
SfePy Documentation

Release version: 2025.1

Robert Cimrman and Contributors

Mar 26, 2025

CONTENTS

1	Documentation	3
1.1	Introduction	3
1.2	Installation	3
1.2.1	Supported Platforms	3
1.2.2	Installing SfePy	4
1.2.3	Using SfePy Docker Images	4
1.2.4	Installing SfePy from Sources	4
1.2.5	Testing Installation	6
1.2.6	Debugging	7
1.3	Tutorial	8
1.3.1	Basic <i>SfePy</i> Usage	8
1.3.2	Basic Notions	8
1.3.3	Running a Simulation	9
1.3.4	Example Problem Description File	10
1.3.5	Interactive Example: Linear Elasticity	13
1.4	User's Guide	19
1.4.1	Running a Simulation	19
1.4.2	Visualization of Results	21
1.4.3	Problem Description File	23
1.4.4	Building Equations in SfePy	39
1.4.5	Term Evaluation	39
1.4.6	Solution Postprocessing	40
1.4.7	Probing	42
1.4.8	Postprocessing filters	42
1.4.9	Solvers	42
1.4.10	Solving Problems in Parallel	45
1.4.11	Isogeometric Analysis	46
1.5	Examples	48
1.5.1	Primer	48
1.5.2	Using Salome with SfePy	72
1.5.3	Preprocessing: <i>FreeCAD/OpenSCAD + Gmsh</i>	75
1.5.4	Material Identification	85
1.5.5	Mesh parametrization	88
1.5.6	Examples	92
1.5.7	Example Applications	717
1.6	Useful Code Snippets and FAQ	717
1.6.1	1. Installation	718
1.6.2	2. Mesh-related tasks	718
1.6.3	3. Regions	719
1.6.4	4. Material parameters	720

1.6.5	5. Solvers	721
1.6.6	6. Postprocessing and visualization	721
1.6.7	7. Finite element method implementation	722
1.7	Theoretical Background	723
1.7.1	Notes on solving PDEs by the Finite Element Method	723
1.7.2	Implementation of Essential Boundary Conditions	726
1.8	Term Overview	727
1.8.1	Term Syntax	727
1.8.2	Term Table	729
2	Development	757
2.1	General Information	757
2.2	Possible Topics	757
2.3	Developer Guide	758
2.3.1	Retrieving the Latest Code	758
2.3.2	SfePy Directory Structure	759
2.3.3	Exploring the Code	760
2.3.4	How to Contribute	761
2.3.5	How to Regenerate Documentation	766
2.3.6	How to Implement a New Term	766
2.3.7	Multi-linear Terms	772
2.3.8	How To Make a Release	774
2.3.9	Module Index	777
	Bibliography	1209
	Python Module Index	1211
	Index	1215

- [genindex](#)
- [modindex](#)
- [search](#)

DOCUMENTATION

1.1 Introduction

SfePy (<http://sfepy.org>) is a software for solving systems of coupled partial differential equations (PDEs) by the finite element method in 1D, 2D and 3D. It can be viewed both as black-box PDE solver, and as a Python package which can be used for building custom applications. The word “simple” means that complex FEM problems can be coded very easily and rapidly.

There is also a preliminary support for the isogeometric analysis, outlined in *Isogeometric Analysis*.

The code is written almost entirely in [Python](#), with exception of the most time demanding routines - those are written in C and wrapped by [Cython](#) or written directly in Cython.

SfePy is a free software released under the [New BSD License](#). It relies on [NumPy](#) and [SciPy](#) (an excellent collection of tools for scientific computations in Python). It is a multi-platform software that should work on Linux, Mac OS X and Windows.

SfePy was originally developed as a flexible framework to quickly implement and test the mathematical models developed during our various research projects. It has evolved, however, to a rather full-featured (yet small) finite element code. Many terms have been implemented that can be used to build the PDEs, see *Term Overview*. *SfePy* comes also with a number of examples that can get you started, check *Examples*, *Gallery* and *Tutorial*. Some more advanced features are discussed in *Primer*.

1.2 Installation

1.2.1 Supported Platforms

SfePy is known to work on various flavors of recent Linux, Intel-based MacOS and Windows. The release 2019.4 was the last with Python 2.7 support. *SfePy* should work with any recent Python 3.x that is supported by [NumPy](#) and [SciPy](#).

Note: Depending on Python installation and OS used, replacing `python` by `python3` might be required in all the commands below (e.g. in *Compilation of C Extension Modules*) in order to use Python 3.

1.2.2 Installing SfePy

The released versions of SfePy can be installed as follows.

- Using `pip`:

```
pip install sfepy
```

- Using `conda`:

1. Install `miniforge`. The miniforge distribution contains the minimal installers for `Conda` and `Mamba` specific to `conda-forge`. The packages are automatically pulled from the `conda-forge` channel. In case of a prior installation, remove the `defaults` channel to avoid a potential `Anaconda` terms of service violation:

```
conda config --remove channels defaults
```

2. Install `sfepy`:

```
conda install sfepy
```

The above commands pull automatically all required dependencies. This, in combination with *installing_from_sources*, enables to try the git version easily. For example with `conda`, use:

```
conda remove --force sfepy
```

to remove the installed `sfepy` package but keep the dependencies installed.

If the installation succeeded, proceed with *Testing Installation*.

1.2.3 Using SfePy Docker Images

Besides the classical installation we also provide official Docker images with a ready-to-run miniforge-based *SfePy* installation.

Before you start using *SfePy* images, you need to first install and configure Docker on your computer. To do this follow official [Docker documentation](#).

Currently available all-in-one image is:

- `sfepy/sfepy-desktop` - an Ubuntu based container containing a full desktop environment in officially supported flavors accessible via any modern web browser.

For available runtime options and further information see `sfepy-docker` project on Github.

1.2.4 Installing SfePy from Sources

The latest stable release can be obtained from the [download](#) page. Otherwise, download the development version of the code from [SfePy git repository](#):

```
git clone git://github.com/sfepy/sfepy.git
```

In case you wish to use a specific release instead of the latest master version, use:

```
git tag -l
```

to see the available releases - the release tags have form `release_<year>.<int>`.

See the [download](#) page for additional download options.

Requirements

Installation prerequisites, required to build *SfePy*:

- a C compiler suite,
- Python 3.x,
- NumPy,
- Cython.
- Cmake
- scikit-build
- ninja

Python packages required for using *SfePy*:

- Pyparsing,
- SciPy,
- meshio for reading and writing mesh files,
- scikit-umfpack for enabling UMFPACK solver for SciPy >= 0.14.0,
- mumpsy or python-mumps for linear direct and Schur solver MUMPS
- Matplotlib for various plots,
- PyTables for storing results in HDF5 files,
- SymPy for some tests and functions,
- igakit for generating IGA domains,
- petsc4py and mpi4py for running parallel examples and using parallel solvers from PETSc,
- slepc4py for eigenvalue problem solvers from SLEPc,
- pymetis for mesh partitioning using Metis,
- Read the Docs Sphinx theme for building documentation,
- psutil for memory requirements checking,
- PyVista for post-processing.

Make sure the dependencies of those packages are also installed (e.g *igakit* requires FORTRAN compiler, *mumpsy* requires MUMPS library, *scikit-umfpack* does not work without UMFPACK, *petsc4py* without PETSc etc.). All dependencies of *meshio* need to be installed for full mesh file format support (when using pip: `pip install meshio[all]`).

SfePy should work both with bleeding edge (Git) and last released versions of *NumPy* and *SciPy*. Please, submit an issue at [Issues](#) page in case this does not hold.

Other dependencies/suggestions:

- To be able to (re)generate the documentation *Sphinx*, *numpydoc* and *LaTeX* are needed (see [How to Regenerate Documentation](#)).
- If *doxygen* is installed, the documentation of data structures and functions can be automatically generated by running:

```
python setup.py doxygendocs
```

- Mesh generation tools use *gmsh* or *tetgen*.

- [IPython](#) is recommended over the regular Python shell to fluently follow some parts of primer/tutorial.

Compilation of C Extension Modules

In the *SfePy* top-level directory:

1. Look at `site_cfg_template.py` and follow the instructions therein. Usually no changes are necessary.
2. For in-place use, compile the extension modules:

```
python setup.py build_ext --inplace
```

After a successful compilation, *SfePy* can be used in-place. However, the `sfepy-*` commands, such as `sfepy-run` are only available after installing the package. Their functionality can be accessed by invoking directly the corresponding scripts in `sfepy/scripts/`.

Installation

SfePy can be used without any installation by running its main scripts and examples from the top-level directory of the distribution or can be installed *locally* or *system-wide*:

- system-wide (may require root privileges):

```
pip install .
```

- local:

```
pip install --user .
```

- development (editable install):

```
pip install -e .
```

The editable install allows working in-place and at the same time the `sfepy-*` commands are available.

If all went well, proceed with [Testing Installation](#).

1.2.5 Testing Installation

After building and/or installing *SfePy* you should check if all the functions are working properly by running the automated tests.

Running Automated Test Suite

The test suite is based on [pytest](#). Install it by:

```
pip install pytest
```

or:

```
conda install pytest
```

when working in [miniforge](#). If *SfePy* was installed, it can be tested using the command:

```
sfepy-test
```

that accepts all of the [pytest](#) options, for example:

```
sfepy-test -vv --durations=0 -m 'not slow' -k test_assembling.py
```

The tests output directory can also be specified:

```
sfepy-test --output-dir=output-tests
```

In general, the tests can be run using:

```
python -c "import sfepy; sfepy.test()"
```

in the *SfePy* top-level directory in case of the in-place build and anywhere else when testing the installed package. Additional [pytest](#) options can be passed as arguments to `sfepy.test()`, for example:

```
python -c "import sfepy; sfepy.test('-vv', '--durations=0', '-m not slow', '-k test_
↪assembling.py')"
```

Analogously to `sfepy-test`, the tests output directory can be specified using:

```
python -c "import sfepy; sfepy.test('--output-dir=output-tests')"
```

See [pytest usage instructions](#) for other options and usage patterns.

To test an in-place build (e.g. in a cloned git repository), the following simpler command can be used in the sources top-level directory:

```
python -m pytest sfepy/tests
python -m pytest -v sfepy/tests/test_assembling.py
```

which will also add the current directory to `sys.path`. If the top-level directory is already in `sys.path` (e.g. using `export PYTHONPATH=.`), the simplest way of invoking `pytest` is:

```
pytest sfepy/tests
pytest -v sfepy/tests/test_assembling.py
```

1.2.6 Debugging

If something goes wrong, edit the `site_cfg.py` config file and set `debug_flags = '-DDEBUG_FMF'` to turn on bound checks in the low level C functions, and recompile the code:

```
python setup.py clean
python setup.py build_ext --inplace
```

Then re-run your code and report the output to the [SfePy mailing list](#).

1.3 Tutorial

1.3.1 Basic SfePy Usage

SfePy package can be used in two basic ways as a:

1. Black-box Partial Differential Equation (PDE) solver,
2. Python package to build custom applications involving solving PDEs by the Finite Element Method (FEM).

This tutorial focuses on the first way and introduces the basic concepts and nomenclature used in the following parts of the documentation. Check also the *Primer* which focuses on a particular problem in detail.

Users not familiar with the finite element method should start with the *Notes on solving PDEs by the Finite Element Method*.

Invoking SfePy from the Command Line

This section introduces the basics of running SfePy from the command line, assuming it was properly installed, see *Installation*.

The command `sfepy-run` is the **most basic starting point** in SfePy. It can be used to run declarative problem description files (see below) as follows:

```
sfepy-run <problem_description_file>
```

Using SfePy Interactively

All functions of SfePy package can be also used interactively (see *Interactive Example: Linear Elasticity* for instance).

We recommend to use the `IPython` interactive shell for the best fluent user experience.

1.3.2 Basic Notions

The simplest way of using SfePy is to solve a system of PDEs defined in a *problem description file*, also referred to as *input file*. In such a file, the problem is described using several keywords that allow one to define the equations, variables, finite element approximations, solvers and solution domain and subdomains (see `sec-problem-description-file` for a full list of those keywords).

The syntax of the *problem description file* is very simple yet powerful, as the file itself is just a regular Python module that can be normally imported – no special parsing is necessary. The keywords mentioned above are regular Python variables (usually of the *dict* type) with special names.

Below we show:

- how to solve a problem given by a problem description file, and
- explain the elements of the file on several examples.

But let us begin with a slight detour...

Sneak Peek: What is Going on Under the Hood

1. A command (usually *sfepy-run* as in this tutorial) or a script reads in an input file.
2. Following the contents of the input file, a *Problem* instance is created – this is the input file coming to life. Let us call the instance *problem*.
 - The *Problem* instance sets up its domain, regions (various sub-domains), fields (the FE approximations), the equations and the solvers. The equations determine the materials and variables in use – only those are fully instantiated, so the input file can safely contain definitions of items that are not used actually.
3. The solution is then obtained by calling *problem.solve()* function, which in turn calls a top-level time-stepping solver. In each step, *problem.time_update()* is called to setup boundary conditions, material parameters and other potentially time-dependent data. The *problem.save_state()* is called at the end of each time step to save the results. This holds also for stationary problems with a single “time step”.

So that is it – using the code a black-box PDE solver shields the user from having to create the *Problem* instance by hand. But note that this is possible, and often necessary when the flexibility of the default solvers is not enough. At the end of the tutorial an example demonstrating the interactive creation of the *Problem* instance is shown, see [Interactive Example: Linear Elasticity](#).

Now let us continue with running a simulation.

1.3.3 Running a Simulation

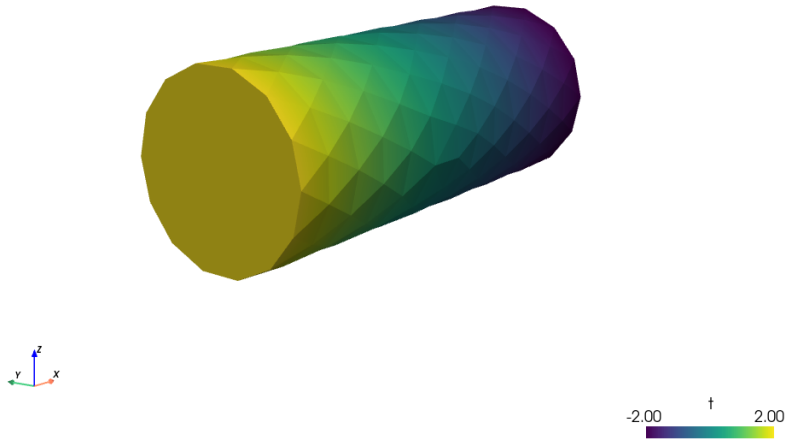
The following commands should be run in the top-level directory of the *SfePy* source tree after compiling the C extension files. See [Installation](#) for full installation instructions.

- Download `sfepy/examples/diffusion/poisson_short_syntax.py`. It represents our sample *SfePy* sec-problem-description-file, which defines the problem to be solved in terms *SfePy* can understand.
- Use the downloaded file in place of `<problem_description_file.py>` and run *sfepy-run* as [described above](#). The successful execution of the command creates output file `cylinder.vtk` in the *SfePy* top-level directory.

Postprocessing the Results

- The *sfepy-view* command can be used for quick postprocessing and visualization of the *SfePy* output files. It requires *pyvista* installed on your system.
- As a simple example, try:

```
sfepy-view cylinder.vtk
```
- The following interactive 3D window should display:



- You can manipulate displayed image using:
 - the left mouse button by itself orbits the 3D view,
 - holding shift and the left mouse button pans the view,
 - holding control and the left mouse button rotates about the screen normal axis,
 - the right mouse button controls the zoom.

1.3.4 Example Problem Description File

Here we discuss the contents of the `sfepy/examples/diffusion/poisson_short_syntax.py` problem description file. For additional examples, see the problem description files in the `sfepy/examples/` directory of *SfePy*.

The problem at hand is the following:

$$c\Delta T = f \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma, \quad (1.1)$$

where $\Gamma \subseteq \Omega$ is a subset of the domain Ω boundary. For simplicity, we set $f \equiv 0$, but we still work with the material constant c even though it has no influence on the solution in this case. We also assume zero fluxes over $\partial\Omega \setminus \Gamma$, i.e. $\frac{\partial T}{\partial n} = 0$ there. The particular boundary conditions used below are $T = 2$ on the left side of the cylindrical domain depicted in the previous section and $T = -2$ on the right side.

The first step to do is to write (1.1) in *weak formulation* (1.15). The $f = 0$, $g = \frac{\partial T}{\partial n} = 0$. So only one term in weak form (1.15) remains:

$$\int_{\Omega} c \nabla T \cdot \nabla s = 0, \quad \forall s \in V_0. \quad (1.2)$$

Comparing the above integral term with the long table in [Term Overview](#), we can see that *SfePy* contains this term under name `dw_laplace`. We are now ready to proceed to the actual problem definition.

Open the `sfepy/examples/diffusion/poisson_short_syntax.py` file in your favorite text editor. Note that the file is a regular Python source code.

```
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'
```

The `filename_mesh` variable points to the file containing the mesh for the particular problem. *SfePy* supports a variety of mesh formats.

```
materials = {
    'coef': ({'val' : 1.0},),
}
```

Here we define just a constant coefficient c of the Poisson equation, using the ‘*values*’ attribute. Other possible attribute is ‘*function*’ for material coefficients computed/obtained at runtime.

Many finite element problems require the definition of material parameters. These can be handled in *SfePy* with material variables which associate the material parameters with the corresponding region of the mesh.

```
regions = {
    'Omega' : 'all', # or 'cells of group 6'
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.099999)', 'facet'),
}
```

Regions assign names to various parts of the finite element mesh. The region names can later be referred to, for example when specifying portions of the mesh to apply boundary conditions to. Regions can be specified in a variety of ways, including by element or by node. Here, ‘*Omega*’ is the elemental domain over which the PDE is solved and ‘*Gamma_Left*’ and ‘*Gamma_Right*’ define surfaces upon which the boundary conditions will be applied.

```
fields = {
    'temperature': ('real', 1, 'Omega', 1)
}
```

A field is used mainly to define the approximation on a (sub)domain, i.e. to define the discrete spaces V_h , where we seek the solution.

The Poisson equation can be used to compute e.g. a temperature distribution, so let us call our field ‘*temperature*’. On the region ‘*Omega*’ it will be approximated using linear finite elements.

A field in a given region defines the finite element approximation. Several variables can use the same field, see below.

```
variables = {
    't': ('unknown field', 'temperature', 0),
    's': ('test field', 'temperature', 't'),
}
```

One field can be used to generate discrete degrees of freedom (DOFs) of several variables. Here the unknown variable (the temperature) is called ‘*t*’, its associated DOF name is ‘*t.0*’ – this will be referred to in the Dirichlet boundary section (*ebc*). The corresponding test variable of the weak formulation is called ‘*s*’. Notice that the ‘*dual*’ item of a test variable must specify the unknown it corresponds to.

For each unknown (or state) variable there has to be a test (or virtual) variable defined, as usual in weak formulation of PDEs.

```
ebcs = {
    't1': ('Gamma_Left', {'t.0' : 2.0}),
```

(continues on next page)

(continued from previous page)

```
't2', ('Gamma_Right', {'t.0' : -2.0}),
}
```

Essential (Dirichlet) boundary conditions can be specified as above.

Boundary conditions place restrictions on the finite element formulation and create a unique solution to the problem. Here, we specify that a temperature of +2 is applied to the left surface of the mesh and a temperature of -2 is applied to the right surface.

```
integrals = {
    'i': 2,
}
```

Integrals specify which numerical scheme to use. Here we are using a 2nd order quadrature over a 3 dimensional space.

```
equations = {
    'Temperature' : ""dw_laplace.i.Omega( coef.val, s, t ) = 0""
}
```

The equation above directly corresponds to the discrete version of (1.2), namely: Find $\mathbf{t} \in V_h$, such that

$$\mathbf{s}^T \left(\int_{\Omega_h} c \mathbf{G}^T \mathbf{G} \right) \mathbf{t} = 0, \quad \forall \mathbf{s} \in V_{h0},$$

where $\nabla u \approx \mathbf{G}u$.

The equations block is the heart of the *SfePy* problem description file. Here, we are specifying that the Laplacian of the temperature (in the weak formulation) is 0, where *coef.val* is a material constant. We are using the 'i' integral defined previously, over the domain specified by the region 'Omega'.

The above syntax is useful for defining *custom integrals* with user-defined quadrature points and weights, see [Integrals](#). The above uniform integration can be more easily achieved by:

```
equations = {
    'Temperature' : ""dw_laplace.2.Omega( coef.val, s, t ) = 0""
}
```

The integration order is specified directly in place of the integral name. The integral definition is superfluous in this case.

```
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
               {'i_max' : 1,
                'eps_a' : 1e-10,
               }),
}
```

Here, we specify the linear and nonlinear solver kinds and options. See [sfepy.solvers.ls](#), [sfepy.solvers.nls](#) and [sfepy.solvers.ts_solvers](#) for available solvers and their parameters.. Even linear problems are solved by a nonlinear solver (KISS rule) – only one iteration is needed and the final residual is obtained for free. Note that we do not need to define a time-stepping solver here - the problem is stationary and the default 'ts.stationary' solver is created automatically.

```
options = {
    'nls' : 'newton',
    'ls' : 'ls',
}
```

The solvers to use are specified in the options block. We can define multiple solvers with different convergence parameters.

That's it! Now it is possible to proceed as described in [Invoking SfePy from the Command Line](#).

1.3.5 Interactive Example: Linear Elasticity

This example shows how to use *SfePy* interactively, but also how to make a custom simulation script. We will use *IPython* interactive shell which allows more flexible and intuitive work (but you can use standard Python shell as well).

We wish to solve the following linear elasticity problem:

$$-\frac{\partial \sigma_{ij}(\underline{u})}{\partial x_j} + f_i = 0 \text{ in } \Omega, \quad \underline{u} = 0 \text{ on } \Gamma_1, \quad u_1 = \bar{u}_1 \text{ on } \Gamma_2, \quad (1.3)$$

where the stress is defined as $\sigma_{ij} = 2\mu e_{ij} + \lambda e_{kk}\delta_{ij}$, λ, μ are the Lamé's constants, the strain is $e_{ij}(\underline{u}) = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$ and f are volume forces. This can be written in general form as $\sigma_{ij}(\underline{u}) = D_{ijkl}e_{kl}(\underline{u})$, where in our case $D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda\delta_{ij}\delta_{kl}$.

In the weak form the equation (1.3) is

$$\int_{\Omega} D_{ijkl}e_{kl}(\underline{u})e_{ij}(\underline{v}) + \int_{\Omega} f_i v_i = 0, \quad (1.4)$$

where $\underline{v}$ is the test function, and both $\underline{u}, \underline{v}$ belong to a suitable function space.

Hint: Whenever you create a new object (e.g. a *Mesh* instance, see below), try to print it using the *print* statement – it will give you insight about the object internals.

The whole example summarized in a script is available below in [Complete Example as a Script](#).

In the *SfePy* top-level directory run

```
ipython
```

```
In [1]: import numpy as nm
In [2]: from sfepy.discrete.fem import Mesh, FEDomain, Field
```

Read a finite element mesh, that defines the domain Ω .

```
In [3]: mesh = Mesh.from_file('meshes/2d/rectangle_tri.mesh')
```

Create a domain. The domain allows defining regions or subdomains.

```
In [4]: domain = FEDomain('domain', mesh)
```

Define the regions – the whole domain Ω , where the solution is sought, and Γ_1, Γ_2 , where the boundary conditions will be applied. As the domain is rectangular, we first get a bounding box to get correct bounds for selecting the boundary edges.

```
In [5]: min_x, max_x = domain.get_mesh_bounding_box()[:, 0]
In [6]: eps = 1e-8 * (max_x - min_x)
In [7]: omega = domain.create_region('Omega', 'all')
In [8]: gamma1 = domain.create_region('Gamma1',
...:                                 'vertices in x < %.10f' % (min_x + eps),
...:                                 'facet')
In [9]: gamma2 = domain.create_region('Gamma2',
...:                                 'vertices in x > %.10f' % (max_x - eps),
...:                                 'facet')
```

Next we define the actual finite element approximation using the *Field* class.

```
In [10]: field = Field.from_args('fu', nm.float64, 'vector', omega,
...:                             approx_order=2)
```

Using the field *fu*, we can define both the unknown variable *u* and the test variable *v*.

```
In [11]: from sfepy.discrete import (FieldVariable, Material, Integral, Function,
...:                                 Equation, Equations, Problem)
In [12]: u = FieldVariable('u', 'unknown', field)
In [13]: v = FieldVariable('v', 'test', field, primary_var_name='u')
```

Before we can define the terms to build the equation of linear elasticity, we have to create also the materials, i.e. define the (constitutive) parameters. The linear elastic material *m* will be defined using the two Lamé constants $\lambda = 1$, $\mu = 1$. The volume forces will be defined also as a material as a constant (column) vector $[0.02, 0.01]^T$.

```
In [14]: from sfepy.mechanics.matcoefs import stiffness_from_lame
In [15]: m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
In [16]: f = Material('f', val=[[0.02], [0.01]])
```

One more thing needs to be defined – the numerical quadrature that will be used to integrate each term over its domain.

```
In [17]: integral = Integral('i', order=3)
```

Now we are ready to define the two terms and build the equations.

```
In [18]: from sfepy.terms import Term
In [19]: t1 = Term.new('dw_lin_elastic(m.D, v, u)',
...:                  integral, omega, m=m, v=v, u=u)
In [20]: t2 = Term.new('dw_volume_lvf(f.val, v)',
...:                  integral, omega, f=f, v=v)
In [21]: eq = Equation('balance', t1 + t2)
In [22]: eqs = Equations([eq])
```

The equations have to be completed by boundary conditions. Let us clamp the left edge Γ_1 , and shift the right edge Γ_2 in the *x* direction a bit, depending on the *y* coordinate.

```
In [23]: from sfepy.discrete.conditions import Conditions, EssentialBC
In [24]: fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})
```

(continues on next page)

(continued from previous page)

```
In [25]: def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):
.....:         val = shift * coors[:,1]**2
.....:         return val
In [26]: bc_fun = Function('shift_u_fun', shift_u_fun,
.....:         extra_args={'shift' : 0.01})
In [27]: shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})
```

The last thing to define before building the problem are the solvers. Here we just use a sparse direct *SciPy solver* and the *SfePy Newton solver* with default parameters. We also wish to store the convergence statistics of the Newton solver. As the problem is linear it should converge in one iteration.

```
In [28]: from sfepy.base.base import IndexedStruct
In [29]: from sfepy.solvers.ls import ScipyDirect
In [30]: from sfepy.solvers.nls import Newton

In [31]: ls = ScipyDirect({})
In [32]: nls_status = IndexedStruct()
In [33]: nls = Newton({}, lin_solver=ls, status=nls_status)
```

Now we are ready to create a *Problem* instance.

```
In [34]: pb = Problem('elasticity', equations=eqs)
```

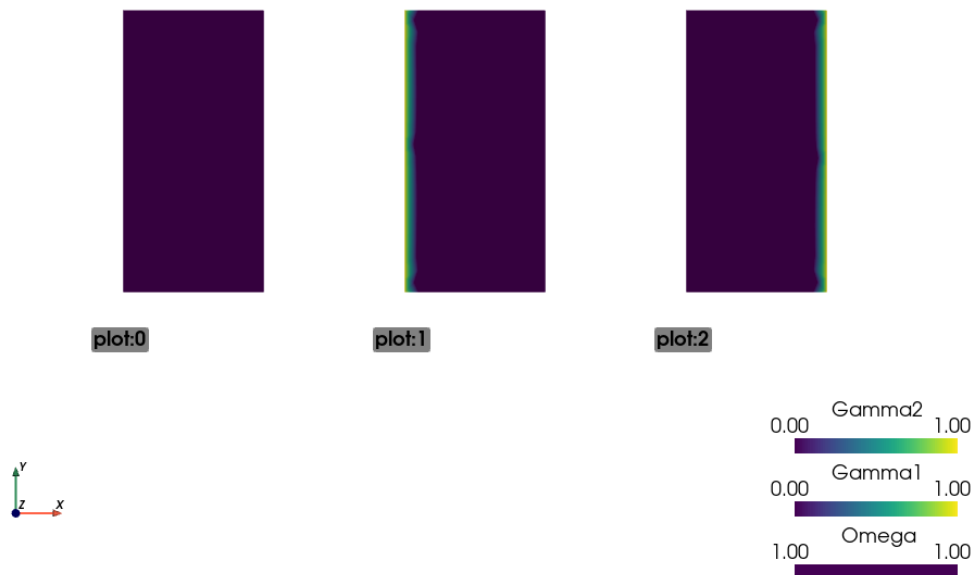
The *Problem* has several handy methods for debugging. Let us try saving the regions into a VTK file.

```
In [35]: pb.save_regions_as_groups('regions')
```

And view them

```
In [36]: ! sfepy-view regions.vtk -2 --grid-vector1 "[2, 0, 0]"
```

You should see this:



Finally, we set the boundary conditions and the top-level solver, solve the problem, save and view the results. For

stationary problems, the top-level solver needs not to be a time-stepping solver - when a nonlinear solver is set instead, the default 'ts.stationary' time-stepping solver is created automatically.

```
In [39]: pb.set_bcs(ebcs=Conditions([fix_u, shift_u]))
In [40]: pb.set_solver(nls)

In [41]: status = IndexedStruct()
In [42]: variables = pb.solve(status=status)

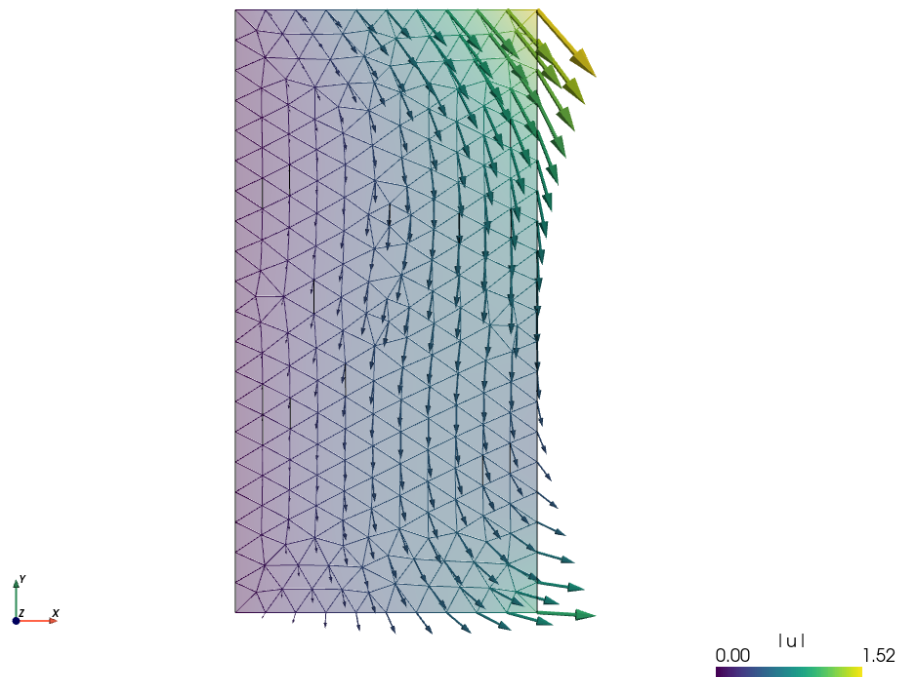
In [43]: print('Nonlinear solver status:\n', nls_status)
In [44]: print('Stationary solver status:\n', status)

In [45]: pb.save_state('linear_elasticity.vtk', variables)
```

This

```
sfepy-view linear_elasticity.vtk -2
```

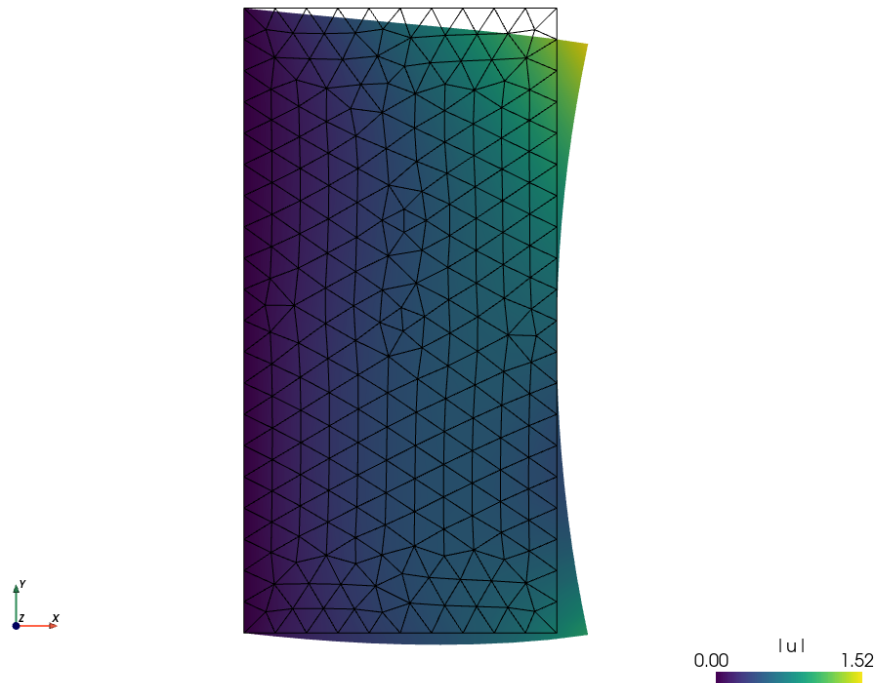
is used to produce the resulting image:



The default view is not very fancy. Let us show the displacements by shifting the mesh. Close the previous window and do

```
sfepy-view linear_elasticity.vtk -2 -f u:wu:p0 1:vw:p0
```

And the result is:



Complete Example as a Script

The source code: `linear_elastic_interactive.py`.

This file should be run from the top-level *SfePy* source directory so it can find the mesh file correctly. Please note that the provided example script may differ from above tutorial in some minor details.

```

1  #!/usr/bin/env python
2  """
3  Linear elasticity example using the imperative API.
4  """
5  from argparse import ArgumentParser
6  import numpy as nm
7
8  import sys
9  sys.path.append('.')
10
11  from sfepy.base.base import IndexedStruct
12  from sfepy.discrete import (FieldVariable, Material, Integral, Function,
13                             Equation, Equations, Problem)
14  from sfepy.discrete.fem import Mesh, FEDomain, Field
15  from sfepy.terms import Term
16  from sfepy.discrete.conditions import Conditions, EssentialBC
17  from sfepy.solvers.ls import ScipyDirect
18  from sfepy.solvers.nls import Newton
19  from sfepy.mechanics.matcoefs import stiffness_from_lame
20
21
22  def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):

```

(continues on next page)

(continued from previous page)

```

23     """
24     Define a displacement depending on the y coordinate.
25     """
26     val = shift * coors[:,1]**2
27
28     return val
29
30
31 def main():
32     from sfepy import data_dir
33
34     parser = ArgumentParser()
35     parser.add_argument('--version', action='version', version='%(prog)s')
36     options = parser.parse_args()
37
38     mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
39     domain = FEDomain('domain', mesh)
40
41     min_x, max_x = domain.get_mesh_bounding_box()[:,0]
42     eps = 1e-8 * (max_x - min_x)
43     omega = domain.create_region('Omega', 'all')
44     gamma1 = domain.create_region('Gamma1',
45                                   'vertices in x < %.10f' % (min_x + eps),
46                                   'facet')
47     gamma2 = domain.create_region('Gamma2',
48                                   'vertices in x > %.10f' % (max_x - eps),
49                                   'facet')
50
51     field = Field.from_args('fu', nm.float64, 'vector', omega,
52                             approx_order=2)
53
54     u = FieldVariable('u', 'unknown', field)
55     v = FieldVariable('v', 'test', field, primary_var_name='u')
56
57     m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
58     f = Material('f', val=[[0.02], [0.01]])
59
60     integral = Integral('i', order=3)
61
62     t1 = Term.new('dw_lin_elastic(m.D, v, u)',
63                  integral, omega, m=m, v=v, u=u)
64     t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
65     eq = Equation('balance', t1 + t2)
66     eqs = Equations([eq])
67
68     fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})
69
70     bc_fun = Function('shift_u_fun', shift_u_fun,
71                       extra_args={'shift' : 0.01})
72     shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})
73
74     ls = ScipyDirect({})

```

(continues on next page)

(continued from previous page)

```

75
76     nls_status = IndexedStruct()
77     nls = Newton({}, lin_solver=ls, status=nls_status)
78
79     pb = Problem('elasticity', equations=eqs)
80     pb.save_regions_as_groups('regions')
81
82     pb.set_bcs(ebcs=Conditions([fix_u, shift_u]))
83
84     pb.set_solver(nls)
85
86     status = IndexedStruct()
87     variables = pb.solve(status=status)
88
89     print('Nonlinear solver status:\n', nls_status)
90     print('Stationary solver status:\n', status)
91
92     pb.save_state('linear_elasticity.vtk', variables)
93
94
95 if __name__ == '__main__':
96     main()

```

1.4 User's Guide

This manual provides reference documentation to *SfePy* from a user's perspective.

1.4.1 Running a Simulation

The following should be run in the top-level directory of the *SfePy* source tree after installing the sfepy package. See [Installation](#) for installation instructions.

Basic Usage

- sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py
 - Creates cylinder.vtk
- sfepy-run sfepy/examples/navier_stokes/stokes.py
 - Creates channels_symm944t.vtk

Running Tests

The tests are based on `pytest` and can be run using:

```
python -c "import sfepy; sfepy.test()"
```

See *Testing Installation* for additional information.

Computations and Examples

Most of the example problems in the `sfepy/examples` directory can be computed by the `sfepy-run` command. Additional (stand-alone) examples can be run directly, e.g.:

```
python sfepy/examples/large_deformation/compare_elastic_materials.py
```

Common Tasks

- Run a simulation:

```
sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py
sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py -o some_results # ->
↳ produces some_results.vtk
```

- Print available terms:

```
sfepy-run --list=terms
```

- Run a simulation and also save Dirichlet boundary conditions:

```
sfepy-run --save-ebc sfepy/examples/diffusion/poisson_short_syntax.py # -> produces
↳ an additional .vtk file with BC visualization
```

- Use a restart file to continue an interrupted simulation:

- **Warning:** This feature is preliminary and does not support terms with internal state.
- Run:

```
sfepy-run sfepy/examples/large_deformation/balloon.py --save-restart=-1
```

and break the computation after a while (hit Ctrl-C). The mode `--save-restart=-1` is currently the only supported mode. It saves a restart file for each time step, and only the last computed time step restart file is kept.

- A file named `'unit_ball.restart-?? .h5'` should be created, where `'??'` indicates the last stored time step. Let us assume it is `'unit_ball.restart-04.h5'`, i.e. the fifth step.
- Restart the simulation by:

```
sfepy-run sfepy/examples/large_deformation/balloon.py --load-restart=unit_ball.
↳ restart-04.h5
```

The simulation should continue from the next time step. Verify that by running:

```
sfepy-run sfepy/examples/large_deformation/balloon.py
```

and compare the residuals printed in the corresponding time steps.

1.4.2 Visualization of Results

Quick visualisation of the *SfePy* results can be done by `sfepy.scripts.resview.py` command, which uses [PyVista](#) visualization toolkit (need to be installed).

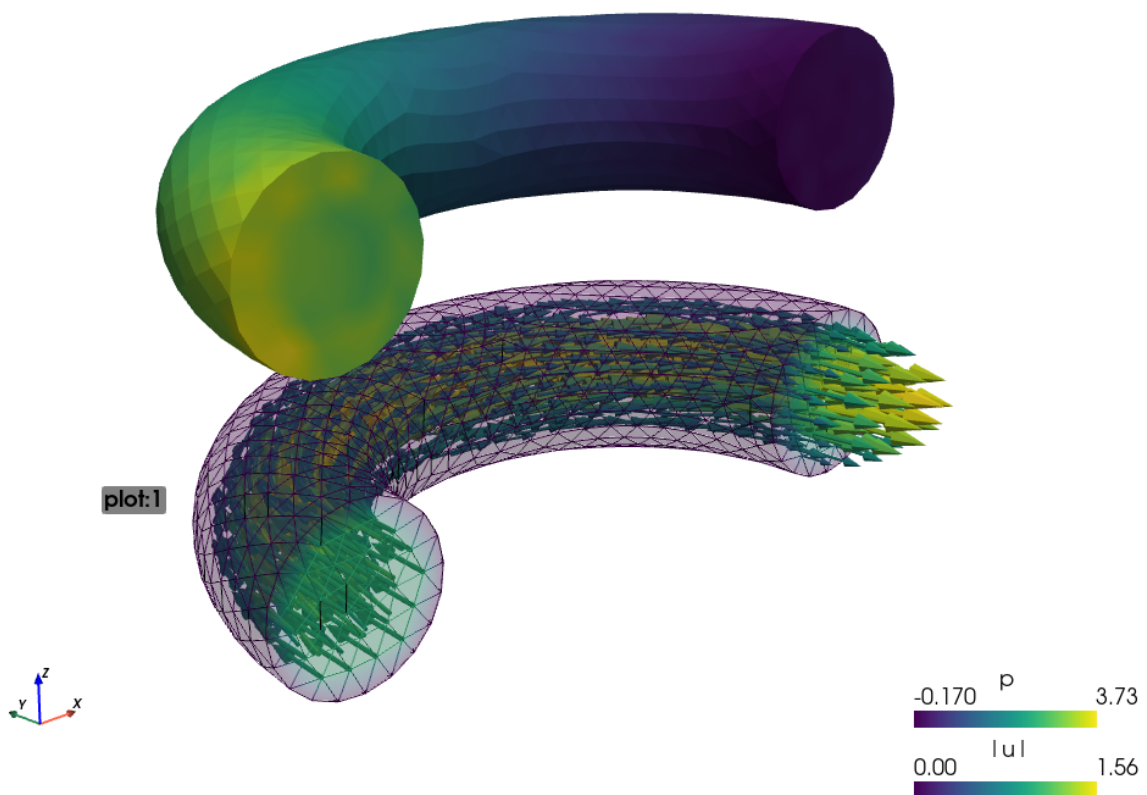
For example, results of a stationary Navier-Stokes flow simulation:

```
sfepy-run sfepy/examples/navier_stokes/navier_stokes.py -o navier_stokes
```

can be viewed using:

```
sfepy-view navier_stokes.vtk
```

produces:

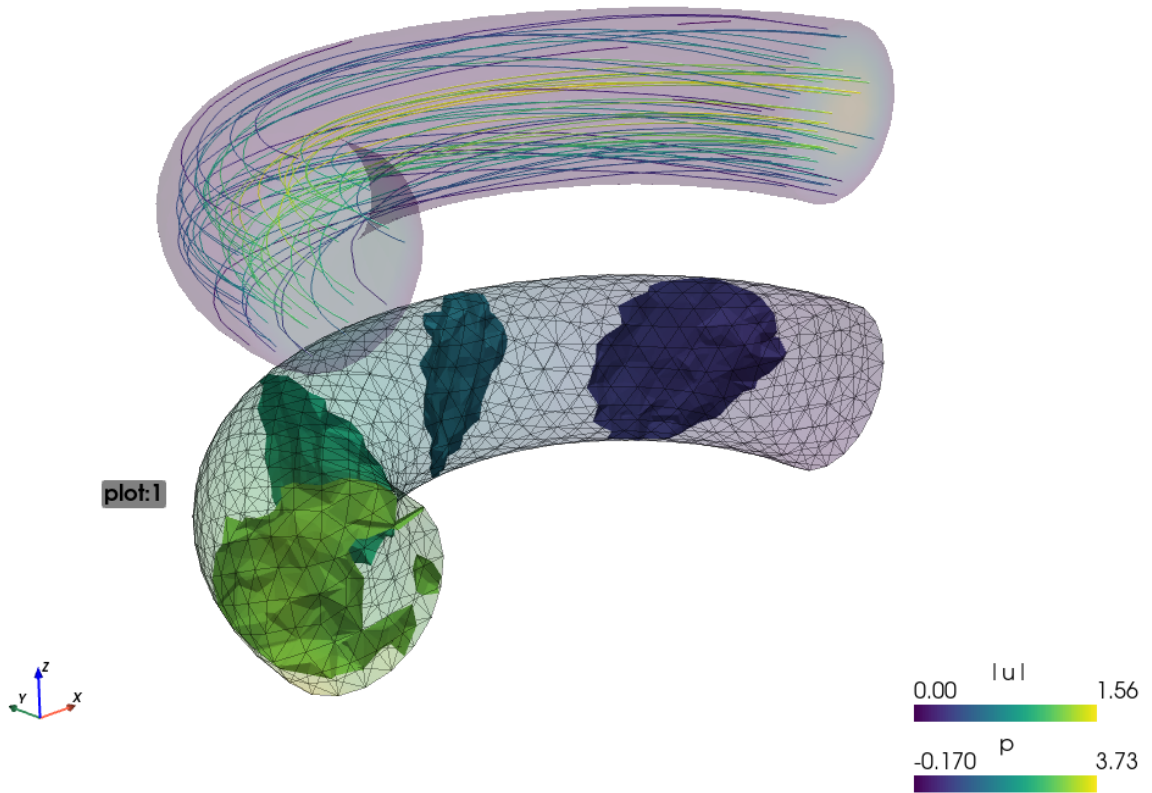


Using:

```
sfepy-view navier_stokes.vtk -f p:i5:p0 p:e:o.2:p0 u:t1000:p1 u:o.2:p1
```

the output is split into plots `plot:0` and `plot:1`, where these plots contain:

- `plot:0`: field `p`, 5 isosurfaces, mesh edges switched on, the surface opacity set to 0.2;
- `plot:1`: vector field `u` streamlines, the surface opacity set to 0.2;



The actual camera position can be printed by pressing the ‘c’ key. The above figures use the following values:

```
--camera-position="-0.236731,-0.352216,0.237044,0.094458,0.0292563,0.0385116,0.266969,0.252278,0.930099"
```

The argument `-o filename.png` takes the screenshot of the produced view:

```
sfepy-view navier_stokes.vtk -o image.png
```

1.4.3 Problem Description File

Here we discuss the basic items that users have to specify in their input files. For complete examples, see the problem description files in the `sfepy/examples/` directory of SfePy.

Long Syntax

Besides the *short syntax* described below there is (due to history) also a *long syntax* which is explained in `problem_desc_file_long`. The short and long syntax can be mixed together in one description file.

FE Mesh

A FE mesh defining a domain geometry can be stored in several formats:

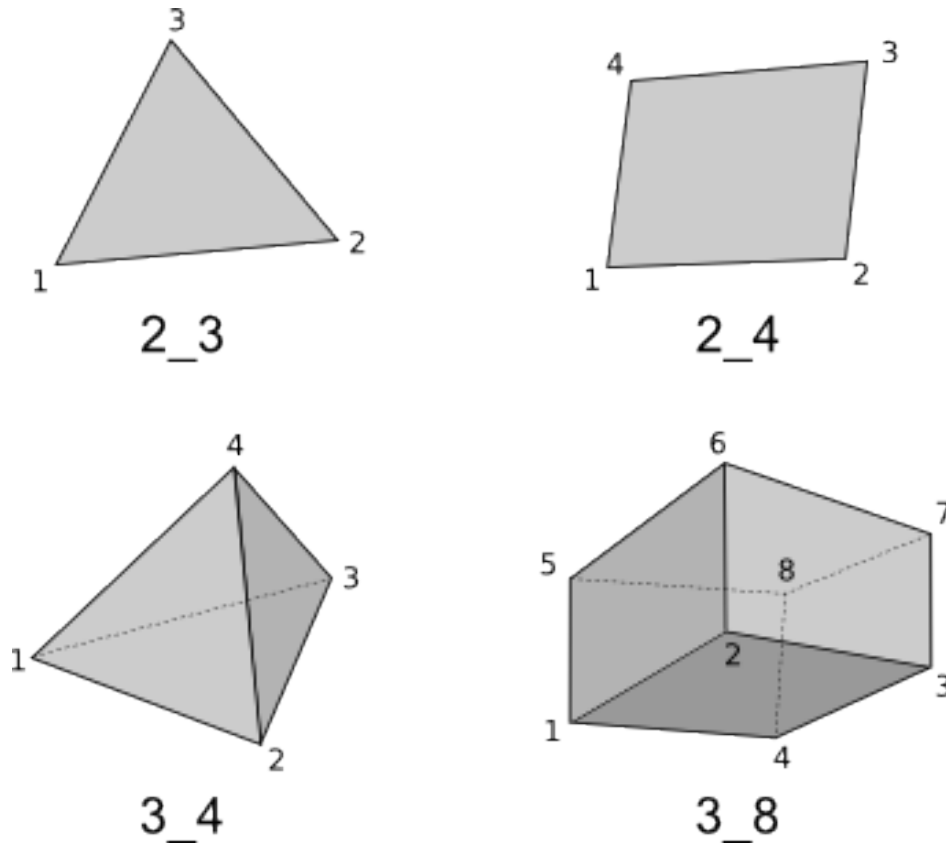
- legacy VTK (`.vtk`)
- custom HDF5 file (`.h5`)
- medit mesh file (`.mesh`)
- tetgen mesh files (`.node`, `.ele`)
- comsol text mesh file (`.txt`)
- abaqus text mesh file (`.inp`)
- avs-ucd text mesh file (`.inp`)
- hypermesh text mesh file (`.hmascii`)
- hermes3d mesh file (`.mesh3d`)
- nastran text mesh file (`.bdf`)
- gambit neutral text mesh file (`.neu`)
- salome/pythonocc med binary mesh file (`.med`)

Example:

```
filename_mesh = 'meshes/3d/cylinder.vtk'
```

The VTK and HDF5 formats can be used for storing the results. The format can be selected in options, see *Miscellaneous*.

The following geometry elements are supported:



Note the orientation of the vertices matters, the figure displays the correct orientation when interpreted in a [right-handed coordinate system](#).

Regions

Regions serve to select a certain part of the computational domain using topological entities of the FE mesh. They are used to define the boundary conditions, the domains of terms and materials etc.

Let us denote D the maximal dimension of topological entities. For volume meshes it is also the dimension of space the domain is embedded in. Then the following topological entities can be defined on the mesh (notation follows [Logg2012]):

topological entity	dimension	co-dimension
vertex	0	D
edge	1	$D - 1$
face	2	$D - 2$
facet	$D - 1$	1
cell	D	0

If $D = 2$, faces are not defined and facets are edges. If $D = 3$, facets are faces.

Following the above definitions, a region can be of different *kind*:

- `cell`, `facet`, `face`, `edge`, `vertex` - entities of higher dimension are not included.
- `cell_only`, `facet_only`, `face_only`, `edge_only`, `vertex_only` - only the specified entities are included, other entities are empty sets, so that set-like operators still work, see below.

- The `cell` kind is the most general and should be used with cell terms. It is also the default if the kind is not specified in region definition.
- The `facet` kind (same as `edge` in 2D and `face` in 3D) is to be used with boundary (facet integral) terms.
- The `vertex` (same as `vertex_only`) kind can be used with point-wise defined terms (e.g. point loads).

The kinds allow a clear distinction between regions of different purpose (cell integration domains, facet domains, etc.) and could be used to lower memory usage.

A region definition involves *topological entity selections* combined with *set-like operators*. The set-like operators can result in intermediate regions that have the `cell` kind. The desired kind is set to the final region, removing unneeded entities. Most entity selectors are defined in terms of vertices and cells - the other entities are computed as needed.

Topological Entity Selection

topological entity selection	explanation
<code>all</code>	all entities of the mesh
<code>vertices of surface</code>	surface of the mesh
<code>vertices of group <integer></code>	vertices of given group
<code>vertices of set <str></code>	vertices of a given named vertex set ²
<code>vertices in <expr></code>	vertices given by an expression ³
<code>vertices by <function></code>	vertices given by a function of coordinates ⁴
<code>vertex <id>[, <id>, ...]</code>	vertices given by their ids
<code>vertex in r.<name of another region></code>	any single vertex in the given region
<code>cells of group <integer></code>	cells of given group
<code>cells by <efunction></code>	cells given by a function of coordinates ⁵
<code>cell <id>[, <id>, ...]</code>	cells given by their ids
<code>copy r.<name of another region></code>	a copy of the given region
<code>r.<name of another region></code>	a reference to the given region

² Only if mesh format supports reading boundary condition vertices as vertex sets.

³ `<expr>` is a logical expression like `(y <= 0.1) & (x < 0.2)`. In 2D use `x`, `y`, in 3D use `x`, `y` and `z`. `&` stands for logical and, `|` stands for logical or.

⁴ `<function>` is a function with signature `fun(coors, domain=None)`, where `coors` are coordinates of mesh vertices.

⁵ `<efunction>` is a function with signature `fun(coors, domain=None)`, where `coors` are coordinates of mesh cell centroids.

topological entity selection footnotes

set-like operator	explanation
+v	vertex union
+e	edge union
+f	face union
+s	facet union
+c	cell union
-v	vertex difference
-e	edge difference
-f	face difference
-s	facet difference
-c	cell difference
*v	vertex intersection
*e	edge intersection
*f	face intersection
*s	facet intersection
*c	cell intersection

Region Definition Syntax

Regions are defined by the following Python dictionary:

```
regions = {
    <name> : (<selection>, [<kind>], [<parent>], [{<misc. options>}]),
}
```

or:

```
regions = {
    <name> : <selection>,
}
```

Example definitions:

```
regions = {
    'Omega' : 'all',
    'Right' : ('vertices in (x > 0.99)', 'facet'),
    'Gamma1' : ("""(cells of group 1 *v cells of group 2)
                +v r.Right""", 'facet', 'Omega'),
    'Omega_B' : 'vertices by get_ball',
}
```

The Omega_B region illustrates the selection by a function (see *Topological Entity Selection*). In this example, the function is:

```
import numpy as nm

def get_ball(coors, domain=None):
    x, y, z = coors[:, 0], coors[:, 1], coors[:, 2]
```

(continues on next page)

(continued from previous page)

```

r = nm.sqrt(x**2 + y**2 + z**2)
flag = nm.where((r < 0.1))[0]

return flag

```

The function needs to be registered in *Functions*:

```

functions = {
    'get_ball' : (get_ball,),
}

```

The mirror region can be defined explicitly as:

```

regions = {
    'Top': ('r.Y *v r.Surf1', 'facet', 'Y', {'mirror_region': 'Bottom'}),
    'Bottom': ('r.Y *v r.Surf2', 'facet', 'Y', {'mirror_region': 'Top'}),
}

```

Fields

Fields correspond to FE spaces:

```

fields = {
    <name> : (<data_type>, <shape>, <region_name>, <approx_order>,
             [<space>, <poly_space_basis>])
}

```

where

- <data_type> is a numpy type (float64 or complex128) or ‘real’ or ‘complex’
- <shape> is the number of DOFs per node: 1 or (1,) or ‘scalar’, space dimension (2, or (2,) or 3 or (3,)) or ‘vector’; it can be other positive integer than just 1, 2, or 3
- <region_name> is the name of region where the field is defined
- <approx_order> is the FE approximation order, e.g. 0, 1, 2, ‘1B’ (1 with bubble)
- <space> is the function space
- <poly_space_basis> is the basis of the FE (usually polynomial) space

Example: scalar P1 elements in 2D on a region Omega:

```

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

```

The following approximation orders can be used:

- simplex elements: 1, 2, ‘1B’, ‘2B’
- tensor product elements: 0, 1, ‘1B’

Optional bubble function enrichment is marked by ‘B’.

The implemented combinations of spaces and bases are listed below, the space column corresponds to <space>, the basis column to <poly_space_basis> and region type to the field region type.

Table 1: Fields

space	basis	region kind	description
H1	bernstein	<i>cell, facet</i>	Bernstein basis approximation with positive-only basis function values.
H1	iga	<i>cell</i>	Bezier extraction based NURBS approximation for isogeometric analysis.
H1	lagrange	<i>cell, facet</i>	Lagrange basis nodal approximation.
H1	lagrange_discontinuous	<i>cell</i>	The C0 constant-per-cell approximation.
H1	lobatto	<i>cell</i>	Hierarchical basis approximation with Lobatto polynomials.
H1	sem	<i>cell, facet</i>	Spectral element method approximation.
H1	serendipity	<i>cell, facet</i>	Lagrange basis nodal serendipity approximation with order <= 3.
H1	shell10x	<i>cell</i>	The approximation for the shell10x element.
L2	constant	<i>cell, facet</i>	The L2 constant-in-a-region approximation.
DG	legendre_discontinuous	<i>cell</i>	Discontinuous Galerkin method approximation with Legendre basis.

Variables

Variables use the FE approximation given by the specified field:

```
variables = {
    <name> : (<kind>, <field_name>, <spec>, [<history>])
}
```

where

- <kind> - 'unknown field', 'test field' or 'parameter field'
- <spec> - in case of: primary variable - order in the global vector of unknowns, dual variable - name of primary variable
- <history> - number of time steps to remember prior to current step

Example:

```
variables = {
    't' : ('unknown field', 'temperature', 0, 1),
    's' : ('test field', 'temperature', 't'),
}
```

Integrals

Define the integral type and quadrature rule. This keyword is optional, as the integration orders can be specified directly in equations (see below):

```
integrals = {
    <name> : <order>
}
```

where

- <name> - the integral name - it has to begin with 'i'!
- <order> - the order of polynomials to integrate, or 'custom' for integrals with explicitly given values and weights

Example:

```
import numpy as nm
N = 2
integrals = {
    'i1' : 2,
    'i2' : ('custom', zip(nm.linspace( 1e-10, 0.5, N ),
                          nm.linspace( 1e-10, 0.5, N )),
          [1./N] * N),
}
```

Essential Boundary Conditions and Constraints

The essential boundary conditions set values of DOFs in some regions, while the constraints constrain or transform values of DOFs in some regions.

Dirichlet Boundary Conditions

The Dirichlet, or essential, boundary conditions apply in a given region given by its name, and, optionally, in selected times. The times can be given either using a list of tuples ($t0$, $t1$) making the condition active for $t0 \leq t < t1$, or by a name of a function taking the time argument and returning True or False depending on whether the condition is active at the given time or not.

Dirichlet (essential) boundary conditions:

```
ebcs = {
    <name> : (<region_name>, [<times_specification>],
            {<dof_specification> : <value>[,
              <dof_specification> : <value>, ...]})
}
```

Example:

```
ebcs = {
    'u1' : ('Left', {'u.all' : 0.0}),
    'u2' : ('Right', [(0.0, 1.0)], {'u.0' : 0.1}),
    'phi' : ('Surface', {'phi.all' : 0.0}),
    'u_yz' : ('Gamma', {'u.[1,2]' : 'rotate_yz'}),
}
```

The `u_yz` condition illustrates calculating the condition value by a function. In this example, it is a function of coordinates `coors` of region nodes:

```
import numpy as nm

def rotate_yz(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d
```

(continues on next page)

(continued from previous page)

```
vec = coor[:,1:3] - centre

angle = 10.0 * ts.step

mtx = rotation_matrix2d(angle)
vec_rotated = nm.dot(vec, mtx)

displacement = vec_rotated - vec

return displacement
```

The function needs to be registered in *Functions*:

```
functions = {
    'rotate_yz' : (rotate_yz,),
}
```

Periodic Boundary Conditions

The periodic boundary conditions tie DOFs of a single variable in two regions that have matching nodes. Can be used with functions in *sfepy.discrete.fem.periodic*.

Periodic boundary conditions:

```
epbcs = {
    <name> : ((<region1_name>, <region2_name>), [<times_specification>],
             {<dof_specification> : <dof_specification>[,
               <dof_specification> : <dof_specification>, ...]},
             <match_function_name>)
```

Example:

```
epbcs = {
    'up1' : (('Left', 'Right'), {'u.all' : 'u.all', 'p.0' : 'p.0'},
            'match_y_line'),
}
```

Linear Combination Boundary Conditions

The linear combination boundary conditions (LCBCs) are more general than the Dirichlet BCs or periodic BCs. They can be used to substitute one set of DOFs in a region by another set of DOFs, possibly in another region and of another variable. The LCBCs can be used only in FEM with nodal (Lagrange) basis.

Available LCBC kinds:

- 'rigid' - in linear elasticity problems, a region moves as a rigid body;
- 'no_penetration' - in flow problems, the velocity vector is constrained to the plane tangent to the surface;
- 'normal_direction' - the velocity vector is constrained to the normal direction;
- 'edge_direction' - the velocity vector is constrained to the mesh edge direction;

- 'integral_mean_value' - all DOFs in a region are summed to a single new DOF;
- 'shifted_periodic' - generalized periodic BCs that work with two different variables and can have a non-zero mutual shift;
- 'match_dofs' - tie DOFs of two fields.

Only the 'shifted_periodic' LCBC needs the second region and the DOF mapping function, see below.

Linear combination boundary conditions:

```
lcbcs = {
    'shifted' : (('Left', 'Right'),
                {'u1.all' : 'u2.all'},
                'match_y_line', 'shifted_periodic',
                'get_shift'),
    'mean' : ('Middle', {'u1.all' : None}, None, 'integral_mean_value'),
}
```

Initial Conditions

Initial conditions are applied prior to the boundary conditions - no special care must be used for the boundary dofs:

```
ics = {
    <name> : (<region_name>, {<dof_specification> : <value>[,
                                <dof_specification> : <value>, ...]},...))
}
```

Example:

```
ics = {
    'ic' : ('Omega', {'T.0' : 5.0}),
}
```

Materials

Materials are used to define constitutive parameters (e.g. stiffness, permeability, or viscosity), and other non-field arguments of terms (e.g. known traction or volume forces). Depending on a particular term, the parameters can be constants, functions defined over FE mesh nodes, functions defined in the elements, etc.

Example:

```
material = {
    'm' : ({'val' : [0.0, -1.0, 0.0]}),
    'm2' : 'get_pars',
    'm3' : (None, 'get_pars'), # Same as the above line.
}
```

Example: different material parameters in regions 'Yc', 'Ym':

```
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
dim = 3
materials = {
    'mat' : ({'D' : {
```

(continues on next page)

(continued from previous page)

```
'Ym': stiffness_from_youngpoisson(dim, 7.0e9, 0.4),
'Yc': stiffness_from_youngpoisson(dim, 70.0e9, 0.2)}
},),
}
```

Defining Material Parameters by Functions

The functions for defining material parameters can work in two modes, distinguished by the *mode* argument. The two modes are 'qp' and 'special'. The first mode is used for usual functions that define parameters in quadrature points (hence 'qp'), while the second one can be used for special values like various flags.

The shape and type of data returned in the 'special' mode can be arbitrary (depending on the term used). On the other hand, in the 'qp' mode all the data have to be numpy float64 arrays with shape (n_coord, n_row, n_col) , where n_coord is the number of quadrature points given by the *coors* argument, $n\_coord = coors.shape[0]$, and (n_row, n_col) is the shape of a material parameter in each quadrature point. For example, for scalar parameters, the shape is $(n\_coord, 1, 1)$. The shape is determined by each term.

Example:

```
def get_pars(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        val = coors[:,0]
        val.shape = (coors.shape[0], 1, 1)

    return {'x_coord' : val}
```

The function needs to be registered in *Functions*:

```
functions = {
    'get_pars' : (get_pars,),
}
```

If a material parameter has the same value in all quadrature points, than it is not necessary to repeat the constant and the array can be with shape $(1, n\_row, n\_col)$.

Equations and Terms

Equations can be built by combining terms listed in *Term Table*.

Examples

- Laplace equation, named integral:

```
equations = {
    'Temperature' : ""dw_laplace.i.Omega( coef.val, s, t ) = 0""
}
```

- Laplace equation, simplified integral given by order:


```
equations = {
    'Temperature' : """dw_laplace.2.Omega( coef.val, s, t ) = 0"""
}
```

- Laplace equation, automatic integration order (not implemented yet!):

```
equations = {
    'Temperature' : """dw_laplace.a.Omega( coef.val, s, t ) = 0"""
}
```

- Navier-Stokes equations:

```
equations = {
    'balance' :
        """+ dw_div_grad.i2.Omega( fluid.viscosity, v, u )
          + dw_convect.i2.Omega( v, u )
          - dw_stokes.i1.Omega( v, p ) = 0""",
    'incompressibility' :
        """dw_stokes.i1.Omega( u, q ) = 0""",
}
```

Configuring Solvers

In SfePy, a non-linear solver has to be specified even when solving a linear problem. The linear problem is/should be then solved in one iteration of the nonlinear solver.

Linear and nonlinear solver:

```
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
               {'i_max' : 1}),
}
```

Solver selection:

```
options = {
    'nls' : 'newton',
    'ls' : 'ls',
}
```

For the case that a chosen linear solver is not available, it is possible to define the `fallback` option of the chosen solver which specifies a possible alternative:

```
solvers = {
    'ls': ('ls.mumps', {'fallback': 'ls2'}),
    'ls2': ('ls.scipy_umfpack', {}),
    'newton': ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}
```

Another possibility is to use a “virtual” solver that ensures an automatic selection of an available solver, see [Virtual Linear Solvers with Automatic Selection](#).

Functions

Functions are a way of customizing *SfePy* behavior. They make it possible to define material properties, boundary conditions, parametric sweeps, and other items in an arbitrary manner. Functions are normal Python functions declared in the Problem Definition file, so they can invoke the full power of Python. In order for *SfePy* to make use of the functions, they must be declared using the function keyword. See the examples below, and also the corresponding sections above.

Examples

See `sfePy/examples/diffusion/poisson_functions.py` for a complete problem description file demonstrating how to use different kinds of functions.

- functions for defining regions:

```
def get_circle(coors, domain=None):
    r = nm.sqrt(coors[:,0]**2.0 + coors[:,1]**2.0)
    return nm.where(r < 0.2)[0]

functions = {
    'get_circle' : (get_circle,),
}
```

- functions for defining boundary conditions:

```
def get_p_edge(ts, coors, bc=None, problem=None):
    if bc.name == 'p_left':
        return nm.sin(nm.pi * coors[:,1])
    else:
        return nm.cos(nm.pi * coors[:,1])

functions = {
    'get_p_edge' : (get_p_edge,),
}

ebcs = {
    'p' : ('Gamma', {'p.0' : 'get_p_edge'}),
}
```

The values can be given by a function of time, coordinates and possibly other data, for example:

```
ebcs = {
    'f1' : ('Gamma1', {'u.0' : 'get_ebc_x'}),
    'f2' : ('Gamma2', {'u.all' : 'get_ebc_all'}),
}

def get_ebc_x(coors, amplitude):
    z = coors[:, 2]
    val = amplitude * nm.sin(z * 2.0 * nm.pi)
    return val
```

(continues on next page)

(continued from previous page)

```
def get_ebc_all(ts, coors):
    val = ts.step * coors
    return val

functions = {
    'get_ebc_x' : (lambda ts, coors, bc, problem, **kwargs:
                   get_ebc_x(coors, 5.0)),
    'get_ebc_all' : (lambda ts, coors, bc, problem, **kwargs:
                     get_ebc_all(ts, coors)),
}
```

Note that when setting more than one component as in `get_ebc_all()` above, the function should return either an array of shape `(coors.shape[0], n_components)`, or the same array flattened to 1D row-by-row (i.e. node-by-node), where `n_components` corresponds to the number of components in the boundary condition definition. For example, with `'u.[0, 1]'`, `n_components` is 2.

- function for defining usual material parameters, using `mode=='qp'` seen above in *Defining Material Parameters by Functions*:

```
def get_pars(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        val = coors[:,0]
        val.shape = (coors.shape[0], 1, 1)

        return {'x_coor' : val}

functions = {
    'get_pars' : (get_pars,),
}
```

The keyword arguments contain both additional use-specified arguments, if any, and the following: `equations`, `term`, `problem`, for cases when the function needs access to the equations, problem, or term instances that requested the parameters that are being evaluated. The full signature of the function is:

```
def get_pars(ts, coors, mode=None,
             equations=None, term=None, problem=None, **kwargs)
```

- function for defining special material parameters, with an extra argument:

```
def get_pars_special(ts, coors, mode=None, extra_arg=None):
    if mode == 'special':
        if extra_arg == 'hello!':
            ic = 0
        else:
            ic = 1
        return {'x_%s' % ic : coors[:,ic]}

functions = {
    'get_pars1' : (lambda ts, coors, mode=None, **kwargs:
                   get_pars_special(ts, coors, mode,
                                   extra_arg='hello!')),
}
```

(continues on next page)

(continued from previous page)

```
# Just another way of adding a function, besides 'functions' keyword.
function_1 = {
    'name' : 'get_pars2',
    'function' : lambda ts, coors, mode=None, **kwargs:
        get_pars_special(ts, coors, mode, extra_arg='hi!'),
}
```

- function combining both kinds of material parameters:

```
def get_pars_both(ts, coors, mode=None, **kwargs):
    out = {}

    if mode == 'special':

        out['flag'] = coors.max() > 1.0

    elif mode == 'qp':

        val = coors[:,1]
        val.shape = (coors.shape[0], 1, 1)

        out['y_coor'] = val

    return out

functions = {
    'get_pars_both' : (get_pars_both,),
}
```

- function for setting values of a parameter variable:

```
variables = {
    'p' : ('parameter field', 'temperature',
          {'setter' : 'get_load_variable'}),
}

def get_load_variable(ts, coors, region=None, variable=None, **kwargs):
    y = coors[:,1]
    val = 5e5 * y
    return val

functions = {
    'get_load_variable' : (get_load_variable,)
}
```

Miscellaneous

The options can be used to select solvers, output file format, output directory, to register functions to be called at various phases of the solution (the *hooks*), and for other settings.

Additional options (including solver selection):

```
options = {
    # int >= 0, uniform mesh refinement level
    'refinement_level' : 0,

    # bool, default: False, if True, allow selecting empty regions with no
    # entities
    'allow_empty_regions' : True,

    # string, output directory
    'output_dir' : 'output/<output_dir>',

    # 'vtk' or 'h5', output file (results) format
    'output_format' : 'vtk',

    # output file format variant compatible with 'output_format'
    'file_format' : 'vtk-ascii',

    # string, nonlinear solver name
    'nls' : 'newton',

    # string, linear solver name
    'ls' : 'ls',

    # string, time stepping solver name
    'ts' : 'ts',

    # The times at which results should be saved:
    # - a sequence of times
    # - or 'all' for all time steps (the default value)
    # - or an int, number of time steps, spaced regularly from t0 to t1
    # - or a function 'is_save(ts)'
    'save_times' : 'all',

    # save a restart file for each time step, only the last computed time
    # step restart file is kept.
    'save_restart' : -1,

    # string, a function to be called after each time step
    'step_hook' : '<step_hook_function>',

    # string, a function to be called after each time step, used to
    # update the results to be saved
    'post_process_hook' : '<post_process_hook_function>',

    # string, as above, at the end of simulation
    'post_process_hook_final' : '<post_process_hook_final_function>',
```

(continues on next page)

(continued from previous page)

```

# string, a function to generate probe instances
'gen_probes' : '<gen_probes_function>',

# string, a function to probe data
'probe_hook' : '<probe_hook_function>',

# string, a function to modify problem definition parameters
'parametric_hook' : '<parametric_hook_function>',

# float, default: 1e-9. If the distance between two mesh vertices
# is less than this value, they are considered identical.
# This affects:
# - periodic regions matching
# - mirror regions matching
# - fixing of mesh doubled vertices
'mesh_eps': 1e-7,

# bool, default: True. If True, the (tangent) matrices and residual
# vectors (right-hand sides) contain only active DOFs, otherwise all
# DOFs (including the ones fixed by the Dirichlet or periodic boundary
# conditions) are included. Note that the rows/columns corresponding to
# fixed DOFs are modified w.r.t. a problem without the boundary
# conditions.
'active_only' : False,

# If active_only is False, i.e. all DOFs are included in tangent
# matrices, the E(P)BC rows and columns are by default set to zeros
# everywhere except the diagonal, which is set to one. Set this to
# False to do nothing instead, for example when manually scaling matrix
# blocks in a multi-physical problem by their largest value magnitude.
'not_active_only_modify_matrix' : True,

# bool, default: False. If True, all DOF connectivities are used to
# pre-allocate the matrix graph. If False, only cell region
# connectivities are used.
'any_dof_conn' : False,

# bool, default: False. If True, automatically transform equations to a
# form suitable for the given solver. Implemented for
# ElastodynamicsBaseTS-based solvers
'auto_transform_equations' : True,

# The maximum number of cells added to the matrix graph together.
'graph_cell_chunk_size' : 1000000,
}

```

- `post_process_hook` enables computing derived quantities, like stress or strain, from the primary unknown variables. See the examples in `sfePy/examples/large_deformation/` directory.
- `parametric_hook` makes it possible to run parametric studies by modifying the problem description programmatically. See `sfePy/examples/diffusion/poisson_parametric_study.py` for an example.
- `output_dir` redirects output files to specified directory

1.4.4 Building Equations in SfePy

Equations in *SfePy* are built using terms, which correspond directly to the integral forms of weak formulation of a problem to be solved. As an example, let us consider the Laplace equation in time interval $t \in [0, t_{\text{final}}]$:

$$\frac{\partial T}{\partial t} + c \Delta T = 0 \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma. \quad (1.5)$$

The weak formulation of (1.5) is: Find $T \in V$, such that

$$\int_{\Omega} s \frac{\partial T}{\partial t} + \int_{\Omega} c \nabla T : \nabla s = 0, \quad \forall s \in V_0, \quad (1.6)$$

where we assume no fluxes over $\partial\Omega \setminus \Gamma$. In the syntax used in *SfePy* input files, this can be written as:

```
dw_dot.i.Omega( s, dT/dt ) + dw_laplace.i.Omega( coef, s, T) = 0
```

which directly corresponds to the discrete version of (1.6): Find $T \in V_h$, such that

$$\mathbf{s}^T \left(\int_{\Omega_h} \boldsymbol{\phi}^T \boldsymbol{\phi} \right) \frac{\partial \mathbf{T}}{\partial t} + \mathbf{s}^T \left(\int_{\Omega_h} c \mathbf{G}^T \mathbf{G} \right) \mathbf{T} = 0, \quad \forall \mathbf{s} \in V_{h0},$$

where $u \approx \boldsymbol{\phi} \mathbf{u}$, $\nabla u \approx \mathbf{G} \mathbf{u}$ for $u \in \{s, T\}$. The integrals over the discrete domain Ω_h are approximated by a numerical quadrature, that is named **i** in our case.

Syntax of Terms in Equations

The terms in equations are written in form:

```
<term_name>.<i>.<r>( <arg1>, <arg2>, ... )
```

where **<i>** denotes an integral name (i.e. a name of numerical quadrature to use) and **<r>** marks a region (domain of the integral). In the following, **<virtual>** corresponds to a test function, **<state>** to a unknown function and **<parameter>** to a known function arguments.

When solving, the individual terms in equations are evaluated in the ‘*weak*’ mode. The evaluation modes are described in the next section.

1.4.5 Term Evaluation

Terms can be evaluated in two ways:

1. implicitly by using them in equations;
2. explicitly using `Problem.evaluate()`. This way is mostly used in the postprocessing.

Each term supports one or more *evaluation modes*:

- ‘*weak*’: Assemble (in the finite element sense) either the vector or matrix depending on *diff_var* argument (the name of variable to differentiate with respect to) of `Term.evaluate()`. This mode is usually used implicitly when building the linear system corresponding to given equations.
- ‘*eval*’: The evaluation mode integrates the term (= integral) over a region. The result has the same dimension as the quantity being integrated. This mode can be used, for example, to compute some global quantities during postprocessing such as fluxes or total values of extensive quantities (mass, volume, energy, ...).
- ‘*el_eval*’: The element evaluation mode results in an array of a quantity integrated over each element of a region.

- `'el_avg'` : The element average mode results in an array of a quantity averaged in each element of a region. This is the mode for postprocessing.
- `'qp'` : The quadrature points mode results in an array of a quantity interpolated into quadrature points of each element in a region. This mode is used when further point-wise calculations with the result are needed. The same element type and number of quadrature points in each element are assumed.

Not all terms support all the modes - consult the documentation of the individual terms. There are, however, certain naming conventions:

- `'dw_*'` terms support `'weak'` mode
- `'dq_*'` terms support `'qp'` mode
- `'d_*'`, `'di_*'` terms support `'eval'` and `'el_eval'` modes
- `'ev_*'` terms support `'eval'`, `'el_eval'`, `'el_avg'` and `'qp'` modes

Note that the naming prefixes are due to history when the `mode` argument to `Problem.evaluate()` and `Term.evaluate()` was not available. Now they are often redundant, but are kept around to indicate the evaluation purpose of each term.

Several examples of using the `Problem.evaluate()` function are shown below.

1.4.6 Solution Postprocessing

A solution to equations given in a problem description file is given by the variables of the 'unknown field' kind, that are set in the solution procedure. By default, those are the only values that are stored into a results file. The solution postprocessing allows computing additional, derived, quantities, based on the primary variables values, as well as any other quantities to be stored in the results.

Let us illustrate this using several typical examples. Let us assume that the postprocessing function is called `'post_process()'`, and is added to options as discussed in *Miscellaneous*, see `'post_process_hook'` and `'post_process_hook_final'`. Then:

- compute stress and strain given the displacements (variable `u`):

```
def post_process(out, problem, variables, extend=False):
    """
    This will be called after the problem is solved.

    Parameters
    -----
    out : dict
        The output dictionary, where this function will store additional
        data.
    problem : Problem instance
        The current Problem instance.
    variables : Variables instance
        The computed state, containing FE coefficients of all the unknown
        variables.
    extend : bool
        The flag indicating whether to extend the output data to the whole
        domain. It can be ignored if the problem is solved on the whole
        domain already.

    Returns
```

(continues on next page)

(continued from previous page)

```

-----
out : dict
    The updated output dictionary.
"""
from sfepy.base.base import Struct

# Cauchy strain averaged in elements.
strain = problem.evaluate('ev_cauchy_strain.i.Omega(u)',
                          mode='el_avg')
out['cauchy_strain'] = Struct(name='output_data',
                             mode='cell', data=strain,
                             dofs=None)

# Cauchy stress averaged in elements.
stress = problem.evaluate('ev_cauchy_stress.i.Omega(solid.D, u)',
                          mode='el_avg')
out['cauchy_stress'] = Struct(name='output_data',
                              mode='cell', data=stress,
                              dofs=None)

return out

```

The full example is *linear_elasticity/linear_elastic_probes.py*.

- compute diffusion velocity given the pressure:

```

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    dvel = pb.evaluate('ev_diffusion_velocity.i.Omega(m.K, p)',
                      mode='el_avg')
    out['dvel'] = Struct(name='output_data',
                        mode='cell', data=dvel, dofs=None)

    return out

```

The full example is *biot-biot_npbc*.

- store values of a non-homogeneous material parameter:

```

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    mu = pb.evaluate('ev_integrate_mat.2.Omega(nonlinear.mu, u)',
                    mode='el_avg', copy_materials=False, verbose=False)
    out['mu'] = Struct(name='mu', mode='cell', data=mu, dofs=None)

    return out

```

The full example is *linear_elasticity/material_nonlinearity.py*.

- compute volume of a region (*u* is any variable defined in the region *Omega*):

```

volume = problem.evaluate('ev_volume.2.Omega(u)')

```

1.4.7 Probing

Probing applies interpolation to output the solution along specified paths. There are two ways of probing:

- *VTK* probes: It is the simple way of probing using the `'post_process_hook'`. It generates matplotlib figures with the probing results and previews of the mesh with the probe paths. See [Primer](#) or [linear_elasticity/its2D_5.py](#) example.
- *SfePy* probes: As mentioned in [Miscellaneous](#), it relies on defining two additional functions, namely the `'gen_probes'` function, that should create the required probes (see [sfepy.discrete.probes](#)), and the `'probe_hook'` function that performs the actual probing of the results for each of the probes. This function can return the probing results, as well as a handle to a corresponding matplotlib figure. See [linear_elasticity/its2D_4.py](#) for additional explanation.

Using [sfepy.discrete.probes](#) allows correct probing of fields with the approximation order greater than one, see [Interactive Example](#) in [Primer](#) or [linear_elasticity/its2D_interactive.py](#) for an example of interactive use.

1.4.8 Postprocessing filters

The following postprocessing functions based on the *VTK* filters are available:

- `'get_vtk_surface'`: extract mesh surface
- `'get_vtk_edges'`: extract mesh edges
- `'get_vtk_by_group'`: extract domain by a material ID
- `'tetrahedralize_vtk_mesh'`: 3D cells are converted to tetrahedral meshes, 2D cells to triangles

The following code demonstrates the use of the postprocessing filters:

```
mesh = problem.domain.mesh
mesh_name = mesh.name[mesh.name.rfind(osp.sep) + 1:]

vtkdata = get_vtk_from_mesh(mesh, out, 'postproc_')
matrix = get_vtk_by_group(vtkdata, 1, 1)

matrix_surf = get_vtk_surface(matrix)
matrix_surf_tri = tetrahedralize_vtk_mesh(matrix_surf)
write_vtk_to_file('%s_mat1_surface.vtk' % mesh_name, matrix_surf_tri)

matrix_edges = get_vtk_edges(matrix)
write_vtk_to_file('%s_mat1_edges.vtk' % mesh_name, matrix_edges)
```

1.4.9 Solvers

This section describes the *time-stepping*, *nonlinear*, *linear*, *eigenvalue* and *optimization* solvers available in *SfePy*. There are many internal and external solvers in the [sfepy.solvers package](#) that can be called using a uniform interface.

Time-stepping solvers

All PDEs that can be described in a problem description file are solved internally by a time-stepping solver. This holds even for stationary problems, where the default single-step solver ('`ts.stationary`') is created automatically. In this way, all problems are treated in a uniform way. The same holds when building a problem interactively, or when writing a script, whenever the `Problem.solve()` function is used for a problem solution.

The following solvers are available:

- `ts.adaptive`: Implicit time stepping solver with an adaptive time step.
- `ts.bathe`: Solve elastodynamics problems by the Bathe method.
- `ts.central_difference`: Solve elastodynamics problems by the explicit central difference method.
- `ts.euler`: Simple forward euler method
- `ts.generalized_alpha`: Solve elastodynamics problems by the generalized α method.
- `ts.multistaged`: Explicit time stepping solver with multistage solve_step method
- `ts.newmark`: Solve elastodynamics problems by the Newmark method.
- `ts.runge_kutta_4`: Classical 4th order Runge-Kutta method,
- `ts.simple`: Implicit time stepping solver with a fixed time step.
- `ts.stationary`: Solver for stationary problems without time stepping.
- `ts.tvd_runge_kutta_3`: 3rd order Total Variation Diminishing Runge-Kutta method
- `ts.velocity_verlet`: Solve elastodynamics problems by the explicit velocity-Verlet method.

See `sfePy.solvers.ts_solvers` for available *time-stepping* solvers and their options.

The following time step controllers are available:

- `tsc.ed_basic`: Adaptive time step I-controller for elastodynamics.
- `tsc.ed_linear`: Adaptive time step controller for elastodynamics and linear problems.
- `tsc.ed_pid`: Adaptive time step PID controller for elastodynamics.
- `tsc.fixed`: Fixed (do-nothing) time step controller.
- `tsc.time_sequence`: Given times sequence time step controller.

See `sfePy.solvers.ts_controllers` for available time step controllers and their options.

Nonlinear Solvers

Almost every problem, even linear, is solved in *SfePy* using a nonlinear solver that calls a linear solver in each iteration. This approach unifies treatment of linear and non-linear problems, and simplifies application of Dirichlet (essential) boundary conditions, as the linear system computes not a solution, but a solution increment, i.e., it always has zero boundary conditions.

The following solvers are available:

- `nls.newton`: Solves a nonlinear system $f(x) = 0$ using the Newton method.
- `nls.oseen`: The Oseen solver for Navier-Stokes equations.
- `nls.petsc`: Interface to PETSc SNES (Scalable Nonlinear Equations Solvers).
- `nls.scipy_root`: Interface to `scipy.optimize.root()`.
- `nls.semismooth_newton`: The semi-smooth Newton method.

See `sfePy.solvers.nls`, `sfePy.solvers.oseen` and `sfePy.solvers.semismooth_newton` for all available *nonlinear* solvers and their options.

Linear Solvers

Choosing a suitable linear solver is key to solving efficiently stationary as well as transient PDEs. *SfePy* allows using a number of external solvers with a unified interface.

The following solvers are available:

- `ls.cholesky`: Interface to scikit-sparse.cholesky solver.
- `ls.cm_pb`: Conjugate multiple problems.
- `ls.mumps`: Interface to MUMPS solver.
- `ls.petsc`: PETSc Krylov subspace solver.
- `ls.pyamg`: Interface to PyAMG solvers.
- `ls.pyamg_krylov`: Interface to PyAMG Krylov solvers.
- `ls.pypardiso`: PyPardiso direct solver.
- `ls.rmm`: Special solver for explicit transient elastodynamics.
- `ls.schur_mumps`: Mumps Schur complement solver.
- `ls.scipy_direct`: Direct sparse solver from SciPy.
- `ls.scipy_iterative`: Interface to SciPy iterative solvers.
- `ls.scipy_superlu`: SuperLU - direct sparse solver from SciPy.
- `ls.scipy_umfpack`: UMFPACK - direct sparse solver from SciPy.

See `sfePy.solvers.ls` for all available *linear* solvers and their options.

Virtual Linear Solvers with Automatic Selection

A “virtual” solver can be used in case it is not clear which external linear solvers are available. Each “virtual” solver selects the first available solver from a pre-defined list.

The following solvers are available:

- `ls.auto_direct`: The automatically selected linear direct solver.
- `ls.auto_iterative`: The automatically selected linear iterative solver.

See `sfePy.solvers.auto_fallback` for all available *virtual* solvers.

Eigenvalue Problem Solvers

The following eigenvalue problem solvers are available:

- `eig.matlab`: Matlab eigenvalue problem solver.
- `eig.octave`: Octave eigenvalue problem solver.
- `eig.primme`: PRIMME eigenvalue problem solver.
- `eig.scipy`: SciPy-based solver for both dense and sparse problems.
- `eig.scipy_lobpcg`: SciPy-based LOBPCG solver for sparse symmetric problems.

- `eig.sgscipy`: SciPy-based solver for dense symmetric problems.
- `eig.slepc`: General SLEPc eigenvalue problem solver.

See `sfepy.solvers.eigen` for available *eigenvalue problem* solvers and their options.

Quadratic Eigenvalue Problem Solvers

The following quadratic eigenvalue problem solvers are available:

- `eig.qevp`: Quadratic eigenvalue problem solver based on the problem linearization.

See `sfepy.solvers.qeigen` for available *quadratic eigenvalue problem* solvers and their options.

Optimization Solvers

The following optimization solvers are available:

- `nls.scipy_fmin_like`: Interface to SciPy optimization solvers `scipy.optimize.fmin_*`.
- `opt.fmin_sd`: Steepest descent optimization solver.

See `sfepy.solvers.optimize` for available *optimization* solvers and their options.

1.4.10 Solving Problems in Parallel

The PETSc-based nonlinear equations solver `'nls.petsc'` and linear system solver `'ls.petsc'` can be used for parallel computations, together with the modules in `sfepy.parallel package`. This feature is **very preliminary**, and can be used only with the commands for interactive use - problem description files are not supported (yet). The key module is `sfepy.parallel.parallel` that takes care of the domain and field DOFs distribution among parallel tasks, as well as parallel assembling to PETSc vectors and matrices.

Current Implementation Drawbacks

- The partitioning of the domain and fields DOFs is not done in parallel and all tasks need to load the whole mesh and define the global fields - those must fit into memory available to each task.
- While all KSP and SNES solver are supported, in principle, most of their options have to be passed using the command-line parameters of PETSc - they are not supported yet in the *SfePy* solver parameters.
- There are no performance statistics yet. The code was tested on a single multi-cpu machine only.
- The global solution is gathered to task 0 and saved to disk serially.
- The `vertices of surface` region selector does not work in parallel, because the region definition is applied to a task-local domain.

Examples

The examples demonstrating the use parallel problem solving in *SfePy* are:

- [*diffusion/poisson_parallel_interactive.py*](#)
- [*multi_physics/biot_parallel_interactive.py*](#)

See their help messages for further information.

1.4.11 Isogeometric Analysis

Isogeometric analysis (IGA) is a recently developed computational approach that allows using the NURBS-based domain description from CAD design tools also for approximation purposes similar to the finite element method.

The implementation in *SfePy* is based on Bezier extraction of NURBS as developed in¹. This approach allows reusing the existing finite element assembling routines, as still the evaluation of weak forms occurs locally in “elements” and the local contributions are then assembled to the global system.

Current Implementation

The IGA code is still very preliminary and some crucial components are missing. The current implementation is also very slow, as it is in pure Python.

The following already works:

- single patch tensor product domain support in 2D and 3D
- region selection based on topological Bezier mesh, see below
- Dirichlet boundary conditions using projections for non-constant values
- evaluation in arbitrary point in the physical domain
- both scalar and vector cell terms work
- term integration over the whole domain as well as a cell subdomain
- simple linearization (output file generation) based on sampling the results with uniform parametric vectors
- basic domain generation with `sfepy/scripts/gen_iga_patch.py` based on [*igakit*](#)

The following is not implemented yet:

- tests
- theoretical convergence rate verification
- facet terms
- other boundary conditions
- proper (adaptive) linearization for post-processing
- support for multiple NURBS patches

¹ Michael J. Borden, Michael A. Scott, John A. Evans, Thomas J. R. Hughes: Isogeometric finite element data structures based on Bezier extraction of NURBS, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, Texas, March 2010.

Domain Description

The domain description is in custom HDF5-based files with `.iga` extension. Such a file contains:

- NURBS patch data (knots, degrees, control points and weights). Those can either be generated using `igakit`, created manually or imported from other tools.
- Bezier extraction operators and corresponding DOF connectivity (computed by *SfePy*).
- Bezier mesh control points, weights and connectivity (computed by *SfePy*).

The Bezier mesh is used to create a topological Bezier mesh - a subset of the Bezier mesh containing the Bezier element corner vertices only. Those vertices are interpolatory (are on the exact geometry) and so can be used for region selections.

Region Selection

The domain description files contain vertex sets for regions corresponding to the patch sides, named `'xiIJ'`, where `I` is the parametric axis (0, 1, or 2) and `J` is 0 or 1 for the beginning and end of the axis knot span. Other regions can be defined in the usual way, using the topological Bezier mesh entities.

Examples

The examples demonstrating the use of IGA in *SfePy* are:

- `diffusion/poisson_iga.py`
- `linear_elasticity/linear_elastic_iga.py`
- `navier_stokes/navier_stokes2d_iga.py`

Their problem description files are almost the same as their FEM equivalents, with the following differences:

- There is `filename_domain` instead of `filename_mesh`.
- Fields are defined as follows:

```
fields = {
    't1' : ('real', 1, 'Omega', None, 'H1', 'iga'),
    't2' : ('real', 1, 'Omega', 'iga', 'H1', 'iga'),
    't3' : ('real', 1, 'Omega', 'iga+%d', 'H1', 'iga'),
}
```

The approximation order in the first definition is `None` as it is given by the NURBS degrees in the domain description. The second definition is equivalent to the first one. The third definition, where `%d` should be a non-negative integer, illustrates how to increase the field's NURBS degrees (while keeping the continuity) w.r.t. the domain NURBS description. It is applied in the `navier_stokes/navier_stokes2d_iga.py` example to the velocity field.

1.5 Examples

This section contains domain-specific tutorials as well as the automatically generated list of the standard examples that come with SfePy.

1.5.1 Primer

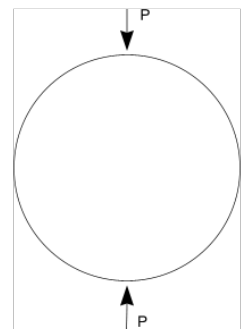
A beginner's tutorial highlighting the basics of *SfePy*.

Introduction

This primer presents a step-by-step walk-through of the process to solve a simple mechanics problem. The typical process to solve a problem using *SfePy* is followed: a model is meshed, a problem definition file is drafted, *SfePy* is run to solve the problem and finally the results of the analysis are visualised.

Problem statement

A popular test to measure the tensile strength of concrete or asphalt materials is the indirect tensile strength (ITS) test pictured below. In this test a cylindrical specimen is loaded across its diameter to failure. The test is usually run by loading the specimen at a constant deformation rate of 50 mm/minute (say) and measuring the load response. When the tensile stress that develops in the specimen under loading exceeds its tensile strength then the specimen will fail. To model this problem using finite elements the indirect tensile test can be simplified to represent a diametrically point loaded disk as shown in the schematic.



The tensile and compressive stresses that develop in the specimen as a result of the point loads P are a function of the diameter D and thickness t of the cylindrical specimen. At the centre of the specimen, the compressive stress is 3 times the tensile stress and the analytical formulation for these are, respectively:

$$\sigma_t = \frac{2P}{\pi t D} \quad (1.7)$$

$$\sigma_c = \frac{6P}{\pi t D} \quad (1.8)$$

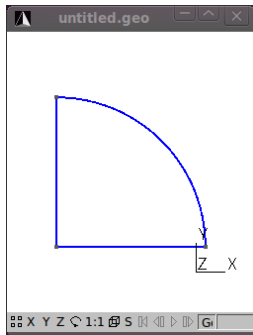
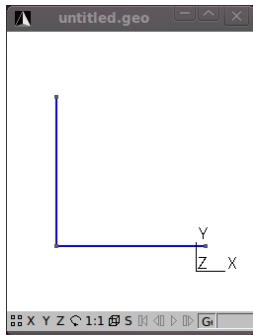
These solutions may be approximated using finite element methods. To solve this problem using *SfePy* the first step is meshing a suitable model.

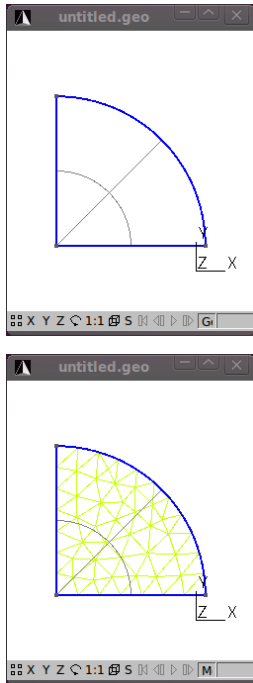
Meshing

Assuming plane strain conditions, the indirect tensile test may be modelled using a 2D finite element mesh. Furthermore, the geometry of the model is symmetrical about the x- and y-axes passing through the centre of the circle. To take advantage of this symmetry only one quarter of the 2D model will be meshed and boundary conditions will be established to indicate this symmetry. The meshing program *Gmsh* is used here to very quickly mesh the model. Follow these steps to model the ITS, creating the file *its2D.mesh* <../meshes/2d/its2D.mesh>:

1. The ITS specimen has a diameter of 150 mm. Using *Gmsh* add (geometry/Elementary Entities/Add) three new points at the following coordinates: (75.0, 0.0, 0.0), (0.0, 0.0, 0.0) and (0.0, 75.0, 0.0).
2. Next add two straight lines connecting the points.
3. Next add a Circle arc connecting two of the points to form the quarter circle segment.
4. Still under *Geometry/Elementary entities* add a Plane Surface.
5. With the geometry of the model defined, add a mesh by clicking on the 2D button under the Mesh functions.

The figures that follow show the various stages in the model process.

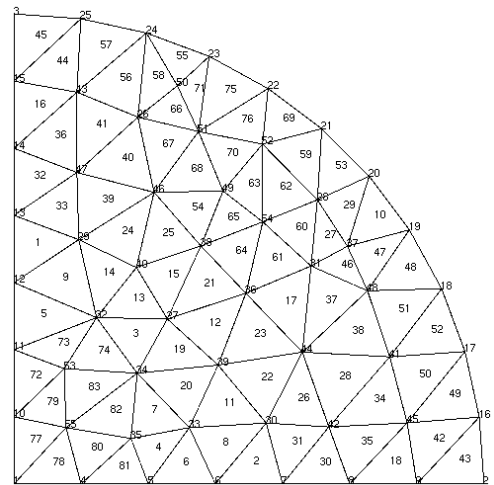




That's the meshing done. Save the mesh in a format that *SfePy* recognizes. For now use the *medit* *.mesh* format e.g. *its2D.mesh*.

Hint: Check the drop down in the *Save As* dialog for the different formats that *Gmsh* can save to.

If you open the *its2D.mesh* file using a text editor you'll notice that *Gmsh* saves the mesh in a 3D format and includes some extra geometry items that should be deleted. Reformatted the mesh file to a 2D format and delete the *Edges* block. Note that when you do this the file cannot be reopened by *Gmsh* so it is always a good idea to also save your meshes in *Gmsh*'s native format as well (Shift-Ctrl-S). Click [here](#) to download the reformatted mesh file that will be used in the tutorial.



You'll notice that the mesh contains 55 vertices (nodes) and 83 triangle elements. The mesh file provides the coordinates of the nodes and the element connectivity. It is important to note that node and element numbering in *SfePy* start at 0 and not 1 as is the case in *Gmsh* and some other meshing programs.

To view *.mesh* files you can use a demo of *medit*. After loading your mesh file with *medit* you can see the node and element numbering by pressing **P** and **F** respectively. The numbering in *medit* starts at 1 as shown. Thus the node at the center of the model in *SfePy* numbering is 0, and elements 76 and 77 are connected to this node. Node and

element numbers can also be viewed in *Gmsh* – under the *mesh* option under the *Visibility* tab enable the *node* and *surface* labels. Note that the surface labels as numbered in *Gmsh* follow on from the line numbering. So to get the corresponding element number in *SfePy* you'll need to subtract the number of lines in the *Gmsh* file + 1. Confused yet? Luckily, *SfePy* provides some useful mesh functions to indicate which elements are connected to which nodes. Nodes and elements can also be identified by defining regions, which is addressed later.

Another open source python option to view *.mesh* files is the appropriately named [Python Mesh Viewer](#).

The next step in the process is coding the *SfePy* problem definition file.

Problem description

The programming of the *problem description file* is well documented in the *SfePy User's Guide*. The problem description file used in the tutorial follows:

```
r"""
Diametrically point loaded 2-D disk. See :ref:`sec-primer`.

Find :math:\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""

from __future__ import absolute_import
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.utils import refine_mesh
from sfepy import data_dir

# Fix the mesh file name if you run this file outside the SfePy directory.
filename_mesh = data_dir + '/meshes/2d/its2D.mesh'

refinement_level = 0
filename_mesh = refine_mesh(filename_mesh, refinement_level)

output_dir = '.' # set this to a valid directory you have write access to

young = 20000.0 # Young's modulus [MPa]
poisson = 0.4 # Poisson's ratio

options = {
    'output_dir' : output_dir,
}

regions = {
    'Omega' : 'all',
```

(continues on next page)

(continued from previous page)

```

'Left' : ('vertices in (x < 0.001)', 'facet'),
'Bottom' : ('vertices in (y < 0.001)', 'facet'),
'Top' : ('vertex 2', 'vertex'),
}

materials = {
    'Asphalt' : ({'D': stiffness_from_youngpoisson(2, young, poisson)},),
    'Load' : ({'val' : [0.0, -1000.0]},),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.2.Omega(Asphalt.D, v, u)
        = dw_point_load.0.Top(Load.val, v)""",
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'XSym' : ('Bottom', {'u.1' : 0.0}),
    'YSym' : ('Left', {'u.0' : 0.0}),
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
    }),
}

```

Download the **Problem description** file and open it in your favourite Python editor. Note that you may wish to change the location of the output directory to somewhere on your drive. You may also need to edit the mesh file name. For the analysis we will assume that the material of the test specimen is linear elastic and isotropic. We define two material constants i.e. Young's modulus and Poisson's ratio. The material is assumed to be asphalt concrete having a Young's modulus of 2,000 MPa and a Poisson's ration of 0.4.

Note: Be consistent in your choice and use of units. In the tutorial we are using Newton (N), millimeters (mm) and megaPascal (MPa). The *sfePy.mechanics.units* module might help you in determining which derived units correspond to given basic units.

The following block of code defines regions on your mesh:

```

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),

```

(continues on next page)

(continued from previous page)

```
'Bottom' : ('vertices in (y < 0.001)', 'facet'),
'Top' : ('vertex 2', 'vertex'),
}
```

Four regions are defined:

1. *'Omega'*: all the elements in the mesh,
2. *'Left'*: the y-axis,
3. *'Bottom'*: the x-axis,
4. *'Top'*: the topmost node. This is where the load is applied.

Having defined the regions these can be used in other parts of your code. For example, in the definition of the boundary conditions:

```
ebcs = {
    'XSym' : ('Bottom', {'u.1' : 0.0}),
    'YSym' : ('Left', {'u.0' : 0.0}),
}
```

Now the power of the regions entity becomes apparent. To ensure symmetry about the x-axis, the vertical or y-displacement of the nodes in the *'Bottom'* region are prevented or set to zero. Similarly, for symmetry about the y-axis, any horizontal or displacement in the x-direction of the nodes in the *'Left'* region or y-axis is prevented.

The load is specified in terms of the *'Load'* material as follows:

```
materials = {
    'Asphalt' : ({
        'lam' : lame_from_youngpoisson(young, poisson)[0],
        'mu' : lame_from_youngpoisson(young, poisson)[1],
    },),
    'Load' : ({'val' : [0.0, -1000.0]}),
}
```

Note the dot in *'val'* – this denotes a special material value, i.e., a value that is not to be evaluated in quadrature points. The load is then applied in equations using the *'dw_point_load.0.Top(Load.val, v)'* term in the topmost node (region *'Top'*).

We provided the material constants in terms of Young's modulus and Poisson's ratio, but the linear elastic isotropic equation used requires as input Lamé's parameters. The *lame_from_youngpoisson()* function is thus used for conversion. Note that to use this function it was necessary to import the function into the code, which was done up front:

```
from sfepy.mechanics.matcoefs import lame_from_youngpoisson
```

Hint: Check out the *sfepy.mechanics.matcoefs* module for other useful material related functions.

That's it – we are now ready to solve the problem.

Running SfePy

One option to solve the problem is to run the `sfepy-run` command: from the command line:

```
sfepy-run its2D_1.py
```

Note: For the purpose of this tutorial it is assumed that the *problem description file* (`its2D_1.py`) is in the current directory. If you have the `its2D_1.py` file in another directory then make sure you include the path to this file as well.

SfePy solves the problem and outputs the solution to the output path (`output_dir`) provided in the script. The output file will be in the VTK format by default if this is not explicitly specified and the name of the output file will be the same as that used for the mesh file except with the `.vtk` extension i.e. `its2D.vtk`.

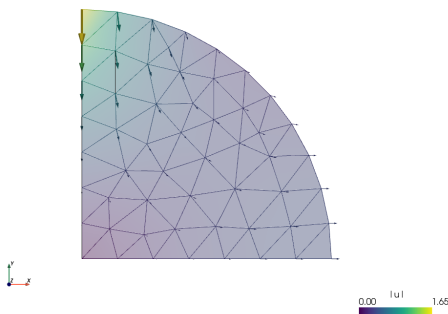
The VTK format is an ASCII format. Open the file using a text editor. You'll notice that the output file includes separate sections:

- POINTS (these are the model nodes),
- CELLS (the model element connectivity),
- VECTORS (the node displacements in the x-, y- and z- directions).

SfePy provides a script to quickly view the solution. To run this script you need to have `pyvista` installed. From the command line issue the following (assuming the correct paths):

```
sfepy-view its2D.vtk -2 -e
```

The `sfepy-view` command generates the image shown below, which shows by default the displacements in the model as arrows and their magnitude as color scale. Cool, but we are more interested in the stresses. To get these we need to modify the problem description file and do some post-processing.



Post-processing

SfePy provides functions to calculate stresses and strains. We'll include a function to calculate these and update the problem material definition and options to call this function as a `post_process_hook()`. Save this file as `its2D_2.py`.

```
r"""
Diametrically point loaded 2-D disk with postprocessing. See
:ref:`sec-primer`.

Find :math:\ul{u} such that:

.. math::
```

(continues on next page)

(continued from previous page)

```

\int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{u}) \epsilon_{kl}(\mathbf{u})
= 0
\;; \quad \forall \mathbf{v} \;;

```

where

```

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;;

```

```

"""

from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg',
                copy_materials=False)

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                  data=strain, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                   data=stress, dofs=None)

    return out

asphalt = materials['Asphalt'][0]
asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
options.update({'post_process_hook' : 'stress_strain',})

```

The updated file imports all of the previous definitions in `its2D_1.py`. The stress function (`de_cauchy_stress()`) requires as input the stiffness tensor – thus it was necessary to update the materials accordingly. The problem options were also updated to call the `stress_strain()` function as a `post_process_hook()`.

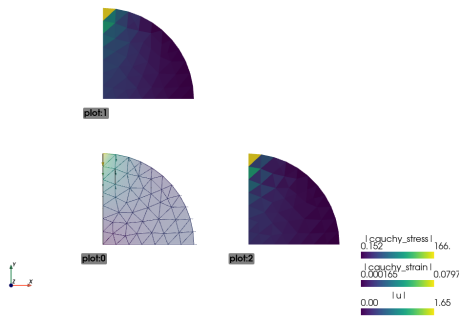
Run *SfePy* to solve the updated problem and view the solution (assuming the correct paths):

```

sfepy-run its2D_2.py
sfepy-view its2D.vtk -2 --max-plots 2

```

In addition to the node displacements, the VTK output shown below now also includes the stresses and strains averaged in the elements:



Remember the objective was to determine the stresses at the centre of the specimen under a load P . The solution as currently derived is expressed in terms of a global displacement vector u . The global (residual) force vector f is a function of the global displacement vector and the global stiffness matrix K as: $f = Ku$. Let's determine the force vector interactively.

Running SfePy in interactive mode

In addition to solving problems using the `sfepy-run` command you can also run *SfePy* interactively (we will use [IPython](#) interactive shell in following examples).

In the *SfePy* top-level directory run

```
ipython
```

issue the following commands:

```
In [1]: from sfepy.applications import solve_pde
```

```
In [2]: pb, variables = solve_pde('its2D_2.py')
```

The problem is solved and the problem definition and solution are provided in the `pb` and `variables` variables respectively. The solution, or in this case, the global displacement vector u , contains the x- and y-displacements at the nodes in the 2D model:

```
In [3]: u = variables()
```

```
In [4]: u
```

```
Out[4]:
array([[ 0.,          0.,          0.37376671, ..., -0.19923848,
         0.08820237, -0.11201528]])
```

```
In [5]: u.shape
```

```
Out[5]: (110,)
```

```
In [6]: u.shape = (55, 2)
```

```
In [7]: u
```

```
Out[7]:
array([[ 0.,          0.],
       [ 0.37376671,  0.],
       [ 0.,         -1.65318152],
```

(continues on next page)

(continued from previous page)

```
...,
[ 0.08716448, -0.23069047],
[ 0.27741356, -0.19923848],
[ 0.08820237, -0.11201528]])
```

Note: We have used the fact, that the state vector contains only one variable (u). In general, the following can be used:

```
In [8]: u = variables.get_state_parts()['u']
```

```
In [9]: u
```

```
Out[9]:
```

```
array([[ 0.          ,  0.          ],
       [ 0.37376671,  0.          ],
       [ 0.          , -1.65318152],
       ...,
       [ 0.08716448, -0.23069047],
       [ 0.27741356, -0.19923848],
       [ 0.08820237, -0.11201528]])
```

Both `variables()` and `variables.get_state_parts()` return a view of the DOF vector, that is why in Out[8] the vector is reshaped according to Out[6]. It is thus possible to set the values of state variables by manipulating the state vector, but shape changes such as the one above are not advised (see In [15] below) - work on a copy instead.

From the above it can be seen that u holds the displacements at the 55 nodes in the model and that the displacement at node 2 (on which the load is applied) is $(0, -1.65318152)$. The global stiffness matrix is saved in `pb` as a [sparse matrix](#):

```
In [10]: K = pb.mtx_a
```

```
In [11]: K
```

```
Out[11]:
```

```
<94x94 sparse matrix of type '<type 'numpy.float64'>'
      with 1070 stored elements in Compressed Sparse Row format>
```

```
In [12]: print(K)
```

```
(0, 0)      2443.95959851
(0, 7)      -2110.99917491
(0, 14)     -332.960423597
(0, 15)     1428.57142857
(1, 1)      2443.95959852
(1, 13)     -2110.99917492
(1, 32)     1428.57142857
(1, 33)     -332.960423596
(2, 2)      4048.78343529
(2, 3)     -1354.87004384
(2, 52)     -609.367453538
(2, 53)     -1869.0018791
(2, 92)     -357.41672785
(2, 93)     1510.24654193
(3, 2)     -1354.87004384
(3, 3)     4121.03202907
(3, 4)     -1696.54911732
(3, 48)     76.2400806561
```

(continues on next page)

(continued from previous page)

```

(3, 49)      -1669.59247304
(3, 52)      -1145.85294856
(3, 53)      2062.13955556
(4, 3)       -1696.54911732
(4, 4)       4410.17902905
(4, 5)       -1872.87344838
(4, 42)      -130.515009576
:           :
(91, 81)     -1610.0550578
(91, 86)     -199.343680224
(91, 87)     -2330.41406097
(91, 90)     -575.80373408
(91, 91)     7853.23899229
(92, 2)      -357.41672785
(92, 8)      1735.59411191
(92, 50)     -464.976034459
(92, 51)     -1761.31189004
(92, 52)     -3300.45367361
(92, 53)     1574.59387937
(92, 88)     -250.325600254
(92, 89)     1334.11823335
(92, 92)     9219.18643706
(92, 93)     -2607.52659081
(93, 2)      1510.24654193
(93, 8)      -657.361661955
(93, 50)     -1761.31189004
(93, 51)     54.1134516246
(93, 52)     1574.59387937
(93, 53)     -315.793227627
(93, 88)     1334.11823335
(93, 89)     -4348.13351285
(93, 92)     -2607.52659081
(93, 93)     9821.16012014

```

In [13]: `K.shape`

Out[13]: (94, 94)

One would expect the shape of the global stiffness matrix K to be (110, 110) i.e. to have the same number of rows and columns as u . This matrix has been reduced by the fixed degrees of freedom imposed by the boundary conditions set at the nodes on symmetry axes. To restore the shape of the matrix, temporarily remove the imposed boundary conditions:

```
In [14]: pb.remove_bcs()
```

Note that the matrix is reallocated, so it contains only zeros at this moment.

Now we can calculate the force vector f :

```
In [15]: f = pb.evaluator.eval_residual(u)
```

This leads to:

```

ValueError: shape mismatch: value array of shape (55,2) could not be broadcast to_
↪ indexing result of shape (110,)

```

– the original shape of the DOF vector needs to be restored:

```
In [16]: u.shape = variables.vec.shape = (110,)
In [17]: f = pb.evaluator.eval_residual(u)

In [18]: f.shape
Out[18]: (110,)

In [19]: f
Out[19]:
array([ -4.73618436e+01,  1.42752386e+02,  1.56921124e-13, ...,
        -2.06057393e-13,  2.13162821e-14, -2.84217094e-14])
```

Remember to restore the original boundary conditions previously removed in step [14]:

```
In [20]: pb.time_update()
```

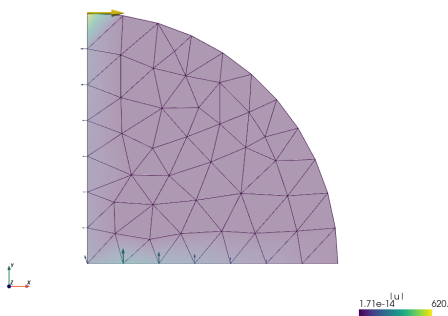
To view the residual force vector, we can save it to a VTK file. This requires setting f to (a copy of) the variables as follows:

```
In [21]: fvars = variables.copy() # Shallow copy, .vec is shared!
In [22]: fvars.init_state(f) # Initialize new .vec with f.
In [23]: out = fvars.create_output()
In [24]: pb.save_state('file.vtk', out=out)
```

Running the `sfepy-view` command on `file.vtk`, i.e.

```
In [25]: ! sfepy-view file.vtk
```

displays the average nodal forces as shown below:



The forces in the x- and y-directions at node 2 are:

```
In [26]: f.shape = (55, 2)
In [27]: f[2]
Out[27]: array([ 6.20373272e+02, -1.13686838e-13])
```

Great, we have an almost zero residual vertical load or force apparent at node 2 i.e. $-1.13686838 \times 10^{-13}$ Newton. Note that these relatively very small numbers can be slightly different, depending on the linear solver kind and version.

Let us now check the stress at node 0, the centre of the specimen.

Generating output at element nodes

Previously we had calculated the stresses in the model but these were averaged from those calculated at Gauss quadrature points within the elements. It is possible to provide custom integrals to allow the calculation of stresses with the Gauss quadrature points at the element nodes. This will provide us a more accurate estimate of the stress at the centre of the specimen located at node 0. The code below outlines one way to achieve this.

```
r"""
Diametrically point loaded 2-D disk with nodal stress calculation. See
:ref:`sec-primer`.

Find :math:\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""

from __future__ import print_function
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.geometry_element import geometry_data
from sfepy.discrete import FieldVariable
from sfepy.discrete.fem import Field
import numpy as nm

gdata = geometry_data['2_3']
nc = len(gdata.coors)

def nodal_stress(out, pb, state, extend=False, integrals=None):
    """
    Calculate stresses at nodal points.
    """

    # Point load.
    mat = pb.get_materials()['Load']
    P = 2.0 * mat.get_data('special', 'val')[1]

    # Calculate nodal stress.
    pb.time_update()

    if integrals is None: integrals = pb.get_integrals()

    stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D, u)', mode='qp',
```

(continues on next page)

(continued from previous page)

```

        integrals=integrals, copy_materials=False)
sfield = Field.from_args('stress', nm.float64, (3,),
                        pb.domain.regions['Omega'])
svar = FieldVariable('sigma', 'parameter', sfield,
                    primary_var_name='(set-to-None)')
svar.set_from_qp(stress, integrals['ivn'])

print('\n=====')
print('Given load = %.2f N' % -P)
print('\nAnalytical solution')
print('=====')
print('Horizontal tensile stress = %.5e MPa/mm' % (-2.*P/(nm.pi*150.)))
print('Vertical compressive stress = %.5e MPa/mm' % (-6.*P/(nm.pi*150.)))
print('\nFEM solution')
print('=====')
print('Horizontal tensile stress = %.5e MPa/mm' % (svar()[0]))
print('Vertical compressive stress = %.5e MPa/mm' % (-svar()[1]))
print('=====')
return out

asphalt = materials['Asphalt'][0]
asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
options.update({'post_process_hook' : 'nodal_stress',})

integrals = {
    'ivn' : ('custom', gdata.coors, [gdata.volume / nc] * nc),
}

```

The output:

```

=====
Given load = 2000.00 N

Analytical solution
=====
Horizontal tensile stress = 8.48826e+00 MPa/mm
Vertical compressive stress = 2.54648e+01 MPa/mm

FEM solution
=====
Horizontal tensile stress = 7.57220e+00 MPa/mm
Vertical compressive stress = 2.58660e+01 MPa/mm
=====

```

Not bad for such a coarse mesh! Re-running the problem using a finer mesh provides a more accurate solution:

```

=====
Given load = 2000.00 N

Analytical solution
=====
Horizontal tensile stress = 8.48826e+00 MPa/mm

```

(continues on next page)

(continued from previous page)

```
Vertical compressive stress = 2.54648e+01 MPa/mm
```

```
FEM solution
```

```
=====
```

```
Horizontal tensile stress = 8.50042e+00 MPa/mm
```

```
Vertical compressive stress = 2.54300e+01 MPa/mm
```

To see how the FEM solution approaches the analytical one, try to play with the uniform mesh refinement level in the *Problem description* file, namely lines 25, 26:

```
refinement_level = 0
filename_mesh = refine_mesh(filename_mesh, refinement_level)
```

The above computation could also be done in the IPython shell:

```
In [23]: from sfepy.applications import solve_pde
In [24]: from sfepy.discrete import (Field, FieldVariable, Material,
                                     Integral, Integrals)
In [25]: from sfepy.discrete.fem.geometry_element import geometry_data

In [26]: gdata = geometry_data['2_3']
In [27]: nc = len(gdata.coors)
In [28]: ivn = Integral('ivn', order=-1,
      ....:              coors=gdata.coors, weights=[gdata.volume / nc] * nc)

In [29]: pb, variables = solve_pde('sfepy/examples/linear_elasticity/its2D_2.py')

In [30]: stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D,u)',
      ....:                   mode='qp', integrals=Integrals([ivn]))
In [31]: sfield = Field.from_args('stress', nm.float64, (3,), pb.domain.regions['Omega'])
In [32]: svar = FieldVariable('sigma', 'parameter', sfield,
      ....:                   primary_var_name='(set-to-None)')
In [33]: svar.set_from_qp(stress, ivn)

In [34]: print('Horizontal tensile stress = %.5e MPa/mm' % (svar()[0]))
Horizontal tensile stress = 7.57220e+00 MPa/mm
In [35]: print('Vertical compressive stress = %.5e MPa/mm' % (-svar()[1]))
Vertical compressive stress = 2.58660e+01 MPa/mm

In [36]: mat = pb.get_materials()['Load']
In [37]: P = 2.0 * mat.get_data('special', 'val')[1]
In [38]: P
Out[38]: -2000.0

In [39]: print('Horizontal tensile stress = %.5e MPa/mm' % (-2.*P/(nm.pi*150.)))
Horizontal tensile stress = 8.48826e+00 MPa/mm
In [40]: print('Vertical compressive stress = %.5e MPa/mm' % (-6.*P/(nm.pi*150.)))
Vertical compressive stress = 2.54648e+01 MPa/mm
```

To wrap this tutorial up let's explore *SfePy*'s probing functions.

Probing

As a bonus for sticking to the end of this tutorial see the following `Problem description` file that provides *SfePy* functions to quickly and neatly probe the solution.

```

r"""
Diametrically point loaded 2-D disk with postprocessing and probes. See
:ref:`sec-primer`.

Find  $\ul{u}$  such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""

from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.postprocess.probes_vtk import Probe

import os
from six.moves import range

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg')

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                data=strain, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                data=stress, dofs=None)

    probe = Probe(out, pb.domain.mesh, probe_view=True)

    ps0 = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
    ps1 = [[75.0, 0.0, 0.0], [0.0, 75.0, 0.0]]

```

(continues on next page)

(continued from previous page)

```

n_point = 10

labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
probes = []
for ip in range(len(ps0)):
    p0, p1 = ps0[ip], ps1[ip]
    probes.append('line%d' % ip)
    probe.add_line_probe('line%d' % ip, p0, p1, n_point)

for ip, label in zip(probes, labels):
    fig = plt.figure()
    plt.clf()
    fig.subplots_adjust(hspace=0.4)
    plt.subplot(311)
    pars, vals = probe(ip, 'u')
    for ic in range(vals.shape[1] - 1):
        plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('displacements')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=10))

    sym_labels = ['11', '22', '12']

    plt.subplot(312)
    pars, vals = probe(ip, 'cauchy_strain')
    for ii in range(vals.shape[1]):
        plt.plot(pars, vals[:, ii], label=r'$\epsilon_{%s}$' % sym_labels[ii],
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('Cauchy strain')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=8))

    plt.subplot(313)
    pars, vals = probe(ip, 'cauchy_stress')
    for ii in range(vals.shape[1]):
        plt.plot(pars, vals[:, ii], label=r'$\sigma_{%s}$' % sym_labels[ii],
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('Cauchy stress')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=8))

    opts = pb.conf.options
    filename_results = os.path.join(opts.get('output_dir'),
                                     'its2D_probe_%s.png' % ip)

    fig.savefig(filename_results)

return out

materials['Asphalt'][0].update({'D' : stiffness_from_youngpoisson(2, young, poisson)})

```

(continues on next page)

(continued from previous page)

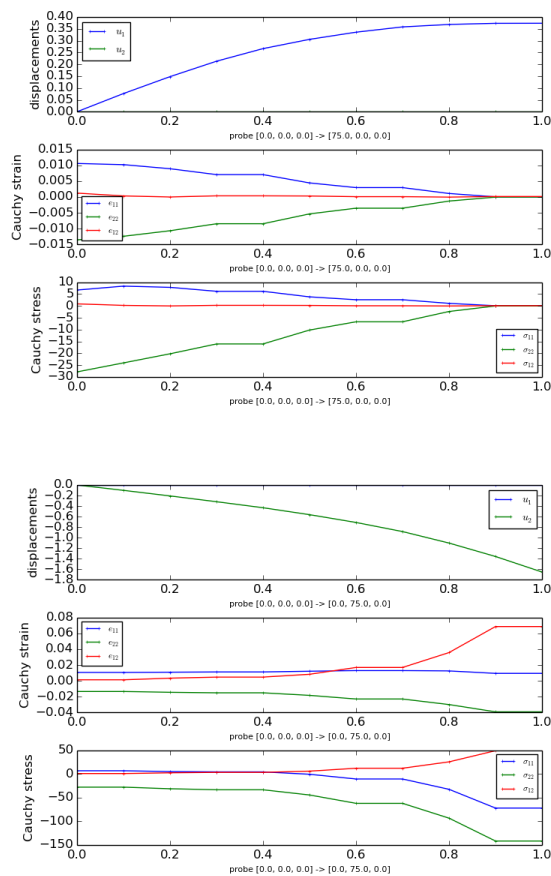
```
options.update({
    'post_process_hook' : 'stress_strain',
})
```

Probing applies interpolation to output the solution along specified paths. For the tutorial, line probing is done along the x- and y-axes of the model.

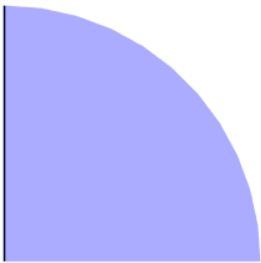
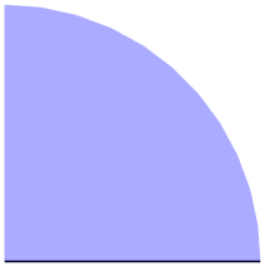
Run *SfePy* to solve the problem and apply the probes:

```
sfepy-run its2D_5.py
```

The probing function will generate the following figures that show the displacements, normal stresses and strains as well as shear stresses and strains along the probe paths. Note that you need [matplotlib](#) installed to run this example.



The probing function also generates previews of the mesh with the probe paths.



Interactive Example

SfePy can be used also interactively by constructing directly the classes that corresponds to the keywords in the problem description files. The following listing shows a script with the same (and more) functionality as the above examples:

```
#!/usr/bin/env python
"""
Diametrically point loaded 2-D disk, using commands for interactive use. See
:ref:`sec-primer`.

The script combines the functionality of all the ``its2D_?.py`` examples and
allows setting various simulation parameters, namely:

- material parameters
- displacement field approximation order
- uniform mesh refinement level

The example shows also how to probe the results as in
:ref:`linear_elasticity-its2D_4`. Using :mod:`sfepy.discrete.probes` allows
correct probing of fields with the approximation order greater than one.

In the SfePy top-level directory the following command can be used to get usage
information::

    python sfepy/examples/linear_elasticity/its2D_interactive.py -h
"""
from __future__ import absolute_import
```

(continues on next page)

(continued from previous page)

```

import sys
from six.moves import range
sys.path.append('.')
from argparse import ArgumentParser, RawDescriptionHelpFormatter

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import assert_, output, ordered_iteritems, IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Integrals,
                             Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.solvers.auto_fallback import AutoDirect
from sfepy.solvers.nls import Newton
from sfepy.discrete.fem.geometry_element import geometry_data
from sfepy.discrete.probes import LineProbe
from sfepy.discrete.projections import project_by_component

from sfepy.examples.linear_elasticity.its2D_2 import stress_strain
from sfepy.examples.linear_elasticity.its2D_3 import nodal_stress

def gen_lines(problem):
    """
    Define two line probes.

    Additional probes can be added by appending to `ps0` (start points) and
    `ps1` (end points) lists.
    """
    ps0 = [[0.0, 0.0], [0.0, 0.0]]
    ps1 = [[75.0, 0.0], [0.0, 75.0]]

    # Use enough points for higher order approximations.
    n_point = 1000

    labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
    probes = []
    for ip in range(len(ps0)):
        p0, p1 = ps0[ip], ps1[ip]
        probes.append(LineProbe(p0, p1, n_point))

    return probes, labels

def probe_results(u, strain, stress, probe, label):
    """
    Probe the results using the given probe and plot the probed values.
    """
    results = {}

    pars, vals = probe(u)

```

(continues on next page)

(continued from previous page)

```

results['u'] = (pars, vals)
pars, vals = probe(strain)
results['cauchy_strain'] = (pars, vals)
pars, vals = probe(stress)
results['cauchy_stress'] = (pars, vals)

fig = plt.figure()
plt.clf()
fig.subplots_adjust(hspace=0.4)
plt.subplot(311)
pars, vals = results['u']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('displacements')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

sym_indices = ['11', '22', '12']

plt.subplot(312)
pars, vals = results['cauchy_strain']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$\epsilon_{%s}$' % sym_indices[ic],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy strain')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

plt.subplot(313)
pars, vals = results['cauchy_stress']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$\sigma_{%s}$' % sym_indices[ic],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy stress')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

return fig, results

helps = {
    'young' : "the Young's modulus [default: %(default)s]",
    'poisson' : "the Poisson's ratio [default: %(default)s]",
    'load' : "the vertical load value (negative means compression)"
    " [default: %(default)s]",
    'order' : 'displacement field approximation order [default: %(default)s]',
    'refine' : 'uniform mesh refinement level [default: %(default)s]',
    'probe' : 'probe the results',
}

def main():
    from sfepy import data_dir

```

(continues on next page)

(continued from previous page)

```

parser = ArgumentParser(description=__doc__,
                        formatter_class=RawDescriptionHelpFormatter)
parser.add_argument('--version', action='version', version='%(prog)s')
parser.add_argument('--young', metavar='float', type=float,
                    action='store', dest='young',
                    default=2000.0, help=helps['young'])
parser.add_argument('--poisson', metavar='float', type=float,
                    action='store', dest='poisson',
                    default=0.4, help=helps['poisson'])
parser.add_argument('--load', metavar='float', type=float,
                    action='store', dest='load',
                    default=-1000.0, help=helps['load'])
parser.add_argument('--order', metavar='int', type=int,
                    action='store', dest='order',
                    default=1, help=helps['order'])
parser.add_argument('-r', '--refine', metavar='int', type=int,
                    action='store', dest='refine',
                    default=0, help=helps['refine'])
parser.add_argument('-p', '--probe',
                    action="store_true", dest='probe',
                    default=False, help=helps['probe'])
options = parser.parse_args()

assert_((0.0 < options.poisson < 0.5),
        "Poisson's ratio must be in ]0, 0.5[!")
assert_((0 < options.order),
        'displacement approximation order must be at least 1!')

output('using values:')
output(" Young's modulus:", options.young)
output(" Poisson's ratio:", options.poisson)
output(' vertical load:', options.load)
output('uniform mesh refinement level:', options.refine)

# Build the problem definition.
mesh = Mesh.from_file(data_dir + '/meshes/2d/its2D.mesh')
domain = FEDomain('domain', mesh)

if options.refine > 0:
    for ii in range(options.refine):
        output('refine %d...' % ii)
        domain = domain.refine()
        output('... %d nodes %d elements'
              % (domain.shape.n_nod, domain.shape.n_el))

omega = domain.create_region('Omega', 'all')
left = domain.create_region('Left',
                            'vertices in x < 0.001', 'facet')
bottom = domain.create_region('Bottom',
                              'vertices in y < 0.001', 'facet')
top = domain.create_region('Top', 'vertex 2', 'vertex')

```

(continues on next page)

(continued from previous page)

```

field = Field.from_args('fu', nm.float64, 'vector', omega,
                        approx_order=options.order)

u = FieldVariable('u', 'unknown', field)
v = FieldVariable('v', 'test', field, primary_var_name='u')

D = stiffness_from_youngpoisson(2, options.young, options.poisson)

asphalt = Material('Asphalt', D=D)
load = Material('Load', values={'val' : [0.0, options.load]})

integral = Integral('i', order=2*options.order)
integral0 = Integral('i', order=0)

t1 = Term.new('dw_lin_elastic(Asphalt.D, v, u)',
              integral, omega, Asphalt=asphalt, v=v, u=u)
t2 = Term.new('dw_point_load(Load.val, v)',
              integral0, top, Load=load, v=v)
eq = Equation('balance', t1 - t2)
eqs = Equations([eq])

xsym = EssentialBC('XSym', bottom, {'u.1' : 0.0})
ysym = EssentialBC('YSym', left, {'u.0' : 0.0})

ls = AutoDirect({})

nls_status = IndexedStruct()
nls = Newton({}, lin_solver=ls, status=nls_status)

pb = Problem('elasticity', equations=eqs)

pb.set_bcs(ebcs=Conditions([xsym, ysym]))

pb.set_solver(nls)

# Solve the problem.
variables = pb.solve()
output(nls_status)

# Postprocess the solution.
out = variables.create_output()
out = stress_strain(out, pb, variables, extend=True)
pb.save_state('its2D_interactive.vtk', out=out)

gdata = geometry_data['2_3']
nc = len(gdata.coors)

integral_vn = Integral('ivn', coors=gdata.coors,
                      weights=[gdata.volume / nc] * nc)

nodal_stress(out, pb, variables, integrals=Integrals([integral_vn]))

```

(continues on next page)

(continued from previous page)

```

if options.probe:
    # Probe the solution.
    probes, labels = gen_lines(pb)

    sfield = Field.from_args('sym_tensor', nm.float64, 3, omega,
                             approx_order=options.order - 1)
    stress = FieldVariable('stress', 'parameter', sfield,
                           primary_var_name='(set-to-None)')
    strain = FieldVariable('strain', 'parameter', sfield,
                           primary_var_name='(set-to-None)')

    cfield = Field.from_args('component', nm.float64, 1, omega,
                             approx_order=options.order - 1)
    component = FieldVariable('component', 'parameter', cfield,
                              primary_var_name='(set-to-None)')

    ev = pb.evaluate
    order = 2 * (options.order - 1)
    strain_qp = ev('ev_cauchy_strain.%d.Omega(u)' % order, mode='qp')
    stress_qp = ev('ev_cauchy_stress.%d.Omega(Asphalt.D, u)' % order,
                   mode='qp', copy_materials=False)

    project_by_component(strain, strain_qp, component, order)
    project_by_component(stress, stress_qp, component, order)

    all_results = []
    for ii, probe in enumerate(probes):
        fig, results = probe_results(u, strain, stress, probe, labels[ii])

        fig.savefig('its2D_interactive_probe_%d.png' % ii)
        all_results.append(results)

    for ii, results in enumerate(all_results):
        output('probe %d:' % ii)
        output.level += 2
        for key, res in ordered_iteritems(results):
            output(key + ':')
            val = res[1]
            output('  min: %+.2e, mean: %+.2e, max: %+.2e'
                   % (val.min(), val.mean(), val.max()))
        output.level -= 2

if __name__ == '__main__':
    main()

```

The script can be run from the *SfePy* top-level directory, assuming the in-place build, as follows:

```
python sfepy/examples/linear_elasticity/its2D_interactive.py
```

The script allows setting several parameters that influence the solution, see:

```
python sfepy/examples/linear_elasticity/its2D_interactive.py -h
```

for the complete list. Besides the material parameters, a uniform mesh refinement level and the displacement field approximation order can be specified. The script demonstrates how to

- project a derived quantity, that is evaluated in quadrature points (e.g. a strain or stress), into a field variable;
- probe the solution defined in the field variables.

Using `sfepy.discrete.probes` allows correct probing of fields with the approximation order greater than one. The end.

1.5.2 Using Salome with SfePy

NOTE This tutorial was created in 2014, so it may be obsolete.

Introduction

Salome is a powerful open-source tool for generating meshes for numerical simulation and post processing the results. This is a short tutorial on using *Salome* as a preprocessor for preparing meshes for use with *SfePy*.

Tutorial prerequisites

This tutorial assumes that you have a working copy of *Salome*. It is possible to build *Salome* from source code. Fortunately, for the less brave, many pre-compiled binaries for different platforms are available at the [Salome download](#) page. Registration for a free account may be required to download from the preceding site.

In addition, this tutorial assumes you have a working copy of *SfePy* with MED read support. See the [Installation](#) for help. Note that it is not actually necessary to “install” *SfePy*; one may run the code from the source directory (see notation below) after compilation of the C extension modules (again, see the installation notes if you are confused).

Note on notation used in this tutorial

We are using the following notations:

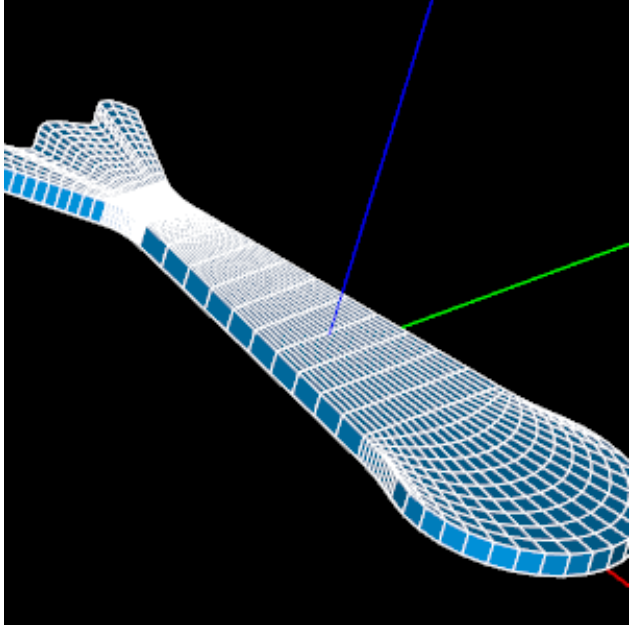
- `<sfepy_root>`: the root directory of the *SfePy* source code
- `<work_dir>`: the working directory where you plan to save your files

Step 1: Using Salome

Salome has its own set of tutorials and community resources. It is suggested you look around on [Salome](#) web site to familiarize yourself with the available resources.

This tutorial follows the EDF Exercise 1 available from the [Salome Tutorial Site](#). Go ahead and complete this tutorial now. We will use the result from there in the following.

This is the mesh you should end up with:



Step 2: Exporting mesh from *Salome*

In the *Salome* MESH module, right click on the mesh object `Mesh_Partition_Hexa` you created in the *Salome* EDF Exercise 1 Tutorial and click `Export to MED file`. Save the file as `Mesh_Partition_Hexa.med` in your working directory `<work_dir>`.

Step 3: Copy *SfePy* project description files

In this tutorial, we will assume that we need to solve a linear elasticity problem on the mesh generated by *Salome*. Since the *Salome* mesh looks a bit like a fish, we will try to simulate the fish waving its tail.

Copy the file `<sfepy_root>/sfepy/examples/linear_elasticity/linear_elastic.py` to `<work_dir>`. Use your favorite python editor to load this file. We will customize this file for our purposes.

Step 4: Modify `linear_elastic.py`

Mesh specification

The first thing we have to do is tell *SfePy* to use our new mesh. Change the line

```
filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'
```

to

```
filename_mesh = 'Mesh_Partition_Hexa.med'
```

Region specification

Next, we have to define sensible Regions for the mesh. We will apply a displacement to the Tail and keep the Top and Bottom of the fish fixed. Change the lines

```
regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
    'SomewhereTop' : ('vertices in (z > 0.017) & (x > 0.03) & (x < 0.07)', 'vertex'),
}
```

to

```
regions = {
    'Omega' : 'all',
    'Tail' : ('vertices in (x < -94)', 'facet'),
    'TopFixed' : ('vertices in (z > 9.999) & (x > 54)', 'facet'),
    'BotFixed' : ('vertices in (z < 0.001) & (x > 54)', 'facet'),
}
```

Field specification

The *Salome* mesh uses hexahedral linear order elements; in *SfePy* notation these are called 3_8, see *User's Guide*.

Just keep the lines

```
fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}
```

Boundary condition specifications

In this section, we tell *SfePy* to fix the top and bottom parts of the “head” of the fish and move the tail 10 units to the side (z direction).

Change the lines

```
ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'Displaced' : ('Right', {'u.0' : 0.01, 'u.[1,2]' : 0.0}),
    'PerturbedSurface' : ('SomewhereTop', {'u.2' : 0.005}),
}
```

to

```
ebcs = {
    'TopFixed' : ('TopFixed', {'u.all' : 0.0}),
    'BotFixed' : ('BotFixed', {'u.all' : 0.0}),
    'Displaced' : ('Tail', {'u.2' : 10, 'u.[0,1]' : 0.0}),
}
```

Step 5: Run *SfePy*

Save your changes to `linear_elastic.py`. Now it's time to run the *SfePy* calculation. In your `<work_dir>` in your terminal type:

```
sfepy-run linear_elastic.py
```

This will run the *SfePy* calculation. Some progress information is printed to your screen and the residual (a measure of the convergence of the solution) is printed for each iteration of the solver. The solver terminates when this residual is less than a certain value. It should only take 1 iteration since we are solving a linear problem. The results will be saved to `Mesh_Partition_Hexa.vtk`.

Now we can view the results of our work. In your terminal, type:

```
sfepy-view Mesh_Partition_Hexa.vtk -f u:wu:f2.0:p0 0:vw:p0
```

You should get the plot with the deformed and undeformed meshes. Notice how the fish is bending its tail in response to the applied displacement.

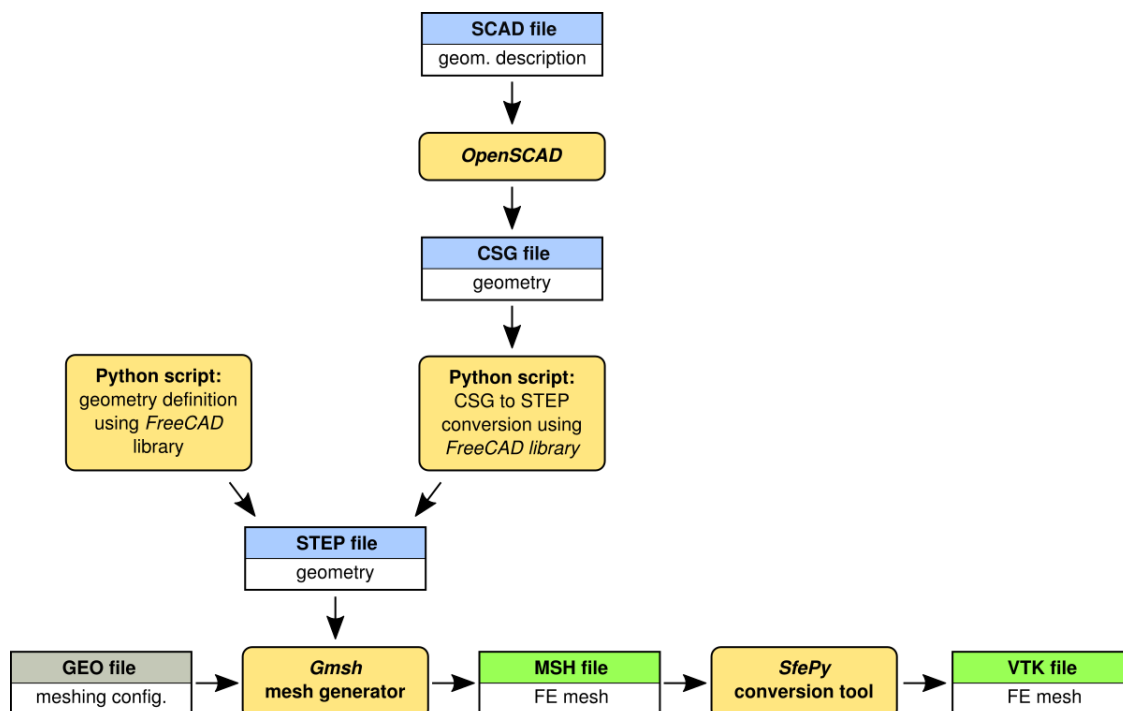
Now you should be able to use meshes created in *Salome* with *SfePy*!

1.5.3 Preprocessing: *FreeCAD/OpenSCAD* + *Gmsh*

Introduction

There are several open source tools for preparing 2D and 3D finite element meshes like *Salome*, *FreeCAD*, *Gmsh*, *Netgen*, etc. Most of them are GUI based geometrical modeling and meshing environments/tools but they also usually allow using their libraries in user scripts. Some of the above mentioned tools are handy for solid modeling, some of them are great for meshing. This tutorial shows how to combine solid geometry modeling functions provided by *FreeCAD* or *OpenSCAD* with meshing functions of *Gmsh*.

The collaboration of modeling, meshing and conversion tools and the workflow are illustrated in the following scheme.



Creating geometry using *FreeCAD*

Functionalities of *FreeCAD* are accessible to Python and can be used to define geometrical models in simple Python scripts. There is a tutorial related to [Python scripting in FreeCAD](#).

The first step in creating a Python script is to set up a path to the *FreeCAD* libraries and import all required modules:

```
1 import sys
2 FREECADPATH = '/usr/lib/freecad/lib/'
3 sys.path.append(FREECADPATH)
4
5 from FreeCAD import Base, newDocument
6 import Part
7 import Draft
8 import ProfileLib.RegularPolygon as Poly
```

Now, a new empty *FreeCAD* document can be defined as:

```
doc = newDocument()
```

All new objects describing the geometry will be added to this document.

In the following lines a geometrical model of a screwdriver handle will be created. Let's start by defining a sphere and a cylinder and join these objects into the one called `uni`:

```
1 radius = 0.01
2 height = 0.1
3
4 cyl = doc.addObject("Part::Cylinder", "cyl")
5 cyl.Radius = radius
6 cyl.Height = height
7
8 sph = doc.addObject("Part::Sphere", "sph")
9 sph.Radius = radius
10
11 uni = doc.addObject("Part::MultiFuse", "uni")
12 uni.Shapes = [cyl, sph]
```

Create a polygon, revolve it around the z-axis to create a solid and use the result as the cutting tool applied to `uni` object:

```
1 ske = doc.addObject('Sketcher::SketchObject', 'Sketch')
2 ske.Placement = Base.Placement(Base.Vector(0, 0, 0),
3                                 Base.Rotation(-0.707107, 0, 0, -0.707107))
4 Poly.makeRegularPolygon('Sketch', 5,
5                          Base.Vector(-1.2 * radius, 0.9 * height, 0),
6                          Base.Vector(-0.8 * radius, 0.9 * height, 0))
7
8 cut = doc.addObject("PartDesign::Revolution", "Revolution")
9 cut.Sketch = ske
10 cut.ReferenceAxis = (ske, ['V_Axis'])
11 cut.Angle = 360.0
12
13 dif = doc.addObject("Part::Cut", "dif")
```

(continues on next page)

(continued from previous page)

```

14 dif.Base = uni
15 dif.Tool = cut

```

Create a cylinder, make a polar array of the cylinder objects and subtract it from the previous result:

```

1 cyl1 = doc.addObject("Part::Cylinder", "cyl1")
2 cyl1.Radius = 0.2 * radius
3 cyl1.Height = 1.1 * height
4 cyl1.Placement = Base.Placement(Base.Vector(-1.1 * radius, 0, -0.2 * height),
5                                   Base.Rotation(0, 0, 0, 1))
6
7 arr = Draft.makeArray(cyl1, Base.Vector(1, 0, 0), Base.Vector(0, 1, 0), 2, 2)
8 arr.ArrayType = "polar"
9 arr.NumberPolar = 6
10
11 dif2 = doc.addObject("Part::Cut", "dif2")
12 dif2.Base = dif
13 dif2.Tool = arr

```

Create a middle hole for the screwdriver metal part:

```

1 cyl2 = doc.addObject("Part::Cylinder", "cyl2")
2 cyl2.Radius = 0.3 * radius
3 cyl2.Height = height
4
5 dif3 = doc.addObject("Part::Cut", "dif3")
6 dif3.Base = dif2
7 dif3.Tool = cyl2

```

Finally, recompute the geometry, export the part to the *STEP* file and save the document in *FreeCAD* format (not really needed for subsequent mesh generation, but may be useful for visualization and geometry check):

```

1 doc.recompute()
2
3 Part.export([dif3], 'screwdriver_handle.step')
4
5 doc.saveAs('screwdriver_handle.FCStd')

```

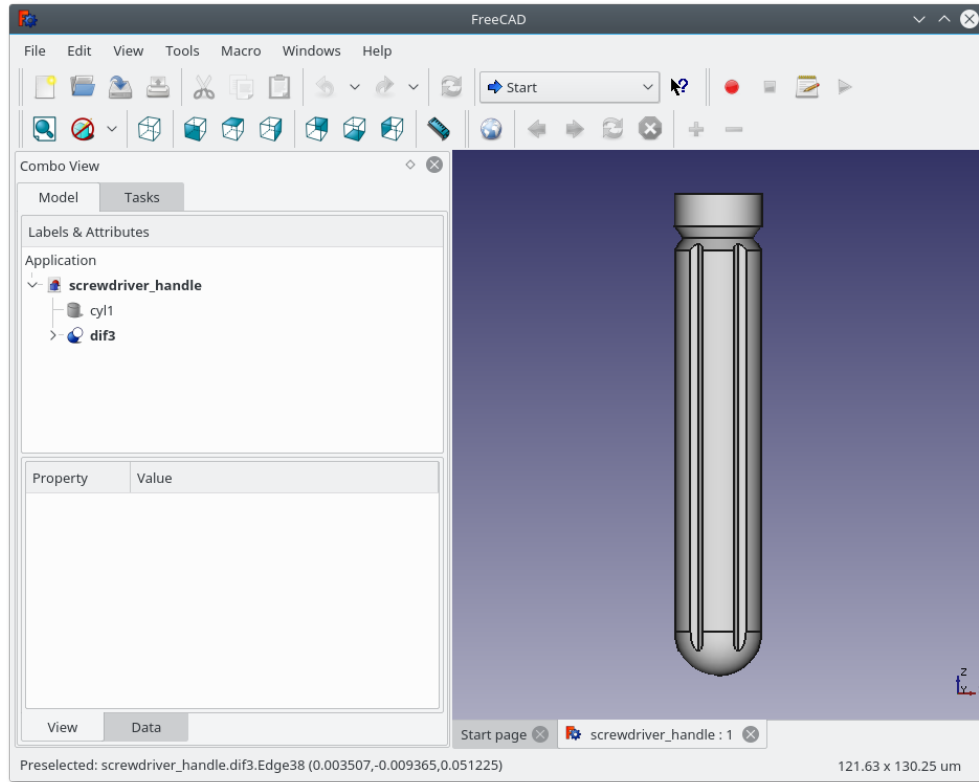
A finite element mesh can be generated directly in *FreeCAD* using *MeshPart* module:

```

1 import MeshPart
2
3 mesh = doc.addObject("Mesh::Feature", "Mesh")
4 mesh.Mesh = MeshPart.meshFromShape(Shape=dif3.Shape, MaxLength=0.002)
5 mesh.Mesh.write("./screwdriver_handle.bdf", "NAS", "mesh")

```

The meshing function of *MeshPart* module is limited to triangular grids so it is better to use *Gmsh* mesh generator which can provide triangular and quadrilateral meshes in 2D or tetrahedral and hexahedral meshes in 3D. *Gmsh* allows to control the meshing process through a wide range of parameters. Meshing by *Gmsh* will be described in section *Gmsh - generating finite element mesh*.



The example of screwdriver handle: `screwdriver_handle.py`.

There are two simple ways how to discover Python calls of *FreeCAD* functions. You can enable “show script commands in python console” in `Edit->Preferences->General->Macro` and the Python console by selecting `View->Views->Python Console` and all subsequent operations will be printed in the console as the Python code. The second way is to switch on the macro recording function (`Macro->Macro recording ...`) which generates a Python script (`FCMacro` file) containing all the code related to actions in the *FreeCAD* graphical interface.

Creating geometry using *OpenSCAD*

The alternative tool for solid geometrical modeling is *OpenSCAD* - “The Programmers Solid 3D CAD Modeller”. It has its own description language based on functional programming that is used to construct solid models using geometrical primitives similar to *FreeCAD*. Solid geometries can be exported to several file formats including *STL* and *CSG*. *OpenSCAD* allows solid modeling based on Constructive Solid Geometry (CSG) principles and extrusion of 2D objects into 3D. The model of a screwdriver handle presented in the previous section can be defined in *OpenSCAD* by the following code (`screwdriver_handle.scad`):

```

1 radius = 0.01;
2 height = 0.1;
3 $fn = 50;
4
5 difference() {
6     difference() {
7         difference() {
8             union() {
9                 cylinder(center=false, h=height, r=radius);
10                sphere(radius);
11            };

```

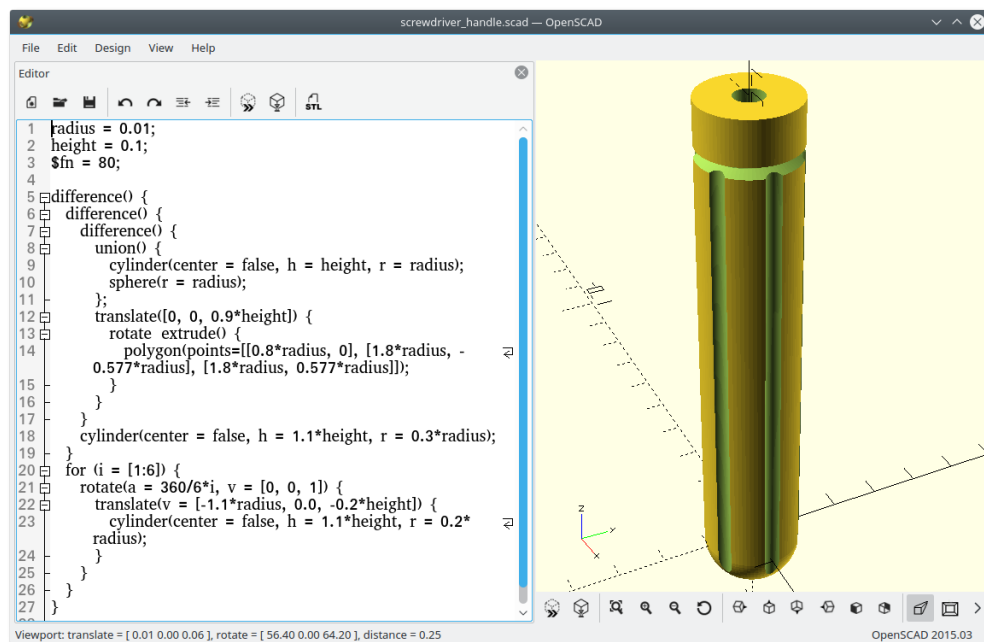
(continues on next page)

(continued from previous page)

```

12     translate([0, 0, 0.9*height])
13     rotate_extrude()
14     polygon([[0.8*radius, 0], [1.8*radius, -0.577*radius], [1.8*radius, 0.
15     ↪ 577*radius]]);
16 }
17 cylinder(center=false, h=1.1*height, r=0.3*radius);
18 }
19 for (i = [1:6]) {
20     rotate([0, 0, 360/6*i])
21     translate([-1.1*radius, 0.0, -0.2*height])
22     cylinder(center=false, h=1.1*height, r=0.2*radius);
23 }

```



To generate a finite element mesh of the solid geometry the model must be exported to a suitable file format. *OpenSCAD* has limited export options, but by using *FreeCAD* import/export functions, it is possible to find a workaround. The *OpenSCAD* model can be exported to the *CSG* file format and *FreeCAD* can be used as a mesh converter to the *STEP* format:

```

1 import sys
2 sys.path.append('/usr/lib/freecad/lib/')
3 sys.path.append('/usr/lib/freecad/Mod/OpenSCAD/')
4
5 import FreeCAD
6 import Part
7 import importCSG
8
9 importCSG.open('screwdriver_handle.csg')
10 Part.export([FreeCAD.ActiveDocument.Objects[-1]], 'screwdriver_handle.step')

```

Gmsh - generating finite element mesh

Gmsh can create finite element meshes using geometrical models imported from *STEP*, *IGES* and *BRep* files (has to be compiled with *OpenCASCADE* support).

The following *GEO* file imports `screwdriver_handle.step` file and defines a field controlling the mesh size (`screwdriver_handle.geo`):

```
1 Merge "screwdriver_handle.step";
2
3 Field[1] = MathEval;
4 Field[1].F = "0.002";
5 Background Field = 1;
```

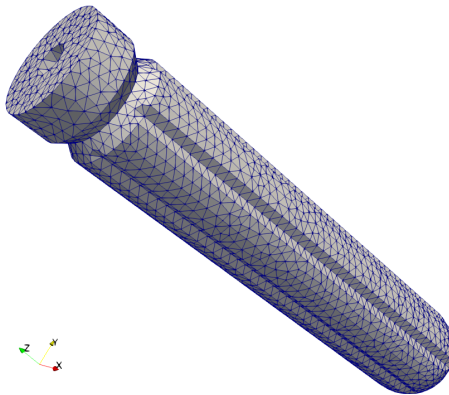
Now, run *Gmsh* generator and export the mesh into the *MSH* format in which all surface and volumetric elements are stored:

```
gmsh -3 -format msh -o screwdriver_handle.msh screwdriver_handle.geo
```

By converting the *MSH* file into the *VTK* format using `sfepy-convert`:

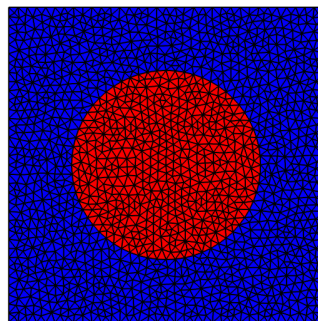
```
sfepy-convert -d 3 screwdriver_handle.msh screwdriver_handle.vtk
```

the surface elements are discarded and only the volumetric mesh is preserved.



Note: planar 2D meshes

To create a planar 2D mesh, such as



that can be described by this *Gmsh* code, the mesh generator can be called as follows:

```
gmsh -2 -format msh -o circle_in_square.msh circle_in_square.geo
```

This, however is not enough to create a truly 2D mesh - the created mesh vertices still have the third, *z*, component which is equal to zero. In order to remove the third component, use:

```
sfepy-convert --2d circle_in_square.msh circle_in_square.h5
```

Now, in the resulting `circle_in_square.h5`, each vertex has only two coordinates. Another way of generating the 2D mesh is to use the legacy VTK format as follows:

```
gmsh -2 -format vtk -o circle_in_square.vtk circle_in_square.geo
sfepy-convert circle_in_square.vtk circle_in_square.h5
```

This is due to the fact that the legacy VTK does not support 2D vertices and so the `VTKMeshIO` reader tries to detect the planar geometry by comparing the *z* components to zero - the `--2d` option of `sfepy-convert` is not needed in this case.

Multipart models

Meshing models composed of parts with different material groups is a little bit tricky task. But there are some more or less general ways of doing that. Here, the method using functions of *Gmsh* for periodic meshes will be shown.

The screwdriver handle example is extended by adding a screwdriver shank. The new part is composed of a cylinder trimmed at one end:

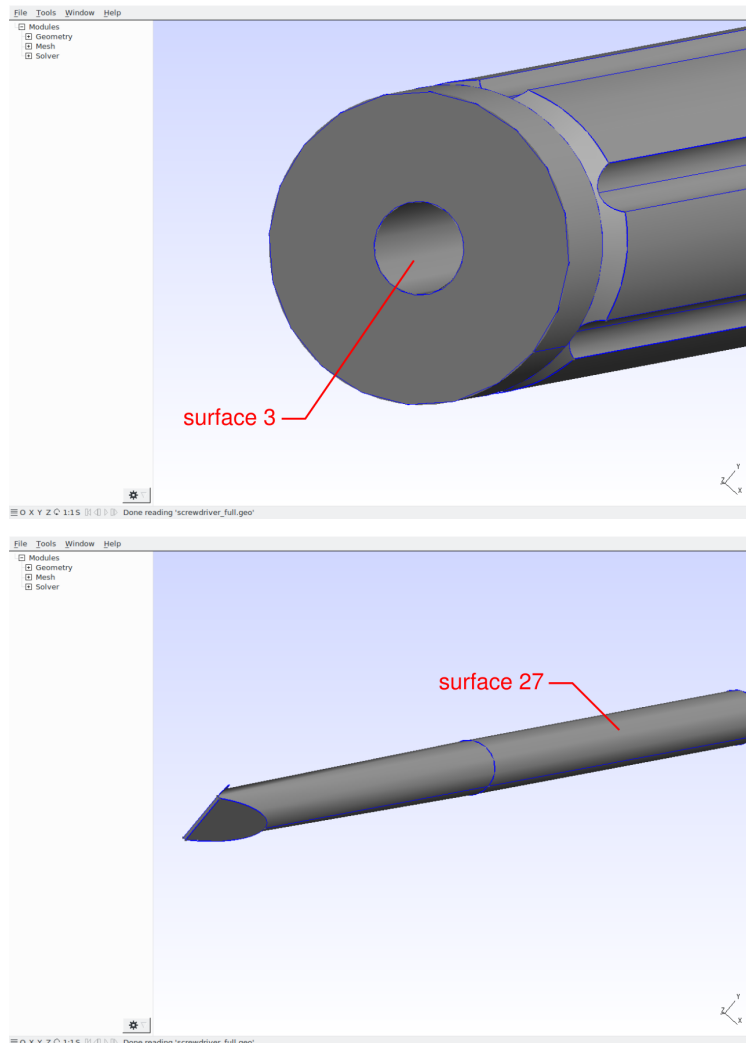
```
1 cyl3 = doc.addObject("Part::Cylinder", "cyl3")
2 cyl3.Radius = 0.3 * radius
3 cyl3.Height = height
4 cyl3.Placement = Base.Placement(Base.Vector(0, 0, height),
5                               Base.Rotation(0, 0, 0, 1))
6
7 tip1 = doc.addObject("Part::Box", "tip1")
8 tip1.Length = radius
9 tip1.Width = 2 * radius
10 tip1.Height = 3 * radius
11 tip1.Placement = Base.Placement(Base.Vector(0, -radius, 1.71 * height),
12                               Base.Rotation(Base.Vector(0, 1, 0), -10),
13                               Base.Vector(0, 0, 3 * radius))
14
15 tip2 = doc.addObject("Part::Mirroring", "tip2")
16 tip2.Source = tip1
17 tip2.Normal = (1, 0, 0)
18
19 tip3 = doc.addObject("Part::MultiFuse", "tip3")
20 tip3.Shapes = [tip1, tip2]
21
22 dif4 = doc.addObject("Part::Cut", "dif4")
23 dif4.Base = cyl3
24 dif4.Tool = tip3
25
26 uni2 = doc.addObject("Part::MultiFuse", "uni2")
27 uni2.Shapes = [cyl2, dif4]
```

The handle and shank are exported to the *STEP* file as two separated parts:

```
1 doc.recompute()
2
3 Part.export([dif3, uni2], 'screwdriver_full.step')
4 doc.saveAs('screwdriver_full.FCStd')
```

The full screwdriver example (handle + shank): `screwdriver_full.py`.

To create a coincidence mesh on the handle and shank interface, it is necessary to identify the interface surfaces and declare them to be periodic in the *GEO* file. The identification has to be done manually in the *Gmsh* graphical interface.



The input file for *Gmsh* is then as follows (`screwdriver_full.geo`):

```
1 Merge "screwdriver_full.step";
2
3 Periodic Surface 5 {7} = 26 {67};
4 Periodic Surface 3 {6, 2, -6, 7} = 27 {68, 69, -68, 67};
5
6 Physical Volume(1) = {1};
7 Physical Volume(2) = {2};
```

(continues on next page)

(continued from previous page)

```

8
9 Field[1] = MathEval;
10 Field[1].F = "0.0015";
11 Background Field = 1;

```

where the first pair of periodic surfaces corresponds to the common circle faces (bottom of the shank) and the second pair to the common cylindrical surfaces. See [Gmsh Reference manual](#) for details on periodic meshing.

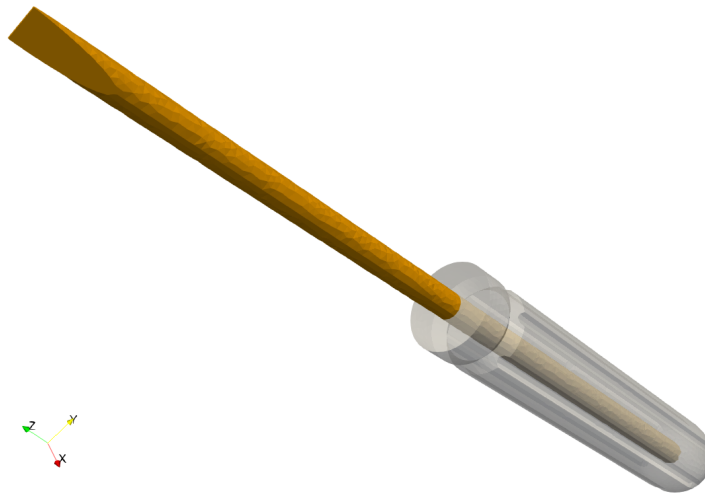
Using the above stated *GEO* file, *Gmsh* creates a mesh containing duplicate vertices on the handle/shank interface. These duplicate vertices can be removed during the conversion to the *VTK* format by giving `--merge` (or just `-m`) argument to *convert_mesh.py* script:

```
sfepy-convert -m screwdriver_full.msh screwdriver_full.vtk
```

In order to extract the cells by the physical groups use the conversion script with `--save-per-mat` argument:

```
sfepy-convert --save-per-mat screwdriver_full.vtk screwdriver.vtk
```

It produces *screwdriver.vtk* containing the original mesh and *screwdriver_matid_1.vtk*, *screwdriver_matid_2.vtk* files containing only the cells of a given physical group and all vertices of the original mesh.



When using *OpenSCAD*, define the full screwdriver geometry as (*screwdriver_full.scad*):

```

1 radius = 0.01;
2 height = 0.1;
3 $fn = 50;
4
5 module tip() {
6   rotate([0, -10, 0])
7   translate([0, -radius, -3*radius])
8   cube([radius, 2*radius, 3*radius], center=false);
9 }
10
11 difference() {
12   difference() {
13     difference() {
14       union() {

```

(continues on next page)

(continued from previous page)

```

15     cylinder(center=false, h=height, r=radius);
16     sphere(radius);
17 };
18     translate([0, 0, 0.9*height])
19     rotate_extrude()
20     polygon([[0.8*radius, 0], [1.8*radius, -0.577*radius], [1.8*radius, 0.
↪ 577*radius]]);
21 }
22     cylinder(center=false, h=height, r=0.3*radius);
23 }
24 for (i = [1:6]) {
25     rotate([0, 0, 360/6*i])
26     translate([-1.1*radius, 0.0, -0.2*height])
27     cylinder(center=false, h=1.1*height, r=0.2*radius);
28 }
29 }
30
31 union() {
32     difference() {
33         translate([0, 0, height])
34         cylinder(center=false, h=height, r=0.3*radius);
35         translate([0, 0, 1.71*height + 3*radius])
36         union() {
37             tip();
38             mirror ([1, 0, 0]) tip();
39         }
40     }
41     cylinder(center=false, h=height, r=0.3*radius);
42 }

```

and convert the *CSG* file to the *STEP* file by:

```

1 import CSG.open('screwdriver_full.csg')
2 top_group = FreeCAD.ActiveDocument.Objects[-1]
3 Part.export(top_group.OutList, 'screwdriver_full.step')

```

Since the different tools for geometry definition have been used, the numbering of geometric objects may differ and the surface and edge numbers have to be changed in the *GEO* file:

```

Periodic Surface 5 {6} = 26 {66};
Periodic Surface 3 {5, 2, -5, 6} = 27 {67, 68, -67, 66};

```

Note: The numbering of objects may vary between *FreeCAD*, *OpenSCAD* and *Gmsh* versions.

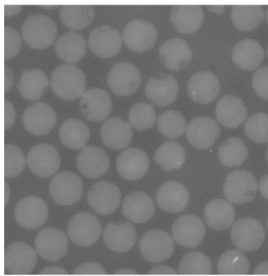
1.5.4 Material Identification

Introduction

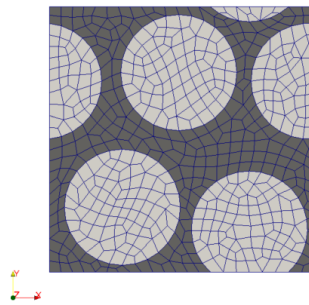
This tutorial shows identification of material parameters of a composite structure using data (force-displacement curves) obtained by a standard tensile test.

Composite structure

The unidirectional long fiber carbon-epoxy composite is considered. Its microstructure was analysed by the scanning electron microscopy and the data, volume fractions and fibers cross-sections, were used to generate a periodic finite element mesh (representative volume element - RVE) representing the random composite structure at the microscopic level (the random structure generation algorithm is described in¹):



cross-section of
fiber composite



generated finite
element mesh

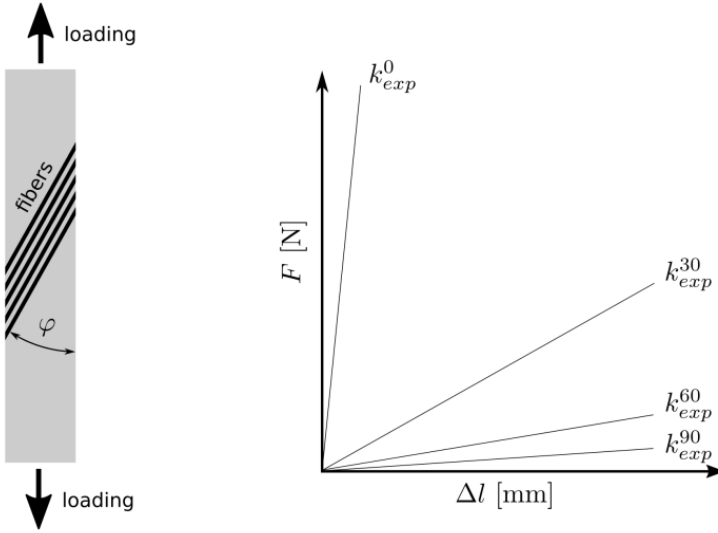
This RVE is used in the micromechanical FE analysis which is based on the two-scale homogenization method.

Material testing

Several carbon-epoxy specimens with different fiber orientations (0, 30, 60 and 90 degrees) were subjected to the tensile test in order to obtain force-elongation dependencies, see². The slopes of the linearized dependencies were used in an objective function of the identification process.

¹ Lubachevsky B. D., How to Simulate Billiards and Similar Systems, Journal of Computational Physics, 94(2), 1991. http://arxiv.org/PS_cache/cond-mat/pdf/0503/0503627v2.pdf

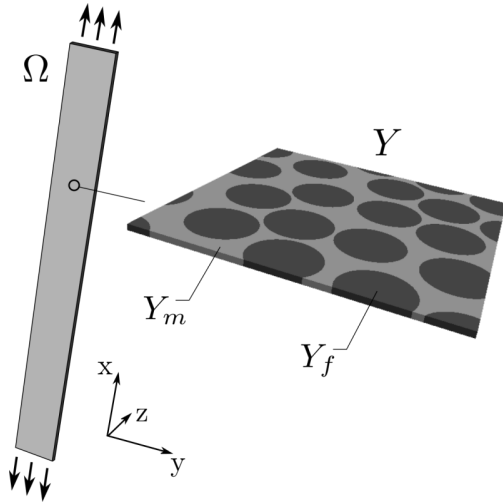
² Srbová H., Kroupa T., Zemčík R., Identification of the Material Parameters of a Unidirectional Fiber Composite Using a Micromodel, Materiali in Tehnologije, 46(5), 2012, 431-434.



Numerical simulation

The linear isotropic material model is used for both components (fiber and matrix) of the composite so only four material parameters (Young's modulus and Poisson's ratio for each component) are necessary to fully describe the mechanical behavior of the structure.

The numerical simulations of the tensile tests are based on the homogenization method applied to the linear elastic problem³. The homogenization procedure results in the microscopic problem solved within the RVE and the macroscopic problem that involves the homogenized elastic coefficients.



³ Pinho-da-Cruz L., Oliveira J. A. and Teixeira-Dias F., Asymptotic homogenization in linear elasticity. Part I: Mathematical formulation and finite element modeling, Computational Materials Science, 45(4), 2009, 1073–1080.

Homogenized coefficients

The problem at the microscopic level is formulated in terms of characteristic response functions and its solution is used to evaluate the homogenized elasticity tensor. The microscopic problem has to be solved with the periodic boundary conditions.

The following *SfePy* description file is used for definition of the microscopic problem: `homogenization_opt_src`.

In the case of the identification process function `get_mat()` obtains the material parameters (Young's modules, Poisson's ratios) from the outer identification loop. Otherwise these parameters are given by values.

Notice the use of `parametric_hook` (*Miscellaneous*) to pass around the optimization parameters.

Macroscopic simulation

The homogenized elasticity problem is solved for the unknown macroscopic displacements and the elongation of the composite specimen is evaluated for a given loading. These values are used to determine the slopes of the calculated force-elongation dependencies which are required by the objective function.

The *SfePy* description file for the macroscopic analysis: `linear_elasticity_opt_src`.

Identification procedure

The identification of material parameters, i.e. the Young's modulus and Poisson's ratio, of the epoxy matrix (E_m, ν_m) and carbon fibers (E_f, ν_f) can be formulated as a minimization of the following objective function:

$$\Phi(\mathbf{x}) = \sum_{i \in \{0, 30, 60, 90\}} \left(1 - \frac{k_{comp}^i(\mathbf{x})}{k_{exp}^i} \right)^2, \quad (1.9)$$

where k_{comp}^i and k_{exp}^i are the computed and measured slopes of the force-elongation tangent lines for a given fiber orientation. This function is minimized using `scipy.optimize.fmin_tnc()`, considering bounds of the identified parameters.

The following steps are performed in each iteration of the optimization loop:

1. Solution of the microscopic problem, evaluation of the homogenized elasticity tensor.
2. Solution of the macroscopic problems for different fiber orientations (0, 30, 60, 90), this is incorporated by appropriate rotation of the elasticity tensor.
3. Evaluation of the objective function.

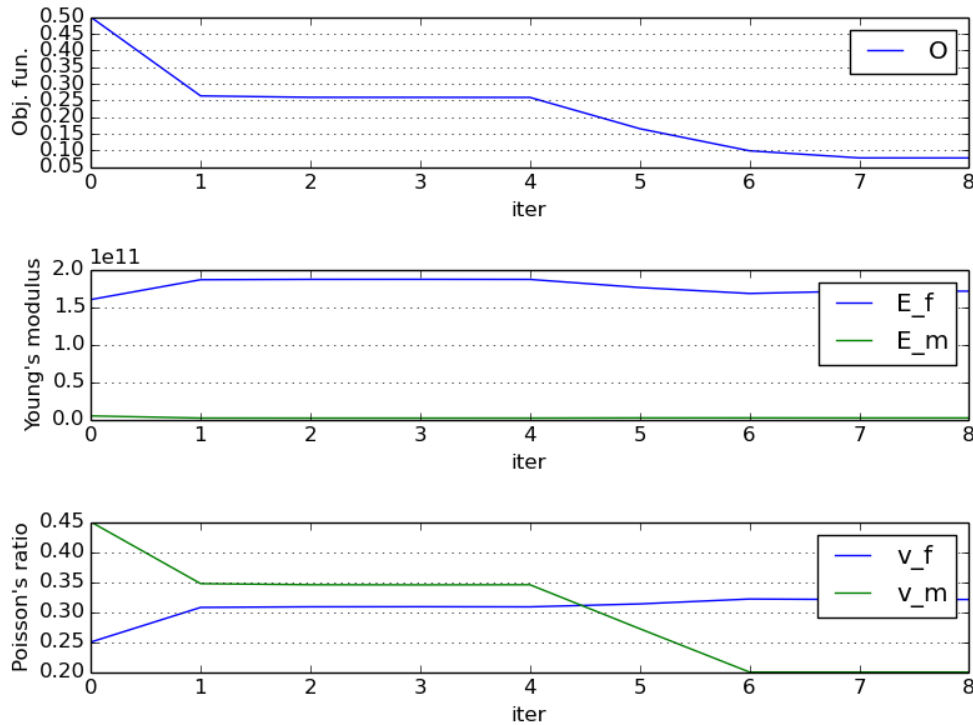
Python script for material identification: `material_opt_src`.

Running identification script

Run the script from the command shell as (from the top-level directory of *SfePy*):

```
$ python sfepy/examples/homogenization/material_opt.py
```

The iteration process is monitored using graphs where the values of the objective function and material parameters are plotted.



The resulting values of E_f , ν_f , E_m , ν_m can be found at the end of the script output:

```
>>> material optimization FINISHED <<<
material_opt_micro: terminated
optimized parameters: [1.71129526e+11 3.20844131e-01 2.33507829e+09 2.00000000e-01]
```

So that:

$$E_f = 171.13 \text{ GPa}$$

$$\nu_f = 3.21$$

$$E_m = 2.34 \text{ GPa}$$

$$\nu_m = 0.20$$

Note: The results may vary across SciPy versions and related libraries.

1.5.5 Mesh parametrization

Introduction

When dealing with shape optimization we usually need to modify a FE mesh using a few optimization parameters describing the mesh geometry. The B-spline parametrization offers an efficient way to do that. A mesh region (2D or 3D) that is to be parametrized is enclosed in the so called spline-box and the positions of all vertices inside the box can be changed by moving the control points of the B-spline curves.

There are two different classes for the B-spline parametrization implemented in *SfePy* (module [sfepy.mesh.splinebox](#)): `SplineBox` and `SplineRegion2D`. The first one defines a rectangular parametrization box in 2D or 3D while the second one allows to set up an arbitrary shaped region of parametrization in 2D.

SplineBox

The rectangular B-spline parametrization is created as follows:

```
1 from sfepy.mesh.splinebox import SplineBox
2
3 spb = SplineBox(<bbox>, <coors>, <nsg>)
```

the first parameter defines the range of the box in each dimension, the second parameter is the array of coordinates (vertices) to be parametrized and the last one (optional) determines the number of control points in each dimension. The number of the control points (ncp) is calculated as:

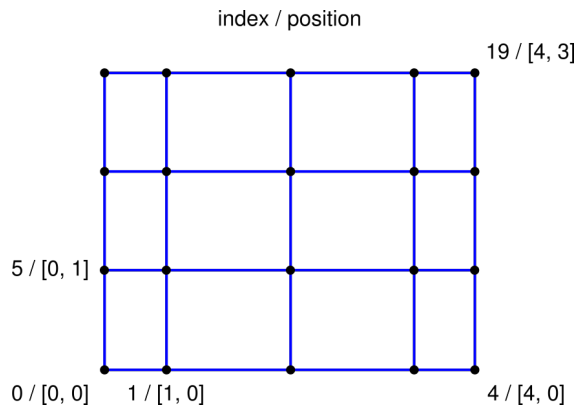
$$ncp_i = nsg_i + degree, \quad i = 1, 2(, 3) \quad (1.10)$$

where *degree* is the degree of the B-spline curve (default value: 3 = cubic spline) and *nsg* is the number of the spline segments (default value: [1,1(,1)] = 4 control points for all dimensions).

The position of the vertices can be modified by moving the control points:

```
spb.move_control_point(<cpoint>, <val>)
```

where <cpoint> is the index or position of the control point, for explanation see the following figure.



The displacement is given by <val>. The modified coordinates of the vertices are evaluated by:

```
new_coors = spb.evaluate()
```

Example

- Create a new 2D SplineBox with the left bottom corner at [-1,-1] and the right top corner at [1, 0.6] which has 5 control points in x-direction and 4 control points in y-direction:

```
1 from sfepy.mesh.splinebox import SplineBox
2 from sfepy.discrete.fem import Mesh
3
4 mesh = Mesh.from_file('meshes/2d/square_tri1.mesh')
5 spb = SplineBox([[-1, 1], [-1, 0.6]], mesh.coors, nsg=[2,1])
```

- Modify the position of mesh coordinates by moving three control points (with indices 1,2 and 3):

```

1 spb.move_control_point(1, [0.1, -0.2])
2 spb.move_control_point(2, [0.2, -0.3])
3 spb.move_control_point(3, [0.0, -0.1])

```

- Evaluate the new coordinates:

```
mesh.cmesh.coors[:] = spb.evaluate()
```

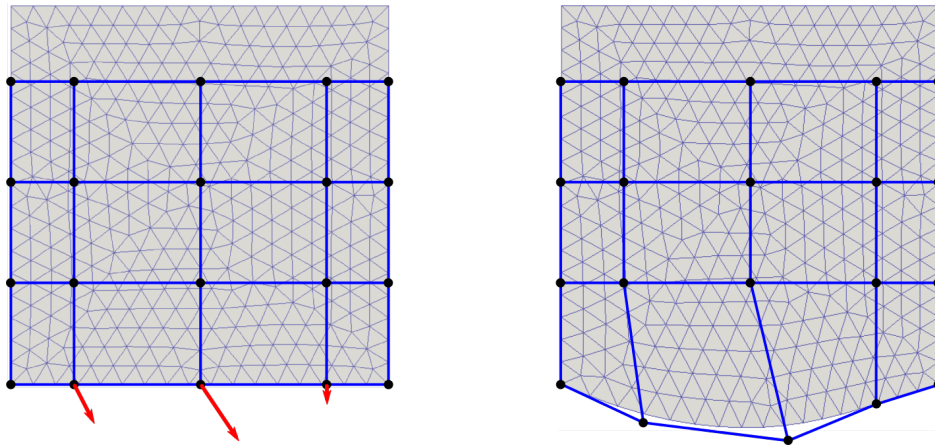
- Write the deformed mesh and the spline control net (the net of control points) into vtk files:

```

spb.write_control_net('square_tri1_splibox.vtk')
mesh.write('square_tri1_deform.vtk')

```

The following figures show the undeformed (left) and deformed (right) mesh and the control net.



SplineRegion2D

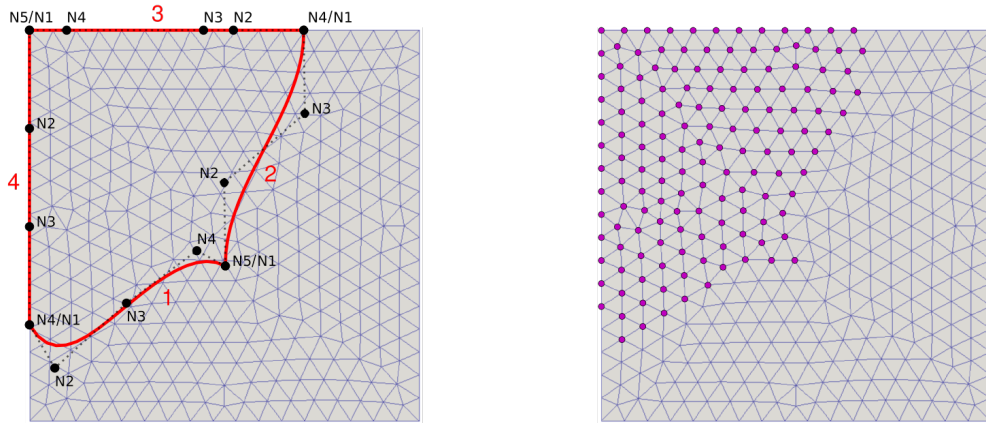
In this case, the region (only in 2D) of parametrization is defined by four B-spline curves:

```

1 from sfepy.mesh.splibox import SplineRegion2D
2
3 spb = SplineRegion2D([<bspl1>, <bspl2>, <bspl3>, <bspl4>], <coors>)

```

The curves must form a closed loop, must be oriented counterclockwise and the opposite curves (<bspl1>, <bspl3> and <bspl2>, <bspl4>) must have the same number of control points and the same knot vectors, see the figure below, on the left.



The position of the selected vertices, depicted in the figure on the right, are driven by the control points in the same way as explained above for `SplineBox`.

Note: Initializing `SplineRegion2D` may be time consuming due to the fact that for all vertex coordinates the spline parameters have to be found using an optimization method in which the B-spline basis is repeatedly evaluated.

Example

- First of all, define four B-spline curves (the default degree of the spline curve is 3) representing the boundary of a parametrization area:

```
1 from sfepy.mesh.bspline import BSpline
2
3 # left / right boundary
4 line_l = nm.array([[ -1,  1], [ -1, .5], [ -1,  0], [ -1, -.5]])
5 line_r = nm.array([[ 0, -.2], [ .1, .2], [ .3, .6], [ .4,  1]])
6
7 sp_l = BSpline()
8 sp_l.approximate(line_l, ncp=4)
9 kn_lr = sp_l.get_knot_vector()
10
11 sp_r = BSpline()
12 sp_r.approximate(line_r, knots=kn_lr)
13
14 # bottom / top boundary
15 line_b = nm.array([[ -1, -.5], [ -.8, -.6], [ -.5, -.4], [ -.2, -.2], [ 0, -.2]])
16 line_t = nm.array([[ .4,  1], [ 0,  1], [ -.2,  1], [ -.6,  1], [ -1,  1]])
17
18 sp_b = BSpline()
19 sp_b.approximate(line_b, ncp=5)
20 kn_bt = sp_b.get_knot_vector()
21
22 sp_t = BSpline()
23 sp_t.approximate(line_t, knots=kn_bt)
```

- Create a new `2D SplineRegion2D` object:

```

1 from sfepy.mesh.splinebox import SplineRegion2D
2
3 spb = SplineRegion2D([sp_b, sp_r, sp_t, sp_l], mesh.coors)

```

- Move the control points:

```

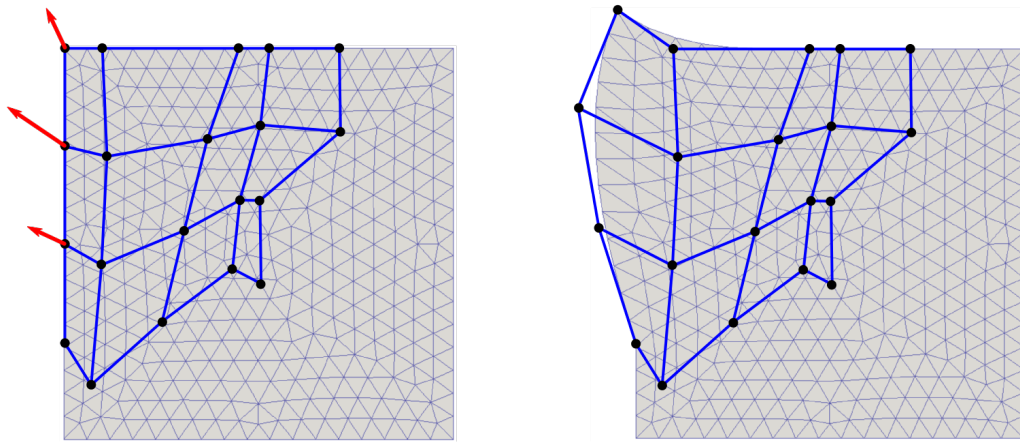
1 spb.move_control_point(5, [-.2, .1])
2 spb.move_control_point(10, [-.3, .2])
3 spb.move_control_point(15, [-.1, .2])

```

- Evaluate the new coordinates:

```
mesh.cmesh.coors[:] = spb.evaluate()
```

The figures below show the undeformed (left) and deformed (right) mesh and the control net.



1.5.6 Examples

acoustics

acoustics/acoustics.py

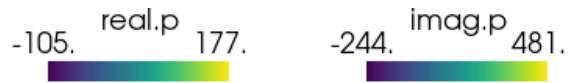
Description

Acoustic pressure distribution.

This example shows how to solve a problem in complex numbers, note the ‘acoustic_pressure’ field definition.

Find p such that:

$$c^2 \int_{\Omega} \nabla q \cdot \nabla p - w^2 \int_{\Omega} qp - iwc \int_{\Gamma_{out}} qp = iwc^2 \rho v_n \int_{\Gamma_{in}} q, \quad \forall q.$$



source code

```

r"""
Acoustic pressure distribution.

This example shows how to solve a problem in complex numbers, note the
'accoustic_pressure' field definition.

Find :math:`p` such that:

.. math::
    c^2 \int_{\Omega} \nabla q \cdot \nabla p
    - w^2 \int_{\Omega} q p
    - i w c \int_{\Gamma_{out}} q p
    = i w c^2 \rho v_n \int_{\Gamma_{in}} q
    \;, \quad \forall q \;.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/2d/special/two_rectangles.mesh'

v_n = 1.0 # m/s

```

(continues on next page)

(continued from previous page)

```

w = 1000.0
c = 343.0 # m/s
rho = 1.55 # kg/m^3

options = {
    'nls' : 'newton',
    'ls' : 'ls',
}

materials = {
    'one' : ({'one' : 1.0},),
}

regions = {
    'Omega' : 'all',
    'Gamma_in' : ('vertices in (x < 0.01)', 'facet'),
    'Gamma_out' : ('vertices in (x > 0.99)', 'facet'),
}

fields = {
    'acoustic_pressure' : ('complex', 1, 'Omega', 1),
}

variables = {
    'p' : ('unknown field', 'acoustic_pressure', 0),
    'q' : ('test field', 'acoustic_pressure', 'p'),
}

ebcs = {
}

integrals = {
    'i' : 2,
}

equations = {
    'Acoustic pressure' :
        """%s * dw_laplace.i.Omega( one.one, q, p )
        - %s * dw_dot.i.Omega( q, p )
        - %s * dw_dot.i.Gamma_out( q, p )
        = %s * dw_integrate.i.Gamma_in( q )"""
        % (c*c, w*w, 1j*w*c, 1j*w*c*c*rho*v_n)
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-1,
        'eps_r' : 1.0,
        'macheps' : 1e-16,
        'lin_red' : 1e-1, # Linear system error < (eps_a * lin_red).
    })
}

```

(continues on next page)

(continued from previous page)

```

        'ls_red'      : 0.1,
        'ls_red_warp' : 0.001,
        'ls_on'       : 1.1,
        'ls_min'      : 1e-5,
        'check'       : 0,
        'delta'       : 1e-6,
    })
}

```

acoustics/acoustics3d.py

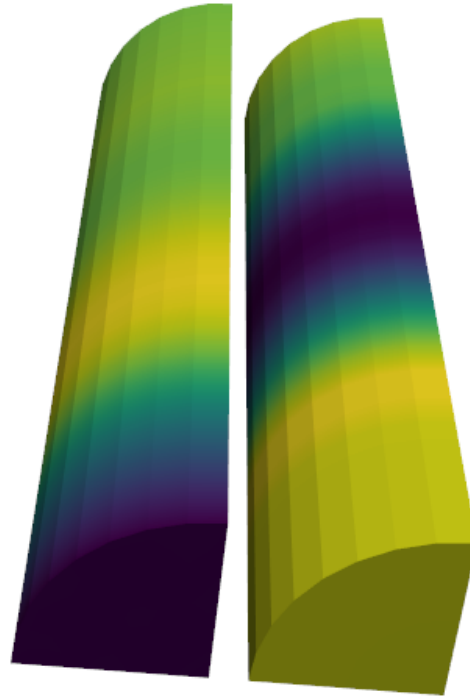
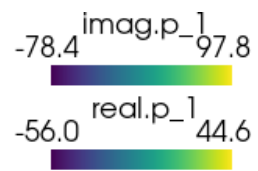
Description

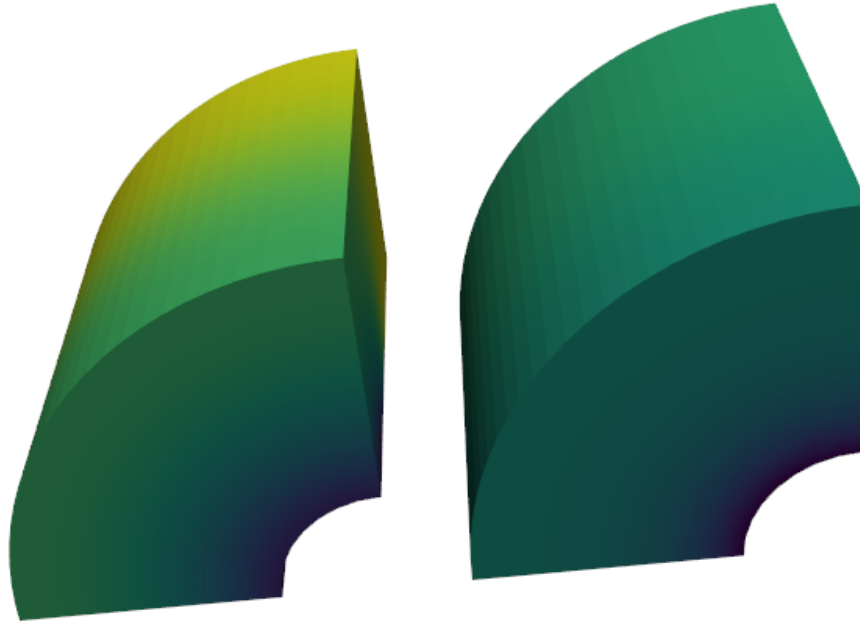
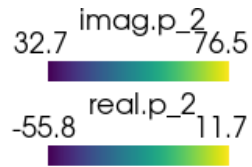
Acoustic pressure distribution in 3D.

Two Laplace equations, one in Ω_1 , other in Ω_2 , connected on the interface region Γ_{12} using traces of variables.

Find two complex acoustic pressures p_1, p_2 such that:

$$\begin{aligned}
 & \int_{\Omega} k^2 qp - \int_{\Omega} \nabla q \cdot \nabla p \\
 & - iw/c \int_{\Gamma_{out}} qp + iw\rho/Z \int_{\Gamma_2} q(p_2 - p_1) + iw\rho/Z \int_{\Gamma_1} q(p_1 - p_2) \\
 & = iw\rho \int_{\Gamma_{in}} v_n q, \quad \forall q.
 \end{aligned}$$





source code

```

r"""
Acoustic pressure distribution in 3D.

Two Laplace equations, one in  $\Omega_1$ , other in
 $\Omega_2$ , connected on the interface region  $\Gamma_{12}$ 
using traces of variables.

Find two complex acoustic pressures  $p_1$ ,  $p_2$  such that:

.. math::
\int_{\Omega} k^2 q p - \int_{\Omega} \nabla q \cdot \nabla p \backslash\backslash
- i w/c \int_{\Gamma_{out}} q p
+ i w \rho/Z \int_{\Gamma_2} q (p_2 - p_1)
+ i w \rho/Z \int_{\Gamma_1} q (p_1 - p_2) \backslash\backslash
= i w \rho \int_{\Gamma_{in}} v_n q
\backslash;, \quad \forall q \backslash;.
"""

from __future__ import absolute_import
from sfepy import data_dir

```

(continues on next page)

(continued from previous page)

```

filename_mesh = data_dir + '/meshes/3d/acoustics_mesh3d.mesh'

freq = 1200
v_n = 1.0 # m/s
c = 343.0 # m/s
rho = 1.55 # kg/m^3
R = 1000
w = 2.0 * freq

k1 = w / c
rhoc1 = rho * c

coef_k = ((1.0 + 0.1472 * (freq / R)**(-0.577))
          + 1j * (-0.1734 * (freq / R)**(-0.595)))
coef_r = ((1.0 + 0.0855 * (freq / R)**(-0.754))
          + 1j * (-0.0765 * (freq / R)**(-0.732)))

k2 = k1 * coef_k
rhoc2 = rhoc1 * coef_r

# perforation geometry parameters
tw = 0.9e-3
dh = 2.49e-3
por = 0.08

# acoustic impedance
Z = rho * c / por * (0.006 + 1j * k1 * (tw + 0.375 * dh
                                     * (1 + rhoc2/rhoc1 * k2/k1)))

regions = {
    'Omega' : 'all',
    'Omega_1' : 'cells of group 1',
    'Omega_2' : 'cells of group 2',
    'Gamma_12' : ('r.Omega_1 *v r.Omega_2', 'facet'),
    'Gamma_12_1' : ('copy r.Gamma_12', 'facet', 'Omega_1'),
    'Gamma_12_2' : ('copy r.Gamma_12', 'facet', 'Omega_2'),
    'Gamma_in' : ('vertices in (z < 0.001)', 'facet'),
    'Gamma_out' : ('vertices in (z > 0.157)', 'facet'),
}

materials = {
}

fields = {
    'acoustic_pressure_1' : ('complex', 'scalar', 'Omega_1', 1),
    'acoustic_pressure_2' : ('complex', 'scalar', 'Omega_2', 1),
}

variables = {
    'p_1' : ('unknown field', 'acoustic_pressure_1'),
    'q_1' : ('test field', 'acoustic_pressure_1', 'p_1'),
    'p_2' : ('unknown field', 'acoustic_pressure_2'),
}

```

(continues on next page)

(continued from previous page)

```

    'q_2' : ('test field', 'acoustic_pressure_2', 'p_2'),
}

ebcs = {
}

integrals = {
    'i' : 2,
}

equations = {
    'Acoustic pressure' :
        """%s * dw_dot.i.Omega_1(q_1, p_1)
        + %s * dw_dot.i.Omega_2(q_2, p_2)
        - dw_laplace.i.Omega_1(q_1, p_1)
        - dw_laplace.i.Omega_2(q_2, p_2)
        - %s * dw_dot.i.Gamma_out(q_1, p_1)
        + %s * dw_jump.i.Gamma_12_1(q_1, p_1, tr(p_2))
        + %s * dw_jump.i.Gamma_12_2(q_2, p_2, tr(p_1))
        = %s * dw_integrate.i.Gamma_in(q_1)"""
        % (k1*k1, k2*k2,
           1j*k1,
           1j*k1*rhoc1 / Z, 1j*k2*rhoc2 / Z,
           1j*k1*rhoc1 * v_n)
}

options = {
    'nls': 'newton',
    'ls': 'ls',
    'split_results_by': 'region',
    'output_dir': 'output',
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
        'macheps' : 1e-16,
        'lin_red' : 1e-1,
        'ls_red' : 0.1,
        'ls_red_warp' : 0.001,
        'ls_on' : 1.1,
        'ls_min' : 1e-5,
        'check' : 0,
        'delta' : 1e-6,
    })
}

```

acoustics/helmholtz_apartment.py

Description

A script demonstrating the solution of the scalar Helmholtz equation for a situation inspired by the physical problem of WiFi propagation in an apartment.

The example is an adaptation of the project found here: <https://bthierry.pages.math.cnrs.fr/course-fem/projet/2017-2018/>

The boundary conditions are defined as perfect radiation conditions, meaning that on the boundary waves will radiate outside.

The PDE for this physical process implies to find $E(x, t)$ for $x \in \Omega$ such that:

$$\begin{cases} \Delta E + k^2 n(x)^2 E = f_s & \forall x \in \Omega \\ \partial_n E(x) - i k n(x) E(x) = 0 & \forall x \text{ on } \partial\Omega \end{cases}$$

Usage

The mesh of the appartement and the different material ids can be visualized with the following:

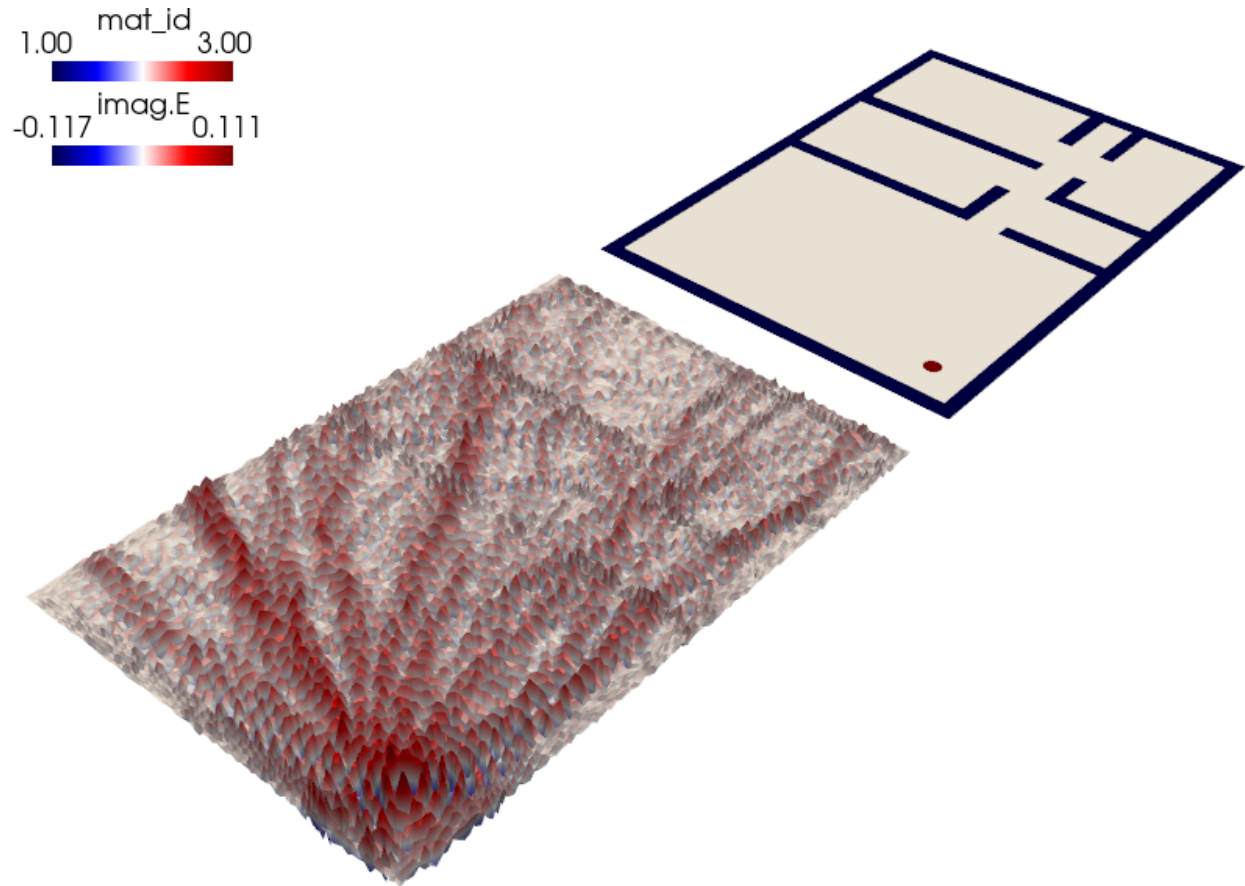
```
sfepy-view meshes/2d/helmholtz_apartment.vtk -e -2
```

The example is run from the top level directory as:

```
sfepy-run sfepy/examples/acoustics/helmholtz_apartment.py
```

The result of the computation can be visualized as follows:

```
sfepy-view helmholtz_apartment.vtk --color-map=seismic -f imag.E:wimag.E:f10%p0 --  
↪ camera-position="-5.14968,-7.27948,7.08783,-0.463265,-0.23358,-0.350532,0.160127,0.  
↪ 664287,0.730124"
```



source code

```
r"""
A script demonstrating the solution of the scalar Helmholtz equation for a
situation inspired by the physical problem of WiFi propagation in an apartment.

The example is an adaptation of the project found here:
https://bthierry.pages.math.cnrs.fr/course-fem/projet/2017-2018/

The boundary conditions are defined as perfect radiation conditions, meaning
that on the boundary waves will radiate outside.

The PDE for this physical process implies to find  $E(x, t)$  for  $x$ 
 $\in \Omega$  such that:

.. math::
    \left\{ \begin{aligned}
        \Delta E + k^2 n(x)^2 E &= f_s \quad \text{for all } x \in \Omega \\
        \partial_n E(x) - i k n(x) E(x) &= 0 \quad \text{for all } x \text{ on } \partial \Omega
    \end{aligned} \right.
```

(continues on next page)

(continued from previous page)

Usage

The mesh of the appartement and the different material ids can be visualized with the following::

```
sfepy-view meshes/2d/helmholtz_apartment.vtk -e -2
```

The example is run from the top level directory as::

```
sfepy-run sfepy/examples/acoustics/helmholtz_apartment.py
```

The result of the computation can be visualized as follows::

```
sfepy-view helmholtz_apartment.vtk --color-map=seismic -f imag.E:wimag.E:f10%p0 --
↪camera-position="-5.14968,-7.27948,7.08783,-0.463265,-0.23358,-0.350532,0.160127,0.
↪664287,0.730124"
"""
```

```
import numpy as nm
from sfepy import data_dir
```

```
f = 2.4 * 1e9 # change this to 2.4 or 5 if you want to simulate frequencies
               # typically used by your Wifi
```

```
c0 = 3e8 # m/s
```

```
k = 2 * nm.pi * f / c0
```

```
filename_mesh = data_dir + "/meshes/2d/helmholtz_apartment.vtk"
```

```
regions = {
    'Omega': 'all',
    'Walls': ('cells of group 1'),
    'Air': ('cells of group 2'),
    'Source': ('cells of group 3'),
    'Gamma': ('vertices of surface', 'facet'),
}
```

air and walls have different material parameters, hence the 1. and 2.4 factors

```
materials = {
    'material': ({'kn_square': {
        'Air': (k * 1.) ** 2,
        'Walls': (k * 2.4) ** 2}, },),
    'boundary': ({'kn': 1j * k * 2.4},),
    'source_func': 'source_func',
}
```

```
def source_func(ts, coors, mode=None, **kwargs):
    """The source function for the antenna."""
    epsilon = 0.1
    x_center = nm.array([-3.23, -1.58])
    if mode == 'qp':
```

(continues on next page)

(continued from previous page)

```

    dists = nm.abs(nm.sum(nm.square(coors - x_center), axis=1))
    val = 3 / nm.pi / epsilon ** 2 * (1 - dists / epsilon)
    return {'val': val[:, nm.newaxis, nm.newaxis]}

functions = {
    'source_func': (source_func,),
}

fields = {
    'electric_field': ('complex', 1, 'Omega', 1),
}

variables = {
    'E': ('unknown field', 'electric_field', 1),
    'q': ('test field', 'electric_field', 'E'),
}

ebcs = {
}

integrals = {
    'i': 2,
}

equations = {
    'Helmholtz equation':
        """- dw_laplace.i.Omega(q, E)
        + dw_dot.i.Omega(material.kn_square, q, E)
        + dw_dot.i.Gamma(boundary.kn, q, E)
        = dw_integrate.i.Source(source_func.val, q)
        """
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
    }),
}

options = {
    'refinement_level': 3,
}

```

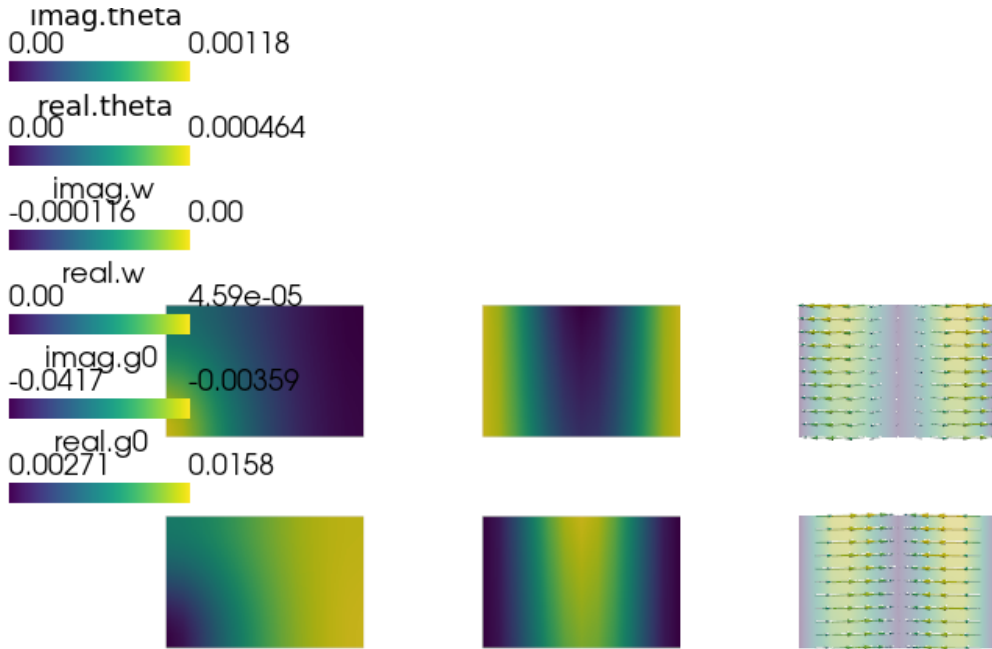
acoustics/vibro_acoustic3d.py**Description**

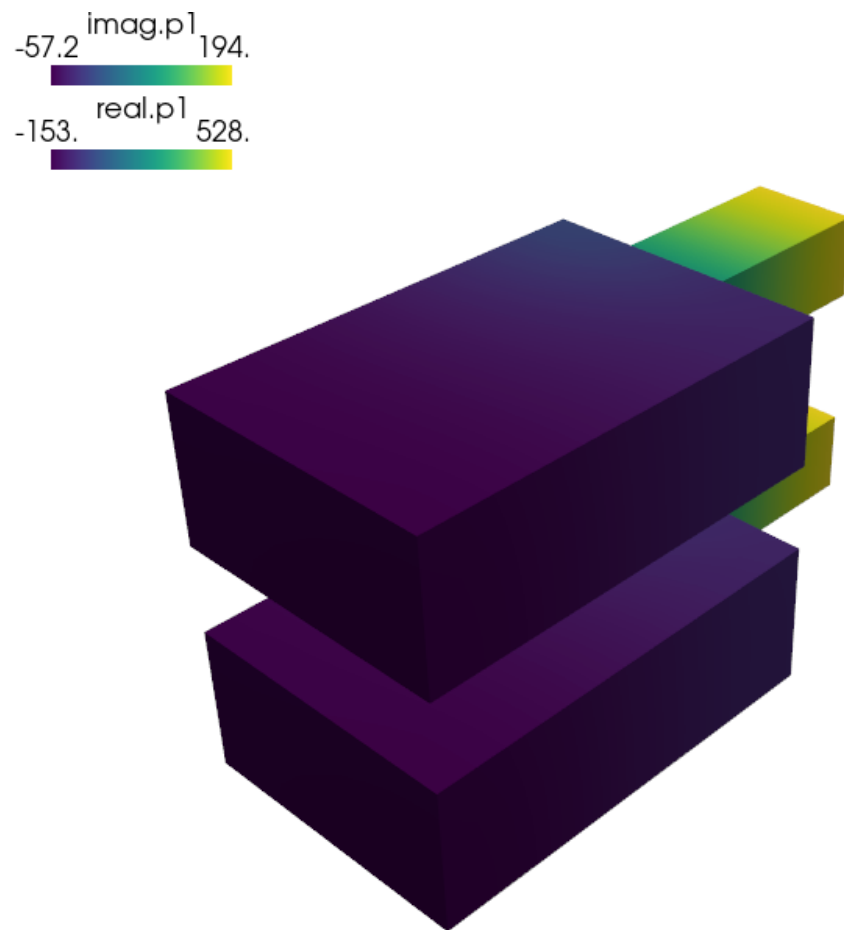
Vibro-acoustic problem

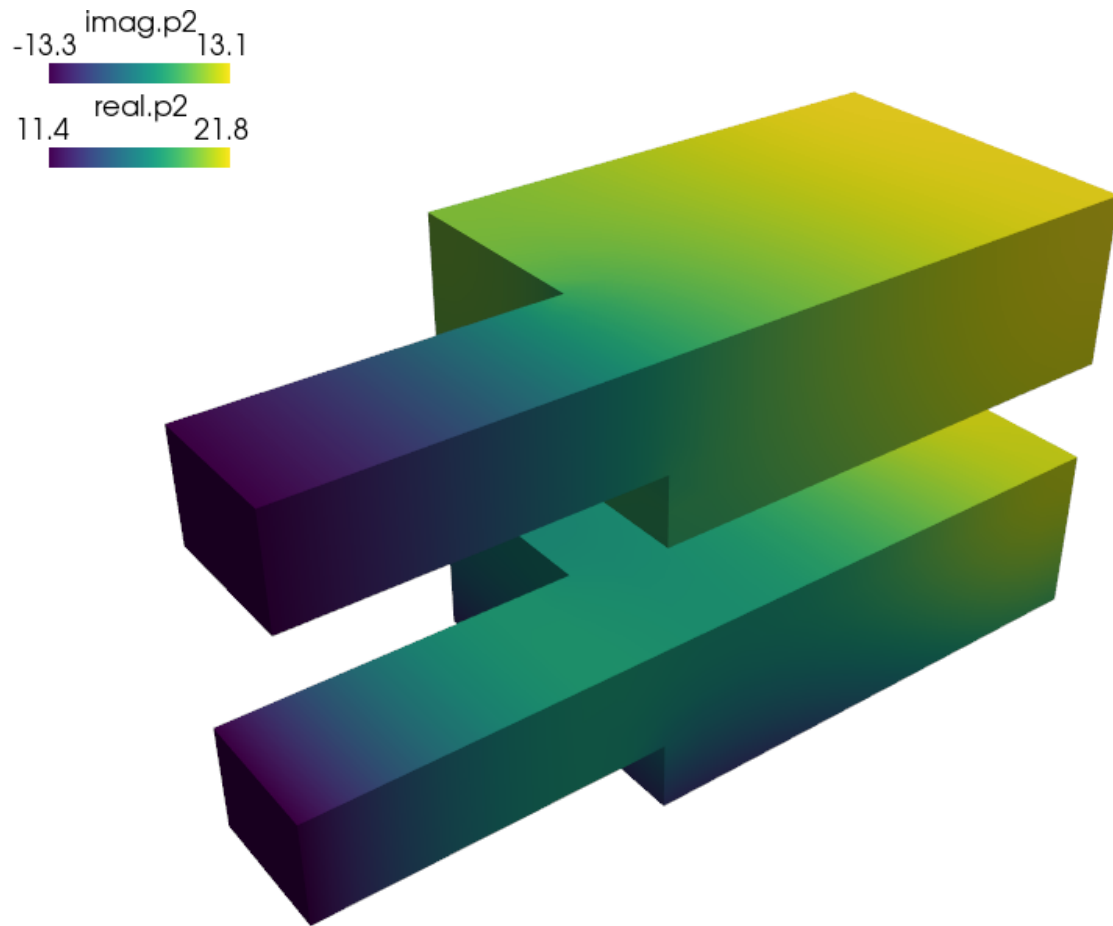
3D acoustic domain with 2D perforated deforming interface.

Problem definition - find p (acoustic pressure), g (transversal acoustic velocity), w (plate deflection) and $\underline{\theta}$ (rotation) such that:

$$\begin{aligned}
 c^2 \int_{\Omega} \nabla q \cdot \nabla p - \omega^2 \int_{\Omega} qp + i\omega c \int_{\Gamma_{in}} qp + i\omega c \int_{\Gamma_{out}} qp - i\omega c^2 \int_{\Gamma_0} (q^+ - q^-)g &= 2i\omega c \int_{\Gamma_{in}} q\bar{p}, \quad \forall q, \\
 -i\omega \int_{\Gamma_0} f(p^+ - p^-) - \omega^2 \int_{\Gamma_0} Ffg + \omega^2 \int_{\Gamma_0} Cfw &= 0, \quad \forall f, \\
 \omega^2 \int_{\Gamma_0} Czg - \omega^2 \int_{\Gamma_0} Szw + \int_{\Gamma_0} \nabla z \cdot \underline{G} \cdot \nabla w - \int_{\Gamma_0} \underline{\theta} \cdot \underline{G} \cdot \nabla z &= 0, \quad \forall z, \\
 -\omega^2 \int_{\Gamma_0} R\underline{\nu} \cdot \underline{\theta} + \int_{\Gamma_0} D_{ijkl}e_{ij}(\underline{\nu})e_{kl}(\underline{\theta}) - \int_{\Gamma_0} \underline{\nu} \cdot \underline{G} \cdot \nabla w + \int_{\Gamma_0} \underline{\nu} \cdot \underline{G} \cdot \underline{\theta} &= 0, \quad \forall \underline{\nu},
 \end{aligned}$$







source code

```

r"""
Vibro-acoustic problem

3D acoustic domain with 2D perforated deforming interface.

Problem definition - find :math:p` (acoustic pressure),
:math:g` (transversal acoustic velocity),
:math:w` (plate deflection) and :math:\ul{\theta}` (rotation) such that:

.. math::
    c^2 \int_{\Omega} \nabla q \cdot \nabla p
    - \omega^2 \int_{\Omega} q p
    + i \omega c \int_{\Gamma_{in}} q p
    + i \omega c \int_{\Gamma_{out}} q p
    - i \omega c^2 \int_{\Gamma_0} (q^{+} - q^{-}) g
    = 2i \omega c \int_{\Gamma_{in}} q \bar{p}
    \;, \quad \forall q \;,

    - i \omega \int_{\Gamma_0} f (p^{+} - p^{-})
    - \omega^2 \int_{\Gamma_0} F f g
    + \omega^2 \int_{\Gamma_0} C f w

```

(continues on next page)

(continued from previous page)

```

= 0
\;, \quad \forall f \;,

\omega^2 \int_{\Gamma_0} C z g
- \omega^2 \int_{\Gamma_0} S z w
+ \int_{\Gamma_0} \nabla z \cdot \mathbf{u}_G \cdot \nabla w
- \int_{\Gamma_0} \mathbf{u}_\theta \cdot \mathbf{u}_G \cdot \nabla z
= 0
\;, \quad \forall z \;,

- \omega^2 \int_{\Gamma_0} R \cdot \mathbf{u}_\nu \cdot \mathbf{u}_\theta
+ \int_{\Gamma_0} D_{ijkl} e_{ij}(\mathbf{u}_\nu) e_{kl}(\mathbf{u}_\theta)
- \int_{\Gamma_0} \mathbf{u}_\nu \cdot \mathbf{u}_G \cdot \nabla w
+ \int_{\Gamma_0} \mathbf{u}_\nu \cdot \mathbf{u}_G \cdot \mathbf{u}_\theta
= 0
\;, \quad \forall \mathbf{u}_\nu \;,
"""
import numpy as nm
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_lame

def define(sound_speed=343.0,
           wave_num=5.5,
           p_inc=300,
           thickness=0.01,
           filename_mesh='/meshes/3d/acoustic_wg.vtk'):

    filename_mesh = data_dir + filename_mesh

    c = sound_speed
    c2 = c**2
    w = wave_num * c
    w2 = w**2
    wc = w * c
    wc2 = w * c2

    regions = {
        'Omega1': 'cells of group 1',
        'Omega2': 'cells of group 2',
        'GammaIn': ('vertices of group 1', 'face'),
        'GammaOut': ('vertices of group 2', 'face'),
        'Gamma_aux': ('r.Omega1 * v r.Omega2', 'face'),
        'Gamma0_1': ('copy r.Gamma_aux', 'face', 'Omega1'),
        'Gamma0_2': ('copy r.Gamma_aux', 'face', 'Omega2'),
        'Gamma0': ('copy r.Gamma_aux', 'cell', None, {'mesh_dim': 2}),
        'Left_': ('vertices in (x < 0.001)', 'edge'),
        'Right_': ('vertices in (x > 0.299)', 'edge'),
        'Gamma0_Left': ('r.Gamma_aux * v r.Left_', 'edge'),
        'Gamma0_Right': ('r.Gamma_aux * v r.Right_', 'edge'),
    }

```

(continues on next page)

(continued from previous page)

```

fields = {
    'pressure1': ('complex', 'scalar', 'Omega1', 1),
    'pressure2': ('complex', 'scalar', 'Omega2', 1),
    'tvelocity': ('complex', 'scalar', 'Gamma0', 1),
    'deflection': ('complex', 'scalar', 'Gamma0', 1),
    'rotation': ('complex', 'vector', 'Gamma0', 1),
}

variables = {
    'p1': ('unknown field', 'pressure1', 0),
    'q1': ('test field', 'pressure1', 'p1'),
    'p2': ('unknown field', 'pressure2', 1),
    'q2': ('test field', 'pressure2', 'p2'),
    'g0': ('unknown field', 'tvelocity', 2),
    'f0': ('test field', 'tvelocity', 'g0'),
    'w': ('unknown field', 'deflection', 3),
    'z': ('test field', 'deflection', 'w'),
    'theta': ('unknown field', 'rotation', 4),
    'nu': ('test field', 'rotation', 'theta'),
}

ebcs = {
    'fixed_l': ('Gamma0_Left', {'w.0': 0.0, 'theta.all': 0.0}),
    'fixed_r': ('Gamma0_Right', {'w.0': 0.0, 'theta.all': 0.0}),
}

options = {
    'split_results_by': 'region',
    'output_dir': 'output',
}

functions = {
}

materials = {
    'ac': ({
        'F': -2.064e+00,
        'c': -1.064e+00,
        'T': 9.202e-01,
        'hG': thickness * 4.5e10 * nm.eye(2),
        'hR': thickness * 0.71,
        'h3R': thickness**3 / 3.0 * 0.71,
        'h3C': thickness**3 / 3.0 * stiffness_from_lame(2, 1e1, 1e0)}, ),
}

equations = {
    'eq_p1': """
        %e * dw_laplace.5.Omega1(q1, p1)
    - %e * dw_dot.5.Omega1(q1, p1)
    + %s * dw_dot.5.GammaIn(q1, p1)
    - %s * dw_dot.5.Gamma0_1(q1, tr(Gamma0, g0))
    = %s * dw_integrate.5.GammaIn(q1)""" % (c2, w2, 1j * wc,

```

(continues on next page)

(continued from previous page)

```

1j * wc2, 2j * wc * p_inc),

'eq_p2': ""
+ %e * dw_laplace.5.Omega2(q2, p2)
- %e * dw_dot.5.Omega2(q2, p2)
+ %s * dw_dot.5.GammaOut(q2, p2)
+ %s * dw_dot.5.Gamma0_2(q2, tr(Gamma0, g0))
= 0"" % (c2, w2, 1j * wc, 1j * wc2),
'eq_g0': ""
- %s * dw_dot.5.Gamma0(f0, tr(Gamma0_1, p1))
+ %s * dw_dot.5.Gamma0(f0, tr(Gamma0_2, p2))
- %e * dw_dot.5.Gamma0(ac.F, f0, g0)
+ %e * dw_dot.5.Gamma0(ac.c, f0, w)
= 0"" % (1j * w, 1j * w, w2, w2),
'eq_w': ""
    %e * dw_dot.5.Gamma0(ac.c, z, g0)
- %e * dw_dot.5.Gamma0(ac.T, z, w)
- %e * dw_dot.5.Gamma0(ac.hR, z, w)
    + dw_diffusion.5.Gamma0(ac.hG, z, w)
    - dw_v_dot_grad_s.5.Gamma0(ac.hG, theta, z)
= 0"" % (w2, w2, w2),
'eq_theta': ""
- %e * dw_dot.5.Gamma0(ac.h3R, nu, theta)
    + dw_lin_elastic.5.Gamma0(ac.h3C, nu, theta)
    - dw_v_dot_grad_s.5.Gamma0(ac.hG, nu, w)
    + dw_dot.5.Gamma0(ac.hG, nu, theta)
= 0"" % (w2, ),
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'nls': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-4,
        'eps_r': 1e-6,
    })
}

return locals()

```

dg

dg/advection_1D.py

Description

Transient advection equation in 1D solved using discontinuous galerkin method.

$$\frac{dp}{dt} + a \cdot dp/dx = 0$$

$$p(t, 0) = p(t, 1)$$

Usage Examples

Run:

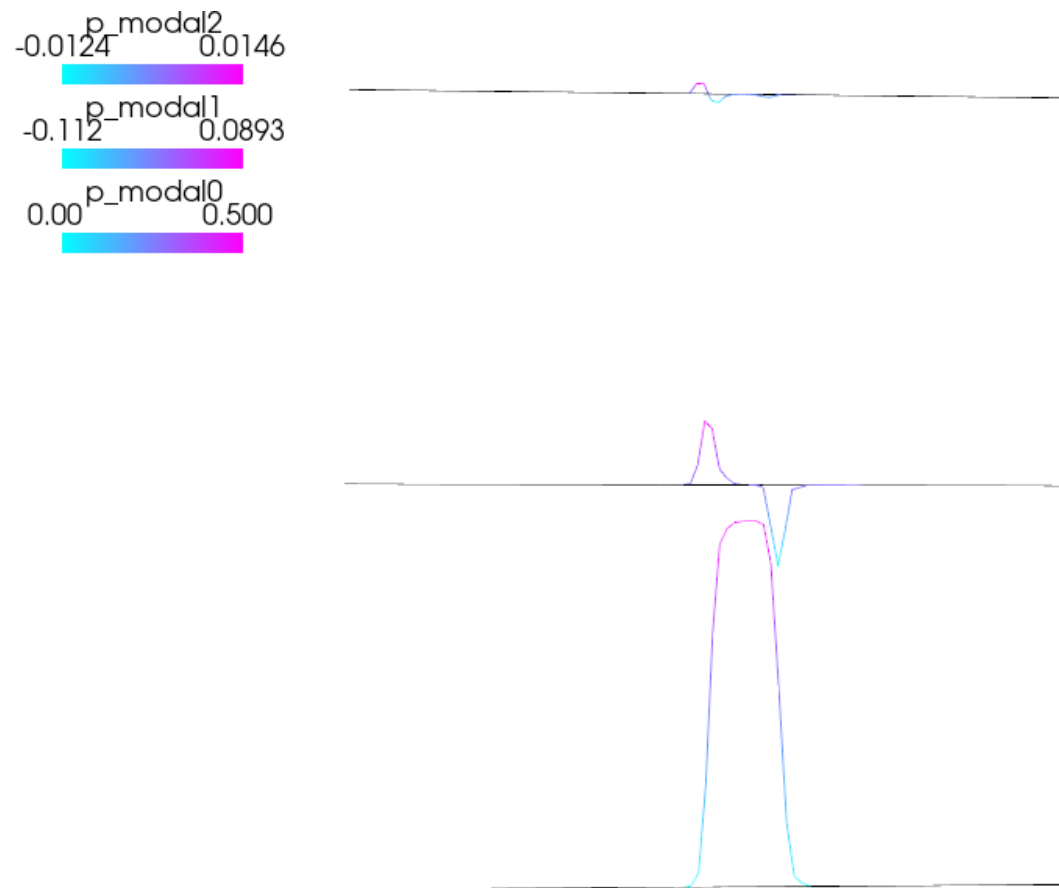
```
sfepy-run sfepy/examples/dg/advection_1D.py
```

To view animated results use `sfepy/examples/dg/dg_plot_1D.py` specifying name of the output in `output/` folder, default is `dg/advection_1D`:

```
python3 sfepy/examples/dg/dg_plot_1D.py dg/advection_1D
```

`dg_plot_1D.py` also accepts full and relative paths:

```
python3 sfepy/examples/dg/dg_plot_1D.py output/dg/advection_1D
```



source code

```
r"""
Transient advection equation in 1D solved using discontinuous galerkin method.

.. math:: \frac{dp}{dt} + a \cdot \frac{dp}{dx} = 0

p(t,0) = p(t,1)
```

(continues on next page)

(continued from previous page)

Usage Examples

*Run::**sfepy-run sfepy/examples/dg/advection_1D.py**To view animated results use ``sfepy/examples/dg/dg\_plot\_1D.py`` specifying name of the output in ``output/`` folder, default is ``dg/advection\_1D``::**python3 sfepy/examples/dg/dg_plot_1D.py dg/advection_1D**``dg\_plot\_1D.py`` also accepts full and relative paths::**python3 sfepy/examples/dg/dg_plot_1D.py output/dg/advection_1D
"""**from sfepy.examples.dg.example_dg_common import *
from sfepy.discrete.dg.limiters import MomentLimiter1D**dim = 1**def define(filename_mesh=None,
 approx_order=2,

 adflux=0.0,
 limit=True,

 cw=None,
 diffcoef=None,
 diffscheme="symmetric",

 cfl=0.4,
 dt=None,
 t1=0.1
):**t0 = 0
transient = True**mstart = 0
mend = 1**diffcoef = None
cw = None**example_name = "advection_1D"
dim = 1**if filename_mesh is None:
 filename_mesh = get_gen_1D_mesh_hook(0, 1, 100)**materials = {*

(continues on next page)

(continued from previous page)

```

    'a': ({'val': [1.0], '.flux': adflux},),
}

regions = {
    'Omega': 'all',
    'Gamma': ('vertices of surface', 'facet'),
    'left': ('vertices in x == 0', 'vertex'),
    'right': ('vertices in x == 1', 'vertex')
}

fields = {
    'f': ('real', 'scalar', 'Omega', str(approx_order) + 'd', 'DG', 'legendre')
}

variables = {
    'p': ('unknown field', 'f', 0, 1),
    'v': ('test field', 'f', 'p'),
}

dgebcs = {
    'u_left': ('left', {'p.all': 0}),
    'u_right': ('right', {'p.all': 0}),
}

dgepbcs_1 = {
    'name' : 'u_r1',
    'region': ['right', 'left'],
    'dofs': {'p.all': 'p.all'},
    'match': 'match_y_line',
}

integrals = {
    'i': 2 * approx_order,
}

equations = {
    'Advection': """
        dw_dot.i.Omega(v, p)
        - dw_s_dot_mgrad_s.i.Omega(a.val, p[-1], v)
        + dw_dg_advect_laxfrie_flux.i.Omega(a.flux, a.val, v, p[-1]) = 0
    """
}

solvers = {
    "tss": ('ts.tvd_runge_kutta_3',
           {"t0" : t0,
            "t1" : t1,
            'limiters': {'f': MomentLimiter1D} if limit else {}
           }),
    'nls': ('nls.newton', {}),
    'ls' : ('ls.scipy_direct', {})
}

```

(continues on next page)

(continued from previous page)

```

}

options = {
    'ts'          : 'tss',
    'nls'         : 'newton',
    'ls'          : 'ls',
    'save_times'  : 100,
    'active_only' : False,
    'pre_process_hook': get_cfl_setup(cfl)
                        if dt is None else
                        get_cfl_setup(dt=dt),
    'output_dir'  : 'output/dg/' + example_name,
    'output_format' : "vtk",
}

functions = {}

def local_register_function(fun):
    try:
        functions.update({fun.__name__: (fun,)})

    except AttributeError: # Already a sfepy Function.
        fun = fun.function
        functions.update({fun.__name__: (fun,)})

    return fun

def four_step_p(x):
    """
    piecewise constant  $(-\infty, 1.8], (1.8, a + 4](a+4, a + 5](a + 5, \infty)$ 
    """
    return nm.pieceswise(x,
                        [x <= mstart,
                         x <= mstart + .4,
                         mstart + .4 < x,
                         mstart + .5 <= x],
                        [0, 0, .5, 0])

@local_register_function
def get_ic(x, ic=None):
    return four_step_p(x)

def analytic_sol(coors, t=None, uset=False):
    x = coors[..., 0]
    if uset:
        res = get_ic(x[..., None] - t[None, ...])
        return res # for animating transient problem

    res = get_ic(x[..., None])
    return res[..., 0]

```

(continues on next page)

(continued from previous page)

```

@local_register_function
def sol_fun(ts, coors, mode="qp", **kwargs):
    t = ts.time
    if mode == "qp":
        return {"p": analytic_sol(coors, t)[..., None, None]}

    ics = {
        'ic': ('Omega', {'p.0': 'get_ic'}),
    }

    return locals()

```

dg/advection_2D.py

Description

Transient advection equation in 2D solved by discontinuous Galerkin method.

$$\frac{dp}{dt} + a \cdot \text{grad } p = 0$$

Usage Examples

Run:

```
sfepy-run sfepy/examples/dg/advection_2D.py
```

Results are saved to output/dg/advection_2D folder by default as .msh files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use File | Open menu or Ctrl + O shortcut, navigate to the output folder, select all .msh files and hit Open, all files should load as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on All view options.... This brings up the Options window with View [0] selected in left column. Under the tab General ensure that Adapt visualization grid is ticked, then you can adjust Maximum recursion depth and Target visualization error to tune the visualization. To see visualization elements (as opposed to mesh elements) go to Visibility tab and tick Draw element outlines, this option is also available from quick options menu as View element outlines or under shortcut Alt+E. In the quick options menu, you can also modify normal raise by clicking View Normal Raise to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

source code

```

r"""
Transient advection equation in 2D solved by discontinuous Galerkin method.

.. math:: \frac{dp}{dt} + a \cdot \text{grad } p = 0

Usage Examples
-----

```

(continues on next page)

(continued from previous page)

Run::

```
sfePy-run sfePy/examples/dg/advection_2D.py
```

Results are saved to output/dg/advection_2D folder by default as ``.msh`` files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use ``File | Open`` menu or Ctrl + O shortcut, navigate to the output folder, select all ``.msh`` files and hit Open, all files should load as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on ``All view options...``. This brings up the Options window with ``View [0]`` selected in left column. Under the tab ``General`` ensure that ``Adapt visualization grid`` is ticked, then you can adjust ``Maximum recursion depth`` and ``Target visualization error`` to tune the visualization. To see visualization elements (as opposed to mesh elements) go to ``Visibility`` tab and tick ``Draw element outlines``, this option is also available from quick options menu as ``View element outlines`` or under shortcut ``Alt+E``. In the quick options menu, you can also modify normal raise by clicking ``View Normal Raise`` to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

```
"""
from sfePy.examples.dg.example_dg_common import *
from sfePy.discrete.dg.limiters import MomentLimiter2D

mesh_center = (0.5, 0.25)
mesh_size = (1.0, 0.5)

def define(filename_mesh=None,
            approx_order=2,

            adflux=0,
            limit=True,

            cw=None,
            diffcoef=None,
            diffscheme="symmetric",

            cfl=0.4,
            dt=None,
            t1=0.01

            ):

    example_name = "advection_2D"
    dim = 2

    diffcoef = None
    cw = None
```

(continues on next page)

(continued from previous page)

```

if filename_mesh is None:
    filename_mesh = get_gen_block_mesh_hook((1., 1.), (20, 20), (.5, .5))

t0 = 0.

angle = 0
# get_common(approx_order, cfl, t0, t1, None, get_ic)
rotm = nm.array([[nm.cos(angle), -nm.sin(angle)],
                  [nm.sin(angle), nm.cos(angle)]])
velo = nm.sum(rotm.T * nm.array([1., 0.]), axis=-1)[: , None]
materials = {
    'a': ({'val': [velo], '.flux': adflux},),
}

regions = {
    'Omega'      : 'all',
    'left': ('vertices in x == 0', 'edge'),
    'right': ('vertices in x == 1', 'edge'),
    'top': ('vertices in y == 1', 'edge'),
    'bottom': ('vertices in y == 0', 'edge')
}

fields = {
    'f': ('real', 'scalar', 'Omega', str(approx_order) + 'd', 'DG', 'legendre') #
}

variables = {
    'p': ('unknown field', 'f', 0, 1),
    'v': ('test field', 'f', 'p'),
}

def gsmooth(x):
    r"""
    .. :math: C_0^{\infty}
    """
    return .3 * nm.piecewise(x, [x <= 0.1, x >= 0.1, .3 < x],
                             [0, lambda x:
                              nm.exp(1 / ((10 * (x - .2)) ** 2 - 1) + 1),
                              0])

def analytic_sol(coors, t):
    x_1 = coors[..., 0]
    x_2 = coors[..., 1]
    sin = nm.sin
    pi = nm.pi
    exp = nm.exp
    # res = four_step_u(x_1) * four_step_u(x_2)
    res = gsmooth(x_1) * gsmooth(x_2)
    return res

@local_register_function

```

(continues on next page)

(continued from previous page)

```

def sol_fun(ts, coors, mode="qp", **kwargs):
    t = ts.time
    if mode == "qp":
        return {"p": analytic_sol(coors, t)[..., None, None]}

def get_ic(x, ic=None):
    return gsmooth(x[..., 0:1]) * gsmooth(x[..., 1:])

functions = {
    'get_ic': (get_ic,)
}

ics = {
    'ic': ('Omega', {'p.0': 'get_ic'}),
}

dgepb1 = {
    'name': 'u_r1',
    'region': ['right', 'left'],
    'dofs': {'p.all': 'p.all'},
    'match': 'match_y_line',
}

integrals = {
    'i': 3 * approx_order,
}

equations = {
    'Advection': """
        dw_dot.i.Omega(v, p)
        - dw_s_dot_mgrad_s.i.Omega(a.val, p[-1], v)
        + dw_dg_advect_laxfrie_flux.i.Omega(a.flux, a.val, v, p[-1]) = 0
    """
}

solvers = {
    "tss": ('ts.tvd_runge_kutta_3',
           {"t0" : t0,
            "t1" : t1,
            'limiters': {"f": MomentLimiter2D} if limit else {})),
    'nls': ('nls.newton', {}),
    'ls' : ('ls.scipy_direct', {})
}

options = {
    'ts' : 'tss',
    'nls' : 'newton',
    'ls' : 'ls',
    'save_times' : 100,
    'active_only' : False,
    'output_dir' : 'output/dg/' + example_name,

```

(continues on next page)

(continued from previous page)

```

    'output_format' : 'msh',
    'file_format'   : 'gmsh-dg',
    'pre_process_hook': get_cfl_setup(cfl) if dt is None else get_cfl_setup(dt=dt)
}

return locals()

```

dg/advection_diffusion_2D.py

Description

Static advection-diffusion equation in 2D solved by discontinuous Galerkin method.

$$a \cdot \text{grad } p - \text{div}(\text{grad } p) = 0$$

Based on

Antonietti, P., & Quarteroni, A. (2013). Numerical performance of discontinuous and stabilized continuous Galerkin methods for convection-diffusion problems.

Usage Examples

Run:

```
sfePy-run sfePy/examples/dg/advection_diffusion_2D.py
```

Results are saved to output/dg/advection_diffusion_2D folder by default as `.msh` files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use **File | Open** menu or **Ctrl + O** shortcut, navigate to the output folder, select all `.msh` files and hit **Open**, all files should load as one item in Post-processing named `p_cell_nodes`.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on **All view options...** This brings up the Options window with **View [0]** selected in left column. Under the tab **General** ensure that **Adapt visualization grid** is ticked, then you can adjust **Maximum recursion depth** and **Target visualization error** to tune the visualization. To see visualization elements (as opposed to mesh elements) go to **Visibility** tab and tick **Draw element outlines**, this option is also available from quick options menu as **View element outlines** or under shortcut **Alt+E**. In the quick options menu, you can also modify normal raise by clicking **View Normal Raise** to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

source code

```

r"""
Static advection-diffusion equation in 2D solved by discontinuous Galerkin method.

.. math:: a \cdot \text{grad}\, , p - \text{div}(\text{grad}\, , p) = 0

Based on

Antonietti, P., & Quarteroni, A. (2013). Numerical performance of discontinuous

```

(continues on next page)

(continued from previous page)

and stabilized continuous Galerkin methods for convection-diffusion problems.

Usage Examples

Run::

```
sfePy-run sfePy/examples/dg/advection_diffusion_2D.py
```

Results are saved to output/dg/advection_diffusion_2D folder by default as ``.msh`` files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use `File | Open` menu or `Ctrl + O` shortcut, navigate to the output folder, select all ``.msh`` files and hit Open, all files should load as one item in Post-processing named `p_cell_nodes`.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on `All view options...`. This brings up the Options window with `View [0]` selected in left column. Under the tab `General` ensure that `Adapt visualization grid` is ticked, then you can adjust `Maximum recursion depth` and `Target visualization error` to tune the visualization. To see visualization elements (as opposed to mesh elements) go to `Visibility` tab and tick `Draw element outlines`, this option is also available from quick options menu as `View element outlines` or under shortcut `Alt+E`. In the quick options menu, you can also modify normal raise by clicking `View Normal Raise` to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise `-1` produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

"""

```
from sfePy.examples.dg.example_dg_common import *
```

```
def define(filename_mesh=None,
            approx_order=3,

            adflux=0,
            limit=False,

            cw=1000,
            diffcoef=1,
            diffscheme="symmetric",

            cfl=None,
            dt=None,
            ):

    cfl = None
    dt = None
```

(continues on next page)

(continued from previous page)

```

functions = {}
def local_register_function(fun):
    try:
        functions.update({fun.__name__: (fun,)})

    except AttributeError: # Already a sfepy Function.
        fun = fun.function
        functions.update({fun.__name__: (fun,)})

    return fun

example_name = "advection_diffusion_2D"
dim = 2

if filename_mesh is None:
    filename_mesh = get_gen_block_mesh_hook((1., 1.), (20, 20), (.5, .5))

velo = [1., 1.]

angle = 0.0 # - nm.pi / 5
rotm = nm.array([[nm.cos(angle), -nm.sin(angle)],
                 [nm.sin(angle), nm.cos(angle)]])
velo = nm.sum(rotm.T * nm.array(velo), axis=-1)[: , None]

regions = {
    'Omega' : 'all',
    'left' : ('vertices in x == 0', 'edge'),
    'right': ('vertices in x == 1', 'edge'),
    'top' : ('vertices in y == 1', 'edge'),
    'bottom': ('vertices in y == 0', 'edge')
}

fields = {
    'f': ('real', 'scalar', 'Omega', str(approx_order) + 'd', 'DG', 'legendre')
}

variables = {
    'p': ('unknown field', 'f', 0),
    'v': ('test field', 'f', 'p'),
}

integrals = {
    'i': 2 * approx_order,
}

@local_register_function
def bc_funs(ts, coors, bc, problem):
    # return 2*coors[..., 1]
    t = ts.dt*ts.step
    x_1 = coors[..., 0]
    x_2 = coors[..., 1]

```

(continues on next page)

(continued from previous page)

```

res = nm.zeros(nm.shape(x_1))

sin = nm.sin
cos = nm.cos
exp = nm.exp
pi = nm.pi

if bc.diff == 0:
    if "left" in bc.name:
        res[:] = 0
    elif "right" in bc.name:
        res[:] = 0
    elif "bottom" in bc.name:
        res[:] = 0 #-2*sin(2*pi*x_1)
    elif "top" in bc.name:
        res[:] = 0

elif bc.diff == 1:
    if "left" in bc.name:
        res = nm.stack((-2*pi*(x_2**2 - x_2),
                        res),
                        axis=-2)
    elif "right" in bc.name:
        res = nm.stack((-2*pi*(x_2**2 - x_2), res,),
                        axis=-2)
    elif "bot" in bc.name:
        res = nm.stack((res,
                        sin(2*pi*x_1)),
                        axis=-2)
    elif "top" in bc.name:
        res = nm.stack((res,
                        -sin(2*pi*x_1)),
                        axis=-2)

return res

@local_register_function
def source_fun(ts, coors, mode="qp", **kwargs):
    # t = ts.dt * ts.step
    eps = diffcoef
    sin = nm.sin
    cos = nm.cos
    exp = nm.exp
    sqrt = nm.sqrt
    pi = nm.pi
    if mode == "qp":
        x_1 = coors[..., 0]
        x_2 = coors[..., 1]
        res = -2*pi*(x_2**2 - x_2)*cos(2*pi*x_1)\
            - 2*(2*pi**2*(x_2**2 - x_2)*sin(2*pi*x_1) - sin(2*pi*x_1))*eps\
            - (2*x_2 - 1)*sin(2*pi*x_1)

```

(continues on next page)

(continued from previous page)

```

        return {"val": res[...], None, None]}

def analytic_sol(coors, t):
    x_1 = coors[..., 0]
    x_2 = coors[..., 1]
    sin = nm.sin
    pi = nm.pi
    res = -(x_2 ** 2 - x_2) * sin(2 * pi * x_1)
    return res

@local_register_function
def sol_fun(ts, coors, mode="qp", **kwargs):
    t = ts.time
    if mode == "qp":
        return {"p": analytic_sol(coors, t)[...], None, None]}

dgebcs = {
    'u_left' : ('left', {'p.all': "bc_funs", 'grad.p.all' : "bc_funs"}),
    'u_top'   : ('top', {'p.all': "bc_funs", 'grad.p.all' : "bc_funs"}),
    'u_bot'   : ('bottom', {'p.all': "bc_funs", 'grad.p.all' : "bc_funs"}),
    'u_right' : ('right', {'p.all': "bc_funs", 'grad.p.all' : "bc_funs"}),
}

materials = {
    'a' : ({'val': [velo], '.flux': adflux},),
    'D' : ({'val': [diffcoef], '.cw': cw},),
    'g' : 'source_fun'
}

equations = {
    'balance': """
        - dw_s_dot_mgrad_s.i.Omega(a.val, p, v)
        + dw_dg_advect_laxfrie_flux.i.Omega(a.flux, a.val, v, p)
        """
        +
        " + dw_laplace.i.Omega(D.val, v, p) " +
        diffusion_schemes_implicit[diffscheme] +
        " + dw_dg_interior_penalty.i.Omega(D.val, D.cw, v, p)" +
        " - dw_volume_lvf.i.Omega(g.val, v)" +
        "= 0"
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',

```

(continues on next page)

(continued from previous page)

```

    'kind' : 'nls.newton',

    'i_max'      : 5,
    'eps_a'      : 1e-8,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 0.99999,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
}

options = {
    'nls'          : 'newton',
    'ls'           : 'ls',
    'output_dir'   : 'output/dg/' + example_name,
    'output_format' : 'msh',
    'file_format'  : 'gmsh-dg'
}
return locals()

```

dg/burgers_2D.py

Description

Burgers equation in 2D solved using discontinuous Galerkin method

$$\frac{dp}{dt} + \operatorname{div} f(p) - \operatorname{div}(\operatorname{grad} p) = 0$$

Based on

Kučera, V. (n.d.). Higher order methods for the solution of compressible flows. Charles University. p. 21 eq. (1.39)

Usage Examples

Run:

```
sfepy-run sfepy/examples/dg/burgers_2D.py
```

Results are saved to output/dg/burgers_2D folder by default as .msh files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use File | Open menu or Ctrl + O shortcut, navigate to the output folder, select all .msh files and hit Open, all files should load as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on All view options.... This brings up the Options window with View [0] selected in left column. Under the tab General ensure that Adapt visualization grid is ticked, then you can adjust Maximum recursion depth and Target visualization error to tune the visualization. To see visualization elements (as opposed to mesh elements) go to Visibility tab and tick Draw element outlines, this option is also available from quick options menu as View element outlines or under

shortcut Alt+E. In the quick options menu, you can also modify normal raise by clicking View Normal Raise to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

source code

```
r"""
Burgers equation in 2D solved using discontinuous Galerkin method

.. math:: \frac{dp}{dt} + \text{div}\backslash,f(p) - \text{div}(\text{grad}\backslash,p) = 0

Based on

Kučera, V. (n.d.). Higher order methods for the solution of compressible flows.
Charles University. p. 21 eq. (1.39)

Usage Examples
-----

Run::

    sfepy-run sfepy/examples/dg/burgers_2D.py

Results are saved to output/dg/burgers_2D folder by default as ``.msh`` files,
the best way to view them is through GMSH (http://gmsh.info/) version 4.6 or
newer. Start GMSH and use ``File | Open`` menu or Ctrl + O shortcut, navigate to
the output folder, select all ``.msh`` files and hit Open, all files should load
as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements,
to modify fidelity of rendering, double click the displayed mesh, quick options
menu should pop up, click on ``All view options...``. This brings up the Options
window with ``View [0]`` selected in left column. Under the tab ``General``
ensure that ``Adapt visualization grid`` is ticked, then you can adjust
``Maximum recursion depth`` and ``Target visualization error`` to tune
the visualization. To see visualization elements (as opposed to mesh elements)
go to ``Visibility`` tab and tick ``Draw element outlines``, this option is also
available from quick options menu as ``View element outlines`` or under shortcut
``Alt+E``. In the quick options menu, you can also modify normal raise by
clicking ``View Normal Raise`` to see solution rendered as surface above the
mesh. Note that for triangular meshes normal raise -1 produces expected raise
above the mesh. This is due to the opposite orientation of the reference
elements in GMSH and SfePy and might get patched in the future.
"""

from sfepy.examples.dg.example_dg_common import *
from sfepy import data_dir

from sfepy.discrete.dg.limiters import MomentLimiter2D, IdentityLimiter

mesh_center = (0, 0)
```

(continues on next page)

(continued from previous page)

```

mesh_size = (2, 2)

def define(filename_mesh=None,
            approx_order=2,

            adflux=0,
            limit=False,

            cw=10,
            diffcoef=0.002,
            diffscheme="symmetric",

            cfl=None,
            dt=1e-5,
            t1=0.01
            ):

    functions = {}
    def local_register_function(fun):
        try:
            functions.update({fun.__name__: (fun,)})

        except AttributeError: # Already a sfepy Function.
            fun = fun.function
            functions.update({fun.__name__: (fun,)})

        return fun

    example_name = "burgers_2D"
    dim = 2

    if filename_mesh is None:
        filename_mesh = data_dir + "/meshes/2d/square_tri2.mesh"

    t0 = 0.
    if dt is None and cfl is None:
        dt = 1e-5

    velo = [1., 1.]

    angle = 0 # - nm.pi / 5
    rotm = nm.array([[nm.cos(angle), -nm.sin(angle)],
                     [nm.sin(angle), nm.cos(angle)]])
    velo = nm.sum(rotm.T * nm.array(velo), axis=-1)[: , None]
    burg_velo = velo.T / nm.linalg.norm(velo)

    regions = {
        'Omega': 'all',
        'left' : ('vertices in x == -1', 'edge'),
        'right': ('vertices in x == 1', 'edge'),
        'top'  : ('vertices in y == 1', 'edge'),

```

(continues on next page)

(continued from previous page)

```

    'bottom': ('vertices in y == -1', 'edge')
}

fields = {
    'f': ('real', 'scalar', 'Omega',
          str(approx_order) + 'd', 'DG', 'legendre')
}

variables = {
    'p': ('unknown field', 'f', 0, 1),
    'v': ('test field', 'f', 'p'),
}

integrals = {
    'i': 5,
}

def analytic_sol(coors, t):
    x_1 = coors[..., 0]
    x_2 = coors[..., 1]
    sin = nm.sin
    pi = nm.pi
    exp = nm.exp
    res = -(exp(-t) - 1)*(sin(5*x_1*x_2) + sin(-4*x_1*x_2 + 4*x_1 + 4*x_2))
    return res

@local_register_function
def sol_fun(ts, coors, mode="qp", **kwargs):
    t = ts.time
    if mode == "qp":
        return {"p": analytic_sol(coors, t)[..., None, None]}

@local_register_function
def bc_funs(ts, coors, bc, problem):
    # return 2*coors[..., 1]
    t = ts.dt*ts.step
    x_1 = coors[..., 0]
    x_2 = coors[..., 1]
    sin = nm.sin
    cos = nm.cos
    exp = nm.exp
    if bc.diff == 0:
        if "left" in bc.name:
            res = -(exp(-t) - 1)*(sin(-5*x_2) + sin(8*x_2 - 4))
        elif "bottom" in bc.name:
            res = -(exp(-t) - 1) * (sin(-5 * x_1) + sin(8 * x_1 - 4))
        elif "right" in bc.name:
            res = -(exp(-t) - 1)*(sin(4) + sin(5*x_2))
        elif "top" in bc.name:
            res = -(exp(-t) - 1)*(sin(4) + sin(5*x_1))

    elif bc.diff == 1:

```

(continues on next page)

(continued from previous page)

```

if "left" in bc.name:
    res = nm.stack(((4*(x_2 - 1)*cos(4) - 5*x_2*cos(5*x_2))*
                    (exp(-t) - 1),
                    -5*(exp(-t) - 1)*cos(5*x_2)),
                    axis=-2)
elif "bottom" in bc.name:
    res = nm.stack(((5*cos(-5*x_1) - 8*cos(8*x_1 - 4))*(exp(-t) - 1),
                    -(5*x_1*cos(-5*x_1) - 4*(x_1 - 1)*cos(8*x_1 - 4))*
                    (exp(-t) - 1)),
                    axis=-2)

elif "right" in bc.name:
    res = nm.stack(((4*(x_2 - 1)*cos(4) - 5*x_2*cos(5*x_2))*
                    (exp(-t) - 1),
                    -5*(exp(-t) - 1)*cos(5*x_2)),
                    axis=-2)
elif "top" in bc.name:
    res = nm.stack((-5*(exp(-t) - 1)*cos(5*x_1),
                    (4*(x_1 - 1)*cos(4) - 5*x_1*cos(5*x_1))*
                    (exp(-t) - 1)),
                    axis=-2)

return res

@local_register_function
def source_fun(ts, coors, mode="qp", **kwargs):
    if mode == "qp":
        t = ts.dt * ts.step
        x_1 = coors[..., 0]
        x_2 = coors[..., 1]
        sin = nm.sin
        cos = nm.cos
        exp = nm.exp
        res = (
            + (5 * x_1 * cos(5 * x_1 * x_2)
              - 4 * (x_1 - 1) * cos(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2)) *
              (exp(-t) - 1) ** 2 * (sin(5 * x_1 * x_2)
                                   - sin(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2))
            + (5 * x_2 * cos(5 * x_1 * x_2)
              - 4 * (x_2 - 1) * cos(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2)) *
              (exp(-t) - 1) ** 2 * (sin(5 * x_1 * x_2)
                                   - sin(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2))
            - diffcoef *
              ((25 * x_1 ** 2 * sin(5 * x_1 * x_2) - 16 * (x_1 - 1) ** 2 *
                sin(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2)) * (exp(-t) - 1)
              + (25 * x_2 ** 2 * sin(5 * x_1 * x_2) - 16 * (x_2 - 1) ** 2 *
                sin(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2)) * (exp(-t) - 1))
            + (sin(5 * x_1 * x_2) - sin(4 * x_1 * x_2 - 4 * x_1 - 4 * x_2)) *
              exp(-t)
        )
    return {"val": res[..., None, None]}

```

(continues on next page)

(continued from previous page)

```

def adv_fun(p):
    vu = velo.T * p[..., None]
    return vu

def adv_fun_d(p):
    v1 = velo.T * nm.ones(p.shape + (1,))
    return v1

def burg_fun(p):
    vu = .5*burg_velo * p[..., None] ** 2
    return vu

def burg_fun_d(p):
    v1 = burg_velo * p[..., None]
    return v1

materials = {
    'a' : ({'val': [velo], '.flux':adflux},),
    'D' : ({'val': [diffcoef], '.Cw': cw},),
    'g' : 'source_fun'
}

ics = {
    'ic': ('Omega', {'p.0': 0}),
}

dgebcs = {
    'u_left' : ('left', {'p.all': 'bc_funs', 'grad.p.all': 'bc_funs'}),
    'u_right' : ('right', {'p.all': 'bc_funs', 'grad.p.all': 'bc_funs'}),
    'u_bottom' : ('bottom', {'p.all': 'bc_funs', 'grad.p.all': 'bc_funs'}),
    'u_top' : ('top', {'p.all': 'bc_funs', 'grad.p.all': 'bc_funs'}),
}

equations = {
    'balance':
        "dw_dot.i.Omega(v, p)" +
        # non-linear hyperbolic terms
        " - dw_ns_dot_grad_s.i.Omega(burg_fun, burg_fun_d, p[-1], v)" +
        " + dw_dg_nonlinear_laxfrie_flux.i.Omega(a.flux, burg_fun, burg_fun_d, v, p[-1])
↪ " +
        # diffusion
        " + dw_laplace.i.Omega(D.val, v, p[-1])" +
        diffusion_schemes_explicit[diffscheme] +
        " - dw_dg_interior_penalty.i.Omega(D.val, D.Cw, v, p[-1])"
        # source
        + " - dw_volume_lvf.i.Omega(g.val, v)"
        " = 0"
}

solvers = {
    "tss.tvd_runge_kutta_3": ('ts.tvd_runge_kutta_3',

```

(continues on next page)

(continued from previous page)

```

        {"t0": t0,
         "t1": t1,
         'limiters': {
             "f": MomentLimiter2D} if limit else {})),
    "tss.euler": ('ts.euler',
                 {"t0"      : t0,
                  "t1"      : t1,
                  'limiters': {"f": MomentLimiter2D} if limit else {})),
    'nls': ('nls.newton', {}),
    'ls' : ('ls.scipy_direct', {})
}

options = {
    'ts'          : 'tss.euler',
    'nls'          : 'nls.newton',
    'ls'           : 'ls.mumps',
    'save_times'   : 100,
    'output_dir'   : 'output/dg/' + example_name,
    'output_format': 'msh',
    'file_format'  : 'gmsh-dg',
    'pre_process_hook': get_cfl_setup(CFL=cfl, dt=dt)
}

return locals()

```

dg/dg_plot_1D.py

Description

Script for plotting 1D DG FEM data stored in VTK files

source code

```

#!/usr/bin/env python
# -*- coding:utf-8 -*-
"""
Script for plotting 1D DG FEM data stored in VTK files
"""

import glob

import matplotlib
matplotlib.use('Qt5Agg')
import matplotlib.pyplot as plt
import argparse

from os.path import join as pjoin
import os
import sys
from glob import glob

from sfepy.discrete.dg.dg_1D_vizualizer import \

```

(continues on next page)

(continued from previous page)

```

(load_1D_vtk, animate_1D_DG_sol, load_state_1D_vtk, plot1D_legendre_dofs,
reconstruct_legendre_dofs)

def load_and_plot_fun(folder, filename, t0, t1, tn,
                      ic_fun=None, exact=None,
                      compare=False, polar=False):
    """
    Parameters
    -----
    folder : str
        folder where to look for files
    filename : str
        used in {name}.i.vtk, i = 0,1, ... tns - 1
    t0 : float
        starting time
    t1 : int
        final time
    tn : int
        number of time steps
    ic_fun : callable
        initital condition
    exact : callable
        exact solution, for transient problems function of space ant time
    compare : bool
    polar : bool
    """

    # load time data
    lmesh, u = load_1D_vtk(folder, filename)
    animate_1D_DG_sol(lmesh, t0, t1, u, tn=tn, ic=ic_fun, exact=exact,
                      delay=100, polar=polar)

    if compare:
        files = glob(pjoin(folder, "*.vtk"))

        n_first = sorted(files)[0].split(".")[2]
        n_last = sorted(files)[-1].split(".")[2]

        print("Plotting files {} and {} to compare first and last".format(n_first, n_
↪last))

        coors, u_end = load_state_1D_vtk(pjoin(folder, filename + "." + n_last + ".vtk"))
        coors, u_start = load_state_1D_vtk(pjoin(folder, filename + "." + n_first + ".vtk"
↪))

        plot1D_legendre_dofs(coors, [u_start.swapaxes(0, 1)[: , :, 0], u_end.swapaxes(0,
↪1)[: , :, 0]])

        plt.figure("Reconstructed solution")
        ww_s, xx = reconstruct_legendre_dofs(coors, None, u_end)
        ww_e, _ = reconstruct_legendre_dofs(coors, None, u_start)

```

(continues on next page)

(continued from previous page)

```

plt.plot(xx, ww_s[:, 0], label="IC")
plt.plot(xx, ww_e[:, 0], label="Last")
plt.legend()
plt.show()

def main(argv):
    parser = argparse.ArgumentParser(description='Plotting of 1D DG data in VTK files',
                                     epilog='(c) 2019 T. Zitka , Man-machine'
                                     + ' Interaction at NTC UWB')
    parser.add_argument("input_name", help="Folder or name of the example in "
                                     + "output folder with VTK data, file names "
                                     + "in format <name>.[0-9]*.vtk. If not "
                                     + "provided asks for the name of the example"
                                     , nargs="?")

    parser.add_argument("-s", "--search", help="Search for different orders in "
                                     + "provided folder",
                        action="store_true", default=False)
    parser.add_argument("-t0", "--start_time", type=float, default=0,
                        help="Start time of the simulation")
    parser.add_argument("-t1", "--end_time", type=float, default=1.,
                        help="End time of the simulation")
    parser.add_argument("-o", "--order", type=int, default=None,
                        help="Order of the approximation, when example folder"
                        + " contains more orders this chooses the one")
    parser.add_argument("-cf", "--compare_final", action="store_true",
                        help="To compare starting and final time - "
                        + "usefull for periodic boundary problems")
    parser.add_argument("-p", "--polar", help="Plot in polar projection",
                        action="store_true")

    if argv is None:
        argv = sys.argv[1:]
    args = parser.parse_args(argv)

    t0 = args.start_time
    t1 = args.end_time
    cf = args.compare_final
    pol = args.polar
    if args.input_name is None:
        input_name = str(input("Please provide name of the example in output/" +
                               " folder or full path to data: "))
    else:
        input_name = args.input_name

    if os.path.isdir(input_name):
        full_infolder_path = os.path.abspath(input_name)
    else:
        input_name = pjoin("output", input_name)
        if os.path.isdir(input_name):
            full_infolder_path = os.path.abspath(input_name)

```

(continues on next page)

(continued from previous page)

```

    else:
        print("Example {} not found in {}".format(input_name,
                                                    os.path.abspath("output")))

        return

print("Input folder found: {}".format(full_infolder_path))
if args.search:
    print("Input folder contains results for orders {}".format([os.path.basename(fol)
                                                                for fol in
                                                                glob(pjoin(full_infolder_path, "[0-9]*"))]))

    if args.order is None:
        order = str(input("Please provide order of approximation, default is 1: "))
        if len(order) == 0:
            order = 1
        else:
            try:
                order = int(order)
            except ValueError:
                print("Value {} for order not understood!".format(order))
                return
    else:
        order = args.order

    print("Looking for results of order {}".format(order))
    full_infolder_path = pjoin(full_infolder_path, str(order))
    if not os.path.isdir(full_infolder_path):
        print("Input folder with order {} not found".format(full_infolder_path))
        return
else:
    order = args.order

contents = glob(pjoin(full_infolder_path, "*.vtk"))
print()
tn = len(contents) # we assume the contents are time step data files
if tn == 0:
    print("Input folder {} is empty!".format(full_infolder_path))
    return
base_name = os.path.basename(contents[0]).split(".")[0]
print("Found {} files, basename is {}".format(tn, base_name))
print("Plotting ...")
load_and_plot_fun(full_infolder_path, base_name, t0, t1, tn,
                  compare=cf, polar=pol)

if __name__ == '__main__':
    main(sys.argv[1:])

```

dg/example_dg_common.py**Description**

Functions common to DG examples

source code

```

"""
Functions common to DG examples
"""
import os
from glob import glob

import numpy as nm

from sfepy.base.base import output
from sfepy.discrete.fem import Mesh
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh

diffusion_schemes_implicit = {
    "symmetric":
        " + dw_dg_diffusion_flux.i.Omega(D.val, p, v)"
        + " + dw_dg_diffusion_flux.i.Omega(D.val, v, p)",
    "non-symmetric":
        " + dw_dg_diffusion_flux.i.Omega(D.val, p, v)"
        + " - dw_dg_diffusion_flux.i.Omega(D.val, v, p)",
    "incomplete":
        " + dw_dg_diffusion_flux.i.Omega(D.val, p, v)}

diffusion_schemes_explicit = {
    "symmetric":
        " - dw_dg_diffusion_flux.i.Omega(D.val, p[-1], v)"
        + " - dw_dg_diffusion_flux.i.Omega(D.val, v, p[-1])",
    "non-symmetric":
        " - dw_dg_diffusion_flux.i.Omega(D.val, p[-1], v)"
        + " + dw_dg_diffusion_flux.i.Omega(D.val, v, p[-1])",
    "incomplete":
        " - dw_dg_diffusion_flux.i.Omega(D.val, p[-1], v)}

functions = {}
def local_register_function(fun):
    try:
        functions.update({fun.__name__: (fun,)})

    except AttributeError: # Already a sfepy Function.
        fun = fun.function
        functions.update({fun.__name__: (fun,)})

    return fun

```

(continues on next page)

(continued from previous page)

```

def get_cfl_setup(CFL=None, dt=None):
    """
    Provide either CFL or dt to create preprocess hook that sets up
    Courant-Friedrichs-Levi stability condition for either advection or
    diffusion.

    Parameters
    -----
    CFL : float, optional
    dt: float, optional

    Returns
    -----
    setup_cfl_condition : callable
        expects sfepy.discrete.problem as argument

    """

    if CFL is None and dt is None:
        raise ValueError("Specify either CFL or dt in CFL setup")

    def setup_cfl_condition(problem):
        """
        Sets up CFL condition for problem ts_conf in problem

        Parameters
        -----
        problem : discrete.problem.Problem
        """
        ts_conf = problem.ts_conf
        mesh = problem.domain.mesh
        dim = mesh.dim
        first_field = list(problem.fields.values())[0]
        first_field_name = list(problem.fields.keys())[0]
        approx_order = first_field.approx_order
        mats = problem.create_materials(['a', 'D'])
        try:
            # make this more general?
            # maybe require material name in parameter
            velo = problem.conf_materials['material_a__0'].values["val"]
            max_velo = nm.max(nm.linalg.norm(velo))
        except KeyError:
            max_velo = 1

        try:
            # make this more general?
            # maybe require material name in parameter
            diffusion = problem.conf_materials['material_D__0'].values["val"]
            max_diffusion = nm.max(nm.linalg.norm(diffusion))
        except KeyError:
            max_diffusion = None

```

(continues on next page)

(continued from previous page)

```

dx = nm.min(problem.domain.mesh.cmesh.get_volumes(dim))

output("Preprocess hook - setup_cfl_condition:...")
output("Approximation order of field {}({}) is {}".format(
    first_field_name, first_field.family_name, approx_order))
output("Space divided into {} cells, {} steps, step size {}".format(
    mesh.n_el, len(mesh.coors), dx))

if dt is None:
    adv_dt = get_cfl_advection(max_velo, dx, approx_order, CFL)
    diff_dt = get_cfl_diffusion(max_diffusion, dx, approx_order, CFL)
    _dt = min(adv_dt, diff_dt)
else:
    output("CFL coefficient {} ignored, dt specified directly".format(CFL))
    _dt = dt

tn = int(nm.ceil((ts_conf.t1 - ts_conf.t0) / _dt))
dtdx = _dt / dx

ts_conf.dt = _dt
ts_conf.n_step = tn
ts_conf.cour = max_velo * dtdx

output("Time divided into {} nodes, {} steps, step size is {}".format(
    tn - 1, tn, _dt))
output("Courant number c = max(norm(a)) * dt/dx = {}".format(
    ts_conf.cour))
output("Time stepping solver is {}".format(ts_conf.kind))
output("... CFL setup done.")

return setup_cfl_condition

def get_cfl_advection(max_velo, dx, approx_order, CFL):
    """
    Parameters
    -----
    max_velo : float
    dx : float
    approx_order : int
    CFL : CFL

    Returns
    -----
    dt : float
    """
    order_corr = 1. / (2 * approx_order + 1)

    dt = dx / max_velo * CFL * order_corr

```

(continues on next page)

(continued from previous page)

```

    if not (nm.isfinite(dt)):
        dt = 1
    output(("CFL advection: CFL coefficient was {0} " +
           "and order correction 1/{1} = {2}")
           .format(CFL, (2 * approx_order + 1), order_corr))
    output("CFL advection: resulting dt={}".format((dt)))
    return dt

def get_cfl_diffusion(max_diffusion, dx, approx_order, CFL,
                      do_order_corr=False):
    """

    Parameters
    -----
    max_diffusion : float
    dx : float
    approx_order : int
    CFL : float
    do_order_corr : bool

    Returns
    -----
    dt : float
    """
    if max_diffusion is None:
        return 1

    if do_order_corr:
        order_corr = 1. / (2 * approx_order + 1)
    else:
        order_corr = 1

    dt = dx**2 / max_diffusion * CFL * order_corr

    if not (nm.isfinite(dt)):
        dt = 1
    output(("CFL diffusion: CFL coefficient was {0} " +
           "and order correction 1/{1} = {2}")
           .format(CFL, (2 * approx_order + 1), order_corr))
    output("CFL diffusion: resulting dt={}".format(dt))
    return dt

def get_gen_1D_mesh_hook(XS, XE, n_nod):
    """

    Parameters
    -----
    XS : float

```

(continues on next page)

(continued from previous page)

```

    leftmost coordinate
    XE : float
    rightmost coordinate
    n_nod : int
        number of nodes, number of cells is then n_nod - 1

Returns
-----
mio : UserMeshIO instance
"""
def mesh_hook(mesh, mode):
    """
    Generate the 1D mesh.
    """
    if mode == 'read':

        coors = nm.linspace(XS, XE, n_nod).reshape((n_nod, 1))
        conn = nm.arange(n_nod, dtype=nm.int32) \
            .repeat(2)[1:-1].reshape((-1, 2))
        mat_ids = nm.zeros(n_nod - 1, dtype=nm.int32)
        descs = ['1_2']

        mesh = Mesh.from_data('uniform_1D{}'.format(n_nod), coors, None,
                              [conn], [mat_ids], descs)

        return mesh

    elif mode == 'write':
        pass

mio = UserMeshIO(mesh_hook)
return mio

def get_gen_block_mesh_hook(dims, shape, centre, mat_id=0, name='block',
                             coors=None, verbose=True):
    """

Parameters
-----
dims : array of 2 or 3 floats
    Dimensions of the block.
shape : array of 2 or 3 ints
    Shape (counts of nodes in x, y, z) of the block mesh.
centre : array of 2 or 3 floats
    Centre of the block.
mat_id : int, optional
    The material id of all elements.
name : string
    Mesh name.
verbose : bool
    If True, show progress of the mesh generation.

Returns

```

(continues on next page)

(continued from previous page)

```

-----
mio : UserMeshIO instance
"""
def mesh_hook(mesh, mode):
    """
    Generate the 1D mesh.
    """
    if mode == 'read':

        mesh = gen_block_mesh(dims, shape, centre, mat_id=mat_id, name=name,
                              coors=coors, verbose=verbose)

        return mesh

    elif mode == 'write':
        pass

mio = UserMeshIO(mesh_hook)
return mio

def clear_folder(clear_format, confirm=False, doit=True):
    """
    Deletes files matching the format

    Parameters
    -----
    clear_format : str
    confirm : bool
    doit : bool
        if False do not delete anything no matter the confirmation

    Returns
    -----
    deleted_anything :
        True if there was something to delete
    """
    files = glob(clear_format)
    if confirm:
        for file in files:
            output("Will delete file {}".format(file))
            doit = input("-----\nDelete files [Y/n]? ").strip() == "Y"

    if doit:
        for file in files:
            os.remove(file)
    return bool(files)

```

dg/imperative_burgers_1D.py**Description**

Burgers equation in 1D solved using discontinuous Galerkin method

source code

```
#!/usr/bin/env python
"""
Burgers equation in 1D solved using discontinuous Galerkin method
"""
import argparse
import sys
sys.path.append('.')
from os.path import join as pjoin

import numpy as nm

from sfepy.examples.dg.example_dg_common import \
    clear_folder, get_gen_1D_mesh_hook
from sfepy.examples.dg.dg_plot_1D import load_and_plot_fun

# sfepy imports
from sfepy.base.base import IndexedStruct
from sfepy.base.base import Struct, configure_output, output
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                             Equation, Equations, Problem)
from sfepy.discrete.conditions import InitialCondition, EssentialBC, Conditions
from sfepy.discrete.dg.fields import DGField
from sfepy.discrete.dg.limiters import MomentLimiter1D
from sfepy.discrete.fem import FEDomain
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.solvers.ts_dg_solvers import TVDRK3StepSolver
from sfepy.terms.terms_dg import Term

def parse_args(argv=None):
    if argv is None:
        argv = sys.argv

    parser = argparse.ArgumentParser(
        description='Solve Burgers equation and display animated results, '
        'change script code to modify the problem.',
        epilog='(c) 2019 T. Zitka , Man-machine Interaction at NTC UWB')
    parser.add_argument('-o', '--output-dir', default='.',
                        help='output directory')
    parser.add_argument('-p', '--plot',
                        action='store_true', dest='plot',
                        default=False, help='plot animated results')
    options = parser.parse_args(argv[1:])
    return options

def main(argv=None):
```

(continues on next page)

(continued from previous page)

```

options = parse_args(argv=argv)

# vvvvvvvvvvvvvvvvvvvvv #
approx_order = 2
# ^^^^^^^^^^^^^^^^^^^^^ #

# Setup output names
outputs_folder = options.output_dir

domain_name = "domain_1D"
problem_name = "iburgers_1D"
output_folder = pjoin(outputs_folder, problem_name, str(approx_order))
output_format = "vtk"
save_timestn = 100
clear_folder(pjoin(output_folder, "." + output_format))
configure_output({'output_screen': True,
                  'output_log_name':
                    pjoin(output_folder,
                          f"last_run_{problem_name}_{approx_order}.txt")})

# -----
# | Get mesh |
# -----
X1 = 0.
XN = 1.
n_nod = 100
n_el = n_nod - 1
mesh = get_gen_1D_mesh_hook(X1, XN, n_nod).read(None)

# -----
# | Create problem components |
# -----

integral = Integral('i', order=approx_order * 2)
domain = FEDomain(domain_name, mesh)
omega = domain.create_region('Omega', 'all')
left = domain.create_region('Gamma1',
                            'vertices in x == %.10f' % X1,
                            'vertex')
right = domain.create_region('Gamma2',
                             'vertices in x == %.10f' % XN,
                             'vertex')
field = DGField('dgfu', nm.float64, 'scalar', omega,
                approx_order=approx_order)

u = FieldVariable('u', 'unknown', field, history=1)
v = FieldVariable('v', 'test', field, primary_var_name='u')

MassT = Term.new('dw_dot(v, u)', integral, omega, u=u, v=v)

velo = nm.array(1.0)

```

(continues on next page)

(continued from previous page)

```

def adv_fun(u):
    vu = velo.T * u[..., None]
    return vu

def adv_fun_d(u):
    v1 = velo.T * nm.ones(u.shape + (1,))
    return v1

burg_velo = velo.T / nm.linalg.norm(velo)

def burg_fun(u):
    vu = burg_velo * u[..., None] ** 2
    return vu

def burg_fun_d(u):
    v1 = 2 * burg_velo * u[..., None]
    return v1

StiffT = Term.new('dw_ns_dot_grad_s(fun, fun_d, u[-1], v)',
                  integral, omega,
                  u=u, v=v,
                  fun=burg_fun, fun_d=burg_fun_d)

# alpha = Material('alpha', val=[.0])
# FluxT = AdvectDGFluxTerm("adv_1f_flux(a.val, v, u)", "a.val, v, u[-1]",
#                          integral, omega, u=u, v=v, a=a, alpha=alpha)

FluxT = Term.new('dw_dg_nonlinear_laxfrie_flux(fun, fun_d, v, u[-1])',
                  integral, omega,
                  u=u, v=v,
                  fun=burg_fun, fun_d=burg_fun_d)

eq = Equation('balance', MassT - StiffT + FluxT)
eqs = Equations([eq])

# -----
# | Create boundary conditions |
# -----
left_fix_u = EssentialBC('left_fix_u', left, {'u.all': 1.0})
right_fix_u = EssentialBC('right_fix_u', right, {'u.all': 0.0})

# -----
# | Create initial condition |
# -----
def ghump(x):

```

(continues on next page)

(continued from previous page)

```

"""
Nice gaussian.
"""

return nm.exp(-200 * x ** 2)

def ic_wrap(x, ic=None):
    return ghump(x - .3)

ic_fun = Function('ic_fun', ic_wrap)
ics = InitialCondition('ic', omega, {'u.0': ic_fun})

# -----
# | Create problem |
# -----
pb = Problem(problem_name,
              equations=eqs,
              conf=Struct(options={"save_times": save_timestn},
                           ics={}, ebc={}, epbs={}, lcbcs={},
                           materials={}),
              active_only=False)
pb.setup_output(output_dir=output_folder, output_format=output_format)
pb.set_ics(Conditions([ics]))

# -----
# | Create limiter |
# -----
limiter = MomentLimiter1D

# -----
# | Set time discretization |
# -----
CFL = .2
max_velo = nm.max(nm.abs(velo))
t0 = 0
t1 = .2
dx = nm.min(mesh.cmesh.get_volumes(1))
dt = dx / max_velo * CFL / (2 * approx_order + 1)
tn = int(nm.ceil((t1 - t0) / dt))
dtdx = dt / dx

# -----
# | Create solver |
# -----
ls = ScipyDirect({})
nls_status = IndexedStruct()
nls = Newton({'is_linear': True}, lin_solver=ls, status=nls_status)

tss_conf = {'t0'      : t0,
            't1'      : t1,
            'n_step'   : tn,
            'limiters': {"dgfu": limiter}}
```

(continues on next page)

(continued from previous page)

```

tss = TVDRK3StepSolver(tss_conf,
                        nls=nls, context=pb, verbose=True)

# -----
# | Solve |
# -----
pb.set_solver(tss)
state_end = pb.solve()

output("Solved equation \n\n\t\t u_t - div(f(u)) = 0\n")
output(f"With IC: {ic_fun.name}")
# output("and EBCs: {}".format(pb.ebcs.names))
# output("and EPBCs: {}".format(pb.epbcs.names))
output("-----")
output(f"Approximation order is {approx_order}")
output(f"Space divided into {mesh.n_el} cells, " +
       f"{len(mesh.coors)} steps, step size is {dx}")
output(f"Time divided into {tn - 1} nodes, {tn} steps, step size is {dt}")
output(f"CFL coefficient was {CFL} and " +
       f"order correction {1 / (2 * approx_order + 1)}")
output(f"Courant number c = max(abs(u)) * dt/dx = {max_velo * dtdx}")
output("-----")
output(f"Time stepping solver is {tss.name}")
output(f"Limiter used: {limiter.name}")
output("=====")

# -----
# | Plot 1D|
# -----
if options.plot:
    load_and_plot_fun(output_folder, domain_name,
                      t0, t1, min(tn, save_timestn),
                      ic_fun)

if __name__ == '__main__':
    main()

```

dg/laplace_2D.py

Description

Laplace equation solved in 2d by discontinuous Galerkin method

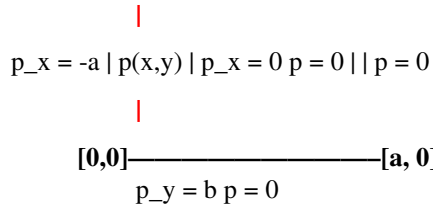
$$-\operatorname{div}(\operatorname{grad} p) = 0$$

on rectangle

$$p = 0 \quad p_y = 0$$

$$[0, b] \text{-----} [a, b]$$

|



solution to this is

$$p(x, y) = 1/2 * x ** 2 - 1/2 * y ** 2 - a * x + b * y$$

Usage Examples

Run:

```
sfePy-run sfePy/examples/dg/laplace_2D.py
```

Results are saved to output/dg/laplace_2D folder by default as .msh files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use File | Open menu or Ctrl + O shortcut, navigate to the output folder, select all .msh files and hit Open, all files should load as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on All view options.... This brings up the Options window with View [0] selected in left column. Under the tab General ensure that Adapt visualization grid is ticked, then you can adjust Maximum recursion depth and Target visualization error to tune the visualization. To see visualization elements (as opposed to mesh elements) go to Visibility tab and tick Draw element outlines, this option is also available from quick options menu as View element outlines or under shortcut Alt+E. In the quick options menu, you can also modify normal raise by clicking View Normal Raise to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

source code

```
r"""
Laplace equation solved in 2d by discontinous Galerkin method

.. math:: - \operatorname{div}(\operatorname{grad}\,p) = 0

on rectangle

      p = 0
      p_y = 0

      [0,b]-----[a, b]
      |             |
      |             |
p_x = -a |         p(x,y)         | p_x = 0
p = 0   |             |         | p = 0
      |             |             |
      |             |             |
      [0,0]-----[a, 0]
              p_y = b
              p = 0

solution to this is
.. math:: p(x,y) = 1/2*x**2 - 1/2*y**2 - a*x + b*y
```

(continues on next page)

(continued from previous page)

Usage Examples

Run::

```
sfepy-run sfepy/examples/dg/laplace_2D.py
```

Results are saved to output/dg/laplace_2D folder by default as ``.msh`` files, the best way to view them is through GMSH (<http://gmsh.info/>) version 4.6 or newer. Start GMSH and use ``File | Open`` menu or Ctrl + O shortcut, navigate to the output folder, select all ``.msh`` files and hit Open, all files should load as one item in Post-processing named p_cell_nodes.

GMSH is capable of rendering high order approximations in individual elements, to modify fidelity of rendering, double click the displayed mesh, quick options menu should pop up, click on ``All view options...``. This brings up the Options window with ``View [0]`` selected in left column. Under the tab ``General`` ensure that ``Adapt visualization grid`` is ticked, then you can adjust ``Maximum recursion depth`` and ``Target visualization error`` to tune the visualization. To see visualization elements (as opposed to mesh elements) go to ``Visibility`` tab and tick ``Draw element outlines``, this option is also available from quick options menu as ``View element outlines`` or under shortcut ``Alt+E``. In the quick options menu, you can also modify normal raise by clicking ``View Normal Raise`` to see solution rendered as surface above the mesh. Note that for triangular meshes normal raise -1 produces expected raise above the mesh. This is due to the opposite orientation of the reference elements in GMSH and SfePy and might get patched in the future.

```
from sfepy.examples.dg.example_dg_common import *

def define(filename_mesh=None,
            approx_order=2,

            adflux=None,
            limit=False,

            cw=100,
            diffcoef=1,
            diffscheme="symmetric",

            cfl=None,
            dt=None,
            ):

    cfl = None
    dt = None

    functions = {}
    def local_register_function(fun):
```

(continues on next page)

(continued from previous page)

```

    try:
        functions.update({fun.__name__: (fun,)})

    except AttributeError: # Already a sfepy Function.
        fun = fun.function
        functions.update({fun.__name__: (fun,)})

    return fun

example_name = "laplace_2D"
dim = 2

if filename_mesh is None:
    filename_mesh = get_gen_block_mesh_hook((1., 1.), (16, 16), (.5, .5))

a = 1
b = 1
c = 0

regions = {
    'Omega'      : 'all',
    'left'       : ('vertices in x == 0', 'edge'),
    'right'      : ('vertices in x == 1', 'edge'),
    'top'        : ('vertices in y == 1', 'edge'),
    'bottom'     : ('vertices in y == 0', 'edge')
}
fields = {
    'f': ('real', 'scalar', 'Omega', str(approx_order) + 'd', 'DG', 'legendre') #
}

variables = {
    'p': ('unknown field', 'f', 0, 1),
    'v': ('test field', 'f', 'p'),
}

def analytic_sol(coors, t):
    x_1, x_2 = coors[..., 0], coors[..., 1]
    res = 1/2*x_1**2 - 1/2*x_2**2 - a*x_1 + b*x_2 + c
    return res

@local_register_function
def sol_fun(ts, coors, mode="qp", **kwargs):
    t = ts.time
    if mode == "qp":
        return {"p": analytic_sol(coors, t)[..., None, None]}

@local_register_function
def bc_funs(ts, coors, bc, problem):
    t = ts.time
    x_1, x_2 = coors[..., 0], coors[..., 1]
    res = nm.zeros(x_1.shape)

```

(continues on next page)

(continued from previous page)

```

    if bc.diff == 0:
        res[:] = analytic_sol(coors, t)

    elif bc.diff == 1:
        res = nm.stack((x_1 - a, -x_2 + b),
                       axis=-2)

    return res

materials = {
    'D' : ({'val': [diffcoef], '.Cw': cw},),
}

dgebcs = {
    'u_left' : ('left', {'p.all': "bc_funs", 'grad.p.all': "bc_funs"}),
    'u_right' : ('right', {'p.all': "bc_funs", 'grad.p.all': "bc_funs"}),
    'u_bottom' : ('bottom', {'p.all': "bc_funs", 'grad.p.all': "bc_funs"}),
    'u_top' : ('top', {'p.all': "bc_funs", 'grad.p.all': "bc_funs"}),
}

integrals = {
    'i': 2 * approx_order,
}

equations = {
    'laplace': " dw_laplace.i.Omega(D.val, v, p) " +
                diffusion_schemes_implicit[diffscheme] +
                " + dw_dg_interior_penalty.i.Omega(D.val, D.Cw, v, p)" +
                " = 0"
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    # 'i_max'      : 5,
    # 'eps_a'      : 1e-8,
    # 'eps_r'      : 1.0,
    # 'macheps'    : 1e-16,
    # 'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    # 'ls_red'     : 0.1,
    # 'ls_red_warp' : 0.001,
    # 'ls_on'      : 0.99999,
    # 'ls_min'     : 1e-5,
    # 'check'      : 0,
    # 'delta'      : 1e-6,
}

```

(continues on next page)

(continued from previous page)

```
options = {
    'nls'          : 'newton',
    'ls'           : 'ls',
    'output_dir'   : 'output/dg/' + example_name,
    'output_format': 'msh',
    'file_format'  : 'gmsh-dg',
    # 'pre_process_hook': get_cfl_setup(cfl)
}
return locals()
```

diffusion

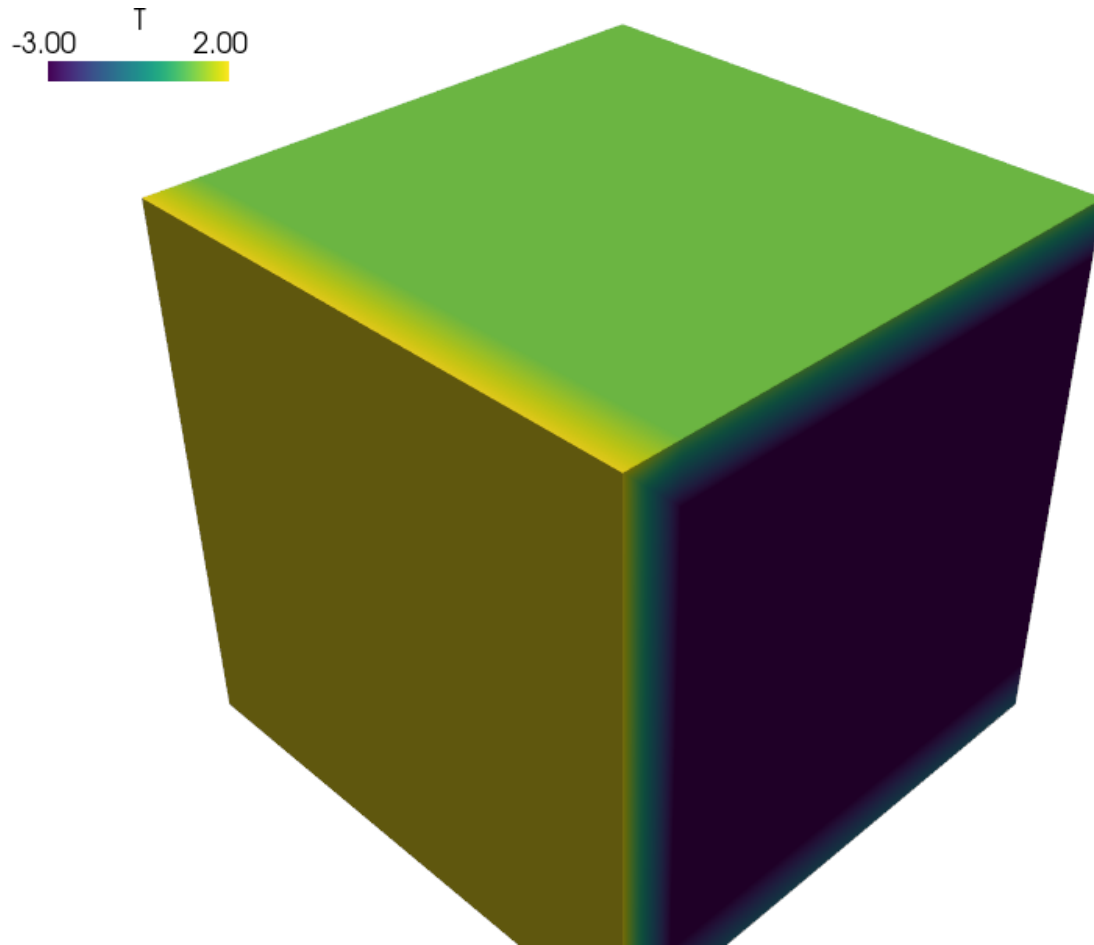
diffusion/cube.py

Description

Laplace equation (e.g. temperature distribution) on a cube geometry with different boundary condition values on the cube sides. This example was used to create the SfePy logo.

Find T such that:

$$\int_{\Omega} c \nabla s \cdot \nabla T = 0, \quad \forall s.$$



source code

```
r"""
Laplace equation (e.g. temperature distribution) on a cube geometry with
different boundary condition values on the cube sides. This example was
used to create the SfePy logo.

Find :math:T such that:

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla T
    = 0
    \;, \quad \forall s \;.
"""
from __future__ import absolute_import
from sfepy import data_dir

#filename_mesh = data_dir + '/meshes/3d/cube_big_tetra.mesh'
filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'

##### Laplace.

material_1 = {
```

(continues on next page)

(continued from previous page)

```

    'name' : 'coef',
    'values' : {'val' : 1.0},
}

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

if filename_mesh.find('cube_medium_hexa.mesh') >= 0:
    region_1000 = {
        'name' : 'Omega',
        'select' : 'cells of group 0',
    }
    integral_1 = {
        'name' : 'i',
        'order' : 1,
    }
    solver_0 = {
        'name' : 'ls',
        'kind' : 'ls.scipy_direct',
    }

elif filename_mesh.find('cube_big_tetra.mesh') >= 0:
    region_1000 = {
        'name' : 'Omega',
        'select' : 'cells of group 6',
    }
    integral_1 = {
        'name' : 'i',
        'quadrature' : 'custom',
        'vals' : [[1./3., 1./3., 1./3.]],
        'weights' : [0.5]
    }
    solver_0 = {
        'name' : 'ls',
        'kind' : 'ls.scipy_iterative',

        'method' : 'cg',
        'i_max' : 1000,
        'eps_r' : 1e-12,
    }

variable_1 = {
    'name' : 'T',
    'kind' : 'unknown field',
    'field' : 'temperature',
    'order' : 0, # order in the global vector of unknowns
}

```

(continues on next page)

(continued from previous page)

```

variable_2 = {
    'name' : 's',
    'kind' : 'test field',
    'field' : 'temperature',
    'dual' : 'T',
}

region_0 = {
    'name' : 'Surface',
    'select' : 'vertices of surface',
    'kind' : 'facet',
}
region_1 = {
    'name' : 'Bottom',
    'select' : 'vertices in (z < -0.4999999)',
    'kind' : 'facet',
}
region_2 = {
    'name' : 'Top',
    'select' : 'vertices in (z > 0.4999999)',
    'kind' : 'facet',
}
region_03 = {
    'name' : 'Left',
    'select' : 'vertices in (x < -0.4999999)',
    'kind' : 'facet',
}

ebc_1 = {
    'name' : 'T0',
    'region' : 'Surface',
    'dofs' : {'T.0' : -3.0},
}
ebc_4 = {
    'name' : 'T1',
    'region' : 'Top',
    'dofs' : {'T.0' : 1.0},
}
ebc_3 = {
    'name' : 'T2',
    'region' : 'Bottom',
    'dofs' : {'T.0' : -1.0},
}
ebc_2 = {
    'name' : 'T3',
    'region' : 'Left',
    'dofs' : {'T.0' : 2.0},
}

equations = {
    'nice_equation' : """dw_laplace.i.Omega( coef.val, s, T ) = 0""",

```

(continues on next page)

(continued from previous page)

```

}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 1,
    'eps_a'      : 1e-10,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 1.1,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
}

```

diffusion/darcy_flow_multicomp.py

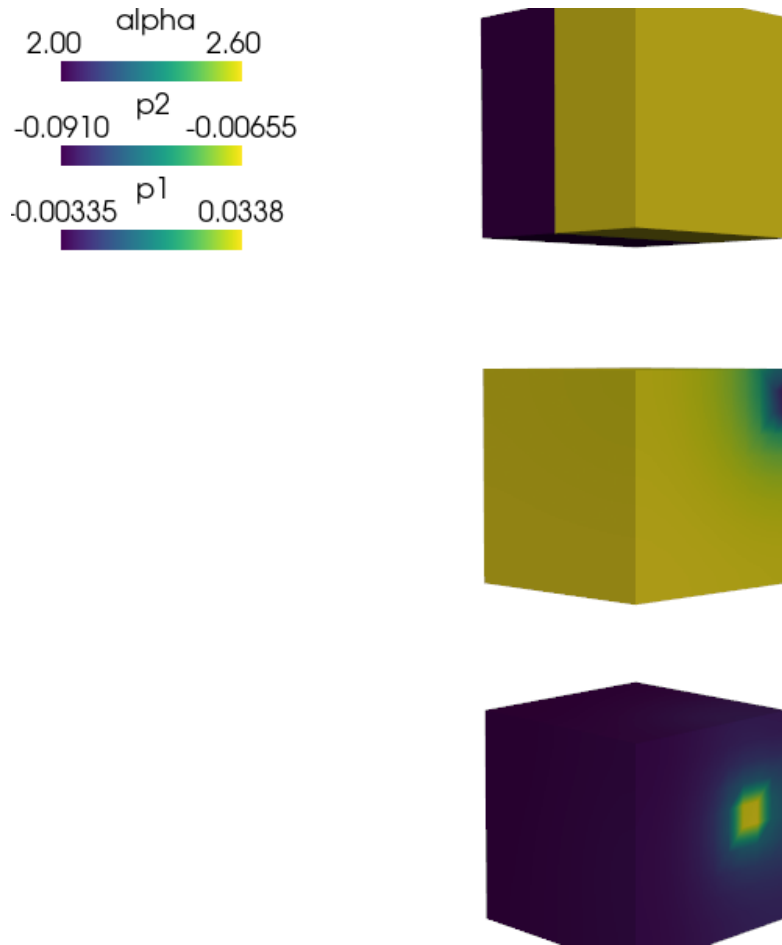
Description

Each of the two equations describes a flow in one compartment of a porous medium. The equations are based on the Darcy flow and the i -th compartment is defined in Ω_i .

$$\int_{\Omega_i} K^i \nabla p^i \cdot \nabla q^i + \int_{\Omega_i} \sum_j \bar{G}_{\alpha_k} (p^i - p^j) q^i = \int_{\Omega_i} f^i q^i,$$

$$\forall q^i \in Q^i, \quad i, j = 1, 2 \quad \text{and} \quad i \neq j,$$

where K^i is the local permeability of the i -th compartment, $\bar{G}_{\alpha_k} = G_j^i$ is the perfusion coefficient related to the compartments i and j , f^i are sources or sinks which represent the external flow into the i -th compartment and p^i is the pressure in the i -th compartment.



source code

```

r"""
Each of the two equations describes a flow in one compartment of a porous
medium. The equations are based on the Darcy flow and the  $i$ -th compartment is
defined in  $\Omega_i$ .

.. math::
    \int_{\Omega_i} K^i \nabla p^i \cdot \nabla q^i + \int_{\Omega_i}
    \sum_j \bar{G} \alpha_k \left( p^i - p^j \right) q^i
    = \int_{\Omega_i} f^i q^i,
.. math::
    \text{forall } q^i \in Q^i, \quad \text{quad } i,j=1,2 \quad \text{and} \quad i \neq j,

where  $K^i$  is the local permeability of the  $i$ -th compartment,
 $\bar{G} \alpha_k = G_{i-j}$  is the perfusion coefficient
related to the compartments  $i$  and  $j$ ,  $f^i$  are
sources or sinks which represent the external flow into the  $i$ -th
compartment and  $p^i$  is the pressure in the  $i$ -th compartment.
"""

from __future__ import absolute_import
from sfepy.base.base import Struct

```

(continues on next page)

(continued from previous page)

```

import numpy as nm
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'
G_bar = 2.0
alpha1 = 1.3
alpha2 = 1.0

materials = {
    'mat': ('mat_fun')
}

regions = {
    'Omega': 'cells of group 0',
    'Sigma_1': ('vertex 0', 'vertex'),
    'Omega1': ('copy r.Omega', 'cell', 'Omega'),
    'Omega2': ('copy r.Omega', 'cell', 'Omega'),
    'Source': 'cell 24',
    'Sink': 'cell 1',
}

fields = {
    'pressure': ('real', 1, 'Omega', 1)
}

variables = {
    'p1': ('unknown field', 'pressure'),
    'q1': ('test field', 'pressure', 'p1'),
    'p2': ('unknown field', 'pressure'),
    'q2': ('test field', 'pressure', 'p2'),
}

ebcs = {
    'P1': ('Sigma_1', {'p1.0' : 0.0}),
}

equations = {
    'komp1': """dw_diffusion.5.Omega1(mat.K, q1, p1)
                + dw_dot.5.Omega1(mat.G_alfa, q1, p1)
                - dw_dot.5.Omega1(mat.G_alfa, q1, p2)
                = dw_integrate.5.Source(mat.f_1, q1)""",

    'komp2': """dw_diffusion.5.Omega2(mat.K, q2, p2)
                + dw_dot.5.Omega2(mat.G_alfa, q2, p2)
                - dw_dot.5.Omega2(mat.G_alfa, q2, p1)
                = dw_integrate.5.Sink(mat.f_2, q2)"""
}

solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton',
               {'i_max' : 1,

```

(continues on next page)

(continued from previous page)

```

        'eps_a'      : 1e-6,
        'eps_r'      : 1.0,
    })
}

def mat_fun(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        nqp, dim = coors.shape
        alpha = nm.zeros((nqp,1,1), dtype=nm.float64)
        alpha[0:nqp // 2,...] = alpha1
        alpha[nqp // 2:,...] = alpha2
        K = nm.eye(dim, dtype=nm.float64)
        K2 = nm.tile(K, (nqp,1,1))
        out = {
            'K' : K2,
            'f_1': 20.0 * nm.ones((nqp,1,1), dtype=nm.float64),
            'f_2': -20.0 * nm.ones((nqp,1,1), dtype=nm.float64),
            'G_alfa': G_bar * alpha,
        }

        return out

functions = {
    'mat_fun': (mat_fun,),
}

options = {
    'post_process_hook': 'postproc',
}

def postproc(out, pb, state, extend=False):
    alpha = pb.evaluate('ev_integrate_mat.5.Omega(mat.G_alfa, p1)',
                        mode='el_avg')
    out['alpha'] = Struct(name='output_data',
                        mode='cell',
                        data=alpha.reshape(alpha.shape[0], 1, 1, 1),
                        dofs=None)

    return out

```

diffusion/laplace_1d.py

Description

Laplace equation in 1D with a variable coefficient.

Because the mesh is trivial in 1D, it is generated by `mesh_hook()`, and registered using `UserMeshIO`.

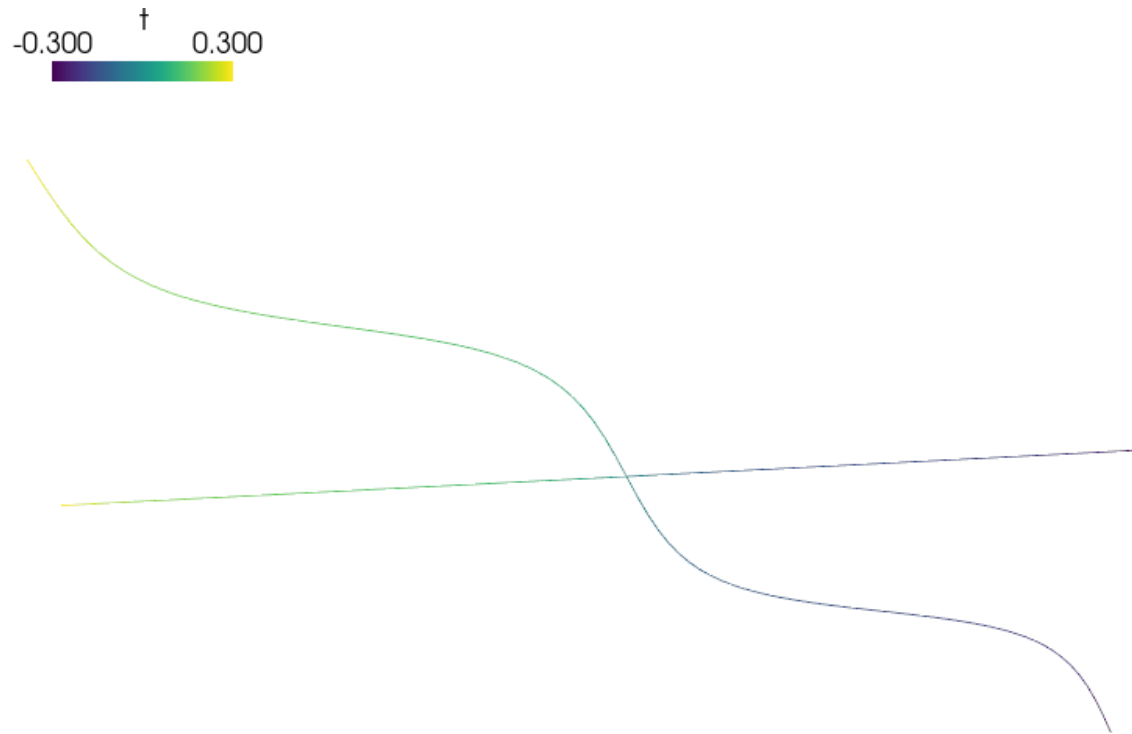
Find t such that:

$$\int_{\Omega} c(x) \frac{ds}{dx} \frac{dt}{dx} = 0, \quad \forall s,$$

where the coefficient $c(x) = 0.1 + \sin(2\pi x)^2$ is computed in `get_coef()`.

View the results using:

```
sfepy-view laplace_1d.vtk -f t:wt 1:vw
```



source code

```
r"""
Laplace equation in 1D with a variable coefficient.

Because the mesh is trivial in 1D, it is generated by :func:`mesh_hook()`, and
registered using :class:`UserMeshIO` <sfepy.discrete.fem.meshio.UserMeshIO>`.

Find :math:`t` such that:

.. math::
    \int_{\Omega} c(x) \, \text{tdiff}[s][x] \, \text{tdiff}[t][x]
    = 0
    \;, \quad \text{forall } s \;,

where the coefficient :math:`c(x) = 0.1 + \sin(2 \pi x)^2` is computed in
:func:`get_coef()`.

View the results using::

    sfepy-view laplace_1d.vtk -f t:wt 1:vw
```

(continues on next page)

(continued from previous page)

```

"""
from __future__ import absolute_import
import numpy as nm
from sfepy.discrete.fem import Mesh
from sfepy.discrete.fem.meshio import UserMeshIO

def mesh_hook(mesh, mode):
    """
    Generate the 1D mesh.
    """
    if mode == 'read':
        n_nod = 101

        coors = nm.linspace(0.0, 1.0, n_nod).reshape((n_nod, 1))
        conn = nm.arange(n_nod, dtype=nm.int32).repeat(2)[1:-1].reshape((-1, 2))
        mat_ids = nm.zeros(n_nod - 1, dtype=nm.int32)
        descs = ['1_2']

        mesh = Mesh.from_data('laplace_1d', coors, None,
                              [conn], [mat_ids], descs)

        return mesh

    elif mode == 'write':
        pass

def get_coef(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        x = coors[:, 0]

        val = 0.1 + nm.sin(2 * nm.pi * x)**2
        val.shape = (coors.shape[0], 1, 1)

        return {'val' : val}

filename_mesh = UserMeshIO(mesh_hook)

materials = {
    'coef' : 'get_coef',
}

functions = {
    'get_coef' : (get_coef,),
}

regions = {
    'Omega' : 'all',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),

```

(continues on next page)

(continued from previous page)

```
}

variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

ebcs = {
    't1' : ('Gamma_Left', {'t.0' : 0.3}),
    't2' : ('Gamma_Right', {'t.0' : -0.3}),
}

integrals = {
    'i' : 2,
}

equations = {
    'Temperature' : ""dw_laplace.i.Omega(coef.val, s, t) = 0""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
}
```

diffusion/laplace_coupling_lcbcs.py

Description

Two Laplace equations with multiple linear combination constraints.

The two equations are coupled by a periodic-like boundary condition constraint with a shift, given as a non-homogeneous linear combination boundary condition.

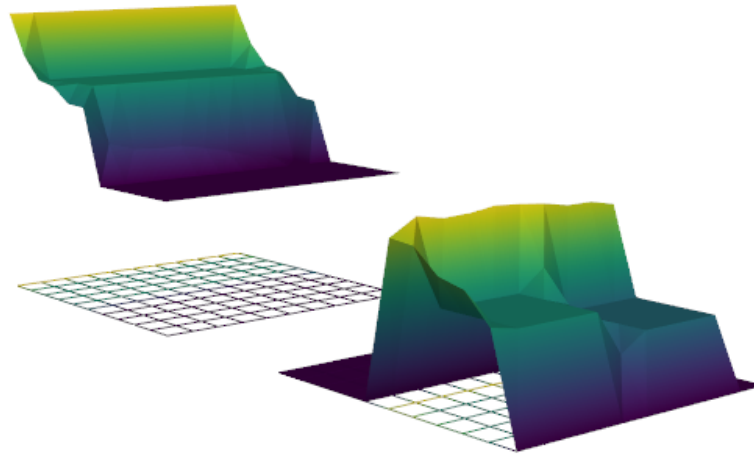
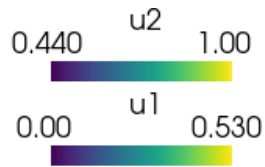
Find u such that:

$$\begin{aligned}
 \int_{\Omega_1} \nabla v_1 \cdot \nabla u_1 &= 0, \quad \forall v_1, \\
 \int_{\Omega_2} \nabla v_2 \cdot \nabla u_2 &= 0, \quad \forall v_2, \\
 u_1 &= 0 \text{ on } \Gamma_{bottom}, \\
 u_2 &= 1 \text{ on } \Gamma_{top}, \\
 u_1(\underline{x}) &= u_2(\underline{x}) + a(\underline{x}) \text{ for } \underline{x} \in \Gamma = \bar{\Omega}_1 \cap \bar{\Omega}_2 \\
 u_1(\underline{x}) &= u_1(\underline{y}) + b(\underline{y}) \text{ for } \underline{x} \in \Gamma_{left}, \underline{y} \in \Gamma_{right}, \underline{y} = P(\underline{x}), \\
 u_1 &= c_{11} \text{ in } \Omega_{m11} \subset \Omega_1, \\
 u_1 &= c_{12} \text{ in } \Omega_{m12} \subset \Omega_1, \\
 u_2 &= c_2 \text{ in } \Omega_{m2} \subset \Omega_2,
 \end{aligned}$$

where $a(\underline{x})$, $b(\underline{y})$ are given functions (shifts), P is the periodic coordinate mapping and c_{11} , c_{12} and c_2 are unknown constant values - the unknown DOFs in Ω_{m11} , Ω_{m12} and Ω_{m2} are replaced by the integral mean values.

View the results using:

```
sfepy-view square_quad.vtk -f u1:wu1:p0 1:vw:p0 u2:wu2:p1 1:vw:p1
```



[source code](#)

```

r"""
Two Laplace equations with multiple linear combination constraints.

The two equations are coupled by a periodic-like boundary condition constraint
with a shift, given as a non-homogeneous linear combination boundary condition.

Find :math:u` such that:

.. math::
    \int_{\Omega_1} \nabla v_1 \cdot \nabla u_1
    = 0
    \quad \forall v_1 \in V_1,

    \int_{\Omega_2} \nabla v_2 \cdot \nabla u_2
    = 0
    \quad \forall v_2 \in V_2,

    u_1 = 0 \quad \text{on } \Gamma_{\text{bottom}},

    u_2 = 1 \quad \text{on } \Gamma_{\text{top}},

    u_1(x) = u_2(x) + a(x) \quad \text{for } x \in \Gamma = \bar{\Omega}_1 \cap \bar{\Omega}_2

    u_1(x) = u_1(y) + b(y) \quad \text{for } x \in \Gamma_{\text{left}}, y \in \Gamma_{\text{right}}, u(y) = P(x) \quad \forall x, y \in \Gamma,

    u_1 = c_{11} \quad \text{in } \Omega_{m1} \subset \Omega_1,

    u_1 = c_{12} \quad \text{in } \Omega_{m2} \subset \Omega_1,

    u_2 = c_2 \quad \text{in } \Omega_{m2} \subset \Omega_2,

where :math:a(x), :math:b(y)` are given functions (shifts),
:math:P` is the periodic coordinate mapping and :math:c_{11}, :math:c_{12}`
and :math:c_2` are unknown constant values - the unknown DOFs in
:math:\Omega_{m1}, :math:\Omega_{m2}` and :math:\Omega_{m2}` are replaced
by the integral mean values.

View the results using::

    sfepy-view square_quad.vtk -f u1:wu1:p0 1:vw:p0 u2:wu2:p1 1:vw:p1
"""
from __future__ import absolute_import
import numpy as nm

import sfepy.discrete.fem.periodic as per
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/2d/square_quad.mesh'

options = {
    'nls' : 'newton',

```

(continues on next page)

(continued from previous page)

```

    'ls' : 'ls',
}

def get_shift1(ts, coors, region):
    val = 0.1 * coors[:, 0]

    return val

def get_shift2(ts, coors, region):
    val = nm.empty_like(coors[:, 1])
    val.fill(0.3)

    return val

functions = {
    'get_shift1' : (get_shift1,),
    'get_shift2' : (get_shift2,),
    'match_y_line' : (per.match_y_line,),
    'match_x_line' : (per.match_x_line,),
}

fields = {
    'scalar1': ('real', 1, 'Omega1', 1),
    'scalar2': ('real', 1, 'Omega2', 1),
}

materials = {
}

variables = {
    'u1' : ('unknown field', 'scalar1', 0),
    'v1' : ('test field', 'scalar1', 'u1'),
    'u2' : ('unknown field', 'scalar2', 1),
    'v2' : ('test field', 'scalar2', 'u2'),
}

regions = {
    'Omega1' : 'cells of group 1',
    'Omega2' : 'cells of group 2',
    'Omega_m1' : 'r.Omega1 -v (r.Gamma +s vertices of surface)',
    'Omega_m11' : 'r.Omega_m1 *v vertices in (x < 0)',
    'Omega_m12' : 'r.Omega_m1 *v vertices in (x > 0)',
    'Omega_m2' : 'r.Omega2 -v (r.Gamma +s vertices of surface)',
    'Left' : ('vertices in (x < -0.499)', 'facet'),
    'Right' : ('vertices in (x > 0.499)', 'facet'),
    'Bottom' : ('vertices in ((y < -0.499))', 'facet'),
    'Top' : ('vertices in ((y > 0.499))', 'facet'),
    'Gamma' : ('r.Omega1 *v r.Omega2 -v vertices of surface', 'facet'),
    'Gamma1' : ('copy r.Gamma', 'facet', 'Omega1'),
    'Gamma2' : ('copy r.Gamma', 'facet', 'Omega2'),
}

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    'fix1' : ('Top', {'u2.all' : 1.0}),
    'fix2' : ('Bottom', {'u1.all' : 0.0}),
}

lcbcs = {
    'shifted1' : (('Gamma1', 'Gamma2'),
                  {'u1.all' : 'u2.all'},
                  'match_x_line', 'shifted_periodic',
                  'get_shift1'),
    'shifted2' : (('Left', 'Right'),
                  {'u1.all' : 'u1.all'},
                  'match_y_line', 'shifted_periodic',
                  'get_shift2'),
    'mean11' : ('Omega_m11', {'u1.all' : None}, None, 'integral_mean_value'),
    'mean12' : ('Omega_m12', {'u1.all' : None}, None, 'integral_mean_value'),
    'mean2' : ('Omega_m2', {'u2.all' : None}, None, 'integral_mean_value'),
}

integrals = {
    'i1' : 2,
}

equations = {
    'eq1' : """
        dw_laplace.i1.Omega1(v1, u1) = 0
    """,
    'eq2' : """
        dw_laplace.i1.Omega2(v2, u2) = 0
    """,
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

```

diffusion/laplace_fluid_2d.py

Description

A Laplace equation that models the flow of “dry water” around an obstacle shaped like a Citroen CX.

Description

As discussed e.g. in the Feynman lectures Section 12-5 of Volume 2 (https://www.feynmanlectures.caltech.edu/II_12.html#Ch12-S5), the flow of an irrotational and incompressible fluid can be modelled with a potential $\underline{v} = \underline{grad}(\phi)$ that obeys

$$\nabla \cdot \underline{\nabla} \phi = \Delta \phi = 0$$

The weak formulation for this problem is to find ϕ such that:

$$\int_{\Omega} \nabla \psi \cdot \nabla \phi = \int_{\Gamma_{left}} \underline{v}_0 \cdot \underline{n} \psi + \int_{\Gamma_{right}} \underline{v}_0 \cdot \underline{n} \psi + \int_{\Gamma_{top}} \underline{v}_0 \cdot \underline{n} \psi + \int_{\Gamma_{bottom}} \underline{v}_0 \cdot \underline{n} \psi, \quad \forall \psi,$$

where $\underline{v}_0$ is the 2D vector defining the far field velocity that generates the incompressible flow.

Since the value of the potential is defined up to a constant value, a Dirichlet boundary condition is set at a single vertex to avoid having a singular matrix.

Usage examples

This example can be run with the following:

```
sfepy-run sfepy/examples/diffusion/laplace_fluid_2d.py
sfepy-view citroen.vtk -f phi:p0 phi:t50:p0 --2d-view
```

Generating the mesh

The mesh can be generated with:

```
gmsh -2 -f msh22 meshes/2d/citroen.geo -o meshes/2d/citroen.msh
sfepy-convert --2d meshes/2d/citroen.msh meshes/2d/citroen.h5
```

0.00 ϕ $3.58\text{e}+03$




source code

```
r"""
A Laplace equation that models the flow of "dry water" around an obstacle
shaped like a Citroen CX.

Description
-----

As discussed e.g. in the Feynman lectures Section 12-5 of Volume 2
(https://www.feynmanlectures.caltech.edu/II\_12.html#Ch12-S5),
the flow of an irrotational and incompressible fluid can be modelled with a
potential  $\mathbf{v} = \text{grad}(\phi)$  that obeys

.. math::
    \nabla \cdot \mathbf{v} = \Delta \phi = 0

The weak formulation for this problem is to find  $\phi$  such that:

.. math::
    \int_{\Omega} \nabla \psi \cdot \nabla \phi
    = \int_{\Gamma_{\text{left}}} \mathbf{v}_0 \cdot \mathbf{n} \, \psi
```

(continues on next page)

(continued from previous page)

```

+ \int_{\Gamma_{right}} \ul{v}_0 \cdot n \, , \, \psi
+ \int_{\Gamma_{top}} \ul{v}_0 \cdot n \, , \, \psi
+ \int_{\Gamma_{bottom}} \ul{v}_0 \cdot n \, , \, \psi
\;, \, \text{quad} \, \text{forall} \, \psi \, \;,

```

where $\ul{v}_0$ is the 2D vector defining the far field velocity that generates the incompressible flow.

Since the value of the potential is defined up to a constant value, a Dirichlet boundary condition is set at a single vertex to avoid having a singular matrix.

Usage examples

This example can be run with the following::

```

sfepy-run sfepy/examples/diffusion/laplace_fluid_2d.py
sfepy-view citroen.vtk -f phi:p0 phi:t50:p0 --2d-view

```

Generating the mesh

The mesh can be generated with::

```

gmsh -2 -f msh22 meshes/2d/citroen.geo -o meshes/2d/citroen.msh
sfepy-convert --2d meshes/2d/citroen.msh meshes/2d/citroen.h5
"""

```

```

import numpy as nm
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/2d/citroen.h5'

v0 = nm.array([1, 0.25])

materials = {
    'm': ({'v0': v0.reshape(-1, 1)}),
}

regions = {
    'Omega': 'all',
    'Gamma_Left': ('vertices in (x < 0.1)', 'facet'),
    'Gamma_Right': ('vertices in (x > 1919.9)', 'facet'),
    'Gamma_Top': ('vertices in (y > 917.9)', 'facet'),
    'Gamma_Bottom': ('vertices in (y < 0.1)', 'facet'),
    'Vertex': ('vertex in r.Gamma_Left', 'vertex'),
}

fields = {
    'u': ('real', 1, 'Omega', 1),
}

variables = {

```

(continues on next page)

(continued from previous page)

```
'phi': ('unknown field', 'u', 0),
'psi': ('test field', 'u', 'phi'),
}

# these EBCS prevent the matrix from being singular, see description
ebcs = {
    'fix': ('Vertex', {'phi.0': 0.0}),
}

integrals = {
    'i': 2,
}

equations = {
    'Laplace equation':
        """dw_laplace.i.Omega( psi, phi )
        = dw_surface_ndot.i.Gamma_Left( m.v0, psi )
        + dw_surface_ndot.i.Gamma_Right( m.v0, psi )
        + dw_surface_ndot.i.Gamma_Top( m.v0, psi )
        + dw_surface_ndot.i.Gamma_Bottom( m.v0, psi )"""
}

solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-10,
    }),
}
```

diffusion/laplace_iga_interactive.py

Description

Laplace equation with Dirichlet boundary conditions solved in a single patch NURBS domain using the isogeometric analysis (IGA) approach, using commands for interactive use.

This script allows the creation of a customisable NURBS surface using igakit built-in CAD routines, which is then saved in custom HDF5-based files with .iga extension.

Notes

The `create_patch` function creates a NURBS-patch of the area between two coplanar nested circles using igakit CAD built-in routines. The created patch is not connected in the orthoradial direction. This is a problem when the disconnected boundary is not perpendicular to the line connecting the two centres of the circles, as the solution then exhibits a discontinuity along this line. A workaround for this issue is to enforce perpendicularity by changing the start angle in function `igakit.cad.circle` (see the code down below for the actual trick). The discontinuity disappears.

Usage Examples

Default options, storing results in this file's parent directory:

```
python3 sfepy/examples/diffusion/laplace_iga_interactive.py
```

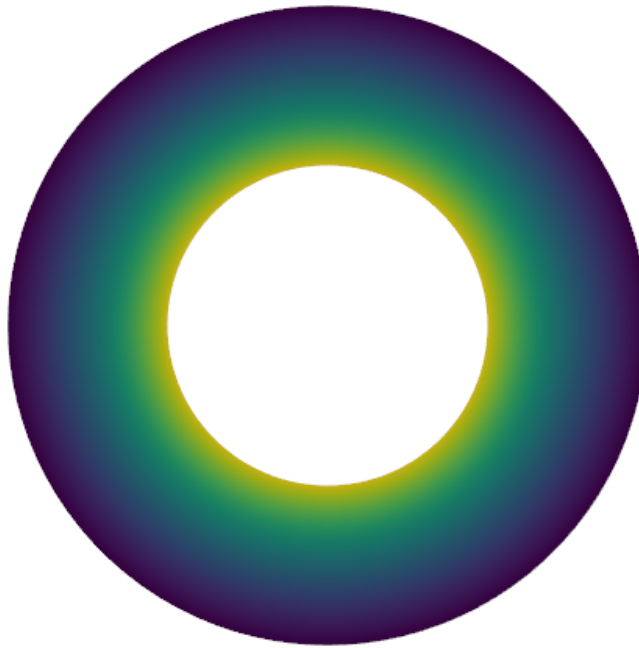
Command line options for tweaking the geometry of the NURBS-patch & more:

```
python3 sfepy/examples/diffusion/laplace_iga_interactive.py --R1=0.7 --C2=0.1,0.1 --
↪viewpatch
```

View the results using:

```
sfepy-view concentric_circles.vtk
```

3.00 u 7.00



source code

```
#!/usr/bin/env python
r"""
Laplace equation with Dirichlet boundary conditions solved in a single patch
NURBS domain using the isogeometric analysis (IGA) approach, using commands
for interactive use.

This script allows the creation of a customisable NURBS surface using igakit
```

(continues on next page)

(continued from previous page)

built-in CAD routines, which is then saved in custom HDF5-based files with .iga extension.

Notes

The ``create\_patch`` function creates a NURBS-patch of the area between two coplanar nested circles using igakit CAD built-in routines. The created patch is not connected in the orthoradial direction. This is a problem when the disconnected boundary is not perpendicular to the line connecting the two centres of the circles, as the solution then exhibits a discontinuity along this line. A workaround for this issue is to enforce perpendicularity by changing the start angle in function ``igakit.cad.circle`` (see the code down below for the actual trick). The discontinuity disappears.

Usage Examples

Default options, storing results in this file's parent directory::

```
python3 sfepy/examples/diffusion/laplace_iga_interactive.py
```

Command line options for tweaking the geometry of the NURBS-patch & more::

```
python3 sfepy/examples/diffusion/laplace_iga_interactive.py --R1=0.7 --C2=0.1,0.1 --
↪viewpatch
```

View the results using::

```
sfepy-view concentric_circles.vtk
"""
```

```
from argparse import RawDescriptionHelpFormatter, ArgumentParser
```

```
import os
```

```
import sys
```

```
sys.path.append('.')
```

```
import numpy as nm
```

```
from sfepy import data_dir
```

```
from sfepy.base.ioutils import ensure_path
```

```
from sfepy.base.base import IndexedStruct
```

```
from sfepy.discrete import (FieldVariable, Integral, Equation, Equations,
                             Problem)
```

```
from sfepy.discrete.iga.domain import IGDomein
```

```
from sfepy.discrete.common.fields import Field
```

```
from sfepy.terms import Term
```

```
from sfepy.discrete.conditions import Conditions, EssentialBC
```

```
from sfepy.solvers.ls import ScipyDirect
```

```
from sfepy.solvers.nls import Newton
```

```
def create_patch(R1, R2, C1, C2, order=2, viewpatch=False):
```

(continues on next page)

(continued from previous page)

```

"""
Create a single 2d NURBS-patch of the area between two coplanar nested
circles using igakit.

Parameters
-----
R1 : float
    Radius of the inner circle.
R2 : float
    Radius of the outer circle.
C1 : list of two floats
    Coordinates of the center of the inner circle given as [x1, y1].
C2 : list of two floats
    Coordinates of the center of the outer circle given as [x2, y2].
order : int, optional
    Degree of the NURBS basis functions. The default is 2.
viewpatch : bool, optional
    When set to True, display the NURBS patch. The default is False.

Returns
-----
None.

"""

from sfepy.discrete.iga.domain_generators import create_from_igakit
import sfepy.discrete.iga.io as io
from igakit.cad import circle, ruled
from igakit.plot import plt as iplt
from numpy import pi

# Assert the inner circle is inside the outer one
inter_centers = nm.sqrt((C2[0]-C1[0])**2 + (C2[1]-C1[1])**2)
assert R2>R1, "Outer circle should have a larger radius than the inner one"
assert inter_centers<R2-R1, "Circles are not nested"

# Geometry Creation
centers_direction = [C2[0]-C1[0], C2[1]-C1[1]]
if centers_direction[0]==0 and centers_direction[1]==0:
    start_angle = 0.0
else:
    start_angle = nm.arctan2(centers_direction[1], centers_direction[0])
c1 = circle(radius=R1, center=C1, angle=(start_angle, start_angle + 2*pi))
c2 = circle(radius=R2, center=C2, angle=(start_angle, start_angle + 2*pi))
srf = ruled(c1,c2).transpose() # make the radial direction first

# Refinement
insert_U = insert_uniformly(srf.knots[0], 6)
insert_V = insert_uniformly(srf.knots[1], 6)
srf.refine(0, insert_U).refine(1, insert_V)

# Setting the NURBS-surface degree

```

(continues on next page)

(continued from previous page)

```

srf.elevate(0, order=srf.degree[0] if order-srf.degree[0] > 0 else 0)
srf.elevate(1, order=srf.degree[1] if order-srf.degree[1] > 0 else 0)

# SfePy .iga file creation
nurbs, bmesh, regions = create_from_igakit(srf, verbose=True)

# Save .iga file in sfepy/meshes/iga
filename_domain = data_dir + '/meshes/iga/concentric_circles.iga'
io.write_iga_data(filename_domain, None, nurbs.knots, nurbs.degrees,
                  nurbs.cps, nurbs.weights, nurbs.cs, nurbs.conn,
                  bmesh.cps, bmesh.weights, bmesh.conn, regions)

if viewpatch:
    iplt.use('matplotlib')
    iplt.figure()
    iplt.plot(srf)
    iplt.show()

def insert_uniformly(U, n):
    """
    Find knots to uniformly add to U.
    [Code from igakit/demo/venturi.py file]

    Given a knot vector U and the number of uniform spans desired,
    find the knots which need to be inserted.

    Parameters
    -----
    U : numpy.ndarray
        Original knot vector for a  $C^{p-1}$  space.
    n : int
        Target number of uniformly-spaced knot spans.

    Returns
    -----
    Knots to be inserted into U
    """
    U0 = U
    dU = (U.max() - U.min()) / float(n) # target dU in knot vector
    idone = 0
    while idone == 0:
        # Add knots in middle of spans which are too large
        Uadd = []
        for i in range(len(U)-1):
            if U[i+1] - U[i] > dU:
                Uadd.append(0.5 * (U[i+1] + U[i]))
        # Now we add these knots (once only, assumes  $C^{p-1}$ )
        if len(Uadd) > 0:
            U = nm.sort(nm.concatenate([U, nm.asarray(Uadd)]))
        else:
            idone = 1
        # And now a little Laplacian smoothing

```

(continues on next page)

(continued from previous page)

```

        for num_iterations in range(5):
            for i in range(len(U)-2):
                if abs(U0[U0.searchsorted(U[i+1])]-U[i+1]) > 1.0e-14:
                    U[i+1] = 0.5*(U[i]+U[i+2])
    return nm.setdiff1d(U,U0)

helps = {
    'output_dir' :
    'output directory',
    'R1' :
    'Inner circle radius [default: %(default)s]',
    'R2' :
    'Outer circle radius [default: %(default)s]',
    'C1' :
    'centre of the inner circle [default: %(default)s]',
    'C2' :
    'centre of the outer circle [default: %(default)s]',
    'order' :
    'field approximation order [default: %(default)s]',
    'viewpatch' :
    'generate a plot of the NURBS-patch',
}

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                            formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('-o', '--output-dir', default='.',
                        help=helps['output_dir'])
    parser.add_argument('--R1', metavar='R1',
                        action='store', dest='R1',
                        default='0.5', help=helps['R1'])
    parser.add_argument('--R2', metavar='R2',
                        action='store', dest='R2',
                        default='1.0', help=helps['R2'])
    parser.add_argument('--C1', metavar='C1',
                        action='store', dest='C1',
                        default='0.0,0.0', help=helps['C1'])
    parser.add_argument('--C2', metavar='C2',
                        action='store', dest='C2',
                        default='0.0,0.0', help=helps['C2'])
    parser.add_argument('--order', metavar='int', type=int,
                        action='store', dest='order',
                        default=2, help=helps['order'])
    parser.add_argument('-v', '--viewpatch',
                        action='store_true', dest='viewpatch',
                        default=False, help=helps['viewpatch'])
    options = parser.parse_args()

    # Creation of the NURBS-patch with igakit
    R1 = eval(options.R1)
    R2 = eval(options.R2)
    C1 = list(eval(options.C1))

```

(continues on next page)

(continued from previous page)

```

C2 = list(eval(options.C2))
order = options.order
viewpatch = options.viewpatch
create_patch(R1, R2, C1, C2, order=order, viewpatch=viewpatch)

# Setting a Domain instance
filename_domain = data_dir + '/meshes/iga/concentric_circles.iga'
domain = IGDDomain.from_file(filename_domain)

# Sub-domains
omega = domain.create_region('Omega', 'all')
Gamma_out = domain.create_region('Gamma_out', 'vertices of set xi01',
                                  kind='facet')
Gamma_in = domain.create_region('Gamma_in', 'vertices of set xi00',
                                 kind='facet')

# Field (featuring order elevation)
order_increase = order - domain.nurbs.degrees[0]
order_increase *= int(order_increase>0)
field = Field.from_args('fu', nm.float64, 'scalar', omega,
                       approx_order='iga', space='H1',
                       poly_space_basis='iga')

# Variables
u = FieldVariable('u', 'unknown', field) # unknown function
v = FieldVariable('v', 'test', field, primary_var_name='u') # test function

# Integral
integral = Integral('i', order=2*field.approx_order)

# Term
t = Term.new('dw_laplace( v, u )', integral, omega, v=v, u=u)

# Equation
eq = Equation('laplace', t)
eqs = Equations([eq])

# Boundary Conditions
u_in = EssentialBC('u_in', Gamma_in, {'u.all' : 7.0})
u_out = EssentialBC('u_out', Gamma_out, {'u.all' : 3.0})

# solvers
ls = ScipyDirect({})
nls_status = IndexedStruct()
nls = Newton({}, lin_solver=ls, status=nls_status)

# problem instance
pb = Problem('potential', equations=eqs, active_only=True)

# Set boundary conditions
pb.set_bcs(ebcs=Conditions([u_in, u_out]))

```

(continues on next page)

(continued from previous page)

```

# solving
pb.set_solver(nls)
status = IndexedStruct()
state = pb.solve(status=status, save_results=True, verbose=True)

# Saving the results to a classic VTK file
filename = os.path.join(options.output_dir, 'concentric_circles.vtk')
ensure_path(filename)
pb.save_state(filename, state)

if __name__ == '__main__':
    main()

```

diffusion/laplace_refine_interactive.py

Description

Example of solving Laplace's equation on a block domain refined with level 1 hanging nodes.

The domain is progressively refined towards the edge/face of the block, where Dirichlet boundary conditions are prescribed by an oscillating function.

Find u such that:

$$\int_{\Omega} \nabla v \cdot \nabla u = 0, \quad \forall v.$$

Notes

The implementation of the mesh refinement with level 1 hanging nodes is a proof-of-concept code with many unresolved issues. The main problem is the fact that a user needs to input the cells to refine at each level, while taking care of the following constraints:

- the level 1 hanging nodes constraint: a cell that has a less-refined neighbour cannot be refined;
- the implementation constraint: a cell with a refined neighbour cannot be refined.

The hanging nodes are treated by a basis transformation/DOF substitution, which has to be applied explicitly by the user:

- call `field.substitute_dofs(subs)` before assembling and solving;
- then call `field.restore_dofs()` before saving results.

Usage Examples

Default options, 2D, storing results in 'output' directory:

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py output
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw
```

Default options, 3D, storing results in 'output' directory:

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py -3 output
sfepy-view output/hanging.vtk -f u:wu:f0.1 1:vw
```

Finer initial domain, 2D, storing results in ‘output’ directory:

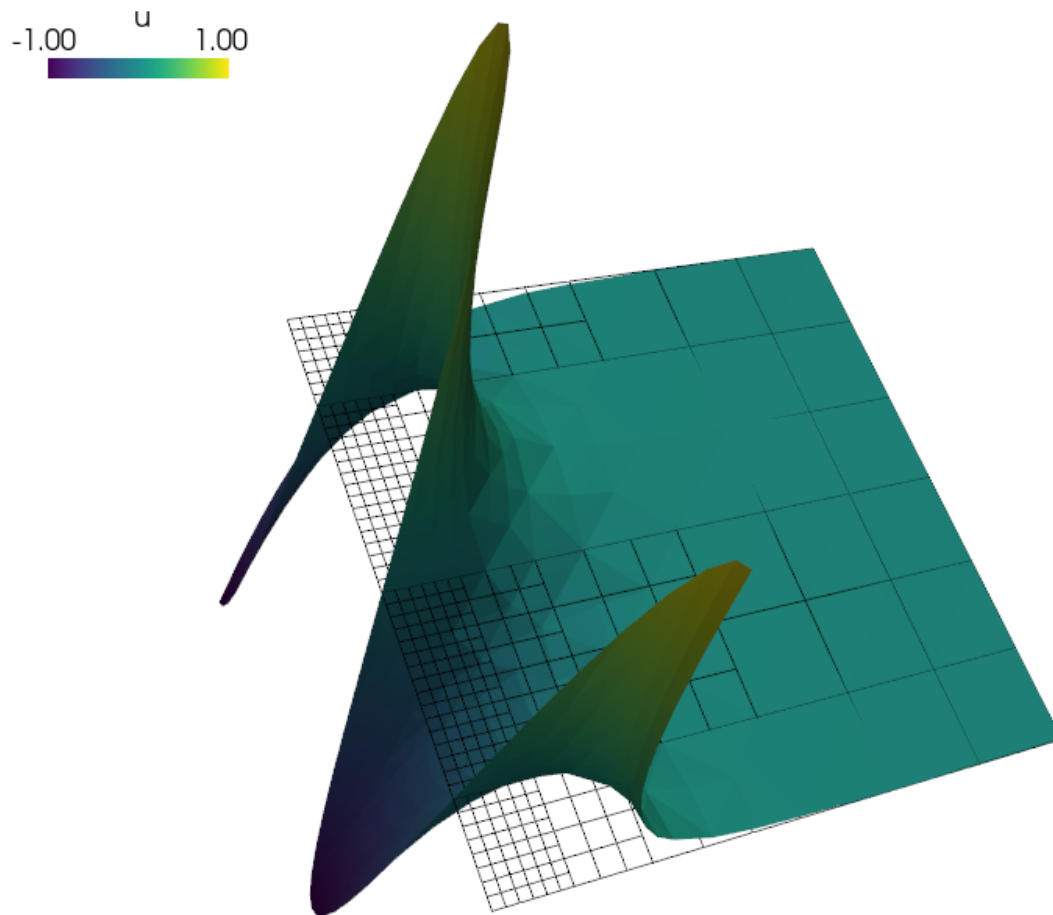
```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py --shape=11,11 output
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw
```

Bi-quadratic approximation, 2D, storing results in ‘output’ directory:

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py --order=2 output
```

```
# View solution with higher order DOFs removed.
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw

# View full solution on a mesh adapted for visualization.
sfepy-view output/hanging_u.vtk -2 -f u:wu 1:vw
```



source code

```
#!/usr/bin/env python
r"""
Example of solving Laplace's equation on a block domain refined with level 1
```

(continues on next page)

(continued from previous page)

hanging nodes.

The domain is progressively refined towards the edge/face of the block, where Dirichlet boundary conditions are prescribed by an oscillating function.

Find u such that:

```
.. math::
    \int_{\Omega} \nabla v \cdot \nabla u = 0
    \;; \quad \forall s \;;
```

Notes

The implementation of the mesh refinement with level 1 hanging nodes is a proof-of-concept code with many unresolved issues. The main problem is the fact that a user needs to input the cells to refine at each level, while taking care of the following constraints:

- the level 1 hanging nodes constraint: a cell that has a less-refined neighbour cannot be refined;*
- the implementation constraint: a cell with a refined neighbour cannot be refined.*

The hanging nodes are treated by a basis transformation/DOF substitution, which has to be applied explicitly by the user:

- call `field.substitute_dofs(subs)` before assembling and solving;*
- then call `field.restore_dofs()` before saving results.*

Usage Examples

Default options, 2D, storing results in 'output' directory::

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py output
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw
```

Default options, 3D, storing results in 'output' directory::

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py -3 output
sfepy-view output/hanging.vtk -f u:wu:f0.1 1:vw
```

Finer initial domain, 2D, storing results in 'output' directory::

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py --shape=11,11 output
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw
```

Bi-quadratic approximation, 2D, storing results in 'output' directory::

```
python3 sfepy/examples/diffusion/laplace_refine_interactive.py --order=2 output
```

(continues on next page)

(continued from previous page)

```

# View solution with higher order DOFs removed.
sfepy-view output/hanging.vtk -2 -f u:wu 1:vw

# View full solution on a mesh adapted for visualization.
sfepy-view output/hanging_u.vtk -2 -f u:wu 1:vw
"""
from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser

import os
import sys
sys.path.append('.')
import numpy as nm

from sfepy.base.base import output, Struct
from sfepy.base.ioutils import ensure_path
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.discrete import (FieldVariable, Integral, Equation, Equations,
                             Function, Problem)
from sfepy.discrete.fem import FEDomain, Field
from sfepy.discrete.conditions import (Conditions, EssentialBC)
import sfepy.discrete.fem.refine_hanging as rh
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.terms import Term

def refine_towards_facet(domain0, grading, axis):
    subs = None
    domain = domain0
    for level, coor in enumerate(grading):
        refine = nm.zeros(domain.mesh.n_el, dtype=nm.uint8)

        region = domain.create_region('aux',
                                      'vertices in (%s %.10f)' % (axis, coor),
                                      add_to_regions=False)

        refine[region.cells] = 1

        domain, subs = rh.refine(domain, refine, subs=subs)

    return domain, subs

helps = {
    'output_dir' :
    'output directory',
    'dims' :
    'dimensions of the block [default: %(default)s]',
    'shape' :
    'shape (counts of nodes in x, y[, z]) of the block [default: %(default)s]',
    'centre' :
    'centre of the block [default: %(default)s]',
    '3d' :
    'generate a 3D block',

```

(continues on next page)

(continued from previous page)

```

'order' :
'field approximation order',
}

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('output_dir', help=helps['output_dir'])
    parser.add_argument('--dims', metavar='dims',
                        action='store', dest='dims',
                        default='1.0,1.0,1.0', help=helps['dims'])
    parser.add_argument('--shape', metavar='shape',
                        action='store', dest='shape',
                        default='7,7,7', help=helps['shape'])
    parser.add_argument('--centre', metavar='centre',
                        action='store', dest='centre',
                        default='0.0,0.0,0.0', help=helps['centre'])
    parser.add_argument('-3', '--3d',
                        action='store_true', dest='is_3d',
                        default=False, help=helps['3d'])
    parser.add_argument('--order', metavar='int', type=int,
                        action='store', dest='order',
                        default=1, help=helps['order'])
    options = parser.parse_args()

    dim = 3 if options.is_3d else 2
    dims = nm.array(eval(options.dims), dtype=nm.float64)[:dim]
    shape = nm.array(eval(options.shape), dtype=nm.int32)[:dim]
    centre = nm.array(eval(options.centre), dtype=nm.float64)[:dim]

    output('dimensions:', dims)
    output('shape:      ', shape)
    output('centre:     ', centre)

    mesh0 = gen_block_mesh(dims, shape, centre, name='block-fem',
                           verbose=True)
    domain0 = FEDomain('d', mesh0)

    bbox = domain0.get_mesh_bounding_box()
    min_x, max_x = bbox[:, 0]
    eps = 1e-8 * (max_x - min_x)

    cnt = (shape[0] - 1) // 2
    g0 = 0.5 * dims[0]
    grading = nm.array([g0 / 2**ii for ii in range(cnt)]) + eps + centre[0] - g0

    domain, subs = refine_towards_facet(domain0, grading, 'x <')

    omega = domain.create_region('Omega', 'all')

    gamma1 = domain.create_region('Gamma1',
                                   'vertices in (x < %.10f)' % (min_x + eps),

```

(continues on next page)

(continued from previous page)

```

                                'facet')
gamma2 = domain.create_region('Gamma2',
                                'vertices in (x > %.10f)' % (max_x - eps),
                                'facet')

field = Field.from_args('fu', nm.float64, 1, omega,
                        approx_order=options.order)

if subs is not None:
    field.substitute_dofs(subs)

u = FieldVariable('u', 'unknown', field)
v = FieldVariable('v', 'test', field, primary_var_name='u')

integral = Integral('i', order=2*options.order)

t1 = Term.new('dw_laplace(v, u)',
              integral, omega, v=v, u=u)
eq = Equation('eq', t1)
eqs = Equations([eq])

def u_fun(ts, coors, bc=None, problem=None):
    """
    Define a displacement depending on the y coordinate.
    """
    if coors.shape[1] == 2:
        min_y, max_y = bbox[:, 1]
        y = (coors[:, 1] - min_y) / (max_y - min_y)

        val = (max_y - min_y) * nm.cos(3 * nm.pi * y)

    else:
        min_y, max_y = bbox[:, 1]
        min_z, max_z = bbox[:, 2]
        y = (coors[:, 1] - min_y) / (max_y - min_y)
        z = (coors[:, 2] - min_z) / (max_z - min_z)

        val = ((max_y - min_y) * (max_z - min_z)
               * nm.cos(3 * nm.pi * y) * (1.0 + 3.0 * (z - 0.5)**2))

    return val

bc_fun = Function('u_fun', u_fun)
fix1 = EssentialBC('shift_u', gamma1, {'u.0' : bc_fun})
fix2 = EssentialBC('fix2', gamma2, {'u.all' : 0.0})

ls = ScipyDirect({})

nls = Newton({}, lin_solver=ls)

pb = Problem('heat', equations=eqs)

```

(continues on next page)

(continued from previous page)

```

pb.set_bcs(ebcs=Conditions([fix1, fix2]))

pb.set_solver(nls)

state = pb.solve(save_results=False)

if subs is not None:
    field.restore_dofs()

filename = os.path.join(options.output_dir, 'hanging.vtk')
ensure_path(filename)

pb.save_state(filename, state)
if options.order > 1:
    pb.save_state(filename, state, linearization=Struct(kind='adaptive',
                                                         min_level=0,
                                                         max_level=8,
                                                         eps=1e-3))

if __name__ == '__main__':
    main()

```

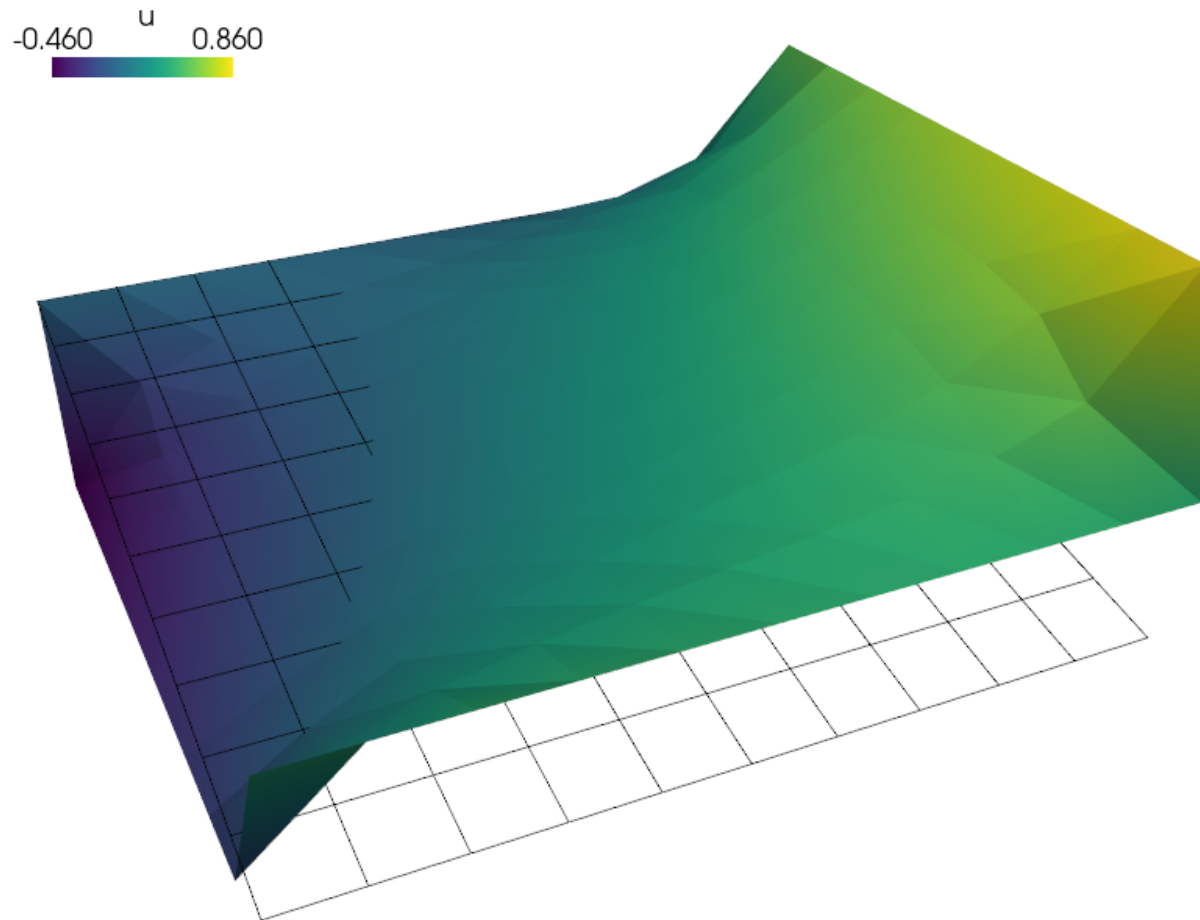
diffusion/laplace_shifted_periodic.py

Description

Laplace equation with shifted periodic BCs.

Display using:

```
sfepy-view laplace_shifted_periodic.vtk -f u:wu:f0.5 1:vw
```



source code

```
#!/usr/bin/env python
"""
Laplace equation with shifted periodic BCs.

Display using::

    sfepy-view laplace_shifted_periodic.vtk -f u:wu:f0.5 1:vw
"""
from __future__ import absolute_import
import sys
sys.path.append('.')
from argparse import ArgumentParser, RawDescriptionHelpFormatter
import numpy as nm

from sfepy.base.base import output
from sfepy.discrete import (FieldVariable, Integral, Equation, Equations,
                             Function, Problem)
from sfepy.discrete.fem import FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import (Conditions, EssentialBC,
                                       LinearCombinationBC)
```

(continues on next page)

(continued from previous page)

```

from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.mesh.mesh_generators import gen_block_mesh
import sfepy.discrete.fem.periodic as per

def run(domain, order, output_dir=''):
    omega = domain.create_region('Omega', 'all')
    bbox = domain.get_mesh_bounding_box()
    min_x, max_x = bbox[:, 0]
    min_y, max_y = bbox[:, 1]
    eps = 1e-8 * (max_x - min_x)
    gamma1 = domain.create_region('Gamma1',
                                  'vertices in (x < %.10f)' % (min_x + eps),
                                  'facet')
    gamma2 = domain.create_region('Gamma2',
                                  'vertices in (x > %.10f)' % (max_x - eps),
                                  'facet')
    gamma3 = domain.create_region('Gamma3',
                                  'vertices in y < %.10f' % (min_y + eps),
                                  'facet')
    gamma4 = domain.create_region('Gamma4',
                                  'vertices in y > %.10f' % (max_y - eps),
                                  'facet')

    field = Field.from_args('fu', nm.float64, 1, omega, approx_order=order)

    u = FieldVariable('u', 'unknown', field)
    v = FieldVariable('v', 'test', field, primary_var_name='u')

    integral = Integral('i', order=2*order)

    t1 = Term.new('dw_laplace(v, u)',
                  integral, omega, v=v, u=u)
    eq = Equation('eq', t1)
    eqs = Equations([eq])

    fix1 = EssentialBC('fix1', gamma1, {'u.0' : 0.4})
    fix2 = EssentialBC('fix2', gamma2, {'u.0' : 0.0})

    def get_shift(ts, coors, region):
        return nm.ones_like(coors[:, 0])

    dof_map_fun = Function('dof_map_fun', per.match_x_line)
    shift_fun = Function('shift_fun', get_shift)

    sper = LinearCombinationBC('sper', [gamma3, gamma4], {'u.0' : 'u.0'},
                               dof_map_fun, 'shifted_periodic',
                               arguments=(shift_fun,))

    ls = ScipyDirect({})
    nls = Newton({}, lin_solver=ls)

```

(continues on next page)

(continued from previous page)

```

pb = Problem('laplace', equations=eqs)
pb.set_output_dir(output_dir)

pb.set_bcs(ebcs=Conditions([fix1, fix2]), lcbcs=Conditions([sper]))

pb.set_solver(nls)

state = pb.solve()

return pb, state

helps = {
    'dims' :
    'dimensions of the block [default: %(default)s]',
    'centre' :
    'centre of the block [default: %(default)s]',
    'shape' :
    'numbers of vertices along each axis [default: %(default)s]',
}

def main():
    parser = ArgumentParser(description=__doc__,
                              formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--version', action='version', version='%(prog)s')
    parser.add_argument('-d', '--dims', metavar='dims',
                        action='store', dest='dims',
                        default='[1.0, 1.0]', help=helps['dims'])
    parser.add_argument('-c', '--centre', metavar='centre',
                        action='store', dest='centre',
                        default='[0.0, 0.0]', help=helps['centre'])
    parser.add_argument('-s', '--shape', metavar='shape',
                        action='store', dest='shape',
                        default='[11, 11]', help=helps['shape'])
    options = parser.parse_args()

    dims = nm.array(eval(options.dims), dtype=nm.float64)
    centre = nm.array(eval(options.centre), dtype=nm.float64)
    shape = nm.array(eval(options.shape), dtype=nm.int32)

    output('dimensions:', dims)
    output('centre:      ', centre)
    output('shape:         ', shape)

    mesh = gen_block_mesh(dims, shape, centre, name='block-fem')
    fe_domain = FEDomain('domain', mesh)

    pb, state = run(fe_domain, 1)
    pb.save_state('laplace_shifted_periodic.vtk', state)

if __name__ == '__main__':
    main()

```

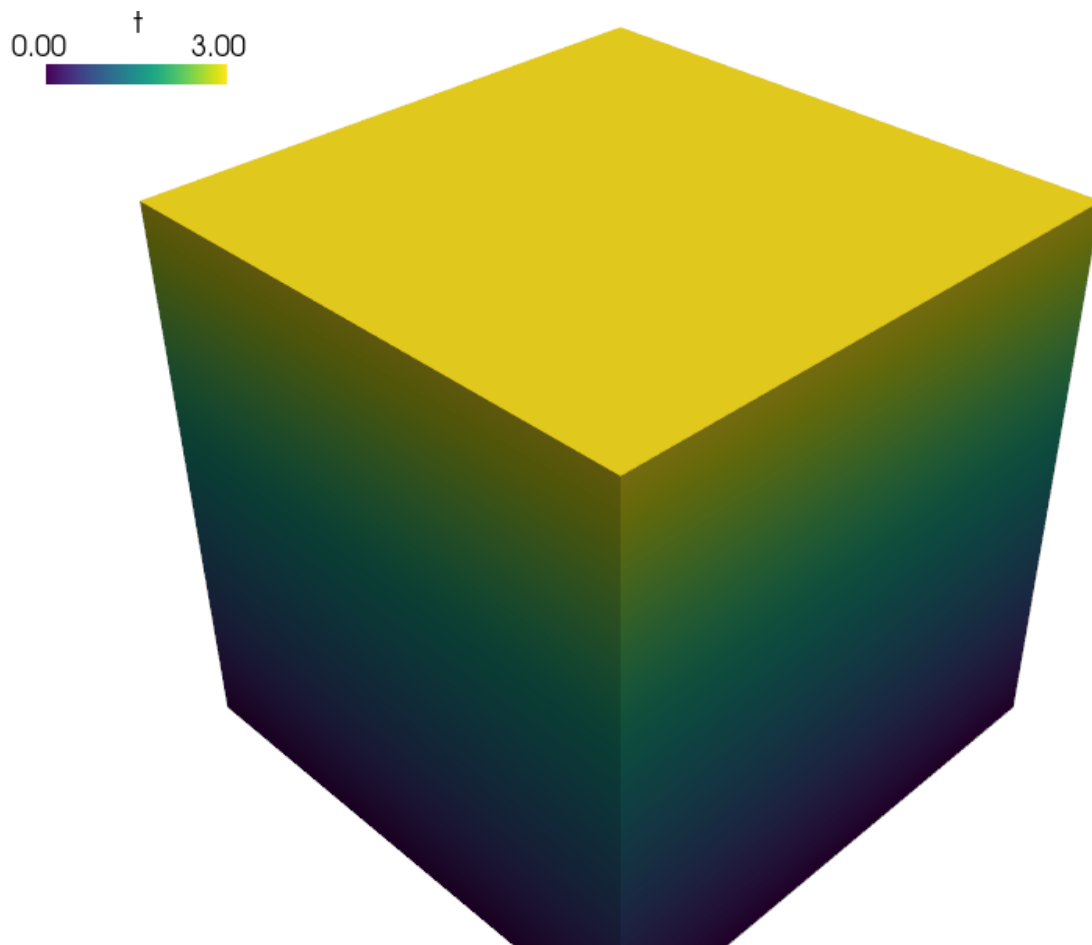
`diffusion/laplace_time_ebcs.py`**Description**

Example explaining how to change Dirichlet boundary conditions depending on time. It is shown on the stationary Laplace equation for temperature, so there is no dynamics, only the conditions change with time.

Five time steps are solved on a cube domain, with the temperature fixed to zero on the bottom face, and set to other values on the left, right and top faces in different time steps.

Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$



source code

```

r"""
Example explaining how to change Dirichlet boundary conditions depending
on time. It is shown on the stationary Laplace equation for temperature,
so there is no dynamics, only the conditions change with time.

Five time steps are solved on a cube domain, with the temperature fixed
to zero on the bottom face, and set to other values on the left, right
and top faces in different time steps.

```

(continues on next page)

(continued from previous page)

```

Find :math:`t` such that:

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    \quad \text{for all } s.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cube_medium_tetra.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',

    'active_only' : False,
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -0.499)', 'facet'),
    'Right' : ('vertices in (x > 0.499)', 'facet'),
    'Bottom' : ('vertices in (z < -0.499)', 'facet'),
    'Top' : ('vertices in (z > 0.499)', 'facet'),
}

materials = {
    'one' : ({'val' : 1.0},),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

ebcs = {
    'fixed' : ('Bottom', {'t.all' : 0}),
    't_t02' : ('Left', [(-0.5, 0.5), (2.5, 3.5)], {'t.all' : 1.0}),
    't_t1' : ('Right', [(0.5, 1.5)], {'t.all' : 2.0}),
    't_t4' : ('Top', 'is_ebc', {'t.all' : 3.0}),
}

def is_ebc(ts):
    if ts.step in (2, 4):
        return True

```

(continues on next page)

(continued from previous page)

```

    else:
        return False

functions = {
    'is_ebc' : (is_ebc,),
}

equations = {
    'eq' : """"dw_laplace.2.Omega( one.val, s, t ) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
    }),
    'ts' : ('ts.simple', {
        't0'         : 0.0,
        't1'         : 4.0,
        'dt'         : None,
        'n_step'     : 5, # has precedence over dt!

        'quasistatic' : True,
        'verbose'     : 1,
    }),
}

```

diffusion/poisson.py

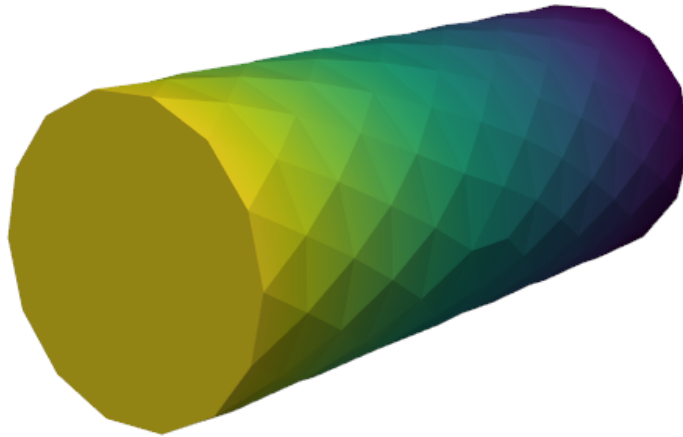
Description

Laplace equation using the long syntax of keywords.

See the tutorial section *Example Problem Description File* for a detailed explanation. See *diffusion/poisson_short_syntax.py* for the short syntax version.

Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$



source code

```
r"""
Laplace equation using the long syntax of keywords.

See the tutorial section :ref:`poisson-example-tutorial` for a detailed
explanation. See :ref:`diffusion-poisson_short_syntax` for the short syntax
version.

Find :math:`t` such that:

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    \;, \quad \forall s \;.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

material_2 = {
    'name' : 'coef',
```

(continues on next page)

(continued from previous page)

```

    'values' : {'val' : 1.0},
}

region_1000 = {
    'name' : 'Omega',
    'select' : 'cells of group 6',
}

region_03 = {
    'name' : 'Gamma_Left',
    'select' : 'vertices in (x < 0.00001)',
    'kind' : 'facet',
}

region_4 = {
    'name' : 'Gamma_Right',
    'select' : 'vertices in (x > 0.099999)',
    'kind' : 'facet',
}

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

variable_1 = {
    'name' : 't',
    'kind' : 'unknown field',
    'field' : 'temperature',
    'order' : 0, # order in the global vector of unknowns
}

variable_2 = {
    'name' : 's',
    'kind' : 'test field',
    'field' : 'temperature',
    'dual' : 't',
}

ebc_1 = {
    'name' : 't1',
    'region' : 'Gamma_Left',
    'dofs' : {'t.0' : 2.0},
}

ebc_2 = {
    'name' : 't2',
    'region' : 'Gamma_Right',
    'dofs' : {'t.0' : -2.0},
}

```

(continues on next page)

(continued from previous page)

```

}

integral_1 = {
    'name' : 'i',
    'order' : 2,
}

equations = {
    'Temperature' : ""dw_laplace.i.Omega( coef.val, s, t ) = 0""
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
    'method' : 'auto',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 1,
    'eps_a'      : 1e-10,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 1.1,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
}

options = {
    'nls' : 'newton',
    'ls'  : 'ls',
}

```

diffusion/poisson_field_dependent_material.py

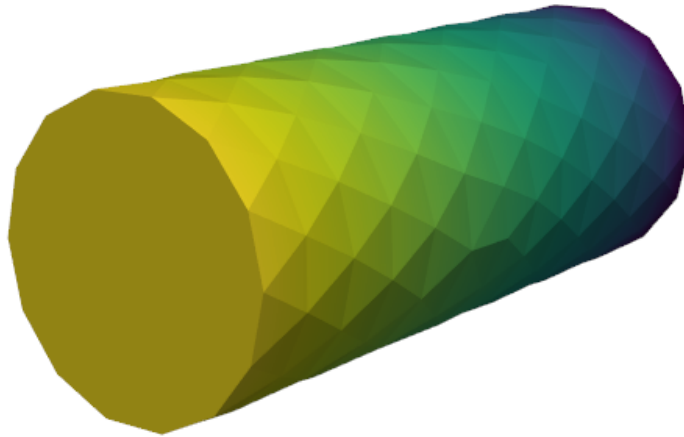
Description

Laplace equation with a field-dependent material parameter.

Find $T(t)$ for $t \in [0, t_{\text{final}}]$ such that:

$$\int_{\Omega} c(T) \nabla s \cdot \nabla T = 0, \quad \forall s.$$

where $c(T)$ is the T dependent diffusion coefficient. Each iteration calculates T and adjusts $c(T)$.



source code

```
r"""
Laplace equation with a field-dependent material parameter.

Find :math:`T(t)` for :math:`t \in [0, t_{\rm final}]` such that:

.. math::
    \int_{\Omega} c(T) \nabla s \cdot \nabla T
    = 0
    \;, \quad \forall s \;.

where :math:`c(T)` is the :math:`T` dependent diffusion coefficient.
Each iteration calculates :math:`T` and adjusts :math:`c(T)`.
```

```
"""
from __future__ import absolute_import
from sfepy import data_dir
from sfepy.base.base import output

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

t0 = 0.0
t1 = 0.1
```

(continues on next page)

(continued from previous page)

```

n_step = 11

def get_conductivity(ts, coors, problem, equations=None, mode=None, **kwargs):
    """
    Calculates the conductivity as  $2+10*T$  and returns it.
    This relation results in larger  $T$  gradients where  $T$  is small.
    """
    if mode == 'qp':
        # T-field values in quadrature points coordinates given by integral i
        # - they are the same as in `coors` argument.
        T_values = problem.evaluate('ev_integrate.i.Omega(T)',
                                    mode='qp', verbose=False)
        val = 2 + 10 * (T_values + 2)

        output('conductivity: min:', val.min(), 'max:', val.max())

        val.shape = (val.shape[0] * val.shape[1], 1, 1)
        return {'val' : val}

materials = {
    'coef' : 'get_conductivity',
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    'T' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 'T'),
}

regions = {
    'Omega' : 'all',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.099999)', 'facet'),
}

ebcs = {
    'T1' : ('Gamma_Left', {'T.0' : 2.0}),
    'T2' : ('Gamma_Right', {'T.0' : -2.0}),
}

functions = {
    'get_conductivity' : (get_conductivity,),
}

ics = {
    'ic' : ('Omega', {'T.0' : 0.0}),
}

integrals = {

```

(continues on next page)

(continued from previous page)

```

    'i' : 1,
}

equations = {
    'Temperature' : """dw_laplace.i.Omega( coef.val, s, T ) = 0""""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
    }),
    'ts' : ('ts.simple', {
        't0' : t0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step, # has precedence over dt!
        'quasistatic' : True,
        'verbose' : 1,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'save_times' : 'all',
}

```

diffusion/poisson_functions.py

Description

Poisson equation with source term.

Find u such that:

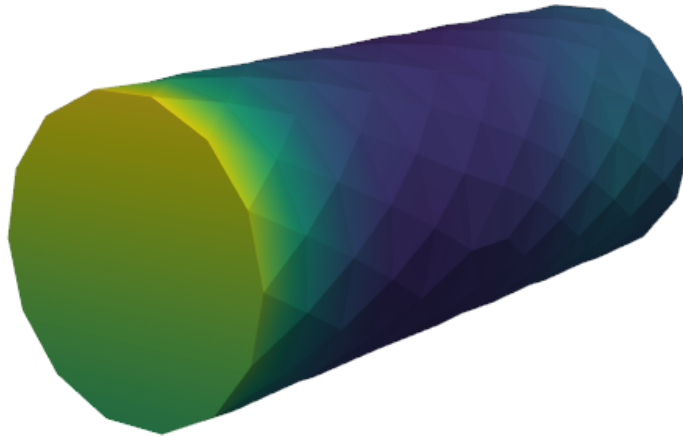
$$\int_{\Omega} c \nabla v \cdot \nabla u = - \int_{\Omega_L} b v = - \int_{\Omega_L} f v p, \quad \forall v,$$

where $b(x) = f(x)p(x)$, p is a given FE field and f is a given general function of space.

This example demonstrates use of functions for defining material parameters, regions, parameter variables or boundary conditions. Notably, it demonstrates the following:

1. How to define a material parameter by an arbitrary function - see the function `get_pars()` that evaluates $f(x)$ in quadrature points.
2. How to define a known function that belongs to a given FE space (field) - this function, $p(x)$, is defined in a FE sense by its nodal values only - see the function `get_load_variable()`.

In order to define the load $b(x)$ directly, the term `dw_dot` should be replaced by `dw_integrate`.



source code

```
r"""
Poisson equation with source term.

Find :math:`u` such that:

.. math::
    \int_{\Omega} c \nabla v \cdot \nabla u
    = - \int_{\Omega_L} b v = - \int_{\Omega_L} f v p
    \;, \quad \text{forall } v \;,

where :math:`b(x) = f(x) p(x)` , :math:`p` is a given FE field and :math:`f` is
a given general function of space.

This example demonstrates use of functions for defining material parameters,
regions, parameter variables or boundary conditions. Notably, it demonstrates
the following:

1. How to define a material parameter by an arbitrary function - see the
   function :code:`get_pars()` that evaluates :math:`f(x)` in quadrature
   points.
2. How to define a known function that belongs to a given FE space (field) -
```

(continues on next page)

(continued from previous page)

```

    this function,  $p(x)$ , is defined in a FE sense by its nodal values
    only - see the function func.get_load_variable().

In order to define the load  $b(x)$  directly, the term dw_dot
should be replaced by dw_integrate.
"""
from __future__ import absolute_import
import numpy as nm
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
}

materials = {
    'm' : ({'c' : 1.0},),
    'load' : 'get_pars',
}

regions = {
    'Omega' : 'all',
    'Omega_L' : 'vertices by get_middle_ball',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.099999)', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
    'velocity' : ('real', 'vector', 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'temperature', 0),
    'v' : ('test field', 'temperature', 'u'),
    'p' : ('parameter field', 'temperature',
          {'setter' : 'get_load_variable'}),
    'w' : ('parameter field', 'velocity',
          {'setter' : 'get_convective_velocity'}),
}

ebcs = {
    'u1' : ('Gamma_Left', {'u.0' : 'get_ebc'}),
    'u2' : ('Gamma_Right', {'u.0' : -2.0}),
}

integrals = {
    'i' : 1,
}

```

(continues on next page)

(continued from previous page)

```

equations = {
    'Laplace equation' :
        """dw_laplace.i.Omega( m.c, v, u )
        - dw_convect_v_grad_s.i.Omega( v, w, u )
        = - dw_dot.i.Omega_L( load.f, v, p )"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

def get_pars(ts, coors, mode=None, **kwargs):
    """
    Evaluate the coefficient `load.f` in quadrature points `coors` using a
    function of space.

    For scalar parameters, the shape has to be set to `(coors.shape[0], 1, 1)`.
    """
    if mode == 'qp':
        x = coors[:, 0]

        val = 55.0 * (x - 0.05)

        val.shape = (coors.shape[0], 1, 1)
        return {'f' : val}

def get_middle_ball(coors, domain=None):
    """
    Get the  $\Omega_L$  region as a function of mesh coordinates.
    """
    x, y, z = coors[:, 0], coors[:, 1], coors[:, 2]

    r1 = nm.sqrt((x - 0.025)**2.0 + y**2.0 + z**2)
    r2 = nm.sqrt((x - 0.075)**2.0 + y**2.0 + z**2)
    flag = nm.where((r1 < 2.3e-2) | (r2 < 2.3e-2))[0]

    return flag

def get_load_variable(ts, coors, region=None, variable=None, **kwargs):
    """
    Define nodal values of 'p' in the nodal coordinates `coors`.
    """
    y = coors[:, 1]

    val = 5e5 * y
    return val

def get_convective_velocity(ts, coors, **kwargs):

```

(continues on next page)

(continued from previous page)

```

"""
Define nodal values of 'w' in the nodal coordinates `coors`.
"""
val = 100.0 * nm.ones_like(coors)

return val

def get_ebc(coors, amplitude):
    """
    Define the essential boundary conditions as a function of coordinates
    `coors` of region nodes.
    """
    z = coors[:, 2]
    val = amplitude * nm.sin(z * 2.0 * nm.pi)
    return val

functions = {
    'get_pars' : (get_pars,),
    'get_load_variable' : (get_load_variable,),
    'get_convective_velocity' : (get_convective_velocity,),
    'get_middle_ball' : (get_middle_ball,),
    'get_ebc' : (lambda ts, coor, bc, problem, **kwargs: get_ebc(coor, 5.0),),
}

```

diffusion/poisson_iga.py

Description

Poisson equation solved in a single patch NURBS domain using the isogeometric analysis (IGA) approach.

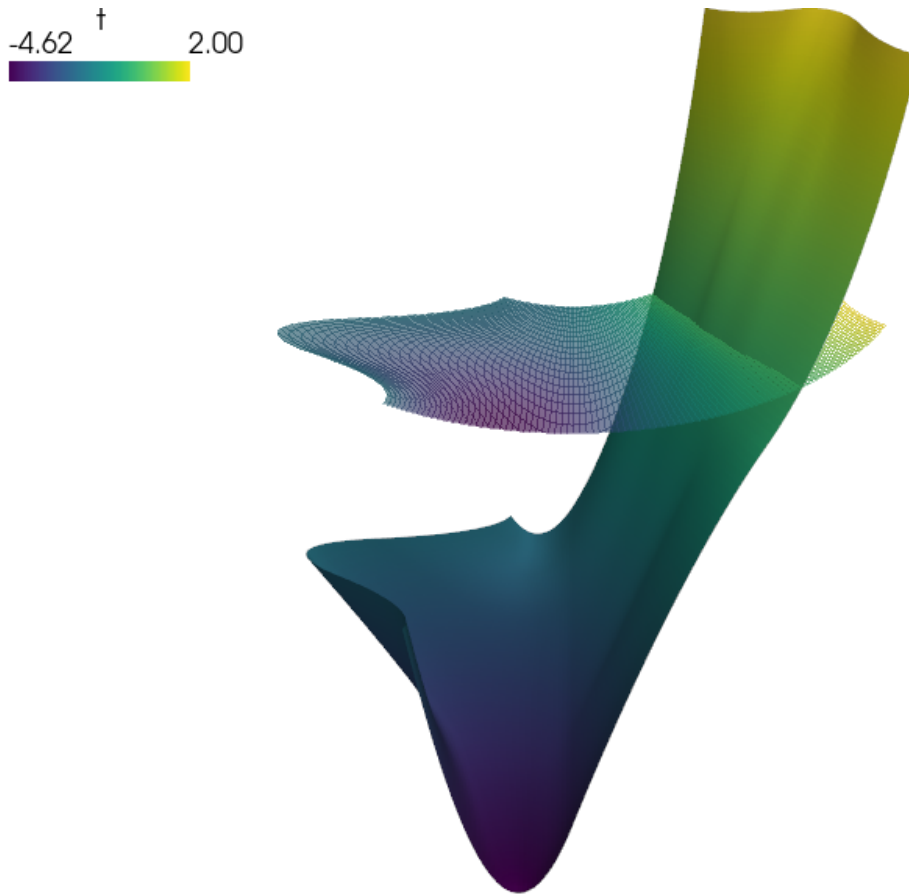
Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = \int_{\Omega_0} f s, \quad \forall s.$$

Try setting the Dirichlet boundary condition (ebcs) on various sides of the domain ('Gamma1', ..., 'Gamma4').

View the results using:

```
sfepy-view patch2d.vtk -f t:wt:f0.4 1:vw
```



source code

```

r"""
Poisson equation solved in a single patch NURBS domain using the isogeometric
analysis (IGA) approach.

Find :math:t such that:

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = \int_{\Omega_0} f s
    \;, \quad \forall s \;.

Try setting the Dirichlet boundary condition (ebcs) on various sides of the
domain ('Gamma1', ..., 'Gamma4').

View the results using::

    sfepy-view patch2d.vtk -f t:wt:f0.4 1:vw
"""
from __future__ import absolute_import
from sfepy import data_dir

```

(continues on next page)

(continued from previous page)

```

filename_domain = data_dir + '/meshes/iga/patch2d.iga'

materials = {
    'm' : ({'c' : 1.0, 'f' : -10.0},),
}

regions = {
    'Omega' : 'all',
    'Omega_0' : 'vertices in (x > 1.5)',
    'Gamma1' : ('vertices of set xi00', 'facet'),
    'Gamma2' : ('vertices of set xi01', 'facet'),
    'Gamma3' : ('vertices of set xi10', 'facet'),
    'Gamma4' : ('vertices of set xi11', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', None, 'H1', 'iga'),
}

variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

ebcs = {
    't1' : ('Gamma3', {'t.0' : 2.0}),
    't2' : ('Gamma4', {'t.0' : -2.0}),
}

integrals = {
    'i' : 3,
}

equations = {
    'Temperature' : """dw_laplace.i.Omega(m.c, s, t)
                        = dw_volume_lvf.i.Omega_0(m.f, s)"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
}

```

diffusion/poisson_neumann.py

Description

The Poisson equation with Neumann boundary conditions on a part of the boundary.

Find T such that:

$$\int_{\Omega} K_{ij} \nabla_i s \nabla_j p T = \int_{\Gamma_N} s g, \quad \forall s,$$

where g is the given flux, $g = \underline{n} \cdot K_{ij} \nabla_j \bar{T}$, and $K_{ij} = c \delta_{ij}$ (an isotropic medium). See the tutorial section [Strong form of Poisson's equation and its integration](#) for a detailed explanation.

The diffusion velocity and fluxes through various parts of the boundary are computed in the `post_process()` function. On 'Gamma_N' (the Neumann condition boundary part), the flux/length should correspond to the given value $g = -50$, while on 'Gamma_N0' the flux should be zero. Use the 'refinement_level' option (see the usage examples below) to check the convergence of the numerical solution to those values. The total flux and the flux through 'Gamma_D' (the Dirichlet condition boundary part) are shown as well.

Usage Examples

Run with the default settings (no refinement):

```
sfepy-run sfepy/examples/diffusion/poisson_neumann.py
```

Refine the mesh twice:

```
sfepy-run sfepy/examples/diffusion/poisson_neumann.py -O "'refinement_level' : 2"
```



source code

```

r"""
The Poisson equation with Neumann boundary conditions on a part of the
boundary.

Find :math:`T` such that:

.. math::
    \int_{\Omega} K_{ij} \nabla_i s \nabla_j p T
    = \int_{\Gamma_N} s g
    \quad \forall s \in V,

```

where g is the given flux, $g = \mathbf{n} \cdot \mathbf{K}_{ij} \nabla_j \bar{T}$, and $K_{ij} = c \delta_{ij}$ (an isotropic medium). See the tutorial section :ref:`poisson-weak-form-tutorial` for a detailed explanation.

The diffusion velocity and fluxes through various parts of the boundary are computed in the :func:`post\_process()` function. On Γ_N (the Neumann condition boundary part), the flux/length should correspond to the given value $g = -50$, while on Γ_0 the flux should be zero. Use the 'refinement_level' option (see the usage examples below) to check the convergence of the numerical solution to those values. The total flux and the

(continues on next page)

(continued from previous page)

flux through 'Gamma_D' (the Dirichlet condition boundary part) are shown as well.

Usage Examples

Run with the default settings (no refinement)::

```
sfePy-run sfePy/examples/diffusion/poisson_neumann.py
```

Refine the mesh twice::

```
sfePy-run sfePy/examples/diffusion/poisson_neumann.py -O "refinement_level" : 2"
"""
```

```
from __future__ import absolute_import
import numpy as nm
```

```
from sfePy.base.base import output, Struct
from sfePy import data_dir
```

```
def post_process(out, pb, state, extend=False):
```

```
    """
```

```
    Calculate :math:\nabla t and compute boundary fluxes.
```

```
    """
```

```
    dv = pb.evaluate('ev_diffusion_velocity.i.Omega(m.K, t)', mode='el_avg',
                     verbose=False)
```

```
    out['dv'] = Struct(name='output_data', mode='cell',
                       data=dv, dofs=None)
```

```
    totals = nm.zeros(3)
```

```
    for gamma in ['Gamma_N', 'Gamma_N0', 'Gamma_D']:
```

```
        flux = pb.evaluate('ev_surface_flux.i.%s(m.K, t)' % gamma,
                           verbose=False)
```

```
        area = pb.evaluate('ev_volume.i.%s(t)' % gamma, verbose=False)
```

```
        flux_data = (gamma, flux, area, flux / area)
```

```
        totals += flux_data[1:]
```

```
        output('%8s flux: % 8.3f length: % 8.3f flux/length: % 8.3f'
               % flux_data)
```

```
    totals[2] = totals[0] / totals[1]
```

```
    output('    total flux: % 8.3f length: % 8.3f flux/length: % 8.3f'
           % tuple(totals))
```

```
    return out
```

```
filename_mesh = data_dir + '/meshes/2d/cross-51-0.34.mesh'
```

```
materials = {
    'flux' : ({'val' : -50.0},),
```

(continues on next page)

(continued from previous page)

```

    'm' : ({'K' : 2.7 * nm.eye(2)}),
}

regions = {
    'Omega' : 'all',
    'Gamma_D' : ('vertices in (x < -0.4999)', 'facet'),
    'Gamma_N0' : ('vertices in (y > 0.4999)', 'facet'),
    'Gamma_N' : ('vertices of surface -s (r.Gamma_D +v r.Gamma_N0)',
                 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

ebcs = {
    't1' : ('Gamma_D', {'t.0' : 5.3}),
}

integrals = {
    'i' : 2
}

equations = {
    'Temperature' : """
        dw_diffusion.i.Omega(m.K, s, t)
        = dw_integrate.i.Gamma_N(flux.val, s)
    """
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',

    'refinement_level' : 0,
    'post_process_hook' : 'post_process',
}

```

diffusion/poisson_nonlinear_material.py**Description**

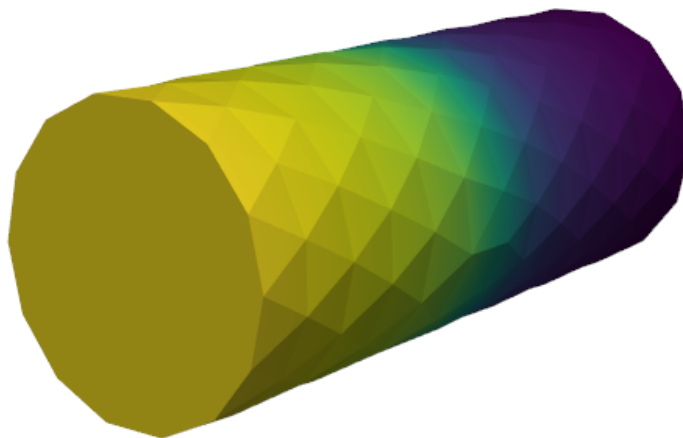
Nonlinear Poisson's equation example demonstrating the nonlinear diffusion and nonlinear volume force terms.

The example is an adaptation of: [diffusion/poisson_field_dependent_material.py](#).

Find T such that:

$$\int_{\Omega} c(T) \nabla s \cdot \nabla T + \int_{\Omega} g(T) \cdot s = 0, \quad \forall s.$$

where $c(T)$ and $g(T)$ are the T dependent coefficients.



source code

```
r"""
Nonlinear Poisson's equation example demonstrating the nonlinear diffusion
and nonlinear volume force terms.

The example is an adaptation of:
:ref:`diffusion-poisson_field_dependent_material`.

Find :math:`T` such that:
```

(continues on next page)

(continued from previous page)

```

.. math::
  \int_{\Omega} c(T) \nabla s \cdot \nabla T + \int_{\Omega} g(T) \cdot s
  = 0
  \;; \quad \forall s \;;.

where :math:`c(T)` and :math:`g(T)` are the :math:`T` dependent coefficients.
"""
from sfepy import data_dir
import numpy as np

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

def get_conductivity(T):
    """
    Calculates the conductivity as  $(1 + T^4)$  and returns it.
    """
    val = 1 + T**4
    return val

def d_get_conductivity(T):
    """
    Calculates the derivative of get_conductivity and returns it.
    """
    val = 4*T**3
    return val

def nl_src(T):
    """
    Calculates a non linear source term as  $\text{np.sinh}(T)$  and returns it.
    """
    val = np.sinh(T)
    return val

def d_nl_src(T):
    """
    Calculates the derivative of the nl_src term and returns it.
    """
    val = np.cosh(T)
    return val

materials = {
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    'T' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 'T'),
}

```

(continues on next page)

(continued from previous page)

```

regions = {
    'Omega' : 'all',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),
}

ebcs = {
    'T1' : ('Gamma_Left', {'T.0' : 2.0}),
    'T2' : ('Gamma_Right', {'T.0' : -2.0}),
}

integrals = {
    'i' : 10,
}

equations = {
    'Temperature' : """
        dw_nl_diffusion.i.Omega(get_conductivity, d_get_conductivity, s, T)
        + dw_volume_nvf.i.Omega(nl_src, d_nl_src, s, T) = 0
    """
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 15,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
        'ls_on' : 10,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'save_times' : 'all',
}

```

diffusion/poisson_parallel_interactive.py

Description

Parallel assembling and solving of a Poisson's equation, using commands for interactive use.

Find u such that:

$$\int_{\Omega} \nabla v \cdot \nabla u = \int_{\Omega} v f, \quad \forall s.$$

Important Notes

- This example requires petsc4py, mpi4py and (optionally) pymetis with their dependencies installed!
- This example generates a number of files - do not use an existing non-empty directory for the `output_dir` argument.
- Use the `--clear` option with care!

Notes

- Each task is responsible for a subdomain consisting of a set of cells (a cell region).
- Each subdomain owns PETSc DOFs within a consecutive range.
- When both global and task-local variables exist, the task-local variables have `_i` suffix.
- This example does not use a nonlinear solver.
- This example can serve as a template for solving a linear single-field scalar problem - just replace the equations in `create_local_problem()`.
- The command line options are saved into `<output_dir>/options.txt` file.

Usage Examples

See all options:

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py -h
```

See PETSc options:

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py -help
```

Single process run useful for debugging with `debug()`:

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel
```

Parallel runs:

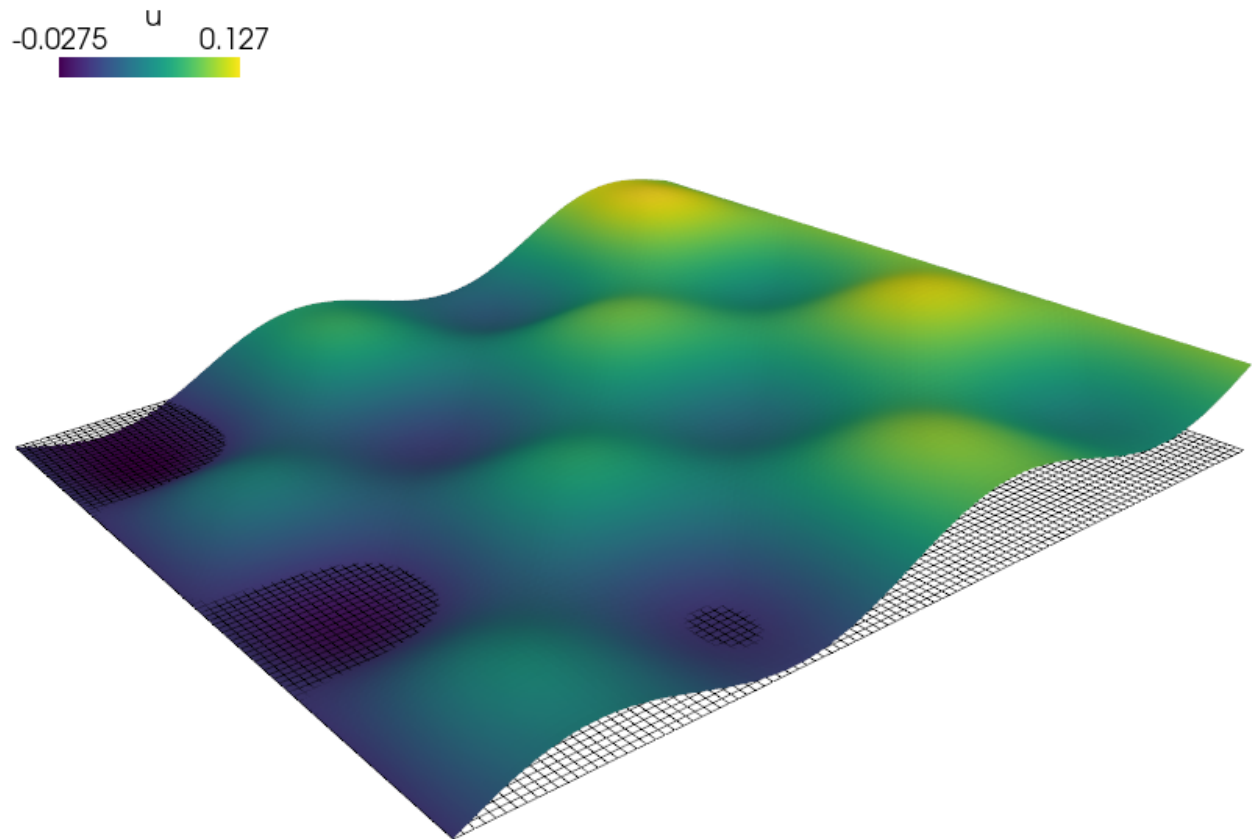
```
mpiexec -n 3 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101
```

```
mpiexec -n 3 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --metis
```

```
mpiexec -n 5 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --verify --metis -ksp_monitor -ksp_converged_reason
```

View the results using:

```
sfepy-view output-parallel/sol.h5 -f u:vu 1:vw
```



source code

```
#!/usr/bin/env python
r"""
Parallel assembling and solving of a Poisson's equation, using commands for
interactive use.

Find :math:u such that:

.. math::
    \int_{\Omega} \nabla v \cdot \nabla u
    = \int_{\Omega} v f
    \;, \quad \text{forall } s \; .

Important Notes
-----

- This example requires petsc4py, mpi4py and (optionally) pymetis with their
  dependencies installed!
- This example generates a number of files - do not use an existing non-empty
  directory for the output_dir argument.
- Use the --clear option with care!
```

(continues on next page)

(continued from previous page)

Notes

- Each task is responsible for a subdomain consisting of a set of cells (a cell region).
- Each subdomain owns PETSc DOFs within a consecutive range.
- When both global and task-local variables exist, the task-local variables have ``\_i`` suffix.
- This example does not use a nonlinear solver.
- This example can serve as a template for solving a linear single-field scalar problem - just replace the equations in `:func:`create_local_problem()``.
- The command line options are saved into `<output_dir>/options.txt` file.

Usage Examples

See all options::

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py -h
```

See PETSc options::

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py -help
```

Single process run useful for debugging with `:func:`debug()``

`<sfepy.base.base.debug>`::`

```
python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel
```

Parallel runs::

```
mpiexec -n 3 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101
```

```
mpiexec -n 3 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --metis
```

```
mpiexec -n 5 python3 sfepy/examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --verify --metis -ksp_monitor -ksp_converged_reason
```

View the results using::

```
sfepy-view output-parallel/sol.h5 -f u:vu 1:vw
"""
```

```

from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser
import os
import sys
sys.path.append('.')
import csv
import numpy as nm

```

(continues on next page)

(continued from previous page)

```

import matplotlib.pyplot as plt

from sfepy.base.base import output, Struct
from sfepy.base.ioutils import ensure_path, remove_files_patterns, save_options
from sfepy.base.timing import Timer
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.discrete.common.region import Region
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                             Equation, Equations, Problem)
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.discrete.evaluate import apply_ebc_to_matrix
from sfepy.terms import Term
from sfepy.solvers.ls import PETScKrylovSolver

import sfepy.parallel.parallel as pl
import sfepy.parallel.plot_parallel_dofs as ppd

def create_local_problem(omega_gi, order):
    """
    Local problem definition using a domain corresponding to the global region
    `omega_gi`.
    """
    mesh = omega_gi.domain.mesh

    # All tasks have the whole mesh.
    bbox = mesh.get_bounding_box()
    min_x, max_x = bbox[:, 0]
    eps_x = 1e-8 * (max_x - min_x)

    mesh_i = Mesh.from_region(omega_gi, mesh, localize=True)
    domain_i = FEDomain('domain_i', mesh_i)
    omega_i = domain_i.create_region('Omega', 'all')

    gamma1_i = domain_i.create_region('Gamma1',
                                       'vertices in (x < %.10f)'
                                       % (min_x + eps_x),
                                       'facet', allow_empty=True)
    gamma2_i = domain_i.create_region('Gamma2',
                                       'vertices in (x > %.10f)'
                                       % (max_x - eps_x),
                                       'facet', allow_empty=True)

    field_i = Field.from_args('fu', nm.float64, 1, omega_i,
                              approx_order=order)

    output('number of local field DOFs:', field_i.n_nod)

    u_i = FieldVariable('u_i', 'unknown', field_i)
    v_i = FieldVariable('v_i', 'test', field_i, primary_var_name='u_i')

    integral = Integral('i', order=2*order)

```

(continues on next page)

(continued from previous page)

```

mat = Material('m', lam=10, mu=5)
t1 = Term.new('dw_laplace(m.lam, v_i, u_i)',
              integral, omega_i, m=mat, v_i=v_i, u_i=u_i)

def _get_load(coors):
    val = nm.ones_like(coors[:, 0])
    for coor in coors.T:
        val *= nm.sin(4 * nm.pi * coor)
    return val

def get_load(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        return {'val' : _get_load(coors).reshape(coors.shape[0], 1, 1)}

load = Material('load', function=Function('get_load', get_load))

t2 = Term.new('dw_volume_lvf(load.val, v_i)',
              integral, omega_i, load=load, v_i=v_i)

eq = Equation('balance', t1 - 100 * t2)
eqs = Equations([eq])

ebc1 = EssentialBC('ebc1', gamma1_i, {'u_i.all' : 0.0})
ebc2 = EssentialBC('ebc2', gamma2_i, {'u_i.all' : 0.1})

pb = Problem('problem_i', equations=eqs, active_only=False)
pb.time_update(ebcs=Conditions([ebc1, ebc2]))
pb.update_materials()

return pb

def verify_save_dof_maps(field, cell_tasks, dof_maps, id_map, options,
                        verbose=False):
    vec = pl.verify_task_dof_maps(dof_maps, id_map, field, verbose=verbose)

    order = options.order
    mesh = field.domain.mesh

    sfield = Field.from_args('aux', nm.float64, 'scalar', field.region,
                           approx_order=order)
    aux = FieldVariable('aux', 'parameter', sfield,
                       primary_var_name='(set-to-None)')
    out = aux.create_output(vec,
                           linearization=Struct(kind='adaptive',
                                                  min_level=order-1,
                                                  max_level=order-1,
                                                  eps=1e-8))

    filename = os.path.join(options.output_dir,
                             'para-domains-dofs.h5')
    if field.is_higher_order():
        out['aux'].mesh.write(filename, out=out)

```

(continues on next page)

(continued from previous page)

```

else:
    mesh.write(filename, out=out)

out = Struct(name='cells', mode='cell',
             data=cell_tasks[:, None, None, None])
filename = os.path.join(options.output_dir,
                        'para-domains-cells.h5')
mesh.write(filename, out={'cells' : out})

def solve_problem(mesh_filename, options, comm):
    order = options.order

    rank, size = comm.Get_rank(), comm.Get_size()

    output('rank', rank, 'of', size)

    stats = Struct()
    timer = Timer('solve_timer')

    timer.start()
    mesh = Mesh.from_file(mesh_filename)
    stats.t_read_mesh = timer.stop()

    timer.start()
    if rank == 0:
        cell_tasks = pl.partition_mesh(mesh, size, use_metis=options.metis,
                                       verbose=True)

    else:
        cell_tasks = None

    stats.t_partition_mesh = timer.stop()

    output('creating global domain and field...')
    timer.start()

    domain = FEDomain('domain', mesh)
    omega = domain.create_region('Omega', 'all')
    field = Field.from_args('fu', nm.float64, 1, omega, approx_order=order)

    stats.t_create_global_fields = timer.stop()
    output('...done in', timer.dt)

    output('distributing field %s...' % field.name)
    timer.start()

    distribute = pl.distribute_fields_dofs
    lfds, gfds = distribute([field], cell_tasks,
                           is_overlap=True,
                           save_inter_regions=options.save_inter_regions,
                           output_dir=options.output_dir,

```

(continues on next page)

(continued from previous page)

```

                                comm=comm, verbose=True)

lfd = lfd[0]

stats.t_distribute_fields_dofs = timer.stop()
output('...done in', timer.dt)

if rank == 0:
    dof_maps = gfds[0].dof_maps
    id_map = gfds[0].id_map

    if options.verify:
        verify_save_dof_maps(field, cell_tasks,
                              dof_maps, id_map, options, verbose=True)

    if options.plot:
        ppd.plot_partitioning([None, None], field, cell_tasks, gfds[0],
                              options.output_dir, size)

output('creating local problem...')
timer.start()

omega_gi = Region.from_cells(lfd.cells, field.domain)
omega_gi.finalize()
omega_gi.update_shape()

pb = create_local_problem(omega_gi, order)

variables = pb.get_initial_state()
eqs = pb.equations

u_i = variables['u_i']
field_i = u_i.field

stats.t_create_local_problem = timer.stop()
output('...done in', timer.dt)

if options.plot:
    ppd.plot_local_dofs([None, None], field, field_i, omega_gi,
                        options.output_dir, rank)

output('allocating global system...')
timer.start()

sizes, drange = pl.get_sizes(lfd.petsc_dofs_range, field.n_nod, 1)
output('sizes:', sizes)
output('drange:', drange)

pdofs = pl.get_local_ordering(field_i, lfd.petsc_dofs_conn)

output('pdofs:', pdofs)

pmtx, psol, prhs = pl.create_petsc_system(pb.mtx_a, sizes, pdofs, drange,

```

(continues on next page)

(continued from previous page)

```

                                is_overlap=True, comm=comm,
                                verbose=True)

stats.t_allocate_global_system = timer.stop()
output('...done in', timer.dt)

output('evaluating local problem...')
timer.start()

variables.fill_state(0.0)
variables.apply_ebc()

rhs_i = eqs.eval_residuals(variables())
# This must be after pl.create_petsc_system() call!
mtx_i = eqs.eval_tangent_matrices(variables(), pb.mtx_a)

stats.t_evaluate_local_problem = timer.stop()
output('...done in', timer.dt)

output('assembling global system...')
timer.start()

apply_ebc_to_matrix(mtx_i, u_i.eq_map.eq_ebc)
pl.assemble_rhs_to_petsc(prhs, rhs_i, pdofs, drange, is_overlap=True,
                        comm=comm, verbose=True)
pl.assemble_mtx_to_petsc(pmtx, mtx_i, pdofs, drange, is_overlap=True,
                        comm=comm, verbose=True)

stats.t_assemble_global_system = timer.stop()
output('...done in', timer.dt)

output('creating solver...')
timer.start()

conf = Struct(method='cg', precondition='gamg', sub_precondition='none',
              i_max=10000, eps_a=1e-50, eps_r=1e-5, eps_d=1e4, verbose=True)
status = {}
ls = PETScKrylovSolver(conf, comm=comm, mtx=pmtx, status=status)

stats.t_create_solver = timer.stop()
output('...done in', timer.dt)

output('solving...')
timer.start()

psol = ls(prhs, psol)

psol_i = pl.create_local_petsc_vector(pdofs)
gather, scatter = pl.create_gather_scatter(pdofs, psol_i, psol, comm=comm)

scatter(psol_i, psol)

```

(continues on next page)

(continued from previous page)

```

sol0_i = variables() - psol_i[...]
psol_i[...] = sol0_i

gather(psol, psol_i)

stats.t_solve = timer.stop()
output('...done in', timer.dt)

output('saving solution...')
timer.start()

variables.set_state(sol0_i)
out = u_i.create_output()

filename = os.path.join(options.output_dir, 'sol_%02d.h5' % comm.rank)
pb.domain.mesh.write(filename, io='auto', out=out)

gather_to_zero = pl.create_gather_to_zero(psol)

psol_full = gather_to_zero(psol)

if comm.rank == 0:
    sol = psol_full[...].copy()[id_map]

    u = FieldVariable('u', 'parameter', field,
                      primary_var_name='(set-to-None)')

    filename = os.path.join(options.output_dir, 'sol.h5')
    if (order == 1) or (options.linearization == 'strip'):
        out = u.create_output(sol)
        mesh.write(filename, io='auto', out=out)

    else:
        out = u.create_output(sol, linearization=Struct(kind='adaptive',
                                                         min_level=0,
                                                         max_level=order,
                                                         eps=1e-3))

        out['u'].mesh.write(filename, io='auto', out=out)

stats.t_save_solution = timer.stop()
output('...done in', timer.dt)

stats.t_total = timer.total

stats.n_dof = sizes[1]
stats.n_dof_local = sizes[0]
stats.n_cell = omega.shape.n_cell
stats.n_cell_local = omega_gi.shape.n_cell

if options.show:
    plt.show()

```

(continues on next page)

(continued from previous page)

```

    return stats

def save_stats(filename, pars, stats, overwrite, rank, comm=None):
    out = stats.to_dict()
    names = sorted(out.keys())
    shape_dict = {'n%d' % ii : pars.shape[ii] for ii in range(pars.dim)}
    keys = ['size', 'rank', 'dim'] + list(shape_dict.keys()) + ['order'] + names

    out['size'] = comm.size
    out['rank'] = rank
    out['dim'] = pars.dim
    out.update(shape_dict)
    out['order'] = pars.order

    if rank == 0 and overwrite:
        with open(filename, 'w') as fd:
            writer = csv.DictWriter(fd, fieldnames=keys)
            writer.writeheader()
            writer.writerow(out)

    else:
        with open(filename, 'a') as fd:
            writer = csv.DictWriter(fd, fieldnames=keys)
            writer.writerow(out)

helps = {
    'output_dir' :
    'output directory',
    'dims' :
    'dimensions of the block [default: %(default)s]',
    'shape' :
    'shape (counts of nodes in x, y, z) of the block [default: %(default)s]',
    'centre' :
    'centre of the block [default: %(default)s]',
    '2d' :
    'generate a 2D rectangle, the third components of the above'
    ' options are ignored',
    'order' :
    'field approximation order',
    'linearization' :
    'linearization used for storing the results with approximation order > 1'
    '[default: %(default)s]',
    'metis' :
    'use metis for domain partitioning',
    'verify' :
    'verify domain partitioning, save cells and DOFs of tasks'
    ' for visualization',
    'plot' :
    'make partitioning plots',
    'save_inter_regions' :
    'save inter-task regions for debugging partitioning problems',

```

(continues on next page)

(continued from previous page)

```

'show' :
'show partitioning plots (implies --plot)',
'stats_filename' :
'name of the stats file for storing elapsed time statistics',
'new_stats' :
'create a new stats file with a header line (overwrites existing!)',
'silent' : 'do not print messages to screen',
'clear' :
'clear old solution files from output directory'
' (DANGEROUS - use with care!)',
}

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('output_dir', help=helps['output_dir'])
    parser.add_argument('--dims', metavar='dims',
                        action='store', dest='dims',
                        default='1.0,1.0,1.0', help=helps['dims'])
    parser.add_argument('--shape', metavar='shape',
                        action='store', dest='shape',
                        default='11,11,11', help=helps['shape'])
    parser.add_argument('--centre', metavar='centre',
                        action='store', dest='centre',
                        default='0.0,0.0,0.0', help=helps['centre'])
    parser.add_argument('-2', '--2d',
                        action='store_true', dest='is_2d',
                        default=False, help=helps['2d'])
    parser.add_argument('--order', metavar='int', type=int,
                        action='store', dest='order',
                        default=1, help=helps['order'])
    parser.add_argument('--linearization', choices=['strip', 'adaptive'],
                        action='store', dest='linearization',
                        default='strip', help=helps['linearization'])
    parser.add_argument('--metis',
                        action='store_true', dest='metis',
                        default=False, help=helps['metis'])
    parser.add_argument('--verify',
                        action='store_true', dest='verify',
                        default=False, help=helps['verify'])
    parser.add_argument('--plot',
                        action='store_true', dest='plot',
                        default=False, help=helps['plot'])
    parser.add_argument('--show',
                        action='store_true', dest='show',
                        default=False, help=helps['show'])
    parser.add_argument('--save-inter-regions',
                        action='store_true', dest='save_inter_regions',
                        default=False, help=helps['save_inter_regions'])
    parser.add_argument('--stats', metavar='filename',
                        action='store', dest='stats_filename',
                        default=None, help=helps['stats_filename'])

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('--new-stats',
                    action='store_true', dest='new_stats',
                    default=False, help=helps['new_stats'])
parser.add_argument('--silent',
                    action='store_true', dest='silent',
                    default=False, help=helps['silent'])
parser.add_argument('--clear',
                    action='store_true', dest='clear',
                    default=False, help=helps['clear'])
options, petsc_opts = parser.parse_known_args()

if options.show:
    options.plot = True

comm = pl.PETSc.COMM_WORLD

output_dir = options.output_dir

filename = os.path.join(output_dir, 'output_log_%02d.txt' % comm.rank)
if comm.rank == 0:
    ensure_path(filename)
comm.barrier()

output.prefix = 'sfepy_%02d:' % comm.rank
output.set_output(filename=filename, combined=options.silent == False)

output('petsc options:', petsc_opts)

mesh_filename = os.path.join(options.output_dir, 'para.h5')

dim = 2 if options.is_2d else 3
dims = nm.array(eval(options.dims), dtype=nm.float64)[:dim]
shape = nm.array(eval(options.shape), dtype=nm.int32)[:dim]
centre = nm.array(eval(options.centre), dtype=nm.float64)[:dim]
output('dimensions:', dims)
output('shape:      ', shape)
output('centre:     ', centre)

if comm.rank == 0:
    from sfepy.mesh.mesh_generators import gen_block_mesh

    if options.clear:
        remove_files_patterns(output_dir,
                              ['*.h5', '*.mesh', '*.txt', '*.png'],
                              ignores=['output_log_%02d.txt' % ii
                                      for ii in range(comm.size)],
                              verbose=True)

    save_options(os.path.join(output_dir, 'options.txt'),
                [['options', vars(options)]])

    mesh = gen_block_mesh(dims, shape, centre, name='block-fem',

```

(continues on next page)

(continued from previous page)

```

        verbose=True)
    mesh.write(mesh_filename, io='auto')

    comm.barrier()

    output('field order:', options.order)

    stats = solve_problem(mesh_filename, options, comm)
    output(stats)

    if options.stats_filename:
        if comm.rank == 0:
            ensure_path(options.stats_filename)
            comm.barrier()

        pars = Struct(dim=dim, shape=shape, order=options.order)
        pl.call_in_rank_order(
            lambda rank, comm:
                save_stats(options.stats_filename, pars, stats, options.new_stats,
                           rank, comm),
            comm
        )

if __name__ == '__main__':
    main()

```

diffusion/poisson_parametric_study.py

Description

Poisson equation.

This example demonstrates parametric study capabilities of Application classes. In particular (written in the strong form):

$$c\Delta t = f \text{ in } \Omega,$$

$$t = 2 \text{ on } \Gamma_1, t = -2 \text{ on } \Gamma_2, f = 1 \text{ in } \Omega_1, f = 0 \text{ otherwise,}$$

where Ω is a square domain, $\Omega_1 \in \Omega$ is a circular domain.

Now let's see what happens if Ω_1 diameter changes.

Run:

```
sfepy-run sfepy/examples/diffusion/poisson_parametric_study.py
```

and then look in 'output/r_omega1' directory, try for example:

```
sfepy-view output/r_omega1/circles_in_square*.vtk -2
```

Remark: this simple case could be achieved also by defining Ω_1 by a time-dependent function and solve the static problem as a time-dependent problem. However, the approach below is much more general.

Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$



source code

```
r"""
Poisson equation.

This example demonstrates parametric study capabilities of Application
classes. In particular (written in the strong form):

.. math::
    c \Delta t = f \text{ in } \Omega,

    t = 2 \text{ on } \Gamma_1 \;;,
    t = -2 \text{ on } \Gamma_2 \;;,
    f = 1 \text{ in } \Omega_1 \;;,
    f = 0 \text{ otherwise,}

where :math:`\Omega` is a square domain, :math:`\Omega_1` in :math:`\Omega` is
a circular domain.
```

(continues on next page)

(continued from previous page)

Now let's see what happens if Ω_1 diameter changes.

Run::

```
sfePy-run sfePy/examples/diffusion/poisson_parametric_study.py
```

and then look in 'output/r_omega1' directory, try for example::

```
sfePy-view output/r_omega1/circles_in_square*.vtk -2
```

Remark: this simple case could be achieved also by defining Ω_1 by a time-dependent function and solve the static problem as a time-dependent problem. However, the approach below is much more general.

Find t such that:

```
.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    \;, \quad \forall s \;.
"""
from __future__ import absolute_import
import os
import numpy as nm

from sfePy import data_dir
from sfePy.base.base import output

# Mesh.
filename_mesh = data_dir + '/meshes/2d/special/circles_in_square.vtk'

# Options. The value of 'parametric_hook' is the function that does the
# parametric study.
options = {
    'nls' : 'newton', # Nonlinear solver
    'ls' : 'ls', # Linear solver

    'parametric_hook' : 'vary_omega1_size',
    'output_dir' : 'output/r_omega1',
}

# Domain and subdomains.
default_diameter = 0.25
regions = {
    'Omega' : 'all',
    'Gamma_1' : ('vertices in (x < -0.999)', 'facet'),
    'Gamma_2' : ('vertices in (x > 0.999)', 'facet'),
    'Omega_1' : 'vertices by select_circ',
}

# FE field defines the FE approximation: 2_3_P1 = 2D, P1 on triangles.
```

(continues on next page)

(continued from previous page)

```

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

# Unknown and test functions (FE sense).
variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

# Dirichlet boundary conditions.
ebcs = {
    't1' : ('Gamma_1', {'t.0' : 2.0}),
    't2' : ('Gamma_2', {'t.0' : -2.0}),
}

# Material coefficient c and source term value f.
material_1 = {
    'name' : 'coef',
    'values' : {
        'val' : 1.0,
    }
}
material_2 = {
    'name' : 'source',
    'values' : {
        'val' : 10.0,
    }
}

# Numerical quadrature and the equation.
integral_1 = {
    'name' : 'i',
    'order' : 2,
}

equations = {
    'Poisson' : """dw_laplace.i.Omega( coef.val, s, t )
                  = dw_volume_lvf.i.Omega_1( source.val, s )"""
}

# Solvers.
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {

```

(continues on next page)

(continued from previous page)

```

'name' : 'newton',
'kind' : 'nls.newton',

'i_max'      : 1,
'eps_a'      : 1e-10,
'eps_r'      : 1.0,
'macheps'    : 1e-16,
'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
'ls_red'     : 0.1,
'ls_red_warp' : 0.001,
'ls_on'      : 1.1,
'ls_min'     : 1e-5,
'check'      : 0,
'delta'      : 1e-6,
}

functions = {
    'select_circ': (lambda coors, domain=None:
                    select_circ(coors[:,0], coors[:,1], 0, default_diameter)),
}

# Functions.
def select_circ( x, y, z, diameter ):
    """Select circular subdomain of a given diameter."""
    r = nm.sqrt( x**2 + y**2 )

    out = nm.where(r < diameter)[0]

    n = out.shape[0]
    if n <= 3:
        raise ValueError( 'too few vertices selected! (%d)' % n )

    return out

def vary_omega1_size( problem ):
    """Vary size of \Omega_1. Saves also the regions into options['output_dir'].

    Input:
        problem: Problem instance
    Return:
        a generator object:
        1. creates new (modified) problem
        2. yields the new (modified) problem and output container
        3. use the output container for some logging
        4. yields None (to signal next iteration to Application)
    """
    from sfepy.discrete import Problem
    from sfepy.solvers.ts import get_print_info

    output.prefix = 'vary_omega1_size:'

    diameters = nm.linspace( 0.1, 0.6, 7 ) + 0.001

```

(continues on next page)

(continued from previous page)

```
ofn_trunk, output_format = problem.ofn_trunk, problem.output_format
output_dir = problem.output_dir
join = os.path.join

conf = problem.conf
cf = conf.get_raw( 'functions' )
n_digit, aux, d_format = get_print_info( len( diameters ) + 1 )
for ii, diameter in enumerate( diameters ):
    output( 'iteration %d: diameter %3.2f' % (ii, diameter) )

    cf['select_circ'] = (lambda coors, domain=None:
                        select_circ(coors[:,0], coors[:,1], 0, diameter),)
    conf.edit('functions', cf)
    problem = Problem.from_conf(conf)

    problem.save_regions( join( output_dir, ('regions_' + d_format) % ii ),
                        ['Omega_1'] )
    region = problem.domain.regions['Omega_1']
    if not region.has_cells():
        raise ValueError('region %s has no cells!' % region.name)

    ofn_trunk = ofn_trunk + '_' + (d_format % ii)
    problem.setup_output(output_filename_trunk=ofn_trunk,
                        output_dir=output_dir,
                        output_format=output_format)

    out = []
    yield problem, out

    out_problem, state = out[-1]

    filename = join( output_dir,
                    ('log_%s.txt' % d_format) % ii )
    fd = open( filename, 'w' )
    log_item = '$r(\Omega_1)$: %f\n' % diameter
    fd.write( log_item )
    fd.write( 'solution:\n' )
    nm.savetxt(fd, state())
    fd.close()

    yield None
```

diffusion/poisson_periodic_boundary_condition.py**Description**

Transient Laplace equation with a localized power source and periodic boundary conditions.

This example is using a mesh generated by gmsh. Both the .geo script used by gmsh to generate the file and the .mesh file can be found in meshes.

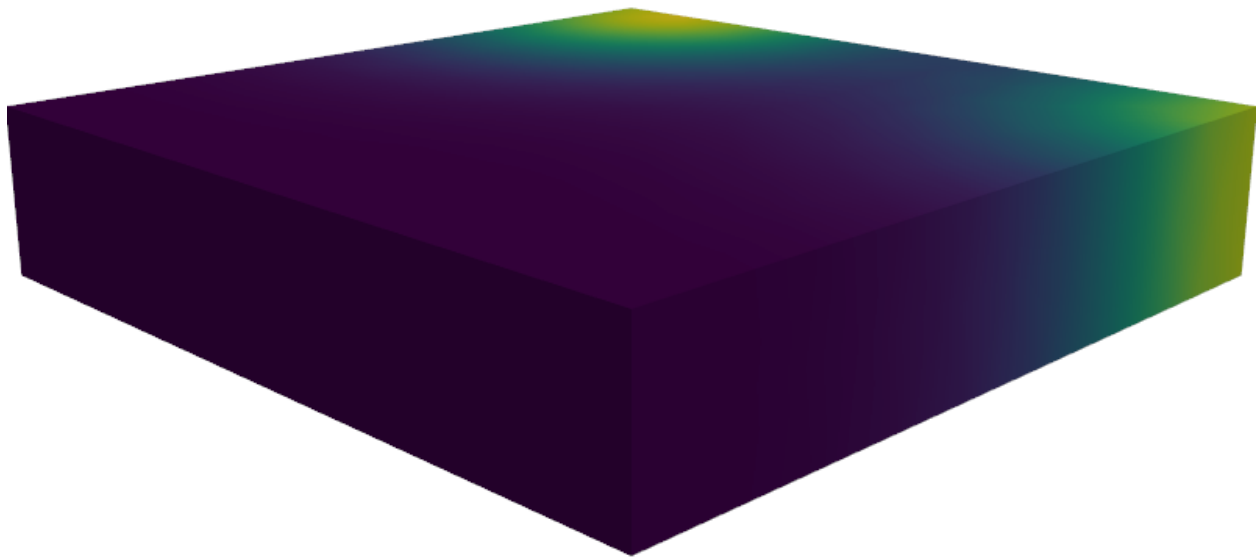
The mesh is suitable for periodic boundary conditions. It consists of a cylinder enclosed by a box in the x and y directions.

The cylinder will act as a power source.

The transient Laplace equation will be solved in time interval $t \in [0, t_{\text{final}}]$.

Find $T(t)$ for $t \in [0, t_{\text{final}}]$ such that:

$$\int_{\Omega} c s \frac{\partial T}{\partial t} + \int_{\Omega} \sigma_2 \nabla s \cdot \nabla T = \int_{\Omega_2} P_3 T, \quad \forall s.$$



source code

```
r"""
Transient Laplace equation with a localized power source and
periodic boundary conditions.
```

(continues on next page)

(continued from previous page)

This example is using a mesh generated by `gmsh`. Both the `.geo` script used by `gmsh` to generate the file and the `.mesh` file can be found in `meshes`.

The mesh is suitable for periodic boundary conditions. It consists of a cylinder enclosed by a box in the x and y directions.

The cylinder will act as a power source.

The transient Laplace equation will be solved in time interval $t \in [0, t_{\text{final}}]$.

Find $T(t)$ for $t \in [0, t_{\text{final}}]$ such that:

```
.. math::
    \int_{\Omega} c \, \frac{\partial T}{\partial t}
    + \int_{\Omega} \sigma_2 \, \nabla s \cdot \nabla T
    = \int_{\Omega_2} P_3 T
    \quad ; \quad \forall s \in V
```

```
from __future__ import absolute_import
from sfepy import data_dir
import numpy as nm
import sfepy.discrete.fem.periodic as per

filename_mesh = data_dir + '/meshes/3d/cylinder_in_box.mesh'

t0 = 0.0
t1 = 1.
n_step = 11
power_per_volume = 1.e2 # Heating power per volume of the cylinder
capacity_cylinder = 1. # Heat capacity of cylinder
capacity_fill = 1. # Heat capacity of filling material
conductivity_cylinder = 1. # Heat conductivity of cylinder
conductivity_fill = 1. # Heat conductivity of filling material

def cylinder_material_func(ts, coors, problem, mode=None, **kwargs):
    """
    Returns the thermal conductivity, the thermal mass, and the power of the
    material in the cylinder.
    """
    if mode == 'qp':
        shape = (coors.shape[0], 1, 1)

        power = nm.empty(shape, dtype=nm.float64)
        if ts.step < 5:
            # The power is turned on in the first 5 steps only.
            power.fill(power_per_volume)

        else:
```

(continues on next page)

(continued from previous page)

```

        power.fill(0.0)

        conductivity = nm.ones(shape) * conductivity_cylinder
        capacity = nm.ones(shape) * capacity_cylinder

        return {'power' : power, 'capacity' : capacity,
                'conductivity' : conductivity}

materials = {
    'cylinder' : 'cylinder_material_func',
    'fill' : ({'capacity' : capacity_fill,
               'conductivity' : conductivity_fill},),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    'T' : ('unknown field', 'temperature', 1, 1),
    's' : ('test field', 'temperature', 'T'),
}

regions = {
    'Omega' : 'all',
    'cylinder' : 'cells of group 444',
    'fill' : 'cells of group 555',
    'Gamma_Left' : ('vertices in (x < -2.4999)', 'facet'),
    'y+' : ('vertices in (y > 2.4999)', 'facet'),
    'y-' : ('vertices in (y < -2.4999)', 'facet'),
    'z+' : ('vertices in (z > 0.4999)', 'facet'),
    'z-' : ('vertices in (z < -0.4999)', 'facet'),
}

ebcs = {
    'T1' : ('Gamma_Left', {'T.0' : 0.0}),
}

# The matching functions link the elements on each side with that on the
# opposing side.
functions = {
    'cylinder_material_func' : (cylinder_material_func,),
    "match_y_plane" : (per.match_y_plane,),
    "match_z_plane" : (per.match_z_plane,),
}

epbcs = {
    # In the y-direction
    'periodic_y' : ([ 'y+', 'y-' ], {'T.0' : 'T.0'}, 'match_y_plane'),
    # and in the z-direction. Due to the symmetry of the problem, this periodic
    # boundary condition is actually not necessary, but we include it anyway.
    'periodic_z' : ([ 'z+', 'z-' ], {'T.0' : 'T.0'}, 'match_z_plane'),

```

(continues on next page)

(continued from previous page)

```
}

ics = {
    'ic' : ('Omega', {'T.0' : 0.0}),
}

integrals = {
    'i' : 1,
}

equations = {
    'Temperature' :
        """dw_dot.i.cylinder( cylinder.capacity, s, dT/dt )
           dw_dot.i.fill( fill.capacity, s, dT/dt )
           dw_laplace.i.cylinder( cylinder.conductivity, s, T )
           dw_laplace.i.fill( fill.conductivity, s, T )
           = dw_integrate.i.cylinder( cylinder.power, s )"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
    }),
    'ts' : ('ts.simple', {
        't0' : t0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step, # has precedence over dt!
        'quasistatic' : False,
        'verbose' : 1,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'output_dir' : 'output',
    'save_times' : 'all',
    'active_only' : False,
}
```

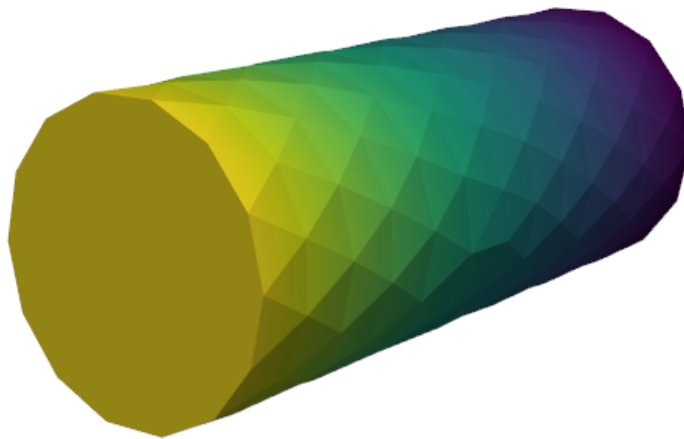
diffusion/poisson_short_syntax.py**Description**

Laplace equation using the short syntax of keywords.

See [diffusion/poisson.py](#) for the long syntax version.

Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$

**source code**

```
r"""
Laplace equation using the short syntax of keywords.

See :ref:`diffusion-poisson` for the long syntax version.

Find :math:`t` such that:

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
```

(continues on next page)

(continued from previous page)

```

    \;;, \quad \forall s \;;.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

materials = {
    'coef' : ({'val' : 1.0},),
}

regions = {
    'Omega' : 'all', # or 'cells of group 6'
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.099999)', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    't' : ('unknown field', 'temperature', 0),
    's' : ('test field', 'temperature', 't'),
}

ebcs = {
    't1' : ('Gamma_Left', {'t.0' : 2.0}),
    't2' : ('Gamma_Right', {'t.0' : -2.0}),
}

integrals = {
    'i' : 2,
}

equations = {
    'Temperature' : ""dw_laplace.i.Omega( coef.val, s, t ) = 0""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
                {'i_max' : 1,
                 'eps_a' : 1e-10,
                }),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
}

```

diffusion/sinbc.py

Description

Laplace equation with Dirichlet boundary conditions given by a sine function and constants.

Find t such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$

The `sfepy.discrete.fem.meshio.UserMeshIO` class is used to refine the original two-element mesh before the actual solution.

The FE polynomial basis and the approximation order can be chosen on the command-line. By default, the fifth order Lagrange polynomial space is used, see `define()` arguments.

This example demonstrates how to visualize higher order approximations of the continuous solution. The adaptive linearization is applied in order to save viewable results, see both the `options` keyword and the `post_process()` function that computes the solution gradient. The linearization parameters can also be specified on the command line.

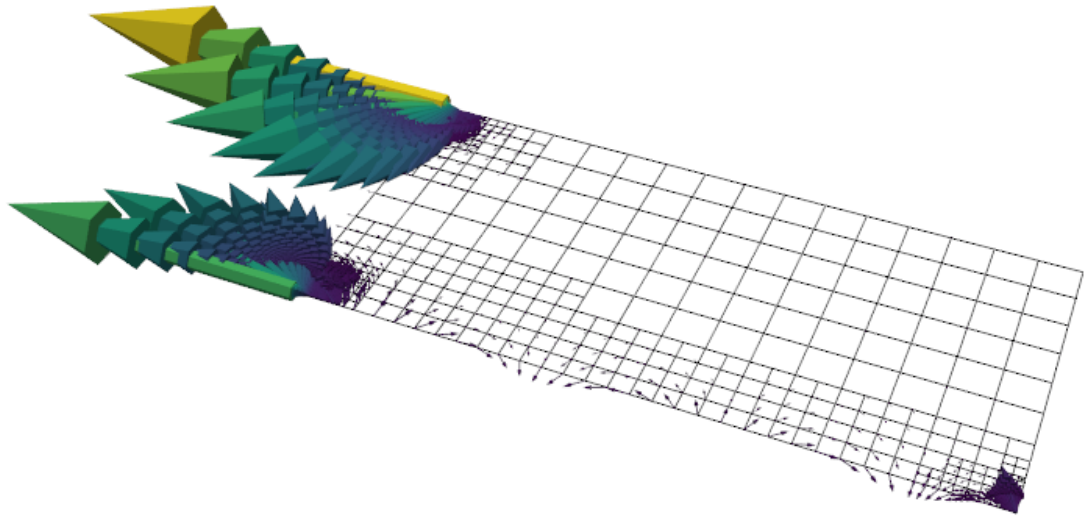
The Lagrange or Bernstein polynomial bases support higher order DOFs in the Dirichlet boundary conditions, unlike the hierarchical Lobatto basis implementation, compare the results of:

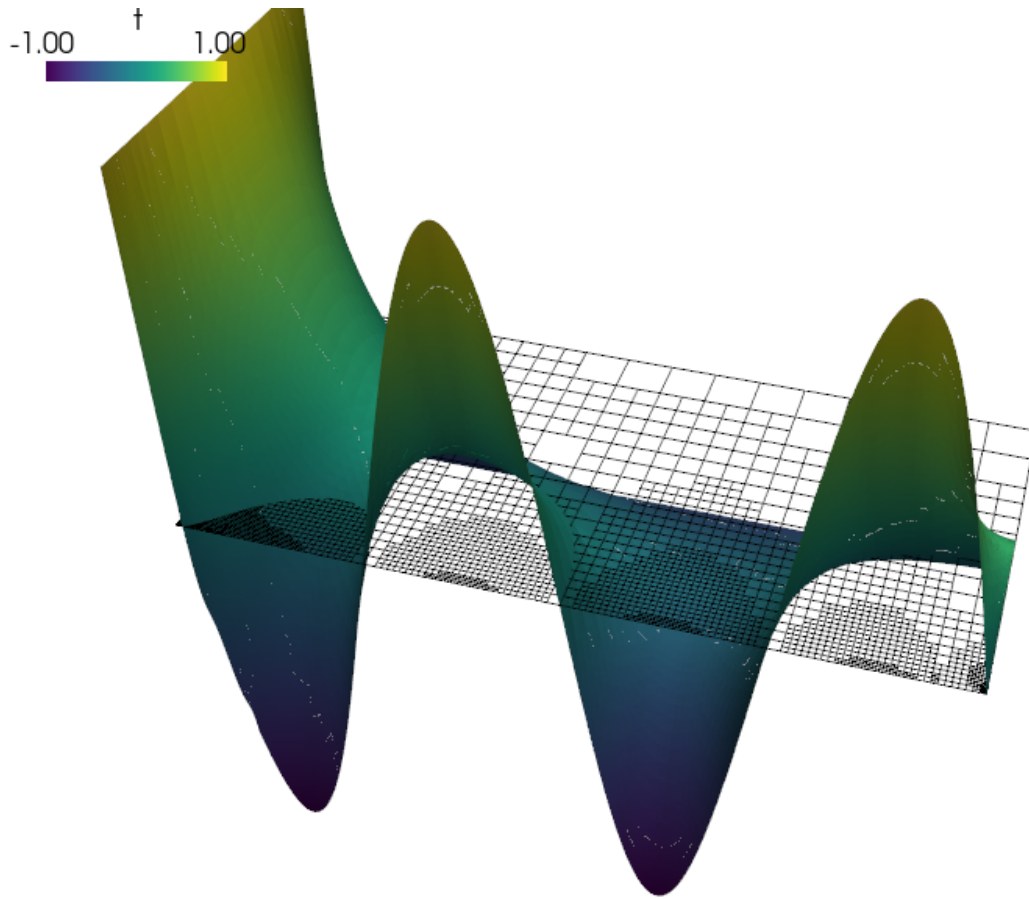
```
sfepy-run sfepy/examples/diffusion/sinbc.py -d basis=lagrange
sfepy-run sfepy/examples/diffusion/sinbc.py -d basis=bernstein
sfepy-run sfepy/examples/diffusion/sinbc.py -d basis=lobatto
```

Use the following commands to view each of the results of the above commands (assuming default output directory and names):

```
sfepy-view 2_4_2_refined_t.vtk -2 -f t:wt
sfepy-view 2_4_2_refined_grad.vtk -2
```

grad
0.0782 137.



source code

```
r"""
Laplace equation with Dirichlet boundary conditions given by a sine function
and constants.
```

Find t such that:

```
.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    \;, \quad \forall s \;.
```

The `:class:`sfepy.discrete.fem.meshio.UserMeshIO`` class is used to refine the original two-element mesh before the actual solution.

The FE polynomial basis and the approximation order can be chosen on the command-line. By default, the fifth order Lagrange polynomial space is used, see `define()` arguments.

This example demonstrates how to visualize higher order approximations of the continuous solution. The adaptive linearization is applied in order to save viewable results, see both the options keyword and the `post_process()`

(continues on next page)

(continued from previous page)

function that computes the solution gradient. The linearization parameters can also be specified on the command line.

The Lagrange or Bernstein polynomial bases support higher order DOFs in the Dirichlet boundary conditions, unlike the hierarchical Lobatto basis implementation, compare the results of::

```
sfePy-run sfePy/examples/diffusion/sinbc.py -d basis=lagrange
sfePy-run sfePy/examples/diffusion/sinbc.py -d basis=bernstein
sfePy-run sfePy/examples/diffusion/sinbc.py -d basis=lobatto
```

Use the following commands to view each of the results of the above commands (assuming default output directory and names)::

```
sfePy-view 2_4_2_refined_t.vtk -2 -f t:wt
sfePy-view 2_4_2_refined_grad.vtk -2
"""
from __future__ import absolute_import
import numpy as nm

from sfePy import data_dir

from sfePy.base.base import output
from sfePy.discrete.fem import Mesh, FEDomain
from sfePy.discrete.fem.meshio import UserMeshIO, MeshIO
from sfePy.homogenization.utils import define_box_regions
from six.moves import range

base_mesh = data_dir + '/meshes/elements/2_4_2.mesh'

def mesh_hook(mesh, mode):
    """
    Load and refine a mesh here.
    """
    if mode == 'read':
        mesh = Mesh.from_file(base_mesh)
        domain = FEDomain(mesh.name, mesh)
        for ii in range(3):
            output('refine %d...' % ii)
            domain = domain.refine()
            output('... %d nodes %d elements'
                  % (domain.shape.n_nod, domain.shape.n_el))

        domain.mesh.name = '2_4_2_refined'

        return domain.mesh

    elif mode == 'write':
        pass

def post_process(out, pb, state, extend=False):
    """
```

(continues on next page)

(continued from previous page)

```

Calculate gradient of the solution.
"""
from sfepy.discrete.fem.fields_base import create_expression_output

aux = create_expression_output('ev_grad.ie.Elements( t )',
                              'grad', 'temperature',
                              pb.fields, pb.get_materials(),
                              pb.get_variables(), functions=pb.functions,
                              mode='qp', verbose=False,
                              min_level=0, max_level=5, eps=1e-3)

out.update(aux)

return out

def define(order=5, basis='lagrange', min_level=0, max_level=5, eps=1e-3):

    filename_mesh = UserMeshIO(mesh_hook)

    # Get the mesh bounding box.
    io = MeshIO.any_from_filename(base_mesh)
    bbox, dim = io.read_bounding_box(ret_dim=True)

    options = {
        'nls' : 'newton',
        'ls' : 'ls',
        'post_process_hook' : 'post_process',
        'linearization' : {
            'kind' : 'adaptive',
            'min_level' : min_level, # Min. refinement level applied everywhere.
            'max_level' : max_level, # Max. refinement level.
            'eps' : eps, # Relative error tolerance.
        },
    }

    materials = {
        'coef' : ({'val' : 1.0},),
    }

    regions = {
        'Omega' : 'all',
    }
    regions.update(define_box_regions(dim, bbox[0], bbox[1], 1e-5))

    fields = {
        'temperature' : ('real', 1, 'Omega', order, 'H1', basis),
    }

    variables = {
        't' : ('unknown field', 'temperature', 0),
        's' : ('test field', 'temperature', 't'),
    }

```

(continues on next page)

(continued from previous page)

```

amplitude = 1.0
def ebc_sin(ts, coor, **kwargs):
    x0 = 0.5 * (coor[:, 1].min() + coor[:, 1].max())
    val = amplitude * nm.sin( (coor[:, 1] - x0) * 2. * nm.pi )
    return val

ebcs = {
    't1' : ('Left', {'t.0' : 'ebc_sin'}),
    't2' : ('Right', {'t.0' : -0.5}),
    't3' : ('Top', {'t.0' : 1.0}),
}

functions = {
    'ebc_sin' : (ebc_sin,),
}

equations = {
    'Temperature' : """dw_laplace.10.Omega(coef.val, s, t) = 0"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

return locals()

```

diffusion/time_advection_diffusion.py

Description

The transient advection-diffusion equation with a given divergence-free advection velocity.

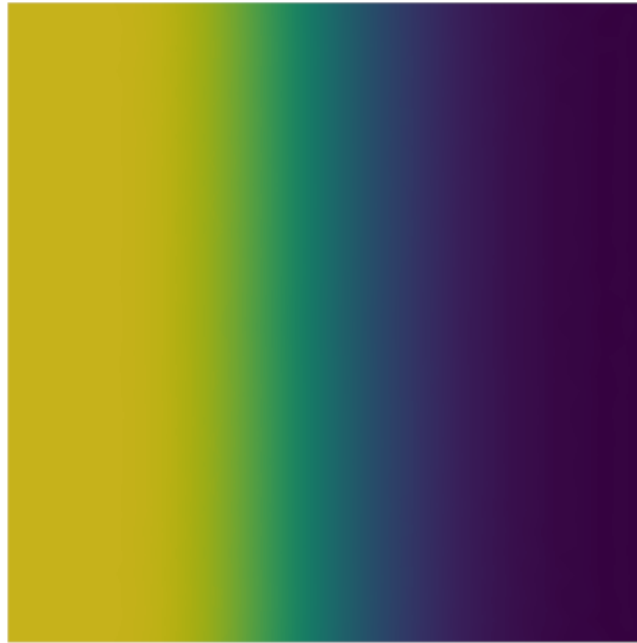
Find u such that:

$$\int_{\Omega} s \frac{\partial u}{\partial t} + \int_{\Omega} s \nabla \cdot (\underline{v} u) + \int_{\Omega} D \nabla s \cdot \nabla u = 0, \quad \forall s.$$

View the results using:

```
sfepy-view square_tri2.*.vtk -f u:wu 1:vw
```

0.00 u 2.00



source code

```

r"""
The transient advection-diffusion equation with a given divergence-free
advection velocity.

Find  $u$  such that:

.. math::
    \int_{\Omega} s \, \text{pdiff}\{u\}_t
    + \int_{\Omega} s \, \text{nabla} \cdot \left( \text{ul}\{v\} u \right)
    + \int_{\Omega} D \, \text{nabla} s \cdot \text{nabla} u
    = 0
    \;, \quad \text{forall } s \; .

View the results using::

    sfepy-view square_tri2.*.vtk -f u:wu 1:vw
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/2d/square_tri2.mesh'

```

(continues on next page)

(continued from previous page)

```

regions = {
    'Omega' : 'all', # or 'cells of group 6'
    'Gamma_Left' : ('vertices in (x < -0.99999)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),
}

fields = {
    'concentration' : ('real', 1, 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'concentration', 0, 1),
    's' : ('test field', 'concentration', 'u'),
}

ebcs = {
    'u1' : ('Gamma_Left', {'u.0' : 2.0}),
    'u2' : ('Gamma_Right', {'u.0' : 0.0}),
}

# Units: D: 0.0001 m^2 / day, v: [0.1, 0] m / day -> time in days.
materials = {
    'm' : ({'D' : 0.0001, 'v' : [[0.1], [0.0]]}),
}

integrals = {
    'i' : 2,
}

equations = {
    'advection-diffusion' :
        """
        dw_dot.i.Omega(s, du/dt)
        + dw_advect_div_free.i.Omega(m.v, s, u)
        + dw_laplace.i.Omega(m.D, s, u)
        = 0
        """
}

solvers = {
    'ts' : ('ts.simple', {
        't0' : 0.0,
        't1' : 10.0,
        'dt' : None,
        'n_step' : 11, # Has precedence over dt.
        'verbose' : 1,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

```

(continues on next page)

(continued from previous page)

```

    'ls' : ('ls.scipy_direct', {}),
}

options = {
    'ts' : 'ts',
    'nls' : 'newton',
    'ls' : 'ls',
    'save_times' : 'all',
}

```

diffusion/time_heat_equation_multi_material.py

Description

Transient heat equation with time-dependent source term, three different material domains and Newton type boundary condition loss term.

Description

This example is inspired by the Laser Powder Bed Fusion additive manufacturing process. A laser source deposits a flux on a circular surface (a full layer of the cylinder being built) at regular time intervals. The heat propagates into the build plate and to the sides of the cylinder into powder which is relatively bad at conducting heat. Heat losses through Newton type heat exchange occur both at the surface where the heat is being deposited and at the bottom plate.

The PDE for this physical process implies to find $T(x, t)$ for $x \in \Omega, t \in [0, t_{\text{final}}]$ such that:

$$\left\{ \begin{array}{ll} \rho(x)c_p(x)\frac{\partial T}{\partial t}(x, t) = -\nabla \cdot (-k(x)\nabla T(x, t)) & \forall x \in \Omega, \forall t \in [0, T_{\text{max}}] \\ -k\nabla T(x, t) \cdot \underline{n} = q(t) + h(T - T_{\infty}) & \forall x \in \Gamma_{\text{source}} \\ -k\nabla T(x, t) \cdot \underline{n} = q(t) + h(T - T_{\infty}) & \forall x \in \Gamma_{\text{plate}} \\ \nabla T(x, t) \cdot \underline{n} = 0 & \forall x \in \Gamma \setminus (\Gamma_{\text{source}} \cup \Gamma_{\text{plate}}) \end{array} \right.$$

The weak formulation solved using *sfePy* is to find a discretized field T that satisfies:

$$\begin{aligned} & \int_{\Omega} \rho c_p \frac{\partial T}{\partial t}(x, t) s + \int_{\Omega} \nabla s \cdot (k(x)\nabla T) = \int_{\Gamma} -k\nabla T \cdot \underline{n} s \\ & = \int_{\Gamma_{\text{source}}} q(t) + \int_{\Gamma_{\text{source}}} h(T - T_{\infty}) + \int_{\Gamma_{\text{plate}}} h(T - T_{\infty}) \quad \forall s \end{aligned}$$

Uniform initial conditions are used as a starting point of the simulation.

Usage examples

The script can be run with:

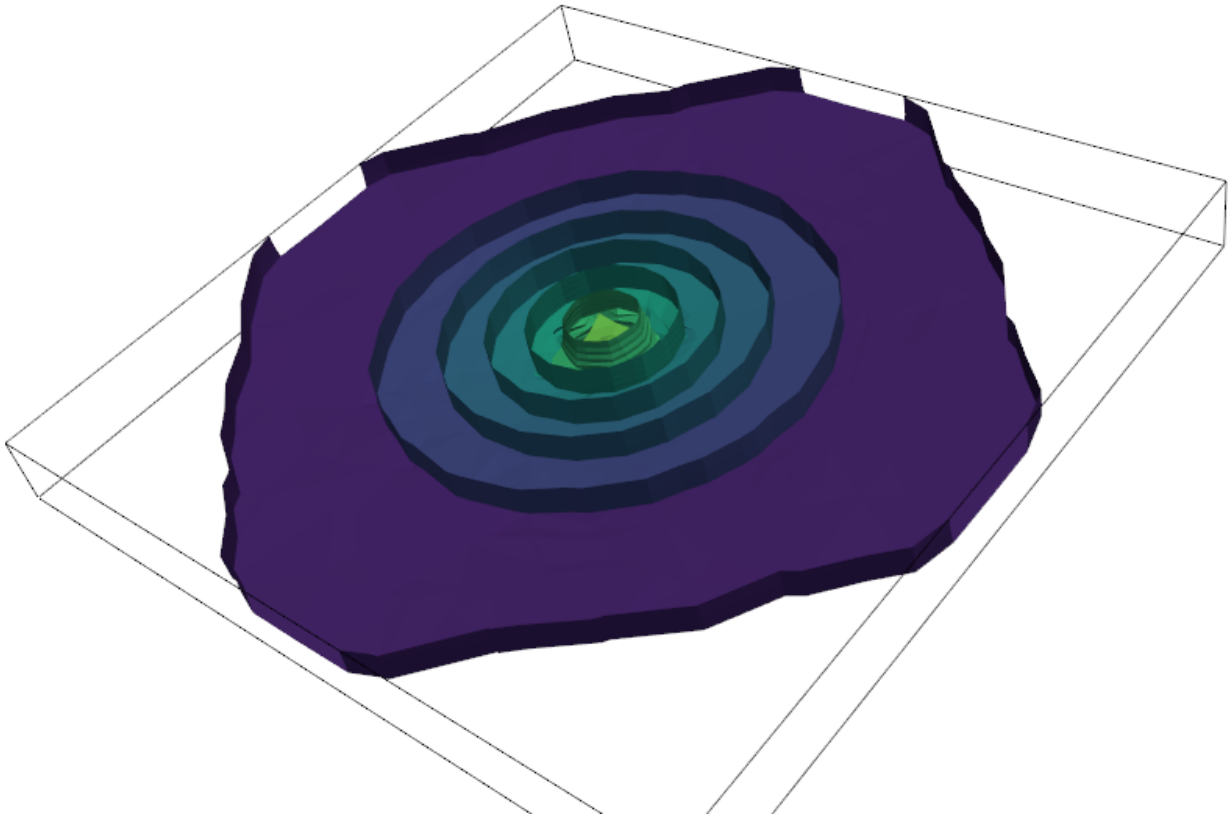
```
sfePy-run sfePy/examples/diffusion/time_heat_equation_multi_material.py
```

The last time-step result field can then be visualized as isosurfaces with:

```
sfePy-view multi_material_cylinder_plate.119.vtk -i 10 -l
```

The resulting time evolution of temperatures is saved as an image file in the output directory (heat_probe_time_evolution.png).

This script uses SI units (meters, kilograms, Joules...) except for temperature, which is expressed in degrees Celsius.



source code

```
r"""
Transient heat equation with time-dependent source term, three different
material domains and Newton type boundary condition loss term.

Description
-----

This example is inspired by the Laser Powder Bed Fusion additive manufacturing
process. A laser source deposits a flux on a circular surface (a full layer of
the cylinder being built) at regular time intervals. The heat propagates into
the build plate and to the sides of the cylinder into powder which is
relatively bad at conducting heat. Heat losses through Newton type heat
exchange occur both at the surface where the heat is being deposited and at the
bottom plate.

The PDE for this physical process implies to find  $T(x, t)$  for  $x$ 
 $\in \Omega$ ,  $t \in [0, t_{\rm final}]$  such that:
```

(continues on next page)

(continued from previous page)

```

.. math::
    \left\{ \begin{aligned}
        & \rho(x) c_p(x) \frac{\partial T}{\partial t}(x, t) \\
        &= -\nabla \cdot \left( -k(x) \underline{\nabla} T(x, t) \right) \quad \& \\
        & \text{for all } x \in \Omega, \text{ for all } t \in [0, T_{\max}] \quad \& \\
        & -k \underline{\nabla} T(x, t) \cdot \underline{n} \\
        &= q(t) + h(T - T_{\infty}) \quad \& \text{for all } x \in \Gamma_{\text{source}} \quad \& \\
        & -k \underline{\nabla} T(x, t) \cdot \underline{n} \\
        &= q(t) + h(T - T_{\infty}) \quad \& \text{for all } x \in \Gamma_{\text{plate}} \quad \& \\
        & \underline{\nabla} T(x, t) \cdot \underline{n} \\
        &= 0 \quad \& \text{for all } x \in \Gamma \setminus (\Gamma_{\text{source}} \cup \Gamma_{\text{plate}}) \\
    \end{aligned} \right.

```

The weak formulation solved using `sfepy` is to find a discretized field T that satisfies:

```

.. math::
    \int_{\Omega} \rho c_p \frac{\partial T}{\partial t} \, dx + \int_{\Omega} \underline{\nabla} s \cdot (k(x) \underline{\nabla} T) \, dx = \int_{\Gamma_{\text{source}}} q(t) \, ds + \int_{\Gamma_{\text{source}}} h(T - T_{\infty}) \, ds + \int_{\Gamma_{\text{plate}}} h(T - T_{\infty}) \, ds \quad \& \text{for all } s

```

Uniform initial conditions are used as a starting point of the simulation.

Usage examples

The script can be run with::

```
sfepy-run sfepy/examples/diffusion/time_heat_equation_multi_material.py
```

The last time-step result field can then be visualized as isosurfaces with::

```
sfepy-view multi_material_cylinder_plate.119.vtk -i 10 -l
```

The resulting time evolution of temperatures is saved as an image file in the output directory (heat_probe_time_evolution.png).

This script uses SI units (meters, kilograms, Joules...) except for temperature, which is expressed in degrees Celsius.

```

"""
import numpy as nm
from sfepy import data_dir
from sfepy.discrete.probes import LineProbe

```

(continues on next page)

(continued from previous page)

```

import matplotlib.pyplot as plt
import os

nominal_heat_flux = 6.36e5
alpha = 0.25
t_start = nm.array([0., 20., 40.]) # times when heating starts (seconds)
t_stop = nm.array([10., 30., 50.]) # times when heating stops (seconds)

T0 = 25. # °C
h = 50. # W/m2/K
nm = 1e-3

filename_mesh = data_dir + '/meshes/3d/multi_material_cylinder_plate.vtk'

materials = {
    'powder': ({'lam': 0.16,
                'rho_cp': 1330. * 650.},),
    'cylinder': ({'lam': 153.,
                  'rho_cp': 2760. * 935.,
                  'lam_vec': 173. * nm.eye(3)}),),
    'plate': ({'lam': 153.,
               'rho_cp': 2660. * 927.,
               'lam_vec': 153. * nm.eye(3)}),),

    'heat_flux_defined_by_func': 'get_flux_value',
    'heat_loss': ({'h_bot': -h, 'T_bot_inf': T0,
                   'h_top': -h, 'T_top_inf': T0},),
}

regions = {
    'Omega': 'all',
    'Omega_Cylinder': 'cells of group 4',
    'Omega_Powder': 'cells of group 3',
    'Omega_Plate': 'cells of group 1 +v cells of group 2',
    'Gamma_Plate': ('vertices in (z < -9.95e-3)', 'facet'),
    'Gamma_Source': ('vertices of surface *v r.Omega_Cylinder', 'face'),
}

def get_flux_value(ts, coors, mode=None, **kwargs):
    """Defines heat flux as a function of time."""
    if mode == 'qp':
        shape = (coors.shape[0], 1, 1)
        if nm.any((ts.time >= t_start) & (ts.time <= t_stop)):
            flux = alpha * nominal_heat_flux
        else:
            flux = 0.
        val = flux * nm.ones(shape, dtype=nm.float64)
        return {'val': val}

functions = {

```

(continues on next page)

(continued from previous page)

```

    'get_flux_value': (get_flux_value,),
}

fields = {
    'temperature': ('real', 1, 'Omega', 1),
}

variables = {
    'T': ('unknown field', 'temperature', 1, 1), # 1 means history=1
    's': ('test field', 'temperature', 'T'),
}

ebcs = {
}

integrals = {
    'i': 2
}

equations = {
    'Temperature': """
        dw_dot.i.Omega_Cylinder(cylinder.rho_cp, s, dT/dt )
    + dw_dot.i.Omega_Plate(plate.rho_cp, s, dT/dt )
    + dw_dot.i.Omega_Powder(powder.rho_cp, s, dT/dt )
    + dw_laplace.i.Omega_Cylinder(cylinder.lam, s, T)
    + dw_laplace.i.Omega_Plate(plate.lam, s, T)
    + dw_laplace.i.Omega_Powder(powder.lam, s, T)
    = dw_integrate.i.Gamma_Source(heat_flux_defined_by_func.val, s)
    + dw_bc_newton.i.Gamma_Source(heat_loss.h_top, heat_loss.T_top_inf, s, T)
    + dw_bc_newton.i.Gamma_Plate(heat_loss.h_bot, heat_loss.T_bot_inf, s, T)
    """
}

ics = {
    'ic1': ('Omega_Powder', {'T.0': T0}),
    'ic2': ('Omega_Plate', {'T.0': T0}),
    'ic3': ('Omega_Cylinder', {'T.0': T0}),
}

def gen_probe():
    """Instantiates a line probe used later by the `step_hook` function."""
    p0, p1 = nm.array([0., 0., -10. * mm]), nm.array([0.0, 0.0, 15. * mm])
    line_probe = LineProbe(p0, p1, n_point=100, share_geometry=True)
    return line_probe

line_probe = gen_probe()
# inits an empty list that will hold the probe results
probe_results = []

```

(continues on next page)

(continued from previous page)

```

def step_hook(pb, ts, variables):
    """
    This implements a function that gets called at every step from the
    time-solver.
    """
    T_field = pb.get_variables()['T']
    pars, vals = line_probe(T_field)
    probe_results.append(vals)

def post_process_hook(out, pb, state, extend=False):
    ts = pb.ts
    if ts.step == ts.n_step - 1:
        fig, (ax1, ax2) = plt.subplots(nrows=2)
        temperature_image = nm.array(probe_results).squeeze()
        m = ax1.imshow(temperature_image.T, origin='lower', aspect='auto')
        ax1.set_xlabel("time step")
        ax1.set_ylabel("distance across build\nplate and cylinder")
        fig.colorbar(m, ax=ax1, label="temperature")
        ax2.plot(temperature_image.T[0], label="bottom")
        ax2.plot(temperature_image.T[-1], label="top")
        ax2.set_xlabel("time step")
        ax2.set_ylabel("temperature (°C)")
        ax2.legend()
        fig.tight_layout()
        fig.savefig(os.path.join(pb.output_dir, 'heat_probe_time_evolution.png'),
                    bbox_inches='tight')

    return out

solvers = {
    'ls': ('ls.auto_direct', {
        # Reuse the factorized linear system from the first time step.
        'use_presolve': True,
        # Speed up the above by omitting the matrix digest check used normally
        # for verification that the current matrix corresponds to the
        # factorized matrix stored in the solver instance. Use with care!
        'use_mtx_digest': False,
    }),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-5,
        'is_linear': True,
    }),
    'ts': ('ts.simple', {
        't0': 0.0,
        't1': 60.,
        'dt': None,
        'n_step': 120,
        'verbose': True,
        'is_linear': True,
    }),
}

```

(continues on next page)

(continued from previous page)

```

}

options = {
    'step_hook': 'step_hook',
    'post_process_hook': 'post_process_hook',
}

```

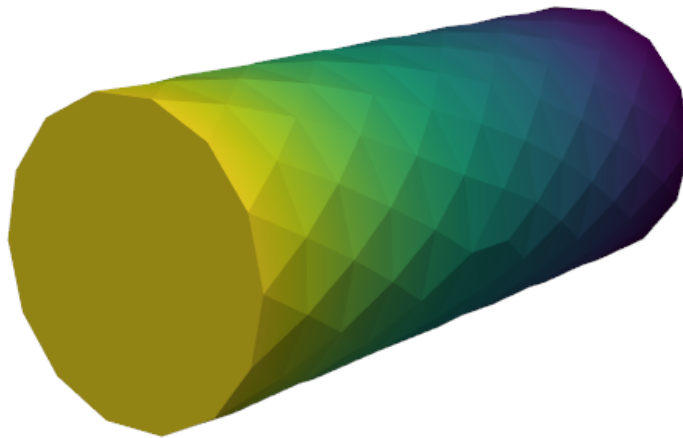
diffusion/time_poisson.py

Description

Transient Laplace equation with non-constant initial conditions given by a function.

Find $T(t)$ for $t \in [0, t_{\text{final}}]$ such that:

$$\int_{\Omega} s \frac{\partial T}{\partial t} + \int_{\Omega} c \nabla s \cdot \nabla T = 0, \quad \forall s.$$



source code

```

r"""
Transient Laplace equation with non-constant initial conditions given by a

```

(continues on next page)

(continued from previous page)

```

function.

Find  $T(t)$  for  $t \in [0, t_{\rm final}]$  such that:

.. math::
    \int_{\Omega} s \, \text{pdiff}\{T\}\{t\}
    + \int_{\Omega} c \, \nabla s \cdot \nabla T
    = 0
    \quad \text{for all } s \text{ ;}.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

t0 = 0.0
t1 = 0.1
n_step = 11

material_2 = {
    'name' : 'coef',
    'values' : {'val' : 0.01},
    'kind' : 'stationary', # 'stationary' or 'time-dependent'
}

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

variable_1 = {
    'name' : 'T',
    'kind' : 'unknown field',
    'field' : 'temperature',
    'order' : 0,
    'history' : 1,
}

variable_2 = {
    'name' : 's',
    'kind' : 'test field',
    'field' : 'temperature',
    'dual' : 'T',
}

regions = {
    'Omega' : 'all',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),
}

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    'T1': ('Gamma_Left', {'T.0' : 2.0}),
    'T2': ('Gamma_Right', {'T.0' : -2.0}),
}

def get_ic(coor, ic):
    """Non-constant initial condition."""
    import numpy as nm
    # Normalize x coordinate.
    mi, ma = coor[:,0].min(), coor[:,0].max()
    nx = (coor[:,0] - mi) / (ma - mi)
    return nm.where( (nx > 0.25) & (nx < 0.75 ), 8.0 * (nx - 0.5), 0.0 )

functions = {
    'get_ic' : (get_ic,),
}

ics = {
    'ic' : ('Omega', {'T.0' : 'get_ic'}),
}

integral_1 = {
    'name' : 'i',
    'order' : 1,
}

equations = {
    'Temperature' :
        """dw_dot.i.Omega( s, dT/dt )
        + dw_laplace.i.Omega( coef.val, s, T ) = 0"""
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
    'use_presolve' : True,
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 1,
    'eps_a'      : 1e-10,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 1.1,
    'ls_min'     : 1e-5,

```

(continues on next page)

(continued from previous page)

```
'check'      : 0,
'delta'      : 1e-6,
'is_linear'   : True,
}

solver_2 = {
    'name' : 'ts',
    'kind' : 'ts.simple',

    't0'    : t0,
    't1'    : t1,
    'dt'     : None,
    'n_step' : n_step, # has precedence over dt!
    'verbose' : 1,
}

options = {
    'nls' : 'newton',
    'ls'  : 'ls',
    'ts'  : 'ts',
    'save_times' : 'all',
}
```

diffusion/time_poisson_explicit.py

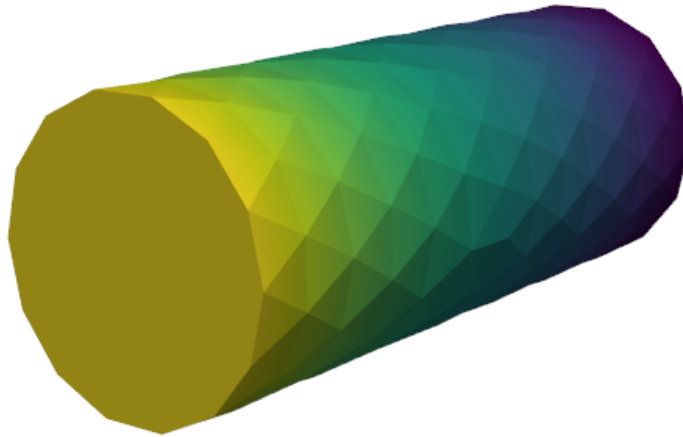
Description

Transient Laplace equation.

The same example as time_poisson.py, but using the short syntax of keywords, and explicit time-stepping.

Find $T(t)$ for $t \in [0, t_{\text{final}}]$ such that:

$$\int_{\Omega} s \frac{\partial T}{\partial t} + \int_{\Omega} c \nabla s \cdot \nabla T = 0, \quad \forall s.$$



source code

```

r"""
Transient Laplace equation.

The same example as time_poisson.py, but using the short syntax of keywords,
and explicit time-stepping.

Find  $T(t)$  for  $t \in [0, t_{\rm final}]$  such that:

.. math::
    \int_{\Omega} s \, \text{pdiff}\{T\}\{t\}
    + \int_{\Omega} c \, \nabla s \cdot \nabla T
    = 0
    \;, \quad \text{forall } s \; ;.
"""
from __future__ import absolute_import
from sfepy import data_dir

from sfepy.examples.diffusion.time_poisson import get_ic

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

```

(continues on next page)

(continued from previous page)

```

materials = {
    'coef' : ({'val' : 0.01},),
}

regions = {
    'Omega' : 'all',
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}

variables = {
    'T' : ('unknown field', 'temperature', 0, 1),
    's' : ('test field', 'temperature', 'T'),
}

ebcs = {
    't1' : ('Gamma_Left', {'T.0' : 2.0}),
    't2' : ('Gamma_Right', {'T.0' : -2.0}),
}

ics = {
    'ic' : ('Omega', {'T.0' : 'get_ic'}),
}

functions = {
    'get_ic' : (get_ic,),
}

integrals = {
    'i' : 1,
}

equations = {
    'Temperature' :
        """dw_dot.i.Omega( s, dT/dt )
        + dw_laplace.i.Omega( coef.val, s, T[-1] ) = 0"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'is_linear' : True,
    }),
    'ts' : ('ts.simple', {
        't0' : 0.0,
        't1' : 0.07,
        'dt' : 0.00002,
    })
}

```

(continues on next page)

(continued from previous page)

```
        'n_step' : None,
        'verbose' : 1,
    }},
}

options = {
    'ls' : 'ls',
    'ts' : 'ts',
    'save_times' : 100,
    'output_format' : 'h5',
}
```

diffusion/time_poisson_interactive.py

Description

Transient Laplace equation (heat equation) with non-constant initial conditions given by a function, using commands for interactive use.

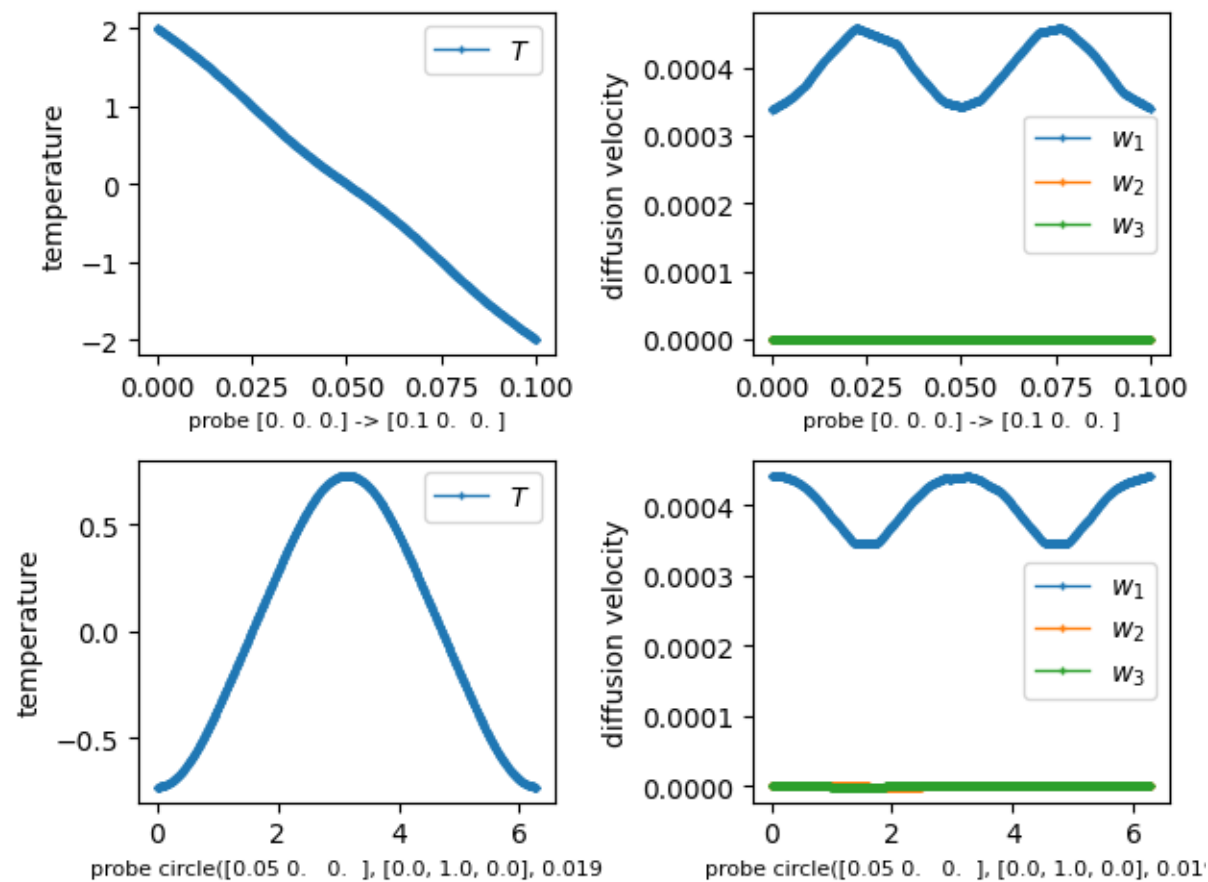
The script allows setting various simulation parameters, namely:

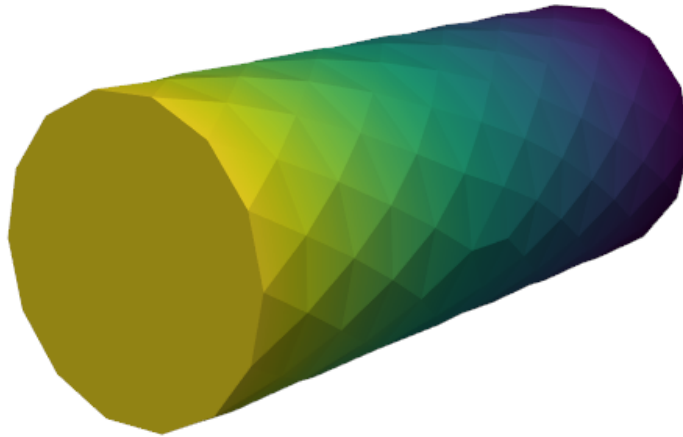
- the diffusivity coefficient
- the max. initial condition value
- temperature field approximation order
- uniform mesh refinement

The example shows also how to probe the results.

In the SfePy top-level directory the following command can be used to get usage information:

```
python sfepy/examples/diffusion/time_poisson_interactive.py -h
```





source code

```
#!/usr/bin/env python
"""
Transient Laplace equation (heat equation) with non-constant initial conditions
given by a function, using commands for interactive use.

The script allows setting various simulation parameters, namely:

- the diffusivity coefficient
- the max. initial condition value
- temperature field approximation order
- uniform mesh refinement

The example shows also how to probe the results.

In the SfePy top-level directory the following command can be used to get usage
information::

    python sfepy/examples/diffusion/time_poisson_interactive.py -h
"""
from __future__ import absolute_import
import sys
```

(continues on next page)

(continued from previous page)

```

from six.moves import range
sys.path.append('.')
from argparse import ArgumentParser, RawDescriptionHelpFormatter

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import assert_, output, ordered_iteritems, IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                            Equation, Equations, Problem)
from sfepy.discrete.problem import prepare_matrix
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC, InitialCondition
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.solvers.ts_solvers import SimpleTimeSteppingSolver
from sfepy.discrete.probes import LineProbe, CircleProbe
from sfepy.discrete.projections import project_by_component

def gen_probes(problem):
    """
    Define a line probe and a circle probe.
    """
    # Use enough points for higher order approximations.
    n_point = 1000

    p0, p1 = nm.array([0.0, 0.0, 0.0]), nm.array([0.1, 0.0, 0.0])
    line = LineProbe(p0, p1, n_point, share_geometry=True)
    # Workaround current probe code shortcoming.
    line.set_options(close_limit=0.5)

    centre = 0.5 * (p0 + p1)
    normal = [0.0, 1.0, 0.0]
    r = 0.019
    circle = CircleProbe(centre, normal, r, n_point, share_geometry=True)
    circle.set_options(close_limit=0.0)

    probes = [line, circle]
    labels = ['%s -> %s' % (p0, p1),
              'circle(%s, %s, %s' % (centre, normal, r)]

    return probes, labels

def probe_results(ax_num, T, dvel, probe, label):
    """
    Probe the results using the given probe and plot the probed values.
    """
    results = {}

    pars, vals = probe(T)
    results['T'] = (pars, vals)

```

(continues on next page)

(continued from previous page)

```

pars, vals = probe(dvel)
results['dvel'] = (pars, vals)

fig = plt.figure(1)

ax = plt.subplot(2, 2, 2 * ax_num + 1)
ax.cla()
pars, vals = results['T']

ax.plot(pars, vals[:, 0], label=r'$T$', lw=1, ls='-', marker='+', ms=3)
dx = 0.05 * (pars[-1] - pars[0])
ax.set_xlim(pars[0] - dx, pars[-1] + dx)
ax.set_ylabel('temperature')
ax.set_xlabel('probe %s' % label, fontsize=8)
ax.legend(loc='best', fontsize=10)

ax = plt.subplot(2, 2, 2 * ax_num + 2)
ax.cla()
pars, vals = results['dvel']
for ic in range(vals.shape[1]):
    ax.plot(pars, vals[:, ic], label=r'$w_{%d}$' % (ic + 1),
            lw=1, ls='-', marker='+', ms=3)
dx = 0.05 * (pars[-1] - pars[0])
ax.set_xlim(pars[0] - dx, pars[-1] + dx)
ax.set_ylabel('diffusion velocity')
ax.set_xlabel('probe %s' % label, fontsize=8)
ax.legend(loc='best', fontsize=10)

return fig, results

helps = {
    'diffusivity' : 'the diffusivity coefficient [default: %(default)s]',
    'ic_max' : 'the max. initial condition value [default: %(default)s]',
    'order' : 'temperature field approximation order [default: %(default)s]',
    'refine' : 'uniform mesh refinement level [default: %(default)s]',
    'probe' : 'probe the results',
    'show' : 'show the probing results figure, if --probe is used',
}

def main():
    from sfepy import data_dir

    parser = ArgumentParser(description=__doc__,
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--version', action='version', version='%(prog)s')
    parser.add_argument('--diffusivity', metavar='float', type=float,
                        action='store', dest='diffusivity',
                        default=1e-5, help=helps['diffusivity'])
    parser.add_argument('--ic-max', metavar='float', type=float,
                        action='store', dest='ic_max',
                        default=2.0, help=helps['ic_max'])

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('--order', metavar='int', type=int,
                    action='store', dest='order',
                    default=2, help=helps['order'])
parser.add_argument('-r', '--refine', metavar='int', type=int,
                    action='store', dest='refine',
                    default=0, help=helps['refine'])
parser.add_argument('-p', '--probe',
                    action="store_true", dest='probe',
                    default=False, help=helps['probe'])
parser.add_argument('-s', '--show',
                    action="store_true", dest='show',
                    default=False, help=helps['show'])
options = parser.parse_args()

assert_((0 < options.order),
        'temperature approximation order must be at least 1!')

output('using values:')
output('  diffusivity:', options.diffusivity)
output('  max. IC value:', options.ic_max)
output('uniform mesh refinement level:', options.refine)

mesh = Mesh.from_file(data_dir + '/meshes/3d/cylinder.mesh')
domain = FEDomain('domain', mesh)

if options.refine > 0:
    for ii in range(options.refine):
        output('refine %d...' % ii)
        domain = domain.refine()
        output('... %d nodes %d elements'
              % (domain.shape.n_nod, domain.shape.n_el))

omega = domain.create_region('Omega', 'all')
left = domain.create_region('Left',
                            'vertices in x < 0.00001', 'facet')
right = domain.create_region('Right',
                             'vertices in x > 0.099999', 'facet')

field = Field.from_args('fu', nm.float64, 'scalar', omega,
                       approx_order=options.order)

T = FieldVariable('T', 'unknown', field, history=1)
s = FieldVariable('s', 'test', field, primary_var_name='T')

m = Material('m', diffusivity=options.diffusivity * nm.eye(3))

integral = Integral('i', order=2*options.order)

t1 = Term.new('dw_diffusion(m.diffusivity, s, T)',
              integral, omega, m=m, s=s, T=T)
t2 = Term.new('dw_dot(s, dT/dt)',
              integral, omega, s=s, T=T)

```

(continues on next page)

(continued from previous page)

```

eq = Equation('balance', t1 + t2)
eqs = Equations([eq])

# Boundary conditions.
ebc1 = EssentialBC('T1', left, {'T.0' : 2.0})
ebc2 = EssentialBC('T2', right, {'T.0' : -2.0})

# Initial conditions.
def get_ic(coors, ic):
    x, y, z = coors.T
    return 2 - 40.0 * x + options.ic_max * nm.sin(4 * nm.pi * x / 0.1)
ic_fun = Function('ic_fun', get_ic)
ic = InitialCondition('ic', omega, {'T.0' : ic_fun})

pb = Problem('heat', equations=eqs)
pb.set_bcs(ebcs=Conditions([ebc1, ebc2]))
pb.set_ics(Conditions([ic]))

variables = pb.get_initial_state()
init_fun, prestep_fun, _poststep_fun = pb.get_tss_functions()

ls = ScipyDirect({})
nls_status = IndexedStruct()
nls = Newton({'is_linear' : True}, lin_solver=ls, status=nls_status)
tss = SimpleTimeSteppingSolver({'t0' : 0.0, 't1' : 100.0, 'n_step' : 11},
                               nls=nls, context=pb, verbose=True)
pb.set_solver(tss)

if options.probe:
    # Prepare probe data.
    probes, labels = gen_probes(pb)

    ev = pb.evaluate
    order = 2 * (options.order - 1)

    gfield = Field.from_args('gu', nm.float64, 'vector', omega,
                             approx_order=options.order - 1)
    dvel = FieldVariable('dvel', 'parameter', gfield,
                         primary_var_name='(set-to-None)')
    cfield = Field.from_args('gu', nm.float64, 'scalar', omega,
                             approx_order=options.order - 1)
    component = FieldVariable('component', 'parameter', cfield,
                              primary_var_name='(set-to-None)')

    nls_options = {'eps_a' : 1e-16, 'i_max' : 1}

    suffix = tss.ts.suffix
    def poststep_fun(ts, vec):
        vec = _poststep_fun(ts, vec)

        # Probe the solution.
        dvel_qp = ev('ev_diffusion_velocity.%d.Omega(m.diffusivity, T)')

```

(continues on next page)

(continued from previous page)

```

        % order, copy_materials=False, mode='qp')
    project_by_component(dvel, dvel_qp, component, order,
                        nls_options=nls_options)

    all_results = []
    for ii, probe in enumerate(probes):
        fig, results = probe_results(ii, T, dvel, probe, labels[ii])

        all_results.append(results)

    plt.tight_layout()
    fig.savefig('time_poisson_interactive_probe_%.png'
                % (suffix % ts.step), bbox_inches='tight')

    for ii, results in enumerate(all_results):
        output('probe %d (%s):' % (ii, probes[ii].name))
        output.level += 2
        for key, res in ordered_iteritems(results):
            output(key + ':')
            val = res[1]
            output('  min: %+.2e, mean: %+.2e, max: %+.2e'
                  % (val.min(), val.mean(), val.max()))
        output.level -= 2

    return vec

else:
    poststep_fun = _poststep_fun

    pb.time_update(tss.ts)
    variables.apply_ebc()

    # This is required if {'is_linear' : True} is passed to Newton.
    mtx = prepare_matrix(pb, variables)
    pb.try_presolve(mtx)

    tss_status = IndexedStruct()
    tss(variables.get_state(pb.active_only, force=True),
        init_fun=init_fun, prestep_fun=prestep_fun, poststep_fun=poststep_fun,
        status=tss_status)

    output(tss_status)

    if options.show:
        plt.show()

if __name__ == '__main__':
    main()

```


homogenization

homogenization/homogenization_opt.py

Description

missing description!

source code

```

from __future__ import absolute_import
import numpy as nm

import sfepy.discrete.fem.periodic as per
from sfepy.discrete.fem.mesh import Mesh
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.homogenization.utils import define_box_regions
import sfepy.homogenization.coefs_base as cb
from sfepy import data_dir

# material function
def get_mat(coors, mode, pb):
    if mode == 'qp':
        cnf = pb.conf
        # get material coefficients
        if hasattr(cnf, 'opt_data'):
            # from optim.
            E_f, nu_f, E_m, nu_m = cnf.opt_data['mat_params']
        else:
            # given values
            E_f, nu_f, E_m, nu_m = 160.e9, 0.28, 5.e9, 0.45

    nqp = coors.shape[0]
    nel = pb.domain.mesh.n_el
    nqpe = nqp // nel
    out = nm.zeros((nqp, 6, 6), dtype=nm.float64)

    # set values - matrix
    D_m = stiffness_from_youngpoisson(3, E_m, nu_m)
    Ym = pb.domain.regions['Ym'].get_cells()
    idx0 = (nm.arange(nqpe)[: ,nm.newaxis] * nm.ones((1, Ym.shape[0])),
            dtype=nm.int32)).T.flatten()
    idxs = (Ym[: ,nm.newaxis] * nm.ones((1, nqpe),
            dtype=nm.int32)).flatten() * nqpe
    out[idxs + idx0,...] = D_m

    # set values - fiber
    D_f = stiffness_from_youngpoisson(3, E_f, nu_f)
    Yf = pb.domain.regions['Yf'].get_cells()
    idx0 = (nm.arange(nqpe)[: ,nm.newaxis] * nm.ones((1, Yf.shape[0])),
            dtype=nm.int32)).T.flatten()
    idxs = (Yf[: ,nm.newaxis] * nm.ones((1, nqpe),
            dtype=nm.int32)).flatten() * nqpe
    out[idxs + idx0,...] = D_f

```

(continues on next page)

(continued from previous page)

```

        return {'D': out}

def optimization_hook(pb):
    cnf = pb.conf
    out = []
    yield pb, out

    if hasattr(cnf, 'opt_data'):
        # store homogenized tensor
        pb.conf.opt_data['D_homog'] = out[-1].D.copy()

    yield None

def define(is_opt=False):
    filename_mesh = data_dir + '/meshes/3d/matrix_fiber_rand.vtk'

    mesh = Mesh.from_file(filename_mesh)
    bbox = mesh.get_bounding_box()

    regions = {
        'Y' : 'all',
        'Ym' : ('cells of group 7', 'cell'),
        'Yf' : ('r.Y -c r.Ym', 'cell'),
    }

    regions.update(define_box_regions(3, bbox[0], bbox[1]))

    functions = {
        'get_mat': (lambda ts, coors, mode=None, problem=None, **kwargs:
                     get_mat(coors, mode, problem)),
        'match_x_plane' : (per.match_x_plane,),
        'match_y_plane' : (per.match_y_plane,),
        'match_z_plane' : (per.match_z_plane,),
    }

    materials = {
        'mat': 'get_mat',
    }

    fields = {
        'corrector' : ('real', 3, 'Y', 1),
    }

    variables = {
        'u': ('unknown field', 'corrector'),
        'v': ('test field', 'corrector', 'u'),
        'Pi': ('parameter field', 'corrector', 'u'),
        'Pi1': ('parameter field', 'corrector', '(set-to-None)'),
        'Pi2': ('parameter field', 'corrector', '(set-to-None)'),
    }

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    'fixed_u' : ('Corners', {'u.all' : 0.0}),
}

epbcs = {
    'periodic_x' : ([ 'Left', 'Right' ], {'u.all' : 'u.all'}, 'match_x_plane'),
    'periodic_y' : ([ 'Near', 'Far' ], {'u.all' : 'u.all'}, 'match_y_plane'),
    'periodic_z' : ([ 'Top', 'Bottom' ], {'u.all' : 'u.all'}, 'match_z_plane'),
}

all_periodic = ['periodic_%s' % ii for ii in ['x', 'y', 'z'][:3]]

options = {
    'coefs': 'coefs',
    'requirements': 'requirements',
    'volume': { 'variables' : ['u'], 'expression' : 'ev_volume.5.Y( u )' },
    'output_dir': 'output',
    'coefs_filename': 'coefs_le',
}

equation_corrs = {
    'balance_of_forces':
        """dw_lin_elastic.5.Y(mat.D, v, u)
        = - dw_lin_elastic.5.Y(mat.D, v, Pi)"""
}

coefs = {
    'D' : {
        'requires' : ['pis', 'corrs_rs'],
        'expression' : 'dw_lin_elastic.5.Y(mat.D, Pi1, Pi2 )',
        'set_variables': [( 'Pi1', ( 'pis', 'corrs_rs' ), 'u' ),
                          ( 'Pi2', ( 'pis', 'corrs_rs' ), 'u' )],
        'class' : cb.CoeffSymSym,
    },
    'vol': {
        'regions': ['Ym', 'Yf'],
        'expression': 'ev_volume.5.%s(u)',
        'class': cb.VolumeFractions,
    },
    'filenames' : {},
}

requirements = {
    'pis' : {
        'variables' : ['u'],
        'class' : cb.ShapeDimDim,
    },
    'corrs_rs' : {
        'requires' : ['pis'],
        'ebcs' : ['fixed_u'],
        'epbcs' : all_periodic,
        'equations' : equation_corrs,
    },
}

```

(continues on next page)

(continued from previous page)

```

        'set_variables' : [('Pi', 'pis', 'u')],
        'class' : cb.CorrDimDim,
        'save_name' : 'corrs_le',
        'is_linear': True,
    },
}

solvers = {
    'ls' : ('ls.auto_direct', {'use_presolve' : True}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-4,
        'problem': 'linear',
    })
}

if is_opt:
    options.update({
        'parametric_hook': 'optimization_hook',
        'float_format': '%.16e',
    })

return locals()

```

homogenization/linear_elastic_mM.py

Description

Linear elasticity with effective macroscopic properties determined according to the theory of homogenization from a periodic microstructure.

source code

```

"""
Linear elasticity with effective macroscopic properties determined
according to the theory of homogenization from a periodic microstructure.
"""

import os
from sfepy import data_dir, base_dir
from sfepy.base.base import nm
from sfepy.homogenization.micmac import get_homog_coefs_linear
from sfepy.homogenization.recovery import save_recovery_region, \
    recover_micro_hook

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    if isinstance(state, dict):
        pass
    else:
        stress = pb.evaluate('ev_cauchy_stress.i.Omega(solid.D, u)',
                              mode='el_avg')

```

(continues on next page)

(continued from previous page)

```

    strain = pb.evaluate('ev_cauchy_strain.i.Omega(u)',
                        mode='el_avg')
    out['cauchy_strain'] = Struct(name='output_data',
                                mode='cell', data=strain,
                                dofs=None)
    out['cauchy_stress'] = Struct(name='output_data',
                                mode='cell', data=stress,
                                dofs=None)

    if pb.conf.options.get('recover_micro', False):
        rname = pb.conf.options.recovery_region
        region = pb.domain.regions[rname]

        filename = os.path.join(os.path.dirname(pb.get_output_name()),
                                'recovery_region.vtk')
        save_recovery_region(pb, rname, filename=filename);

        rstrain = pb.evaluate('ev_cauchy_strain.i.%s(u)' % rname,
                              mode='el_avg')[:, 0, ...]

        recover_micro_hook(pb.conf.options.micro_filename,
                           region, {'strain': rstrain}, 0.01,
                           output_dir=pb.conf.options.output_dir)

    return out

def get_elements(coors, domain=None):
    return nm.arange(50, domain.shape.n_el, 100)

regenerate = True

def get_homog(ts, coors, mode=None,
              equations=None, term=None, problem=None, **kwargs):
    global regenerate

    out = get_homog_coefs_linear(ts, coors, mode, regenerate=regenerate,
                                micro_filename=options['micro_filename'],
                                output_dir=problem.conf.options.output_dir)

    regenerate = False

    return out

functions = {
    'get_elements' : (get_elements,),
    'get_homog' : (get_homog,),
}

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),

```

(continues on next page)

(continued from previous page)

```

    'Right' : ('vertices in (x > 0.099)', 'facet'),
    'Recovery' : 'cells by get_elements',
}

materials = {
    'solid' : 'get_homog',
}

fields = {
    '3_displacement' : ('real', 3, 'Omega', 1),
}

integrals = {
    'i' : 1,
}

variables = {
    'u' : ('unknown field', '3_displacement', 0),
    'v' : ('test field', '3_displacement', 'u'),
}

ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'PerturbedSurface' : ('Right', {'u.0' : 0.02, 'u.1' : 0.0, 'u.2' : 0.0}),
}

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
    }),
}

micro_filename = base_dir \
    + '/examples/homogenization/linear_homogenization_up.py'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'output_dir' : 'output',
    'post_process_hook' : 'post_process',
    'output_prefix' : 'macro:',
    'recover_micro' : True,
    'recovery_region' : 'Recovery',
    'micro_filename' : micro_filename,
}

```

homogenization/linear_elasticity_opt.py**Description**

missing description!

source code

```

from __future__ import absolute_import
import numpy as nm

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy import data_dir

def mesh_hook(mesh, mode):
    if mode == 'read':
        mesh = gen_block_mesh([0.0098, 0.0011, 0.1], [5, 3, 17],
                               [0, 0, 0.05], name='specimen',
                               verbose=False)

        return mesh

    elif mode == 'write':
        pass

def optimization_hook(pb):
    cnf = pb.conf
    out = []
    yield pb, out

    state = out[-1][1].get_state_parts()
    coors = pb.domain.cmesh.coors
    displ = state['u'].reshape((coors.shape[0], 3))
    # elongation
    mcoors = coors[cnf.mnodes, 2]
    mdispl = displ[cnf.mnodes, 2]
    dl = (mdispl[1] - mdispl[0]) / (mcoors[1] - mcoors[0])

    if hasattr(cnf, 'opt_data'):
        # compute slope of the force-elongation curve
        cnf.opt_data['k'] = cnf.F / dl

    yield None

def get_mat(coors, mode, pb):
    if mode == 'qp':
        # get material data
        if hasattr(pb.conf, 'opt_data'):
            # from homogenization
            D = pb.conf.opt_data['D_homog']
        else:
            # given values
            D = stiffness_from_youngpoisson(3, 150.0e9, 0.3)

```

(continues on next page)

(continued from previous page)

```

    nqp = coors.shape[0]
    return {'D': nm.tile(D, (nqp, 1, 1))}

def define(is_opt=False):
    filename_mesh = UserMeshIO(mesh_hook)
    mnodes = (107, 113) # nodes for elongation eval.

    regions = {
        'Omega': 'all',
        'Bottom': ('vertices in (z < 0.001)', 'facet'),
        'Top': ('vertices in (z > 0.099)', 'facet'),
    }

    functions = {
        'get_mat': (lambda ts, coors, mode=None, problem=None, **kwargs:
                     get_mat(coors, mode, problem)),
    }

    S = 1.083500e-05 # specimen cross-section
    F = 5.0e3 # force
    materials = {
        'solid': 'get_mat',
        'load': ({'val': F / S},),
    }

    fields = {
        'displacement': ('real', 'vector', 'Omega', 1),
    }

    variables = {
        'u': ('unknown field', 'displacement', 0),
        'v': ('test field', 'displacement', 'u'),
    }

    ebcs = {
        'FixedBottom': ('Bottom', {'u.all': 0.0}),
        'FixedTop': ('Top', {'u.0': 0.0, 'u.1': 0.0}),
    }

    equations = {
        'balance_of_forces' :
        """dw_lin_elastic.5.Omega(solid.D, v, u)
        = dw_surface_ltr.5.Top(load.val, v)""",
    }

    solvers = {
        'ls': ('ls.scipy_direct', {}),
        'newton': ('nls.newton', {'eps_a': 1e-6, 'eps_r': 1.e-6,
                                   'check': 0, 'problem': 'nonlinear'}),
    }

```

(continues on next page)

(continued from previous page)

```
options = {  
    'parametric_hook': 'optimization_hook',  
    'output_dir' : 'output',  
}  
  
return locals()
```

homogenization/linear_homogenization.py

Description

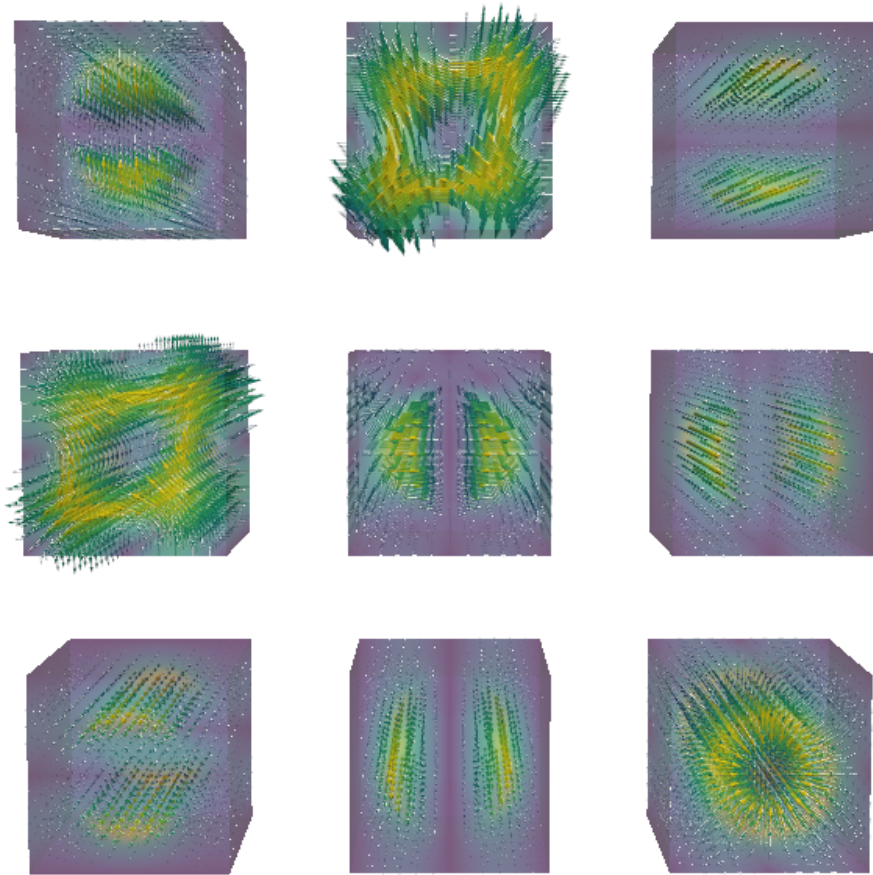
Compute homogenized elastic coefficients for a given heterogeneous linear elastic microstructure.

See [1] for details or [2] and [3] for a quick explanation.

[1] D. Cioranescu, J.S.J. Paulin: Homogenization in open sets with holes. *Journal of Mathematical Analysis and Applications* 71(2), 1979, pages 590-607. [https://doi.org/10.1016/0022-247X\(79\)90211-7](https://doi.org/10.1016/0022-247X(79)90211-7)

[2] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias: Asymptotic homogenisation in linear elasticity. Part I: Mathematical formulation and finite element modelling. *Computational Materials Science* 45(4), 2009, pages 1073-1080. <http://dx.doi.org/10.1016/j.commatsci.2009.02.025>

[3] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias: Asymptotic homogenisation in linear elasticity. Part II: Finite element procedures and multiscale applications. *Computational Materials Science* 45(4), 2009, pages 1081-1096. <http://dx.doi.org/10.1016/j.commatsci.2009.01.027>



source code

```
r"""
Compute homogenized elastic coefficients for a given heterogeneous linear
elastic microstructure.

See [1] for details or [2] and [3] for a quick explanation.

[1] D. Cioranescu, J.S.J. Paulin: Homogenization in open sets with holes.
Journal of Mathematical Analysis and Applications 71(2), 1979, pages 590-607.
https://doi.org/10.1016/0022-247X(79)90211-7

[2] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias:
Asymptotic homogenisation in linear elasticity.
Part I: Mathematical formulation and finite element modelling.
Computational Materials Science 45(4), 2009, pages 1073-1080.
http://dx.doi.org/10.1016/j.commatsci.2009.02.025

[3] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias:
Asymptotic homogenisation in linear elasticity.
Part II: Finite element procedures and multiscale applications.
Computational Materials Science 45(4), 2009, pages 1081-1096.
http://dx.doi.org/10.1016/j.commatsci.2009.01.027
```

(continues on next page)

(continued from previous page)

```

"""

from __future__ import absolute_import
import sfepy.discrete.fem.periodic as per
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.homogenization.utils import define_box_regions
import sfepy.homogenization.coefs_base as cb
from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.homogenization.recovery import compute_micro_u,\
    compute_stress_strain_u, compute_mac_stress_part

def recovery_le(pb, corrs, macro):

    out = {}

    dim = corrs['corrs_le']['u_00'].shape[1]
    mic_u = - compute_micro_u(corrs['corrs_le'], macro['strain'], 'u', dim)

    out['u_mic'] = Struct(name='output_data',
                        mode='vertex', data=mic_u)

    stress_Y, strain_Y = \
        compute_stress_strain_u(pb, 'i', 'Y', 'mat.D', 'u', mic_u)
    stress_Y += \
        compute_mac_stress_part(pb, 'i', 'Y', 'mat.D', 'u', macro['strain'])

    strain = macro['strain'] + strain_Y

    out['cauchy_strain'] = Struct(name='output_data',
                                mode='cell', data=strain)
    out['cauchy_stress'] = Struct(name='output_data',
                                mode='cell', data=stress_Y)

    return out

filename_mesh = data_dir + '/meshes/3d/matrix_fiber.mesh'
dim = 3
region_lbn = (0, 0, 0)
region_rtf = (1, 1, 1)

regions = {
    'Y': 'all',
    'Ym': 'cells of group 1',
    'Yc': 'cells of group 2',
}
regions.update(define_box_regions(dim, region_lbn, region_rtf))

materials = {
    'mat': ({'D': {'Ym': stiffness_from_youngpoisson(dim, 7.0e9, 0.4),

```

(continues on next page)

(continued from previous page)

```

        'Yc': stiffness_from_youngpoisson(dim, 70.0e9, 0.2)}},),
}

fields = {
    'corrector': ('real', dim, 'Y', 1),
}

variables = {
    'u': ('unknown field', 'corrector', 0),
    'v': ('test field', 'corrector', 'u'),
    'Pi': ('parameter field', 'corrector', 'u'),
    'Pi1': ('parameter field', 'corrector', '(set-to-None)'),
    'Pi2': ('parameter field', 'corrector', '(set-to-None)'),
}

functions = {
    'match_x_plane': (per.match_x_plane,),
    'match_y_plane': (per.match_y_plane,),
    'match_z_plane': (per.match_z_plane,),
}

ebcs = {
    'fixed_u': ('Corners', {'u.all': 0.0}),
}

if dim == 3:
    epbcs = {
        'periodic_x': (['Left', 'Right'], {'u.all': 'u.all'},
                        'match_x_plane'),
        'periodic_y': (['Near', 'Far'], {'u.all': 'u.all'},
                        'match_y_plane'),
        'periodic_z': (['Top', 'Bottom'], {'u.all': 'u.all'},
                        'match_z_plane'),
    }
else:
    epbcs = {
        'periodic_x': (['Left', 'Right'], {'u.all': 'u.all'},
                        'match_x_plane'),
        'periodic_y': (['Bottom', 'Top'], {'u.all': 'u.all'},
                        'match_y_plane'),
    }

all_periodic = ['periodic_%s' % ii for ii in ['x', 'y', 'z'][:dim]]

integrals = {
    'i': 2,
}

options = {
    'coefs': 'coefs',
    'requirements': 'requirements',
    'ls': 'ls', # linear solver to use

```

(continues on next page)

(continued from previous page)

```

    'volume': {'expression': 'ev_volume.i.Y(u)'},
    'output_dir': 'output',
    'coefs_filename': 'coefs_le',
    'recovery_hook': 'recovery_le',
}

equation_corrs = {
    'balance_of_forces':
        """dw_lin_elastic.i.Y(mat.D, v, u) =
        - dw_lin_elastic.i.Y(mat.D, v, Pi)"""
}

expr_coefs = """dw_lin_elastic.i.Y(mat.D, Pi1, Pi2)"""

coefs = {
    'D': {
        'requires': ['pis', 'corrs_rs'],
        'expression': expr_coefs,
        'set_variables': [('Pi1', ('pis', 'corrs_rs'), 'u'),
                          ('Pi2', ('pis', 'corrs_rs'), 'u')],
        'class': cb.CoeffSymSym,
    },
    'filenames': {},
}

requirements = {
    'pis': {
        'variables': ['u'],
        'class': cb.ShapeDimDim,
        'save_name': 'corrs_pis',
    },
    'corrs_rs': {
        'requires': ['pis'],
        'ebcs': ['fixed_u'],
        'epbcs': all_periodic,
        'equations': equation_corrs,
        'set_variables': [('Pi', 'pis', 'u')],
        'class': cb.CorrDimDim,
        'save_name': 'corrs_le',
        'is_linear': True,
    },
}

solvers = {
    'ls': ('ls.auto_direct', {'use_presolve' : True}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-4,
    })
}

```

homogenization/linear_homogenization_postproc.py

Description

This example shows how to use the VTK postprocessing functions.

source code

```
"""
This example shows how to use the VTK postprocessing functions.
"""

from __future__ import absolute_import
import os.path as osp
from .linear_homogenization import *
from sfepy.postprocess.utils_vtk import get_vtk_from_mesh,\
    get_vtk_by_group, get_vtk_surface, get_vtk_edges, write_vtk_to_file,\
    tetrahedralize_vtk_mesh

options.update({
    'post_process_hook' : 'post_process',
})

def post_process(out, problem, state, extend=False):

    mesh = problem.domain.mesh
    mesh_name = mesh.name[mesh.name.rfind(osp.sep) + 1:]

    vtkdata = get_vtk_from_mesh(mesh, out, 'postproc_')
    matrix = get_vtk_by_group(vtkdata, 1, 1)

    matrix_surf = get_vtk_surface(matrix)
    matrix_surf_tri = tetrahedralize_vtk_mesh(matrix_surf)
    write_vtk_to_file('%s_mat1_surface.vtk' % mesh_name, matrix_surf_tri)

    matrix_edges = get_vtk_edges(matrix)
    write_vtk_to_file('%s_mat1_edges.vtk' % mesh_name, matrix_edges)

    return out
```

homogenization/linear_homogenization_up.py

Description

Compute homogenized elastic coefficients for a given heterogeneous linear elastic microstructure, see [1] for details or [2] and [3] for a quick explanation. The mixed formulation, where displacements and pressures are as unknowns, is used in this example.

[1] D. Cioranescu, J.S.J. Paulin: Homogenization in open sets with holes. *Journal of Mathematical Analysis and Applications* 71(2), 1979, pages 590-607. [https://doi.org/10.1016/0022-247X\(79\)90211-7](https://doi.org/10.1016/0022-247X(79)90211-7)

[2] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias: Asymptotic homogenisation in linear elasticity. Part I: Mathematical formulation and finite element modelling. *Computational Materials Science* 45(4), 2009, pages 1073-1080. <http://dx.doi.org/10.1016/j.commatsci.2009.02.025>

[3] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias: Asymptotic homogenisation in linear elasticity. Part II: Finite element procedures and multiscale applications. Computational Materials Science 45(4), 2009, pages 1081-1096. <http://dx.doi.org/10.1016/j.commatsci.2009.01.027>

source code

```

r"""
Compute homogenized elastic coefficients for a given heterogeneous linear
elastic microstructure, see [1] for details or [2] and [3] for a quick
explanation. The mixed formulation, where displacements and pressures are
as unknowns, is used in this example.

[1] D. Cioranescu, J.S.J. Paulin: Homogenization in open sets with holes.
Journal of Mathematical Analysis and Applications 71(2), 1979, pages 590-607.
https://doi.org/10.1016/0022-247X\(79\)90211-7

[2] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias:
Asymptotic homogenisation in linear elasticity.
Part I: Mathematical formulation and finite element modelling.
Computational Materials Science 45(4), 2009, pages 1073-1080.
http://dx.doi.org/10.1016/j.commatsci.2009.02.025

[3] J. Pinho-da-Cruz, J.A. Oliveira, F. Teixeira-Dias:
Asymptotic homogenisation in linear elasticity.
Part II: Finite element procedures and multiscale applications.
Computational Materials Science 45(4), 2009, pages 1081-1096.
http://dx.doi.org/10.1016/j.commatsci.2009.01.027
"""

from __future__ import absolute_import
import numpy as nm

import sfepy.discrete.fem.periodic as per
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson_mixed,\
    bulk_from_youngpoisson
from sfepy.homogenization.utils import define_box_regions, get_box_volume
import sfepy.homogenization.coefs_base as cb

from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.homogenization.recovery import compute_micro_u,\
    compute_stress_strain_u, compute_mac_stress_part, add_stress_p

def recovery_le(pb, corrs, macro):
    out = {}
    dim = corrs['corrs_le']['u_00'].shape[1]
    mic_u = - compute_micro_u(corrs['corrs_le'], macro['strain'], 'u', dim)
    mic_p = - compute_micro_u(corrs['corrs_le'], macro['strain'], 'p', dim)

    out['u_mic'] = Struct(name='output_data',
                        mode='vertex', data=mic_u)
    out['p_mic'] = Struct(name='output_data', mode='cell',
                        data=mic_p[:, nm.newaxis, :, nm.newaxis])

```

(continues on next page)

(continued from previous page)

```

stress_Y, strain_Y = \
    compute_stress_strain_u(pb, 'i', 'Y', 'mat.D', 'u', mic_u)
stress_Y += \
    compute_mac_stress_part(pb, 'i', 'Y', 'mat.D', 'u', macro['strain'])
add_stress_p(stress_Y, pb, 'i', 'Y', 'p', mic_p)

strain = macro['strain'] + strain_Y

out['cauchy_strain'] = Struct(name='output_data',
                             mode='cell', data=strain)
out['cauchy_stress'] = Struct(name='output_data',
                              mode='cell', data=stress_Y)

return out

dim = 3
filename_mesh = data_dir + '/meshes/3d/matrix_fiber.mesh'
region_lbn = (0, 0, 0)
region_rtf = (1, 1, 1)

regions = {
    'Y': 'all',
    'Ym': 'cells of group 1',
    'Yc': 'cells of group 2',
}
regions.update(define_box_regions(dim, region_lbn, region_rtf))

materials = {
    'mat': ({'D': {'Ym': stiffness_from_youngpoisson_mixed(dim, 7.0e9, 0.4),
                  'Yc': stiffness_from_youngpoisson_mixed(dim, 70.0e9, 0.2)},
            'gamma': {'Ym': 1.0/bulk_from_youngpoisson(7.0e9, 0.4),
                      'Yc': 1.0/bulk_from_youngpoisson(70.0e9, 0.2)}}),
}

fields = {
    'corrector_u': ('real', dim, 'Y', 1),
    'corrector_p': ('real', 1, 'Y', 0),
}

variables = {
    'u': ('unknown field', 'corrector_u'),
    'v': ('test field', 'corrector_u', 'u'),
    'p': ('unknown field', 'corrector_p'),
    'q': ('test field', 'corrector_p', 'p'),
    'Pi': ('parameter field', 'corrector_u', 'u'),
    'Pi1u': ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pi2u': ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pi1p': ('parameter field', 'corrector_p', '(set-to-None)'),
    'Pi2p': ('parameter field', 'corrector_p', '(set-to-None)'),
}

```

(continues on next page)

(continued from previous page)

```

}

functions = {
    'match_x_plane': (per.match_x_plane,),
    'match_y_plane': (per.match_y_plane,),
    'match_z_plane': (per.match_z_plane,),
}

ebcs = {
    'fixed_u': ('Corners', {'u.all': 0.0}),
}

if dim == 3:
    epbcs = {
        'periodic_x': (['Left', 'Right'], {'u.all': 'u.all'},
                        'match_x_plane'),
        'periodic_y': (['Near', 'Far'], {'u.all': 'u.all'},
                        'match_y_plane'),
        'periodic_z': (['Top', 'Bottom'], {'u.all': 'u.all'},
                        'match_z_plane'),
    }
else:
    epbcs = {
        'periodic_x': (['Left', 'Right'], {'u.all': 'u.all'},
                        'match_x_plane'),
        'periodic_y': (['Bottom', 'Top'], {'u.all': 'u.all'},
                        'match_y_plane'),
    }

all_periodic = ['periodic_%s' % ii for ii in ['x', 'y', 'z'][:dim]]

integrals = {
    'i': 2,
}

options = {
    'coefs': 'coefs',
    'requirements': 'requirements',
    'ls': 'ls', # linear solver to use
    'volume': {'value': get_box_volume(dim, region_lbn, region_rtf), },
    'output_dir': 'output',
    'coefs_filename': 'coefs_le_up',
    'recovery_hook': 'recovery_le',
    'multiprocessing': False,
}

equation_corrs = {
    'balance_of_forces':
        """ dw_lin_elastic.i.Y(mat.D, v, u)
            - dw_stokes.i.Y(v, p) =
            - dw_lin_elastic.i.Y(mat.D, v, Pi)""",
    'pressure_constraint':

```

(continues on next page)

(continued from previous page)

```

        """- dw_stokes.i.Y(u, q)
        - dw_dot.i.Y(mat.gamma, q, p) =
        + dw_stokes.i.Y(Pi, q)""",
    }

coefs = {
    'elastic_u': {
        'requires': ['pis', 'corrs_rs'],
        'expression': 'dw_lin_elastic.i.Y(mat.D, Pi1u, Pi2u)',
        'set_variables': [('Pi1u', ('pis', 'corrs_rs'), 'u'),
                          ('Pi2u', ('pis', 'corrs_rs'), 'u')],
        'class': cb.CoeffSymSym,
    },
    'elastic_p': {
        'requires': ['corrs_rs'],
        'expression': 'dw_dot.i.Y(mat.gamma, Pi1p, Pi2p)',
        'set_variables': [('Pi1p', 'corrs_rs', 'p'),
                          ('Pi2p', 'corrs_rs', 'p')],
        'class': cb.CoeffSymSym,
    },
    'D': {
        'requires': ['c.elastic_u', 'c.elastic_p'],
        'class': cb.CoeffSum,
    },
    'filenames': {},
}

requirements = {
    'pis': {
        'variables': ['u'],
        'class': cb.ShapeDimDim,
    },
    'corrs_rs': {
        'requires': ['pis'],
        'ebcs': ['fixed_u'],
        'epbcs': all_periodic,
        'equations': equation_corrs,
        'set_variables': [('Pi', 'pis', 'u')],
        'class': cb.CorrDimDim,
        'save_name': 'corrs_le',
        'is_linear': True,
    },
}

solvers = {
    'ls': ('ls.auto_direct', {'use_presolve' : True}),
    # 'ls': ('ls.auto_iterative', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e2,
    })
}

```

homogenization/material_opt.py

Description

See the *Material Identification* tutorial for a comprehensive description of this example.

source code

```
#!/usr/bin/env python
"""
See the :ref:`sec-mat_optim` tutorial for a comprehensive description of this
example.
"""
from __future__ import print_function
from __future__ import absolute_import
import sys
sys.path.append('.')

import numpy as nm
from scipy.optimize import fmin_tnc

import sfepy
from sfepy.base.base import Struct
from sfepy.base.log import Log

class MaterialOptimizer(object):

    @staticmethod
    def create_app(filename, is_homog=False, **kwargs):
        from sfepy.base.conf import ProblemConf, get_standard_keywords
        from sfepy.homogenization.homogen_app import HomogenizationApp
        from sfepy.applications import PDESolverApp

        required, other = get_standard_keywords()
        if is_homog:
            required.remove('equations')

        conf = ProblemConf.from_file(filename, required, other,
                                     define_args=kwargs)
        options = Struct(output_filename_trunk=None,
                         save_ebc=False,
                         save_ebc_nodes=False,
                         save_regions=False,
                         save_regions_as_groups=False,
                         solve_not=False)

        if is_homog:
            app = HomogenizationApp(conf, options, 'material_opt_micro:')
        else:
            app = PDESolverApp(conf, options, 'material_opt_macro:')

        app.conf.opt_data = {}
        opts = conf.options
```

(continues on next page)

(continued from previous page)

```

    if hasattr(opts, 'parametric_hook'): # Parametric study.
        parametric_hook = conf.get_function(opts.parametric_hook)
        app.parametrize(parametric_hook)

    return app

def x_norm2real(self, x):
    return x * (self.x_U - self.x_L) + self.x_L

def x_real2norm(self, x):
    return (x - self.x_L) / (self.x_U - self.x_L)

def __init__(self, macro_fn, micro_fn, x0, x_L, x_U, exp_data):
    self.macro_app = self.create_app(macro_fn, is_homog=False, is_opt=True)
    self.micro_app = self.create_app(micro_fn, is_homog=True, is_opt=True)
    self.x_L = nm.array(x_L)
    self.x_U = nm.array(x_U)
    self.x0 = self.x_real2norm(nm.array(x0))
    self.x = []
    self.eval_f = []
    self.exp_data = exp_data

    @staticmethod
    def rotate_mat(D, angle):
        s = nm.sin(angle)
        c = nm.cos(angle)
        s2 = s**2
        c2 = c**2
        sc = s * c
        T = nm.array([[c2, 0, s2, 0, 2*sc, 0],
                      [0, 1, 0, 0, 0, 0],
                      [s2, 0, c2, 0, -2*sc, 0],
                      [0, 0, 0, c, 0, -s],
                      [-sc, 0, sc, 0, c2 - s2, 0],
                      [0, 0, 0, s, 0, c]])

        return nm.dot(nm.dot(T, D), T.T)

def matopt_eval(self, x):
    mic_od = self.micro_app.conf.opt_data
    mac_od = self.macro_app.conf.opt_data

    mic_od['coefs'] = {}
    mic_od['mat_params'] = x
    self.micro_app()

    D = mic_od['D_homog']
    val = 0.0
    aux = []
    for phi, exp_k in self.exp_data:
        print('phi = %d' % phi)

```

(continues on next page)

(continued from previous page)

```

        mac_od['D_homog'] = self.rotate_mat(D, nm.deg2rad(phi))
        self.macro_app()

        comp_k = mac_od['k']
        val += (1.0 - comp_k / exp_k)**2
        aux.append((comp_k, exp_k))

    val = nm.sqrt(val)
    self.x.append(x)
    self.eval_f.append(val)

    return val

def iter_step(self, x, first_step=False):
    if first_step:
        self.log = Log(['0'], ['E_f', 'E_m'], ['v_f', 'v_m']],
            ylabel=[ 'Obj. fun.', "Young's modulus", "Poisson's ratio"],
            xlabel=[ 'iter', 'iter', 'iter'],
            aggregate=0)
        self.istep = 0
        self.log(0.5, x[0], x[2], x[1], x[3],
            x=[0, 0, 0, 0])
    else:
        self.istep += 1
        self.log(self.eval_f[-1], x[0], x[2], x[1], x[3],
            x=(self.istep,)*4)

def material_optimize(self):
    x0 = self.x0
    bnds = zip(self.x_real2norm(self.x_L), self.x_real2norm(self.x_U))
    feval = lambda x: self.matopt_eval(self.x_norm2real(x))
    istep = lambda x: self.iter_step(self.x_norm2real(x))
    self.iter_step(self.x_norm2real(x0), first_step=True)

    print('>>> material optimization START <<<')
    xopt = fmin_tnc(feval, x0, approx_grad=True, bounds=list(bnds),
        xtol=1e-3, callback=istep)
    print('>>> material optimization FINISHED <<<')

    self.log(finished=True)
    return self.x_norm2real(xopt[0])

def main():
    srcdir = sfepy.base_dir + '/examples/homogenization/'
    micro_filename = srcdir + 'homogenization_opt.py'
    macro_filename = srcdir + 'linear_elasticity_opt.py'

    exp_data = zip([0, 30, 60, 90], [1051140., 197330., 101226., 95474.])
    mo = MaterialOptimizer(macro_filename, micro_filename,
        [160.e9, 0.25, 5.e9, 0.45],
        [120e9, 0.2, 2e9, 0.2],
        [200e9, 0.45, 8e9, 0.45],

```

(continues on next page)

(continued from previous page)

```
list(exp_data))

optim_par = mo.material_optimize()
print('optimized parameters: ', optim_par)

if __name__ == '__main__':
    main()
```

homogenization/nonlinear_homogenization.py

Description

missing description!

source code

```
# -*- coding: utf-8 -*-
import numpy as nm
from sfepy.homogenization.utils import define_box_regions
import sfepy.homogenization.coefs_base as cb
import sfepy.discrete.fem.periodic as per
from sfepy.base.base import Struct
from sfepy.terms.terms_hyperelastic_ul import\
    HyperElasticULFamilyData, NeoHookeanULTerm, BulkPenaltyULTerm
from sfepy.terms.extmods.terms import sym2nonsym
from sfepy.discrete.functions import ConstantFunctionByRegion
from sfepy import data_dir
import sfepy.linalg as la

def recovery_hook(pb, ncoors, region, ts,
                  naming_scheme='step_iel', recovery_file_tag=''):
    from sfepy.base.ioutils import get_print_info
    from sfepy.homogenization.recovery import get_output_suffix
    import os.path as op

    for ii, icell in enumerate(region.cells):
        out = {}
        pb.set_mesh_coors(ncoors[ii], update_fields=True,
                           clear_all=False, actual=True)
        stress = pb.evaluate('ev_integrate_mat.3.Y(mat_he.S, u)',
                              mode='el_avg')

        out['cauchy_stress'] = Struct(name='output_data',
                                      mode='cell',
                                      data=stress,
                                      dofs=None)

        strain = pb.evaluate('ev_integrate_mat.3.Y(mat_he.E, u)',
                              mode='el_avg')

        out['green_strain'] = Struct(name='output_data',
```

(continues on next page)

(continued from previous page)

```

        mode='cell',
        data=strain,
        dofs=None)

out['displacement'] = Struct(name='output_data',
                             mode='vertex',
                             data=ncoors[ii] - pb.get_mesh_coors(),
                             dofs=None)

output_dir = pb.conf.options.get('output_dir', '.')
format = get_print_info(pb.domain.mesh.n_el, fill='0')[1]
suffix = get_output_suffix(icell, ts, naming_scheme, format,
                           pb.output_format)

micro_name = pb.get_output_name(extra='recovered_'
                                + recovery_file_tag + suffix)
filename = op.join(output_dir, op.basename(micro_name))
fpv = pb.conf.options.get('split_results_by', None)
pb.save_state(filename, out=out, split_results_by=fpv)

def def_mat(ts, mode, coors, term, pb):
    if not (mode == 'qp'):
        return

    if not hasattr(pb, 'family_data'):
        pb.family_data = HyperElasticULFamilyData()

    update_var = pb.conf.options.mesh_update_variable
    if pb.equations is None:
        state_u = pb.create_variables([update_var])[update_var]
    else:
        state_u = pb.get_variables()[update_var]

    if state_u.data[0] is None:
        state_u.init_data()

    state_u.set_data(
        pb.domain.get_mesh_coors(actual=True) - pb.domain.get_mesh_coors())
    state_u.field.clear_mappings()
    family_data = pb.family_data(state_u, term.region, term.integral,
                                list(term.geometry_types.values())[0])

    if len(state_u.field.mappings0) == 0:
        state_u.field.save_mappings()

    n_el, n_qp, dim, n_en, n_c = state_u.get_data_shape(term.integral,
                                                         term.act_integration,
                                                         term.region.name)

    conf_mat = pb.conf.materials
    solid_key = [key for key in conf_mat.keys() if 'solid' in key][0]

```

(continues on next page)

(continued from previous page)

```

solid_mat = conf_mat[solid_key].values
mat = {}
for mat_key in ['mu', 'K']:
    mat_fun = ConstantFunctionByRegion({mat_key: solid_mat[mat_key]})
    mat[mat_key] = mat_fun.function(ts=ts, coors=coors, mode='qp',
    term=term, problem=pb)[mat_key].reshape((n_el, n_qp, 1, 1))

shape = family_data.green_strain.shape[:2]
sym = family_data.green_strain.shape[-2]
dim2 = dim**2

fargs = [family_data.get(name)
          for name in NeoHookeanULTerm.family_data_names]
stress = nm.empty(shape + (sym, 1), dtype=nm.float64)
tanmod = nm.empty(shape + (sym, sym), dtype=nm.float64)
NeoHookeanULTerm.stress_function(stress, mat['mu'], *fargs)
NeoHookeanULTerm.tan_mod_function(tanmod, mat['mu'], *fargs)

fargs = [family_data.get(name)
          for name in BulkPenaltyULTerm.family_data_names]
stress_p = nm.empty(shape + (sym, 1), dtype=nm.float64)
tanmod_p = nm.empty(shape + (sym, sym), dtype=nm.float64)
BulkPenaltyULTerm.stress_function(stress_p, mat['K'], *fargs)
BulkPenaltyULTerm.tan_mod_function(tanmod_p, mat['K'], *fargs)

stress_ns = nm.zeros(shape + (dim2, dim2), dtype=nm.float64)
tanmod_ns = nm.zeros(shape + (dim2, dim2), dtype=nm.float64)
sym2nonsym(stress_ns, stress + stress_p)
sym2nonsym(tanmod_ns, tanmod + tanmod_p)

npts = nm.prod(shape)
J = family_data.det_f
mtx_f = family_data.mtx_f.reshape((npts, dim, dim))

out = {
    'E': 0.5 * (la.dot_sequences(mtx_f, mtx_f, 'ATB') - nm.eye(dim)),
    'A': ((tanmod_ns + stress_ns) / J).reshape((npts, dim2, dim2)),
    'S': ((stress + stress_p) / J).reshape((npts, sym, 1)),
}

return out

```

```

filename_mesh = data_dir + '/meshes/2d/special/circle_in_square_small.mesh'
dim = 2

```

```

options = {
    'coefs': 'coefs',
    'requirements': 'requirements',
    'volume': {'expression': 'ev_volume.5.Y(u)'},
    'output_dir': './output',
    'coefs_filename': 'coefs_hyper_homog',

```

(continues on next page)

(continued from previous page)

```

    'multiprocessing': True,
    'chunks_per_worker': 2,
    'micro_update': {'coors': [('corrs_rs', 'u', 'mtx_e')]},
    'mesh_update_variable': 'u',
    'recovery_hook': 'recovery_hook',
    'store_micro_idx': [49, 81],
}

fields = {
    'displacement': ('real', 'vector', 'Y', 1),
}

functions = {
    'match_x_plane': (per.match_x_plane,),
    'match_y_plane': (per.match_y_plane,),
    'mat_fce': (lambda ts, coors, mode=None, term=None, problem=None, **kwargs:
                def_mat(ts, mode, coors, term, problem)),
}

materials = {
    'mat_he': 'mat_fce',
    'solid': ({'K': {'Ym': 1000, 'Yc': 1000},
               'mu': {'Ym': 100, 'Yc': 10},
               },),
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
    'Pi': ('parameter field', 'displacement', 'u'),
    'Pi1u': ('parameter field', 'displacement', '(set-to-None)'),
    'Pi2u': ('parameter field', 'displacement', '(set-to-None)'),
}

regions = {
    'Y': 'all',
    'Ym': 'cells of group 1',
    'Yc': 'cells of group 2',
}

regions.update(define_box_regions(dim, (0., 0.), (1., 1.)))

ebcs = {
    'fixed_u': ('Corners', {'u.all': 0.0}),
}

epbcs = {
    'periodic_ux': (['Left', 'Right'], {'u.all': 'u.all'}, 'match_x_plane'),
    'periodic_uy': (['Bottom', 'Top'], {'u.all': 'u.all'}, 'match_y_plane'),
}

coefs = {

```

(continues on next page)

(continued from previous page)

```

'A': {
    'requires': ['pis', 'corrs_rs'],
    'expression': 'dw_nonsym_elastic.3.Y(mat_he.A, Pi1u, Pi2u)',
    'set_variables': [('Pi1u', ('pis', 'corrs_rs'), 'u'),
                      ('Pi2u', ('pis', 'corrs_rs'), 'u')],
    'class': cb.CoeffNonSymNonSym,
},
'S': {
    'expression': 'ev_integrate_mat.3.Y(mat_he.S, u)',
    'class': cb.CoeffOne,
}
}

requirements = {
    'pis': {
        'variables': ['u'],
        'class': cb.ShapeDimDim,
    },
    'corrs_rs': {
        'requires': ['pis'],
        'ebcs': ['fixed_u'],
        'epbcs': ['periodic_ux', 'periodic_uy'],
        'equations': {
            'balance_of_forces':
                """dw_nonsym_elastic.3.Y(mat_he.A, v, u)
                = - dw_nonsym_elastic.3.Y(mat_he.A, v, Pi)"""
        },
        'set_variables': [('Pi', 'pis', 'u')],
        'class': cb.CorrDimDim,
        'save_name': 'corrs_hyper_homog',
    },
}

solvers = {
    'ls': ('ls.auto_direct', {'use_presolve' : True}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-4,
        'problem': 'nonlinear',
    }),
}

```

homogenization/nonlinear_hyperelastic_mM.py**Description**

Homogenized nonlinear hyperelastic material with evolving microstructure deformation in each macroscopic quadrature point.

Run in parallel using:

```
mpirun -n 4 sfepy-run --app=bvp-mM --debug-mpi sfepy/examples/homogenization/nonlinear_
↪hyperelastic_mM.py
```

**source code**

```
"""
Homogenized nonlinear hyperelastic material with evolving microstructure
deformation in each macroscopic quadrature point.

Run in parallel using::

    mpirun -n 4 sfepy-run --app=bvp-mM --debug-mpi sfepy/examples/homogenization/
↪nonlinear_hyperelastic_mM.py
"""
import numpy as nm
```

(continues on next page)

(continued from previous page)

```

import six

from sfepy.base import data_dir, base_dir
from sfepy.base.base import Struct, output
from sfepy.terms.terms_hyperelastic_ul import HyperElasticULFamilyData
from sfepy.homogenization.micmac import get_homog_coefs_nonlinear
import sfepy.linalg as la
from sfepy.discrete.evaluate import Evaluator

hyperelastic_data = {}

def post_process(out, pb, state, extend=False):
    if isinstance(state, dict):
        pass
    else:
        pb.update_materials_flag = 2
        stress = pb.evaluate('ev_integrate_mat.1.Omega(solid.S, u)',
                             mode='el_avg')

        out['cauchy_stress'] = Struct(name='output_data',
                                     mode='cell',
                                     data=stress,
                                     dofs=None)

        strain = pb.evaluate('ev_integrate_mat.1.Omega(solid.E, u)',
                              mode='el_avg')

        out['green_strain'] = Struct(name='output_data',
                                     mode='cell',
                                     data=strain,
                                     dofs=None)

        pb.update_materials_flag = 0

    if pb.conf.options.get('recover_micro', False):
        happ = pb.homogen_app
        if pb.ts.step == 0:
            rname = pb.conf.options.recovery_region
            rcells = pb.domain.regions[rname].get_cells()
            sh = hyperelastic_data['homog_mat_shape']

            happ.app_options.store_micro_idx = sh[1] * rcells
        else:
            hpb = happ.problem
            recovery_hook = hpb.conf.options.get('recovery_hook', None)
            if recovery_hook is not None:
                recovery_hook = hpb.conf.get_function(recovery_hook)
                rname = pb.conf.options.recovery_region
                rcoors = []
                for ii in happ.app_options.store_micro_idx:
                    key = happ.get_micro_cache_key('coors', ii, pb.ts.step)

```

(continues on next page)

(continued from previous page)

```

        if key in happ.micro_state_cache:
            rcoors.append(happ.micro_state_cache[key])

    recovery_hook(hpb, rcoors, pb.domain.regions[rname], pb.ts)

    return out

def get_homog_mat(ts, coors, mode, term=None, problem=None, **kwargs):
    if problem.update_materials_flag == 2 and mode == 'qp':
        out = hyperelastic_data['homog_mat']
        return {k: nm.array(v) for k, v in six.iteritems(out)}
    elif problem.update_materials_flag == 0 or not mode == 'qp':
        return

    output('get_homog_mat')
    dim = problem.domain.mesh.dim

    update_var = problem.conf.options.mesh_update_variables[0]
    state_u = problem.equations.variables[update_var]
    state_u.field.clear_mappings()
    family_data = problem.family_data(state_u, term.region, term.integral,
                                      term.geometry_types['u'])

    mtx_f = family_data.mtx_f.reshape((coors.shape[0],)
                                      + family_data.mtx_f.shape[-2:])

    if hasattr(problem, 'mtx_f_prev'):
        rel_mtx_f = la.dot_sequences(mtx_f, nm.linalg.inv(problem.mtx_f_prev),
                                    'AB')
    else:
        rel_mtx_f = mtx_f

    problem.mtx_f_prev = mtx_f.copy()

    macro_data = {'mtx_e': rel_mtx_f - nm.eye(dim)} # '*' - macro strain
    out = get_homog_coefs_nonlinear(ts, coors, mode, macro_data,
                                   term=term, problem=problem,
                                   iteration=problem.iiter, **kwargs)

    out['E'] = 0.5 * (la.dot_sequences(mtx_f, mtx_f, 'ATB') - nm.eye(dim))

    hyperelastic_data['time'] = ts.step
    hyperelastic_data['homog_mat_shape'] = family_data.det_f.shape[:2]
    hyperelastic_data['homog_mat'] = \
        {k: nm.array(v) for k, v in six.iteritems(out)}

    return out

def ulf_iteration_hook(pb, nls, vec, it, err, err0):
    Evaluator.new_ulf_iteration(pb, nls, vec, it, err, err0)

```

(continues on next page)

(continued from previous page)

```

pb.iiter = it
pb.update_materials_flag = True
pb.update_materials()
pb.update_materials_flag = False

class MyEvaluator(Evaluator):
    def eval_residual(self, vec, is_full=False):
        if not is_full:
            vec = self.problem.equations.make_full_vec(vec)
            vec_r = self.problem.equations.eval_residuals(vec * 0)

        return vec_r

def ulf_init(pb):
    pb.family_data = HyperElasticULFamilyData()
    pb_vars = pb.get_variables()
    pb_vars['u'].init_data()

    pb.update_materials_flag = True
    pb.iiter = 0

options = {
    'output_dir': 'output',
    'mesh_update_variables': ['u'],
    'nls_iter_hook': ulf_iteration_hook,
    'pre_process_hook': ulf_init,
    'micro_filename': (base_dir +
                       '/examples/homogenization/nonlinear_homogenization.py'),
    'recover_micro': True,
    'recovery_region': 'Recovery',
    'post_process_hook': post_process,
    'user_evaluator': MyEvaluator,
}

materials = {
    'solid': 'get_homog',
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
}

filename_mesh = data_dir + '/meshes/2d/its2D.mesh'

```

(continues on next page)

(continued from previous page)

```

regions = {
    'Omega': 'all',
    'Left': ('vertices in (x < 0.001)', 'facet'),
    'Bottom': ('vertices in (y < 0.001)', 'facet'),
    'Recovery': ('cell 49, 81', 'cell'),
}

ebcs = {
    'l': ('Left', {'u.all': 0.0}),
    'b': ('Bottom', {'u.all': 'move_bottom'}),
}

centre = nm.array([0, 0], dtype=nm.float64)

def move_bottom(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d

    vec = coor[:, 0:2] - centre
    angle = 3 * ts.step
    print('angle:', angle)
    mtx = rotation_matrix2d(angle)
    out = nm.dot(vec, mtx) - vec

    return out

functions = {
    'move_bottom': (move_bottom,),
    'get_homog': (get_homog_mat,),
}

equations = {
    'balance_of_forces':
        """dw_nonsym_elastic.1.Omega(solid.A, v, u)
        = - dw_lin_prestress.1.Omega(solid.S, v)""",
}

solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'eps_a': 1e-3,
        'eps_r': 1e-3,
        'i_max': 20,
    }),
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'n_step': 3 + 1,
        'verbose': 1,
    })
}

```

(continues on next page)

(continued from previous page)

```
}  
})
```

homogenization/perfusion_micro.py

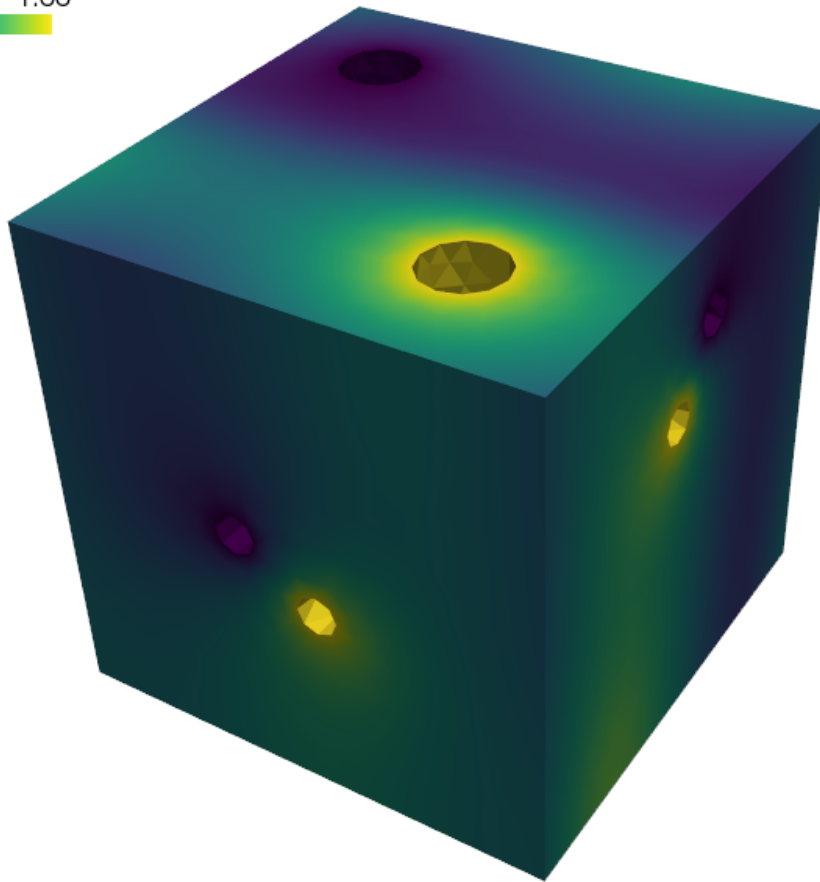
Description

Homogenization of the Darcy flow in a thin porous layer.

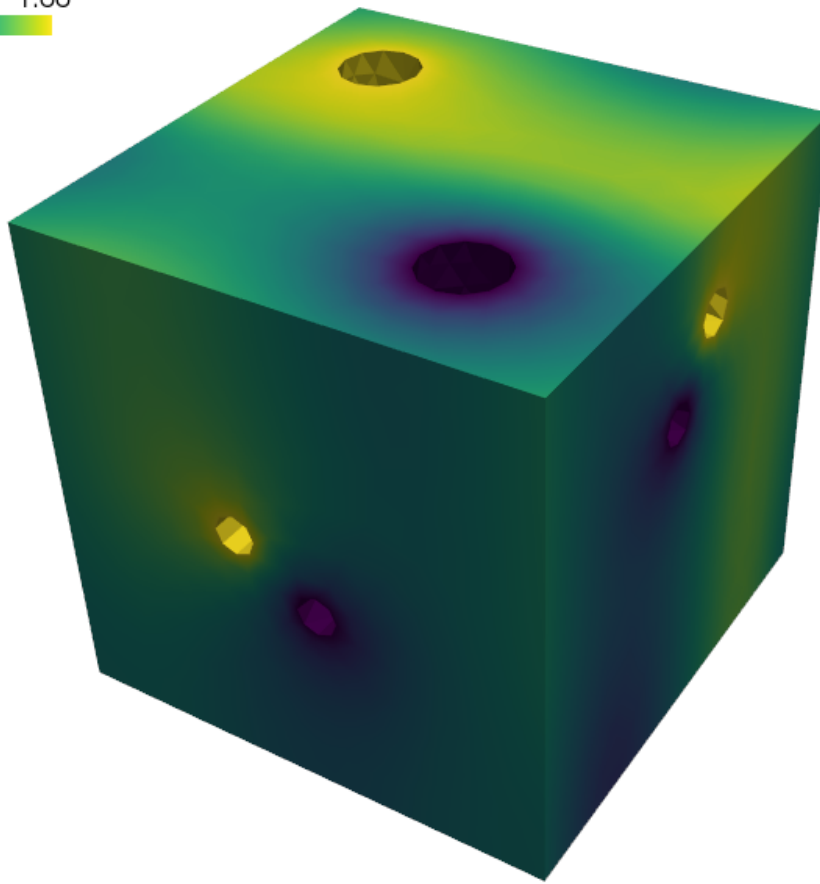
The reference cell is composed of the matrix representing the dual porosity and of two disconnected channels representing the primary porosity, see paper [1].

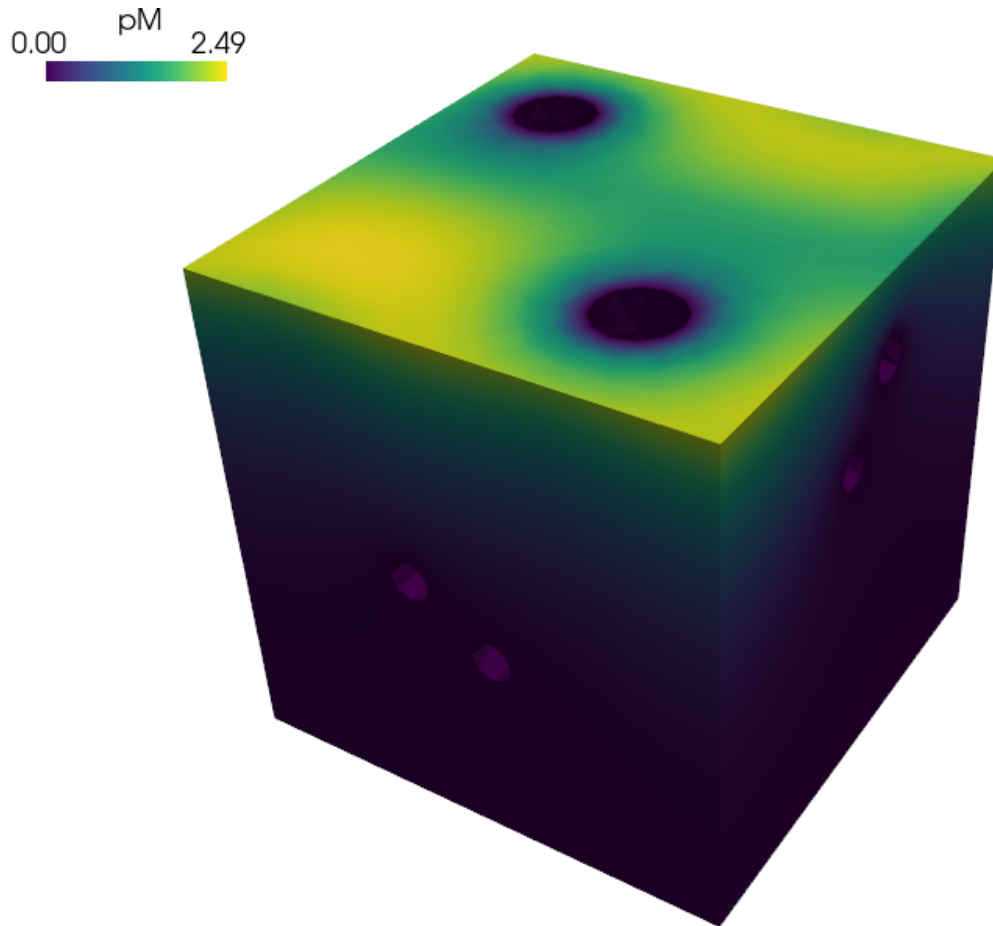
[1] E. Rohan, V. Lukeš: Modeling Tissue Perfusion Using a Homogenized Model with Layer-wise Decomposition. IFAC Proceedings Volumes 45(2), 2012, pages 1029-1034. <https://doi.org/10.3182/20120215-3-AT-3016.00182>

0.00 pM 1.00

0.00 pM 1.00





source code

```
# -*- coding: utf-8
r"""
Homogenization of the Darcy flow in a thin porous layer.

The reference cell is composed of the matrix representing the dual porosity
and of two disconnected channels representing the primary porosity,
see paper [1].

[1] E. Rohan, V. Lukeš: Modeling Tissue Perfusion Using a Homogenized
Model with Layer-wise Decomposition. IFAC Proceedings Volumes 45(2), 2012,
pages 1029-1034.
https://doi.org/10.3182/20120215-3-AT-3016.00182
"""

from __future__ import absolute_import
from sfepy.discrete.fem.periodic import match_x_plane, match_y_plane
import sfepy.homogenization.coefs_base as cb
import numpy as nm
from sfepy import data_dir
import six
from six.moves import range
```

(continues on next page)

(continued from previous page)

```

def get_mats(pk, ph, pe, dim):
    m1 = nm.eye(dim, dtype=nm.float64) * pk
    m1[-1, -1] = pk / ph
    m2 = nm.eye(dim, dtype=nm.float64) * pk
    m2[-1, -1] = pk / ph ** 2

    return m1, m2

def recovery_perf(pb, corrs, macro):
    from sfepy.homogenization.recovery import compute_p_from_macro
    from sfepy.base.base import Struct

    slev = ''

    micro_nnod = pb.domain.mesh.n_nod

    centre_Y = nm.sum(pb.domain.mesh.coors, axis=0) / micro_nnod
    nodes_Y = {}

    channels = {}
    for k in six.iterkeys(macro):
        if 'press' in k:
            channels[k[-1]] = 1

    channels = list(channels.keys())

    varnames = ['pM']
    for ch in channels:
        nodes_Y[ch] = pb.domain.regions['Y' + ch].vertices
        varnames.append('p' + ch)

    pvars = pb.create_variables(varnames)

    press = {}

    # matrix
    press['M'] = \
        corrs['corrs_%s_gamma_p' % pb_def['name']][['pM']] * macro['g_p'] + \
        corrs['corrs_%s_gamma_m' % pb_def['name']][['pM']] * macro['g_m']

    out = {}
    # channels
    for ch in channels:
        press_mac = macro['press' + ch][0, 0]
        press_mac_grad = macro['pressg' + ch]
        nnod = corrs['corrs_%s_pi%s' % (pb_def['name'], ch)] \
            ['p%s_0' % ch].shape[0]

        press_mic = nm.zeros((nnod, 1))
        for key, val in \

```

(continues on next page)

(continued from previous page)

```

six.iteritems(corr[six.iteritems(corr['corrs_%s_pi%s' % (pb_def['name'], ch)]]):
    kk = int(key[-1])
    press_mic += val * press_mac_grad[kk, 0]

for key in six.iterkeys(corr):
    if ('_gamma_' + ch in key):
        kk = int(key[-1]) - 1
        press_mic += corr[key]['p' + ch] * macro['g' + ch][kk]

press_mic += \
    compute_p_from_macro(press_mac_grad[nm.newaxis,nm.newaxis, :, :],
        micro_coors[nodes_Y[ch]], 0,
        centre=centre_Y, extdim=-1).reshape((nnod, 1))

press[ch] = press_mac + eps0 * press_mic

out[slev + 'p' + ch] = Struct(name='output_data',
    mode='vertex',
    data=press[ch],
    var_name='p' + ch,
    dofs=None)

pvars['p' + ch].set_data(press_mic)
dvel = pb.evaluate('ev_diffusion_velocity.iV.Y%s(mat1%s.k, p%s)'
    % (ch, ch, ch),
    var_dict={'p' + ch: pvars['p' + ch]},
    mode='el_avg')

out[slev + 'w' + ch] = Struct(name='output_data',
    mode='cell',
    data=dvel,
    var_name='w' + ch,
    dofs=None)

press['M'] += corr['corrs_%s_eta%s' % (pb_def['name'], ch)]['pM']\
    * press_mac

pvars['pM'].set_data(press['M'])
dvel = pb.evaluate('%e * ev_diffusion_velocity.iV.YM(mat1M.k, pM)' % eps0,
    var_dict={'pM': pvars['pM']}, mode='el_avg')

out[slev + 'pM'] = Struct(name='output_data',
    mode='vertex',
    data=press['M'],
    var_name='pM',
    dofs=None)

out[slev + 'wM'] = Struct(name='output_data',
    mode='cell',
    data=dvel,
    var_name='wM',
    dofs=None)

```

(continues on next page)

(continued from previous page)

```

    return out

geoms = {
    '2_4': ['2_4_Q1', '2', 5],
    '3_8': ['3_8_Q1', '4', 5],
    '3_4': ['3_4_P1', '3', 3],
}

pb_def = {
    'name': '3d_2ch',
    'mesh_filename': data_dir + '/meshes/3d/perfusion_micro3d.mesh',
    'dim': 3,
    'geom': geoms['3_4'],
    'eps0': 1.0e-2,
    'param_h': 1.0,
    'param_kappa_m': 0.1,
    'matrix_mat_el_grp': 3,
    'channels': {
        'A': {
            'mat_el_grp': 1,
            'fix_nd_grp': (4, 1),
            'io_nd_grp': [1, 2, 3],
            'param_kappa_ch': 1.0,
        },
        'B': {
            'mat_el_grp': 2,
            'fix_nd_grp': (14, 11),
            'io_nd_grp': [11, 12, 13],
            'param_kappa_ch': 2.0,
        },
    },
}

filename_mesh = pb_def['mesh_filename']
eps0 = pb_def['eps0']
param_h = pb_def['param_h']

# integrals
integrals = {
    'iV': 2,
    'iS': 2,
}

functions = {
    'match_x_plane': (match_x_plane,),
    'match_y_plane': (match_y_plane,),
}

aux = []
for ch, val in six.iteritems(pb_def['channels']):

```

(continues on next page)

(continued from previous page)

```

    aux.append('r.bYM' + ch)

# basic regions
regions = {
    'Y': 'all',
    'YM': 'cells of group %d' % pb_def['matrix_mat_el_grp'],
    # periodic boundaries
    'Pl': ('vertices in (x < 0.001)', 'facet'),
    'Pr': ('vertices in (x > 0.999)', 'facet'),
    'PLYM': ('r.Pl *v r.YM', 'facet'),
    'PrYM': ('r.Pr *v r.YM', 'facet'),
    'bYMp': ('r.bYp *v r.YM', 'facet', 'YM'),
    'bYmM': ('r.bYm *v r.YM', 'facet', 'YM'),
    'bYmPm': ('r.bYmP +v r.bYmM', 'facet', 'YM'),
}

# matrix/channel boundaries
regions.update({
    'bYMchs': (' +v '.join(aux), 'facet', 'YM'),
    'YmMchs': 'r.YM -v r.bYMchs',
})

# boundary conditions Gamma+/-
ebcs = {
    'gamma_pm_bYMchs': ('bYMchs', {'pM.0': 0.0}),
    'gamma_pm_YmMchs': ('YmMchs', {'pM.0': 1.0}),
}

# periodic boundary conditions - matrix, X-direction
epbcs = {'periodic_xYM': (['PLYM', 'PrYM'], {'pM.0': 'pM.0'}, 'match_x_plane')}
lcbcs = {}

all_periodicYM = ['periodic_%sYM' % ii for ii in ['x', 'y'][:pb_def['dim']-1]]
all_periodicY = {}

if pb_def['dim'] == 2:
    regions.update({
        'bYm': ('vertices in (y < 0.001)', 'facet'),
        'bYp': ('vertices in (y > 0.999)', 'facet'),
    })
if pb_def['dim'] == 3:
    regions.update({
        'Pn': ('vertices in (y < 0.001)', 'facet'),
        'Pf': ('vertices in (y > 0.999)', 'facet'),
        'PnYM': ('r.Pn *v r.YM', 'facet'),
        'PfYM': ('r.Pf *v r.YM', 'facet'),
        'bYm': ('vertices in (z < 0.001)', 'facet'),
        'bYp': ('vertices in (z > 0.999)', 'facet'),
    })
# periodic boundary conditions - matrix, Y-direction
epbcs.update({
    'periodic_yYM': (['PnYM', 'PfYM'], {'pM.0': 'pM.0'}, 'match_y_plane'),

```

(continues on next page)

(continued from previous page)

```

    })

reg_io = {}
ebcs_eta = {}
ebcs_gamma = {}

# generate regions, ebcs, epbcs
for ch, val in six.iteritems(pb_def['channels']):

    all_periodicY[ch] = ['periodic_%sY%s' % (ii, ch)
                        for ii in ['x', 'y'][:pb_def['dim']-1]]

    # channels: YA, fixedYA, bYMA (matrix/channel boundaries)
    regions.update({
        'Y' + ch: 'cells of group %d' % val['mat_el_grp'],
        'bYM' + ch: ('r.YM *v r.Y' + ch, 'facet', 'YM'),
        'PLY' + ch: ('r.Pl *v r.Y' + ch, 'facet'),
        'PrY' + ch: ('r.Pr *v r.Y' + ch, 'facet'),
    })

    if 'fix_nd_grp' in val:
        regions.update({
            'fixedY' + ch: ('vertices of group %d' % val['fix_nd_grp'][0],
                           'vertex'),
        })

    ebcs_eta[ch] = []
    for ch2, val2 in six.iteritems(pb_def['channels']):
        aux = 'eta%s_bYM%s' % (ch, ch2)
        if ch2 == ch:
            ebcs.update({aux: ('bYM' + ch2, {'pM.0': 1.0})})
        else:
            ebcs.update({aux: ('bYM' + ch2, {'pM.0': 0.0})})

        ebcs_eta[ch].append(aux)

    # boundary conditions
    # periodic boundary conditions - channels, X-direction
    epbcs.update({
        'periodic_xY' + ch: ([ 'PLY' + ch, 'PrY' + ch],
                             {'p%s.0' % ch: 'p%s.0' % ch},
                             'match_x_plane'),
    })

    if pb_def['dim'] == 3:
        regions.update({
            'PnY' + ch: ('r.Pn *v r.Y' + ch, 'facet'),
            'PfY' + ch: ('r.Pf *v r.Y' + ch, 'facet'),
        })
        # periodic boundary conditions - channels, Y-direction
        epbcs.update({
            'periodic_yY' + ch: ([ 'PnY' + ch, 'PfY' + ch],

```

(continues on next page)

(continued from previous page)

```

        {'p%s.0' % ch: 'p%s.0' % ch},
        'match_y_plane'),

    })

    reg_io[ch] = []
    aux_bY = []
    # channel: inputs/outputs
    for i_io in range(len(val['io_nd_grp'])):
        io = '%s_%d' % (ch, i_io+1)

        # regions
        aux = val['io_nd_grp'][i_io]
        if 'fix_nd_grp' in val and val['fix_nd_grp'][1] == aux:
            regions.update({
                'bY%s' % io: ('vertices of group %d +v r.fixedY%s' % (aux, ch),
                              'facet', 'Y%s' % ch),
            })
        else:
            regions.update({
                'bY%s' % io: ('vertices of group %d' % aux,
                              'facet', 'Y%s' % ch),
            })

        aux_bY.append('r.bY%s' % io)
        reg_io[ch].append('bY%s' % io)

    regions.update({
        'bY' + ch: (' +v '.join(aux_bY), 'facet', 'Y' + ch),
    })

    # channel: inputs/outputs
    for i_io in range(len(val['io_nd_grp'])):
        io = '%s_%d' % (ch, i_io + 1)
        ion = '%s_n%d' % (ch, i_io + 1)
        regions.update({
            'bY%s' % ion: ('r.bY%s -v r.bY%s' % (ch, io), 'facet', 'Y%s' % ch),
        })

        # boundary conditions
        aux = 'fix_p%s_bY%s' % (ch, ion)
        ebcs.update({
            aux: ('bY%s' % ion, {'p%s.0' % ch: 0.0}),
        })

        lcbcs.update({
            'imv' + ch: ('Y' + ch, {'ls%s.all' % ch: None}, None,
                              'integral_mean_value')
        })

    matk1, matk2 = get_mats(pb_def['param_kappa_m'], param_h, eps0, pb_def['dim'])

```

(continues on next page)

(continued from previous page)

```

materials = {
    'mat1M': ({'k': matk1},),
    'mat2M': ({'k': matk2},),
}

fields = {
    'corrector_M': ('real', 'scalar', 'YM', 1),
    'vel_M': ('real', 'vector', 'YM', 1),
    'vol_all': ('real', 'scalar', 'Y', 1),
}

variables = {
    'pM': ('unknown field', 'corrector_M'),
    'qM': ('test field', 'corrector_M', 'pM'),
    'Pi_M': ('parameter field', 'corrector_M', '(set-to-None)'),
    'corr_M': ('parameter field', 'corrector_M', '(set-to-None)'),
    'corr1_M': ('parameter field', 'corrector_M', '(set-to-None)'),
    'corr2_M': ('parameter field', 'corrector_M', '(set-to-None)'),
    'wM': ('parameter field', 'vel_M', '(set-to-None)'),
    'vol_all': ('parameter field', 'vol_all', '(set-to-None)'),
}

# generate regions for channel inputs/outputs
for ch, val in six.iteritems(pb_def['channels']):

    matk1, matk2 = get_mats(val['param_kappa_ch'], param_h,
                             eps0, pb_def['dim'])

    materials.update({
        'mat1' + ch: ({'k': matk1},),
        'mat2' + ch: ({'k': matk2},),
    })

    fields.update({
        'corrector_' + ch: ('real', 'scalar', 'Y' + ch, 1),
        'vel_' + ch: ('real', 'vector', 'Y' + ch, 1),
    })

    variables.update({
        'p' + ch: ('unknown field', 'corrector_' + ch),
        'q' + ch: ('test field', 'corrector_' + ch, 'p' + ch),
        'Pi_' + ch: ('parameter field', 'corrector_' + ch, '(set-to-None)'),
        'corr1_' + ch: ('parameter field', 'corrector_' + ch, '(set-to-None)'),
        'corr2_' + ch: ('parameter field', 'corrector_' + ch, '(set-to-None)'),
        'w' + ch: ('unknown field', 'vel_' + ch),
        # lagrange mutltipliers - integral mean value
        'ls' + ch: ('unknown field', 'corrector_' + ch),
        'lv' + ch: ('test field', 'corrector_' + ch, 'ls' + ch),
    })

options = {
    'coefs': 'coefs',
    'requirements': 'requirements',

```

(continues on next page)

(continued from previous page)

```

    'ls': 'ls', # linear solver to use
    'volumes': {
        'total': {
            'variables': ['vol_all'],
            'expression': ""ev_volume.iV.Y(vol_all)""",
        },
        'one': {
            'value': 1.0,
        }
    },
    'output_dir': './output',
    'split_results_by': 'region',
    'coefs_filename': 'coefs_perf_' + pb_def['name'],
    'coefs_info': {'eps0': eps0},
    'recovery_hook': 'recovery_perf',
    'multiprocessing': False,
}

for ipm in ['p', 'm']:
    options['volumes'].update({
        'bYM' + ipm: {
            'variables': ['pM'],
            'expression': "ev_volume.iS.bYM%s(pM)" % ipm,
        },
        'bY' + ipm: {
            'variables': ['vol_all'],
            'expression': "ev_volume.iS.bY%s(vol_all)" % ipm,
        }
    })

for ch in six.iterkeys(reg_io):
    for ireg in reg_io[ch]:
        options['volumes'].update({
            ireg: {
                'variables': ['p' + ch],
                'expression': "ev_volume.iS.%s(p%s)" % (ireg, ch),
            }
        })

coefs = {
    'vol_bYMpm': {
        'regions': ['bYMp', 'bYMm'],
        'expression': 'ev_volume.iS.%s(pM)',
        'class': cb.VolumeFractions,
    },
    'filenames': {},
}

requirements = {
    'corrs_one_YM': {
        'variable': ['pM'],
        'ebcs': ['gamma_pm_YMmchs', 'gamma_pm_bYMchs'],

```

(continues on next page)

(continued from previous page)

```

        'epbcs': [],
        'save_name': 'corrs_one_YM',
        'class': cb.CorrSetBCS,
        'is_linear': True,
    },
}

for ipm in ['p', 'm']:
    requirements.update({
        'corrs_gamma_' + ipm: {
            'requires': [],
            'ebcs': ['gamma_pm_bYMchs'],
            'epbcs': all_periodicYM,
            'equations': {
                'eq_gamma_pm': """dw_diffusion.iV.YM(mat2M.k, qM, pM) =
                                %e * dw_integrate.iS.bYM%s(qM)"""
                                % (1.0/param_h, ipm),
            },
            'class': cb.CorrOne,
            'save_name': 'corrs_%s_gamma_%s' % (pb_def['name'], ipm),
            'is_linear': True,
        },
    })

for ipm2 in ['p', 'm']:
    coefs.update({
        'H' + ipm + ipm2: { # test+
            'requires': ['corrs_gamma_' + ipm],
            'set_variables': [('corr_M', 'corrs_gamma_' + ipm, 'pM')],
            'expression': 'ev_integrate.iS.bYM%s(corr_M)' % ipm2,
            'set_volume': 'bYp',
            'class': cb.CoeffOne,
        },
    })

def get_channel(keys, bn):
    for ii in keys:
        if bn in ii:
            return ii[(ii.rfind(bn) + len(bn)):]

    return None

def set_corrpis(variables, ir, ic, mode, **kwargs):
    ch = get_channel(list(kwargs.keys()), 'pis_')
    pis = kwargs['pis_' + ch]
    corrs_pi = kwargs['corrs_pi' + ch]

    if mode == 'row':
        val = pis.states[ir]['p' + ch] + corrs_pi.states[ir]['p' + ch]
        variables['corr1_' + ch].set_data(val)

```

(continues on next page)

(continued from previous page)

```

elif mode == 'col':
    val = pis.states[ic]['p' + ch] + corrs_pi.states[ic]['p' + ch]
    variables['corr2_' + ch].set_data(val)

def set_corr_S(variables, ir, *args, **kwargs):
    ch = get_channel(list(kwargs.keys()), 'pis_')
    io = get_channel(list(kwargs.keys()), 'corrs_gamma_')

    pis = kwargs['pis_' + ch]
    corrs_gamma = kwargs['corrs_gamma_' + io]

    pi = pis.states[ir]['p' + ch]
    val = corrs_gamma.state['p' + ch]
    variables['corr1_' + ch].set_data(pi)
    variables['corr2_' + ch].set_data(val)

def set_corr_cc(variables, ir, *args, **kwargs):
    ch = get_channel(list(kwargs.keys()), 'pis_')
    pis = kwargs['pis_' + ch]
    corrs_pi = kwargs['corrs_pi' + ch]

    pi = pis.states[ir]['p' + ch]
    pi = pi - nm.mean(pi)
    val = pi + corrs_pi.states[ir]['p' + ch]
    variables['corr1_' + ch].set_data(val)

for ch, val in six.iteritems(pb_def['channels']):
    coefs.update({
        'G' + ch: { # test+
            'requires': ['corrs_one' + ch, 'corrs_eta' + ch],
            'set_variables': [('corr1_M', 'corrs_one' + ch, 'pM'),
                             ('corr2_M', 'corrs_eta' + ch, 'pM')],
            'expression': 'dw_diffusion.iV.YM(mat2M.k, corr1_M, corr2_M)',
            'class': cb.CoeffOne,
        },
        'K' + ch: { # test+
            'requires': ['pis_' + ch, 'corrs_pi' + ch],
            'set_variables': set_corrpis,
            'expression': 'dw_diffusion.iV.Y%s(mat2%s.k, corr1_%s, corr2_%s)\' \
                          % ((ch,) * 4),
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDimDim,
        },
    })

requirements.update({
    'pis_' + ch: {
        'variables': ['p' + ch],
        'class': cb.ShapeDim,
    }
})

```

(continues on next page)

(continued from previous page)

```

    },
    'corrs_one' + ch: {
        'variable': ['pM'],
        'ebcs': ebcs_eta[ch],
        'epbcs': [],
        'save_name': 'corrs_%s_one%s' % (pb_def['name'], ch),
        'class': cb.CorrSetBCS,
    },
    },
    'corrs_eta' + ch: {
        'ebcs': ebcs_eta[ch],
        'epbcs': all_periodicYM,
        'equations': {
            'eq_eta': 'dw_diffusion.iV.YM(mat2M.k, qM, pM) = 0',
        },
        'class': cb.CorrOne,
        'save_name': 'corrs_%s_eta%s' % (pb_def['name'], ch),
        'is_linear': True,
    },
    },
    'corrs_pi' + ch: {
        'requires': ['pis_' + ch],
        'set_variables': [('Pi_' + ch, 'pis_' + ch, 'p' + ch)],
        'ebcs': [],
        'epbcs': all_periodicY[ch],
        'lcbcs': ['imv' + ch],
        'equations': {
            'eq_pi': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                        + dw_dot.iV.Y%s(q%s, ls%s)
                        = - dw_diffusion.iV.Y%s(mat2%s.k, q%s, Pi_%s)"""
                        % ((ch,) * 11),
            'eq_imv': 'dw_dot.iV.Y%s(lv%s, p%s) = 0' % ((ch,) * 3),
        },
        'dim': pb_def['dim'] - 1,
        'class': cb.CorrDim,
        'save_name': 'corrs_%s_pi%s' % (pb_def['name'], ch),
    },
    },
})

for ipm in ['p', 'm']:
    coefs.update({
        'E' + ipm + ch: { # test+
            'requires': ['corrs_eta' + ch],
            'set_variables': [('corr_M', 'corrs_eta' + ch, 'pM')],
            'expression': 'ev_integrate.iS.bYM%s(corr_M)' % ipm,
            'set_volume': 'bYp',
            'class': cb.CoeffOne,
        },
        'F' + ipm + ch: { # test+
            'requires': ['corrs_one' + ch, 'corrs_gamma_' + ipm],
            'set_variables': [('corr1_M', 'corrs_one' + ch, 'pM'),
                             ('corr2_M', 'corrs_gamma_' + ipm, 'pM')],
            'expression': """dw_diffusion.iV.YM(mat2M.k, corr1_M, corr2_M)
                            - %e * ev_integrate.iS.bYM%s(corr1_M)"""
    })

```

(continues on next page)

(continued from previous page)

```

        % (1.0/param_h, ipm),
        'class': cb.CoeffOne,
    },
})

for i_io in range(len(val['io_nd_grp'])):
    io = '%s_%d' % (ch, i_io + 1)

    coefs.update({
        'S' + io: { # [Rohan1] (4.28), test+
            'requires': ['corrs_gamma_' + io, 'pis_' + ch],
            'set_variables': set_corr_S,
            'expression': 'dw_diffusion.iV.Y%s(mat2%s.k,corr1_%s,corr2_%s)'
                % ((ch,) * 4),
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDim,
        },
        'P' + io: { # test+
            'requires': ['pis_' + ch, 'corrs_pi' + ch],
            'set_variables': set_corr_cc,
            'expression': 'ev_integrate.iS.bY%s(corr1_%s)\'
                % (io, ch),
            'set_volume': 'bYp',
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDim,
        },
        'S_test' + io: {
            'requires': ['corrs_pi' + ch],
            'set_variables': [('corr1_' + ch, 'corrs_pi' + ch, 'p' + ch)],
            'expression': '%e * ev_integrate.iS.bY%s(corr1_%s)\'
                % (1.0 / param_h, io, ch),
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDim,
        },
    })

    requirements.update({
        'corrs_gamma_' + io: {
            'requires': [],
            'variables': ['p' + ch, 'q' + ch],
            'ebcs': [],
            'epbcs': all_periodicY[ch],
            'lcbcs': ['imv' + ch],
            'equations': {
                'eq_gamma': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                    + dw_dot.iV.Y%s(q%s, ls%s)
                    = %e * dw_integrate.iS.bY%s(q%s)"""
                    % ((ch,) * 7 + (1.0/param_h, io, ch)),
                'eq_imv': 'dw_dot.iV.Y%s(lv%s, p%s) = 0'
                    % ((ch,) * 3),
            },
            'class': cb.CorrOne,
        },
    })

```

(continues on next page)

(continued from previous page)

```

        'save_name': 'corrs_%s_gamma_%s' % (pb_def['name'], io),
    },
})

for i_io2 in range(len(val['io_nd_grp'])):
    io2 = '%s_%d' % (ch, i_io2 + 1)
    io12 = '%s_%d' % (io, i_io2 + 1)
    coefs.update({
        'R' + io12: { # test+
            'requires': ['corrs_gamma_' + io2],
            'set_variables': [('corr1_' + ch, 'corrs_gamma_' + io2,
                                'p' + ch)],
            'expression': 'ev_integrate.iS.bY%s(corr1_%s)'\
                           % (io, ch),
            'set_volume': 'bYp',
            'class': cb.CoeffOne,
        },
    })

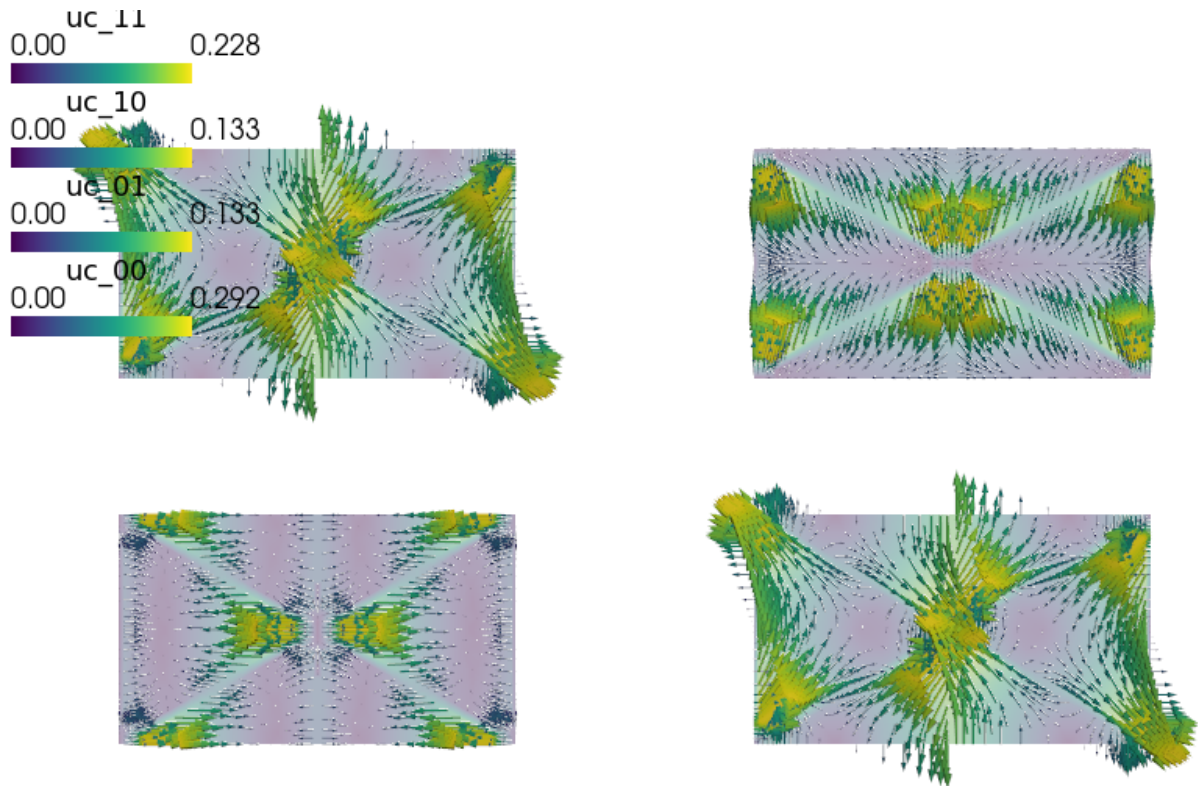
solvers = {
    'ls': ('ls.auto_direct', {'use_presolve' : True}),
    'newton': ('nls.newton', {
        'i_max': 1,
    })
}

```

homogenization/rs_correctors.py

Description

Compute homogenized elastic coefficients for a given microstructure.



source code

```
#!/usr/bin/env python
"""
Compute homogenized elastic coefficients for a given microstructure.
"""
from __future__ import print_function
from __future__ import absolute_import
from argparse import ArgumentParser
import sys
import six
sys.path.append('.')

import numpy as nm

from sfepy import data_dir
import sfepy.discrete.fem.periodic as per
from sfepy.homogenization.utils import define_box_regions

def define_regions(filename):
    """
    Define various subdomains for a given mesh file.
    """
```

(continues on next page)

(continued from previous page)

```

regions = {}
dim = 2

regions['Y'] = 'all'

eog = 'cells of group %d'
if filename.find('osteonT1') >= 0:
    mat_ids = [11, 39, 6, 8, 27, 28, 9, 2, 4, 14, 12, 17, 45, 28, 15]
    regions['Ym'] = ' +c '.join((eog % im) for im in mat_ids)
    wx = 0.865
    wy = 0.499

regions['Yc'] = 'r.Y -c r.Ym'

# Sides and corners.
regions.update(define_box_regions(2, (wx, wy)))

return dim, regions

def get_pars(ts, coor, mode=None, term=None, **kwargs):
    """
    Define material parameters: :math:D_{ijkl} (elasticity), in a given region.
    """
    if mode == 'qp':
        dim = coor.shape[1]
        sym = (dim + 1) * dim // 2

        out = {}

        # in 1e+10 [Pa]
        lam = 1.7
        mu = 0.3
        o = nm.array([1.] * dim + [0.] * (sym - dim), dtype = nm.float64)
        oot = nm.outer(o, o)
        out['D'] = lam * oot + mu * nm.diag(o + 1.0)

        for key, val in six.iteritems(out):
            out[key] = nm.tile(val, (coor.shape[0], 1, 1))

        channels_cells = term.region.domain.regions['Yc'].cells
        n_cell = term.region.get_n_cells()
        val = out['D'].reshape((n_cell, -1, 3, 3))
        val[channels_cells] *= 1e-1

        return out

##
# Mesh file.
filename_mesh = data_dir + '/meshes/2d/special/osteonT1_11.mesh'

##
# Define regions (subdomains, boundaries) - $Y$, $Y_i$, ...

```

(continues on next page)

(continued from previous page)

```

# depending on a mesh used.
dim, regions = define_regions(filename_mesh)

functions = {
    'get_pars' : (lambda ts, coors, **kwargs:
                  get_pars(ts, coors, **kwargs)),
    'match_x_plane' : (per.match_x_plane,),
    'match_y_plane' : (per.match_y_plane,),
    'match_z_plane' : (per.match_z_plane,),
    'match_x_line' : (per.match_x_line,),
    'match_y_line' : (per.match_y_line,),
}

##
# Define fields: 'displacement' in $Y$,
# 'pressure_m' in $Y_m$.
fields = {
    'displacement' : ('real', dim, 'Y', 1),
}

##
# Define corrector variables: unknown displacements: uc, test: vc
# displacement-like variables: Pi, Pi1, Pi2
variables = {
    'uc'      : ('unknown field', 'displacement', 0),
    'vc'      : ('test field', 'displacement', 'uc'),
    'Pi'      : ('parameter field', 'displacement', 'uc'),
    'Pi1'     : ('parameter field', 'displacement', None),
    'Pi2'     : ('parameter field', 'displacement', None),
}

##
# Periodic boundary conditions.
if dim == 3:
    epbcs = {
        'periodic_x' : ([ 'Left', 'Right'], {'uc.all' : 'uc.all'},
                        'match_x_plane'),
        'periodic_y' : ([ 'Near', 'Far'], {'uc.all' : 'uc.all'},
                        'match_y_plane'),
        'periodic_z' : ([ 'Top', 'Bottom'], {'uc.all' : 'uc.all'},
                        'match_z_plane'),
    }
else:
    epbcs = {
        'periodic_x' : ([ 'Left', 'Right'], {'uc.all' : 'uc.all'},
                        'match_y_line'),
        'periodic_y' : ([ 'Bottom', 'Top'], {'uc.all' : 'uc.all'},
                        'match_x_line'),
    }

##
# Dirichlet boundary conditions.

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    'fixed_u' : ('Corners', {'uc.all' : 0.0}),
}

##
# Material defining constitutive parameters of the microproblem.
materials = {
    'm' : 'get_pars',
}

##
# Numerical quadratures for volume (i3 - order 3) integral terms.
integrals = {
    'i3' : 3,
}

##
# Homogenized coefficients to compute.
def set_elastic(variables, ir, ic, mode, pis, corrs_rs):
    mode2var = {'row' : 'Pi1', 'col' : 'Pi2'}

    val = pis.states[ir, ic]['uc'] + corrs_rs.states[ir, ic]['uc']

    variables[mode2var[mode]].set_data(val)

coefs = {
    'E' : {
        'requires' : ['pis', 'corrs_rs'],
        'expression' : 'dw_lin_elastic.i3.Y(m.D, Pi1, Pi2)',
        'set_variables' : set_elastic,
    },
}

all_periodic = ['periodic_%s' % ii for ii in ['x', 'y', 'z'][:dim] ]
requirements = {
    'pis' : {
        'variables' : ['uc'],
    },
    ##
    # Steady state correctors  $\bar{\omega}^{rs}$ .
    'corrs_rs' : {
        'requires' : ['pis'],
        'save_variables' : ['uc'],
        'ebcs' : ['fixed_u'],
        'epbcs' : all_periodic,
        'equations' : {'eq' : """"dw_lin_elastic.i3.Y(m.D, vc, uc)
                        = - dw_lin_elastic.i3.Y(m.D, vc, Pi)""""},
        'set_variables' : [('Pi', 'pis', 'uc')],
        'save_name' : 'corrs_elastic',
        'is_linear' : True,
    },
}

```

(continues on next page)

(continued from previous page)

```

##
# Solvers.
solvers = {
    'ls' : ('ls.auto_direct', {'use_presolve' : True}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-8,
        'eps_r' : 1e-2,
    })
}

#####
# Mini-application below, computing the homogenized elastic coefficients.
helps = {
    'no_pauses' : 'do not make pauses',
}

def main():
    import os
    from sfepy.base.base import spause, output
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.discrete import Problem
    import sfepy.homogenization.coefs_base as cb

    parser = ArgumentParser(description=__doc__)
    parser.add_argument('--version', action='version', version='% (prog)s')
    parser.add_argument('-n', '--no-pauses',
                        action="store_true", dest='no_pauses',
                        default=False, help=helps['no_pauses'])
    options = parser.parse_args()

    if options.no_pauses:
        def spause(*args):
            output(*args)

    nm.set_printoptions(precision=3)

    spause(r""">>>
First, this file will be read in place of an input
(problem description) file.
Press 'q' to quit the example, press any other key to continue...""")
    required, other = get_standard_keywords()
    required.remove('equations')
    # Use this file as the input file.
    conf = ProblemConf.from_file(__file__, required, other)
    print(list(conf.to_dict().keys()))
    spause(r""">>>
...the read input as a dict (keys only for brevity).
['q'/other key to quit/continue...]""")

    spause(r""">>>

```

(continues on next page)

(continued from previous page)

```

Now the input will be used to create a Problem instance.
['q'/other key to quit/continue...]"")
    problem = Problem.from_conf(conf, init_equations=False)
    # The homogenization mini-apps need the output_dir.
    output_dir = ''
    problem.output_dir = output_dir
    print(problem)
    spause(r"">>>
...the Problem instance.
['q'/other key to quit/continue...]"")

    spause(r"">>>
The homogenized elastic coefficient  $E_{ijkl}$  is expressed
using  $\Pi$  operators, computed now. In fact, those operators are permuted
coordinates of the mesh nodes.
['q'/other key to quit/continue...]"")
    req = conf.requirements['pis']
    mini_app = cb.ShapeDimDim('pis', problem, req)
    mini_app.setup_output(save_formats=['vtk'])
    pis = mini_app()
    print(pis)
    spause(r"">>>
...the  $\Pi$  operators.
['q'/other key to quit/continue...]"")

    spause(r"">>>
Next,  $E_{ijkl}$  needs so called steady state correctors  $\bar{\omega}^{rs}$ ,
computed now.
['q'/other key to quit/continue...]"")
    req = conf.requirements['corrs_rs']

    save_name = req.get('save_name', '')
    name = os.path.join(output_dir, save_name)

    mini_app = cb.CorrDimDim('steady rs correctors', problem, req)
    mini_app.setup_output(save_formats=['vtk'])
    corrs_rs = mini_app(data={'pis': pis})
    print(corrs_rs)
    spause(r"">>>
...the  $\bar{\omega}^{rs}$  correctors.
The results are saved in: %s.%s

Try to display them with:

    sfepy-view %s.%s

['q'/other key to quit/continue...]" % (2 * (name, problem.output_format)))

    spause(r"">>>
Then the volume of the domain is needed.
['q'/other key to quit/continue...]"")
    volume = problem.evaluate('ev_volume.i3.Y(uc)')

```

(continues on next page)

(continued from previous page)

```

print(volume)

spause(r""">>>
...the volume.
['q'/other key to quit/continue...]""")

spause(r""">>>
Finally, $E_{ijkl}$ can be computed.
['q'/other key to quit/continue...]""")
mini_app = cb.CoeffSymSym('homogenized elastic tensor',
                           problem, conf.coefs['E'])
c_e = mini_app(volume, data={'pis': pis, 'corrs_rs' : corrs_rs})
print(r""">>>
The homogenized elastic coefficient $E_{ijkl}$, symmetric storage
with rows, columns in 11, 22, 12 ordering: """)
print(c_e)

if __name__ == '__main__':
    main()

```

large_deformation

large_deformation/active_fibres.py

Description

Nearly incompressible hyperelastic material model with active fibres.

Large deformation is described using the total Lagrangian formulation. Models of this kind can be used in biomechanics to model biological tissues, e.g. muscles.

Find $\underline{u}$ such that:

$$\int_{\Omega^{(0)}} \left(\underline{\underline{S}}^{\text{eff}}(\underline{u}) + K(J-1) J \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV = 0, \quad \forall \underline{v},$$

where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
J	$\det(\underline{\underline{F}})$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $\underline{\underline{C}} = \underline{\underline{F}}^T \underline{\underline{F}}$
$\underline{\underline{E}}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j} \right)$
$\underline{\underline{S}}^{\text{eff}}(\underline{u})$	effective second Piola-Kirchhoff stress tensor

The effective stress $\underline{\underline{S}}^{\text{eff}}(\underline{u})$ incorporates also the effects of the active fibres in two preferential directions:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} \left(\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1} \right) + \sum_{k=1}^2 \tau^k \underline{\underline{\omega}}^k.$$

The first term is the neo-Hookean term and the sum add contributions of the two fibre systems. The tensors $\underline{\underline{\omega}}^k = \underline{\underline{d}}^k \underline{\underline{d}}^k$ are defined by the fibre system direction vectors $\underline{\underline{d}}^k$ (unit).

For the one-dimensional tensions τ^k holds simply (^{k} omitted):

$$\tau = Af_{\max} \exp \left\{ -\left(\frac{\epsilon - \epsilon_{\text{opt}}}{s} \right)^2 \right\}, \epsilon = \underline{\underline{E}} : \underline{\underline{\omega}}.$$

Usage Examples

- Run with the Newton solver:

```
sfepy-run sfepy/examples/large_deformation/active_fibres.py
```

- Run with the matrix-free Newton-Krylov solver from SciPy:

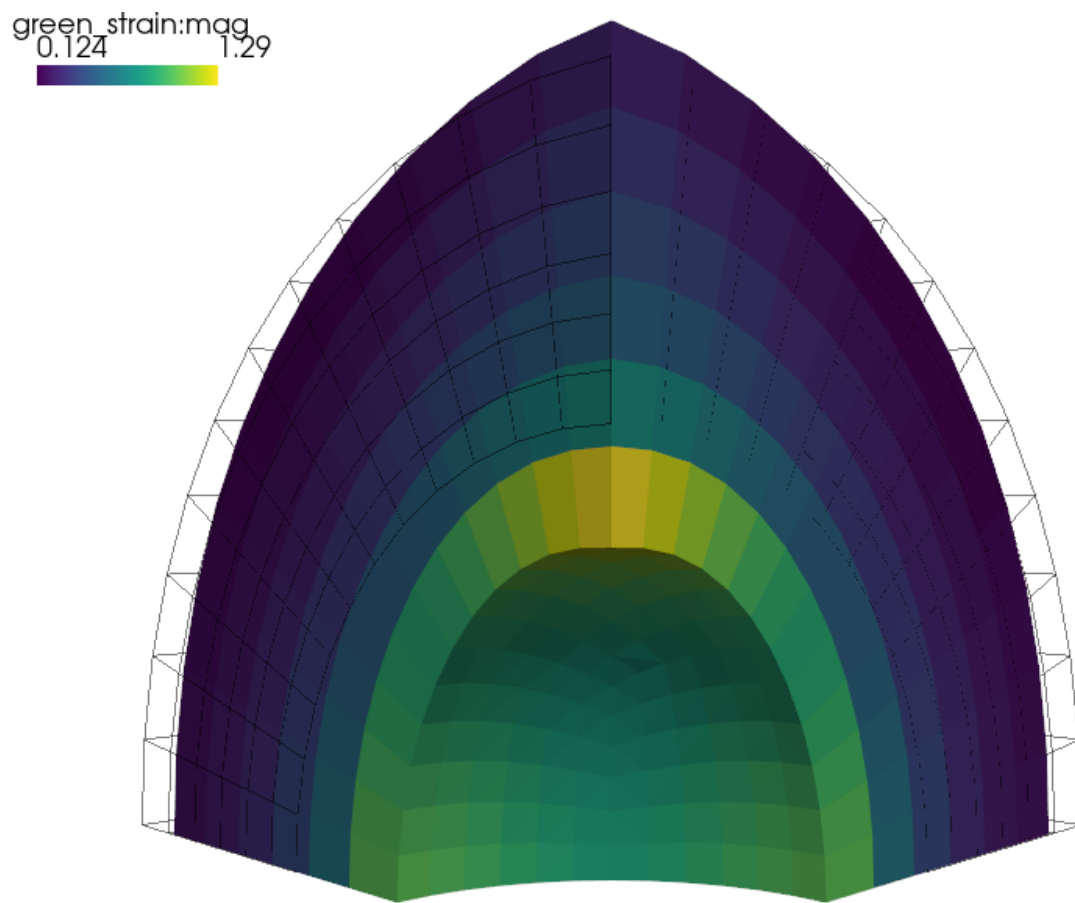
```
sfepy-run sfepy/examples/large_deformation/active_fibres.py -d solver=root
```

- Visualize the Green strain tensor magnitude on the deforming mesh:

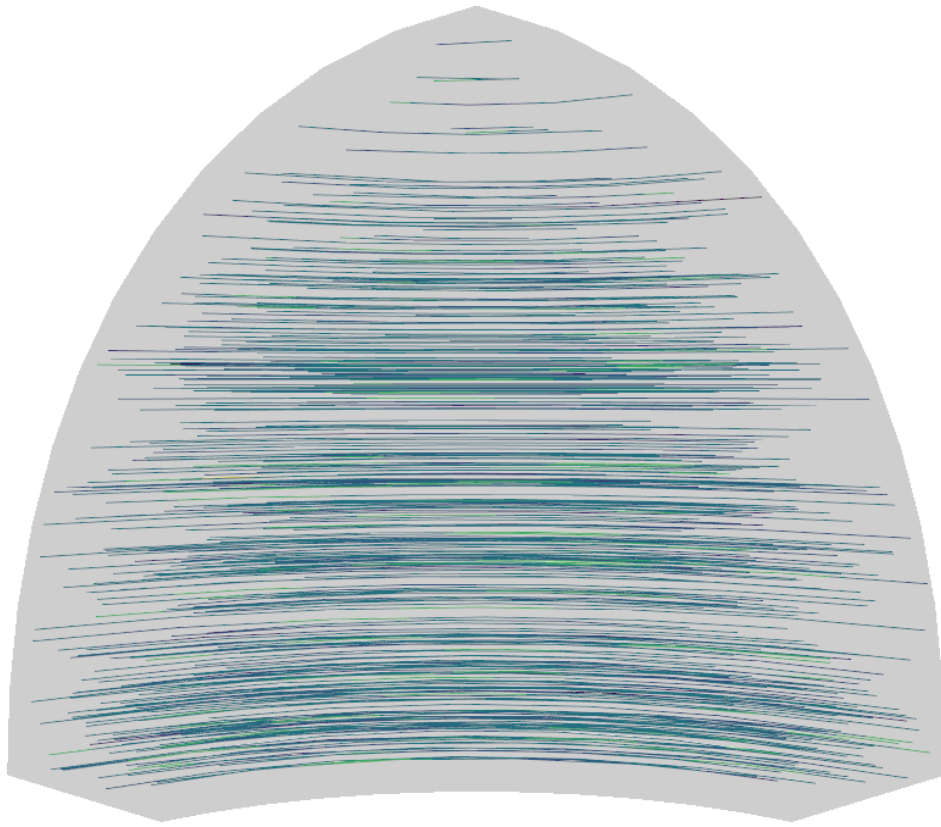
```
sfepy-view output/hsphere8.h5 -f green_strain:wu:f1:p0 1:vw:o0.3:p0 --color-  
↪limits=0.0,1.3
```

- Visualize the stresses in active fibres on the deforming mesh:

```
sfepy-view output/hsphere8.h5 -f f1_stress:wu:f1:p0 1:vw:o0.3:p0 --color-limits=0,20  
sfepy-view output/hsphere8.h5 -f f2_stress:wu:f1:p0 1:vw:o0.3:p0 --color-limits=0,30  
  
sfepy-view output/hsphere8.h5 -f f1_stress:wu:f1:p0 1:vw:o0.3:p0 f2_stress:wu:f1:p1 ↪  
↪1:vw:o0.3:p1 --grid-vector1="1.2,-1.2,0" --color-limits=0,30
```







source code

```
# -*- coding: utf-8 -*-
r"""
Nearly incompressible hyperelastic material model with active fibres.

Large deformation is described using the total Lagrangian formulation.
Models of this kind can be used in biomechanics to model biological
tissues, e.g. muscles.

Find  $\mathbf{u}$  such that:

.. math::
    \int_{\Omega} \left( S \cdot \text{eff}(\mathbf{u}) + K(J-1) \right) \delta C^{-1} \, dV = 0
    \quad \forall \mathbf{v} \;,

where

.. list-table::
   :widths: 20 80
```

(continues on next page)

(continued from previous page)

```

* - :math:`\ul{F}`
  - deformation gradient :math:`F_{ij} = \pdiff{x_i}{X_j}`
* - :math:`J`
  - :math:`\det(F)`
* - :math:`\ul{C}`
  - right Cauchy-Green deformation tensor :math:`C = F^T F`
* - :math:`\ul{E}(\ul{u})`
  - Green strain tensor :math:`E_{ij} = \frac{1}{2}(\pdiff{u_i}{X_j} + \pdiff{u_j}{X_i} + \pdiff{u_m}{X_i}\pdiff{u_m}{X_j})`
* - :math:`\ul{S}\eff(\ul{u})`
  - effective second Piola-Kirchhoff stress tensor

```

The effective stress :math:`\ul{S}\eff(\ul{u})` incorporates also the effects of the active fibres in two preferential directions:

```

.. math::
    \ul{S}\eff(\ul{u}) = \mu J^{-\frac{2}{3}}(\ul{I}
    - \frac{1}{3}\tr(\ul{C}) \ul{C}^{-1})
    + \sum_{k=1}^2 \tau^k \ul{\omega}^k
    \;.

```

The first term is the neo-Hookean term and the sum add contributions of the two fibre systems. The tensors :math:`\ul{\omega}^k = \ul{d}^k \ul{d}^k` are defined by the fibre system direction vectors :math:`\ul{d}^k` (unit).

For the one-dimensional tensions :math:`\tau^k` holds simply (:math:`^k` omitted):

```

.. math::
    \tau = A f_{\rm max} \exp\{\left\{-(\frac{\epsilon - \epsilon_{\rm opt}}{\epsilon - \epsilon_{\rm opt}})^2\right\}\} \mbox{ , } \epsilon = \ul{E} : \ul{\omega}
    \;.

```

Usage Examples

- Run with the Newton solver::

```
sfepy-run sfepy/examples/large_deformation/active_fibres.py
```

- Run with the matrix-free Newton-Krylov solver from SciPy::

```
sfepy-run sfepy/examples/large_deformation/active_fibres.py -d solver=root
```

- Visualize the Green strain tensor magnitude on the deforming mesh::

```
sfepy-view output/hsphere8.h5 -f green_strain:wu:f1:p0 1:vw:o0.3:p0 --color-limits=0.
↪ 0,1.3
```

- Visualize the stresses in active fibres on the deforming mesh::

(continues on next page)

(continued from previous page)

```

sfepy-view output/hsphere8.h5 -f f1_stress:wu:f1:p0 1:vw:o0.3:p0 --color-limits=0,20
sfepy-view output/hsphere8.h5 -f f2_stress:wu:f1:p0 1:vw:o0.3:p0 --color-limits=0,30

sfepy-view output/hsphere8.h5 -f f1_stress:wu:f1:p0 1:vw:o0.3:p0 f2_stress:wu:f1:p1,
↪1:vw:o0.3:p1 --grid-vector1="1.2,-1.2,0" --color-limits=0,30
"""
from functools import partial

import numpy as nm

from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.base.ioutils import edit_filename
from sfepy.homogenization.utils import define_box_regions

def get_pars_fibres(ts, coors, mode=None, which=0, vf=1.0, **kwargs):
    """
    Parameters
    -----
    ts : TimeStepper
        Time stepping info.
    coors : array_like
        The physical domain coordinates where the parameters should be defined.
    mode : 'qp' or 'special'
        Call mode.
    which : int
        Fibre system id.
    vf : float
        Fibre system volume fraction.
    """
    if mode != 'qp': return

    fmax = 100.0
    eps_opt = 0.01
    s = 1.0

    tt = ts.time * 2.0 * nm.pi

    x, y, z = coors[:,0], coors[:,1], coors[:,2]
    # Spherical coordinates.
    r = nm.sqrt(x**2 + y**2 + z**2)
    theta = nm.arccos(z / r)[:,None]
    phi = nm.arctan2(y, x)[:,None]
    act = 0.5 * (1.0 + nm.sin(tt - (0.5 * nm.pi)))

    if which == 0: # system 1 - meridians
        fdir = nm.c_[nm.cos(theta) * nm.cos(phi),
                     nm.cos(theta) * nm.sin(phi),
                     - nm.sin(theta)]

    elif which == 1: # system 2 - parallels
        fdir = nm.c_[-nm.sin(phi), nm.cos(phi), nm.zeros_like(phi)]

```

(continues on next page)

(continued from previous page)

```

else:
    raise ValueError(f'unknown fibre system {which}!')

nfdir = nm.linalg.norm(fdir, axis=1, keepdims=True)
fdir /= nm.where(nfdir > 0.0, nfdir, 1.0)

shape = (coors.shape[0], 1, 1)
out = {
    'fmax' : vf * nm.tile(fmax, shape),
    'eps_opt' : nm.tile(eps_opt, shape),
    's' : nm.tile(s, shape),
    'fdir' : fdir.reshape((coors.shape[0], 3, 1)),
    'act' : nm.tile(act, shape),
}

return out

def stress_strain(out, pb, state, extend=False):
    ev = partial(pb.evaluate, mode='el_avg', verbose=False)

    strain = ev('dw_tl_he_neohook.i.Omega(solid.mu, v, u)', term_mode='strain')
    out['green_strain'] = Struct(name='result', mode='cell', data=strain)

    stress = ev('dw_tl_he_neohook.i.Omega(solid.mu, v, u)', term_mode='stress')
    out['neohook_stress'] = Struct(name='result', mode='cell', data=stress)

    stress = ev('dw_tl_bulk_penalty.i.Omega(solid.K, v, u)', term_mode='stress')
    out['bulk_stress'] = Struct(name='output_data', mode='cell', data=stress)

    stress = ev(
        'dw_tl_fib_a.i.Omega(f1.fmax, f1.eps_opt, f1.s, f1.fdir, f1.act, v, u)',
        term_mode='stress',
    )
    out['f1_stress'] = Struct(name='result', mode='cell', data=stress)

    stress = ev(
        'dw_tl_fib_a.i.Omega(f2.fmax, f2.eps_opt, f2.s, f2.fdir, f2.act, v, u)',
        term_mode='stress',
    )
    out['f2_stress'] = Struct(name='result', mode='cell', data=stress)

    ts = pb.get_timestepper()
    if ts.step == 0:
        coors = pb.domain.get_mesh_coors()
        fout = {}
        for ii in range(2):
            aux = get_pars_fibres(ts, coors, mode='qp', which=ii, vf=1.0)
            fout[f'fdir{ii}'] = Struct(name='result', mode='vertex',
                                      data=aux['fdir'])

        filename = edit_filename(pb.get_output_name(),

```

(continues on next page)

(continued from previous page)

```

        suffix='_fibres', new_ext='.vtk')
    pb.save_state(filename, out=fout)

    return out

def define(solver='newton', refine=0, n_step=21, output_dir='output'):
    filename_mesh = data_dir + '/meshes/3d/hsphere8.vtk'

    vf_matrix = 0.5
    vf_fibres1 = 0.2
    vf_fibres2 = 0.3

    options = {
        'nls' : solver,
        'ls' : 'ls',
        'ts' : 'ts',
        'refinement_level' : refine,
        'save_times' : 'all',
        'post_process_hook' : 'stress_strain',
        'output_format' : 'h5',
        'output_dir' : output_dir,
    }

    fields = {
        'displacement': (nm.float64, 3, 'Omega', 1),
    }

    materials = {
        'solid' : ({
            'K' : vf_matrix * 1e3, # bulk modulus
            'mu' : vf_matrix * 20e0, # shear modulus of neoHookean term
        },),
        'f1' : 'get_pars_fibres1',
        'f2' : 'get_pars_fibres2',
    }

    functions = {
        'get_pars_fibres1' : (lambda ts, coors, mode=None, **kwargs:
            get_pars_fibres(ts, coors, mode=mode, which=0,
                vf=vf_fibres1, **kwargs)),
        'get_pars_fibres2' : (lambda ts, coors, mode=None, **kwargs:
            get_pars_fibres(ts, coors, mode=mode, which=1,
                vf=vf_fibres2, **kwargs)),
    }

    variables = {
        'u' : ('unknown field', 'displacement', 0),
        'v' : ('test field', 'displacement', 'u'),
    }

    dim = 3
    bbox = [[0] * dim, [25e-6] * dim]

```

(continues on next page)

(continued from previous page)

```

bregions = define_box_regions(dim, bbox[0], bbox[1])
regions = {
    'Omega' : 'all',
    'Left' : bregions['Left'],
    'Near' : bregions['Near'],
    'Bottom' : bregions['Bottom'],
}

ebcs = {
    'e0' : ('Left', {'u.0' : 0.0}),
    'e1' : ('Near', {'u.1' : 0.0}),
    'e2' : ('Bottom', {'u.2' : 0.0}),
}

integrals = {
    'i' : 2,
}

equations = {
    'balance' : """
        dw_tl_he_neohook.i.Omega(solid.mu, v, u)
        + dw_tl_bulk_penalty.i.Omega(solid.K, v, u)
        + dw_tl_fib_a.i.Omega(f1.fmax, f1.eps_opt, f1.s, f1.fdir, f1.act, v, u)
        + dw_tl_fib_a.i.Omega(f2.fmax, f2.eps_opt, f2.s, f2.fdir, f2.act, v, u)
        = 0
    """,
}

solvers = {
    'ls' : ('ls.auto_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 10,
        'eps_a' : 0.0,
        'eps_r' : 1e-5,
        'eps_mode' : 'or',
        'macheps' : 1e-16,
        'lin_red' : None, # Linear system error < (eps_a * lin_red).
        'ls_red' : 0.1,
        'ls_red_warp' : 0.001,
        'ls_on' : 10.0,
        'ls_min' : 1e-5,
        'check' : 0,
        'delta' : 1e-6,
        'report_status' : True,
    }),
    'root' : ('nls.scipy_root', {
        'method' : 'krylov',
        'use_jacobian' : False,
        'report_status' : True,
        'options' : {
            'ftol' : 1e-3,
        },
    }),
}

```

(continues on next page)

(continued from previous page)

```

    'ts' : ('ts.simple', {
        't0' : 0.0,
        't1' : 1.0 * n_step / 21.0,
        'dt' : None,
        'quasistatic' : True,
        'n_step' : n_step, # has precedence over dt.
        'verbose' : 1,
    }),
}

return locals()

```

large_deformation/balloon.py

Description

Inflation of a Mooney-Rivlin hyperelastic balloon.

This example serves as a verification of the membrane term (`dw_t1_membrane`, [TLMembraneTerm](#)) implementation.

Following Rivlin 1952 and Dumais, the analytical relation between a relative stretch $L = r/r_0$ of a thin (membrane) sphere made of the Mooney-Rivlin material of the undeformed radius r_0 , membrane thickness h_0 and the inner pressure p is

$$p = 4 \frac{h_0}{r_0} \left(\frac{1}{L} - \frac{1}{L^7} \right) (c_1 + c_2 L^2) ,$$

where c_1, c_2 are the Mooney-Rivlin material parameters.

In the equations below, only the surface of the domain is mechanically important - a stiff 2D membrane is embedded in the 3D space and coincides with the balloon surface. The volume is very soft, to simulate a fluid-filled cavity. A similar model could be used to model e.g. plant cells. The balloon surface is loaded by prescribing the inner volume change $\omega(t)$. The fluid pressure in the cavity is a single scalar value, enforced either by the 'integral_mean_value' linear combination condition, when `use_lcbcs` argument of `define()` is set to `True` (default), or by the L^2 constant approximation.

Find $\underline{u}(\underline{X})$ and a constant p such that:

- balance of forces:

$$\int_{\Omega^{(0)}} \left(\underline{S}^{\text{eff}}(\underline{u}) - p J \underline{C}^{-1} \right) : \delta \underline{E}(\underline{v}; \underline{v}) \, dV + \int_{\Gamma^{(0)}} \underline{S}^{\text{eff}}(\underline{u}) \delta \underline{E}(\underline{\tilde{u}}; \underline{\tilde{v}}) h_0 \, dS = 0 , \quad \forall \underline{v} \in [H_0^1(\Omega)]^3 ,$$

- volume conservation:

$$\int_{\Omega_0} [\omega(t) - J(u)] q \, dx = 0 \quad \forall q \in L^2(\Omega) ,$$

where

$\underline{F}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
J	$\det(F)$
$\underline{C}$	right Cauchy-Green deformation tensor $C = F^T F$
$\underline{E}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j} \right)$
$\underline{S}^{\text{eff}}(\underline{u})$	effective second Piola-Kirchhoff stress tensor

The effective stress $\underline{\underline{S}}^{\text{eff}}(\underline{u})$ is given by:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} \left(\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1} \right) + \kappa J^{-\frac{4}{3}} \left(\text{tr}(\underline{\underline{C}} \underline{\underline{I}} - \underline{\underline{C}} - \frac{2}{6} ((\text{tr} \underline{\underline{C}})^2 - \text{tr}(\underline{\underline{C}}^2)) \underline{\underline{C}}^{-1} \right).$$

The $\tilde{u}$ and $\tilde{v}$ variables correspond to $\underline{u}$, $\underline{v}$, respectively, transformed to the membrane coordinate frame.

Use the following command to show a comparison of the FEM solution with the above analytical relation (notice the nonlinearity of the dependence):

```
sfepy-run sfepy/examples/large_deformation/balloon.py -d 'plot=True'
```

or:

```
sfepy-run sfepy/examples/large_deformation/balloon.py -d 'plot=True, use_lcbcs=False'
```

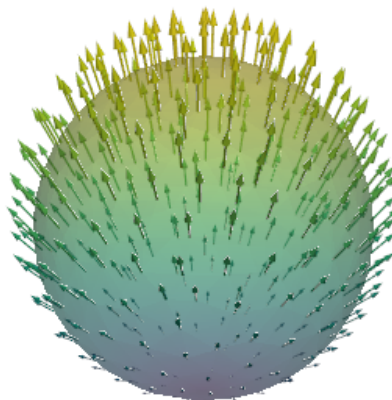
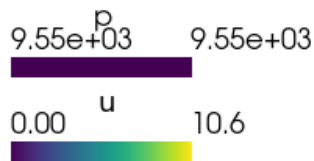
The agreement should be very good, even though the mesh is coarse.

View the results using:

```
sfepy-view unit_ball.h5 -f u:wu:s12:p0 p:s12:p1
```

This example uses the adaptive time-stepping solver ('ts.adaptive') with the default adaptivity function `adapt_time_step()`. Plot the used time steps by:

```
python3 sfepy/scripts/plot_times.py unit_ball.h5
```



source code

```

r"""
Inflation of a Mooney-Rivlin hyperelastic balloon.

This example serves as a verification of the membrane term (`dw_tl_membrane`,
:class:`TLMembraneTerm <sfe.py.terms.terms_membrane.TLMembraneTerm>`)
implementation.

Following Rivlin 1952 and Dumaïs, the analytical relation between a
relative stretch  $L = r / r_0$  of a thin (membrane) sphere made of the
Mooney-Rivlin material of the undeformed radius  $r_0$ , membrane
thickness  $h_0$  and the inner pressure  $p$  is

.. math::

    p = 4 \frac{h_0}{r_0} \left( \frac{1}{L} - \frac{1}{L^7} \right) (c_1 + c_2 L^2) \;,

where  $c_1$ ,  $c_2$  are the Mooney-Rivlin material parameters.

In the equations below, only the surface of the domain is mechanically
important - a stiff 2D membrane is embedded in the 3D space and coincides with
the balloon surface. The volume is very soft, to simulate a fluid-filled
cavity. A similar model could be used to model e.g. plant cells. The balloon
surface is loaded by prescribing the inner volume change  $\omega(t)$ .
The fluid pressure in the cavity is a single scalar value, enforced either by
the `integral_mean_value` linear combination condition, when `use_lcbcs`
argument of :func:define() is set to `True` (default), or by the
 $L^2$  constant approximation.

Find  $\mathbf{u}(\mathbf{X})$  and a constant  $p$  such that:

- balance of forces:

.. math::
    \int_{\Omega} \left( \mathbf{S} \cdot \mathbf{e}(\mathbf{u}) - p \mathbf{I} \right) : \mathbf{e}(\mathbf{v}) \, dV + \int_{\Gamma} \mathbf{S} \cdot \mathbf{e}(\tilde{\mathbf{u}}) : \mathbf{e}(\tilde{\mathbf{v}}) \, dS = 0 \;, \quad \forall \mathbf{v} \in [H^1_0(\Omega)]^3 \;,

- volume conservation:

.. math::
    \int_{\Omega} \left[ \omega(t) - J(\mathbf{u}) \right] \, dx = 0 \;, \quad \forall q \in L^2(\Omega) \;,

where

.. list-table::
   :widths: 20 80

   * - :math:`\mathbf{F}`

```

(continues on next page)

(continued from previous page)

```

- deformation gradient :math:`F_{ij} = \text{pdiff}\{x_i\}\{X_j\}`
* - :math:`J`
- :math:`\det(F)`
* - :math:`\text{ull}\{C\}`
- right Cauchy-Green deformation tensor :math:`C = F^T F`
* - :math:`\text{ull}\{E\}(\text{ul}\{u\})`
- Green strain tensor :math:`E_{ij} = \frac{1}{2}(\text{pdiff}\{u_i\}\{X_j\} + \text{pdiff}\{u_j\}\{X_i\} + \text{pdiff}\{u_m\}\{X_i\}\text{pdiff}\{u_m\}\{X_j\})`
* - :math:`\text{ull}\{S\}\text{eff}(\text{ul}\{u\})`
- effective second Piola-Kirchhoff stress tensor

```

The effective stress :math:`\text{ull}\{S\}\text{eff}(\text{ul}\{u\})` is given by:

```

.. math::
    \text{ull}\{S\}\text{eff}(\text{ul}\{u\}) = \mu J^{-\frac{2}{3}}(\text{ull}\{I}
    - \frac{1}{3}\text{tr}(\text{ull}\{C\}) \text{ull}\{C\}^{-1})
    + \kappa J^{-\frac{4}{3}}(\text{tr}(\text{ull}\{C\}\text{ull}\{I\} - \text{ull}\{C\}
    - \frac{2}{6}(\text{tr}(\text{ull}\{C\})^2 - \text{tr}(\text{ull}\{C\}^2))\text{ull}\{C\}^{-1})
    \;.

```

The :math:`\tilde{\text{ul}}\{u\}` and :math:`\tilde{\text{ul}}\{v\}` variables correspond to :math:`\text{ul}\{u\}`, :math:`\text{ul}\{v\}`, respectively, transformed to the membrane coordinate frame.

Use the following command to show a comparison of the FEM solution with the above analytical relation (notice the nonlinearity of the dependence)::

```
sfepy-run sfepy/examples/large_deformation/balloon.py -d 'plot=True'
```

or::

```
sfepy-run sfepy/examples/large_deformation/balloon.py -d 'plot=True, use_lcbcs=False'
```

The agreement should be very good, even though the mesh is coarse.

View the results using::

```
sfepy-view unit_ball.h5 -f u:wu:s12:p0 p:s12:p1
```

This example uses the adaptive time-stepping solver (``ts.adaptive``) with the default adaptivity function :func:`adapt\_time\_step` (`<sfepy.solvers.ts_solvers.adapt_time_step>`). Plot the used time steps by::

```

python3 sfepy/scripts/plot_times.py unit_ball.h5
"""
import os
import numpy as nm

from sfepy.base.base import Output
from sfepy.discrete.fem import MeshIO
from sfepy.linalg import get_coors_in_ball
from sfepy import data_dir

```

(continues on next page)

(continued from previous page)

```

output = Output('balloon:')

def get_nodes(coors, radius, eps, mode):
    if mode == 'ax1':
        centre = nm.array([0.0, 0.0, -radius], dtype=nm.float64)

    elif mode == 'ax2':
        centre = nm.array([0.0, 0.0, radius], dtype=nm.float64)

    elif mode == 'equator':
        centre = nm.array([radius, 0.0, 0.0], dtype=nm.float64)

    else:
        raise ValueError('unknown mode %s!' % mode)

    return get_coors_in_ball(coors, centre, eps)

def get_volume(ts, coors, region=None, **kwargs):
    rs = 1.0 + 1.0 * ts.time

    rv = get_rel_volume(rs)
    output('relative stretch:', rs)
    output('relative volume:', rv)

    out = nm.empty((coors.shape[0],), dtype=nm.float64)
    out.fill(rv)

    return out

def get_rel_volume(rel_stretch):
    """
    Get relative volume  $V/V_0$  from relative stretch  $r/r_0$  of a ball.
    """
    return nm.power(rel_stretch, 3.0)

def get_rel_stretch(rel_volume):
    """
    Get relative stretch  $r/r_0$  from relative volume  $V/V_0$  of a ball.
    """
    return nm.power(rel_volume, 1.0/3.0)

def get_balloon_pressure(rel_stretch, h0, r0, c1, c2):
    """
    Rivlin 1952 + Dumaïs:

    
$$P = 4h_0/r_0 * (1/L - 1/L^7) * (C_1 + L^2 * C_2)$$

    """
    L = rel_stretch
    p = 4.0 * h0 / r0 * (1.0/L - 1.0/L**7) * (c1 + c2 * L**2)

    return p

```

(continues on next page)

(continued from previous page)

```

def plot_radius(problem, state):
    import matplotlib.pyplot as plt

    from sfepy.postprocess.time_history import extract_time_history

    if problem.conf.use_lcbcs:
        ths, ts = extract_time_history('unit_ball.h5', 'p e 0')
        p = ths['p'][0]

    else:
        # Vertex 0 is not used in any cell...
        ths, ts = extract_time_history('unit_ball.h5', 'p n 1')
        p = ths['p'][1]

    L = 1.0 + ts.times[:p.shape[0]]

    L2 = 1.0 + nm.linspace(ts.times[0], ts.times[-1], 1000)
    p2 = get_balloon_pressure(L2, 1e-2, 1, 3e5, 3e4)

    plt.rcParams['lines.linewidth'] = 3
    plt.rcParams['font.size'] = 14

    plt.plot(L2, p2, 'r', label='theory')
    plt.plot(L, p, 'b*', ms=12, label='FEM')

    plt.title('Mooney-Rivlin hyperelastic balloon inflation')
    plt.xlabel(r'relative stretch $r/r_0$')
    plt.ylabel(r'pressure $p$')

    plt.legend(loc='best')
    plt.tight_layout()

    fig = plt.gcf()
    fig.savefig('balloon_pressure_stretch.pdf')

    plt.show()

def define(plot=False, use_lcbcs=True):
    filename_mesh = data_dir + '/meshes/3d/unit_ball.mesh'

    conf_dir = os.path.dirname(__file__)
    io = MeshIO.any_from_filename(filename_mesh, prefix_dir=conf_dir)
    bbox = io.read_bounding_box()
    dd = bbox[1] - bbox[0]

    radius = bbox[1, 0]
    eps = 1e-8 * dd[0]

    options = {
        'nls' : 'newton',
        'ls'  : 'ls',

```

(continues on next page)

(continued from previous page)

```

        'ts' : 'ts',
        'save_times' : 'all',
        'output_dir' : '.',
        'output_format' : 'h5',
    }

    if plot:
        options['post_process_hook_final'] = plot_radius

    fields = {
        'displacement': (nm.float64, 3, 'Omega', 1),
    }
    if use_lcbcs:
        fields['pressure'] = (nm.float64, 1, 'Omega', 0)

    else:
        fields['pressure'] = (nm.float64, 1, 'Omega', 0, 'L2', 'constant')

    materials = {
        'solid' : ({
            'mu' : 50, # shear modulus of neoHookean term
            'kappa' : 0.0, # shear modulus of Mooney-Rivlin term
        },),
        'walls' : ({
            'mu' : 3e5, # shear modulus of neoHookean term
            'kappa' : 3e4, # shear modulus of Mooney-Rivlin term
            'h0' : 1e-2, # initial thickness of wall membrane
        },),
    }

    variables = {
        'u' : ('unknown field', 'displacement', 0),
        'v' : ('test field', 'displacement', 'u'),
        'p' : ('unknown field', 'pressure', 1),
        'q' : ('test field', 'pressure', 'p'),
        'omega' : ('parameter field', 'pressure', {'setter' : 'get_volume'}),
    }

    regions = {
        'Omega' : 'all',
        'Ax1' : ('vertices by get_ax1', 'vertex'),
        'Ax2' : ('vertices by get_ax2', 'vertex'),
        'Equator' : ('vertices by get_equator', 'vertex'),
        'Surface' : ('vertices of surface', 'facet'),
    }

    ebc = {
        'fix1' : ('Ax1', {'u.all' : 0.0}),
        'fix2' : ('Ax2', {'u.[0, 1]' : 0.0}),
        'fix3' : ('Equator', {'u.1' : 0.0}),
    }

```

(continues on next page)

(continued from previous page)

```

if use_lcbcs:
    lcbcs = {
        'pressure' : ('Omega', {'p.all' : None},
                       None, 'integral_mean_value'),
    }

equations = {
    'balance'
        : """dw_tl_he_neohook.2.Omega(solid.mu, v, u)
            + dw_tl_he_mooney_rivlin.2.Omega(solid.kappa, v, u)
            + dw_tl_membrane.2.Surface(walls.mu, walls.kappa, walls.h0, v, u)
            + dw_tl_bulk_pressure.2.Omega(v, u, p)
            = 0""",
    'volume'
        : """dw_tl_volume.2.Omega(q, u)
            = dw_dot.2.Omega(q, omega)""",
}

solvers = {
    'ls' : ('ls.auto_direct', {
        # This setting causes a new factorization in each Newton step
        # without computing the digest.
        'use_presolve' : False,
        'use_mtx_digest' : False,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 8,
        'eps_a' : 1e-4,
        'eps_r' : 1e-8,
        'macheps' : 1e-16,
        # Do not check linear system solution accuracy in each step.
        'lin_red' : None,
        'ls_red' : 0.5,
        'ls_red_warp' : 0.1,
        'ls_on' : 100.0,
        'ls_min' : 1e-5,
        'check' : 0,
        'delta' : 1e-6,
        'report_status' : True,
    }),
    'ts' : ('ts.adaptive', {
        't0' : 0.0,
        't1' : 5.0,
        'dt' : None,
        'n_step' : 11,

        'dt_red_factor' : 0.8,
        'dt_red_max' : 1e-3,
        'dt_inc_factor' : 1.25,
        'dt_inc_on_iter' : 4,
        'dt_inc_wait' : 3,
    })
}

```

(continues on next page)

(continued from previous page)

```
        'verbose' : 1,
        'quasistatic' : True,
    }},
}

functions = {
    'get_ax1' : (lambda coors, domain:
        get_nodes(coors, radius, eps, 'ax1'),),
    'get_ax2' : (lambda coors, domain:
        get_nodes(coors, radius, eps, 'ax2'),),
    'get_equator' : (lambda coors, domain:
        get_nodes(coors, radius, eps, 'equator'),),
    'get_volume' : (get_volume,),
}

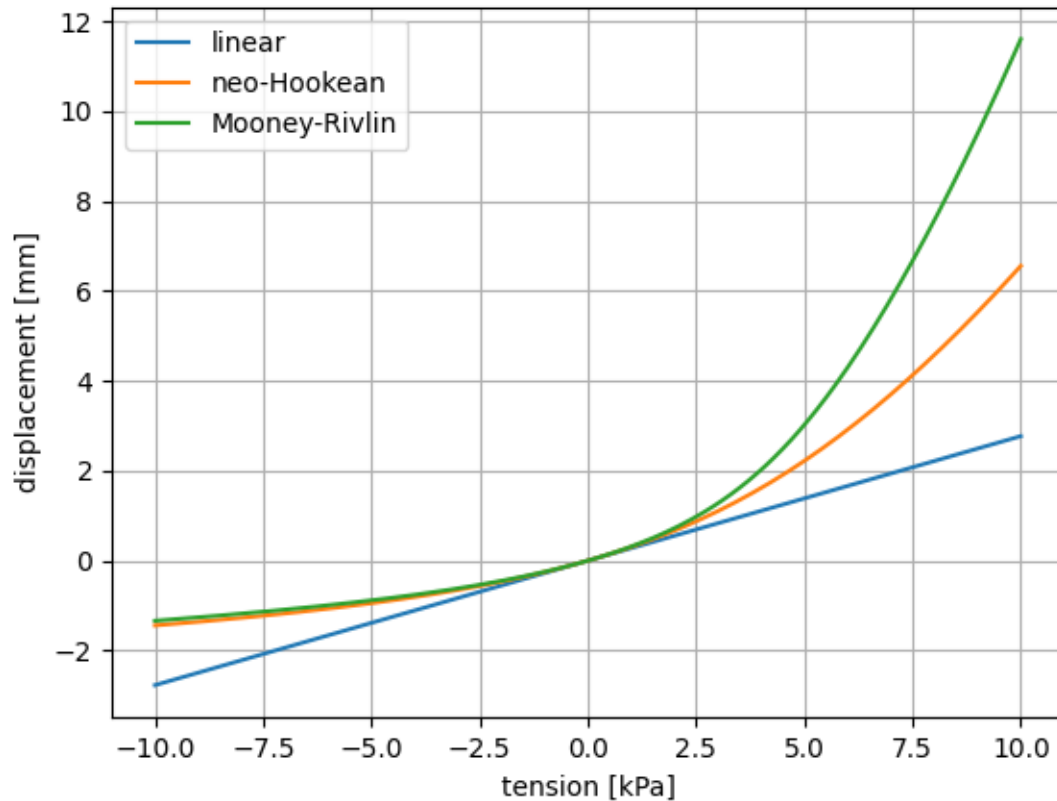
return locals()
```

large_deformation/compare_elastic_materials.py

Description

Compare various elastic materials w.r.t. uniaxial tension/compression test.

Requires Matplotlib.



source code

```
#!/usr/bin/env python
"""
Compare various elastic materials w.r.t. uniaxial tension/compression test.

Requires Matplotlib.
"""
from __future__ import absolute_import
from argparse import ArgumentParser, RawDescriptionHelpFormatter
import sys
import six
sys.path.append('.')

import numpy as nm

def define():
    """Define the problem to solve."""
    from sfepy.discrete.fem.meshio import UserMeshIO
    from sfepy.mesh.mesh_generators import gen_block_mesh
    from sfepy.mechanics.matcoefs import stiffness_from_lame

    def mesh_hook(mesh, mode):
        """
```

(continues on next page)

(continued from previous page)

```

Generate the block mesh.
"""
if mode == 'read':
    mesh = gen_block_mesh([2, 2, 3], [2, 2, 4], [0, 0, 1.5], name='el3',
                          verbose=False)

    return mesh

elif mode == 'write':
    pass

filename_mesh = UserMeshIO(mesh_hook)

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'save_times' : 'all',
}

functions = {
    'linear_tension' : (linear_tension,),
    'linear_compression' : (linear_compression,),
    'empty' : (lambda ts, coor, mode, region, ig: None,),
}

fields = {
    'displacement' : ('real', 3, 'Omega', 1),
}

# Coefficients are chosen so that the tangent stiffness is the same for all
# material for zero strains.
# Young modulus = 10 kPa, Poisson's ratio = 0.3
materials = {
    'solid' : ({
        'K' : 8.333, # bulk modulus
        'mu_nh' : 3.846, # shear modulus of neoHookean term
        'mu_mr' : 1.923, # shear modulus of Mooney-Rivlin term
        'kappa' : 1.923, # second modulus of Mooney-Rivlin term
        # elasticity for LE term
        'D' : stiffness_from_lame(dim=3, lam=5.769, mu=3.846),
    },),
    'load' : 'empty',
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Bottom' : ('vertices in (z < 0.1)', 'facet'),
}

```

(continues on next page)

(continued from previous page)

```

    'Top' : ('vertices in (z > 2.9)', 'facet'),
}

ebcs = {
    'fixb' : ('Bottom', {'u.all' : 0.0}),
    'fixt' : ('Top', {'u.[0,1]' : 0.0}),
}

integrals = {
    'i' : 1,
    'isurf' : 2,
}

equations = {
    'linear' : """dw_lin_elastic.i.Omega(solid.D, v, u)
                  = dw_surface_ltr.isurf.Top(load.val, v)""",
    'neo-Hookean' : """dw_tl_he_neohook.i.Omega(solid.mu_nh, v, u)
                      + dw_tl_bulk_penalty.i.Omega(solid.K, v, u)
                      = dw_surface_ltr.isurf.Top(load.val, v)""",
    'Mooney-Rivlin' : """dw_tl_he_neohook.i.Omega(solid.mu_mr, v, u)
                       + dw_tl_he_mooney_rivlin.i.Omega(solid.kappa, v, u)
                       + dw_tl_bulk_penalty.i.Omega(solid.K, v, u)
                       = dw_surface_ltr.isurf.Top(load.val, v)""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 5,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
    }),
    'ts' : ('ts.simple', {
        't0' : 0,
        't1' : 1,
        'dt' : None,
        'n_step' : 101, # has precedence over dt!
        'verbose' : 1,
    }),
}

return locals()

##
# Pressure tractions.
def linear_tension(ts, coor, mode=None, **kwargs):
    if mode == 'qp':
        val = nm.tile(0.1 * ts.step, (coor.shape[0], 1, 1))
        return {'val' : val}

def linear_compression(ts, coor, mode=None, **kwargs):
    if mode == 'qp':
        val = nm.tile(-0.1 * ts.step, (coor.shape[0], 1, 1))

```

(continues on next page)

(continued from previous page)

```

    return {'val' : val}

def store_top_u(displacements):
    """Function _store() will be called at the end of each loading step. Top
    displacements will be stored into `displacements`."""
    def _store(problem, ts, state):

        top = problem.domain.regions['Top']
        top_u = problem.get_variables()['u'].get_state_in_region(top)
        displacements.append(nm.mean(top_u[:,-1]))

    return _store

def solve_branch(problem, branch_function):
    displacements = {}
    for key, eq in six.iteritems(problem.conf.equations):
        problem.set_equations({key : eq})

        load = problem.get_materials()['load']
        load.set_function(branch_function)

        out = []
        problem.solve(save_results=False, step_hook=store_top_u(out))
        displacements[key] = nm.array(out, dtype=nm.float64)

    return displacements

helps = {
    'no_plot' : 'do not show plot window',
}

def main():
    from sfepy.base.base import output
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.discrete import Problem
    from sfepy.base.plotutils import plt

    parser = ArgumentParser(description=__doc__,
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--version', action='version', version='%s')
    parser.add_argument('-n', '--no-plot',
                        action="store_true", dest='no_plot',
                        default=False, help=helps['no_plot'])
    options = parser.parse_args()

    required, other = get_standard_keywords()
    # Use this file as the input file.
    conf = ProblemConf.from_file(__file__, required, other)

    # Create problem instance, but do not set equations.
    problem = Problem.from_conf(conf, init_equations=False)

```

(continues on next page)

(continued from previous page)

```

# Solve the problem. Output is ignored, results stored by using the
# step_hook.
u_t = solve_branch(problem, linear_tension)
u_c = solve_branch(problem, linear_compression)

# Get pressure load by calling linear_*() for each time step.
ts = problem.get_timestepper()
load_t = nm.array([linear_tension(ts, nm.array([[0.0]]), 'qp')['val']
                  for aux in ts.iter_from(0)],
                  dtype=nm.float64).squeeze()
load_c = nm.array([linear_compression(ts, nm.array([[0.0]]), 'qp')['val']
                  for aux in ts.iter_from(0)],
                  dtype=nm.float64).squeeze()

# Join the branches.
displacements = {}
for key in u_t.keys():
    displacements[key] = nm.r_[u_c[key][::-1], u_t[key]]
load = nm.r_[load_c[::-1], load_t]

if plt is None:
    output('matplotlib cannot be imported, printing raw data!')
    output(displacements)
    output(load)
else:
    legend = []
    for key, val in six.iteritems(displacements):
        plt.plot(load, val)
        legend.append(key)

    plt.legend(legend, loc = 2)
    plt.xlabel('tension [kPa]')
    plt.ylabel('displacement [mm]')
    plt.grid(True)

    plt.gcf().savefig('pressure_displacement.png')

    if not options.no_plot:
        plt.show()

if __name__ == '__main__':
    main()

```

large_deformation/gen_yeoh_tl_up_interactive.py

Description

Incompressible generalized Yeoh hyperelastic material model.

In this model, the deformation energy density per unit reference volume is given by

$$W = K_1 (\bar{I}_1 - 3)^m + K_2 (\bar{I}_1 - 3)^p + K_3 (\bar{I}_1 - 3)^q$$

where $\bar{I}_1$ is the first main invariant of the deviatoric part of the right Cauchy-Green deformation tensor $\underline{\underline{C}}$, the coefficients K_1, K_2, K_3 and exponents m, p, q are material parameters. Only a single term (`dw_tl_he_genyeoh`) is used in this example for the sake of simplicity.

Components of the second Piola-Kirchhoff stress are in the case of an incompressible material

$$S_{ij} = 2 \frac{\partial W}{\partial C_{ij}} - p F_{ik}^{-1} F_{kj}^{-T} ,$$

where p is the hydrostatic pressure.

The large deformation is described using the total Lagrangian formulation in this example. The incompressibility is treated by mixed displacement-pressure formulation. The weak formulation is: Find the displacement field $\underline{u}$ and pressure field p such that:

$$\begin{aligned} \int_{\Omega^{(0)}} \underline{\underline{S}}^{\text{eff}}(\underline{u}, p) : \underline{\underline{E}}(\underline{v}) \, dV &= 0 , \quad \forall \underline{v} , \\ \int_{\Omega^{(0)}} q (J(\underline{u}) - 1) \, dV &= 0 , \quad \forall q . \end{aligned}$$

The following formula holds for the axial true (Cauchy) stress in the case of uniaxial stress:

$$\sigma(\lambda) = \frac{2}{3} m K_1 \left(\lambda^2 + \frac{2}{\lambda} - 3 \right)^{m-1} \left(\lambda - \frac{1}{\lambda^2} \right) ,$$

where $\lambda = l/l_0$ is the prescribed stretch (l_0 and l being the original and deformed specimen length respectively).

The boundary conditions are set so that a state of uniaxial stress is achieved, i.e. appropriate components of displacement are fixed on the “Left”, “Bottom”, and “Near” faces and a monotonously increasing displacement is prescribed on the “Right” face. This prescribed displacement is then used to calculate λ and to convert the second Piola-Kirchhoff stress to the true (Cauchy) stress.

Note on material parameters

The three-term generalized Yeoh model is meant to be used for modelling of filled rubbers. The following choice of parameters is suggested [1] based on experimental data and stability considerations:

$$\begin{aligned} K_1 &> 0, \\ K_2 &< 0, \\ K_3 &> 0, \\ 0.7 &< m < 1, \\ m &< p < q. \end{aligned}$$

Usage Examples

Default options:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py -p
```

Using different mesh fineness:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--shape "5, 5, 5"
```

Different dimensions of the computational domain:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--dims "2, 1, 3"
```

Different length of time interval and/or number of time steps:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
-t 0,15,21
```

Use higher approximation order (the `-t` option to decrease the time step is required for convergence here):

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--order 2 -t 0,2,21
```

Change material parameters:

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py -m 2,1
```

View the results using sfepy-view

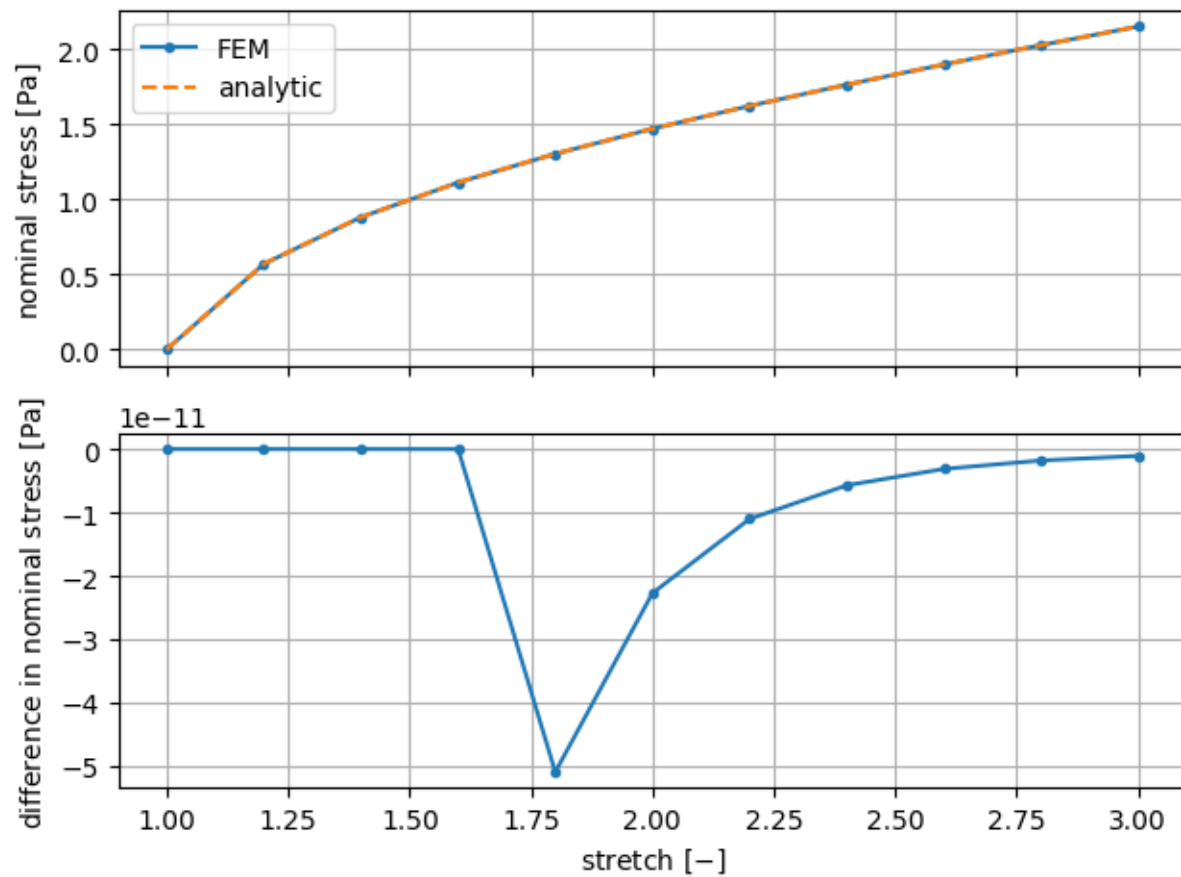
Show pressure on deformed mesh (use PgDn/PgUp to jump forward/back):

```
sfepy-view --fields=p:f1:wu:p1 domain.??\.vtk
```

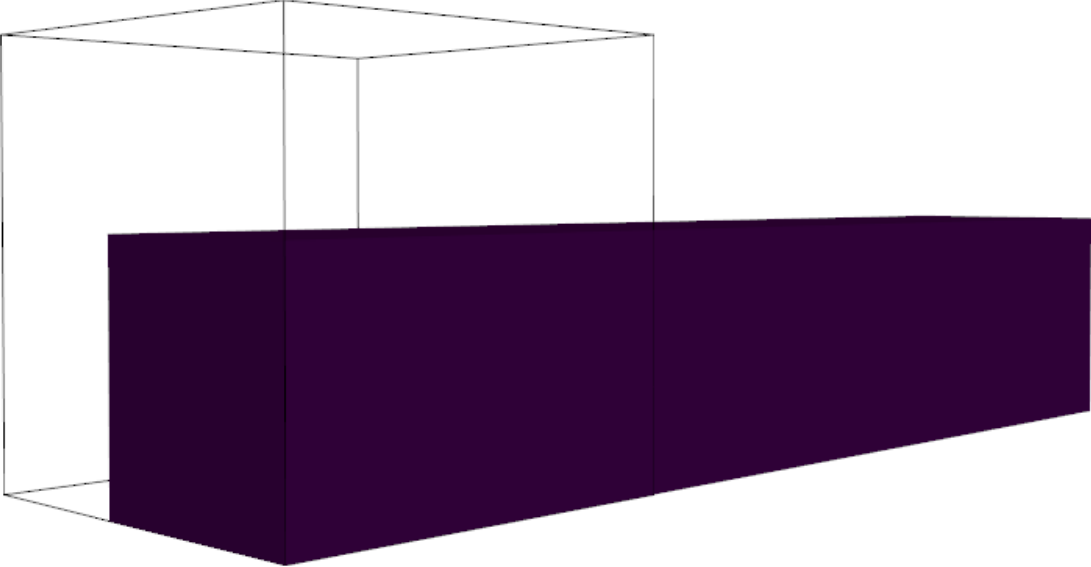
Show the axial component of stress (second Piola-Kirchhoff):

```
sfepy-view --fields=stress:c0 domain.??\.vtk
```

[1] Travis W. Hohenberger, Richard J. Windslow, Nicola M. Pugno, James J. C. Busfield. A constitutive Model For Both Low and High Strain Nonlinearities In Highly Filled Elastomers And Implementation With User-Defined Material Subroutines In Abaqus. Rubber Chemistry And Technology, Vol. 92, No. 4, Pp. 653-686 (2019)



0.717 stress 0.717



source code

```
#!/usr/bin/env python
r"""
Incompressible generalized Yeoh hyperelastic material model.

In this model, the deformation energy density per unit reference volume is
given by

.. math::
    W =
    K_1 \, \left( \overline{I_1} - 3 \right)^m
    + K_2 \, \left( \overline{I_1} - 3 \right)^p
    + K_3 \, \left( \overline{I_1} - 3 \right)^q

where  $\overline{I_1}$  is the first main invariant of the deviatoric part
of the right Cauchy-Green deformation tensor  $\mathbf{C}$ , the coefficients
 $K_1, K_2, K_3$  and exponents  $m, p, q$  are material parameters.
Only a single term (dw_tl_he_genyeoh
<sfePy.terms.terms_hyperelastic_tl.GenYeohTLTerm>) is used in this example for
the sake of simplicity.
```

Components of the second Piola-Kirchhoff stress are in the case of an

(continues on next page)

(continued from previous page)

incompressible material

```
.. math::
    S_{ij} = 2 \, \, \text{pdiff}\{W\}\{C_{ij}\} - p \, \, F^{\{-1\}}_{\{ik\}} \, \, F^{\{-T\}}_{\{kj\}} \, \, ;,
```

where p is the hydrostatic pressure.

The large deformation is described using the total Lagrangian formulation in this example. The incompressibility is treated by mixed displacement-pressure formulation. The weak formulation is:

Find the displacement field u and pressure field p such that:

```
.. math::
    \intl{\Omega}\{S\} \, \text{eff}(u, p) : \text{E}(v)
    \, \text{d}V = 0
    \, \, ;, \, \text{quad} \, \text{forall} \, v \, \, ;,

    \intl{\Omega}\{q\} \, (J(u)-1) \, \text{d}V = 0
    \, \, ;, \, \text{quad} \, \text{forall} \, q \, \, ;.
```

The following formula holds for the axial true (Cauchy) stress in the case of uniaxial stress:

```
.. math::
    \sigma(\lambda) =
        \frac{2}{3} \, \, m \, \, K_1 \, \,
        \left( \lambda^2 + \frac{2}{3} \lambda - 3 \right)^{m-1} \, \,
        \left( \lambda - \frac{1}{3} \lambda^2 \right) \, \, ;,
```

where $\lambda = l/l_0$ is the prescribed stretch (l_0 and l being the original and deformed specimen length respectively).

The boundary conditions are set so that a state of uniaxial stress is achieved, i.e. appropriate components of displacement are fixed on the "Left", "Bottom", and "Near" faces and a monotonously increasing displacement is prescribed on the "Right" face. This prescribed displacement is then used to calculate λ and to convert the second Piola-Kirchhoff stress to the true (Cauchy) stress.

Note on material parameters

The three-term generalized Yeoh model is meant to be used for modelling of filled rubbers. The following choice of parameters is suggested [1] based on experimental data and stability considerations:

```
:math:`K_1 > 0`,

:math:`K_2 < 0`,

:math:`K_3 > 0`,
```

(continues on next page)

(continued from previous page)

```
:math:`0.7 < m < 1`,
```

```
:math:`m < p < q`.
```

Usage Examples

Default options::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py -p
```

Using different mesh fineness::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--shape "5, 5, 5"
```

Different dimensions of the computational domain::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--dims "2, 1, 3"
```

Different length of time interval and/or number of time steps::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
-t 0,15,21
```

Use higher approximation order (the ``-t`` option to decrease the time step is required for convergence here)::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py \
--order 2 -t 0,2,21
```

Change material parameters::

```
python3 sfepy/examples/large_deformation/gen_yeoh_tl_up_interactive.py -m 2,1
```

View the results using ``sfepy-view``

Show pressure on deformed mesh (use PgDn/PgUp to jump forward/back)::

```
sfepy-view --fields=p:f1:wu:p1 domain.??.vtk
```

Show the axial component of stress (second Piola-Kirchhoff)::

```
sfepy-view --fields=stress:c0 domain.??.vtk
```

(continues on next page)

(continued from previous page)

```
[1] Travis W. Hohenberger, Richard J. Windslow, Nicola M. Pugno, James J. C.
Busfield. A constitutive Model For Both Low and High Strain Nonlinearities In
Highly Filled Elastomers And Implementation With User-Defined Material
Subroutines In Abaqus. Rubber Chemistry And Technology, Vol. 92, No. 4, Pp.
653-686 (2019)
"""
from __future__ import print_function, absolute_import
import argparse
import sys

SFEPY_DIR = '.'
sys.path.append(SFEPY_DIR)

import matplotlib.pyplot as plt
import numpy as np

from sfepy.base.base import IndexedStruct, Struct
from sfepy.discrete import (
    FieldVariable, Material, Integral, Function, Equation, Equations, Problem)
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.discrete.fem import FEDomain, Field
from sfepy.homogenization.utils import define_box_regions
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.solvers.ts_solvers import SimpleTimeSteppingSolver
from sfepy.terms import Term

DIMENSION = 3

def get_displacement(ts, coors, bc=None, problem=None):
    """
    Define the time-dependent displacement.
    """
    out = 1. * ts.time * coors[:, 0]
    return out

def _get_analytic_stress(stretches, coef, exp):
    out = np.array([
        2 * coef * exp * (stretch**2 + 2 / stretch - 3)**(exp - 1)
        * (stretch - stretch**-2)
        if (stretch**2 + 2 / stretch > 3) else 0.
        for stretch in stretches])
    return out

def plot_graphs(
    material_parameters, global_stress, global_displacement,
    undeformed_length):
    """
    Plot a comparison of the nominal stress computed by the FEM and using the
    analytic formula.
    """
```

(continues on next page)

(continued from previous page)

```

Parameters
-----
material_parameters : list or tuple of float
    The  $K_1$  coefficient and exponent  $m$ .
global_displacement
    The total displacement for each time step, from the FEM.
global_stress
    The true (Cauchy) stress for each time step, from the FEM.
undeformed_length : float
    The length of the undeformed specimen.
"""
coef, exp = material_parameters

stretch = 1 + np.array(global_displacement) / undeformed_length

# axial stress values
stress_fem_2pk = np.array([sig for sig in global_stress])
stress_fem = stress_fem_2pk * stretch
stress_analytic = _get_analytic_stress(stretch, coef, exp)

fig, (ax_stress, ax_difference) = plt.subplots(nrows=2, sharex=True)

ax_stress.plot(stretch, stress_fem, '.-', label='FEM')
ax_stress.plot(stretch, stress_analytic, '--', label='analytic')

ax_difference.plot(stretch, stress_fem - stress_analytic, '.-')

ax_stress.legend(loc='best').set_draggable(True)
ax_stress.set_ylabel(r'nominal stress  $\mathrm{[Pa]}$ ')
ax_stress.grid()

ax_difference.set_ylabel(r'difference in nominal stress  $\mathrm{[Pa]}$ ')
ax_difference.set_xlabel(r'stretch  $\mathrm{[-]}$ ')
ax_difference.grid()
plt.tight_layout()

return fig

def stress_strain(
    out, problem, _state, order=1, global_stress=None,
    global_displacement=None, **_):
    """
    Compute the stress and the strain and add them to the output.

    Parameters
    -----
    out : dict
        Holds the results of the finite element computation.
    problem : sfepy.discrete.Problem
    order : int
        The approximation order of the displacement field.
    global_displacement

```

(continues on next page)

(continued from previous page)

```

    Total displacement for each time step, current value will be appended.
global_stress
    The true (Cauchy) stress for each time step, current value will be
    appended.

Returns
-----
out : dict
"""
strain = problem.evaluate(
    'dw_tl_he_genyeoh.%d.Omega(m1.par, v, u)' % (2*order),
    mode='el_avg', term_mode='strain', copy_materials=False)

out['green_strain'] = Struct(
    name='output_data', mode='cell', data=strain, dofs=None)

stress_1 = problem.evaluate(
    'dw_tl_he_genyeoh.%d.Omega(m1.par, v, u)' % (2*order),
    mode='el_avg', term_mode='stress', copy_materials=False)
stress_p = problem.evaluate(
    'dw_tl_bulk_pressure.%d.Omega(v, u, p)' % (2*order),
    mode='el_avg', term_mode='stress', copy_materials=False)
stress = stress_1 + stress_p

out['stress'] = Struct(
    name='output_data', mode='cell', data=stress, dofs=None)

global_stress.append(stress[0, 0, 0, 0])
global_displacement.append(get_displacement(
    problem.ts, np.array([[1., 0, 0]]))[0])

return out

def main(cli_args):
    dims = parse_argument_list(cli_args.dims, float)
    shape = parse_argument_list(cli_args.shape, int)
    centre = parse_argument_list(cli_args.centre, float)
    material_parameters = parse_argument_list(cli_args.material_parameters,
                                              float)

    order = cli_args.order

    ts_vals = cli_args.ts.split(',')
    ts = {
        't0' : float(ts_vals[0]), 't1' : float(ts_vals[1]),
        'n_step' : int(ts_vals[2])}

    do_plot = cli_args.plot

    ### Mesh and regions ###
    mesh = gen_block_mesh(
        dims, shape, centre, name='block', verbose=False)
    domain = FEDomain('domain', mesh)

```

(continues on next page)

(continued from previous page)

```

omega = domain.create_region('Omega', 'all')

lbn, rtf = domain.get_mesh_bounding_box()
box_regions = define_box_regions(3, lbn, rtf)
regions = dict([
    [r, domain.create_region(r, box_regions[r][0], box_regions[r][1])]
    for r in box_regions])

### Fields ###
scalar_field = Field.from_args(
    'fu', np.float64, 'scalar', omega, approx_order=order-1)
vector_field = Field.from_args(
    'fv', np.float64, 'vector', omega, approx_order=order)

u = FieldVariable('u', 'unknown', vector_field, history=1)
v = FieldVariable('v', 'test', vector_field, primary_var_name='u')
p = FieldVariable('p', 'unknown', scalar_field, history=1)
q = FieldVariable('q', 'test', scalar_field, primary_var_name='p')

### Material ###
coefficient, exponent = material_parameters
m_1 = Material(
    'm1', par=[coefficient, exponent],
)

### Boundary conditions ###
x_sym = EssentialBC('x_sym', regions['Left'], {'u.0' : 0.0})
y_sym = EssentialBC('y_sym', regions['Near'], {'u.1' : 0.0})
z_sym = EssentialBC('z_sym', regions['Bottom'], {'u.2' : 0.0})
disp_fun = Function('disp_fun', get_displacement)
displacement = EssentialBC(
    'displacement', regions['Right'], {'u.0' : disp_fun})
ebcs = Conditions([x_sym, y_sym, z_sym, displacement])

### Terms and equations ###
integral = Integral('i', order=2*order+1)

term_1 = Term.new(
    'dw_tl_he_genyeoh(m1.par, v, u)',
    integral, omega, m1=m_1, v=v, u=u)
term_pressure = Term.new(
    'dw_tl_bulk_pressure(v, u, p)',
    integral, omega, v=v, u=u, p=p)

term_volume_change = Term.new(
    'dw_tl_volume(q, u)',
    integral, omega, q=q, u=u, term_mode='volume')
term_volume = Term.new(
    'dw_integrate(q)',
    integral, omega, q=q)

```

(continues on next page)

(continued from previous page)

```

eq_balance = Equation('balance', term_1 + term_pressure)
eq_volume = Equation('volume', term_volume_change - term_volume)
equations = Equations([eq_balance, eq_volume])

### Solvers ###
ls = ScipyDirect({})
nls_status = IndexedStruct()
nls = Newton(
    {'i_max' : 20},
    lin_solver=ls, status=nls_status
)

### Problem ###
pb = Problem('hyper', equations=equations)
pb.set_bcs(ebcs=ebcs)
pb.set_ics(ics=Conditions([]))
tss = SimpleTimeSteppingSolver(ts, nls=nls, context=pb)
pb.set_solver(tss)

### Solution ###
axial_stress = []
axial_displacement = []
def stress_strain_fun(*args, **kwargs):
    return stress_strain(
        *args, order=order, global_stress=axial_stress,
        global_displacement=axial_displacement, **kwargs)

pb.solve(save_results=True, post_process_hook=stress_strain_fun)

if do_plot:
    fig = plot_graphs(
        material_parameters, axial_stress, axial_displacement,
        undeformed_length=dims[0])

    fig.savefig('gen_yeoh_tl_up_comparison.png', bbox_inches='tight')

    if cli_args.show:
        plt.show()

def parse_argument_list(cli_arg, type_fun=None, value_separator=','):
    """
    Split the command-line argument into a list of items of given type.

    Parameters
    -----
    cli_arg : str
    type_fun : function
        A function to be called on each substring of `cli_arg`; default: str.
    value_separator : str
    """
    if type_fun is None:
        type_fun = str

```

(continues on next page)

(continued from previous page)

```

out = [type_fun(value) for value in cli_arg.split(value_separator)]
return out

def parse_args():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(
        description=__doc__,
        formatter_class=argparse.RawDescriptionHelpFormatter)
    parser.add_argument(
        '--order', type=int, default=1, help='The approximation order of the '
        'displacement field [default: %(default)s]')
    parser.add_argument(
        '-m', '--material-parameters', default='0.5, 0.9',
        help='Material parameters - coefficient and exponent - of a single '
        'term of the generalized Yeoh hyperelastic model. '
        '[default: %(default)s]')
    parser.add_argument(
        '--dims', default="1.0, 1.0, 1.0",
        help='Dimensions of the block [default: %(default)s]')
    parser.add_argument(
        '--shape', default='2, 2, 2',
        help='Shape (counts of nodes in x, y, z) of the block [default: '
        '%(default)s]')
    parser.add_argument(
        '--centre', default='0.5, 0.5, 0.5',
        help='Centre of the block [default: %(default)s]')
    parser.add_argument(
        '-p', '--plot', action='store_true', default=False,
        help='Whether to plot a comparison with analytical formula.')
    parser.add_argument(
        '-n', '--no-show', dest='show', action='store_false', default=True,
        help='Do not show matplotlib figures.')
    parser.add_argument(
        '-t', '--ts',
        type=str, default='0.0,2.0,11',
        help='Start time, end time, and number of time steps [default: '
        '"%(default)s"]')
    return parser.parse_args()

if __name__ == '__main__':
    args = parse_args()
    main(args)

```

large_deformation/hyperelastic.py**Description**

Nearly incompressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the total Lagrangian formulation. Models of this kind can be used to model e.g. rubber or some biological materials.

Find $\underline{u}$ such that:

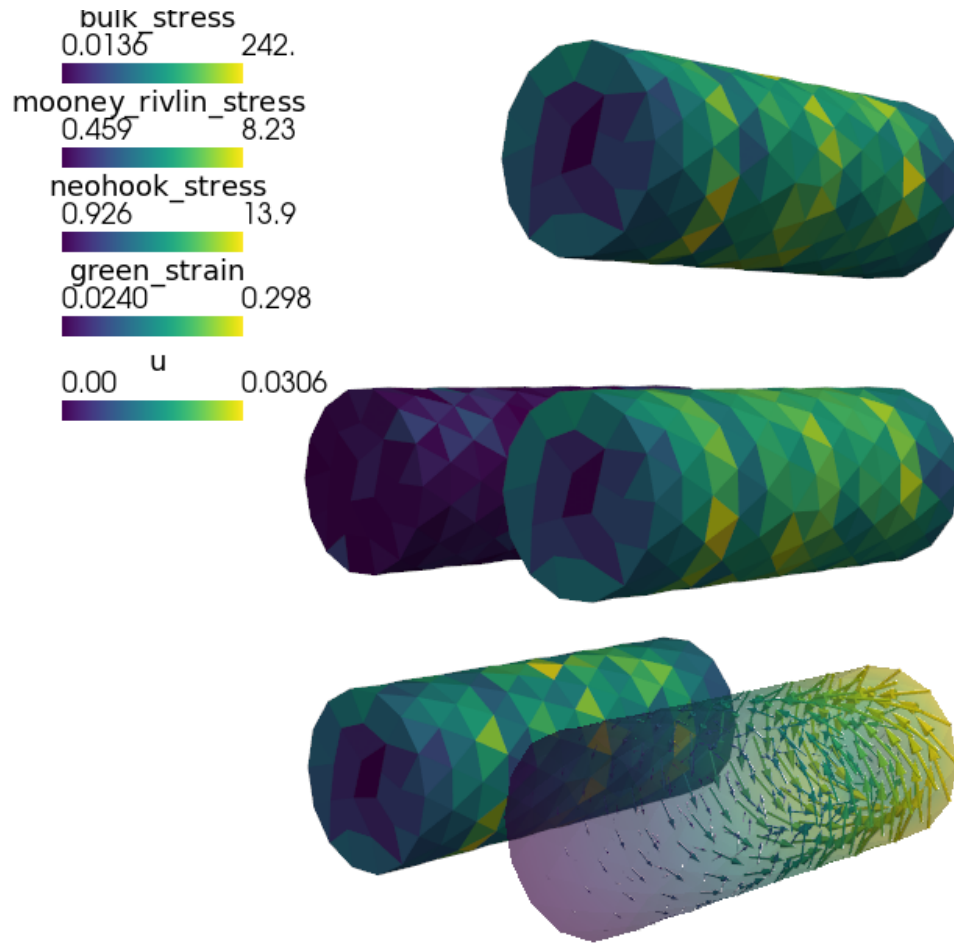
$$\int_{\Omega^{(0)}} \left(\underline{\underline{S}}^{\text{eff}}(\underline{u}) + K(J - 1) J \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV = 0, \quad \forall \underline{v},$$

where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
J	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = F^T F$
$\underline{\underline{E}}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j} \right)$
$\underline{\underline{S}}^{\text{eff}}(\underline{u})$	effective second Piola-Kirchhoff stress tensor

The effective stress $\underline{\underline{S}}^{\text{eff}}(\underline{u})$ is given by:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} \left(\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1} \right) + \kappa J^{-\frac{4}{3}} \left(\text{tr}(\underline{\underline{C}} \underline{\underline{I}} - \underline{\underline{C}} - \frac{2}{6} ((\text{tr} \underline{\underline{C}})^2 - \text{tr}(\underline{\underline{C}}^2)) \underline{\underline{C}}^{-1} \right).$$



source code

```
# -*- coding: utf-8 -*-
r"""
Nearly incompressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the total Lagrangian formulation.
Models of this kind can be used to model e.g. rubber or some biological
materials.

Find :math:\ul{u}` such that:

.. math::
\intl{\Omega\sub{suz}} \left( \ul{S}\eff(\ul{u})
+ K(J-1)\; J \ul{C}^{-1} \right) : \delta \ul{E}(\ul{v}) \difd{V}
= 0
\;, \quad \forall \ul{v} \;,

where

.. list-table::
:widths: 20 80
```

(continues on next page)

(continued from previous page)

```

* - :math:`\ul{F}`
  - deformation gradient :math:`F_{ij} = \pdiff{x_i}{X_j}`
* - :math:`J`
  - :math:`\det(F)`
* - :math:`\ul{C}`
  - right Cauchy-Green deformation tensor :math:`C = F^T F`
* - :math:`\ul{E}(\ul{u})`
  - Green strain tensor :math:`E_{ij} = \frac{1}{2}(\pdiff{u_i}{X_j} + \pdiff{u_j}{X_i} + \pdiff{u_m}{X_i}\pdiff{u_m}{X_j})`
* - :math:`\ul{S}\eff(\ul{u})`
  - effective second Piola-Kirchhoff stress tensor

```

The effective stress :math:`\ul{S}\eff(\ul{u})` is given by:

```

.. math::
    \ul{S}\eff(\ul{u}) = \mu J^{-\frac{2}{3}}(\ul{I}
    - \frac{1}{3}\tr(\ul{C}) \ul{C}^{-1})
    + \kappa J^{-\frac{4}{3}} (\tr(\ul{C})\ul{I} - \ul{C}
    - \frac{2}{6}((\tr(\ul{C}))^2 - \tr(\ul{C}^2))\ul{C}^{-1})
    \;.
"""
from __future__ import print_function
from __future__ import absolute_import
import numpy as nm

from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'save_times' : 'all',
    'post_process_hook' : 'stress_strain',
}

field_1 = {
    'name' : 'displacement',
    'dtype' : nm.float64,
    'shape' : 3,
    'region' : 'Omega',
    'approx_order' : 1,
}

material_1 = {
    'name' : 'solid',
    'values' : {
        'K' : 1e3, # bulk modulus
        'mu' : 20e0, # shear modulus of neoHookean term
        'kappa' : 10e0, # shear modulus of Mooney-Rivlin term
    }

```

(continues on next page)

(continued from previous page)

```

    }
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}

##
# Dirichlet BC + related functions.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0}),
    'r' : ('Right', {'u.0' : 0.0, 'u.[1,2]' : 'rotate_yz'}),
}

centre = nm.array( [0, 0], dtype = nm.float64 )

def rotate_yz(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d

    vec = coor[:,1:3] - centre

    angle = 10.0 * ts.step
    print('angle:', angle)

    mtx = rotation_matrix2d( angle )
    vec_rotated = nm.dot( vec, mtx )

    displacement = vec_rotated - vec

    return displacement

functions = {
    'rotate_yz' : (rotate_yz,),
}

def stress_strain( out, problem, state, extend = False ):
    from sfepy.base.base import Struct, debug

    ev = problem.evaluate
    strain = ev('dw_tl_he_neohook.i.Omega( solid.mu, v, u )',
                mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=strain, dofs=None)

    stress = ev('dw_tl_he_neohook.i.Omega( solid.mu, v, u )',

```

(continues on next page)

(continued from previous page)

```

        mode='el_avg', term_mode='stress')
out['neohook_stress'] = Struct(name='output_data',
                             mode='cell', data=stress, dofs=None)

stress = ev('dw_tl_he_mooney_rivlin.i.Omega( solid.kappa, v, u )',
           mode='el_avg', term_mode='stress')
out['mooney_rivlin_stress'] = Struct(name='output_data',
                                     mode='cell', data=stress, dofs=None)

stress = ev('dw_tl_bulk_penalty.i.Omega( solid.K, v, u )',
           mode='el_avg', term_mode='stress')
out['bulk_stress'] = Struct(name='output_data',
                           mode='cell', data=stress, dofs=None)

return out

##
# Balance of forces.
integral_1 = {
    'name' : 'i',
    'order' : 1,
}
equations = {
    'balance' : """dw_tl_he_neohook.i.Omega( solid.mu, v, u )
                  + dw_tl_he_mooney_rivlin.i.Omega( solid.kappa, v, u )
                  + dw_tl_bulk_penalty.i.Omega( solid.K, v, u )
                  = 0""",
}

##
# Solvers etc.
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 5,
    'eps_a'      : 1e-10,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 1.1,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
}

```

(continues on next page)

(continued from previous page)

```

solver_2 = {
    'name' : 'ts',
    'kind' : 'ts.simple',

    't0'    : 0,
    't1'    : 1,
    'dt'    : None,
    'n_step' : 11, # has precedence over dt!
    'verbose' : 1,
}

```

large_deformation/hyperelastic_tl_up_interactive.py

Description

Incompressible Mooney-Rivlin hyperelastic material model.

In this model, the deformation energy density per unit reference volume is given by

$$W = C_{(10)} (\bar{I}_1 - 3) + C_{(01)} (\bar{I}_2 - 3) ,$$

where $\bar{I}_1$ and $\bar{I}_2$ are the first and second main invariants of the deviatoric part of the right Cauchy-Green deformation tensor $\underline{\underline{C}}$. The coefficients $C_{(10)}$ and $C_{(01)}$ are material parameters.

Components of the second Piola-Kirchhoff stress are in the case of an incompressible material

$$S_{ij} = 2 \frac{\partial W}{\partial C_{ij}} - p F_{ik}^{-1} F_{kj}^{-T} ,$$

where p is the hydrostatic pressure.

The large deformation is described using the total Lagrangian formulation in this example. The incompressibility is treated by mixed displacement-pressure formulation. The weak formulation is: Find the displacement field $\underline{u}$ and pressure field p such that:

$$\begin{aligned} \int_{\Omega^{(0)}} \underline{\underline{S}}^{\text{eff}}(\underline{u}, p) : \underline{\underline{E}}(\underline{v}) \, dV &= 0 , \quad \forall \underline{v} , \\ \int_{\Omega^{(0)}} q (J(\underline{u}) - 1) \, dV &= 0 , \quad \forall q . \end{aligned}$$

The following formula holds for the axial true (Cauchy) stress in the case of uniaxial stress:

$$\sigma(\lambda) = 2 \left(C_{(10)} + \frac{C_{(01)}}{\lambda} \right) \left(\lambda^2 - \frac{1}{\lambda} \right) ,$$

where $\lambda = l/l_0$ is the prescribed stretch (l_0 and l being the original and deformed specimen length respectively).

The boundary conditions are set so that a state of uniaxial stress is achieved, i.e. appropriate components of displacement are fixed on the “Left”, “Bottom”, and “Near” faces and a monotonously increasing displacement is prescribed on the “Right” face. This prescribed displacement is then used to calculate λ and to convert the second Piola-Kirchhoff stress to the true (Cauchy) stress.

Note on material parameters

The relationship between material parameters used in the *SfePy* hyperelastic terms (*NeoHookeanTLTerm*, *MooneyRivlinTLTerm*) and the ones used in this example is:

$$\begin{aligned}\mu &= 2 C_{(10)} , \\ \kappa &= 2 C_{(01)} .\end{aligned}$$

Usage Examples

Default options:

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula:

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py -p
```

Using different mesh fineness:

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
--shape "5, 5, 5"
```

Different dimensions of the computational domain:

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
--dims "2, 1, 3"
```

Different length of time interval and/or number of time steps:

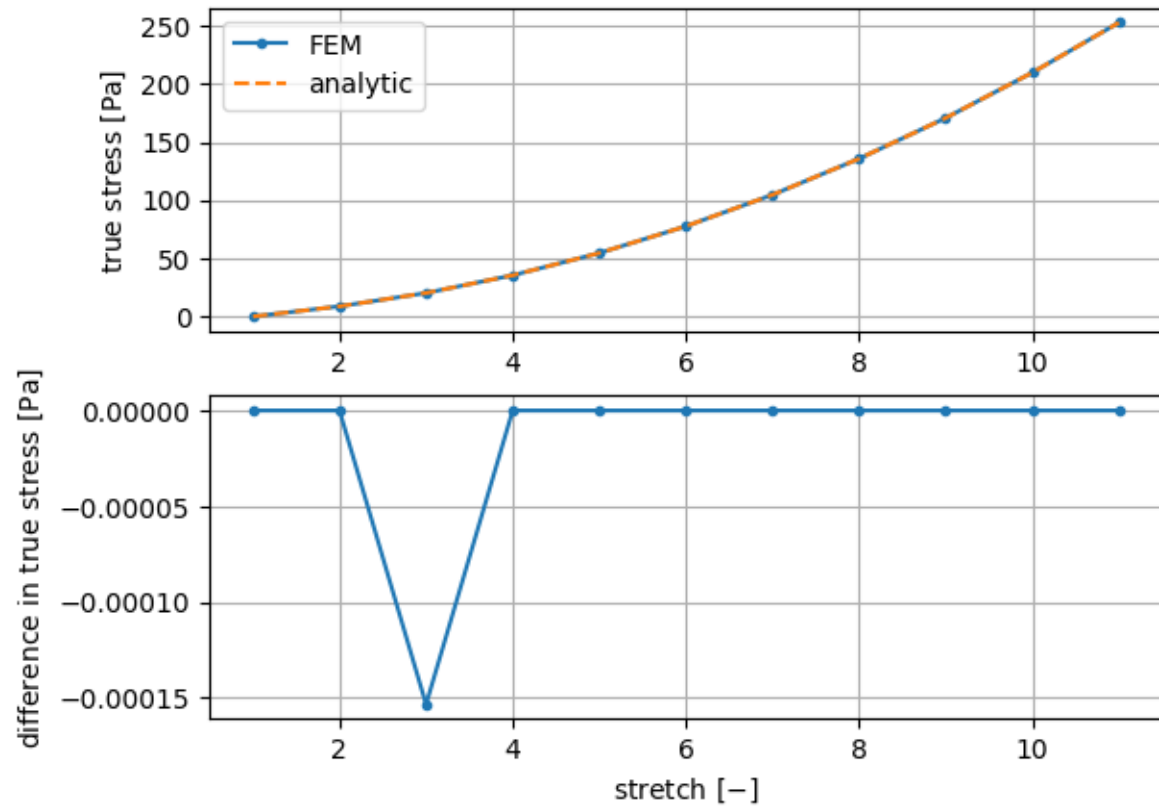
```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
-t 0,15,21
```

Use higher approximation order (the `-t` option to decrease the time step is required for convergence here):

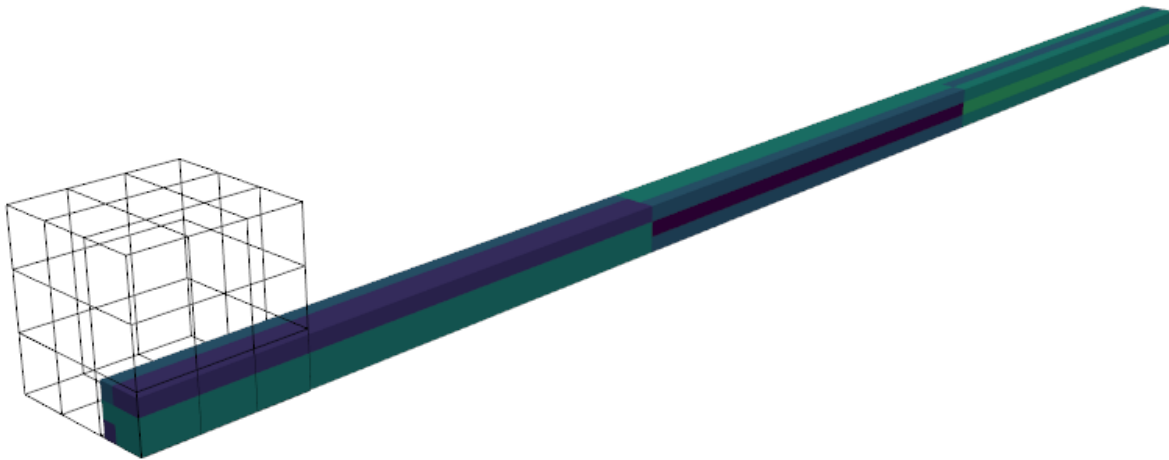
```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
--order 2 -t 0,2,21
```

Change material parameters:

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py -m 2,1
```

stress
2.09 2.09

source code

```
#!/usr/bin/env python
r"""
Incompressible Mooney-Rivlin hyperelastic material model.

In this model, the deformation energy density per unit reference volume is
given by

.. math::
    W = C_{(10)} \, \left( \overline{I_1} - 3 \right)
      + C_{(01)} \, \left( \overline{I_2} - 3 \right) \, ;,

where  $\overline{I_1}$  and  $\overline{I_2}$  are the first
and second main invariants of the deviatoric part of the right
Cauchy-Green deformation tensor  $\mathbf{C}$ . The coefficients
 $C_{(10)}$  and  $C_{(01)}$  are material parameters.

Components of the second Piola-Kirchhoff stress are in the case of an
incompressible material

.. math::
    S_{ij} = 2 \, \frac{\partial W}{\partial C_{ij}} - p \, \mathbf{F}^{-1}_{ik} \, \mathbf{F}^{-T}_{kj} \, ;,
```

(continues on next page)

(continued from previous page)

where p is the hydrostatic pressure.

The large deformation is described using the total Lagrangian formulation in this example. The incompressibility is treated by mixed displacement-pressure formulation. The weak formulation is:

Find the displacement field u and pressure field p such that:

```
.. math::
    \intl{\Omega} \text{eff}(u, p) : E(v)
    \text{d}V = 0
    \quad \forall v,
    \intl{\Omega} q, (J(u)-1) \text{d}V = 0
    \quad \forall q.
```

The following formula holds for the axial true (Cauchy) stress in the case of uniaxial stress:

```
.. math::
    \sigma(\lambda) =
    2 \left( C_{(10)} + \frac{C_{(01)}}{\lambda} \right) \lambda,
    \left( \lambda^2 - \frac{1}{\lambda} \right);
```

where $\lambda = l/l_0$ is the prescribed stretch (l_0 and l being the original and deformed specimen length respectively).

The boundary conditions are set so that a state of uniaxial stress is achieved, i.e. appropriate components of displacement are fixed on the "Left", "Bottom", and "Near" faces and a monotonously increasing displacement is prescribed on the "Right" face. This prescribed displacement is then used to calculate λ and to convert the second Piola-Kirchhoff stress to the true (Cauchy) stress.

Note on material parameters

The relationship between material parameters used in the *SfePy* hyperelastic terms (`NeoHookeanTLTerm`

```
<sfe.py.terms.hyperelastic_tl.NeoHookeanTLTerm>,
:class:MooneyRivlinTLTerm
<sfe.py.terms.hyperelastic_tl.MooneyRivlinTLTerm>)
and the ones used in this example is:
```

```
.. math::
    \mu = 2, C_{(10)};
    \kappa = 2, C_{(01)}.
```

Usage Examples

(continues on next page)

(continued from previous page)

Default options::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py -p
```

Using different mesh fineness::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
  --shape "5, 5, 5"
```

Different dimensions of the computational domain::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
  --dims "2, 1, 3"
```

Different length of time interval and/or number of time steps::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
  -t 0,15,21
```

Use higher approximation order (the ``-t`` option to decrease the time step is required for convergence here)::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py \
  --order 2 -t 0,2,21
```

Change material parameters::

```
$ python sfepy/examples/large_deformation/hyperelastic_tl_up_interactive.py -m 2,1
"""
```

```
from __future__ import print_function, absolute_import
import argparse
import sys
```

```
SFEPY_DIR = '.'
sys.path.append(SFEPY_DIR)
```

```
import matplotlib.pyplot as plt
import numpy as np
```

```
from sfepy.base.base import IndexedStruct, Struct
from sfepy.discrete import (
    FieldVariable, Material, Integral, Function, Equation, Equations, Problem)
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.discrete.fem import FEDomain, Field
from sfepy.homogenization.utils import define_box_regions
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.solvers.ls import ScipyDirect
```

(continues on next page)

(continued from previous page)

```

from sfepy.solvers.nls import Newton
from sfepy.solvers.ts_solvers import SimpleTimeSteppingSolver
from sfepy.terms import Term

DIMENSION = 3

def get_displacement(ts, coors, bc=None, problem=None):
    """
    Define the time-dependent displacement.
    """
    out = 1. * ts.time * coors[:, 0]
    return out

def plot_graphs(
    material_parameters, global_stress, global_displacement,
    undeformed_length):
    """
    Plot a comparison of the true stress computed by the FEM and using the
    analytic formula.

    Parameters
    -----
    material_parameters : list or tuple of float
        The C10 and C01 coefficients.
    global_displacement
        The total displacement for each time step, from the FEM.
    global_stress
        The true (Cauchy) stress for each time step, from the FEM.
    undeformed_length : float
        The length of the undeformed specimen.
    """
    c10, c01 = material_parameters

    stretch = 1 + np.array(global_displacement) / undeformed_length

    # axial stress values
    stress_fem_2pk = np.array([sig for sig in global_stress])
    stress_fem = stress_fem_2pk * stretch**2
    stress_analytic = 2 * (c10 + c01/stretch) * (stretch**2 - 1./stretch)

    fig = plt.figure()
    ax_stress = fig.add_subplot(211)
    ax_difference = fig.add_subplot(212)

    ax_stress.plot(stretch, stress_fem, '.-', label='FEM')
    ax_stress.plot(stretch, stress_analytic, '--', label='analytic')

    ax_difference.plot(stretch, stress_fem - stress_analytic, '.-')

    ax_stress.legend(loc='best').set_draggable(True)
    ax_stress.set_ylabel(r'true stress $\mathrm{[Pa]}$')
    ax_stress.grid()

```

(continues on next page)

(continued from previous page)

```

ax_difference.set_ylabel(r'difference in true stress $\mathrm{[Pa]}$')
ax_difference.set_xlabel(r'stretch $\mathrm{[-]}$')
ax_difference.grid()

return fig

def stress_strain(
    out, problem, _state, order=1, global_stress=None,
    global_displacement=None, **_):
    """
    Compute the stress and the strain and add them to the output.

    Parameters
    -----
    out : dict
        Holds the results of the finite element computation.
    problem : sfepy.discrete.Problem
    order : int
        The approximation order of the displacement field.
    global_displacement
        Total displacement for each time step, current value will be appended.
    global_stress
        The true (Cauchy) stress for each time step, current value will be
        appended.

    Returns
    -----
    out : dict
    """
    strain = problem.evaluate(
        'dw_tl_he_neohook.%d.Omega(m.mu, v, u)' % (2*order),
        mode='el_avg', term_mode='strain', copy_materials=False)

    out['green_strain'] = Struct(
        name='output_data', mode='cell', data=strain, dofs=None)

    stress_l0 = problem.evaluate(
        'dw_tl_he_neohook.%d.Omega(m.mu, v, u)' % (2*order),
        mode='el_avg', term_mode='stress', copy_materials=False)
    stress_01 = problem.evaluate(
        'dw_tl_he_mooney_rivlin.%d.Omega(m.kappa, v, u)' % (2*order),
        mode='el_avg', term_mode='stress', copy_materials=False)
    stress_p = problem.evaluate(
        'dw_tl_bulk_pressure.%d.Omega(v, u, p)' % (2*order),
        mode='el_avg', term_mode='stress', copy_materials=False)
    stress = stress_l0 + stress_01 + stress_p

    out['stress'] = Struct(
        name='output_data', mode='cell', data=stress, dofs=None)

    global_stress.append(stress[0, 0, 0, 0])

```

(continues on next page)

(continued from previous page)

```

global_displacement.append(np.max(out['u'].data[:, 0]))

return out

def main(cli_args):
    dims = parse_argument_list(cli_args.dims, float)
    shape = parse_argument_list(cli_args.shape, int)
    centre = parse_argument_list(cli_args.centre, float)
    material_parameters = parse_argument_list(cli_args.material_parameters,
                                              float)

    order = cli_args.order

    ts_vals = cli_args.ts.split(',')
    ts = {
        't0' : float(ts_vals[0]), 't1' : float(ts_vals[1]),
        'n_step' : int(ts_vals[2])}

    do_plot = cli_args.plot

    ### Mesh and regions ###
    mesh = gen_block_mesh(
        dims, shape, centre, name='block', verbose=False)
    domain = FEDomain('domain', mesh)

    omega = domain.create_region('Omega', 'all')

    lbn, rtf = domain.get_mesh_bounding_box()
    box_regions = define_box_regions(3, lbn, rtf)
    regions = dict([
        [r, domain.create_region(r, box_regions[r][0], box_regions[r][1])]
        for r in box_regions])

    ### Fields ###
    scalar_field = Field.from_args(
        'fu', np.float64, 'scalar', omega, approx_order=order-1)
    vector_field = Field.from_args(
        'fv', np.float64, 'vector', omega, approx_order=order)

    u = FieldVariable('u', 'unknown', vector_field, history=1)
    v = FieldVariable('v', 'test', vector_field, primary_var_name='u')
    p = FieldVariable('p', 'unknown', scalar_field, history=1)
    q = FieldVariable('q', 'test', scalar_field, primary_var_name='p')

    ### Material ###
    c10, c01 = material_parameters
    m = Material(
        'm', mu=2*c10, kappa=2*c01,
    )

    ### Boundary conditions ###
    x_sym = EssentialBC('x_sym', regions['Left'], {'u.0' : 0.0})
    y_sym = EssentialBC('y_sym', regions['Near'], {'u.1' : 0.0})

```

(continues on next page)

(continued from previous page)

```

z_sym = EssentialBC('z_sym', regions['Bottom'], {'u.2' : 0.0})
disp_fun = Function('disp_fun', get_displacement)
displacement = EssentialBC(
    'displacement', regions['Right'], {'u.0' : disp_fun})
ebcs = Conditions([x_sym, y_sym, z_sym, displacement])

### Terms and equations ###
integral = Integral('i', order=2*order)

term_neohook = Term.new(
    'dw_tl_he_neohook(m.mu, v, u)',
    integral, omega, m=m, v=v, u=u)
term_mooney = Term.new(
    'dw_tl_he_mooney_rivlin(m.kappa, v, u)',
    integral, omega, m=m, v=v, u=u)
term_pressure = Term.new(
    'dw_tl_bulk_pressure(v, u, p)',
    integral, omega, v=v, u=u, p=p)

term_volume_change = Term.new(
    'dw_tl_volume(q, u)',
    integral, omega, q=q, u=u, term_mode='volume')
term_volume = Term.new(
    'dw_integrate(q)',
    integral, omega, q=q)

eq_balance = Equation('balance', term_neohook+term_mooney+term_pressure)
eq_volume = Equation('volume', term_volume_change-term_volume)
equations = Equations([eq_balance, eq_volume])

### Solvers ###
ls = ScipyDirect({})
nls_status = IndexedStruct()
nls = Newton(
    {'i_max' : 5},
    lin_solver=ls, status=nls_status
)

### Problem ###
pb = Problem('hyper', equations=equations)
pb.set_bcs(ebcs=ebcs)
pb.set_ics(ics=Conditions([]))
tss = SimpleTimeSteppingSolver(ts, nls=nls, context=pb)
pb.set_solver(tss)

### Solution ###
axial_stress = []
axial_displacement = []
def stress_strain(*args, **kwargs):
    return stress_strain(
        *args, order=order, global_stress=axial_stress,
        global_displacement=axial_displacement, **kwargs)

```

(continues on next page)

(continued from previous page)

```

pb.solve(save_results=True, post_process_hook=stress_strain_fun)

if do_plot:
    fig = plot_graphs(
        material_parameters, axial_stress, axial_displacement,
        undeformed_length=dims[0])

    fig.savefig('hyperelastic_tl_up_comparison.png', bbox_inches='tight')

    if cli_args.show:
        plt.show()

def parse_argument_list(cli_arg, type_fun=None, value_separator=','):
    """
    Split the command-line argument into a list of items of given type.

    Parameters
    -----
    cli_arg : str
    type_fun : function
        A function to be called on each substring of `cli_arg`; default: str.
    value_separator : str
    """
    if type_fun is None:
        type_fun = str
    out = [type_fun(value) for value in cli_arg.split(value_separator)]
    return out

def parse_args():
    """Parse command line arguments."""
    parser = argparse.ArgumentParser(
        description=__doc__,
        formatter_class=argparse.RawDescriptionHelpFormatter)
    parser.add_argument(
        '--order', type=int, default=1, help='The approximation order of the '
        'displacement field [default: %(default)s]')
    parser.add_argument(
        '-m', '--material-parameters', default='1.0, 0.5',
        help='Material parameters - C10, C01 - of the two-parametric '
        'Mooney-Rivlin hyperelastic model. [default: %(default)s]')
    parser.add_argument(
        '--dims', default='1.0, 1.0, 1.0',
        help='Dimensions of the block [default: %(default)s]')
    parser.add_argument(
        '--shape', default='4, 4, 4',
        help='Shape (counts of nodes in x, y, z) of the block [default: '
        '%(default)s]')
    parser.add_argument(
        '--centre', default='0.5, 0.5, 0.5',
        help='Centre of the block [default: %(default)s]')
    parser.add_argument(

```

(continues on next page)

(continued from previous page)

```
    '-p', '--plot', action='store_true', default=False,
    help='Whether to plot a comparison with analytical formula.')
parser.add_argument(
    '-n', '--no-show', dest='show', action='store_false', default=True,
    help='Do not show matplotlib figures.')
parser.add_argument(
    '-t', '--ts',
    type=str, default='0.0,10.0,11',
    help='Start time, end time, and number of time steps [default: '
    '"%(default)s"]')
return parser.parse_args()

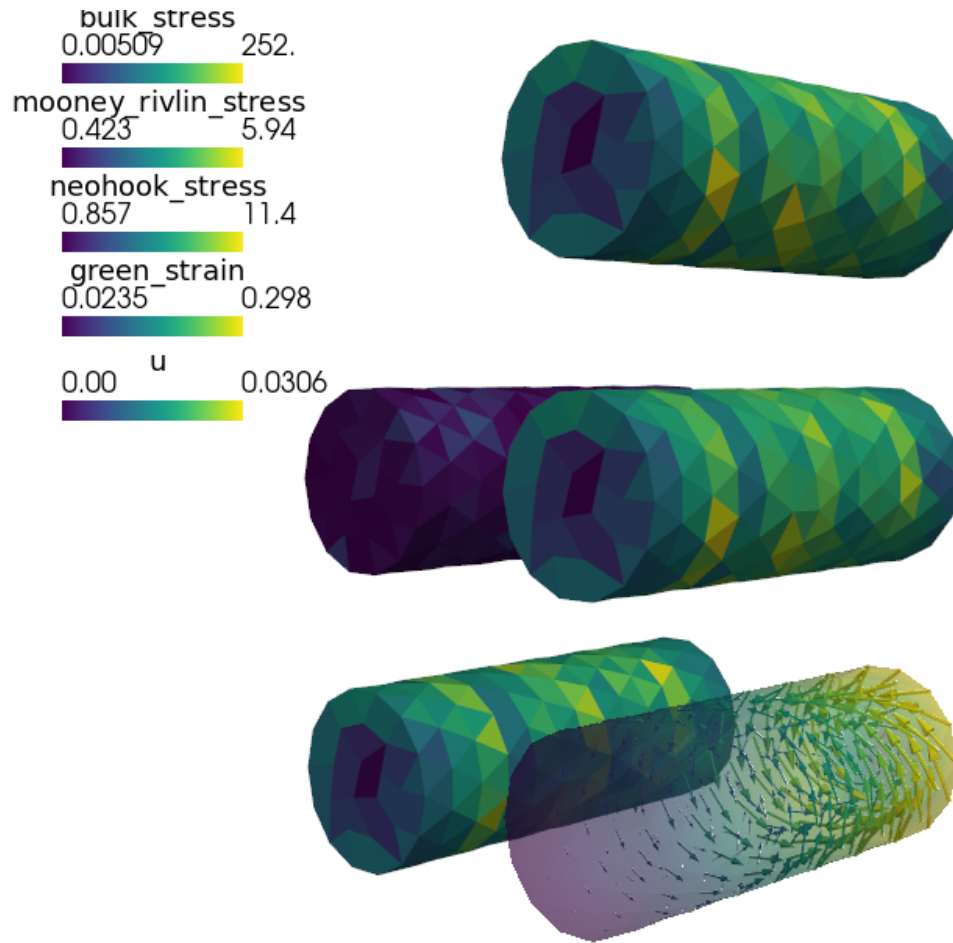
if __name__ == '__main__':
    args = parse_args()
    main(args)
```

large_deformation/hyperelastic_ul.py

Description

Nearly incompressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the updated Lagrangian formulation. Models of this kind can be used to model e.g. rubber or some biological materials.



source code

```
# -*- coding: utf-8 -*-
r"""
Nearly incompressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the updated Lagrangian formulation.
Models of this kind can be used to model e.g. rubber or some biological
materials.
"""
from __future__ import print_function
from __future__ import absolute_import
import numpy as nm
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

options = {
    'nls': 'newton',
    'ls': 'ls',
    'ts': 'ts',
    'ulf': True,
    'mesh_update_variables': ['u'],
```

(continues on next page)

(continued from previous page)

```

    'output_dir': 'output',
    'post_process_hook': 'stress_strain',
    'report_nls_status': True,
    'log_nls_status': True,
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
}

materials = {
    'solid': ({'K': 1e3, # bulk modulus
               'mu': 20e0, # shear modulus of neoHookean term
               'kappa': 10e0, # shear modulus of Mooney-Rivlin term
               },),
}

variables = {
    'u': ('unknown field', 'displacement', 0),
    'v': ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}

##
# Dirichlet BC + related functions.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0}),
    'r' : ('Right', {'u.0' : 0.0, 'u.[1,2]' : 'rotate_yz'}),
}

centre = nm.array( [0, 0], dtype = nm.float64 )

def rotate_yz(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d

    vec = coor[:,1:3] - centre

    angle = 10.0 * ts.step
    print('angle:', angle)

    mtx = rotation_matrix2d( angle )
    vec_rotated = nm.dot( vec, mtx )

    displacement = vec_rotated - vec

    return displacement

```

(continues on next page)

(continued from previous page)

```

functions = {
    'rotate_yz' : (rotate_yz,),
}

def stress_strain( out, problem, state, extend = False ):
    from sfepy.base.base import Struct

    ev = problem.evaluate
    strain = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=strain, dofs=None)

    stress = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='stress')
    out['neohook_stress'] = Struct(name='output_data',
                                   mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_he_mooney_rivlin.3.Omega( solid.kappa, v, u )',
               mode='el_avg', term_mode='stress')
    out['mooney_rivlin_stress'] = Struct(name='output_data',
                                          mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_bulk_penalty.3.Omega( solid.K, v, u )',
               mode='el_avg', term_mode='stress')
    out['bulk_stress'] = Struct(name='output_data',
                                mode='cell', data=stress, dofs=None)

    return out

equations = {
    'balance': """dw_ul_he_neohook.3.Omega( solid.mu, v, u )
                  + dw_ul_he_mooney_rivlin.3.Omega(solid.kappa, v, u)
                  + dw_ul_bulk_penalty.3.Omega( solid.K, v, u )
                  = 0""",
}

##
# Solvers etc.
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 25,
        'eps_a': 1e-8,
        'eps_r': 1.0,
        'macheps': 1e-16,
        'lin_red': 1e-2, # Linear system error < (eps_a * lin_red).
        'ls_red': 0.1,
        'ls_red_warp': 0.001,
        'ls_on': 1.1,
        'ls_min': 1e-5,
        'check': 0,
    })
}

```

(continues on next page)

(continued from previous page)

```

        'delta': 1e-6,
    }),
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'dt': None,
        'n_step': 11, # has precedence over dt!
        'verbose' : 1,
    }),
}

```

large_deformation/hyperelastic_ul_by_fun.py

Description

example import failed!

source code

```

# -*- coding: utf-8 -*-
r"""
Compressible Neo-Hookean hyperelastic material model. The tangent modulus
and the stress tensor are calculated by a user defined function.
"""
from functools import partial

import numpy as nm
from hyperelastic_ul import (filename_mesh, options, regions, fields,
                             variables, ebcs, functions)

mu, K = 20., 1000.

# function which returns tangent modulus and stress tensor
def get_hyperelastic_mat(family_data, mode):
    from sfepy.terms.extmods.terms import sym2nonsym
    from sfepy.terms.terms_hyperelastic_ul import (NeoHookeanULTerm,
                                                    BulkPenaltyULTerm)

    n_el, n_qp, sym, _ = family_data.green_strain.shape
    dim = family_data.mtx_f.shape[-1]
    dim2 = dim**2

    ones = nm.ones((n_el, n_qp, 1, 1), dtype=nm.float64)
    mat_mu = ones * mu
    mat_K = ones * K

    fargs = [family_data.get(name)
              for name in NeoHookeanULTerm.family_data_names]
    stress = nm.empty((n_el, n_qp, sym, 1), dtype=nm.float64)
    NeoHookeanULTerm.stress_function(stress, mat_mu, *fargs)

    if mode == 'tan_mod':

```

(continues on next page)

(continued from previous page)

```

        tanmod = nm.empty((n_el, n_qp, sym, sym), dtype=nm.float64)
        NeoHookeanULTerm.tan_mod_function(tanmod, mat_mu, *fargs)

    fargs = [family_data.get(name)
              for name in BulkPenaltyULTerm.family_data_names]
    stress_p = nm.empty((n_el, n_qp, sym, 1), dtype=nm.float64)
    BulkPenaltyULTerm.stress_function(stress_p, mat_K, *fargs)

    if mode == 'tan_mod':
        tanmod_p = nm.empty((n_el, n_qp, sym, sym), dtype=nm.float64)
        BulkPenaltyULTerm.tan_mod_function(tanmod_p, mat_K, *fargs)

        tanmod_ns = nm.zeros((n_el, n_qp, dim2, dim2), dtype=nm.float64)
        sym2nonsym(tanmod_ns, tanmod + tanmod_p)

        stress_ns = nm.zeros((n_el, n_qp, dim2, dim2), dtype=nm.float64)
        sym2nonsym(stress_ns, stress + stress_p)

        out = tanmod_ns + stress_ns

    elif mode == 'stress':
        out = stress + stress_p

    else:
        raise ValueError()

    return out

def stress_strain(out, problem, state, extend = False):
    from sfepy.base.base import Struct

    ev = partial(problem.evaluate, mode='el_avg',
                  get_hyperelastic_mat=get_hyperelastic_mat)
    stress = ev('dw_ul_he_by_fun.i.Omega(get_hyperelastic_mat, v, u)',
                term_mode='stress')
    out['cauchy_stress'] = Struct(name='output_data',
                                  mode='cell', data=stress)
    strain = ev('dw_ul_he_by_fun.i.Omega(get_hyperelastic_mat, v, u)',
                term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                  mode='cell', data=strain)

    return out

integrals = {
    'i': 2,
}

materials = {
}

```

(continues on next page)

(continued from previous page)

```
equations = {
    'balance': 'dw_ul_he_by_fun.i.Omega(get_hyperelastic_mat, v, u) = 0',
}

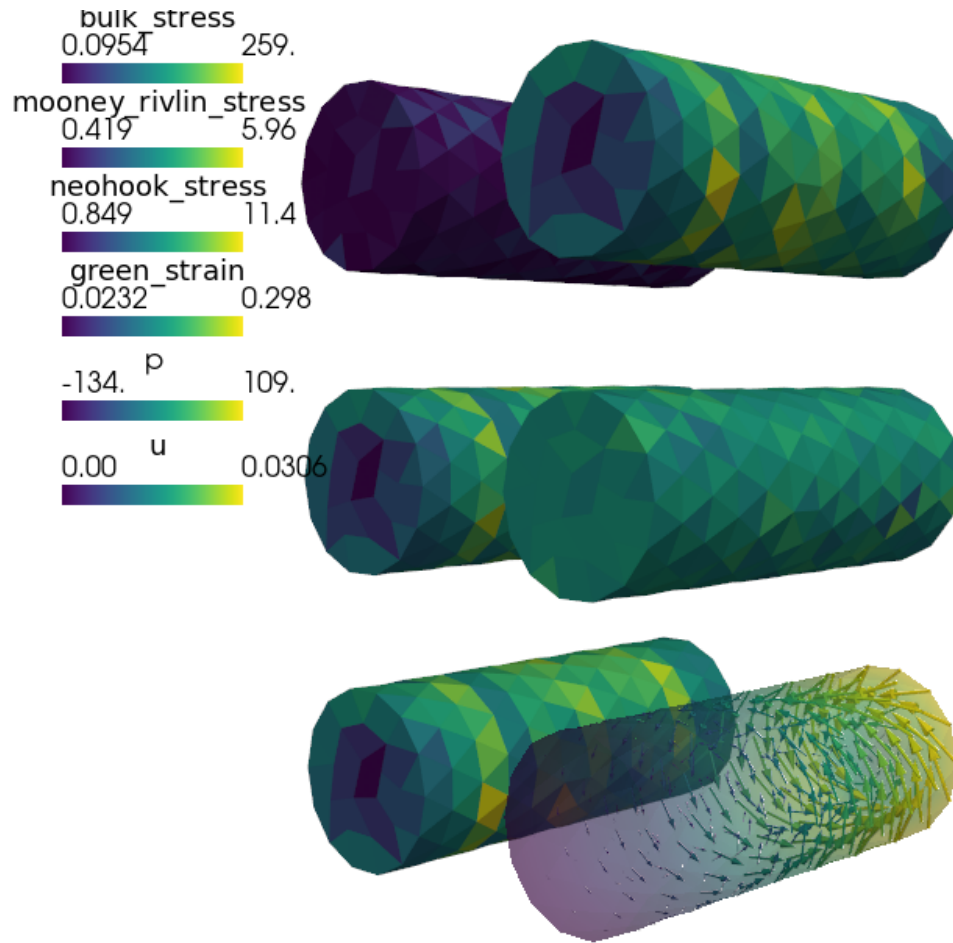
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 20,
        'eps_a': 1e-5,
        'eps_r': 1e-3,
    }),
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'n_step': 11,
    }),
}
```

large_deformation/hyperelastic_ul_up.py

Description

Compressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the updated Lagrangian formulation. Incompressibility is treated by mixed displacement-pressure formulation. Models of this kind can be used to model e.g. rubber or some biological materials.



source code

```
# -*- coding: utf-8 -*-
r"""
Compressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the updated Lagrangian formulation.
Incompressibility is treated by mixed displacement-pressure formulation.
Models of this kind can be used to model e.g. rubber or some biological
materials.
"""
from __future__ import print_function
from __future__ import absolute_import
import numpy as nm
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

options = {
    'nls': 'newton',
    'ls': 'ls',
    'ts': 'ts',
    'ulf': True,
```

(continues on next page)

(continued from previous page)

```

    'mesh_update_variables': ['u'],
    'output_dir': 'output',
    'post_process_hook': 'stress_strain',
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
    'pressure': ('real', 'scalar', 'Omega', 0),
}

materials = {
    'solid': ({'iK': 1.0 / 1e3, # bulk modulus
               'mu': 20e0, # shear modulus of neoHookean term
               'kappa': 10e0, # shear modulus of Mooney-Rivlin term
               },),
}

variables = {
    'u': ('unknown field', 'displacement', 0),
    'v': ('test field', 'displacement', 'u'),
    'p': ('unknown field', 'pressure', 1),
    'q': ('test field', 'pressure', 'p'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}

##
# Dirichlet BC + related functions.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0}),
    'r' : ('Right', {'u.0' : 0.0, 'u.[1,2]' : 'rotate_yz'}),
}

centre = nm.array( [0, 0], dtype = nm.float64 )

def rotate_yz(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d

    vec = coor[:,1:3] - centre

    angle = 10.0 * ts.step
    print('angle:', angle)

    mtx = rotation_matrix2d( angle )
    vec_rotated = nm.dot( vec, mtx )

    displacement = vec_rotated - vec

```

(continues on next page)

(continued from previous page)

```

    return displacement

functions = {
    'rotate_yz' : (rotate_yz,),
}

def stress_strain( out, problem, state, extend = False ):
    from sfepy.base.base import Struct

    ev = problem.evaluate
    strain = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
                mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=strain, dofs=None)

    stress = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
                mode='el_avg', term_mode='stress')
    out['neohook_stress'] = Struct(name='output_data',
                                   mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_he_mooney_rivlin.3.Omega( solid.kappa, v, u )',
                mode='el_avg', term_mode='stress')
    out['mooney_rivlin_stress'] = Struct(name='output_data',
                                         mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_bulk_pressure.3.Omega( v, u, p )',
                mode='el_avg', term_mode='stress')
    out['bulk_stress'] = Struct(name='output_data',
                                mode='cell', data=stress, dofs=None)

    return out

equations = {
    'balance': """dw_ul_he_neohook.3.Omega( solid.mu, v, u )
                  + dw_ul_he_mooney_rivlin.3.Omega(solid.kappa, v, u)
                  + dw_ul_bulk_pressure.3.Omega( v, u, p ) = 0""",
    'volume': """dw_ul_volume.3.Omega( q, u )
                 + dw_ul_compressible.3.Omega( solid.iK, q, p, u ) = 0""",
}

##
# Solvers etc.
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 25,
        'eps_a': 1e-8,
        'eps_r': 1.0,
        'macheps': 1e-16,
        'lin_red': 1e-2, # Linear system error < (eps_a * lin_red).
        'ls_red': 0.1,
        'ls_red_warp': 0.001,
    })
}

```

(continues on next page)

(continued from previous page)

```

    'ls_on': 1.1,
    'ls_min': 1e-5,
    'check': 0,
    'delta': 1e-6,
    },
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'dt': None,
        'n_step': 11, # has precedence over dt!
        'verbose' : 1,
    }),
}

```

large_deformation/perfusion_tl.py

Description

Porous nearly incompressible hyperelastic material with fluid perfusion.

Large deformation is described using the total Lagrangian formulation. Models of this kind can be used in biomechanics to model biological tissues, e.g. muscles.

Find $\underline{u}$ such that:

(equilibrium equation with boundary tractions)

$$\int_{\Omega^{(0)}} \left(\underline{\underline{S}}^{\text{eff}} - pJ\underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV + \int_{\Gamma_0^{(0)}} \underline{\nu} \cdot \underline{\underline{F}}^{-1} \cdot \underline{\underline{\sigma}} \cdot \underline{v} \, dS = 0, \quad \forall \underline{v},$$

(mass balance equation (perfusion))

$$\int_{\Omega^{(0)}} qJ(\underline{u}) + \int_{\Omega^{(0)}} \underline{\underline{K}}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial X} \frac{\partial p}{\partial X} = \int_{\Omega^{(0)}} qJ(\underline{u}^{(n-1)}) , \quad \forall q ,$$

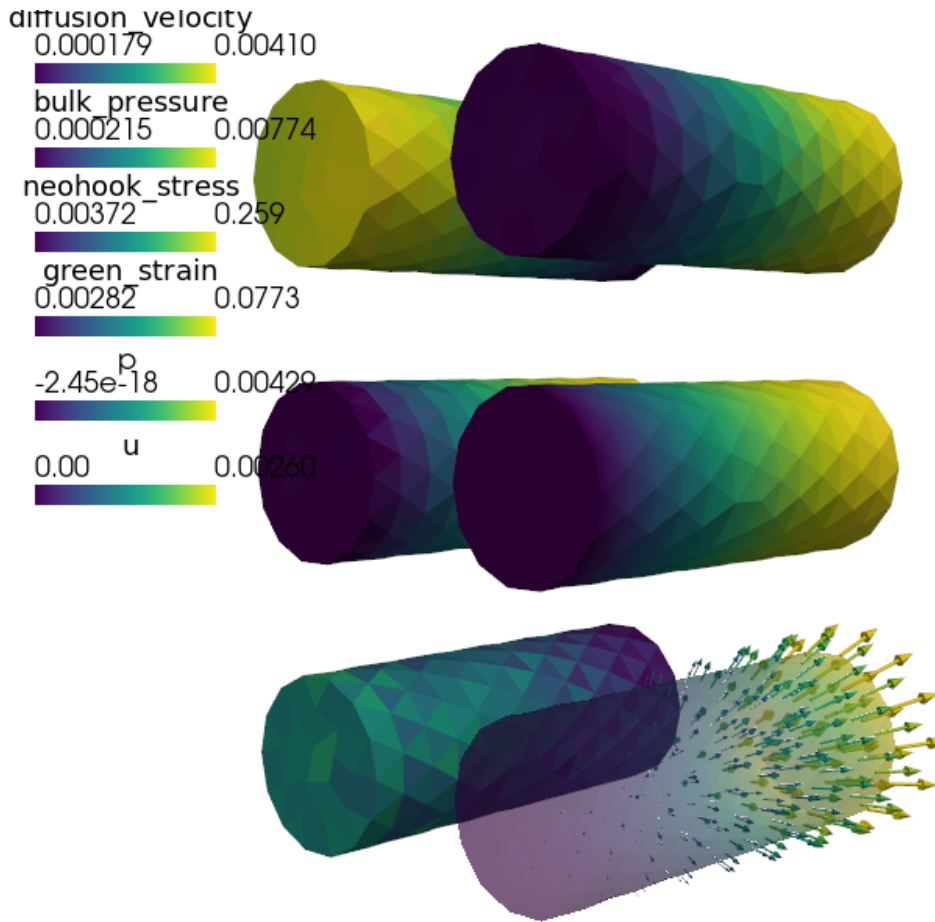
where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
J	$\det(\underline{\underline{F}})$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $\underline{\underline{C}} = \underline{\underline{F}}^T \underline{\underline{F}}$
$\underline{\underline{E}}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j} \right)$
$\underline{\underline{S}}^{\text{eff}}(\underline{u})$	effective second Piola-Kirchhoff stress tensor

The effective (neo-Hookean) stress $\underline{\underline{S}}^{\text{eff}}(\underline{u})$ is given by:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} \left(\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1} \right) .$$

The linearized deformation-dependent permeability is defined as $\underline{\underline{K}}(\underline{u}) = J \underline{\underline{F}}^{-1} \underline{\underline{k}}(J) \underline{\underline{F}}^{-T}$, where $\underline{u}$ relates to the previous time step ($n - 1$) and $f(J) = \max \left(0, \left(1 + \frac{(J-1)}{N_f} \right) \right)^2$ expresses the dependence on volume compression/expansion.



source code

```
# -*- coding: utf-8 -*-
r"""
Porous nearly incompressible hyperelastic material with fluid perfusion.

Large deformation is described using the total Lagrangian formulation.
Models of this kind can be used in biomechanics to model biological
tissues, e.g. muscles.

Find :math:\ul{u} such that:

(equilibrium equation with boundary tractions)

.. math::
\intl{\Omega\ul{suz}} \left( \ul{S}\ul{eff} - p J \ul{C}^{-1} \right) : \delta \ul{E}(\ul{v}) \difd{V}
+ \intl{\Gamma_0\ul{suz}} \ul{\nu} \cdot \ul{F}^{-1} \cdot \ul{\sigma}
\cdot \ul{v} J \difd{S}
= 0
\;, \quad \forall \ul{v} \;,

(mass balance equation (perfusion))
```

(continues on next page)

(continued from previous page)

```

.. math::
    \intl{\Omega\text{suz}} q J(\text{ul}\{u\})
    + \intl{\Omega\text{suz}} \text{ull}\{K\}(\text{ul}\{u\}\text{sunm}) : \pdiff{q}{X} \pdiff{p}{X}
    = \intl{\Omega\text{suz}} q J(\text{ul}\{u\}\text{sunm})
    \;; \quad \text{forall } q \;;

```

where

```

.. list-table::
   :widths: 20 80

   * - :math:`\text{ull}\{F\}`
     - deformation gradient :math:`F_{ij} = \pdiff{x_i}{X_j}`
   * - :math:`J`
     - :math:`\det(F)`
   * - :math:`\text{ull}\{C\}`
     - right Cauchy-Green deformation tensor :math:`C = F^T F`
   * - :math:`\text{ull}\{E\}(\text{ul}\{u\})`
     - Green strain tensor :math:`E_{ij} = \frac{1}{2}(\pdiff{u_i}{X_j} + \pdiff{u_j}{X_i} + \pdiff{u_m}{X_i}\pdiff{u_m}{X_j})`
   * - :math:`\text{ull}\{S\}\text{eff}(\text{ul}\{u\})`
     - effective second Piola-Kirchhoff stress tensor

```

The effective (neo-Hookean) stress :math:`\text{ull}\{S\}\text{eff}(\text{ul}\{u\})` is given by:

```

.. math::
    \text{ull}\{S\}\text{eff}(\text{ul}\{u\}) = \mu J^{-\frac{2}{3}}(\text{ull}\{I\}
    - \frac{1}{3}\text{tr}(\text{ull}\{C\}) \text{ull}\{C\}^{-1})
    \;.

```

The linearized deformation-dependent permeability is defined as :math:`\text{ull}\{K\}(\text{ul}\{u\}) = J \text{ull}\{F\}^{-1} \text{ull}\{k\} f(J) \text{ull}\{F\}^{-T}` , where :math:`\text{ul}\{u\}` relates to the previous time step :math:`(n-1)` and :math:`f(J) = \max\left(0, \left(1 + \frac{J - 1}{N\_f}\right)\right)^2` expresses the dependence on volume compression/expansion.

```

"""

```

```

from __future__ import absolute_import
import numpy as nm

```

```

from sfepy import data_dir

```

```

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

```

```

# Time-stepping parameters.

```

```

t0 = 0.0

```

```

t1 = 1.0

```

```

n_step = 21

```

(continues on next page)

(continued from previous page)

```

from sfepy.solvers.ts import TimeStepper
ts = TimeStepper(t0, t1, None, n_step)

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'save_times' : 'all',
    'post_process_hook' : 'post_process',
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
    'pressure' : ('real', 1, 'Omega', 1),
}

materials = {
    # Perfused solid.
    'ps' : ({
        'mu' : 20e0, # shear modulus of neoHookean term
        'k' : ts.dt * nm.eye(3, dtype=nm.float64), # reference permeability
        'N_f' : 1.0, # reference porosity
    },),
    # Surface pressure traction.
    'traction' : 'get_traction',
}

variables = {
    'u' : ('unknown field', 'displacement', 0, 1),
    'v' : ('test field', 'displacement', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}

##
# Dirichlet BC.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0, 'p.0' : 'get_pressure'}),
}

##
# Balance of forces.
integrals = {
    'i1' : 1,
    'i2' : 2,

```

(continues on next page)

(continued from previous page)

```

}

equations = {
    'force_balance'
        : """dw_tl_he_neohook.i1.Omega( ps.mu, v, u )
            + dw_tl_bulk_pressure.i1.Omega( v, u, p )
            + dw_tl_surface_traction.i2.Right( traction.pressure, v, u )
            = 0""",
    'mass_balance'
        : """dw_tl_volume.i1.Omega( q, u )
            + dw_tl_diffusion.i1.Omega( ps.k, ps.N_f, q, p, u[-1])
            = dw_tl_volume.i1.Omega( q, u[-1] )"""
}

def post_process(out, problem, state, extend=False):
    from sfepy.base.base import Struct, debug

    val = problem.evaluate('dw_tl_he_neohook.i1.Omega( ps.mu, v, u )',
                           mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=val, dofs=None)

    val = problem.evaluate('dw_tl_he_neohook.i1.Omega( ps.mu, v, u )',
                           mode='el_avg', term_mode='stress')
    out['neohook_stress'] = Struct(name='output_data',
                                   mode='cell', data=val, dofs=None)

    val = problem.evaluate('dw_tl_bulk_pressure.i1.Omega( v, u, p )',
                           mode='el_avg', term_mode='stress')
    out['bulk_pressure'] = Struct(name='output_data',
                                   mode='cell', data=val, dofs=None)

    val = problem.evaluate('dw_tl_diffusion.i1.Omega( ps.k, ps.N_f, q, p, u[-1] )',
                           mode='el_avg', term_mode='diffusion_velocity')
    out['diffusion_velocity'] = Struct(name='output_data',
                                       mode='cell', data=val, dofs=None)

    return out

##
# Solvers etc.
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max' : 7,
    'eps_a' : 1e-10,

```

(continues on next page)

(continued from previous page)

```

'eps_r'      : 1.0,
'macheps'    : 1e-16,
'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
'ls_red'     : 0.1,
'ls_red_warp': 0.001,
'ls_on'      : 1.1,
'ls_min'     : 1e-5,
'check'      : 0,
'delta'      : 1e-6,
}

solver_2 = {
    'name' : 'ts',
    'kind' : 'ts.simple',

    't0'    : t0,
    't1'    : t1,
    'dt'    : None,
    'n_step' : n_step, # has precedence over dt!
    'verbose' : 1,
}

##
# Functions.
def get_traction(ts, coors, mode=None):
    """
    Pressure traction.

    Parameters
    -----
    ts : TimeStepper
        Time stepping info.
    coors : array_like
        The physical domain coordinates where the parameters should be defined.
    mode : 'qp' or 'special'
        Call mode.
    """
    if mode != 'qp': return

    tt = ts.nt * 2.0 * nm.pi

    dim = coors.shape[1]
    val = 0.05 * nm.sin(tt) * nm.eye(dim, dtype=nm.float64)
    val[1,0] = val[0,1] = 0.5 * val[0,0]

    shape = (coors.shape[0], 1, 1)
    out = {
        'pressure' : nm.tile(val, shape),
    }

    return out

```

(continues on next page)

(continued from previous page)

```
def get_pressure(ts, coor, **kwargs):
    """Internal pressure Dirichlet boundary condition."""
    tt = ts.nt * 2.0 * nm.pi

    val = nm.zeros((coor.shape[0],), dtype=nm.float64)

    val[:] = 1e-2 * nm.sin(tt)

    return val

functions = {
    'get_traction' : (lambda ts, coors, mode=None, **kwargs:
                      get_traction(ts, coors, mode=mode),),
    'get_pressure' : (get_pressure,),
}
```

linear_elasticity

linear_elasticity/dispersion_analysis.py

Description

Dispersion analysis of a heterogeneous finite scale periodic cell.

The periodic cell mesh has to contain two subdomains Y1 (with the cell ids 1), Y2 (with the cell ids 2), so that different material properties can be defined in each of the subdomains (see `--pars` option). The command line parameters can be given in any consistent unit set, for example the basic SI units. The `--unit-multipliers` option can be used to rescale the input units to ones more suitable to the simulation, for example to prevent having different matrix blocks with large differences of matrix entries magnitudes. The results are then in the rescaled units.

Usage Examples

Default material parameters, a square periodic cell with a spherical inclusion, logs also standard pressure dilatation and shear waves, no eigenvectors:

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/circle_
↳ in_square.mesh --log-std-waves --eigs-only
```

As above, with custom eigenvalue solver parameters, and different number of eigenvalues, mesh size and units used in the calculation:

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/circle_
↳ in_square.mesh --solver-conf="kind='eig.scipy', method='eigsh', tol=1e-10,
↳ maxiter=1000, which='LM', sigma=0" --log-std-waves -n 5 --range=0,640,101 --mode=omega_
↳ --unit-multipliers=1e-6,1e-2,1e-3 --mesh-size=1e-2 --eigs-only
```

Default material parameters, a square periodic cell with a square inclusion, and a very small mesh to allow comparing the omega and kappa modes (full matrix solver required!):

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_2m.mesh -
↳ -solver-conf="kind='eig.scipy', method='eigh'" --log-std-waves -n 10 --range=0,640,101
↳ --mesh-size=1e-2 --mode=omega --eigs-only --no-legends --unit-multipliers=1e-6,1e-2,1e-
↳ 3 -o output/omega

python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_2m.mesh -
↳ -solver-conf="kind='eig.qevp', method='companion', mode='inverted', solver={kind='eig.
↳ scipy', method='eig'}" --log-std-waves -n 500 --range=0,4000000,1001 --mesh-size=1e-2 -
↳ -mode=kappa --eigs-only --no-legends --unit-multipliers=1e-6,1e-2,1e-3 -o output/kappa
```

View/compare the resulting logs:

```
python script/plot_logs.py output/omega/frequencies.txt --no-legends -g 1 -o mode-omega.
↳ png
python script/plot_logs.py output/kappa/wave-numbers.txt --no-legends -o mode-kappa.png
python script/plot_logs.py output/kappa/wave-numbers.txt --no-legends --swap-axes -o
↳ mode-kappa-t.png
```

In contrast to the heterogeneous square periodic cell, a homogeneous square periodic cell (the region Y2 is empty):

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_1m.mesh -
↳ -solver-conf="kind='eig.scipy', method='eigh'" --log-std-waves -n 10 --range=0,640,101
↳ --mesh-size=1e-2 --mode=omega --eigs-only --no-legends --unit-multipliers=1e-6,1e-2,1e-
↳ 3 -o output/omega-h

python script/plot_logs.py output/omega-h/frequencies.txt --no-legends -g 1 -o mode-
↳ omega-h.png
```

Use the Brillouin stepper:

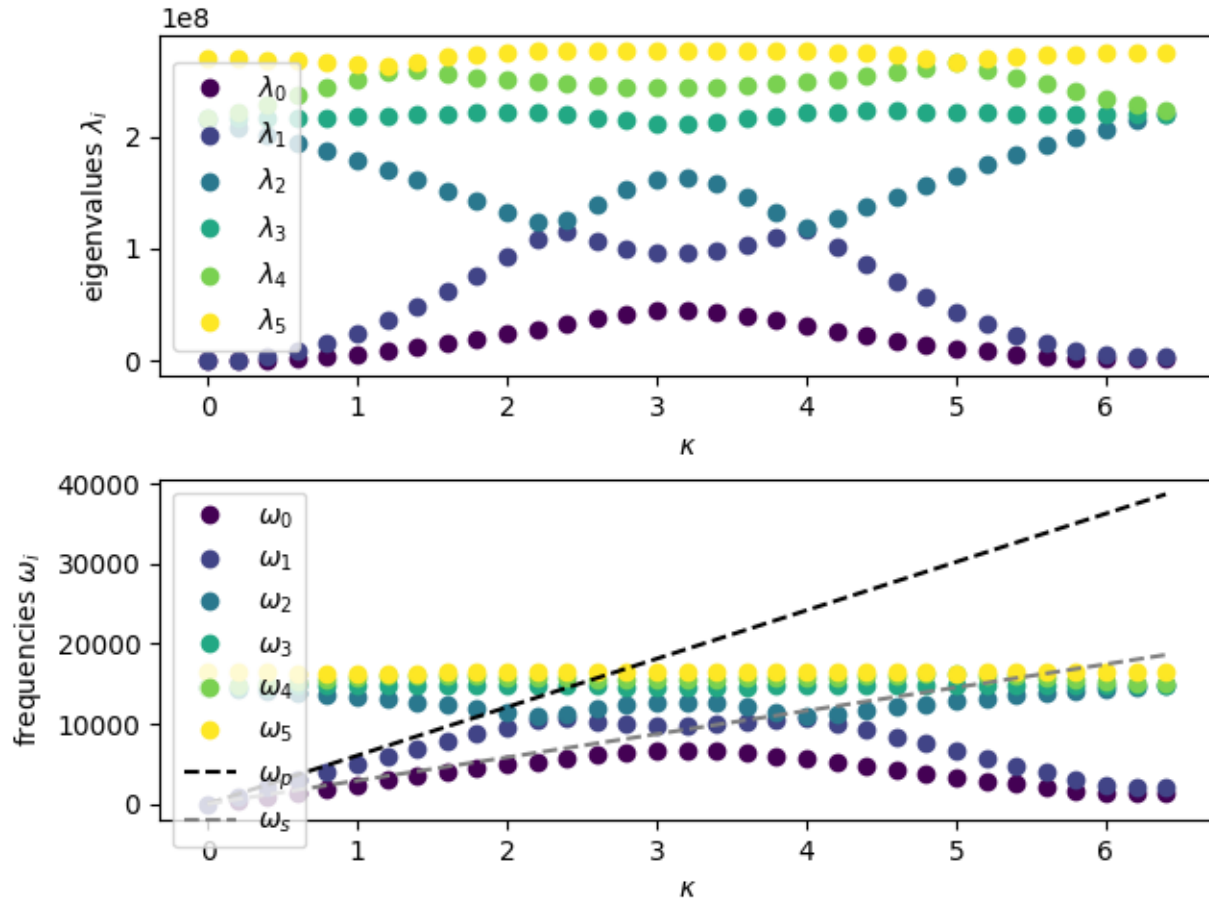
```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/circle_
↳ in_square.mesh --log-std-waves -n=60 --eigs-only --no-legends --stepper=brillouin

python script/plot_logs.py output/frequencies.txt -g 0 --rc="'font.size':14, 'lines.
↳ linewidth' : 3, 'lines.markersize' : 4" -o brillouin-stepper-kappas.png

python script/plot_logs.py output/frequencies.txt -g 1 --no-legends --rc="'font.size':14,
↳ 'lines.linewidth' : 3, 'lines.markersize' : 4" -o brillouin-stepper-omegas.png
```

Additional arguments can be passed to the problem configuration's `define()` function using the `--define-kwargs` option. In this file, only the mesh vertex separation parameter `mesh_eps` can be used:

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/circle_
↳ in_square.mesh --log-std-waves --eigs-only --define-kwargs="mesh_eps=1e-10" --save-
↳ regions
```



source code

```
#!/usr/bin/env python
"""
Dispersion analysis of a heterogeneous finite scale periodic cell.

The periodic cell mesh has to contain two subdomains Y1 (with the cell ids 1),
Y2 (with the cell ids 2), so that different material properties can be defined
in each of the subdomains (see ``--pars`` option). The command line parameters
can be given in any consistent unit set, for example the basic SI units. The
``--unit-multipliers`` option can be used to rescale the input units to ones
more suitable to the simulation, for example to prevent having different
matrix blocks with large differences of matrix entries magnitudes. The results
are then in the rescaled units.

Usage Examples
-----

Default material parameters, a square periodic cell with a spherical inclusion,
logs also standard pressure dilatation and shear waves, no eigenvectors::

    python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/
    ↪ circle_in_square.mesh --log-std-waves --eigs-only
```

(continues on next page)

(continued from previous page)

As above, with custom eigenvalue solver parameters, and different number of eigenvalues, mesh size and units used in the calculation::

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/
↪circle_in_square.mesh --solver-conf="kind='eig.scipy', method='eigsh', tol=1e-10,
↪maxiter=10000, which='LM', sigma=0" --log-std-waves -n 5 --range=0,640,101 --mode=omega -
↪-unit-multipliers=1e-6,1e-2,1e-3 --mesh-size=1e-2 --eigs-only
```

Default material parameters, a square periodic cell with a square inclusion, and a very small mesh to allow comparing the omega and kappa modes (full matrix solver required!)::

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_2m.
↪mesh --solver-conf="kind='eig.scipy', method='eigh'" --log-std-waves -n 10 --range=0,640,
↪101 --mesh-size=1e-2 --mode=omega --eigs-only --no-legends --unit-multipliers=1e-6,1e-
↪2,1e-3 -o output/omega
```

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_2m.
↪mesh --solver-conf="kind='eig.qevp', method='companion', mode='inverted', solver={kind='eig.
↪scipy', method='eig'}" --log-std-waves -n 500 --range=0,4000000,1001 --mesh-size=1e-2 --
↪mode=kappa --eigs-only --no-legends --unit-multipliers=1e-6,1e-2,1e-3 -o output/kappa
```

View/compare the resulting logs::

```
python script/plot_logs.py output/omega/frequencies.txt --no-legends -g 1 -o mode-
↪omega.png
python script/plot_logs.py output/kappa/wave-numbers.txt --no-legends -o mode-kappa.png
python script/plot_logs.py output/kappa/wave-numbers.txt --no-legends --swap-axes -o
↪mode-kappa-t.png
```

In contrast to the heterogeneous square periodic cell, a homogeneous square periodic cell (the region Y2 is empty)::

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_1m.
↪mesh --solver-conf="kind='eig.scipy', method='eigh'" --log-std-waves -n 10 --range=0,640,
↪101 --mesh-size=1e-2 --mode=omega --eigs-only --no-legends --unit-multipliers=1e-6,1e-
↪2,1e-3 -o output/omega-h
```

```
python script/plot_logs.py output/omega-h/frequencies.txt --no-legends -g 1 -o mode-
↪omega-h.png
```

Use the Brillouin stepper::

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/
↪circle_in_square.mesh --log-std-waves -n=60 --eigs-only --no-legends --
↪stepper=brillouin

python script/plot_logs.py output/frequencies.txt -g 0 --rc="'font.size':14, 'lines.
↪linewidth' : 3, 'lines.markersize' : 4" -o brillouin-stepper-kappas.png

python script/plot_logs.py output/frequencies.txt -g 1 --no-legends --rc="'font.size
↪':14, 'lines.linewidth' : 3, 'lines.markersize' : 4" -o brillouin-stepper-omegas.png
```

(continues on next page)

(continued from previous page)

Additional arguments can be passed to the problem configuration's `:func:`define()`` function using the `--define-kwargs` option. In this file, only the mesh vertex separation parameter `mesh_eps` can be used::

```
python sfepy/examples/linear_elasticity/dispersion_analysis.py meshes/2d/special/
↪circle_in_square.mesh --log-std-waves --eigs-only --define-kwargs="mesh_eps=1e-10" --
↪save-regions
"""
from __future__ import absolute_import
import os
import sys
sys.path.append('.')
import gc
from copy import copy
from argparse import ArgumentParser, RawDescriptionHelpFormatter

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import import_file, output, Struct
from sfepy.base.conf import dict_from_string, ProblemConf
from sfepy.base.ioutils import ensure_path, remove_files_patterns, save_options
from sfepy.base.log import Log
from sfepy.discrete.fem import MeshIO
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson as stiffness
import sfepy.mechanics.matcoefs as mc
from sfepy.mechanics.units import apply_unit_multipliers, apply_units_to_pars
import sfepy.discrete.fem.periodic as per
from sfepy.discrete.fem.meshio import convert_complex_output
from sfepy.homogenization.utils import define_box_regions
from sfepy.discrete import Problem
from sfepy.mechanics.tensors import get_von_mises_stress
from sfepy.solvers import Solver
from sfepy.solvers.ts import get_print_info, TimeStepper
from sfepy.linalg.utils import output_array_stats, max_diff_csr

pars_kinds = {
    'young1' : 'stress',
    'poisson1' : 'one',
    'density1' : 'density',
    'young2' : 'stress',
    'poisson2' : 'one',
    'density2' : 'density',
}

def define(filename_mesh, pars, approx_order, refinement_level, solver_conf,
           plane='strain', post_process=False, mesh_eps=1e-8):
    io = MeshIO.any_from_filename(filename_mesh)
    bbox = io.read_bounding_box()
    dim = bbox.shape[1]
```

(continues on next page)

(continued from previous page)

```

options = {
    'absolute_mesh_path' : True,
    'refinement_level' : refinement_level,
    'allow_empty_regions' : True,
    'post_process_hook' : 'compute_von_mises' if post_process else None,
}

fields = {
    'displacement': ('complex', dim, 'Omega', approx_order),
}

materials = {
    'm' : ({
        'D' : {'Y1' : stiffness(dim,
                                young=pars.young1,
                                poisson=pars.poisson1,
                                plane=plane),
              'Y2' : stiffness(dim,
                                young=pars.young2,
                                poisson=pars.poisson2,
                                plane=plane)}},
        'density' : {'Y1' : pars.density1, 'Y2' : pars.density2},
    },),
    'wave' : 'get_wdir',
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Y1' : 'cells of group 1',
    'Y2' : 'cells of group 2',
}
regions.update(define_box_regions(dim,
                                  bbox[0], bbox[1], mesh_eps))

ebcs = {
}

if dim == 3:
    epbcs = {
        'periodic_x' : (['Left', 'Right'], {'u.all' : 'u.all'},
                        'match_x_plane'),
        'periodic_y' : (['Near', 'Far'], {'u.all' : 'u.all'},
                        'match_y_plane'),
        'periodic_z' : (['Top', 'Bottom'], {'u.all' : 'u.all'},
                        'match_z_plane'),
    }
else:

```

(continues on next page)

(continued from previous page)

```

    epbcs = {
        'periodic_x' : (['Left', 'Right'], {'u.all' : 'u.all'},
                        'match_y_line'),
        'periodic_y' : (['Bottom', 'Top'], {'u.all' : 'u.all'},
                        'match_x_line'),
    }

    per.set_accuracy(mesh_eps)
    functions = {
        'match_x_plane' : (per.match_x_plane,),
        'match_y_plane' : (per.match_y_plane,),
        'match_z_plane' : (per.match_z_plane,),
        'match_x_line' : (per.match_x_line,),
        'match_y_line' : (per.match_y_line,),
        'get_wdir' : (get_wdir,),
    }

    integrals = {
        'i' : 2 * approx_order,
    }

    equations = {
        'K' : 'dw_lin_elastic.i.Omega(m.D, v, u)',
        'S' : 'dw_elastic_wave.i.Omega(m.D, wave.vec, v, u)',
        'R' : ""1j * dw_elastic_wave_cauchy.i.Omega(m.D, wave.vec, u, v)
              - 1j * dw_elastic_wave_cauchy.i.Omega(m.D, wave.vec, v, u)""",
        'M' : 'dw_dot.i.Omega(m.density, v, u)',
    }

    solver_0 = solver_conf.copy()
    solver_0['name'] = 'eig'

    return locals()

def get_wdir(ts, coors, mode=None,
             equations=None, term=None, problem=None, wdir=None, **kwargs):
    if mode == 'special':
        return {'vec' : wdir}

def set_wave_dir(pb, wdir):
    materials = pb.get_materials()
    wave_mat = materials['wave']
    wave_mat.set_extra_args(wdir=wdir)

def save_materials(output_dir, pb, options):
    stiffness = pb.evaluate('ev_integrate_mat.2.Omega(m.D, u)',
                           mode='el_avg', copy_materials=False, verbose=False)
    young, poisson = mc.youngpoisson_from_stiffness(stiffness,
                                                    plane=options.plane)
    density = pb.evaluate('ev_integrate_mat.2.Omega(m.density, u)',
                         mode='el_avg', copy_materials=False, verbose=False)

```

(continues on next page)

(continued from previous page)

```

out = {}
out['young'] = Struct(name='young', mode='cell',
                      data=young[... , None, None])
out['poisson'] = Struct(name='poisson', mode='cell',
                        data=poisson[... , None, None])
out['density'] = Struct(name='density', mode='cell', data=density)
materials_filename = os.path.join(output_dir, 'materials.vtk')
pb.save_state(materials_filename, out=out)

def get_std_wave_fun(pb, options):
    stiffness = pb.evaluate('ev_integrate_mat.2.Omega(m.D, u)',
                           mode='el_avg', copy_materials=False, verbose=False)
    young, poisson = mc.youngpoisson_from_stiffness(stiffness,
                                                    plane=options.plane)
    density = pb.evaluate('ev_integrate_mat.2.Omega(m.density, u)',
                          mode='el_avg', copy_materials=False, verbose=False)

    lam, mu = mc.lame_from_youngpoisson(young, poisson,
                                       plane=options.plane)

    alam = nm.average(lam)
    amu = nm.average(mu)
    adensity = nm.average(density)

    cp = nm.sqrt((alam + 2.0 * amu) / adensity)
    cs = nm.sqrt(amu / adensity)
    output('average p-wave speed:', cp)
    output('average shear wave speed:', cs)

    log_names = [r'$\omega_p$', r'$\omega_s$']
    log_plot_kwargs = [{'ls' : '--', 'color' : 'k'},
                       {'ls' : '--', 'color' : 'gray'}]

    if options.mode == 'omega':
        fun = lambda wmag, wdir: (cp * wmag, cs * wmag)

    else:
        fun = lambda wmag, wdir: (wmag / cp, wmag / cs)

    return fun, log_names, log_plot_kwargs

def get_stepper(rng, pb, options):
    if options.stepper == 'linear':
        stepper = TimeStepper(rng[0], rng[1], dt=None, n_step=rng[2])
        return stepper

    bbox = pb.domain.mesh.get_bounding_box()

    bzone = 2.0 * nm.pi / (bbox[1] - bbox[0])

    num = rng[2] // 3

    class BrillouinStepper(Struct):

```

(continues on next page)

(continued from previous page)

```

"""
Step over 1. Brillouin zone in xy plane.
"""
def __init__(self, t0, t1, dt=None, n_step=None, step=None, **kwargs):
    Struct.__init__(self, t0=t0, t1=t1, dt=dt, n_step=n_step, step=step)

    self.n_digit, self.format, self.suffix = get_print_info(self.n_step)

def __iter__(self):
    ts = TimeStepper(0, bzone[0], dt=None, n_step=num)
    for ii, val in ts:
        yield ii, val, nm.array([1.0, 0.0])
        if ii == (num-2): break

    ts = TimeStepper(0, bzone[1], dt=None, n_step=num)
    for ii, k1 in ts:
        wdir = nm.array([bzone[0], k1])
        val = nm.linalg.norm(wdir)
        wdir = wdir / val
        yield num + ii, val, wdir
        if ii == (num-2): break

    wdir = nm.array([bzone[0], bzone[1]])
    val = nm.linalg.norm(wdir)
    wdir = wdir / val
    ts = TimeStepper(0, 1, dt=None, n_step=num)
    for ii, _ in ts:
        yield 2 * num + ii, val * (1.0 - float(ii)/(num-1)), wdir

stepper = BrillouinStepper(0, 1, n_step=rng[2])

return stepper

def compute_von_mises(out, pb, state, extend=False, wmag=None, wdir=None):
    """
    Calculate the von Mises stress.
    """
    stress = pb.evaluate('ev_cauchy_stress.i.Omega(m.D, u)', mode='el_avg')

    vms = get_von_mises_stress(stress.squeeze())
    vms.shape = (vms.shape[0], 1, 1, 1)
    out['von_mises_stress'] = Struct(name='output_data', mode='cell',
                                    data=vms)

    return out

def save_eigenvectors(filename, svecs, wmag, wdir, pb):
    if svecs is None: return

    variables = pb.set_default_state()
    # Make full eigenvectors (add DOFs fixed by boundary conditions).
    vecs = nm.empty((variables.di.n_dof_total, svecs.shape[1]),

```

(continues on next page)

(continued from previous page)

```

        dtype=svecs.dtype)
    for ii in range(svecs.shape[1]):
        vecs[:, ii] = variables.make_full_vec(svecs[:, ii])

    # Save the eigenvectors.
    out = {}

    pp_name = pb.conf.options.get('post_process_hook')
    pp = getattr(pb.conf.funmod, pp_name if pp_name is not None else '',
                 lambda out, *args, **kwargs: out)

    for ii in range(svecs.shape[1]):
        variables.set_state(vecs[:, ii])
        aux = variables.create_output()
        aux2 = {}
        pp(aux2, pb, variables, wmag=wmag, wdir=wdir)
        aux.update(convert_complex_output(aux2))
        out.update({key + '%03d' % ii : aux[key] for key in aux})

    pb.save_state(filename, out=out)

def assemble_matrices(define, mod, pars, set_wave_dir, options, wdir=None):
    """
    Assemble the blocks of dispersion eigenvalue problem matrices.
    """
    define_dict = define(filename_mesh=options.mesh_filename,
                          pars=pars,
                          approx_order=options.order,
                          refinement_level=options.refine,
                          solver_conf=options.solver_conf,
                          plane=options.plane,
                          post_process=options.post_process,
                          **options.define_kwargs)

    conf = ProblemConf.from_dict(define_dict, mod)

    pb = Problem.from_conf(conf)
    pb.dispersion_options = options
    pb.set_output_dir(options.output_dir)
    dim = pb.domain.shape.dim

    # Set the normalized wave vector direction to the material(s).
    if wdir is None:
        wdir = nm.asarray(options.wave_dir[:dim], dtype=nm.float64)
        wdir = wdir / nm.linalg.norm(wdir)
    set_wave_dir(pb, wdir)

    bbox = pb.domain.mesh.get_bounding_box()
    size = (bbox[1] - bbox[0]).max()
    scaling0 = apply_unit_multipliers([1.0], ['length'],
                                       options.unit_multipliers)[0]
    scaling = scaling0

```

(continues on next page)

(continued from previous page)

```

if options.mesh_size is not None:
    scaling *= options.mesh_size / size
output('scaling factor of periodic cell mesh coordinates:', scaling)
output('new mesh size with applied unit multipliers:', scaling * size)
pb.domain.mesh.coors[:] *= scaling
pb.set_mesh_coors(pb.domain.mesh.coors, update_fields=True)

bzone = 2.0 * nm.pi / (scaling * size)
output('1. Brillouin zone size:', bzone * scaling)
output('1. Brillouin zone size with applied unit multipliers:', bzone)

pb.time_update()
pb.update_materials()

# Assemble the matrices.
mtx = {}
for key, eq in pb.equations.iteritems():
    mtx[key] = mtx = pb.mtx_a.copy()
    mtx = eq.evaluate(mode='weak', dw_mode='matrix', asm_obj=mtx)
    mtx.eliminate_zeros()
    output_array_stats(mtx.data, 'nonzeros in %s' % key)

    output('symmetry checks:')
    output('%s - %s^T:' % (key, key), max_diff_csr(mtx, mtx.T))
    output('%s - %s^H:' % (key, key), max_diff_csr(mtx, mtx.H))

return pb, wdir, bzone, mtx

def setup_n_eigs(options, pb, mtx):
    """
    Setup the numbers of eigenvalues based on options and numbers of DOFs.
    """
    solver_n_eigs = n_eigs = options.n_eigs
    n_dof = mtx['K'].shape[0]
    if options.mode == 'omega':
        if options.n_eigs > n_dof:
            n_eigs = n_dof
            solver_n_eigs = None
    else:
        if options.n_eigs > 2 * n_dof:
            n_eigs = 2 * n_dof
            solver_n_eigs = None

    return solver_n_eigs, n_eigs

def build_evp_matrices(mtx, val, mode, pb):
    """
    Build the matrices of the dispersion eigenvalue problem.
    """
    if mode == 'omega':
        mtx_a = mtx['K'] + val**2 * mtx['S'] + val * mtx['R']

```

(continues on next page)

(continued from previous page)

```

    output('A - A^H:', max_diff_csr(mtx_a, mtx_a.H))

    evp_mtxs = (mtx_a, mtxs['M'])

else:
    evp_mtxs = (mtxs['S'], mtxs['R'], mtxs['K'] - val**2 * mtxs['M'])

return evp_mtxs

def process_evp_results(eigs, svecs, val, wdir, bzone, pb, mtxs, options,
                       std_wave_fun=None):
    """
    Transform eigenvalues to either omegas or kappas, depending on `mode`.
    Transform eigenvectors, if available, depending on `mode`.
    Return also the values to log.
    """
    if options.mode == 'omega':
        omegas = nm.sqrt(eigs)

        output('eigs, omegas:')
        for ii, om in enumerate(omegas):
            output('{:>3}. {: .10e}, {:.10e}'.format(ii, eigs[ii], om))

        if options.stepper == 'linear':
            out = tuple(eigs) + tuple(omegas)

        else:
            out = tuple(val * wdir) + tuple(omegas)

        if std_wave_fun is not None:
            out = out + std_wave_fun(val, wdir)

        return omegas, svecs, out

    else:
        kappas = eigs.copy()
        rks = kappas.copy()

        # Mask modes far from 1. Brillouin zone.
        max_kappa = 1.2 * bzone
        kappas[kappas.real > max_kappa] = nm.nan

        # Mask non-physical modes.
        kappas[kappas.real < 0] = nm.nan
        kappas[nm.abs(kappas.imag) > 1e-10] = nm.nan
        out = tuple(kappas.real)

        output('raw kappas, masked real part:',)
        for ii, kr in enumerate(kappas.real):
            output('{:>3}. {: 23.5e}, {:.10e}'.format(ii, rks[ii], kr))

        if svecs is not None:

```

(continues on next page)

(continued from previous page)

```

        n_dof = mtxs['K'].shape[0]
        # Select only vectors corresponding to physical modes.
        ii = nm.isfinite(kappas.real)
        svecs = svecs[:n_dof, ii]

        if std_wave_fun is not None:
            out = out + tuple(ii if ii <= max_kappa else nm.nan
                              for ii in std_wave_fun(val, wdir))

        return kappas, svecs, out

helps = {
    'pars' :
    'material parameters in Y1, Y2 subdomains in basic units.'
    ' The default parameters are:'
    ' young1, poisson1, density1, young2, poisson2, density2'
    ' [default: %(default)s]',
    'conf' :
    'if given, an alternative problem description file with apply_units() and'
    ' define() functions [default: %(default)s]',
    'define_kwargs' : 'additional keyword arguments passed to define()',
    'mesh_size' :
    'desired mesh size (max. of bounding box dimensions) in basic units'
    ' - the input periodic cell mesh is rescaled to this size'
    ' [default: %(default)s]',
    'unit_multipliers' :
    'basic unit multipliers (time, length, mass) [default: %(default)s]',
    'plane' :
    'for 2D problems, plane strain or stress hypothesis selection'
    ' [default: %(default)s]',
    'wave_dir' : 'the wave vector direction (will be normalized)'
    ' [default: %(default)s]',
    'mode' : 'solution mode: omega = solve a generalized EVP for omega,'
    ' kappa = solve a quadratic generalized EVP for kappa'
    ' [default: %(default)s]',
    'stepper' : 'the range stepper. For "brillouin", only the number'
    ' of items from --range is used'
    ' [default: %(default)s]',
    'range' : 'the wave vector magnitude / frequency range'
    ' (like numpy.linspace) depending on the mode option'
    ' [default: %(default)s]',
    'order' : 'displacement field approximation order [default: %(default)s]',
    'refine' : 'number of uniform mesh refinements [default: %(default)s]',
    'n_eigs' : 'the number of eigenvalues to compute [default: %(default)s]',
    'eigs_only' : 'compute only eigenvalues, not eigenvectors',
    'post_process' : 'post-process eigenvectors',
    'solver_conf' : 'eigenvalue problem solver configuration options'
    ' [default: %(default)s]',
    'save_regions' : 'save defined regions into'
    ' <output_directory>/regions.vtk',
    'save_materials' : 'save material parameters into'
    ' <output_directory>/materials.vtk',

```

(continues on next page)

(continued from previous page)

```

'log_std_waves' : 'log also standard pressure dilatation and shear waves',
'no_legends' :
'do not show legends in the log plots',
'no_show' :
'do not show the log figure',
'silent' : 'do not print messages to screen',
'clear' :
'clear old solution files from output directory',
'output_dir' :
'output directory [default: %(default)s]',
'mesh_filename' :
'input periodic cell mesh file name [default: %(default)s]',
}

def main():
    # Aluminium and epoxy.
    default_pars = '70e9,0.35,2.799e3,3.8e9,0.27,1.142e3'
    default_solver_conf = ("kind='eig.scipy',method='eigsh',tol=1.0e-5,"
                           "maxiter=1000,which='LM',sigma=0.0")

    parser = ArgumentParser(description=__doc__,
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--pars', metavar='name1=value1,name2=value2,...'
                        ' or value1,value2,...',
                        action='store', dest='pars',
                        default=default_pars, help=helps['pars'])
    parser.add_argument('--conf', metavar='filename',
                        action='store', dest='conf',
                        default=None, help=helps['conf'])
    parser.add_argument('--define-kwargs', metavar='dict-like',
                        action='store', dest='define_kwargs',
                        default=None, help=helps['define_kwargs'])
    parser.add_argument('--mesh-size', type=float, metavar='float',
                        action='store', dest='mesh_size',
                        default=None, help=helps['mesh_size'])
    parser.add_argument('--unit-multipliers',
                        metavar='c_time,c_length,c_mass',
                        action='store', dest='unit_multipliers',
                        default='1.0,1.0,1.0', help=helps['unit_multipliers'])
    parser.add_argument('--plane', action='store', dest='plane',
                        choices=['strain', 'stress'],
                        default='strain', help=helps['plane'])
    parser.add_argument('--wave-dir', metavar='float,float[,float]',
                        action='store', dest='wave_dir',
                        default='1.0,0.0,0.0', help=helps['wave_dir'])
    parser.add_argument('--mode', action='store', dest='mode',
                        choices=['omega', 'kappa'],
                        default='omega', help=helps['mode'])
    parser.add_argument('--stepper', action='store', dest='stepper',
                        choices=['linear', 'brillouin'],
                        default='linear', help=helps['stepper'])
    parser.add_argument('--range', metavar='start,stop,count',

```

(continues on next page)

(continued from previous page)

```

        action='store', dest='range',
        default='0,6.4,33', help=helps['range'])
parser.add_argument('--order', metavar='int', type=int,
                    action='store', dest='order',
                    default=1, help=helps['order'])
parser.add_argument('--refine', metavar='int', type=int,
                    action='store', dest='refine',
                    default=0, help=helps['refine'])
parser.add_argument('-n', '--n-eigs', metavar='int', type=int,
                    action='store', dest='n_eigs',
                    default=6, help=helps['n_eigs'])
group = parser.add_mutually_exclusive_group()
group.add_argument('--eigs-only',
                  action='store_true', dest='eigs_only',
                  default=False, help=helps['eigs_only'])
group.add_argument('--post-process',
                  action='store_true', dest='post_process',
                  default=False, help=helps['post_process'])
parser.add_argument('--solver-conf', metavar='dict-like',
                    action='store', dest='solver_conf',
                    default=default_solver_conf, help=helps['solver_conf'])
parser.add_argument('--save-regions',
                    action='store_true', dest='save_regions',
                    default=False, help=helps['save_regions'])
parser.add_argument('--save-materials',
                    action='store_true', dest='save_materials',
                    default=False, help=helps['save_materials'])
parser.add_argument('--log-std-waves',
                    action='store_true', dest='log_std_waves',
                    default=False, help=helps['log_std_waves'])
parser.add_argument('--no-legends',
                    action='store_false', dest='show_legends',
                    default=True, help=helps['no_legends'])
parser.add_argument('--no-show',
                    action='store_false', dest='show',
                    default=True, help=helps['no_show'])
parser.add_argument('--silent',
                    action='store_true', dest='silent',
                    default=False, help=helps['silent'])
parser.add_argument('-c', '--clear',
                    action='store_true', dest='clear',
                    default=False, help=helps['clear'])
parser.add_argument('-o', '--output-dir', metavar='path',
                    action='store', dest='output_dir',
                    default='output', help=helps['output_dir'])
parser.add_argument('mesh_filename', default='',
                    help=helps['mesh_filename'])
options = parser.parse_args()

output_dir = options.output_dir

output.set_output(filename=os.path.join(output_dir, 'output_log.txt'),

```

(continues on next page)

(continued from previous page)

```

        combined=options.silent == False)

if options.conf is not None:
    mod = import_file(options.conf)

else:
    mod = sys.modules[__name__]

pars_kinds = mod.pars_kinds
define = mod.define
set_wave_dir = mod.set_wave_dir
setup_n_eigs = mod.setup_n_eigs
build_evp_matrices = mod.build_evp_matrices
save_materials = mod.save_materials
get_std_wave_fun = mod.get_std_wave_fun
get_stepper = mod.get_stepper
process_evp_results = mod.process_evp_results
save_eigenvectors = mod.save_eigenvectors

try:
    options.pars = dict_from_string(options.pars)

except:
    aux = [float(ii) for ii in options.pars.split(',')]
    options.pars = {key : aux[ii]
                    for ii, key in enumerate(pars_kinds.keys())}

options.unit_multipliers = [float(ii)
                            for ii in options.unit_multipliers.split(',')]
options.wave_dir = [float(ii)
                   for ii in options.wave_dir.split(',')]
aux = options.range.split(',')
options.range = [float(aux[0]), float(aux[1]), int(aux[2])]
options.solver_conf = dict_from_string(options.solver_conf)
options.define_kwargs = dict_from_string(options.define_kwargs)

if options.clear:
    remove_files_patterns(output_dir,
                          ['*.h5', '*.vtk', '*.txt'],
                          ignores=['output_log.txt'],
                          verbose=True)

filename = os.path.join(output_dir, 'options.txt')
ensure_path(filename)
save_options(filename, [('options', vars(options))],
             quote_command_line=True)

pars = apply_units_to_pars(options.pars, pars_kinds,
                           options.unit_multipliers)
output('material parameter names and kinds:')
output(pars_kinds)
output('material parameters with applied unit multipliers:')

```

(continues on next page)

(continued from previous page)

```

output(pars)

pars = Struct(**pars)

if options.mode == 'omega':
    rng = copy(options.range)
    rng[:2] = apply_unit_multipliers(options.range[:2],
                                     ['wave_number', 'wave_number'],
                                     options.unit_multipliers)
    output('wave number range with applied unit multipliers:', rng)

else:
    if options.stepper == 'brillouin':
        raise ValueError('Cannot use "brillouin" stepper in kappa mode!')

    rng = copy(options.range)
    rng[:2] = apply_unit_multipliers(options.range[:2],
                                     ['frequency', 'frequency'],
                                     options.unit_multipliers)
    output('frequency range with applied unit multipliers:', rng)

pb, wdir, bzone, mtxs = assemble_matrices(define, mod, pars, set_wave_dir,
                                          options)

dim = pb.domain.shape.dim

if dim != 2:
    options.plane = 'strain'

if options.save_regions:
    pb.save_regions_as_groups(os.path.join(output_dir, 'regions'))

if options.save_materials:
    save_materials(output_dir, pb, options)

conf = pb.solver_confs['eig']
eig_solver = Solver.any_from_conf(conf)

n_eigs, options.n_eigs = setup_n_eigs(options, pb, mtxs)

get_color = lambda ii: plt.cm.viridis((float(ii)
                                         / (max(options.n_eigs, 2) - 1)))
plot_kwargs = [{'color' : get_color(ii), 'ls' : '', 'marker' : 'o'}
                for ii in range(options.n_eigs)]
get_color_dim = lambda ii: plt.cm.viridis((float(ii) / (max(dim, 2) - 1)))
plot_kwargs_dim = [{'color' : get_color_dim(ii), 'ls' : '', 'marker' : 'o'}
                   for ii in range(dim)]

log_names = []
log_plot_kwargs = []
if options.log_std_waves:
    std_wave_fun, log_names, log_plot_kwargs = get_std_wave_fun(
        pb, options)

```

(continues on next page)

(continued from previous page)

```

else:
    std_wave_fun = None

stepper = get_stepper(rng, pb, options)

if options.mode == 'omega':
    eigenshapes_filename = os.path.join(output_dir,
                                         'frequency-eigenshapes-%s.vtk'
                                         % stepper.suffix)

    if options.stepper == 'linear':
        log = Log([[r'$\lambda_{%d}$' % ii for ii in range(options.n_eigs)],
                  [r'$\omega_{%d}$'
                   % ii for ii in range(options.n_eigs)] + log_names],
                  plot_kwargs=[plot_kwargs, plot_kwargs + log_plot_kwargs],
                  formats=[[ '{:.12e}' * options.n_eigs,
                           '{:.12e}' * (options.n_eigs + len(log_names)) ],
                           yscales=['linear', 'linear'],
                           xlabel=[r'$\kappa$', r'$\kappa$'],
                           ylabel=[r'eigenvalues $\lambda_i$',
                                    r'frequencies $\omega_i$'],
                           show_legends=options.show_legends,
                           is_plot=options.show,
                           log_filename=os.path.join(output_dir, 'frequencies.txt'),
                           aggregate=1000, sleep=0.1)

    else:
        log = Log([[r'$\kappa_{%d}$' % ii for ii in range(dim)],
                  [r'$\omega_{%d}$'
                   % ii for ii in range(options.n_eigs)] + log_names],
                  plot_kwargs=[plot_kwargs_dim,
                               plot_kwargs + log_plot_kwargs],
                  formats=[[ '{:.12e}' * dim,
                           '{:.12e}' * (options.n_eigs + len(log_names)) ],
                           yscales=['linear', 'linear'],
                           xlabel=[r'', r''],
                           ylabel=[r'wave vector $\kappa$',
                                    r'frequencies $\omega_i$'],
                           show_legends=options.show_legends,
                           is_plot=options.show,
                           log_filename=os.path.join(output_dir, 'frequencies.txt'),
                           aggregate=1000, sleep=0.1)

    for aux in stepper:
        if options.stepper == 'linear':
            iv, wmag = aux

        else:
            iv, wmag, wdir = aux

        output('step %d: wave vector %s' % (iv, wmag * wdir))

```

(continues on next page)

(continued from previous page)

```

if options.stepper == 'brillouin':
    pb, _, bzone, mtxs = assemble_matrices(
        define, mod, pars, set_wave_dir, options, wdir=wdir)

    evp_mtxs = build_evp_matrices(mtxs, wmag, options.mode, pb)

    if options.eigs_only:
        eigs = eig_solver(*evp_mtxs, n_eigs=n_eigs,
                          eigenvectors=False)
        svecs = None
    else:
        eigs, svecs = eig_solver(*evp_mtxs, n_eigs=n_eigs,
                                eigenvectors=True)

    omegas, svecs, out = process_evp_results(
        eigs, svecs, wmag, wdir, bzone, pb, mtxs, options,
        std_wave_fun=std_wave_fun
    )
    if options.stepper == 'linear':
        log(*out, x=[wmag, wmag])
    else:
        log(*out, x=[iv, iv])

    save_eigenvectors(eigenshapes_filename % iv, svecs, wmag, wdir, pb)

    gc.collect()

log(save_figure=os.path.join(output_dir, 'frequencies.png'))
log(finished=True)

else:
    eigenshapes_filename = os.path.join(output_dir,
                                         'wave-number-eigenshapes-%s.vtk'
                                         % stepper.suffix)

    log = Log([[r'$\kappa_{%d}$' % ii for ii in range(options.n_eigs)]
               + log_names],
              plot_kwargs=[plot_kwargs + log_plot_kwargs],
              formats=[[ '{:.12e}' * (options.n_eigs + len(log_names)) ],
                      yscale=['linear'],
                      xlabel=[r'$\omega$'],
                      ylabel=[r'wave numbers $\kappa_i$'],
                      show_legends=options.show_legends,
                      is_plot=options.show,
                      log_filename=os.path.join(output_dir, 'wave-numbers.txt'),
                      aggregate=1000, sleep=0.1)

    for io, omega in stepper:
        output('step %d: frequency %s' % (io, omega))

```

(continues on next page)

(continued from previous page)

```

evp_mtxs = build_evp_matrices(mtxs, omega, options.mode, pb)

if options.eigs_only:
    eigs = eig_solver(*evp_mtxs, n_eigs=n_eigs,
                      eigenvectors=False)
    svecs = None

else:
    eigs, svecs = eig_solver(*evp_mtxs, n_eigs=n_eigs,
                             eigenvectors=True)

kappas, svecs, out = process_evp_results(
    eigs, svecs, omega, wdir, bzone, pb, mtxs, options,
    std_wave_fun=std_wave_fun
)
log(*out, x=[omega])

save_eigenvectors(eigenshapes_filename % io, svecs, kappas, wdir,
                  pb)

gc.collect()

log(save_figure=os.path.join(output_dir, 'wave-numbers.png'))
log(finished=True)

if __name__ == '__main__':
    main()

```

linear_elasticity/elastic_contact_planes.py

Description

Elastic contact planes simulating an indentation test.

Four contact planes bounded by polygons (triangles in this case) form a very rigid pyramid shape simulating an indenter.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \sum_{i=1}^4 \int_{\Gamma_i} \underline{v} \cdot f^i(d(\underline{u})) \underline{n}^i = 0 ,$$

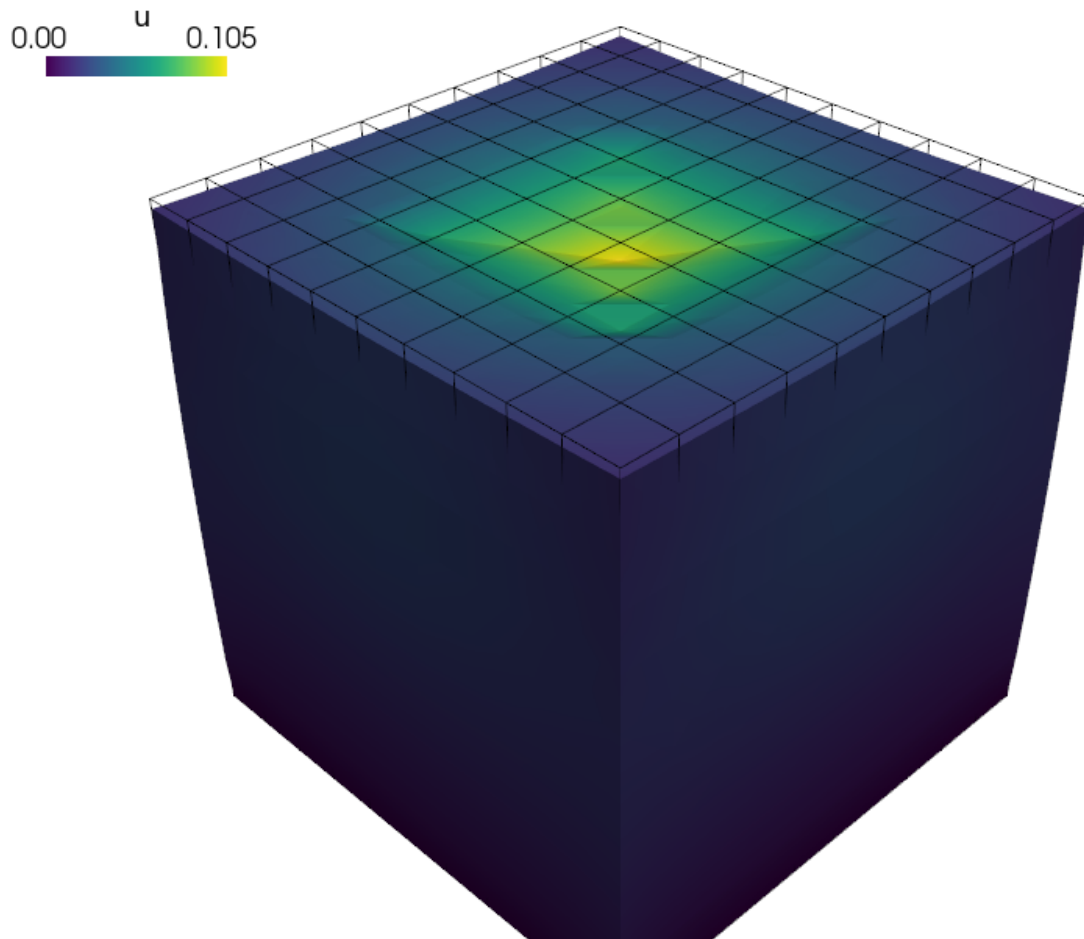
where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} .$$

Notes

Even though the material is linear elastic and small deformations are used, the problem is highly nonlinear due to contacts with the planes.

Checking the tangent matrix by finite differences by setting ‘check’ in ‘nls’ solver configuration to nonzero is rather tricky - the active contact points must not change during the test. This can be ensured by a sufficient initial penetration and large enough contact boundary polygons (hard!), or by tweaking the `dw_contact_plane` term to mask points only by undeformed coordinates.



source code

```
r"""
Elastic contact planes simulating an indentation test.

Four contact planes bounded by polygons (triangles in this case) form a very
rigid pyramid shape simulating an indenter.

Find :math:`\ul{u}` such that:

.. math::
\int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
+ \sum_{i=1}^4 \int_{\Gamma_i} \ul{v} \cdot f^i(d(\ul{u})) \ \ul{n}^i
= 0 \ ;,
```

(continues on next page)

(continued from previous page)

where

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;
```

Notes

Even though the material is linear elastic and small deformations are used, the problem is highly nonlinear due to contacts with the planes.

Checking the tangent matrix by finite differences by setting 'check' in 'nls' solver configuration to nonzero is rather tricky - the active contact points must not change during the test. This can be ensured by a sufficient initial penetration and large enough contact boundary polygons (hard!), or by tweaking the `dw_contact_plane` term to mask points only by undeformed coordinates.

"""

```
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_lame
from six.moves import range

filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'

k = 1e5 # Elastic plane stiffness for positive penetration.
f0 = 1e2 # Force at zero penetration.
dn = 0.2 # x or y component magnitude of normals.
ds = 0.25 # Boundary polygon size in horizontal directions.
az = 0.4 # Anchor z coordinate.

options = {
    'ts' : 'ts',
    'nls' : 'newton',
    'ls' : 'lsd',

    'output_format': 'vtk',
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
}

materials = {
    'solid' : ({
        'D': stiffness_from_lame(dim=3, lam=5.769, mu=3.846),
    },),
    'cp0' : ({
        'f' : [k, f0],
        '.n' : [dn, 0.0, 1.0],
        '.a' : [0.0, 0.0, az],
    },)
```

(continues on next page)

(continued from previous page)

```

        '.bs' : [[0.0, 0.0, az],
                 [-ds, -ds, az],
                 [-ds, ds, az]],
    },),
    'cp1' : ({
        'f' : [k, f0],
        '.n' : [-dn, 0.0, 1.0],
        '.a' : [0.0, 0.0, az],
        '.bs' : [[0.0, 0.0, az],
                 [ds, -ds, az],
                 [ds, ds, az]],
    },),
    'cp2' : ({
        'f' : [k, f0],
        '.n' : [0.0, dn, 1.0],
        '.a' : [0.0, 0.0, az],
        '.bs' : [[0.0, 0.0, az],
                 [-ds, -ds, az],
                 [ds, -ds, az]],
    },),
    'cp3' : ({
        'f' : [k, f0],
        '.n' : [0.0, -dn, 1.0],
        '.a' : [0.0, 0.0, az],
        '.bs' : [[0.0, 0.0, az],
                 [-ds, ds, az],
                 [ds, ds, az]],
    },),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Bottom' : ('vertices in (z < -0.499)', 'facet'),
    'Top' : ('vertices in (z > 0.499)', 'facet'),
}

ebcs = {
    'fixed' : ('Bottom', {'u.all' : 0.0}),
}

equations = {
    'elasticity' :
    """dw_lin_elastic.2.Omega(solid.D, v, u)
    + dw_contact_plane.2.Top(cp0.f, cp0.n, cp0.a, cp0.bs, v, u)
    + dw_contact_plane.2.Top(cp1.f, cp1.n, cp1.a, cp1.bs, v, u)
    + dw_contact_plane.2.Top(cp2.f, cp2.n, cp2.a, cp2.bs, v, u)
    + dw_contact_plane.2.Top(cp3.f, cp3.n, cp3.a, cp3.bs, v, u)
    """
}

```

(continues on next page)

(continued from previous page)

```

    = 0""",
}

solvers = {
    'lsd' : ('ls.scipy_direct', {}),
    'lsi' : ('ls.petsc', {
        'method' : 'cg',
        'eps_r' : 1e-8,
        'i_max' : 3000,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 10,
        'eps_a' : 1e-10,
        'check' : 0,
        'delta' : 1e-6,
    }),
}

def main():
    import os
    from itertools import product

    import numpy as nm
    import matplotlib.pyplot as plt

    from sfepy.discrete.fem import MeshIO
    import sfepy.linalg as la
    from sfepy.mechanics.contact_bodies import (ContactPlane, plot_polygon,
                                                plot_points)

    conf_dir = os.path.dirname(__file__)
    io = MeshIO.any_from_filename(filename_mesh, prefix_dir=conf_dir)
    bb = io.read_bounding_box()
    outline = [vv for vv in product(*bb.T)]

    ax = plot_points(None, nm.array(outline), 'r*')
    for name in ['cp%d' % ii for ii in range(4)]:
        cpc = materials[name][0]
        cp = ContactPlane(cpc['.a'], cpc['.n'], cpc['.bs'])

        v1, v2 = la.get_perpendiculars(cp.normal)

        ax = plot_polygon(ax, cp.bounds)
        ax = plot_polygon(ax, nm.r_[cp.anchor[None, :],
                                    cp.anchor[None, :] + cp.normal[None, :]])
        ax = plot_polygon(ax, nm.r_[cp.anchor[None, :],
                                    cp.anchor[None, :] + v1])
        ax = plot_polygon(ax, nm.r_[cp.anchor[None, :],
                                    cp.anchor[None, :] + v2])

    plt.show()

```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':  
    main()
```

linear_elasticity/elastic_contact_sphere.py

Description

Elastic contact sphere simulating an indentation test.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Gamma} \underline{v} \cdot \underline{f}(d(\underline{u})) \underline{n}(\underline{u}) = 0 ,$$

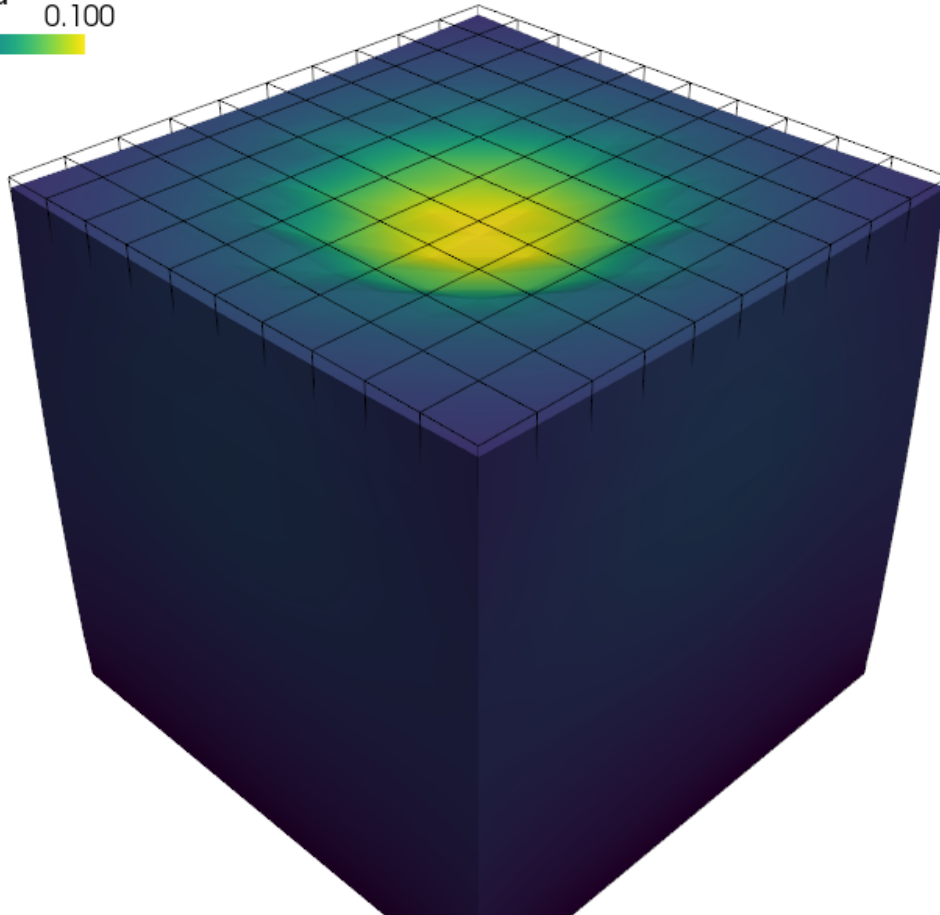
where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} .$$

Notes

Even though the material is linear elastic and small deformations are used, the problem is highly nonlinear due to contacts with the sphere. See also `elastic_contact_planes.py` example.

0.00 $\underline{u}$ 0.100

source code

```

r"""
Elastic contact sphere simulating an indentation test.

Find :math:\ul{u}` such that:

.. math::
\int_{\Omega} D_{ijkl} \ul{e}_{ij}(\ul{v}) \ul{e}_{kl}(\ul{u})
+ \int_{\Gamma} \ul{v} \cdot f(d(\ul{u})) \ul{n}(\ul{u})
= 0 \ ;,

where

.. math::
D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
\lambda \delta_{ij} \delta_{kl}
\ ;.

Notes
-----

Even though the material is linear elastic and small deformations are used, the
problem is highly nonlinear due to contacts with the sphere. See also
elastic_contact_planes.py example.
"""
from __future__ import absolute_import
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_lame

filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'

k = 1e5 # Elastic sphere stiffness for positive penetration.
f0 = 1e-2 # Force at zero penetration.

options = {
    'nls' : 'newton',
    'ls' : 'ls',

    'output_format': 'vtk',
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
}

materials = {
    'solid' : ({
        'D': stiffness_from_lame(dim=3, lam=5.769, mu=3.846),
    },),
    'cs' : ({
        'f' : [k, f0],
        'c' : [0.0, 0.0, 1.2],
    })
}

```

(continues on next page)

(continued from previous page)

```

        '.r' : 0.8,
    },),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Bottom' : ('vertices in (z < -0.499)', 'facet'),
    'Top' : ('vertices in (z > 0.499)', 'facet'),
}

ebcs = {
    'fixed' : ('Bottom', {'u.all' : 0.0}),
}

equations = {
    'elasticity' :
        """dw_lin_elastic.2.Omega(solid.D, v, u)
        + dw_contact_sphere.2.Top(cs.f, cs.c, cs.r, v, u)
        = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 20,
        'eps_a' : 1e-1,
        'ls_on' : 2.0,
        'check' : 0,
        'delta' : 1e-6,
    }),
}

def main():
    import os
    from itertools import product

    import numpy as nm
    import matplotlib.pyplot as plt

    from sfepy.discrete.fem import MeshIO
    from sfepy.mechanics.contact_bodies import ContactSphere, plot_points

    conf_dir = os.path.dirname(__file__)
    io = MeshIO.any_from_filename(filename_mesh, prefix_dir=conf_dir)
    bb = io.read_bounding_box()
    outline = [vv for vv in product(*bb.T)]

```

(continues on next page)

(continued from previous page)

```

ax = plot_points(None, nm.array(outline), 'r*')
csc = materials['cs'][0]
cs = ContactSphere(csc['.c'], csc['.r'])

pps = (bb[1] - bb[0]) * nm.random.rand(5000, 3) + bb[0]
mask = cs.mask_points(pps, 0.0)

ax = plot_points(ax, cs.centre[None, :], 'b*', ms=30)
ax = plot_points(ax, pps[mask], 'kv')
ax = plot_points(ax, pps[~mask], 'r.')

plt.show()

if __name__ == '__main__':
    main()

```

linear_elasticity/elastic_shifted_periodic.py

Description

Linear elasticity with linear combination constraints and periodic boundary conditions.

The linear combination constraints are used to apply periodic boundary conditions with a shift in the second axis direction.

Find $\underline{u}$ such that:

$$\begin{aligned}
 \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) &= - \int_{\Gamma_{bottom}} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \quad \forall \underline{v}, \\
 \underline{u} &= 0 \text{ on } \Gamma_{left}, \\
 u_1 &= u_2 = 0 \text{ on } \Gamma_{right}, \\
 \underline{u}(\underline{x}) &= \underline{u}(\underline{y}) \text{ for } \underline{x} \in \Gamma_{bottom}, \underline{y} \in \Gamma_{top}, \underline{y} = P_1(\underline{x}), \\
 \underline{u}(\underline{x}) &= \underline{u}(\underline{y}) + a(\underline{y}) \text{ for } \underline{x} \in \Gamma_{near}, \underline{y} \in \Gamma_{far}, \underline{y} = P_2(\underline{x}),
 \end{aligned}$$

where

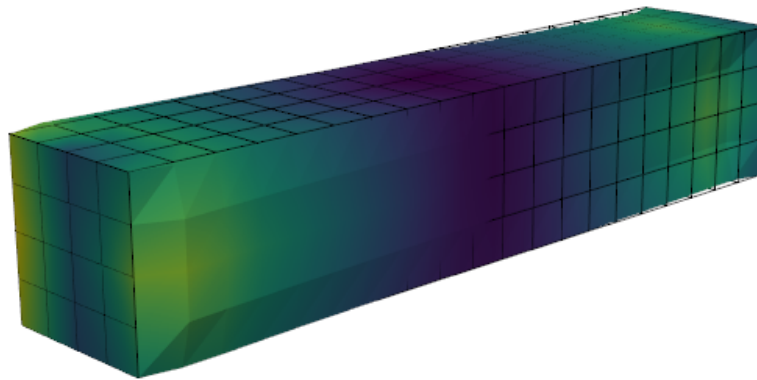
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

and the traction $\underline{\sigma} \cdot \underline{n} = \bar{p} \underline{I} \cdot \underline{n}$ is given in terms of traction pressure $\bar{p}$. The function $a(\underline{y})$ is given (the shift), P_1 and P_2 are the periodic coordinate mappings.

View the results using:

```
sfepy-view block.vtk -f u:wu:f2.0:p0 1:vw:p0 von_mises_stress:p1
```

von_mises_stress
0.124 2.43

source code

```
r"""
Linear elasticity with linear combination constraints and periodic boundary
conditions.

The linear combination constraints are used to apply periodic boundary
conditions with a shift in the second axis direction.

Find :math:\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ul{e}_{ij}(\ul{v}) \ul{e}_{kl}(\ul{u})
    = - \int_{\Gamma_{\text{bottom}}} \ul{v} \cdot \ul{\sigma} \cdot \ul{n}
    \;, \quad \text{forall } \ul{v} \;,

    \ul{u} = 0 \text{ on } \Gamma_{\text{left}} \;,

    u_1 = u_2 = 0 \text{ on } \Gamma_{\text{right}} \;,

    \ul{u}(\ul{x}) = \ul{u}(\ul{y}) \text{ for }
    \ul{x} \in \Gamma_{\text{bottom}}, \ul{y} \in \Gamma_{\text{top}},
    \ul{y} = P_1(\ul{x}) \;,

```

(continues on next page)

(continued from previous page)

$$\begin{aligned} \mathbf{u}(\mathbf{x}) &= \mathbf{u}(\mathbf{y}) + \mathbf{a}(\mathbf{y}) \quad \text{for } \\ \mathbf{x} &\in \Gamma_{\text{near}}, \quad \mathbf{y} \in \Gamma_{\text{far}}, \\ \mathbf{y} &= P_2(\mathbf{x}) \quad ;, \end{aligned}$$

where

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;
```

and the traction $\boldsymbol{\sigma} \cdot \mathbf{n} = \bar{p} \mathbf{I} \cdot \mathbf{n}$ is given in terms of traction pressure $\bar{p}$. The function $\mathbf{a}(\mathbf{y})$ is given (the shift), P_1 and P_2 are the periodic coordinate mappings.

View the results using::

```
sfepy-view block.vtk -f u:wu:f2.0:p0 1:vw:p0 von_mises_stress:p1
"""
from __future__ import absolute_import
import numpy as nm

from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.mechanics.tensors import get_von_mises_stress
import sfepy.discrete.fem.periodic as per
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/block.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process'
}

def post_process(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = pb.evaluate
    stress = ev('ev_cauchy_stress.2.Omega(solid.D, u)', mode='el_avg')

    vms = get_von_mises_stress(stress.squeeze())
    vms.shape = (vms.shape[0], 1, 1, 1)
    out['von_mises_stress'] = Struct(name='output_data', mode='cell',
                                     data=vms, dofs=None)

    return out
```

(continues on next page)

(continued from previous page)

```

def linear_tension(ts, coor, mode=None, **kwargs):
    if mode == 'qp':
        val = 0.1 * nm.sin(coor[:, 0] / 10.)

        return {'val' : val.reshape((coor.shape[0], 1, 1))}

def get_shift(ts, coors, region=None):
    val = nm.zeros_like(coors, dtype=nm.float64)

    val[:, 1] = 0.1 * coors[:, 0]
    return val

functions = {
    'get_shift' : (get_shift,),
    'linear_tension' : (linear_tension,),
    'match_y_plane' : (per.match_y_plane,),
    'match_z_plane' : (per.match_z_plane,),
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
}

materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=5.769, mu=3.846),
    },),
    'load' : (None, 'linear_tension')
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
    'Right' : ('vertices in (x > 4.99)', 'facet'),
    'Bottom' : ('vertices in (z < -0.99)', 'facet'),
    'Top' : ('vertices in (z > 0.99)', 'facet'),
    'Near' : ('vertices in (y < -0.99)', 'facet'),
    'Far' : ('vertices in (y > 0.99)', 'facet'),
}

ebcs = {
    'fix1' : ('Left', {'u.all' : 0.0}),
    'fix2' : ('Right', {'u.[1,2]' : 0.0}),
}

epbcs = {

```

(continues on next page)

(continued from previous page)

```

    'periodic' : ([ 'Bottom', 'Top'], { 'u.all' : 'u.all'}, 'match_z_plane'),
}

lcbcs = {
    'shifted' : (('Near', 'Far'),
                 { 'u.all' : 'u.all'},
                 'match_y_plane', 'shifted_periodic',
                 'get_shift'),
}

equations = {
    'elasticity' : """
        dw_lin_elastic.2.Omega(solid.D, v, u)
        = -dw_surface_ltr.2.Bottom(load.val, v)
    """,
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
    }),
}

```

linear_elasticity/elastodynamic.py

Description

The linear elastodynamics solution of an iron plate impact problem.

Find $\underline{u}$ such that:

$$\int_{\Omega} \rho \underline{v} \frac{\partial^2 \underline{u}}{\partial t^2} + \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

Notes

The used elastodynamics solvers expect that the total vector of DOFs contains three blocks in this order: the displacements, the velocities, and the accelerations. This is achieved by defining three unknown variables 'u', 'du', 'ddu' and the corresponding test variables, see the *variables* definition. Then the solver can automatically extract the mass, damping (zero here), and stiffness matrices as diagonal blocks of the global matrix. Note also the use of the 'dw_zero' (do-nothing) term that prevents the velocity-related variables to be removed from the equations in the absence of a damping term. This manual declaration of variables and 'dw_zero' can be avoided by setting the 'auto_transform_equations' option to True, see [linear_elasticity/seismic_load.py](#) or [multi_physics/piezo_elastodynamic.py](#).

Usage Examples

Run with the default settings (the Newmark method, 3D problem, results stored in output/ed/):

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py
```

Solve using the Bathe method:

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -O "tss_name='tsb'"
```

View the resulting displacements on the deforming mesh (1000x magnified), Cauchy strain and stress using:

```
sfepy-view output/ed/user_block.h5 -f u:wu:f1e3:p0 1:vw:p0 cauchy_strain:p1 cauchy_
↪ stress:p2
```

Solve in 2D using the explicit Velocity-Verlet method with adaptive time-stepping and save all time steps (see plot_times.py use below):

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3), ↪
↪ shape=(61, 61), tss_name='tsvv', tsc_name='tscedb', adaptive=True, save_times='all'"
```

View the resulting velocities on the deforming mesh (1000x magnified) using:

```
sfepy-view output/ed/user_block.h5 -2 --grid-vector1=1.2,0,0 -f du:wu:f1e3:p0 1:vw:p0
```

Plot the adaptive time steps (available at times according to 'save_times' option!):

```
python3 sfepy/scripts/plot_times.py output/ed/user_block.h5 -l
```

Again, solve in 2D using the explicit Velocity-Verlet method with adaptive time-stepping and save all time steps. Now the used time step control is suitable for linear problems solved by a direct solver: it employs a heuristic that tries to keep the time step size constant for several consecutive steps, reducing so the need for a new matrix factorization. Run:

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3), ↪
↪ shape=(61, 61), tss_name='tsvv', tsc_name='tscedl', adaptive=True, save_times='all'"
```

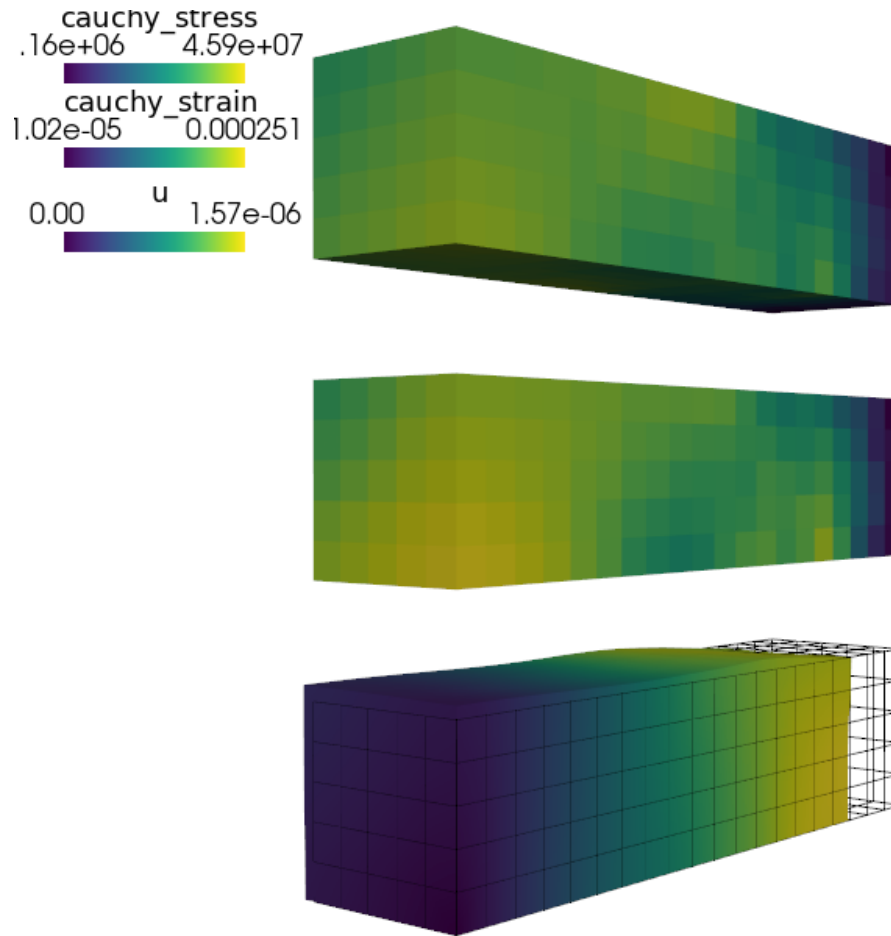
The resulting velocities and adaptive time steps can again be plotted by the commands shown above.

Use the central difference explicit method with the reciprocal mass matrix algorithm¹ and view the resulting stress waves:

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3), ↪
↪ shape=(61, 61), tss_name=tscd, tsc_name=tscedl, adaptive=False, ls_name=lsrmm, mass_
↪ beta=0.5, mass_lumping=row_sum, fast_rmm=True, save_times=all"

sfepy-view output/ed/user_block.h5 -2 --grid-vector1=1.2,0,0 -f cauchy_stress:wu:f1e3:p0 ↪
↪ 1:vw:p0
```

¹ González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018. Inverse mass matrix via the method of localized Lagrange multipliers. International Journal for Numerical Methods in Engineering 113, 277–295. <https://doi.org/10.1002/nme.5613>



source code

```

r"""
The linear elastodynamics solution of an iron plate impact problem.

Find :math:\ul{u} such that:

.. math::
    \int_{\Omega} \rho \ul{v} \text{pdiff}\ul{u}\{t\}
    + \int_{\Omega} D_{ijkl} \text{e}_{ij}(\ul{v}) \text{e}_{kl}(\ul{u})
    = 0
    \;, \quad \text{forall } \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

Notes
-----

```

(continues on next page)

(continued from previous page)

The used elastodynamics solvers expect that the total vector of DOFs contains three blocks in this order: the displacements, the velocities, and the accelerations. This is achieved by defining three unknown variables ``u``, ``du``, ``ddu`` and the corresponding test variables, see the `variables` definition. Then the solver can automatically extract the mass, damping (zero here), and stiffness matrices as diagonal blocks of the global matrix. Note also the use of the `dw_zero` (do-nothing) term that prevents the velocity-related variables to be removed from the equations in the absence of a damping term. This manual declaration of variables and `dw_zero` can be avoided by setting the `auto_transform_equations` option to True, see :ref:`linear\_elasticity-seismic\_load` or :ref:`multi\_physics-piezo\_elastodynamic`.

Usage Examples

Run with the default settings (the Newmark method, 3D problem, results stored in `output/ed/`)::

```
sfePy-run sfePy/examples/linear_elasticity/elastodynamic.py
```

Solve using the Bathe method::

```
sfePy-run sfePy/examples/linear_elasticity/elastodynamic.py -O "tss_name='tsb'"
```

View the resulting displacements on the deforming mesh (1000x magnified), Cauchy strain and stress using::

```
sfePy-view output/ed/user_block.h5 -f u:wu:f1e3:p0 1:vw:p0 cauchy_strain:p1 cauchy_
↪ stress:p2
```

Solve in 2D using the explicit Velocity-Verlet method with adaptive time-stepping and save all time steps (see `plot_times.py` use below)::

```
sfePy-run sfePy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3),
↪ shape=(61, 61), tss_name='tsvv', tsc_name='tscedb', adaptive=True, save_times='all'"
```

View the resulting velocities on the deforming mesh (1000x magnified) using::

```
sfePy-view output/ed/user_block.h5 -2 --grid-vector1=1.2,0,0 -f du:wu:f1e3:p0 1:vw:p0
```

Plot the adaptive time steps (available at times according to 'save_times' option!)::

```
python3 sfePy/scripts/plot_times.py output/ed/user_block.h5 -l
```

Again, solve in 2D using the explicit Velocity-Verlet method with adaptive time-stepping and save all time steps. Now the used time step control is suitable for linear problems solved by a direct solver: it employs a heuristic that tries to keep the time step size constant for several consecutive steps, reducing so the need for a new matrix factorization. Run::

(continues on next page)

(continued from previous page)

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3),
↳shape=(61, 61), tss_name='tsvv', tsc_name='tscedl', adaptive=True, save_times='all'"
```

The resulting velocities and adaptive time steps can again be plotted by the commands shown above.

Use the central difference explicit method with the reciprocal mass matrix algorithm [1]_ and view the resulting stress waves::

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic.py -d "dims=(5e-3, 5e-3),
↳shape=(61, 61), tss_name=tscd, tsc_name=tscedl, adaptive=False, ls_name=lsrmm, mass_
↳beta=0.5, mass_lumping=row_sum, fast_rmm=True, save_times='all'"
```

```
sfepy-view output/ed/user_block.h5 -2 --grid-vector1=1.2,0,0 -f cauchy_
↳stress:wu:f1e3:p0 1:vw:p0
```

```
.. [1] González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018.
    Inverse mass matrix via the method of localized Lagrange multipliers.
    International Journal for Numerical Methods in Engineering 113, 277-295.
    https://doi.org/10.1002/nme.5613
"""
```

```
import numpy as nm
```

```
import sfepy.mechanics.matcoefs as mc
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
```

```
def define(
    E=200e9, nu=0.3, rho=7800,
    plane='strain',
    dims=(1e-2, 2.5e-3, 2.5e-3),
    shape=(21, 6, 6),
    v0=1.0,
    ct1=1.5,
    dt=None,
    edt_safety=0.2,
    tss_name='tsn',
    tsc_name='tscedl',
    adaptive=False,
    ls_name='lsd',
    mass_beta=0.0,
    mass_lumping='none',
    fast_rmm=False,
    active_only=False,
    save_times=20,
    output_dir='output/ed',
):
    """
    Parameters
    -----
    E, nu, rho: material parameters
    plane: plane strain or stress hypothesis
```

(continues on next page)

(continued from previous page)

```

dims: physical dimensions of the block (L, d, x)
shape: numbers of mesh vertices along each axis
v0: initial impact velocity
ct1: final time in L / "longitudinal wave speed" units
dt: time step (None means automatic)
edt_safety: safety factor time step multiplier for explicit schemes,
    if dt is None
tss_name: time stepping solver name (see "solvers" section)
tsc_name: time step controller name (see "solvers" section)
adaptive: use adaptive time step control
ls_name: linear system solver name (see "solvers" section)
mass_beta: averaged mass matrix parameter 0 <= beta <= 1
mass_lumping: mass matrix lumping ('row_sum', 'hrz' or 'none')
fast_rmm: use zero inertia term with lsrmm
save_times: number of solutions to save
output_dir: output directory
    """
    dim = len(dims)

    lam, mu = mc.lame_from_youngpoisson(E, nu, plane=plane)
    # Longitudinal and shear wave propagation speeds.
    cl = nm.sqrt((lam + 2.0 * mu) / rho)
    cs = nm.sqrt(mu / rho)

    # Element size.
    L, d = dims[:2]
    H = L / (nm.max(shape) - 1)

    # Time-stepping parameters.
    if dt is None:
        # For implicit schemes, dt based on the Courant number C0 = dt * cl / H
        # equal to 1.
        dt = H / cl # C0 = 1
        if tss_name in ('tsvv', 'tscd'):
            # For explicit schemes, use a safety margin.
            dt *= edt_safety

    t1 = ct1 * L / cl

    def mesh_hook(mesh, mode):
        """
        Generate the block mesh.
        """
        if mode == 'read':
            mesh = gen_block_mesh(dims, shape, 0.5 * nm.array(dims),
                                  name='user_block', verbose=False)
            return mesh

        elif mode == 'write':
            pass

    def post_process(out, problem, state, extend=False):

```

(continues on next page)

(continued from previous page)

```

"""
Calculate and output strain and stress for given displacements.
"""
from sfepy.base.base import Struct

ev = problem.evaluate
strain = ev('ev_cauchy_strain.i.Omega(u)', mode='el_avg', verbose=False)
stress = ev('ev_cauchy_stress.i.Omega(solid.D, u)', mode='el_avg',
            copy_materials=False, verbose=False)

out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                              data=strain)
out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                              data=stress)

return out

filename_mesh = UserMeshIO(mesh_hook)

regions = {
    'Omega' : 'all',
    'Impact' : ('vertices in (x < 1e-12)', 'facet'),
}
if dim == 3:
    regions.update({
        'Symmetry-y' : ('vertices in (y < 1e-12)', 'facet'),
        'Symmetry-z' : ('vertices in (z < 1e-12)', 'facet'),
    })

# Iron.
materials = {
    'solid' : ({
        'D': mc.stiffness_from_youngpoisson(dim=dim, young=E, poisson=nu,
                                             plane=plane),
        'rho': rho,
        '.lumping' : mass_lumping,
        '.beta' : mass_beta,
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i' : 2,
}

# Notes:
# 1. The order of the variables in the solution vector is specified here
#    (3rd tuple member), since that specific order is expected by the
#    elastodynamic time-stepping solvers.

```

(continues on next page)

(continued from previous page)

```

# 2. For the same reason, we won't explicitly define below the equations
#   du = du/dt and ddu = ddu/dt - these are implicitly defined by
#   the time-stepping solver. see the `step()` method of the solvers.
variables = {
    'u' : ('unknown field', 'displacement', 0),
    'du' : ('unknown field', 'displacement', 1),
    'ddu' : ('unknown field', 'displacement', 2),
    'v' : ('test field', 'displacement', 'u'),
    'dv' : ('test field', 'displacement', 'du'),
    'ddv' : ('test field', 'displacement', 'ddu'),
}
# The mapping of variables for the elastodynamics solvers - keys are given,
# values correspond to the names of the actual variables.
var_names = {'u' : 'u', 'du' : 'du', 'ddu' : 'ddu'}

ebcs = {
    'Impact' : ('Impact', {'u.0' : 0.0, 'du.0' : 0.0, 'ddu.0' : 0.0}),
}
if dim == 3:
    ebcs.update({
        'Symmetry-y' : ('Symmetry-y',
                        {'u.1' : 0.0, 'du.1' : 0.0, 'ddu.1' : 0.0}),
        'Symmetry-z' : ('Symmetry-z',
                        {'u.2' : 0.0, 'du.2' : 0.0, 'ddu.2' : 0.0}),
    })

def get_ic(coor, ic, mode='u'):
    val = nm.zeros_like(coor)
    if mode == 'u':
        val[:, 0] = 0.0

    elif mode == 'du':
        val[:, 0] = -1.0

    return val

functions = {
    'get_ic_u' : (get_ic,),
    'get_ic_du' : (lambda coor, ic: get_ic(coor, None, mode='du'),),
}

ics = {
    'ic' : ('Omega', {'u.all' : 'get_ic_u', 'du.all' : 'get_ic_du'}),
}

if (ls_name == 'lsrmm') and fast_rmm:
    # Speed up residual calculation, as M is not used with lsrmm.
    term = 'dw_zero.i.Omega(ddv, ddu)'

else:
    term = 'de_mass.i.Omega(solid.rho, solid.lumping, solid.beta, ddv, ddu)'

```

(continues on next page)

(continued from previous page)

```

equations = {
    'balance_of_forces' :
    term + """
    + dw_zero.i.Omega(dv, du)
    + dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

solvers = {
    'lsd' : ('ls.auto_direct', {
        # Reuse the factorized linear system from the first time step.
        'use_presolve' : True,
        # Speed up the above by omitting the matrix digest check used
        # normally for verification that the current matrix corresponds to
        # the factorized matrix stored in the solver instance. Use with
        # care!
        'use_mtx_digest' : False,
    }),
    'lsi' : ('ls.petsc', {
        'method' : 'cg',
        'precond' : 'icc',
        'i_max' : 150,
        'eps_a' : 1e-32,
        'eps_r' : 1e-8,
        'verbose' : 2,
    }),
    'lsrmm' : ('ls.rmm', {
        'rmm_term' : """de_mass.i.Omega(solid.rho, solid.lumping,
                                         solid.beta, ddv, ddu)""",
        'debug' : False,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
        'eps_r' : 1e-6,
    }),
    'tsvv' : ('ts.velocity_verlet', {
        # Explicit method.
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'var_names' : var_names,
        'verbose' : 1,
    }),
    'tscd' : ('ts.central_difference', {
        # Explicit method. Supports ls.rmm.
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,

```

(continues on next page)

(continued from previous page)

```

        'n_step' : None,

        'is_linear' : True,

        'var_names' : var_names,
        'verbose' : 1,
    }},
    'tsn' : ('ts.newmark', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'beta' : 0.25,
        'gamma' : 0.5,

        'var_names' : var_names,
        'verbose' : 1,
    }},
    'tsga' : ('ts.generalized_alpha', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'rho_inf' : 0.5,
        'alpha_m' : None,
        'alpha_f' : None,
        'beta' : None,
        'gamma' : None,

        'var_names' : var_names,
        'verbose' : 1,
    }},
    'tsb' : ('ts.bathe', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'var_names' : var_names,
        'verbose' : 1,
    }},
    'tscedb' : ('tsced_basic', {
        'eps_r' : (1e-4, 1e-1),
        'eps_a' : (1e-8, 5e-2),

```

(continues on next page)

(continued from previous page)

```

        'fmin' : 0.3,
        'fmax' : 2.5,
        'fsafety' : 0.85,
    }},
    'tscedl' : ('tsc.ed_linear', {
        'eps_r' : (1e-4, 1e-1),
        'eps_a' : (1e-8, 5e-2),
        'fmin' : 0.3,
        'fmax' : 2.5,
        'fsafety' : 0.85,
        'red_factor' : 0.9,
        'inc_wait' : 10,
        'min_inc_factor' : 1.5,
    }},
}

options = {
    'ts' : tss_name,
    'tsc' : tsc_name if adaptive else None,
    'nls' : 'newton',
    'ls' : ls_name,

    'save_times' : save_times,

    'active_only' : active_only,
    'auto_transform_equations' : False,

    'output_format' : 'h5',
    'output_dir' : output_dir,
    'post_process_hook' : 'post_process',
}

return locals()

```

linear_elasticity/elastodynamic_identification.py

Description

The linear elastodynamics solution of an iron plate impact problem with identification of material parameters from simulated measurement data.

Find $\underline{u}$ such that:

$$\int_{\Omega} \rho \underline{v} \frac{\partial^2 \underline{u}}{\partial t^2} + \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$\begin{aligned} D_{ijkl} &= \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}, \\ \lambda &= E\nu/((1+\nu)(1-2\nu)), \\ \mu &= E/2(1+\nu). \end{aligned}$$

Usage Examples

- Run without the identification:

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic_identification.py
sfepy-view output/edi/user_block.h5 -f u:wu:f1e3:p0 1:vw:p0
```

- Get help:

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py -h
```

- Run the identification with default parameters, show live plot of convergence and launch ipython shell after the computation:

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --plot-log
↪ --shell
```

Result figures are in output/edi, if not changed using `--output-dir` option.

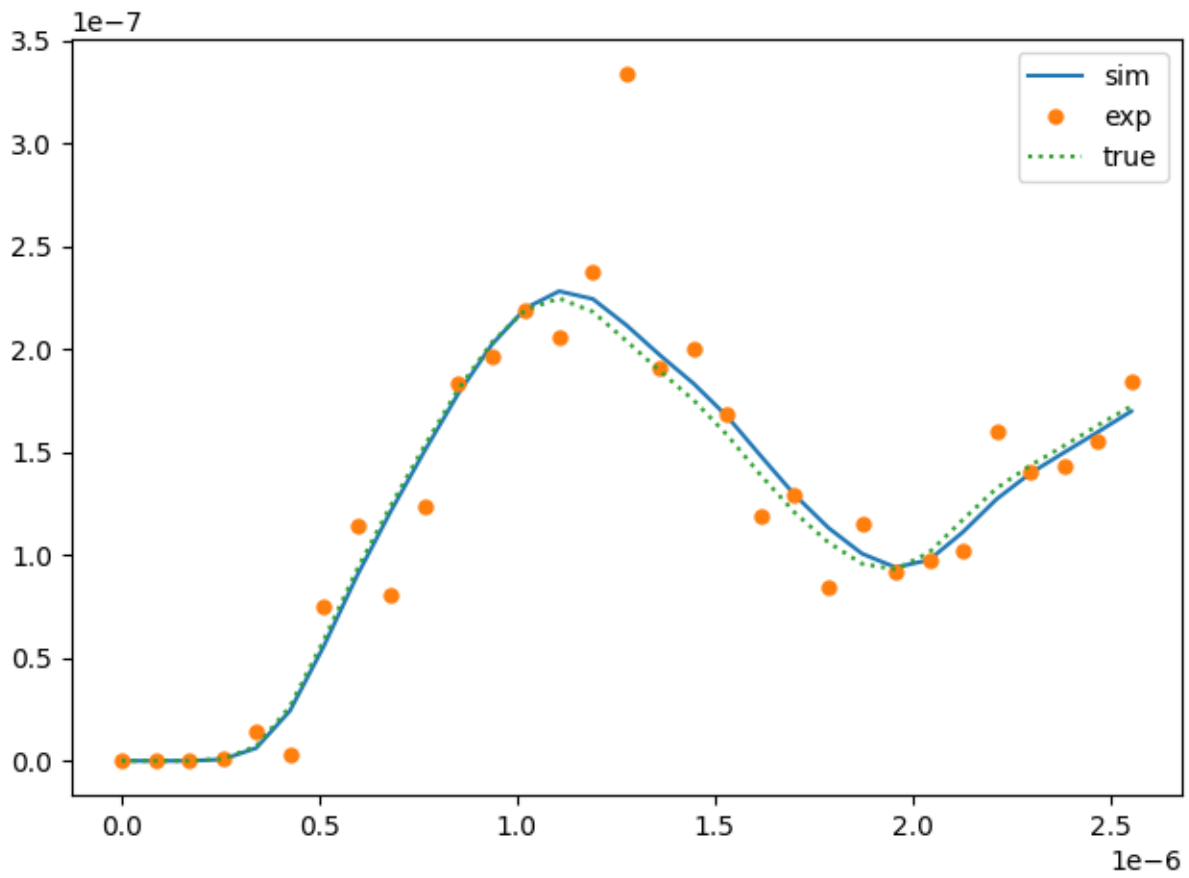
- Check the Jacobian matrix by finite differences:

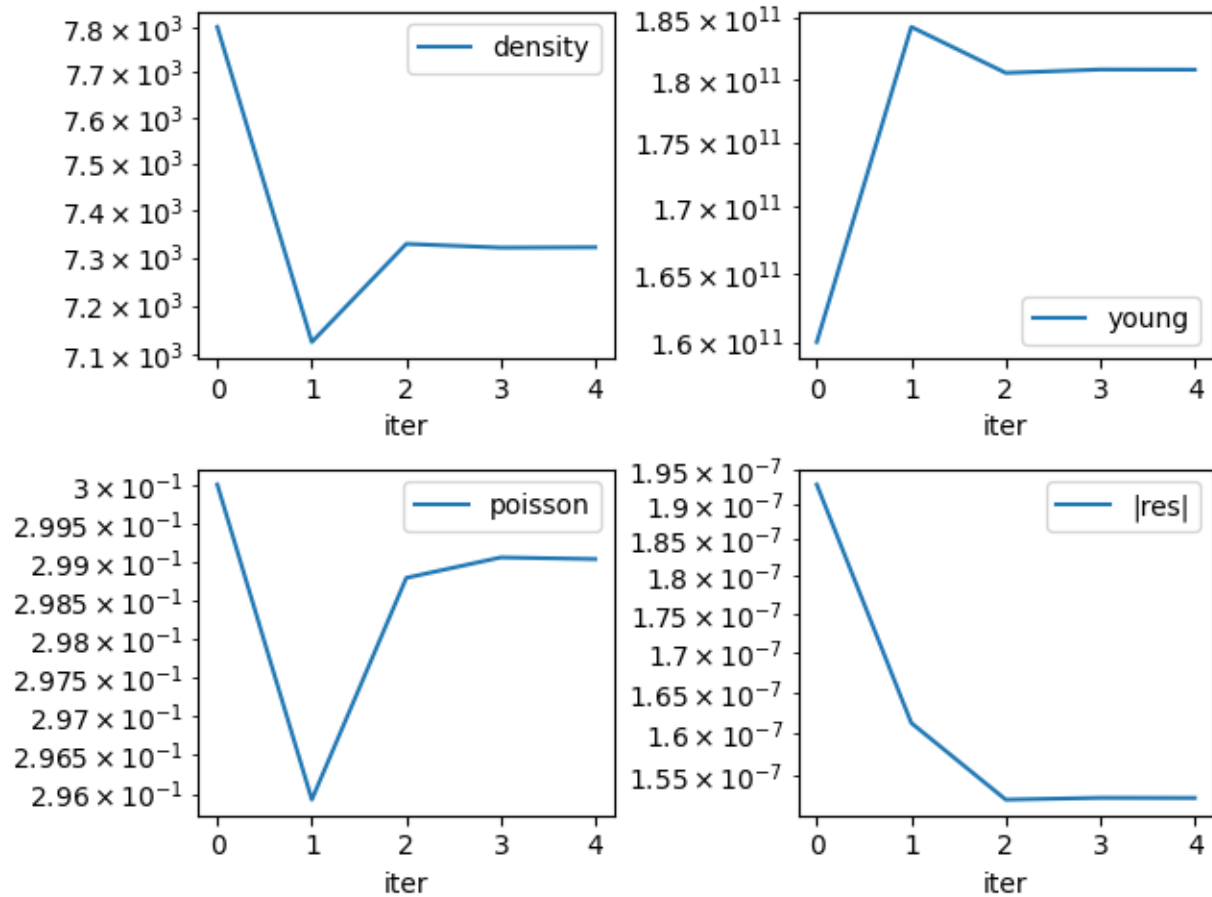
```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --opt-
↪ conf=max_nfev=1 --check-jac --shell
```

- Identify also the damping parameters (zero by default):

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --par-
↪ names=young,poisson,density,alpha,beta --plot-log --shell
```

See also [linear_elasticity/elastodynamic.py](#).





source code

```
r"""
The linear elastodynamics solution of an iron plate impact problem with
identification of material parameters from simulated measurement data.

Find :math:\ul{u} such that:

.. math::
\int_{\Omega} \rho \ul{v} \pdiff{\ul{u}}{t}
+ \int_{\Omega} D_{ijkl} e_{ij}(\ul{v}) e_{kl}(\ul{u})
= 0
\;, \quad \forall \ul{v} \;,

where

.. math::
D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
\lambda \delta_{ij} \delta_{kl} \;,

\lambda = E \nu / ((1 + \nu)(1 - 2\nu)), \quad \mu = E / 2(1 + \nu)
\;.

Usage Examples
```

(continues on next page)

(continued from previous page)

- Run without the identification::

```
sfepy-run sfepy/examples/linear_elasticity/elastodynamic_identification.py
sfepy-view output/edi/user_block.h5 -f u:wu:f1e3:p0 1:vw:p0
```

- Get help::

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py -h
```

- Run the identification with default parameters, show live plot of convergence and launch ipython shell after the computation::

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --plot-log -
↪-shell
```

Result figures are in output/edi, if not changed using --output-dir option.

- Check the Jacobian matrix by finite differences::

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --opt-
↪conf=max_nfev=1 --check-jac --shell
```

- Identify also the damping parameters (zero by default)::

```
python3 sfepy/examples/linear_elasticity/elastodynamic_identification.py --par-
↪names=young,poisson,density,alpha,beta --plot-log --shell
```

See also :ref:`linear\_elasticity-elastodynamic`.

"""

```
from argparse import ArgumentParser, RawDescriptionHelpFormatter
import sys
import os.path as op
from functools import partial
```

```
import numpy as nm
import scipy.sparse as sps
import matplotlib.pyplot as plt
```

```
from sfepy.base.base import output, Struct, IndexedStruct
from sfepy.base.conf import dict_from_string as parse_as_dict
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.homogenization.utils import define_box_regions
from sfepy.solvers import register_solver, Solver
from sfepy.base.timing import Timer
import sfepy.mechanics.matcoefs as mc
```

```
def define(
    young=200e9, poisson=0.3, density=7800,
    alpha=0.0, beta=0.0,
```

(continues on next page)

(continued from previous page)

```

        plane='strain',
        dims=(1e-2, 2.5e-3, 2.5e-3),
        shape=(21, 6, 6),
        v0=1.0,
        ct1=1.5,
        dt=None,
        active_only=False,
        save_times=20,
        output_dir='output/edi',
        **kwargs,
    ):
        """
        Parameters
        -----
        young, poisson, density: material parameters
        plane: plane strain or stress hypothesis
        dims: physical dimensions of the block (L, d, x)
        shape: numbers of mesh vertices along each axis
        v0: initial impact velocity
        ct1: final time in L / "longitudinal wave speed" units
        dt: time step (None means automatic)
        save_times: number of time steps to save
        output_dir: output directory
        """
        dim = len(dims)

        lam, mu = mc.lame_from_youngpoisson(young, poisson, plane=plane)
        # Longitudinal and shear wave propagation speeds.
        cl = nm.sqrt((lam + 2.0 * mu) / density)
        cs = nm.sqrt(mu / density)

        # Element size.
        L, d = dims[:2]
        H = L / (nm.max(shape) - 1)

        # Time-stepping parameters.
        if dt is None:
            # For implicit schemes, dt based on the Courant number C0 = dt * cl / H
            # equal to 1.
            dt = H / cl # C0 = 1

        t1 = ct1 * L / cl

        def mesh_hook(mesh, mode):
            """
            Generate the block mesh.
            """
            if mode == 'read':
                mesh = gen_block_mesh(dims, shape, 0.5 * nm.array(dims),
                                      name='user_block', verbose=False)
            return mesh

```

(continues on next page)

(continued from previous page)

```

elif mode == 'write':
    pass

filename_mesh = UserMeshIO(mesh_hook)

bbox = [[0] * dim, dims]
regions = define_box_regions(dim, bbox[0], bbox[1], 1e-5)

dx = dims[0] / (shape[0] - 1)
sx = int(0.2 * (shape[0] - 1)) * dx
regions.update({
    'Omega' : 'all',
    'Plane' : ('vertices in (x > %.12f) & (x < %.12f)'
               % (0.99999 * sx, 1.00001 * sx),
               'facet'),
})
if dim == 3:
    regions.update({
        'Sensor' : ('r.Plane *v r.Top *v r.Far', 'vertex'),
    })
else:
    regions.update({
        'Sensor' : ('r.Plane *v r.Top', 'vertex'),
    })

# Iron.
materials = {
    'm' : ({
        'young' : young,
        'poisson' : poisson,
        'density' : density,
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i' : 2,
}

# Notes:
# 1. The order of the variables in the solution vector is specified here
#    (3rd tuple member), since that specific order is expected by the
#    elastodynamic time-stepping solvers.
# 2. For the same reason, we won't explicitly define below the equations
#    du = du/dt and ddu = ddu/dt - these are implicitly defined by
#    the time-stepping solver. see the `step()` method of the solvers.
variables = {
    'u' : ('unknown field', 'displacement', 0),

```

(continues on next page)

(continued from previous page)

```

    'du' : ('unknown field', 'displacement', 1),
    'ddu' : ('unknown field', 'displacement', 2),
    'v' : ('test field', 'displacement', 'u'),
    'dv' : ('test field', 'displacement', 'du'),
    'ddv' : ('test field', 'displacement', 'ddu'),
}
# The mapping of variables for the elastodynamics solvers - keys are given,
# values correspond to the names of the actual variables.
var_names = {'u' : 'u', 'du' : 'du', 'ddu' : 'ddu'}

ebcs = {
    'Impact' : ('Left', {'u.0' : 0.0, 'du.0' : 0.0, 'ddu.0' : 0.0}),
}
if dim == 3:
    ebcs.update({
        'Symmetry-y' : ('Near',
                        {'u.1' : 0.0, 'du.1' : 0.0, 'ddu.1' : 0.0}),
        'Symmetry-z' : ('Bottom',
                        {'u.2' : 0.0, 'du.2' : 0.0, 'ddu.2' : 0.0}),
    })

def get_ic(coor, ic, mode='u'):
    val = nm.zeros_like(coor)
    if mode == 'u':
        val[:, 0] = 0.0

    elif mode == 'du':
        val[:, 0] = -1.0

    return val

functions = {
    'get_ic_u' : (get_ic,),
    'get_ic_du' : (lambda coor, ic: get_ic(coor, None, mode='du'),),
}

ics = {
    'ic' : ('Omega', {'u.all' : 'get_ic_u', 'du.all' : 'get_ic_du'}),
}

equations = {
    'eq' :
    f"""
    + dw_mass_ad.i.Omega(m.density, ddv, ddu)
    + {alpha} * dw_mass_ad.i.Omega(m.density, dv, du)
    + {beta} * dw_lin_elastic_yp_ad.i.Omega(m.young, m.poisson, dv, du)
    + dw_lin_elastic_yp_ad.i.Omega(m.young, m.poisson, v, u) = 0""",
}

solvers = {
    'lsd' : ('ls.auto_direct', {
    # 'lsd' : ('ls.scipy_direct', {

```

(continues on next page)

(continued from previous page)

```

        # Reuse the factorized linear system from the first time step.
        'use_presolve' : True,
        # Speed up the above by omitting the matrix digest check used
        # normally for verification that the current matrix corresponds to
        # the factorized matrix stored in the solver instance. Use with
        # care!
        'use_mtx_digest' : False,
        # Increase when getting MUMPS error -9.
        'memory_relaxation' : 50,
    }},
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-6,
        'eps_r'      : 1e-6,
    }},
    'tsn' : ('ts.newmark', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'beta' : 0.25,
        'gamma' : 0.5,

        'var_names' : var_names,
        'verbose' : 1,
    }},
    },

    options = {
        'ts' : 'tsn',
        'tsc' : None,
        'nls' : 'newton',
        'ls' : 'lsd',

        'save_times' : save_times,

        'active_only' : active_only,
        'auto_transform_equations' : False,

        'output_format' : 'h5',
        'output_dir' : output_dir,
    }

    return locals()

from sfepy.solvers.ts_solvers import NewmarkTS, _cache

class NewmarkSATS(NewmarkTS):
    r"""

```

(continues on next page)

(continued from previous page)

Solve elastodynamics problems by the Newmark method.

The method was introduced in [1]. Common settings [2]:

```
=====
name          kind      beta gamma Omega_crit
=====
trapezoidal rule:  implicit 1/4  1/2  unconditional
linear acceleration: implicit 1/6  1/2  :math:`2\sqrt{3}`
Fox-Goodwin:      implicit 1/12 1/2  :math:`\sqrt{6}`
central difference: explicit 0   1/2  2
=====
```

All of these methods are 2-order of accuracy.

[1] Newmark, N. M. (1959) A method of computation for structural dynamics. Journal of Engineering Mechanics, ASCE, 85 (EM3) 67-94.

[2] Arnaud Delaplace, David Ryckelynck: Solvers for Computational Mechanics
"""

```
name = 'ts.newmark_sa'
```

```
def create_nlst(self, nls, dt, gamma, beta, u0, e0, v0, a0, pack, unpack):
    dt2 = dt**2
    iue, iu, ie, iv, ia = pack.indices

    if iue != iu:
        raise ValueError('extra variables not supported!')

    cc0 = (1.0 - gamma) * dt
    cc = gamma * dt
    ck0 = (0.5 - beta) * dt2
    ck = beta * dt2

    def v(a):
        return v0 + cc0 * a0 + cc * a

    def u(a):
        return u0 + dt * v0 + ck0 * a0 + ck * a

    def fun(at):
        vec = nm.r_[u(at), v(at), at]

        aux = nls.fun(vec)

        rt = aux[iu] + aux[iv] + aux[ia]
        return rt

    @_cache(self, 'matrix', self.conf.is_linear)
    def fun_grad(at):
        vec = None if self.conf.is_linear else nm.r_[u(at), v(at), at]
        M, C, K = self.get_matrices(nls, vec, unpack)
```

(continues on next page)

(continued from previous page)

```

Kt = M + cc * C + ck * K
return Kt

@_cache(self, 'matrix_sa', self.conf.is_linear and (self.ts.step > 0))
def efun_grad(at):
    vec = None if self.conf.is_linear else nm.r_[u(at), v(at), at]
    M, C, K = self.get_matrices(nls, vec, unpack)

    size = at.shape[0]
    I = sps.eye(size, dtype=nm.float64)
    zz = sps.dia_matrix((size, size), dtype=nm.float64)
    if self.ts.step > 0:
        duda = -ck * I
        dvda = -cc * I

    else:
        duda = dvda = zz
        # The initial state might depend on parameters - it does not in
        # this example as u0 = 0, v0 = const and a0 = 0, but the
        # following code should work in the general case if the next
        # line is commented out.
        K = C = zz

    return sps.bmat([[K, C, M],
                    [I, zz, duda],
                    [zz, I, dvda]])

@_cache(self, 'matrix_sa0', self.conf.is_linear and (self.ts.step > 0))
def efun_grad0(at):
    if self.ts.step > 0:
        size = at.shape[0]
        I = sps.eye(size, dtype=nm.float64)
        zz = sps.dia_matrix((size, size), dtype=nm.float64)
        duda0 = -ck0 * I
        dudv0 = -dt * I
        dudu0 = -I
        dvda0 = -cc0 * I
        dvdv0 = -I
        return sps.bmat([[zz, zz, zz],
                        [dudu0, dudv0, duda0],
                        [zz, dvdv0, dvda0]])

    else:
        return sps.csr_matrix((self.sa_info.n_dof, self.sa_info.n_dof),
                               dtype=nm.float64)

def efun_grad_par(at):
    mtx = nm.zeros((self.sa_info.n_dof, self.sa_info.n_par),
                   dtype=nm.float64, order='F')
    mtx = self.context.equations.evaluate(
        mode='weak', dw_mode='sensitivity',

```

(continues on next page)

(continued from previous page)

```

        diff_vars=self.sa_info.par_names, asm_obj=mtx,
    )
    for ic, name in enumerate(self.sa_info.par_names):
        if name == 'alpha': # =  $M v$ 
            info = self.sa_info.par_info['alpha']

            elif name == 'beta': # =  $K v$ 
                info = self.sa_info.par_info['beta']

            else:
                continue

        term = self.context.equations[info[1]].terms[info[2]]
        val, iels, status = term.evaluate(mode='weak',
                                         diff_var=None,
                                         standalone=False,
                                         ret_status=True)

        val /= term.sign
        term.assemble_to(mtx[:, ic], val, iels)

    Vt = mtx[iu] + mtx[iv] + mtx[ia]
    zz = nm.zeros_like(Vt, dtype=nm.float64)
    return nm.block([[Vt],
                    [zz],
                    [zz]])

    nlst = nls.copy()
    nlst.fun = fun
    nlst.fun_grad = fun_grad
    nlst.efun_grad = efun_grad
    nlst.efun_grad0 = efun_grad0
    nlst.efun_grad_par = efun_grad_par
    nlst.u = u
    nlst.v = v

    self.nlst = nlst

    return nlst

register_solver(NewmarkSATS)

def apply_sensor(pb, ts, state, out):
    us = state['u'].get_state_in_region(pb.domain.regions['Sensor'])
    output('sensor', ts.step, us)
    out.append(us)

def update_pars(materials, equations, pars, par_names, par_info):
    """
    Materials and options are updated in place.
    """
    done = set()
    for key in par_names:

```

(continues on next page)

(continued from previous page)

```

    if key in done: continue

    info = par_info[key]
    ip = par_names.index(key)
    if info[0] == 'term': # Update term coefficient.
        equations[info[1]].terms[info[2]].sign = pars[ip]

    else: # Update materials.
        mat = materials[info[0]]
        mkeys = mat.get_keys(info[1])
        for mkey in mkeys:
            if key == info[2]:
                val = pars[ip]

            else:
                raise ValueError

        mat.datas[mkey][info[2]][:, :] = val

    output('updated:', par_names)

def eval_fun(pars, data, pb, options, par_names, par_info, opt_data, plog,
             inodir):
    opt_data.tfun.start()

    materials = pb.get_materials()
    update_pars(materials, pb.equations, pars, par_names, par_info)

    pb.ts.set_step() # Reset ts.
    pb.get_solver().clear_lin_solver() # No digest -> clear manually.

    out = []
    pb.solve(save_results=False, step_hook=partial(apply_sensor, out=out))
    usens = nm.concatenate(out)

    sim_data = usens[:, -1]
    res = sim_data - data

    plog(*(tuple(pars) + (nm.linalg.norm(res),)))

    times = pb.ts.times
    fig, ax = plt.subplots()
    ax.plot(times, sim_data, label='sim')
    ax.plot(times, data, ls='', marker='o', mew=0, label='exp')
    ax.plot(times, opt_data.true_data, ls=':', label='true')
    ax.legend()
    plt.tight_layout()
    fig.savefig(inodir(f'res{opt_data.n_fun:05d}.png'), bbox_inches='tight')

    opt_data.n_fun += 1
    opt_data.usens = usens

```

(continues on next page)

(continued from previous page)

```

    dt = opt_data.tfun.stop()
    output(f'function evaluation done in {dt} s.')

    return res

def eval_jac_fd(pars, *args, **kwargs):
    v0 = eval_fun(pars, *args, **kwargs)

    eps = 1e-6
    jac = []
    for ip, par in enumerate(pars):
        pars1 = pars.copy()
        dp = eps * par
        pars1[ip] += dp

        v1 = eval_fun(pars1, *args, **kwargs)
        jac.append((v1 - v0) / dp)

    jac = nm.array(jac).T

    return jac

def eval_jac(pars, data, pb, options, par_names, par_info, opt_data, plog,
            inodir):
    opt_data.tjac.start()

    materials = pb.get_materials()
    update_pars(materials, pb.equations, pars, par_names, par_info)

    pb.ts.set_step() # Reset ts.
    pb.get_solver().clear_lin_solver() # No digest -> clear manually.

    variables = pb.get_initial_state()
    n_dof = variables.adi.n_dof_total
    n_par = len(par_names)

    _tss = pb.get_solver()
    conf = _tss.conf.copy(name='tss_sa')
    conf.kind = 'ts.newmark_sa'

    tss = Solver.any_from_conf(conf, nls=_tss.nls, tsc=_tss.tsc, context=pb)
    tss.sa_info = Struct(
        par_info=par_info, par_names=par_names, n_par=n_par, n_dof=n_dof,
    )
    tss.matrix_sa = tss.matrix_sa0 = tss.matrix_pars = None

    ls = Solver.any_from_conf(tss.nls.lin_solver.conf)

    _init_fun, prestep_fun, _poststep_fun = pb.get_tss_functions(
        update_bcs=True, update_materials=True,
        save_results=False,
        step_hook=None, post_process_hook=None,

```

(continues on next page)

(continued from previous page)

```

)

fu = pb.fields['displacement']
ii = fu.get_dofs_in_region(pb.domain.regions['Sensor'])[0]
idof = int(fu.n_components * ii + 2 + variables.adi.indx['u'].start)

def init_fun(tss, ts, vec0):
    vec0 = _init_fun(ts, vec0)
    # Previous step dy/da.
    tss.gy0 = nm.zeros((tss.sa_info.n_dof, tss.sa_info.n_par),
                       dtype=nm.float64, order='F')
    return vec0

jac = []
def poststep_fun(tss, ts, vec):
    # EBCs need to be applied here because of algebraically computed
    # variables in elastodynamics solvers.
    variables.set_state(vec, pb.active_only, apply_ebc=True)

    unpack = tss.unpack
    utp, etp, vtp, atp = unpack(vec)
    if ts.step == 0:
        nlst = tss.create_nlst(tss.nls, ts.dt, tss.conf.gamma, tss.conf.beta,
                              utp, etp, vtp, atp, tss.pack, unpack)

    else:
        nlst = tss.nlst

    Atp = nlst.efun_grad(atp)
    Wtp = nlst.efun_grad0(atp)
    Vtp = nlst.efun_grad_par(atp)

    ry = -(Vtp + Wtp @ tss.gy0)

    if options.multi_rhs:
        gy = ls(ry, mtx=Atp)

    else:
        gy = nm.empty_like(ry)
        for ic in range(gy.shape[1]):
            gy[:, ic] = ls(ry[:, ic], mtx=Atp)

    if ts.step == 0:
        # No digest -> clear manually.
        ls.clear()

    jac.append(gy[idof])

    tss.gy0 = gy
    pb.advance(ts)

    return vec

```

(continues on next page)

(continued from previous page)

```

tss.set_dof_info(variables.adi)
status = IndexedStruct()
tss(variables.get_state(pb.active_only, force=True),
     init_fun=init_fun,
     prestep_fun=prestep_fun,
     poststep_fun=poststep_fun,
     status=status)

jac = nm.array(jac)

if options.check_jac:
    jac_fd = eval_jac_fd(pars, data, pb, options, par_names, par_info,
                        opt_data, plog, inodir)
    output('Jacobian:')
    output(jac)
    output('Jacobian using finite differences:')
    output(jac_fd)
    output('Relative difference:')
    output((jac_fd - jac) / nm.where(jac_fd, jac_fd, 1.0))

    opt_data.jac = jac.copy()
    opt_data.jac_fd = jac_fd

opt_data.n_jac += 1

dt = opt_data.tjac.stop()
output(f'jacobian evaluation done in {dt} s.')

return jac

def parse_args(args=None):
    helps = {}

    sdefault_opt_conf = """ftol=1e-08, xtol=1e-4, gtol=None, method='trf',
                          f_scale=3e-8, loss='soft_l1', verbose=2
                          """

    opts = dict(
        par_names = ('density,young,poisson', 'parameters to be identified'),
        opt_conf = (sdefault_opt_conf, 'optimization solver options'),
        jac = (True, 'do not use the semi-analytical Jacobian calculation'),
        check_jac = (False,
                     'check the Jacobian using finite differences'),
        multi_rhs = (False, 'solve all rhs of sensitivity analysis in one call'),
        young = (200e9, """Young's modulus"""),
        poisson = (0.3, """Poisson's ratio"""),
        density = (7800.0, 'density'),
        alpha = (0.0, 'proportional damping coefficient (M)'),
        beta = (0.0, 'proportional damping coefficient (K)'),
        dims = ('1e-2,2.5e-3,2.5e-3',
                'physical dimensions of the block (L, d, x)'),
        shape = ('21,6,6',

```

(continues on next page)

(continued from previous page)

```

        'numbers of mesh vertices along each axis'),
    v0 = (1.0, 'initial impact velocity'),
    ct1 = (1.5, 'final time in L / "longitudinal wave speed" units'),
    dt = (None, 'time step (None means automatic)'),
    active_only = (False,
        'vectors and matrices contain only active DOFs'),
    save_times = (20, 'number of time steps to save'),
    output_dir = ('output/edi', 'output directory'),
    plot_log = (False, 'show live log figures'),
    shell = (False, 'run ipython shell after all computations'),
    debug = (False,
        'automatically start debugger when an exception is raised'),
)

parser = ArgumentParser(description=__doc__.rstrip(),
                        formatter_class=RawDescriptionHelpFormatter)

dhelp = ' [default: %(default)s]'
for key, (val, msg) in opts.items():
    helps[key] = msg
    action = 'store'
    vtype = type(val)
    choices = None
    option = key
    if val is True:
        action = 'store_false'
        option = 'no_' + key

    elif val is False:
        action = 'store_true'

    elif isinstance(val, tuple):
        choices = val
        vtype = type(val[0])
        val = val[0]

    elif isinstance(val, list):
        vtype = type(val[1])
        val = val[0]

    if action == 'store':
        helps[key] += dhelp
        parser.add_argument('--' + option.replace('_', '-'),
                            type=vtype,
                            action=action, dest=key, choices=choices,
                            default=val, help=helps[key])
    else:
        parser.add_argument('--' + option.replace('_', '-'),
                            action=action, dest=key,
                            default=val, help=helps[key])
options = parser.parse_args(args=args)

```

(continues on next page)

(continued from previous page)

```

options.dims = [float(ii) for ii in options.dims.split(',')]
options.shape = [int(ii) for ii in options.shape.split(',')]
options.par_names = [ii.strip() for ii in options.par_names.split(',')]
aux = parse_as_dict(sdefault_opt_conf)
aux.update(parse_as_dict(options.opt_conf))
options.opt_conf = aux

options.plane = 'strain'
options.opt_bounds = {
    'density' : [0.5 * options.density, 1.5 * options.density],
    'young' : [0.5 * options.young, 1.5 * options.young],
    'poisson' : [0.9 * options.poisson, 1.1 * options.poisson],
    'alpha' : [0, 1e4],
    'beta' : [0, 1e-4],
}

return options, helps

def main():
    from scipy.optimize import least_squares

    from sfepy.base.base import output, Struct
    from sfepy.base.conf import ProblemConf
    from sfepy.discrete import Problem
    from sfepy.base.log import Log

    options, helps = parse_args()
    if options.debug:
        from sfepy.base.base import debug_on_error; debug_on_error()

    conf = ProblemConf.from_dict(define(**vars(options)), sys.modules[__name__])

    pb = Problem.from_conf(conf)

    out = []
    pb.solve(save_results=False, step_hook=partial(apply_sensor, out=out))
    usens = nm.concatenate(out)

    size = nm.abs(usens[:, -1]).max()
    true_data = usens[:, -1]
    data = true_data.copy()
    i0 = int(0.2 * options.shape[0])
    data[i0:] += nm.random.default_rng(seed=42).normal(
        0, 0.1 * size, len(data[i0:]),
    )
    # Add an outlier.
    data[max(len(data) // 2, i0)] += 0.5 * size

    par_info = {
        'density' : ('m', 'Omega', 'density', options.density),
        'young' : ('m', 'Omega', 'young', options.young * 0.8),
        'poisson' : ('m', 'Omega', 'poisson', options.poisson),

```

(continues on next page)

(continued from previous page)

```

    'alpha' : ('term', 'eq', 1, 1000),
    'beta' : ('term', 'eq', 2, 1e-5),
}
par_names = options.par_names
all_par_names = set(par_info.keys())
if not set(par_names).issubset(all_par_names):
    unknown = set(par_names).difference(all_par_names)
    raise ValueError(f'parameters {unknown} not in {all_par_names}!')

labels = [[name] for name in par_names] + [['|res|']]
nlog = len(labels)
formats = [['{: .16e}' for ii in range(len(labs))] for labs in labels]
inodir = partial(op.join, options.output_dir)
plog = Log(labels,
            formats=formats,
            yscale=['log'] * nlog,
            xlabel=['iter'] * nlog,
            ylabel=None,
            show_legends=True,
            is_plot=options.plot_log,
            log_filename=inodir('pars.txt'),
            aggregate=1000, sleep=0.1)

pars0 = [par_info[name][-1] for name in par_names]
pars0 = nm.array(pars0, dtype=nm.float64)
x_scale = [par if abs(par) > 0 else 1.0 for par in pars0]
bounds = [options.opt_bounds.get(name, (-nm.inf, nm.inf))
           for name in par_names]
bounds = tuple(zip(*bounds))
opt_data = Struct(name='opt_data', true_data=true_data, n_fun=0, n_jac=0,
                  tfun=Timer('fun'), tjac=Timer('jac'))

timer = Timer(start=True)
res = least_squares(
    eval_fun, pars0,
    jac=eval_jac if options.jac else '2-point',
    bounds=bounds,
    x_scale=x_scale,
    args=(data, pb, options, par_names, par_info, opt_data, plog, inodir),
    **options.opt_conf
)
elapsed = timer.stop()
output(f'identification done in {elapsed} s.')
output(res)
od = opt_data
tfi = od.tfun.total / od.n_fun
tji = od.tjac.total / max(od.n_jac, 1)
tfe = tfi * len(pars0)
output(f'evaluations: function: {od.n_fun}, jacobian: {od.n_jac}')
output(f'elapsed: function: {od.tfun.total}, jacobian {od.tjac.total}')
output(f'elapsed/it: function: {tfi}, jacobian {tji}')
output(f'expected finite difference jacobian time: {tfe}')

```

(continues on next page)

(continued from previous page)

```
plog(save_figure=inodir('pars.png'))

if options.shell:
    from sfepy.base.base import shell; shell()

plog(finished=True)

if __name__ == '__main__':
    main()
```

linear_elasticity/its2D_1.py

Description

Diametrically point loaded 2-D disk. See *Primer*.

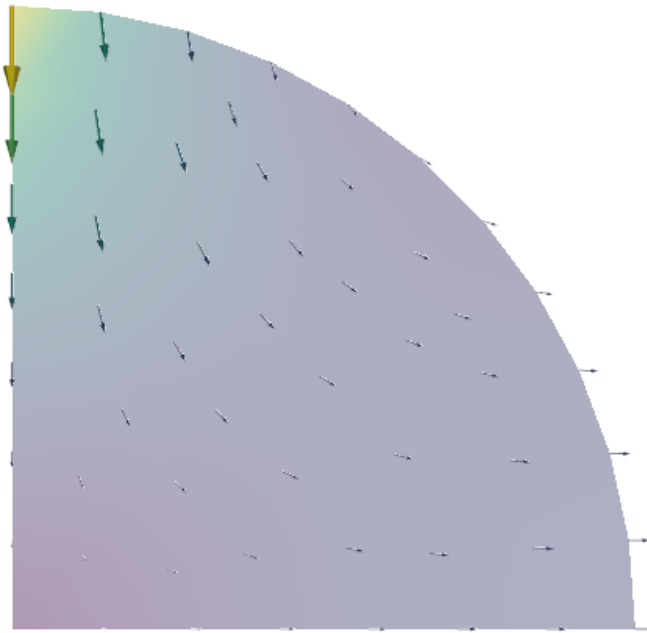
Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

0.00 u 1.65



source code

```

r"""
Diametrically point loaded 2-D disk. See :ref:`sec-primer`.

Find  $u$  such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(v) e_{kl}(u)
    = 0
    \quad \forall v,
where
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad .
"""
from __future__ import absolute_import
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.utils import refine_mesh
from sfepy import data_dir

```

(continues on next page)

(continued from previous page)

```

# Fix the mesh file name if you run this file outside the SfePy directory.
filename_mesh = data_dir + '/meshes/2d/its2D.mesh'

refinement_level = 0
filename_mesh = refine_mesh(filename_mesh, refinement_level)

output_dir = '.' # set this to a valid directory you have write access to

young = 20000.0 # Young's modulus [MPa]
poisson = 0.4 # Poisson's ratio

options = {
    'output_dir' : output_dir,
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Bottom' : ('vertices in (y < 0.001)', 'facet'),
    'Top' : ('vertex 2', 'vertex'),
}

materials = {
    'Asphalt' : ({'D': stiffness_from_youngpoisson(2, young, poisson)}),
    'Load' : ({'.val' : [0.0, -1000.0]}),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.2.Omega(Asphalt.D, v, u)
        = dw_point_load.0.Top(Load.val, v)""",
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'XSym' : ('Bottom', {'u.1' : 0.0}),
    'YSym' : ('Left', {'u.0' : 0.0}),
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,

```

(continues on next page)

(continued from previous page)

```

    'eps_a' : 1e-6,
  }},
}

```

linear_elasticity/its2D_2.py

Description

Diametrically point loaded 2-D disk with postprocessing. See *Primer*.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```
r"""
Diametrically point loaded 2-D disk with postprocessing. See
:ref:`sec-primer`.

Find :math:\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ul{v} e_{ij} \ul{v} e_{kl} \ul{u}
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""

from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg',
               copy_materials=False)

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                 data=strain, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                 data=stress, dofs=None)

    return out

asphalt = materials['Asphalt'][0]
asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
options.update({'post_process_hook' : 'stress_strain',})
```

linear_elasticity/its2D_3.py**Description**

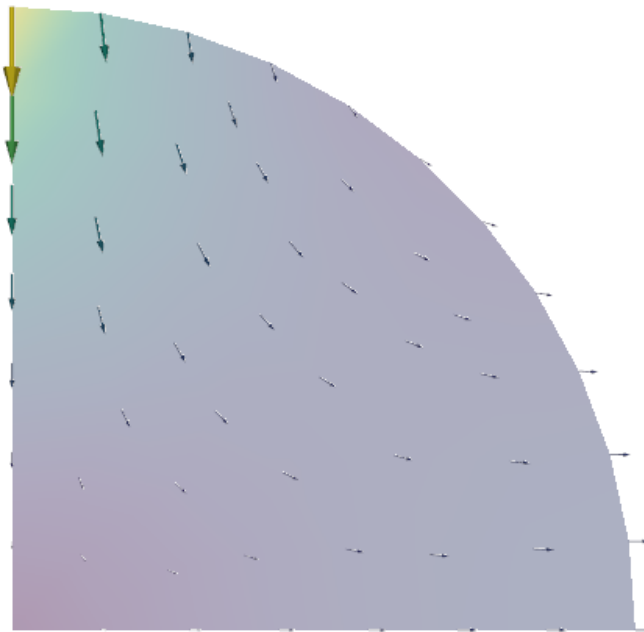
Diametrically point loaded 2-D disk with nodal stress calculation. See [Primer](#).

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```

r"""
Diametrically point loaded 2-D disk with nodal stress calculation. See
:ref:`sec-primer`.

Find :math:`\underline{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})

```

(continues on next page)

(continued from previous page)

```

    = 0
    \;, \quad \forall v \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""
from __future__ import print_function
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.geometry_element import geometry_data
from sfepy.discrete import FieldVariable
from sfepy.discrete.fem import Field
import numpy as nm

gdata = geometry_data['2_3']
nc = len(gdata.coors)

def nodal_stress(out, pb, state, extend=False, integrals=None):
    """
    Calculate stresses at nodal points.
    """

    # Point load.
    mat = pb.get_materials()['Load']
    P = 2.0 * mat.get_data('special', 'val')[1]

    # Calculate nodal stress.
    pb.time_update()

    if integrals is None: integrals = pb.get_integrals()

    stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D, u)', mode='qp',
                        integrals=integrals, copy_materials=False)
    sfield = Field.from_args('stress', nm.float64, (3,),
                          pb.domain.regions['Omega'])
    svar = FieldVariable('sigma', 'parameter', sfield,
                        primary_var_name='(set-to-None)')
    svar.set_from_qp(stress, integrals['ivn'])

    print('\n=====')
    print('Given load = %.2f N' % -P)
    print('\nAnalytical solution')
    print('=====')
    print('Horizontal tensile stress = %.5e MPa/mm' % (-2.*P/(nm.pi*150.)))
    print('Vertical compressive stress = %.5e MPa/mm' % (-6.*P/(nm.pi*150.)))
    print('\nFEM solution')

```

(continues on next page)

(continued from previous page)

```

print('=====')
print('Horizontal tensile stress = %.5e MPa/mm' % (svar()[0]))
print('Vertical compressive stress = %.5e MPa/mm' % (-svar()[1]))
print('=====')
return out

asphalt = materials['Asphalt'][0]
asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
options.update({'post_process_hook' : 'nodal_stress',})

integrals = {
    'ivn' : ('custom', gdata.coors, [gdata.volume / nc] * nc),
}

```

linear_elasticity/its2D_4.py

Description

Diametrically point loaded 2-D disk with postprocessing and probes. See *Primer*.

1. solve the problem:

```
sfepy-run sfepy/examples/linear_elasticity/its2D_4.py
```

2. optionally, view the results:

```
sfepy-view its2D.h5 -2
```

3. optionally, convert results to VTK, and view again ((assumes running from the sfepy directory):

```
python3 sfepy/scripts/extractor.py -d its2D.h5 sfepy-view its2D.0.vtk -2
```

4. probe the data:

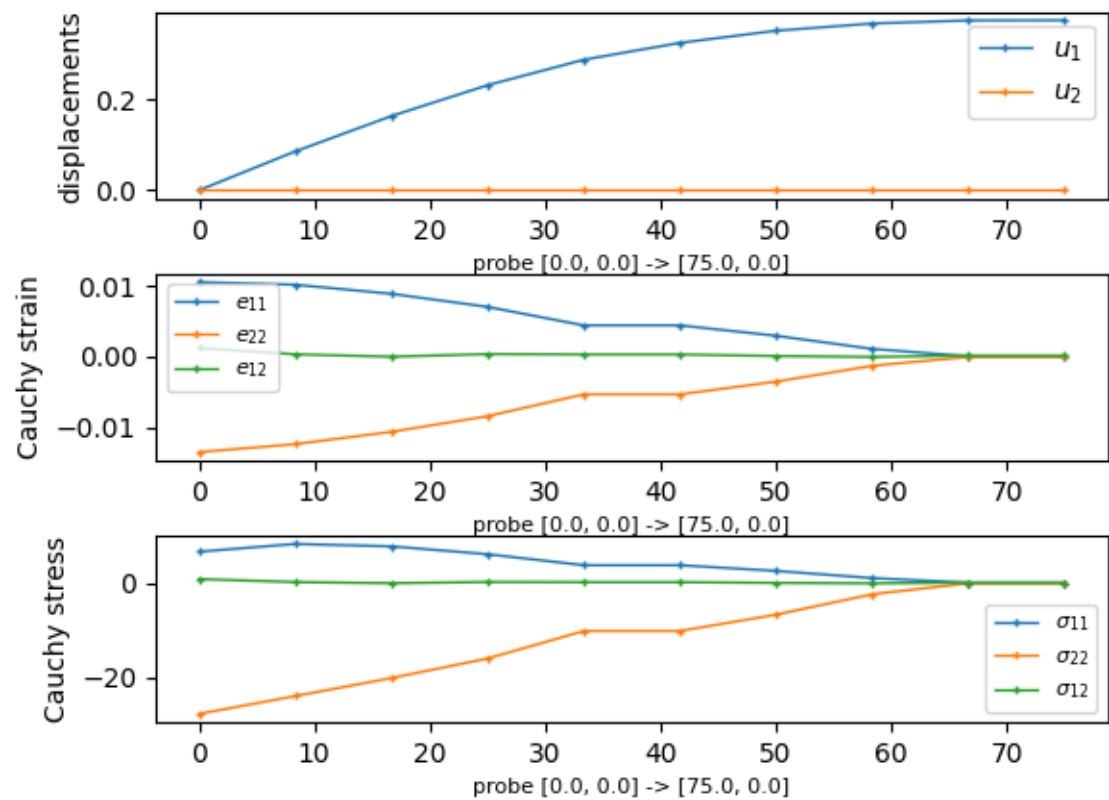
```
sfepy-probe sfepy/examples/linear_elasticity/its2D_4.py its2D.h5
```

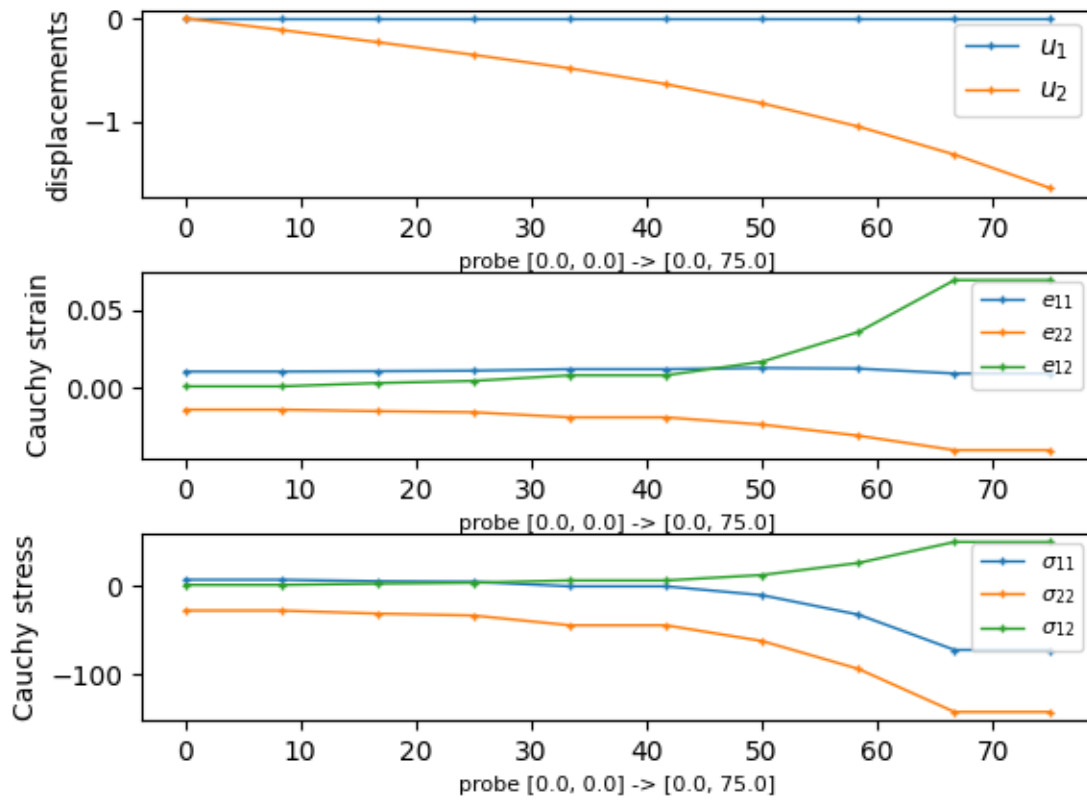
Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$







source code

```
r"""
Diametrically point loaded 2-D disk with postprocessing and probes. See
:ref:`sec-primer`.

1. solve the problem::

    sfepy-run sfepy/examples/linear_elasticity/its2D_4.py

2. optionally, view the results::

    sfepy-view its2D.h5 -2

3. optionally, convert results to VTK, and view again ((assumes running from
the sfepy directory)::

    python3 sfepy/scripts/extractor.py -d its2D.h5
    sfepy-view its2D.0.vtk -2

4. probe the data::

    sfepy-probe sfepy/examples/linear_elasticity/its2D_4.py its2D.h5
```

(continues on next page)

(continued from previous page)

```

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;; \quad \forall \ul{v} \;;

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;;
    """

from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from six.moves import range

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg')

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                  data=strain, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                   data=stress, dofs=None)

    return out

def gen_lines(problem):
    from sfepy.discrete.probes import LineProbe
    ps0 = [[0.0, 0.0], [0.0, 0.0]]
    ps1 = [[75.0, 0.0], [0.0, 75.0]]

    # Use adaptive probe with 10 initial points.
    n_point = 10

    labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
    probes = []
    for ip in range(len(ps0)):
        p0, p1 = ps0[ip], ps1[ip]
        probes.append(LineProbe(p0, p1, n_point))

```

(continues on next page)

(continued from previous page)

```

    return probes, labels

def probe_hook(data, probe, label, problem):
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm

    def get_it(name, var_name):
        var = problem.create_variables([var_name])[var_name]
        var.set_data(data[name].data)

        pars, vals = probe(var)
        vals = vals.squeeze()
        return pars, vals

    results = {}
    results['u'] = get_it('u', 'u')
    results['cauchy_strain'] = get_it('cauchy_strain', 's')
    results['cauchy_stress'] = get_it('cauchy_stress', 's')

    fig = plt.figure()
    plt.clf()
    fig.subplots_adjust(hspace=0.4)
    plt.subplot(311)
    pars, vals = results['u']
    for ic in range(vals.shape[1]):
        plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('displacements')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=10))

    sym_indices = ['11', '22', '12']

    plt.subplot(312)
    pars, vals = results['cauchy_strain']
    for ic in range(vals.shape[1]):
        plt.plot(pars, vals[:,ic], label=r'$\epsilon_{%s}$' % sym_indices[ic],
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('Cauchy strain')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=8))

    plt.subplot(313)
    pars, vals = results['cauchy_stress']
    for ic in range(vals.shape[1]):
        plt.plot(pars, vals[:,ic], label=r'$\sigma_{%s}$' % sym_indices[ic],
                 lw=1, ls='-', marker='+', ms=3)
    plt.ylabel('Cauchy stress')
    plt.xlabel('probe %s' % label, fontsize=8)
    plt.legend(loc='best', prop=fm.FontProperties(size=8))

```

(continues on next page)

(continued from previous page)

```

    return plt.gcf(), results

materials['Asphalt'][0].update({'D' : stiffness_from_youngpoisson(2, young, poisson)})

# Update fields and variables to be able to use probes for tensors.
fields.update({
    'sym_tensor': ('real', 3, 'Omega', 0),
})

variables.update({
    's' : ('parameter field', 'sym_tensor', None),
})

options.update({
    'output_format'      : 'h5', # VTK reader cannot read cell data yet for probing
    'post_process_hook'  : 'stress_strain',
    'gen_probes'         : 'gen_lines',
    'probe_hook'         : 'probe_hook',
})

```

linear_elasticity/its2D_5.py

Description

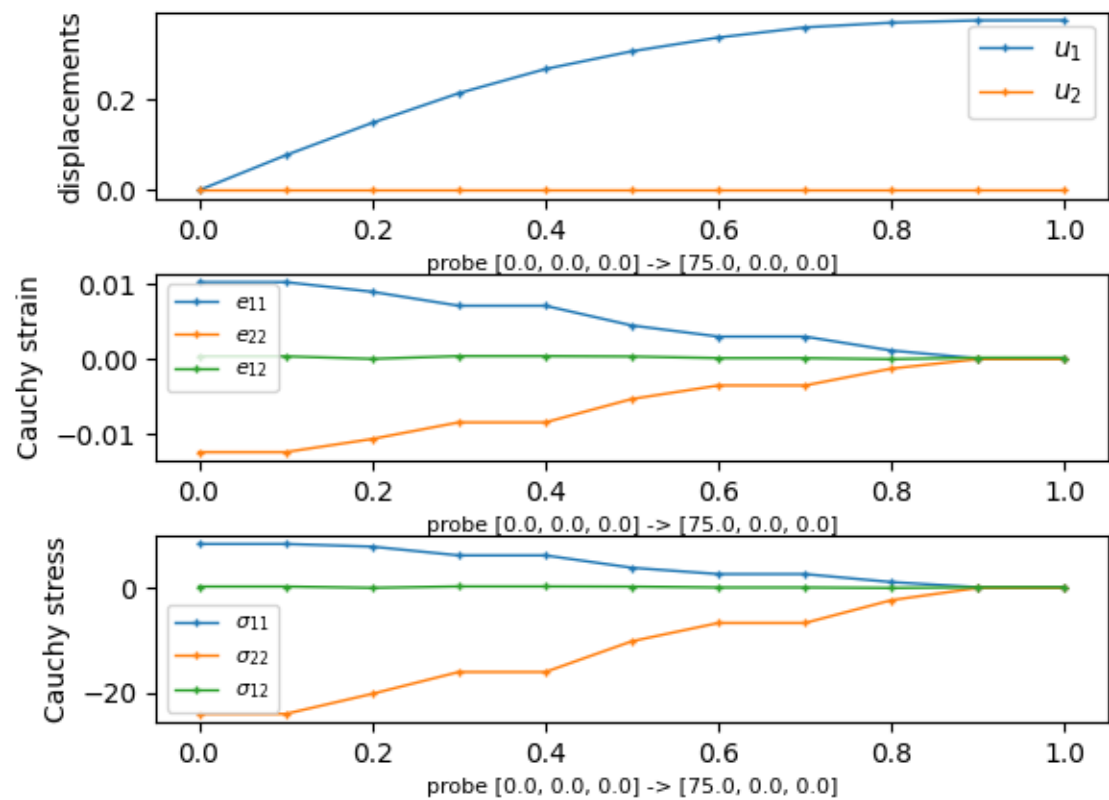
Diametrically point loaded 2-D disk with postprocessing and probes. See [Primer](#).

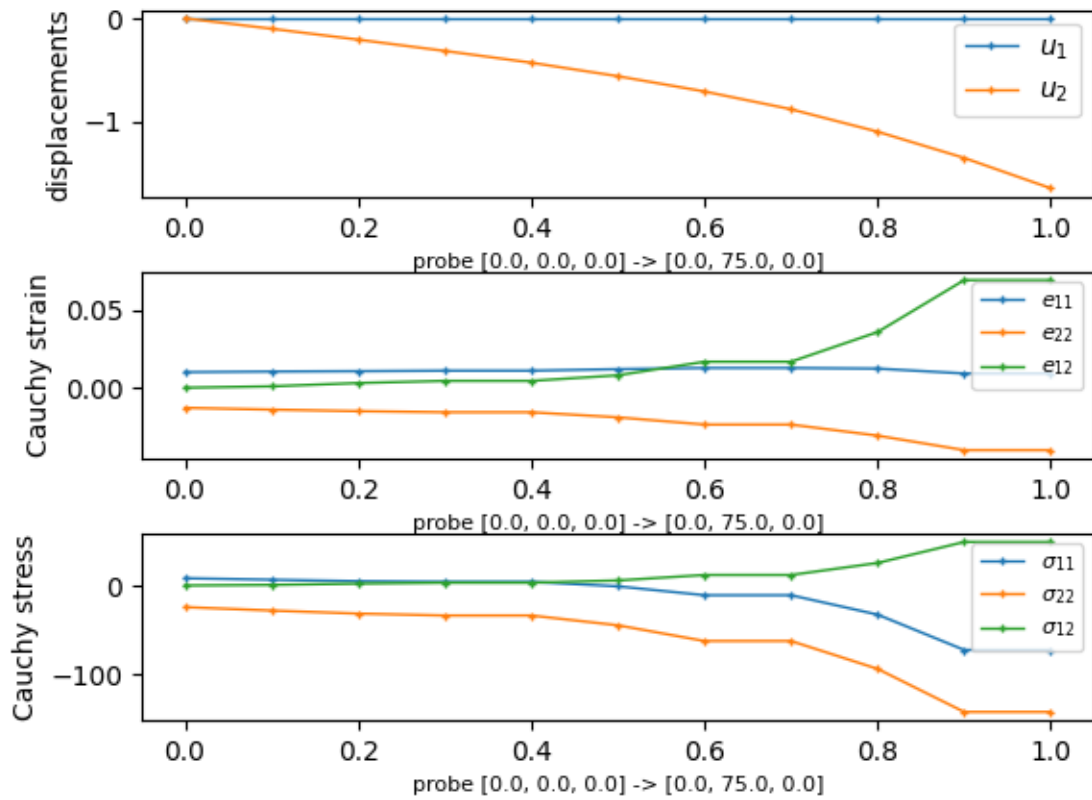
Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$







source code

```

r"""
Diametrically point loaded 2-D disk with postprocessing and probes. See
:ref:`sec-primer`.

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ul{e}_{ij}(\ul{v}) \ul{e}_{kl}(\ul{u})
    = 0
    \quad \forall \ul{v};

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad ;
"""
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

```

(continues on next page)

(continued from previous page)

```

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.postprocess.probes_vtk import Probe

import os
from six.moves import range

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg')

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                  data=strain, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                  data=stress, dofs=None)

    probe = Probe(out, pb.domain.mesh, probe_view=True)

    ps0 = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
    ps1 = [[75.0, 0.0, 0.0], [0.0, 75.0, 0.0]]
    n_point = 10

    labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
    probes = []
    for ip in range(len(ps0)):
        p0, p1 = ps0[ip], ps1[ip]
        probes.append('line%d' % ip)
        probe.add_line_probe('line%d' % ip, p0, p1, n_point)

    for ip, label in zip(probes, labels):
        fig = plt.figure()
        plt.clf()
        fig.subplots_adjust(hspace=0.4)
        plt.subplot(311)
        pars, vals = probe(ip, 'u')
        for ic in range(vals.shape[1] - 1):
            plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
                     lw=1, ls='-', marker='+', ms=3)
        plt.ylabel('displacements')
        plt.xlabel('probe %s' % label, fontsize=8)
        plt.legend(loc='best', prop=fm.FontProperties(size=10))

        sym_labels = ['11', '22', '12']

        plt.subplot(312)

```

(continues on next page)

(continued from previous page)

```

pars, vals = probe(ip, 'cauchy_strain')
for ii in range(vals.shape[1]):
    plt.plot(pars, vals[:, ii], label=r'$\epsilon_{%s}$' % sym_labels[ii],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy strain')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', prop=fm.FontProperties(size=8))

plt.subplot(313)
pars, vals = probe(ip, 'cauchy_stress')
for ii in range(vals.shape[1]):
    plt.plot(pars, vals[:, ii], label=r'$\sigma_{%s}$' % sym_labels[ii],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy stress')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', prop=fm.FontProperties(size=8))

opts = pb.conf.options
filename_results = os.path.join(opts.get('output_dir'),
                                'its2D_probe_%s.png' % ip)

fig.savefig(filename_results)

return out

materials['Asphalt'][0].update({'D' : stiffness_from_youngpoisson(2, young, poisson)})

options.update({
    'post_process_hook' : 'stress_strain',
})

```

linear_elasticity/its2D_interactive.py

Description

Diametrically point loaded 2-D disk, using commands for interactive use. See *Primer*.

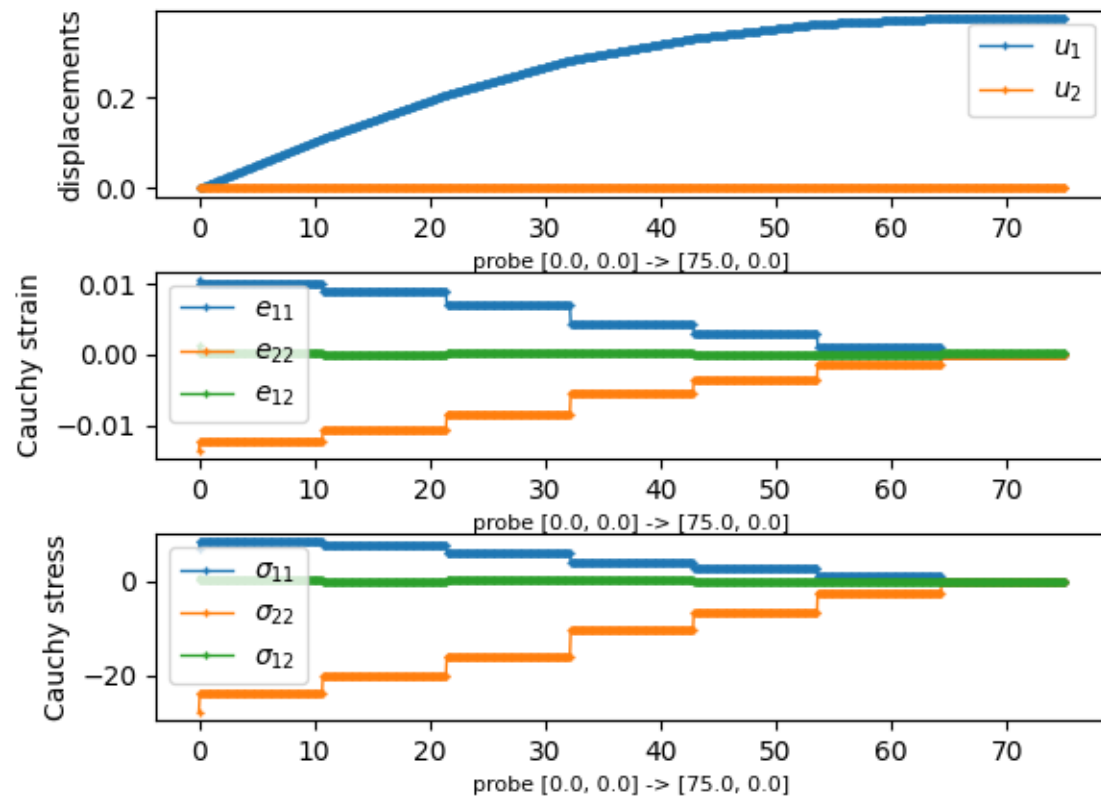
The script combines the functionality of all the `its2D_?.py` examples and allows setting various simulation parameters, namely:

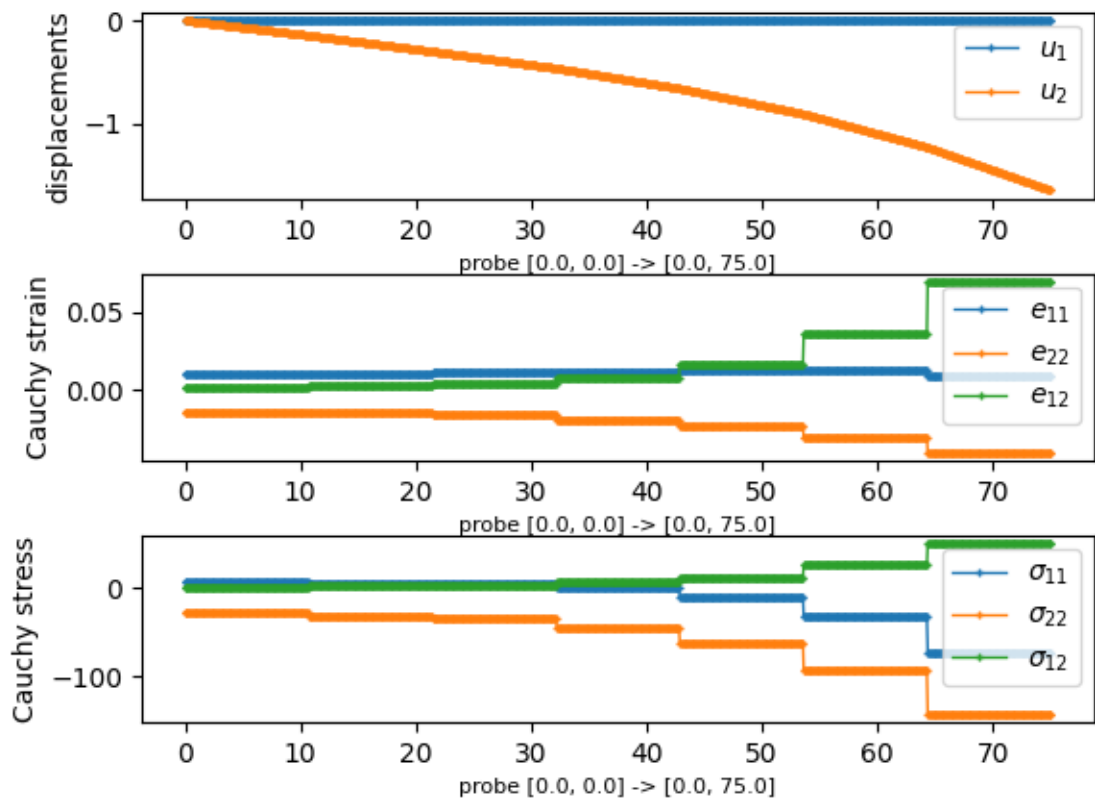
- material parameters
- displacement field approximation order
- uniform mesh refinement level

The example shows also how to probe the results as in *linear_elasticity/its2D_4.py*. Using *sfePy.discrete.probes* allows correct probing of fields with the approximation order greater than one.

In the SfePy top-level directory the following command can be used to get usage information:

```
python sfepy/examples/linear_elasticity/its2D_interactive.py -h
```





source code

```
#!/usr/bin/env python
"""
Diametrically point loaded 2-D disk, using commands for interactive use. See
:ref:`sec-primer`.

The script combines the functionality of all the ``its2D_?.py`` examples and
allows setting various simulation parameters, namely:

- material parameters
- displacement field approximation order
- uniform mesh refinement level

The example shows also how to probe the results as in
:ref:`linear_elasticity-its2D_4`. Using :mod:`sfepy.discrete.probes` allows
correct probing of fields with the approximation order greater than one.

In the SfePy top-level directory the following command can be used to get usage
information::

    python sfepy/examples/linear_elasticity/its2D_interactive.py -h
"""
```

(continues on next page)

(continued from previous page)

```

from __future__ import absolute_import
import sys
from six.moves import range
sys.path.append('.')
from argparse import ArgumentParser, RawDescriptionHelpFormatter

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import assert_, output, ordered_iteritems, IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Integrals,
                            Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.solvers.auto_fallback import AutoDirect
from sfepy.solvers.nls import Newton
from sfepy.discrete.fem.geometry_element import geometry_data
from sfepy.discrete.probes import LineProbe
from sfepy.discrete.projections import project_by_component

from sfepy.examples.linear_elasticity.its2D_2 import stress_strain
from sfepy.examples.linear_elasticity.its2D_3 import nodal_stress

def gen_lines(problem):
    """
    Define two line probes.

    Additional probes can be added by appending to `ps0` (start points) and
    `ps1` (end points) lists.
    """
    ps0 = [[0.0, 0.0], [0.0, 0.0]]
    ps1 = [[75.0, 0.0], [0.0, 75.0]]

    # Use enough points for higher order approximations.
    n_point = 1000

    labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
    probes = []
    for ip in range(len(ps0)):
        p0, p1 = ps0[ip], ps1[ip]
        probes.append(LineProbe(p0, p1, n_point))

    return probes, labels

def probe_results(u, strain, stress, probe, label):
    """
    Probe the results using the given probe and plot the probed values.
    """
    results = {}

```

(continues on next page)

(continued from previous page)

```

pars, vals = probe(u)
results['u'] = (pars, vals)
pars, vals = probe(strain)
results['cauchy_strain'] = (pars, vals)
pars, vals = probe(stress)
results['cauchy_stress'] = (pars, vals)

fig = plt.figure()
plt.clf()
fig.subplots_adjust(hspace=0.4)
plt.subplot(311)
pars, vals = results['u']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('displacements')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

sym_indices = ['11', '22', '12']

plt.subplot(312)
pars, vals = results['cauchy_strain']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$\epsilon_{%s}$' % sym_indices[ic],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy strain')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

plt.subplot(313)
pars, vals = results['cauchy_stress']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$\sigma_{%s}$' % sym_indices[ic],
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy stress')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

return fig, results

helps = {
    'young' : "the Young's modulus [default: %(default)s]",
    'poisson' : "the Poisson's ratio [default: %(default)s]",
    'load' : "the vertical load value (negative means compression)"
    " [default: %(default)s]",
    'order' : 'displacement field approximation order [default: %(default)s]',
    'refine' : 'uniform mesh refinement level [default: %(default)s]',
    'probe' : 'probe the results',
}

def main():

```

(continues on next page)

(continued from previous page)

```

from sfepy import data_dir

parser = ArgumentParser(description=__doc__,
                          formatter_class=RawDescriptionHelpFormatter)
parser.add_argument('--version', action='version', version='%(prog)s')
parser.add_argument('--young', metavar='float', type=float,
                    action='store', dest='young',
                    default=2000.0, help=helps['young'])
parser.add_argument('--poisson', metavar='float', type=float,
                    action='store', dest='poisson',
                    default=0.4, help=helps['poisson'])
parser.add_argument('--load', metavar='float', type=float,
                    action='store', dest='load',
                    default=-1000.0, help=helps['load'])
parser.add_argument('--order', metavar='int', type=int,
                    action='store', dest='order',
                    default=1, help=helps['order'])
parser.add_argument('-r', '--refine', metavar='int', type=int,
                    action='store', dest='refine',
                    default=0, help=helps['refine'])
parser.add_argument('-p', '--probe',
                    action="store_true", dest='probe',
                    default=False, help=helps['probe'])
options = parser.parse_args()

assert_((0.0 < options.poisson < 0.5),
        "Poisson's ratio must be in ]0, 0.5[!")
assert_((0 < options.order),
        'displacement approximation order must be at least 1!')

output('using values:')
output(" Young's modulus:", options.young)
output(" Poisson's ratio:", options.poisson)
output(' vertical load:', options.load)
output('uniform mesh refinement level:', options.refine)

# Build the problem definition.
mesh = Mesh.from_file(data_dir + '/meshes/2d/its2D.mesh')
domain = FEDomain('domain', mesh)

if options.refine > 0:
    for ii in range(options.refine):
        output('refine %d...' % ii)
        domain = domain.refine()
        output('... %d nodes %d elements'
              % (domain.shape.n_nod, domain.shape.n_el))

omega = domain.create_region('Omega', 'all')
left = domain.create_region('Left',
                            'vertices in x < 0.001', 'facet')
bottom = domain.create_region('Bottom',
                              'vertices in y < 0.001', 'facet')

```

(continues on next page)

(continued from previous page)

```

top = domain.create_region('Top', 'vertex 2', 'vertex')

field = Field.from_args('fu', nm.float64, 'vector', omega,
                        approx_order=options.order)

u = FieldVariable('u', 'unknown', field)
v = FieldVariable('v', 'test', field, primary_var_name='u')

D = stiffness_from_youngpoisson(2, options.young, options.poisson)

asphalt = Material('Asphalt', D=D)
load = Material('Load', values={'val' : [0.0, options.load]})

integral = Integral('i', order=2*options.order)
integral0 = Integral('i', order=0)

t1 = Term.new('dw_lin_elastic(Asphalt.D, v, u)',
              integral, omega, Asphalt=asphalt, v=v, u=u)
t2 = Term.new('dw_point_load(Load.val, v)',
              integral0, top, Load=load, v=v)
eq = Equation('balance', t1 - t2)
eqs = Equations([eq])

xsym = EssentialBC('XSym', bottom, {'u.1' : 0.0})
ysym = EssentialBC('YSym', left, {'u.0' : 0.0})

ls = AutoDirect({})

nls_status = IndexedStruct()
nls = Newton({}, lin_solver=ls, status=nls_status)

pb = Problem('elasticity', equations=eqs)

pb.set_bcs(ebcs=Conditions([xsym, ysym]))

pb.set_solver(nls)

# Solve the problem.
variables = pb.solve()
output(nls_status)

# Postprocess the solution.
out = variables.create_output()
out = stress_strain(out, pb, variables, extend=True)
pb.save_state('its2D_interactive.vtk', out=out)

gdata = geometry_data['2_3']
nc = len(gdata.coors)

integral_vn = Integral('ivn', coors=gdata.coors,
                      weights=[gdata.volume / nc] * nc)

```

(continues on next page)

(continued from previous page)

```

nodal_stress(out, pb, variables, integrals=Integrals([integral_vn]))

if options.probe:
    # Probe the solution.
    probes, labels = gen_lines(pb)

    sfield = Field.from_args('sym_tensor', nm.float64, 3, omega,
                            approx_order=options.order - 1)
    stress = FieldVariable('stress', 'parameter', sfield,
                          primary_var_name='(set-to-None)')
    strain = FieldVariable('strain', 'parameter', sfield,
                          primary_var_name='(set-to-None)')

    cfield = Field.from_args('component', nm.float64, 1, omega,
                            approx_order=options.order - 1)
    component = FieldVariable('component', 'parameter', cfield,
                             primary_var_name='(set-to-None)')

    ev = pb.evaluate
    order = 2 * (options.order - 1)
    strain_qp = ev('ev_cauchy_strain.%d.Omega(u)' % order, mode='qp')
    stress_qp = ev('ev_cauchy_stress.%d.Omega(Asphalt.D, u)' % order,
                  mode='qp', copy_materials=False)

    project_by_component(strain, strain_qp, component, order)
    project_by_component(stress, stress_qp, component, order)

    all_results = []
    for ii, probe in enumerate(probes):
        fig, results = probe_results(u, strain, stress, probe, labels[ii])

        fig.savefig('its2D_interactive_probe_%d.png' % ii)
        all_results.append(results)

    for ii, results in enumerate(all_results):
        output('probe %d:' % ii)
        output.level += 2
        for key, res in ordered_iteritems(results):
            output(key + ':')
            val = res[1]
            output('  min: %+.2e, mean: %+.2e, max: %+.2e'
                  % (val.min(), val.mean(), val.max()))
        output.level -= 2

if __name__ == '__main__':
    main()

```


linear_elasticity/linear_elastic.py**Description**

Linear elasticity with given displacements.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

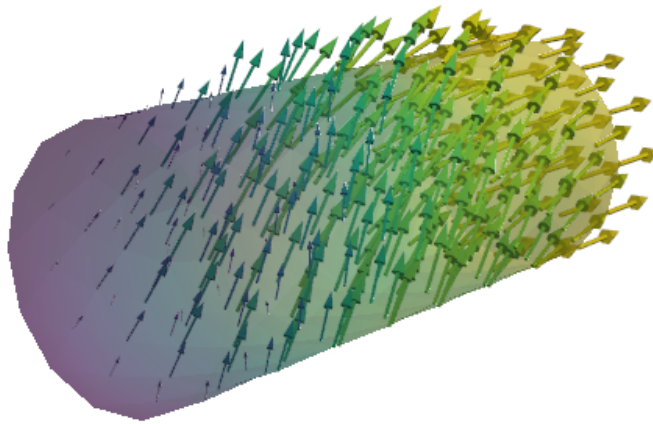
This example models a cylinder that is fixed at one end while the second end has a specified displacement of 0.01 in the x direction (this boundary condition is named 'Displaced'). There is also a specified displacement of 0.005 in the z direction for points in the region labeled 'SomewhereTop'. This boundary condition is named 'PerturbedSurface'. The region 'SomewhereTop' is specified as those vertices for which:

```
(z > 0.017) & (x > 0.03) & (x < 0.07)
```

The displacement field (three DOFs/node) in the 'Omega region' is approximated using P1 (four-node tetrahedral) finite elements. The material is linear elastic and its properties are specified as Lamé parameters λ and μ (see http://en.wikipedia.org/wiki/Lam%C3%A9_parameters)

The output is the displacement for each vertex, saved by default to cylinder.vtk. View the results using:

```
sfepy-view cylinder.vtk -f u:wu 1:vw
```



source code

```
# -*- coding: utf-8 -*-
r"""
Linear elasticity with given displacements.

Find :math:\ul{u} such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

This example models a cylinder that is fixed at one end while the second end
has a specified displacement of 0.01 in the x direction (this boundary
condition is named ``Displaced``). There is also a specified displacement of
```

(continues on next page)

(continued from previous page)

0.005 in the z direction for points in the region labeled ``SomewhereTop``. This boundary condition is named ``PerturbedSurface``. The region ``SomewhereTop`` is specified as those vertices for which::

$$(z > 0.017) \ \& \ (x > 0.03) \ \& \ (x < 0.07)$$

The displacement field (three DOFs/node) in the ``Omega region`` is approximated using P1 (four-node tetrahedral) finite elements. The material is linear elastic and its properties are specified as Lamé parameters :math:`\lambda` and :math:`\mu` (see http://en.wikipedia.org/wiki/Lam%C3%A9_parameters)

The output is the displacement for each vertex, saved by default to cylinder.vtk. View the results using::

```
sfePy-view cylinder.vtk -f u:wu 1:vw
"""
from __future__ import absolute_import
from sfePy import data_dir
from sfePy.mechanics.matcoefs import stiffness_from_lame

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
    'SomewhereTop' : ('vertices in (z > 0.017) & (x > 0.03) & (x < 0.07)',
                     'vertex'),
}

materials = {
    'solid' : ({'D': stiffness_from_lame(dim=3, lam=1e1, mu=1e0)},),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i' : 1,
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'Displaced' : ('Right', {'u.0' : 0.01, 'u.[1,2]' : 0.0}),
}
```

(continues on next page)

(continued from previous page)

```
'PerturbedSurface' : ('SomewhereTop', {'u.2' : 0.005}),
}

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
    }),
}
```

linear_elasticity/linear_elastic_damping.py

Description

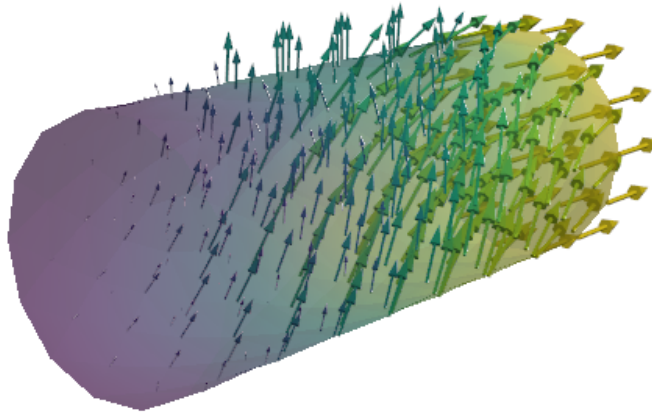
Time-dependent linear elasticity with a simple damping.

Find $\underline{u}$ such that:

$$\int_{\Omega} c \, \underline{v} \cdot \frac{\partial \underline{u}}{\partial t} + \int_{\Omega} D_{ijkl} \, e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \, \delta_{ij}\delta_{kl}.$$



source code

```

r"""
Time-dependent linear elasticity with a simple damping.

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} c \, \ul{v} \cdot \pdiff{\ul{u}}{t}
    + \int_{\Omega} D_{ijkl} \, e_{ij}(\ul{v}) \, e_{kl}(\ul{u})
    = 0
    \quad \forall \ul{v};

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad.
"""
from __future__ import print_function
from __future__ import absolute_import
from copy import deepcopy

```

(continues on next page)

(continued from previous page)

```

import numpy as nm
from sfepy.examples.linear_elasticity.linear_elastic import \
    filename_mesh, materials, regions, fields, ebcs, \
    integrals, solvers

def print_times(problem, state):
    print(nm.array(problem.ts.times))

options = {
    'ts' : 'ts',
    'save_times' : 'all',
    'post_process_hook_final' : print_times,
    'output_format' : 'h5',
}

variables = {
    'u' : ('unknown field', 'displacement', 0, 1),
    'v' : ('test field', 'displacement', 'u'),
}

# Put density to 'solid'.
materials = deepcopy(materials)
materials['solid'][0].update({'c' : 1000.0})

# Moving the PerturbedSurface region.
ebcs = deepcopy(ebcs)
ebcs['PerturbedSurface'][1].update({'u.0' : 'ebc_sin'})

def ebc_sin(ts, coors, **kwargs):
    val = 0.01 * nm.sin(2.0*nm.pi*ts.nt)
    return nm.tile(val, (coors.shape[0],))

equations = {
    'balance_of_forces in time' :
    """dw_dot.i.Omega( solid.c, v, du/dt )
    + dw_lin_elastic.i.Omega( solid.D, v, u ) = 0""",
}

def adapt_time_step(ts, status, adt, problem, verbose=False):
    if ts.time > 0.5:
        ts.set_time_step(0.1)

    return True

solvers = deepcopy(solvers) # Do not spoil linear_elastic.py namespace in tests.
solvers.update({
    'ts' : ('ts.adaptive', {
        't0' : 0.0,
        't1' : 1.0,
        'dt' : None,
        'n_step' : 101,

```

(continues on next page)

(continued from previous page)

```

        'adapt_fun' : adapt_time_step,
        'verbose' : 1,
    }},
})

ls = solvers['ls']
ls[1].update({'use_presolve' : True})

functions = {
    'ebc_sin' : (ebc_sin,),
}

```

linear_elasticity/linear_elastic_iga.py

Description

Linear elasticity solved in a single patch NURBS domain using the isogeometric analysis (IGA) approach.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

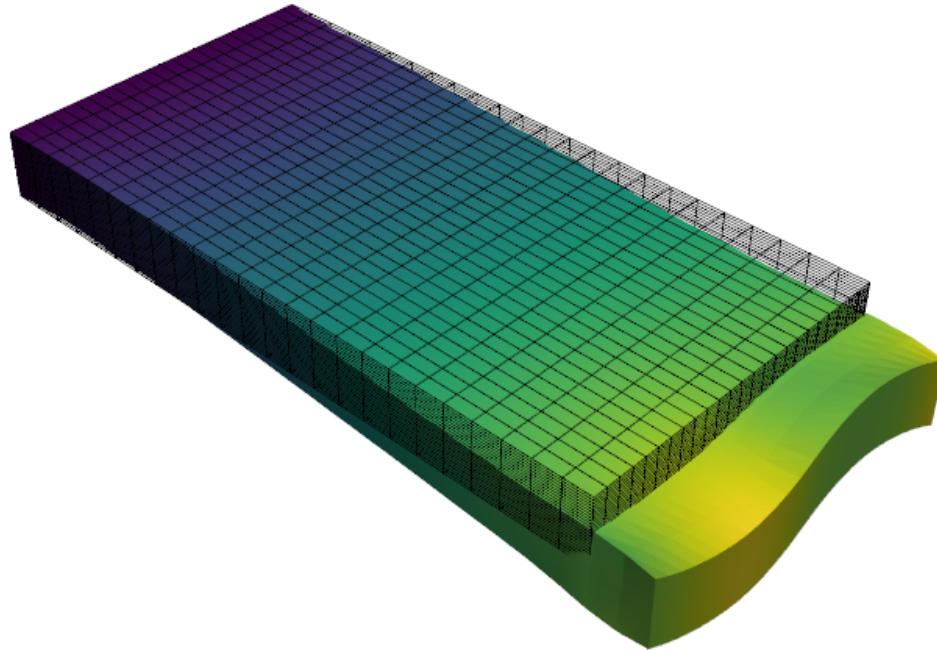
The domain geometry was created by:

```
sfepy-mesh iga-patch -d [1,0.5,0.1] -s [11,5,3] --degrees [2,2,2] -o meshes/iga/block3d.
↪ iga
```

View the results using:

```
sfepy-view block3d.vtk -f u:wu 1:vw
```

0.00 u 0.128



source code

```

r"""
Linear elasticity solved in a single patch NURBS domain using the isogeometric
analysis (IGA) approach.

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

The domain geometry was created by::

    sfepy-mesh iga-patch -d [1,0.5,0.1] -s [11,5,3] --degrees [2,2,2] -o meshes/iga/

```

(continues on next page)

(continued from previous page)

```

↪ block3d.iga

View the results using::

    sfepy-view block3d.vtk -f u:vu 1:vw
"""
from __future__ import absolute_import
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy import data_dir

filename_domain = data_dir + '/meshes/iga/block3d.iga'

regions = {
    'Omega' : 'all',
    'Gamma1' : ('vertices of set xi00', 'facet'),
    'Gamma2' : ('vertices of set xi01', 'facet'),
}

materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=5.769, mu=3.846),
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', None, 'H1', 'iga'),
}

integrals = {
    'i' : 3,
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'u1' : ('Gamma1', {'u.all' : 0.0}),
    'u2' : ('Gamma2', {'u.0' : 0.1, 'u.[1,2]' : 'get_ebcs'}),
}

def get_ebcs(ts, coors, **kwargs):
    import numpy as nm

    aux = nm.empty_like(coors[:, 1:])
    aux[:, 0] = 0.1 * coors[:, 1]
    aux[:, 1] = -0.05 + 0.03 * nm.sin(coors[:, 1] * 5 * nm.pi)

    return aux

functions = {

```

(continues on next page)

(continued from previous page)

```
'get_ebcs' : (get_ebcs,),
}

equations = {
    'balance_of_forces' : """dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

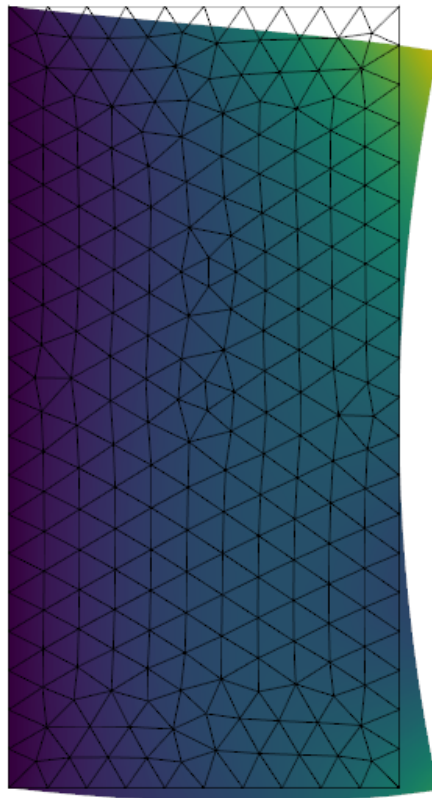
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
    }),
}
```

linear_elasticity/linear_elastic_interactive.py

Description

Linear elasticity example using the imperative API.

0.00 u 1.52



source code

```
#!/usr/bin/env python
"""
Linear elasticity example using the imperative API.
"""
from argparse import ArgumentParser
import numpy as nm

import sys
sys.path.append('.')

from sfepy.base.base import IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                             Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.mechanics.matcoefs import stiffness_from_lame

def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):
    """
    Define a displacement depending on the y coordinate.
    """
    val = shift * coors[:,1]**2

    return val

def main():
    from sfepy import data_dir

    parser = ArgumentParser()
    parser.add_argument('--version', action='version', version='% (prog)s')
    options = parser.parse_args()

    mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
    domain = FEDomain('domain', mesh)

    min_x, max_x = domain.get_mesh_bounding_box()[:,0]
    eps = 1e-8 * (max_x - min_x)
    omega = domain.create_region('Omega', 'all')
    gamma1 = domain.create_region('Gamma1',
                                   'vertices in x < %.10f' % (min_x + eps),
                                   'facet')
    gamma2 = domain.create_region('Gamma2',
                                   'vertices in x > %.10f' % (max_x - eps),
                                   'facet')

    field = Field.from_args('fu', nm.float64, 'vector', omega,
                           approx_order=2)
```

(continues on next page)

(continued from previous page)

```

u = FieldVariable('u', 'unknown', field)
v = FieldVariable('v', 'test', field, primary_var_name='u')

m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
f = Material('f', val=[[0.02], [0.01]])

integral = Integral('i', order=3)

t1 = Term.new('dw_lin_elastic(m.D, v, u)',
              integral, omega, m=m, v=v, u=u)
t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
eq = Equation('balance', t1 + t2)
eqs = Equations([eq])

fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})

bc_fun = Function('shift_u_fun', shift_u_fun,
                  extra_args={'shift' : 0.01})
shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})

ls = ScipyDirect({})

nls_status = IndexedStruct()
nls = Newton({}, lin_solver=ls, status=nls_status)

pb = Problem('elasticity', equations=eqs)
pb.save_regions_as_groups('regions')

pb.set_bcs(ebcs=Conditions([fix_u, shift_u]))

pb.set_solver(nls)

status = IndexedStruct()
variables = pb.solve(status=status)

print('Nonlinear solver status:\n', nls_status)
print('Stationary solver status:\n', status)

pb.save_state('linear_elasticity.vtk', variables)

if __name__ == '__main__':
    main()

```

linear_elasticity/linear_elastic_probes.py

Description

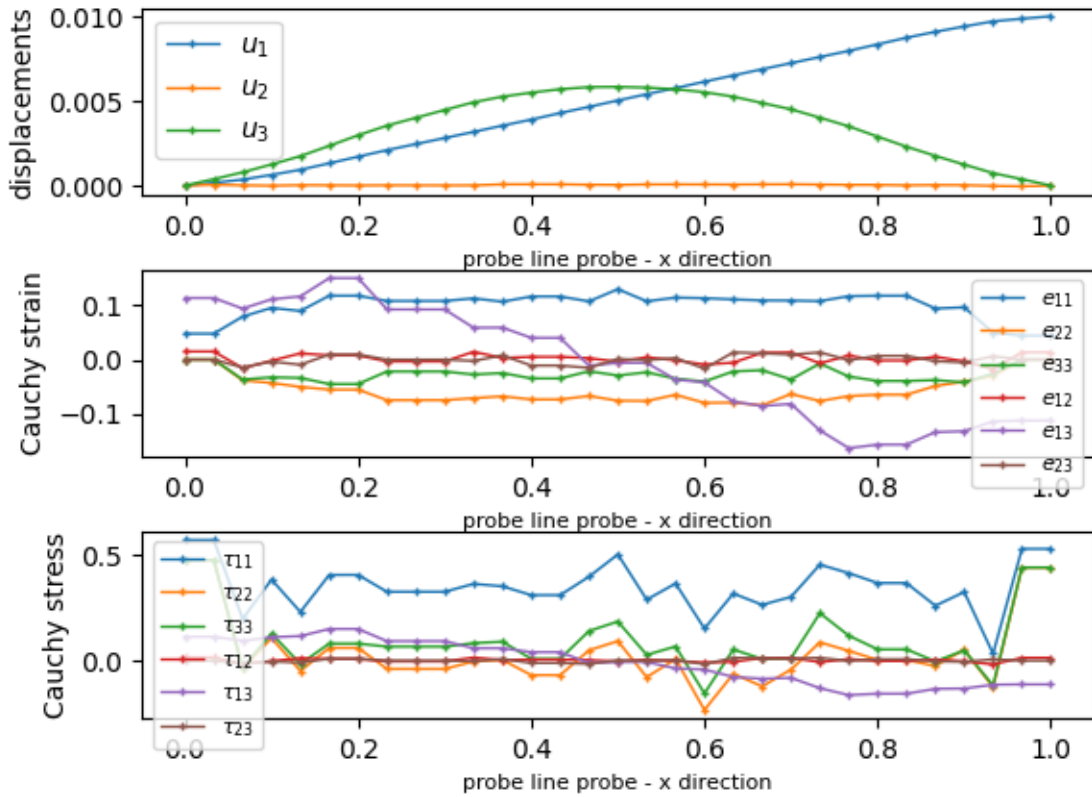
This example shows how to use the `post_process_hook` to probe the output data.

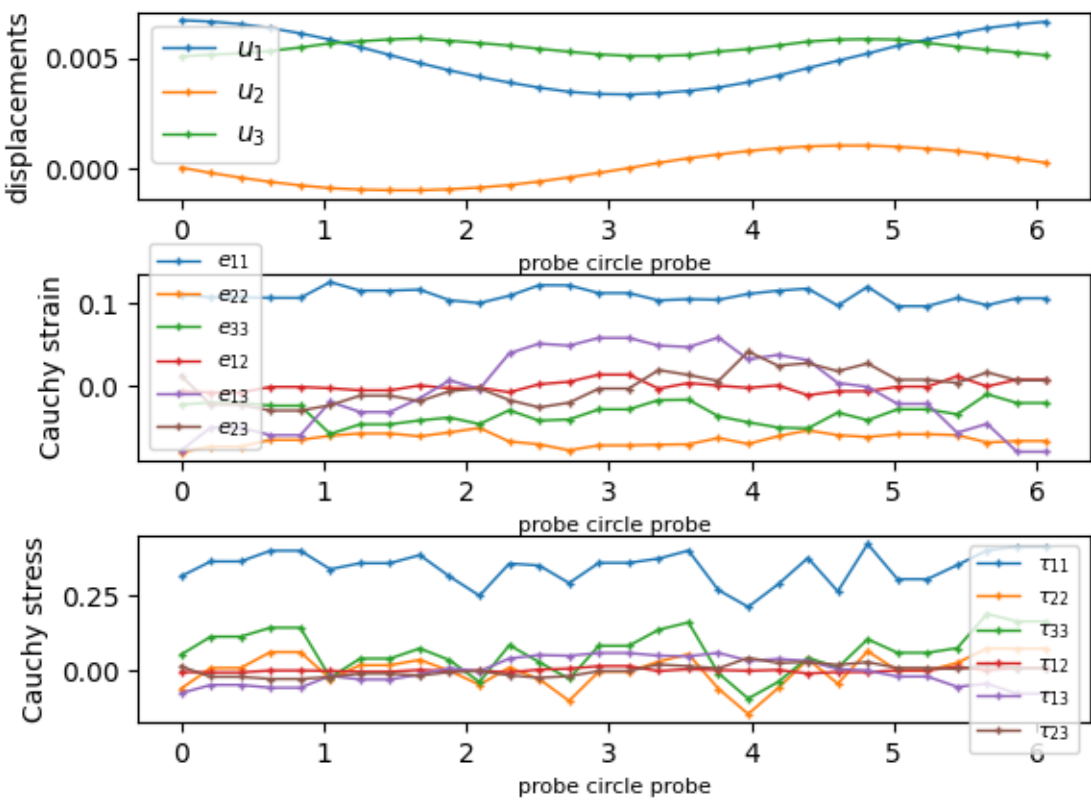
Find $\underline{u}$ such that:

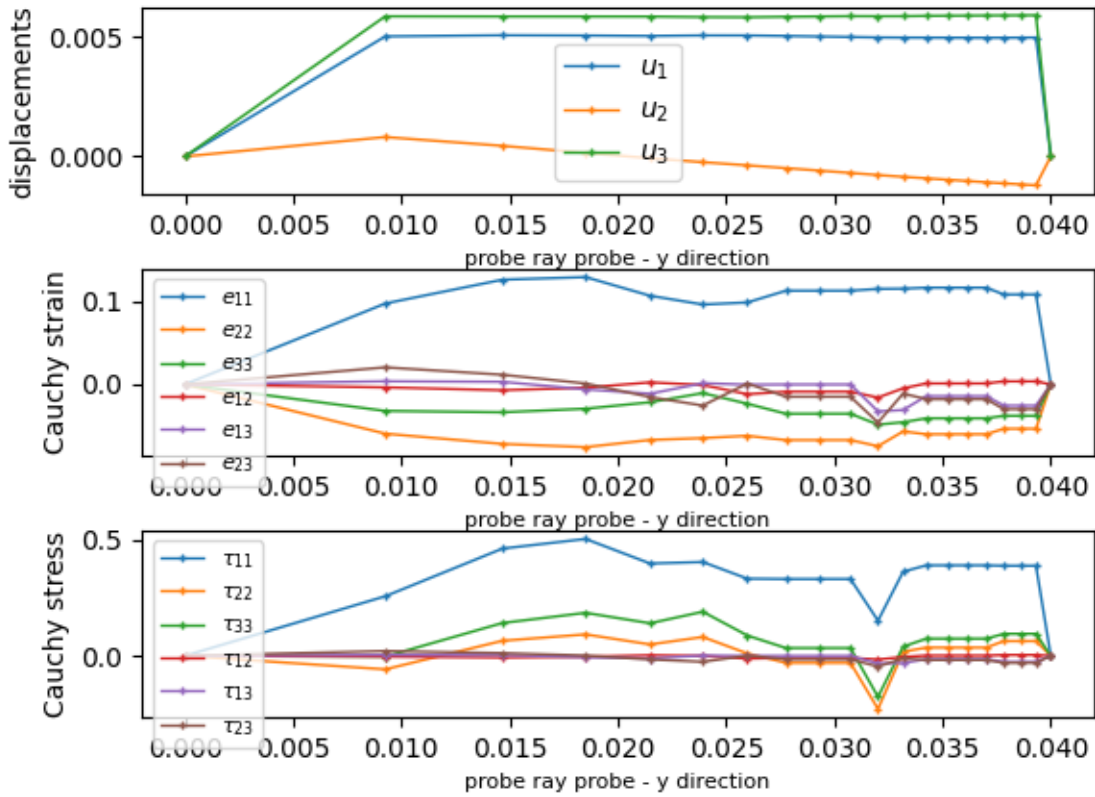
$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

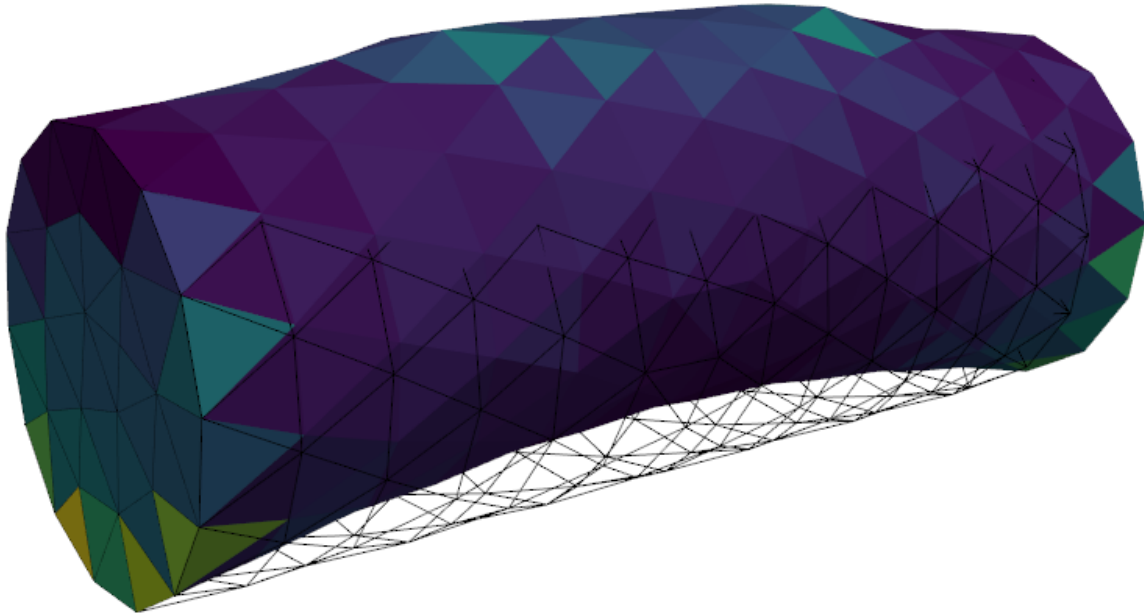
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$







cauchy_stress
0.0822 2.48



source code

```
r"""
This example shows how to use the post_process_hook to probe the output data.

Find  $\mathbf{u}$  such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
    = 0
    \quad \forall \mathbf{v};
where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad .
"""
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.linear_elastic import *
import os
import numpy as nm
```

(continues on next page)

(continued from previous page)

```

from sfepy.base.base import Struct
try:
    from sfepy.postprocess.probes_vtk import Probe

except ImportError:
    Probe = None

from six.moves import range

# Define options.
options = {
    'output_dir' : '.',
    'post_process_hook' : 'post_process',
}

# The function returning the probe parametrization.
def par_fun(idx):
    return nm.log(idx + 1) / nm.log(20) * 0.04

# Define the function post_process, that will be called after the problem is
# solved.
def post_process(out, problem, variables, extend=False):
    """
    This will be called after the problem is solved.

    Parameters
    -----
    out : dict
        The output dictionary, where this function will store additional data.
    problem : Problem instance
        The current Problem instance.
    variables : Variables instance
        The computed state, containing FE coefficients of all the unknown
        variables.
    extend : bool
        The flag indicating whether to extend the output data to the whole
        domain. It can be ignored if the problem is solved on the whole domain
        already.

    Returns
    -----
    out : dict
        The updated output dictionary.
    """
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm

    # Cauchy strain averaged in elements.
    strain = problem.evaluate('ev_cauchy_strain.i.Omega( u )',
                              mode='el_avg')
    out['cauchy_strain'] = Struct(name='output_data',

```

(continues on next page)

(continued from previous page)

```

                                mode='cell', data=strain,
                                dofs=None)
# Cauchy stress averaged in elements.
stress = problem.evaluate('ev_cauchy_stress.i.Omega( solid.D, u )',
                           mode='el_avg')
out['cauchy_stress'] = Struct(name='output_data',
                              mode='cell', data=stress,
                              dofs=None)

if Probe is None:
    # Do not probe if vtk cannot be imported.
    return out

# Define three line probes in axial directions.
mesh = problem.domain.mesh

bbox = mesh.get_bounding_box()
cx, cy, cz = 0.5 * bbox.sum(axis=0)

labels = []
probe_names = []

probe = Probe(out, mesh)

# line probe
labels.append('line probe - x direction')
probe_names.append('line')
probe.add_line_probe('line',
                     [bbox[0,0], cy, cz],
                     [bbox[1,0], cy, cz],
                     30)

# circle probe
labels.append('circle probe')
probe_names.append('circle')
probe.add_circle_probe('circle',
                       [cx, cy, cz],
                       [0, 0, 1],
                       0.015,
                       30)

# ray probe
labels.append('ray probe - y direction')
probe_names.append('ray')
probe.add_ray_probe('ray',
                    [cx, bbox[0,1], cz],
                    [0, 1, 0],
                    par_fun, 20)

# Gnerate matplotlib figures with the probe plot.
for probe_name, label in zip(probe_names, labels):

```

(continues on next page)

(continued from previous page)

```

fig = plt.figure()
plt.clf()
fig.subplots_adjust(hspace=0.4)

plt.subplot(311)
pars, vals = probe(probe_name, 'u')
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('displacements')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', prop=fm.FontProperties(size=10))

sym_labels = ['11', '22', '33', '12', '13', '23']

plt.subplot(312)
pars, vals = probe(probe_name, 'cauchy_strain')
for ii, lab in enumerate(sym_labels):
    plt.plot(pars, vals[:,ii], label=r'$e_{%s}$' % lab,
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy strain')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', prop=fm.FontProperties(size=8))

plt.subplot(313)
pars, vals = probe(probe_name, 'cauchy_stress')
for ii, lab in enumerate(sym_labels):
    plt.plot(pars, vals[:,ii], label=r'$\tau_{%s}$' % lab,
             lw=1, ls='-', marker='+', ms=3)
plt.ylabel('Cauchy stress')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', prop=fm.FontProperties(size=8))

opts = problem.conf.options
filename_results = os.path.join(opts.get('output_dir'),
                                'cylinder_probe_%s.png' % probe_name)

fig.savefig(filename_results)

return out

```

linear_elasticity/linear_elastic_tractions.py

Description

Linear elasticity with pressure traction load on a surface and constrained to one-dimensional motion.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} .$$

and $\underline{\underline{\sigma}} \cdot \underline{n} = \bar{p} \underline{I} \cdot \underline{n}$ with given traction pressure $\bar{p}$.

The function `verify_tractions()` is called after the solution to verify that the inner surface tractions correspond to the load applied to the external surface. Try running the example with different approximation orders and/or uniform refinement levels:

- the default options:

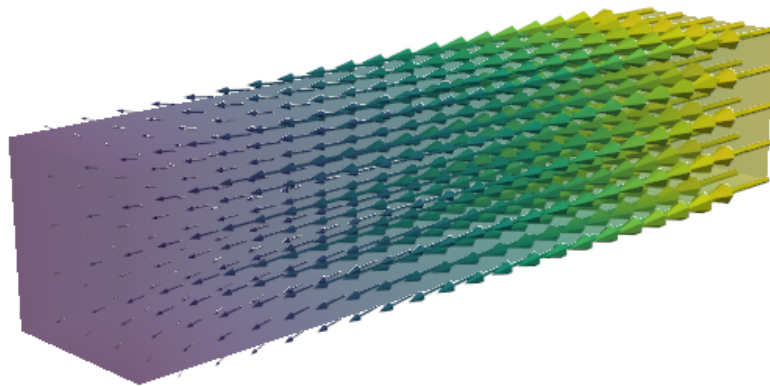
```
sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -0.1
↪refinement_level=0 -d approx_order=1
```

- refine once:

```
sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -0.1
↪refinement_level=1 -d approx_order=1
```

- use the tri-quadratic approximation (Q2):

```
sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -0.1
↪refinement_level=0 -d approx_order=2
```



source code

```

r"""
Linear elasticity with pressure traction load on a surface and constrained to
one-dimensional motion.

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \, e_{ij}(\ul{v}) \, e_{kl}(\ul{u})
    = - \int_{\Gamma_{right}} \ul{v} \cdot \ul{\sigma} \cdot \ul{n}
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

and :math:`\ul{\sigma} \cdot \ul{n} = \bar{p} \ul{I} \cdot \ul{n}`
with given traction pressure :math:`\bar{p}`.

The function :func:`verify_tractions()` is called after the solution to verify
that the inner surface tractions correspond to the load applied to the external
surface. Try running the example with different approximation orders and/or uniform
↪ refinement levels:

- the default options::

    sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -O refinement_
    ↪ level=0 -d approx_order=1

- refine once::

    sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -O refinement_
    ↪ level=1 -d approx_order=1

- use the tri-quadratic approximation (Q2)::

    sfepy-run sfepy/examples/linear_elasticity/linear_elastic_tractions.py -O refinement_
    ↪ level=0 -d approx_order=2
"""
from __future__ import absolute_import
import numpy as nm
from sfepy.base.base import output
from sfepy.mechanics.matcoefs import stiffness_from_lame

def linear_tension(ts, coor, mode=None, **kwargs):
    if mode == 'qp':
        val = nm.tile(1.0, (coor.shape[0], 1, 1))

        return {'val' : val}

def verify_tractions(out, problem, state, extend=False):

```

(continues on next page)

(continued from previous page)

```

"""
Verify that the inner surface tractions correspond to the load applied
to the external surface.
"""
from sfepy.mechanics.tensors import get_full_indices
from sfepy.discrete import Material, Function

load_force = problem.evaluate(
    'ev_integrate_mat.2.Right(load.val, u)'
)
output('surface load force:', load_force)

def eval_force(region_name):
    strain = problem.evaluate(
        'ev_cauchy_strain.i.%s(u)' % region_name, mode='qp',
        verbose=False,
    )
    D = problem.evaluate(
        'ev_integrate_mat.i.%s(solid.D, u)' % region_name,
        mode='qp',
        verbose=False,
    )

    normal = nm.array([1, 0, 0], dtype=nm.float64)

    s2f = get_full_indices(len(normal))
    stress = nm.einsum('cqij,cqjk->cqik', D, strain)
    # Full (matrix) form of stress.
    mstress = stress[..., s2f, 0]

    # Force in normal direction.
    force = nm.einsum('cqij,i,j->cq', mstress, normal, normal)

    def get_force(ts, coors, mode=None, **kwargs):
        if mode == 'qp':
            return {'force' : force.reshape(coors.shape[0], 1, 1)}
    aux = Material('aux', function=Function('get_force', get_force))

    middle_force = - problem.evaluate(
        'ev_integrate_mat.i.%s(aux.force, u)' % region_name,
        aux=aux,
        verbose=False,
    )
    output('%s section axial force:' % region_name, middle_force)

    eval_force('Left')
    eval_force('Middle')
    eval_force('Right')

    return out

def define(approx_order=1):

```

(continues on next page)

(continued from previous page)

```

"""Define the problem to solve."""
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/block.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'verify_tractions',
}

functions = {
    'linear_tension' : (linear_tension,),
}

fields = {
    'displacement': ('real', 3, 'Omega', approx_order),
}

materials = {
    'solid' : ({'D': stiffness_from_lame(3, lam=5.769, mu=3.846)}),
    'load' : (None, 'linear_tension')
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
    # Use a parent region to select only facets belonging to cells in the
    # parent region. Otherwise, each facet is in the region two times, with
    # opposite normals.
    'Middle' : ('vertices in (x > -1e-10) & (x < 1e-10)', 'facet', 'Rhalf'),
    'Rhalf' : 'vertices in x > -1e-10',
    'Right' : ('vertices in (x > 4.99)', 'facet'),
}

ebcs = {
    'fixb' : ('Left', {'u.all' : 0.0}),
    'fixt' : ('Right', {'u.[1,2]' : 0.0}),
}

integrals = {
    'i' : 2 * approx_order,
}

##
# Balance of forces.
equations = {

```

(continues on next page)

(continued from previous page)

```

    'elasticity' :
        """dw_lin_elastic.i.Omega( solid.D, v, u )
           = - dw_surface_ltr.i.Right( load.val, v )""",
    }

    ##
    # Solvers etc.
    solvers = {
        'ls' : ('ls.auto_direct', {}),
        'newton' : ('nls.newton',
            { 'i_max'      : 1,
              'eps_a'      : 1e-10,
              'eps_r'      : 1.0,
              'macheps'    : 1e-16,
              # Linear system error < (eps_a * lin_red).
              'lin_red'    : 1e-2,
              'ls_red'     : 0.1,
              'ls_red_warp' : 0.001,
              'ls_on'      : 1.1,
              'ls_min'     : 1e-5,
              'check'      : 0,
              'delta'      : 1e-6,
            })
    }

    return locals()

```

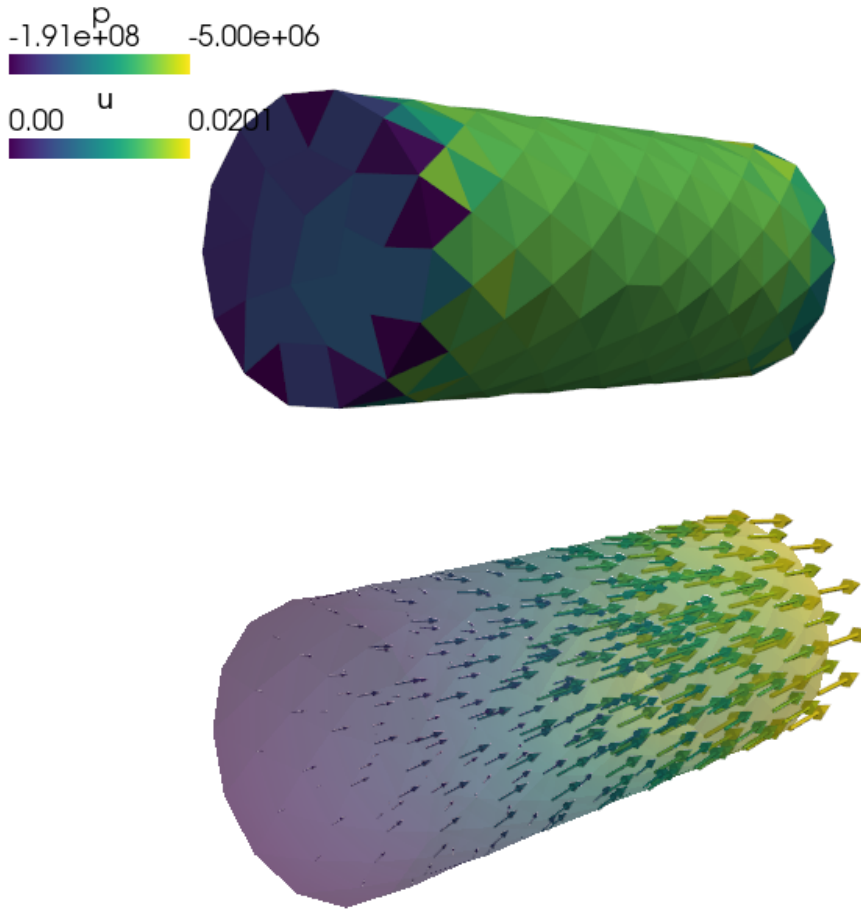
linear_elasticity/linear_elastic_up.py

Description

Nearly incompressible linear elasticity in mixed displacement-pressure formulation with comments.

Find $\underline{u}$, p such that:

$$\begin{aligned}
 \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \nabla \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\
 - \int_{\Omega} q \nabla \cdot \underline{u} - \int_{\Omega} \gamma q p &= 0, \quad \forall q.
 \end{aligned}$$



source code

```
r"""
Nearly incompressible linear elasticity in mixed displacement-pressure
formulation with comments.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u})
    - \int_{\Omega} p \nabla \cdot \mathbf{v}
    = 0
    \quad \forall \mathbf{v};

    - \int_{\Omega} q \nabla \cdot \mathbf{u}
    - \int_{\Omega} \gamma q p
    = 0
    \quad \forall q;.
"""
#!
#! Linear Elasticity
#! =====
#$ \centerline{Example input file, \today}
```

(continues on next page)

(continued from previous page)

```

#!/ This file models a cylinder that is fixed at one end while the
#!/ second end has a specified displacement of 0.02 in the x direction
#!/ (this boundary condition is named PerturbedSurface).
#!/ The output is the displacement for each node, saved by default to
#!/ simple_out.vtk. The material is linear elastic.
from __future__ import absolute_import
from sfepy import data_dir

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson_mixed, bulk_from_
↪youngpoisson

#!/ Mesh
#!/ ----

dim = 3
approx_u = '3_4_P1'
approx_p = '3_4_P0'
order = 2
filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'
#!/ Regions
#!/ -----
#!/ Whole domain 'Omega', left and right ends.
regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}
#!/ Materials
#!/ -----
#!/ The linear elastic material model is used.
materials = {
    'solid' : ({'D' : stiffness_from_youngpoisson_mixed(dim, 0.7e9, 0.4),
                'gamma' : 1.0/bulk_from_youngpoisson(0.7e9, 0.4)},),
}
#!/ Fields
#!/ -----
#!/ A field is used to define the approximation on a (sub)domain
fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
    'pressure' : ('real', 'scalar', 'Omega', 0),
}
#!/ Integrals
#!/ -----
#!/ Define the integral type Volume/Surface and quadrature rule.
integrals = {
    'i' : order,
}
#!/ Variables
#!/ -----
#!/ Define displacement and pressure fields and corresponding fields
#!/ for test variables.

```

(continues on next page)

(continued from previous page)

```

variables = {
    'u' : ('unknown field', 'displacement'),
    'v' : ('test field', 'displacement', 'u'),
    'p' : ('unknown field', 'pressure'),
    'q' : ('test field', 'pressure', 'p'),
}
#! Boundary Conditions
#! -----
#! The left end of the cylinder is fixed (all DOFs are zero) and
#! the 'right' end has non-zero displacements only in the x direction.
ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'PerturbedSurface' : ('Right', {'u.0' : 0.02, 'u.1' : 0.0}),
}
#! Equations
#! -----
#! The weak formulation of the linear elastic problem.
equations = {
    'balance_of_forces' :
        """ dw_lin_elastic.i.Omega( solid.D, v, u )
           - dw_stokes.i.Omega( v, p )
           = 0 """ ,
    'pressure constraint' :
        """ - dw_stokes.i.Omega( u, q )
           - dw_dot.i.Omega( solid.gamma, q, p )
           = 0 """ ,
}
#! Solvers
#! -----
#! Define linear and nonlinear solver.
#! Even linear problems are solved by a nonlinear solver - only one
#! iteration is needed and the final residual is obtained for free.
solvers = {
    'ls': ('ls.schur_mumps', {
        'schur_variables': ['p'],
        'fallback': 'ls2'
    }),
    'ls2': ('ls.scipy_umfpack', {'fallback': 'ls3'}),
    'ls3': ('ls.scipy_superlu', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-2,
        'eps_r'      : 1e-10,
    }),
}
#! Options
#! -----
#! Various problem-specific options.
options = {
    'output_dir' : './output',
    'absolute_mesh_path' : True,
}

```

linear_elasticity/linear_viscoelastic.py

Description

Linear viscoelasticity with pressure traction load on a surface and constrained to one-dimensional motion.

The fading memory terms require an unloaded initial configuration, so the load starts in the second time step. The load is then held for the first half of the total time interval, and released afterwards.

This example uses exponential fading memory kernel $\mathcal{H}_{ijkl}(t) = \mathcal{H}_{ijkl}(0)e^{-dt}$ with decay d . Two equation kinds are supported - ‘th’ and ‘eth’. In ‘th’ mode the tabulated kernel is linearly interpolated to required times using `interp_conv_mat()`. In ‘eth’ mode, the computation is exact for exponential kernels.

Find $\underline{u}$ such that:

$$\begin{aligned} & \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \\ & + \int_{\Omega} \left[\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl} \left(\frac{\partial \underline{u}}{\partial \tau}(\tau) \right) d\tau \right] e_{ij}(\underline{v}) \\ & = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \quad \forall \underline{v}, \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

$\mathcal{H}_{ijkl}(0)$ has the same structure as D_{ijkl} and $\underline{\underline{\sigma}} \cdot \underline{n} = \bar{p} \underline{I} \cdot \underline{n}$ with given traction pressure $\bar{p}$.

Notes

Because this example is run also as a test, it uses by default very few time steps. Try changing that.

Visualization

The output file is assumed to be ‘block.h5’ in the working directory. Change it appropriately for your situation.

Deforming mesh

Try to run the following:

```
sfepy-view block.h5 -s 20 -f u:wu:file0:p0 1:vw:p0 total_stress:p1
```

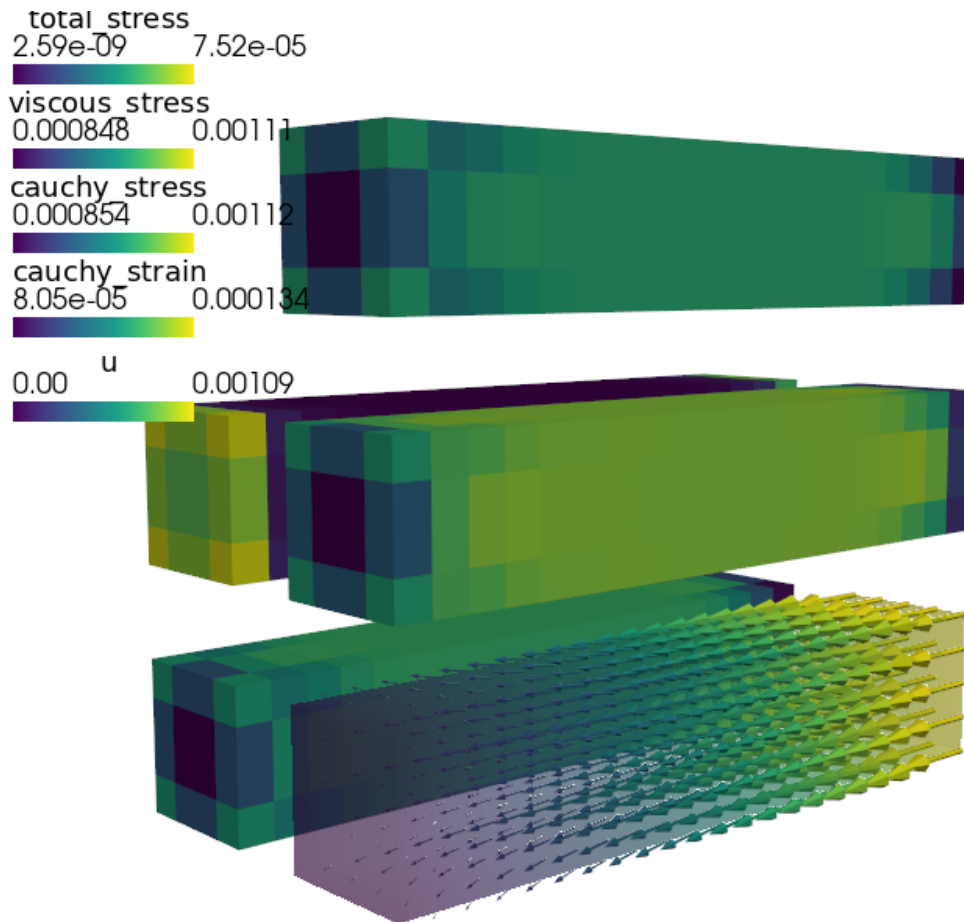
to see the results.

Time history plots

Run the following:

```
python3 sfepy/examples/linear_elasticity/linear_viscoelastic.py -h
python3 sfepy/examples/linear_elasticity/linear_viscoelastic.py block.h5
```

Try comparing 'th' and 'eth' versions, e.g., for $n\_step = 201$, and $f\_n\_step = 51$. There is a visible notch on viscous stress curves in the 'th' mode, as the fading memory kernel is cut off before it goes close enough to zero.



source code

```
#!/usr/bin/env python
r"""
Linear viscoelasticity with pressure traction load on a surface and constrained
to one-dimensional motion.

The fading memory terms require an unloaded initial configuration, so the load
starts in the second time step. The load is then held for the first half of the
total time interval, and released afterwards.

This example uses exponential fading memory kernel
:math:`\mathcal{H}_{ijkl}(t) = \mathcal{H}_{ijkl}(0) e^{-d t}` with decay
:math:`d`. Two equation kinds are supported - 'th' and 'eth'. In 'th'
```

(continues on next page)

(continued from previous page)

mode the tabulated kernel is linearly interpolated to required times using `:func:'interp_conv_mat()'`. In 'eth' mode, the computation is exact for exponential kernels.

Find $\ul{u}$ such that:

```
.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u}) \ \
    + \int_{\Omega} \left[ \int_0^t
    \ Hcal_{ijkl}(t-\tau) \ , e_{kl}(\pdiff{\ul{u}}{\tau}(\tau)) \ difd{\tau}
    \right] \ , e_{ij}(\ul{v}) \ \
    = - \int_{\Gamma_{right}} \ul{v} \cdot \ull{\sigma} \cdot \ul{n}
    \ ;, \quad \forall \ul{v} \ ;,
```

where

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \ ;,
```

$\Hcal_{ijkl}(0)$ has the same structure as D_{ijkl} and $\ull{\sigma} \cdot \ul{n} = \bar{p} \ \ull{I} \cdot \ul{n}$ with given traction pressure $\bar{p}$.

Notes

Because this example is run also as a test, it uses by default very few time steps. Try changing that.

Visualization

The output file is assumed to be 'block.h5' in the working directory. Change it appropriately for your situation.

Deforming mesh

AAAAAAAAAAAAAAAA

Try to run the following::

```
sfepy-view block.h5 -s 20 -f u:wu:f1e0:p0 l:vw:p0 total_stress:p1
```

to see the results.

Time history plots

AAAAAAAAAAAAAAAA

Run the following::

```
python3 sfepy/examples/linear_elasticity/linear_viscoelastic.py -h
```

(continues on next page)

(continued from previous page)

```
python3 sfepy/examples/linear_elasticity/linear_viscoelastic.py block.h5

Try comparing 'th' and 'eth' versions, e.g., for n_step = 201, and f_n_step =
51. There is a visible notch on viscous stress curves in the 'th' mode, as the
fading memory kernel is cut off before it goes close enough to zero.
"""
import numpy as nm

import sys
sys.path.append('.')

from sfepy.base.base import output
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.homogenization.utils import interp_conv_mat
from sfepy import data_dir

def linear_tension(ts, coors, mode=None, verbose=True, **kwargs):
    if mode == 'qp':
        val = 1.0 * ((ts.step > 0) and (ts.nt <= 0.5))

        if verbose:
            output('load:', val)

        val = nm.tile(val, (coors.shape[0], 1, 1))

        return {'val' : val}

def get_exp_fading_kernel(coef0, decay, times):
    val = coef0[None, ...] * nm.exp(-decay * times[:, None, None])
    return val

def get_th_pars(ts, coors, mode=None, times=None, kernel=None, **kwargs):
    out = {}

    if mode == 'special':
        out['H'] = interp_conv_mat(kernel, ts, times)

    elif mode == 'qp':
        out['H0'] = kernel[0][None, ...]
        out['Hd'] = nm.array([[[kernel[1, 0, 0] / kernel[0, 0, 0]]]])

    return out

def post_process(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
```

(continues on next page)

(continued from previous page)

```

        data=strain, dofs=None)

estress = ev('ev_cauchy_stress.2.Omega(solid.D, u)', mode='el_avg')
out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                             data=estress, dofs=None)

ts = pb.get_timestepper()
if pb.conf.mode == 'th':
    vstress = ev('ev_cauchy_stress_th.2.Omega(ts, th.H, du/dt)',
                 ts=ts, mode='el_avg')
    out['viscous_stress'] = Struct(name='output_data', mode='cell',
                                   data=vsstress, dofs=None)

else:
    # The eth terms require 'preserve_caches=True' in order to have correct
    # fading memory history.
    vstress = ev('ev_cauchy_stress_eth.2.Omega(ts, th.H0, th.Hd, du/dt)',
                 ts=ts, mode='el_avg', preserve_caches=True)
    out['viscous_stress'] = Struct(name='output_data', mode='cell',
                                   data=vsstress, dofs=None)

out['total_stress'] = Struct(name='output_data', mode='cell',
                             data=estress + vstress, dofs=None)

return out

def define(verbose=False):
    filename_mesh = data_dir + '/meshes/3d/block.mesh'

    ## Configure below. ##

    # Time stepping times.
    t0 = 0.0
    t1 = 20.0
    n_step = 21

    # Fading memory times.
    f_t0 = 0.0
    f_t1 = 5.0
    f_n_step = 6

    decay = 0.8
    mode = 'eth'

    ## Configure above. ##

    times = nm.linspace(f_t0, f_t1, f_n_step)
    kernel = get_exp_fading_kernel(stiffness_from_lame(3, lam=1.0, mu=1.0),
                                   decay, times)

    dt = (t1 - t0) / (n_step - 1)
    fading_memory_length = min(int((f_t1 - f_t0) / dt) + 1, n_step)

```

(continues on next page)

(continued from previous page)

```

output('fading memory length:', fading_memory_length, verbose=verbose)

options = {
    'ts' : 'ts',
    'nls' : 'newton',
    'ls' : 'ls',

    'output_format' : 'h5',
    'post_process_hook' : 'post_process',
}

functions = {
    'linear_tension' : (linear_tension,),
    'get_pars' : (lambda ts, coors, mode=None, **kwargs:
                   get_th_pars(ts, coors, mode, times=times, kernel=kernel,
                               **kwargs),),
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
}

materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=5.769, mu=3.846),
    },),
    'th' : 'get_pars',
    'load' : 'linear_tension',
}

variables = {
    'u' : ('unknown field', 'displacement', 0, fading_memory_length),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
    'Right' : ('vertices in (x > 4.99)', 'facet'),
}

ebcs = {
    'fixb' : ('Left', {'u.all' : 0.0}),
    'fixt' : ('Right', {'u.[1,2]' : 0.0}),
}

if mode == 'th':
    # General form with tabulated kernel.
    equations = {
        'elasticity' :
        """dw_lin_elastic.2.Omega( solid.D, v, u )
          + dw_lin_elastic_th.2.Omega( ts, th.H, v, du/dt )

```

(continues on next page)

(continued from previous page)

```

        = - dw_surface_ltr.2.Right( load.val, v )""",
    }

    else:
        # Fast form that is exact for exponential kernels.
        equations = {
            'elasticity' :
            """dw_lin_elastic.2.Omega( solid.D, v, u )
            + dw_lin_elastic_eth.2.Omega( ts, th.H0, th.Hd, v, du/dt )
            = - dw_surface_ltr.2.Right( load.val, v )""",
        }

    solvers = {
        'ls' : ('ls.scipy_direct', {}),
        'newton' : ('nls.newton', {
            'i_max' : 1,
            'eps_a' : 1e-10,
        }),
        'ts' : ('ts.simple', {
            't0' : t0,
            't1' : t1,
            'dt' : None,
            'n_step' : n_step,
            'quasistatic' : True,
            'verbose' : 1,
        }),
    }

    return locals()

def main():
    """
    Plot the load, displacement, strain and stresses w.r.t. time.
    """
    from argparse import ArgumentParser, RawDescriptionHelpFormatter
    import matplotlib.pyplot as plt

    from sfepy.base.base import Struct
    import sfepy.postprocess.time_history as th

    msgs = {
        'node': 'plot displacements in given node [default: %(default)s]',
        'element': 'plot tensors in given element [default: %(default)s]',
    }

    parser = ArgumentParser(description=__doc__,
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument metavar='OUTPUT_FILE', dest='output_file',
                        help='output file in HDF5 format')
    parser.add_argument('-n', '--node', type=int, metavar='ii',
                        action='store', dest='node',
                        default=512, help=msgs['node'])

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('-e', '--element', type=int, metavar='ii',
                    action='store', dest='element',
                    default=299, help=msgs['element'])
options = parser.parse_args()

filename = options.output_file

tensor_names = ['cauchy_strain',
                'cauchy_stress', 'viscous_stress', 'total_stress']
extract = ('u n %d, ' % options.node) \
          + ', '.join('%s e %d' % (name, options.element)
                      for name in tensor_names)
ths, ts = th.extract_time_history(filename, extract)

load = [linear_tension(ts, nm.array([0]),
                      mode='qp', verbose=False)['val'].squeeze()
        for ii in ts]
load = nm.array(load)

conf = Struct(**define(verbose=True))
normalized_kernel = conf.kernel[:, 0, 0] / conf.kernel[0, 0, 0]

plt.figure(1, figsize=(8, 10))
plt.subplots_adjust(hspace=0.3,
                    top=0.95, bottom=0.05, left=0.07, right=0.95)

plt.subplot(311)
plt.plot(conf.times, normalized_kernel, lw=3)
plt.title('fading memory decay')
plt.xlabel('time')

plt.subplot(312)
plt.plot(ts.times, load, lw=3)
plt.title('load')
plt.xlabel('time')

displacements = ths['u'][options.node]

plt.subplot(313)
plt.plot(ts.times, displacements, lw=3)
plt.title('displacement components, node %d' % options.node)
plt.xlabel('time')
plt.tight_layout()

plt.figure(2, figsize=(8, 10))
plt.subplots_adjust(hspace=0.35,
                    top=0.95, bottom=0.05, left=0.07, right=0.95)

for ii, tensor_name in enumerate(tensor_names):
    tensor = ths[tensor_name][options.element]

    plt.subplot(411 + ii)

```

(continues on next page)

(continued from previous page)

```

plt.plot(ts.times, tensor, lw=3)
plt.title('%s components, element %d' % (tensor_name, options.element))
plt.xlabel('time')

plt.tight_layout()

plt.show()

if __name__ == '__main__':
    main()

```

linear_elasticity/material_nonlinearity.py

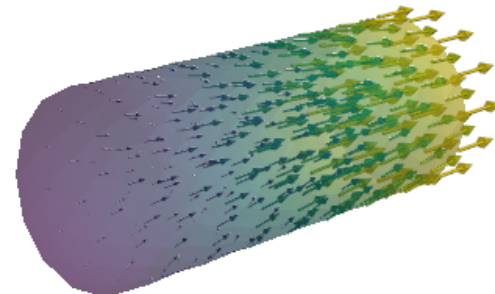
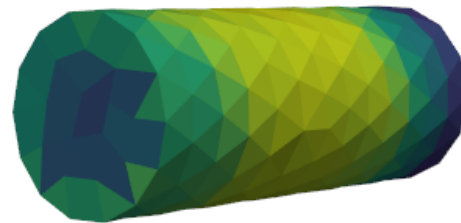
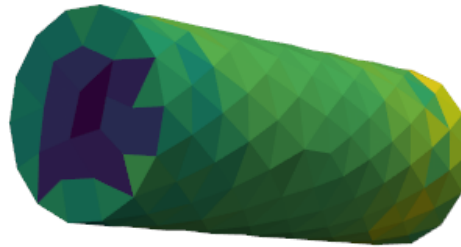
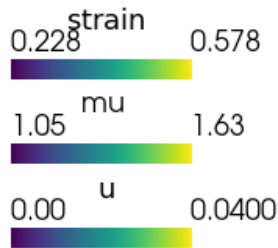
Description

Example demonstrating how a linear elastic term can be used to solve an elasticity problem with a material nonlinearity.

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```
# -*- coding: utf-8 -*-
r"""
Example demonstrating how a linear elastic term can be used to solve an
elasticity problem with a material nonlinearity.

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
    = 0
    \quad \forall \mathbf{v} \in V,
where
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad ;
"""
from __future__ import absolute_import
import numpy as nm

from sfepy.linalg import norm_l2_along_axis
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_lame

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    mu = pb.evaluate('ev_integrate_mat.2.Omega(nonlinear.mu, u)',
                     mode='el_avg', copy_materials=False, verbose=False)
    out['mu'] = Struct(name='mu', mode='cell', data=mu, dofs=None)

    strain = pb.evaluate('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    out['strain'] = Struct(name='strain', mode='cell', data=strain, dofs=None)

    return out

strains = [None]

def get_pars(ts, coors, mode='qp',
             equations=None, term=None, problem=None, **kwargs):
    """
    The material nonlinearity function - the Lamé coefficient  $\mu$ 
    depends on the strain.
    """
    if mode != 'qp': return

    val = nm.empty(coors.shape[0], dtype=nm.float64)
    val.fill(1e0)
```

(continues on next page)

(continued from previous page)

```

order = term.integral.order
uvar = equations.variables['u']

strain = problem.evaluate('ev_cauchy_strain.%d.Omega(u)' % order,
                          u=uvar, mode='qp')

if ts.step > 0:
    strain0 = strains[-1]

else:
    strain0 = strain

dstrain = (strain - strain0) / ts.dt
dstrain.shape = (strain.shape[0] * strain.shape[1], strain.shape[2])

norm = norm_l2_along_axis(dstrain)

val += norm

# Store history.
strains[0] = strain
return {'D': stiffness_from_lame(dim=3, lam=1e1, mu=val),
        'mu': val.reshape(-1, 1, 1)}

def pull(ts, coors, **kwargs):
    val = nm.empty_like(coors[:,0])
    val.fill(0.01 * ts.step)

    return val

functions = {
    'get_pars' : (get_pars,),
    'pull' : (pull,),
}

options = {
    'ts' : 'ts',
    'output_format' : 'h5',
    'save_times' : 'all',

    'post_process_hook' : 'post_process',
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Right' : ('vertices in (x > 0.099)', 'facet'),
}

materials = {
    'nonlinear' : 'get_pars',
}

```

(continues on next page)

(continued from previous page)

```

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
    'Displaced' : ('Right', {'u.0' : 'pull', 'u.[1,2]' : 0.0}),
}

equations = {
    'balance_of_forces in time' :
        """dw_lin_elastic.2.Omega(nonlinear.D, v, u) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
        {'i_max' : 1,
         'eps_a' : 1e-10,
         'eps_r' : 1.0,
        }),
    'ts' : ('ts.simple',
        {'t0' : 0.0,
         't1' : 1.0,
         'dt' : None,
         'n_step' : 5,
         'quasistatic' : True,
         'verbose' : 1,
        }),
}

```

linear_elasticity/mixed_mesh.py

Description

A linear elastic beam loaded with a continuous force. The FE mesh consists of hexahedral and tetrahedral elements.

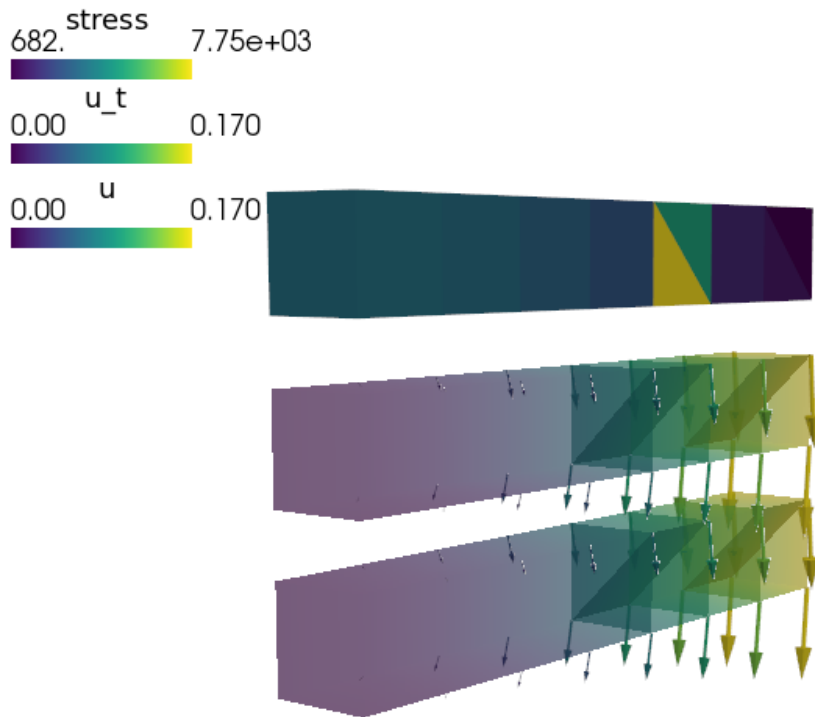
The displacement at the beam end is compared to the reference solution calculated on the homogeneous hexahedral mesh.

Running the simulation:

```
sfepy-run sfepy/examples/linear_elasticity/mixed_mesh.py
```

Viewing the results:

```
sfepy-view beam_h* -f u:s0:wu:e:p0 u:s1:wu:e:p0 --camera-position="1.2,-0.6,0.1,0.4,0.1,-
↪0.1,-0.2,0.1,1"
```



source code

```

r"""
A linear elastic beam loaded with a continuous force. The FE mesh consists
of hexehedral and tetrahedral elements.

The displacement at the beam end is compared to the reference
solution calculated on the homogeneous hexahedral mesh.

Running the simulation::

    sfepy-run sfepy/examples/linear_elasticity/mixed_mesh.py

Viewing the results::

    sfepy-view beam_h* -f u:s0:wu:e:p0 u:s1:wu:e:p0 --camera-position="1.2,-0.6,0.1,0.4,
    ↪0.1,-0.1,-0.2,0.1,1"
"""
from __future__ import absolute_import
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.base.base import Struct
import numpy as nm

```

(continues on next page)

(continued from previous page)

```

def reference_solution(pb):
    from sfepy.discrete import Problem

    conf = pb.conf.copy()
    conf.filename_mesh = data_dir + '/meshes/3d/beam_h7.mesh'
    for k in list(conf.regions.keys()):
        if '_t' in k:
            del conf.regions[k]

    for k in list(conf.fields.keys()):
        if '_t' in k:
            del conf.fields[k]

    for k in list(conf.variables.keys()):
        if '_t' in k:
            del conf.variables[k]

    conf.equations = {
        'balance_of_forces':
            """dw_lin_elastic.i.Omega(solid.D, v, u)
            = dw_surface_ltr.i.Top(force.val, v)"""
    }

    rpb = Problem.from_conf(conf)
    rpb.set_output_dir(pb.output_dir)

    return rpb.solve().get_state_parts()['u']

def get_force(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        F = 1e3

        val = nm.zeros_like(coors)[..., None]
        val[:, 2, 0] = -coors[:, 0] / 0.7 * F

        return {'val': val}

def post_proces(out, pb, state, extend=False):
    c1 = pb.domain.regions['Omega'].cells
    c2 = pb.domain.regions['Omega_t'].cells
    ed = pb.domain.regions['Edge'].vertices

    S = nm.zeros((pb.domain.cmesh.n_el, 1, 6, 1), dtype=nm.float64)

    S[c1] = pb.evaluate('ev_cauchy_stress.i.Omega(solid.D, u)',
                        mode='el_avg')
    S[c2] = pb.evaluate('ev_cauchy_stress.i.Omega_t(solid.D, u_t)',
                        mode='el_avg')

```

(continues on next page)

(continued from previous page)

```

out['stress'] = Struct(name='out', data=S, mode='cell')

u_nd = out['u'].data[ed[0], :]
ref_u_nd = reference_solution(pb).reshape(-1, 3)[ed[0], :]

du = nm.linalg.norm((ref_u_nd - u_nd) / nm.linalg.norm(ref_u_nd))
print(f'Relative difference with respect to the reference solution: {du}')

return out

filename_mesh = data_dir + '/meshes/3d/beam_h5t12.mesh'

options = {
    'post_process_hook': 'post_proces',
}

regions = {
    'Omega_': 'all',
    'Omega': ('cells of group 1', 'cell', None, {'vertices_from': 'Omega_'}),
    'Omega_t': ('cells of group 2', 'cell', None, {'vertices_from': 'Omega_'}),
    'Left': ('vertices in (x < 0.01)', 'facet'),
    'Right': ('vertices in (x > 0.69)', 'facet'),
    'Top_': ('vertices in (z > 0.09)', 'facet'),
    'Bottom': ('vertices in (z < 0.01)', 'facet'),
    'Edge': ('r.Bottom *v r.Right', 'vertex'),
    'Top': ('r.Top_ *f r.Omega', 'facet'),
    'Top_t': ('r.Top_ *f r.Omega_t', 'facet'),
}

functions = {
    'get_force' : (get_force,),
}

materials = {
    'solid': ({'D': stiffness_from_youngpoisson(dim=3, young=1e6, poisson=0.3)},),
    'force': 'get_force',
    'vforce': ({'.val' : [0.0, 0.0, -7]}),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
    'displacement_t': ('real', 'vector', 'Omega_t', 1),
}

integrals = {
    'i': 2,
}

variables = {
    'u': ('unknown field', 'displacement', 0),

```

(continues on next page)

(continued from previous page)

```

    'v': ('test field', 'displacement', 'u'),
    'u_t': ('unknown field', 'displacement_t', 0),
    'v_t': ('test field', 'displacement_t', 'u_t'),
}

ebcs = {
    'Fixed': ('Left', {'u.all' : 0.0}),
}

equations = {
    'balance_of_forces':
        """dw_lin_elastic.i.Omega(solid.D, v, u)
        + dw_lin_elastic.i.Omega_t(solid.D, v_t, u_t)
        = dw_surface_ltr.i.Top(force.val, v)
        + dw_surface_ltr.i.Top_t(force.val, v_t)"""
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-6,
    }),
}

```

linear_elasticity/modal_analysis.py

Description

Modal analysis of a linear elastic block in 2D or 3D.

The dimension of the problem is determined by the length of the vector in `--dims` option.

Optionally, a mesh file name can be given as a positional argument. In that case, the mesh generation options are ignored.

The default material properties correspond to aluminium in the following units:

- length: m
- mass: kg
- stiffness / stress: Pa
- density: kg / m³

Examples

- Run with the default arguments:

```
python sfepy/examples/linear_elasticity/modal_analysis.py
```

- Fix bottom surface of the domain:

```
python sfepy/examples/linear_elasticity/modal_analysis.py -b cantilever
```

- Increase mesh resolution:

```
python sfepy/examples/linear_elasticity/modal_analysis.py -s 31,31
```

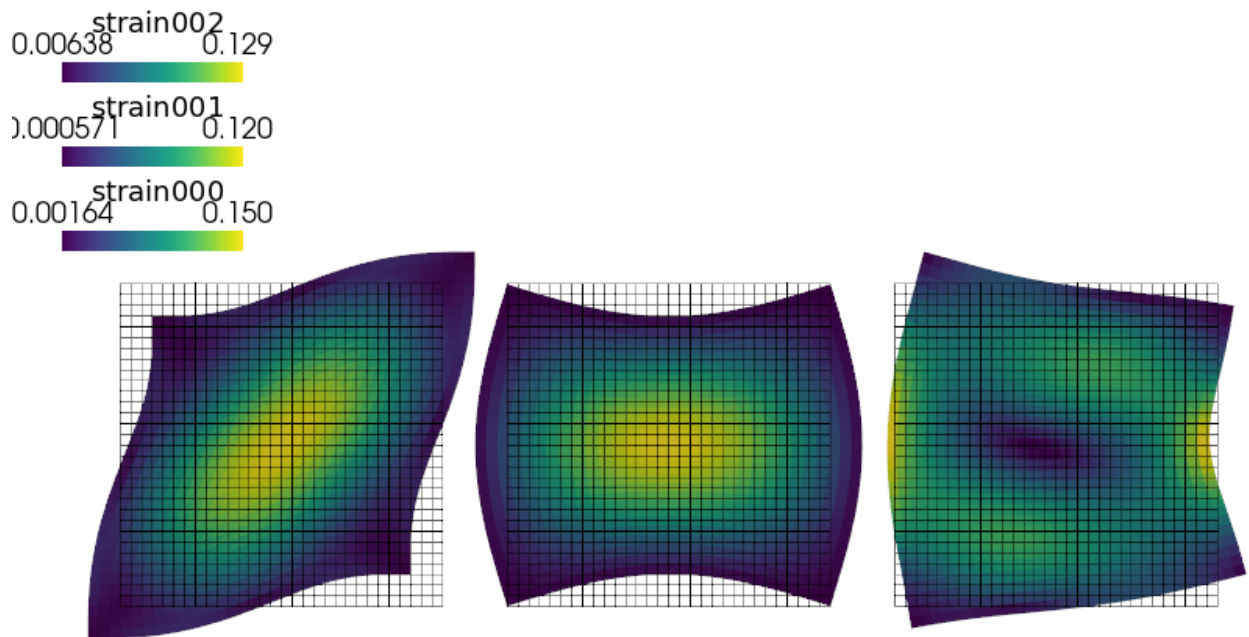
- Use 3D domain:

```
python sfepy/examples/linear_elasticity/modal_analysis.py -d 1,1,1 -c 0,0,0 -s 8,8,8
```

- Change the eigenvalue problem solver to LOBPCG:

```
python sfepy/examples/linear_elasticity/modal_analysis.py --solver="eig.scipy_lobpcg,i_max:100,largest:False"
```

See `sfepy.solvers.eigen` for available solvers.



source code

```
#!/usr/bin/env python
"""
Modal analysis of a linear elastic block in 2D or 3D.

The dimension of the problem is determined by the length of the vector
in ``--dims`` option.

Optionally, a mesh file name can be given as a positional argument. In that
case, the mesh generation options are ignored.

The default material properties correspond to aluminium in the following units:

- length: m
- mass: kg
- stiffness / stress: Pa
- density: kg / m^3

Examples
-----

- Run with the default arguments::

    python sfepy/examples/linear_elasticity/modal_analysis.py

- Fix bottom surface of the domain::

    python sfepy/examples/linear_elasticity/modal_analysis.py -b cantilever

- Increase mesh resolution::

    python sfepy/examples/linear_elasticity/modal_analysis.py -s 31,31

- Use 3D domain::

    python sfepy/examples/linear_elasticity/modal_analysis.py -d 1,1,1 -c 0,0,0 -s 8,8,8

- Change the eigenvalue problem solver to LOBPCG::

    python sfepy/examples/linear_elasticity/modal_analysis.py --solver="eig.scipy_lobpcg,
    ↪ i_max:100,largest:False"

    See :mod:`sfepy.solvers.eigen` for available solvers.
"""
from __future__ import absolute_import
import sys
import six
from six.moves import range
sys.path.append('.')
from argparse import ArgumentParser, RawDescriptionHelpFormatter

import numpy as nm
```

(continues on next page)

(continued from previous page)

```

import scipy.sparse.linalg as sla

from sfepy.base.base import assert_, output, Struct
from sfepy.discrete import (FieldVariable, Material, Integral, Integrals,
                             Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.solvers import Solver

helps = {
    'dims' :
        'dimensions of the block [default: %(default)s]',
    'centre' :
        'centre of the block [default: %(default)s]',
    'shape' :
        'numbers of vertices along each axis [default: %(default)s]',
    'bc_kind' :
        'kind of Dirichlet boundary conditions on the bottom and top surfaces,'
        ' one of: free, cantilever, fixed [default: %(default)s]',
    'axis' :
        'the axis index of the block that the bottom and top surfaces are related'
        ' to [default: %(default)s]',
    'young' : "the Young's modulus [default: %(default)s]",
    'poisson' : "the Poisson's ratio [default: %(default)s]",
    'density' : "the material density [default: %(default)s]",
    'order' : 'displacement field approximation order [default: %(default)s]',
    'n_eigs' : 'the number of eigenvalues to compute [default: %(default)s]',
    'ignore' : 'if given, the number of eigenvalues to ignore (e.g. rigid'
        ' body modes); has precedence over the default setting determined by'
        ' --bc-kind [default: %(default)s]',
    'solver' : 'the eigenvalue problem solver to use. It should be given'
        ' as a comma-separated list: solver_kind,option0:value0,option1:value1,...'
        ' [default: %(default)s]',
}

def main():
    parser = ArgumentParser(description=__doc__,
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--version', action='version', version='%(prog)s')
    parser.add_argument('-d', '--dims', metavar='dims',
                        action='store', dest='dims',
                        default='[1.0, 1.0]', help=helps['dims'])
    parser.add_argument('-c', '--centre', metavar='centre',
                        action='store', dest='centre',
                        default='[0.0, 0.0]', help=helps['centre'])
    parser.add_argument('-s', '--shape', metavar='shape',
                        action='store', dest='shape',
                        default='[11, 11]', help=helps['shape'])
    parser.add_argument('-b', '--bc-kind', metavar='kind',

```

(continues on next page)

(continued from previous page)

```

        action='store', dest='bc_kind',
        choices=['free', 'cantilever', 'fixed'],
        default='free', help=helps['bc_kind'])
parser.add_argument('-a', '--axis', metavar='0, ..., dim, or -1',
                    type=int, action='store', dest='axis',
                    default=-1, help=helps['axis'])
parser.add_argument('--young', metavar='float', type=float,
                    action='store', dest='young',
                    default=6.80e+10, help=helps['young'])
parser.add_argument('--poisson', metavar='float', type=float,
                    action='store', dest='poisson',
                    default=0.36, help=helps['poisson'])
parser.add_argument('--density', metavar='float', type=float,
                    action='store', dest='density',
                    default=2700.0, help=helps['density'])
parser.add_argument('--order', metavar='int', type=int,
                    action='store', dest='order',
                    default=1, help=helps['order'])
parser.add_argument('-n', '--n-eigs', metavar='int', type=int,
                    action='store', dest='n_eigs',
                    default=6, help=helps['n_eigs'])
parser.add_argument('-i', '--ignore', metavar='int', type=int,
                    action='store', dest='ignore',
                    default=None, help=helps['ignore'])
parser.add_argument('--solver', metavar='solver', action='store',
                    dest='solver',
                    default= \
                    "eig.scipy,method:'eigsh',tol:1e-5,maxiter:1000",
                    help=helps['solver'])
parser.add_argument('filename', nargs='?', default=None)
options = parser.parse_args()

aux = options.solver.split(',')
kwargs = {}
for option in aux[1:]:
    key, val = option.split(':')
    kwargs[key.strip()] = eval(val)
eig_conf = Struct(name='evp', kind=aux[0], **kwargs)

output('using values:')
output("  Young's modulus:", options.young)
output("  Poisson's ratio:", options.poisson)
output('  density:', options.density)
output('displacement field approximation order:', options.order)
output('requested %d eigenvalues' % options.n_eigs)
output('using eigenvalue problem solver:', eig_conf.kind)
output.level += 1
for key, val in six.iteritems(kwargs):
    output('%s: %r' % (key, val))
output.level -= 1

assert_((0.0 < options.poisson < 0.5),

```

(continues on next page)

(continued from previous page)

```

        "Poisson's ratio must be in ]0, 0.5[!")
    assert_((0 < options.order),
            'displacement approximation order must be at least 1!')

    filename = options.filename
    if filename is not None:
        mesh = Mesh.from_file(filename)
        dim = mesh.dim
        dims = nm.diff(mesh.get_bounding_box(), axis=0)

    else:
        dims = nm.array(eval(options.dims), dtype=nm.float64)
        dim = len(dims)

        centre = nm.array(eval(options.centre), dtype=nm.float64)[:dim]
        shape = nm.array(eval(options.shape), dtype=nm.int32)[:dim]

        output('dimensions:', dims)
        output('centre:      ', centre)
        output('shape:         ', shape)

        mesh = gen_block_mesh(dims, shape, centre, name='mesh')

    output('axis:          ', options.axis)
    assert_((-dim <= options.axis < dim), 'invalid axis value!')

    eig_solver = Solver.any_from_conf(eig_conf)

    # Build the problem definition.
    domain = FEDomain('domain', mesh)

    bbox = domain.get_mesh_bounding_box()
    min_coor, max_coor = bbox[:, options.axis]
    eps = 1e-8 * (max_coor - min_coor)
    ax = 'xyz'[:dim][options.axis]

    omega = domain.create_region('Omega', 'all')
    bottom = domain.create_region('Bottom',
                                  'vertices in (%s < %.10f)'
                                  % (ax, min_coor + eps),
                                  'facet')
    bottom_top = domain.create_region('BottomTop',
                                       'r.Bottom +v vertices in (%s > %.10f)'
                                       % (ax, max_coor - eps),
                                       'facet')

    field = Field.from_args('fu', nm.float64, 'vector', omega,
                           approx_order=options.order)

    u = FieldVariable('u', 'unknown', field)
    v = FieldVariable('v', 'test', field, primary_var_name='u')

```

(continues on next page)

(continued from previous page)

```

mtx_d = stiffness_from_youngpoisson(dim, options.young, options.poisson)

m = Material('m', D=mtx_d, rho=options.density)

integral = Integral('i', order=2*options.order)

t1 = Term.new('dw_lin_elastic(m.D, v, u)', integral, omega, m=m, v=v, u=u)
t2 = Term.new('dw_dot(m.rho, v, u)', integral, omega, m=m, v=v, u=u)
eq1 = Equation('stiffness', t1)
eq2 = Equation('mass', t2)
lhs_eqs = Equations([eq1, eq2])

pb = Problem('modal', equations=lhs_eqs)

if options.bc_kind == 'free':
    pb.time_update()
    n_rbm = dim * (dim + 1) // 2

elif options.bc_kind == 'cantilever':
    fixed = EssentialBC('Fixed', bottom, {'u.all' : 0.0})
    pb.time_update(ebcs=Conditions([fixed]))
    n_rbm = 0

elif options.bc_kind == 'fixed':
    fixed = EssentialBC('Fixed', bottom_top, {'u.all' : 0.0})
    pb.time_update(ebcs=Conditions([fixed]))
    n_rbm = 0

else:
    raise ValueError('unsupported BC kind! (%s)' % options.bc_kind)

if options.ignore is not None:
    n_rbm = options.ignore

pb.update_materials()

# Assemble stiffness and mass matrices.
mtx_k = eq1.evaluate(mode='weak', dw_mode='matrix', asm_obj=pb.mtx_a)
mtx_m = mtx_k.copy()
mtx_m.data[:] = 0.0
mtx_m = eq2.evaluate(mode='weak', dw_mode='matrix', asm_obj=mtx_m)

try:
    eigs, svecs = eig_solver(mtx_k, mtx_m, options.n_eigs + n_rbm,
                             eigenvectors=True)

except sla.ArpackNoConvergence as ee:
    eigs = ee.eigenvalues
    svecs = ee.eigenvectors
    output('only %d eigenvalues converged!' % len(eigs))

output('%d eigenvalues converged (%d ignored as rigid body modes)' %

```

(continues on next page)

(continued from previous page)

```

        (len(eigs), n_rbm))

eigs = eigs[n_rbm:]
svecs = svecs[:, n_rbm:]

omegas = nm.sqrt(eigs)
freqs = omegas / (2 * nm.pi)

output('number |          eigenvalue | angular frequency '
       '|          frequency')
for ii, eig in enumerate(eigs):
    output('%6d | %17.12e | %17.12e | %17.12e'
           % (ii + 1, eig, omegas[ii], freqs[ii]))

# Make full eigenvectors (add DOFs fixed by boundary conditions).
variables = pb.set_default_state()

vecs = nm.empty((variables.di.n_dof_total, svecs.shape[1]),
                dtype=nm.float64)
for ii in range(svecs.shape[1]):
    vecs[:, ii] = variables.make_full_vec(svecs[:, ii])

# Save the eigenvectors.
out = {}
for ii in range(eigs.shape[0]):
    variables.set_state(vecs[:, ii])
    aux = variables.create_output()
    strain = pb.evaluate('ev_cauchy_strain.i.Omega(u)',
                        integrals=Integrals([integral]),
                        mode='el_avg', verbose=False)
    out['u%03d' % ii] = aux.popitem()[1]
    out['strain%03d' % ii] = Struct(mode='cell', data=strain)

pb.save_state('eigenshapes.vtk', out=out)
pb.save_regions_as_groups('regions')

if __name__ == '__main__':
    main()

```

linear_elasticity/modal_analysis_declarative.py

Description

Modal analysis of a wheel set.

The first six modes are the rigid body modes because no boundary conditions are applied.

Running the simulation:

```
sfePy-run sfePy/examples/linear_elasticity/modal_analysis_declarative.py
```

The eigenvalues are saved to wheelset_eigs.txt and the eigenvectors to wheelset.vtk. View the results using:

```
sfepy-view wheelset.vtk -f u003:wu003:f30%:p0 1:vw:p0
```

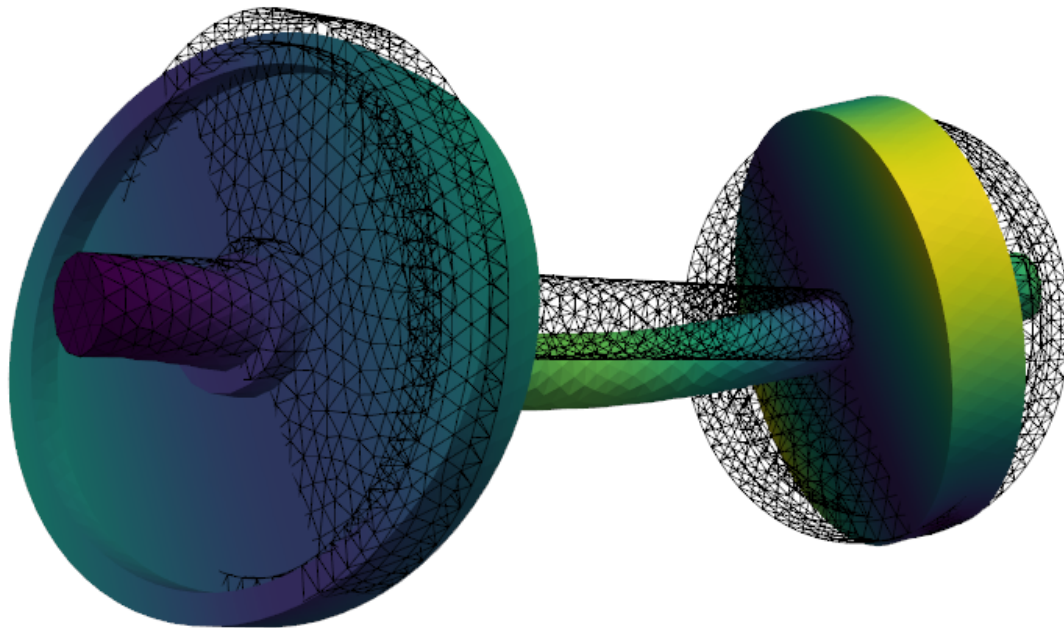
The first six frequencies calculated by SfePy:

```
[11.272, 11.322, 34.432, 80.711, 80.895, 93.149]
```

The results of modal analysis performed in Ansys:

```
[11.306, 11.316, 34.486, 80.901, 81.139, 93.472]
```

0.00 u003 0.0604



source code

```
"""
Modal analysis of a wheel set.

The first six modes are the rigid body modes because no boundary
conditions are applied.

Running the simulation::

    sfepy-run sfepy/examples/linear_elasticity/modal_analysis_declarative.py

The eigenvalues are saved to wheelset_eigs.txt and the eigenvectors to
```

(continues on next page)

(continued from previous page)

wheelset.vtk. View the results using::

```
sfePy-view wheelset.vtk -f u003:wu003:f30%p0 1:vw:p0
```

The first six frequencies calculated by SfePy::

```
[11.272, 11.322, 34.432, 80.711, 80.895, 93.149]
```

The results of modal analysis performed in Ansys::

```
[11.306, 11.316, 34.486, 80.901, 81.139, 93.472]
```

```
"""
```

```
import numpy as nm
from sfePy.base.base import output
from sfePy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfePy import data_dir
```

```
def report_eigs(pb, evp):
    eigs = evp.eigs
    n_rbm = evp.pb.conf.n_rbm

    output('%d eigenvalues converged (%d ignored as rigid body modes)' %
           (len(eigs), n_rbm))

    eigs = eigs[n_rbm:]

    omegas = nm.sqrt(eigs)
    freqs = omegas / (2 * nm.pi)

    output('number |          eigenvalue |  angular frequency '
           '|          frequency')
    for ii, eig in enumerate(eigs):
        output('%6d | %17.12e | %17.12e | %17.12e'
               % (ii + 1, eig, omegas[ii], freqs[ii]))
```

```
def define(n_eigs=6, approx_order=1, density=7850., young=210e9, poisson=0.3):
    filename_mesh, dim = data_dir + '/meshes/3d/wheelset.vtk', 3

    n_rbm = 0

    options = {
        'n_eigs': n_eigs + n_rbm,
        'eigs_only': False,
        'post_process_hook_final': 'report_eigs',
        'evps': 'eig',
    }

    regions = {
        'Omega': 'all',
        'Fix': ('vertices in (x < -1.08999)', 'vertex'),
```

(continues on next page)

(continued from previous page)

```

}

materials = {
    'm': ({
        'D': stiffness_from_youngpoisson(dim, young, poisson),
        'rho': density,
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', approx_order),
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
}

integrals = {
    'i': 2 * approx_order,
}

equations = {
    'lhs': 'dw_lin_elastic.i.Omega(m.D, v, u)',
    'rhs': 'dw_dot.i.Omega(m.rho, v, u)',
}

ebcs = {
    'fix': ('Fix', {'u.all': 0.0}) # fix rigid body modes
}

solvers = {
    # 'eig': ('eig.matlab', {
    #     'method': 'eigs',
    #     'which': 'sm',
    #     'eps': 1e-6,
    # }),
    # 'eig': ('eig.primme', {
    #     'which': 'SM',
    #     'tol': 1e-8,
    # }),
    'eig': ('eig.scipy', {
        'method': 'eigsh',
        'which': 'LM',
        'sigma': 0,
        'tol': 1e-8,
        'linear_solver': ('ls.scipy_superlu', {}),
        # 'linear_solver': ('ls.cholesky', {}),
        # 'linear_solver': ('ls.mumps', {}),
        # 'ls.cholesky' and 'ls.mumps' linear solvers are much faster,
        # but scikit-sparse package and MUMPS library are required
        'maxiter': 1000,
    }),
}

```

(continues on next page)

(continued from previous page)

```
    }},  
}  
  
return locals()
```

linear_elasticity/multi_node_lcbcs.py

Description

Linear elasticity with multi node linear combination constraints.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0, \quad \forall \underline{v},$$

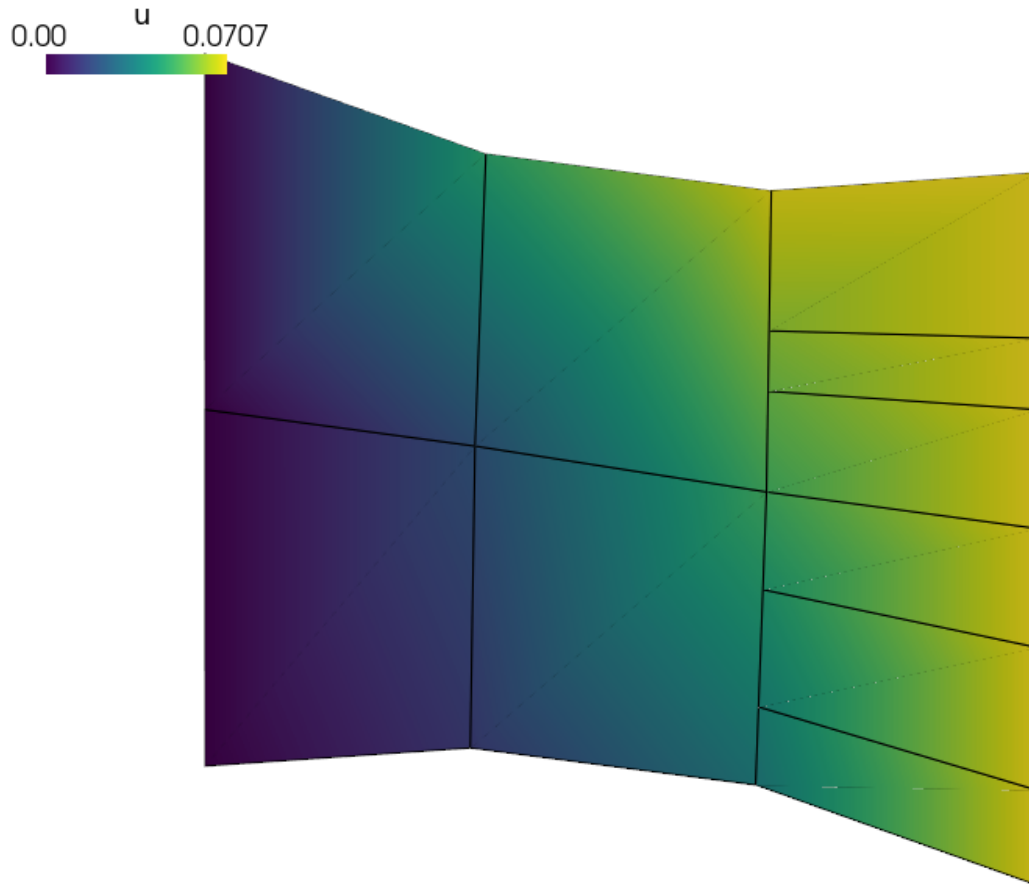
where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

The hanging nodes in 'Dependent_nodes' region are bounded to the nodes in 'Independent_nodes' region using the lcbcs` (``multi_node_combination) conditions

View the results using:

```
sfepy-view hanging_nodes.vtk -f u:wu:e -2
```



source code

```
# -*- coding: utf-8 -*-
r"""
Linear elasticity with multi node linear combination constraints.

Find :math:\ul{u} such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

The hanging nodes in ``Dependent_nodes`` region are bounded
to the nodes in ``Independent_nodes`` region using the ``lcbcs``
(``multi_node_combination``) conditions
```

(continues on next page)

(continued from previous page)

View the results using::

```

    sfepy-view hanging_nodes.vtk -f u:wu:e -2
    """
import numpy as nm
from sfepy import data_dir
from sfepy.base.base import output
from sfepy.mechanics.matcoefs import stiffness_from_lame

def node_map_fun(coors0, coors1):
    """
    Map hanging nodes (coors0) and coincident edges (edge nodes in coors1)
    on a line.
    """

    ldir = coors1[-1] - coors1[0]
    ldir = ldir / nm.linalg.norm(ldir)
    if ldir[0] > 1e-9:
        par0 = coors0[:, 0] / ldir[0]
        par1 = coors1[:, 0] / ldir[0]
    else:
        par0 = coors0[:, 1] / ldir[1]
        par1 = coors1[:, 1] / ldir[1]

    idxs1 = nm.argsort(par1)
    nmap = []
    for par in par0:
        edge = nm.where(nm.logical_and(par1[:-1] < par, par < par1[1:]))[0][0]
        de = par1[edge + 1] - par1[edge]
        nmap.append((idxs1[edge], idxs1[edge + 1],
                     1 - (par - par1[edge]) / de,
                     1 - (par1[edge + 1] - par) / de))

    return (nm.arange(par0.shape[0]),
            nm.array([k[:2] for k in nmap]),
            nm.array([k[2:] for k in nmap]))

def check_lcs(out, pb, state, extend=None):
    displ = state.get_state_parts()['u'].reshape((-1, 2))
    coors = pb.domain.mesh.coors

    hnodes_map = {12: [2, 6], 14: [2, 6], 16: [6, 10], 18: [6, 10]}

    ok = True
    for nd, ends in hnodes_map.items():
        dy = coors[ends[1], 1] - coors[ends[0], 1]
        t = (coors[nd, 1] - coors[ends[0], 1]) / dy
        u_exp = displ[ends[0]] * (1 - t) + displ[ends[1]] * t
        ok = ok and (nm.linalg.norm(displ[nd] - u_exp) < 1e-9)

```

(continues on next page)

(continued from previous page)

```

result = {True: 'passed', False: 'failed'}[ok]
output(f'multi node linear condition test {result}!')

# tests_lcbcs hook:
nls_status = pb.get_solver().status.nls_status
if hasattr(nls_status, 'conditions'):
    nls_status.conditions.append(int(not ok))

return out

filename_mesh = data_dir + '/meshes/2d/special/hanging_nodes.mesh'

options = {
    'post_process_hook': check_lcs,
}

regions = {
    'Omega': 'all',
    'Omega1': 'cells of group 1',
    'Omega2': 'cells of group 2',
    'Inside_nodes': ('vertices in (x > 0.199) & (x < 0.201)', 'vertex'),
    'Independent_nodes': ('r.Inside_nodes *v r.Omega1', 'vertex'),
    'Dependent_nodes': ('r.Inside_nodes -v r.Independent_nodes', 'vertex'),
    'Left': ('vertices in (x < 0.001)', 'facet'),
    'Right': ('vertices in (x > 0.299)', 'facet'),
}

materials = {
    'solid': ({'D': stiffness_from_lame(dim=2, lam=1e1, mu=1e0)},),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i': 1,
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
}

ebcs = {
    'Fixed': ('Left', {'u.all': 0.0}),
    'Displaced': ('Right', {'u.0': 0.05, 'u.1': -0.05}),
}

functions = {

```

(continues on next page)

(continued from previous page)

```

    'node_map_fun': (node_map_fun,),
}

lcbcs = {
    'hanging': ([ 'Dependent_nodes', 'Independent_nodes'], {'u.all': 'u.all'},
                'node_map_fun', 'multi_node_combination', None),
}

equations = {
    'balance_of_forces' :
    """dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-10,
    }),
}

```

linear_elasticity/multi_point_constraints.py

Description

The linear elasticity of two discs connected using multi-point constraints.

Find $\underline{u}$, $\underline{u}_c$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \sum_{i,j \in \Omega_c} K_{kl} ((u_c)_l^{(j)} - (u_c)_l^{(i)}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

$$K_{kl} = K_{kl}(\underline{d}, \underline{k}),$$

$u_c^{(j)}$ are the DOFs - two displacements and one rotation in 2D - of the node j in Ω_c , which is a subdomain composed of two 1D spring terms with the generalized stiffness matrix K_{kl} depending on the directions $\underline{d}$ of the springs and the stiffness vector $\underline{k}$.

The deformation is governed by the Dirichlet conditions applied to one of the spring end points, see the *dofs* argument of `define()` below.

Usage Examples

- Save and display boundary regions:

```
sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py --save-
↪regions-as-groups --solve-not

sfepy-view annulus-c_regions.vtk -2e -f Gamma1:p0 Gamma2:p1 Gamma3:p3 Gamma4:p4 --
↪max-plots=4 --color-map=summer
```

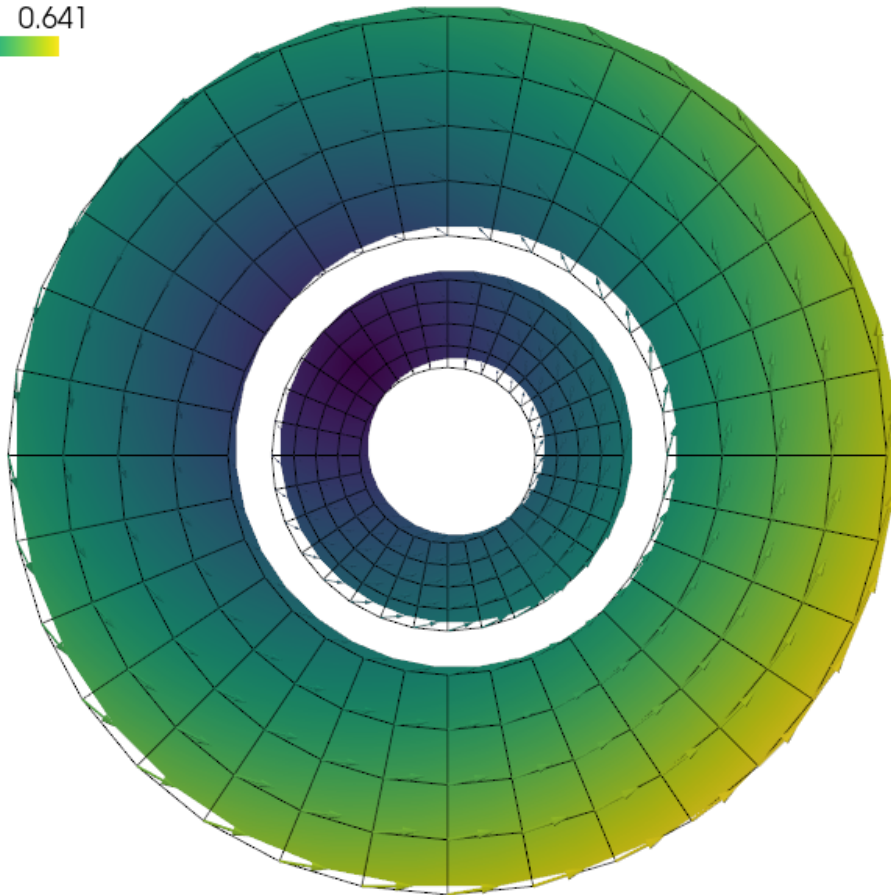
- Run:

```
sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py
sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=(0,1)
↪"
sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=(0,
↪1), is_rot=False"
sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=2"
```

- Display results:

```
sfepy-view annulus-c.vtk -2e
sfepy-view annulus-c.vtk -2e -f u:wu:f1:p0 1:vw:p0 u:gu:p0
```

0.00 **u** 0.641

source code

```

r"""
The linear elasticity of two discs connected using multi-point constraints.

Find  $\mathbf{u}$ ,  $\mathbf{u}_c$  such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
    + \sum_{i,j \in \Omega_c} K_{kl} ((u_c)^{(j)}_l - (u_c)^{(i)}_l)
    = 0
    \quad \forall \mathbf{v};

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl};

    K_{kl} = K_{kl}(\mathbf{d}, \mathbf{k});

 $(u_c)^{(j)}$  are the DOFs - two displacements and one rotation in 2D -
of the node  $j$  in  $\Omega_c$ , which is a subdomain composed of two
1D spring terms with the generalized stiffness matrix  $K_{kl}$  depending
on the directions  $\mathbf{d}$  of the springs and the stiffness vector
 $\mathbf{k}$ .

The deformation is governed by the Dirichlet conditions applied to one of the
spring end points, see the dofs argument of define() below.

Usage Examples
-----

- Save and display boundary regions::

    sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py --save-regions-
    ↪as-groups --solve-not

    sfepy-view annulus-c_regions.vtk -2e -f Gamma1:p0 Gamma2:p1 Gamma3:p3 Gamma4:p4 --
    ↪max-plots=4 --color-map=summer

- Run::

    sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py
    sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=(0,1)"
    sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=(0,1),
    ↪is_rot=False"
    sfepy-run sfepy/examples/linear_elasticity/multi_point_constraints.py -d "dofs=2"

- Display results::

    sfepy-view annulus-c.vtk -2e
    sfepy-view annulus-c.vtk -2e -f u:wu:f1:p0 1:vw:p0 u:gu:p0

```

(continues on next page)

(continued from previous page)

```

"""
import numpy as nm

from sfepy.mesh.mesh_generators import gen_cylinder_mesh
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.discrete.fem.mesh import Mesh
from sfepy.linalg import get_coors_in_ball
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.mechanics.tensors import dim2sym

def get_weights(mcoors, scoors):
    n_nod, dim = scoors.shape
    out = nm.empty((n_nod, dim2sym(dim)))
    out[:] = nm.linspace(0, 10, n_nod)[:, None]
    return out

def define(dims=(1, 1, 2, 2, 0), shape=(5, 32, 0), order=1, dofs=(0, 1, 2),
           is_rot=True, output_dir='.'):
    if not isinstance(dofs, tuple):
        dofs = (dofs,)

    nuc = 2 + is_rot
    dofs0 = tuple(set(range(nuc)).difference(dofs))

    sdofs = ','.join([f'{ii}' for ii in dofs])
    sdofs0 = ','.join([f'{ii}' for ii in dofs0])

    def mesh_hook(mesh, mode):
        if mode == 'read':
            _mesh = gen_cylinder_mesh(dims, shape, (0, 0, 0),
                                      make_2d=True)
            _mesh2 = _mesh.copy()
            _mesh2.transform_coors(nm.eye(2, dtype=nm.float64) * 2.5)
            _mesh = _mesh + _mesh2

            # Add constraint vertices (and cells) to the base mesh.
            coors, vgroups, conns, mat_ids, descs = _mesh._get_io_data()
            nc = coors.shape[0]

            coors = nm.r_[coors, nm.zeros((4, 2), dtype=nm.float64)]
            vgroups = nm.r_[vgroups, [1, 2, 3, 4]]
            conns.append([[nc, nc + 1],
                         [nc + 2, nc + 3]])
            mat_ids.append([1, 2])
            descs.append('1_2')
            mesh = Mesh.from_data('annulus-c',
                                coors, vgroups, conns, mat_ids, descs)

            return mesh

        elif mode == 'write':
            pass

```

(continues on next page)

(continued from previous page)

```

filename_mesh = UserMeshIO(mesh_hook)

regions = {
    'Omega' : 'all',
    'Gamma1' : ('vertices by get_gamma1', 'facet'),
    'Gamma2' : ('vertices by get_gamma2', 'facet'),
    'Gamma3' : ('vertices by get_gamma3', 'facet'),
    'Gamma4' : ('vertices by get_gamma4', 'facet'),
    'Omega12C' : ('(vertices of group 1) +v (vertices of group 2)',
                  'cell', None, {'cell_tdim': 1}),
    'Gamma1C' : ('vertices of group 1', 'vertex'),
    'Gamma2C' : ('vertices of group 2', 'vertex'),
    'Omega34C' : ('(vertices of group 3) +v (vertices of group 4)',
                  'cell', None, {'cell_tdim': 1}),
    'Gamma3C' : ('vertices of group 3', 'vertex'),
    'Gamma4C' : ('vertices of group 4', 'vertex'),
    'OmegaC' : ('r.Omega12C +c r.Omega34C', 'cell', None, {'cell_tdim': 1})
}

centre = [0, 0]
functions = {
    'get_gamma1' : (lambda coors, domain:
                    get_coors_in_ball(coors, centre, 1.1, 0.9),),
    'get_gamma2' : (lambda coors, domain:
                    get_coors_in_ball(coors, centre, 2.1, 1.9),),
    'get_gamma3' : (lambda coors, domain:
                    get_coors_in_ball(coors, centre, 2.6, 2.4),),
    'get_gamma4' : (lambda coors, domain:
                    get_coors_in_ball(coors, centre, 5.1, 4.9),),
    'get_weights' : (get_weights,),
}

fields = {
    'fu' : ('real', 2, 'Omega', order),
    'fc' : ('real', nuc, 'OmegaC', 1),
}

variables = {
    'u' : ('unknown field', 'fu', 0),
    'v' : ('test field', 'fu', 'u'),
    'uc' : ('unknown field', 'fc', 1),
    'vc' : ('test field', 'fc', 'uc'),
}

ebcs = {
    'uc' : ('Gamma1C', {f'uc.[{sdofs}]' : 0.1},),
}
if dofs0:
    ebcs.update({
        'uc0' : ('Gamma1C', {f'uc.[{sdofs0}]' : 0.0},),
    })
if not is_rot:

```

(continues on next page)

(continued from previous page)

```

ebcs.update({
    'u0' : ('Gamma4', {'u.all' : 0.0}),
})

lcbcs = {
    'rigid' : (('Gamma1', 'Gamma2C'), {'u.all' : 'uc.all'},
              None, 'rigid2'),
    'avg1' : (('Gamma2', 'Gamma3C'), {'u.all' : 'uc.all'},
              'get_weights', 'average_force'),
    'avg2' : (('Gamma3', 'Gamma4C'), {'u.all' : 'uc.all'},
              None, 'average_force'),
}

materials = {
    'm' : ({
        'D' : stiffness_from_lame(dim=2, lam=1e1, mu=1e0),
        'ks' : [[1e+5], [1e+5], [1e+5]][:nuc],
        'dvec' : [[0.01], [0.01]],
    },),
}

integrals = {
    'i' : 2 * order,
}

if is_rot:
    equations = {
        'eq1' :
        """dw_lin_elastic.i.Omega(m.D, v, u)
        + dw_lin_dspring_rot.0.Omega12C(m.dvec, m.ks, vc, uc)
        + dw_lin_dspring_rot.0.Omega34C(m.dvec, m.ks, vc, uc)
        = 0
        """,
    }
else:
    equations = {
        'eq1' :
        """dw_lin_elastic.i.Omega(m.D, v, u)
        + dw_lin_dspring.0.Omega12C(m.dvec, m.ks, vc, uc)
        + dw_lin_dspring.0.Omega34C(m.dvec, m.ks, vc, uc)
        = 0
        """,
    }

solvers = {
    'ls' : ('ls.auto_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : -1e-10,
        'check' : 0,
    }),
}

```

(continues on next page)

(continued from previous page)

```

}

options = {
    'output_dir' : output_dir,
    'nls' : 'newton',
    'ls' : 'ls',
}

return locals()

```

linear_elasticity/nodal_lcbcs.py

Description

Linear elasticity with nodal linear combination constraints.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

and $\underline{\underline{\sigma}} \cdot \underline{n} = \bar{p} \underline{\underline{I}} \cdot \underline{n}$ with given traction pressure $\bar{p}$. The constraints are given in terms of coefficient matrices and right-hand sides, see the lcbcs keyword below. For instance, 'nlcbc1' in the 3D mesh case corresponds to

$$\begin{aligned} u_0 - u_1 + u_2 &= 0 \\ u_0 + 0.5u_1 + 0.1u_2 &= 0.05 \end{aligned}$$

that should hold in the 'Top' region.

This example demonstrates how to pass command line options to a problem description file using `--define` option of `sfePy-run`. Try:

```
sfePy-run sfePy/examples/linear_elasticity/nodal_lcbcs.py --define='dim: 3'
```

to use a 3D mesh, instead of the default 2D mesh. The example also shows that the nodal constraints can be used in place of the Dirichlet boundary conditions. Try:

```
sfePy-run sfePy/examples/linear_elasticity/nodal_lcbcs.py --define='use_ebcs: False'
```

to replace ebcs with the 'nlcbc4' constraints. The results should be the same for the two cases. Both options can be combined:

```
sfePy-run sfePy/examples/linear_elasticity/nodal_lcbcs.py --define='dim: 3, use_ebcs:
↪False'
```

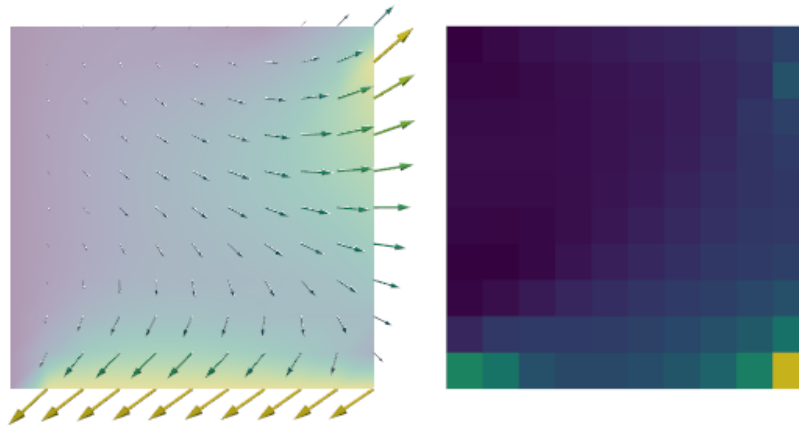
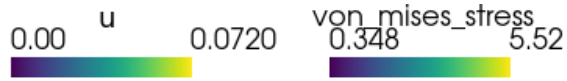
The `post_process()` function is used both to compute the von Mises stress and to verify the linear combination constraints.

View the 2D results using:


```
sfepy-view square_quad.vtk -2
```

View the 3D results using:

```
sfepy-view cube_medium_tetra.vtk
```



source code

```
r"""
Linear elasticity with nodal linear combination constraints.

Find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \, e_{ij}(\ul{v}) \, e_{kl}(\ul{u})
    = - \int_{\Gamma_{right}} \ul{v} \cdot \ull{\sigma} \cdot \ul{n}
    \;, \quad \text{forall } \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
```

(continues on next page)

(continued from previous page)

```
\;.
```

and $\sigma \cdot n = \bar{p} I \cdot n$ with given traction pressure $\bar{p}$. The constraints are given in terms of coefficient matrices and right-hand sides, see the `lcbcs` keyword below. For instance, `nlcbc1` in the 3D mesh case corresponds to

```
.. math::
    u_0 - u_1 + u_2 = 0 \ \
    u_0 + 0.5 u_1 + 0.1 u_2 = 0.05
```

that should hold in the `Top` region.

This example demonstrates how to pass command line options to a problem description file using `--define` option of `sfepy-run`. Try::

```
sfepy-run sfepy/examples/linear_elasticity/nodal_lcbcs.py --define='dim: 3'
```

to use a 3D mesh, instead of the default 2D mesh. The example also shows that the nodal constraints can be used in place of the Dirichlet boundary conditions. Try::

```
sfepy-run sfepy/examples/linear_elasticity/nodal_lcbcs.py --define='use_ebcs: False'
```

to replace `ebcs` with the `nlcbc4` constraints. The results should be the same for the two cases. Both options can be combined::

```
sfepy-run sfepy/examples/linear_elasticity/nodal_lcbcs.py --define='dim: 3, use_ebcs:
False'
```

The `post_process()` function is used both to compute the von Mises stress and to verify the linear combination constraints.

View the 2D results using::

```
sfepy-view square_quad.vtk -2
```

View the 3D results using::

```
sfepy-view cube_medium_tetra.vtk
"""
```

```
from __future__ import absolute_import
import numpy as nm
```

```
from sfepy.base.base import output, assert_
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.mechanics.tensors import get_von_mises_stress
from sfepy import data_dir
```

```
def post_process(out, pb, state, extend=False):
```

```
    """
```

```
    Calculate and output strain and stress for given displacements.
```

(continues on next page)

(continued from previous page)

```

"""
from sfepy.base.base import Struct

ev = pb.evaluate
stress = ev('ev_cauchy_stress.2.Omega(m.D, u)', mode='el_avg')

vms = get_von_mises_stress(stress.squeeze())
vms.shape = (vms.shape[0], 1, 1, 1)
out['von_mises_stress'] = Struct(name='output_data', mode='cell',
                                data=vms, dofs=None)

dim = pb.domain.shape.dim

us = state().reshape((-1, dim))

field = pb.fields['displacement']

if dim == 2:
    ii = field.get_dofs_in_region(pb.domain.regions['Top'])
    output('top LCBC (u.0 - u.1 = 0):')
    output('\n', nm.c_[us[ii], nm.diff(us[ii], 1)])

    ii = field.get_dofs_in_region(pb.domain.regions['Bottom'])
    output('bottom LCBC (u.0 + u.1 = -0.1):')
    output('\n', nm.c_[us[ii], nm.sum(us[ii], 1)])

    ii = field.get_dofs_in_region(pb.domain.regions['Right'])
    output('right LCBC (u.0 + u.1 = linspace(0, 0.1)):')
    output('\n', nm.c_[us[ii], nm.sum(us[ii], 1)])

else:
    ii = field.get_dofs_in_region(pb.domain.regions['Top'])
    output('top LCBC (u.0 - u.1 + u.2 = 0):')
    output('\n', nm.c_[us[ii], us[ii, 0] - us[ii, 1] + us[ii, 2]])
    output('top LCBC (u.0 + 0.5 u.1 + 0.1 u.2 = 0.05):')
    output('\n', nm.c_[us[ii],
                        us[ii, 0] + 0.5 * us[ii, 1] + 0.1 * us[ii, 2]])

    ii = field.get_dofs_in_region(pb.domain.regions['Bottom'])
    output('bottom LCBC (u.2 - 0.1 u.1 = 0.2):')
    output('\n', nm.c_[us[ii], us[ii, 2] - 0.1 * us[ii, 1]])

    ii = field.get_dofs_in_region(pb.domain.regions['Right'])
    output('right LCBC (u.0 + u.1 + u.2 = linspace(0, 0.1)):')
    output('\n', nm.c_[us[ii], nm.sum(us[ii], 1)])

return out

def define(dim=2, use_ebcs=True):
    assert_(dim in (2, 3))

    if dim == 2:

```

(continues on next page)

(continued from previous page)

```

    filename_mesh = data_dir + '/meshes/2d/square_quad.mesh'

else:
    filename_mesh = data_dir + '/meshes/3d/cube_medium_tetra.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process'
}

def get_constraints(ts, coors, region=None):
    mtx = nm.ones((coors.shape[0], 1, dim), dtype=nm.float64)

    rhs = nm.arange(coors.shape[0], dtype=nm.float64)[: , None]

    rhs *= 0.1 / (coors.shape[0] - 1)

    return mtx, rhs

functions = {
    'get_constraints' : (get_constraints,),
}

fields = {
    'displacement': ('real', dim, 'Omega', 1),
}

materials = {
    'm' : ({
        'D' : stiffness_from_lame(dim, lam=5.769, mu=3.846),
    },),
    'load' : ({'val' : -1.0},),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

regions = {
    'Omega' : 'all',
    'Bottom' : ('vertices in (y < -0.499) -v r.Left', 'facet'),
    'Top' : ('vertices in (y > 0.499) -v r.Left', 'facet'),
    'Left' : ('vertices in (x < -0.499)', 'facet'),
    'Right' : ('vertices in (x > 0.499) -v (r.Bottom +v r.Top)', 'facet'),
}

if dim == 2:
    lcbcs = {
        'nlcbc1' : ('Top', {'u.all' : None}, None, 'nodal_combination',
                    ([[1.0, -1.0]], [0.0])),

```

(continues on next page)

(continued from previous page)

```

        'nlcbc2' : ('Bottom', {'u.all' : None}, None, 'nodal_combination',
                      ([[1.0, 1.0]], [-0.1])),
        'nlcbc3' : ('Right', {'u.all' : None}, None, 'nodal_combination',
                      'get_constraints'),
    }

    else:
        lcbcs = {
            'nlcbc1' : ('Top', {'u.all' : None}, None, 'nodal_combination',
                          ([[1.0, -1.0, 1.0], [1.0, 0.5, 0.1]], [0.0, 0.05])),
            'nlcbc2' : ('Bottom', {'u.[2,1]' : None}, None, 'nodal_combination',
                          ([[1.0, -0.1]], [0.2])),
            'nlcbc3' : ('Right', {'u.all' : None}, None, 'nodal_combination',
                          'get_constraints'),
        }

    if use_ebcs:
        ebcs = {
            'fix' : ('Left', {'u.all' : 0.0}),
        }
    else:
        ebcs = {}

    lcbcs.update({
        'nlcbc4' : ('Left', {'u.all' : None}, None, 'nodal_combination',
                      (nm.eye(dim), nm.zeros(dim))),
    })

    equations = {
        'elasticity' : """
            dw_lin_elastic.2.Omega(m.D, v, u)
            = -dw_surface_ltr.2.Right(load.val, v)
            """
    }

    solvers = {
        'ls' : ('ls.scipy_direct', {}),
        'newton' : ('nls.newton', {
            'i_max' : 1,
            'eps_a' : 1e-10,
        }),
    }

    return locals()

```

linear_elasticity/prestress_fibres.py

Description

Linear elasticity with a given prestress in one subdomain and a (pre)strain fibre reinforcement in the other.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Omega_1} \sigma_{ij} e_{ij}(\underline{v}) + \int_{\Omega_2} D_{ijkl}^f e_{ij}(\underline{v}) (d_k d_l) = 0, \quad \forall \underline{v},$$

where

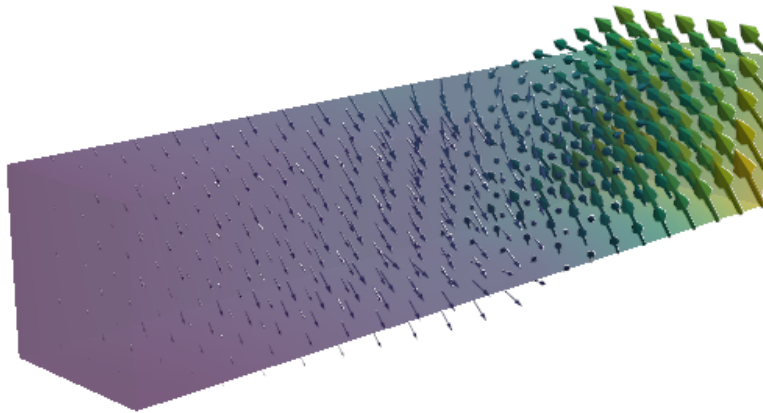
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

The stiffness of fibres D_{ijkl}^f is defined analogously, $\underline{d}$ is the unit fibre direction vector and σ_{ij} is the prestress.

Visualization

Use the following to see the deformed structure with 10x magnified displacements:

```
sfepy-view block.vtk -f u:wu:f5 1:vw
```



source code

```

r"""
Linear elasticity with a given prestress in one subdomain and a (pre)strain
fibre reinforcement in the other.

Find :math:`\ul{u}` such that:

.. math::
\int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
+ \int_{\Omega_1} \sigma_{ij} \ e_{ij}(\ul{v})
+ \int_{\Omega_2} D^f_{ijkl} \ e_{ij}(\ul{v}) \ \left(d_k \ d_l\right)
= 0
\;, \quad \forall \ul{v} \;,

where

.. math::
D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
\lambda \delta_{ij} \delta_{kl}
\;.

The stiffness of fibres :math:`D^f_{ijkl}` is defined analogously,
:math:`\ul{d}` is the unit fibre direction vector and :math:`\sigma_{ij}` is
the prestress.

Visualization
-----

Use the following to see the deformed structure with 10x magnified
displacements::

    sfepy-view block.vtk -f u:wu:f5 1:vw
"""
from __future__ import absolute_import
import numpy as nm
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/block.mesh'

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
    'Omega1' : 'vertices in (x < 0.001)',
    'Omega2' : 'vertices in (x > -0.001)',
}

materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=1e2, mu=1e1),
        'prestress' : 0.1 * nm.array([[1.0], [1.0], [1.0],
                                      [0.5], [0.5], [0.5]]),
                                     dtype=nm.float64),
        'DF' : stiffness_from_lame(3, lam=8e0, mu=8e-1),
    })

```

(continues on next page)

(continued from previous page)

```

        'nu' : nm.array([[ -0.5], [0.0], [0.5]], dtype=nm.float64),
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'Fixed' : ('Left', {'u.all' : 0.0}),
}

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.2.Omega( solid.D, v, u )
        + dw_lin_prestress.2.Omega1( solid.prestress, v )
        + dw_lin_strain_fib.2.Omega2( solid.DF, solid.nu, v )
        = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

```

linear_elasticity/seismic_load.py

Description

The linear elastodynamics of an elastic body loaded by a given base motion.

Find $\underline{u}$ such that:

$$\begin{aligned}
 \int_{\Omega} \rho \underline{v} \frac{\partial^2 \underline{u}}{\partial t^2} + \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) &= 0, \quad \forall \underline{v}, \\
 u_1(t) &= 10^{-5} \sin(\omega t) \sin(kx_2) \text{ on } \Gamma_{\text{Seismic}}, \\
 \omega &= c_L k,
 \end{aligned}$$

where c_L is the longitudinal wave propagation speed, $k = 2\pi/L$, L is the length of the domain and

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

See [linear_elasticity/elastodynamic.py](#) example for notes on elastodynamics solvers.

Usage Examples

Run with the default settings (the Newmark method, 2D problem, results stored in output/seismic/):

```
sfepy-run sfepy/examples/linear_elasticity/seismic_load.py -o tsn
```

View the resulting displacements on the deforming mesh (10x magnified):

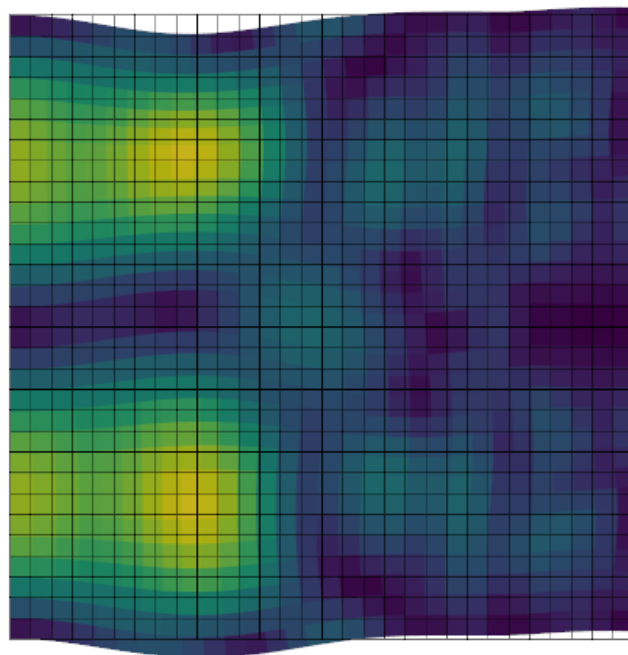
```
sfepy-view output/seismic/tsn.h5 -2 -f u:wu:f10:p0 1:vw:p0
```

Use the central difference explicit method with the reciprocal mass matrix algorithm¹ and view the resulting stress waves:

```
sfepy-run sfepy/examples/linear_elasticity/seismic_load.py -d "dims=(5e-3, 5e-3),  
↳ shape=(51, 51), tss_name=tscd, ls_name=lsrmm, mass_beta=0.5, mass_lumping=row_sum,  
↳ fast_rmm=True, save_times=all" -o tscd
```

```
sfepy-view output/seismic/tscd.h5 -2 -f cauchy_stress:wu:f10:p0 1:vw:p0
```

cauchy_stress
1.09e+07 -3.98e+09

source code

¹ González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018. Inverse mass matrix via the method of localized Lagrange multipliers. International Journal for Numerical Methods in Engineering 113, 277–295. <https://doi.org/10.1002/nme.5613>

```

r"""
The linear elastodynamics of an elastic body loaded by a given base motion.

Find :math:\mathbf{u} such that:

.. math::
\int_{\Omega} \rho \mathbf{v} \cdot \frac{d\mathbf{u}}{dt}
+ \int_{\Omega} \mathbf{D}_{ijkl} \mathbf{e}_{ij}(\mathbf{v}) \mathbf{e}_{kl}(\mathbf{u})
= 0
\;, \quad \forall \mathbf{v} \;, \quad \forall
u_1(t) = 10^{-5} \sin(\omega t) \sin(k x_2)
\quad \text{on } \Gamma_{\text{Seismic}} \;, \quad \forall
\omega = c_L k \;,

where :math:c_L is the longitudinal wave propagation speed, :math:k = 2 \pi / L,
:math:L is the length of the domain and

.. math::
\mathbf{D}_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
\lambda \delta_{ij} \delta_{kl}
\;.

See :ref:`linear_elasticity-elastodynamic` example for notes on elastodynamics
solvers.

Usage Examples
-----

Run with the default settings (the Newmark method, 2D problem, results stored
in ``output/seismic/``)::

    sfepy-run sfepy/examples/linear_elasticity/seismic_load.py -o tsn

View the resulting displacements on the deforming mesh (10x magnified)::

    sfepy-view output/seismic/tsn.h5 -2 -f u:wu:f10:p0 1:vw:p0

Use the central difference explicit method with the reciprocal mass matrix
algorithm [1]_ and view the resulting stress waves::

    sfepy-run sfepy/examples/linear_elasticity/seismic_load.py -d "dims=(5e-3, 5e-3),
↪ shape=(51, 51), tss_name=tscd, ls_name=lsrmm, mass_beta=0.5, mass_lumping=row_sum,
↪ fast_rmm=True, save_times=all" -o tscd

    sfepy-view output/seismic/tscd.h5 -2 -f cauchy_stress:wu:f10:p0 1:vw:p0

.. [1] González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018.
    Inverse mass matrix via the method of localized Lagrange multipliers.
    International Journal for Numerical Methods in Engineering 113, 277-295.
    https://doi.org/10.1002/nme.5613
"""
import numpy as nm

```

(continues on next page)

(continued from previous page)

```

import sfepy.mechanics.matcoefs as mc
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh

def define(
    E=200e9, nu=0.3, rho=7800,
    plane='strain',
    dims=(5e-3, 5e-3),
    shape=(31, 31),
    v0=1.0,
    ct1=1.5,
    dt=None,
    edt_safety=0.2,
    tss_name='tsn',
    tsc_name='tsced1',
    adaptive=False,
    ls_name='lsd',
    mass_beta=0.0,
    mass_lumping='none',
    fast_rmm=False,
    active_only=False,
    save_times=20,
    output_dir='output/seismic',
):
    """
    Parameters
    -----
    E, nu, rho: material parameters
    plane: plane strain or stress hypothesis
    dims: physical dimensions of the block (L, d, x)
    shape: numbers of mesh vertices along each axis
    v0: initial impact velocity
    ct1: final time in L / "longitudinal wave speed" units
    dt: time step (None means automatic)
    edt_safety: safety factor time step multiplier for explicit schemes,
               if dt is None
    tss_name: time stepping solver name (see "solvers" section)
    tsc_name: time step controller name (see "solvers" section)
    adaptive: use adaptive time step control
    ls_name: linear system solver name (see "solvers" section)
    mass_beta: averaged mass matrix parameter  $0 \leq \beta \leq 1$ 
    mass_lumping: mass matrix lumping ('row_sum', 'hrz' or 'none')
    fast_rmm: use zero inertia term with lsrmm
    save_times: number of solutions to save
    output_dir: output directory
    """
    dim = len(dims)

    lam, mu = mc.lame_from_youngpoisson(E, nu, plane=plane)
    # Longitudinal and shear wave propagation speeds.
    cl = nm.sqrt((lam + 2.0 * mu) / rho)
    cs = nm.sqrt(mu / rho)

```

(continues on next page)

(continued from previous page)

```

# Element size.
L, d = dims[:2]
H = L / (nm.max(shape) - 1)

# Time-stepping parameters.
if dt is None:
    # For implicit schemes, dt based on the Courant number C0 = dt * cl / H
    # equal to 1.
    dt = H / cl # C0 = 1
    if tss_name in ('tsvv', 'tscd'):
        # For explicit schemes, use a safety margin.
        dt *= edt_safety

t1 = ct1 * L / cl

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, 0.5 * nm.array(dims),
                              name='user_block', verbose=False)
        return mesh

    elif mode == 'write':
        pass

def post_process(out, problem, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    from sfepy.base.base import Struct

    ev = problem.evaluate
    strain = ev('ev_cauchy_strain.i.Omega(u)', mode='el_avg', verbose=False)
    stress = ev('ev_cauchy_stress.i.Omega(solid.D, u)', mode='el_avg',
               copy_materials=False, verbose=False)

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                 data=strain)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                 data=stress)

    return out

filename_mesh = UserMeshIO(mesh_hook)

regions = {
    'Omega' : 'all',
    'Seismic' : ('vertices in (x < 1e-12)', 'facet'),
}

```

(continues on next page)

(continued from previous page)

```

# Iron.
materials = {
    'solid' : ({
        'D': mc.stiffness_from_youngpoisson(dim=dim, young=E, poisson=nu,
                                           plane=plane),
        'rho': rho,
        '.lumping' : mass_lumping,
        '.beta' : mass_beta,
    },),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i' : 2,
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

def get_ebcs(ts, coors, mode='u'):
    y = coors[:, 1]
    amplitude = 0.00001
    k = 2 * nm.pi / L
    omega = cl * k
    if mode == 'u':
        val = amplitude * nm.sin(ts.time * omega) * nm.sin(k * y)

    elif mode == 'du':
        val = amplitude * omega * nm.cos(ts.time * omega) * nm.sin(k * y)

    elif mode == 'ddu':
        val = -amplitude * omega**2 * nm.sin(ts.time * omega) * nm.sin(k * y)

    return val

functions = {
    'get_u' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor),),
    'get_du' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='du'),),
    'get_ddu' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='ddu'),),
}

ebcs = {
    'Seismic' : ('Seismic', {'u.0' : 'get_u', 'du.0' : 'get_du',
                             'ddu.0' : 'get_ddu'}),
}

```

(continues on next page)

(continued from previous page)

```

ics = {
    'ic' : ('Omega', {'u.all' : 0.0, 'du.all' : 0.0}),
}

if (ls_name == 'lsrmm') and fast_rmm:
    # Speed up residual calculation, as M is not used with lsrmm.
    term = 'dw_zero.i.Omega(v, ddu)'

else:
    term = 'de_mass.i.Omega(solid.rho, solid.lumping, solid.beta, v, ddu)'

equations = {
    'balance_of_forces' :
    term + '+ dw_lin_elastic.i.Omega(solid.D, v, u) = 0',
}

solvers = {
    'lsd' : ('ls.auto_direct', {
        # Reuse the factorized linear system from the first time step.
        'use_presolve' : True,
        # Speed up the above by omitting the matrix digest check used
        # normally for verification that the current matrix corresponds to
        # the factorized matrix stored in the solver instance. Use with
        # care!
        'use_mtx_digest' : False,
    }),
    'lsi' : ('ls.petsc', {
        'method' : 'cg',
        'precond' : 'icc',
        'i_max' : 150,
        'eps_a' : 1e-32,
        'eps_r' : 1e-8,
        'verbose' : 2,
    }),
    'lsrmm' : ('ls.rmm', {
        'rmm_term' : ""de_mass.i.Omega(solid.rho, solid.lumping,
                                         solid.beta, v, ddu)""",
        'debug' : False,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
        'eps_r' : 1e-6,
        'ls_on' : 1e100,
    }),
    'tsvv' : ('ts.velocity_verlet', {
        # Explicit method.
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,
    })
}

```

(continues on next page)

(continued from previous page)

```

        'is_linear' : True,

        'verbose' : 1,
    },
    'tscd' : ('ts.central_difference', {
        # Explicit method. Supports ls.rmm.
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'verbose' : 1,
    },
    ),
    'tsn' : ('ts.newmark', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'beta' : 0.25,
        'gamma' : 0.5,

        'verbose' : 1,
    },
    ),
    'tsga' : ('ts.generalized_alpha', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

        'rho_inf' : 0.5,
        'alpha_m' : None,
        'alpha_f' : None,
        'beta' : None,
        'gamma' : None,

        'verbose' : 1,
    },
    ),
    'tsb' : ('ts.bathe', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,

```

(continues on next page)

(continued from previous page)

```

        'verbose' : 1,
    }),
    'tscedb' : ('tsc.ed_basic', {
        'eps_r' : (1e-4, 1e-1),
        'eps_a' : (1e-8, 5e-2),
        'fmin' : 0.3,
        'fmax' : 2.5,
        'fsafety' : 0.85,
    }),
    'tscedl' : ('tsc.ed_linear', {
        'eps_r' : (1e-4, 1e-1),
        'eps_a' : (1e-8, 5e-2),
        'fmin' : 0.3,
        'fmax' : 2.5,
        'fsafety' : 0.85,
        'red_factor' : 0.9,
        'inc_wait' : 10,
        'min_inc_factor' : 1.5,
    }),
}

options = {
    'ts' : tss_name,
    'tsc' : tsc_name if adaptive else None,
    'nls' : 'newton',
    'ls' : ls_name,

    'save_times' : save_times,

    'active_only' : active_only,
    'auto_transform_equations' : True,

    'output_format' : 'h5',
    'output_dir' : output_dir,
    'post_process_hook' : 'post_process',
}

return locals()

```

linear_elasticity/shell10x_cantilever.py

Description

Bending of a long thin cantilever beam, declarative problem description.

The example demonstrates use of the `dw_shell10x` term.

Find displacements of the central plane $\underline{u}$, and rotations $\underline{\alpha}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}, \underline{\beta}) e_{kl}(\underline{u}, \underline{\alpha}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{f}, \quad \forall \underline{v},$$

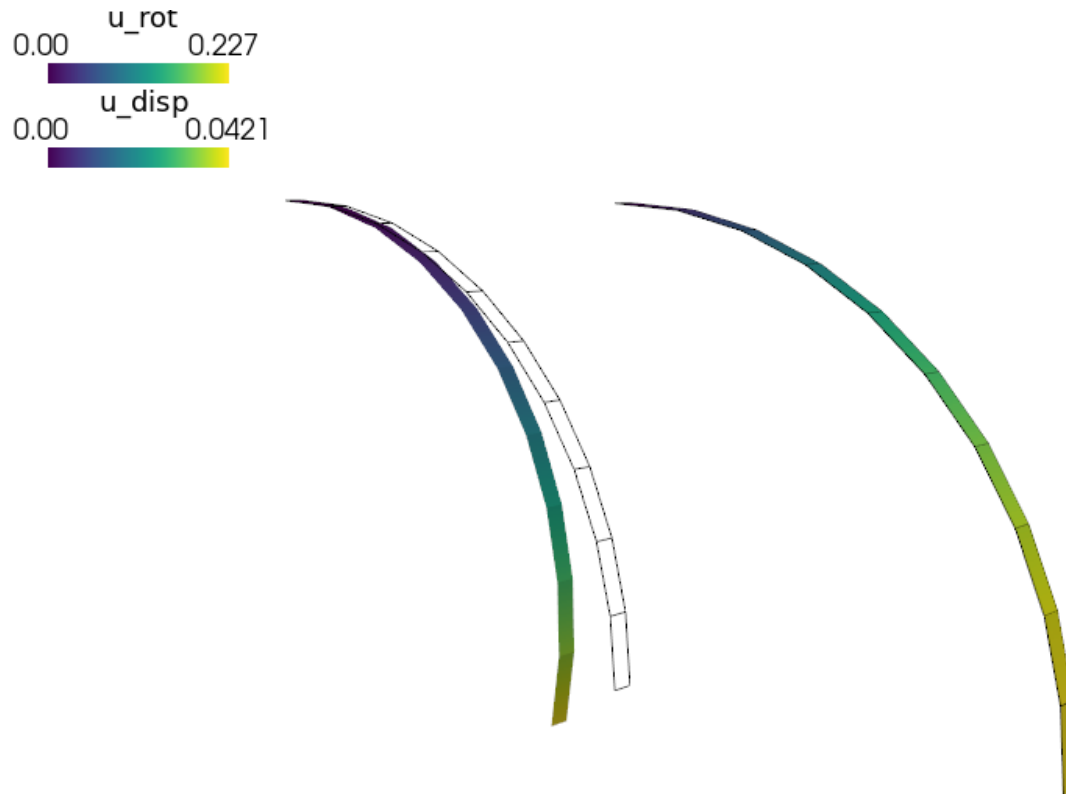
where D_{ijkl} is the isotropic elastic tensor, given using the Young's modulus E and the Poisson's ratio ν .

The variable `u` below holds both $\underline{u}$ and $\underline{\alpha}$ DOFs. For visualization, it is saved as two fields `u_disp` and `u_rot`, corresponding to $\underline{u}$ and $\underline{\alpha}$, respectively.

See also [linear_elasticity/shell10x_cantilever_interactive.py](#) example.

View the results using:

```
sfepy-view shell10x.vtk -f u_disp:wu_disp 1:vw
```



source code

```
r"""
Bending of a long thin cantilever beam, declarative problem description.

The example demonstrates use of the
:class:`dw_shell10x` <sfepy.terms.terms_shells.Shell10XTerm>` term.

Find displacements of the central plane :math:`\underline{u}``, and rotations
:math:`\underline{\alpha}`` such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\underline{v}, \underline{\beta})
    \epsilon_{kl}(\underline{u}, \underline{\alpha})
    = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{f}
    \quad \text{for all } \underline{v};
```

(continues on next page)

(continued from previous page)

where D_{ijkl} is the isotropic elastic tensor, given using the Young's modulus E and the Poisson's ratio ν .

The variable `u` below holds both u and u_α DOFs. For visualization, it is saved as two fields `u_disp` and `u_rot`, corresponding to u and u_α , respectively.

See also `:ref:linear_elasticity-shell10x_cantilever_interactive` example.

View the results using::

```
sfePy-view shell10x.vtk -f u_disp:wu_disp 1:vw
"""
from __future__ import absolute_import
from sfepy.base.base import output
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.discrete import Integral
import sfepy.mechanics.shell10x as sh

import sfepy.examples.linear_elasticity.shell10x_cantilever_interactive as sci

# Beam dimensions.
dims = [0.2, 0.01, 0.001]
thickness = dims[2]

transform = 'bend' # None, 'bend' or 'twist'

# Mesh resolution: increase to improve accuracy.
shape = [11, 2]

# Material parameters.
young = 210e9
poisson = 0.3

# Loading force.
force = -1.0

def mesh_hook(mesh, mode):
    """
    Generate the beam mesh.
    """
    if mode == 'read':
        mesh = sci.make_mesh(dims[:2], shape, transform=transform)
        return mesh

def post_process(out, problem, state, extend=False):
    u = problem.get_variables()['u']
    gamma2 = problem.domain.regions['Gamma2']

    dofs = u.get_state_in_region(gamma2)
    output('DOFs along the loaded edge:')
```

(continues on next page)

(continued from previous page)

```

output('\n%s' % dofs)

if transform != 'twist':
    label, ii = {None : ('u_3', 2), 'bend' : ('u_1', 0)}[transform]

    u_exact = sci.get_analytical_displacement(dims, young, force,
                                              transform=transform)
    output('max. %s displacement:' % label, dofs[0, ii])
    output('analytical value:', u_exact)

return out

filename_mesh = UserMeshIO(mesh_hook)

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process',
}

if transform is None:
    pload = [[0.0, 0.0, force / shape[1], 0.0, 0.0, 0.0]] * shape[1]

elif transform == 'bend':
    pload = [[force / shape[1], 0.0, 0.0, 0.0, 0.0, 0.0]] * shape[1]

elif transform == 'twist':
    pload = [[0.0, force / shape[1], 0.0, 0.0, 0.0, 0.0]] * shape[1]

materials = {
    'm' : ({
        'D' : sh.create_elastic_tensor(young=young, poisson=poisson),
        '.drill' : 1e-7,
    },),
    'load' : ({
        '.val' : pload,
    },)
}

xmin = (-0.5 + 1e-12) * dims[0]
xmax = (0.5 - 1e-12) * dims[0]

regions = {
    'Omega' : 'all',
    'Gamma1' : ('vertices in (x < %.14f)' % xmin, 'facet'),
    'Gamma2' : ('vertices in (x > %.14f)' % xmax, 'facet'),
}

fields = {
    'fu': ('real', 6, 'Omega', 1, 'H1', 'shell10x'),
}

```

(continues on next page)

(continued from previous page)

```

variables = {
    'u' : ('unknown field', 'fu', 0),
    'v' : ('test field', 'fu', 'u'),
}

ebcs = {
    'fix' : ('Gammal', {'u.all' : 0.0}),
}

# Custom integral.
aux = Integral('i', order=3)
qp_coors, qp_weights = aux.get_qp('3_8')
qp_coors[:, 2] = thickness * (qp_coors[:, 2] - 0.5)
qp_weights *= thickness

integrals = {
    'i' : ('custom', qp_coors, qp_weights),
}

equations = {
    'elasticity' :
    """dw_shell10x.i.Omega(m.D, m.drill, v, u)
    = dw_point_load.i.Gamma2(load.val, v)""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-7,
    }),
}

```

linear_elasticity/shell10x_cantilever_interactive.py

Description

Bending of a long thin cantilever beam, imperative problem description.

The example demonstrates use of the `dw_shell10x` term.

Find displacements of the central plane $\underline{u}$, and rotations $\underline{\alpha}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}, \underline{\beta}) e_{kl}(\underline{u}, \underline{\alpha}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{f}, \quad \forall \underline{v},$$

where D_{ijkl} is the isotropic elastic tensor, given using the Young's modulus E and the Poisson's ratio ν .

The variable `u` below holds both $\underline{u}$ and $\underline{\alpha}$ DOFs. For visualization, it is saved as two fields `u_disp` and `u_rot`, corresponding to $\underline{u}$ and $\underline{\alpha}$, respectively.

The material, loading and discretization parameters can be given using command line options.

Besides the default straight beam, two coordinate transformations can be applied (see the `--transform` option):

- bend: the beam is bent
- twist: the beam is twisted

For the straight and bent beam a comparison with the analytical solution coming from the Euler-Bernoulli theory is shown.

See also [linear_elasticity/shell10x_cantilever.py](#) example.

Usage Examples

See all options:

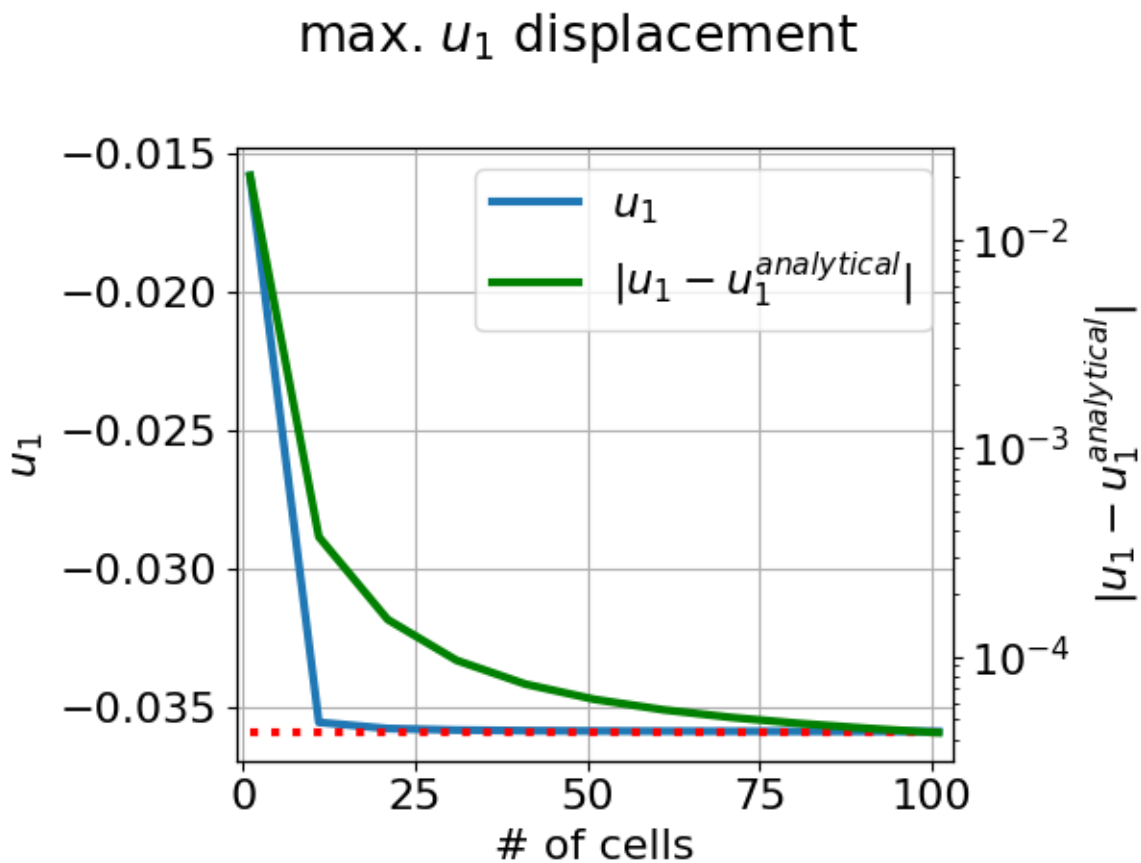
```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py -h
```

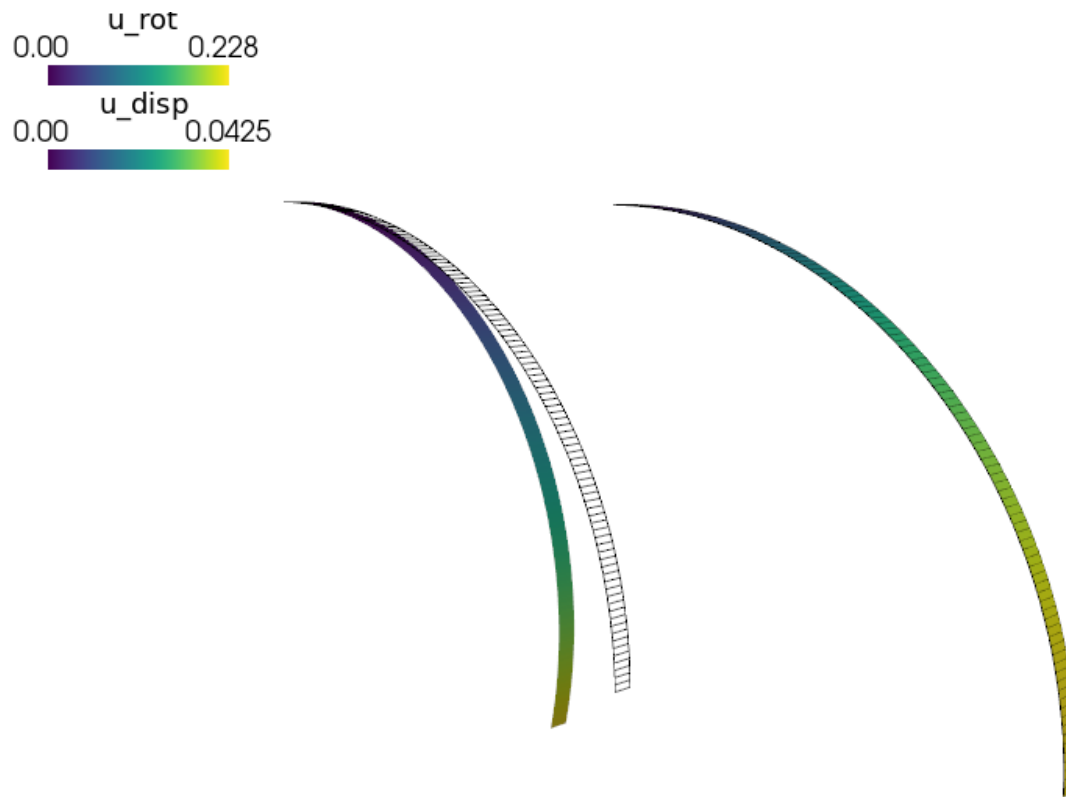
Apply the bending transformation to the beam domain coordinates, plot convergence curves w.r.t. number of elements:

```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py output -t_u
↪ bend -p
```

Apply the twisting transformation to the beam domain coordinates, change number of cells:

```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py output -t_u
↪ twist -n 2,51,3
```





source code

```
#!/usr/bin/env python
r"""
Bending of a long thin cantilever beam, imperative problem description.

The example demonstrates use of the
:class:`dw_shell10x <sfepy.terms.terms_shells.Shell10XTerm>` term.

Find displacements of the central plane  $u$ , and rotations
 $\alpha$  such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(u, \beta)
    e_{kl}(u, \alpha)
    = - \int_{\Gamma} \gamma v \cdot f
    \quad \text{for all } v \text{ ;},

where  $D_{ijkl}$  is the isotropic elastic tensor, given using the Young's
modulus  $E$  and the Poisson's ratio  $\nu$ .

The variable u below holds both  $u$  and  $\alpha$ 
DOFs. For visualization, it is saved as two fields u_disp and u_rot,
```

(continues on next page)

(continued from previous page)

corresponding to u and α , respectively.

The material, loading and discretization parameters can be given using command line options.

Besides the default straight beam, two coordinate transformations can be applied (see the `--transform` option):

- bend: the beam is bent
- twist: the beam is twisted

For the straight and bent beam a comparison with the analytical solution coming from the Euler-Bernoulli theory is shown.

See also `linear_elasticity-shell10x_cantilever` example.

Usage Examples

See all options::

```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py -h
```

Apply the bending transformation to the beam domain coordinates, plot convergence curves w.r.t. number of elements::

```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py output -t
↪ bend -p
```

Apply the twisting transformation to the beam domain coordinates, change number of cells::

```
python3 sfepy/examples/linear_elasticity/shell10x_cantilever_interactive.py output -t
↪ twist -n 2,51,3
```

```
"""
from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser
import os
import sys
from six.moves import range
sys.path.append('.')

import numpy as nm

from sfepy.base.base import output, IndexedStruct
from sfepy.base.ioutils import ensure_path
from sfepy.discrete import (FieldVariable, Material, Integral,
                             Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.solvers.auto_fallback import AutoDirect
```

(continues on next page)

(continued from previous page)

```

from sfepy.solvers.nls import Newton
from sfepy.linalg import make_axis_rotation_matrix
from sfepy.mechanics.tensors import transform_data
from sfepy.mesh.mesh_generators import gen_block_mesh
import sfepy.mechanics.shell10x as sh

def make_mesh(dims, shape, transform=None):
    """
    Generate a 2D rectangle mesh in 3D space, and optionally apply a coordinate
    transform.
    """
    _mesh = gen_block_mesh(dims, shape, [0, 0], name='shell10x', verbose=False)

    coors = nm.c_[_mesh.coors, nm.zeros(_mesh.n_nod, dtype=nm.float64)]
    coors = nm.ascontiguousarray(coors)

    conns = [_mesh.get_conn(_mesh.descs[0])]

    mesh = Mesh.from_data(_mesh.name, coors, _mesh.cmesh.vertex_groups, conns,
                        [_mesh.cmesh.cell_groups], _mesh.descs)

    if transform == 'bend':
        bbox = mesh.get_bounding_box()
        x0, x1 = bbox[:, 0]

        angles = 0.5 * nm.pi * (coors[:, 0] - x0) / (x1 - x0)
        mtx = make_axis_rotation_matrix([0, -1, 0], angles[:, None, None])

        coors = mesh.coors.copy()
        coors[:, 0] = 0
        coors[:, 2] = (x1 - x0)

        mesh.coors[:] = transform_data(coors, mtx=mtx)
        mesh.coors[:, 0] -= 0.5 * (x1 - x0)

    elif transform == 'twist':
        bbox = mesh.get_bounding_box()
        x0, x1 = bbox[:, 0]

        angles = 0.5 * nm.pi * (coors[:, 0] - x0) / (x1 - x0)
        mtx = make_axis_rotation_matrix([-1, 0, 0], angles[:, None, None])

        mesh.coors[:] = transform_data(mesh.coors, mtx=mtx)

    return mesh

def make_domain(dims, shape, transform=None):
    """
    Generate a 2D rectangle domain in 3D space, define regions.
    """
    xmin = (-0.5 + 1e-12) * dims[0]
    xmax = (0.5 - 1e-12) * dims[0]

```

(continues on next page)

(continued from previous page)

```

mesh = make_mesh(dims, shape, transform=transform)
domain = FEDomain('domain', mesh)
domain.create_region('Omega', 'all')
domain.create_region('Gamma1', 'vertices in (x < %.14f)' % xmin, 'facet')
domain.create_region('Gamma2', 'vertices in (x > %.14f)' % xmax, 'facet')

return domain

def solve_problem(shape, dims, young, poisson, force, transform=None):
    domain = make_domain(dims[:2], shape, transform=transform)

    omega = domain.regions['Omega']
    gamma1 = domain.regions['Gamma1']
    gamma2 = domain.regions['Gamma2']

    field = Field.from_args('fu', nm.float64, 6, omega, approx_order=1,
                           poly_space_basis='shell10x')
    u = FieldVariable('u', 'unknown', field)
    v = FieldVariable('v', 'test', field, primary_var_name='u')

    thickness = dims[2]
    if transform is None:
        pload = [[0.0, 0.0, force / shape[1], 0.0, 0.0, 0.0]] * shape[1]

    elif transform == 'bend':
        pload = [[force / shape[1], 0.0, 0.0, 0.0, 0.0, 0.0]] * shape[1]

    elif transform == 'twist':
        pload = [[0.0, force / shape[1], 0.0, 0.0, 0.0, 0.0]] * shape[1]

    m = Material('m', D=sh.create_elastic_tensor(young=young, poisson=poisson),
                 values={'drill' : 1e-7})
    load = Material('load', values={'val' : pload})

    aux = Integral('i', order=3)
    qp_coors, qp_weights = aux.get_qp('3_8')
    qp_coors[:, 2] = thickness * (qp_coors[:, 2] - 0.5)
    qp_weights *= thickness

    integral = Integral('i', coors=qp_coors, weights=qp_weights, order='custom')

    t1 = Term.new('dw_shell10x(m.D, m.drill, v, u)',
                  integral, omega, m=m, v=v, u=u)
    t2 = Term.new('dw_point_load(load.val, v)',
                  integral, gamma2, load=load, v=v)
    eq = Equation('balance', t1 - t2)
    eqs = Equations([eq])

    fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})

    ls = AutoDirect({})

```

(continues on next page)

(continued from previous page)

```

nls_status = IndexedStruct()
nls = Newton({}, lin_solver=ls, status=nls_status)

pb = Problem('elasticity with shell10x', equations=eqs)
pb.set_bcs(ebcs=Conditions([fix_u]))
pb.set_solver(nls)

state = pb.solve()

return pb, state, u, gamma2

def get_analytical_displacement(dims, young, force, transform=None):
    """
    Returns the analytical value of the max. displacement according to
    Euler-Bernoulli theory.
    """
    l, b, h = dims

    if transform is None:
        moment = b * h**3 / 12.0
        u = force * l**3 / (3 * young * moment)

    elif transform == 'bend':
        u = force * 3.0 * nm.pi * l**3 / (young * b * h**3)

    elif transform == 'twist':
        u = None

    return u

helps = {
    'output_dir' : 'output directory',
    'dims' :
    'dimensions of the cantilever [default: %(default)s]',
    'nx' :
    'the range for the numbers of cells in the x direction'
    '[default: %(default)s]',
    'transform' :
    'the transformation of the domain coordinates [default: %(default)s]',
    'young' : "the Young's modulus [default: %(default)s]",
    'poisson' : "the Poisson's ratio [default: %(default)s]",
    'force' : "the force load [default: %(default)s]",
    'plot' : 'plot the max. displacement w.r.t. number of cells',
    'no_show' : 'do not show matplotlib figures',
    'silent' : 'do not print messages to screen',
}

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('output_dir', help=helps['output_dir'])

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('-d', '--dims', metavar='l,w,t',
                    action='store', dest='dims',
                    default='0.2,0.01,0.001', help=helps['dims'])
parser.add_argument('-n', '--nx', metavar='start,stop,step',
                    action='store', dest='nx',
                    default='2,103,10', help=helps['nx'])
parser.add_argument('-t', '--transform', choices=['none', 'bend', 'twist'],
                    action='store', dest='transform',
                    default='none', help=helps['transform'])
parser.add_argument('--young', metavar='float', type=float,
                    action='store', dest='young',
                    default=210e9, help=helps['young'])
parser.add_argument('--poisson', metavar='float', type=float,
                    action='store', dest='poisson',
                    default=0.3, help=helps['poisson'])
parser.add_argument('--force', metavar='float', type=float,
                    action='store', dest='force',
                    default=-1.0, help=helps['force'])
parser.add_argument('-p', '--plot',
                    action="store_true", dest='plot',
                    default=False, help=helps['plot'])
parser.add_argument('--no-show',
                    dest='show', action='store_false',
                    default=True, help=helps['no_show'])
parser.add_argument('--silent',
                    action='store_true', dest='silent',
                    default=False, help=helps['silent'])
options = parser.parse_args()

dims = nm.array([float(ii) for ii in options.dims.split(',')],
                dtype=nm.float64)
nxs = tuple([int(ii) for ii in options.nx.split(',')])
young = options.young
poisson = options.poisson
force = options.force

output_dir = options.output_dir

odir = lambda filename: os.path.join(output_dir, filename)

filename = odir('output_log.txt')
ensure_path(filename)
output.set_output(filename=filename, combined=options.silent == False)

output('output directory:', output_dir)
output('using values:')
output(" dimensions:", dims)
output(" nx range:", nxs)
output(" Young's modulus:", options.young)
output(" Poisson's ratio:", options.poisson)
output(' force:', options.force)
output(' transform:', options.transform)

```

(continues on next page)

(continued from previous page)

```

if options.transform == 'none':
    options.transform = None

u_exact = get_analytical_displacement(dims, young, force,
                                     transform=options.transform)

if options.transform is None:
    ilog = 2
    labels = ['u_3']

elif options.transform == 'bend':
    ilog = 0
    labels = ['u_1']

elif options.transform == 'twist':
    ilog = [0, 1, 2]
    labels = ['u_1', 'u_2', 'u_3']

label = ', '.join(labels)

log = []
for nx in range(*nxs):
    shape = (nx, 2)

    pb, state, u, gamma2 = solve_problem(shape, dims, young, poisson, force,
                                         transform=options.transform)

    dofs = u.get_state_in_region(gamma2)
    output('DOFs along the loaded edge:')
    output('\n%s' % dofs)

    log.append([nx - 1] + nm.array(dofs[0, ilog], ndmin=1).tolist())

pb.save_state(odir('shell10x_cantilever.vtk'), state)

log = nm.array(log)

output('max. %s displacement w.r.t. number of cells:' % label)
output('\n%s' % log)
output('analytical value:', u_exact)

if options.plot:
    import matplotlib.pyplot as plt

    plt.rcParams.update({
        'lines.linewidth' : 3,
        'font.size' : 16,
    })

    fig, ax1 = plt.subplots()
    fig.suptitle('max. %s$ displacement' % label)

```

(continues on next page)

(continued from previous page)

```

for ic in range(log.shape[1] - 1):
    ax1.plot(log[:, 0], log[:, ic + 1], label=r'%s$' % labels[ic])
ax1.set_xlabel('# of cells')
ax1.set_ylabel(r'%s$' % label)
ax1.grid(which='both')

lines1, labels1 = ax1.get_legend_handles_labels()

if u_exact is not None:
    ax1.hlines(u_exact, log[0, 0], log[-1, 0],
              'r', 'dotted', label=r'%s^{analytical}$' % label)

    ax2 = ax1.twinx()
    # Assume single log column.
    ax2.semilogy(log[:, 0], nm.abs(log[:, 1] - u_exact), 'g',
                 label=r'$|s - %s^{analytical}|$' % (label, label))
    ax2.set_ylabel(r'$|s - %s^{analytical}|$' % (label, label))

    lines2, labels2 = ax2.get_legend_handles_labels()

else:
    lines2, labels2 = [], []

ax1.legend(lines1 + lines2, labels1 + labels2, loc='best')

plt.tight_layout()
ax1.set_xlim([log[0, 0] - 2, log[-1, 0] + 2])

suffix = {None: 'straight',
          'bend' : 'bent', 'twist' : 'twisted'}[options.transform]
fig.savefig(odir('shell10x_cantilever_convergence_%s.png' % suffix))

if options.show:
    plt.show()

if __name__ == '__main__':
    main()

```

linear_elasticity/truss_bridge.py

Description

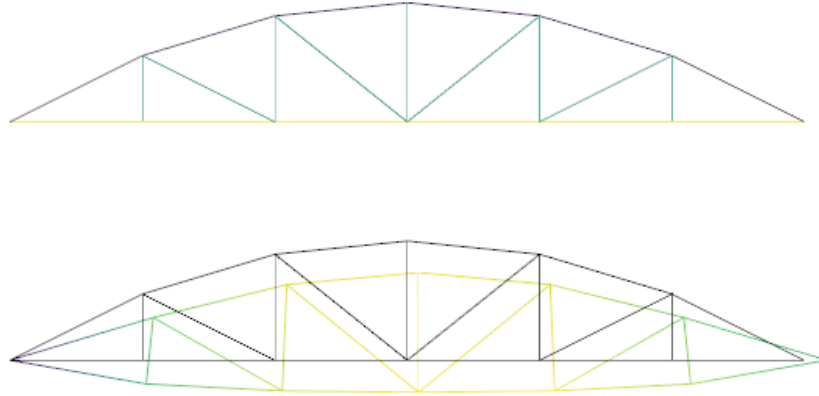
An example demonstrating the usage of the truss elements in 2D. The bridge structure is fixed on the left and supported on the right.

Running the simulation:

```
sfepy-run sfepy/examples/linear_elasticity/truss_bridge.py
```

Viewing the results:

```
sfepy-view bridge.vtk -f u:wu:p0 1:vw:p0 S:e:p1 --2d-view
```



source code

```
r"""
An example demonstrating the usage of the truss elements in 2D.
The bridge structure is fixed on the left and supported on the right.

Running the simulation::

    sfepy-run sfepy/examples/linear_elasticity/truss_bridge.py

Viewing the results::

    sfepy-view bridge.vtk -f u:wu:p0 1:vw:p0 S:e:p1 --2d-view
"""
import numpy as nm
from sfepy.base.base import Struct
from sfepy import data_dir

def post_process(out, pb, state, extend=False):
    valS = pb.evaluate('ev_lin_truss_force.0.Truss(truss.EA, u)', mode='el_avg')
```

(continues on next page)

(continued from previous page)

```

out['S'] = Struct(name='output_data', mode='cell',
                  region_name='Truss', data=valS)

print('### nodal displacements: ')
print(out['u'].data)

return out

def get_pars(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        EA = nm.array([2, 2, 2, 2, 2, 2,
                       10, 10, 10, 10, 10, 10,
                       3, 3, 3, 3, 3,
                       1, 1, 1, 1]).reshape(-1, 1, 1) * 1000

        return {'EA': EA}

filename_mesh = data_dir + "/meshes/2d/bridge.vtk"

options = {
    'post_process_hook': 'post_process',
}

regions = {
    'Truss': 'cells of group 1',
    'Left': ('vertices in (x < 0.01)', 'vertex'),
    'Right': ('vertices in (x > 59.99)', 'vertex'),
    'Bottom': ('vertex 2, 4, 6, 8, 10', 'vertex'),
}

materials = {
    'force': ({'.val': nm.array([[0, -10],
                                [0, -10],
                                [0, -16],
                                [0, -10],
                                [0, -10]])}),
    'truss': 'get_pars',
}

functions = {
    'get_pars': (get_pars,),
}

fields = {
    'displacement': ('real', 'vector', 'Truss', 1),
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
}

```

(continues on next page)

(continued from previous page)

```
}

ebcs = {
    'Fixed_Left': ('Left', {'u.all': 0.0}),
    'Support_Right': ('Right', {'u.1': 0.0}),
}

equations = {
    'balance_of_forces':
        """dw_lin_truss.0.Truss(truss.EA, v, u)
        = dw_point_load.0.Bottom(force.val, v)""",
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {}),
}
```

linear_elasticity/truss_bridge3d.py

Description

An example demonstrating the usage of the truss structural elements in 3D. The mixed (solid and structural elements) bridge structure is fixed on the left and supported on the right.

Running the simulation:

```
sfepy-run sfepy/examples/linear_elasticity/truss_bridge3d.py
```

Viewing the results:

```
sfepy-view bridge3d_S*.vtk -f u_solid:s0:wu_solid:f1e3:p0 u_struct:s1:wu_struct:f1e3:p0
```


u_{solid}
0.00 0.000590



u_struct
0.00 0.000478



source code

```

r"""
An example demonstrating the usage of the truss structural elements in 3D.
The mixed (solid and structural elements) bridge structure is fixed
on the left and supported on the right.

Running the simulation::

    sfepy-run sfepy/examples/linear_elasticity/truss_bridge3d.py

Viewing the results::

    sfepy-view bridge3d_S*.vtk -f u_solid:s0:wu_solid:f1e3:p0 u_struct:s1:wu_struct:f1e3:p0
"""
import numpy as nm
from sfepy.base.base import Struct
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy import data_dir

def post_process(out, pb, state, extend=False):
    valP = pb.evaluate('ev_lin_truss_force.0.Struct(truss.EA, u)',

```

(continues on next page)

(continued from previous page)

```

        mode='el_avg')
out['P'] = Struct(name='output_data', mode='cell',
                  region_name='Struct', data=valP)

valS = pb.evaluate('ev_cauchy_stress.i.Solid(solid.D, u)', mode='el_avg')
out['S'] = Struct(name='output_data', mode='cell',
                  region_name='Solid', data=valS)

u = out.pop('u').data
u_solid = u[pb.domain.regions['Solid'].vertices, :]
out['u_solid'] = Struct(name='output_data', mode='vertex',
                       region_name='Solid', data=u_solid)
u_truss = u[pb.domain.regions['Struct'].vertices, :]
out['u_struct'] = Struct(name='output_data', mode='vertex',
                        region_name='Struct', data=u_truss)

return out

filename_mesh = data_dir + "/meshes/3d/bridge3d.vtk"

options = {
    'post_process_hook': 'post_process',
    'split_results_by': 'region',
}

regions = {
    'Solid': 'cells of group 1',
    'Struct': 'cells of group 2',
    'Omega': ('r.Solid +v r.Struct', 'cell', None, {'finalize': False}),
    'Left': ('vertices in (x < 0.01)', 'vertex'),
    'Right': ('vertices in (x > 11.99)', 'vertex'),
    'Bottom': ('vertices in (z < -0.199)', 'vertex'),
    'Top': ('vertices in (z > -0.001)', 'facet'),
    'LeftBottom': ('r.Left *v r.Bottom', 'vertex'),
    'RightBottom': ('r.Right *v r.Bottom', 'vertex'),
}

materials = {
    'force': ({'val': nm.array([[0, 0, -1000.]]).T},),
    'truss': ({'EA': 210e9 * (0.02**2 * nm.pi)},),
    'solid': ({'D': stiffness_from_youngpoisson(3, 10e9, 0.3)},),
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
}

```

(continues on next page)

(continued from previous page)

```

integrals = {
    'i': 2,
}

ebcs = {
    'Fixed_Left': ('LeftBottom', {'u.all': 0.0}),
    'Support_Right': ('RightBottom', {'u.2': 0.0}),
}

equations = {
    'balance_of_forces':
        """dw_lin_elastic.i.Solid(solid.D, v, u)
        + dw_lin_truss.0.Struct(truss.EA, v, u)
        = dw_surface_ltr.i.Top(force.val, v)""",
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {'eps_a': 1e-6, 'eps_r': 1e-9}),
}

```

linear_elasticity/two_bodies_contact.py

Description

Contact of two elastic bodies with a penalty function for enforcing the contact constraints.

Find $\underline{u}$ such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Gamma_c} \varepsilon_N \langle g_N(\underline{u}) \rangle \underline{n} \underline{v} = 0, \quad \forall \underline{v},$$

where $\varepsilon_N \langle g_N(\underline{u}) \rangle$ is the penalty function, ε_N is the normal penalty parameter, $\langle g_N(\underline{u}) \rangle$ are the Macaulay's brackets of the gap function $g_N(\underline{u})$ and

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

This example also demonstrates use of an experimental contact term based on the IPC Toolkit [1].

[1] <https://github.com/ipc-sim/ipc-toolkit>

Usage examples:

```

# Check regions and EBC nodes.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py --save-regions-as-
→groups --save-ebc-nodes --solve-not

sfepy-view two_bodies_ebc_nodes.vtk
sfepy-view two_bodies_regions.h5

# Run with default parameters and view results.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py

```

(continues on next page)

(continued from previous page)

```

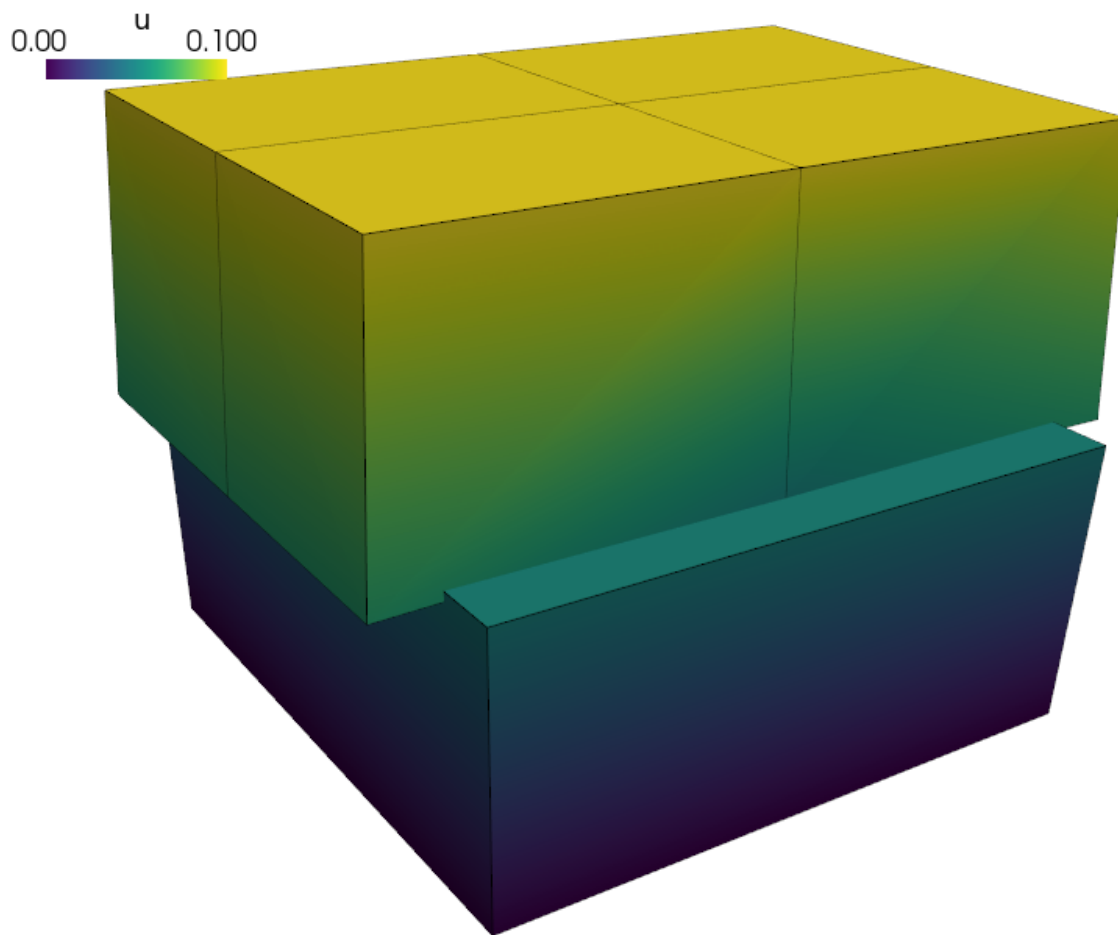
sfepy-view output/contact/two_bodies.h5
sfepy-view output/contact/two_bodies.h5 -f u:wu:f1:p0 1:vw:wu:f1:p0

# Plot the nonlinear solver convergence.
python3 sfepy/scripts/plot_logs.py output/contact/log.txt

# Run with default parameters and IPC toolkit, view results.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py -d "contact='ipc'"
sfepy-view output/contact/two_bodies.h5 -f u:wu:f1:p0 1:vw:wu:f1:p0

# Plot the IPC contact parameters evolution.
python3 sfepy/scripts/plot_logs.py output/contact/clog.txt

```



source code

```

r"""
Contact of two elastic bodies with a penalty function for enforcing the contact
constraints.

Find  $\ul{u}$  such that:

```

(continues on next page)

(continued from previous page)

```
.. math::
    \int_{\{\Omega\}} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    + \int_{\{\Gamma_c\}} \varepsilon_N \ \langle g_N(\ul{u}) \ \rangle \ul{n} \ \ul{v}
    = 0
    \;; \quad \forall \ul{v} \;;
```

where $\varepsilon_N \ \langle g_N(\ul{u}) \ \rangle$ is the penalty function, ε_N is the normal penalty parameter, $\langle g_N(\ul{u}) \ \rangle$ are the Macaulay's brackets of the gap function $g_N(\ul{u})$ and

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;;
```

This example also demonstrates use of an experimental contact term based on the IPC Toolkit [1].

[1] <https://github.com/ipc-sim/ipc-toolkit>

Usage examples::

```
# Check regions and EBC nodes.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py --save-regions-as-
→groups --save-ebc-nodes --solve-not

sfepy-view two_bodies_ebc_nodes.vtk
sfepy-view two_bodies_regions.h5

# Run with default parameters and view results.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py

sfepy-view output/contact/two_bodies.h5
sfepy-view output/contact/two_bodies.h5 -f u:wu:f1:p0 1:vw:wu:f1:p0

# Plot the nonlinear solver convergence.
python3 sfepy/scripts/plot_logs.py output/contact/log.txt

# Run with default parameters and IPC toolkit, view results.
sfepy-run sfepy/examples/linear_elasticity/two_bodies_contact.py -d "contact='ipc'"
sfepy-view output/contact/two_bodies.h5 -f u:wu:f1:p0 1:vw:wu:f1:p0

# Plot the IPC contact parameters evolution.
python3 sfepy/scripts/plot_logs.py output/contact/clog.txt
"""
import os.path as op
from functools import partial
import inspect

from sfepy.base.base import output
from sfepy.base.log import Log
```

(continues on next page)

(continued from previous page)

```

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.meshio import UserMeshIO

import numpy as nm

def get_bbox(dims, centre, eps=0.0):
    dims = nm.asarray(dims)
    centre = nm.asarray(centre)

    bbox = nm.r_[[centre - (0.5 - eps) * dims], [centre + (0.5 - eps) * dims]]
    return bbox

def gen_two_bodies(dims0, shape0, centre0, dims1, shape1, centre1, shift1):
    from sfepy.discrete.fem import Mesh
    from sfepy.mesh.mesh_generators import gen_block_mesh

    m0 = gen_block_mesh(dims0, shape0, centre0)
    m1 = gen_block_mesh(dims1, shape1, centre1)

    coors = nm.concatenate((m0.coors, m1.coors + shift1), axis=0)

    desc = m0.descs[0]
    c0 = m0.get_conn(desc)
    c1 = m1.get_conn(desc)
    conn = nm.concatenate((c0, c1 + m0.n_nod), axis=0)

    ngroups = nm.zeros(coors.shape[0], dtype=nm.int32)
    ngroups[m0.n_nod:] = 1

    mat_id = nm.zeros(conn.shape[0], dtype=nm.int32)
    mat_id[m0.n_el:] = 1

    name = 'two_bodies'

    mesh = Mesh.from_data(name, coors, ngroups, [conn], [mat_id], m0.descs)

    return mesh

def find_contact_ipc_term(equations):
    for eq in equations:
        for term in eq.terms:
            if term.name == 'dw_contact_ipc':
                break

        else:
            continue

        break

    else:
        raise ValueError('no dw_contact_ipc in equations!')

```

(continues on next page)

(continued from previous page)

```

    return term

def apply_line_search(vec_x0, vec_dx0, it, err_last, conf, fun,
                     timers, log=None, context=None, clog=None):
    """
    Apply a backtracking line-search with continuous collision detection from
    IPC toolkit.
    """
    pb = context
    term = find_contact_ipc_term(pb.equations)

    select_other = lambda term: term.name != 'dw_contact_ipc'

    ls = 1.0
    vec_dx = vec_dx0

    variables = pb.get_variables()

    ci = term.get_contact_info(variables['u'])
    ci.it = it

    ls_it = 0
    while 1:
        vec_x = vec_x0 - vec_dx
        if it > 0:
            # Determine max. step size using continuous collision detection.
            u0 = variables.make_full_vec(vec_x0).reshape((-1, ci.dim))
            u1 = variables.make_full_vec(vec_x).reshape((-1, ci.dim))
            x0 = ci.smesh.coors + u0[ci.nods]
            x1 = ci.smesh.coors + u1[ci.nods]

            max_step_size = term.ipc.compute_collision_free_stepsize(
                ci.collision_mesh, x0, x1,
            )
            if max_step_size < 1.0:
                vec_x = vec_x0 - max_step_size * vec_dx

            ls = min(ls, max_step_size)

        timers.residual.start()

        ci.ls_it = ls_it

        try:
            vec_r1 = fun(vec_x, select_term=select_other)

            vec_r1_full = pb.equations.make_full_vec(vec_r1, force_value=0.0)
            ci.e_grad_full = vec_r1_full

            val, iels = term.evaluate(mode='weak', dw_mode='vector',
                                     standalone=False, ci=ci)

```

(continues on next page)

(continued from previous page)

```

vec_r2 = nm.zeros_like(vec_r1)
term.assemble_to(vec_r2, val, iels, mode='vector')

vec_r = vec_r1 + vec_r2

except ValueError:
    if (it == 0) or (ls < conf.ls_min):
        output('giving up!')
        raise

    else:
        ok = False

else:
    ok = True

timers.residual.stop()

if ok:
    err = nm.linalg.norm(vec_r)
    if not nm.isfinite(err):
        output('residual:', vec_r)
        output(nm.isfinite(vec_r).all())
        raise ValueError('infs or nans in the residual')

    if log is not None:
        log(err, it)

    if clog is not None:
        if it > 0:
            clog(ci.min_distance, ci.barrier_stiffness,
                 ci.bp_grad_norm, ci.e_grad_norm)

        else:
            clog(nm.nan, nm.nan, nm.nan, nm.nan)

    if (it == 0) or (err < (err_last * conf.ls_on)):
        break

    red = conf.ls_red
    output('linesearch: iter %d, (%.5e < %.5e) (new ls: %e)'
          % (it, err, err_last * conf.ls_on, red * ls))

else: # Failure.
    if conf.give_up_warp:
        output('giving up!')
        break

    red = conf.ls_red_warp
    output('residual computation failed for iter %d'
          ' (new ls: %e)!' % (it, red * ls))

```

(continues on next page)

(continued from previous page)

```

    if ls < conf.ls_min:
        output('linesearch failed, continuing anyway')
        break

    ls *= red
    vec_dx = ls * vec_dx0
    ls_it += 1

    return vec_x, vec_r, err, ok

def markdown_table_from_dict(adict):
    header = '| option | value |\n'
    separator = '| --- | --- |\n'
    rows = '\n'.join([f'| {key} | {val} |' for key, val in adict.items()])

    return header + separator + rows

def define(
    dims0=(1.0, 1.0, 0.5),
    shape0=(2, 2, 2),
    centre0=(0, 0, -0.25),

    dims1=(1.2, 0.8, 0.5),
    shape1=(3, 3, 2),
    centre1=(0, 0, 0.25),

    shift10=(0.0, 0.0, 1e-4),
    shift11=(0.0, 0.0, -0.1),

    young=1.0,
    poisson=0.3,
    rho=1.0,

    cm=None,
    ck=0.0,
    dhat=1e-2,
    pspd='NONE',

    t1=1000,
    n_step=5,
    contact='builtin',

    output_dir='output/contact',
    verbose=True,
):
    args = locals()

    inodir = partial(op.join, output_dir)

    output.set_output(filename=inodir('output_log.txt'), quiet=not verbose,
                      combined=True)

```

(continues on next page)

(continued from previous page)

```

signature = inspect.signature(define)
arg_names = list(signature.parameters.keys())
options = {name : args[name] for name in arg_names}
table = markdown_table_from_dict(options)
with open(inodir('options.md'), 'w') as fd:
    fd.write(table)

dim = len(dims0)
shape0 = shape0[:dim]
centre0 = centre0[:dim]

dims1 = dims1[:dim]
shape1 = shape1[:dim]
centre1 = centre1[:dim]

shift10 = shift10[:dim]
shift11 = shift11[:dim]

shift11 = nm.array(shift11)

clog = Log([r'$d$', [r'$k$', [r'$\nabla B$', [r'$\nabla E$']],
            xlabel=[', ', 'all iterations', 'all iterations'],
            ylabel=[r'$d$', r'$k$', r'$\nabla B$', r'$\nabla E$'],
            yscale=['log', 'linear', 'linear', 'linear'],
            is_plot=False,
            log_filename=inodir('clog.txt'),
            format=[['%.8e']] * 4)

def mesh_hook(mesh, mode):
    if mode == 'read':
        return gen_two_bodies(dims0, shape0, centre0,
                               dims1, shape1, centre1, shift10)

    elif mode == 'write':
        pass

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct
    from sfepy.discrete.fem import extend_cell_data

    ev = pb.evaluate
    gap = ev('dw_contact.i.Contact(contact.epss, v, u)',
             mode='el_avg', term_mode='gap')
    gap = extend_cell_data(gap, pb.domain, 'Contact', val=0.0,
                           is_surface=True)
    out['gap'] = Struct(name='output_data',
                        mode='cell', data=gap, dofs=None)

    return out

filename_mesh = UserMeshIO(mesh_hook)

```

(continues on next page)

(continued from previous page)

```

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'output_dir' : output_dir,
    'output_format' : 'h5',
    'post_process_hook' : 'post_process',
}

bbox0 = get_bbox(dims0, centre0, eps=1e-5)
bbox1 = get_bbox(dims1, nm.asarray(centre1) + nm.asarray(shift10), eps=1e-5)

if dim == 2:
    regions = {
        'Omega' : 'all',
        'Omega0' : 'cells of group 0',
        'Omega1' : 'cells of group 1',
        'Bottom' : ('vertices in (y < %.12e)' % bbox0[0, 1], 'facet'),
        'Top' : ('vertices in (y > %.12e)' % bbox1[1, 1], 'facet'),
        'Contact0' : ('(vertices in (y > %.12e) *v r.Omega0)' % bbox0[1, 1],
            'facet'),
        'Contact1' : ('(vertices in (y < %.12e) *v r.Omega1)' % bbox1[0, 1],
            'facet'),
        'Contact' : ('r.Contact0 +s r.Contact1', 'facet')
    }

else:
    regions = {
        'Omega' : 'all',
        'Omega0' : 'cells of group 0',
        'Omega1' : 'cells of group 1',
        'Bottom' : ('vertices in (z < %.12e)' % bbox0[0, 2], 'facet'),
        'Top' : ('vertices in (z > %.12e)' % bbox1[1, 2], 'facet'),
        'Contact0' : ('(vertices in (z > %.12e) *v r.Omega0)' % bbox0[1, 2],
            'facet'),
        'Contact1' : ('(vertices in (z < %.12e) *v r.Omega1)' % bbox1[0, 2],
            'facet'),
        'Contact' : ('r.Contact0 +s r.Contact1', 'facet')
    }

fields = {
    'displacement': ('real', dim, 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}

ebcs = {
    'fixb' : ('Bottom', {'u.all' : 0.0}),
    'fixt' : ('Top', {'u.all' : 'move_top'}),
}

```

(continues on next page)

(continued from previous page)

```

def move_top(ts, coors, bc, problem, **kwargs):
    val = nm.empty_like(coors)
    val[:] = ts.nt * shift11
    return val

functions = {
    'move_top' : (move_top,),
}

volume = nm.prod(dims0) + nm.prod(dims1)
mass = volume * rho

materials = {
    'solid' : ({
        'D' : stiffness_from_youngpoisson(dim, young=young, poisson=poisson),
    },),
    'contact' : ({
        '.m' : cm if cm is not None else mass,
        '.k' : ck, # 0 = Adaptive barrier stiffness.
        '.dhat' : dhat,
        '.Pspd' : pspd,
        '.epss' : 2e+1,
    },),
}

integrals = {
    'i' : 2,
}

if contact == 'builtin':
    equations = {
        'elasticity' :
        """
            dw_lin_elastic.2.Omega(solid.D, v, u)
            + dw_contact.i.Contact(contact.epss, v, u)
            = 0
        """,
    }

elif contact == 'ipc':
    equations = {
        'elasticity' :
        """
            dw_lin_elastic.2.Omega(solid.D, v, u)
            + dw_contact_ipc.i.Contact(
                contact.m, contact.k, contact.dhat, contact.Pspd, v, u
            )
            = 0
        """,
    }

```

(continues on next page)

(continued from previous page)

```

else:
    raise ValueError("contact argument can be one of 'builtin', 'ipc'!")

if contact == 'ipc':
    apply_ls = partial(apply_line_search, clog=clog)

else:
    apply_ls = None

solvers = {
    'ls' : ('ls.auto_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 30,
        'eps_a' : 1e-8,
        'eps_r' : 1e-8,
        'eps_mode' : 'or',
        'macheps' : 1e-16,
        # Linear system error < (eps_a * lin_red).
        'lin_red' : None,
        'line_search_fun' : apply_ls,
        'ls_red' : 0.5,
        'ls_red_warp' : 0.5,
        'ls_on' : 1.0,
        'ls_min' : 1e-5,
        'check' : 0,
        'delta' : 1e-8,
        'log' : {'text' : inodir('log.txt')},
        'log_vlines' : 'solve',
    }),
    'ts' : ('ts.simple', {
        't0' : 0.0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step,
        'quasistatic' : True,
        'verbose' : 1,
    }),
}

return locals()

```

linear_elasticity/wedge_mesh.py

Description

A linear elastic beam loaded with a continuous force. The FE meshes consisting of hexahedral, tetrahedral, and wedge elements are used in the simulation and the results are compared.

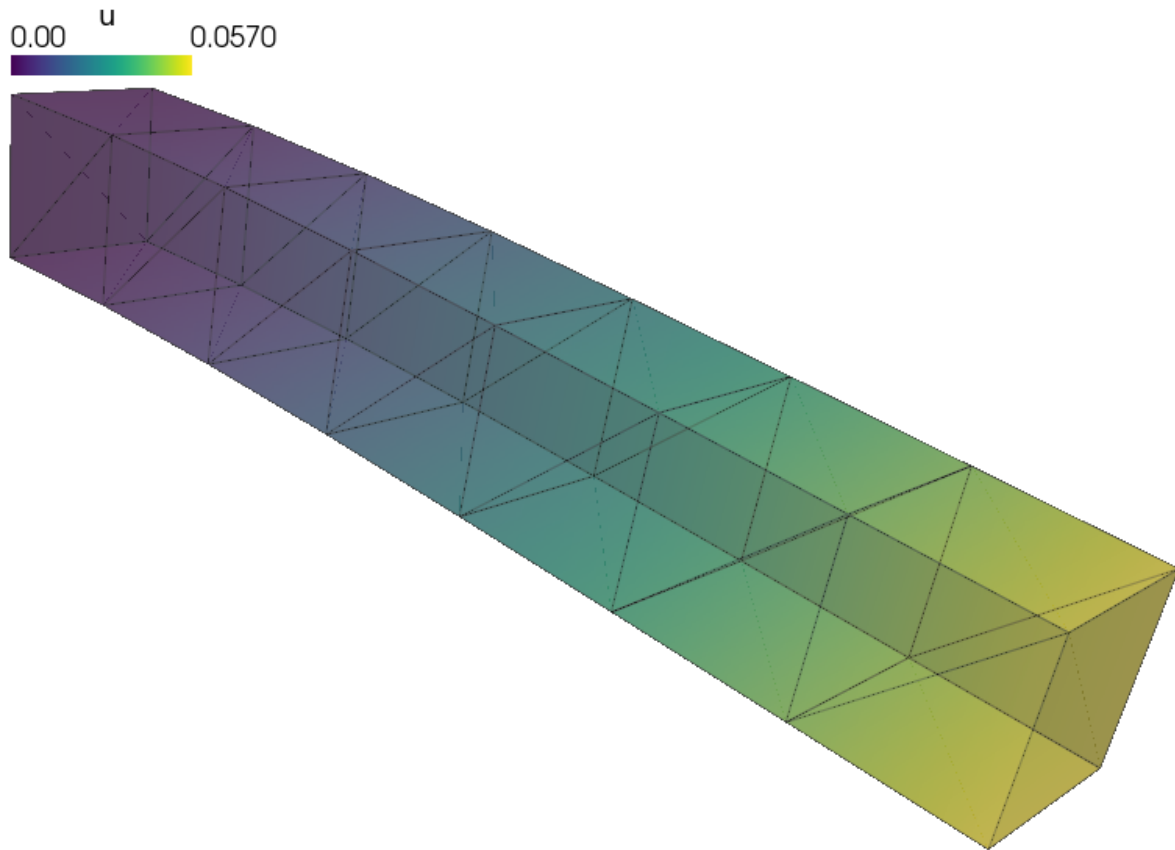
The displacement at the beam end is compared to the reference solution calculated on the homogeneous hexahedral mesh.

Running the simulation:

```
sfepy-run sfepy/examples/linear_elasticity/wedge_mesh.py
```

Viewing the results:

```
sfepy-view output/beam_h7.vtk output/beam_t42.vtk output/beam_w14.vtk -f u:s0:wu:e:p0_
↪u:s1:wu:e:p0 u:s2:wu:e:p0 --camera-position="1.2,-0.6,0.1,0.4,0.1,-0.1,-0.2,0.1,1"
```



source code

```
r"""
A linear elastic beam loaded with a continuous force. The FE meshes consisting
of hexehedral, tetrahedral, and wedge elements are used in the simulation and
the results are compared.
```

```
The displacement at the beam end is compared to the reference
solution calculated on the homogeneous hexahedral mesh.
```

```
Running the simulation::
```

```
    sfepy-run sfepy/examples/linear_elasticity/wedge_mesh.py
```

```
Viewing the results::
```

(continues on next page)

(continued from previous page)

```

    sfepy-view output/beam_h7.vtk output/beam_t42.vtk output/beam_w14.vtk -f
↪ u:s0:wu:e:p0 u:s1:wu:e:p0 u:s2:wu:e:p0 --camera-position="1.2,-0.6,0.1,0.4,0.1,-0.1,-0.
↪ 2,0.1,1"
"""
import os.path as osp
import numpy as nm
from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.base.base import Struct
from sfepy.discrete import Problem
from sfepy.discrete.fem import Mesh
from sfepy.discrete.fem.meshio import UserMeshIO
from scipy.spatial.transform import Rotation

global_dict = {}

def mesh_hook(mesh, mode):
    if mode == 'read' and 'mesh_hook_param' in global_dict:
        fname, angle = global_dict['mesh_hook_param']
        mesh = Mesh.from_file(fname)

        center = nm.average(mesh.cmesh.coors, axis=0)
        coors = mesh.cmesh.coors - center

        rot = Rotation.from_rotvec(nm.array([1., 0, 0]) * nm.deg2rad(angle))
        mesh.cmesh.coors[:] = rot.apply(coors) + center

    return mesh

def test_meshes(pb0):
    out = []
    conf = pb0.conf.copy()
    ok = True

    for mesh_group in meshes:
        displ = []
        for mesh in mesh_group:
            if isinstance(mesh, tuple):
                fname = osp.join(data_dir, 'meshes', '3d', mesh[0])
                angle = mesh[1]
                global_dict['mesh_hook_param'] = (fname, angle)
                conf.filename_mesh = UserMeshIO(mesh_hook)
            else:
                conf.filename_mesh = osp.join(data_dir, 'meshes', '3d', mesh)

        pb = Problem.from_conf(conf)
        pb.set_output_dir(pb0.output_dir)

    yield pb, out

```

(continues on next page)

(continued from previous page)

```

        displ.append(out[-1][1]['u'].data[-1].reshape((-1, 3)))

    yield None

    err = nm.array([d[-1, 2] - displ[0][-1, 2] for d in displ[1:]]) < 0.01
    ok = ok and err.all()

    print(f'wedge elements test: {["failed!", "passed"][int(ok)]}')

def get_force(ts, coors, mode=None, **kwargs):
    if mode == 'qp':
        force = 1e3

        val = nm.zeros_like(coors)[..., None]
        val[:, 2, 0] = -coors[:, 0] / 0.7 * force

        return {'val': val}

def post_proces(out, pb, state, extend=False):
    S = pb.evaluate('ev_cauchy_stress.i.Omega(solid.D, u)', mode='el_avg')
    out['stress'] = Struct(name='out', data=S, mode='cell')

    return out

meshes = [
    ['beam_h7.mesh', ('beam_w14.vtk', 90), ('beam_w14.vtk', 270)],
    ['beam_t42.mesh', 'beam_w14.vtk', ('beam_w14.vtk', 180)],
]

def define():
    filename_mesh = osp.join(data_dir, 'meshes', '3d', meshes[0][0])

    options = {
        'post_process_hook': 'post_proces',
        'parametric_hook': 'test_mesher',
        'output_dir': 'output',
    }

    regions = {
        'Omega': 'all',
        'Left': ('vertices in (x < 0.01)', 'facet'),
        'Right': ('vertices in (x > 0.69)', 'facet'),
        'Top': ('vertices in (z > 0.09)', 'facet'),
        'Bottom': ('vertices in (z < 0.01)', 'facet'),
        'Edge': ('r.Bottom *v r.Right', 'vertex'),
    }

    functions = {
        'get_force': (get_force,),
    }

```

(continues on next page)

(continued from previous page)

```
}

materials = {
    'solid': ({'D': stiffness_from_youngpoisson(dim=3, young=1e6,
                                                poisson=0.3)},),
    'force': 'get_force',
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}

integrals = {
    'i': 2,
}

variables = {
    'u': ('unknown field', 'displacement', 0),
    'v': ('test field', 'displacement', 'u'),
}

ebcs = {
    'Fixed': ('Left', {'u.all' : 0.0}),
}

equations = {
    'balance_of_forces':
        """dw_lin_elastic.i.Omega(solid.D, v, u)
        = dw_surface_ltr.i.Top(force.val, v)"""
}

solvers = {
    'ls': ('ls.auto_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-6,
    }),
}

return locals()
```

miscellaneous

miscellaneous/live_plot.py

Description

Live plot demonstration.

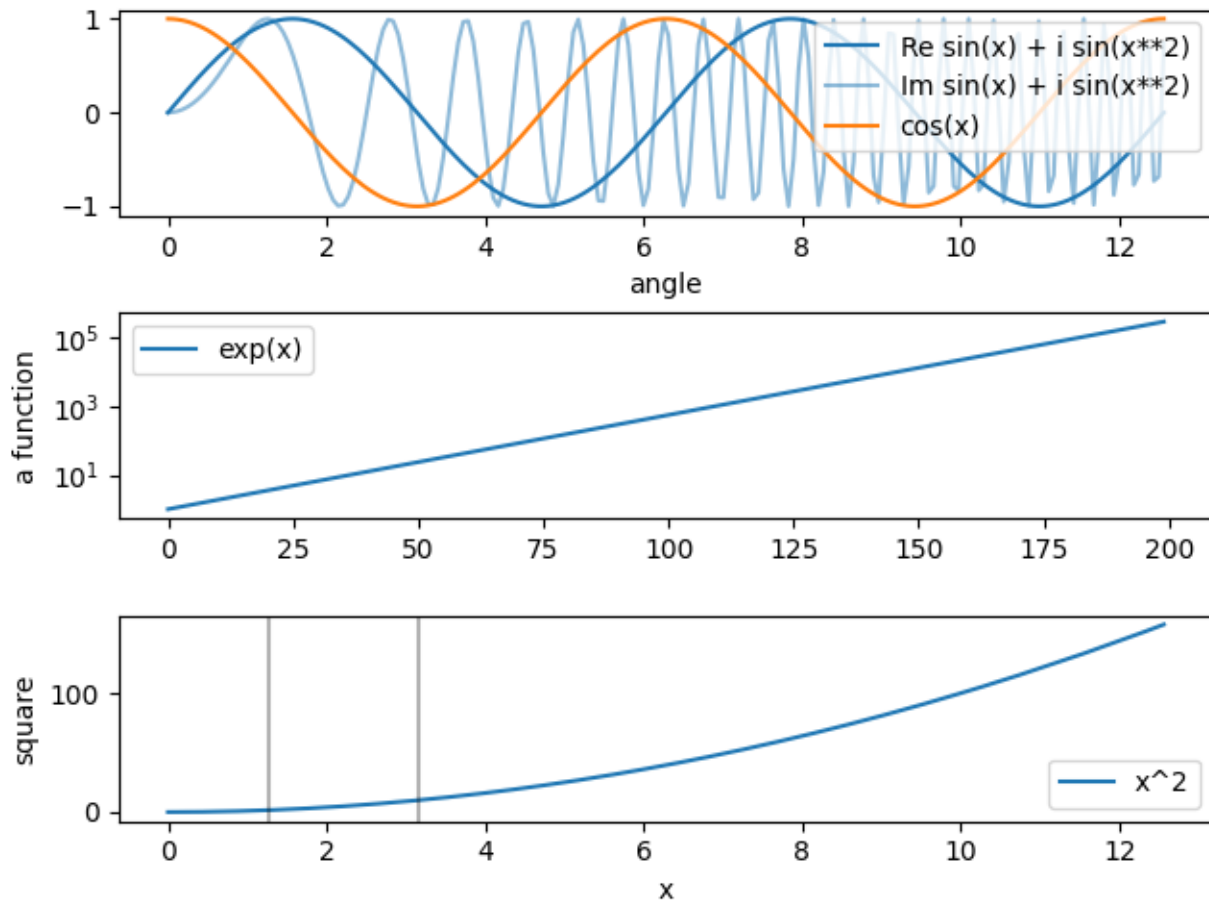
Usage Example

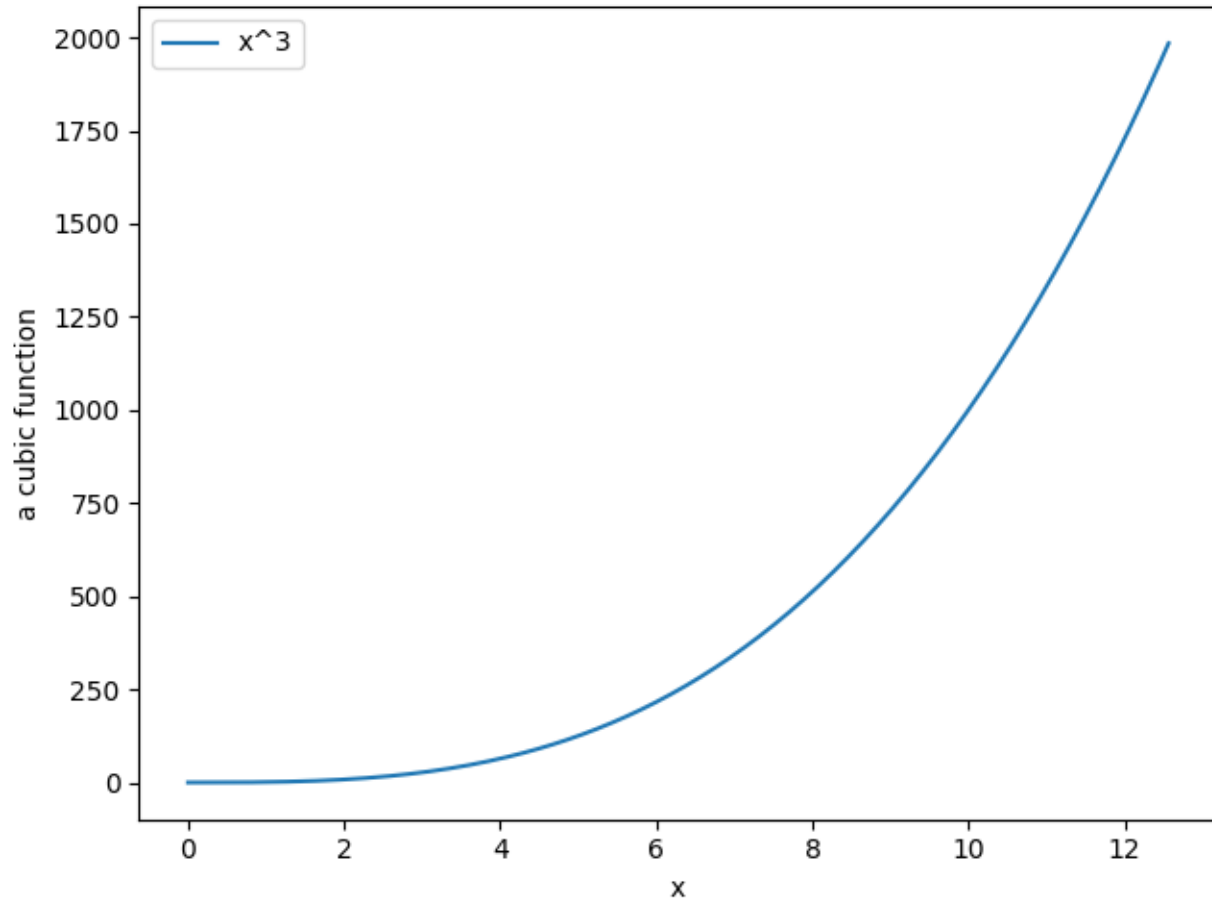
- Run and plot two logs on the fly:

```
python3 sfepy/examples/miscellaneous/live_plot.py --plot-log
```

- Run and store the two logs, plot them later:

```
python3 sfepy/examples/miscellaneous/live_plot.py
python3 sfepy/scripts/plot_logs.py live_plot.txt
python3 sfepy/scripts/plot_logs.py live_plot2.txt
```





source code

```
"""
Live plot demonstration.

Usage Example
-----

- Run and plot two logs on the fly::

    python3 sfepy/examples/miscellaneous/live_plot.py --plot-log

- Run and store the two logs, plot them later::

    python3 sfepy/examples/miscellaneous/live_plot.py

    python3 sfepy/scripts/plot_logs.py live_plot.txt
    python3 sfepy/scripts/plot_logs.py live_plot2.txt
"""
from argparse import ArgumentParser, RawDescriptionHelpFormatter
import os.path as op
from functools import partial

import numpy as nm
```

(continues on next page)

(continued from previous page)

```

from sfepy.base.base import output, pause
from sfepy.base.log import Log

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('-o', '--output-dir', default='.',
                        help='output directory')
    parser.add_argument('-p', '--plot-log', action='store_true',
                        default=False, help='show live log figures')
    options = parser.parse_args()

    inodir = partial(op.join, options.output_dir)

    log = Log([('sin(x) + i sin(x**2)', 'cos(x)'], ['exp(x)'], ['x^2']),
              yscale=['linear', 'log', 'linear'],
              xlabel=['angle', None, 'x'],
              ylabel=[None, 'a function', 'square'],
              aggregate=1000, sleep=0.05,
              is_plot=options.plot_log,
              log_filename=inodir('live_plot.txt'))

    # Plot groups can be added on the fly but then they are missing in the log
    # header - sfepy/scripts/plot_logs.py cannot plot them.
    # log.add_group(['x^2'], yscale='linear', xlabel='new x',
    #                ylabel='square', formats=['%+g'])

    log2 = Log(['x^3'],
               yscale=['linear'],
               xlabel=['x'], ylabel=['a cubic function'],
               aggregate=1000, sleep=0.05,
               is_plot=options.plot_log,
               log_filename=inodir('live_plot2.txt'),
               formats=[['{:.5e}']])

    for ii, x in enumerate(nm.linspace(0, 4.0 * nm.pi, 200)):
        output('x: ', x)

        log(nm.sin(x)+1j*nm.sin(x**2), nm.cos(x), nm.exp(x), x**2,
            x=[x, None, x])

        if (ii == 20) or (ii == 50):
            log.plot_vlines([2], color='g', linewidth=2)

        log2(x**3, x=[x])

    print(log)
    print(log2)

    log(save_figure=inodir('live_plot.png'))
    log2(save_figure=inodir('live_plot2.png'))

```

(continues on next page)

(continued from previous page)

```

if options.plot_log:
    pause()

log(finished=True)
log2(finished=True)

if __name__ == '__main__':
    main()

```

miscellaneous/refine_evp.py

Description

Plot the convergence of eigenvalues (or corresponding frequencies) of an eigenvalue problem to an analytical solution, when applying the uniform mesh refinement.

Uses the PRIMME eigenvalue solver by default (`pip install primme`).

Usage Examples

- Run without the convergence analysis, use the spectral element method (SEM) basis of order 5:

```
sfepy-run sfepy/examples/miscellaneous/refine_evp.py -d order=5,basis=sem
```

- Get help:

```
python3 sfepy/examples/miscellaneous/refine_evp.py -h
```

- Plot the convergence of the smallest eigenvalue of the Laplace Dirichlet problem:

```
python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2
```

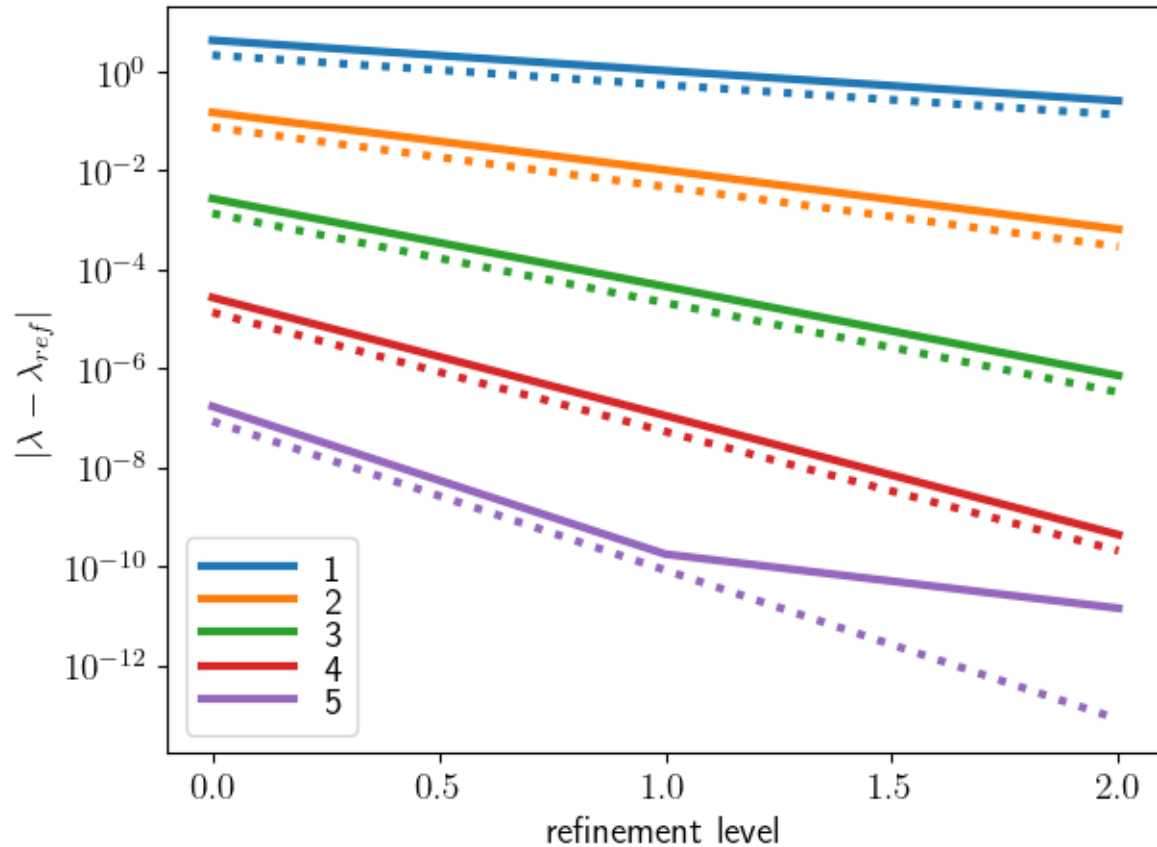
- Plot the convergence of the smallest frequency of the 1D elastic bar vibration problem, show relative errors:

```
python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
↪kind=elasticity --transform=freqs --relative
```

- Using the 1D elastic bar vibration problem, compare the SEM results with the FEM + row-sum mass matrix lumping. Plot also the sparsity patterns of the mass (M) and stiffness (K) matrices:

```
python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
↪evps=primme --kind=elasticity-lumping --transform=freqs --relative --beta=1 --
↪mass-lumping='row_sum' --sparsity

python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
↪evps=primme --kind=elasticity --basis=sem --transform=freqs --relative --beta=0 --
↪mass-lumping='none' --sparsity
```



source code

```
"""
Plot the convergence of eigenvalues (or corresponding frequencies) of an
eigenvalue problem to an analytical solution, when applying the uniform
mesh refinement.

Uses the PRIMME eigenvalue solver by default (`pip install primme`).

Usage Examples
-----

- Run without the convergence analysis, use the spectral element method (SEM)
  basis of order 5::

    sfepy-run sfepy/examples/miscellaneous/refine_evp.py -d order=5,basis=sem

- Get help::

    python3 sfepy/examples/miscellaneous/refine_evp.py -h

- Plot the convergence of the smallest eigenvalue of the Laplace Dirichlet
  problem::

    python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2
```

(continues on next page)

(continued from previous page)

```

- Plot the convergence of the smallest frequency of the 1D elastic bar
  vibration problem, show relative errors::

    python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
    ↪ kind=elasticity --transform=freqs --relative

- Using the 1D elastic bar vibration problem, compare the SEM results with the
  FEM + row-sum mass matrix lumping. Plot also the sparsity patterns of the
  mass (M) and stiffness (K) matrices::

    python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
    ↪ evps=primme --kind=elasticity-lumping --transform=freqs --relative --beta=1 --mass-
    ↪ lumping='row_sum' --sparsity

    python3 sfepy/examples/miscellaneous/refine_evp.py --max-order=5 --max-refine=2 --
    ↪ evps=primme --kind=elasticity --basis=sem --transform=freqs --relative --beta=0 --mass-
    ↪ lumping='none' --sparsity
    """
from argparse import ArgumentParser, RawDescriptionHelpFormatter
import sys
import os.path as op
from functools import partial
from itertools import product

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import output
from sfepy.base.conf import dict_from_string as parse_as_dict
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.base.conf import ProblemConf
from sfepy.discrete import Problem
from sfepy.solvers import Solver
from sfepy.discrete.fem.poly_spaces import SEMTensorProductPolySpace

def define(order=1, refine=0, evps='primme', n_eigs=1, eigs_only=True,
           kind='laplace', basis='lagrange', beta=1.0, mass_lumping='none',
           transform='none', output_dir='output', **kwargs):

    options = {
        'evps' : evps,
        'n_eigs' : n_eigs,
        'eigs_only' : eigs_only,

        'active_only' : True,
        'refinement_level' : refine,

        'output_dir' : output_dir,
    }

```

(continues on next page)

(continued from previous page)

```

if kind == 'laplace':
    dims, shape, centre = [1.0, 1.0], [3, 3], [0.5, 0.5]

    regions = {
        'Omega' : 'all',
        'Gamma' : ('vertices of surface', 'facet'),
    }

    materials = {
    }

    fields = {
        'fu' : ('real', 'scalar', 'Omega', order, 'H1', basis),
    }

    variables = {
        'u' : ('unknown field', 'fu', 0),
        'v' : ('test field', 'fu', 'u'),
    }

    ebcs = {
        'zeros' : ('Gamma', {'u.0' : 0.0}),
    }

    if basis != 'sem':
        integrals = {
            'i' : 2 * order,
        }

    else:
        from sfepy.discrete.fem.geometry_element import GeometryElement
        dim = len(dims)
        gel = GeometryElement(f'{dim}_{2**dim}')
        ps = SEMTensorProductPolySpace(None, gel, order)
        integrals = {
            'i' : ('custom', ps.node_coors, ps.node_weights),
        }

    equations = {
        'lhs' : ""dw_laplace.i.Omega(v, u)"",
        'rhs' : ""dw_dot.i.Omega(v, u)"",
    }
    def eval_exact(n_eigs):
        num = int(nm.ceil(nm.sqrt(n_eigs)))
        rng = range(1, num+1)
        eigs = [(ir**2 + ic**2) * nm.pi**2 for ir, ic in product(rng, rng)]
        return nm.array(sorted(eigs)[:n_eigs], dtype=nm.float64)

    else:
        import sfepy.mechanics.matcoefs as mc

        dims, shape, centre = [1.0], [2], [0.5]

```

(continues on next page)

(continued from previous page)

```

regions = {
    'Omega' : 'all',
    'Gamma' : ('vertices in (x < %.16e)' % 1e-8, 'facet'),
}
dim = len(dims)
young = 69e9
density = 2700.0
poisson = 0.0
plane = 'strain'
materials = {
    'm' : ({
        'rho' : density,
        'D' : mc.stiffness_from_youngpoisson(
            dim, young=young, poisson=poisson, plane=plane
        ),
    },),
    'c' : ({
        '.beta' : beta,
        '.mass_lumping' : mass_lumping,
    },),
}

fields = {
    'fu' : ('real', dim, 'Omega', order, 'H1', basis),
}

variables = {
    'u' : ('unknown field', 'fu', 0),
    'v' : ('test field', 'fu', 'u'),
}

ebcs = {
    'zeros' : ('Gamma', {'u.all' : 0.0}),
}

if basis != 'sem':
    integrals = {
        'i' : 2 * order,
    }
else:
    from sfepy.discrete.fem.geometry_element import GeometryElement
    dim = len(dims)
    gel = GeometryElement(f'{dim}_{2**dim}')
    ps = SEMTensorProductPolySpace(None, gel, order)

    integrals = {
        'i' : ('custom', ps.node_coors, ps.node_weights),
    }

equations = {

```

(continues on next page)

(continued from previous page)

```

        'lhs' : ""dw_lin_elastic.i.Omega(m.D, v, u)""",
        'rhs' : ""dw_dot.i.Omega(m.rho, v, u)""",
    }
    if kind == 'elasticity-lumping':
        equations.update({
            'rhs' : ""de_mass.i.Omega(m.rho, c.mass_lumping, c.beta,
                                     v, u)""",
        })

    def eval_exact(n_eigs):
        c0 = nm.sqrt(young / density)

        ns = nm.arange(1, n_eigs+1)
        freqs = 0.5 * (ns - 0.5) * c0 / dims[0]

        if transform == 'none':
            eigs = (2 * nm.pi * freqs)**2

        else:
            eigs = freqs

        return eigs

    def mesh_hook(mesh, mode):
        """
        Generate the block mesh.
        """
        if mode == 'read':
            mesh = gen_block_mesh(dims, shape, centre,
                                  name='evp-test', verbose=False)

            return mesh

        elif mode == 'write':
            pass

    filename_mesh = UserMeshIO(mesh_hook)

    solvers = {
        'matlab': ('eig.matlab', {
            'method': 'eigs',
            'which': 'sm',
            'eps': 1e-14,
        }),
        'primme' : ('eig.primme', {
            'which' : 'SM',
            # Prevent hanging with 1x1 matrices and tol = 1e-14.
            'tol' : 1e-13 if refine == 0 else 1e-14,
        }),
        'scipy' : ('eig.scipy', {
            'method' : 'eigsh',
            'tol' : 1e-14,
            # 'maxiter' : 150,

```

(continues on next page)

(continued from previous page)

```

        # Compute the eigenvalues near 0 (= the smallest eigenvalues) using
        # the shift-invert mode.
        'which' : 'LM',
        'sigma' : 0.0,
    }},
}

return locals()

def get_figname(name, options):
    rel = 'r' if options.relative else 'a'
    figname = (f'{name}-{options.kind}-{options.basis}-{options.evps}'
               f'-{options.transform}-{rel}{options.fig_suffix}')
    return figname

def parse_args(args=None):
    helps = {}

    opts = dict(
        max_order = (1, 'max. approximation order'),
        max_refine = (0, 'max. uniform refinement level'),
        evps = ('primme', 'solver name'),
        n_eigs = (1, 'number of eigenvalues'),
        kind = (('laplace', 'elasticity', 'elasticity-lumping'), 'problem kind'),
        basis = (('lagrange', 'lobatto', 'sem'), 'field basis'),
        beta = (1.0, 'averaged mass matrix parameter'),
        mass_lumping = ('none', 'mass matrix lumping algorithm'),
        transform = (('none', 'freqs'), 'eigenvalues transformation'),
        relative = (False, 'plot relative errors'),
        sparsity = (False, 'plot sparsity patterns of matrices'),
        fig_suffix = ('.pdf', 'file suffix for saving figures'),
        plot_rc_params = ('', 'matplotlib resources'),
        output_dir = ('output', 'output directory'),
        show = (True, 'do not show matplotlib figures'),
        shell = (False, 'run ipython shell after all computations'),
        debug = (False,
                 'automatically start debugger when an exception is raised'),
    )

    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)

    dhelp = ' [default: %(default)s]'
    for key, (val, msg) in opts.items():
        helps[key] = msg
        action = 'store'
        vtype = type(val)
        choices = None
        option = key
        if val is True:
            action = 'store_false'

```

(continues on next page)

(continued from previous page)

```

        option = 'no_' + key

    elif val is False:
        action = 'store_true'

    elif isinstance(val, tuple):
        choices = val
        vtype = type(val[0])
        val = val[0]

    elif isinstance(val, list):
        vtype = type(val[1])
        val = val[0]

    if action == 'store':
        helps[key] += dhelp
        parser.add_argument('--' + option.replace('_', '-'),
                            type=vtype,
                            action=action, dest=key, choices=choices,
                            default=val, help=helps[key])
    else:
        parser.add_argument('--' + option.replace('_', '-'),
                            action=action, dest=key,
                            default=val, help=helps[key])
options = parser.parse_args(args=args)

options.plot_rc_params = parse_as_dict(options.plot_rc_params)

return options, helps

def main():
    options, helps = parse_args()
    if options.debug:
        from sfepy.base.base import debug_on_error; debug_on_error()

    inodir = partial(op.join, options.output_dir)

    orders = range(1, options.max_order + 1)

    all_eigs = []
    for order, refine in product(
        orders,
        range(options.max_refine + 1),
    ):
        conf = ProblemConf.from_dict(define(order=order, refine=refine,
                                           **vars(options)),
                                   sys.modules[__name__])
        pb = Problem.from_conf(conf)

        pb.time_update()
        mtx_a = pb.evaluate(pb.conf.equations['lhs'], mode='weak',
                           auto_init=True, dw_mode='matrix')

```

(continues on next page)

(continued from previous page)

```

mtx_b = pb.evaluate(pb.conf.equations['rhs'], mode='weak',
                    dw_mode='matrix')
eig = Solver.any_from_conf(pb.get_solver_conf(options.evps))

n_eigs = min(mtx_a.shape[0], options.n_eigs)
try:
    eigs = eig(mtx_a, mtx_b, n_eigs, eigenvectors=False)
    eigs = nm.pad(eigs, (0, options.n_eigs - n_eigs), 'constant',
                  constant_values=(nm.nan, nm.nan))

except:
    eigs = nm.full(options.n_eigs, nm.nan)

all_eigs.append(eigs)

eigs_nums = nm.array(all_eigs).reshape(
    (options.max_order, -1, options.n_eigs)
)
eigs_ana = pb.conf.eval_exact(options.n_eigs)

if options.transform == 'freqs':
    eigs_nums = nm.sqrt(eigs_nums) / (2 * nm.pi)

ref = nm.arange(options.max_refine + 1)
hs = 1.0 / 2.0**ref

plt.rcParams['text.usetex'] = True
plt.rcParams['font.size'] = 14
plt.rcParams['lines.linewidth'] = 3
plt.rcParams['legend.handlelength'] = 2.5
plt.rcParams['legend.labelspacing'] = 0.0
plt.rcParams.update(options.plot_rc_params)

x = r'\lambda' if options.transform == 'none' else 'f'
if options.relative:
    ylabel = f'$|{x}|/{x}_{\{ref\}} - 1|'$

else:
    ylabel = f'$|{x} - {x}_{\{ref\}}|'$

for ie in range(options.n_eigs):
    fig, ax = plt.subplots()
    colors = plt.cm.tab10.colors
    for io, order in enumerate(orders):
        eigs_num = eigs_nums[io, :, ie]
        eig_ana = eigs_ana[ie]

        error = nm.abs(eigs_num - eig_ana)
        if options.relative:
            error /= eig_ana

```

(continues on next page)

(continued from previous page)

```

output('order:', order)
output('analytical:', eig_ana)
output('numerical :', eigs_num)

t0 = error[0] / 2
terror = t0 * hs**(2 * order)

ax.semilogy(ref, error, color=colors[io], label=order)
ax.semilogy(ref, terror, color=colors[io], ls=':')

ax.set_xlabel('refinement level')
ax.set_ylabel(ylabel)
ax.legend()

fig.tight_layout()
fig.savefig(inudir(get_figname(f'h-refinement-{ie}', options)),
            bbox_inches='tight')

if options.sparsity:
    fig, ax = plt.subplots()
    ax.spy(mtx_a, marker='.', ms=4)
    fig.tight_layout()
    fig.savefig(inudir(get_figname('K-sparsity', options)),
                bbox_inches='tight')

    fig, ax = plt.subplots()
    ax.spy(mtx_b, marker='.', ms=4)
    fig.tight_layout()
    fig.savefig(inudir(get_figname('M-sparsity', options)),
                bbox_inches='tight')

if options.show:
    plt.show()

if options.shell:
    from sfepy.base.base import shell; shell()

if __name__ == '__main__':
    main()

```

multi_physics

multi_physics/biot.py

Description

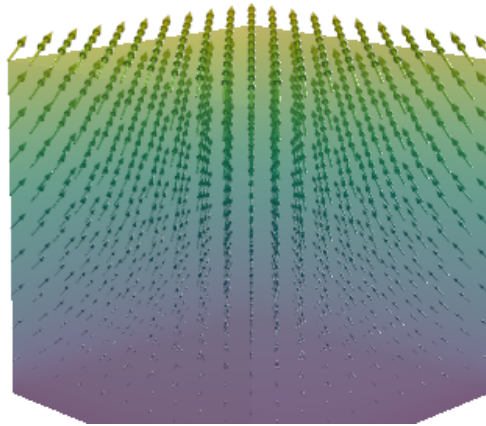
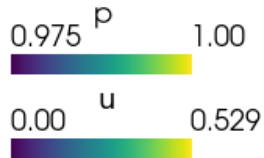
Biot problem - deformable porous medium.

Find $\underline{u}$, p such that:

$$\begin{aligned} \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) + \int_{\Omega} K_{ij} \nabla_i q \nabla_j p &= 0, \quad \forall q, \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```
r"""
Biot problem - deformable porous medium.

Find :math:\underline{u}, :math:p` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\underline{v}) \ e_{kl}(\underline{u})
    - \int_{\Omega} p \ \alpha_{ij} \ e_{ij}(\underline{v})
    = 0
    \;, \quad \forall \underline{v} \;
```

(continues on next page)

(continued from previous page)

```

\int_{\Omega} q \ \alpha_{ij} e_{ij}(\ul{u})
+ \int_{\Omega} K_{ij} \ \nabla_i q \ \nabla_j p
= 0
\;; \quad \forall q \;;

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;;.
"""

from __future__ import absolute_import
import numpy as nm

from sfepy import data_dir
from sfepy.mechanics.matcoefs import stiffness_from_lame

filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'

regions = {
    'Omega' : 'all',
    'Bottom' : ('vertices in (z < -0.4999999)', 'facet'),
    'Top' : ('vertices in (z > 0.4999999)', 'facet'),
    'Left' : ('vertices in (x < -0.4999999)', 'facet'),
}

field_1 = {
    'name' : 'displacement',
    'dtype' : nm.float64,
    'shape' : (3,),
    'region' : 'Omega',
    'approx_order' : 1,
}

field_2 = {
    'name' : 'pressure',
    'dtype' : nm.float64,
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

ebcs = {
    'fix_u' : ('Bottom', {'u.all' : 0.0}),

```

(continues on next page)

(continued from previous page)

```

    'load_u' : ('Top', {'u.2' : 0.2}),
    'load_p' : ('Left', {'p.all' : 1.0}),
}

material_1 = {
    'name' : 'm',
    'values' : {
        'D': stiffness_from_lame(dim=3, lam=1.7, mu=0.3),
        'alpha' : nm.array( [[0.132], [0.132], [0.132],
                             [0.092], [0.092], [0.092]],
                             dtype = nm.float64 ),
        'K' : nm.array( [[2.0, 0.2, 0.0], [0.2, 1.0, 0.0], [0.0, 0.0, 0.5]],
                         dtype = nm.float64 ),
    }
}

integral_1 = {
    'name' : 'i1',
    'order' : 1,
}

integral_2 = {
    'name' : 'i2',
    'order' : 2,
}

equations = {
    'eq_1' :
        """dw_lin_elastic.i2.Omega( m.D, v, u )
        - dw_biot.i1.Omega( m.alpha, v, p )
        = 0""",
    'eq_2' :
        """dw_biot.i1.Omega( m.alpha, u, q ) + dw_diffusion.i1.Omega( m.K, q, p )
        = 0""",
}

solver_0 = {
    'name' : 'ls_d',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 1,
    'eps_a'      : 1e-10,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
}

```

(continues on next page)

(continued from previous page)

```

'ls_on'      : 1.1,
'ls_min'     : 1e-5,
'check'      : 0,
'delta'      : 1e-6,
}

```

multi_physics/biot_npbc.py

Description

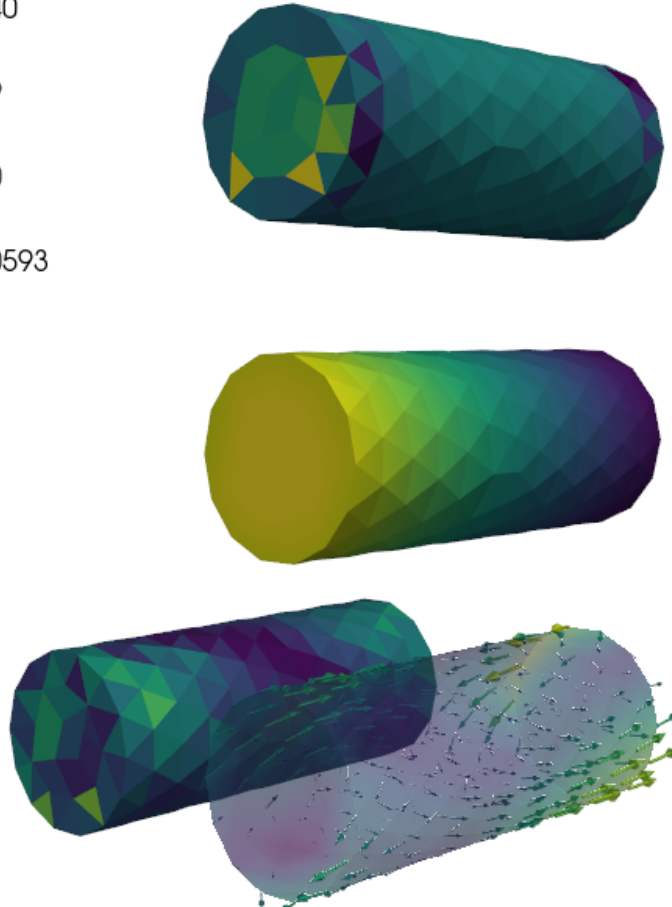
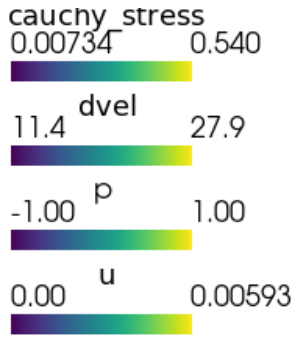
Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region.

Find $\underline{u}$, p such that:

$$\begin{aligned}
 \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) &= 0, \quad \forall \underline{v}, \\
 \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) + \int_{\Omega} K_{ij} \nabla_i q \nabla_j p &= 0, \quad \forall q, \\
 \underline{u} \cdot \underline{n} &= 0 \text{ on } \Gamma_{walls},
 \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```

r"""
Biot problem - deformable porous medium with the no-penetration boundary
condition on a boundary region.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
    - \int_{\Omega} p \alpha_{ij} \epsilon_{ij}(\mathbf{v})
    = 0
    \quad \forall \mathbf{v};

    \int_{\Omega} q \alpha_{ij} \epsilon_{ij}(\mathbf{u})
    + \int_{\Omega} K_{ij} \nabla_i q \nabla_j p
    = 0
    \quad \forall q;

    \mathbf{u} \cdot \mathbf{n} = 0 \quad \text{on } \Gamma_{\text{walls}};

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad .
"""

from __future__ import absolute_import
import os
import numpy as nm

from sfepy.linalg import get_coors_in_tube
from sfepy.mechanics.matcoefs import stiffness_from_lame

def define():
    from sfepy import data_dir

    filename = data_dir + '/meshes/3d/cylinder.mesh'
    output_dir = 'output'
    return define_input(filename, output_dir)

def cinc_simple(coors, mode):
    axis = nm.array([1, 0, 0], nm.float64)
    if mode == 0: # In
        centre = nm.array([0.0, 0.0, 0.0], nm.float64)
        radius = 0.019
        length = 0.00002
    elif mode == 1: # Out
        centre = nm.array([0.1, 0.0, 0.0], nm.float64)
        radius = 0.019
        length = 0.00002
    elif mode == 2: # Rigid

```

(continues on next page)

(continued from previous page)

```

        centre = nm.array([0.05, 0.0, 0.0], nm.float64)
        radius = 0.015
        length = 0.03
    else:
        raise ValueError('unknown mode %s!' % mode)

    return get_coors_in_tube(coors,
                            centre, axis, -1, radius, length)

def define_regions(filename):
    if filename.find('simple.mesh'):
        dim = 3
        regions = {
            'Omega' : 'all',
            'Walls' : ('vertices of surface -v (r.Outlet +f r.Inlet)', 'facet'),
            'Inlet' : ('vertices by cinc_simple0', 'facet'),
            'Outlet' : ('vertices by cinc_simple1', 'facet'),
            'Rigid' : 'vertices by cinc_simple2',
        }
    else:
        raise ValueError('unknown mesh %s!' % filename)

    return regions, dim

def get_pars(ts, coor, mode, output_dir='.', **kwargs):
    if mode == 'qp':
        n_nod, dim = coor.shape
        sym = (dim + 1) * dim // 2

        out = {}
        out['D'] = nm.tile(stiffness_from_lame(dim, lam=1.7, mu=0.3),
                           (coor.shape[0], 1, 1))

        aa = nm.zeros((sym, 1), dtype=nm.float64)
        aa[:dim] = 0.132
        aa[dim:sym] = 0.092
        out['alpha'] = nm.tile(aa, (coor.shape[0], 1, 1))

        perm = nm.eye(dim, dtype=nm.float64)
        out['K'] = nm.tile(perm, (coor.shape[0], 1, 1))

    return out

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    dvel = pb.evaluate('ev_diffusion_velocity.i.Omega( m.K, p )',
                       mode='el_avg')
    out['dvel'] = Struct(name='output_data',
                        mode='cell', data=dvel, dofs=None)

```

(continues on next page)

(continued from previous page)

```

stress = pb.evaluate('ev_cauchy_stress.i.Omega( m.D, u )',
                    mode='el_avg')
out['cauchy_stress'] = Struct(name='output_data',
                             mode='cell', data=stress, dofs=None)

return out

def define_input(filename, output_dir):

    filename_mesh = filename
    options = {
        'output_dir' : output_dir,
        'output_format' : 'vtk',
        'post_process_hook' : 'post_process',

        'ls' : 'ls',
        'nls' : 'newton',
    }

    functions = {
        'cinc_simple0' : (lambda coors, domain:
                        cinc_simple(coors, 0)),
        'cinc_simple1' : (lambda coors, domain:
                        cinc_simple(coors, 1)),
        'cinc_simple2' : (lambda coors, domain:
                        cinc_simple(coors, 2)),
        'get_pars' : (lambda ts, coors, mode=None, **kwargs:
                        get_pars(ts, coors, mode,
                                output_dir=output_dir, **kwargs)),
    }

    regions, dim = define_regions(filename_mesh)

    field_1 = {
        'name' : 'displacement',
        'dtype' : nm.float64,
        'shape' : dim,
        'region' : 'Omega',
        'approx_order' : 1,
    }

    field_2 = {
        'name' : 'pressure',
        'dtype' : nm.float64,
        'shape' : 1,
        'region' : 'Omega',
        'approx_order' : 1,
    }

    variables = {
        'u' : ('unknown field', 'displacement', 0),
        'v' : ('test field', 'displacement', 'u'),
        'p' : ('unknown field', 'pressure', 1),
        'q' : ('test field', 'pressure', 'p'),
    }

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    'inlet' : ('Inlet', {'p.0' : 1.0, 'u.all' : 0.0}),
    'outlet' : ('Outlet', {'p.0' : -1.0}),
}

lcbcs = {
    'rigid' : ('Outlet', {'u.all' : None}, None, 'rigid'),
    'no_penetration' : ('Walls', {'u.all' : None}, None,
                        'no_penetration', None),
}

material_1 = {
    'name' : 'm',
    'function' : 'get_pars',
}

integral_1 = {
    'name' : 'i',
    'order' : 2,
}

equations = {
    'eq_1' :
        """dw_lin_elastic.i.Omega( m.D, v, u )
        - dw_biot.i.Omega( m.alpha, v, p )
        = 0""",
    'eq_2' :
        """dw_biot.i.Omega( m.alpha, u, q )
        + dw_diffusion.i.Omega( m.K, q, p )
        = 0""",
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct', # Direct solver.
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',
}

return locals()

```

multi_physics/biot_npbclagrange.py**Description**

Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region enforced using Lagrange multipliers.

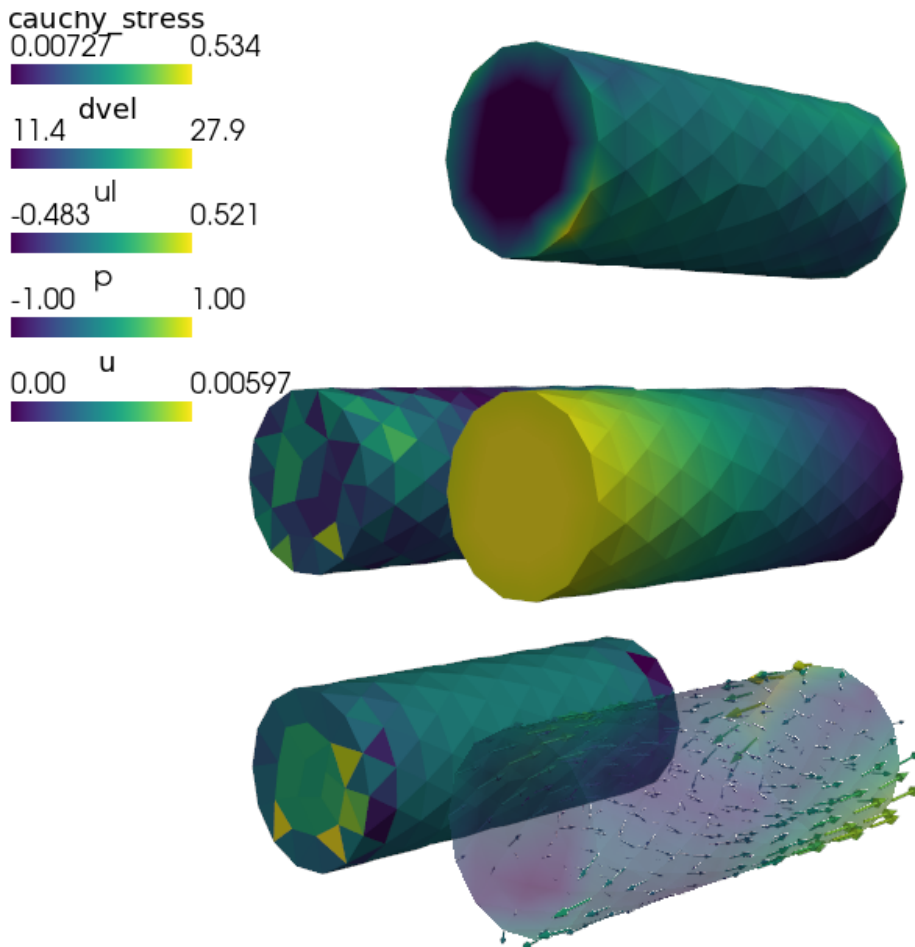
The non-penetration condition is enforced weakly using the Lagrange multiplier λ . There is also a rigid body movement constraint imposed on the Γ_{outlet} region using the linear combination boundary conditions.

Find $\underline{u}$, p and λ such that:

$$\begin{aligned} \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) + \int_{\Gamma_{walls}} \lambda \underline{n} \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) + \int_{\Omega} K_{ij} \nabla_i q \nabla_j p &= 0, \quad \forall q, \\ \int_{\Gamma_{walls}} \hat{\lambda} \underline{n} \cdot \underline{u} &= 0, \quad \forall \hat{\lambda}, \\ \underline{u} \cdot \underline{n} &= 0 \text{ on } \Gamma_{walls}, \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



[source code](#)


```

r"""
Biot problem - deformable porous medium with the no-penetration boundary
condition on a boundary region enforced using Lagrange multipliers.

The non-penetration condition is enforced weakly using the Lagrange
multiplier  $\lambda$ . There is also a rigid body movement
constraint imposed on the  $\Gamma_{\text{outlet}}$  region using the
linear combination boundary conditions.

Find  $u$ ,  $p$  and  $\lambda$  such that:

.. math::
\int_{\Omega} D_{ijkl} e_{ij}(u,v) e_{kl}(u)
- \int_{\Omega} p \alpha_{ij} e_{ij}(u,v)
+ \int_{\Gamma_{\text{walls}}} \lambda u_n \cdot u_v
= 0
\;, \quad \forall v \;,

\int_{\Omega} q \alpha_{ij} e_{ij}(u)
+ \int_{\Omega} K_{ij} \nabla_i q \nabla_j p
= 0
\;, \quad \forall q \;,

\int_{\Gamma_{\text{walls}}} \hat{\lambda} u_n \cdot u
= 0
\;, \quad \forall \hat{\lambda} \;,

u \cdot u_n = 0 \quad \text{on } \Gamma_{\text{walls}} \;,

where

.. math::
D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
\lambda \delta_{ij} \delta_{kl}
\;.
"""
from __future__ import absolute_import
from sfepy.examples.multi_physics.biot_npb import (cinc_simple,
                                                    define_regions, get_pars)

def define():
    from sfepy import data_dir

    filename = data_dir + '/meshes/3d/cylinder.mesh'
    output_dir = 'output'
    return define_input(filename, output_dir)

def post_process(out, pb, state, extend=False):
    from sfepy.base.base import Struct

    dvel = pb.evaluate('ev_diffusion_velocity.2.Omega( m.K, p )',
                       mode='el_avg')
    out['dvel'] = Struct(name='output_data', var_name='p',

```

(continues on next page)

(continued from previous page)

```

        mode='cell', data=dvel, dofs=None)

stress = pb.evaluate('ev_cauchy_stress.2.Omega( m.D, u )',
                    mode='el_avg')
out['cauchy_stress'] = Struct(name='output_data', var_name='u',
                             mode='cell', data=stress, dofs=None)

return out

def define_input(filename, output_dir):

    filename_mesh = filename
    options = {
        'output_dir' : output_dir,
        'output_format' : 'vtk',
        'post_process_hook' : 'post_process',
        'ls' : 'ls',
        'nls' : 'newton',
    }

    functions = {
        'cinc_simple0' : (lambda coors, domain:
                        cinc_simple(coors, 0)),
        'cinc_simple1' : (lambda coors, domain:
                        cinc_simple(coors, 1)),
        'cinc_simple2' : (lambda coors, domain:
                        cinc_simple(coors, 2)),
        'get_pars' : (lambda ts, coors, mode=None, **kwargs:
                        get_pars(ts, coors, mode,
                                output_dir=output_dir, **kwargs)),
    }

    regions, dim = define_regions(filename_mesh)

    fields = {
        'displacement': ('real', 'vector', 'Omega', 1),
        'pressure': ('real', 'scalar', 'Omega', 1),
        'multiplier': ('real', 'scalar', 'Walls', 1),
    }

    variables = {
        'u' : ('unknown field', 'displacement', 0),
        'v' : ('test field', 'displacement', 'u'),
        'p' : ('unknown field', 'pressure', 1),
        'q' : ('test field', 'pressure', 'p'),
        'ul' : ('unknown field', 'multiplier', 2),
        'vl' : ('test field', 'multiplier', 'ul'),
    }

    ebcs = {
        'inlet' : ('Inlet', {'p.0' : 1.0, 'u.all' : 0.0}),
        'outlet' : ('Outlet', {'p.0' : -1.0}),
    }

```

(continues on next page)

(continued from previous page)

```

lcbcs = {
    'rigid' : ('Outlet', {'u.all' : None}, None, 'rigid'),
}

materials = {
    'm' : 'get_pars',
}

equations = {
    'eq_1' :
        """dw_lin_elastic.2.Omega( m.D, v, u )
        - dw_biot.2.Omega( m.alpha, v, p )
        + dw_non_penetration.2.Walls( v, ul )
        = 0""",
    'eq_2' :
        """dw_biot.2.Omega( m.alpha, u, q )
        + dw_diffusion.2.Omega( m.K, q, p )
        = 0""",
    'eq_3' :
        """dw_non_penetration.2.Walls( u, vl )
        = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {}),
}

return locals()

```

multi_physics/biot_parallel_interactive.py

Description

Parallel assembling and solving of a Biot problem (deformable porous medium), using commands for interactive use.

Find $\underline{u}$, p such that:

$$\begin{aligned}
 \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) &= 0, \quad \forall \underline{v}, \\
 \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) + \int_{\Omega} K_{ij} \nabla_i q \nabla_j p &= 0, \quad \forall q,
 \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

Important Notes

- This example requires petsc4py, mpi4py and (optionally) pymetis with their dependencies installed!
- This example generates a number of files - do not use an existing non-empty directory for the `output_dir` argument.
- Use the `--clear` option with care!

Notes

- Each task is responsible for a subdomain consisting of a set of cells (a cell region).
- Each subdomain owns PETSc DOFs within a consecutive range.
- When both global and task-local variables exist, the task-local variables have `_i` suffix.
- This example shows how to use a nonlinear solver from PETSc.
- This example can serve as a template for solving a (non)linear multi-field problem - just replace the equations in `create_local_problem()`.
- The material parameter α_{ij} is artificially high to be able to see the pressure influence on displacements.
- The command line options are saved into `<output_dir>/options.txt` file.

Usage Examples

See all options:

```
python3 sfepy/examples/multi_physics/biot_parallel_interactive.py -h
```

See PETSc options:

```
python3 sfepy/examples/multi_physics/biot_parallel_interactive.py -help
```

Single process run useful for debugging with `debug()`:

```
python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel
```

Parallel runs:

```
mpiexec -n 3 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-
↪parallel -2 --shape=101,101

mpiexec -n 3 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-
↪parallel -2 --shape=101,101 --metis

mpiexec -n 8 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-
↪parallel -2 --shape 101,101 --metis -snes_monitor -snes_view -snes_converged_reason -
↪ksp_monitor
```

Using FieldSplit preconditioner:

```

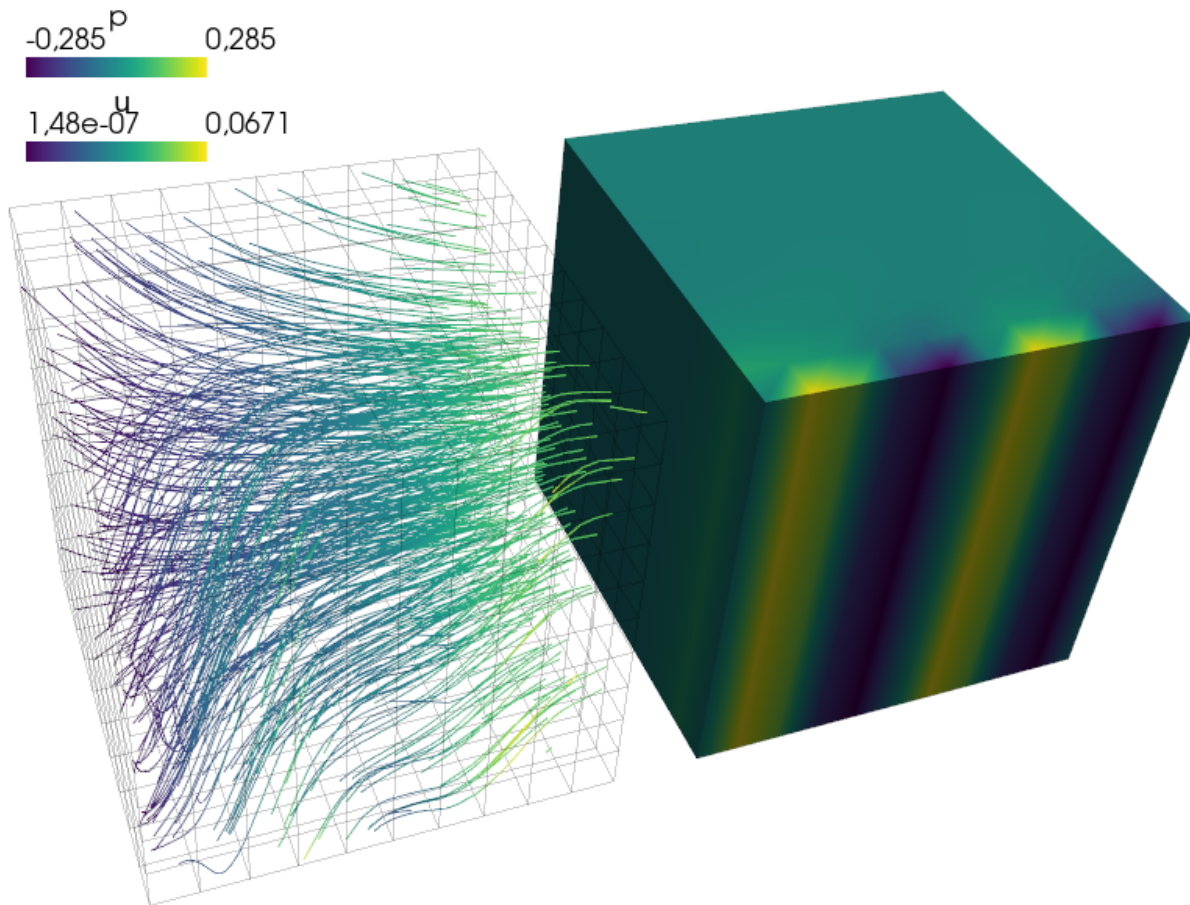
mpirun -n 2 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-
→parallel --shape=101,101 -snes_monitor -snes_converged_reason -ksp_monitor -pc_type_
→fieldsplit

mpirun -n 8 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-
→parallel --shape=1001,1001 --metis -snes_monitor -snes_converged_reason -ksp_monitor -
→pc_type fieldsplit -pc_fieldsplit_type additive

```

View the results using:

```
sfepy-view output-parallel/sol.h5
```



source code

```

#!/usr/bin/env python
r"""
Parallel assembling and solving of a Biot problem (deformable porous medium),
using commands for interactive use.

Find  $\ul{u}$ ,  $p$  such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\ul{v}) e_{kl}(\ul{u})

```

(continues on next page)

(continued from previous page)

```

- \int_{\Omega} p \alpha_{ij} e_{ij}(\mathbf{v})
= 0
\;, \quad \forall \mathbf{v} \;,

\int_{\Omega} q \alpha_{ij} e_{ij}(\mathbf{u})
+ \int_{\Omega} K_{ij} \nabla_i q \nabla_j p
= 0
\;, \quad \forall q \;,

```

where

```

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.

```

Important Notes

- This example requires `petsc4py`, `mpi4py` and (optionally) `pymetis` with their dependencies installed!
- This example generates a number of files - do not use an existing non-empty directory for the `output_dir` argument.
- Use the `--clear` option with care!

Notes

- Each task is responsible for a subdomain consisting of a set of cells (a cell region).
- Each subdomain owns PETSc DOFs within a consecutive range.
- When both global and task-local variables exist, the task-local variables have `_i` suffix.
- This example shows how to use a nonlinear solver from PETSc.
- This example can serve as a template for solving a (non)linear multi-field problem - just replace the equations in `:func: create_local_problem()`.
- The material parameter `alpha_{ij}` is artificially high to be able to see the pressure influence on displacements.
- The command line options are saved into `<output_dir>/options.txt` file.

Usage Examples

See all options::

```
python3 sfepy/examples/multi_physics/biot_parallel_interactive.py -h
```

See PETSc options::

```
python3 sfepy/examples/multi_physics/biot_parallel_interactive.py -help
```

Single process run useful for debugging with `:func: debug()`

(continues on next page)

(continued from previous page)

`<sfepy.base.base.debug>`::``python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel``Parallel runs::``mpirun -n 3 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel -2 --shape=101,101``mpirun -n 3 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel -2 --shape=101,101 --metis``mpirun -n 8 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel -2 --shape 101,101 --metis -snes_monitor -snes_view -snes_converged_reason -ksp_monitor``Using FieldSplit preconditioner::``mpirun -n 2 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel --shape=101,101 -snes_monitor -snes_converged_reason -ksp_monitor -pc_type fieldsplit``mpirun -n 8 python3 sfepy/examples/multi_physics/biot_parallel_interactive.py output-parallel --shape=1001,1001 --metis -snes_monitor -snes_converged_reason -ksp_monitor -pc_type fieldsplit -pc_fieldsplit_type additive``View the results using::``sfepy-view output-parallel/sol.h5``"""`

```
from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser
import os
import sys
sys.path.append('.')
```

```
import numpy as nm
```

```
from sfepy.base.base import output, Struct
from sfepy.base.ioutils import ensure_path, remove_files_patterns, save_options
from sfepy.base.timing import Timer
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.discrete.common.region import Region
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                             Equation, Equations, Problem)
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.terms import Term
from sfepy.solvers.ls import PETScKrylovSolver
from sfepy.solvers.nls import PETScNonlinearSolver
from sfepy.mechanics.matcoefs import stiffness_from_lame

import sfepy.parallel.parallel as pl
```

(continues on next page)

(continued from previous page)

```

from sfepy.parallel.evaluate import PETScParallelEvaluator

def create_local_problem(omega_gi, orders):
    """
    Local problem definition using a domain corresponding to the global region
    `omega_gi`.
    """
    order_u, order_p = orders

    mesh = omega_gi.domain.mesh

    # All tasks have the whole mesh.
    bbox = mesh.get_bounding_box()
    min_x, max_x = bbox[:, 0]
    eps_x = 1e-8 * (max_x - min_x)

    min_y, max_y = bbox[:, 1]
    eps_y = 1e-8 * (max_y - min_y)

    mesh_i = Mesh.from_region(omega_gi, mesh, localize=True)
    domain_i = FEDomain('domain_i', mesh_i)
    omega_i = domain_i.create_region('Omega', 'all')

    gamma1_i = domain_i.create_region('Gamma1',
                                      'vertices in (x < %.10f)'
                                      % (min_x + eps_x),
                                      'facet', allow_empty=True)
    gamma2_i = domain_i.create_region('Gamma2',
                                      'vertices in (x > %.10f)'
                                      % (max_x - eps_x),
                                      'facet', allow_empty=True)
    gamma3_i = domain_i.create_region('Gamma3',
                                      'vertices in (y < %.10f)'
                                      % (min_y + eps_y),
                                      'facet', allow_empty=True)

    field1_i = Field.from_args('fu', nm.float64, mesh.dim, omega_i,
                              approx_order=order_u)

    field2_i = Field.from_args('fp', nm.float64, 1, omega_i,
                              approx_order=order_p)

    output('field 1: number of local DOFs:', field1_i.n_nod)
    output('field 2: number of local DOFs:', field2_i.n_nod)

    u_i = FieldVariable('u_i', 'unknown', field1_i, order=0)
    v_i = FieldVariable('v_i', 'test', field1_i, primary_var_name='u_i')
    p_i = FieldVariable('p_i', 'unknown', field2_i, order=1)
    q_i = FieldVariable('q_i', 'test', field2_i, primary_var_name='p_i')

    if mesh.dim == 2:
        alpha = 1e2 * nm.array([[0.132], [0.132], [0.092]])

```

(continues on next page)

(continued from previous page)

```

else:
    alpha = 1e2 * nm.array([[0.132], [0.132], [0.132],
                           [0.092], [0.092], [0.092]])

mat = Material('m', D=stiffness_from_lame(mesh.dim, lam=10, mu=5),
              k=1, alpha=alpha)
integral = Integral('i', order=2*(max(order_u, order_p)))

t11 = Term.new('dw_lin_elastic(m.D, v_i, u_i)',
               integral, omega_i, m=mat, v_i=v_i, u_i=u_i)
t12 = Term.new('dw_biot(m.alpha, v_i, p_i)',
               integral, omega_i, m=mat, v_i=v_i, p_i=p_i)
t21 = Term.new('dw_biot(m.alpha, u_i, q_i)',
               integral, omega_i, m=mat, u_i=u_i, q_i=q_i)
t22 = Term.new('dw_laplace(m.k, q_i, p_i)',
               integral, omega_i, m=mat, q_i=q_i, p_i=p_i)

eq1 = Equation('eq1', t11 - t12)
eq2 = Equation('eq1', t21 + t22)
eqs = Equations([eq1, eq2])

ebc1 = EssentialBC('ebc1', gamma1_i, {'u_i.all' : 0.0})
ebc2 = EssentialBC('ebc2', gamma2_i, {'u_i.0' : 0.05})
def bc_fun(ts, coors, **kwargs):
    val = 0.3 * nm.sin(4 * nm.pi * (coors[:, 0] - min_x) / (max_x - min_x))
    return val

fun = Function('bc_fun', bc_fun)
ebc3 = EssentialBC('ebc3', gamma3_i, {'p_i.all' : fun})

pb = Problem('problem_i', equations=eqs, active_only=False)
pb.time_update(ebcs=Conditions([ebc1, ebc2, ebc3]))
pb.update_materials()

return pb

def solve_problem(mesh_filename, options, comm):
    order_u = options.order_u
    order_p = options.order_p

    rank, size = comm.Get_rank(), comm.Get_size()

    output('rank', rank, 'of', size)

    stats = Struct()
    timer = Timer('solve_timer')

    timer.start()
    mesh = Mesh.from_file(mesh_filename)
    stats.t_read_mesh = timer.stop()

```

(continues on next page)

(continued from previous page)

```

timer.start()
if rank == 0:
    cell_tasks = pl.partition_mesh(mesh, size, use_metis=options.metis,
                                   verbose=True)

else:
    cell_tasks = None

stats.t_partition_mesh = timer.stop()

output('creating global domain and fields...')
timer.start()

domain = FEDomain('domain', mesh)
omega = domain.create_region('Omega', 'all')
field1 = Field.from_args('fu', nm.float64, mesh.dim, omega,
                        approx_order=order_u)
field2 = Field.from_args('fp', nm.float64, 1, omega,
                        approx_order=order_p)
fields = [field1, field2]

stats.t_create_global_fields = timer.stop()
output('...done in', timer.dt)

output('distributing fields...')
timer.start()

distribute = pl.distribute_fields_dofs
lfds, gfds = distribute(fields, cell_tasks,
                        is_overlap=True,
                        use_expand_dofs=True,
                        save_inter_regions=options.save_inter_regions,
                        output_dir=options.output_dir,
                        comm=comm, verbose=True)

stats.t_distribute_fields_dofs = timer.stop()
output('...done in', timer.dt)

output('creating local problem...')
timer.start()

cells = lfds[0].cells

omega_gi = Region.from_cells(cells, domain)
omega_gi.finalize()
omega_gi.update_shape()

pb = create_local_problem(omega_gi, [order_u, order_p])

variables = pb.get_initial_state()

variables.fill_state(0.0)

```

(continues on next page)

(continued from previous page)

```

variables.apply_ebc()

stats.t_create_local_problem = timer.stop()
output('...done in', timer.dt)

output('allocating global system...')
timer.start()

sizes, drange, pdofs = pl.setup_composite_dofs(lfds, fields, variables,
                                              verbose=True)
pmtx, psol, prhs = pl.create_petsc_system(pb.mtx_a, sizes, pdofs, drange,
                                         is_overlap=True, comm=comm,
                                         verbose=True)

stats.t_allocate_global_system = timer.stop()
output('...done in', timer.dt)

output('creating solver...')
timer.start()

conf = Struct(method='bcgsl', precondition='jacobi', sub_precondition='none',
              i_max=10000, eps_a=1e-50, eps_r=1e-6, eps_d=1e4,
              verbose=True)
status = {}
ls = PETScKrylovSolver(conf, comm=comm, mtx=pmtx, status=status)

field_ranges = {}
for ii, variable in enumerate(variables.iter_state(ordered=True)):
    field_ranges[variable.name] = lfds[ii].petsc_dofs_range

ls.set_field_split(field_ranges, comm=comm)

ev = PETScParallelEvaluator(pb, pdofs, drange, True,
                           psol, comm, verbose=True)

nls_status = {}
conf = Struct(method='newtonls',
              i_max=5, eps_a=0, eps_r=1e-5, eps_s=0.0,
              verbose=True)
nls = PETScNonlinearSolver(conf, pmtx=pmtx, prhs=prhs, comm=comm,
                           fun=ev.eval_residual,
                           fun_grad=ev.eval_tangent_matrix,
                           lin_solver=ls, status=nls_status)

stats.t_create_solver = timer.stop()
output('...done in', timer.dt)

output('solving...')
timer.start()

variables.apply_ebc()

```

(continues on next page)

(continued from previous page)

```

ev.psoli[...] = variables()
ev.gather(psol, ev.psoli)

psol = nls(psol)

ev.scatter(ev.psoli, psol)
sol0_i = ev.psoli[...]

stats.t_solve = timer.stop()
output('...done in', timer.dt)

output('saving solution...')
timer.start()

variables.set_state(sol0_i)
out = variables.create_output()

filename = os.path.join(options.output_dir, 'sol_%02d.h5' % comm.rank)
pb.domain.mesh.write(filename, io='auto', out=out)

gather_to_zero = pl.create_gather_to_zero(psol)

psol_full = gather_to_zero(psol)

if comm.rank == 0:
    sol = psol_full[...].copy()

    u = FieldVariable('u', 'parameter', field1,
                      primary_var_name='(set-to-None)')
    remap = gfds[0].id_map
    ug = sol[remap]

    p = FieldVariable('p', 'parameter', field2,
                      primary_var_name='(set-to-None)')
    remap = gfds[1].id_map
    pg = sol[remap]

    if (((order_u == 1) and (order_p == 1))
        or (options.linearization == 'strip')):
        out = u.create_output(ug)
        out.update(p.create_output(pg))
        filename = os.path.join(options.output_dir, 'sol.h5')
        mesh.write(filename, io='auto', out=out)

    else:
        out = u.create_output(ug, linearization=Struct(kind='adaptive',
                                                       min_level=0,
                                                       max_level=order_u,
                                                       eps=1e-3))

        filename = os.path.join(options.output_dir, 'sol_u.h5')
        out['u'].mesh.write(filename, io='auto', out=out)

```

(continues on next page)

(continued from previous page)

```

        out = p.create_output(pg, linearization=Struct(kind='adaptive',
                                                    min_level=0,
                                                    max_level=order_p,
                                                    eps=1e-3))

        filename = os.path.join(options.output_dir, 'sol_p.h5')
        out['p'].mesh.write(filename, io='auto', out=out)

    stats.t_save_solution = timer.stop()
    output('...done in', timer.dt)

    stats.t_total = timer.total

    stats.n_dof = sizes[1]
    stats.n_dof_local = sizes[0]
    stats.n_cell = omega.shape.n_cell
    stats.n_cell_local = omega_gi.shape.n_cell

    return stats

helps = {
    'output_dir' :
    'output directory',
    'dims' :
    'dimensions of the block [default: %(default)s]',
    'shape' :
    'shape (counts of nodes in x, y, z) of the block [default: %(default)s]',
    'centre' :
    'centre of the block [default: %(default)s]',
    '2d' :
    'generate a 2D rectangle, the third components of the above'
    ' options are ignored',
    'u-order' :
    'displacement field approximation order',
    'p-order' :
    'pressure field approximation order',
    'linearization' :
    'linearization used for storing the results with approximation order > 1'
    '[default: %(default)s]',
    'metis' :
    'use metis for domain partitioning',
    'save_inter_regions' :
    'save inter-task regions for debugging partitioning problems',
    'stats_filename' :
    'name of the stats file for storing elapsed time statistics',
    'new_stats' :
    'create a new stats file with a header line (overwrites existing!)',
    'silent' : 'do not print messages to screen',
    'clear' :
    'clear old solution files from output directory'
    '(DANGEROUS - use with care!)',

```

(continues on next page)

(continued from previous page)

```

}

def main():
    parser = ArgumentParser(description=__doc__.rstrip(),
                             formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('output_dir', help=helps['output_dir'])
    parser.add_argument('--dims', metavar='dims',
                        action='store', dest='dims',
                        default='1.0,1.0,1.0', help=helps['dims'])
    parser.add_argument('--shape', metavar='shape',
                        action='store', dest='shape',
                        default='11,11,11', help=helps['shape'])
    parser.add_argument('--centre', metavar='centre',
                        action='store', dest='centre',
                        default='0.0,0.0,0.0', help=helps['centre'])
    parser.add_argument('-2', '--2d',
                        action='store_true', dest='is_2d',
                        default=False, help=helps['2d'])
    parser.add_argument('--u-order', metavar='int', type=int,
                        action='store', dest='order_u',
                        default=1, help=helps['u-order'])
    parser.add_argument('--p-order', metavar='int', type=int,
                        action='store', dest='order_p',
                        default=1, help=helps['p-order'])
    parser.add_argument('--linearization', choices=['strip', 'adaptive'],
                        action='store', dest='linearization',
                        default='strip', help=helps['linearization'])
    parser.add_argument('--metis',
                        action='store_true', dest='metis',
                        default=False, help=helps['metis'])
    parser.add_argument('--save-inter-regions',
                        action='store_true', dest='save_inter_regions',
                        default=False, help=helps['save_inter_regions'])
    parser.add_argument('--stats', metavar='filename',
                        action='store', dest='stats_filename',
                        default=None, help=helps['stats_filename'])
    parser.add_argument('--new-stats',
                        action='store_true', dest='new_stats',
                        default=False, help=helps['new_stats'])
    parser.add_argument('--silent',
                        action='store_true', dest='silent',
                        default=False, help=helps['silent'])
    parser.add_argument('--clear',
                        action='store_true', dest='clear',
                        default=False, help=helps['clear'])
    options, petsc_opts = parser.parse_known_args()

    comm = pl.PETSc.COMM_WORLD

    output_dir = options.output_dir

    filename = os.path.join(output_dir, 'output_log-%02d.txt' % comm.rank)

```

(continues on next page)

(continued from previous page)

```

if comm.rank == 0:
    ensure_path(filename)
comm.barrier()

output.prefix = 'sfepy_%02d:' % comm.rank
output.set_output(filename=filename, combined=options.silent == False)

output('petsc options:', petsc_opts)

mesh_filename = os.path.join(options.output_dir, 'para.h5')

dim = 2 if options.is_2d else 3
dims = nm.array(eval(options.dims), dtype=nm.float64)[:dim]
shape = nm.array(eval(options.shape), dtype=nm.int32)[:dim]
centre = nm.array(eval(options.centre), dtype=nm.float64)[:dim]

output('dimensions:', dims)
output('shape:      ', shape)
output('centre:     ', centre)

if comm.rank == 0:
    from sfepy.mesh.mesh_generators import gen_block_mesh

    if options.clear:
        remove_files_patterns(output_dir,
                               ['*.h5', '*.mesh', '*.txt'],
                               ignores=['output_log_%02d.txt' % ii
                                       for ii in range(comm.size)],
                               verbose=True)

    save_options(os.path.join(output_dir, 'options.txt'),
                 [('options', vars(options))])

    mesh = gen_block_mesh(dims, shape, centre, name='block-fem',
                          verbose=True)
    mesh.write(mesh_filename, io='auto')

comm.barrier()

output('field u order:', options.order_u)
output('field p order:', options.order_p)

stats = solve_problem(mesh_filename, options, comm)
output(stats)

if options.stats_filename:
    from sfepy.examples.diffusion.poisson_parallel_interactive import save_stats
    if comm.rank == 0:
        ensure_path(options.stats_filename)
        comm.barrier()

    pars = Struct(dim=dim, shape=shape, order=options.order_u)

```

(continues on next page)

(continued from previous page)

```

    pl.call_in_rank_order(
        lambda rank, comm:
            save_stats(options.stats_filename, pars, stats, options.new_stats,
                       rank, comm),
        comm
    )

if __name__ == '__main__':
    main()

```

multi_physics/biot_short_syntax.py

Description

Biot problem - deformable porous medium with a no-penetration boundary condition imposed in the weak sense on a boundary region, using the short syntax of keywords.

The Biot coefficient tensor α_{ij} is non-symmetric. The mesh resolution can be changed by editing the *shape* variable.

This example demonstrates how to set up various linear solvers and preconditioners (see *solvers* dict):

- 'direct' (a direct solver from SciPy), 'iterative-s' (an iterative solver from SciPy), 'iterative-p' (an iterative solver from PETSc) solvers can be used as the main linear solver.
- 'direct', 'cg-s' (several iterations of CG from SciPy), 'cg-p' (several iterations of CG from PETSc), 'pyamg' (an algebraic multigrid solver) solvers can be used as preconditioners for the matrix blocks on the diagonal.

See `setup_precond()` and try to modify it.

The PETSc solvers can be configured also using command line options. For example, set 'ls' : 'iterative-p' in *options*, and run:

```
sfepy-run sfepy/examples/multi_physics/biot_short_syntax.py -ksp_monitor
```

or simply run:

```
sfepy-run sfepy/examples/multi_physics/biot_short_syntax.py -O "ls='iterative-p'"
```

to monitor the PETSc iterative solver convergence. It will diverge without preconditioning, see `matvec_bj()`, `matvec_j()` for further details.

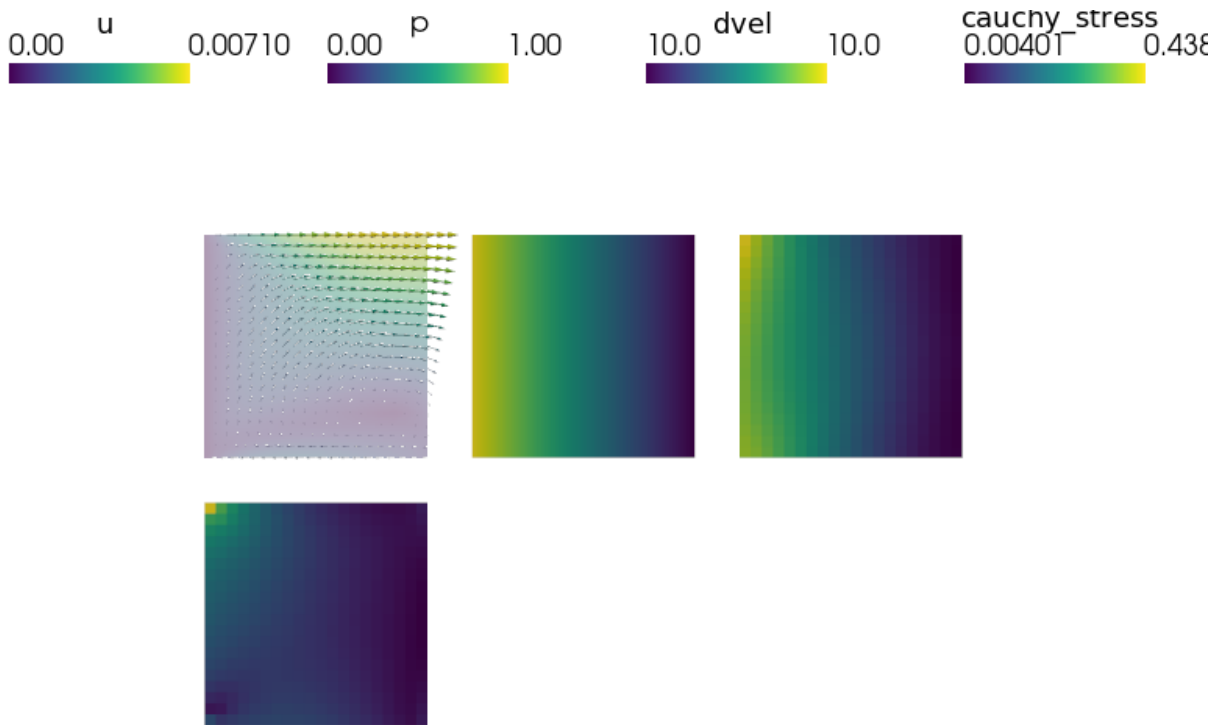
The PETSc options can also be set in the solver configuration - try uncommenting the 'ksp_*' or 'pc_*' parameters in 'iterative-p'. Uncommenting all the lines leads to, among other things, using the GMRES method with no preconditioning and the condition number estimate computation. Compare the condition number estimates with and without a preconditioning (try, for example, using 'precond' : 'mg' or 'pc_type' : 'mg').

Find $\underline{u}$, p such that:

$$\begin{aligned}
 \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) + \int_{\Gamma_{TB}} \varepsilon(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u}) &= 0, \quad \forall \underline{v}, \\
 - \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) - \int_{\Omega} K_{ij} \nabla_i q \nabla_j p &= 0, \quad \forall q,
 \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```
r"""
Biot problem - deformable porous medium with a no-penetration boundary
condition imposed in the weak sense on a boundary region, using the short
syntax of keywords.

The Biot coefficient tensor  $\alpha_{ij}$  is non-symmetric. The mesh
resolution can be changed by editing the `shape` variable.

This example demonstrates how to set up various linear solvers and
preconditioners (see `solvers` dict):

- `direct` (a direct solver from SciPy), `iterative-s` (an iterative solver
  from SciPy), `iterative-p` (an iterative solver from PETSc) solvers can be
  used as the main linear solver.
- `direct`, `cg-s` (several iterations of CG from SciPy), `cg-p` (several
  iterations of CG from PETSc), `pyamg` (an algebraic multigrid solver)
  solvers can be used as preconditioners for the matrix blocks on the diagonal.

See func:setup_precond() and try to modify it.

The PETSc solvers can be configured also using command line options. For
```

(continues on next page)

(continued from previous page)

example, set ``ls` : 'iterative-p`` in `options`, and run::

```
sfePy-run sfePy/examples/multi_physics/biot_short_syntax.py -ksp_monitor
```

or simply run::

```
sfePy-run sfePy/examples/multi_physics/biot_short_syntax.py -O "ls='iterative-p'"
```

to monitor the PETSc iterative solver convergence. It will diverge without preconditioning, see :func:`matvec\_bj()`, :func:`matvec\_j()` for further details.

The PETSc options can also be set in the solver configuration - try uncommenting the ``ksp\_\*`` or ``pc\_\*`` parameters in ``'iterative-p``. Uncommenting all the lines leads to, among other things, using the GMRES method with no preconditioning and the condition number estimate computation. Compare the condition number estimates with and without a preconditioning (try, for example, using ``'precond' : 'mg`` or ``'pc\_type' : 'mg``).

Find :math:\ul{u}, :math:p` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    - \int_{\Omega} p \ \alpha_{ij} \ e_{ij}(\ul{v})
    + \int_{\Gamma_{TB}} \ \varepsilon \ (\ul{n} \cdot \ul{v}) \ (\ul{n} \cdot \ul{u})
    = 0
    \;, \quad \forall \ul{v} \;,

    - \int_{\Omega} q \ \alpha_{ij} \ e_{ij}(\ul{u})
    - \int_{\Omega} K_{ij} \ \nabla_i q \ \nabla_j p
    = 0
    \;, \quad \forall q \;,
```

where

```
.. math::
    D_{ijkl} = \mu \ (\delta_{ik} \ \delta_{jl} + \delta_{il} \ \delta_{jk}) +
    \lambda \ \delta_{ij} \ \delta_{kl}
    \;.
```

```
"""
```

```
from __future__ import absolute_import
import numpy as nm
```

```
from sfePy.base.base import Struct
from sfePy.mechanics.matcoefs import stiffness_from_lame
from sfePy.discrete.fem.meshio import UserMeshIO
from sfePy.mesh.mesh_generators import gen_block_mesh
```

```
def get_pars(ts, coor, mode, **kwargs):
    """
    Define the material parameters.
    """
```

(continues on next page)

(continued from previous page)

```

if mode == 'qp':
    n_nod, dim = coord.shape

    out = {}
    out['D'] = stiffness_from_lame(dim, lam=1.7, mu=0.3)[None, ...]

    out['alpha'] = nm.array([[[0.132, 0.092],
                               [0.052, 0.132]]])

    out['K'] = nm.eye(dim, dtype=nm.float64)[None, ...]

    out['np_eps'] = nm.array([[[[1e5]]]])

    return out

def post_process(out, pb, state, extend=False):
    """
    Compute derived quantities of interest..
    """
    from sfepy.base.base import Struct

    dvel = pb.evaluate('ev_diffusion_velocity.i.Omega(m.K, p)',
                       mode='el_avg')
    out['dvel'] = Struct(name='output_data',
                        mode='cell', data=dvel, dofs=None)

    stress = pb.evaluate('ev_cauchy_stress.i.Omega(m.D, u)',
                        mode='el_avg')
    out['cauchy_stress'] = Struct(name='output_data',
                                mode='cell', data=stress, dofs=None)

    return out

# Mesh dimensions.
dims = [0.1, 0.1]

# Mesh resolution: increase to improve accuracy.
shape = [21, 21]

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, [0, 0], name='user_block',
                              verbose=False)

        return mesh

    elif mode == 'write':
        pass

filename_mesh = UserMeshIO(mesh_hook)

```

(continues on next page)

(continued from previous page)

```

materials = {
    'coef' : ({'val' : 1.0},),
}

regions = {
    'Omega' : 'all', # or 'cells of group 6'
    'GammaL' : ('vertices in (x < -0.0499)', 'facet'),
    'GammaR' : ('vertices in (x > 0.0499)', 'facet'),
    'GammaTB' : ('vertices of surface -s (r.GammaL +s r.GammaR)', 'facet')
}

fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
    'pressure': ('real', 'scalar', 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

ebcs = {
    'inlet' : ('GammaL', {'p.0' : 1.0, 'u.all' : 0.0}),
    'outlet' : ('GammaR', {'p.0' : 0.0}),
}

integrals = {
    'i' : 2,
}

materials = {
    'm' : 'get_pars',
}

functions = {
    'get_pars' : (get_pars,),
}

equations = {
    'eq_1' :
        """+ dw_lin_elastic.i.Omega(m.D, v, u)
          - dw_biot.i.Omega(m.alpha, v, p)
          + dw_non_penetration_p.i.GammaTB(m.np_eps, v, u)
          = 0""",
    'eq_2' :
        """- dw_biot.i.Omega(m.alpha, u, q)
          - dw_diffusion.i.Omega(m.K, q, p)
          = 0""",
}

```

(continues on next page)

(continued from previous page)

```

def setup_precond(mtx, problem):
    """
    Setup a preconditioner for `mtx`.
    """
    import scipy.sparse.linalg as spla
    from sfepy.solvers import Solver

    # Get active DOF indices for u, p.
    adi = problem.get_variables().adi
    iu = adi.indx['u']
    ip = adi.indx['p']

    # Get the diagonal blocks of the linear system matrix.
    K = mtx[iu, iu]
    M = mtx[ip, ip]

    # Create solvers for K, M blocks to be used in matvec_bj(). A different
    # solver for each block could be used.
    conf = problem.solver_confs['direct']
    # conf = problem.solver_confs['cg-s']
    # conf = problem.solver_confs['cg-p']
    # conf = problem.solver_confs['pyamg']
    ls1 = Solver.any_from_conf(conf, mtx=K, context=problem)
    ls2 = Solver.any_from_conf(conf, mtx=M, context=problem)

def matvec_bj(vec):
    """
    The application of the Block Jacobi preconditioner.

    The exact version (as with the `direct` solver) can be obtained also
    by using the following PETScs command-line options, together with the
    `iterative-p` solver::

        -ksp_monitor -pc_type fieldsplit -pc_fieldsplit_type additive -fieldsplit_u
    ↪ksp_type preonly -fieldsplit_u_pc_type lu -fieldsplit_p_ksp_type preonly -fieldsplit_p_
    ↪pc_type lu

    The inexact version (20 iterations of a CG solver for each block, as
    with the `cg-s` or `cg-p` solvers) can be obtained also by using
    the following PETScs command-line options, together with the
    `iterative-p` solver::

        -ksp_monitor -pc_type fieldsplit -pc_fieldsplit_type additive -fieldsplit_u
    ↪ksp_type cg -fieldsplit_u_pc_type none -fieldsplit_p_ksp_type cg -fieldsplit_p_pc_type_
    ↪none -fieldsplit_u_ksp_max_it 20 -fieldsplit_p_ksp_max_it 20
    """
    vu = ls1(vec[iu])
    vp = ls2(vec[ip])

    return nm.r_[vu, vp]

```

(continues on next page)

(continued from previous page)

```

def matvec_j(vec):
    """
    The application of the Jacobi (diagonal) preconditioner.

    The same effect can be obtained also by using the following PETScs
    command-line options, together with the ``iterative-p`` solver::

        -ksp_monitor -pc_type jacobi
    """
    D = mtx.diagonal()

    return vec / D

    # Create the preconditioner, using one of matvec_bj() or matvec_j().
    precondition = Struct(name='precond', shape=mtx.shape, matvec=matvec_bj)
    precondition = spla.aslinearoperator(precondition)

    return precondition

method = 'gmres'
i_max = 20
eps_r = 1e-8

solvers = {
    'direct' : ('ls.scipy_direct', {}),

    'iterative-s' : ('ls.scipy_iterative', {
        'method' : method,
        'i_max' : i_max,
        'eps_r' : eps_r,
        'setup_precond': setup_precond,
        'verbose' : 2,
    }),
    'cg-s' : ('ls.scipy_iterative', {
        'method' : 'cg',
        'i_max' : 20,
        'eps_r' : 1e-6,
        'verbose' : 0,
    }),

    'iterative-p' : ('ls.petsc', {
        'method' : method,
        'precond' : 'none',
        'i_max' : i_max,
        'eps_r' : eps_r,
        'verbose' : 2,
        # 'ksp_converged_reason' : None,
        # 'ksp_monitor_true_residual' : None,
        # 'ksp_monitor_singular_value' : None,
        # 'ksp_final_residual' : None,
        # 'ksp_type' : 'gmres', # Overrides `method`.
        # 'ksp_max_it' : 500,
    })
}

```

(continues on next page)

(continued from previous page)

```

        # 'ksp_gmres_restart' : 1000,
        # 'pc_type' : 'none', # Overrides `precond`.
    }),
    'cg-p' : ('ls.petsc', {
        'method' : 'cg',
        'precond' : 'none',
        'i_max' : 20,
        'eps_r' : 1e-6,
        'verbose' : 0,
    }),

    'pyamg' : ('ls.pyamg', {
        'method' : 'smoothed_aggregation_solver',
        'i_max' : 20,
        'eps_r' : 1e-6,
        'verbose' : 0,
    }),

    'newton' : ('nls.newton',
        {'i_max' : 1,
         'eps_r' : 1e-6,
         'eps_a' : 1.0,
        }),
    },
}

options = {
    'nls' : 'newton',
    'ls' : 'iterative-s',

    'post_process_hook' : 'post_process',
}

```

multi_physics/piezo_elasticity.py

Description

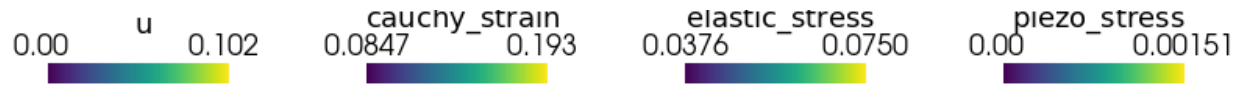
Piezo-elasticity problem - linear elastic material with piezoelectric effects.

Find $\underline{u}, \phi$ such that:

$$\begin{aligned}
 -\omega^2 \int_Y \rho \underline{v} \cdot \underline{u} + \int_Y D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{Y_2} g_{kij} e_{ij}(\underline{v}) \nabla_k \phi &= 0, \quad \forall \underline{v}, \\
 \int_{Y_2} g_{kij} e_{ij}(\underline{u}) \nabla_k \psi + \int_Y K_{ij} \nabla_i \psi \nabla_j \phi &= 0, \quad \forall \psi,
 \end{aligned}$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$



source code

```

r"""
Piezo-elasticity problem - linear elastic material with piezoelectric
effects.

Find  $\mathbf{u}$ ,  $\phi$  such that:

.. math::
    - \omega^2 \int_Y \rho \mathbf{v} \cdot \mathbf{u}
    + \int_Y D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u})
    - \int_{Y_2} g_{kij} e_{ij}(\mathbf{v}) \nabla_k \phi
    = 0
    \quad \forall \mathbf{v};

    \int_{Y_2} g_{kij} e_{ij}(\mathbf{u}) \nabla_k \phi
    + \int_Y K_{ij} \nabla_i \phi \nabla_j \phi
    = 0
    \quad \forall \phi;

where

.. math::

```

(continues on next page)

(continued from previous page)

```

    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
    """
    from __future__ import absolute_import
    import os
    import numpy as nm

    from sfepy import data_dir
    from sfepy.discrete.fem import MeshIO
    from sfepy.mechanics.matcoefs import stiffness_from_lame
    import six

    def post_process(out, pb, state, extend=False):
        """
        Calculate and output the strain and stresses for the given state.
        """
        from sfepy.base.base import Struct
        from sfepy.discrete.fem import extend_cell_data

        ev = pb.evaluate
        strain = ev('ev_cauchy_strain.i.Y(u)', mode='el_avg')
        stress = ev('ev_cauchy_stress.i.Y(inclusion.D, u)', mode='el_avg')

        piezo = -ev('ev_piezo_stress.i.Y2(inclusion.coupling, phi)',
                    mode='el_avg')
        piezo = extend_cell_data(piezo, pb.domain, 'Y2', val=0.0)

        piezo_strain = ev('ev_piezo_strain.i.Y(inclusion.coupling, u)',
                          mode='el_avg')

        out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                                     data=strain, dofs=None)
        out['elastic_stress'] = Struct(name='output_data', mode='cell',
                                      data=stress, dofs=None)
        out['piezo_stress'] = Struct(name='output_data', mode='cell',
                                    data=piezo, dofs=None)
        out['piezo_strain'] = Struct(name='output_data', mode='cell',
                                    data=piezo_strain, dofs=None)
        out['total_stress'] = Struct(name='output_data', mode='cell',
                                    data=stress + piezo, dofs=None)

        return out

    filename_mesh = data_dir + '/meshes/2d/special/circle_in_square.mesh'
    ## filename_mesh = data_dir + '/meshes/2d/special/circle_in_square_small.mesh'
    ## filename_mesh = data_dir + '/meshes/3d/special/cube_sphere.mesh'
    ## filename_mesh = data_dir + '/meshes/2d/special/cube_cylinder.mesh'

    omega = 1
    omega_squared = omega**2

```

(continues on next page)

(continued from previous page)

```

conf_dir = os.path.dirname(__file__)
io = MeshIO.any_from_filename(filename_mesh, prefix_dir=conf_dir)
bbox, dim = io.read_bounding_box(ret_dim=True)

geom = {3 : '3_4', 2 : '2_3'}[dim]

x_left, x_right = bbox[:,0]

options = {
    'post_process_hook' : 'post_process',
}

regions = {
    'Y' : 'all',
    'Y1' : 'cells of group 1',
    'Y2' : 'cells of group 2',
    'Y2_Surface': ('r.Y1 *v r.Y2', 'facet'),
    'Left' : ('vertices in (x < %f)' % (x_left + 1e-3), 'facet'),
    'Right' : ('vertices in (x > %f)' % (x_right - 1e-3), 'facet'),
}

fields = {
    'displacement' : ('real', dim, 'Y', 1),
    'potential' : ('real', 1, 'Y', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'phi' : ('unknown field', 'potential', 1),
    'psi' : ('test field', 'potential', 'phi'),
}

ebcs = {
    'u1' : ('Left', {'u.all' : 0.0}),
    'u2' : ('Right', {'u.0' : 0.1}),
    'phi' : ('Y2_Surface', {'phi.all' : 0.0}),
}

def get_inclusion_pars(ts, coor, mode=None, **kwargs):
    """TODO: implement proper 3D -> 2D transformation of constitutive
    matrices."""
    if mode == 'qp':
        _, dim = coor.shape
        sym = (dim + 1) * dim // 2

        dielectric = nm.eye(dim, dtype=nm.float64)
        # !!!
        coupling = nm.ones((dim, sym), dtype=nm.float64)
        # coupling[0,1] = 0.2

        out = {

```

(continues on next page)

(continued from previous page)

```

        # Lamé coefficients in 1e+10 Pa.
        'D' : stiffness_from_lame(dim=2, lam=0.1798, mu=0.148),
        # dielectric tensor
        'dielectric' : dielectric,
        # piezoelectric coupling
        'coupling' : coupling,
        'density' : nm.array([[0.1142]]), # in 1e4 kg/m3
    }

    for key, val in six.iteritems(out):
        out[key] = val[None, ...]

    return out

materials = {
    'inclusion' : (None, 'get_inclusion_pars')
}

functions = {
    'get_inclusion_pars' : (get_inclusion_pars,),
}

integrals = {
    'i' : 2,
}

equations = {
    '1' : """- %f * dw_dot.i.Y(inclusion.density, v, u)
              + dw_lin_elastic.i.Y(inclusion.D, v, u)
              - dw_piezo_coupling.i.Y2(inclusion.coupling, v, phi)
              = 0""" % omega_squared,
    '2' : """dw_piezo_coupling.i.Y2(inclusion.coupling, u, psi)
              + dw_diffusion.i.Y(inclusion.dielectric, psi, phi)
              = 0""",
}

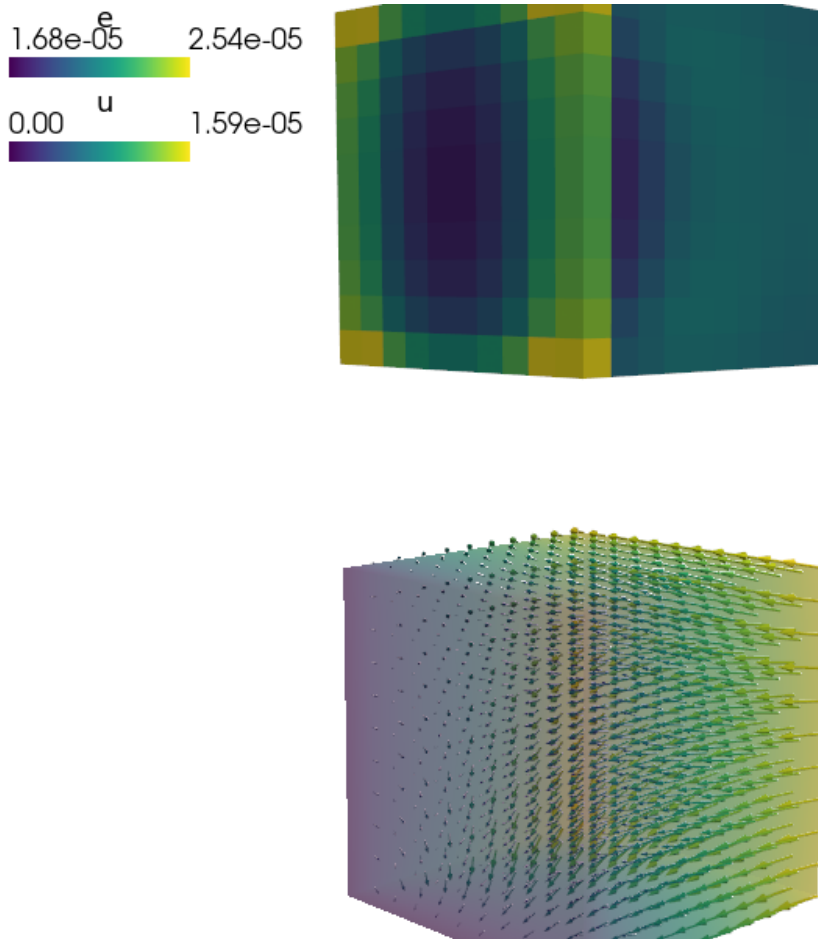
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
                {'i_max' : 1,
                 'eps_a' : 1e-10,
                }),
}

```

multi_physics/piezo_elasticity_macro.py**Description**

Piezo-elasticity problem - homogenization of a piezoelectric linear elastic matrix with embedded metallic electrodes, see [1] for details.

[1] E.Rohan, V.Lukes: Homogenization of the fluid-saturated piezoelectric porous media. International Journal of Solids and Structures 147, 2018, pages 110-125. <https://doi.org/10.1016/j.ijsolstr.2018.05.017>



source code

```
"""
Piezo-elasticity problem - homogenization of a piezoelectric linear elastic
matrix with embedded metallic electrodes, see [1] for details.

[1] E.Rohan, V.Lukes: Homogenization of the fluid-saturated piezoelectric
porous media. International Journal of Solids and Structures 147, 2018,
pages 110-125. https://doi.org/10.1016/j.ijsolstr.2018.05.017
"""

import numpy as nm
from sfepy import data_dir, base_dir
from sfepy.base.base import Struct
```

(continues on next page)

(continued from previous page)

```

from sfepy.homogenization.micmac import get_homog_coefs_linear
import os.path as osp
from sfepy.homogenization.recovery import recover_micro_hook

def post_process(out, pb, state, extend=False):
    # evaluate macroscopic strain
    strain = pb.evaluate('ev_cauchy_strain.i2.Omega(u)', mode='el_avg')
    out['e'] = Struct(name='output_data', mode='cell', dofs=None,
                      var_name='u', data=strain)

    # micro recovery
    rreg = pb.domain.regions['Recovery']
    dim = rreg.dim

    state_dict = state.get_state_parts()
    displ = state_dict['u']

    displ = displ.reshape((displ.shape[0] // dim, dim))
    macro = {
        'u': ('val', 'svar', displ),
        'strain': ('cauchy_strain', 'svar', displ),
        '_phi': pb.conf.phi,
    }

    def_args = {
        'eps0': pb.conf.eps0,
        'filename_mesh': pb.conf.filename_mesh_micro,
    }

    pvar = pb.create_variables(['svar'])

    recover_micro_hook(pb.conf.filename_micro, rreg, macro, pb.conf.eps0,
                      region_mode=pb.conf.region_mode,
                      eval_mode=pb.conf.eval_mode,
                      eval_vars=pvar, define_args=def_args)

    return out

def get_homog_fun(fname):
    return lambda ts, coors, mode=None, problem=None, **kwargs:\
        get_homog(coors, mode, problem, fname, **kwargs)

def get_homog(coors, mode, pb, micro_filename, **kwargs):
    if not (mode == 'qp'):
        return

    nqp = coors.shape[0]
    coefs_filename = osp.join(pb.conf.options.get('output_dir', '.'),
                                   'coefs_piezo.h5')

```

(continues on next page)

(continued from previous page)

```

def_args = {
    'eps0': pb.conf.eps0,
    'filename_mesh': pb.conf.filename_mesh_micro,
}

coefs = get_homog_coefs_linear(0, 0, None,
                              micro_filename=micro_filename,
                              coefs_filename=coefs_filename,
                              define_args=def_args)

Vf = coefs['V0'] * pb.conf.phi[0] + coefs['V1'] * pb.conf.phi[1]

out = {
    'A': nm.tile(coefs['A'], (nqp, 1, 1)),
    'Vf': nm.tile(Vf[:, nm.newaxis], (nqp, 1, 1)),
}

return out

def define(region_mode='el_centers', eval_mode='constant'):
    """
    Parameters
    -----
    region_mode : {'el_centers', 'tiled'}
    eval_mode : {'constant', 'continuous'}
    """
    eps0 = 1. / 30 # real size of the reference cell

    phi = nm.array([1, -1]) * 1e4 # prescribed el. potential

    filename_mesh = data_dir + '/meshes/3d/cube_medium_hexa.mesh'

    # define the micro problem - homogenization procedure
    filename_micro = base_dir + \
        '/examples/multi_physics/piezo_elasticity_micro.py'
    filename_mesh_micro = data_dir + '/meshes/3d/piezo_mesh_micro.vtk'

    fields = {
        'displacement': ('real', 'vector', 'Omega', 1),
        'sfield': ('real', 'scalar', 'Omega', 1),
    }

    variables = {
        'u': ('unknown field', 'displacement'),
        'v': ('test field', 'displacement', 'u'),
        'svar': ('parameter field', 'sfield', 'set-to-none'),
    }

    # define material - homogenization
    functions = {

```

(continues on next page)

(continued from previous page)

```

    'get_homog': (get_homog_fun(filename_micro),),
}

materials = {
    'hom': 'get_homog',
}

integrals = {
    'i2': 2,
}

regions = {
    'Omega': 'all',
    'Left': ('vertices in (x < -0.4999)', 'facet'),
    'Recovery': ('cell 266'),
}

ebcs = {
    'fixed_u': ('Left', {'u.all': 0.0}),
}

equations = {
    'balance_of_forces': """
        dw_lin_elastic.i2.Omega(hom.A, v, u)
        =
        - dw_lin_prestress.i2.Omega(hom.Vf, v)""",
}

solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton',
               {'i_max': 10,
                'eps_a': 1e-3,
                'eps_r': 1e-3,
                'problem': 'nonlinear',
                })
}

options = {
    'output_dir': 'output',
    'nls': 'newton',
    'post_process_hook': 'post_process',
}

return locals()

```

multi_physics/piezo_elasticity_micro.py

Description

Piezo-elasticity problem - homogenization of a piezoelectric linear elastic matrix with embedded metallic electrodes, see [1] for details.

[1] E.Rohan, V.Lukes: Homogenization of the fluid-saturated piezoelectric porous media. International Journal of Solids and Structures 147, 2018, pages 110-125. <https://doi.org/10.1016/j.ijsolstr.2018.05.017>

source code

```
r"""
Piezo-elasticity problem - homogenization of a piezoelectric linear elastic
matrix with embedded metallic electrodes, see [1] for details.

[1] E.Rohan, V.Lukes: Homogenization of the fluid-saturated piezoelectric
porous media. International Journal of Solids and Structures 147, 2018,
pages 110-125. https://doi.org/10.1016/j.ijsolstr.2018.05.017
"""

import os
import numpy as nm

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.homogenization.utils import coor_to_sym, define_box_regions
from sfepy.discrete.fem.mesh import Mesh
from sfepy.base.base import Struct
import sfepy.discrete.fem.periodic as per
import sfepy.homogenization.coefs_base as cb

# Recover fields at the microscopical level
def recovery_micro(pb, corrs, macro):
    eps0 = macro['eps0']
    mesh = pb.domain.mesh
    regions = pb.domain.regions
    dim = mesh.dim

    if len(macro['u'].shape) == 2:
        mac_strain_Ymc = mac_strain_Ym = macro['strain'][None, ...]
        mac_u_Ymc = macro['u'][None, ...]
    else:
        Ymc_map = regions['Ymc'].get_entities(0)
        Ym_map = regions['Ym'].get_entities(0)
        mac_strain_Ymc = macro['strain'][Ymc_map, ...]
        mac_strain_Ym = macro['strain'][Ym_map, ...]
        mac_u_Ymc = macro['u'][Ymc_map, ...]

    # deformation
    u1, phi = 0, 0

    for ii in range(2):
        u1 += corrs['corrs_k%d' % ii]['u'] * macro['_phi'][ii]
        phi += corrs['corrs_k%d' % ii]['r'] * macro['_phi'][ii]
```

(continues on next page)

(continued from previous page)

```

for ii in range(dim):
    for jj in range(dim):
        kk = coord_to_sym(ii, jj, dim)
        sidx = f'_{ii}_{jj}'
        phi += corrs['corrs_rs']['r' + sidx] * mac_strain_Ym[:, kk]
        u1 += corrs['corrs_rs']['u' + sidx] * mac_strain_Ymc[:, kk]

u = mac_u_Ymc[:, 0] + eps0 * u1
mvar = pb.create_variables(['u', 'r', 'svar'])
e_mac_Ymc = [None] * macro['strain'].shape[1]

if mac_strain_Ymc.shape[0] > 1:
    for ii in range(dim):
        for jj in range(dim):
            kk = coord_to_sym(ii, jj, dim)
            mvar['svar'].set_data(mac_strain_Ymc[:, kk])
            aux = pb.evaluate('ev_integrate.i2.Ymc(svar)',
                              mode='el_avg',
                              var_dict={'svar': mvar['svar']})
            e_mac_Ymc[kk] = aux.squeeze()

    e_mac_Ymc = nm.vstack(e_mac_Ymc).T[:, nm.newaxis, :, nm.newaxis]
else:
    e_mac_Ymc = mac_strain_Ymc

mvar['r'].set_data(phi)
E_mic = pb.evaluate('ev_grad.i2.Ym(r)',
                    mode='el_avg',
                    var_dict={'r': mvar['r']}) / eps0

mvar['u'].set_data(u1)
e_mic = pb.evaluate('ev_cauchy_strain.i2.Ymc(u)',
                    mode='el_avg',
                    var_dict={'u': mvar['u']})
e_mic += e_mac_Ymc

out = {
    'u': (u, 'u', 'p'),
    'u1': (u1, 'u', 'p'),
    'e_mic': (e_mic, 'u', 'c'),
    'phi': (phi, 'r', 'p'),
    'E_mic': (E_mic, 'r', 'c'),
}

out_struct = {}
for k, v in out.items():
    out_struct[k] = Struct(name='output_data',
                           mode='cell' if v[2] == 'c' else 'vertex',
                           data=v[0],
                           var_name=v[1],
                           dofs=None)

```

(continues on next page)

(continued from previous page)

```

    return out_struct

# Define the local problems and the homogenized coefficients,
# eps0 is the real size of the reference cell.
def define(eps0=1e-3, filename_mesh='meshes/3d/piezo_mesh_micro.vtk'):
    filename_mesh = os.path.join(data_dir, filename_mesh)
    mesh = Mesh.from_file(filename_mesh)
    bbox = mesh.get_bounding_box()
    regions = define_box_regions(mesh.dim, bbox[0], bbox[1], eps=1e-3)

    regions.update({
        'Ymc': 'all',
        # matrix
        'Ym': 'cells of group 1',
        'Ym_left': ('r.Ym *v r.Left', 'vertex'),
        'Ym_right': ('r.Ym *v r.Right', 'vertex'),
        'Ym_bottom': ('r.Ym *v r.Bottom', 'vertex'),
        'Ym_top': ('r.Ym *v r.Top', 'vertex'),
        'Ym_far': ('r.Ym *v r.Far', 'vertex'),
        'Ym_near': ('r.Ym *v r.Near', 'vertex'),
        'Gamma_ms': ('r.Ym *v r.Yc', 'facet', 'Ym'),
        # conductors
        'Yc': ('r.Yc1 +c r.Yc2', 'cell'),
        'Yc1': 'cells of group 2',
        'Yc2': 'cells of group 3',
        'Gamma_s1': ('r.Ym *v r.Yc1', 'facet', 'Ym'),
        'Gamma_s2': ('r.Ym *v r.Yc2', 'facet', 'Ym'),
    })

    options = {
        'coefs_filename': 'coefs_piezo',
        'volume': {'value': nm.prod(bbox[1] - bbox[0])},
        'coefs': 'coefs',
        'requirements': 'requirements',
        'output_dir': 'output',
        'split_results_by': 'region',
        'absolute_mesh_path': True,
        'multiprocessing': False,
        'recovery_hook': recovery_micro,
    }

    fields = {
        'displacement': ('real', 'vector', 'Ymc', 1),
        'potential': ('real', 'scalar', 'Ym', 1),
        'sfield': ('real', 'scalar', 'Ymc', 1),
    }

    variables = {
        # displacement
        'u': ('unknown field', 'displacement'),

```

(continues on next page)

(continued from previous page)

```

    'v': ('test field', 'displacement', 'u'),
    'Pi_u': ('parameter field', 'displacement', 'u'),
    'U1': ('parameter field', 'displacement', '(set-to-None)'),
    'U2': ('parameter field', 'displacement', '(set-to-None)'),
    # potential
    'r': ('unknown field', 'potential'),
    's': ('test field', 'potential', 'r'),
    'Pi_r': ('parameter field', 'potential', 'r'),
    'R1': ('parameter field', 'potential', '(set-to-None)'),
    'R2': ('parameter field', 'potential', '(set-to-None)'),
    # auxiliary
    'svar': ('parameter field', 'sfield', '(set-to-None)'),
}

epbcs = {
    'p_ux': (['Left', 'Right'], {'u.all': 'u.all'}, 'match_x_plane'),
    'p_uy': (['Near', 'Far'], {'u.all': 'u.all'}, 'match_y_plane'),
    'p_uz': (['Bottom', 'Top'], {'u.all': 'u.all'}, 'match_z_plane'),
    'p_rx': (['Ym_left', 'Ym_right'], {'r.0': 'r.0'}, 'match_x_plane'),
    'p_ry': (['Ym_near', 'Ym_far'], {'r.0': 'r.0'}, 'match_y_plane'),
    'p_rz': (['Ym_bottom', 'Ym_top'], {'r.0': 'r.0'}, 'match_z_plane'),
}

periodic = {
    'per_u': ['per_u_x', 'per_u_y', 'per_u_z'],
    'per_r': ['per_r_x', 'per_r_y', 'per_r_z'],
}

# rescale piezoelectric material parameters
mat_g_sc, mat_d_sc = (eps0, eps0**2)

materials = {
    'elastic': ({
        'D': {
            'Ym': nm.array([[1.504, 0.656, 0.659, 0, 0, 0],
                           [0.656, 1.504, 0.659, 0, 0, 0],
                           [0.659, 0.659, 1.455, 0, 0, 0],
                           [0, 0, 0, 0.424, 0, 0],
                           [0, 0, 0, 0, 0.439, 0],
                           [0, 0, 0, 0, 0, 0.439]]) * 1e11,
            'Yc': stiffness_from_youngpoisson(3, 200e9, 0.25)}},),
    'piezo': ({
        'g': nm.array([[0, 0, 0, 0, 11.404, 0],
                       [0, 0, 0, 0, 0, 11.404],
                       [-4.322, -4.322, 17.360, 0, 0, 0]]) / mat_g_sc,
        'd': nm.array([[1.284, 0, 0],
                       [0, 1.284, 0],
                       [0, 0, 1.505]]) * 1e-8 / mat_d_sc},),
}

functions = {
    'match_x_plane': (per.match_x_plane,),

```

(continues on next page)

(continued from previous page)

```

    'match_y_plane': (per.match_y_plane,),
    'match_z_plane': (per.match_z_plane,),
}

ebcs = {
    'fixed_u': ('Corners', {'u.all': 0.0}),
    'fixed_r': ('Gamma_ms', {'r.all': 0.0}),
    'fixed_r1_s1': ('Gamma_s1', {'r.0': 1.0}),
    'fixed_r0_s1': ('Gamma_s1', {'r.0': 0.0}),
    'fixed_r1_s2': ('Gamma_s2', {'r.0': 1.0}),
    'fixed_r0_s2': ('Gamma_s2', {'r.0': 0.0}),
}

integrals = {
    'i2': 2,
}

solvers = {
    'ls_d': ('ls.auto_direct', {'use_presolve' : True}),
    'ls_i': ('ls.scipy_iterative', {}),
    'ns_ea6': ('nls.newton', {'eps_a': 1e6, 'eps_r': 1e-3,}),
    'ns_ea0': ('nls.newton', {'eps_a': 1e0, 'eps_r': 1e-3,}),
}

coefs = {
    'A1': {
        'status': 'auxiliary',
        'requires': ['pis_u', 'corrs_rs'],
        'expression': 'dw_lin_elastic.i2.Ymc(elastic.D, U1, U2)',
        'set_variables': [(('U1', ('corrs_rs', 'pis_u'), 'u'),
                           ('U2', ('corrs_rs', 'pis_u'), 'u'))],
        'class': cb.CoeffSymSym,
    },
    'A2': {
        'status': 'auxiliary',
        'requires': ['corrs_rs'],
        'expression': 'dw_diffusion.i2.Ym(piezo.d, R1, R2)',
        'set_variables': [(('R1', 'corrs_rs', 'r'),
                           ('R2', 'corrs_rs', 'r'))],
        'class': cb.CoeffSymSym,
    },
    'A': {
        'requires': ['c.A1', 'c.A2'],
        'expression': 'c.A1 + c.A2',
        'class': cb.CoeffEval,
    },
    'vol': {
        'regions': ['Ym', 'Yc1', 'Yc2'],
        'expression': 'ev_volume.i2.%s(svar)',
        'class': cb.VolumeFractions,
    },
    'eps0': {

```

(continues on next page)

(continued from previous page)

```

        'requires': [],
        'expression': '%e' % eps0,
        'class': cb.CoeffEval,
    },
    'filenames': {},
}

requirements = {
    'pis_u': {
        'variables': ['u'],
        'class': cb.ShapeDimDim,
    },
    'pis_r': {
        'variables': ['r'],
        'class': cb.ShapeDim,
    },
    'corrs_rs': {
        'requires': ['pis_u'],
        'ebcs': ['fixed_u', 'fixed_r'],
        'epbcs': ['p_ux', 'p_uy', 'p_uz', 'p_rx', 'p_ry', 'p_rz'],
        'equations': {
            'eq1':
                """dw_lin_elastic.i2.Ymc(elastic.D, v, u)
                - dw_piezo_coupling.i2.Ym(piezo.g, v, r)
                = - dw_lin_elastic.i2.Ymc(elastic.D, v, Pi_u)""",
            'eq2':
                """
                - dw_piezo_coupling.i2.Ym(piezo.g, u, s)
                - dw_diffusion.i2.Ym(piezo.d, s, r)
                = dw_piezo_coupling.i2.Ym(piezo.g, Pi_u, s)""",
        },
        'set_variables': [('Pi_u', 'pis_u', 'u')],
        'class': cb.CorrDimDim,
        'save_name': 'corrs_rs',
        'solvers': {'ls': 'ls_d', 'nls': 'ns_ea6'},
        'is_linear': True,
    },
}

# define requirements and coefficients related to conductors
bc_conductors = [
    ['fixed_r1_s1', 'fixed_r0_s2'], # phi = 1 on S1, phi = 0 on S2
    ['fixed_r1_s2', 'fixed_r0_s1'], # phi = 0 on S1, phi = 1 on S2
]

for k in range(2):
    sk = '%d' % k

    requirements.update({
        'corrs_k' + sk: {
            'requires': ['pis_r'],
            'ebcs': ['fixed_u'] + bc_conductors[k],

```

(continues on next page)

(continued from previous page)

```

        'epbcs': ['p_ux', 'p_uy', 'p_uz', 'p_rx', 'p_ry', 'p_rz'],
        'equations': {
            'eq1':
                """dw_lin_elastic.i2.Ymc(elastic.D, v, u)
                - dw_piezo_coupling.i2.Ym(piezo.g, v, r)
                = 0""",
            'eq2':
                """
                - dw_piezo_coupling.i2.Ym(piezo.g, u, s)
                - dw_diffusion.i2.Ym(piezo.d, s, r)
                = 0"""
        },
        'class': cb.CorrOne,
        'save_name': 'corrs_k' + sk,
        'solvers': {'ls': 'ls_d', 'nls': 'ns_ea0'},
        'is_linear': True,
    },
})

coefs.update({
    'V1_' + sk: {
        'status': 'auxiliary',
        'requires': ['pis_u', 'corrs_k' + sk],
        'expression': 'dw_lin_elastic.i2.Ymc(elastic.D, U1, U2)',
        'set_variables': [('U1', 'corrs_k' + sk, 'u'),
                          ('U2', 'pis_u', 'u')],
        'class': cb.CoeffSym,
    },
    'V2_' + sk: {
        'status': 'auxiliary',
        'requires': ['pis_u', 'corrs_k' + sk],
        'expression': 'dw_piezo_coupling.i2.Ym(piezo.g, U1, R1)',
        'set_variables': [('R1', 'corrs_k' + sk, 'r'),
                          ('U1', 'pis_u', 'u')],
        'class': cb.CoeffSym,
    },
    'V' + sk: {
        'requires': ['c.V1_' + sk, 'c.V2_' + sk],
        'expression': 'c.V1_%s - c.V2_%s' % (sk, sk),
        'class': cb.CoeffEval,
    },
})

return locals()

```

multi_physics/piezo_elastodynamic.py

Description

The linear elastodynamics of a piezoelectric body loaded by a given base motion.

The generated voltage between the bottom and top surface electrodes is recorded and plotted. The scalar potential on the top surface electrode is modeled using a constant L^2 field. The Nitsche's method is used to weakly apply the (unknown) potential Dirichlet boundary condition on the top surface.

Find the displacements $\underline{u}(t)$, the potential $p(t)$ and the constant potential on the top electrode $\bar{p}(t)$ such that:

$$\begin{aligned} \int_{\Omega} \rho \underline{v} \cdot \ddot{\underline{u}} + \int_{\Omega} C_{ijkl} \varepsilon_{ij}(\underline{v}) \varepsilon_{kl}(\underline{u}) - \int_{\Omega} e_{kij} \varepsilon_{ij}(\underline{v}) \nabla_k p &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} e_{kij} \varepsilon_{ij}(\underline{u}) \nabla_k q + \int_{\Omega} \kappa_{ij} \nabla_i \psi \nabla_j p - \int_{\Gamma_{top}} \kappa_{ij} \nabla_j p n_i q + \int_{\Gamma_{top}} \kappa_{ij} \nabla_j q n_i (p - \bar{p}) + \int_{\Gamma_{top}} k q (p - \bar{p}) &= 0, \quad \forall q, \\ \int_{\Gamma_{top}} \kappa_{ij} \nabla_j \dot{p} n_i + \bar{p}/R &= 0, \end{aligned}$$

where C_{ijkl} is the matrix of elastic properties under constant electric field intensity, e_{kij} the piezoelectric modulus, κ_{ij} the permittivity under constant deformation, k a penalty parameter and R the external circuit resistance (e.g. of an oscilloscope used to measure the voltage between the electrodes).

Usage Examples

Run with the default settings, results stored in `output/piezo-ed/`:

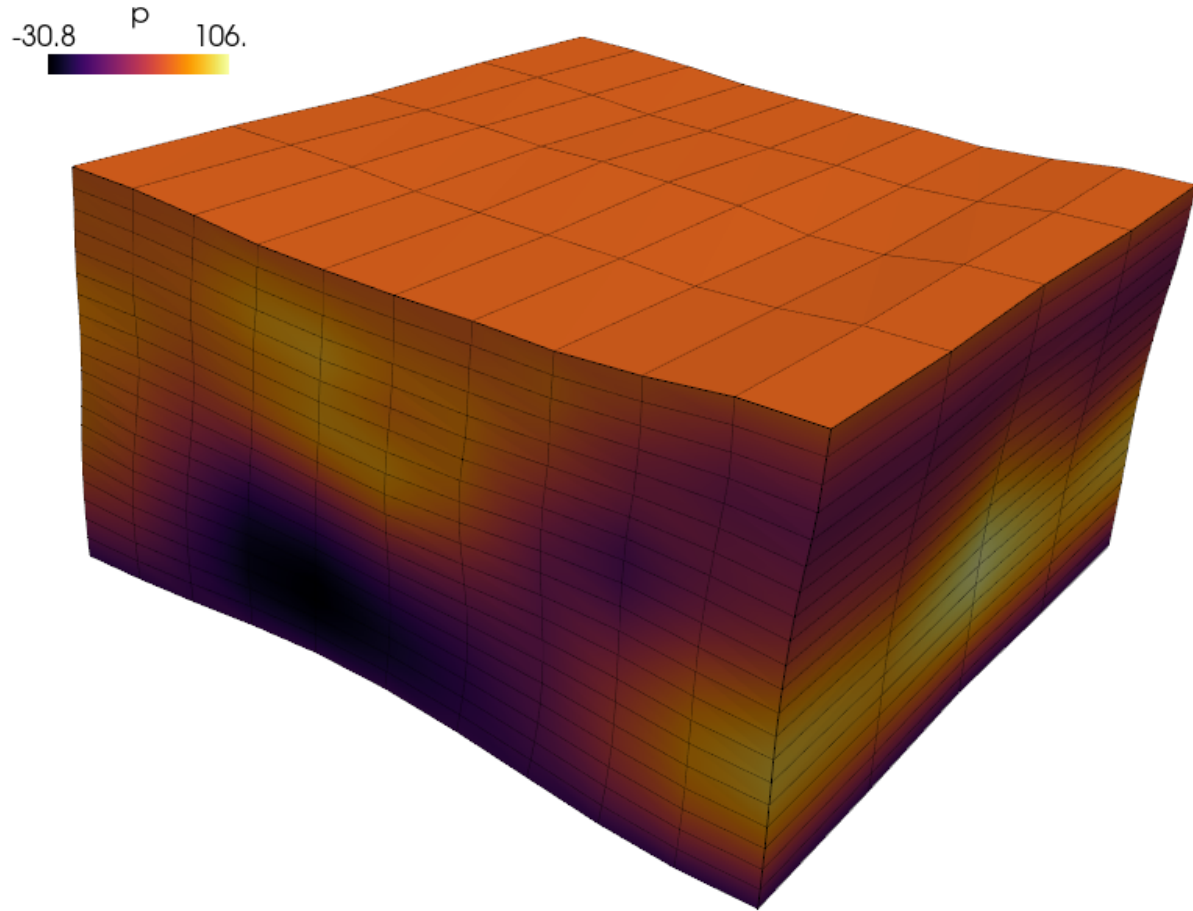
```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py
```

The `define()` arguments, see below, can be set using the `-d` option:

```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py -d "order=2, ct1=2.5"
```

View the resulting potential p on a deformed mesh (2000x magnified):

```
sfepy-view output/piezo-ed/user_block.h5 -f p:wu:f2000:p0 1:vw:wu:f2000:p0 --color-  
map=inferno
```



source code

```
r"""
The linear elastodynamics of a piezoelectric body loaded by a given base
motion.

The generated voltage between the bottom and top surface electrodes is recorded
and plotted. The scalar potential on the top surface electrode is modeled using
a constant  $L^2$  field. The Nitsche's method is used to weakly apply the
(unknown) potential Dirichlet boundary condition on the top surface.

Find the displacements  $\mathbf{u}(t)$ , the potential  $p(t)$  and the
constant potential on the top electrode  $\bar{p}(t)$  such that:

.. math::
\int_{\Omega} \rho \mathbf{v} \cdot \ddot{\mathbf{u}}
+ \int_{\Omega} C_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
- \int_{\Omega} e_{kij} \epsilon_{ij}(\mathbf{v}) \nabla_k p
= 0
\;, \quad \forall \mathbf{v} \;,

\int_{\Omega} e_{kij} \epsilon_{ij}(\mathbf{u}) \nabla_k q
+ \int_{\Omega} \kappa_{ij} \nabla_i \psi \nabla_j p
```

(continues on next page)

(continued from previous page)

```

- \int_{\Gamma_{top}} \kappa_{ij} \nabla_j p n_i q
+ \int_{\Gamma_{top}} \kappa_{ij} \nabla_j q n_i (p - \bar{p})
+ \int_{\Gamma_{top}} k q (p - \bar{p})
= 0
\;; \quad \forall q \;;

\int_{\Gamma_{top}} \kappa_{ij} \nabla_j \dot{p} n_i + \bar{p} / R = 0 \;;,

```

where C_{ijkl} is the matrix of elastic properties under constant electric field intensity, e_{kij} the piezoelectric modulus, κ_{ij} the permittivity under constant deformation, k a penalty parameter and R the external circuit resistance (e.g. of an oscilloscope used to measure the voltage between the electrodes).

Usage Examples

Run with the default settings, results stored in `output/piezo-ed/`:

```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py
```

The `func:define()` arguments, see below, can be set using the `-d` option:

```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py -d "order=2, ct1=2.5"
```

View the resulting potential p on a deformed mesh (2000x magnified):

```
sfepy-view output/piezo-ed/user_block.h5 -f p:wu:f2000:p0 1:vw:wu:f2000:p0 --color-
map=inferno
"""
```

```
from functools import partial
```

```
import numpy as nm
```

```
from sfepy.base.base import output
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.homogenization.utils import define_box_regions
```

```
def post_process(out, problem, state, extend=False, pcs=None):
```

```
    """
```

```
    Calculate and output strain, stress and electric field vector for the given
    displacements and potential.
    """
```

```
    from sfepy.base.base import Struct
```

```
    ev = problem.evaluate
```

```
    strain = ev('ev_cauchy_strain.i.Omega(u)', mode='el_avg', verbose=False)
```

```
    stress = ev('ev_cauchy_stress.i.Omega(m.C, u)', mode='el_avg',
                copy_materials=False, verbose=False)
```

```
    E = ev('ev_grad.i.Omega(p)', mode='el_avg', verbose=False)
```

(continues on next page)

(continued from previous page)

```

out['cauchy_strain'] = Struct(name='output_data', mode='cell',
                             data=strain)
out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                              data=stress)
out['E'] = Struct(name='output_data', mode='cell', data=E)

top = problem.domain.regions['Top']
p_top = state['p'].get_state_in_region(top)
# = state['pc'](), but we want to test .get_state_in_region()
pc_top = state['pc'].get_state_in_region(top)

output('pc:', pc_top)
output('|p - pc|_top:', nm.linalg.norm(p_top - pc_top))
if pcs is not None:
    pcs.append(pc_top[0, 0])

return out

def plot_voltage(problem, state, pcs=None):
    import os.path as op
    import matplotlib.pyplot as plt

    ts = problem.get_timestepper()

    fig, ax = plt.subplots()
    ax.plot(ts.times, pcs)
    ax.set_xlabel('$t$ [s]')
    ax.set_ylabel(r'$\bar{p}$ [V]')

    fig.tight_layout()
    fig.savefig(op.join(problem.output_dir, 'voltage.pdf'))

def define(
    dims=(1e-2, 1e-2, 5e-3),
    shape=(5, 11, 21),
    order=1,
    amplitude=0.0000001,
    ct1=1.5,
    dt=None,
    tss_name='tsn',
    tsc_name='tsced1',
    adaptive=False,
    ls_name='lsd',
    active_only=False,
    save_times='all',
    output_dir='output/piezo-ed',
):
    """
    Parameters
    -----
    dims: physical dimensions of the block mesh
    shape: numbers of mesh vertices along each axis

```

(continues on next page)

(continued from previous page)

```

order: the FE approximation order
amplitude: the seismic load amplitude
ct1: final time in min(dims) / "longitudinal wave speed" units
dt: time step (None means automatic)
tss_name: time stepping solver name (see "solvers" section)
tsc_name: time step controller name (see "solvers" section)
adaptive: use adaptive time step control
ls_name: linear system solver name (see "solvers" section)
save_times: number of solutions to save
output_dir: output directory
"""

dim = len(dims)
assert dim == 3

# A PZT 5-H material, Voigt notation, strain - electric displacement form.
epsT = nm.array([[1700., 0, 0],
                 [0, 1700., 0],
                 [0, 0, 1450.0]])
dv = 1e-12 * nm.array([[0, 0, 0, 0, 741., 0],
                       [0, 0, 0, 741, 0, 0],
                       [-274., -274., 593., 0, 0, 0]]) # C / N = m / V

# Convert to stress - electric displacement form.
CEv = nm.array([[1.27e+011, 8.02e+010, 8.47e+010, 0, 0, 0],
                [8.02e+010, 1.27e+011, 8.47e+010, 0, 0, 0],
                [8.47e+010, 8.47e+010, 1.17e+011, 0, 0, 0],
                [0, 0, 0, 2.34e+010, 0, 0],
                [0, 0, 0, 0, 2.30e+010, 0],
                [0, 0, 0, 0, 0, 2.35e+010]])

ev = dv @ CEv
epsS = epsT - dv @ ev.T

# SfePy: 11 22 33 12 13 23
# Voigt: 11 22 33 23 13 12
ii = [0, 1, 2, 5, 4, 3]
ix, iy = nm.meshgrid(ii, ii, sparse=True)
CE = CEv[ix, iy]
e = ev[:, ii]

eps0 = 8.8541878128e-12
kappa = epsS * eps0

# Longitudinal and shear wave propagation speeds.
mu = CE[-1, -1]
lam = CE[0, 0] - 2 * mu
rho = 7800
cl = nm.sqrt((lam + 2.0 * mu) / rho)
cs = nm.sqrt(mu / rho)

# Element size.
L = nm.min(dims)
H = L / (nm.max(shape) - 1)

```

(continues on next page)

(continued from previous page)

```

# Time-stepping parameters.
if dt is None:
    # For implicit schemes, dt based on the Courant number  $C0 = dt * c1 / H$ 
    # equal to 1.
    dt = H / c1 #  $C0 = 1$ 

t1 = ct1 * L / c1

# Time history record of pc.
pcs = []
_post_process = partial(post_process, pcs=pcs)
_plot_voltage = partial(plot_voltage, pcs=pcs)

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, 0.5 * nm.array(dims),
                              name='user_block', verbose=False)
        return mesh

    elif mode == 'write':
        pass

filename_mesh = UserMeshIO(mesh_hook)

bbox = [[0] * dim, dims]
regions = define_box_regions(dim, bbox[0], bbox[1], 1e-5)
regions.update({
    'Omega' : 'all',
})

materials = {
    'inclusion' : (None, 'get_inclusion_pars')
}

fields = {
    'displacement' : ('real', 'vector', 'Omega', order),
    'potential' : ('real', 'scalar', 'Omega', order),
    'constant' : ('real', 'scalar', 'Top', 0, 'L2', 'constant'),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'p' : ('unknown field', 'potential', 1, 1),
    'q' : ('test field', 'potential', 'p'),
    'pc' : ('unknown field', 'constant', 2, 1),
    'qc' : ('test field', 'constant', 'pc'),
}

```

(continues on next page)

(continued from previous page)

```

materials = {
    'm' : ({
        'C': CE,
        'e' : e,
        'kappa' : kappa,
        'rho': rho,
        'penalty': 1,
        'iR' : 1.0 / (15e6 * dims[0] * dims[1]), # 1 / (R * top_area).
    },),
}

integrals = {
    'i' : 2 * order,
}

def get_ebcs(ts, coors, mode='u'):
    y = coors[:, 1]
    k = 2 * nm.pi / dims[1]
    shift = nm.pi / 3
    omega = cl * k
    time = ts.time
    if mode == 'u':
        val = (amplitude * nm.sin(time * omega) * nm.sin(k * y + shift))

    elif mode == 'du':
        val = (amplitude * omega * nm.cos(time * omega)
               * nm.sin(k * y + shift))

    elif mode == 'ddu':
        val = (-amplitude * omega**2 * nm.sin(time * omega)
               * nm.sin(k * y + shift))

    return val

functions = {
    'get_u' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor),),
    'get_du' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='du'),),
    'get_ddu' : (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='ddu'),),
}

ebcs = {
    'Seismic' : ('Bottom', {'u.2' : 'get_u', 'du.2' : 'get_du',
                             'ddu.2' : 'get_ddu'}),
    'Pot0' : ('Bottom', {'p.all' : 0.0}),
}

ics = {
    'ic' : ('Omega', {'u.all' : 0.0, 'du.all' : 0.0, 'p.0' : 0.0}),
}

equations = {

```

(continues on next page)

(continued from previous page)

```

'1' : """dw_dot.i.Omega(m.rho, v, ddu)
      + dw_lin_elastic.i.Omega(m.C, v, u)
      - dw_piezo_coupling.i.Omega(m.e, v, p)
      = 0""",
'2' : """dw_piezo_coupling.i.Omega(m.e, u, q)
      + dw_diffusion.i.Omega(m.kappa, q, p)
      - de_surface_flux.i.Top(m.kappa, q, p)
      + de_surface_flux.i.Top(m.kappa, p, q)
      - de_surface_flux.i.Top(m.kappa, pc, q)
      + dw_dot.i.Top(m.penalty, q, p)
      - dw_dot.i.Top(m.penalty, q, pc)
      = 0""",
'3' : """de_surface_flux.i.Top(m.kappa, qc, dp/dt)
      + 0.5 * dw_dot.i.Top(m.iR, qc, pc)
      + 0.5 * dw_dot.i.Top(m.iR, qc, pc[-1])
      = 0""",
}

solvers = {
    'lsd' : ('ls.auto_direct', {
        # Reuse the factorized linear system from the first time step.
        'use_presolve' : True,
        # Speed up the above by omitting the matrix digest check used
        # normally for verification that the current matrix corresponds to
        # the factorized matrix stored in the solver instance. Use with
        # care!
        'use_mtx_digest' : False,
        # Increase when getting MUMPS error -9.
        'memory_relaxation' : 20,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
        'eps_r' : 1e-6,
        'ls_on' : 1e100,
    }),
    'tsn' : ('ts.newmark', {
        't0' : 0.0,
        't1' : t1,
        'dt' : dt,
        'n_step' : None,

        'is_linear' : True,
        # Without this the adaptive time-stepping cannot work.
        'has_time_derivatives' : adaptive,

        'beta' : 0.25,
        'gamma' : 0.5,

        'verbose' : 1,
    }),
    'tscedl' : ('tsc.ed_linear', {

```

(continues on next page)

(continued from previous page)

```

        'eps_r' : (1e-4, 1e-2),
        'eps_a' : (1e-8, 1e-3),
        'fmin' : 0.3,
        'fmax' : 2.5,
        'fsafety' : 0.85,
        'red_factor' : 0.9,
        'inc_wait' : 10,
        'min_inc_factor' : 1.5,
    }},
}

options = {
    'ts' : tss_name,
    'tsc' : tsc_name if adaptive else None,
    'nls' : 'newton',
    'ls' : ls_name,

    'save_times' : save_times,

    'active_only' : active_only,
    'auto_transform_equations' : True,

    'output_format' : 'h5',
    'output_dir' : output_dir,
    'post_process_hook' : _post_process,
    'post_process_hook_final' : _plot_voltage,
}

return locals()

```

multi_physics/thermal_electric.py

Description

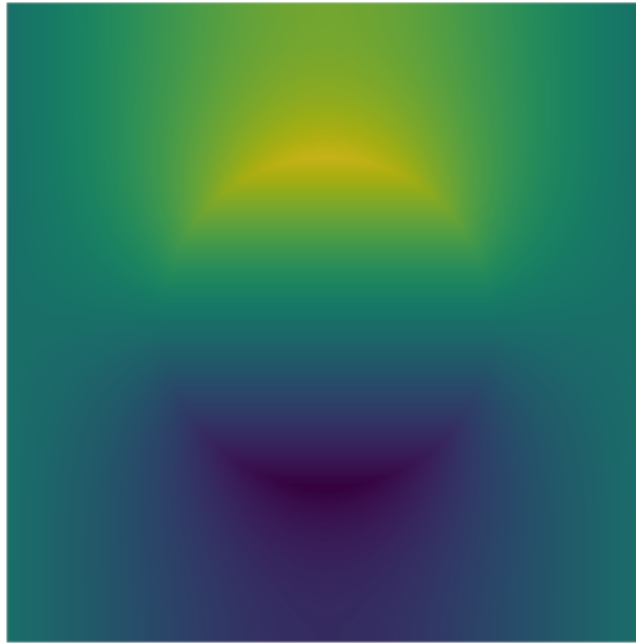
First solve the stationary electric conduction problem. Then use its results to solve the evolutionary heat conduction problem.

Run this example as on a command line:

```
$ python <path_to_this_file>/thermal_electric.py
```



phi
-1.00 1.00

source code

```
#!/usr/bin/env python
"""
First solve the stationary electric conduction problem. Then use its
results to solve the evolutionary heat conduction problem.

Run this example as on a command line::

    $ python <path_to_this_file>/thermal_electric.py
"""
from __future__ import absolute_import
import sys
sys.path.append( '.' )
import os

from sfepy import data_dir

filename_mesh = data_dir + '/meshes/2d/special/circle_in_square.mesh'

# Time stepping for the heat conduction problem.
t0 = 0.0
t1 = 0.5
```

(continues on next page)

(continued from previous page)

```

n_step = 11

# Material parameters.
specific_heat = 1.2

#####

cwd = os.path.split(os.path.join(os.getcwd(), __file__))[0]

options = {
    'absolute_mesh_path' : True,
    'output_dir' : os.path.join(cwd, 'output')
}

regions = {
    'Omega' : 'all',
    'Omega1' : 'cells of group 1',
    'Omega2' : 'cells of group 2',
    'Omega2_Surface': ('r.Omega1 *v r.Omega2', 'facet'),
    'Left' : ('vertices in (x < %f)' % -0.4999, 'facet'),
    'Right' : ('vertices in (x > %f)' % 0.4999, 'facet'),
}

materials = {
    'm' : ({
        'thermal_conductivity' : 2.0,
        'electric_conductivity' : 1.5,
    },),
}

# The fields use the same approximation, so a single field could be used
# instead.
fields = {
    'temperature': ('real', 1, 'Omega', 1),
    'potential' : ('real', 1, 'Omega', 1),
}

variables = {
    'T' : ('unknown field', 'temperature', 0, 1),
    's' : ('test field', 'temperature', 'T'),
    'phi' : ('unknown field', 'potential', 1),
    'psi' : ('test field', 'potential', 'phi'),
    'phi_known' : ('parameter field', 'potential', '(set-to-None)'),
}

ics = {
    'ic' : ('Omega', {'T.0' : 0.0}),
}

ebcs = {
    'left' : ('Left', {'T.0' : 0.0, 'phi.0' : 0.0}),
    'right' : ('Right', {'T.0' : 2.0, 'phi.0' : 0.0}),
}

```

(continues on next page)

(continued from previous page)

```

    'inside' : ('Omega2_Surface', {'phi.0' : 'set_electric_bc'}),
}

def set_electric_bc(coor):
    y = coor[:,1]
    ymin, ymax = y.min(), y.max()
    val = 2.0 * (((y - ymin) / (ymax - ymin)) - 0.5)
    return val

functions = {
    'set_electric_bc' : (lambda ts, coor, bc, problem, **kwargs:
                        set_electric_bc(coor),),
}

equations = {
    '2' : """%.12e * dw_dot.2.Omega( s, dT/dt )
            + dw_laplace.2.Omega( m.thermal_conductivity, s, T )
            = dw_electric_source.2.Omega( m.electric_conductivity,
            s, phi_known ) """ % specific_heat,
    '1' : """dw_laplace.2.Omega( m.electric_conductivity, psi, phi ) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
        'problem'    : 'nonlinear',
    }),
    'ts' : ('ts.simple', {
        't0'         : t0,
        't1'         : t1,
        'dt'         : None,
        'n_step'     : n_step, # has precedence over dt!
        'verbose'    : 1,
    }),
}

def main():
    from sfepy.base.base import output
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.discrete import Problem

    output.prefix = 'therel:'

    required, other = get_standard_keywords()
    conf = ProblemConf.from_file(__file__, required, other)

    problem = Problem.from_conf(conf, init_equations=False)

    # Setup output directory according to options above.
    problem.setup_default_output()

```

(continues on next page)

(continued from previous page)

```
# First solve the stationary electric conduction problem.
problem.set_equations({'eq' : conf.equations['1']})
state_el = problem.solve()
problem.save_state(problem.get_output_name(suffix = 'el'), state_el)

# Then solve the evolutionary heat conduction problem, using state_el.
problem.set_equations({'eq' : conf.equations['2']})
phi_var = problem.get_variables()['phi_known']
phi_var.set_data(state_el())
problem.solve()

output('results saved in %s' % problem.get_output_name(suffix = '*'))

if __name__ == '__main__':
    main()
```

multi_physics/thermo_elasticity.py

Description

Thermo-elasticity with a given temperature distribution.

Uses *dw_biot* term with an isotropic coefficient for thermo-elastic coupling.

For given body temperature T and background temperature T_0 find $\underline{u}$ such that:

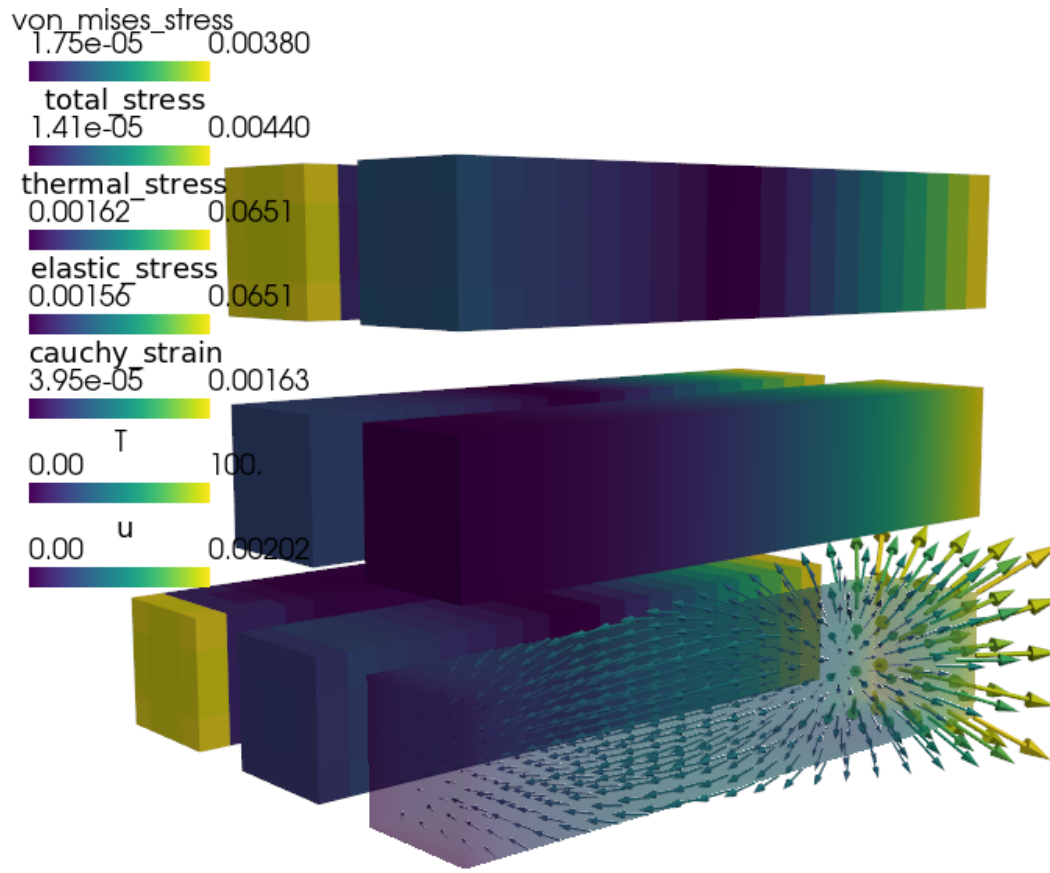
$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} (T - T_0) \alpha_{ij} e_{ij}(\underline{v}) = 0, \quad \forall \underline{v},$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

$$\alpha_{ij} = (3\lambda + 2\mu)\alpha\delta_{ij}$$

and α is the thermal expansion coefficient.



source code

```

r"""
Thermo-elasticity with a given temperature distribution.

Uses `dw_biot` term with an isotropic coefficient for thermo-elastic coupling.

For given body temperature :math:`T` and background temperature
:math:`T_0` find :math:`\ul{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
    - \int_{\Omega} (T - T_0) \ \alpha_{ij} \ e_{ij}(\ul{v})
    = 0
    \;, \quad \forall \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;, \;

```

(continues on next page)

(continued from previous page)

```

    \alpha_{ij} = (3 \lambda + 2 \mu) \alpha \delta_{ij}

and :math:`\alpha` is the thermal expansion coefficient.
"""
from __future__ import absolute_import
import numpy as np

from sfepy.base.base import Struct
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.mechanics.tensors import get_von_mises_stress
from sfepy import data_dir

# Material parameters.
lam = 10.0
mu = 5.0
thermal_expandability = 1.25e-5
T0 = 20.0 # Background temperature.

filename_mesh = data_dir + '/meshes/3d/block.mesh'

def get_temperature_load(ts, coors, region=None, **kwargs):
    """
    Temperature load depends on the `x` coordinate.
    """
    x = coors[:, 0]
    return (x - x.min())**2 - T0

def post_process(out, pb, state, extend=False):
    """
    Compute derived quantities: strain, stresses. Store also the loading
    temperature.
    """
    ev = pb.evaluate

    strain = ev('ev_cauchy_strain.2.Omega( u )', mode='el_avg')
    out['cauchy_strain'] = Struct(name='output_data',
                                mode='cell', data=strain,
                                dofs=None)

    e_stress = ev('ev_cauchy_stress.2.Omega( solid.D, u )', mode='el_avg')
    out['elastic_stress'] = Struct(name='output_data',
                                mode='cell', data=e_stress,
                                dofs=None)

    t_stress = ev('ev_biot_stress.2.Omega( solid.alpha, T )', mode='el_avg')
    out['thermal_stress'] = Struct(name='output_data',
                                mode='cell', data=t_stress,
                                dofs=None)

    out['total_stress'] = Struct(name='output_data',
                                mode='cell', data=e_stress + t_stress,
                                dofs=None)

```

(continues on next page)

(continued from previous page)

```

out['von_mises_stress'] = aux = out['total_stress'].copy()
vms = get_von_mises_stress(aux.data.squeeze())
vms.shape = (vms.shape[0], 1, 1, 1)
out['von_mises_stress'].data = vms

val = pb.get_variables()['T']()
val.shape = (val.shape[0], 1)
out['T'] = Struct(name='output_data',
                  mode='vertex', data=val + T0,
                  dofs=None)

return out

options = {
    'post_process_hook' : 'post_process',

    'nls' : 'newton',
    'ls' : 'ls',
}

functions = {
    'get_temperature_load' : (get_temperature_load,),
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
    'temperature': ('real', 1, 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'T' : ('parameter field', 'temperature',
          {'setter' : 'get_temperature_load'}),
}

ebcs = {
    'fix_u' : ('Left', {'u.all' : 0.0}),
}

eye_sym = np.array([[1], [1], [1], [0], [0], [0]], dtype=np.float64)
materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=lam, mu=mu),
        'alpha' : (3.0 * lam + 2.0 * mu) * thermal_expandability * eye_sym
    },),
}

```

(continues on next page)

(continued from previous page)

```

equations = {
    'balance_of_forces' :
        """dw_lin_elastic.2.Omega( solid.D, v, u )
        - dw_biot.2.Omega( solid.alpha, v, T )
        = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 1,
        'eps_a'      : 1e-10,
    }),
}

```

multi_physics/thermo_elasticity_ess.py

Description

Thermo-elasticity with a computed temperature demonstrating equation sequence solver.

Uses *dw_biot* term with an isotropic coefficient for thermo-elastic coupling.

The equation sequence solver ('ess' in solvers) automatically solves first the temperature distribution and then the elasticity problem with the already computed temperature.

Find $\underline{u}$, T such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} (T - T_0) \alpha_{ij} e_{ij}(\underline{v}) = 0, \quad \forall \underline{v},$$

$$\int_{\Omega} \nabla s \cdot \nabla T = 0, \quad \forall s.$$

where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

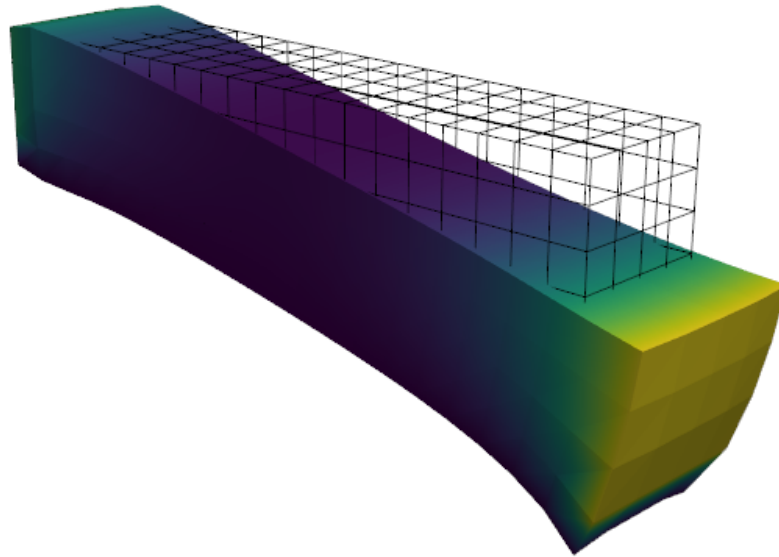
$$\alpha_{ij} = (3\lambda + 2\mu)\alpha\delta_{ij},$$

T_0 is the background temperature and α is the thermal expansion coefficient.

Notes

The gallery image was produced by (plus proper view settings):

```
sfepy-view block.vtk -f T:p1 u:wu:f1000:p0 u:vw:p0
```

source code

```
r"""
Thermo-elasticity with a computed temperature demonstrating equation sequence
solver.

Uses `dw_biot` term with an isotropic coefficient for thermo-elastic coupling.

The equation sequence solver (`ess` in `solvers`) automatically solves
first the temperature distribution and then the elasticity problem with the
already computed temperature.

Find :math:\ul{u}`, :math:T` such that:

.. math::
    \int_{\Omega} D_{ijkl} \ul{e}_{ij}(\ul{v}) \ul{e}_{kl}(\ul{u})
    - \int_{\Omega} (T - T_0) \alpha_{ij} \ul{e}_{ij}(\ul{v})
    = 0
    \quad \forall \ul{v} \;,

    \int_{\Omega} \nabla s \cdot \nabla T
    = 0
    \quad \forall s \;.
```

(continues on next page)

(continued from previous page)

where

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;, \; \;

    \alpha_{ij} = (3 \lambda + 2 \mu) \alpha \delta_{ij} \;,

:math:`T_0` is the background temperature and :math:`\alpha` is the thermal
expansion coefficient.
```

Notes

The gallery image was produced by (plus proper view settings)::

```
sfepy-view block.vtk -f T:p1 u:wu:f1000:p0 u:vw:p0
"""
from __future__ import absolute_import
import numpy as np

from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy import data_dir

# Material parameters.
lam = 10.0
mu = 5.0
thermal_expandability = 1.25e-5
T0 = 20.0 # Background temperature.

filename_mesh = data_dir + '/meshes/3d/block.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',

    'block_solve' : True,
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -4.99)', 'facet'),
    'Right' : ('vertices in (x > 4.99)', 'facet'),
    'Bottom' : ('vertices in (z < -0.99)', 'facet'),
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
    'temperature': ('real', 1, 'Omega', 1),
}
```

(continues on next page)

(continued from previous page)

```

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'T' : ('unknown field', 'temperature', 1),
    's' : ('test field', 'temperature', 'T'),
}

ebcs = {
    'u0' : ('Left', {'u.all' : 0.0}),
    't0' : ('Left', {'T.0' : 20.0}),
    't2' : ('Bottom', {'T.0' : 0.0}),
    't1' : ('Right', {'T.0' : 30.0}),
}

eye_sym = np.array([[1], [1], [1], [0], [0], [0]], dtype=np.float64)
materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=lam, mu=mu),
        'alpha' : (3.0 * lam + 2.0 * mu) * thermal_expandability * eye_sym
    },),
}

equations = {
    'balance_of_forces' : """
        + dw_lin_elastic.2.Omega(solid.D, v, u)
        - dw_biot.2.Omega(solid.alpha, v, T)
        = 0
    """,
    'temperature' : """
        + dw_laplace.1.Omega(s, T)
        = 0
    """
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

```

navier_stokes

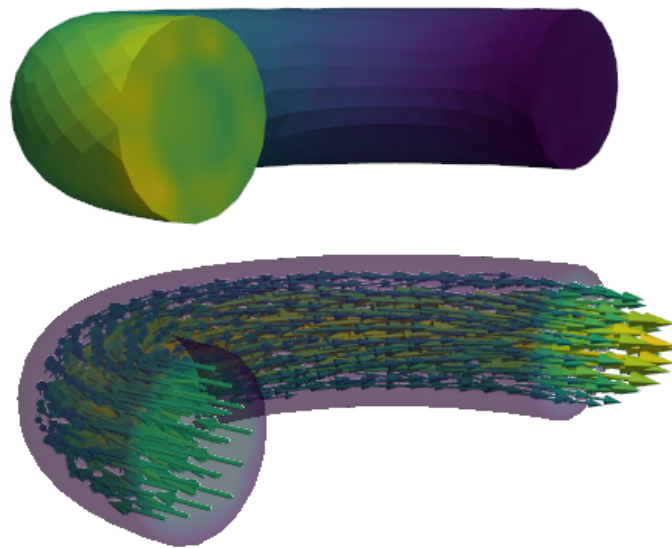
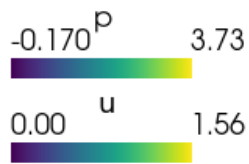
navier_stokes/navier_stokes.py

Description

Navier-Stokes equations for incompressible fluid flow.

Find $\underline{u}$, p such that:

$$\begin{aligned} \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q. \end{aligned}$$



source code

```
r"""
Navier-Stokes equations for incompressible fluid flow.

Find :math:\underline{u}, :math:p` such that:

.. math::
    \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}
```

(continues on next page)

(continued from previous page)

```

+ \int_{\Omega} ((\ul{u} \cdot \nabla) \ul{u}) \cdot \ul{v}
- \int_{\Omega} p \nabla \cdot \ul{v}
= 0
\;, \quad \forall \ul{v} \;,

\int_{\Omega} q \nabla \cdot \ul{u}
= 0
\;, \quad \forall q \;.
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/elbow2.mesh'

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'verify_incompressibility',

    # Options for saving higher-order variables.
    # Possible kinds:
    #   'strip' ... just remove extra DOFs (ignores other linearization
    #               options)
    #   'adaptive' ... adaptively refine linear element mesh.
    'linearization' : {
        'kind' : 'strip',
        'min_level' : 1, # Min. refinement level to achieve everywhere.
        'max_level' : 2, # Max. refinement level.
        'eps' : 1e-1, # Relative error tolerance.
    },
}

field_1 = {
    'name' : '3_velocity',
    'dtype' : 'real',
    'shape' : (3,),
    'region' : 'Omega',
    'approx_order' : '1B',
}

field_2 = {
    'name' : 'pressure',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

# Can use logical operations '&' (and), '|' (or).
region_1000 = {
    'name' : 'Omega',
    'select' : 'cells of group 6',

```

(continues on next page)

(continued from previous page)

```

}

region_0 = {
    'name' : 'Walls',
    'select' : 'vertices of surface -v (r.Outlet +v r.Inlet)',
    'kind' : 'facet',
}
region_1 = {
    'name' : 'Inlet',
    'select' : 'vertices by cinc0', # In
    'kind' : 'facet',
}
region_2 = {
    'name' : 'Outlet',
    'select' : 'vertices by cinc1', # Out
    'kind' : 'facet',
}

ebc_1 = {
    'name' : 'Walls',
    'region' : 'Walls',
    'dofs' : {'u.all' : 0.0},
}
ebc_2 = {
    'name' : 'Inlet',
    'region' : 'Inlet',
    'dofs' : {'u.1' : 1.0, 'u.[0,2]' : 0.0},
}

material_1 = {
    'name' : 'fluid',
    'values' : {
        'viscosity' : 1.25e-3,
        'density' : 1e0,
    },
}

variable_1 = {
    'name' : 'u',
    'kind' : 'unknown field',
    'field' : '3_velocity',
    'order' : 0,
}
variable_2 = {
    'name' : 'v',
    'kind' : 'test field',
    'field' : '3_velocity',
    'dual' : 'u',
}
variable_3 = {
    'name' : 'p',
    'kind' : 'unknown field',

```

(continues on next page)

(continued from previous page)

```

        'field' : 'pressure',
        'order' : 1,
    }
variable_4 = {
    'name' : 'q',
    'kind' : 'test field',
    'field' : 'pressure',
    'dual' : 'p',
}
variable_5 = {
    'name' : 'pp',
    'kind' : 'parameter field',
    'field' : 'pressure',
    'like' : 'p',
}

integral_1 = {
    'name' : 'i1',
    'order' : 2,
}
integral_2 = {
    'name' : 'i2',
    'order' : 3,
}

##
# Stationary Navier-Stokes equations.
equations = {
    'balance' :
        """+ dw_div_grad.i2.Omega( fluid.viscosity, v, u )
          + dw_convect.i2.Omega( v, u )
          - dw_stokes.i1.Omega( v, p ) = 0""",
    'incompressibility' :
        """dw_stokes.i1.Omega( u, q ) = 0""",
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 5,
    'eps_a'      : 1e-8,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'     : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,

```

(continues on next page)

(continued from previous page)

```

'ls_on'      : 0.99999,
'ls_min'     : 1e-5,
'check'      : 0,
'delta'      : 1e-6,
}

def verify_incompressibility(out, problem, variables, extend=False):
    """This hook is normally used for post-processing (additional results can
    be inserted into `out` dictionary), but here we just verify the weak
    incompressibility condition."""
    from sfepy.base.base import nm, output, assert_

    one = nm.ones((variables['p'].field.n_nod,), dtype=nm.float64)
    variables.set_state_parts({'p' : one})
    zero = problem.evaluate('dw_stokes.i1.Omega(u, p)')
    output('div(u) = %.3e' % zero)

    assert_(abs(zero) < 1e-14)

    return out

##
# Functions.
import os.path as op
import sys

sys.path.append(data_dir) # Make installed example work.
import sfepy.examples.navier_stokes.utils as utils

cinc_name = 'cinc_' + op.splitext(op.basename(filename_mesh))[0]
cinc = getattr(utils, cinc_name)

functions = {
    'cinc0' : (lambda coors, domain=None: cinc(coors, 0),),
    'cinc1' : (lambda coors, domain=None: cinc(coors, 1),),
}

```

navier_stokes/navier_stokes2d.py

Description

Navier-Stokes equations for incompressible fluid flow in 2D.

Find $\underline{u}$, p such that:

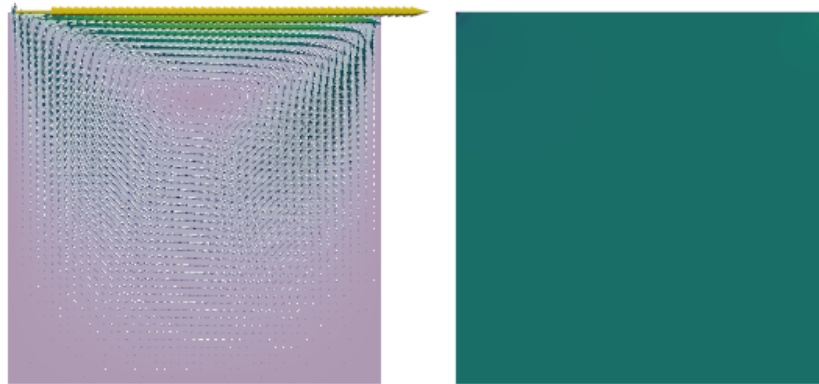
$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} = 0, \quad \forall \underline{v},$$

$$\int_{\Omega} q \nabla \cdot \underline{u} = 0, \quad \forall q.$$

The mesh is created by `gen_block_mesh()` function.

View the results using:


```
sfepy-view user_block.vtk -2
```



source code

```
# -*- coding: utf-8 -*-
r"""
Navier-Stokes equations for incompressible fluid flow in 2D.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
\int_{\Omega} \nu \nabla \mathbf{v} : \nabla \mathbf{u}
+ \int_{\Omega} ((\mathbf{u} \cdot \nabla) \mathbf{u}) \cdot \mathbf{v}
- \int_{\Omega} p \nabla \cdot \mathbf{v}
= 0
\quad \forall \mathbf{v};

\int_{\Omega} q \nabla \cdot \mathbf{u}
= 0
\quad \forall q.

The mesh is created by ``gen_block_mesh()`` function.
```

(continues on next page)

(continued from previous page)

View the results using::

```

    sfepy-view user_block.vtk -2
"""
from __future__ import absolute_import
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh

# Mesh dimensions.
dims = [0.1, 0.1]

# Mesh resolution: increase to improve accuracy.
shape = [51, 51]

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, [0, 0], name='user_block',
                              verbose=False)

        return mesh

    elif mode == 'write':
        pass

filename_mesh = UserMeshIO(mesh_hook)

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -0.0499)', 'facet'),
    'Right' : ('vertices in (x > 0.0499)', 'facet'),
    'Bottom' : ('vertices in (y < -0.0499)', 'facet'),
    'Top' : ('vertices in (y > 0.0499)', 'facet'),
    'Walls' : ('r.Left +v r.Right +v r.Bottom', 'facet'),
}

materials = {
    'fluid' : ({'viscosity' : 1.00e-2},),
}

fields = {
    'velocity': ('real', 'vector', 'Omega', 2),
    'pressure': ('real', 'scalar', 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'velocity', 0),
    'v' : ('test field', 'velocity', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

```

(continues on next page)

(continued from previous page)

```

ebcs = {
    '1_Walls' : ('Walls', {'u.all' : 0.0}),
    '0_Driven' : ('Top', {'u.0' : 1.0, 'u.1' : 0.0}),
    'Pressure' : ('Bottom', {'p.0' : 0.0}),
}

integrals = {
    'i' : 4,
}

equations = {
    'balance' :
        """+ dw_div_grad.i.Omega(fluid.viscosity, v, u)
        + dw_convect.i.Omega(v, u)
        - dw_stokes.i.Omega(v, p) = 0""",

    'incompressibility' :
        """dw_stokes.i.Omega(u, q) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 15,
        'eps_a'       : 1e-10,
        'eps_r'       : 1.0,
    }),
}

```

navier_stokes/navier_stokes2d_iga.py

Description

Navier-Stokes equations for incompressible fluid flow in 2D solved in a single patch NURBS domain using the isogeometric analysis (IGA) approach.

Find $\underline{u}, p$ such that:

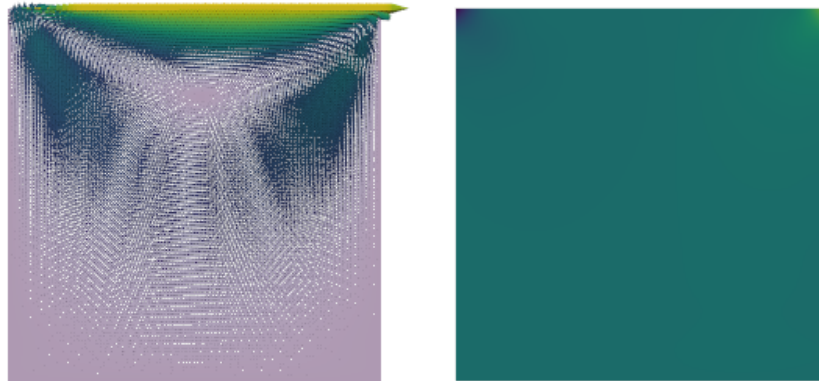
$$\begin{aligned}
 \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\
 \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q.
 \end{aligned}$$

The domain geometry was created by:

```
sfepy-mesh iga-patch -2 -d 0.1,0.1 -s 10,10 -o meshes/iga/block2d.iga
```

View the results using:

```
sfepy-view block2d.vtk -2
```



source code

```
# -*- coding: utf-8 -*-
r"""
Navier-Stokes equations for incompressible fluid flow in 2D solved in a single
patch NURBS domain using the isogeometric analysis (IGA) approach.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
\int_{\Omega} \nu \nabla \mathbf{v} : \nabla \mathbf{u}
+ \int_{\Omega} ((\mathbf{u} \cdot \nabla) \mathbf{u}) \cdot \mathbf{v}
- \int_{\Omega} p \nabla \cdot \mathbf{v}
= 0
\quad \forall \mathbf{v};

\int_{\Omega} q \nabla \cdot \mathbf{u}
= 0
\quad \forall q.

The domain geometry was created by:

sfePy-mesh iga-patch -2 -d 0.1,0.1 -s 10,10 -o meshes/iga/block2d.iga
```

(continues on next page)

(continued from previous page)

View the results using::

```

    sfepy-view block2d.vtk -2
"""
from __future__ import absolute_import
from sfepy import data_dir

filename_domain = data_dir + '/meshes/iga/block2d.iga'

regions = {
    'Omega' : 'all',
    'Left' : ('vertices of set xi00', 'facet'),
    'Right' : ('vertices of set xi01', 'facet'),
    'Bottom' : ('vertices of set xi10', 'facet'),
    'Top' : ('vertices of set xi11', 'facet'),
    'Walls' : ('r.Left +v r.Right +v r.Bottom', 'facet'),
}

materials = {
    'fluid' : ({'viscosity' : 1.00e-2},),
}

fields = {
    'velocity': ('real', 'vector', 'Omega', 'iga+1', 'H1', 'iga'),
    'pressure': ('real', 'scalar', 'Omega', 'iga', 'H1', 'iga'),
}

variables = {
    'u' : ('unknown field', 'velocity', 0),
    'v' : ('test field', 'velocity', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

ebcs = {
    '1_Walls' : ('Walls', {'u.all' : 0.0}),
    '0_Driven' : ('Top', {'u.0' : 1.0, 'u.1' : 0.0}),
    'Pressure' : ('Bottom', {'p.0' : 0.0}),
}

integrals = {
    'i' : 6,
}

equations = {
    'balance' :
        """+ dw_div_grad.i.Omega(fluid.viscosity, v, u)
          + dw_convect.i.Omega(v, u)
          - dw_stokes.i.Omega(v, p) = 0""",

    'incompressibility' :

```

(continues on next page)

(continued from previous page)

```

        """dw_stokes.i.Omega(u, q) = 0""",
    }

    solvers = {
        'ls' : ('ls.scipy_direct', {}),
        'newton' : ('nls.newton', {
            'i_max'      : 15,
            'eps_a'       : 1e-10,
            'eps_r'       : 1.0,
        }),
    }

```

navier_stokes/stabilized_navier_stokes.py

Description

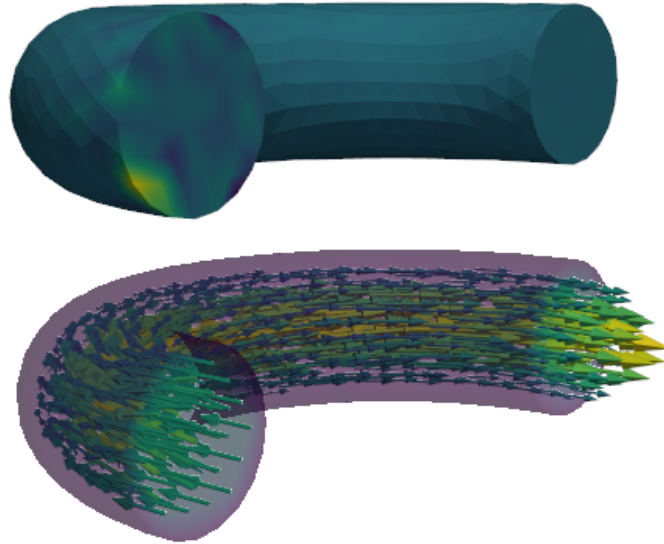
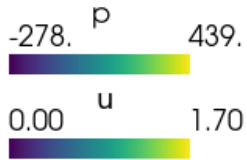
Stabilized Navier-Stokes problem with grad-div, SUPG and PSPG stabilization solved by a custom Oseen solver.

The stabilization terms are described in [1].

[1] G. Matthies and G. Lube. On streamline-diffusion methods of inf-sup stable discretisations of the generalised Oseen problem. Number 2007-02 in Preprint Series of Institut fuer Numerische und Angewandte Mathematik, Georg-August-Universitaet Goettingen, 2007.

Find $\underline{u}, p$ such that:

$$\begin{aligned}
 & \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} \int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} \\
 & + \gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v}) \\
 & + \sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v}) \\
 & + \sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v}) = 0, \quad \forall \underline{v}, \\
 & \int_{\Omega} q \nabla \cdot \underline{u} \\
 & + \sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q \\
 & + \sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q = 0, \quad \forall q.
 \end{aligned}$$



source code

```
r"""
Stabilized Navier-Stokes problem with grad-div, SUPG and PSPG stabilization
solved by a custom Oseen solver.

The stabilization terms are described in [1].

[1] G. Matthies and G. Lube. On streamline-diffusion methods of inf-sup stable
discretisations of the generalised Oseen problem. Number 2007-02 in Preprint
Series of Institut fuer Numerische und Angewandte Mathematik,
Georg-August-Universitaet Goettingen, 2007.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
\begin{array}{l}
\int_{\Omega} \nu \nabla \mathbf{v} : \nabla \mathbf{u} \\
\int_{\Omega} ((\mathbf{b} \cdot \nabla) \mathbf{u}) \cdot \mathbf{v} \\
- \int_{\Omega} p \nabla \cdot \mathbf{v} \\
+ \gamma \int_{\Omega} (\nabla \cdot \mathbf{u}) \cdot (\nabla \cdot \mathbf{v}) \\
+ \sum_{K \in \mathcal{T}_h} \Delta_K ((\mathbf{b} \cdot \nabla) \mathbf{u}) \cdot (\mathbf{b} \cdot \nabla) \mathbf{v} \\
\end{array}
```

(continues on next page)

(continued from previous page)

```

+ \sum_{K \in \mathcal{I}_{cal\_h}} \int_{T_K} \delta_K \nabla p \cdot (\nabla b \cdot \nabla v)
= 0
\;, \quad \forall v \;,
\end{array}

\begin{array}{l}
\int_{\Omega} q \nabla \cdot u \, \\
+ \sum_{K \in \mathcal{I}_{cal\_h}} \int_{T_K} \tau_K (\nabla b \cdot \nabla u) \cdot \nabla q \, \\
+ \sum_{K \in \mathcal{I}_{cal\_h}} \int_{T_K} \tau_K \nabla p \cdot \nabla q \, \\
= 0
\;, \quad \forall q \;.
\end{array}
"""
from __future__ import absolute_import
from sfepy.solvers.oseen import StabilizationFunction
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/elbow2.mesh'

options = {
    'solution' : 'steady',
    'nls' : 'oseen',
    'ls' : 'ls',
}

regions = {
    'Omega' : 'all',
    'Walls' : ('vertices of surface -v (r.Outlet +v r.Inlet)', 'facet'),
    'Inlet' : ('vertices by cinc0', 'facet'),
    'Outlet' : ('vertices by cinc1', 'facet'),
}

fields = {
    'velocity' : ('real', 3, 'Omega', 1),
    'pressure' : ('real', 1, 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'velocity', 0),
    'v' : ('test field', 'velocity', 'u'),
    'b' : ('parameter field', 'velocity', 'u'),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

ebcs = {
    'Walls_velocity' : ('Walls', {'u.all' : 0.0}),
    'Inlet_velocity' : ('Inlet', {'u.1' : 1.0, 'u.[0,2]' : 0.0}),
}

```

(continues on next page)

(continued from previous page)

```

materials = {
    'fluid' : ({'viscosity' : 1.25e-5,
                'density' : 1e0},),
    'stabil' : 'stabil',
}

integrals = {
    'i1' : 2,
    'i2' : 3,
}

##
# Stationary Navier-Stokes equations with grad-div, SUPG and PSPG stabilization.
equations = {
    'balance' :
        """ dw_div_grad.i2.Omega( fluid.viscosity, v, u )
            + dw_lin_convect.i2.Omega( v, b, u )
            - dw_stokes.i1.Omega( v, p )
            + dw_st_grad_div.i1.Omega( stabil.gamma, v, u )
            + dw_st_supg_c.i1.Omega( stabil.delta, v, b, u )
            + dw_st_supg_p.i1.Omega( stabil.delta, v, b, p )
            = 0""",
    'incompressibility' :
        """ dw_stokes.i1.Omega( u, q )
            + dw_st_pspg_c.i1.Omega( stabil.tau, q, b, u )
            + dw_st_pspg_p.i1.Omega( stabil.tau, q, p )
            = 0""",
}

solver_1 = {
    'name' : 'oseen',
    'kind' : 'nls.oseen',

    'stabil_mat' : 'stabil',

    'adimensionalize' : False,
    'check_navier_stokes_residual' : False,

    'i_max'      : 10,
    'eps_a'      : 1e-8,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).

    # Uncomment the following to get a convergence log.
    ## 'log'      : {'text' : 'oseen_log.txt',
    ##               'plot' : 'oseen_log.png'},
}

solver_2 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',

```

(continues on next page)

(continued from previous page)

```

}

##
# Functions.
import os.path as op
import sys

sys.path.append(data_dir) # Make installed example work.
import sfepy.examples.navier_stokes.utils as utils

cinc_name = 'cinc_' + op.splitext(op.basename(filename_mesh))[0]
cinc = getattr(utils, cinc_name)

name_map = {'p' : 'p', 'q' : 'q', 'u' : 'u', 'b' : 'b', 'v' : 'v',
            'fluid' : 'fluid', 'omega' : 'omega', 'i1' : 'i1', 'i2' : 'i2',
            'viscosity' : 'viscosity', 'velocity' : 'velocity',
            'gamma' : 'gamma', 'delta' : 'delta', 'tau' : 'tau'}

functions = {
    'cinc0' : (lambda coors, domain=None: cinc(coors, 0)),
    'cinc1' : (lambda coors, domain=None: cinc(coors, 1)),
    'stabil' : (StabilizationFunction(name_map)),
}

```

navier_stokes/stokes.py

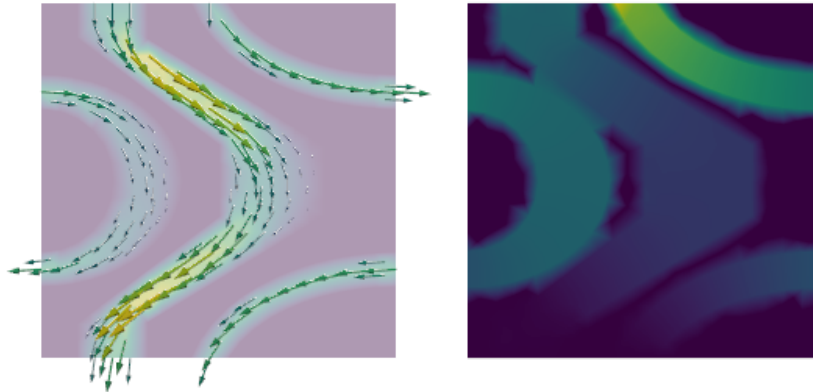
Description

Stokes equations for incompressible fluid flow.

This example demonstrates fields defined on subdomains as well as use of periodic boundary conditions.

Find $\underline{u}$, p such that:

$$\begin{aligned}
 \int_{Y_1 \cup Y_2} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{Y_1 \cup Y_2} p \nabla \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\
 \int_{Y_1 \cup Y_2} q \nabla \cdot \underline{u} &= 0, \quad \forall q.
 \end{aligned}$$



source code

```

r"""
Stokes equations for incompressible fluid flow.

This example demonstrates fields defined on subdomains as well as use of
periodic boundary conditions.

Find  $\mathbf{u}$ ,  $p$  such that:

.. math::
    \int_{Y_1 \cup Y_2} \nu \nabla \mathbf{v} : \nabla \mathbf{u}
    - \int_{Y_1 \cup Y_2} p \nabla \cdot \mathbf{v}
    = 0
    \quad \forall \mathbf{v};

    \int_{Y_1 \cup Y_2} q \nabla \cdot \mathbf{u}
    = 0
    \quad \forall q;

"""
from __future__ import absolute_import
from sfepy import data_dir
from sfepy.discrete.fem.periodic import match_y_line

```

(continues on next page)

(continued from previous page)

```

filename_mesh = data_dir + '/meshes/2d/special/channels_symm944t.mesh'

if filename_mesh.find( 'symm' ):
    region_1 = {
        'name' : 'Y1',
        'select' : ""cells of group 3"",
    }
    region_2 = {
        'name' : 'Y2',
        'select' : ""cells of group 4 +c cells of group 6
                    +c cells of group 8"",
    }
    region_4 = {
        'name' : 'Y1Y2',
        'select' : ""r.Y1 +c r.Y2"",
    }
    region_5 = {
        'name' : 'Walls',
        'select' : ""r.EBCGamma1 +v r.EBCGamma2"",
        'kind' : 'facet',
    }
    region_310 = {
        'name' : 'EBCGamma1',
        'select' : ""(cells of group 1 *v cells of group 3)
                    +v
                    (cells of group 2 *v cells of group 3)
                    "",
        'kind' : 'facet',
    }
    region_320 = {
        'name' : 'EBCGamma2',
        'select' : ""(cells of group 5 *v cells of group 4)
                    +v
                    (cells of group 1 *v cells of group 4)
                    +v
                    (cells of group 7 *v cells of group 6)
                    +v
                    (cells of group 2 *v cells of group 6)
                    +v
                    (cells of group 9 *v cells of group 8)
                    +v
                    (cells of group 2 *v cells of group 8)
                    "",
        'kind' : 'facet',
    }

w2 = 0.499
# Sides.
region_20 = {
    'name' : 'Left',

```

(continues on next page)

(continued from previous page)

```

        'select' : 'vertices in (x < %.3f)' % -w2,
        'kind' : 'facet',
    }
    region_21 = {
        'name' : 'Right',
        'select' : 'vertices in (x > %.3f)' % w2,
        'kind' : 'facet',
    }
    region_22 = {
        'name' : 'Bottom',
        'select' : 'vertices in (y < %.3f)' % -w2,
        'kind' : 'facet',
    }
    region_23 = {
        'name' : 'Top',
        'select' : 'vertices in (y > %.3f)' % w2,
        'kind' : 'facet',
    }

    field_1 = {
        'name' : '2_velocity',
        'dtype' : 'real',
        'shape' : (2,),
        'region' : 'Y1Y2',
        'approx_order' : 2,
    }

    field_2 = {
        'name' : 'pressure',
        'dtype' : 'real',
        'shape' : (1,),
        'region' : 'Y1Y2',
        'approx_order' : 1,
    }

    variable_1 = {
        'name' : 'u',
        'kind' : 'unknown field',
        'field' : '2_velocity',
        'order' : 0,
    }
    variable_2 = {
        'name' : 'v',
        'kind' : 'test field',
        'field' : '2_velocity',
        'dual' : 'u',
    }
    variable_3 = {
        'name' : 'p',
        'kind' : 'unknown field',
        'field' : 'pressure',
        'order' : 1,
    }

```

(continues on next page)

(continued from previous page)

```

}
variable_4 = {
    'name' : 'q',
    'kind' : 'test field',
    'field' : 'pressure',
    'dual' : 'p',
}

integral_1 = {
    'name' : 'i',
    'order' : 2,
}

equations = {
    'balance' :
        """dw_div_grad.i.Y1Y2( fluid.viscosity, v, u )
        - dw_stokes.i.Y1Y2( v, p ) = 0""",
    'incompressibility' :
        """dw_stokes.i.Y1Y2( u, q ) = 0""",
}

material_1 = {
    'name' : 'fluid',
    'values' : {
        'viscosity' : 1.0,
        'density' : 1e0,
    },
}

ebc_1 = {
    'name' : 'walls',
    'region' : 'Walls',
    'dofs' : {'u.all' : 0.0},
}

ebc_2 = {
    'name' : 'top_velocity',
    'region' : 'Top',
    'dofs' : {'u.1' : -1.0, 'u.0' : 0.0},
}

ebc_10 = {
    'name' : 'bottom_pressure',
    'region' : 'Bottom',
    'dofs' : {'p.0' : 0.0},
}

epbc_1 = {
    'name' : 'u_rl',
    'region' : ['Left', 'Right'],
    'dofs' : {'u.all' : 'u.all', 'p.0' : 'p.0'},
    'match' : 'match_y_line',
}

```

(continues on next page)

(continued from previous page)

```

functions = {
    'match_y_line' : (match_y_line,),
}

solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 2,
    'eps_a'      : 1e-8,
    'eps_r'      : 1e-2,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 1.1,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
}

save_format = 'hdf5' # 'hdf5' or 'vtk'

```

navier_stokes/stokes_slip_bc.py

Description

Incompressible Stokes flow with Navier (slip) boundary conditions, flow driven by a moving wall and a small diffusion for stabilization.

This example demonstrates the use of *no-penetration* and *edge direction* boundary conditions together with Navier or slip boundary conditions. Alternatively the *no-penetration* boundary conditions can be applied in a weak sense using the penalty term `dw_non_penetration_p`.

Find $\underline{u}, p$ such that:

$$\begin{aligned}
 \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{\Omega} p \nabla \cdot \underline{v} + \int_{\Gamma_1} \beta \underline{v} \cdot (\underline{u} - \underline{u}_d) + \int_{\Gamma_2} \beta \underline{v} \cdot \underline{u} &= 0, \quad \forall \underline{v}, \\
 \int_{\Omega} \mu \nabla q \cdot \nabla p + \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q,
 \end{aligned}$$

where ν is the fluid viscosity, β is the slip coefficient, μ is the (small) numerical diffusion coefficient, Γ_1 is the top wall that moves with the given driving velocity $\underline{u}_d$ and Γ_2 are the remaining walls. The Navier conditions are in effect on both Γ_1, Γ_2 and are expressed by the corresponding integrals in the equations above.

The *no-penetration* boundary conditions are applied on Γ_1, Γ_2 , except the vertices of the block edges, where the *edge direction* boundary conditions are applied.

The penalty term formulation is given by the following equations.

Find $\underline{u}$, p such that:

$$\begin{aligned} \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{\Omega} p \nabla \cdot \underline{v} + \int_{\Gamma_1} \beta \underline{v} \cdot (\underline{u} - \underline{u}_d) + \int_{\Gamma_2} \beta \underline{v} \cdot \underline{u} + \int_{\Gamma_1 \cup \Gamma_2} \epsilon (\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u}) &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} \mu \nabla q \cdot \nabla p + \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q, \end{aligned}$$

where ϵ is the penalty coefficient (sufficiently large). The *no-penetration* boundary conditions are applied on Γ_1, Γ_2 .

Optionally, Dirichlet boundary conditions can be applied on the inlet in the both cases, see below.

For large meshes use the 'ls_i' linear solver - PETSc + petsc4py are needed in that case.

Several parameters can be set using the `--define` option of `sfepy-run`, see `define()` and the examples below.

Examples

Specify the inlet velocity and a finer mesh:

```
sfepy-run sfepy/examples/navier_stokes/stokes_slip_bc -d shape="(11,31,31),u_inlet=0.5"
sfepy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Use the penalty term formulation and einsum-based terms with the default (numpy) backend:

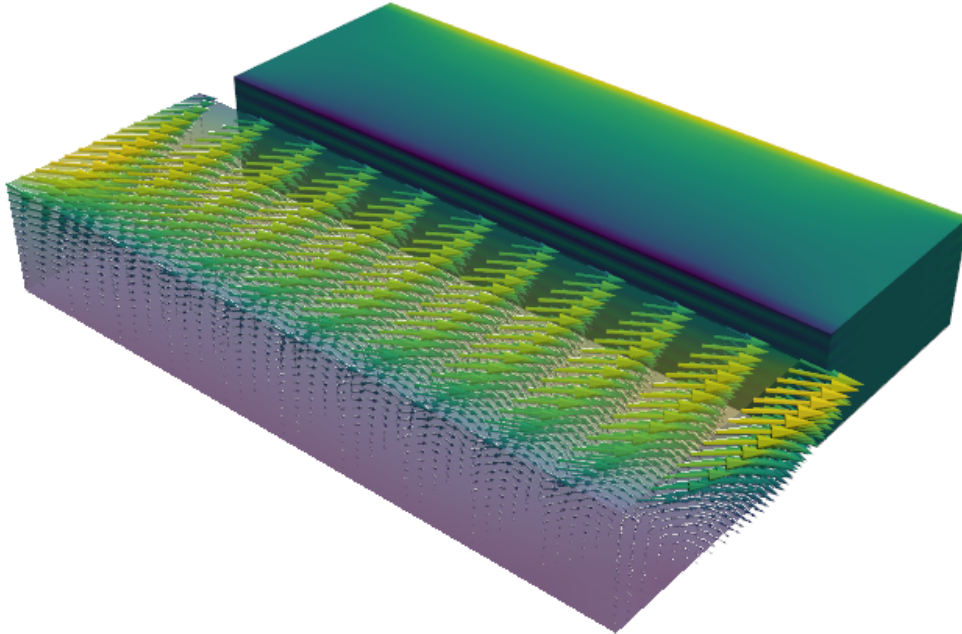
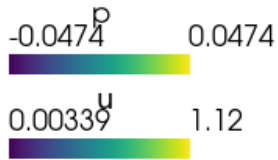
```
sfepy-run sfepy/examples/navier_stokes/stokes_slip_bc -d "mode=penalty,term_mode=einsum"
sfepy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Change backend to `opt_einsum` (needs to be installed) and use the quadratic velocity approximation order:

```
sfepy-run sfepy/examples/navier_stokes/stokes_slip_bc -d "u_order=2,mode=penalty,term_
↪mode=einsum,backend=opt_einsum,optimize=auto"
sfepy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Note the pressure field distribution improvement w.r.t. the previous examples. If PETSc + petsc4py are installed, try using the iterative solver to speed up the solution:

```
sfepy-run sfepy/examples/navier_stokes/stokes_slip_bc -d "u_order=2,ls=ls_i,mode=penalty,
↪term_mode=einsum,backend=opt_einsum,optimize=auto"
sfepy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

source code

```
r"""
Incompressible Stokes flow with Navier (slip) boundary conditions, flow driven
by a moving wall and a small diffusion for stabilization.

This example demonstrates the use of `no-penetration` and `edge direction`
boundary conditions together with Navier or slip boundary conditions.
Alternatively the `no-penetration` boundary conditions can be applied in a weak
sense using the penalty term ``dw_non_penetration_p``.

Find :math:\mathbf{u}, :math:p` such that:

.. math::
\int_{\Omega} \nu \nabla \mathbf{v} : \nabla \mathbf{u}
- \int_{\Omega} p \nabla \cdot \mathbf{v}
+ \int_{\Gamma_1} \beta \mathbf{v} \cdot (\mathbf{u} - \mathbf{u}_d)
+ \int_{\Gamma_2} \beta \mathbf{v} \cdot \mathbf{u}
= 0
\quad \forall \mathbf{v} \quad ;,

\int_{\Omega} \mu \nabla q \cdot \nabla p
+ \int_{\Omega} q \nabla \cdot \mathbf{u}
```

(continues on next page)

(continued from previous page)

```
= 0
\;, \quad \forall q \;,
```

where ν is the fluid viscosity, β is the slip coefficient, μ is the (small) numerical diffusion coefficient, Γ_1 is the top wall that moves with the given driving velocity u_d and Γ_2 are the remaining walls. The Navier conditions are in effect on both Γ_1 , Γ_2 and are expressed by the corresponding integrals in the equations above.

The 'no-penetration' boundary conditions are applied on Γ_1 , Γ_2 , except the vertices of the block edges, where the 'edge direction' boundary conditions are applied.

The penalty term formulation is given by the following equations.

Find u , p such that:

```
.. math::
\int_{\Omega} \nu \nabla u : \nabla v : \nabla u
- \int_{\Omega} p \nabla \cdot v
+ \int_{\Gamma_1} \beta v \cdot (u - u_d)
+ \int_{\Gamma_2} \beta v \cdot u
+ \int_{\Gamma_1 \cup \Gamma_2} \epsilon (n \cdot v)
(n \cdot u)
= 0
\;, \quad \forall v \;,

\int_{\Omega} \mu \nabla q \cdot \nabla p
+ \int_{\Omega} q \nabla \cdot u
= 0
\;, \quad \forall q \;,
```

where ϵ is the penalty coefficient (sufficiently large). The 'no-penetration' boundary conditions are applied on Γ_1 , Γ_2 .

Optionally, Dirichlet boundary conditions can be applied on the inlet in the both cases, see below.

For large meshes use the `ls_i` linear solver - PETSc + petsc4py are needed in that case.

Several parameters can be set using the `--define` option of `sfepy-run`, see `func:define()` and the examples below.

Examples

Specify the inlet velocity and a finer mesh::

```
sfepy-run sfepy/examples/navier_stokes/stokes_slip_bc -d shape="(11,31,31),u_inlet=0.5"
```

(continues on next page)

(continued from previous page)

```
sfeipy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Use the penalty term formulation and einsum-based terms with the default (numpy) backend::

```
sfeipy-run sfeipy/examples/navier_stokes/stokes_slip_bc -d "mode=penalty,term_mode=einsum"
sfeipy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Change backend to opt_einsum (needs to be installed) and use the quadratic velocity approximation order::

```
sfeipy-run sfeipy/examples/navier_stokes/stokes_slip_bc -d "u_order=2,mode=penalty,term_
mode=einsum,backend=opt_einsum,optimize=auto"
sfeipy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

Note the pressure field distribution improvement w.r.t. the previous examples. If PETSc + petsc4py are installed, try using the iterative solver to speed up the solution::

```
sfeipy-run sfeipy/examples/navier_stokes/stokes_slip_bc -d "u_order=2,ls=ls_i,
mode=penalty,term_mode=einsum,backend=opt_einsum,optimize=auto"
sfeipy-view -f p:p0 u:o.4:p1 u:g:f0.2:p1 -- user_block.vtk
```

```
import os
from functools import partial
import numpy as nm
```

```
from sfeipy.base.base import assert_, output
from sfeipy.discrete.fem.meshio import UserMeshIO
from sfeipy.mesh.mesh_generators import gen_block_mesh
from sfeipy.homogenization.utils import define_box_regions
```

```
def define(dims=(3, 1, 0.5), shape=(11, 15, 15), u_order=1, refine=0,
           ls='ls_d', u_inlet=None, mode='lcbc', term_mode='original',
           backend='numpy', optimize='optimal', verbosity=0, output_dir='',
           save_lcbc_vecs=False):
```

```
    """
```

```
    Parameters
```

```
    -----
```

```
    dims : tuple
```

```
        The block domain dimensions.
```

```
    shape : tuple
```

```
        The mesh resolution: increase to improve accuracy.
```

```
    u_order : int
```

```
        The velocity field approximation order.
```

```
    refine : int
```

```
        The refinement level.
```

```
    ls : 'ls_d' or 'ls_i'
```

```
        The pre-configured linear solver name.
```

```
    u_inlet : float, optional
```

```
        The x-component of the inlet velocity.
```

```
    mode : 'lcbc' or 'penalty'
```

(continues on next page)

(continued from previous page)

```

    The alternative formulations.
term_mode : 'original' or 'einsum'
    The switch to use either the original or new experimental einsum-based
    terms.
backend : str
    The einsum mode backend.
optimize : str
    The einsum mode optimization (backend dependent).
verbosity : 0, 1, 2, 3
    The verbosity level of einsum-based terms.
output_dir : str
    The output directory.
save_lcbc_vecs : bool
    If True, save the no_penetration and edge_direction LCBC vectors.
"""
output('dims: {}, shape: {}, u_order: {}, refine: {}, u_inlet: {}'.format(
    dims, shape, u_order, refine, u_inlet))
output('linear solver: {}'.format(ls))
output('mode: {}, term_mode: {}'.format(mode, term_mode))
if term_mode == 'einsum':
    output('backend: {}, optimize: {}, verbosity: {}'.format(
        backend, optimize, verbosity))

assert_(mode in {'lcbc', 'penalty'})
assert_(term_mode in {'original', 'einsum'})
if u_order > 1:
    assert_(mode == 'penalty', msg='set mode=penalty to use u_order > 1!')
dims = nm.array(dims, dtype=nm.float64)
shape = nm.array(shape, dtype=nm.int32)

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, [0, 0, 0], name='user_block',
                               verbose=False)
        return mesh

    elif mode == 'write':
        pass

filename_mesh = UserMeshIO(mesh_hook)

regions = define_box_regions(3, 0.5 * dims)
regions.update({
    'Omega' : 'all',
    'Edges_v' : (""(r.Near *v r.Bottom) +v
                  (r.Bottom *v r.Far) +v
                  (r.Far *v r.Top) +v
                  (r.Top *v r.Near)""", 'edge'),
    'Gamma1_f' : ('copy r.Top', 'face'),

```

(continues on next page)

(continued from previous page)

```

    'Gamma2_f' : ('r.Near +v r.Bottom +v r.Far', 'face'),
    'Gamma_f' : ('r.Gamma1_f +v r.Gamma2_f', 'face'),
    'Gamma_v' : ('r.Gamma_f -v r.Edges_v', 'face'),
    'Inlet_f' : ('r.Left -v r.Gamma_f', 'face'),
})

fields = {
    'velocity' : ('real', 3, 'Omega', u_order),
    'pressure' : ('real', 1, 'Omega', 1),
}

def get_u_d(ts, coors, region=None, **kwargs):
    """
    Given stator velocity.
    """
    out = nm.zeros_like(coors)
    out[:] = [1.0, 1.0, 0.0]

    return out

functions = {
    'get_u_d' : (get_u_d,),
}

variables = {
    'u' : ('unknown field', 'velocity', 0),
    'v' : ('test field', 'velocity', 'u'),
    'u_d' : ('parameter field', 'velocity',
             {'setter' : 'get_u_d'}),
    'p' : ('unknown field', 'pressure', 1),
    'q' : ('test field', 'pressure', 'p'),
}

materials = {
    'm' : ({
        'nu' : 1e-3,
        'beta' : 1e-2,
        'mu' : 1e-10,
    },),
}

ebcs = {
}
if u_inlet is not None:
    ebcs['inlet'] = ('Inlet_f', {'u.0' : u_inlet, 'u.[1, 2]' : 0.0})

indir = partial(os.path.join, output_dir)

if mode == 'lcbc':
    lcbs = {
        'walls' : ('Gamma_v', {'u.all' : None}, None, 'no_penetration',
                    indir('normals_Gamma.vtk') if save_lcbc_vecs else None),
    }

```

(continues on next page)

(continued from previous page)

```

        'edges' : ('Edges_v', [(-0.5, 1.5)], {'u.all' : None}, None,
                  'edge_direction',
                  indir('edges_Edges.vtk') if save_lcbbc_vecs else None),
    }

    if term_mode == 'original':
        equations = {
            'balance' :
                """dw_div_grad.5.Omega(m.nu, v, u)
                - dw_stokes.5.Omega(v, p)
                + dw_dot.5.Gamma1_f(m.beta, v, u)
                + dw_dot.5.Gamma2_f(m.beta, v, u)
                =
                + dw_dot.5.Gamma1_f(m.beta, v, u_d)""",
            'incompressibility' :
                """dw_laplace.5.Omega(m.mu, q, p)
                + dw_stokes.5.Omega(u, q) = 0""",
        }

    else:
        equations = {
            'balance' :
                """de_div_grad.5.Omega(m.nu, v, u)
                - de_stokes.5.Omega(v, p)
                + de_dot.5.Gamma1_f(m.beta, v, u)
                + de_dot.5.Gamma2_f(m.beta, v, u)
                =
                + de_dot.5.Gamma1_f(m.beta, v, u_d)""",
            'incompressibility' :
                """de_laplace.5.Omega(m.mu, q, p)
                + de_stokes.5.Omega(u, q) = 0""",
        }

    else:
        materials['m'][0]['np_eps'] = 1e3

    if term_mode == 'original':
        equations = {
            'balance' :
                """dw_div_grad.5.Omega(m.nu, v, u)
                - dw_stokes.5.Omega(v, p)
                + dw_dot.5.Gamma1_f(m.beta, v, u)
                + dw_dot.5.Gamma2_f(m.beta, v, u)
                + dw_non_penetration_p.5.Gamma1_f(m.np_eps, v, u)
                + dw_non_penetration_p.5.Gamma2_f(m.np_eps, v, u)
                =
                + dw_dot.5.Gamma1_f(m.beta, v, u_d)""",
            'incompressibility' :
                """dw_laplace.5.Omega(m.mu, q, p)
                + dw_stokes.5.Omega(u, q) = 0""",
        }

```

(continues on next page)

(continued from previous page)

```

else:
    equations = {
        'balance' :
            """de_div_grad.5.Omega(m.nu, v, u)
            - de_stokes.5.Omega(v, p)
            + de_dot.5.Gamma1_f(m.beta, v, u)
            + de_dot.5.Gamma2_f(m.beta, v, u)
            + de_non_penetration_p.5.Gamma1_f(m.np_eps, v, u)
            + de_non_penetration_p.5.Gamma2_f(m.np_eps, v, u)
            =
            + de_dot.5.Gamma1_f(m.beta, v, u_d)""",
        'incompressibility' :
            """de_laplace.5.Omega(m.mu, q, p)
            + de_stokes.5.Omega(u, q) = 0""",
    }

solvers = {
    'ls_d' : ('ls.auto_direct', {}),
    'ls_i' : ('ls.petsc', {
        'method' : 'bcgsl', # ksp_type
        'precond' : 'bjacobi', # pc_type
        'sub_precond' : 'ilu', # sub_pc_type
        'eps_a' : 0.0, # abstol
        'eps_r' : 1e-12, # rtol
        'eps_d' : 1e10, # Divergence tolerance.
        'i_max' : 200, # maxits
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}

options = {
    'nls' : 'newton',
    'ls' : ls,
    'eterm': {
        'verbosity' : verbosity,
        'backend_args' : {
            'backend' : backend,
            'optimize' : optimize,
            'layout' : None,
        },
    },
    'refinement_level' : refine,
    'output_dir' : output_dir,
}

return locals()

```

navier_stokes/utils.py

Description

missing description!

[source code](#)

```
##
# Functions.
from __future__ import absolute_import
import numpy as nm

from sfepy.linalg import get_coors_in_tube

# last revision: 01.08.2007
def cinc_cylinder(coors, mode):
    axis = nm.array([1, 0, 0], nm.float64)
    if mode == 0: # In
        centre = nm.array([-0.00001, 0.0, 0.0], nm.float64)
        radius = 0.019
        length = 0.00002
    elif mode == 1: # Out
        centre = nm.array([0.09999, 0.0, 0.0], nm.float64)
        radius = 0.019
        length = 0.00002
    else:
        centre = nm.array([0.05, 0.0, 0.0], nm.float64)
        radius = 0.012
        length = 0.04

    return get_coors_in_tube(coors, centre, axis, -1.0, radius, length)

def cinc_elbow2(coors, mode):
    if mode == 0: # In
        centre = nm.array([0.0, -0.00001, 0.0], nm.float64)
    else: # Out
        centre = nm.array([0.2, -0.00001, 0.0], nm.float64)

    axis = nm.array([0, 1, 0], nm.float64)
    radius = 0.029
    length = 0.00002

    return get_coors_in_tube(coors, centre, axis, -1.0, radius, length)
```

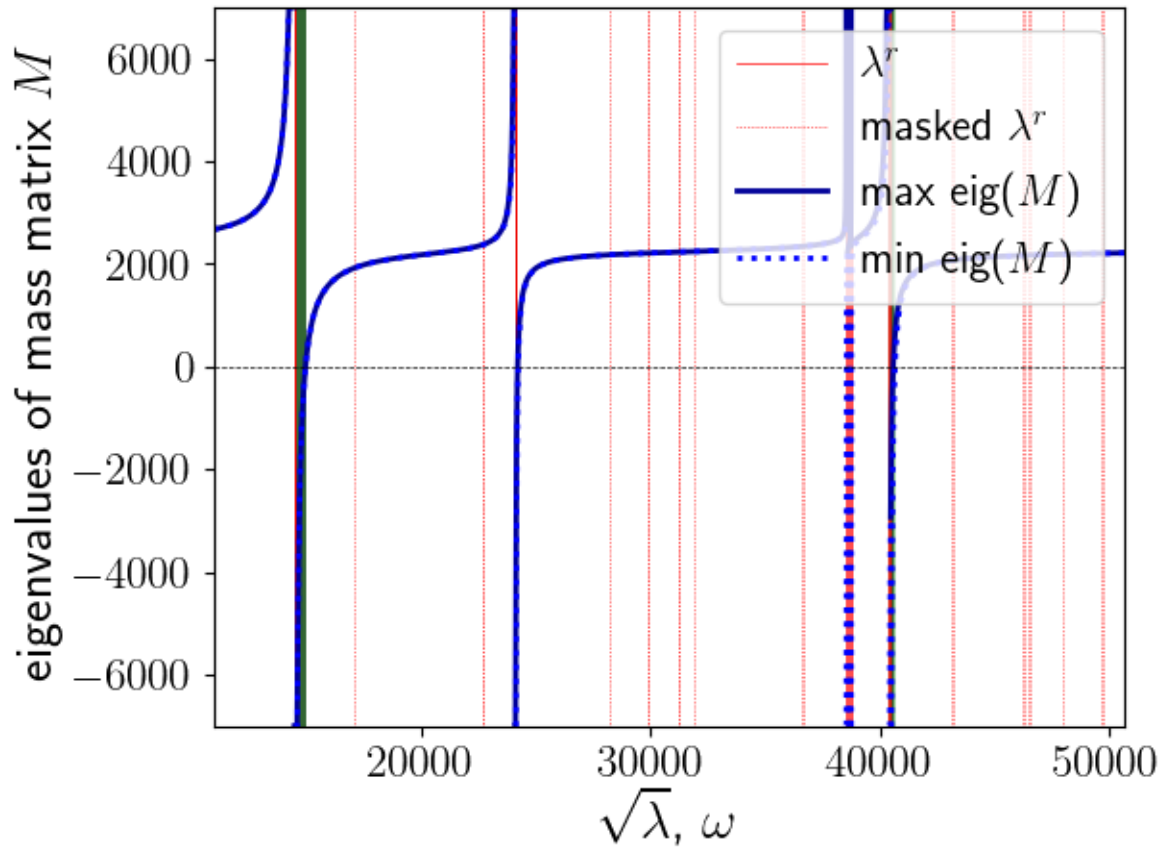

phononic

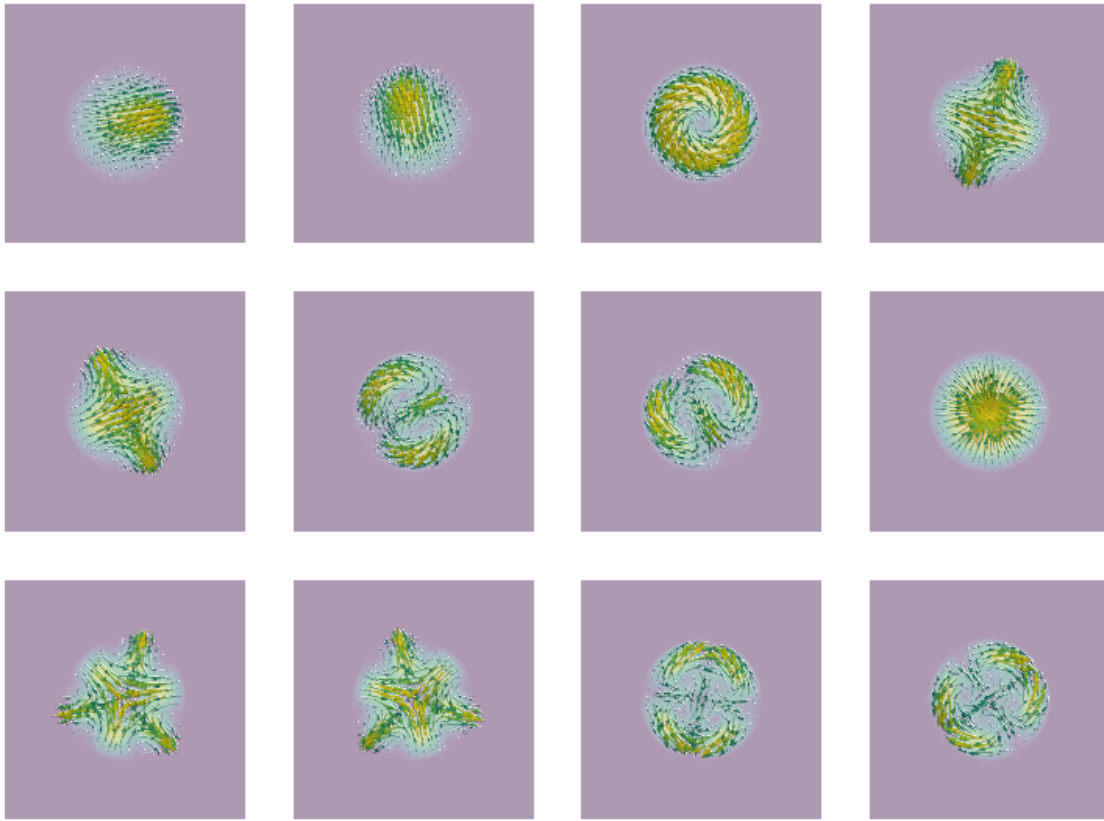
phononic/band_gaps.py

Description

Acoustic band gaps in a strongly heterogeneous elastic body, detected using homogenization techniques.

A reference periodic cell contains two domains: the stiff matrix Y_m and the soft (but heavy) inclusion Y_c .





source code

```
"""
Acoustic band gaps in a strongly heterogeneous elastic body, detected using
homogenization techniques.

A reference periodic cell contains two domains: the stiff matrix :math:`Y_m`
and the soft (but heavy) inclusion :math:`Y_c`.
"""
from __future__ import absolute_import
from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.base.iutils import InDir
from sfepy.homogenization.coefficients import Coefficients

from sfepy.examples.phononic.band_gaps_conf import (BandGapsConf, get_pars,
                                                    clip, clip_sqrt)

clip, clip_sqrt # Make pyflakes happy...

incwd = InDir(__file__)

filename = data_dir + '/meshes/2d/special/circle_in_square.mesh'
```

(continues on next page)

(continued from previous page)

```

output_dir = incwd('output/band_gaps')

# aluminium, SI units
D_m = get_pars(2, 5.898e10, 2.681e10)
density_m = 2799.0

# epoxy, SI units
D_c = get_pars(2, 1.798e9, 1.48e9)
density_c = 1142.0

mat_pars = Coefficients(D_m=D_m, density_m=density_m,
                        D_c=D_c, density_c=density_c)

region_selects = Struct(matrix='cells of group 1',
                        inclusion='cells of group 2')

corrs_save_names = {'evp' : 'evp', 'corrs_rs' : 'corrs_rs'}

options = {
    'plot_transform_angle' : None,
    'plot_transform_wave' : ('clip_sqrt', (0, 7000)),
    'plot_transform' : ('clip', (-7000, 7000)),

    'fig_name' : 'band_gaps',
    'fig_name_angle' : 'band_gaps_angle',
    'fig_name_wave' : 'band_gaps_wave',
    'fig_suffix' : '.pdf',

    'coefs_filename' : 'coefs.txt',

    'incident_wave_dir' : [1.0, 1.0],

    'plot_options' : {
        'show' : True,
        'legend' : True,
    },
    'plot_labels' : {
        'band_gaps' : {
            'resonance' : r'$\lambda^r$',
            'masked' : r'masked $\lambda^r$',
            'eig_min' : r'min eig($M$)',
            'eig_max' : r'max eig($M$)',
            'x_axis' : r'$\sqrt{\lambda}$, $\omega$',
            'y_axis' : r'eigenvalues of mass matrix $M$',
        },
    },
    'plot_rsc' : {
        'params' : {'axes.labelsize': 'x-large',
                    'font.size': 14,
                    'legend.fontsize': 'large',
                    'legend.loc': 'upper right',

```

(continues on next page)

(continued from previous page)

```
        'xtick.labelsize': 'large',
        'ytick.labelsize': 'large',
        'text.usetex': True},
    },
    'multiprocessing' : False,
    'float_format' : '%.16e',
}

evp_options = {
    'eigensolver' : 'eig.sgscipy',
    'save_eig_vectors' : (12, 0),
    'scale_epsilon' : 1.0,
    'elasticity_contrast' : 1.0,
}

eigenmomenta_options = {
    # eigenmomentum threshold,
    'threshold' : 1e-2,
    # eigenmomentum threshold is relative w.r.t. largest one,
    'threshold_is_relative' : True,
}

band_gaps_options = {
    'eig_range' : (0, 30), # -> freq_range
                        # = sqrt(eigs[slice(*eig_range)][[0, -1]])
    # 'fixed_freq_range' : (0.1, 3e7),
    'freq_margins' : (10, 10), # % of freq_range
    'freq_eps' : 1e-7, # frequency
    'zero_eps' : 1e-12, # zero finding
    'freq_step' : 0.0001, # % of freq_range

    'log_save_name' : 'band_gaps.log',
    'raw_log_save_name' : 'raw_eigensolution.npz',
}

conf = BandGapsConf(filename, 1, region_selects, mat_pars, options,
                    evp_options, eigenmomenta_options, band_gaps_options,
                    corrs_save_names=corrs_save_names, incwd=incwd,
                    output_dir=output_dir)

define = lambda: conf.conf.to_dict()
```

phononic/band_gaps_conf.py**Description**

Configuration classes for acoustic band gaps in a strongly heterogeneous elastic body.

source code

```

"""
Configuration classes for acoustic band gaps in a strongly heterogeneous
elastic body.
"""
from __future__ import absolute_import
import numpy as nm

from sfepy.base.base import get_default, import_file, Struct
from sfepy.base.conf import ProblemConf
from sfepy.discrete.fem import MeshIO
import sfepy.discrete.fem.periodic as per
from sfepy.mechanics.matcoefs import stiffness_from_lame, TransformToPlane
from sfepy.homogenization.utils import define_box_regions, get_lattice_volume
import sfepy.homogenization.coefs_base as cb
import sfepy.homogenization.coefs_phononic as cp

per.set_accuracy(1e-8)

def get_pars(dim, lam, mu):
    c = stiffness_from_lame(3, lam, mu)
    if dim == 2:
        tr = TransformToPlane()
        try:
            c = tr.tensor_plane_stress(c3=c)
        except:
            sym = (dim + 1) * dim // 2
            c = nm.zeros((sym, sym), dtype=nm.float64)

    return c

def set_coef_d(variables, ir, ic, mode, pis, corrs_rs):
    mode2var = {'row' : 'u1_m', 'col' : 'u2_m'}

    val = pis.states[ir, ic]['u_m'] + corrs_rs.states[ir, ic]['u_m']

    variables[mode2var[mode]].set_data(val)

class BandGapsConf(Struct):
    """
    Configuration class for acoustic band gaps in a strongly heterogeneous
    elastic body.
    """

    def __init__(self, filename, approx, region_selects, mat_pars, options,
                 evp_options, eigenmomenta_options, band_gaps_options,
                 coefs_save_name='coefs',

```

(continues on next page)

(continued from previous page)

```

        corrs_save_names=None,
        incwd=None,
        output_dir=None, **kwargs):
Struct.__init__(self, approx=approx, region_selects=region_selects,
                mat_pars=mat_pars, options=options,
                evp_options=evp_options,
                eigenmomenta_options=eigenmomenta_options,
                band_gaps_options=band_gaps_options,
                **kwargs)
self.incwd = get_default(incwd, lambda x: x)

self.conf = Struct()
self.conf.filename_mesh = self.incwd(filename)

output_dir = get_default(output_dir, self.incwd('output'))

default = {'evp' : 'evp', 'corrs_rs' : 'corrs_rs'}
self.corrs_save_names = get_default(corrs_save_names,
                                    default)

io = MeshIO.any_from_filename(self.conf.filename_mesh)
self.bbox, self.dim = io.read_bounding_box(ret_dim=True)
rpc_axes = nm.eye(self.dim, dtype=nm.float64) \
            * (self.bbox[1] - self.bbox[0])

self.conf.options = options
self.conf.options.update({
    'output_dir' : output_dir,

    'volume' : {
        'value' : get_lattice_volume(rpc_axes),
    },

    'coefs' : 'coefs',
    'requirements' : 'requirements',

    'coefs_filename' : coefs_save_name,
})

self.conf.mat_pars = mat_pars

self.conf.solvers = self.define_solvers()
self.conf.regions = self.define_regions()
self.conf.materials = self.define_materials()
self.conf.fields = self.define_fields()
self.conf.variables = self.define_variables()
(self.conf.ebcs, self.conf.epbcs,
 self.conf.lcbcs, self.all_periodic) = self.define_bcs()
self.conf.functions = self.define_functions()
self.conf.integrals = self.define_integrals()

self.equations, self.expr_coefs = self.define_equations()

```

(continues on next page)

(continued from previous page)

```

self.conf.coefs = self.define_coefs()
self.conf.requirements = self.define_requirements()

def __call__(self):
    return ProblemConf.from_dict(self.conf.__dict__,
                                import_file(__file__))

def define_solvers(self):
    solvers = {
        'ls_d' : ('ls.auto_direct', {'use_presolve' : True}),
        'ls_i' : ('ls.scipy_iterative', {
            'method' : 'cg',
            'i_max'   : 1000,
            'eps_a'   : 1e-12,
        }),
        'newton' : ('nls.newton', {
            'i_max' : 1,
            'eps_a' : 1e-4,
        }),
    }

    return solvers

def define_regions(self):
    regions = {
        'Y' : 'all',
        'Y_m' : self.region_selects.matrix,
        'Y_c' : self.region_selects.inclusion,
        'Gamma_mc': ('r.Y_m *v r.Y_c', 'facet'),
    }

    regions.update(define_box_regions(self.dim,
                                     self.bbox[0], self.bbox[1], 1e-5))

    return regions

def define_materials(self):
    materials = {
        'm' : ({
            'D_m' : self.mat_pars.D_m,
            'density_m' : self.mat_pars.density_m,
            'D_c' : self.mat_pars.D_c,
            'density_c' : self.mat_pars.density_c,
        }, None, None, {'special_constant' : True}),
    }
    return materials

def define_fields(self):
    fields = {
        'vector_Y_m' : ('real', self.dim, 'Y_m', self.approx),
        'vector_Y_c' : ('real', self.dim, 'Y_c', self.approx),
    }

```

(continues on next page)

(continued from previous page)

```

        'scalar_Y' : ('real', 1, 'Y', 1),
    }
    return fields

def define_variables(self):
    variables = {
        'u_m' : ('unknown field', 'vector_Y_m'),
        'v_m' : ('test field', 'vector_Y_m', 'u_m'),
        'Pi'   : ('parameter field', 'vector_Y_m', '(set-to-None)'),
        'u1_m'  : ('parameter field', 'vector_Y_m', '(set-to-None)'),
        'u2_m'  : ('parameter field', 'vector_Y_m', '(set-to-None)'),

        'u_c' : ('unknown field', 'vector_Y_c'),
        'v_c' : ('test field', 'vector_Y_c', 'u_c'),

        'aux'  : ('parameter field', 'scalar_Y', '(set-to-None)'),
    }
    return variables

def define_bcs(self):
    ebcs = {
        'fixed_corners' : ('Corners', {'u_m.all' : 0.0}),
        'fixed_gamma_mc' : ('Gamma_mc', {'u_c.all' : 0.0}),
    }

    epbcs = {}
    all_periodic = []
    for vn in ['u_m']:
        val = {'%s.all' % vn : '%s.all' % vn}

        epbcs.update({
            'periodic_%s_x' % vn : ([ 'Left', 'Right'], val,
                                    'match_y_line'),
            'periodic_%s_y' % vn : ([ 'Top', 'Bottom'], val,
                                    'match_x_line'),
        })
        all_periodic.extend(['periodic_%s_x' % vn, 'periodic_%s_y' % vn])

    lcbcs = {}

    return ebcs, epbcs, lcbcs, all_periodic

def define_functions(self):
    functions = {
        'match_x_line' : (per.match_x_line,),
        'match_y_line' : (per.match_y_line,),
    }

    return functions

def define_integrals(self):
    integrals = {

```

(continues on next page)

(continued from previous page)

```

        'i' : 2,
    }

    return integrals

def define_equations(self):
    equations = {}
    equations['corrs_rs'] = {
        'balance_of_forces' :
        """dw_lin_elastic.i.Y_m( m.D_m, v_m, u_m )
        = - dw_lin_elastic.i.Y_m( m.D_m, v_m, Pi )""",
    }
    equations['evp'] = {
        'lhs' : """dw_lin_elastic.i.Y_c( m.D_c, v_c, u_c )""",
        'rhs' : """dw_dot.i.Y_c( m.density_c, v_c, u_c )""",
    }

    expr_coefs = {
        'D' : """dw_lin_elastic.i.Y_m( m.D_m, u1_m, u2_m )""",
        'VF' : """ev_volume.i.%s(aux)""",
        'ema' : """ev_integrate.i.Y_c( m.density_c, u_c )""",
    }

    return equations, expr_coefs

def define_coefs(self):
    from copy import copy

    ema_options = copy(self.eigenmomenta_options)
    ema_options.update({'var_name' : 'u_c'})

    dispersion_options = copy(self.band_gaps_options)
    dispersion_options.update({'log_save_name' : 'dispersion.log'})

    coefs = {
        # Basic.
        'VF' : {
            'regions' : ['Y_m', 'Y_c'],
            'expression' : self.expr_coefs['VF'],
            'class' : cb.VolumeFractions,
        },
        'dv_info' : {
            'requires' : ['c.VF'],
            'region_to_material' : {'Y_m' : ('m', 'density_m'),
                                   'Y_c' : ('m', 'density_c')},
            'class' : cp.DensityVolumeInfo,
        },
        'eigenmomenta' : {
            'requires' : ['evp', 'c.dv_info'],
            'expression' : self.expr_coefs['ema'],
            'options' : ema_options,
        },
    }

```

(continues on next page)

(continued from previous page)

```

        'class' : cp.Eigenmomenta,
    },
    'M' : {
        'requires' : ['evp', 'c.dv_info', 'c.eigenmomenta'],
        'class' : cp.AcousticMassTensor,
    },
    'band_gaps' : {
        'requires' : ['evp', 'c.eigenmomenta', 'c.M'],
        'options' : self.band_gaps_options,
        'class' : cp.BandGaps,
    },
    # Dispersion.
    'D' : {
        'requires' : ['pis', 'corrs_rs'],
        'expression' : self.expr_coefs['D'],
        'set_variables' : set_coef_d,
        'class' : cb.CoeffSymSym,
    },
    'Gamma' : {
        'requires' : ['c.D'],
        'options' : {
            'mode' : 'simple',
            'incident_wave_dir' : None,
        },
        'class' : cp.ChristoffelAcousticTensor,
    },
    'dispersion' : {
        'requires' : ['evp', 'c.eigenmomenta', 'c.M', 'c.Gamma'],
        'options' : dispersion_options,
        'class' : cp.BandGaps,
    },
    'polarization_angles' : {
        'requires' : ['c.dispersion'],
        'options' : {
            'incident_wave_dir' : None,
        },
        'class' : cp.PolarizationAngles,
    },
    # Phase velocity.
    'phase_velocity' : {
        'requires' : ['c.dv_info', 'c.Gamma'],
        'options' : {
            'eigensolver' : 'eig.sgscipy',
        },
        'class' : cp.PhaseVelocity,
    },
    'filenames' : {},
}

return coefs

```

(continues on next page)

(continued from previous page)

```

def define_requirements(self):
    requirements = {
        # Basic.
        'evp' : {
            'ebcs' : ['fixed_gamma_mc'],
            'epbcs' : None,
            'equations' : self.equations['evp'],
            'save_name' : self.corrs_save_names['evp'],
            'options' : self.evp_options,
            'class' : cp.SimpleEVP,
        },

        # Dispersion.
        'pis' : {
            'variables' : ['u_m'],
            'class' : cb.ShapeDimDim,
        },
        'corrs_rs' : {
            'requires' : ['pis'],
            'ebcs' : ['fixed_corners'],
            'epbcs' : self.all_periodic,
            'equations' : self.equations['corrs_rs'],
            'set_variables' : [('Pi', 'pis', 'u_m')],
            'save_name' : self.corrs_save_names['corrs_rs'],
            'is_linear' : True,
            'class' : cb.CorrDimDim,
            'is_linear' : True,
        },
    }
    return requirements

class BandGapsRigidConf(BandGapsConf):
    """
    Configuration class for acoustic band gaps in a strongly heterogeneous
    elastic body with rigid inclusions.
    """

    def define_regions(self):
        regions = BandGapsConf.define_regions(self)
        regions['Y_cr'] = regions['Y_c']
        regions.update({
            'Y_r' : 'vertices by select_yr',
            'Y_c' : 'r.Y_cr -c r.Y_r',
        })
        return regions

    def define_materials(self):
        materials = BandGapsConf.define_materials(self)
        materials['m'][0].update({
            'D_r' : self.mat_pars.D_r,
            'density_r' : self.mat_pars.density_r,

```

(continues on next page)

(continued from previous page)

```

    })
    return materials

def define_fields(self):
    fields = {
        'vector_Y_cr' : ('real', self.dim, 'Y_cr', self.approx),

        'scalar_Y' : ('real', 1, 'Y', 1),
    }
    return fields

def define_variables(self):
    variables = {
        'u' : ('unknown field', 'vector_Y_cr'),
        'v' : ('test field', 'vector_Y_cr', 'u'),

        'aux' : ('parameter field', 'scalar_Y', '(set-to-None)'),
    }
    return variables

def define_bcs(self):
    ebcs = {
        'fixed_gamma_mc' : ('Gamma_mc', {'u.all' : 0.0}),
    }
    lcbcs = {
        'rigid' : ('Y_r', {'u.all' : None}, None, 'rigid'),
    }

    return ebcs, {}, lcbcs, []

def define_functions(self):
    functions = BandGapsConf.define_functions(self)
    functions.update({
        'select_yr' : (self.select_yr,),
    })

    return functions

def define_equations(self):
    equations = {}

    # dw_lin_elastic.i.Y_r( m.D_r, v, u ) should have no effect!
    equations['evp'] = {
        'lhs' : """"dw_lin_elastic.i.Y_c( m.D_c, v, u )
                    + dw_lin_elastic.i.Y_r( m.D_r, v, u )""",
        'rhs' : """"dw_dot.i.Y_c( m.density_c, v, u )
                    + dw_dot.i.Y_r( m.density_r, v, u )""",
    }

    expr_coefs = {
        'VF' : """"ev_volume.i.%s(aux)""",
        'ema' : """"ev_integrate.i.Y_c( m.density_c, u )

```

(continues on next page)

(continued from previous page)

```

        + ev_integrate.i.Y_r( m.density_r, u )""",
    }

    return equations, expr_coefs

def define_coefs(self):
    from copy import copy

    ema_options = copy(self.eigenmomenta_options)
    ema_options.update({'var_name' : 'u'})

    coefs = {
        # Basic.
        'VF' : {
            'regions' : ['Y_m', 'Y_cr', 'Y_c', 'Y_r'],
            'expression' : self.expr_coefs['VF'],
            'class' : cb.VolumeFractions,
        },
        'dv_info' : {
            'requires' : ['c.VF'],
            'region_to_material' : {'Y_m' : ('m', 'density_m'),
                                   'Y_c' : ('m', 'density_c'),
                                   'Y_r' : ('m', 'density_r')},
            'class' : cp.DensityVolumeInfo,
        },
        'eigenmomenta' : {
            'requires' : ['evp', 'c.dv_info'],
            'expression' : self.expr_coefs['ema'],
            'options' : ema_options,
            'class' : cp.Eigenmomenta,
        },
        'M' : {
            'requires' : ['evp', 'c.dv_info', 'c.eigenmomenta'],
            'class' : cp.AcousticMassTensor,
        },
        'band_gaps' : {
            'requires' : ['evp', 'c.eigenmomenta', 'c.M'],
            'options' : self.band_gaps_options,
            'class' : cp.BandGaps,
        },
        'filenames' : {},
    }

    return coefs

def define_requirements(self):
    requirements = {
        # Basic.
        'evp' : {
            'ebcs' : ['fixed_gamma_mc'],

```

(continues on next page)

(continued from previous page)

```
        'epbcs' : None,
        'lcbcs' : ['rigid'],
        'equations' : self.equations['evp'],
        'save_name' : self.corrs_save_names['evp'],
        'options' : self.evp_options,
        'class' : cp.SimpleEVP,
    },
}
return requirements

def clip(data, plot_range):
    return nm.clip(data, *plot_range)

def clip_sqrt(data, plot_range):
    return nm.clip(nm.sqrt(data), *plot_range)

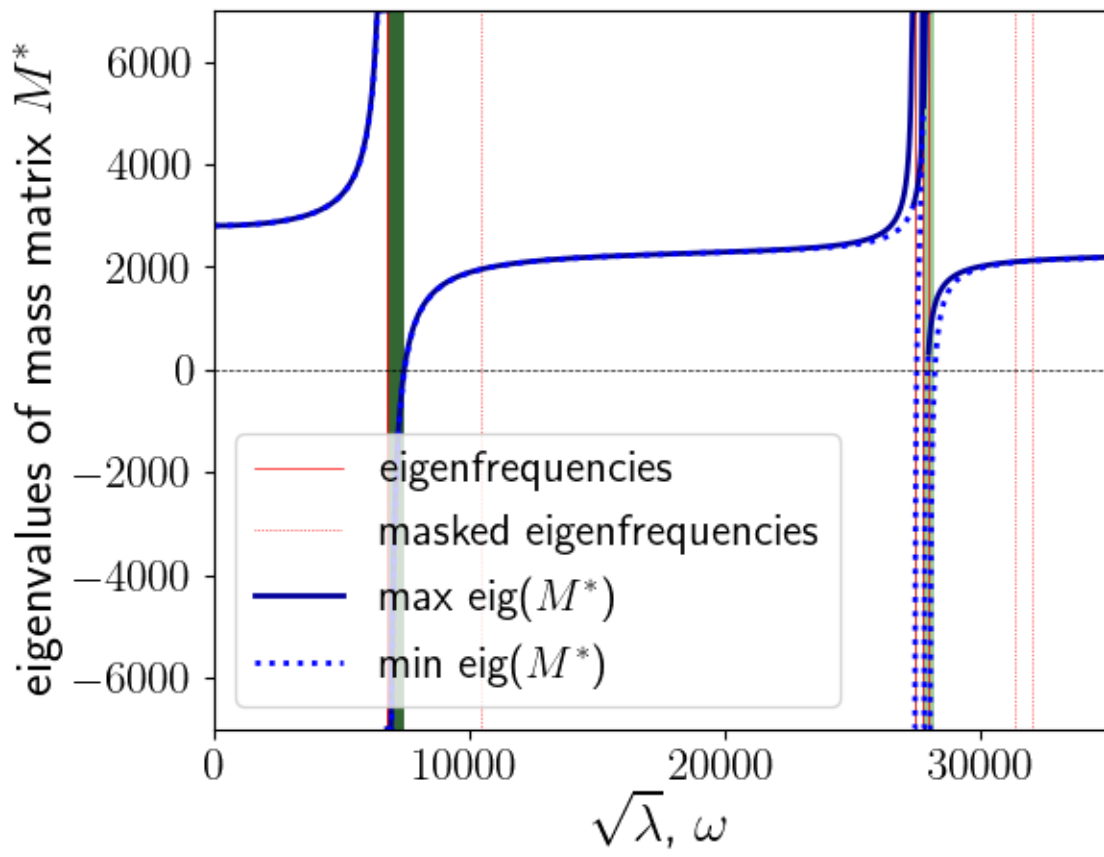
def normalize(data, plot_range):
    aux = nm.arctan(data)
    return clip(aux, plot_range)
```

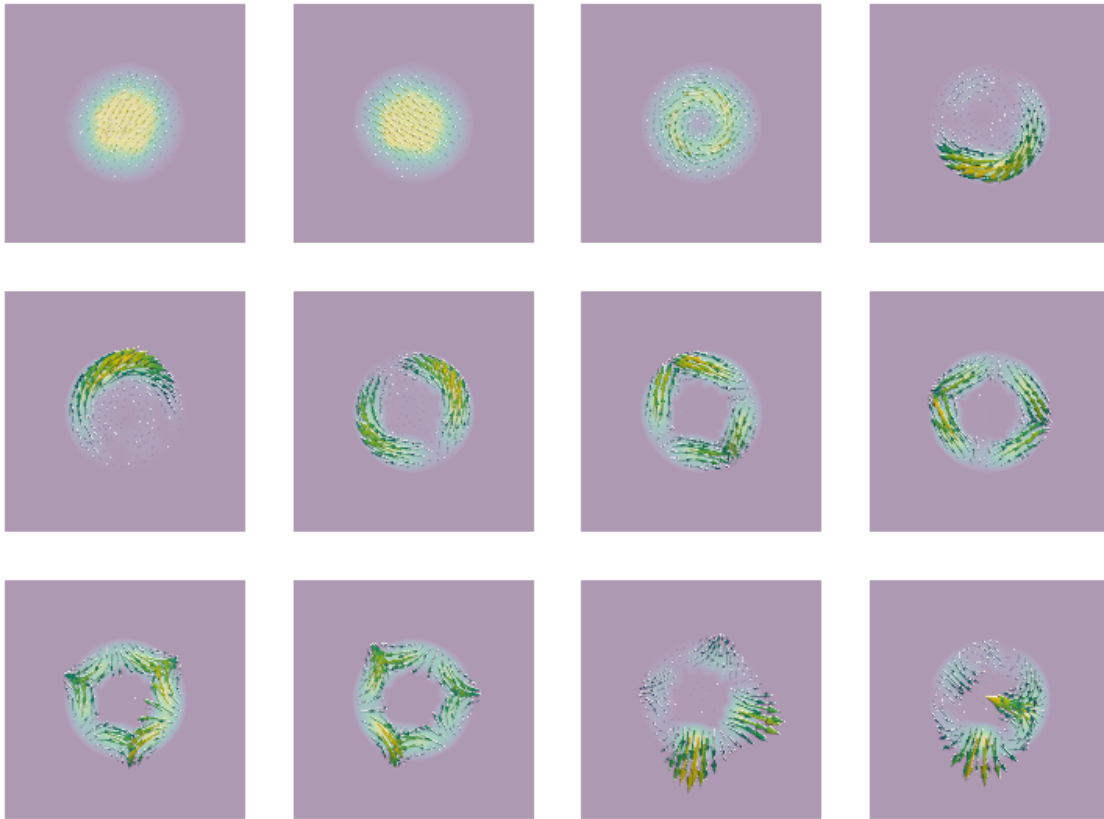
phononic/band_gaps_rigid.py

Description

Acoustic band gaps in a strongly heterogeneous elastic body with a rigid inclusion, detected using homogenization techniques.

A reference periodic cell contains three domains: the stiff matrix Y_m and the soft inclusion Y_c enclosing the rigid heavy sub-inclusion Y_r .





source code

```

"""
Acoustic band gaps in a strongly heterogeneous elastic body with a rigid
inclusion, detected using homogenization techniques.

A reference periodic cell contains three domains: the stiff matrix  $Y_m$ 
and the soft inclusion  $Y_c$  enclosing the rigid heavy sub-inclusion
 $Y_r$ .
"""
from __future__ import absolute_import
import numpy as nm

from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.base.ioutils import InDir
from sfepy.discrete.fem import extend_cell_data
from sfepy.linalg import norm_l2_along_axis
from sfepy.homogenization.coefficients import Coefficients

from sfepy.examples.phononic.band_gaps_conf import (BandGapsRigidConf,
                                                    get_pars, clip)

```

(continues on next page)

(continued from previous page)

```

clip # Make pyflakes happy...

incwd = InDir(__file__)

dim = 2

if dim == 3:
    filename = data_dir + '/meshes/3d/special/cube_sphere.mesh'
else:
    filename = data_dir + '/meshes/2d/special/circle_in_square.mesh'

output_dir = incwd('output/band_gaps_rigid')

# Rigid inclusion diameter.
yr_diameter = 0.125

# aluminium, SI units
D_m = get_pars(2, 5.898e10, 2.681e10)
density_m = 2799.0

# epoxy, SI units
D_c = get_pars(2, 1.798e9, 1.48e9)
density_c = 1142.0

# lead, SI units, does not matter
D_r = get_pars(dim, 4.074e10, 5.556e9)
density_r = 11340.0

mat_pars = Coefficients(D_m=D_m, density_m=density_m,
                        D_c=D_c, density_c=density_c,
                        D_r=D_r, density_r=density_r)

region_selects = Struct(matrix='cells of group 1',
                        inclusion='cells of group 2')

corrs_save_names = {'evp' : 'evp'}

evp_options = {
    'eigensolver' : 'eig.sgscipy',
    'save_eig_vectors' : (12, 0),
    'scale_epsilon' : 1.0,
    'elasticity_contrast' : 1.0,
}

eigenmomenta_options = {
    # eigenmomentum threshold,
    'threshold' : 1e-1,
    # eigenmomentum threshold is relative w.r.t. largest one,
    'threshold_is_relative' : True,
}

```

(continues on next page)

(continued from previous page)

```

band_gaps_options = {
    'fixed_freq_range' : (0., 35000.), # overrides eig_range!

    'freq_eps' : 1e-7, # frequency
    'zero_eps' : 1e-12, # zero finding
    'freq_step' : 0.01, # % of freq_range

    'log_save_name' : 'band_gaps.log',
    'raw_log_save_name' : 'raw_eigensolution.npz',
}

options = {
    'post_process_hook' : 'post_process',

    'plot_transform' : ('clip', (-7000, 7000)),

    'fig_name' : 'band_gaps',
    'fig_suffix' : '.pdf',

    'coefs_filename' : 'coefs.txt',

    'plot_options' : {
        'show' : True, # Show figure.
        'legend' : True, # Show legend.
    },
    'float_format' : '%.16e',
}

def select_yr_circ(coors, diameter=None):
    r = norm_l2_along_axis(coors)
    out = nm.where(r < diameter)[0]

    if out.shape[0] <= 3:
        raise ValueError('too few nodes selected! (%d)' % out.shape[0])

    return out

def _select_yr_circ(coors, domain=None, diameter=None):
    return select_yr_circ(coors, diameter=yr_diameter)

def post_process(out, problem, mtx_phi):
    for key in list(out.keys()):
        ii = int(key[1:])
        vec = mtx_phi[:,ii].copy()
        problem.set_default_state(vec)

        strain = problem.evaluate('ev_cauchy_strain.i.Y_c(u)',
                                verbose=False, mode='el_avg')
        strain = extend_cell_data(strain, problem.domain, 'Y_c')
        out['strain%03d' % ii] = Struct(name='output_data',
                                mode='cell', data=strain,

```

(continues on next page)

(continued from previous page)

```

                                dofs=None)

    return out

conf = BandGapsRigidConf(filename, 1, region_selects, mat_pars, options,
                        evp_options, eigenmomenta_options, band_gaps_options,
                        corrs_save_names=corrs_save_names, incwd=incwd,
                        output_dir=output_dir, select_yr=_select_yr_circ)

define = lambda: conf.conf.to_dict()

```

quantum

quantum/boron.py

Description

Boron atom with 1 electron.

See [quantum/quantum_common.py](#).



[source code](#)

```
"""
Boron atom with 1 electron.

See :ref:`quantum-quantum_common`.
"""
from __future__ import absolute_import
from sfepy.linalg import norm_l2_along_axis

from sfepy.examples.quantum.quantum_common import common

def get_exact(n_eigs, box_size, dim):
    Z = 5
    if dim == 2:
        eigs = [-float(Z)**2/2/(n-0.5)**2/4
                for n in [1] + [2]*3 + [3]*5 + [4]*8 + [5]*15]

    elif dim == 3:
        eigs = [-float(Z)**2/2/n**2 for n in [1] + [2]*2**2 + [3]*3**2]

    return eigs

def fun_v(ts, coor, mode=None, **kwargs):
    if not mode == 'qp': return

    out = {}
    C = 0.5
    r = norm_l2_along_axis(coor, axis=1)
    V = - C * 5.0 / r

    V.shape = (V.shape[0], 1, 1)
    out['V'] = V
    return out

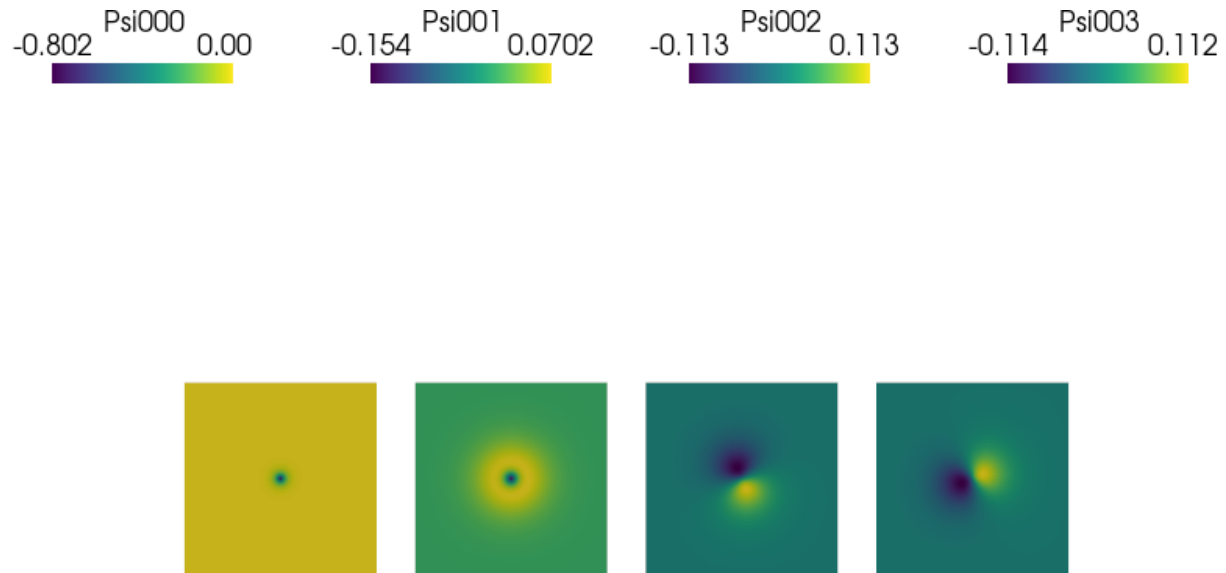
def define(n_eigs=10, tau=-15):
    l = common(fun_v, get_exact=get_exact, n_eigs=n_eigs, tau=tau)
    return l
```

quantum/hydrogen.py

Description

Hydrogen atom.

See *quantum/quantum_common.py*.



source code

```
"""
Hydrogen atom.

See :ref:`quantum-quantum_common`.
"""
from __future__ import absolute_import
from sfepy.linalg import norm_l2_along_axis

from sfepy.examples.quantum.quantum_common import common

def get_exact(n_eigs, box_size, dim):
    Z = 1
    if dim == 2:
        eigs = [-float(Z)**2/2/(n-0.5)**2/4
                for n in [1] + [2]*3 + [3]*5 + [4]*8 + [5]*15]

    elif dim == 3:
        eigs = [-float(Z)**2/2/n**2 for n in [1] + [2]*2**2 + [3]*3**2]

    return eigs
```

(continues on next page)

(continued from previous page)

```
def fun_v(ts, coor, mode=None, **kwargs):
    if not mode == 'qp': return

    out = {}
    C = 0.5
    r = norm_l2_along_axis(coor, axis=1)
    V = - C * 1.0 / r

    V.shape = (V.shape[0], 1, 1)
    out['V'] = V
    return out

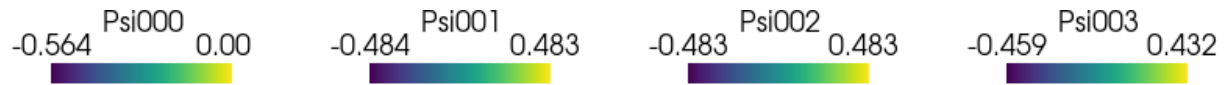
def define(n_eigs=5, tau=-1.0):
    l = common(fun_v, get_exact=get_exact, n_eigs=n_eigs, tau=tau)
    return l
```

quantum/oscillator.py

Description

Quantum oscillator.

See *quantum/quantum_common.py*.



source code

```
"""
Quantum oscillator.

See :ref:`quantum-quantum_common`.
"""
from __future__ import absolute_import
from sfepy.linalg import norm_l2_along_axis

from sfepy.examples.quantum.quantum_common import common

def get_exact(n_eigs, box_size, dim):
    if dim == 2:
        eigs = [1] + [2]*2 + [3]*3 + [4]*4 + [5]*5 + [6]*6

    elif dim == 3:
        eigs = [float(1)/2 + x for x in [1] + [2]*3 + [3]*6 + [4]*10]

    return eigs

def fun_v(ts, coor, mode=None, **kwargs):
    if not mode == 'qp': return
```

(continues on next page)

(continued from previous page)

```
out = {}
C = 0.5
val = C * norm_l2_along_axis(coor, axis=1, squared=True)

val.shape = (val.shape[0], 1, 1)
out['V'] = val
return out

def define(n_eigs=20, tau=0.0):
    l = common(fun_v, get_exact=get_exact, n_eigs=n_eigs, tau=tau)
    return l
```

quantum/quantum_common.py

Description

Common code for basic electronic structure examples.

It covers only simple single electron problems, e.g. well, oscillator, hydrogen atom and boron atom with 1 electron - see the corresponding files in this directory, where potentials (`fun_v()`) as well as exact solutions (`get_exact()`) for those problems are defined.

Notes

The same code should work also with a 3D (box) mesh, but a very fine mesh would be required. Also in the 2D case, finer mesh and/or higher approximation order means higher accuracy.

Try changing C, F and L parameters in `meshes/quantum/square.geo` and regenerate the mesh using gmsh:

```
gmsh -2 -format mesh meshes/quantum/square.geo -o meshes/quantum/square.mesh
./script/convert_mesh.py -2 meshes/quantum/square.mesh meshes/quantum/square.mesh
```

The `script/convert_mesh.py` call makes the mesh planar, as gmsh saves 2D medit meshes including the zero z coordinates.

Also try changing approximation order ('`approx_order`') of the field below.

Usage Examples

The following examples are available and can be run using:

```
sfepy-run sfepy/examples/quantum/boron.py
sfepy-run sfepy/examples/quantum/hydrogen.py
sfepy-run sfepy/examples/quantum/oscillator.py
sfepy-run sfepy/examples/quantum/well.py
```

source code


```

"""
Common code for basic electronic structure examples.

It covers only simple single electron problems, e.g. well, oscillator, hydrogen
atom and boron atom with 1 electron - see the corresponding files in this
directory, where potentials (:func:`fun_v()`) as well as exact solutions
(:func:`get_exact()`) for those problems are defined.

Notes
-----

The same code should work also with a 3D (box) mesh, but a very fine mesh would
be required. Also in the 2D case, finer mesh and/or higher approximation order
means higher accuracy.

Try changing C, F and L parameters in ``meshes/quantum/square.geo`` and
regenerate the mesh using gmsh::

    gmsh -2 -format mesh meshes/quantum/square.geo -o meshes/quantum/square.mesh
    ./script/convert_mesh.py -2 meshes/quantum/square.mesh meshes/quantum/square.mesh

The ``script/convert_mesh.py`` call makes the mesh planar, as gmsh saves 2D
medit meshes including the zero z coordinates.

Also try changing approximation order ('approx_order') of the field below.

Usage Examples
-----

The following examples are available and can be run using::

    sfepy-run sfepy/examples/quantum/boron.py
    sfepy-run sfepy/examples/quantum/hydrogen.py
    sfepy-run sfepy/examples/quantum/oscillator.py
    sfepy-run sfepy/examples/quantum/well.py
"""
from __future__ import absolute_import
from sfepy.base.base import output
from sfepy import data_dir

def common(fun_v, get_exact=None, n_eigs=5, tau=0.0):

    def report_eigs(pb, evp):
        from numpy import NaN

        bounding_box = pb.domain.mesh.get_bounding_box()
        box_size = bounding_box[1][0] - bounding_box[0][0]
        output('box_size: %f' % box_size)
        output('eigenvalues:')

        if get_exact is not None:
            eeigs = get_exact(n_eigs, box_size, pb.domain.shape.dim)

```

(continues on next page)

(continued from previous page)

```

        output('n      exact      FEM      error')
        for ie, eig in enumerate(evp.eigs):
            if ie < len(eeigs):
                exact = eeigs[ie]
                err = 100*abs((exact - eig)/exact)
            else:
                exact = NaN
                err = NaN
            output('%d:  %.8f  %.8f  %7.4f%%' % (ie, exact, eig, err))

    else:
        output('n      FEM')
        for ie, eig in enumerate(evp.eigs):
            output('%d:  %.8f' % (ie, eig))

filename_mesh = data_dir + '/meshes/quantum/square.mesh'

options = {
    'n_eigs' : n_eigs,
    'eigs_only' : False,
    'post_process_hook_final' : 'report_eigs',

    'evps' : 'eig',
}

regions = {
    'Omega' : 'all',
    'Surface' : ('vertices of surface', 'facet'),
}

materials = {
    'm' : ({'val' : 0.5},),
    'mat_v' : 'fun_v',
}

functions = {
    'fun_v' : (fun_v,),
}

approx_order = 2
fields = {
    'field_Psi' : ('real', 'scalar', 'Omega', approx_order),
}

variables = {
    'Psi' : ('unknown field', 'field_Psi', 0),
    'v' : ('test field', 'field_Psi', 'Psi'),
}

ebcs = {
    'ZeroSurface' : ('Surface', {'Psi.0' : 0.0}),
}

```

(continues on next page)

(continued from previous page)

```

integrals = {
    'i' : 2 * approx_order,
}

equations = {
    'lhs' : """dw_laplace.i.Omega(m.val, v, Psi)
               + dw_dot.i.Omega(mat_v.V, v, Psi)""",
    'rhs' : """dw_dot.i.Omega(v, Psi)""",
}

solvers = {
    'eig' : ('eig.scipy', {
        'method' : 'eigsh',
        'tol' : 1e-10,
        'maxiter' : 150,

        # Compute the eigenvalues near tau using the shift-invert mode.
        'which' : 'LM',
        'sigma' : tau,
    }),
}

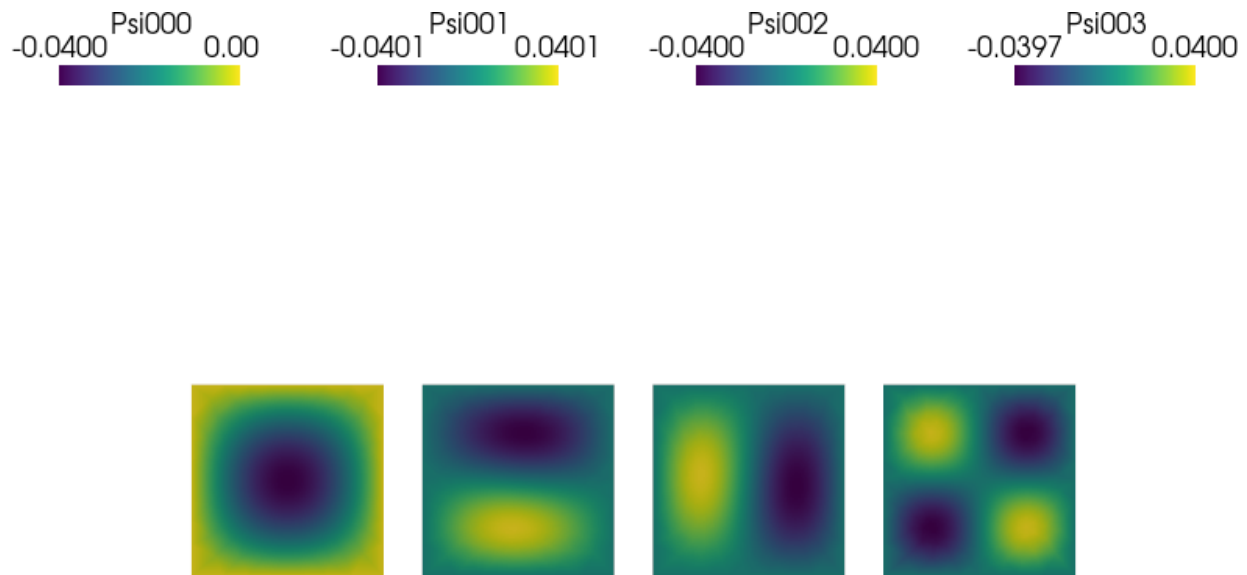
return locals()

```

quantum/well.py**Description**

Quantum potential well.

See *quantum/quantum_common.py*.



source code

```
"""
Quantum potential well.

See :ref:`quantum-quantum_common`.
"""
from __future__ import absolute_import

from sfepy.examples.quantum.quantum_common import common

def get_exact(n_eigs, box_size, dim):
    from numpy import pi

    if dim == 2:
        eigs = [pi**2/(2*box_size**2)*x
                for x in [2, 5, 5, 8, 10, 10, 13, 13, 17, 17, 18, 20, 20]]

    elif dim == 3:
        eigs = [pi**2/(2*box_size**2)*x
                for x in [3, 6, 6, 6, 9, 9, 9, 11, 11, 11,
                        12, 14, 14, 14, 14, 14, 14, 17, 17, 17]]
```

(continues on next page)

(continued from previous page)

```

    return eigs

def fun_v(ts, coor, mode=None, **kwargs):
    from numpy import zeros_like

    if not mode == 'qp': return

    out = {}
    val = zeros_like(coor[:,0])

    val.shape = (val.shape[0], 1, 1)
    out['V'] = val
    return out

def define(n_eigs=10, tau=0.0):
    l = common(fun_v, get_exact=get_exact, n_eigs=n_eigs, tau=tau)
    return l

```

1.5.7 Example Applications

- Homogenization of peristaltic flows in piezoelectric porous media (2023)
- Deformation of a foam-reinforced shell beam (2023)
- Two-scale numerical simulation of a large deforming fluid-saturated porous structure (2021)
- Homogenization of the vibro-acoustic transmission on perforated plates with embedded resonators (2021)
- Multiscale numerical modelling of perfusion in deformable double porous media described by the Biot-Darcy-Brinkman model (2020)
- Homogenization of piezoelectric porous media (2020)
- Numerical simulation of viscous flow in deformable double porous media (2020)
- Numerical simulations of large-deforming fluid-saturated porous media using an Eulerian incremental formulation (2017)
- Fish heart model (2010)
- Phononic materials (2010)

Note that older examples do not reflect the current state of SfePy.

1.6 Useful Code Snippets and FAQ

Code examples below that use *sfePy*-* scripts assume the *sfePy* package to be installed, see also [Installation](#).

1.6.1 1. Installation

1.1 No module named 'sfepy.discrete.common.extmods.mappings'

When installing SfePy from sources or using the git version, its extension modules have to be compiled before using the package, see *Compilation of C Extension Modules*.

1.2 ModuleNotFoundError: No module named 'sfepy' + in-place compilation

The extension modules are compiled in place, but `ModuleNotFoundError: No module named 'sfepy'` shows up when running some interactive examples/scripts/modules from the SfePy source directory.

On some platforms the current directory is not in the `sys.path` directory list. Add it using:

```
export PYTHONPATH=.
```

or add the following code prior to `sfepy` imports into the module:

```
import sys
sys.path.append('.')
```

1.6.2 2. Mesh-related tasks

2.1 Checking and fixing a mesh

To check mesh errors such as double vertices or disconnected components, try the following.

- Show the mesh Euler characteristic, number of components and other information:

```
sfepy-mesh info -d cylinder.mesh
```

- Fix double/disconnected vertices:

```
sfepy-convert -m bad.mesh maybe-good.mesh
```

2.2 Convert a mesh to another format

- Simple conversion:

```
sfepy-convert mesh.format1 mesh.format2
```

- Scaling the mesh anisotropically:

```
sfepy-convert -s 2,4,3 cylinder.mesh cylinder-scaled.mesh
```

2.3 Remove lower-dimensional entities from a mesh

Use `sfepy-convert` with the `-d <dimension>` option, where `<dimension>` is the topological dimension of cells that should be in the mesh. For example, `-d 2` stores only the 2D cells.

2.4 Gmsh format selection

It is suggested to use `msh22` format instead of the default `msh4` when generating a mesh with `gmsh`:

```
gmsh -2 cylinder.geo -o cylinder.msh -format msh22
```

`msh22` seems to be more reliable and foolproof when converting.

1.6.3 3. Regions

3.1 Verify that regions are correctly defined

- Using the problem description files (declarative API):

```
sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py --save-regions-as-groups-  
→--solve-not  
sfepy-view -e cylinder_regions.vtk
```

- In a script (imperative API):

```
problem.save_regions_as_groups('regions')
```

3.2 Define a region using a function of coordinates with the imperative API

Examples:

- A facet region defined using a function of mesh vertex coordinates:

```
from sfepy.discrete import Function, Functions

def _get_region(coors, domain=None):
    ii = np.nonzero(coors[:,0] < 0.5)[0]
    return ii

get_region = Function('get_region', _get_region)
region = domain.create_region(
    'Region', 'vertices by get_region', 'facet',
    functions=Functions([get_region]),
)
```

- Analogously a cell region defined using the coordinates of cell centroids:

```
# ...
region = domain.create_region(
    'Region', 'cells by get_region', 'cell',
    functions=Functions([get_region]),
)
```

1.6.4 4. Material parameters

4.1 How to set material parameters per region with the imperative API?

Example: define rho, D to have different values in regions omega1, omega2:

```
m = Material('m', values={'rho': {'omega1': 2700, 'omega2': 6000},
                          'D': {'omega1': D1, 'omega2': D2}})
```

4.2 How to implement state dependent materials?

Besides writing a custom solver, one can use pseudo-time-stepping for this purpose, as demonstrated in [linear_elasticity/material_nonlinearity.py](#) or [diffusion/poisson_field_dependent_material.py](#). Note that the examples are contrived, and in practice care must be taken to ensure convergence.

4.3 2D and 3D linear elasticity consistency

Why are results of a 2D elasticity simulation not consistent with a properly constrained 3D elasticity simulation?

Possible reason: when using the Young's modulus and Poisson's ratio as input parameters, and then calling [stiffness_from_youngpoisson\(\)](#), note that the default value of the plane argument is 'strain', corresponding to the plane strain assumption, see also [lame_from_youngpoisson\(\)](#). Try setting plane='stress'.

4.4 How to set material parameters by a function with the imperative API?

Example (also showing the full material function signature):

```
from sfepy.discrete import Material, Function

def get_pars(ts, coors, mode=None,
             equations=None, term=None, problem=None, **kwargs):
    value1 = a_function(ts.t, coors)
    value2 = another_function(ts.step, coors)
    if mode == 'qp':
        out = {
            'value1' : value1.reshape(coors.shape[0], 1, 1),
            'value2' : value2.reshape(coors.shape[0], 1, 1),
        }
    return out
m = Material('m', function=Function('get_pars', get_pars))
```

4.5 How to get cells corresponding to coordinates in a material function?

The full signature of the material function is:

```
def get_pars(ts, coors, mode=None,
            equations=None, term=None, problem=None, **kwargs)
```

Thus it has access to `term.region.cells`, hence access to the cells that correspond to the coordinates. The length of the `coors` is `n_cell * n_qp`, where `n_qp` is the number of quadrature points per cell, and `n_cell = len(term.region.cells)`, so that `coors.reshape((n_cell, n_qp, -1))` can be used.

4.6 Ordering of symmetric tensor components - no Voigt notation!

Due to historical reasons, SfePy does not use the [Voigt notation](#) for storing 3D symmetric tensors. Instead, the ordering found in Crisfield¹ is used, where e.g. the stress tensor components stored in a vector are ordered as [11, 22, 33, 12, 13, 23]. That is, the 12 and 23 components are swapped w.r.t. the Voigt notation. The same holds for the strain vector or the 6×6 stiffness matrix.

1.6.5 5. Solvers

5.1 How to work with solvers/preconditioners?

See [multi_physics/biot_short_syntax.py](#) (user-defined preconditioners) or [navier_stokes/stokes_slip_bc.py](#) (petsc solver setup).

5.2 How to get the linear system components: the matrix and the right-hand side?

To get the residual vector $\mathbf{r}$ (see *Implementation of Essential Boundary Conditions*) and the tangent matrix $\mathbf{K}$, the imperative API can be used as follows:

```
# pb is a Problem instance,
pb.set_bcs(ebcs=Conditions([...])) # Set Dirichlet boundary conditions.
pb.set_ics(Conditions([...])) # Set initial conditions (if any).
variables = pb.get_initial_state()
pb.time_update()
pb.update_materials()
variables.apply_ebc()
r = pb.equations.eval_residuals(variables())
K = pb.equations.eval_tangent_matrices(variables(), pb.mtx_a)
```

See also [diffusion/poisson_parallel_interactive.py](#).

1.6.6 6. Postprocessing and visualization

6.1 Higher order DOF visualization for an approximation order greater than one

By default, the additional, higher order DOFs, are not used in the VTK/HDF5 results files ('strip' linearization kind). To see the influence of those DOFs, 'adaptive' linearization has to be used, see [diffusion/sinbc.py](#) (declarative API) and [diffusion/laplace_refine_interactive.py](#) or [multi_physics/biot_parallel_interactive.py](#) (imperative API, search linearization).

¹ M. A. Crisfield, Non-Linear Finite Element Analysis of Solids and Structures (John Wiley & Sons, Chichester, 1996).

6.2 Visualization of various FEM-related information

- Quadrature rules:

```
python3 sfepy/scripts/plot_quadratures.py
```

- Facet orientations - run in the source code directory and make sure the current directory is in the Python's path list (see ``Miscellaneous``):

```
python3 sfepy/postprocess/plot_facets.py
```

- Global and local numberings of mesh topological entities (cells, faces, edges, vertices):

```
python3 sfepy/scripts/plot_mesh.py meshes/elements/2_4_2.mesh
```

The global numbers serve as indices into connectivities. In the plot, the global numbers are on the entities, the cell-local ones are inside the cells next to each entity towards the cell centroids.

1.6.7 7. Finite element method implementation

7.1 Finite element approximation (field) order and numerical quadrature order

SfePy supports reading only straight-facet (linear approximation) meshes, nevertheless field orders higher than one can be used, because internally, the mesh elements are enriched with the required additional nodes. The calculation then occurs on such an augmented mesh with appropriate higher order elements.

The quadrature order equal to two-times the field order (used in many examples) works well for bilinear forms with constant (on each element) material parameters. For example, a dot product involves integrating $u * v$, so if the approximation order of u and v is 1, their product's order is 2. Of course, there are terms that could use a lower quadrature order, or higher, depending on the data. Increased quadrature order is required e.g. in terms with highly oscillating material coefficients.

Example:

```
approx_order = 2
# The finite element approximation order.
fields = {
    'displacement': ('real', 3, 'Omega', approx_order),
}
# The numerical quadrature order.
integrals = {
    'i' : 2 * approx_order,
}
```

7.2 Numbering of DOFs

Locally (in a connectivity row), the DOFs are stored DOF-by-DOF (u_0 in all local nodes, u_1 in all local nodes, ...).

Globally (in a state vector), the DOFs are stored node-by-node (u_0 , u_1 , ..., u_X in node 0, u_0 , u_1 , ..., u_X in node 1, ...).

See also `create_adof_conn()`.

7.3 Where is the code that calculates the element (e.g. stiffness) matrix?

The code that computes the per element residuals and matrices is organized in terms, see [Term Overview](#) - click on the term class name and then “source” link to see the code. The original terms are implemented in C, newer terms tend to be implemented directly in Python. The structure and attributes of a term class are described in [How to Implement a New Term](#).

7.4 What structural elements are available in SfePy?

The code is currently focused on solid elements. The only supported structural elements are

- shell10x, see [linear_elasticity/shell10x_cantilever.py](#),
- Mooney-Rivlin membrane with plain stress assumption, see [large_deformation/balloon.py](#),
- linear truss, see [linear_elasticity/truss_bridge.py](#), [linear_elasticity/truss_bridge3d.py](#),
- linear spring elements, see [linear_elasticity/multi_point_constraints.py](#).

1.7 Theoretical Background

This part introduces parts the theoretical mathematical background necessary to use SfePy effectively. It also discusses some implementation choices done in SfePy.

Contents:

1.7.1 Notes on solving PDEs by the Finite Element Method

The Finite Element Method (FEM) is the numerical method for solving Partial Differential Equations (PDEs). FEM was developed in the middle of XX. century and now it is widely used in different areas of science and engineering, including mechanical and structural design, biomedicine, electrical and power design, fluid dynamics and other. FEM is based on a very elegant mathematical theory of weak solution of PDEs. In this section we will briefly discuss basic ideas underlying FEM.

Strong form of Poisson’s equation and its integration

Let us start our discussion about FEM with the strong form of Poisson’s equation

$$\Delta T = f(x), \quad x \in \Omega, \quad (1.11)$$

$$T = u(x), \quad x \in \Gamma_D, \quad (1.12)$$

$$\nabla T \cdot \mathbf{n} = g(x), \quad x \in \Gamma_N, \quad (1.13)$$

where $\Omega \subset \mathbb{R}^n$ is the solution domain with the boundary $\partial\Omega$, Γ_D is the part of the boundary where Dirichlet boundary conditions are given, Γ_N is the part of the boundary where Neumann boundary conditions are given, $T(x)$ is the unknown function to be found, $f(x)$, $u(x)$, $g(x)$ are known functions.

FEM is based on a weak formulation. The weak form of the equation (1.11) is

$$\int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = 0,$$

where s is a **test** function. Integrating this equation by parts

$$\begin{aligned} 0 &= \int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = \int_{\Omega} \nabla \cdot (\nabla T) \cdot s \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega = \\ &= - \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega + \int_{\Omega} \nabla \cdot (\nabla T \cdot s) \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega \end{aligned}$$

and applying Gauss theorem we obtain:

$$0 = - \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega + \int_{\Gamma_D \cup \Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega$$

or

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_D \cup \Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega.$$

The surface integral term can be split into two integrals, one over the Dirichlet part of the surface and second over the Neumann part

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_D} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma + \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega. \quad (1.14)$$

The equation (1.14) is the initial weak form of the Poisson's problem (1.11)–(1.13). But we can not work with it without applying the boundary conditions. So it is time to talk about the boundary conditions.

Dirichlet Boundary Conditions

On the Dirichlet part of the surface we have two restrictions. One is the Dirichlet boundary conditions $T(x) = u(x)$ as they are, and the second is the integral term over Γ_D in equation (1.14). To be consistent we have to use only the Dirichlet conditions and avoid the integral term. To implement this we can take the function $T \in V(\Omega)$ and the test function $s \in V_0(\Omega)$, where

$$V(\Omega) = \{v(x) \in H^1(\Omega)\},$$

$$V_0(\Omega) = \{v(x) \in H^1(\Omega); v(x) = 0, x \in \Gamma_D\}.$$

In other words the unknown function T must be continuous together with its gradient in the domain. In contrast the test function s must be also continuous together with its gradient in the domain but it should be zero on the surface Γ_D .

With this requirement the integral term over Dirichlet part of the surface is vanishing and the weak form of the Poisson equation for $T \in V(\Omega)$ and $s \in V_0(\Omega)$ becomes

$$\begin{aligned} \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega &= \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \\ T(x) &= u(x), \quad x \in \Gamma_D. \end{aligned}$$

That is why Dirichlet conditions in FEM terminology are called **Essential Boundary Conditions**. These conditions are not a part of the weak form and they are used as they are.

Neumann Boundary Conditions

The Neumann boundary conditions correspond to the known flux $g(x) = \nabla T \cdot \mathbf{n}$. The integral term over the Neumann surface in the equation (1.14) contains exactly the same flux. So we can use the known function $g(x)$ in the integral term:

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega,$$

where test function s also belongs to the space V_0 .

That is why Neumann conditions in FEM terminology are called **Natural Boundary Conditions**. These conditions are a part of weak form terms.

The weak form of the Poisson's equation

Now we can write the resulting weak form for the Poisson's problem (1.11)–(1.13). For any test function $s \in V_0(\Omega)$ find $T \in V(\Omega)$ such that

$$\boxed{\begin{aligned} \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega &= \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \quad \text{and} \\ T(x) &= u(x), \quad x \in \Gamma_D. \end{aligned}} \quad (1.15)$$

Discussion of discretization and meshing

It is planned to have an example of the discretization based on the Poisson's equation weak form (1.15). For now, please refer to the wikipedia page [Finite Element Method](#) for a basic description of the discretization and meshing.

Numerical solution of the problem

To solve numerically given problem based on the weak form (1.15) we have to go through 5 steps:

1. Define geometry of the domain Ω and surfaces Γ_D and Γ_N .
2. Define the known functions f , u and g .
3. Define the unknown function T and the test functions s .
4. Define essential boundary conditions (Dirichlet conditions) $T(x) = u(x), x \in \Gamma_D$.
5. Define equation and natural boundary conditions (Neumann conditions) as the set of all integral terms $\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega, \int_{\Gamma_N} g \cdot s \, d\Gamma, \int_{\Omega} f \cdot s \, d\Omega$.

1.7.2 Implementation of Essential Boundary Conditions

The essential boundary conditions can be applied in several ways. Here we describe the implementation used in SfePy.

Motivation

Let us solve a linear system $Ax = b$ with $n \times n$ matrix A with n_f values in the x vector known. The known values can be for example EBC values on a boundary, if A comes from a PDE discretization. If we put the known fixed values into a vector x_f , that has the same size as x , and has zeros in positions that are not fixed, we can easily construct a $n \times n_r$ matrix T that maps the reduced vector x_r of size $n_r = n - n_f$, where the fixed values are removed, to the full vector x :

$$x = Tx_r + x_f .$$

With that the reduced linear system with a $n_r \times n_r$ can be formed:

$$T^T ATx_r = T^T(b - Ax_f)$$

that can be solved by a linear solver. We can see, that the (non-zero) known values are now on the right-hand side of the linear system. When the known values are all zero, we have simply

$$T^T ATx_r = T^T b ,$$

which is convenient, as it allows simply throwing away the A and b entries corresponding to the known values already during the finite element assembling.

Implementation

All PDEs in SfePy are solved in a uniform way as a system of non-linear equations

$$f(u) = 0 ,$$

where f is the nonlinear function and u the vector of unknown DOFs. This system is solved iteratively by the Newton method

$$u^{new} = u^{old} - \left(\frac{df}{du^{old}}\right)^{-1} f(u^{old})$$

until a convergence criterion is met. Each iteration involves solution of the system of linear equations

$$K\Delta u = r ,$$

where the tangent matrix K and the residual r are

$$K \equiv \frac{df}{du^{old}} ,$$
$$r \equiv f(u^{old}) .$$

Then

$$u^{new} = u^{old} - \Delta u .$$

If the initial (old) vector u^{old} contains the values of EBCs at correct positions, the increment Δu is zero at those positions. This allows us to assemble directly the reduced matrix $T^T K T$, the right-hand side $T^T r$, and ignore the values of EBCs during assembling. The EBCs are satisfied automatically by applying them to the initial guess u^0 , that is given to the Newton solver.

Linear Problems

For linear problems we have

$$\begin{aligned} f(u) &\equiv Au - b = 0, \\ \frac{df}{du} &= A, \end{aligned}$$

and so the Newton method converges in a single iteration:

$$u^{new} = u^{old} - A^{-1}(Au^{old} - b) = A^{-1}b.$$

Evaluation of Residual and Tangent Matrix

The evaluation of the residual f as well as the tangent matrix K within the Newton solver proceeds in the following steps:

- The EBCs are applied to the full DOF vector u .
- The reduced vector u_r is passed to the Newton solver.
- Newton iteration loop:
 - Evaluation of f_r or K_r :
 1. u is reconstructed from u_r ;
 2. local element contributions are evaluated using u ;
 3. local element contributions are assembled into f_r or K_r - values corresponding to fixed DOF positions are thrown away.
 - The reduced system $K_r \Delta u_r = r_r$ is solved.
 - Solution is updated: $u_r \leftarrow u_r - \Delta u_r$.
 - The loop is terminated if a stopping condition is satisfied, the solver returns the final u_r .
- The final u is reconstructed from u_r .

1.8 Term Overview

1.8.1 Term Syntax

In general, the syntax of a term call is:

$$\langle \text{term name} \rangle . \langle i \rangle . \langle r \rangle (\langle \text{arg1} \rangle , \langle \text{arg2} \rangle , \dots),$$

where $\langle i \rangle$ denotes an integral name (i.e. a name of numerical quadrature to use) and $\langle r \rangle$ marks a region (domain of the integral).

The following notation is used:

Table 2: Notation.

symbol	meaning
Ω	cell (volume) (sub)domain
Γ	facet (surface) (sub)domain
$\mathcal{D}$	cell or facet (sub)domain
d	dimension of space
t	time
y	any function
$\underline{y}$	any vector function
$\underline{n}$	unit outward normal
q	scalar test or parameter function
p	scalar unknown or parameter function
$\underline{v}$	vector test or parameter function
$\underline{w}, \underline{u}$	vector unknown or parameter function
$\underline{e}(\underline{u})$	Cauchy strain tensor ($\frac{1}{2}((\nabla \underline{u}) + (\nabla \underline{u})^T)$)
$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
J	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = F^T F$
$\underline{\underline{E}}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j})$
$\underline{\underline{S}}$	second Piola-Kirchhoff stress tensor
$\underline{f}$	vector volume forces
f	scalar volume force (source)
ρ	density
ν	kinematic viscosity
$c, \underline{c}, \underline{\underline{c}}$	any constant
$\delta_{ij}, \underline{\underline{I}}$	Kronecker delta, identity matrix
$\text{tr } \underline{\bullet}$	trace of a second order tensor ($\sum_{i=1}^d \bullet_{ii}$)
$\text{dev } \underline{\bullet}$	deviator of a second order tensor ($\underline{\bullet} - \frac{1}{d} \text{tr } \underline{\bullet}$)
$T_K \in \mathcal{T}_h$	K -th element of triangulation (= mesh) $\mathcal{T}_h$ of domain Ω
$K \leftarrow \mathcal{I}_h$	K is assigned values from $\{0, 1, \dots, N_h - 1\} \equiv \mathcal{I}_h$ in ascending order

The suffix “₀” denotes a quantity related to a previous time step.

Term names are (usually) prefixed according to the following conventions:

Table 3: Term name prefixes.

pre- fix	meaning	evaluation modes	meaning
dw	discrete weak	‘weak’	terms having a virtual (test) argument and zero or more unknown arguments, used for FE assembling
ev	evaluate	‘eval’, ‘el_eval’, ‘el_avg’, ‘qp’	terms having all arguments known, modes ‘el_avg’, ‘qp’ are not supported by all <i>ev_</i> terms
de	discrete einsum	any (work in progress)	multi-linear terms defined using an enriched einsum notation

Evaluation modes ‘eval’, ‘el_avg’ and ‘qp’ are defined as follows:

Table 4: Evaluation modes.

mode	definition
‘eval’	$\int_{\mathcal{D}}(\cdot)$
‘el_avg’	vector for $K \leftarrow \mathcal{I}_h : \int_{T_K}(\cdot) / \int_{T_K} 1$
‘qp’	$(\cdot) _{qp}$

1.8.2 Term Table

Below we list all the terms available in automatically generated tables. The first column lists the name, the second column the argument lists and the third column the mathematical definition of each term. The terms are divided into the following tables:

- *Table of basic terms*
- *Table of large deformation terms* (total/updated Lagrangian formulation)
- *Table of sensitivity terms*
- *Table of special terms*
- *Table of multi-linear terms*

The notation <virtual> corresponds to a test function, <state> to a unknown function and <parameter> to a known function. By <material> we denote material (constitutive) parameters, or, in general, any given function of space and time that parameterizes a term, for example a given traction force vector.

Table of basic terms

Table 5: Basic terms

name/class	arguments	definition	examples
dw_advect_div_fre AdvectDivFreeTerm	<material> <virtual>, <state>	$\int_{\Omega} \nabla \cdot (\underline{y} p) q = \int_{\Omega} \underbrace{((\nabla \cdot \underline{y}) + \underline{y} \cdot \nabla) p}_{\equiv 0} q$	tim.adv.dif
dw_bc_newton BCNewtonTerm	<material_ <material_ <virtual>, <state>	$\int_{\Gamma} \alpha q (p - p_{\text{outer}})$	tim.heq.equ.mul.mat
dw_biot BiotTerm	<material> <virtual/ param_v>, <state/ param_s> <material> <state>, <virtual>	$\int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}), \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u})$	bio.npb.lag , the.ela.ess , bio , bio.npb , the.ela , bio.sho.syn
ev_biot_stress BiotStressTerm	<material> <parameter>	$- \int_{\Omega} \alpha_{ij} p$	
ev_cauchy_strain CauchyStrainTerm	<parameter>	$\int_{\mathcal{D}} \underline{e}(\underline{w})$	
ev_cauchy_stress CauchyStressTerm	<material> <parameter>	$\int_{\mathcal{D}} D_{ijkl} e_{kl}(\underline{w})$	
dw_contact ContactTerm	<material> <virtual>, <state>	$\int_{\Gamma_c} \varepsilon_N \langle g_N(\underline{u}) \rangle \underline{n} \underline{v}$	two.bod.con
dw_contact_ipc ContactIPCTerm	<material_ <material_ <material_ <material_ <virtual>, <state>	$\int_{\Gamma_c} \sum_{k \in C} \nabla b(d_k, \underline{u})$	

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_contact_plane <i>ContactPlaneTerm</i>	<material_> <material_> <material_> <material_> <virtual>, <state>	$\int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}$	<i>ela.con.pla</i>
dw_contact_sphere <i>ContactSphereTerm</i>	<material_> <material_> <material_> <virtual>, <state>	$\int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}(\underline{u})$	<i>ela.con.sph</i>
dw_convect <i>ConvectTerm</i>	<virtual>, <state>	$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$	<i>nav.sto</i> , <i>nav.sto</i> , <i>nav.sto.iga</i>
dw_convect_v_grad <i>ConvectVGradSTerm</i>	<virtual>, <state_v>, <state_s>	$\int_{\Omega} q(\underline{u} \cdot \nabla p)$	<i>poi.fun</i>
ev_def_grad <i>DeformationGradient</i>	<parameter>	$\underline{\underline{F}} = \frac{\partial \underline{x}}{\partial \underline{X}} _{qp} = \underline{\underline{I}} + \frac{\partial \underline{u}}{\partial \underline{X}} _{qp},$ $\underline{x} = \underline{X} + \underline{u}, J = \det(\underline{\underline{F}})$	
dw_dg_advect_lax <i>AdvectionDGFlux</i>	<opt_mater> <material_> <virtual>, <state>	$\int_{\partial T_K} \underline{n} \cdot \underline{f}^*(p_{in}, p_{out}) q$ where $\underline{f}^*(p_{in}, p_{out}) = \underline{a} \frac{p_{in} + p_{out}}{2} + (1 - \alpha) \underline{n} C \frac{p_{in} - p_{out}}{2},$	<i>adv.dif.2D</i> , <i>adv.1D</i> , <i>adv.2D</i>
dw_dg_diffusion_f <i>DiffusionDGFlux</i>	<material_> <state>, <virtual> <material_> <virtual>, <state>	$\int_{\partial T_K} D \langle \nabla p \rangle [q], \int_{\partial T_K} D \langle \nabla q \rangle [p]$ where $\langle \nabla \phi \rangle = \frac{\nabla \phi_{in} + \nabla \phi_{out}}{2}$ $[\phi] = \phi_{in} - \phi_{out}$	<i>lap.2D</i> , <i>adv.dif.2D</i> , <i>bur.2D</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_dg_interior_penetration <i>DiffusionInteriorPenetration</i>	<material> <material> <virtual>, <state>	$\int_{\partial T_K} \bar{D} C_w \frac{Ord^2}{d(\partial T_K)} [p][q]$ where $[\phi] = \phi_{in} - \phi_{out}$	<i>lap.2D</i> , <i>adv.dif.2D</i> , <i>bur.2D</i>
dw_dg_nonlinear_hyperelasticity <i>NonlinearHyperelasticity</i>	<opt_material> <fun>, <fun_d>, <virtual>, <state>	$\int_{\partial T_K} \underline{n} \cdot \underline{f}^*(p_{in}, p_{out}) q$ where $\underline{f}^*(p_{in}, p_{out}) = \frac{\underline{f}(p_{in}) + \underline{f}(p_{out})}{2} + (1 - \alpha) \underline{n} C \frac{p_{in} - p_{out}}{2},$	<i>bur.2D</i>
dw_diffusion <i>DiffusionTerm</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p$	<i>bio.npb.lag</i> , <i>vib.aco</i> , <i>pie.ela</i> , <i>bio</i> , <i>bio.npb</i> , <i>poi.neu</i> , <i>pie.ela</i> , <i>dar.flo.mul</i> , <i>bio.sho.syn</i>
dw_diffusion_coupled <i>DiffusionCoupledTerm</i>	<material> <virtual/ param_1>, <state/ param_2> <material> <state>, <virtual>	$\int_{\Omega} p K_j \nabla_j q, \int_{\Omega} q K_j \nabla_j p$	
dw_diffusion_reaction <i>DiffusionRTerm</i>	<material> <virtual>	$\int_{\Omega} K_j \nabla_j q$	
ev_diffusion_velocity <i>DiffusionVelocityTerm</i>	<material> <parameter>	$- \int_{\mathcal{D}} K_{ij} \nabla_j p$	
ev_div <i>DivTerm</i>	<opt_material> <parameter>	$\int_{\mathcal{D}} \nabla \cdot \underline{u}, \int_{\mathcal{D}} c \nabla \cdot \underline{u}$	

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_div <i>DivOperatorTerm</i>	<opt_mater <virtual>	$\int_{\Omega} \nabla \cdot \underline{v} \text{ or } \int_{\Omega} c \nabla \cdot \underline{v}$	
dw_div_grad <i>DivGradTerm</i>	<opt_mater <virtual/ param_1> <state/ param_2>	$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nabla \underline{v} : \nabla \underline{u}$	<i>nav.sto, sto.sli.bc,</i> <i>nav.sto,</i> <i>nav.sto.iga,</i> <i>sta.nav.sto, sto</i>
dw_dot <i>DotProductTerm</i>	<opt_mater <virtual/ param_1> <state/ param_2>	$\int_{\mathcal{D}} qp, \int_{\mathcal{D}} \underline{v} \cdot \underline{u}$ $\int_{\Gamma} \underline{v} \cdot \underline{np}, \int_{\Gamma} q \underline{n} \cdot \underline{u},$ $\int_{\mathcal{D}} c qp, \int_{\mathcal{D}} c \underline{v} \cdot \underline{u}, \int_{\mathcal{D}} \underline{v} \cdot \underline{c} \cdot \underline{u}$	<i>aco, vib.aco,</i> <i>sto.sli.bc,</i> <i>osc, poi.fun,</i> <i>poi.per.bou.con,</i> <i>tim.heq.equ.mul.mat,</i> <i>aco,</i> <i>mod.ana.dec,</i> <i>adv.2D, bor,</i> <i>bur.2D, the.ele,</i> <i>hyd, lin.ela.up,</i> <i>tim.poi.exp, wel,</i> <i>bal, pie.ela,</i> <i>dar.flo.mul,</i> <i>adv.1D,</i> <i>lin.ela.dam,</i> <i>pie.ela, hel.apa,</i> <i>ref.evp, tim.poi,</i> <i>tim.adv.dif</i>
dw_elastic_wave <i>ElasticWaveTerm</i>	<material_ <material_ <virtual> <state>	$\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) g_{kl}(\underline{u})$	
dw_elastic_wave_c <i>ElasticWaveCauchy</i>	<material_ <material_ <virtual> <state> <material_ <material_ <state> <virtual>	$\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) e_{kl}(\underline{u})$ $\int_{\Omega} D_{ijkl} g_{ij}(\underline{u}) e_{kl}(\underline{v})$	
dw_electric_source <i>ElectricSourceTerm</i>	<material> <virtual> <parameter>	$\int_{\Omega} cs (\nabla \phi)^2$	<i>the.ele</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
ev_grad <i>GradTerm</i>	<opt_mater <parameter>	$\int_{\mathcal{D}} \nabla p \text{ or } \int_{\mathcal{D}} \nabla \underline{u}$ $\int_{\mathcal{D}} c \nabla p \text{ or } \int_{\mathcal{D}} c \nabla \underline{u}$	
dw_integrate <i>IntegrateOperator</i>	<opt_mater <virtual>	$\int_{\mathcal{D}} q \text{ or } \int_{\mathcal{D}} cq$	<i>aco</i> , <i>vib.aco</i> , <i>poi.neu</i> , <i>hel.apa</i> , <i>poi.per.bou.con</i> , <i>tim.heq.equ.mul.mat</i> , <i>aco</i> , <i>dar.flo.mul</i>
ev_integrate <i>IntegrateTerm</i>	<opt_mater <parameter>	$\int_{\mathcal{D}} y, \int_{\mathcal{D}} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n}$ $\int_{\mathcal{D}} cy, \int_{\mathcal{D}} c\underline{y}, \int_{\Gamma} c\underline{y} \cdot \underline{n} \text{ flux}$	
ev_integrate_mat <i>IntegrateMaterialTerm</i>	<material> <parameter>	$\int_{\mathcal{D}} c$	
dw_jump <i>SurfaceJumpTerm</i>	<opt_mater <virtual>, <state_1>, <state_2>	$\int_{\Gamma} cq(p_1 - p_2)$	<i>aco</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_laplace <i>LaplaceTerm</i>	<opt_mater <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} c \nabla q \cdot \nabla p$	<i>aco, vib.aco,</i> <i>sto.sli.bc,</i> <i>osc, poi.fun,</i> <i>poi.per.bou.con,</i> <i>cub,</i> <i>poi.fie.dep.mat,</i> <i>tim.heq.equ.mul.mat,</i> <i>aco, poi.iga,</i> <i>poi.par.stu, bor,</i> <i>lap.cou.lcb,</i> <i>bur.2D, the.ele,</i> <i>lap.flu.2d, hyd,</i> <i>poi, adv.dif.2D,</i> <i>tim.poi.exp,</i> <i>lap.2D, wel,</i> <i>poi.sho.syn,</i> <i>the.ela.ess,</i> <i>hel.apa, ref.evp,</i> <i>lap.tim.ebc,</i> <i>lap.1d, tim.poi,</i> <i>tim.adv.dif, sin</i> <i>sta.nav.sto</i>
dw_lin_convect <i>LinearConvectTerm</i>	<virtual>, <parameter> <state>	$\int_{\Omega} ((\underline{w} \cdot \nabla) \underline{u}) \cdot \underline{v}$ $((\underline{w} \cdot \nabla) \underline{u}) _{qp}$	
dw_lin_convect2 <i>LinearConvect2Term</i>	<material> <virtual>, <state>	$\int_{\Omega} ((\underline{c} \cdot \nabla) \underline{u}) \cdot \underline{v}$ $((\underline{c} \cdot \nabla) \underline{u}) _{qp}$	
dw_lin_dspring <i>LinearDSpringTerm</i>	<opt_mater <material> <virtual>, <state>	$f_k^{(i)} = -f_k^{(j)} = K_{kl}(u_l^{(j)} - u_l^{(i)})$ $\forall \text{ elements } T_K^{i,j}$ in a region connecting nodes i, j	
dw_lin_dspring_rot <i>LinearDRotSpringTerm</i>	<opt_mater <material> <virtual>, <state>	$f_k^{(i)} = -f_k^{(j)} = K_{kl}(u_l^{(j)} - u_l^{(i)})$ $\forall \text{ elements } T_K^{i,j}$ in a region connecting nodes i, j	<i>mul.poi.con</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_lin_elastic <i>LinearElasticTensor</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$	<i>vib.aco</i> , <i>pre.fib</i> , <i>lin.vis</i> , <i>pie.ela.mac</i> , <i>the.ela</i> , <i>lin.ela.iga</i> , <i>lin.ela</i> , <i>its.3</i> , <i>sei.loa</i> , <i>ela.shi.per</i> , <i>bio.npb.lag</i> , <i>two.bod.con</i> , <i>mod.ana.dec</i> , <i>mat.non</i> , <i>bio</i> , <i>tru.bri</i> , <i>ela.con.sph</i> , <i>its.4</i> , <i>lin.ela.tra</i> , <i>com.ela.mat</i> , <i>lin.ela.mM</i> , <i>wed.mes</i> , <i>nod.lcb</i> , <i>its.2</i> , <i>lin.ela.up</i> , <i>its.1</i> , <i>bio.npb</i> , <i>lin.ela.opt</i> , <i>mix.mes</i> , <i>mul.poi.con</i> , <i>pie.ela</i> , <i>mul.nod.lcb</i> , <i>bio.sho.syn</i> , <i>ela.con.pla</i> , <i>lin.ela.dam</i> , <i>the.ela.ess</i> , <i>pie.ela</i> , <i>ela</i>
dw_lin_elastic_iso <i>LinearElasticIsotropicTensor</i>	<material_ <material_ <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$ with $D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$	
dw_lin_elastic_lam <i>LinearElasticLaminate</i>	<material_ <material_ <virtual>, <state>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$ with $D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$	

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_lin_elastic_yp <i>LinearElasticYP</i>	<material> <material> <virtual>, <state>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$ with $D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$ where $\lambda = E\nu/((1+\nu)(1-2\nu)),$ $\mu = E/2(1+\nu)$	<i>ela.ide</i>
dw_lin_prestress <i>LinearPrestress</i>	<material> <virtual/ param>	$\int_{\Omega} \sigma_{ij} e_{ij}(\underline{v})$	<i>non.hyp.mM</i> , <i>pre.fib</i> , <i>pie.ela.mac</i>
dw_lin_spring <i>LinearSpringTerm</i>	<material> <virtual>, <state>	$\underline{f}^{(i)} = -\underline{f}^{(j)} = k(\underline{u}^{(j)} - \underline{u}^{(i)})$ $\forall$ elements $T_K^{i,j}$ in a region connecting nodes i, j	
dw_lin_strain_fib <i>LinearStrainFib</i>	<material> <material> <virtual>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) (d_k d_l)$	<i>pre.fib</i>
dw_lin_truss <i>LinearTrussTerm</i>	<material> <virtual>, <state>	$F^{(i)} = -F^{(j)} = EA/l(U^{(j)} - U^{(i)})$ $\forall$ elements $T_K^{i,j}$ in a region connecting nodes i, j	<i>tru.bri</i> , <i>tru.bri</i>
ev_lin_truss_force <i>LinearTrussInte</i>	<material> <parameter>	$F = EA/l(U^{(j)} - U^{(i)})$ $\forall$ elements $T_K^{i,j}$ in a region connecting nodes i, j	
dw_mass_ad <i>MassADTerm</i>	<material> <virtual>, <state>	$\int_{\mathcal{D}} \rho \underline{v} \cdot \underline{u}$	<i>ela.ide</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_nl_diffusion <i>NonlinearDiffusion</i>	<fun>, <dfun>, <virtual>, <state>	$\int_{\Omega} \nabla q \cdot \nabla p f(p)$	<i>poi.non.mat</i>
dw_non_penetration <i>NonPenetration1</i>	<opt_mater <virtual>, <state> <opt_mater <state>, <virtual>	$\int_{\Gamma} c \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} c \hat{\lambda} \underline{n} \cdot \underline{u}$ $\int_{\Gamma} \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} \hat{\lambda} \underline{n} \cdot \underline{u}$	<i>bio.npb.lag</i>
dw_non_penetration <i>NonPenetration2</i>	<material> <virtual>, <state>	$\int_{\Gamma} c(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u})$	<i>bio.sho.syn</i>
dw_nonsym_elastic <i>NonsymElasticTensor</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} \underline{\underline{D}} \nabla \underline{u} : \nabla \underline{v}$	<i>non.hyp.mM</i>
dw_ns_dot_grad_s <i>NonlinearScalar</i>	<fun>, <fun_d>, <virtual>, <state> <fun>, <fun_d>, <state>, <virtual>	$\int_{\Omega} q \cdot \nabla \cdot \underline{f}(p) = \int_{\Omega} q \cdot \operatorname{div} \underline{f}(p), \int_{\Omega} \underline{f}(p) \cdot \nabla q$	<i>bur.2D</i>
dw_piezo_coupling <i>PiezoCouplingTensor</i>	<material> <virtual/ param_v>, <state/ param_s> <material> <state>, <virtual>	$\int_{\Omega} g_{kij} e_{ij}(\underline{v}) \nabla_k p$ $\int_{\Omega} g_{kij} e_{ij}(\underline{u}) \nabla_k q$	<i>pie.ela, pie.ela</i>
ev_piezo_strain <i>PiezoStrainTensor</i>	<material> <parameter>	$\int_{\Omega} g_{kij} e_{ij}(\underline{u})$	
ev_piezo_stress <i>PiezoStressTensor</i>	<material> <parameter>	$\int_{\Omega} g_{kij} \nabla_k p$	

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_point_load <i>ConcentratedPointLoad</i>	<material> <virtual>	$\underline{f}^i = \bar{f}^i \quad \forall \text{ FE node } i \text{ in a region}$	<i>its.1</i> , <i>its.4</i> , <i>tru.bri</i> , <i>its.3</i> , <i>she.can</i> , <i>its.2</i>
dw_point_lspring <i>LinearPointSpring</i>	<material> <virtual>, <state>	$\underline{f}^i = -k\underline{u}^i \quad \forall \text{ FE node } i \text{ in a region}$	
dw_s_dot_grad_is <i>ScalarDotGradIs</i>	<material> <virtual>, <state>	$Z^i = \int_{\Omega} q \nabla_i p$	
dw_s_dot_mgrad_s <i>ScalarDotMGradS</i>	<material> <virtual>, <state> <material> <state>, <virtual>	$\int_{\Omega} q \underline{y} \cdot \nabla p, \int_{\Omega} p \underline{y} \cdot \nabla q$	<i>adv.dif.2D</i> , <i>adv.1D</i> , <i>adv.2D</i>
dw_shell10x <i>Shell10XTerm</i>	<material> <material> <virtual>, <state>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$	<i>she.can</i>
dw_stokes <i>StokesTerm</i>	<opt_mater <virtual/ param_v>, <state/ param_s> <opt_mater <state>, <virtual>	$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u}$ or $\int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$	<i>nav.sto</i> , <i>sto.sli.bc</i> , <i>lin.ela.up</i> , <i>nav.sto</i> , <i>nav.sto.iga</i> , <i>sta.nav.sto</i> , <i>sto</i>
dw_stokes_wave <i>StokesWaveTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\underline{\kappa} \cdot \underline{u})$	
dw_stokes_wave_d <i>StokesWaveDivTerm</i>	<material> <virtual>, <state> <material> <state>, <virtual>	$\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\nabla \cdot \underline{u}), \int_{\Omega} (\underline{\kappa} \cdot \underline{u})(\nabla \cdot \underline{v})$	
ev_sum_vals <i>SumNodalValues</i>	<parameter>		

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_surface_flux <i>SurfaceFluxOpen</i>	<opt_mater <virtual>, <state>	$\int_{\Gamma} q \underline{n} \cdot \underline{K} \cdot \nabla p$	
ev_surface_flux <i>SurfaceFluxTerm</i>	<material> <parameter>	$\int_{\Gamma} \underline{n} \cdot K_{ij} \nabla_j p$	
dw_surface_ltr <i>LinearTractionTerm</i>	<opt_mater <virtual/ param>	$\int_{\Gamma} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \int_{\Gamma} \underline{v} \cdot \underline{n},$	<i>lin.vis,</i> <i>lin.ela.opt,</i> <i>tru.bri,</i> <i>lin.ela.tra,</i> <i>com.ela.mat,</i> <i>mix.mes,</i> <i>wed.mes,</i> <i>ela.shi.per,</i> <i>nod.lcb</i>
ev_surface_moment <i>SurfaceMomentTerm</i>	<material> <parameter>	$\int_{\Gamma} \underline{n} (\underline{x} - \underline{x}_0)$	
dw_surface_ndot <i>SurfaceNormalDot</i>	<material> <virtual/ param>	$\int_{\Gamma} q \underline{c} \cdot \underline{n}$	<i>lap.flu.2d</i>
ev_surface_piezo <i>SurfacePiezoFlux</i>	<material> <parameter>	$\int_{\Gamma} g_{kij} e_{ij}(\underline{u}) n_k$	
dw_v_dot_grad_s <i>VectorDotGradScal</i>	<opt_mater <virtual/ param_v>, <state/ param_s> <opt_mater <state>, <virtual>	$\begin{aligned} &\int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q \\ &\int_{\Omega} c \underline{v} \cdot \nabla p, \int_{\Omega} c \underline{u} \cdot \nabla q \\ &\int_{\Omega} \underline{v} \cdot (\underline{c} \nabla p), \int_{\Omega} \underline{u} \cdot (\underline{c} \nabla q) \end{aligned}$	<i>vib.aco</i>

continues on next page

Table 5 – continued from previous page

name/class	arguments	definition	examples
dw_vm_dot_s <i>VectorDotScalar</i>	<material> <virtual/ param_v>, <state/ param_s> <material> <state>, <virtual>	$\int_{\Omega} \underline{v} \cdot \underline{cp}, \int_{\Omega} \underline{u} \cdot \underline{cq}$	
ev_volume <i>VolumeTerm</i>	<parameter>	$\int_{\mathcal{D}} 1$	
dw_volume_lvf <i>LinearVolumeForm</i>	<material> <virtual>	$\int_{\Omega} \underline{f} \cdot \underline{v} \text{ or } \int_{\Omega} f q$	<i>adv.dif.2D,</i> <i>poi.iga, bur.2D,</i> <i>poi.par.stu</i>
dw_volume_nvf <i>NonlinearVolumeForm</i>	<fun>, <dfun>, <virtual>, <state>	$\int_{\Omega} q f(p)$	<i>poi.non.mat</i>
ev_volume_surface <i>VolumeSurfaceTerm</i>	<parameter>	$1/D \int_{\Gamma} \underline{x} \cdot \underline{n}$	
dw_zero <i>ZeroTerm</i>	<virtual>, <state>	0	<i>ela</i>

Table of sensitivity terms

Table 6: Sensitivity terms

name/class	arguments	definition	examples
dw_adj_convect1 <i>AdjConvect1Term</i>	<virtual>, <state>, <parameter>	$\int_{\Omega} ((\underline{v} \cdot \nabla) \underline{u}) \cdot \underline{w}$	
dw_adj_convect2 <i>AdjConvect2Term</i>	<virtual>, <state>, <parameter>	$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{v}) \cdot \underline{w}$	
dw_adj_div_grad <i>AdjDivGradTerm</i>	<material> <material> <virtual>, <parameter>	$w \delta_u \Psi(\underline{u}) \circ \underline{v}$	
ev_sd_convect <i>SDConvectTerm</i>	<parameter> <parameter> <parameter>	$\int_{\Omega} [u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \underline{v}) - u_k \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i]$	
ev_sd_diffusion <i>SDDiffusionTerm</i>	<material> <parameter> <parameter> <parameter>	$\int_{\Omega} \hat{K}_{ij} \nabla_i q \nabla_j p$ $\hat{K}_{ij} = K_{ij} \left(\delta_{ik} \delta_{jl} \nabla \cdot \underline{v} - \delta_{ik} \frac{\partial \mathcal{V}_j}{\partial x_l} - \delta_{jl} \frac{\partial \mathcal{V}_i}{\partial x_k} \right)$	
de_sd_diffusion <i>ESDDiffusionTerm</i>	<material> <virtual/ param_1>, <state/ param_2>, <parameter>	$\int_{\Omega} \hat{K}_{ij} \nabla_i q \nabla_j p$ $\hat{K}_{ij} = K_{ij} \left(\delta_{ik} \delta_{jl} \nabla \cdot \underline{v} - \delta_{ik} \frac{\partial \mathcal{V}_j}{\partial x_l} - \delta_{jl} \frac{\partial \mathcal{V}_i}{\partial x_k} \right)$	
ev_sd_div <i>SDDivTerm</i>	<parameter> <parameter> <parameter>	$\int_{\Omega} p [(\nabla \cdot \underline{w})(\nabla \cdot \underline{v}) - \frac{\partial \mathcal{V}_k}{\partial x_i} \frac{\partial w_i}{\partial x_k}]$	

continues on next page

Table 6 – continued from previous page

name/class	arguments	definition	examples
ev_sd_div_grad <i>SDDivGradTerm</i>	<opt_mater <parameter <parameter <parameter	$\int_{\Omega} \hat{I} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \hat{I} \nabla \underline{v} : \nabla \underline{u}$ $\hat{I}_{ijkl} = \delta_{ik} \delta_{jl} \nabla \cdot \underline{v} - \delta_{ik} \delta_{js} \frac{\partial \mathcal{V}_l}{\partial x_s} - \delta_{is} \delta_{jl} \frac{\partial \mathcal{V}_k}{\partial x_s}$	
de_sd_div_grad <i>ESDDivGradTerm</i>	<opt_mater <virtual/ param_1>, <state/ param_2>, <parameter	$\int_{\Omega} \hat{I} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \hat{I} \nabla \underline{v} : \nabla \underline{u}$ $\hat{I}_{ijkl} = \delta_{ik} \delta_{jl} \nabla \cdot \underline{v} - \delta_{ik} \delta_{js} \frac{\partial \mathcal{V}_l}{\partial x_s} - \delta_{is} \delta_{jl} \frac{\partial \mathcal{V}_k}{\partial x_s}$	
ev_sd_dot <i>SDDotTerm</i>	<parameter <parameter <parameter	$\int_{\Omega} pq(\nabla \cdot \underline{v}), \int_{\Omega} (\underline{u} \cdot \underline{w})(\nabla \cdot \underline{v})$	
de_sd_dot <i>ESDDotTerm</i>	<opt_mater <virtual/ param_1>, <state/ param_2>, <parameter	$\int_{\Omega} qp(\nabla \cdot \underline{v}), \int_{\Omega} (\underline{v} \cdot \underline{u})(\nabla \cdot \underline{v})$ $\int_{\Omega} cqp(\nabla \cdot \underline{v}), \int_{\Omega} c(\underline{v} \cdot \underline{u})(\nabla \cdot \underline{v})$ $\int_{\Omega} \underline{v} \cdot (\underline{M} \underline{u})(\nabla \cdot \underline{v})$	
de_sd_lin_elastic <i>ESDLinearElastic</i>	<material> <virtual/ param_1>, <state/ param_2>, <parameter	$\int_{\Omega} \hat{D}_{ijkl} \frac{\partial v_i}{\partial x_j} \frac{\partial u_k}{\partial x_l}$ $\hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \underline{v}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$	
ev_sd_lin_elastic <i>SDLinearElastic</i>	<material> <parameter <parameter <parameter	$\int_{\Omega} \hat{D}_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$ $\hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \underline{v}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$	

continues on next page

Table 6 – continued from previous page

name/class	arguments	definition	examples
ev_sd_piezo_coupl <i>SDPiezoCoupling</i>	<material> <parameter> <parameter> <parameter>	$\int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{u}) \nabla_k p$ $\hat{g}_{kij} = g_{kij}(\nabla \cdot \underline{v}) - g_{kil} \frac{\partial \mathcal{V}_j}{\partial x_l} - g_{lij} \frac{\partial \mathcal{V}_k}{\partial x_l}$	
de_sd_piezo_coupl <i>ESDPiezoCoupling</i>	<material> <virtual/ param_v>, <state/ param_s>, <parameter> <material> <state>, <virtual>, <parameter>	$\int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{v}) \nabla_k p, \int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{u}) \nabla_k q$ $\hat{g}_{kij} = g_{kij}(\nabla \cdot \underline{v}) - g_{kil} \frac{\partial \mathcal{V}_j}{\partial x_l} - g_{lij} \frac{\partial \mathcal{V}_k}{\partial x_l}$	
de_sd_stokes <i>ESDStokesTerm</i>	<opt_mater <virtual/ param_v>, <state/ param_s>, <parameter> <opt_mater <state>, <virtual>, <parameter>	$\int_{\Omega} p \hat{I}_{ij} \frac{\partial v_i}{\partial x_j}, \int_{\Omega} q \hat{I}_{ij} \frac{\partial u_i}{\partial x_j}$ $\hat{I}_{ij} = \delta_{ij} \nabla \cdot \underline{v} - \frac{\partial \mathcal{V}_j}{\partial x_i}$	
ev_sd_surface_inte <i>SDSurfaceIntegral</i>	<parameter> <parameter>	$\int_{\Gamma} p \nabla \cdot \underline{v}$	
de_sd_surface_ltr <i>ESDLinearTractio</i>	<opt_mater <virtual/ param>, <parameter>	$\int_{\Gamma} \underline{v} \cdot [(\hat{\underline{\sigma}} \nabla \cdot \underline{v} - \hat{\underline{\sigma}} \nabla \underline{v}) \underline{n}]$ $\hat{\underline{\sigma}} = \underline{I}, \hat{\underline{\sigma}} = c \underline{I} \text{ or } \hat{\underline{\sigma}} = \underline{\sigma}$	
ev_sd_surface_ltr <i>SDLinearTractio</i>	<opt_mater <parameter> <parameter>	$\int_{\Gamma} \underline{v} \cdot (\underline{\sigma} \underline{n}), \int_{\Gamma} \underline{v} \cdot \underline{n},$	

continues on next page

Table 6 – continued from previous page

name/class	argu- ments	definition	examples
de_sd_v_dot_grad ESDVectorDotGrad	<opt_mater <virtual/ param_v>, <state/ param_s>, <parameter <opt_mater <state>, <virtual>, <parameter	$\int_{\Omega} \hat{I}_{ij} \frac{\partial p}{\partial x_j} v_i, \int_{\Omega} \hat{I}_{ij} \frac{\partial q}{\partial x_j} u_i$ $\hat{I}_{ij} = \delta_{ij} \nabla \cdot \mathcal{V} - \frac{\partial \mathcal{V}_j}{\partial x_i}$	

Table of large deformation terms

Table 7: Large deformation terms

name/class	arguments	definition	examples
dw_tl_bulk_active <i>BulkActiveTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_bulk_penalty <i>BulkPenaltyTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	<i>hyp, act.fib,</i> <i>com.ela.mat</i>
dw_tl_bulk_pressure <i>BulkPressureTLTerm</i>	<virtual>, <state>, <state_p>	$\int_{\Omega} S_{ij}(p) \delta E_{ij}(\underline{u}; \underline{v})$	<i>bal, per.tl</i>
dw_tl_diffusion <i>DiffusionTLTerm</i>	<material> <material> <virtual>, <state>, <parameter>	$\int_{\Omega} \underline{K}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial \underline{X}} \frac{\partial p}{\partial \underline{X}}$	<i>per.tl</i>
dw_tl_fib_a <i>FibresActiveTLTerm</i>	<material> <material> <material> <material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	<i>act.fib</i>
dw_tl_fib_e <i>FibresExponentialTLTerm</i>	<material> <material> <material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_he_genyeoh <i>GenYeohTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_he_mooney <i>MooneyRivlinTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	<i>hyp,</i> <i>com.ela.mat,</i> <i>bal</i>

continues on next page

Table 7 – continued from previous page

name/class	arguments	definition	examples
dw_tl_he_neohook <i>NeoHookeanTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	<i>per.tl</i> , <i>hyp</i> , <i>act.fib</i> , <i>com.ela.mat</i> , <i>bal</i>
dw_tl_he_neohook <i>NeoHookeanTLADTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_he_ogden <i>OgdenTLTerm</i>	<material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_he_ogden_a <i>OgdenTLADTerm</i>	<material> <material> <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$	
dw_tl_membrane <i>TLMembraneTerm</i>	<material> <material> <material> <virtual>, <state>		<i>bal</i>
ev_tl_surface_flux <i>SurfaceFluxTLTerm</i>	<material> <material> <parameter> <parameter>	$\int_{\Gamma} \underline{\nu} \cdot \underline{K}(\underline{u}^{(n-1)}) \frac{\partial p}{\partial \underline{X}}$	
dw_tl_surface_trac <i>SurfaceTractionTerm</i>	<opt_mater> <virtual>, <state>	$\int_{\Gamma} \underline{\nu} \cdot \underline{F}^{-1} \cdot \underline{\sigma} \cdot \underline{v} J$	<i>per.tl</i>
dw_tl_volume <i>VolumeTLTerm</i>	<virtual>, <state>	$\int_{\Omega} q J(\underline{u})$ volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$ rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$	<i>bal</i> , <i>per.tl</i>
ev_tl_volume_surf <i>VolumeSurfaceTerm</i>	<parameter>	$1/D \int_{\Gamma} \underline{\nu} \cdot \underline{F}^{-1} \cdot \underline{x} J$	

continues on next page

Table 7 – continued from previous page

name/class	arguments	definition	examples
dw_ul_bulk_penalty <i>BulkPenaltyULTerms</i>	<material> <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$	<i>hyp.ul</i>
dw_ul_bulk_pressure <i>BulkPressureULTerms</i>	<virtual>, <state>, <state_p>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$	<i>hyp.ul.up</i>
dw_ul_compressibility <i>CompressibilityTerm</i>	<material> <virtual>, <state>, <parameter>	$\frac{\int_{\Omega} 1}{\gamma p q}$	<i>hyp.ul.up</i>
dw_ul_he_by_fun <i>HyperelasticByFunction</i>	<fun>, <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$	<i>hyp.ul.by.fun</i>
dw_ul_he_mooney <i>MooneyRivlinULTerms</i>	<material> <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$	<i>hyp.ul, hyp.ul.up</i>
dw_ul_he_neohook <i>NeoHookeanULTerms</i>	<material> <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$	<i>hyp.ul, hyp.ul.up</i>
dw_ul_volume <i>VolumeULTerm</i>	<virtual>, <state>	$\int_{\Omega} qJ(\underline{u})$ volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$ rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$	<i>hyp.ul.up</i>

Table of special terms

Table 8: Special terms

name/class	arguments	definition	examples
dw_biot_eth <i>BiotETHTerm</i>	<ts>, <material_ <material_ <virtual>, <state> <ts>, <material_ <material_ <state>, <virtual>	$\int_{\Omega} \left[\int_0^t \alpha_{ij}(t-\tau) p(\tau) d\tau \right] e_{ij}(\underline{v}) ,$ $\int_{\Omega} \left[\int_0^t \alpha_{ij}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$	
dw_biot_th <i>BiotTHTerm</i>	<ts>, <material> <virtual>, <state> <ts>, <material> <state>, <virtual>	$\int_{\Omega} \left[\int_0^t \alpha_{ij}(t-\tau) p(\tau) d\tau \right] e_{ij}(\underline{v}) ,$ $\int_{\Omega} \left[\int_0^t \alpha_{ij}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$	
ev_cauchy_stress_e <i>CauchyStressETH</i>	<ts>, <material_ <material_ <parameter	$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau$	
ev_cauchy_stress_t <i>CauchyStressTH</i>	<ts>, <material> <parameter	$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau$	
dw_lin_elastic_eth <i>LinearElasticETH</i>	<ts>, <material_ <material_ <virtual>, <state>	$\int_{\Omega} \left[\int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$	<i>lin.vis</i>
dw_lin_elastic_th <i>LinearElasticTH</i>	<ts>, <material> <virtual>, <state>	$\int_{\Omega} \left[\int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$	
ev_of_ns_surf_min <i>NSOFSurfMinDP</i>	<material_ <material_ <parameter	$\delta \Psi(p) = \delta \left(\int_{\Gamma_{in}} p - \int_{\Gamma_{out}} b_{press} \right)$	

continues on next page

Table 8 – continued from previous page

name/class	arguments	definition	examples
dw_of_ns_surf_min <i>NSOFSurfMinDP</i>	<material> <virtual>	$w\delta_p\Psi(p) \circ q$	
ev_sd_st_grad_div <i>SDGradDivStabiliz</i>	<material> <parameter> <parameter> <parameter>	$\gamma \int_{\Omega} [(\nabla \cdot \underline{u})(\nabla \cdot \underline{w})(\nabla \cdot \underline{v}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot \underline{w}) - (\nabla \cdot \underline{u}) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i}]$	
ev_sd_st_pspg_c <i>SDPSPGCStabiliz</i>	<material> <parameter> <parameter> <parameter> <parameter>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \left[\frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i) (\nabla \cdot \underline{v}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k) \right]$	
ev_sd_st_pspg_p <i>SDPSPGPStabiliz</i>	<material> <parameter> <parameter> <parameter>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K [(\nabla r \cdot \nabla p)(\nabla \cdot \underline{v}) - \frac{\partial r}{\partial x_k} (\nabla \mathcal{V}_k \cdot \nabla p) - (\nabla r \cdot \nabla \mathcal{V}_k) \frac{\partial p}{\partial x_k}]$	
ev_sd_st_supg_c <i>SDSUPGCStabiliz</i>	<material> <parameter> <parameter> <parameter> <parameter>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [(\underline{b} \cdot \nabla u_k)(\underline{b} \cdot \nabla w_k)(\nabla \cdot \underline{v}) - (\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial u_k}{\partial x_i} (\underline{b} \cdot \nabla w_k) - \frac{\partial w_k}{\partial x_i}]$	
dw_st_adj1_supg <i>SUPGPAdj1Stabiliz</i>	<material> <virtual>, <state>, <parameter>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p(\underline{v} \cdot \nabla \underline{w})$	
dw_st_adj2_supg <i>SUPGPAdj2Stabiliz</i>	<material> <virtual>, <parameter> <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla r(\underline{v} \cdot \nabla \underline{u})$	
dw_st_adj_supg_c <i>SUPGCAdjStabiliz</i>	<material> <virtual>, <parameter> <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [((\underline{v} \cdot \nabla) \underline{u})((\underline{u} \cdot \nabla) \underline{w}) + ((\underline{u} \cdot \nabla) \underline{u})((\underline{v} \cdot \nabla) \underline{w})]$	
dw_st_grad_div <i>GradDivStabiliz</i>	<material> <virtual>, <state>	$\gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v})$	<i>sta.nav.sto</i>

continues on next page

Table 8 – continued from previous page

name/class	arguments	definition	examples
dw_st_pspg_c <i>PSPGCStabilization</i>	<material> <virtual>, <parameter> <state>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$	<i>sta.nav.sto</i>
dw_st_pspg_p <i>PSPGPStabilization</i>	<opt_material> <virtual/ param_1>, <state/ param_2>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q$	<i>sta.nav.sto</i>
dw_st_supg_c <i>SUPGCStabilization</i>	<material> <virtual>, <parameter> <state>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v})$	<i>sta.nav.sto</i>
dw_st_supg_p <i>SUPGPStabilization</i>	<material> <virtual>, <parameter> <state>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v})$	<i>sta.nav.sto</i>
dw_volume_dot_w <i>DotSPProductVolume</i>	<ts>, <material_1> <material_2> <virtual>, <state>	$\int_{\Omega} \left[\int_0^t \mathcal{G}(t - \tau) p(\tau) \, d\tau \right] q$	
dw_volume_dot_w <i>DotSPProductVolume</i>	<ts>, <material> <virtual>, <state>	$\int_{\Omega} \left[\int_0^t \mathcal{G}(t - \tau) p(\tau) \, d\tau \right] q$	

Table of multi-linear terms

Table 9: Multi-linear terms

name/class	arguments	definition	examples
de_cauchy_stress <i>ECauchyStressTerm</i>	<material> <parameter>	$\int_{\Omega} D_{ijkl} e_{kl}(\underline{u})$	
de_convect <i>EConvectTerm</i>	<opt_material> <virtual/param_1>, <state/param_2>	$\int_{\Omega} c((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$	
de_diffusion <i>EDiffusionTerm</i>	<material> <virtual/param_1>, <state/param_2>	$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p$	
de_div <i>EDivTerm</i>	<opt_material> <virtual/param>	$\int_{\Omega} \nabla \cdot \underline{v}, \int_{\Omega} c \nabla \cdot \underline{v}$	
de_div_grad <i>EDivGradTerm</i>	<opt_material> <virtual/param_1>, <state/param_2>	$\int_{\Omega} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}$	
de_dot <i>EDotTerm</i>	<opt_material> <virtual/param_1>, <state/param_2>	$\begin{aligned} \int_{\mathcal{D}} qp, \int_{\mathcal{D}} \underline{v} \cdot \underline{u} \\ \int_{\mathcal{D}} cqp, \int_{\mathcal{D}} c\underline{v} \cdot \underline{u} \\ \int_{\mathcal{D}} \underline{v} \cdot (\underline{c} \underline{u}) \end{aligned}$	
de_grad <i>EGradTerm</i>	<opt_material> <parameter>	$\int_{\Omega} \nabla \underline{v}, \int_{\Omega} c \nabla \underline{v}, \int_{\Omega} \underline{c} \cdot \nabla \underline{v}, \int_{\Omega} \underline{c} \cdot \nabla \underline{v}$	
de_integrate <i>EIntegrateOperator</i>	<opt_material> <virtual>	$\int_{\mathcal{D}} q \text{ or } \int_{\mathcal{D}} cq$	

continues on next page

Table 9 – continued from previous page

name/class	arguments	definition	examples
de_laplace <i>ELaplaceTerm</i>	<opt_mater <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} \nabla q \cdot \nabla p, \int_{\Omega} c \nabla q \cdot \nabla p$	
de_lin_convect <i>ELinearConvect</i>	<virtual/ param_1>, <parameter <state/ param_3>	$\int_{\Omega} ((\underline{w} \cdot \nabla) \underline{u}) \cdot \underline{v}$	
de_lin_elastic <i>ELinearElastic</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$	
de_m_flexo <i>MixedFlexoTerm</i>	<virtual/ param_v>, <state/ param_t> <state>, <virtual>	$\int_{\Omega} v_{i,j} a_{ij}$ $\int_{\Omega} u_{i,j} \delta a_{ij}$	
de_m_flexo_coupli <i>MixedFlexoCoupli</i>	<material> <virtual/ param_t>, <state/ param_s> <material> <state>, <virtual>	$\int_{\Omega} f_{ijkl} e_{jk,l}(\underline{\delta w}) \nabla_i p$ $\int_{\Omega} f_{ijkl} e_{jk,l}(\underline{w}) \nabla_i q$	
de_m_sg_elastic <i>MixedStrainGrad</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} a_{ijklmn} e_{ij,k}(\underline{\delta w}) e_{lm,n}(\underline{w})$	
de_mass <i>MassTerm</i>	<material> <material> <material> <virtual>, <state>	$M^C = \int_{\mathcal{D}} \rho \underline{v} \cdot \underline{u}$ $M^L = \text{lumping}(M^C)$ $M^A = (1 - \beta) M^C + \beta M^L$ $A = \sum_e A_e$ $C = \sum_e A_e^T (M_e^A)^{-1} A_e$	<i>ela, sei.loa</i>

continues on next page

Table 9 – continued from previous page

name/class	arguments	definition	examples
de_non_penetration <i>ENonPenetration</i>	<material> <virtual>, <state>	$\int_{\Gamma} c(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u})$	
de_nonsym_elastic <i>ENonSymElastic</i>	<material> <virtual/ param_1>, <state/ param_2>	$\int_{\Omega} \underline{\underline{D}} \nabla \underline{v} : \nabla \underline{u}$	
de_s_dot_mgrad_s <i>EScalarDotMGrad</i>	<material> <virtual/ param_1>, <state/ param_2> <material> <state>, <virtual>	$\int_{\Omega} q \underline{y} \cdot \nabla p, \int_{\Omega} p \underline{y} \cdot \nabla q$	
de_stokes <i>EStokesTerm</i>	<opt_mater <virtual/ param_v>, <state/ param_s> <opt_mater <state>, <virtual>	$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u}$ $\int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$	
de_stokes_traction <i>StokesTraction</i>	<opt_mater <virtual/ param_1>, <state/ param_2>	$\int_{\Gamma} \nu \underline{v} \cdot (\nabla \underline{u} \cdot \underline{n})$	
de_surface_flux <i>SurfaceFluxOpen</i>	<material> <virtual/ param_1>, <state/ param_2> <material> <state>, <virtual>	$\int_{\Gamma} q \underline{n} \cdot \underline{\underline{K}} \cdot \nabla p, \int_{\Gamma} p \underline{n} \cdot \underline{\underline{K}} \cdot \nabla q$	<i>pie.ela</i>
de_surface_ltr <i>ELinearTraction</i>	<opt_mater <virtual/ param>	$\int_{\Gamma} \underline{v} \cdot \underline{n}, \int_{\Gamma} c \underline{v} \cdot \underline{n}$ $\int_{\Gamma} \underline{v} \cdot (\underline{\underline{\sigma}} \underline{n}), \int_{\Gamma} \underline{v} \cdot \underline{f}$	

continues on next page

Table 9 – continued from previous page

name/class	argu- ments	definition	examples
de_surface_piezo_ <i>SurfacePiezoFlu</i>	<material> <virtual/ param_1>, <state/ param_2> <material> <state>, <virtual>	$\int_{\Gamma} q g_{kij} e_{ij}(\underline{u}) n_k, \int_{\Gamma} p g_{kij} e_{ij}(\underline{v}) n_k$	

DEVELOPMENT

The SfePy development takes place in the [sfepy/sfepy](#) repository on Github. The users and developers can also communicate using the [mailing list](#).

2.1 General Information

We are interested in any contribution. There are many ways how you can contribute:

- You can report bugs using [our mailing list](#). You can also add a bug report (or a comment) into the [issues](#).
- You can contribute interesting examples/tutorials.
- You can blog about how you use SfePy (let us know!).
- You can help with improving our documentation and these pages.
- ...

To get acquainted with SfePy, you can start by reading the [Tutorial](#) and [Primer](#) sections of the documentation and trying out the examples that come with the sources. Your first contribution could be pinpointing anything that is not clear in the docs.

We also recommend reading the [How to Contribute](#) section of our [Developer Guide](#).

2.2 Possible Topics

Several specific topics that we wish to address in the future are listed below. If you would like to contribute code/advice to our project with respect to these topics, do not hesitate to contact us (either directly: cimrman3@ntc.zcu.cz, or on [our mailing list](#))

- finish/improve IGA implementation (see [Isogeometric Analysis](#)):
 - support multiple patches
 - efficient quadrature formulas
 - local refinement?
- discretization methods:
 - implement vector elements (Nedelec, Raviart-Thomas, ...)
 - implement the discontinuous Galerkin method
- material models: plasticity, viscoplasticity, damage, ...
- improve parallelization (see [Solving Problems in Parallel](#)):

- cluster installation with fast BLAS
 - parallel code speed-up
 - remove (some of) the serial parts
 - preconditioning for multi-physics problems
- solvers:
 - better defaults/recommendations for iterative solvers (**PETSc**) with respect to large problems
 - dynamics/time-stepping solvers, interface PETSc time-steppers
 - interface more sparse linear solvers (or enable via PETSc), for example **BDDCML**
 - interface more eigenvalue problem solvers
- visualization of large data
- automatic differentiation:
 - for tangent matrices
 - for identification of (material) parameters
- core data structures & programming:
 - using octree-based(?) mesh representation for local refinement
 - continue with/improve the current hanging nodes implementation
 - exploit lazy evaluation

See also the [enhancement issues](#).

2.3 Developer Guide

This section purports to document the *SfePy* internals. It is mainly useful for those who wish to contribute to the development of *SfePy* and understand the inner workings of the code.

We use [git](#) to track source code, documentation, examples, and other files related to the project.

It is not necessary to learn git in order to contribute to *SfePy* but we strongly suggest you do so as soon as possible - it is an extremely useful tool not just for writing code, but also for tracking revisions of articles, Ph.D. theses, books, ... it will also look well in your CV :-). It is also much easier for us to integrate changes that are in form of a github pull request than in another form.

2.3.1 Retrieving the Latest Code

The first step is to obtain the latest development version of the code from the [SfePy git repository](#):

```
git clone git://github.com/sfepy/sfepy.git
```

For development, it is preferable to build the extension modules in place (see *Compilation of C Extension Modules*):

```
python setup.py build_ext --inplace
```

On Unix-like systems, you can simply type `make` in the top-level folder to build in-place.

After the initial compilation, or after making changes, do not forget to run the tests, see *Testing Installation*.

2.3.2 SfePy Directory Structure

Here we list and describe the directories that are in the main sfepy directory.

Table 1: Top directory structure.

name	description
<i>build/</i>	directory created by the build process (generated)
<i>doc/</i>	source files of this documentation
<i>meshes/</i>	finite element mesh files in various formats shared by the examples
<i>output/</i>	default output directory for storing results of the examples
<i>sfepy/</i>	the source code including examples and tests
<i>tools/</i>	various helper scripts (build, documentation generation etc.)

New users/developers (after going through the [Tutorial](#)) should explore the *sfepy/examples/* directory. For developers, the principal directory is *sfepy/*, which has the following contents:

Table 2: *sfepy/* directory structure.

name	description	field-specific
<i>applications/</i>	top level application classes (e.g. PDESolverApp)	
<i>base/</i>	common utilities and classes used by most of the other modules	
<i>discrete/</i>	general classes and modules for describing a discrete problem, taking care of boundary conditions, degrees of freedom, approximations, variables, equations, meshes, regions, quadratures, etc. Discretization-specific classes are in subdirectories: <i>common/</i> - common parent classes for discretization-specific classes <i>fem/</i> - finite element specific classes <i>iga/</i> - isogeometric analysis specific classes	
<i>examples/</i>	the examples using both the declarative and imperative problem description API	
<i>homogenization/</i>	the homogenization engine and supporting modules - highly specialized code, one of the reasons of <i>SfePy</i> existence	•
<i>linalg/</i>	linear algebra functions not covered by NumPy and SciPy	
<i>mechanics/</i>	modules for (continuum) mechanics: elastic constant conversions, tensor, units utilities, etc.	•
<i>mesh/</i>	some utilities to interface with tetgen and triangle mesh generators	
<i>parallel/</i>	modules supporting parallel assembling and solution of problems	
<i>postprocess/</i>	Matplotlib and VTK based post-processing modules	
<i>scripts/</i>	the main as well as auxiliary scripts	
<i>solvers/</i>	interface classes to various internal/external solvers (linear, nonlinear, eigenvalue, optimization, time stepping)	
<i>terms/</i>	implementation of the terms (weak formulation integrals), see Term Overview	
<i>tests/</i>	the tests	

The directories in the “field-specific” column are mostly interesting for specialists working in the respective fields.

The *discrete/* is the heart of the code, while the *terms/* contains the particular integral forms usable to build equations - new term writers should look there.

2.3.3 Exploring the Code

It is convenient to install IPython to have the tab completion available. Moreover, all SfePy classes can be easily examined by printing them:

```
1 In [1]: from sfepy.discrete.fem import Mesh
2
3 In [2]: mesh = Mesh.from_file('meshes/2d/rectangle_tri.mesh')
4 sfepy: reading mesh [line2, tri3, quad4, tetra4, hexa8] (meshes/2d/rectangle_tri.mesh)...
5 sfepy: ...done in 0.00 s
6
7 In [3]: print(mesh)
8 Mesh:meshes/2d/rectangle_tri
9   cmesh:
10     CMesh: n_coor: 258, dim 2, tdim: 2, n_el 454
11   desc:
12     list: ['2_3']
13   dim:
14     2
15   dims:
16     list: [2]
17   io:
18     None
19   n_el:
20     454
21   n_nod:
22     258
23   name:
24     meshes/2d/rectangle_tri
25   nodal_bcs:
26     dict with keys: []
```

We recommend going through the interactive example in the tutorial *Interactive Example: Linear Elasticity* in this way, printing all the variables.

Another useful tool is the `debug()` function, that can be used as follows:

```
from sfepy.base.base import debug; debug()
```

Try to use it in the examples with user defined functions to explore their parameters etc. It works best with IPython installed, as then the tab completion is available also when debugging.

2.3.4 How to Contribute

Read this section if you wish to contribute some work to the *SfePy* project - everyone is welcome to contribute. Contributions can be made in a variety of forms, not just code. Reporting bugs and contributing to the documentation, tutorials, and examples is in great need!

Below we describe

1. where to report problems or find existing issues and additional development suggestions
2. what to do to apply changes/fixes
3. what to do after you made your changes/fixes

Reporting problems

Reporting a bug is the first way in which to contribute to an open source project

Short version: go to the main [SfePy](#) site and follow the links given there.

When you encounter a problem, try searching that site first - an answer may already be posted in the [SfePy mailing list](#) (to which we suggest you subscribe...), or the problem might have been added to the [SfePy issues](#). As is true in any open source project, doing your homework by searching for existing known problems greatly reduces the burden on the developers by eliminating duplicate issues. If you find your problem already exists in the issue tracker, feel free to gather more information and append it to the issue. In case the problem is not there, create a new issue with proper labels for the issue type and priority, and/or ask us using the mailing list.

Note: A google account (e.g., gmail account) is needed to join the mailing list. A github account is needed for working with the source code repository and issues.

Note: When reporting a problem, try to provide as much information as possible concerning the version of *SfePy*, the OS / Linux distribution, and the versions of *Python*, *NumPy* and *SciPy*, and other prerequisites. The versions found on your system can be printed by running:

```
python setup.py --help
```

If you are a new user, please let us know what difficulties you have with this documentation. We greatly welcome a variety of contributions not limited to code only.

Contributing changes

Note: To avoid duplicating work, it is highly advised that you contact the developers on the mailing list or create an enhancement issue before starting work on a non-trivial feature.

Before making any changes, read the [Notes on commits and patches](#).

Using git and github

The preferred way to contribute to *SfePy* is to fork the main repository on github, then submit a “pull request” (PR):

1. [Create a github account](#) if you do not already have one.
2. Fork the project repository: click on the “Fork” button near the top of the [sfepy git repository](#) page. This creates a copy of the repository under your account on the github server.
3. Clone your fork to your computer:

```
git clone git@github.com:YourLogin/sfepy.git
```

4. If you have never used git before, introduce yourself to git and make (optionally) some handy aliases either in `.gitconfig` in your home directory (global settings for all your git projects), or directly in `.git/config` in the repository:

```
1 [user]
2     email = mail@mail.org
3     name = Name Surname
4
5 [color]
6     ui = auto
7     interactive = true
8
9 [alias]
10    ci = commit
11    di = diff --color-words
12    st = status
13    co = checkout
```

5. Create a feature branch to hold your changes:

```
git checkout -b my-feature
```

Then you can start to make your changes. Do not work in the master branch!

6. Modify some files and use git to track your local changes. The changed added/modified files can be listed using:

```
git status
```

and the changes can be reviewed using:

```
git diff
```

A more convenient way of achieving the above is to run:

```
gitk --all
```

in order to visualize of project history (all branches). There are other GUIs for this purpose, e.g. `qgit`. You may need to install those tools, as they usually are not installed with git by default. Record a set of changes by:

```
1 # schedule some of the changed files for the next commit
2 git add file1 file2 ...
3 # an editor will pop up where you should describe the commit
4 git commit
```

We recommend `git gui` command in case you want to add and commit only some changes in a modified file.

Note: Do not be afraid to experiment - git works with your *local* copy of the repository, so it is not possible to damage the master repository. It is always possible to re-clone a fresh copy, in case you do something that is really bad.

7. The commit(s) now reflect changes, but only in your *local* git repository. To update your github repository with your new commit(s), run:

```
git push origin my-feature:my-feature
```

- Finally, when your feature is ready, and all tests pass, go to the github page of your sfepy repository fork, and click “Pull request” to send your changes to the maintainers for review. Continuous integration (CI) will run and check that the changes pass tests on Windows, Linux and Mac using Github Actions. The results will be displayed in the Pull Request discussion. The CI setup is located in the file `.github/workflows/build_and_test_matrix.yml`. It is recommended to check that your contribution complies with the [Notes on commits and patches](#).

In the above setup, your origin remote repository points to `YourLogin/sfepy.git`. If you wish to fetch/merge from the main repository instead of your forked one, you will need to add another remote to use instead of origin. The main repository is usually called “upstream”. To add it, type:

```
git remote add upstream https://github.com/sfepy/sfepy.git
```

To synchronize your repository with the upstream, proceed as follows:

- Fetch the upstream changes:

```
git fetch upstream
```

Never start with `git pull upstream`!

- Check the changes of the upstream master branch. You can use `gitk --all` to visualize all your and remote branches. The upstream master is named `remotes/upstream/master`.
- Make sure all your local changes are either committed in a feature branch or stashed (see `git stash`). Then reset your master to the upstream master:

```
git checkout master
git reset --hard upstream/master
```

Warning The above will remove all your local commits in the master branch that are not in `upstream/master`, and also reset all the changes in your non-committed modified files!

Optionally, the reset command can be run conveniently in `gitk` by right-clicking on a commit you want to reset the current branch onto.

- Optionally, rebase your feature branch onto the upstream master:

```
git checkout my-feature
git rebase upstream/master
```

This is useful, for example, when the upstream master contains a change you need in your feature branch.

For additional information, see, for example, the [gitwash](#) git tutorial, or its incarnation [NumPy gitwash](#).

Notes on commits and patches

- Follow our [Coding style](#).
- Do not use lines longer than 79 characters (exception: tables of values, e.g., quadratures).
- Write descriptive docstrings in correct style, see [Docstring standard](#).
- There should be one patch for one topic - do not mix unrelated things in one patch. For example, when you add a new function, then notice a typo in docstring in a nearby function and correct it, create two patches: one fixing the docstring, the other adding the new function.

- The commit message and description should clearly state what the patch does. Try to follow the style of other commit messages. Some interesting notes can be found at tbaggery.com, namely that the commit message is better to be written in the present tense: “fix bug” and not “fixed bug”.

Without using git

Without using git, send the modified files to the [SfePy mailing list](#) or attach them using [gist](#) to the corresponding issue at the [Issues](#) web page. Do not forget to describe the changes properly, and to follow the spirit of *Notes on commits and patches* and the *Coding style*.

Coding style

All the code in SfePy should try to adhere to python style guidelines, see [PEP-0008](#).

There are some additional recommendations:

- Prefer whole words to abbreviations in public APIs - there is completion after all. If some abbreviation is needed (*really* too long name), try to make it as comprehensible as possible. Also check the code for similar names - try to name things consistently with the existing code. Examples:
 - yes: `equation`, `transform_variables()`, `filename`
 - rather not: `eq`, `transvar()`, `fname`
- Functions have usually form `<action>_<subject>()` e.g.: `save_data()`, `transform_variables()`, do not use `data_save()`, `variable_transform()` etc.
- Variables like `V`, `c`, `A`, `b`, `x` should be tolerated only locally when expressing mathematical ideas.

Really minor recommendations:

- Avoid single letter names, if you can:
 - not even for loop variables - use e.g. `ir`, `ic`, ... instead of `i`, `j` for rows and columns
 - not even in generators, as they “leak” (this is fixed in Python 3.x)

These are recommendations only, we will not refuse code just on the ground that it uses slightly different formatting, as long as it follows the PEP.

Note: some old parts of the code might not follow the PEP, yet. We fix them progressively as we update the code.

Docstring standard

We use [sphinx](#) with the [numpydoc](#) extension to generate this documentation. Refer to the sphinx site for the possible markup constructs.

Basically (with a little tweak), we try to follow the NumPy/SciPy docstring standard as described in [NumPy documentation guide](#). See also the complete [docstring example](#). It is exaggerated a bit to show all the possibilities. Use your common sense here - the docstring should be sufficient for a new user to use the documented object. A good way to remember the format is to type:

```
In [1]: import numpy as nm
In [2]: nm.sin?
```

in *ipython*. The little tweak mentioned above is the starting newline:

```

1 def function(arg1, arg2):
2     """
3     This is a function.
4
5     Parameters
6     -----
7     arg1 : array
8         The coordinates of ...
9     arg2 : int
10        The dimension ...
11
12    Returns
13    -----
14    out : array
15        The resulting array of shape ....
16    """

```

It seems visually better than:

```

1 def function(arg1, arg2):
2     """This is a function.
3
4     Parameters
5     -----
6     arg1 : array
7         The coordinates of ...
8     arg2 : int
9         The dimension ...
10
11    Returns
12    -----
13    out : array
14        The resulting array of shape ....
15    """

```

When using $\LaTeX$ in a docstring, use a raw string:

```

1 def function():
2     r"""
3     This is a function with :math:\mbox{\LaTeX} math:
4     :math:\frac{1}{\pi}.
5     """

```

to prevent Python from interpreting and consuming the backslashes in common escape sequences like `'\n'`, `'\f'` etc.

2.3.5 How to Regenerate Documentation

The following steps summarize how to regenerate this documentation.

1. Install `sphinx`, `numpydoc`, `'sphinx-rtd-theme'` and `IPython`. Do not forget to set the path to `numpydoc` in `site_cfg.py` if it is not installed in a standard location for Python packages on your platform. A recent L^AT_EX distribution is required, too, for example [TeX Live](#). Depending on your OS/platform, it can be in the form of one or several packages.
2. Edit the `.rst` files in `doc/` directory using your favorite text editor - the ReST format is really simple, so nothing fancy is needed. Follow the existing files in `doc/`; for reference also check [reStructuredText Primer](#), [Sphinx Markup Constructs](#) and [docutils reStructuredText](#).
 - When adding a new Python module, add a corresponding documentation file into `doc/src/sfepy/<path>`, where `<path>` should reflect the location of the module in `sfepy/`.
 - Figures belong to `doc/images`; subdirectories can be used.
3. (Re)generate the documentation (assuming GNU make is installed):

```
cd doc
make html
```

4. View it (substitute your favorite browser):

```
firefox _build/html/index.html
```

2.3.6 How to Implement a New Term

Warning Implementing a new term usually involves C. As Cython is now supported by our build system, it should not be that difficult. Python-only terms are possible as well.

Note There is an experimental way (newly from version 2021.1) of implementing multi-linear terms that is much easier than what is described here, see [Multi-linear Terms](#).

Notes on terminology

Term integrals are over domains of the *cell* or *facet* kinds. For meshes with elements of the topological dimension $t \leq d$, where d is the space dimension, *cells* have the topological t , while facets $t - 1$. For example, in 3D meshes cell = volume, facet = surface, while in 2D cell = area, facet = curve.

Introduction

A term in *SfePy* usually corresponds to a single integral term in (weak) integral formulation of an equation. Both cell and facet integrals are supported. There are three types of arguments a term can have:

- *variables*, i.e. the unknown, test or parameter variables declared by the *variables* keyword, see `sec-problem-description-file`,
- *materials*, corresponding to material and other parameters (functions) that are known, declared by the *materials* keyword,
- *user data* - anything, but user is responsible for passing them to the evaluation functions.

SfePy terms are subclasses of `sfe.py.terms.terms.Term`. The purpose of a term is to implement a (vectorized) function that evaluates the term contribution to residual/matrix and/or evaluates the term integral in elements of the term region. Many such functions are currently implemented in C, but some terms are pure Python, vectorized using NumPy.

Evaluation modes

A term can support several evaluation modes, as described in *Term Evaluation*.

Basic attributes

A term class should inherit from `sfe.py.terms.terms.Term` base class. The simplest possible term with cell integration and ‘weak’ evaluation mode needs to have the following attributes and methods:

- `docstring` (not really required per se, but we require it);
- `name` attribute - the name to be used in *equations*;
- `arg_types` attribute - the types of arguments the term accepts;
- `integration` attribute, optional - the kind of integral the term implements, one of ‘*cell*’ (the default, if not given), ‘*facet*’ or ‘*facet_extra*’;
- `function()` static method - the assembling function;
- `get_fargs()` method - the method that takes term arguments and converts them to arguments for `function()`.

Argument types

The argument types can be (“[_*]” denotes an optional suffix):

- ‘*material*[_*]’ for a material parameter, i.e. any function that can be evaluated in quadrature points and that is not a variable;
- ‘*opt_material*[_*]’ for an optional material parameter, that can be left out - there can be only one in a term and it must be the first argument;
- ‘*virtual*’ for a virtual (test) variable (no value defined), ‘*weak*’ evaluation mode;
- ‘*state*[_*]’ for state (unknown) variables (have value), ‘*weak*’ evaluation mode;
- ‘*parameter*[_*]’ for parameter variables (have known value), any evaluation mode.

Only one ‘*virtual*’ variable is allowed in a term.

Integration kinds

The integration kinds have the following meaning:

- ‘*cell*’ for cell integral over a region that contains elements; uses cell connectivity for assembling;
- ‘*facet*’ for facet integral over a region that contains faces; uses facet connectivity for assembling;
- ‘*facet_extra*’ for facet integral over a region that contains faces; uses cell connectivity for assembling - this is needed if full gradients of a variable are required on the boundary.

function()

The *function()* static method has always the following arguments:

```
out, *args
```

where *out* is the already preallocated output array (change it in place!) and **args* are any other arguments the function requires. These function arguments have to be provided by the *get_fargs()* method. The function returns zero *status* on success, nonzero on failure.

The *out* array has shape $(n\_el, 1, n\_row, n\_col)$, where *n_el* is the number of elements and *n_row*, *n_col* are matrix dimensions of the value on a single element.

get_fargs()

The *get_fargs()* method has always the same structure of arguments:

- positional arguments corresponding to *arg_types* attribute:

- example for a typical weak term:

- * for:

```
arg_types = ('material', 'virtual', 'state')
```

the positional arguments are:

```
material, virtual, state
```

- keyword arguments common to all terms:

```
mode=None, term_mode=None, diff_var=None, **kwargs
```

here:

- *mode* is the actual evaluation mode, default is 'eval';
 - *term_mode* is an optional term sub-mode influencing what the term should return (example: *dw_tl_he_neohook* term has 'strain' and 'stress' evaluation sub-modes);
 - *diff_var* is taken into account in the 'weak' evaluation mode. It is either *None* (residual mode) or a name of variable with respect to differentiate to (matrix mode);
 - ***kwargs* are any other arguments that the term supports.

The *get_fargs()* method returns arguments for *function()*.

Additional attributes

These attributes are used mostly in connection with the *tests/test_term_call_modes.py* test for automatic testing of term calls.

- *arg_shapes* attribute - the possible shapes of term arguments;
- *geometries* attribute - the list of reference element geometries that the term supports;
- *mode* attribute - the default evaluation mode.

Argument shapes

The argument shapes are specified using a dict of the following form:

```
arg_shapes = {'material' : 'D, D', 'virtual' : (1, 'state'),
              'state' : 1, 'parameter_1' : 1, 'parameter_2' : 1}
```

The keys are the argument types listed in the `arg_types` attribute, for example:

```
arg_types = (('material', 'virtual', 'state'),
             ('material', 'parameter_1', 'parameter_2'))
```

The values are the shapes containing either integers, or ‘D’ (for space dimension) or ‘S’ (symmetric storage size corresponding to the space dimension). For materials, the shape is a string ‘*nr, nc*’ or a single value, denoting a special-valued term, or *None* denoting an optional material that is left out. For state and parameter variables, the shape is a single value. For virtual variables, the shape is a tuple of a single shape value and a name of the corresponding state variable; the name can be *None*.

When several alternatives are possible, a list of dicts can be used. For convenience, only the shapes of arguments that change w.r.t. a previous dict need to be included, as the values of the other shapes are taken from the previous dict. For example, the following corresponds to a case, where an optional material has either the shape (1, 1) in each point, or is left out:

```
1 arg_types = ('opt_material', 'parameter')
2 arg_shapes = [{'opt_material' : '1, 1', 'parameter' : 1},
3               {'opt_material' : None}]
```

Geometries

The default that most terms use is a list of all the geometries:

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

In that case, the attribute needs not to be define explicitly.

Examples

Let us now discuss the implementation of a simple weak term *dw_integrate* defined as $\int_D cq$, where *c* is a weight (material parameter) and *q* is a virtual variable. This term is implemented as follows:

```
1 class IntegrateOperatorTerm(Term):
2     r"""
3     Integral of a test function weighted by a scalar function
4     :math:`c`.
5
6     :Definition:
7
8     .. math::
9         \int_{\mathcal{D}} q \, \mathit{or} \, \int_{\mathcal{D}} c \, q
10
11     :Arguments:
12         - material : :math:`c` (optional)
```

(continues on next page)

(continued from previous page)

```

13     - virtual : :math:`q`
14     """
15     name = 'dw_integrate'
16     arg_types = ('opt_material', 'virtual')
17     arg_shapes = [{'opt_material' : '1, 1', 'virtual' : (1, None)},
18                   {'opt_material' : None}]
19     integration = ('cell', 'facet')
20
21     @staticmethod
22     def function(out, material, bf, geo):
23         bf_t = nm.tile(bf.transpose((0, 1, 3, 2)), (out.shape[0], 1, 1, 1))
24         bf_t = nm.ascontiguousarray(bf_t)
25         if material is not None:
26             status = geo.integrate(out, material * bf_t)
27         else:
28             status = geo.integrate(out, bf_t)
29         return status
30
31     def get_fargs(self, material, virtual,
32                  mode=None, term_mode=None, diff_var=None, **kwargs):
33         assert_(virtual.n_components == 1)
34         geo, _ = self.get_mapping(virtual)
35
36         return material, geo.bf, geo

```

- lines 2-14: the docstring - always write one!
- line 15: the name of the term, that can be referred to in equations;
- line 16: the argument types - here the term takes a single material parameter, and a virtual variable;
- lines 17-18: the possible argument shapes
- line 19: the integration mode is choosen according to a given domain
- lines 21-29: the term function
 - its arguments are:
 - * the output array *out*, already having the required shape,
 - * the material coefficient (array) *mat* evaluated in physical quadrature points of elements of the term region,
 - * a base function (array) *bf* evaluated in the quadrature points of a reference element and
 - * a reference element (geometry) mapping *geo*.
 - line 23: transpose the base function and tile it so that is has the correct shape - it is repeated for each element;
 - line 24: ensure C contiguous order;
 - lines 25-28: perform numerical integration in C - *geo.integrate()* requires the C contiguous order;
 - line 29: return the status.
- lines 31-36: prepare arguments for the function above:
 - line 33: verify that the variable is scalar, as our implementation does not support vectors;

- line 34: get reference element mapping corresponding to the virtual variable;
- line 36: return the arguments for the function.

A more complex term that involves an unknown variable and has two call modes, is `dw_s_dot_mgrad_s`, defined as $\int_{\Omega} q \underline{y} \cdot \nabla p$ in the ‘grad_state’ mode or $\int_{\Omega} p \underline{y} \cdot \nabla q$ in the ‘grad_virtual’ mode, where $\underline{y}$ is a vector material parameter, q is a virtual variable, and p is a state variable:

```

1  class ScalarDotMGradScalarTerm(Term):
2      r"""
3      Volume dot product of a scalar gradient dotted with a material vector
4      with a scalar.
5
6      :Definition:
7
8      .. math::
9          \int_{\Omega} q \, \underline{y} \cdot \nabla p \, \mathrm{d}x
10         \int_{\Omega} p \, \underline{y} \cdot \nabla q \, \mathrm{d}x
11
12     :Arguments 1:
13         - material : :math:\underline{y}`
14         - virtual  : :math:q`
15         - state    : :math:p`
16
17     :Arguments 2:
18         - material : :math:\underline{y}`
19         - state    : :math:p`
20         - virtual  : :math:q`
21     """
22     name = 'dw_s_dot_mgrad_s'
23     arg_types = (('material', 'virtual', 'state'),
24                  ('material', 'state', 'virtual'))
25     arg_shapes = [{'material' : 'D, 1',
26                     'virtual/grad_state' : (1, None),
27                     'state/grad_state' : 1,
28                     'virtual/grad_virtual' : (1, None),
29                     'state/grad_virtual' : 1}]
30     modes = ('grad_state', 'grad_virtual')
31
32     @staticmethod
33     def function(out, out_qp, geo, fmode):
34         status = geo.integrate(out, out_qp)
35         return status
36
37     def get_fargs(self, mat, var1, var2,
38                  mode=None, term_mode=None, diff_var=None, **kwargs):
39         vg1, _ = self.get_mapping(var1)
40         vg2, _ = self.get_mapping(var2)
41
42         if diff_var is None:
43             if self.mode == 'grad_state':
44                 geo = vg1
45                 bf_t = vg1.bf.transpose((0, 1, 3, 2))
46                 val_qp = self.get(var2, 'grad')

```

(continues on next page)

(continued from previous page)

```

47         out_qp = bf_t * dot_sequences(mat, val_qp, 'ATB')
48
49     else:
50         geo = vg2
51         val_qp = self.get(var1, 'val')
52         out_qp = dot_sequences(vg2.bfg, mat, 'ATB') * val_qp
53
54         fmode = 0
55
56     else:
57         if self.mode == 'grad_state':
58             geo = vg1
59             bf_t = vg1.bf.transpose((0, 1, 3, 2))
60             out_qp = bf_t * dot_sequences(mat, vg2.bfg, 'ATB')
61
62         else:
63             geo = vg2
64             out_qp = dot_sequences(vg2.bfg, mat, 'ATB') * vg1.bf
65
66         fmode = 1
67
68     return out_qp, geo, fmode

```

Only interesting differences with respect to the previous example will be discussed:

- the argument types and shapes (lines 23-29) have to be specified for all the call modes (line 30)
- the term function (lines 32-35) just integrates the element contributions, as all the other calculations are done by the `get_fargs()` function.
- the `get_fargs()` function (lines 37-68) contains:
 - residual computation (lines 43-54) for both modes
 - matrix computation (lines 57-66) for both modes

Concluding remarks

This is just a very basic introduction to the topic of new term implementation. Do not hesitate to ask the [SfePy mailing list](#), and look at the source code of the already implemented terms.

2.3.7 Multi-linear Terms

tentative documentation, the enriched einsum notation is still in flux

Multi-linear terms can be implemented simply by using the following enriched einsum notation:

Table 3: The enriched einsum notation for defining multi-linear terms.

symbol	meaning	example
$\emptyset$	scalar	p
i	i -th vector component	u_i
$i.j$	gradient: derivative of i -th vector component w.r.t. j -th coordinate component	$\frac{\partial u_i}{\partial x_j}$
$i:j$	symmetric gradient	$\frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$
$s(i:j) \rightarrow$	vector storage of symmetric second order tensor, I is the vector component	Cauchy strain tensor $e_{ij}(\underline{u})$

The examples below present the new way of implementing the terms shown in the original [Examples](#), using `sfePy.terms.terms_multilinear.ETermBase`.

Examples

- `de_integrate` defined as $\int_{\Omega} c q$, where c is a weight (material parameter) and q is a virtual variable:

```

1 class EIntegrateOperatorTerm(ETermBase):
2     r"""
3     Volume and surface integral of a test function weighted by a scalar
4     function :math:`c`.
5
6     :Definition:
7
8     .. math::
9         \int_{\text{cal}\{D\}} q \, \text{or} \, \int_{\text{cal}\{D\}} c \, q
10
11     :Arguments:
12         - material : :math:`c` (optional)
13         - virtual  : :math:`q`
14     """
15     name = 'de_integrate'
16     arg_types = ('opt_material', 'virtual')
17     arg_shapes = [{'opt_material' : '1, 1', 'virtual' : (1, None)},
18                  {'opt_material' : None}]
19
20     def get_function(self, mat, virtual, mode=None, term_mode=None,
21                     diff_var=None, **kwargs):
22         if mat is None:
23             fun = self.make_function(
24                 '0', virtual, diff_var=diff_var,
25             )
26
27         else:
28             fun = self.make_function(
29                 '00,0', mat, virtual, diff_var=diff_var,
30             )
31
32     return fun

```

- `de_s_dot_mgrad_s` defined as $\int_{\Omega} q \underline{y} \cdot \nabla p$ in the `'grad_state'` mode or $\int_{\Omega} p \underline{y} \cdot \nabla q$ in the `'grad_virtual'` mode,

where y is a vector material parameter, q is a virtual variable, and p is a state variable:

```

1 class EScalarDotMGradScalarTerm(ETermBase):
2     r"""
3     Volume dot product of a scalar gradient dotted with a material vector with
4     a scalar.
5
6     :Definition:
7
8     .. math::
9         \int_{\Omega} q \, \ul{y} \cdot \nabla p \, \mbox{ , }
10        \int_{\Omega} p \, \ul{y} \cdot \nabla q
11
12     :Arguments 1:
13         - material : :math:\ul{y}`
14         - virtual  : :math:q`
15         - state    : :math:p`
16
17     :Arguments 2:
18         - material : :math:\ul{y}`
19         - state    : :math:p`
20         - virtual  : :math:q`
21     """
22     name = 'de_s_dot_mgrad_s'
23     arg_types = (('material', 'virtual', 'state'),
24                  ('material', 'state', 'virtual'))
25     arg_shapes = [{'material' : 'D, 1',
26                     'virtual/grad_state' : (1, None),
27                     'state/grad_state' : 1,
28                     'virtual/grad_virtual' : (1, None),
29                     'state/grad_virtual' : 1}]
30     modes = ('grad_state', 'grad_virtual')
31
32     def get_function(self, mat, var1, var2, mode=None, term_mode=None,
33                     diff_var=None, **kwargs):
34         return self.make_function(
35             'i0,0,0.i', mat, var1, var2, diff_var=diff_var,
36             )

```

2.3.8 How To Make a Release

Release Tasks

A few notes on what to do during a release.

Things to check before a release

1. synchronize module documentation (dry run):

```
$ python3 tools/sync_module_docs.py doc/src/ . -n
```

2. regenerate gallery page and examples:

```
$ rm -rf doc/examples/
$ python3 tools/gen_gallery.py
```

3. create temporary/testing tarball:

```
$ python3 setup.py sdist
```

4. check in-place build:

```
$ # unpack the tarball
$ # cd into

$ python3 setup.py build_ext --inplace
$ python3 sfepy/scripts/test_install.py
```

5. check that documentation can be built:

```
$ # copy site_cfg.py
$ python3 setup.py htmldocs
$ firefox doc/_build/html/index.html
```

or use:

```
$ cd doc/
$ make html
$ firefox _build/html/index.html
```

try also:

```
$ python3 setup.py pdfdocs
```

6. check installed build:

```
$ python3 -m pip install . --user
$ cd
$ sfepy-test
```

then remove the installed files so that they do not interfere with the local build

7. create final tarball

- update doc/release_notes.rst, with the help of:

```
$ python3 tools/gen_release_notes.py 2019.2
```

- update doc/news.rst, doc/archived_news.rst
- change version number (sfepy/version.py) so that previous release tarball is not overwritten!
- set `is_release = True` in `site_cfg.py`

- update pdfdocs:

```
$ python3 setup.py pdfdocs
```

- create tarball:

```
$ python3 setup.py sdist
```

8. tag the release using:

```
$ git tag release_XXXX.X
```

Useful Git commands

- log

```
git log --pretty=format:"%s%n%b%n" --topo-order --reverse release_2016.4..HEAD
```

- who has contributed since <date>:

```
git log --after=<date> | grep Author | sort | uniq
git log release_2012.1..HEAD | grep Author | sort -k3 | uniq
git shortlog -s -n release_2012.3..HEAD

git rev-list --committer="Name Surname" --since=6.months.ago HEAD | wc
git rev-list --author="Name Surname" --since=6.months.ago HEAD | wc
# ?no-merges
```

- misc:

```
git archive --format=tar HEAD | gzip > name.tar.gz
```

Web update and file uploading

- upload the sources to TestPyPI and test:

- bump version number as needed when testing
- upload commands:

```
python3 setup.py sdist # set ``is_release = True`` in site_cfg.py
python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/
↳ sfepy-2022.3[.-]*
```

- testing:

```
python3 -m venv venv
source venv/bin/activate
python3 -m pip install -U --index-url https://test.pypi.org/simple/ --extra-
↳ index-url https://pypi.org/simple/ sfepy
python3 -m pip install pytest
python3 -c "import sfepy; sfepy.test('-v', '--durations=0')"
```

(continues on next page)

(continued from previous page)

```
deactivate
# rm -rf venv
```

- upload the sources to PyPI and test:

- upload commands:

```
python3 setup.py sdist # set ``is_release = True`` in site_cfg.py
python3 -m twine upload dist/sfepy-2022.3[.-]*
```

- testing:

```
python3 -m venv venv
source venv/bin/activate
python3 -m pip install -U sfepy
python3 -m pip install pytest
python3 -c "import sfepy; sfepy.test('-v', '--durations=0')"
deactivate
# rm -rf venv
```

- make a pull request with the updated version in `sfepy-feedstock/recipe/meta.yaml` from a fork (e.g. <https://github.com/rc/sfepy-feedstock>) of <https://github.com/conda-forge/sfepy-feedstock>.
- publish development docs also as new release docs
- send announcement to
 - `sfepy@python.org`
 - optionally to `scipy-dev@python.org`, `scipy-user@python.org`, `numpy-discussion@python.org`, `python-announce-list@python.org`

2.3.9 Module Index

sfepy package

sfepy.config module

class `sfepy.config.Config`

Read and access the site configuration from the current directory.

`compile_flags()`

`debug_flags()` → list

`debug_warped_cells()`

`is_release()`

`numpydoc_path()`

`python_include()`

`python_version()`

refmap_memory_factor()

system()

sfepy.config.compose_system_compile_flags(*is_posix: bool*) → list

Provides a list of compile flags that tries to be as similar as possible to what Python itself was built with. This has been done historically by `numpy.distutils` (now deprecated) and a squeezed version of it is brought over to here.

sfepy.config.get_top_dir()

Return SfePy installation directory information.

sfepy.config.has_attr(*obj, attr*)

sfepy.version module

sfepy.version.get_version(*version='2025.1'*)

Get sfepy version.

Appends current git commit hash to *version* if *is_release* is set to True in the site configuration.

sfepy.applications package

sfepy.applications.application module

class sfepy.applications.application.Application(*conf, options, output_prefix, **kwargs*)

Base class for applications.

Subclasses should implement: `__init__()`, `call()`.

Automates parametric studies, see `parametrize()`.

call_basic(***kwargs*)

call_parametrized(***kwargs*)

parametrize(*parametric_hook*)

Add *parametric_hook*, set `__call__()` to `call_parametrized()`.

restore()

Remove *parametric_hook*, restore `__call__()` to `call_basic()`.

setup_options()

sfepy.applications.evp_solver_app module

Eigenvalue problem solver application.

class sfepy.applications.evp_solver_app.EVPSolverApp(*conf, options, output_prefix, **kwargs*)

Solve an eigenvalue problem.

call(*status=None*)

make_full(*svecs*)

static process_options(*options*)

Application options setup. Sets default values for missing non-compulsory options.

save_results(*eigs, vecs, out=None, mesh_results_name=None, eig_results_name=None*)

setup_options()

setup_output()

Setup various file names for the output directory given by *self.problem.output_dir*.

solve_eigen_problem()

sfepy.applications.pde_solver_app module

class sfepy.applications.pde_solver_app.PDESolverApp(*conf, options, output_prefix, init_equations=True, **kwargs*)

call(*status=None*)

load_dict(*filename*)

Utility function to load a dictionary *data* from a HDF5 file *filename*.

static process_options(*options*)

Application options setup. Sets default values for missing non-compulsory options.

save_dict(*filename, data*)

Utility function to save a dictionary *data* to a HDF5 file *filename*.

setup_options()

setup_output_info(*problem, options*)

Modifies both problem and options!

sfepy.applications.pde_solver_app.assign_standard_hooks(*obj, get, conf*)

Set standard hook function attributes from *conf* to *obj* using the *get* function.

sfepy.applications.pde_solver_app.save_only(*conf, save_names, problem=None*)

Save information available prior to setting equations and solving them.

sfepy.applications.pde_solver_app.solve_pde(*conf, define_args=None, options=None, status=None, **app_options*)

Solve a system of partial differential equations (PDEs).

This function is a convenience wrapper that creates and runs an instance of [PDESolverApp](#).

Parameters

conf

[str or ProblemConf instance] Either the name of the problem description file defining the PDEs, or directly the ProblemConf instance.

define_args

[dict, optional] Keyword arguments for `define()` function of the problem description file.

options

[options] The command-line options.

status

[dict-like] The object for storing the solver return status.

app_options

[kwargs] The keyword arguments that can override application-specific options.

sfepy.base package**sfepy.base.base module**

`sfepy.base.base.debug(frame=None, frames_back=1)`

Start debugger on line where it is called, roughly equivalent to:

```
import pdb; pdb.set_trace()
```

First, this function tries to start an *IPython*-enabled debugger using the *IPython* API.

When this fails, the plain old *pdb* is used instead.

With *IPython*, one can say in what frame the debugger can stop.

class `sfepy.base.base.Container(objs=None, **kwargs)`

append(*obj*)

as_dict()

Return stored objects in a dictionary with object names as keys.

extend(*objs*)

Extend the container items by the sequence *objs*.

get(*ii*, *default=None*, *msg_if_none=None*)

Get an item from Container - a wrapper around Container.__getitem__() with defaults and custom error message.

Parameters

ii

[int or str] The index or name of the item.

default

[any, optional] The default value returned in case the item *ii* does not exist.

msg_if_none

[str, optional] If not None, and if *default* is None and the item *ii* does not exist, raise ValueError with this message.

get_names()

has_key(*ii*)

insert(*ii*, *obj*)

iteritems()

iterkeys()

itervalues()

print_names()

remove_name(*name*)

update(*objs=None*)

A dict-like update for Struct attributes.

class sfepy.base.base.**IndexedStruct**(**kwargs)

setdefault(*key, default=None*)

class sfepy.base.base.**OneTypeList**(*item_class, seq=None*)

find(*name, ret_indx=False*)

get_names()

print_names()

class sfepy.base.base.**Output**(*prefix, filename=None, quiet=False, combined=False, append=False, **kwargs*)

Factory class providing output (print) functions. All SfePy printing should be accomplished by this class.

Examples

```
>>> from sfepy.base.base import Output
>>> output = Output('sfepy:')
>>> output(1, 2, 3, 'hello')
sfepy: 1 2 3 hello
>>> output.prefix = 'my_cool_app:'
>>> output(1, 2, 3, 'hello')
my_cool_app: 1 2 3 hello
```

get_output_function()

get_output_prefix()

property prefix

set_output(*filename=None, quiet=False, combined=False, append=False*)

Set the output mode.

If *quiet* is *True*, no messages are printed to screen. If simultaneously *filename* is not *None*, the messages are logged into the specified file.

If *quiet* is *False*, more combinations are possible. If *filename* is *None*, output is to screen only, otherwise it is to the specified file. Moreover, if *combined* is *True*, both the ways are used.

Parameters

filename

[str or file object] Print messages into the specified file.

quiet

[bool] Do not print anything to screen.

combined

[bool] Print both on screen and into the specified file.

append

[bool] Append to an existing file instead of overwriting it. Use with *filename*.

set_output_prefix(*prefix*)

class sfepy.base.base.**Struct**(***kwargs*)

copy(*deep=False, name=None*)

Make a (deep) copy of self.

Parameters:

deep

[bool] Make a deep copy.

name

[str] Name of the copy, with default self.name + ‘_copy’.

get(*key, default=None, msg_if_none=None*)

A dict-like get() for Struct attributes.

set_default(*key, default=None*)

Behaves like dict.setdefault().

str_all()

str_class()

As __str__(), but for class attributes.

to_dict()

update(*other, **kwargs*)

A dict-like update for Struct attributes.

sfepy.base.base.**as_float_or_complex**(*val*)

Try to cast val to Python float, and if this fails, to Python complex type.

sfepy.base.base.**assert_**(*condition, msg='assertion failed!'*)

sfepy.base.base.**check_names**(*names1, names2, msg*)

Check if all names in names1 are in names2, otherwise raise IndexError with the provided message msg.

sfepy.base.base.**configure_output**(*options*)

Configure the standard output() function using *output_log_name* and *output_screen* attributes of *options*.

Parameters

options

[Struct or dict] The options with *output_screen* and *output_log_name* items. Defaults are provided if missing.

sfepy.base.base.**debug**(*frame=None, frames_back=1*)

Start debugger on line where it is called, roughly equivalent to:

```
import pdb; pdb.set_trace()
```

First, this function tries to start an *IPython*-enabled debugger using the *IPython* API.

When this fails, the plain old *pdb* is used instead.

With *IPython*, one can say in what frame the debugger can stop.

sfepy.base.base.**debug_on_error**()

Start debugger at the line where an exception was raised.

`sfepy.base.base.dict_extend(d1, d2)`

`sfepy.base.base.dict_from_keys_init(keys, seq_class=None)`

`sfepy.base.base.dict_to_array(adict)`

Convert a dictionary of nD arrays of the same shapes with non-negative integer keys to a single (n+1)D array.

`sfepy.base.base.dict_to_struct(*args, **kwargs)`

Convert a dict instance to a Struct instance.

`sfepy.base.base.edit_dict_strings(str_dict, old, new, recur=False)`

Replace substrings *old* with *new* in string values of dictionary *str_dict*. Both *old* and *new* can be lists of the same length - items in *old* are replaced by items in *new* with the same index.

Parameters

str_dict

[dict] The dictionary with string values or tuples containing strings.

old

[str or list of str] The old substring or list of substrings.

new

[str or list of str] The new substring or list of substrings.

recur

[bool] If True, edit tuple values recursively.

Returns

new_dict

[dict] The dictionary with edited strings.

`sfepy.base.base.edit_tuple_strings(str_tuple, old, new, recur=False)`

Replace substrings *old* with *new* in items of tuple *str_tuple*. Non-string items are just copied to the new tuple.

Parameters

str_tuple

[tuple] The tuple with string values.

old

[str] The old substring.

new

[str] The new substring.

recur

[bool] If True, edit items that are tuples recursively.

Returns

new_tuple

[tuple] The tuple with edited strings.

`sfepy.base.base.find_subclasses(context, classes, omit_unnamed=False, name_attr='name')`

Find subclasses of the given classes in the given context.

Examples

```
>>> solver_table = find_subclasses(vars().items(),
                                   [LinearSolver, NonlinearSolver,
                                    TimeSteppingSolver, EigenvalueSolver,
                                    OptimizationSolver])
```

`sfepy.base.base.get_arguments(omit=None)`

Get a calling function's arguments.

Returns:

args

[dict] The calling function's arguments.

`sfepy.base.base.get_debug()`

Utility function providing `debug()` function.

`sfepy.base.base.get_default(arg, default, msg_if_none=None)`

`sfepy.base.base.get_default_attr(obj, attr, default, msg_if_none=None)`

`sfepy.base.base.get_subdict(adict, keys)`

Get a sub-dictionary of *adict* with given *keys*.

`sfepy.base.base.import_file(filename, package_name=None, can_reload=True)`

Import a file as a module. The module is explicitly reloaded to prevent undesirable interactions.

`sfepy.base.base.insert_as_static_method(cls, name, function)`

`sfepy.base.base.insert_method(instance, function)`

`sfepy.base.base.insert_static_method(cls, function)`

`sfepy.base.base.invert_dict(d, is_val_tuple=False, unique=True)`

Invert a dictionary by making its values keys and vice versa.

Parameters

d

[dict] The input dictionary.

is_val_tuple

[bool] If True, the *d* values are tuples and new keys are the tuple items.

unique

[bool] If True, the *d* values are unique and so the mapping is one to one. If False, the *d* values (possibly) repeat, so the inverted dictionary will have as items lists of corresponding keys.

Returns

di

[dict] The inverted dictionary.

`sfepy.base.base.ipython_shell(frame=0)`

`sfepy.base.base.is_derived_class(cls, parent)`

`sfepy.base.base.is_integer(var)`

`sfepy.base.base.is_sequence(var)`

`sfepy.base.base.is_string(var)`

`sfepy.base.base.iter_dict_of_lists(dol, return_keys=False)`

`sfepy.base.base.load_classes(filenamees, classes, package_name=None, ignore_errors=False, name_attr='name')`

For each filename in filenamees, load all subclasses of classes listed.

`sfepy.base.base.ordered_iteritems(adict)`

`sfepy.base.base.pause(msg=None)`

Prints the line number and waits for a keypress.

If you press: “q” it will call `sys.exit()` any other key ... it will continue execution of the program

This is useful for debugging.

`sfepy.base.base.print_structs(objs)`

Print Struct instances in a container, works recursively. Debugging utility function.

`sfepy.base.base.python_shell(frame=0)`

`sfepy.base.base.remap_dict(d, map)`

Utility function to remap state dict keys according to `var_map`.

`sfepy.base.base.select_by_names(objs_all, names, replace=None, simple=True)`

`sfepy.base.base.set_defaults(dict_, defaults)`

`sfepy.base.base.shell(frame=0)`

Embed an IPython (if available) or regular Python shell in the given frame.

`sfepy.base.base.spause(msg=None)`

Waits for a keypress.

If you press: “q” it will call `sys.exit()` any other key ... it will continue execution of the program

This is useful for debugging. This function is called from `pause()`.

`sfepy.base.base.structify(obj)`

Convert a (nested) dict *obj* into a (nested) Struct.

`sfepy.base.base.try_imports(imports, fail_msg=None)`

Try import statements until one succeeds.

Parameters

imports

[list] The list of import statements.

fail_msg

[str] If not None and no statement succeeds, a *ValueError* is raised with the given message, appended to all failed messages.

Returns

locals

[dict] The dictionary of imported modules.

`sfepy.base.base.update_dict_recursively(dst, src, tuples_too=False, overwrite_by_none=True)`

Update *dst* dictionary recursively using items in *src* dictionary.

Parameters

dst

[dict] The destination dictionary.

src

[dict] The source dictionary.

tuples_too

[bool] If True, recurse also into dictionaries that are members of tuples.

overwrite_by_none

[bool] If False, do not overwrite destination dictionary values by None.

Returns

dst

[dict] The destination dictionary.

`sfepy.base.base.use_method_with_name(instance, method, new_name)`

sfepy.base.compat module

This module contains functions that have different names or behavior depending on NumPy and Scipy versions.

`sfepy.base.compat.in1d(ar1, ar2, assume_unique=False, invert=False, *, kind=None)`

Test whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as *ar1* that is True where an element of *ar1* is in *ar2* and False otherwise.

We recommend using `isin()` instead of *in1d* for new code.

Parameters

ar1

[(M,) array_like] Input array.

ar2

[array_like] The values against which to test each value of *ar1*.

assume_unique

[bool, optional] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

invert

[bool, optional] If True, the values in the returned array are inverted (that is, False where an element of *ar1* is in *ar2* and True otherwise). Default is False. `np.in1d(a, b, invert=True)` is equivalent to (but is faster than) `np.invert(np.in1d(a, b))`.

kind

[{None, 'sort', 'table'}, optional] The algorithm to use. This will not affect the final result, but will affect the speed and memory use. The default, None, will select automatically based on memory considerations.

- If 'sort', will use a mergesort-based approach. This will have a memory usage of roughly 6 times the sum of the sizes of *ar1* and *ar2*, not accounting for size of dtypes.

- If 'table', will use a lookup table approach similar to a counting sort. This is only available for boolean and integer arrays. This will have a memory usage of the size of *ar1* plus the max-min value of *ar2*. *assume_unique* has no effect when the 'table' option is used.
- If None, will automatically choose 'table' if the required memory allocation is less than or equal to 6 times the sum of the sizes of *ar1* and *ar2*, otherwise will use 'sort'. This is done to not use a large amount of memory by default, even though 'table' may be faster in most cases. If 'table' is chosen, *assume_unique* will have no effect.

Added in version 1.8.0.

Returns

in1d

[(M,) ndarray, bool] The values *ar1[in1d]* are in *ar2*.

See also

isin

Version of this function that preserves the shape of *ar1*.

numpy.lib.arraysetops

Module with a number of other functions for performing set operations on arrays.

Notes

in1d can be considered as an element-wise function version of the python keyword *in*, for 1-D sequences. *in1d(a, b)* is roughly equivalent to `np.array([item in b for item in a])`. However, this idea fails if *ar2* is a set, or similar (non-sequence) container: As *ar2* is converted to an array, in those cases `asarray(ar2)` is an object array rather than the expected array of contained values.

Using `kind='table'` tends to be faster than `kind='sort'` if the following relationship is true: $\log_{10}(\text{len}(\text{ar2})) > (\log_{10}(\text{max}(\text{ar2}) - \text{min}(\text{ar2})) - 2.27) / 0.927$, but may use greater memory. The default value for *kind* will be automatically selected based only on memory usage, so one may manually set `kind='table'` if memory constraints can be relaxed.

Added in version 1.4.0.

Examples

```
>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
>>> mask
array([ True, False,  True, False,  True])
>>> test[mask]
array([0, 2, 0])
>>> mask = np.in1d(test, states, invert=True)
>>> mask
array([False,  True, False,  True, False])
>>> test[mask]
array([1, 5])
```

```
sfePy.base.compat.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *,
                          equal_nan=True)
```

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

Parameters

ar

[array_like] Input array. Unless *axis* is specified, this will be flattened if it is not already 1-D.

return_index

[bool, optional] If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

return_inverse

[bool, optional] If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.

return_counts

[bool, optional] If True, also return the number of times each unique item appears in *ar*.

axis

[int or None, optional] The axis to operate on. If None, *ar* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.

Added in version 1.13.0.

equal_nan

[bool, optional] If True, collapses multiple NaN values in the return array into one.

Added in version 1.24.

Returns

unique

[ndarray] The sorted unique values.

unique_indices

[ndarray, optional] The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

unique_inverse

[ndarray, optional] The indices to reconstruct the original array from the unique array. Only provided if *return_inverse* is True.

unique_counts

[ndarray, optional] The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

Added in version 1.9.0.

 See also**numpy.lib.arraysetops**

Module with a number of other functions for performing set operations on arrays.

repeat

Repeat elements of an array.

Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order. The flattened subarrays are then viewed as a structured type with each element given a label, with the effect that we end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the unique rows of a 2D array

```
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'], dtype='<U1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'], dtype='<U1')
```

Reconstruct the input array from the unique values and inverse:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
```

(continues on next page)

(continued from previous page)

```
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

Reconstruct the input values from the unique values and counts:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> values, counts = np.unique(a, return_counts=True)
>>> values
array([1, 2, 3, 4, 6])
>>> counts
array([1, 3, 1, 1, 1])
>>> np.repeat(values, counts)
array([1, 2, 2, 2, 3, 4, 6]) # original order not preserved
```

sfepy.base.conf module

Problem description file handling.

Notes

Short syntax: key is suffixed with ‘__<number>’ to prevent collisions with long syntax keys -> both cases can be used in a single input.

class `sfepy.base.conf.ProblemConf`(*define_dict*, *funmod=None*, *filename=None*, *required=None*, *other=None*, *verbose=True*, *override=None*, *setup=True*)

Problem configuration, corresponding to an input (problem description file). It validates the input using lists of required and other keywords that have to/can appear in the input. Default keyword lists can be obtained by `sfepy.base.conf.get_standard_keywords()`.

ProblemConf instance is used to construct a Problem instance via `Problem.from_conf(conf)`.

add_missing(*conf*)

Add missing values from another problem configuration.

Missing keys/values are added also to values that are dictionaries.

Parameters

conf

[ProblemConf instance] The other configuration.

edit(*key*, *newval*)

static from_dict(*dict_*, *funmod*, *required=None*, *other=None*, *verbose=True*, *override=None*, *setup=True*)

static from_file(*filename*, *required=None*, *other=None*, *verbose=True*, *define_args=None*, *override=None*, *setup=True*)

Loads the problem definition from a file.

The filename can either contain plain definitions, or it can contain the `define()` function, in which case it will be called to return the input definitions.

The job of the `define()` function is to return a dictionary of parameters. How the dictionary is constructed is not our business, but the usual way is to simply have a function `define()` along these lines in the input file:

```
def define():
    options = {
        'save_eig_vectors' : None,
        'eigen_solver' : 'eigen1',
    }
    region_2 = {
        'name' : 'Surface',
        'select' : 'nodes of surface',
    }
    return locals()
```

Optionally, the `define()` function can accept additional arguments that should be defined using the *define_args* tuple or dictionary.

static from_file_and_options(*filename*, *options*, *required=None*, *other=None*, *verbose=True*, *define_args=None*, *setup=True*)

Utility function, a wrapper around `ProblemConf.from_file()` with possible override taken from *options*.

static from_module(*module*, *required=None*, *other=None*, *verbose=True*, *override=None*, *setup=True*)

get_function(*name*)

Get a function object given its name.

It can be either in *ProblemConf.funmod*, or a *ProblemConf* attribute directly.

Parameters

name

[str or function or None] The function name or directly the function.

Returns

fun

[function or None] The required function, or None if *name* was None.

get_item_by_name(*key*, *item_name*)

Return item with name *item_name* in configuration group given by *key*.

get_raw(*key=None*)

setup(*define_dict=None*, *funmod=None*, *filename=None*, *required=None*, *other=None*)

transform_input()

transform_input_trivial()

Trivial input transformations.

update_conf(*conf*)

Update configuration by values in another problem configuration.

Values that are dictionaries are updated in-place by `dict.update()`.

Parameters

conf

[ProblemConf instance] The other configuration.

validate(*required=None*, *other=None*)

`sfepy.base.conf.dict_from_options(options)`

Return a dictionary that can be used to construct/override a ProblemConf instance based on *options*.

See `--conf` and `--options` options of the `simple.py` script.

`sfepy.base.conf.dict_from_string(string, allow_tuple=False, free_word=False)`

Parse *string* and return a dictionary that can be used to construct/override a ProblemConf instance.

`sfepy.base.conf.get_standard_keywords()`

`sfepy.base.conf.transform_conditions(adict, prefix)`

`sfepy.base.conf.transform_dgebcs(adict)`

`sfepy.base.conf.transform_dgepbcs(adict)`

`sfepy.base.conf.transform_ebcs(adict)`

`sfepy.base.conf.transform_epbcs(adict, prefix='epbc')`

`sfepy.base.conf.transform_fields(adict)`

`sfepy.base.conf.transform_functions(adict)`

`sfepy.base.conf.transform_ics(adict)`

`sfepy.base.conf.transform_integrals(adict)`

`sfepy.base.conf.transform_lcbcs(adict)`

`sfepy.base.conf.transform_materials(adict)`

`sfepy.base.conf.transform_regions(adict)`

`sfepy.base.conf.transform_solvers(adict)`

`sfepy.base.conf.transform_to_i_struct_1(adict)`

`sfepy.base.conf.transform_to_struct_01(adict)`

`sfepy.base.conf.transform_to_struct_1(adict)`

`sfepy.base.conf.transform_to_struct_10(adict)`

`sfepy.base.conf.transform_variables(adict)`

`sfepy.base.conf.tuple_to_conf(name, vals, order)`

Convert a configuration tuple *vals* into a Struct named *name*, with attribute names given in and ordered by *order*.

Items in *order* at indices outside the length of *vals* are ignored.

sfepy.base.getch module

`getch()`-like unbuffered character reading from stdin on both Windows and Unix

`_Getch` classes inspired by Danny Yoo, `iskeydown()` based on code by Zachary Pincus.

sfepy.base.goptions module

Various global options/parameters.

Notes

Inspired by rcParams of matplotlib.

class sfepy.base.goptions.**ValidatedDict**

A dictionary object including validation.

keys()

Return sorted list of keys.

validate = {'check_term_finiteness': <function validate_bool>, 'verbose':
<function validate_bool>}

values()

Return values in order of sorted keys.

sfepy.base.goptions.**validate_bool**(val)

Convert b to a boolean or raise a ValueError.

sfepy.base.ioutils module

class sfepy.base.ioutils.**Cached**(data)

The wrapper class that marks data, that should be checked during saving, whether it has been stored to the hdf5 file already and if so, a softlink to the already created instance is created instead of saving.

class sfepy.base.ioutils.**DataMarker**(data)

The Base class for classes for marking data to be handled in a special way during saving to a HDF5 file by write_to_hdf5(). The usage is simple: just “decorate” the desired data element, e.g.:

```
data = [data1, Cached(data2)]
write_to_hdf5(... , ... , data)
```

unpack_data()

One can request unpacking of the wrappers during saving.

Returns

object

The original object, if possible, or self.

class sfepy.base.ioutils.**DataSoftLink**(type, destination, cache=None)

This object is written to the HDF5 file as a softlink to the given path. The destination of the softlink should contain only data, so the structure {type: type, data: softlink_to(destination)} is created in the place where the softlink is written.

get_type()

unpack_data()

One can request unpacking of the wrappers during saving.

Returns

object

The original object, if possible, or self.

write_data(*fd*, *group*, *cache=None*)

Create the softlink to the destination and handle the caching.

class sfepy.base.ioutils.**HDF5BaseData**

When storing values to HDF5, special classes can be used that wrap the stored data and modify the way the storing is done. This class is the base of those.

unpack_data()

One can request unpacking of the wrappers during saving.

Returns**object**

The original object, if possible, or self.

class sfepy.base.ioutils.**HDF5ContextManager**(*filename*, **args*, ***kwargs*)

class sfepy.base.ioutils.**HDF5Data**

Some data written to the HDF5 file can have a custom format. Descendants of this class should have the method `.write_data()` or redefine the `.write()` method.

write(*fd*, *group*, *name*, *cache=None*)

Write a data structure to the HDF5 file.

Create the following structure in the HDF5 file: {type: self.get_type(), anything written by self.write_data()}

Parameters**fd: tables.File**

The hdf5 file handle the data should be written in.

group: tables.group.Group

The group the data will be stored to

name: str

Name of node that will be appended to group and will contain the data

cache: dict or None, optional

Store for already cached objects with structs id(obj) : /path/to Can be used for not storing the one object twice.

write_data(*fd*, *group*)

Write data to the HDF5 file. Redefine this function in sub-classes.

Parameters**fd: tables.File**

The hdf5 file handle the data should be written to.

group: tables.group.Group

The group the data should be stored to.

class sfepy.base.ioutils.**InDir**(*filename*)

Store the directory name a file is in, and prepend this name to other files.

Examples

```
>>> indir = InDir('output/file1')
>>> print indir('file2')
```

class sfepy.base.ioutils.**SoftLink**(*destination*)

This object is written to the HDF5 file as a softlink to the given path.

write(*fd, group, name, cache=None*)

Create the softlink to the destination.

class sfepy.base.ioutils.**Uncached**(*data*)

The wrapper class that marks data, that should be always stored to the hdf5 file, even if the object has been already stored at a different path in the file and so it would have been stored by a softlink otherwise (IGDomain, Mesh and sparse matrices behave so).

sfepy.base.ioutils.**dec**(*val, encoding='utf-8'*)

Decode given bytes using the specified encoding.

sfepy.base.ioutils.**edit_filename**(*filename, prefix="", suffix="", new_ext=None*)

Edit a file name by add a prefix, inserting a suffix in front of a file name extension or replacing the extension.

Parameters

filename

[str] The file name.

prefix

[str] The prefix to be added.

suffix

[str] The suffix to be inserted.

new_ext

[str, optional] If not None, it replaces the original file name extension.

Returns

new_filename

[str] The new file name.

sfepy.base.ioutils.**enc**(*string, encoding='utf-8'*)

Encode given string or bytes using the specified encoding.

sfepy.base.ioutils.**ensure_path**(*filename*)

Check if path to *filename* exists and if not, create the necessary intermediate directories.

sfepy.base.ioutils.**get_or_create_hdf5_group**(*fd, path, from_group=None*)

sfepy.base.ioutils.**get_print_info**(*n_step, fill=None*)

Returns the max. number of digits in range(*n_step*) and the corresponding format string.

Examples:

```
>>> get_print_info(11)
(2, '%2d')
>>> get_print_info(8)
(1, '%1d')
>>> get_print_info(100)
```

(continues on next page)

(continued from previous page)

```
(2, '%2d')
>>> get_print_info(101)
(3, '%3d')
>>> get_print_info(101, fill='0')
(3, '%03d')
```

`sfepy.base.ioutils.get_trunk(filename)`

`sfepy.base.ioutils.locate_files(pattern, root_dir='.', **kwargs)`

Locate all files matching given filename pattern in and below supplied root directory.

The `**kwargs` arguments are passed to `os.walk()`.

`sfepy.base.ioutils.look_ahead_line(fd)`

Read and return a line from the given file object. Saves the current position in the file before the reading occurs and then, after the reading, restores the saved (original) position.

`sfepy.base.ioutils.path_of_hdf5_group(group)`

`sfepy.base.ioutils.read_array(fd, n_row, n_col, dtype)`

Read a NumPy array of shape (n_row, n_col) from the given file object and cast it to type `dtype`. If `n_col` is `None`, determine the number of columns automatically.

`sfepy.base.ioutils.read_dict_hdf5(filename, level=0, group=None, fd=None)`

`sfepy.base.ioutils.read_from_hdf5(fd, group, cache=None)`

Read custom data from a HDF5 file group saved by `write_to_hdf5()`.

The data are stored in a general (possibly nested) structure: {

 ‘type’: string type identifier ‘data’: stored data ‘cache’: string, optional - another possible location
 of object

}

Parameters

fd: `tables.File`

The hdf5 file handle the data should be restored from.

group: `tables.group.Group`

The group in the hdf5 file the data will be restored from.

cache: dict or None

Some objects (e.g. Mesh instances) can be stored on more places in the HDF5 file tree using softlinks, so when the data are restored, the restored objects are stored and searched in cache so that they are created only once. The keys to cache are the (real) paths of the created objects. Moreover, if some stored object has a ‘cache’ key (see e.g. `DataSoftLink` class), and the object with a given ‘path’ has been already created, it is returned instead of creating a new object. Otherwise, the newly created object is associated both with its real path and with the cache key path.

The caching is not active for scalar data types.

Returns

data

[object] The restored custom data.

`sfepy.base.ioutils.read_list(fd, n_item, dtype)`

`sfepy.base.ioutils.read_sparse_matrix_from_hdf5(fd, group, output_format=None)`

Read sparse matrix from given data group of hdf5 file

Parameters

fd: `tables.File`

The hdf5 file handle the matrix will be read from.

group: `tables.group.group`

The hdf5 file group of the file the matrix will be read from.

output_format: {'csr', 'csc', None}, optional

The resulting matrix will be in CSR or CSC format if this parameter is not None (which is default), otherwise it will be in the format the matrix was stored.

Returns

`scipy.sparse.base.spmatrix`

Readed matrix

`sfepy.base.ioutils.read_sparse_matrix_hdf5(filename, output_format=None)`

`sfepy.base.ioutils.read_token(fd)`

Read a single token (sequence of non-whitespace characters) from the given file object.

Notes

Consumes the first whitespace character after the token.

`sfepy.base.ioutils.remove_files(root_dir, **kwargs)`

Remove all files and directories in supplied root directory.

The `**kwargs` arguments are passed to `os.walk()`.

`sfepy.base.ioutils.remove_files_patterns(root_dir, patterns, ignores=None, verbose=False)`

Remove files with names satisfying the given glob patterns in a supplied root directory. Files with patterns in `ignores` are omitted.

`sfepy.base.ioutils.save_options(filename, options_groups, save_command_line=True, quote_command_line=False)`

Save groups of options/parameters into a file.

Each option group has to be a sequence with two items: the group name and the options in {key : value} form.

`sfepy.base.ioutils.skip_read_line(fd, no_eof=False)`

Read the first non-empty line (if any) from the given file object. Return an empty string at EOF, if `no_eof` is False. If it is True, raise the EOFError instead.

`sfepy.base.ioutils.write_dict_hdf5(filename, adict, level=0, group=None, fd=None)`

`sfepy.base.ioutils.write_sparse_matrix_hdf5(filename, mtx, name='a sparse matrix')`

Assume CSR/CSC.

`sfepy.base.ioutils.write_sparse_matrix_to_hdf5(fd, group, mtx)`

Write sparse matrix to given data group of hdf5 file

Parameters

group: tables.group.group

The hdf5 file group the matrix will be read from.

mtx: scipy.sparse.base.spmatrix

The writed matrix

`sfepy.base.ioutils.write_to_hdf5(fd, group, name, data, cache=None, unpack_markers=False)`

Save custom data to a HDF5 file group to be restored by `read_from_hdf5()`.

Allows saving lists, dicts, numpy arrays, scalars, sparse matrices, meshes and iga domains and all pickleable objects.

Parameters

fd: tables.File

The hdf5 file handle the data should be written in.

group: tables.group.Group

The group the data will be stored to.

name: str

The name of the node that will be appended to the group and will contain the data.

data: object

Data to be stored in the HDF5 file.

cache: dict or None

The cache where the paths to stored objects (currently meshes and iga domains) are stored, so subsequent attempts to store such objects create only softlinks to the initially stored object. The `id()` of objects serve as the keys into the cache. Mark the object with `Cached()` or `Uncached()` for (no) softlinking.

unpack_markers:

If True, the input data is modified so that `Cached` and `Uncached` markers are removed from all sub-elements of the data.

Returns

tables.group.Group

The HDF5 group the data was stored to.

sfepy.base.log module

```
class sfepy.base.log.Log(data_names=None, plot_kwargs=None, xlabel=None, ylabel=None,
                        yscale=None, show_legends=True, is_plot=True, aggregate=100, sleep=1.0,
                        log_filename=None, formats=None)
```

Log data and (optionally) plot them in the second process via `LogPlotter`.

```
add_group(names, plot_kwargs=None, yscale=None, xlabel=None, ylabel=None, formats=None)
```

Add a new data group. Notify the plotting process if it is already running.

`count = -1`

```
static from_conf(conf, data_names)
```

Parameters

data_names

[list of lists of str] The data names grouped by subplots: `[[name1, name2, ...], [name3, name4, ...], ...]`, where `name<n>` are strings to display in (sub)plot legends.

get_log_name()

plot_data(*igs*)

plot_vlines(*igs=None, **kwargs*)

Plot vertical lines in axes given by *igs* at current x locations to mark some events.

terminate()

sfepy.base.log.get_logging_conf(*conf, log_name='log'*)

Check for a log configuration ('log' attribute by default) in *conf*. Supply default values if necessary.

Parameters

conf

[Struct] The configuration object.

log_name

[str, optional] The name of the log configuration attribute in *conf*.

Returns

log

[dict] The dictionary {'plot' : <figure_file>, 'text' : <text_log_file>}. One or both values can be None.

sfepy.base.log.iter_names(*data_names, igs=None*)

sfepy.base.log.plot_log(*axs, log, info, xticks=None, yticks=None, xnbins=None, ynbins=None, groups=None, show_legends=True, swap_axes=False*)

Plot log data returned by [read_log\(\)](#) into a specified figure.

Parameters

axs

[sequence of matplotlib.axes.Axes] The list of axes for the log data plots.

log

[dict] The log with data names as keys and (*xs*, *ys*, *vlines*) as values.

info

[dict] The log plot configuration with subplot numbers as keys.

xticks

[list of arrays, optional] The list of x-axis ticks (array or None) for each subplot.

yticks

[list of arrays, optional] The list of y-axis ticks (array or None) for each subplot.

xnbins

[list, optional] The list of x-axis number of bins (int or None) for each subplot.

ynbins

[list, optional] The list of y-axis number of bins (int or None) for each subplot.

groups

[list, optional] The list of data groups subplots. If not given, all groups are plotted.

show_legends

[bool] If True, show legends in plots.

swap_axes

[bool] If True, swap the axes of the plots.

`sfepy.base.log.read_log(filename)`

Read data saved by *Log* into a text file.

Parameters

filename

[str] The name of a text log file.

Returns

log

[dict] The log with data names as keys and (xs, ys, vlines) as values.

info

[dict] The log plot configuration with subplot numbers as keys.

`sfepy.base.log.write_log(output, log, info)`

sfepy.base.log_plotter module

Plotting class to be used by Log.

class `sfepy.base.log_plotter.LogPlotter(aggregate=100, sleep=1.0)`

LogPlotter to be used by *sfepy.base.log.Log*.

apply_commands()

make_axes()

output = **Output**

poll_draw()

process_command(command)

terminate()

`sfepy.base.log_plotter.draw_data(ax, xdata, ydata, label, plot_kwargs, swap_axes=False)`

Draw log data to a given axes, obeying *swap_axes*.

sfepy.base.mem_usage module

Memory usage functions.

`sfepy.base.mem_usage.get_mem_usage(obj, usage=None, name=None, traversal_order=None, level=0)`

Get lower bound of memory usage of an object.

Takes into account strings, numpy arrays and scipy CSR sparse matrices, descends into sequences, mappings and objects.

Parameters

obj

[any object] The object to be measured.

usage

[dict] The dict with memory usage records, serving also as a cache of already traversed objects.

name

[str] The name to be given to the object in its record.

traversal_order

[list, internal] The traversal order of the object.

level

[int, internal] The recurrence level.

Returns**usage**

[int] The object's lower bound of memory usage.

`sfepy.base.mem_usage.print_mem_usage(usage, order_by='usage', direction='up', print_key=False)`

Print memory usage dictionary.

Parameters**usage**

[dict] The dict with memory usage records.

order_by

['usage', 'name', 'kind', 'nrefs', 'traversal_order', or 'level'] The sorting field name.

direction

['up' or 'down'] The sorting direction.

print_key

[bool] If True, print also the record key (object's id).

`sfepy.base.mem_usage.raise_if_too_large(size, factor=1.0)`

Raise MemoryError if the total system memory is lower than *size* times safety *factor*. Use *factor=None* for skipping the memory check.

sfepy.base.multiproc module

Multiprocessing functions.

`sfepy.base.multiproc.get_multiproc(mpi=False)`

`sfepy.base.multiproc.get_num_workers()`

Get the number of slave nodes.

`sfepy.base.multiproc.is_remote_dict(d)`

sfepy.base.multiproc_mpi module

Multiprocessing functions.

class `sfepy.base.multiproc_mpi.MPIFileHandler(filename, mode=4, comm=<mpi4py.MPI.Intracomm object>)`

MPI file class for logging process communication.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, `_handlers`, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden `close()` methods.

emit(*record*)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using `traceback.print_exception` and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

class `sfePy.base.multiproc_mpi.MPILogFile`(*comm, filename, mode*)

close()

sync()

write(*msg*)

class `sfePy.base.multiproc_mpi.RemoteDict`(*name, mutable=False*)

Remote dictionary class - slave side.

get(*key, default=None*)

keys()

update(*other*)

class `sfePy.base.multiproc_mpi.RemoteDictMaster`(*name, mutable=False, soft_set=False, *args*)

Remote dictionary class - master side.

remote_get(*key, slave*)

remote_get_in(*key, slave*)

remote_get_keys(*slave*)

remote_get_len(*slave*)

remote_set(*data, slave, mutable=False*)

class `sfePy.base.multiproc_mpi.RemoteInt`(*remote_dict, value=None*)

Remote integer class, data saved in RemoteDict.

class `IntDesc`

value

class `sfePy.base.multiproc_mpi.RemoteLock`

Remote lock class - lock and unlock restricted access to the master.

acquire()

release()

class `sfePy.base.multiproc_mpi.RemoteQueue`(*name*)

Remote queue class - slave side.

get()

put(*value*)

```

class sfepy.base.multiproc_mpi.RemoteQueueMaster(name, mode='fifo', *args)
    Remote queue class - master side.

    clean()

    get()

    static get_gdict_key(name)

    put(value)

    remote_get(slave)

    remote_put(value, slave)

sfepy.base.multiproc_mpi.cpu_count()
    Get the number of MPI nodes.

sfepy.base.multiproc_mpi.enum(*sequential)

sfepy.base.multiproc_mpi.get_dict(name, mutable=False, clear=False, soft_set=False)
    Get the remote dictionary.

sfepy.base.multiproc_mpi.get_int_value(name, init_value=0)
    Get the remote integer value.

sfepy.base.multiproc_mpi.get_logger(log_filename='multiproc_mpi.log')
    Get the MPI logger which log information into a shared file.

sfepy.base.multiproc_mpi.get_queue(name)
    Get the queue.

sfepy.base.multiproc_mpi.get_slaves()
    Get the list of slave nodes

sfepy.base.multiproc_mpi.is_remote_dict(d)
    Return True if 'd' is RemoteDict or RemoteDictMaster instance.

sfepy.base.multiproc_mpi.master_loop()
    Run the master loop - wait for requests from slaves.

sfepy.base.multiproc_mpi.master_send_continue()
    Send 'continue' to all slaves.

sfepy.base.multiproc_mpi.master_send_task(task, data)
    Send task to all slaves.

sfepy.base.multiproc_mpi.set_logging_level(log_level='info')

sfepy.base.multiproc_mpi.slave_get_task(name='')
    Start the slave nodes.

sfepy.base.multiproc_mpi.slave_task_done(task='')
    Stop the slave nodes.

sfepy.base.multiproc_mpi.tags
    alias of Enum

sfepy.base.multiproc_mpi.wait_for_tag(wtag, num=1)

```

sfepy.base.multiproc_proc module

Multiprocessing functions - using multiprocessing (process based) module.

class sfepy.base.multiproc_proc.**MyQueue**

get()

put(value)

sfepy.base.multiproc_proc.**get_dict**(name, clear=False, **kwargs)

Get the remote dictionary.

sfepy.base.multiproc_proc.**get_int_value**(name, val0=0)

Get the remote integer value.

sfepy.base.multiproc_proc.**get_list**(name, clear=False)

Get the remote list.

sfepy.base.multiproc_proc.**get_lock**(name)

Get the global lock.

sfepy.base.multiproc_proc.**get_manager**()

Get the multiprocessing manager. If not in the global cache, create a new instance.

Returns

manager

[manager] The multiprocessing manager.

sfepy.base.multiproc_proc.**get_mpdict_value**(mode, key, clear=False)

Get the item from the global multiprocessing cache.

Parameters

mode

[str] The type of the required object.

key

[immutable type] The key of the required object.

clear

[bool] If True, clear the dictionary or list (for modes 'dict' and 'list').

Returns

value

[remote object] The remote object.

sfepy.base.multiproc_proc.**get_queue**(name)

Get the global queue.

sfepy.base.multiproc_proc.**is_remote_dict**(d)

Return True if 'd' is instance.

sfepy.base.parse_conf module

Create pyparsing grammar for problem configuration and options.

`sfepy.base.parse_conf.create_bnf(allow_tuple=False, free_word=False)`

`sfepy.base.parse_conf.cvt_array_index(toks)`

`sfepy.base.parse_conf.cvt_cmplx(toks)`

`sfepy.base.parse_conf.cvt_int(toks)`

`sfepy.base.parse_conf.cvt_none(toks)`

`sfepy.base.parse_conf.cvt_real(toks)`

`sfepy.base.parse_conf.get_standard_type_defs(word={W:(*+--9@-Z_a-z) [{{Suppress('(') Forward: None} Suppress('')} Forward: None]})`

Return dict of the pyparsing base lexical elements.

The compound types (tuple, list, dict) can contain compound types or simple types such as integers, floats and words.

Parameters**word**

[lexical element] A custom lexical element for word.

Returns**defs**

[dict] The dictionary with the following items:

- tuple: (... , ... , ...)
- list: [... , ... , ...]
- dict: {...:..., ...:..., ...} or {...=..., ...=..., ...}
- list_item: any of preceding compound types or simple types

`sfepy.base.parse_conf.list_dict(word={W:(*+--9@-Z_a-z) [{{Suppress('(') Forward: None} Suppress('')} Forward: None]})`

Return the pyparsing lexical element, that parses a string either as a list or as a dictionary.

Parameters**word**

[lexical element] A custom lexical element for word.

Returns**ld**

[lexical element] The returned lexical element parses a string in the form ..., .., ... or key1:..., key2=..., key3: ... where ... is a list_item from [get_standard_type_defs\(\)](#) and interprets it as a list or a dictionary.

`sfepy.base.parse_conf.list_of(element, *elements)`

Return lexical element that parses a list of items. The items can be a one or several lexical elements. For example, result of `list_of(real, integer)` parses list of real or integer numbers.

sfepy.base.plotutils module

`sfepy.base.plotutils.font_size(size)`

`sfepy.base.plotutils.iplot(*args, **kwargs)`

`sfepy.base.plotutils.plot_matrix_diff(mtx1, mtx2, delta, legend, mode)`

`sfepy.base.plotutils.print_matrix_diff(title, legend, mtx1, mtx2, mtx_da, mtx_dr, iis)`

`sfepy.base.plotutils.set_axes_font_size(ax, size)`

`sfepy.base.plotutils.spy(mtx, eps=None, color='b', **kwargs)`
Show sparsity structure of a *scipy.sparse* matrix.

`sfepy.base.plotutils.spy_and_show(mtx, **kwargs)`

sfepy.base.reader module

class `sfepy.base.reader.Reader(directory)`

Reads and executes a Python file as a script with `execfile()`, storing its locals. Then sets the `__dict__` of a new instance of `obj_class` to the stored locals.

Example:

```
>>> class A:
>>>     pass
```

```
>>> read = Reader( '.' )
>>> instance_of_a = read( A, 'file.py' )
```

It is equivalent to:

```
>>> mod = __import__( 'file' )
>>> instance_of_a = A()
>>> instance_of_a.__dict__.update( mod.__dict__ )
```

The first way does not create the 'file.pyc'...

sfepy.base.resolve_deps module

Functions for resolving dependencies.

`sfepy.base.resolve_deps.get_nums(deps)`
Get number of prerequisite names for each name in dependencies.

`sfepy.base.resolve_deps.remove_known(deps, known)`
Remove known names from dependencies.

`sfepy.base.resolve_deps.resolve(deps)`
Resolve dependencies among equations so that smaller blocks are solved first.

The dependencies are given in terms of variable names.

Parameters

deps

[dict] The dependencies as a dictionary with names as keys and sets of prerequisite names as values.

Returns**order**

[list] The list of blocks in the order of solving. Each block is a list of names.

`sfepy.base.resolve_deps.solvable(deps, names)`

Return True if *names* form a solvable block, i.e. the set of names equals to the set of their prerequisites.

`sfepy.base.resolve_deps.try_block(deps, num)`

Return generator of lists of solvable blocks of the length *num*.

sfepy.base.testing module

class `sfepy.base.testing.NLSStatus(**kwargs)`

Custom nonlinear solver status storing stopping condition of all time steps.

`sfepy.base.testing.assert_equal(a, b, msg='assertion of equality failed!')`

`sfepy.base.testing.check_conditions(conditions)`

`sfepy.base.testing.compare_vectors(vec1, vec2, allowed_error=1e-08, label1='vec1', label2='vec2', norm=None)`

`sfepy.base.testing.eval_coor_expression(expression, coor)`

`sfepy.base.testing.report(*argc)`

All tests should print via this function.

`sfepy.base.testing.run_declarative_example(ex_filename, define_args=None, output_dir="", ext='.vtk', remove_prefix="")`

Run a declarative example in *ex_filename* given relatively to `sfepy.base_dir`.

sfepy.base.timing module

Elapsed time measurement utilities.

class `sfepy.base.timing.Timer(name='timer', start=False)`

`add(dt)`

`reset()`

`start(reset=False)`

`stop()`

class `sfepy.base.timing.Timers(names)`

`create(name)`

`get_dts()`

get_totals()

start(*name*)

stop(*name*)

sfepy.discrete package

This package implements various PDE discretization schemes (FEM or IGA).

sfepy.discrete.conditions module

The Dirichlet, periodic and linear combination boundary condition classes, as well as the initial condition class.

class `sfepy.discrete.conditions.Condition`(*name*, ***kwargs*)

Common boundary condition methods.

canonicalize_dof_names(*dofs*)

Canonize the DOF names using the full list of DOFs of a variable.

Assumes single condition instance.

iter_single()

Create a single condition instance for each item in self.dofs and yield it.

class `sfepy.discrete.conditions.Conditions`(*objs=None*, ***kwargs*)

Container for various conditions.

canonicalize_dof_names(*dofs*)

Canonize the DOF names using the full list of DOFs of a variable.

static from_conf(*conf*, *regions*)

group_by_variables(*groups=None*)

Group boundary conditions of each variable. Each condition is a group is a single condition.

Parameters

groups

[dict, optional] If present, update the *groups* dictionary.

Returns

out

[dict] The dictionary with variable names as keys and lists of single condition instances as values.

sort()

Sort boundary conditions by their key.

zero_dofs()

Set all boundary condition values to zero, if applicable.

class `sfepy.discrete.conditions.DGEssentialBC`(**args*, *diff=0*, ***kwargs*)

This class is empty, it serves the same purpose as EssentialBC, and is created only for branching in dof_info.py

class `sfepy.discrete.conditions.DGPeriodicBC`(*name*, *regions*, *dofs*, *match*, *key=''*, *times=None*)

This class is empty, it serves the same purpose as PeriodicBC, and is created only for branching in dof_info.py


```
class sfepy.discrete.conditions.EssentialBC(name, region, dofs, key="", times=None)
```

Essential boundary condidion.

Parameters

name

[str] The boundary condition name.

region

[Region instance] The region where the boundary condition is applied.

dofs

[dict] The boundary condition specification defining the constrained DOFs and their values.

key

[str, optional] The sorting key.

times

[list or str, optional] The list of time intervals or a function returning True at time steps, when the condition applies.

zero_dofs()

Set all essential boundary condition values to zero.

```
class sfepy.discrete.conditions.InitialCondition(name, region, dofs, key="")
```

Initial condidion.

Parameters

name

[str] The initial condition name.

region

[Region instance] The region where the initial condition is applied.

dofs

[dict] The initial condition specification defining the constrained DOFs and their values.

key

[str, optional] The sorting key.

```
class sfepy.discrete.conditions.LinearCombinationBC(name, regions, dofs, dof_map_fun, kind, key="",  
                                                    times=None, arguments=None)
```

Linear combination boundary condidion.

Parameters

name

[str] The boundary condition name.

regions

[list of two Region instances] The constrained (master) DOFs region and the new (slave) DOFs region. The latter can be None if new DOFs are not field variable DOFs.

dofs

[dict] The boundary condition specification defining the constrained DOFs and the new DOFs (can be None).

dof_map_fun

[str] The name of function for mapping the constrained DOFs to new DOFs (can be None).

kind

[str] The linear combination condition kind.

key

[str, optional] The sorting key.

times

[list or str, optional] The list of time intervals or a function returning True at time steps, when the condition applies.

arguments: tuple, optional

Additional arguments, depending on the condition kind.

canonicalize_dof_names(*dofs0*, *dofs1=None*)

Canonize the DOF names using the full list of DOFs of a variable.

Assumes single condition instance.

get_var_names()

Get names of variables corresponding to the constrained and new DOFs.

class sfepy.discrete.conditions.**PeriodicBC**(*name*, *regions*, *dofs*, *match*, *key=""*, *times=None*)

Periodic boundary condition.

Parameters

name

[str] The boundary condition name.

regions

[list of two Region instances] The master region and the slave region where the DOFs should match.

dofs

[dict] The boundary condition specification defining the DOFs in the master region and the corresponding DOFs in the slave region.

match

[str] The name of function for matching corresponding nodes in the two regions.

key

[str, optional] The sorting key.

times

[list or str, optional] The list of time intervals or a function returning True at time steps, when the condition applies.

canonicalize_dof_names(*dofs*)

Canonize the DOF names using the full list of DOFs of a variable.

Assumes single condition instance.

sfepy.discrete.conditions.**get_condition_value**(*val*, *functions*, *kind*, *name*)

Check a boundary/initial condition value type and return the value or corresponding function.

sfepy.discrete.equations module

Classes of equations composed of terms.

class sfepy.discrete.equations.**Equation**(*name, terms, setup=True*)

collect_conn_info(*conn_info*)

collect_materials()

Collect materials present in the terms of the equation.

collect_variables()

Collect variables present in the terms of the equation.

Ensures that corresponding primary variables of test/parameter variables are always in the list, even if they are not directly used in the terms.

evaluate(*mode='eval', dw_mode='vector', term_mode=None, diff_vars=None, asm_obj=None, select_term=None, assemble=None*)

Evaluate the equation.

Parameters**mode**

[one of 'eval', 'el_eval', 'el_avg', 'qp', 'weak'] The evaluation mode.

dw_mode

[one of 'vector', 'matrix', 'sensitivity'] The particular evaluation mode if *mode* is 'weak'.

term_mode

[str] The term evaluation mode, used mostly if *mode* is 'eval' in some terms.

diff_vars

[list of str] The names of parameters with respect to the equation is differentiated if *dw_mode* is 'sensitivity'.

asm_obj

[ndarray or spmatrix] The object for storing the evaluation result in the 'weak' mode.

select_term

[function(term)] Optional boolean function returning True for terms that should be evaluated.

assemble

[function(*args, **kwargs)] Optional alternative FE assembling function with arguments: (equation, it, term, asm_obj, vals, iels, mode=, diff_var=), where *term* is the evaluated term, *it* is the term's order in the *equation* and (asm_obj, vals, iels, mode=, diff_var=) are the Term.assemble_to() arguments.

Returns**out**

[result] The evaluation result. In 'weak' mode it is the *asm_obj*.

static from_desc(*name, desc, variables, regions, materials, integrals, user=None, eterm_options=None, allow_derivatives=False*)

class sfepy.discrete.equations.**Equations**(*equations*)

add_equation(*equation*)

Add a new equation.

Parameters

equation

[Equation instance] The new equation.

advance(*ts*)

apply_etc(*vec=None, force_values=None*)

Apply essential (Dirichlet) boundary conditions to equations' variables, or a given vector.

apply_ic(*vec=None, force_values=None*)

Apply initial conditions to equations' variables, or a given vector.

collect_conn_info()

Collect connectivity information as defined by the equations.

collect_materials()

Collect materials present in the terms of all equations.

collect_variables()

Collect variables present in the terms of all equations.

create_matrix_graph(*any_dof_conn=False, rdc=None, cdc=None, shape=None, active_only=True, chunk_size=200000, verbose=True*)

Create tangent matrix graph, i.e. preallocate and initialize the sparse storage needed for the tangent matrix. Order of DOF connectivities is not important.

Parameters

any_dof_conn

[bool] By default, only cell region DOF connectivities are used, with the exception of trace facet DOF connectivities. If True, any DOF connectivities are used.

rdc, cdc

[arrays, optional] Additional row and column DOF connectivities, corresponding to the variables used in the equations.

shape

[tuple, optional] The required shape, if it is different from the shape determined by the equations variables. This may be needed if additional row and column DOF connectivities are passed in.

active_only

[bool] If True, the matrix graph has reduced size and is created with the reduced (active DOFs only) numbering.

chunk_size

[int] The maximum number of cells added to the graph in one [create_dof_graph\(\)](#) call.

verbose

[bool] If False, reduce verbosity.

Returns

matrix

[csr_matrix] The matrix graph in the form of a CSR matrix with preallocated structure and zero data.

create_reduced_vec()

create_subequations(*var_names*, *known_var_names=None*)

Create sub-equations containing only terms with the given virtual variables.

Parameters

var_names

[list] The list of names of virtual variables.

known_var_names

[list] The list of names of (already) known state variables.

Returns

subequations

[Equations instance] The sub-equations.

create_vec()

eval_residuals(*state*, *by_blocks=False*, *names=None*, *by_terms=False*, *select_term=None*, *assemble=None*)

Evaluate (assemble) residual vectors.

Parameters

state

[array] The vector of DOF values. Note that it is needed only in nonlinear terms.

by_blocks

[bool] If True, return the individual blocks composing the whole residual vector. Each equation should then correspond to one required block and should be named as '*block_name*, *test_variable_name*, *unknown_variable_name*'.

names

[list of str, optional] Optionally, select only blocks with the given *names*, if *by_blocks* is True.

by_terms

[bool] If True, return a dict of the individual term residuals indexed by (<equation name>, <term index>).

select_term

[function(term)] Optional boolean function returning True for terms that should be evaluated.

assemble

[function(*args, **kwargs)] Optional alternative FE assembling function with arguments: (equation, it, term, asm_obj, vals, iels, mode=, diff_var=), where *term* is the evaluated term, *it* is the term's order in the *equation* and (asm_obj, vals, iels, mode=, diff_var=) are the Term.assemble_to() arguments.

Returns

out

[array or dict of array] The assembled residual vector. If *by_blocks* is True, a dictionary is returned instead, with keys given by *block_name* part of the individual equation names.

eval_tangent_matrices(*state*, *tangent_matrix*, *by_blocks=False*, *names=None*, *by_terms=False*, *select_term=None*, *assemble=None*)

Evaluate (assemble) tangent matrices.

Parameters**state**

[array] The vector of DOF values. Note that it is needed only in nonlinear terms.

tangent_matrix

[csr_matrix] The preallocated CSR matrix with zero data.

by_blocks

[bool] If True, return the individual blocks composing the whole matrix. Each equation should then correspond to one required block and should be named as '*block_name*', '*test_variable_name*', '*unknown_variable_name*'.

names

[list of str, optional] Optionally, select only blocks with the given *names*, if *by_blocks* is True.

by_terms

[bool] If True, return a dict of the individual term matrices indexed by (<equation name>, <term index>).

select_term

[function(term)] Optional boolean function returning True for terms that should be evaluated.

assemble

[function(*args, **kwargs)] Optional alternative FE assembling function with arguments: (equation, it, term, asm_obj, vals, iels, mode=, diff_var=), where *term* is the evaluated term, *it* is the term's order in the *equation* and (asm_obj, vals, iels, mode=, diff_var=) are the Term.assemble_to() arguments.

Returns**out**

[csr_matrix or dict of csr_matrix] The assembled matrix. If *by_blocks* is True, a dictionary is returned instead, with keys given by *block_name* part of the individual equation names.

evaluate(*names=None*, *mode='eval'*, *dw_mode='vector'*, *term_mode=None*, *diff_vars=None*, *asm_obj=None*, *select_term=None*, *assemble=None*)

Evaluate the equations.

Parameters**names**

[str or sequence of str, optional] Evaluate only equations of the given name(s).

mode

[one of 'eval', 'el_avg', 'qp', 'weak'] The evaluation mode.

dw_mode

[one of 'vector', 'matrix', 'sensitivity'] The particular evaluation mode if *mode* is 'weak'.

term_mode

[str] The term evaluation mode, used mostly if *mode* is 'eval' in some terms.

diff_vars

[list of str] The names of parameters with respect to the equations are differentiated if *dw_mode* is 'sensitivity'.

asm_obj

[ndarray or spmatrix] The object for storing the evaluation result in the 'weak' mode.

select_term

[function(*term*)] Optional boolean function returning True for terms that should be evaluated.

assemble

[function(**args*, ***kwargs*)] Optional alternative FE assembling function with arguments: (equation, it, term, asm_obj, vals, iels, mode=, diff_var=), where *term* is the evaluated term, *it* is the term's order in the *equation* and (asm_obj, vals, iels, mode=, diff_var=) are the Term.assemble_to() arguments.

Returns**out**

[dict or result] The evaluation result. In 'weak' mode it is the *asm_obj*. Otherwise, it is a dict of results with equation names as keys or a single result for a single equation.

static from_conf(*conf*, *variables*, *regions*, *materials*, *integrals*, *user*=None, *eterm_options*=None, *allow_derivatives*=False, *verbose*=True)

get_domain()

get_graph_conns(*any_dof_conn*=False, *rdcs*=None, *cdcs*=None, *irs*=None, *ics*=None, *active_only*=True)

Get DOF connectivities needed for creating tangent matrix graph.

Parameters**any_dof_conn**

[bool] By default, only cell DOF connectivities are used, with the exception of trace facet DOF connectivities. If True, any kind of DOF connectivities is allowed.

rdcs, cdcs

[list of arrays, optional] Additional row and column DOF connectivities, corresponding to the variables used in the equations.

irs, ics: list of ints, optional

Additional row and column block indices for each row and column connectivity pair.

active_only

[bool] If True, the active DOF connectivities have reduced size and are created with the reduced (active DOFs only) numbering.

Returns**rdcs, cdcs**

[list of arrays] The row and column DOF connectivities defining the matrix graph blocks.

irs, ics

[list of ints] Row and column block indices for each row and column connectivity pair.

get_lcbc_operator()**get_variable**(*name*)**get_variable_dependencies()**

For each virtual variable get names of state/parameter variables that are present in terms with that virtual variable.

The virtual variables define the actual equations and their dependencies define the variables needed to evaluate the equations.

Returns

deps

[dict] The dependencies as a dictionary with virtual variable names as keys and sets of state/parameter variables as values.

get_variable_names()

Return the list of names of all variables used in equations.

init_state(*vec=None*)**init_time**(*ts*)**invalidate_term_caches()**

Invalidate evaluate caches of variables present in equations.

make_full_vec(*svec, force_value=None*)

Make a full DOF vector satisfying E(P)BCs from a reduced DOF vector.

print_terms()

Print names of equations and their terms.

reduce_vec(*vec, follow_epbc=False*)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

Notes

If 'follow_epbc' is True, values of EPBC master dofs are not simply thrown away, but added to the corresponding slave dofs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

reset_materials()

Clear material data so that next materials.time_update() is performed even for stationary materials.

set_data(*data, step=0, ignore_unknown=False*)

Set data (vectors of DOF values) of variables.

Parameters

data

[dict] The dictionary of {variable_name : data vector}.

step

[int, optional] The time history step, 0 (default) = current.

ignore_unknown

[bool, optional] Ignore unknown variable names if *data* is a dict.

set_state(*vec, reduced=False, force=False, preserve_caches=False*)**setup_initial_conditions**(*ics, functions=None*)**time_update**(*ts, ebc=None, epbc=None, lcbs=None, functions=None, problem=None, active_only=True, verbose=True*)

Update the equations for current time step.

The update involves creating the mapping of active DOFs from/to all DOFs for all state variables, the setup of linear combination boundary conditions operators and the setup of active DOF connectivities.

Parameters

ts
[TimeStepper instance] The time stepper.

ebcs
[Conditions instance, optional] The essential (Dirichlet) boundary conditions.

epbcs
[Conditions instance, optional] The periodic boundary conditions.

lbcs
[Conditions instance, optional] The linear combination boundary conditions.

functions
[Functions instance, optional] The user functions for boundary conditions, materials, etc.

problem
[Problem instance, optional] The problem that can be passed to user functions as a context.

active_only
[bool] If True, the active DOF connectivities and matrix graph have reduced size and are created with the reduced (active DOFs only) numbering.

verbose
[bool] If False, reduce verbosity.

Returns

graph_changed
[bool] The flag set to True if the current time step set of active boundary conditions differs from the set of the previous time step.

time_update_materials(*ts, mode='normal', problem=None, verbose=True*)

Update data materials for current time and possibly also state.

Parameters

ts
[TimeStepper instance] The time stepper.

mode
['normal', 'update' or 'force'] The update mode, see [sfepy.discrete.materials.Material.time_update\(\)](#).

problem
[Problem instance, optional] The problem that can be passed to user functions as a context.

verbose
[bool] If False, reduce verbosity.

`sfepy.discrete.equations.create_dof_graph(rdc, cdc, shape, active_only=False)`

Create the DOF graph corresponding to a row and column DOF connectivity.

`sfepy.discrete.equations.create_matrix_graph(rdcs, cdcs, irs, ics, rdi, cdi, active_only=True, chunk_size=200000)`

Created the matrix graph using (active) DOF connectivities.

Parameters

rdcs, cdcs
[list of arrays] Row and column DOF connectivities, corresponding to the variables used in the equations.

irs, ics

[list of ints] Row and column block indices for each row and column connectivity pair.

rdi, cdi

[DofInfo] Row and column DOF info.

active_only

[bool] If True, the matrix graph has reduced size and is created with the reduced (active DOFs only) numbering.

chunk_size

[int] The maximum number of cells added to the graph in one `create_dof_graph()` call.

Returns

graph

[boolean csr_matrix] The matrix graph.

`sfepy.discrete.equations.get_expression_arg_names(expression, strip_dots=True)`

Parse expression and return set of all argument names. For arguments with attribute-like syntax (e.g. materials), if `strip_dots` is True, only base argument names are returned.

`sfepy.discrete.equations.parse_definition(equation_def)`

Parse equation definition string to create term description list.

sfepy.discrete.evaluate module

class sfepy.discrete.evaluate.Evaluator(*problem, matrix_hook=None, assemble=None*)

This class provides the functions required by a nonlinear solver for a given problem.

eval_residual(*vec, is_full=False, select_term=None*)

eval_tangent_matrix(*vec, mtx=None, is_full=False, select_term=None*)

make_full_vec(*vec*)

static new_ulf_iteration(*problem, nls, vec, it, err, err0*)

`sfepy.discrete.evaluate.apply_ebc_to_matrix(mtx, ebc_rows, epbc_rows=None)`

Apply E(P)BC to matrix rows: put 1 to the diagonal for EBC DOFs, 1 to the diagonal for master EPBC DOFs, -1 to the [master, slave] entries. It is assumed, that the matrix contains zeros in EBC and master EPBC DOFs rows and columns.

When used within a nonlinear solver, the actual values on the EBC DOFs diagonal positions do not matter, as the residual is zero at those positions.

`sfepy.discrete.evaluate.assemble_by_blocks(conf_equations, problem, ebcs=None, epbcs=None, dw_mode='matrix', active_only=True)`

Instead of a global matrix, return its building blocks as defined in `conf_equations`. The name and row/column variables of each block have to be encoded in the equation's name, as in:

```
conf_equations = {
    'A,v,u' : "dw_lin_elastic.i1.Y2( inclusion.D, v, u )",
}
```

Notes

ebcs, *epbcs* must be either lists of BC names, or BC configuration dictionaries.

```
sfepy.discrete.evaluate.create_evaluable(expression, fields, materials, variables, integrals,
                                         regions=None, ebcs=None, epbcs=None, lbcs=None,
                                         ts=None, functions=None, auto_init=False, mode='eval',
                                         extra_args=None, active_only=True, eterm_options=None,
                                         verbose=True, kwargs=None)
```

Create evaluable object (equations and corresponding variables) from the *expression* string.

Parameters

expression

[str] The expression to evaluate.

fields

[dict] The dictionary of fields used in *variables*.

materials

[Materials instance] The materials used in the expression.

variables

[Variables instance] The variables used in the expression.

integrals

[Integrals instance] The integrals to be used.

regions

[Region instance or list of Region instances] The region(s) to be used. If not given, the regions defined within the fields domain are used.

ebcs

[Conditions instance, optional] The essential (Dirichlet) boundary conditions for 'weak' mode.

epbcs

[Conditions instance, optional] The periodic boundary conditions for 'weak' mode.

lbcs

[Conditions instance, optional] The linear combination boundary conditions for 'weak' mode.

ts

[TimeStepper instance, optional] The time stepper.

functions

[Functions instance, optional] The user functions for boundary conditions, materials etc.

auto_init

[bool] Set values of all variables to all zeros.

mode

[one of 'eval', 'el_avg', 'qp', 'weak'] The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el_avg' element averages and 'eval' means integration over each term region.

extra_args

[dict, optional] Extra arguments to be passed to terms in the expression.

active_only

[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

eterm_options

[dict, optional] The einsum-based terms evaluation options.

verbose

[bool] If False, reduce verbosity.

kwargs

[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression.

Returns**equation**

[Equation instance] The equation that is ready to be evaluated.

variables

[Variables instance] The variables used in the equation.

`sfepy.discrete.evaluate.eval_equations(equations, variables, names=None, preserve_caches=False, mode='eval', dw_mode='vector', term_mode=None, active_only=True, any_dof_conn=False, verbose=True)`

Evaluate the equations.

Parameters**equations**

[Equations instance] The equations returned by `create_evaluable()`.

variables

[Variables instance] The variables returned by `create_evaluable()`.

names

[str or sequence of str, optional] Evaluate only equations of the given name(s).

preserve_caches

[bool] If True, do not invalidate evaluate caches of variables.

mode

[one of ‘eval’, ‘el_avg’, ‘qp’, ‘weak’] The evaluation mode - ‘weak’ means the finite element assembling, ‘qp’ requests the values in quadrature points, ‘el_avg’ element averages and ‘eval’ means integration over each term region.

dw_mode

[‘vector’ or ‘matrix’] The assembling mode for ‘weak’ evaluation mode.

term_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

active_only

[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

any_dof_conn

[bool] If True, in ‘weak’ *mode* and ‘matrix’ *dw_mode*, all DOF connectivities are used to pre-allocate the matrix graph. If False, only cell region connectivities are used.

verbose

[bool] If False, reduce verbosity.

Returns**out**

[dict or result] The evaluation result. In ‘weak’ mode it is the vector or sparse matrix, depending on *dw_mode*. Otherwise, it is a dict of results with equation names as keys or a single result for a single equation.

`sfepy.discrete.evaluate.eval_in_els_and_qp`(*expression, iels, coors, fields, materials, variables, functions=None, mode='eval', term_mode=None, extra_args=None, active_only=True, verbose=True, kwargs=None*)

Evaluate an expression in given elements and points.

Parameters**expression**

[str] The expression to evaluate.

fields

[dict] The dictionary of fields used in *variables*.

materials

[Materials instance] The materials used in the expression.

variables

[Variables instance] The variables used in the expression.

functions

[Functions instance, optional] The user functions for materials etc.

mode

[one of ‘eval’, ‘el_avg’, ‘qp’] The evaluation mode - ‘qp’ requests the values in quadrature points, ‘el_avg’ element averages and ‘eval’ means integration over each term region.

term_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

extra_args

[dict, optional] Extra arguments to be passed to terms in the expression.

active_only

[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

verbose

[bool] If False, reduce verbosity.

kwargs

[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression.

Returns**out**

[array] The result of the evaluation.

sfepy.discrete.evaluate_variable module

sfepy.discrete.evaluate_variable.**eval_complex**(*vec, conn, geo, mode, shape, bf=None*)

Evaluate basic derived quantities of a complex variable given its DOF vector, connectivity and reference mapping.

sfepy.discrete.evaluate_variable.**eval_real**(*vec, conn, geo, mode, shape, bf=None*)

Evaluate basic derived quantities of a real variable given its DOF vector, connectivity and reference mapping.

sfepy.discrete.functions module

class sfepy.discrete.functions.**ConstantFunction**(*values, no_tile=False*)

Function with constant values.

class sfepy.discrete.functions.**ConstantFunctionByRegion**(*values*)

Function with constant values in regions.

class sfepy.discrete.functions.**Function**(*name, function, is_constant=False, extra_args=None*)

Base class for user-defined functions.

set_extra_args(***extra_args*)

set_function(*function, is_constant=False*)

class sfepy.discrete.functions.**Functions**(*objs=None, **kwargs*)

Container to hold all user-defined functions.

static from_conf(*conf*)

sfepy.discrete.functions.**make_sfepy_function**(*fun_or_name=None*)

Convenience decorator to quickly create [*sfepy.discrete.functions.Function*](#) objects.

Has two modes of use either without parameter:

```
@make_sfepy_function
def my_function(...):
    ...
```

or with name:

```
@make_sfepy_function("new_name_for_my_function")
def my_function(...):
    ...
```

Parameters

fun_or_name

[string, optional] Name to be saved within *Function* instance, if None name of decorated function is used.

Returns

new_fun

[*sfepy.discrete.functions.Function*] With attribute name set to provided name or original function name.

sfepy.discrete.integrals module

Classes for accessing quadrature points and weights for various reference element geometries.

class sfepy.discrete.integrals.**Integral**(*name*, *order=1*, *coors=None*, *weights=None*, *bounds=None*, *tp_fix=1.0*, *weight_fix=1.0*, *symmetric=False*)

Wrapper class around quadratures.

get_qp(*geometry*)

Get quadrature point coordinates and corresponding weights for given geometry. For built-in quadratures, the integration order is given by *self.order*.

Parameters

geometry

[str] The geometry key describing the integration domain, see the keys of *sfepy.discrete.quadratures.quadrature_tables*.

Returns

coors

[array] The coordinates of quadrature points.

weights: array

The quadrature weights.

integrate(*function*, *order=1*, *geometry='1_2'*)

Integrate numerically a given scalar function.

Parameters

function

[callable(coors)] The function of space coordinates to integrate.

order

[int, optional] The integration order. For tensor product geometries, this is the 1D (line) order.

geometry

[str] The geometry key describing the integration domain. Default is '1_2', i.e. a line integral in [0, 1]. For other values see the keys of *sfepy.discrete.quadratures.quadrature_tables*.

Returns

val

[float] The value of the integral.

class sfepy.discrete.integrals.**Integrals**(*objs=None*, ***kwargs*)

Container for instances of [Integral](#).

static from_conf(*conf*)

get(*name*)

Return existing or new integral.

Parameters

name

[str] The name can either be a non-negative integer, a string representation of a non-negative integer (the integral order) or 'a' (automatic order) or a string beginning with 'i' (existing custom integral name).

sfepy.discrete.materials module

```
class sfepy.discrete.materials.Material(name, kind='time-dependent', function=None, values=None,
                                         flags=None, **kwargs)
```

A class holding constitutive and other material parameters.

Example input:

```
material_2 = {
    'name' : 'm',
    'values' : {'E' : 1.0},
}
```

Material parameters are passed to terms using the dot notation, i.e. 'm.E' in our example case.

```
__init__(name, kind='time-dependent', function=None, values=None, flags=None, **kwargs)
```

A material is defined either by a function, or by a set of constant values, potentially distinct per region. Therefore, either *function* must be specified, or a combination of *values* and ***kwargs*.

For constant materials, ***kwargs* are simply combined with *values* into a dictionary mapping material parameter names to parameter values. The parameter values may either be specified as a constant value, or as another dictionary mapping region names to constant values (see [sfepy.discrete.functions.ConstantFunctionByRegion](#)).

Special material parameters, that are not evaluated in quadrature points - for example flags or geometry independent data - are denoted by parameter names starting with '.' - in this case the *values* argument need to be used, or a function that returns the parameters when `mode == 'special'`.

Parameters**name**

[str] The name of the material.

kind

['time-dependent' or 'stationary'] The kind of the material.

function

[function] The function for setting up the material values.

values

[dict] Constant material values.

flags

[dict, optional] Special flags.

****kwargs**

[keyword arguments, optional] Constant material values passed by their names.

```
static from_conf(conf, functions)
```

Construct Material instance from configuration.

```
get_constant_data(name)
```

Get constant data by name.

```
get_data(key, name)
```

name can be a dict - then a Struct instance with data as attributes named as the dict keys is returned.

```
get_keys(region_name=None)
```

Get all data keys.

Parameters**region_name**

[str] If not None, only keys with this region are returned.

iter_terms(*equations*, *only_new=True*)

Iterate terms for which the material data should be evaluated.

reduce_on_datas(*reduce_fun*, *init=0.0*)

For non-special values only!

reset()

Clear all data created by a call to `time_update()`, set `self.mode` to None.

set_all_data(*datas*)

Use the provided data, set mode to 'user'.

set_data(*key*, *qps*, *data*)

Set the material data in quadrature points.

Parameters**key**

[tuple] The (region_name, integral_name) data key.

qps

[Struct] Information about the quadrature points.

data

[dict] The material data.

set_extra_args(***extra_args*)

Extra arguments passed to the material function.

set_function(*function*)

time_update(*ts*, *equations*, *mode='normal'*, *problem=None*)

Evaluate material parameters in physical quadrature points.

Parameters**ts**

[TimeStepper instance] The time stepper.

equations

[Equations instance] The equations using the materials.

mode

['normal', 'update' or 'force'] The update mode. In 'force' mode, `self.datas` is cleared and all updates are redone. In 'update' mode, existing data are preserved and new can be added. The 'normal' mode depends on other attributes: for stationary (`self.kind == 'stationary'`) materials and materials in 'user' mode, nothing is done if `self.datas` is not empty. For time-dependent materials (`self.kind == 'time-dependent'`, the default) that are not constant, i.e., are given by a user function, 'normal' mode behaves like 'force' mode. For constant materials it behaves like 'update' mode - existing data are reused.

problem

[Problem instance, optional] The problem that can be passed to user functions as a context.

update_data(*key, ts, equations, term, problem=None*)

Update the material parameters in quadrature points.

Parameters

key

[tuple] The (region_name, integral_name) data key.

ts

[TimeStepper] The time stepper.

equations

[Equations] The equations for which the update occurs.

term

[Term] The term for which the update occurs.

problem

[Problem, optional] The problem definition for which the update occurs.

update_special_constant_data(*equations=None, problem=None*)

Update the special constant material parameters.

Parameters

equations

[Equations] The equations for which the update occurs.

problem

[Problem, optional] The problem definition for which the update occurs.

update_special_data(*ts, equations, problem=None*)

Update the special material parameters.

Parameters

ts

[TimeStepper] The time stepper.

equations

[Equations] The equations for which the update occurs.

problem

[Problem, optional] The problem definition for which the update occurs.

class sfepy.discrete.materials.**Materials**(*objs=None, **kwargs*)

static from_conf(*conf, functions, wanted=None*)

Construct Materials instance from configuration.

reset()

Clear material data so that next materials.time_update() is performed even for stationary materials.

time_update(*ts, equations, mode='normal', problem=None, verbose=True*)

Update material parameters for given time, problem, and equations.

Parameters

ts

[TimeStepper instance] The time stepper.

equations

[Equations instance] The equations using the materials.

mode

['normal', 'update' or 'force'] The update mode, see [Material.time_update\(\)](#).

problem

[Problem instance, optional] The problem that can be passed to user functions as a context.

verbose

[bool] If False, reduce verbosity.

sfepy.discrete.parse_equations module

class sfepy.discrete.parse_equations.**TermParse**

sfepy.discrete.parse_equations.**collect_term**(*term_descs*, *lc*)

sfepy.discrete.parse_equations.**create_bnf**(*term_descs*)

term_descs .. list of TermParse objects (sign, term_name, term_arg_names), where sign can be real or complex multiplier

sfepy.discrete.parse_equations.**rhs**(*lc*)

sfepy.discrete.parse_regions module

Grammar for selecting regions of a domain.

Regions serve for selection of certain parts of the computational domain represented as a finite element mesh. They are used to define the boundary conditions, the domains of terms and materials etc.

Notes

History: pre-git versions already from 13.06.2006.

sfepy.discrete.parse_regions.**create_bnf**(*stack*)

sfepy.discrete.parse_regions.**join_tokens**(*str*, *loc*, *toks*)

sfepy.discrete.parse_regions.**print_leaf**(*level*, *op*)

sfepy.discrete.parse_regions.**print_op**(*level*, *op*, *item1*, *item2*)

sfepy.discrete.parse_regions.**print_stack**(*stack*)

sfepy.discrete.parse_regions.**replace**(*what*, *keep=False*)

sfepy.discrete.parse_regions.**replace_with_region**(*what*, *r_index*)

sfepy.discrete.parse_regions.**to_stack**(*stack*)

sfepy.discrete.parse_regions.**visit_stack**(*stack*, *op_visitor*, *leaf_visitor*)

sfePy.discrete.probes module

Classes for probing values of Variables, for example, along a line.

class `sfePy.discrete.probes.CircleProbe`(*centre, normal, radius, n_point, share_geometry=True*)

Probe variables along a circle.

If `n_point` is positive, that number of evenly spaced points is used. If `n_point` is `None` or non-positive, an adaptive refinement based on element diameters is used and the number of points and their spacing are determined automatically. If it is negative, `-n_point` is used as an initial guess.

get_points(*refine_flag=None*)

Get the probe points.

Returns**pars**

[array_like] The independent coordinate of the probe.

points

[array_like] The probe points, parametrized by `pars`.

is_cyclic = `True`

static parse_report(*lines*)

Parse report lines to get the probe parameters.

report()

Report the probe parameters.

class `sfePy.discrete.probes.IntegralProbe`(*name, problem, expressions, labels*)

Evaluate integral expressions.

class `sfePy.discrete.probes.LineProbe`(*p0, p1, n_point, share_geometry=True*)

Probe variables along a line.

If `n_point` is positive, that number of evenly spaced points is used. If `n_point` is `None` or non-positive, an adaptive refinement based on element diameters is used and the number of points and their spacing are determined automatically. If it is negative, `-n_point` is used as an initial guess.

get_points(*refine_flag=None*)

Get the probe points.

Returns**pars**

[array_like] The independent coordinate of the probe.

points

[array_like] The probe points, parametrized by `pars`.

static parse_report(*lines*)

Parse report lines to get the probe parameters.

report()

Report the probe parameters.

class `sfePy.discrete.probes.PointsProbe`(*points, share_geometry=True*)

Probe variables in given points.

get_points(*refine_flag=None*)

Get the probe points.

Returns

pars

[array_like] The independent coordinate of the probe.

points

[array_like] The probe points, parametrized by *pars*.

static parse_report(*lines*)

Parse report lines to get the probe parameters.

refine_points(*variable, points, cache*)

No refinement for this probe.

report()

Report the probe parameters.

class sfepy.discrete.probes.**Probe**(*name, share_geometry=True, n_point=None, **kwargs*)

Base class for all point probes. Enforces two points minimum.

cache = Struct:probe_shared_evaluate_cache

get_actual_cache(*pars, cache, hash_chunk_size=100000*)

Return the actual evaluate cache, which is a combination of the (mesh-based) evaluate cache and probe-specific data, like the reference element coordinates. The reference element coordinates are reused, if the sha1 hash of the probe parameter vector does not change.

get_evaluate_cache()

Return the evaluate cache for domain-related data given by *self.share_geometry*.

is_cyclic = False

probe(*variable, mode='val', ret_points=False*)

Probe the given variable.

Parameters

variable

[Variable instance] The variable to be sampled along the probe.

mode

[{'val', 'grad'}, optional] The evaluation mode: the variable value (default) or the variable value gradient.

ret_points

[bool] If True, return also the probe points.

Returns

pars

[array] The parametrization of the probe points.

points

[array, optional] If *ret_points* is True, the coordinates of points corresponding to *pars*, where the *variable* is evaluated.

vals

[array] The probed values.

static refine_pars(*pars*, *refine_flag*, *cyclic_val=None*)

Refine the probe parametrization based on the *refine_flag*.

refine_points(*variable*, *points*, *cells*)

Mark intervals between points for a refinement, based on element sizes at those points. Assumes the points to be ordered.

Returns

refine_flag

[bool array] True at places corresponding to intervals between subsequent points that need to be refined.

report()

Report the probe parameters.

reset_refinement()

Reset the probe refinement state.

set_n_point(*n_point*)

Set the number of probe points.

Parameters

n_point

[int] The (fixed) number of probe points, when positive. When non-positive, the number of points is adaptively increased starting from -*n_point*, until the neighboring point distance is less than the diameter of the elements enclosing the points. When None, it is set to -10.

set_options(*close_limit=None*, *size_hint=None*)

Set the probe options.

Parameters

close_limit

[float] The maximum limit distance of a point from the closest element allowed for extrapolation.

size_hint

[float] Element size hint for the refinement of probe parametrization.

class sfepy.discrete.probes.**RayProbe**(*p0*, *dirvec*, *p_fun*, *n_point*, *both_dirs*, *share_geometry=True*)

Probe variables along a ray. The points are parametrized by a function of radial coordinates from a given point in a given direction.

gen_points(*sign*)

Generate the probe points and their parametrization.

get_points(*refine_flag=None*)

Get the probe points.

Returns

pars

[array_like] The independent coordinate of the probe.

points

[array_like] The probe points, parametrized by *pars*.

static parse_report(*lines*)

Parse report lines to get the probe parameters.

refine_points(*variable, points, cache*)

No refinement for this probe.

report()

Report the probe parameters.

`sfepy.discrete.probes.get_data_name(fd)`

Try to read next data name in file *fd*.

Returns

name

[str] The data name.

nc

[int] The number of data columns.

`sfepy.discrete.probes.parse_scalar(line)`

`sfepy.discrete.probes.parse_vector(line)`

`sfepy.discrete.probes.read_header(fd)`

Read the probe data header from file descriptor *fd*.

Returns

header

[Struct instance] The probe data header.

`sfepy.discrete.probes.read_results(filename, only_names=None)`

Read probing results from a file.

Parameters

filename

[str or file object] The probe results file name.

Returns

header

[Struct instance] The probe data header.

results

[dict] The dictionary of probing results. Keys are data names, values are the probed values.

`sfepy.discrete.probes.write_results(filename, probe, results)`

Write probing results into a file.

Parameters

filename

[str or file object] The output file name.

probe

[Probe subclass instance] The probe used to obtain the results.

results

[dict] The dictionary of probing results. Keys are data names, values are the probed values.

sfepy.discrete.problem module

class sfepy.discrete.problem.**Problem**(name, conf=None, functions=None, domain=None, fields=None, equations=None, auto_conf=True, active_only=True)

Problem definition, the top-level class holding all data necessary to solve a problem.

It can be constructed from a [ProblemConf](#) instance using *Problem.from_conf()* or directly from a problem description file using *Problem.from_conf_file()*

For interactive use, the constructor requires only the *equations*, *nls* and *ls* keyword arguments, see below.

Parameters

name

[str] The problem name.

conf

[ProblemConf instance, optional] The [ProblemConf](#) describing the problem.

functions

[Functions instance, optional] The user functions for boundary conditions, materials, etc.

domain

[Domain instance, optional] The solution [Domain](#).

fields

[dict, optional] The dictionary of [Field](#) instances.

equations

[Equations instance, optional] The [Equations](#) to solve. This argument is required when *auto_conf* is True.

auto_conf

[bool] If True, fields and domain are determined by *equations*.

active_only

[bool] If True, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs, see below.

Notes

The Problem is by default created with *active_only* set to True. Then the (tangent) matrices and residual vectors (right-hand sides) have reduced sizes and contain only the active DOFs, i.e., DOFs not constrained by EBCs or EPBCs.

Setting *active_only* to False results in full-size vectors and matrices. Then the matrix size non-zeros structure does not depend on the actual E(P)BCs applied. It must be False when using parallel PETSc solvers.

The active DOF connectivities contain all DOFs, with the E(P)BC-constrained ones stored as *-1 - <DOF number>*, so that the full connectivities can be reconstructed for the matrix graph creation. However, the negative entries mean that the assembled matrices/residuals have zero values at positions corresponding to constrained DOFs.

The resulting linear system then provides a solution increment, that has to be added to the initial guess used to compute the residual, just like in the Newton iterations. The increment of the constrained DOFs is automatically zero.

When solving with a direct solver, the diagonal entries of a matrix at positions corresponding to constrained DOFs has to be set to ones, so that the matrix is not singular, see [sfepy.discrete.evaluate.apply_ebc_to_matrix\(\)](#), which is called automatically in [sfepy.discrete.evaluate.Evaluator](#).

`eval_tangent_matrix()`. It is not called automatically in `Problem.evaluate()`. Note that setting the diagonal entries to one might not be necessary with iterative solvers, as the zero matrix rows match the zero residual rows, i.e. if the reduced matrix would be regular, then the right-hand side (the residual) is orthogonal to the kernel of the matrix.

advance(*ts=None*)

block_solve(*state0=None, status=None, save_results=True, step_hook=None, post_process_hook=None, report_nls_status=False, log_nls_status=False, verbose=True*)

Call `Problem.solve()` sequentially for the individual matrix blocks of a block-triangular matrix. It is called by `Problem.solve()` if the ‘*block_solve*’ option is set to True.

clear_equations()

copy(*name=None*)

Make a copy of Problem.

create_evaluable(*expression, try_equations=True, auto_init=False, preserve_caches=False, copy_materials=True, integrals=None, ebcs=None, epbcs=None, lcbcs=None, ts=None, functions=None, mode='eval', var_dict=None, strip_variables=True, extra_args=None, active_only=True, eterm_options=None, verbose=True, **kwargs*)

Create evaluable object (equations and corresponding variables) from the *expression* string. Convenience function calling `create_evaluable()` with defaults provided by the Problem instance *self*.

The evaluable can be repeatedly evaluated by calling `eval_equations()`, e.g. for different values of variables.

Parameters

expression

[str] The expression to evaluate.

try_equations

[bool] Try to get variables from *self.equations*. If this fails, variables can either be provided in *var_dict*, as keyword arguments, or are created automatically according to the expression.

auto_init

[bool] Set values of all variables to all zeros.

preserve_caches

[bool] If True, do not invalidate evaluate caches of variables.

copy_materials

[bool] Work with a copy of *self.equations.materials* instead of reusing them. Safe but can be slow.

integrals

[Integrals instance, optional] The integrals to be used. Automatically created as needed if not given.

ebcs

[Conditions instance, optional] The essential (Dirichlet) boundary conditions for ‘weak’ mode. If not given, *self.ebcs* are used.

epbcs

[Conditions instance, optional] The periodic boundary conditions for ‘weak’ mode. If not given, *self.epbcs* are used.

lcbcs

[Conditions instance, optional] The linear combination boundary conditions for ‘weak’ mode. If not given, *self.lcbcs* are used.

ts

[TimeStepper instance, optional] The time stepper. If not given, *self.ts* is used.

functions

[Functions instance, optional] The user functions for boundary conditions, materials etc. If not given, *self.functions* are used.

mode

[one of 'eval', 'el_avg', 'qp', 'weak'] The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el_avg' element averages and 'eval' means integration over each term region.

var_dict

[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression. Use this if the name of a variable conflicts with one of the parameters of this method.

strip_variables

[bool] If False, the variables in *var_dict* or *kwargs* not present in the expression are added to the actual variables as a context.

extra_args

[dict, optional] Extra arguments to be passed to terms in the expression.

active_only

[bool] If True, in 'weak' mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

eterm_options

[dict, optional] The einsum-based terms evaluation options.

verbose

[bool] If False, reduce verbosity.

****kwargs**

[keyword arguments] Additional variables can be passed as keyword arguments, see *var_dict*.

Returns**equations**

[Equations instance] The equations that can be evaluated.

variables

[Variables instance] The corresponding variables. Set their values and use *eval_equations()*.

Examples

problem is Problem instance.

```
>>> out = problem.create_evaluable('ev_integrate.il.Omega(u)')
>>> equations, variables = out
```

vec is a vector of coefficients compatible with the field of 'u' - let's use all ones.

```
>>> vec = nm.ones((variables['u'].n_dof,), dtype=nm.float64)
>>> variables['u'].set_data(vec)
>>> vec_qp = eval_equations(equations, variables, mode='qp')
```

Try another vector:

```
>>> vec = 3 * nm.ones((variables['u'].n_dof,), dtype=nm.float64)
>>> variables['u'].set_data(vec)
>>> vec_qp = eval_equations(equations, variables, mode='qp')
```

create_materials(*mat_names=None*)

Create materials with names in *mat_names*. Their definitions have to be present in *self.conf.materials*.

Notes

This method does not change *self.equations*, so it should not have any side effects.

create_state()

create_subproblem(*var_names, known_var_names*)

Create a sub-problem with equations containing only terms with the given virtual variables.

Parameters

var_names

[list] The list of names of virtual variables.

known_var_names

[list] The list of names of (already) known state variables.

Returns

subpb

[Problem instance] The sub-problem.

create_variables(*var_names=None*)

Create variables with names in *var_names*. Their definitions have to be present in *self.conf.variables*.

Notes

This method does not change *self.equations*, so it should not have any side effects.

eval_equations(*names=None, preserve_caches=False, mode='eval', dw_mode='vector', term_mode=None, active_only=True, any_dof_conn=False, verbose=True*)

Evaluate (some of) the problem's equations, convenience wrapper of *eval_equations()*.

Parameters

names

[str or sequence of str, optional] Evaluate only equations of the given name(s).

preserve_caches

[bool] If True, do not invalidate evaluate caches of variables.

mode

[one of 'eval', 'el_avg', 'qp', 'weak'] The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el_avg' element averages and 'eval' means integration over each term region.

dw_mode

['vector' or 'matrix'] The assembling mode for 'weak' evaluation mode.

term_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

verbose

[bool] If False, reduce verbosity.

Returns**out**

[dict or result] The evaluation result. In 'weak' mode it is the vector or sparse matrix, depending on *dw_mode*. Otherwise, it is a dict of results with equation names as keys or a single result for a single equation.

evaluate(*expression*, *try_equations*=True, *auto_init*=False, *preserve_caches*=False, *copy_materials*=True, *integrals*=None, *ebcs*=None, *epbcs*=None, *lbcs*=None, *ts*=None, *functions*=None, *mode*='eval', *dw_mode*='vector', *term_mode*=None, *var_dict*=None, *strip_variables*=True, *ret_variables*=False, *any_dof_conn*=False, *active_only*=True, *eterm_options*=None, *verbose*=True, *extra_args*=None, ***kwargs*)

Evaluate an expression, convenience wrapper of [*Problem.create_evaluable\(\)*](#) and [*eval_equations\(\)*](#).

Parameters**dw_mode**

['vector' or 'matrix'] The assembling mode for 'weak' evaluation mode.

term_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

ret_variables

[bool] If True, return the variables that were created to evaluate the expression.

other

[arguments] See docstrings of [*Problem.create_evaluable\(\)*](#).

Returns**out**

[array] The result of the evaluation.

variables

[Variables instance] The variables that were created to evaluate the expression. Only provided if *ret_variables* is True.

static from_conf(*conf*, *init_fields*=True, *init_equations*=True, *init_solvers*=True)

static from_conf_file(*conf_filename*, *required*=None, *other*=None, *init_fields*=True, *init_equations*=True, *init_solvers*=True)

get_default_ts(*t0*=None, *t1*=None, *dt*=None, *n_step*=None, *step*=None)

get_dim(*get_sym*=False)

Returns mesh dimension, symmetric tensor dimension (if *get_sym* is True).

get_ebc_indices()

Get indices of E(P)BC-constrained DOFs in the full global state vector.

get_evaluator(*reuse=False*)

Either create a new Evaluator instance (*reuse == False*), or return an existing instance, created in a preceding call to `Problem.init_solvers()`.

get_initial_state(*vec=None*)

Create a zero state and apply initial conditions.

get_integrals(*names=None*)

Get integrals, initialized from problem configuration if available.

Parameters

names

[list, optional] If given, only the named integrals are returned.

Returns

integrals

[Integrals instance] The requested integrals.

get_ls()

get_materials()

get_mesh_coors(*actual=False*)

get_meshes_from_region(*reg_names*)

get_nls()

get_nls_functions()

Returns functions to be used by a nonlinear solver to evaluate the nonlinear function value (the residual) and its gradient (the tangent matrix) corresponding to the problem equations.

Returns

fun

[function] The function `fun(x)` for computing the residual.

fun_grad

[function] The function `fun_grad(x)` for computing the tangent matrix.

iter_hook

[function] The optional (user-defined) function to be called before each nonlinear solver iteration iteration.

get_output_name(*suffix=None, extra=None, mode=None*)

Return default output file name, based on the output directory, output format, step suffix and mode. If present, the extra string is put just before the output format suffix.

get_restart_filename(*ts=None*)

If restarts are allowed in problem definition options, return the restart file name, based on the output directory and time step.

get_solver()

get_solver_conf(*name*)

get_timestepper()

get_tss()

```
get_tss_functions(update_bcs=True, update_materials=True, save_results=True, step_hook=None,  
                  post_process_hook=None)
```

Get the problem-dependent functions required by the time-stepping solver during the solution process.

Parameters

update_bcs

[bool, optional] If True, update the boundary conditions in each *prestep_fun* call.

update_materials

[bool, optional] If True, update the values of material parameters in each *prestep_fun* call.

save_results

[bool, optional] If True, save the results in each *poststep_fun* call.

step_hook

[callable, optional] The optional user-defined function that is called in each *poststep_fun* call before saving the results.

post_process_hook

[callable, optional] The optional user-defined function that is passed in each *poststep_fun* to [Problem.save_state\(\)](#).

Returns

init_fun

[callable] The initialization function called before the actual time-stepping.

prestep_fun

[callable] The function called in each time (sub-)step prior to the nonlinear solver call.

poststep_fun

[callable] The function called at the end of each time step.

```
get_variables(auto_create=False)
```

```
init_solvers(status=None, ls_conf=None, nls_conf=None, tsc_conf=None, ts_conf=None, force=False)
```

Create and initialize solver instances.

Parameters

status

[dict-like, IndexedStruct, optional] The user-supplied object to hold the time-stepping/nonlinear solver convergence statistics.

ls_conf

[Struct, optional] The linear solver options.

nls_conf

[Struct, optional] The nonlinear solver options.

tsc_conf

[Struct, optional] The time step controller options.

ts_conf

[Struct, optional] The time stepping solver options.

force

[bool] If True, re-create the solver instances even if they already exist in *self.nls* attribute.

```
init_time(ts)
```

is_linear()

load_restart()(*filename*, *ts=None*)

Load the current state and time step from a restart file.

Alternatively, a regular output file in the HDF5 format can be used in place of the restart file. In that case the restart is only approximate, because higher order field DOFs (if any) were stripped out. Files with the adaptive linearization are not supported. Use with caution!

Parameters

filename

[str] The restart file name.

ts

[TimeStepper instance, optional] The time stepper. If not given, a default one is created. Otherwise, it is modified in place.

Returns

variables

[Variables instance] The loaded variables.

refine_uniformly()(*level*)

Refine the mesh uniformly *level*-times.

Notes

This operation resets almost everything (fields, equations, ...) - it is roughly equivalent to creating a new Problem instance with the refined mesh.

remove_bcs()

Convenience function to remove boundary conditions.

reset()

save_ebc()(*filename*, *ebcs=None*, *epbcs=None*, *force=True*, *default=0.0*)

Save essential boundary conditions as state variables.

Parameters

filename

[str] The output file name.

ebcs

[Conditions instance, optional] The essential (Dirichlet) boundary conditions. If not given, *self.conf.ebcs* are used.

epbcs

[Conditions instance, optional] The periodic boundary conditions. If not given, *self.conf.epbcs* are used.

force

[bool] If True, sequential nonzero values are forced to individual *ebcs* so that the conditions are visible even when zero.

default

[float] The default constant value of state vector.

save_regions(*filename_trunk*, *region_names=None*)

Save regions as meshes.

Parameters

filename_trunk

[str] The output filename without suffix.

region_names

[list, optional] If given, only the listed regions are saved.

save_regions_as_groups(*filename_trunk*, *region_names=None*)

Save regions in a single mesh but mark them by using different element/node group numbers.

See `Domain.save_regions_as_groups()` for more details.

Parameters

filename_trunk

[str] The output filename without suffix.

region_names

[list, optional] If given, only the listed regions are saved.

save_restart(*filename*, *ts=None*)

Save the current state and time step to a restart file.

Parameters

filename

[str] The restart file name.

ts

[TimeStepper instance, optional] The time stepper. If not given, a default one is created.

Notes

Does not support terms with internal state.

save_state(*filename*, *state=None*, *out=None*, *fill_value=None*, *post_process_hook=None*,
linearization=None, *split_results_by=None*, ***kwargs*)

Parameters

split_results_by

[None, 'region', 'variable'] If 'region' or 'variable', data of each region/variable are stored in a separate file. If None, it is set to the application option value.

linearization

[Struct or None] The linearization configuration for higher order approximations. If its kind is 'adaptive', *split_results_by* is assumed 'variable'.

select_bcs(*ebc_names=None*, *epbc_names=None*, *lcbc_names=None*, *create_matrix=False*,
any_dof_conn=None)

select_materials(*material_names*, *only_conf=False*)

select_variables(*variable_names*, *only_conf=False*)

set_bcs(*ebcs=None*, *epbcs=None*, *lbcbs=None*)

Update boundary conditions.

set_conf_solvers(*conf_solvers=None, options=None*)

Choose which solvers should be used. If solvers are not set in *options*, use the ones named *ls*, *nls*, *ts* and optionally *tsc*. If such solver names do not exist, use the first of each required solver kind listed in *conf_solvers*.

set_default_state(*vec=None*)

Return variables with an initialized state.

A convenience function that obtains the problem equations' variables, initializes the state ones with zeros (default) or using *vec* and then returns the variables.

set_equations(*conf_equations=None, user=None, keep_solvers=False, make_virtual=False*)

Set equations of the problem using the *equations* problem description entry.

Fields and Regions have to be already set.

set_equations_instance(*equations, keep_solvers=False*)

Set equations of the problem to *equations*.

set_fields(*conf_fields=None*)

set_ics(*ics=None*)

Set the initial conditions to use.

set_linear(*is_linear*)

set_materials(*conf_materials=None*)

Set definition of materials.

set_mesh_coors(*coors, update_fields=False, actual=False, clear_all=True, extra_dofs=False*)

Set mesh coordinates.

Parameters

coors

[array] The new coordinates.

update_fields

[bool] If True, update also coordinates of fields.

actual

[bool] If True, update the actual configuration coordinates, otherwise the undeformed configuration ones.

set_output_dir(*output_dir=None*)

Set the directory for output files.

The directory is created if it does not exist.

set_regions(*conf_regions=None, conf_materials=None, functions=None, allow_empty=False*)

set_solver(*solver, status=None*)

Set a time-stepping or nonlinear solver to be used in [Problem.solve\(\)](#) call.

Parameters

solver

[NonlinearSolver or TimeSteppingSolver instance] The nonlinear or time-stepping solver.

status

[dict-like, optional] The user-supplied object to hold the solver convergence statistics.

Notes

A copy of the solver is used, and the nonlinear solver functions are set to those returned by [Problem.get_nls_functions\(\)](#), if not set already. If a nonlinear solver is set, a default StationarySolver instance is created automatically as the time-stepping solver. Also sets *self.ts* attribute.

If *self.conf.options.auto_transform_equations* is True (the default is False), the problem equations are automatically transformed to a form suitable for the given solver. Implemented for [ElastodynamicsBaseTS](#)-based solvers. If it is False, *solver.var_names* have to be defined.

set_variables(*conf_variables=None*)

Set definition of variables.

setup_default_output(*conf=None, options=None*)

Provide default values to *Problem.setup_output()* from *conf.options* and *options*.

setup_hooks(*options=None*)

Setup various hooks (user-defined functions), as given in *options*.

Supported hooks:

- *matrix_hook*
 - check/modify tangent matrix in each nonlinear solver iteration
- *nls_iter_hook*
 - called prior to every iteration of nonlinear solver, if the solver supports that
 - takes the Problem instance (*self*) as the first argument

setup_output(*output_filename_trunk=None, output_dir=None, output_format=None, file_format=None, float_format=None, split_results_by=None, linearization=None*)

Sets output options to given values, or uses the defaults for each argument that is None.

solve(*state0=None, status=None, force_values=None, var_data=None, update_bcs=True, update_materials=True, save_results=True, step_hook=None, post_process_hook=None, post_process_hook_final=None, report_nls_status=False, log_nls_status=False, verbose=True*)

Solve the problem equations by calling the top-level solver.

Before calling this function the top-level solver has to be set, see [Problem.set_solver\(\)](#). Also, the boundary conditions and the initial conditions (for time-dependent problems) has to be set, see [Problem.set_bcs\(\)](#), [Problem.set_ics\(\)](#).

Parameters

state0

[array, optional] If given, the initial state - then the initial conditions stored in the Problem instance are ignored. By default, the initial state is created and the initial conditions are applied automatically.

status

[dict-like, optional] The user-supplied object to hold the solver convergence statistics.

force_values

[dict of floats or float, optional] If given, the supplied values override the values of the essential boundary conditions.

var_data

[dict, optional] A dictionary of {variable_name : data vector} used to initialize parameter variables.

update_bcs

[bool, optional] If True, update the boundary conditions in each *prestep_fun* call. See [Problem.get_tss_functions\(\)](#).

update_materials

[bool, optional] If True, update the values of material parameters in each *prestep_fun* call. See [Problem.get_tss_functions\(\)](#).

save_results

[bool, optional] If True, save the results in each *poststep_fun* call. See [Problem.get_tss_functions\(\)](#).

step_hook

[callable, optional] The optional user-defined function that is called in each *poststep_fun* call before saving the results. See [Problem.get_tss_functions\(\)](#).

post_process_hook

[callable, optional] The optional user-defined function that is passed in each *poststep_fun* to [Problem.save_state\(\)](#). See [Problem.get_tss_functions\(\)](#).

post_process_hook_final

[callable, optional] The optional user-defined function that is called after the top-level solver returns.

report_nls_status: bool, optional

If True, print summary non-linear solver info.

log_nls_status: bool, optional

If True, log non-linear solver info into a CSV file.

Returns**variables**

[Variables] The variables with the final time step state.

time_update(*ts=None, ebcs=None, epbcs=None, lbcs=None, functions=None, create_matrix=False, is_matrix=True, any_dof_conn=None*)

try_presolve(*mtx*)

update_equations(*ts=None, ebcs=None, epbcs=None, lbcs=None, functions=None, create_matrix=False, is_matrix=True, any_dof_conn=None*)

Update equations for current time step.

The tangent matrix graph is automatically recomputed if the set of active essential or periodic boundary conditions changed w.r.t. the previous time step.

Parameters**ts**

[TimeStepper instance, optional] The time stepper. If not given, *self.ts* is used.

ebcs

[Conditions instance, optional] The essential (Dirichlet) boundary conditions. If not given, *self.ebcs* are used.

epbcs

[Conditions instance, optional] The periodic boundary conditions. If not given, *self.epbcs* are used.

lcbs

[Conditions instance, optional] The linear combination boundary conditions. If not given, *self.lcbs* are used.

functions

[Functions instance, optional] The user functions for boundary conditions, materials, etc. If not given, *self.functions* are used.

create_matrix

[bool] If True, force the matrix graph computation.

is_matrix

[bool] If False, the matrix is not created. Has precedence over *create_matrix*.

any_dof_conn

[bool or None, default False] If True, all DOF connectivities are used to pre-allocate the matrix graph. If False, only cell region connectivities are used. If None, the value is, if available, taken from *conf.options*.

update_materials(*ts=None, mode='normal', verbose=True*)

Update materials used in equations.

Parameters**ts**

[TimeStepper instance] The time stepper.

mode

['normal', 'update' or 'force'] The update mode, see [Material.time_update\(\)](#).

verbose

[bool] If False, reduce verbosity.

update_time_stepper(*ts*)**sfepy.discrete.problem.make_is_save**(*options*)

Given problem options, return a callable that determines whether to save results of a time step.

sfepy.discrete.problem.prepare_matrix(*problem, state*)

Pre-assemble tangent system matrix.

sfepy.discrete.projections module

Construct projections between FE spaces.

sfepy.discrete.projections.create_mass_matrix(*field*)

Create scalar mass matrix corresponding to the given field.

Returns**mtx**

[csr_matrix] The mass matrix in CSR format.

sfepy.discrete.projections.make_h1_projection_data(*target, eval_data*)

Project scalar data given by a material-like *eval_data()* function to a scalar *target* field variable using the H^1 dot product.

sfepy.discrete.projections.make_l2_projection(*target, source, ls=None, nls_options=None*)

Project a scalar *source* field variable to a scalar *target* field variable using the L^2 dot product.

`sfepy.discrete.projections.make_l2_projection_data(target, eval_data, order=None, ls=None, nls_options=None)`

Project scalar data to a scalar *target* field variable using the L^2 dot product.

Parameters

target

[FieldVariable instance] The target variable.

eval_data

[callable or array] Either a material-like function *eval_data()*, or an array of values in quadrature points that has to be reshaped to the shape required by *order*.

order

[int, optional] The quadrature order. If not given, it is set to $2 * \text{target.field.approx\_order}$.

`sfepy.discrete.projections.project_by_component(tensor, tensor_qp, component, order, ls=None, nls_options=None)`

Wrapper around `make_l2_projection_data()` for non-scalar fields.

`sfepy.discrete.projections.project_to_facets(region, fun, dpn, field)`

Project a function *fun* to the *field* in facets of the given *region*.

sfepy.discrete.quadratures module

quadrature_tables are organized as follows:

```
quadrature_tables = {
    '<geometry1>' : {
        order1 : QuadraturePoints(args1),
        order2 : QuadraturePoints(args2),
        ...
    },
    '<geometry2>' : {
        order1 : QuadraturePoints(args1),
        order2 : QuadraturePoints(args2),
        ...
    },
    ...
}
```

Note The order for quadratures on tensor product domains ('2_4', '3_8' geometries) in case of composite Gauss quadratures (products of 1D quadratures) holds for each component separately, so the actual polynomial order may be much higher (up to *order * dimension*).

Naming conventions in problem description files:

```
`<family>_<order>_<dimension>`
```

Integral 'family' is just an arbitrary name given by user.

Low order quadrature coordinates and weights copied from The Finite Element Method Displayed by Gouri Dhatt and Gilbert Touzat, Wiley-Interscience Production, 1984.

The line integral (geometry '1_2') coordinates and weights are from Abramowitz, M. and Stegun, I.A., Handbook of Mathematical Functions, Dover Publications, New York, 1972. The triangle (geometry '2_3') coordinates and weights are from Dunavant, D.A., High Degree Efficient Symmetrical Gaussian Quadrature Rules for the Triangle, Int. J. Num.

Meth. Eng., 21 (1985) pp 1129-1148 - only rules with points inside the reference triangle are used. The actual values were copied from PHAML (<http://math.nist.gov/phaml/>), see also Mitchell, W.F., PHAML User's Guide, NISTIR 7374, 2006.

Quadrature rules for the quadrilateral (geometry '2_4') and hexahedron (geometry '3_8') of order higher than 5 are computed as the tensor product of the line (geometry '1_2') rules.

Quadrature rules for the triangle (geometry '2_3') and tetrahedron (geometry '3_4') of order higher than 19 and 6, respectively follow A. Grundmann and H.M. Moeller, Invariant integration formulas for the n-simplex by combinatorial methods, SIAM J. Numer. Anal. 15 (1978), 282–290. The generating function was adapted from pytools/hegde codes (<http://mathema.tician.de/software/hegde>) by Andreas Kloeckner.

```
class sfepy.discrete.quadratures.QuadraturePoints(data, coors=None, weights=None, bounds=None,
                                                  tp_fix=1.0, weight_fix=1.0, symmetric=False)
```

Representation of a set of quadrature points.

Parameters

data

[array_like] The array of shape $(n\_point, dim + 1)$ of quadrature point coordinates (first dim columns) and weights (the last column).

coors

[array_like, optional] Optionally, instead of using *data*, the coordinates and weights can be provided separately - *data* are then ignored.

weights

[array_like, optional] Optionally, instead of using *data*, the coordinates and weights can be provided separately - *data* are then ignored.

bounds

[(float, float), optional] The coordinates and weights should correspond to a reference element in $[0, 1] \times dim$. Provide the correct bounds if this is not the case.

tp_fix

[float, optional] The value that is used to multiply the tensor product element volume (= 1.0) to get the correct volume.

weight_fix

[float, optional] The value that is used to multiply the weights to get the correct values.

symmetric

[bool] If True, the integral is 1D and the given coordinates and weights are symmetric w.r.t. the centre of bounds; only the non-negative coordinates are given.

```
static from_table(geometry, order)
```

Create a new *QuadraturePoints* instance, given reference element geometry name and polynomial order. For tensor product geometries, the polynomial order is the 1D (line) order.

```
sfepy.discrete.quadratures.get_actual_order(geometry, order)
```

Return the actual integration order for given geometry.

Parameters

geometry

[str] The geometry key describing the integration domain, see the keys of *quadrature_tables*.

Returns

order

[int] If *order* is in quadrature tables it is this value. Otherwise it is the closest higher order. If no higher order is available, a warning is printed and the highest available order is used.

sfepy.discrete.simplex_cubature module

Generate simplex quadrature points. Code taken and adapted from pytools/hedge by Andreas Kloeckner.

`sfepy.discrete.simplex_cubature.factorial(n)`

`sfepy.discrete.simplex_cubature.generate_decreasing_nonnegative_tuples_summing_to(n, length,
min=0,
max=None)`

`sfepy.discrete.simplex_cubature.generate_permutations(original)`

Generate all permutations of the list ``original``.

Nicked from <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/252178>

`sfepy.discrete.simplex_cubature.generate_unique_permutations(original)`

Generate all unique permutations of the list ``original``.

`sfepy.discrete.simplex_cubature.get_simplex_cubature(order, dimension)`

Cubature on an $M\{n\}$ -simplex.

cf. A. Grundmann and H.M. Moeller, Invariant integration formulas for the n -simplex by combinatorial methods, SIAM J. Numer. Anal. 15 (1978), 282–290.

This cubature rule has both negative and positive weights. It is exact for polynomials up to order $2s + 1$, where s is given as *order*. The integration domain is the unit simplex

$$T_n := \{(x_1, \dots, x_n) : x_i \geq -1, \sum_i x_i \leq -1\}$$

`sfepy.discrete.simplex_cubature.wandering_element(length, wanderer=1, landscape=0)`

sfepy.discrete.variables module

Classes of variables for equations/terms.

class `sfepy.discrete.variables.DGFieldVariable(name, kind, field, order=None,
primary_var_name=None, special=None, flags=None,
history=None, **kwargs)`

Field variable specifically intended for use with DGFields, bypasses application of EBC and EPBC as this is done in DGField.

Is instance checked in `create_adof_conns`.

apply_ebc(*vec*, *offset*=0, *force_values*=None)

Apply essential (Dirichlet) and periodic boundary conditions to vector *vec*, starting at *offset*.

get_full(*r_vec*, *r_offset*=0, *force_value*=None, *vec*=None, *offset*=0)

Get the full DOF vector satisfying E(P)BCs from a reduced DOF vector.

Notes

The reduced vector starts in *r_vec* at *r_offset*. Passing a *force_value* overrides the EBC values. Optionally, *vec* argument can be provided to store the full vector (in place) starting at *offset*.

class `sfePy.discrete.variables.FieldVariable`(*name*, *kind*, *field*, *order=None*, *primary_var_name=None*, *special=None*, *flags=None*, *history=None*, ***kwargs*)

A finite element field variable.

field .. field description of variable (borrowed)

apply_ebc(*vec*, *offset=0*, *force_values=None*)

Apply essential (Dirichlet) and periodic boundary conditions to vector *vec*, starting at *offset*.

apply_ic(*vec*, *offset=0*, *force_values=None*)

Apply initial conditions conditions to vector *vec*, starting at *offset*.

clear_evaluate_cache()

Clear current evaluate cache.

create_output(*vec=None*, *key=None*, *extend=True*, *fill_value=None*, *linearization=None*)

Convert the DOF vector to a dictionary of output data usable by `Mesh.write()`.

Parameters

vec

[array, optional] An alternative DOF vector to be used instead of the variable DOF vector.

key

[str, optional] The key to be used in the output dictionary instead of the variable name.

extend

[bool] Extend the DOF values to cover the whole domain.

fill_value

[float or complex] The value used to fill the missing DOF values if *extend* is True.

linearization

[Struct or None] The linearization configuration for higher order approximations.

equation_mapping(*bcs*, *var_di*, *ts*, *functions*, *problem=None*, *warn=False*)

Create the mapping of active DOFs from/to all DOFs.

Sets *n_adof*.

Returns

active_bcs

[set] The set of boundary conditions active in the current time.

evaluate(*mode='val'*, *region=None*, *integral=None*, *integration=None*, *step=0*, *time_derivative=None*, *trace_region=None*, *dt=None*, *bf=None*)

Evaluate various quantities related to the variable according to *mode* in quadrature points defined by *integral*.

The evaluated data are cached in the variable instance in *evaluate_cache* attribute.

Parameters

mode

[one of 'val', 'grad', 'div', 'cauchy_strain'] The evaluation mode.

region

[Region instance, optional] The region where the evaluation occurs. If None, the underlying field region is used.

integral

[Integral instance, optional] The integral defining quadrature points in which the evaluation occurs. If None, the first order volume integral is created. Must not be None for surface integrations.

integration

['cell', 'facet', 'facet_extra', or 'point'] The term integration type. If None, it is derived from the region kind.

step

[int, default 0] The time step (0 means current, -1 previous, ...).

time_derivative

[None or 'dt'] If not None, return time derivative of the data, approximated by the backward finite difference.

trace_region

[None or str] If not None, evaluate of trace of the variable on a boundary region.

dt

[float, optional] The time step to be used if *derivative* is 'dt'. If None, the *dt* attribute of the variable is used.

bf

[Base function, optional] The base function to be used in 'val' mode.

Returns**out**

[array] The 4-dimensional array of shape $(n_{el}, n_{qp}, n_{row}, n_{col})$ with the requested data, where n_{row}, n_{col} depend on *mode*.

evaluate_at(*coors*, *mode*='val', *strategy*='general', *close_limit*=0.1, *get_cells_fun*=None, *cache*=None, *ret_cells*=False, *ret_status*=False, *ret_ref_coors*=False, *verbose*=False)

Evaluate the variable in the given physical coordinates. Convenience wrapper around [Field.evaluate_at\(\)](#), see its docstring for more details.

get_data_shape(*integral*, *integration*='cell', *region_name*=None)

Get element data dimensions for given approximation.

Parameters**integral**

[Integral instance] The integral describing used numerical quadrature.

integration

['cell', 'facet', 'facet_extra', 'point' or 'custom'] The term integration mode.

region_name

[str] The name of the region of the integral.

Returns**data_shape**

[5 ints] The $(n_{el}, n_{qp}, dim, n_{en}, n_{comp})$ for volume shape kind, $(n_{fa}, n_{qp}, dim, n_{fn}, n_{comp})$ for surface shape kind and $(n_{nod}, 0, 0, 1, n_{comp})$ for point shape kind.

Notes

- n_{el} , n_{fa} = number of elements/facets
- n_{qp} = number of quadrature points per element/facet
- dim = spatial dimension
- n_{en} , n_{fn} = number of element/facet nodes
- n_{comp} = number of variable components in a point/node
- n_{nod} = number of element nodes

get_dof_conn(*region_name*, *dct*, *trace_region=None*)

Get active dof connectivity of a variable.

Notes

The primary and dual variables must have the same Region.

get_dof_info(*active=False*)

get_element_diameters(*cells*, *mode*, *square=False*)

Get diameters of selected elements.

get_field()

get_full(*r_vec*, *r_offset=0*, *force_value=None*, *vec=None*, *offset=0*)

Get the full DOF vector satisfying E(P)BCs from a reduced DOF vector.

Notes

The reduced vector starts in *r_vec* at *r_offset*. Passing a *force_value* overrides the EBC values. Optionally, *vec* argument can be provided to store the full vector (in place) starting at *offset*.

get_interp_coors(*strategy='interpolation'*, *interp_term=None*)

Get the physical coordinates to interpolate into, based on the strategy used.

get_mapping(*region*, *integral*, *integration*, *get_saved=False*, *return_key=False*)

Get the reference element mapping of the underlying field.

 See also

`sfepy.discrete.common.fields.Field.get_mapping`

get_reduced(*vec*, *offset=0*, *follow_epbc=False*)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

Notes

The full vector starts in *vec* at *offset*. If ‘follow_epbc’ is True, values of EPBC master DOFs are not simply thrown away, but added to the corresponding slave DOFs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

get_state_in_region(*region*, *reshape=True*, *step=0*)

Get DOFs of the variable in the given region.

Parameters

region

[Region] The selected region.

reshape

[bool] If True, reshape the DOF vector to a 2D array with the individual components as columns. Otherwise a 1D DOF array of the form [all DOFs in region node 0, all DOFs in region node 1, ...] is returned.

step

[int, default 0] The time step (0 means current, -1 previous, ...).

Returns

out

[array] The selected DOFs.

has_ebc(*vec=None*, *force_values=None*)

has_same_mesh(*other*)

Returns

flag

[int] The flag can be either ‘different’ (different meshes), ‘deformed’ (slightly deformed same mesh), or ‘same’ (same).

invalidate_evaluate_cache(*step=0*)

Invalidate variable data in evaluate cache for time step given by *step* (0 is current, -1 previous, ...).

This should be done, for example, prior to every nonlinear solver iteration.

save_as_mesh(*filename*)

Save the field mesh and the variable values into a file for visualization. Only the vertex values are stored.

set_from_function(*fun*, *step=0*)

Set the variable data (the vector of DOF values) using a function of space coordinates.

Parameters

fun

[callable] The function of coordinates returning DOF values of shape (*n_coor*, *n_components*).

step

[int, optional] The time history step, 0 (default) = current.

set_from_mesh_vertices(*data*)

Set the variable using values at the mesh vertices.

set_from_other(*other*, *strategy*='projection', *close_limit*=0.1)

Set the variable using another variable. Undefined values (e.g. outside the other mesh) are set to numpy.nan, or extrapolated.

Parameters

strategy

['projection' or 'interpolation'] The strategy to set the values: the L² orthogonal projection (not implemented!), or a direct interpolation to the nodes (nodal elements only!)

Notes

If the other variable uses the same field mesh, the coefficients are set directly.

set_from_qp(*data_qp*, *integral*, *step*=0)

Set DOFs of variable using values in quadrature points corresponding to the given integral.

setup_initial_conditions(*ics*, *di*, *functions*, *warn*=False)

Setup of initial conditions.

time_update(*ts*, *functions*)

Store time step, set variable data for variables with the setter function.

class sfepy.discrete.variables.**Variable**(*name*, *kind*, *order*=None, *primary_var_name*=None, *special*=None, *flags*=None, ***kwargs*)

advance(*ts*)

Advance in time the DOF state history. A copy of the DOF vector is made to prevent history modification.

static from_conf(*key*, *conf*, *fields*)

get_dual()

Get the dual variable.

Returns

var

[Variable instance] The primary variable for non-state variables, or the dual variable for state variables.

get_initial_condition()

get_primary()

Get the corresponding primary variable.

Returns

var

[Variable instance] The primary variable, or *self* for state variables or if *primary_var_name* is None, or None if no other variables are defined.

get_primary_name()

init_data(*step*=0)

Initialize the dof vector data of time step *step* to zeros.

init_history()

Initialize data of variables with history.

is_complex()

is_finite(*step=0, derivative=None, dt=None*)

is_kind(*kind*)

is_parameter()

is_real()

is_state()

is_state_or_parameter()

is_virtual()

set_constant(*val=0.0, step=0*)

Set the variable dof vector data of time step *step* to a scalar *val*.

set_data(*data=None, indx=None, step=0, preserve_caches=False*)

Set data (vector of DOF values) of the variable.

Parameters

data

[array] The vector of DOF values.

indx

[int, optional] If given, *data[indx]* is used.

step

[int, optional] The time history step, 0 (default) = current.

preserve_caches

[bool] If True, do not invalidate evaluate caches of the variable.

time_update(*ts, functions*)

Implemented in subclasses.

class sfepy.discrete.variables.**Variables**(*variables=None*)

Container holding instances of Variable.

advance(*ts*)

apply_ebc(*vec=None, force_values=None*)

Apply essential (Dirichlet) and periodic boundary conditions to state all variables or the given vector *vec*.

apply_ic(*vec=None, force_values=None*)

Apply initial conditions to all state variables or the given vector *vec*.

check_vec_size(*vec, reduced=False*)

Check whether the shape of the DOF vector corresponds to the total number of DOFs of the state variables.

Parameters

vec

[array] The vector of DOF values.

reduced

[bool] If True, the size of the DOF vector should be reduced, i.e. without DOFs fixed by boundary conditions.

create_output(*vec=None, fill_value=None, var_info=None, extend=True, linearization=None*)

Creates an output dictionary with state variables data, that can be passed as ‘out’ kwarg to `Mesh.write()`.

Then the dictionary entries are formed by components of the state vector corresponding to unknown variables according to kind of linearization given by *linearization*.

create_reduced_vec()

create_vec()

equation_mapping(*ebcs, epbcs, ts, functions, problem=None, active_only=True*)

Create the mapping of active DOFs from/to all DOFs for all state variables.

Parameters

ebcs

[Conditions instance] The essential (Dirichlet) boundary conditions.

epbcs

[Conditions instance] The periodic boundary conditions.

ts

[TimeStepper instance] The time stepper.

functions

[Functions instance] The user functions for boundary conditions.

problem

[Problem instance, optional] The problem that can be passed to user functions as a context.

active_only

[bool] If True, the active DOF info `self.adi` uses the reduced (active DOFs only) numbering. Otherwise it is the same as `self.di`.

Returns

active_bcs

[set] The set of boundary conditions active in the current time.

fill_state(*value*)

Fill the DOF vector with given value.

static from_conf(*conf, fields*)

get_dual_names()

Get names of pairs of dual variables.

Returns

duals

[dict] The dual names as virtual name : state name pairs.

get_idx(*var_name, reduced=False, allow_dual=False*)

get_lcbc_operator()

get_matrix_shape()

get_reduced_state(*follow_epbc=False, force=False*)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

get_state(*reduced=False, follow_epbc=False, force=False*)

get_state_parts(*vec=None*)

Return parts of a state vector corresponding to individual state variables.

Parameters

vec

[array, optional] The state vector. If not given, then the data stored in the variables are returned instead.

Returns

out

[dict] The dictionary of the state parts.

get_vec_part(*vec, var_name, reduced=False*)

has_ebc(*vec=None, force_values=None, verbose=False*)

has_virtuals()

init_history()

init_state(*vec=None*)

invalidate_evaluate_caches(*step=0*)

iter_state(*ordered=True*)

link_duals()

Link state variables with corresponding virtual variables, and assign link to self to each variable instance.

Usually, when solving a PDE in the weak form, each state variable has a corresponding virtual variable.

make_full_vec(*svec, force_value=None, vec=None*)

Make a full DOF vector satisfying E(P)BCs from a reduced DOF vector.

Parameters

svec

[array] The reduced DOF vector.

force_value

[float, optional] Passing a *force_value* overrides the EBC values.

vec

[array, optional] If given, the buffer for storing the result (zeroed).

Returns

vec

[array] The full DOF vector.

reduce_vec(*vec, follow_epbc=False, svec=None*)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

Notes

If 'follow_epbc' is True, values of EPBC master dofs are not simply thrown away, but added to the corresponding slave dofs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

set_adof_conns(*adof_conns*)

Set all active DOF connectivities to *self* as well as relevant sub-dicts to the individual variables.

set_data(*data*, *step*=0, *ignore_unknown*=False, *preserve_caches*=False)

Set data (vectors of DOF values) of variables.

Parameters

data

[array] The state vector or dictionary of {variable_name : data vector}.

step

[int, optional] The time history step, 0 (default) = current.

ignore_unknown

[bool, optional] Ignore unknown variable names if *data* is a dict.

preserve_caches

[bool] If True, do not invalidate evaluate caches of variables.

set_full_state(*vec*, *force*=False, *preserve_caches*=False)

Set the full DOF vector (including EBC and PBC DOFs). If *var_name* is given, set only the DOF sub-vector corresponding to the given variable. If *force* is True, setting variables with LCBC DOFs is allowed.

set_reduced_state(*r_vec*, *preserve_caches*=False)

Set the reduced DOF vector, with EBC and PBC DOFs removed.

Parameters

r_vec

[array] The reduced DOF vector corresponding to the variables.

preserve_caches

[bool] If True, do not invalidate evaluate caches of variables.

set_state(*vec*, *reduced*=False, *force*=False, *preserve_caches*=False, *apply_ebc*=False)

set_state_parts(*parts*, *vec*=None, *force*=False)

Set parts of the DOF vector corresponding to individual state variables.

Parameters

parts

[dict] The dictionary of the DOF vector parts.

force

[bool] If True, proceed even with LCBCs present.

set_vec_part(*vec*, *var_name*, *part*, *reduced*=False)

setup_dof_info(*make_virtual*=False)

Setup global DOF information.

setup_dtype()

Setup data types of state variables - all have to be of the same data type, one of `nm.float64` or `nm.complex128`.

setup_initial_conditions(*ics, functions*)**setup_lcbc_operators**(*lcbcs, ts=None, functions=None*)

Prepare linear combination BC operator matrix and right-hand side vector.

setup_ordering()

Setup ordering of variables.

time_update(*ts, functions, verbose=True*)**sfepy.discrete.variables.create_adof_conn**(*eq, conn, dpn, offset*)

Given a node connectivity, number of DOFs per node and equation mapping, create the active dof connectivity.

Locally (in a connectivity row), the DOFs are stored DOF-by-DOF (`u_0` in all local nodes, `u_1` in all local nodes, ...).

Globally (in a state vector), the DOFs are stored node-by-node (`u_0, u_1, ..., u_X` in node 0, `u_0, u_1, ..., u_X` in node 1, ...).

sfepy.discrete.variables.create_adof_conns(*conn_info, var_idx=None, active_only=True, verbose=True*)

Create active DOF connectivities for all variables referenced in *conn_info*.

If a variable has not the equation mapping, a trivial mapping is assumed and connectivity with all DOFs active is created.

DOF connectivity key is a tuple (primary variable name, region name, type, trace_region).

Notes

If *active_only* is False, the DOF connectivities contain all DOFs, with the E(P)BC-constrained ones stored as *-1* - *<DOF number>*, so that the full connectivities can be reconstructed for the matrix graph creation.

sfepy.discrete.variables.expand_basis(*basis, dpn*)

Expand basis for variables with several components (DOFs per node), in a way compatible with [`create\_adof\_conn\(\)`](#), according to *dpn* (DOF-per-node count).

sfepy.discrete.common sub-package

Common lower-level code and parent classes for FEM and IGA.

sfepy.discrete.common.dof_info module

Classes holding information on global DOFs and mapping of all DOFs - equations (active DOFs).

Helper functions for the equation mapping.

class sfepy.discrete.common.dof_info.DofInfo(*name*)

Global DOF information, i.e. ordering of DOFs of the state (unknown) variables in the global state vector.

append_raw(*name*, *n_dof*)

Append raw DOFs.

Parameters

name

[str] The name of variable the DOFs correspond to.

n_dof

[int] The number of DOFs.

append_variable(*var*, *active=False*, *shared=None*)

Append DOFs of the given variable.

Parameters

var

[Variable instance] The variable to append.

active

[bool, optional] When True, only active (non-constrained) DOFs are considered.

get_info(*var_name*)

Return information on DOFs of the given variable.

Parameters

var_name

[str] The name of the variable.

get_n_dof_total()

Return the total number of DOFs of all state variables.

get_subset_info(*var_names*)

Return global DOF information for selected variables only. Silently ignores non-existing variable names.

Parameters

var_names

[list] The names of the selected variables.

class sfepy.discrete.common.dof_info.**EquationMap**(*name*, *dof_names*, *var_di*)

Map all DOFs to equations for active DOFs.

get_operator()

Get the matrix operator R corresponding to the equation mapping, such that the restricted matrix A_r can be obtained from the full matrix A by $A_r = R^T A R$. All the matrices are w.r.t. a single variables that uses this mapping.

Returns

mtx

[coo_matrix] The matrix R .

map_equations(*bcs*, *field*, *ts*, *functions*, *problem=None*, *warn=False*)

Create the mapping of active DOFs from/to all DOFs.

Parameters

bcs

[Conditions instance] The Dirichlet or periodic boundary conditions (single condition instances). The dof names in the conditions must already be canonized.

field

[Field instance] The field of the variable holding the DOFs.

ts

[TimeStepper instance] The time stepper.

functions

[Functions instance] The registered functions.

problem

[Problem instance, optional] The problem that can be passed to user functions as a context.

warn

[bool, optional] If True, warn about BC on non-existent nodes.

Returns**active_bcs**

[set] The set of boundary conditions active in the current time.

Notes

- Periodic bc: master and slave DOFs must belong to the same field (variables can differ, though).

`sfepy.discrete.common.dof_info.expand_nodes_to_dofs(nods, n_dof_per_node)`

Expand DOF node indices into DOFs given a constant number of DOFs per node.

`sfepy.discrete.common.dof_info.expand_nodes_to_equations(nods, dof_names, all_dof_names)`

Expand vector of node indices to equations (DOF indices) based on the DOF-per-node count.

DOF names must be already canonized.

Returns**eq**

[array] The equations/DOF indices in the node-by-node order.

`sfepy.discrete.common.dof_info.group_chains(chain_list)`

Group EPBC chains.

`sfepy.discrete.common.dof_info.is_active_bc(bc, ts=None, functions=None)`

Check whether the given boundary condition is active in the current time.

Returns**active**

[bool] True if the condition *bc* is active.

`sfepy.discrete.common.dof_info.resolve_chains(master_slave, chains)`

Resolve EPBC chains - e.g. in corner nodes.

sfepy.discrete.common.domain module

class `sfepy.discrete.common.domain.Domain`(*name*, *mesh=None*, *nurbs=None*, *bmesh=None*,
regions=None, *verbose=False*)

create_extra_tdim_region(*region*, *functions*, *tdim*)

create_region(*name*, *select*, *kind='cell'*, *parent=None*, *check_parents=True*, *extra_options=None*,
functions=None, *add_to_regions=True*, *allow_empty=False*)

Region factory constructor. Append the new region to `self.regions` list.

create_regions(*region_defs*, *functions=None*, *allow_empty=False*)

get_centroids(*dim*)

Return the coordinates of centroids of mesh entities with dimension *dim*.

has_faces()

reset_regions()

Reset the list of regions associated with the domain.

save_regions(*filename*, *region_names=None*)

Save regions as individual meshes.

Parameters

filename

[str] The output filename.

region_names

[list, optional] If given, only the listed regions are saved.

save_regions_as_groups(*filename*, *region_names=None*)

Save regions in a single mesh but mark them by using different element/node group numbers.

If regions overlap, the result is undetermined, with exception of the whole domain region, which is marked by group id 0.

Region masks are also saved as scalar point data for output formats that support this.

Parameters

filename

[str] The output filename.

region_names

[list, optional] If given, only the listed regions are saved.

`sfepy.discrete.common.domain.region_leaf`(*domain*, *regions*, *rdef*, *functions*, *tdim*)

Create/setup a region instance according to *rdef*.

`sfepy.discrete.common.domain.region_op`(*level*, *op_code*, *item1*, *item2*)

`sfepy.discrete.common.extmods._fmfield` module

`sfepy.discrete.common.extmods._geommech` module

Low level functions.

```
sfepy.discrete.common.extmods._geommech.geme_mulAVSB3py(vs, inp)
```

`sfepy.discrete.common.extmods.assemble` module

Low level finite element assembling functions.

```
sfepy.discrete.common.extmods.assemble.assemble_matrix(mtx, rows, cols, mtx_in_els, iels, sign,
                                                         row_conn, col_conn)
```

```
sfepy.discrete.common.extmods.assemble.assemble_matrix_b(mtx, rows, cols, mtx_in_els, iels, sign,
                                                           row_conn, col_conn)
```

```
sfepy.discrete.common.extmods.assemble.assemble_matrix_complex(mtx, rows, cols, mtx_in_els, iels,
                                                                sign, row_conn, col_conn)
```

```
sfepy.discrete.common.extmods.assemble.assemble_matrix_complex_b(mtx, rows, cols, mtx_in_els,
                                                                    iels, sign, row_conn, col_conn)
```

```
sfepy.discrete.common.extmods.assemble.assemble_vector(vec, vec_in_els, iels, sign, conn)
```

```
sfepy.discrete.common.extmods.assemble.assemble_vector_complex(vec, vec_in_els, iels, sign, conn)
```

`sfepy.discrete.common.extmods.cmapping` module

Low level reference mapping functionality.

```
class sfepy.discrete.common.extmods.cmapping.CMapping
```

`sfepy.discrete.common.extmods.cmesh` module

C Mesh data structures and functions.

```
class sfepy.discrete.common.extmods.cmesh.CConnectivity
```

Notes

The memory is allocated/freed in C - this class just wraps NumPy arrays around that data without copying.

```
cprint()
```

```
indices
```

```
n_incident
```

```
num
```

offset

offsets

class sfepy.discrete.common.extmods.cmesh.CMesh

cell_groups

cell_types

conns

coors

cprint(*header_only=1*)

create_new(*entities=None, dent=0, localize=False*)

Create a new CMesh instance, with cells corresponding to the given *entities* of dimension *dent*.

Parameters

entities

[array, optional] The selected topological entities of the mesh to be in the new mesh. If not given, a copy of the mesh based on the cell-vertex connectivity is returned.

dent

[int, optional] The topological dimension of the entities.

localize

[bool] If True, strip the vertices not used in the the resulting sub-mesh cells and renumber the connectivity.

Returns

cmesh

[CMesh] The new mesh with the cell-vertex connectivity. Other connectivities have to be created and local entities need to be set manually.

dim

edge_oris

entities

face_oris

facet_oris

free_connectivity(*d1, d2*)

classmethod from_data(*coors, vertex_groups, conns, mat_ids, desc, copy_coors=True*)

Fill CMesh data using Python data.

get_cell_conn()

get_centroids(*dim*)

Return the coordinates of centroids of mesh entities with dimension *dim*.

get_complete(*dim, entities, dent*)

Get entities of dimension *dim* that are completely given by entities of dimension *dent* listed in *entities*.

get_conn(*d1*, *d2*)

get_conn_as_graph(*d1*, *d2*)

Get *d1* -> *d2* connectivity as a sparse matrix graph (values = ones).

For safety, creates a copy of the connectivity arrays. The connectivity is created if necessary.

get_facet_normals(*which=-1*)

Return the normals of facets for each mesh cell. The normals can be accessed using the cell-facet connectivity.

If *which* is -1, two normals of each quadrilateral face are averaged. If it is 0 or 1, the corresponding normal is used.

get_incident(*dim*, *entities*, *dent*, *ret_offsets=False*)

Get non-unique entities *indices* of dimension *dim* that are contained in entities of dimension *dent* listed in *entities*. As each of entities can be in several entities of dimension *dent*, *offsets* array is returned optionally.

get_local_entities(*key*)

get_local_ids(*entities*, *dent*, *incident*, *offsets*, *dim*)

Get local ids of *entities* of dimension *dent* in non-unique entities *incident* of dimension *dim* (with given *offsets* per *entities*) incident to *entities*, see `mesh_get_incident()`.

The function searches *entities* in *incident* -> *entities* connectivity for each non-unique entity in *incident*.

get_orientations(*dim*, *codim=None*)

Get orientations of entities of dimension *dim*. Alternatively, co-dimension can be specified using *codim* argument.

get_surface_facets()

Get facets (edges in 2D, faces in 3D) on the mesh surface.

get_volumes(*dim*)

Return the volumes of mesh entities with dimension *dim* > 0.

key_to_index

n_coor

n_el

num

set_local_entities(*gels*)

setup_connectivity(*d1*, *d2*)

setup_entities()

Set up mesh edge (2D and 3D) and face connectivities (3D only) as well as their orientations.

tdim

vertex_groups

`sfepy.discrete.common.extmods.cmesh.cmem_statistics()`

`sfepy.discrete.common.extmods.cmesh.get_cmem_usage()`

`sfepy.discrete.common.extmods.cmesh.orient_elements(flag, cmesh, cells, dcells, v_roots, v_vecs, swap_from, swap_to)`

Swap element nodes so that its volume is positive.

sfepy.discrete.common.extmods.crefcoors module**class sfepy.discrete.common.extmods.crefcoors.CBasisContext****sfepy.discrete.common.extmods.crefcoors.evaluate_in_rc**(*out, ref_coors, cells, status, source_vals, conn, diff, _ctx*)

Evaluate source field DOF values or gradients in the given reference element coordinates using the given interpolation.

1. Evaluate basis functions or gradients of basis functions in the reference coordinates. For gradients, transform the values to the material coordinates. 2. Interpolate source values using the basis functions/gradients.

Interpolation uses field approximation connectivity.

sfepy.discrete.common.extmods.crefcoors.find_ref_coors(*ref_coors, cells, status, coors, cmesh, candidates, offsets, allow_extrapolation, qp_eps, close_limit, _ctx*)**sfepy.discrete.common.extmods.crefcoors.find_ref_coors_convex**(*ref_coors, cells, status, coors, cmesh, centroids, normals0, normals1, ics, allow_extrapolation, qp_eps, close_limit, _ctx*)**sfepy.discrete.common.fields module****class sfepy.discrete.common.fields.Field**(***kwargs*)

Base class for fields.

clear_mappings(*clear_all=False*)

Clear current reference mappings.

create_eval_mesh()

Create a mesh for evaluating the field. The default implementation returns None, because this mesh is for most fields the same as the one created by *Field.create_mesh()*.

evaluate_at(*coors, source_vals, mode='val', strategy='general', close_limit=0.1, get_cells_fun=None, cache=None, ret_cells=False, ret_status=False, ret_ref_coors=False, verbose=False*)

Evaluate source DOF values corresponding to the field in the given coordinates using the field interpolation.

Parameters**coors**

[array, shape (n_coor, dim)] The coordinates the source values should be interpolated into.

source_vals

[array, shape (n_nod, n_components)] The source DOF values corresponding to the field.

mode

[{'val', 'grad', 'div', 'cauchy_strain'}, optional] The evaluation mode: the field value (default), the field value gradient, divergence, or cauchy strain.

strategy

[{'general', 'convex'}, optional] The strategy for finding the elements that contain the coordinates. For convex meshes, the 'convex' strategy might be faster than the 'general' one.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

get_cells_fun

[callable, optional] If given, a function with signature `get_cells_fun(coors, cmesh, **kwargs)` returning cells and offsets that potentially contain points with the coordinates *coors*. Applicable only when *strategy* is 'general'. When not given, `get_potential_cells()` is used.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as *cache.ref_coors*, *cache.cells* and *cache.status*, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored. See `Field.get_evaluate_cache()`.

ret_ref_coors

[bool, optional] If True, return also the found reference element coordinates.

ret_status

[bool, optional] If True, return also the enclosing cell status for each point.

ret_cells

[bool, optional] If True, return also the cell indices the coordinates are in.

verbose

[bool] If False, reduce verbosity.

Returns**vals**

[array] The interpolated values with shape `(n_coor, n_components, 1)` or gradients with shape `(n_coor, n_components, dim)` according to the *mode*. If *ret_status* is False, the values where the status is greater than one are set to `numpy.nan`.

ref_coors

[array] The found reference element coordinates, if *ret_ref_coors* is True.

cells

[array] The cell indices, if *ret_ref_coors* or *ret_cells* or *ret_status* are True.

status

[array] The status, if *ret_ref_coors* or *ret_status* are True, with the following meaning: 0 is success, 1 is extrapolation within *close_limit*, 2 is extrapolation outside *close_limit*, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If *close_limit* is 0, then for the 'general' strategy the status 5 indicates points outside of the field domain that had no potential cells.

static from_args(*name*, *dtype*, *shape*, *region*, *approx_order*=1, *space*='H1',
poly_space_basis='lagrange')

Create a Field subclass instance corresponding to a given space.

Parameters**name**

[str] The field name.

dtype

[numpy.dtype] The field data type: float64 or complex128.

shape

[int/tuple/str] The field shape: 1 or (1,) or 'scalar', space dimension (2, or (2,)) or 3 or (3,)) or 'vector', or a tuple. The field shape determines the shape of the FE base functions and is related to the number of components of variables and to the DOF per node count, depending on the field kind.

region

[Region] The region where the field is defined.

approx_order

[int/str] The FE approximation order, e.g. 0, 1, 2, '1B' (1 with bubble).

space

[str] The function space name.

poly_space_basis

[str] The name of polynomial space base.

Notes

Assumes one cell type for the whole region!

static from_conf(conf, regions)

Create a Field subclass instance based on the configuration.

get_mapping(region, integral, integration, get_saved=False, return_key=False)

For given region, integral and integration type, get a reference mapping, i.e. jacobians, element volumes and base function derivatives for Volume-type geometries, and jacobians, normals and base function derivatives for Surface-type geometries corresponding to the field approximation.

The mappings are cached in the field instance in *mappings* attribute. The mappings can be saved to *mappings0* using *Field.save_mappings*. The saved mapping can be retrieved by passing *get_saved=True*. If the required (saved) mapping is not in cache, a new one is created.

Returns**geo**

[PyCMapping instance] The reference mapping.

mapping

[FEMapping or IGMMapping instance] The mapping.

key

[tuple] The key of the mapping in *mappings* or *mappings0*.

save_mappings()

Save current reference mappings to *mappings0* attribute.

set_dofs(fun=0.0, region=None, dpn=None, warn=None)

Set the values of DOFs in a given *region* using a function of space coordinates or value *fun*.

If *fun* is a function, the l2 projection that is global for all region facets is used to set the DOFs.

If *dpn > 1*, and *fun* is a function, it has to return the values point-by-point, i.e. all components in the first point, in the second point etc., concatenated to an array that is reshapable to the shape (*n_point*, *dpn*).

Parameters**fun**

[float or array of length dpn or callable] The DOF values.

region

[Region] The region containing the DOFs.

dpn

[int, optional] The DOF-per-node count. If not given, the number of field components is used.

warn

[str, optional] The warning message printed when the region selects no DOFs.

Returns**nods**

[array, shape (n_dof,)] The field DOFs (or node indices) given by the region.

vals

[array, shape (n_dof, dpn)] The values of the DOFs, node-by-node when raveled in C (row-major) order.

Notes

The nodal basis fields (lagrange) reimplement this function to set DOFs directly.

The hierarchical basis field (lobatto) do not support surface mappings, so also reimplement this function.

`sfepy.discrete.common.fields.fields_from_conf(conf, regions)`

`sfepy.discrete.common.fields.parse_approx_order(approx_order)`

Parse the uniform approximation order value (str or int).

`sfepy.discrete.common.fields.parse_shape(shape, dim)`

`sfepy.discrete.common.fields.setup_extra_data(conn_info)`

Setup extra data required for non-volume integration.

sfepy.discrete.common.global_interp module

Global interpolation functions.

`sfepy.discrete.common.global_interp.get_potential_cells(coors, cmesh, centroids=None, extrapolate=True)`

Get cells that potentially contain points with the given physical coordinates.

Parameters**coors**

[array] The physical coordinates.

cmesh

[CMesh instance] The cmesh defining the cells.

centroids

[array, optional] The centroids of the cells.

extrapolate

[bool] If True, even the points that are surely outside of the cmesh are considered and assigned potential cells.

Returns

potential_cells

[array] The indices of the cells that potentially contain the points.

offsets

[array] The offsets into *potential_cells* for each point: a point *ip* is potentially in cells *potential_cells[offsets[ip]:offsets[ip+1]]*.

```
sfepy.discrete.common.global_interp.get_ref_coors(field, coors, strategy='general', close_limit=0.1,  
                                                  get_cells_fun=None, cache=None, verbose=False)
```

Get reference element coordinates and elements corresponding to given physical coordinates.

Parameters**field**

[Field instance] The field defining the approximation.

coors

[array] The physical coordinates.

strategy

[{'general', 'convex'}, optional] The strategy for finding the elements that contain the coordinates. For convex meshes, the 'convex' strategy might be faster than the 'general' one.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

get_cells_fun

[callable, optional] If given, a function with signature *get_cells_fun(coors, cmesh, **kwargs)* returning cells and offsets that potentially contain points with the coordinates *coors*. Applicable only when *strategy* is 'general'. When not given, *get_potential_cells()* is used.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as *cache.ref_coors*, *cache.cells* and *cache.status*, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

verbose

[bool] If False, reduce verbosity.

Returns**ref_coors**

[array] The reference coordinates.

cells

[array] The cell indices corresponding to the reference coordinates.

status

[array] The status: 0 is success, 1 is extrapolation within *close_limit*, 2 is extrapolation outside *close_limit*, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If *close_limit* is 0, then for the 'general' strategy the status 5 indicates points outside of the field domain that had no potential cells.

```
sfepy.discrete.common.global_interp.get_ref_coors_convex(field, coors, close_limit=0.1,  
                                                         cache=None, verbose=False)
```

Get reference element coordinates and elements corresponding to given physical coordinates.

Parameters

field

[Field instance] The field defining the approximation.

coors

[array] The physical coordinates.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as *cache.ref_coors*, *cache.cells* and *cache.status*, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

verbose

[bool] If False, reduce verbosity.

Returns**ref_coors**

[array] The reference coordinates.

cells

[array] The cell indices corresponding to the reference coordinates.

status

[array] The status: 0 is success, 1 is extrapolation within *close_limit*, 2 is extrapolation outside *close_limit*, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells.

Notes

Outline of the algorithm for finding $\mathbf{x}_i$ such that $\mathbf{X}(\mathbf{x}_i) = \mathbf{P}$:

1. make inverse connectivity - for each vertex have cells it is in.
2. find the closest vertex $\mathbf{V}$.
3. choose initial cell: i_0 = first from cells incident to $\mathbf{V}$.
4. while not $\mathbf{P}$ in $\mathbf{C}_{i_0}$, change $\mathbf{C}_{i_0}$ towards $\mathbf{P}$, check if $\mathbf{P}$ in new $\mathbf{C}_{i_0}$.

```
sfe.py.discrete.common.global_interp.get_ref_coors_general(field, coors, close_limit=0.1,
                                                           get_cells_fun=None, cache=None,
                                                           verbose=False)
```

Get reference element coordinates and elements corresponding to given physical coordinates.

Parameters**field**

[Field instance] The field defining the approximation.

coors

[array] The physical coordinates.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

get_cells_fun

[callable, optional] If given, a function with signature `get_cells_fun(coors, cmesh, **kwargs)` returning cells and offsets that potentially contain points with the coordinates *coors*. When not given, `get_potential_cells()` is used.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as `cache.ref_coors`, `cache.cells` and `cache.status`, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

verbose

[bool] If False, reduce verbosity.

Returns**ref_coors**

[array] The reference coordinates.

cells

[array] The cell indices corresponding to the reference coordinates.

status

[array] The status: 0 is success, 1 is extrapolation within *close_limit*, 2 is extrapolation outside *close_limit*, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If *close_limit* is 0, then status 5 indicates points outside of the field domain that had no potential cells.

sfepy.discrete.common.mappings module

Reference-physical domain mappings.

class sfepy.discrete.common.mappings.**Mapping**(**kwargs)

Base class for mappings.

static from_args(region, kind='v')

Create mapping from reference to physical entities in a given region, given the integration kind ('v' or 's').

This mapping can be used to compute the physical quadrature points.

Parameters**region**

[Region instance] The region defining the entities.

kind

['v' or 's'] The kind of the entities: 'v' - cells, 's' - facets.

Returns**mapping**

[FEMapping or IGMapping instance] The requested mapping.

class sfepy.discrete.common.mappings.**PhysicalQPs**(num=0)

Physical quadrature points in a region.

get_shape(rshape)

Get shape from raveled shape.

class sfepy.discrete.common.mappings.**PyCMapping**(*bf, det, volume, bfg, normal, dim*)

Class for storing mapping data. Primary data in numpy arrays. Data for C functions translated to FMFields and embedded in CMapping.

integrate(*out, field, mode=0*)

sfepy.discrete.common.mappings.**get_jacobian**(*field, integral, region=None, integration='volume'*)

Get the jacobian of reference mapping corresponding to *field*.

Parameters

field

[Field instance] The field defining the reference mapping.

integral

[Integral instance] The integral defining quadrature points.

region

[Region instance, optional] If given, use the given region instead of *field* region.

integration

[one of ('volume', 'surface', 'surface_extra')] The integration type.

Returns

jac

[array] The jacobian merged for all element groups.

➡ See also

[*get_mapping_data*](#)

Notes

Assumes the same element geometry in all element groups of the field!

sfepy.discrete.common.mappings.**get_mapping_data**(*name, field, integral, region=None, integration='volume'*)

General helper function for accessing reference mapping data.

Get data attribute *name* from reference mapping corresponding to *field* in *region* in quadrature points of the given *integral* and *integration* type.

Parameters

name

[str] The reference mapping attribute name.

field

[Field instance] The field defining the reference mapping.

integral

[Integral instance] The integral defining quadrature points.

region

[Region instance, optional] If given, use the given region instead of *field* region.

integration

[one of ('volume', 'surface', 'surface_extra')] The integration type.

Returns**data**

[array] The required data merged for all element groups.

Notes

Assumes the same element geometry in all element groups of the field!

`sfepy.discrete.common.mappings.get_normals(field, integral, region)`

Get the normals of element faces in *region*.

Parameters**field**

[Field instance] The field defining the reference mapping.

integral

[Integral instance] The integral defining quadrature points.

region

[Region instance] The given of the element faces.

Returns**normals**

[array] The normals merged for all element groups.

 See also

[*get_mapping_data*](#)

Notes

Assumes the same element geometry in all element groups of the field!

`sfepy.discrete.common.mappings.get_physical_qps(region, integral, map_kind=None)`

Get physical quadrature points corresponding to the given region and integral.

sfepy.discrete.common.poly_spaces module

class `sfepy.discrete.common.poly_spaces.PolySpace(name, geometry, order)`

Abstract polynomial space class.

static any_from_args(*name, geometry, order, basis='lagrange', force_bubble=False*)

Construct a particular polynomial space classes according to the arguments passed in.

eval_basis(*coors, diff=0, ori=None, force_axis=False, transform=None, suppress_errors=False, eps=1e-15*)

Evaluate the basis or its first or second derivatives in points given by coordinates. The real work is done in `_eval_basis()` implemented in subclasses.

Note that the second derivative code is a work-in-progress and only *coors* and *transform* arguments are used.

Parameters

coors

[array_like] The coordinates of points where the basis is evaluated. See Notes.

diff

[0, 1 or 2] If nonzero, return the given derivative.

ori

[array_like, optional] Optional orientation of element facets for per element basis.

force_axis

[bool] If True, force the resulting array shape to have one more axis even when *ori* is None.

transform

[array_like, optional] The basis transform array.

suppress_errors

[bool] If True, do not report points outside the reference domain.

eps

[float] Accuracy for comparing coordinates.

Returns

basis

[array] The basis (shape (n_coors, 1, n_basis)) or its first derivative (shape (n_coors, dim, n_basis)) or its second derivative (shape (n_coors, dim, dim, n_basis)) evaluated in the given points. An additional axis is pre-pended of length n_cell, if *ori* is given, or of length 1, if *force_axis* is True.

Notes

If `coors.ndim == 3`, several point sets are assumed, with equal number of points in each of them. This is the case, for example, of the values of the volume basis functions on the element facets. The indexing (of `bf_b(g)`) is then `(ifa,iqp,:,n_ep)`, so that the facet can be set in C using `FMF_SetCell`.

```
keys = {(0, 1): 'simplex', (1, 2): 'simplex', (2, 3): 'simplex', (2, 4):
'tensor_product', (3, 4): 'simplex', (3, 6): 'wedge', (3, 8): 'tensor_product'}
```

```
static suggest_name(geometry, order, basis='lagrange', force_bubble=False)
```

Suggest the polynomial space name given its constructor parameters.

```
sfepy.discrete.common.poly_spaces.register_poly_space(cls)
```

```
sfepy.discrete.common.poly_spaces.transform_basis(transform, bf)
```

Transform a basis *bf* using *transform* array of matrices.

sfepy.discrete.common.region module

```
class sfepy.discrete.common.region.Region(name, definition, domain, parse_def, kind='cell',
parent=None, tdim=None)
```

Region defines a subset of a FE domain.

Region kinds:

- `cell_only`, `facet_only`, `face_only`, `edge_only`, `vertex_only` - only the specified entities are included, others are empty sets (so that the operators are still defined)

- cell, facet, face, edge, vertex - entities of higher dimension are not included

The 'cell' kind is the most general and it is the default.

Region set-like operators: + (union), - (difference), * (intersection), followed by one of ('v', 'e', 'f', 'c', and 's') for vertices, edges, faces, cells, and facets.

Created: 31.10.2005

property cells

contains(*other*)

Return True if the region contains the *other* region.

The check is performed using entities corresponding to the other region kind.

copy()

Make a copy based on the region kind.

delete_zero_faces(*eps=1e-14*)

property edges

eval_op_cells(*other, op*)

eval_op_edges(*other, op*)

eval_op_faces(*other, op*)

eval_op_facets(*other, op*)

eval_op_vertices(*other, op*)

property faces

property facets

finalize(*allow_empty=False*)

Initialize the entities corresponding to the region kind and regenerate all already existing (accessed) entities of lower topological dimension from the kind entities.

static from_cells(*cells, domain, name='region', kind='cell', parent=None*)

Create a new region containing given cells.

Parameters

cells

[array] The array of cells.

domain

[Domain instance] The domain containing the facets.

name

[str, optional] The name of the region.

kind

[str, optional] The kind of the region.

parent

[str, optional] The name of the parent region.

Returns

obj

[Region instance] The new region.

static from_facets(*facets, domain, name='region', kind='facet', parent=None*)

Create a new region containing given facets.

Parameters

facets

[array] The array with indices to unique facets.

domain

[Domain instance] The domain containing the facets.

name

[str, optional] The name of the region.

kind

[str, optional] The kind of the region.

parent

[str, optional] The name of the parent region.

Returns

obj

[Region instance] The new region.

static from_vertices(*vertices, domain, name='region', kind='cell'*)

Create a new region containing given vertices.

Parameters

vertices

[array] The array of vertices.

domain

[Domain instance] The domain containing the vertices.

name

[str, optional] The name of the region.

kind

[str, optional] The kind of the region.

Returns

obj

[Region instance] The new region.

get_cell_indices(*cells, true_cells_only=True*)

Return indices of *cells* in the region cells.

Raises `ValueError` if *true_cells_only* is `True` and the region kind does not allow cells. For *true_cells_only* equal to `False`, cells incident to facets are returned if the region itself contains no cells.

Raises `ValueError` if all *cells* are not in the region cells.

get_cells(*true_cells_only=True*)

Get cells of the region.

Raises `ValueError` if *true_cells_only* is `True` and the region kind does not allow cells. For *true_cells_only* equal to `False`, cells incident to facets are returned if the region itself contains no cells. Obeys parent region, if given.

get_charfun(*by_cell=False, val_by_id=False*)

Return the characteristic function of the region as a vector of values defined either in the mesh vertices (*by_cell* == *False*) or cells. The values are either 1 (*val_by_id* == *False*) or sequential id + 1.

get_edge_graph()

Return the graph of region edges as a sparse matrix having uid(k) + 1 at (i, j) if vertex[i] is connected with vertex[j] by the edge k.

Degenerate edges are ignored.

get_entities(*dim*)

Return mesh entities of dimension *dim*.

get_facet_indices()

Return an array (per group) of (iel, ifa) for each facet. A facet can be in 1 (surface) or 2 (inner) cells.

get_mirror_region(*name*)

get_n_cells(*is_surface=False*)

Get number of region cells.

Parameters

is_surface

[bool] If True, number of edges or faces according to domain dimension is returned instead.

Returns

n_cells

[int] The number of cells.

has_cells()

light_copy(*name, parse_def, tdim=None*)

set_kind(*kind*)

set_kind_tdim()

setup_from_highest(*dim, allow_lower=True, allow_empty=False*)

Setup entities of topological dimension *dim* using the available entities of the highest topological dimension.

setup_from_vertices(*dim*)

Setup entities of topological dimension *dim* using the region vertices.

setup_mirror_region(*mirror_name=None, ret_name=False*)

Find the corresponding mirror region, set up element mapping.

update_shape()

Update shape of each group according to region vertices, edges, faces and cells.

property vertices

`sfepy.discrete.common.region.are_disjoint(r1, r2)`

Check if the regions *r1* and *r2* are disjoint.

Uses vertices for the check - *_only regions not allowed.

`sfepy.discrete.common.region.get_dependency_graph(region_defs)`

Return a dependency graph and a name-sort name mapping for given region definitions.

`sfepy.discrete.common.region.get_parents(selector)`

Given a region selector, return names of regions it is based on.

`sfepy.discrete.common.region.sort_by_dependency(graph)`

sfepy.discrete.fem sub-package

sfepy.discrete.fem.domain module

Computational domain, consisting of the mesh and regions.

class `sfepy.discrete.fem.domain.FEDomain(name, mesh, verbose=False, **kwargs)`

Domain is divided into groups, whose purpose is to have homogeneous data shapes.

clear_surface_groups()

Remove surface group data.

create_surface_group(region)

Create a new surface group corresponding to *region* if it does not exist yet.

Notes

Surface groups define surface facet connectivity that is needed for [`sfepy.discrete.fem.mappings.FEMapping`](#).

fix_element_orientation(geom_els=None, force_check=False)

Ensure element vertices ordering giving positive cell volumes.

get_conn(ret_gel=False, tdim=None, cells=None)

Get the cell-vertex connectivity and, if *ret_gel* is True, also the corresponding reference geometry element. If *tdim* is not None get the connectivity of the cells with topological dimension *tdim*.

get_diameter()

Return the diameter of the domain.

Notes

The diameter corresponds to the Friedrichs constant.

get_element_diameters(cells, volume, mode, square=True)

get_mesh_bounding_box()

Return the bounding box of the underlying mesh.

Returns

bbox

[ndarray (2, dim)] The bounding box with min. values in the first row and max. values in the second row.

get_mesh_coors(actual=False)

Return the coordinates of the underlying mesh vertices.

refine()

Uniformly refine the domain mesh.

Returns**domain**

[FEDomain instance] The new domain with the refined mesh.

Notes

Works only for meshes with single element type! Does not preserve node groups!

sfepy.discrete.fem.extmods.bases module

Polynomial base functions and related utilities.

class sfepy.discrete.fem.extmods.bases.**CLagrangeContext**

baseId

cprint()

e_coors_max

evaluate(*coors*, *diff=False*, *eps=1e-15*, *check_errors=True*)

geo_ctx

iel

is_bubble

mbfg

mesh_conn

mesh_coors

sfepy.discrete.fem.extmods.lobatto_bases module

Interface to Lobatto bases.

sfepy.discrete.fem.extmods.lobatto_bases.**eval_lobatto1d**(*coors*, *order*)

Evaluate 1D Lobatto functions of the given order in given points.

sfepy.discrete.fem.extmods.lobatto_bases.**eval_lobatto_tensor_product**(*coors*, *nodes*, *cmin*, *cmax*,
order, *diff=False*)

Evaluate tensor product Lobatto functions of the given order in given points.

Base functions are addressed using the *nodes* array with rows corresponding to individual functions and columns to 1D indices (= orders when ≥ 1) into *lobatto[]* and *d_lobatto[]* lists for each axis.

sfepy.discrete.fem.facets module

Helper functions related to mesh facets and Lagrange FE approximation.

Line: ori - iter:

0 - iter0 1 - iter1

Triangle: ori - iter:

0 - iter21 1 - iter12 3 - iter02 4 - iter20 6 - iter10 7 - iter01

Possible couples:

1, 4, 7 <-> 0, 3, 6

Square: ori - iter:

0 - iter10x01y 7 - iter10y01x

11 - iter01y01x 30 - iter01x10y 33 - iter10x10y 52 - iter01y10x 56 - iter10y10x 63 - iter01x01y

Possible couples:

7, 33, 52, 63 <-> 0, 11, 30, 56

`_quad_ori_groups`:

$i < j < k < l$

all faces are permuted to

$l - k ||| i - j$

ijkl

which is the same as

$l - j ||| i - k$

ikjl

$k - l ||| i - j$

ijlk

- start at one vertex and go around clock-wise or anticlock-wise

-> 8 groups of 3 -> same face nodes order in ijkl (63), ikjl (59), ijlk (31) ilkj (11), iklj (15), iljk (43) jkli (7), jlki (3), kjli (6) kjil (56), jkil (57), ljik (48) lijk (52), likj (20), kijl (60) lkji (0), ljki (4), klji (1) klj (33), lkij (32), jlik (41) jilk (30), kilj (22), jilk (62)

sfepy.discrete.fem.facets.build_orientation_map(*n_fp*)

The keys are binary masks of the lexicographical ordering of facet vertices. A bit *i* set to one means $v[i] < v[i+1]$.

The values are [*original_order*, *permutation*], where *permutation* can be used to sort facet vertices lexicographically. Hence *permuted_facet* = *facet*[*permutation*].

sfepy.discrete.fem.facets.get_facet_dof_permutations(*n_fp*, *order*)

Prepare DOF permutation vector for each possible facet orientation.

sfepy.discrete.fem.facets.iter0(*num*)

sfepy.discrete.fem.facets.iter01(*num*)

sfepy.discrete.fem.facets.iter01x01y(*num*)

```
sfepy.discrete.fem.facets.iter01x10y(num)
sfepy.discrete.fem.facets.iter01y01x(num)
sfepy.discrete.fem.facets.iter01y10x(num)
sfepy.discrete.fem.facets.iter02(num)
sfepy.discrete.fem.facets.iter1(num)
sfepy.discrete.fem.facets.iter10(num)
sfepy.discrete.fem.facets.iter10x01y(num)
sfepy.discrete.fem.facets.iter10x10y(num)
sfepy.discrete.fem.facets.iter10y01x(num)
sfepy.discrete.fem.facets.iter10y10x(num)
sfepy.discrete.fem.facets.iter12(num)
sfepy.discrete.fem.facets.iter20(num)
sfepy.discrete.fem.facets.iter21(num)
sfepy.discrete.fem.facets.make_line_matrix(order)
sfepy.discrete.fem.facets.make_square_matrix(order)
sfepy.discrete.fem.facets.make_triangle_matrix(order)
```

sfepy.discrete.fem.fe_surface module

```
class sfepy.discrete.fem.fe_surface.FEPhantomSurface(name, region, volume_econn)
```

A phantom surface of the region with tdim=2.

```
class sfepy.discrete.fem.fe_surface.FESurface(name, region, efaces, volume_econn,
                                              volume_region=None)
```

Description of a surface of a finite element domain.

```
static from_region(name, region, ret_gel=False)
```

```
get_connectivity(local=False, trace_region=None)
```

Return the surface element connectivity.

Parameters

local

[bool] If True, return local connectivity w.r.t. surface nodes, otherwise return global connectivity w.r.t. all mesh nodes.

trace_trace

[None or str] If not None, return mirror connectivity according to *local*.

```
set_orientation_map()
```


setup_mirror_connectivity(*region, mirror_name*)

Setup mirror surface connectivity required to integrate over a mirror region.

1. Get orientation of the faces: a) for own elements -> ooris b) for mirror elements -> moris
2. orientation -> permutation.

sfepy.discrete.fem.fields_base module

Notes

Important attributes of continuous (order > 0) Field and SurfaceField instances:

- *vertex_remap* : *econn*[:, :*n_vertex*] = *vertex_remap*[*conn*]
- *vertex_remap_i* : *conn* = *vertex_remap_i*[*econn*[:, :*n_vertex*]]

where *conn* is the mesh vertex connectivity, *econn* is the region-local field connectivity.

class sfepy.discrete.fem.fields_base.FEField(*name, dtype, shape, region, approx_order=1*)

Base class for finite element fields.

Notes

- *interp*s and hence *node_descs* are per region (must have single geometry!)

Field shape information:

- *shape* - the shape of the basis functions in a point
- *n_components* - the number of DOFs per FE node
- *val_shape* - the shape of field value (the product of DOFs and basis functions) in a point

average_qp_to_vertices(*data_qp, integral*)

Average data given in quadrature points in region elements into region vertices.

$$u_n = \sum_e (u_{e,avg} * area_e) / \sum_e area_e = \sum_e \int_{area_e} u / \sum_e area_e$$

clear_qp_basis()

Remove cached quadrature points and basis functions.

create_bqp(*region_name, integral*)

create_bqp_key(*integral, bkey*)

create_mapping(*region, integral, integration, return_mapping=True*)

Create a new reference mapping.

Compute jacobians, element volumes and basis function derivatives for Volume-type geometries (volume mappings), and jacobians, normals and basis function derivatives for Surface-type geometries (surface mappings).

Notes

- surface mappings are defined on the surface region
- surface mappings require field order to be > 0

create_mesh(*extra_nodes=True*)

Create a mesh from the field region, optionally including the field extra nodes.

create_output(*dofs*, *var_name*, *dof_names=None*, *key=None*, *extend=True*, *fill_value=None*, *linearization=None*)

Convert the DOFs corresponding to the field to a dictionary of output data usable by Mesh.write().

Parameters

dofs

[array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

var_name

[str] The variable name corresponding to *dofs*.

dof_names

[tuple of str] The names of DOF components.

key

[str, optional] The key to be used in the output dictionary instead of the variable name.

extend

[bool] Extend the DOF values to cover the whole domain.

fill_value

[float or complex] The value used to fill the missing DOF values if *extend* is True.

linearization

[Struct or None] The linearization configuration for higher order approximations.

Returns

out

[dict] The output dictionary.

eval_basis(*key*, *derivative*, *integral*, *iels=None*, *from_geometry=False*, *basis_only=True*)

extend_dofs(*dofs*, *fill_value=None*)

Extend DOFs to the whole domain using the *fill_value*, or the smallest value in *dofs* if *fill_value* is None.

get_coor(*nods=None*)

Get coordinates of the field nodes.

Parameters

nods

[array, optional] The indices of the required nodes. If not given, the coordinates of all the nodes are returned.

get_data_shape(*integral*, *integration='cell'*, *region_name=None*)

Get element data dimensions.

Parameters

integral

[Integral instance] The integral describing used numerical quadrature.

integration

['cell', 'facet', 'facet_extra', 'point' or 'custom'] The term integration mode.

region_name

[str] The name of the region of the integral.

Returns**data_shape**

[4 ints] The $(n_{el}, n_{qp}, dim, n_{en})$ for volume shape kind, $(n_{fa}, n_{qp}, dim, n_{fn})$ for surface shape kind and $(n_{nod}, 0, 0, 1)$ for point shape kind.

Notes

Integration modes: - 'cell': integration over cells/elements - 'facet': integration over cell facets (faces, edges) - 'facet_extra': same as 'facet' but also the normal derivatives

are evaluated

- 'point': point integration
- 'custom': user defined integration

Dimensions: - n_{el} , n_{fa} = number of elements/facets - n_{qp} = number of quadrature points per element/facet - dim = spatial dimension - n_{en} , n_{fn} = number of element/facet nodes - n_{nod} = number of element nodes

get_dofs_in_region(region, merge=True)

Return indices of DOFs that belong to the given region.

get_econn(conn_type, region, trace_region=None, local=False)

Get extended connectivity of the given type in the given region.

Parameters**conn_type: tuple or string**

DOF connectivity type, eg. ('cell', 3) or 'cell'. If the topological dimension not specified, it is taken from region.tdim.

region: sfepy.discrete.common.region.Region

The region for which the connectivity is required.

trace_region: None or string

If not None, return mirror connectivity according to *local*.

local: bool

If True, return local connectivity w.r.t. facet nodes, otherwise return global connectivity w.r.t. all mesh nodes.

Returns**econn: numpy.ndarray**

The extended connectivity array.

get_evaluate_cache(cache=None, share_geometry=False, verbose=False)

Get the evaluate cache for `Variable.evaluate_at()`.

Parameters**cache**

[Struct instance, optional] Optionally, use the provided instance to store the cache data.

share_geometry

[bool] Set to True to indicate that all the evaluations will work on the same region. Certain data are then computed only for the first probe and cached.

verbose

[bool] If False, reduce verbosity.

Returns**cache**

[Struct instance] The evaluate cache.

get_output_approx_order()

Get the approximation order used in the output file.

get_qp(key, integral)

Get quadrature points and weights corresponding to the given key and integral. The key is 'v', 's#' or 'b#', where # is the number of face vertices. For 'b#', the quadrature must already be created by calling [FEField.create_bqp\(\)](#), usually through [FEField.create_mapping\(\)](#).

get_true_order()

Get the true approximation order depending on the reference element geometry.

For example, for P1 (linear) approximation the true order is 1, while for Q1 (bilinear) approximation in 2D the true order is 2.

get_vertices()

Return indices of vertices belonging to the field region.

interp_to_qp(dofs)

Interpolate DOFs into quadrature points.

The quadrature order is given by the field approximation order.

Parameters**dofs**

[array] The array of DOF values of shape $(n\_nod, n\_component)$.

Returns**data_qp**

[array] The values interpolated into the quadrature points.

integral

[Integral] The corresponding integral defining the quadrature points.

is_higher_order()

Return True, if the field's approximation order is greater than one.

linearize(dofs, min_level=0, max_level=1, eps=0.0001)

Linearize the solution for post-processing.

Parameters**dofs**

[array, shape $(n\_nod, n\_component)$] The array of DOFs reshaped so that each column corresponds to one component.

min_level

[int] The minimum required level of mesh refinement.

max_level

[int] The maximum level of mesh refinement.

eps

[float] The relative tolerance parameter of mesh adaptivity.

Returns**mesh**

[Mesh instance] The adapted, nonconforming, mesh.

vdofs

[array] The DOFs defined in vertices of *mesh*.

levels

[array of ints] The refinement level used for each element group.

remove_extra_dofs(dofs)

Remove DOFs defined in higher order nodes (order > 1).

restore_dofs(store=False)

Undoes the effect of *FEField.substitute_dofs()*.

restore_substituted(vec)

Restore values of the unused DOFs using the transpose of the applied basis transformation.

set_basis_transform(transform)

Set local element basis transformation.

The basis transformation is applied in *FEField.eval_basis()* and *FEField.create_mapping()*.

Parameters**transform**

[array, shape (*n_cell*, *n_ep*, *n_ep*)] The array with (*n_ep*, *n_ep*) transformation matrices for each cell in the field's region, where *n_ep* is the number of element DOFs.

set_coors(coors, extra_dofs=False)

Set coordinates of field nodes.

setup_bar_data(field, region)**setup_coors()**

Setup coordinates of field nodes.

setup_extra_data(info)**setup_point_data(field, region)****setup_surface_data(region, trace_region=None)**

nodes[leconn] == econn

substitute_dofs(subs, restore=False)

Perform facet DOF substitutions according to *subs*.

Modifies *self.econn* in-place and sets *self.econn0*, *self.unused_dofs* and *self.basis_transform*.

class sfepy.discrete.fem.fields_base.H1Mixin(kwargs)**

Methods of fields specific to H1 space.

```
sfepy.discrete.fem.fields_base.create_expression_output(expression, name, primary_field_name,  
                                                    fields, materials, variables,  
                                                    functions=None, mode='eval',  
                                                    term_mode=None, extra_args=None,  
                                                    verbose=True, kwargs=None, min_level=0,  
                                                    max_level=1, eps=0.0001)
```

Create output mesh and data for the expression using the adaptive linearizer.

Parameters

expression

[str] The expression to evaluate.

name

[str] The name of the data.

primary_field_name

[str] The name of field that defines the element groups and polynomial spaces.

fields

[dict] The dictionary of fields used in *variables*.

materials

[Materials instance] The materials used in the expression.

variables

[Variables instance] The variables used in the expression.

functions

[Functions instance, optional] The user functions for materials etc.

mode

[one of 'eval', 'el_avg', 'qp'] The evaluation mode - 'qp' requests the values in quadrature points, 'el_avg' element averages and 'eval' means integration over each term region.

term_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

extra_args

[dict, optional] Extra arguments to be passed to terms in the expression.

verbose

[bool] If False, reduce verbosity.

kwargs

[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression.

min_level

[int] The minimum required level of mesh refinement.

max_level

[int] The maximum level of mesh refinement.

eps

[float] The relative tolerance parameter of mesh adaptivity.

Returns

out

[dict] The output dictionary.

```
sfepy.discrete.fem.fields_base.eval_nodal_coors(coors, mesh_coors, region, poly_space,  
                                                geom_poly_space, econn, only_extra=True)
```

Compute coordinates of nodes corresponding to *poly_space*, given mesh coordinates and *geom_poly_space*.

```
sfepy.discrete.fem.fields_base.get_eval_expression(expression, fields, materials, variables,  
                                                  functions=None, mode='eval', term_mode=None,  
                                                  extra_args=None, verbose=True, kwargs=None)
```

Get the function for evaluating an expression given a list of elements, and reference element coordinates.

```
sfepy.discrete.fem.fields_base.set_mesh_coors(domain, fields, coors, update_fields=False,  
                                              actual=False, clear_all=True, extra_dofs=False)
```

sfepy.discrete.fem.fields_hierarchic module

```
class sfepy.discrete.fem.fields_hierarchic.H1HierarchicVolumeField(name, dtype, shape, region,  
                                                                approx_order=1)
```

Hierarchical basis approximation with Lobatto polynomials.

```
create_basis_context()
```

Create the context required for evaluating the field basis.

```
family_name = 'volume_H1_lobatto'
```

```
set_dofs(fun=0.0, region=None, dpn=None, warn=None)
```

Set the values of DOFs in a given *region* using a function of space coordinates or value *fun*.

sfepy.discrete.fem.fields_l2 module

```
class sfepy.discrete.fem.fields_l2.L2ConstantSurfaceField(name, dtype, shape, region,  
                                                         approx_order=0)
```

```
family_name = 'surface_L2_constant'
```

```
class sfepy.discrete.fem.fields_l2.L2ConstantVolumeField(name, dtype, shape, region,  
                                                         approx_order=0)
```

The L2 constant-in-a-region approximation.

```
create_mapping(region, integral, integration, return_mapping=True)
```

```
create_output(dofs, var_name, dof_names=None, key=None, extend=True, fill_value=None,  
             linearization=None)
```

```
eval_basis(key, derivative, integral, iels=None, from_geometry=False, base_only=True)
```

```
family_name = 'volume_L2_constant'
```

```
get_coor(nods=None)
```

Returns the barycenter of the field region.

Parameters

nods

[array, optional] Ignored.

get_data_shape(*integral*, *integration*='cell', *region_name*=None)

Get element data dimensions.

Parameters

integral

[Integral instance] The integral describing used numerical quadrature.

integration

['cell'] The term integration type. Ignored.

region_name

[str] The name of the region of the integral.

Returns

data_shape

[4 ints] The (*n_el*, *n_qp*, *dim*, *n_en*) for volume shape kind, (*n_fa*, *n_qp*, *dim*, *n_fn*) for surface shape kind.

Notes

- *n_el*, *n_fa* = number of elements/facets
- *n_qp* = number of quadrature points per element/facet
- *dim* = spatial dimension
- *n_en*, *n_fn* = number of element/facet nodes
- *n_nod* = number of element nodes

get_dofs_in_region(*region*, *merge*=True)

Return indices of DOFs that belong to the given region.

get_econn(*conn_type*, *region*, *trace_region*=None, *local*=False)

Get extended connectivity of the given type in the given region.

Parameters

conn_type: tuple or string

DOF connectivity type, ignored.

region: sfepy.discrete.common.region.Region

The region for which the connectivity is required.

trace_region: None or string

Ignored.

local: bool

Ignored.

Returns

econn: numpy.ndarray

The extended connectivity array.

setup_extra_data(*info*)

sfepy.discrete.fem.fields_nodal module**Notes**

Important attributes of continuous (order > 0) Field and SurfaceField instances:

- `vertex_remap : econn[:, :n_vertex] = vertex_remap[conn]`
- `vertex_remap_i : conn = vertex_remap_i[econn[:, :n_vertex]]`

where `conn` is the mesh vertex connectivity, `econn` is the region-local field connectivity.

class sfepy.discrete.fem.fields_nodal.GlobalNodalLikeBasis(**kwargs)

get_surface_basis(region)

Get basis for projections to region's facets.

Notes

Cannot be used for all fields because IGA does not support surface mappings.

class sfepy.discrete.fem.fields_nodal.H1DiscontinuousField(name, dtype, shape, region, approx_order=1)

The C0 constant-per-cell approximation.

average_to_vertices(dofs)

Average DOFs of the discontinuous field into the field region vertices.

extend_dofs(dofs, fill_value=None)

Extend DOFs to the whole domain using the `fill_value`, or the smallest value in `dofs` if `fill_value` is None.

family_name = 'volume_H1_lagrange_discontinuous'

remove_extra_dofs(dofs)

Remove DOFs defined in higher order nodes (order > 1).

class sfepy.discrete.fem.fields_nodal.H1NodalMixin(**kwargs)

create_basis_context()

Create the context required for evaluating the field basis.

set_dofs(fun=0.0, region=None, dpn=None, warn=None)

Set the values of DOFs in a given `region` using a function of space coordinates or value `fun`.

class sfepy.discrete.fem.fields_nodal.H1NodalSurfaceField(name, dtype, shape, region, approx_order=1)

A field defined on a surface region.

family_name = 'surface_H1_lagrange'

interp_v_vals_to_n_vals(vec)

Interpolate a function defined by vertex DOF values using the FE surface geometry basis (P1 or Q1) into the extra nodes, i.e. define the extra DOF values.

class sfepy.discrete.fem.fields_nodal.H1NodalVolumeField(name, dtype, shape, region, approx_order=1)

Lagrange basis nodal approximation.

```
family_name = 'volume_H1_lagrange'
```

```
interp_v_vals_to_n_vals(vec)
```

Interpolate a function defined by vertex DOF values using the FE geometry basis (P1 or Q1) into the extra nodes, i.e. define the extra DOF values.

```
class sfepy.discrete.fem.fields_nodal.H1SEMSurfaceField(name, dtype, shape, region,  
                                                       approx_order=1)
```

```
family_name = 'surface_H1_sem'
```

```
class sfepy.discrete.fem.fields_nodal.H1SEMVolumField(name, dtype, shape, region,  
                                                       approx_order=1)
```

Spectral element method approximation.

Uses the Lagrange basis with Legendre-Gauss-Lobatto nodes and quadrature.

```
create_basis_context()
```

Create the context required for evaluating the field basis.

```
family_name = 'volume_H1_sem'
```

```
class sfepy.discrete.fem.fields_nodal.H1SNodalSurfaceField(name, dtype, shape, region,  
                                                           approx_order=1)
```

```
family_name = 'surface_H1_serendipity'
```

```
class sfepy.discrete.fem.fields_nodal.H1SNodalVolumField(name, dtype, shape, region,  
                                                         approx_order=1)
```

Lagrange basis nodal serendipity approximation with order ≤ 3 .

```
create_basis_context()
```

Create the context required for evaluating the field basis.

```
family_name = 'volume_H1_serendipity'
```

sfepy.discrete.fem.fields_positive module

```
class sfepy.discrete.fem.fields_positive.H1BernsteinSurfaceField(name, dtype, shape, region,  
                                                                  approx_order=1)
```

```
family_name = 'surface_H1_bernstein'
```

```
class sfepy.discrete.fem.fields_positive.H1BernsteinVolumField(name, dtype, shape, region,  
                                                                approx_order=1)
```

Bernstein basis approximation with positive-only basis function values.

```
create_basis_context()
```

Create the context required for evaluating the field basis.

```
family_name = 'volume_H1_bernstein'
```

sfepy.discrete.fem.geometry_element module

GeometryElement describes the geometric entities of a finite element mesh.

Notes

- geometry_data: surface facets are assumed to be of the same kind for each geometry element - wedges or pyramids are not supported.
- the orientation is a tuple: (root1, vertices of direction vectors, swap from, swap to, root2, ...)

class sfepy.discrete.fem.geometry_element.GeometryElement(name)

The geometric entities of a finite element mesh.

create_surface_facet()

Create a GeometryElement instance corresponding to this instance surface facet.

get_conn_permutations()

Get all possible connectivity permutations corresponding to different spatial orientations of the geometry element.

get_edges_per_face()

Return the indices into self.edges per face.

get_grid(n_nod)

Get a grid of *n_nod* interpolation points, including the geometry element vertices. The number of points must correspond to a valid number of FE nodes for each geometry.

get_interpolation_name()

Get the name of corresponding linear interpolant.

get_surface_entities()

Return self.vertices in 1D, self.edges in 2D and self.faces in 3D.

sfepy.discrete.fem.geometry_element.create_geometry_elements(names=None)

Utility function to create GeometryElement instances.

Parameters**names**

[str, optional] The names of the entity, one of the keys in geometry_data dictionary. If None, all keys of geometry_data are used.

Returns**gels**

[dict] The dictionary of geometry elements with names as keys.

sfepy.discrete.fem.geometry_element.setup_orientation(vecs_tuple)

sfepy.discrete.fem.history module

```
class sfepy.discrete.fem.history.Histories(objs=None, **kwargs)

    static from_file_hdf5(filename, var_names)
        TODO: do not read entire file, provide data on demand.

class sfepy.discrete.fem.history.History(name, th=None, steps=None, times=None)

    append(item, step, time)

    static from_sequence(seq, name)
```

sfepy.discrete.fem.lcbc_operators module

Operators for enforcing linear combination boundary conditions in nodal FEM setting.

```
class sfepy.discrete.fem.lcbc_operators.AverageForceOperator(name, regions, dof_names,
                                                             dof_map_fun, variables, ts=None,
                                                             functions=None)
```

Transformation matrix operator for average force multi-point constraint LCBCs.

Unlike in other operators, the *regions* and *dof_names* couples are ordered as (independent nodes/DOFs, dependent nodes/DOFs), to allow a simple interchange with the `rigid2` LCBC in a problem description. Functionally it corresponds to the RBE3 multi-point constraint in MSC/Nastran.

Rotation DOFs in independent nodes are not supported (ignored), see the comment (*) in the code. This is because the independent field is assumed to come from solid elements.

A simplified version for the dependent field without the rotation DOFs is also supported.

```
kind = 'average_force'
```

```
class sfepy.discrete.fem.lcbc_operators.EdgeDirectionOperator(name, regions, dof_names,
                                                             dof_map_fun, filename, variables,
                                                             ts=None, functions=None)
```

Transformation matrix operator for edges direction LCBCs.

The substitution (in 3D) is:

$$[u_1, u_2, u_3]^T = [d_1, d_2, d_3]^T w,$$

where $\underline{d}$ is an edge direction vector averaged into a node. The new DOF is w .

```
get_vectors(nodes, region, field, filename=None)
```

```
kind = 'edge_direction'
```

```
class sfepy.discrete.fem.lcbc_operators.IntegralMeanValueOperator(name, regions, dof_names,
                                                                    dof_map_fun, variables,
                                                                    ts=None, functions=None)
```

Transformation matrix operator for integral mean value LCBCs. All DOFs in a region are summed to form a single new DOF.

```
kind = 'integral_mean_value'
```

```
class sfepy.discrete.fem.lcbc_operators.LCBCOperator(name, regions, dof_names, dof_map_fun,  
                                                    variables, functions=None)
```

Base class for LCBC operators.

setup()

```
class sfepy.discrete.fem.lcbc_operators.LCBCOperators(name, variables, functions=None)
```

Container holding instances of LCBCOperator subclasses for a single variable.

add_from_bc(*bc, ts*)

Create a new LCBC operator described by *bc*, and add it to the container.

Parameters

bc

[LinearCombinationBC instance] The LCBC condition description.

ts

[TimeStepper instance] The time stepper.

append(*op*)

finalize()

Call this after all LCBCs of the variable have been added.

Initializes the global column indices and DOF counts.

make_global_operator(*adi, new_only=False*)

Assemble all LCBC operators into a single matrix.

Parameters

adi

[DofInfo] The active DOF information.

new_only

[bool] If True, the operator columns will contain only new DOFs.

Returns

mtx_lc

[csr_matrix] The global LCBC operator in the form of a CSR matrix.

rhs_lc

[array] The right-hand side for non-homogeneous LCBCs.

lcdi

[DofInfo] The global active LCBC-constrained DOF information.

```
class sfepy.discrete.fem.lcbc_operators.MRLCBCOperator(name, regions, dof_names, dof_map_fun,  
                                                       variables, functions=None)
```

Base class for model-reduction type LCBC operators.

These operators are applied to a single field, and replace its DOFs in a given region by new DOFs. In case some field DOFs are to be preserved, those have to be “copied” explicitly, by setting the corresponding row of the operator matrix to a single value one (see, for example, [NoPenetrationOperator](#)).

setup()

treat_pbcs(*dofs, master*)

Treat dofs with periodic BC.

class sfepy.discrete.fem.lcbc_operators.**MatchDOFsOperator**(*name, regions, dof_names, dof_map_fun, variables, ts, functions*)

Transformation matrix operator for match DOFs boundary conditions.

This operator ties DOFs of two fields in two disjoint regions together. It does not create any new DOFs.

kind = 'match_dofs'

class sfepy.discrete.fem.lcbc_operators.**MultiNodeLCOperator**(*name, regions, dof_names, dof_map_fun, constraints, variables, ts, functions*)

Transformation matrix operator that defines the DOFs at one (dependent) node as a linear combination of the DOFs at some other (independent) nodes.

The linear combination is given by:

$$\bar{u}_i = \sum_{j=1}^n c^j u_i^j ,$$

for all i in a given set of DOFs. $j = 1, \dots, n$ are the linear constraint indices and c^j are given weights of the independent nodes.

kind = 'multi_node_combination'

class sfepy.discrete.fem.lcbc_operators.**NoPenetrationOperator**(*name, regions, dof_names, dof_map_fun, filename, variables, ts=None, functions=None*)

Transformation matrix operator for no-penetration LCBCs.

kind = 'no_penetration'

class sfepy.discrete.fem.lcbc_operators.**NodalLCOperator**(*name, regions, dof_names, dof_map_fun, constraints, variables, ts=None, functions=None*)

Transformation matrix operator for the general linear combination of DOFs in each node of a field in the given region.

The DOFs can be fully constrained - then the operator corresponds to enforcing Dirichlet boundary conditions.

The linear combination is given by:

$$\sum_{j=1}^n A_{ij} u_j = b_i , \forall i ,$$

where $u_j, j = 1, \dots, n$ are the DOFs in the node and $i = 1, \dots, m, m < n$, are the linear constraint indices.

SymPy is used to solve the constraint linear system in each node for the dependent DOF(s).

kind = 'nodal_combination'

class sfepy.discrete.fem.lcbc_operators.**NormalDirectionOperator**(*name, regions, dof_names, dof_map_fun, filename, variables, ts=None, functions=None*)

Transformation matrix operator for normal direction LCBCs.

The substitution (in 3D) is:

$$[u_1, u_2, u_3]^T = [n_1, n_2, n_3]^T w$$

The new DOF is w .

```
get_vectors(nodes, region, field, filename=None)
```

```
kind = 'normal_direction'
```

```
class sfepy.discrete.fem.lcbc_operators.Rigid2Operator(name, regions, dof_names, dof_map_fun,
                                                    variables, ts=None, functions=None)
```

Transformation matrix operator for rigid body multi-point constraint LCBCs.

For each dependent node i in the first region, its DOFs u_j^i are given by the displacement $\bar{u}_j$ and rotation r_j in the single independent node in the second region:

$$u_j^i = \bar{u}_j + S_{ij}r_j$$

where the spin matrix S_{ij} is computed using the coordinates relative to the independent node, i.e. $\underline{x} - \bar{\underline{x}}$. Functionally it corresponds to the RBE2 multi-point constraint in MSC/Nastran.

A simplified version for fields without the rotation DOFs is also supported.

```
kind = 'rigid2'
```

```
class sfepy.discrete.fem.lcbc_operators.Rigid0Operator(name, regions, dof_names, dof_map_fun,
                                                    variables, ts=None, functions=None)
```

Transformation matrix operator for rigid LCBCs.

```
kind = 'rigid'
```

```
class sfepy.discrete.fem.lcbc_operators.ShiftedPeriodicOperator(name, regions, dof_names,
                                                                dof_map_fun, shift_fun,
                                                                variables, ts, functions)
```

Transformation matrix operator for shifted periodic boundary conditions.

This operator ties existing DOFs of two fields in two disjoint regions together. Unlike [MRLCBCOperator](#) subclasses, it does not create any new DOFs.

```
kind = 'shifted_periodic'
```

sfepy.discrete.fem.linearizer module

Linearization of higher order solutions for the purposes of visualization.

```
sfepy.discrete.fem.linearizer.create_output(eval_dofs, eval_coors, n_el, ps, min_level=0, max_level=2,
                                             eps=0.0001)
```

Create mesh with linear elements that approximates DOFs returned by `eval_dofs()` corresponding to a higher order approximation with a relative precision given by `eps`. The DOFs are evaluated in physical coordinates returned by `eval_coors()`.

```
sfepy.discrete.fem.linearizer.get_eval_coors(coors, conn, ps)
```

Get default function for evaluating physical coordinates given a list of elements and reference element coordinates.

```
sfepy.discrete.fem.linearizer.get_eval_dofs(dofs, dof_conn, ps, ori=None)
```

Get default function for evaluating field DOFs given a list of elements and reference element coordinates.

sfepy.discrete.fem.mappings module

Finite element reference mappings.

class sfepy.discrete.fem.mappings.**FEMapping**(*coors, conn, poly_space=None, gel=None, order=1*)

Base class for finite element mappings.

eval_basis(*coors, diff=False, grad_axes=None*)

Evaluate basis functions or their gradients in given coordinates.

get_geometry()

Return reference element geometry as a GeometryElement instance.

get_mapping(*qp_coors, weights, bf=None, poly_space=None, ori=None, transform=None, is_face=False, fc_bf_map=None, extra=(None, None, None)*)

Get the mapping for given quadrature points, weights, and polynomial space.

Parameters

qp_coors: **numpy.ndarray**

The coordinates of the integration points.

weights:

The integration weights.

bf: **numpy.ndarray**

The basis functions.

poly_space: **PolySpace** instance

The PolySpace instance.

ori: **numpy.ndarray**

Element orientation, used by hierarchical basis.

transform: **numpy.ndarray**

The transformation matrix applied to the basis functions.

is_face: **bool**

Is it the boundary of a region?

fc_bf_map: **tuple**

The additional info to remap face derivatives of wedge elements: - `fc_bf_map[0]`: id of face group (triangle or quad) - `fc_bf_map[1]`: position of inplane derivatives (xy-axes)

extra: **tuple**

The extra data for surface derivatives: - the derivatives of the field boundary basis functions with

respect to the reference coordinates

- the boundary connectivity
- the derivatives of the domain boundary basis functions with respect to the reference coordinates

Returns

pymap: **PyCMapping** instance

The domain mapping data.

get_physical_qps(*qp_coors*)

Get physical quadrature points corresponding to given reference element quadrature points.

Returns**qps**

[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

set_basis_indices(*indices*)

Set indices to cell-based basis that give the facet-based basis.

```
sfe.py.discrete.fem.mappings.eval_mapping_data_in_qp(coors, conn, bf_g, weights, ebf_g=None,
                                                    is_face=False, eps=1e-15, se_conn=None,
                                                    se_bf_bg=None, ecoors=None)
```

Evaluate mapping data.

Parameters**coors: numpy.ndarray**

The nodal coordinates.

conn: numpy.ndarray

The element connectivity.

bf_g: numpy.ndarray

The derivatives of the domain basis functions with respect to the reference coordinates.

weights: numpy.ndarray

The weights of the quadrature points.

ebf_g: numpy.ndarray

The derivatives of the field basis functions with respect to the reference coordinates.

is_face: bool

Is it the boundary of a region?

eps: float

The tolerance for the normal vectors calculation.

se_conn: numpy.ndarray

The connectivity for the calculation of surface derivatives.

se_bf_bg: numpy.ndarray

The surface basis function derivatives with respect to the reference coordinates.

ecoors: numpy.ndarray

The element nodal coordinates.

Returns**det: numpy.ndarray**

The determinant of the mapping evaluated in integration points.

volume: numpy.ndarray

The element (volume or surface) volumes in integration points.

bfg: numpy.ndarray

The derivatives of the basis functions with respect to the spatial coordinates. Can be evaluated either for surface elements if *bf_g*, *se_conn*, and *se_bf_bg* are given.

normal: numpy.ndarray

The normal vectors for the surface elements in integration points.

`sfepy.discrete.fem.mappings.transform_coors_to_lower_dim(coors, to_dim)`

Transform element coordinates into XY plane.

See:

<https://math.stackexchange.com/questions/1167717/transform-a-plane-to-the-xy-plane>
https://en.wikipedia.org/wiki/Rotation_matrix#Rotation_matrix_from_axis_and_angle

[https:](https://en.wikipedia.org/wiki/Rotation_matrix#Rotation_matrix_from_axis_and_angle)

sfepy.discrete.fem.mesh module

class `sfepy.discrete.fem.mesh.Mesh(name='mesh', cmesh=None)`

The Mesh class is a light proxy to CMesh.

Input and output is handled by the MeshIO class and subclasses.

property coors

copy(*name=None*)

Make a deep copy of the mesh.

Parameters

name

[str] Name of the copied mesh.

create_conn_graph(*verbose=True*)

Create a graph of mesh connectivity.

Returns

graph

[csr_matrix] The mesh connectivity graph as a SciPy CSR matrix.

static from_data(*name, coors, ngroups, conns, mat_ids, desc, nodal_bcs=None*)

Create a mesh from mesh IO data.

static from_file(*filename=None, io='auto', prefix_dir=None, omit_facets=False, file_format=None*)

Read a mesh from a file.

Parameters

filename

[string or function or MeshIO instance or Mesh instance] The name of file to read the mesh from. For convenience, a mesh creation function or a MeshIO instance or directly a Mesh instance can be passed in place of the file name.

io

[*MeshIO instance] Passing *MeshIO instance has precedence over filename.

prefix_dir

[str] If not None, the filename is relative to that directory.

omit_facets

[bool] If True, do not read cells of lower dimension than the space dimension (faces and/or edges). Only some MeshIO subclasses support this!

static from_region(*region, mesh_in, localize=False, is_surface=False, tdim=None*)

Create a mesh corresponding to cells, or lower dimensional entities according tdim parameter, of a given region. If *is_surface* is True, then tdim = dim - 1.

get_bounding_box()

get_cmesh(*desc*)

get_conn(*desc*, *ret_cells=False*, *tdim=None*)

Get the rectangular cell-vertex connectivity corresponding to *desc*. If *ret_cells* is True, the corresponding cells are returned as well.

transform_coors(*mtx_t*, *ref_coors=None*)

Transform coordinates of the mesh by the given transformation matrix.

Parameters

mtx_t

[array] The transformation matrix T (2D array). It is applied depending on its shape:

- (*dim*, *dim*): $x = T * x$
- (*dim*, *dim* + 1): $x = T[:, :-1] * x + T[:, -1]$

ref_coors

[array, optional] Alternative coordinates to use for the transformation instead of the mesh coordinates, with the same shape as *self.coors*.

write(*filename=None*, *io=None*, *out=None*, *float_format=None*, *file_format=None*, ***kwargs*)

Write mesh + optional results in *out* to a file.

Parameters

filename

[str, optional] The file name. If None, the mesh name is used instead.

io

[MeshIO instance or 'auto', optional] Passing 'auto' respects the extension of *filename*.

out

[dict, optional] The output data attached to the mesh vertices and/or cells.

float_format

[str, optional] The format string used to print floats in case of a text file format.

****kwargs**

[dict, optional] Additional arguments that can be passed to the *MeshIO* instance.

sfepy.discrete.fem.mesh.find_map(*x1*, *x2*, *allow_double=False*, *join=True*)

Find a mapping between common coordinates in *x1* and *x2*, such that $x1[cmap[:,0]] == x2[cmap[:,1]]$

sfepy.discrete.fem.mesh.fix_double_nodes(*coor*, *ngroups*, *conns*)

Detect and attempt fixing double nodes in a mesh.

The double nodes are nodes having the same coordinates w.r.t. precision given by *eps*.

sfepy.discrete.fem.mesh.get_min_vertex_distance(*coor*, *guess*)

Can miss the minimum, but is enough for our purposes.

sfepy.discrete.fem.mesh.get_min_vertex_distance_naive(*coor*)

sfepy.discrete.fem.mesh.make_mesh(*coor*, *ngroups*, *conns*, *mesh_in*)

Create a mesh reusing *mat_ids* and *descs* of *mesh_in*.

sfepy.discrete.fem.mesh.merge_mesh(*x1*, *ngroups1*, *conn1*, *mat_ids1*, *x2*, *ngroups2*, *conn2*, *mat_ids2*, *cmap*)

Merge two meshes in common coordinates found in *x1*, *x2*.

Notes

Assumes the same number and kind of element groups in both meshes!

```
sfepy.discrete.fem.mesh.set_accuracy(eps)
```

sfepy.discrete.fem.meshio module

```
class sfepy.discrete.fem.meshio.ANSYSCDBMeshIO(filename, **kwargs)
```

```
    format = 'ansys_cdb'
```

```
    static guess(filename)
```

```
    static make_format(format, nchar=1000)
```

```
    read(mesh, **kwargs)
```

```
    read_bounding_box()
```

```
    read_dimension(ret_fd=False)
```

```
    write(filename, mesh, out=None, **kwargs)
```

```
class sfepy.discrete.fem.meshio.ComsolMeshIO(filename, **kwargs)
```

```
    format = 'comsol'
```

```
    read(mesh, **kwargs)
```

```
    write(filename, mesh, out=None, **kwargs)
```

```
class sfepy.discrete.fem.meshio.GmshIO(filename, file_format=None, **kwargs)
```

Used to read and write data in .msh format when file_format gmsh-dg is specified. Tailored for use with Discontinuous galerking methods, mesh and ElementNodeData with InterpolationScheme can be written and read. It however omits mat_ids and node_groups.

For details on format see [1].

For details on representing and visualization of DG FEM data using gmsh see [2].

[1] <http://gmsh.info/doc/texinfo/gmsh.html#File-formats>

[2] Remacle, J.-F., Chevaugneon, N., Marchandise, E., & Geuzaine, C. (2007). Efficient visualization of high-order finite elements. International Journal for Numerical Methods in Engineering, 69(4), 750-771. <https://doi.org/10.1002/nme.1787>

```
    format = 'gmshio'
```

```
    load_slices = {'all': slice(0, None, None), 'first': slice(0, 1, None), 'last': slice(-1, None, None)}
```

```
    read_data(step=None, filename=None, cache=None)
```

Reads file or files with basename filename or self.filename. Considers all files to contain data from time steps of solution of single transient problem i.e. all data have the same shape, mesh and same interpolation scheme in case of ElementNodeData. Does not read mulitple NodeData or ElementData. For stationary problems just reads one file with time 0.0 and time step 0.

Providing filename allows reading multiple files of format *basename.*[0-9].msh*

Parameters**step**

[String, int, optional] “all”, “last”, “first” or number of step to read: if “all” read all files with the basename and varying step, if “last” read only last step of all files with the filename, if “first” reads step=0, if None reads file with filename provided or specified in object.

filename

[string, optional] Filename of the files to use, if None filename from object is used. Base-name is extracted as *basename.*[0-9].msh*

cache

[has no effect]

Returns**out**

[dictionary] Keys represent name of data, values are Structs with attributes:

data

[list, array] For ElementNodeData with shape (n_cell, n_cell_dof) contains for each time step. For other contains array of data from last time step.

time

[list] Contains times.

time_n

[list] Contains time step numbers.

scheme

[Struct] Interpolation scheme used in data, only one interpolation scheme is allowed.

scheme_name

[str] Name of the interpolation scheme, repeated for convenience.

mode

[str] Represents of type of data. cell_nodes : for ElementNodeData; vertex or cell : Note that for vertex and cell data reading multiple time steps does not work yet.

Notes

The interpolation scheme *Struct* contains the following items:

name

[string] Name of the scheme.

F

[array] Coefficients matrix as defined in [1] and [2].

P

[array] Exponents matrix as defined in [1] and [2].

write(filename, mesh, out=None, ts=None, **kwargs)

Writes mesh and data, handles cell DOFs data from DGField as ElementNodeData.

Omits gmsh:ref for cells and vertices i.e. mat_ids and node_groups to prevent cluttering the GMSH post-processing.

Parameters**filename**

[string] Path to file.

mesh
[sfepy.discrete.fem.mesh.Mesh] Computational mesh to write.

out
[dictionary] Keys represent name of the data, values are Structs with attributes:

data
[array] For ElementNodeData shape is (n_cell, n_cell_dof)

mode
[str] Represents type of data, cell_nodes for ElementNodeData.
For ElementNodeData:

scheme
[Struct] Interpolation scheme used in data, only one interpolation scheme is allowed.

scheme_name
[str] Name of the interpolation scheme, associated with data, repeated for convenience.

ts
[sfepy.solvers.ts.TimeStepper instance, optional] Provides data to write time step.

Notes

The interpolation scheme *Struct* contains the following items:

name
[string] Name of the scheme.

F
[array] Coefficients matrix as defined in [1] and [2].

P
[array] Exponents matrix as defined in [1] and [2].

```
class sfepy.discrete.fem.meshio.HDF5MeshIO(filename, **kwargs)
```

```
    format = 'hdf5'
```

```
    read(mesh=None, **kwargs)
```

```
    read_bounding_box(ret_fd=False, ret_dim=False)
```

```
    read_data(step, filename=None, cache=None)
```

```
    read_data_header(dname, step=None, filename=None)
```

```
    read_dimension(ret_fd=False)
```

```
    read_last_step(filename=None)
```

The default implementation: just return 0 as the last step.

```
    static read_mesh_from_hdf5(filename, group=None, mesh=None)
```

Read the mesh from a HDF5 file.

filename: str or tables.File

The HDF5 file to read the mesh from.

group: tables.group.Group or str, optional

The HDF5 file group to read the mesh from. If None, the root group is used.

mesh: `sfepy.discrete.fem.Mesh` or `None`

If `None`, the new mesh is created and returned, otherwise content of this argument is replaced by the read mesh.

Returns

`sfepy.discrete.fem.Mesh`
readed mesh

`read_time_history(node_name, indx, filename=None)`

`read_time_stepper(filename=None)`

`read_times(filename=None)`

Read true time step data from individual time steps.

Returns

steps
[array] The time steps.

times
[array] The times of the time steps.

nts
[array] The normalized times of the time steps, in [0, 1].

`read_variables_time_history(var_names, ts, filename=None)`

`string = <module 'string' from '/usr/lib/python3.10/string.py'>`

`write(filename, mesh, out=None, ts=None, cache=None, xdmf=False, **kwargs)`

`static write_mesh_to_hdf5(filename, group, mesh, force_3d=False)`

Write mesh to a hdf5 file.

filename: `str` or `tables.File`
The HDF5 file to write the mesh to.

group: `tables.group.Group` or `None` or `str`
The HDF5 file group to write the mesh to. If `None`, the root group is used. The group can be given as a path from root, e.g. `/path/to/mesh`

mesh: `sfepy.discrete.fem.Mesh`
The mesh to write.

`static write_xdmf_file(filename, **kwargs)`

`class sfepy.discrete.fem.meshio.HDF5XdmfMeshIO(filename, **kwargs)`

`format = 'hdf5-xdmf'`

`write(filename, mesh, out=None, ts=None, cache=None, **kwargs)`

`class sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO(filename, **kwargs)`

`format = 'hascii'`

`read(mesh, **kwargs)`

`read_dimension()`

```
write(filename, mesh, out=None, **kwargs)
```

```
class sfepy.discrete.fem.meshio.Mesh3DMeshIO(filename, **kwargs)
```

```
format = 'mesh3d'
```

```
read(mesh, **kwargs)
```

```
read_dimension()
```

```
class sfepy.discrete.fem.meshio.MeshIO(filename, **kwargs)
```

The abstract class for importing and exporting meshes.

Read the docstring of the Mesh() class. Basically all you need to do is to implement the read() method:

```
def read(self, mesh, **kwargs):
    nodes = ...
    ngroups = ...
    conns = ...
    mat_ids = ...
    descs = ...
    mesh._set_io_data(nodes, ngroups, conns, mat_ids, descs)
    return mesh
```

See the Mesh class' docstring how the nodes, ngroups, conns, mat_ids and descs should look like. You just need to read them from your specific format from disk.

To write a mesh to disk, just implement the write() method and use the information from the mesh instance (e.g. nodes, conns, mat_ids and descs) to construct your specific format.

Optionally, subclasses can implement read_data() to read also computation results. This concerns mainly the subclasses with implemented write() supporting the 'out' kwarg.

The default implementation of read_last_step() just returns 0. It should be reimplemented in subclasses capable of storing several steps.

```
static any_from_filename(filename, prefix_dir=None, file_format=None, mode='r')
```

Create a MeshIO instance according to the kind of *filename*.

Parameters

filename

[str, function or MeshIO subclass instance] The name of the mesh file. It can be also a user-supplied function accepting two arguments: *mesh*, *mode*, where *mesh* is a Mesh instance and *mode* is one of 'read', 'write', or a MeshIO subclass instance.

prefix_dir

[str] The directory name to prepend to *filename*.

Returns

io

[MeshIO subclass instance] The MeshIO subclass instance corresponding to the kind of *filename*.

```
call_msg = 'called an abstract MeshIO instance!'
```

```
format = None
```

```
get_filename_trunk()
```


get_vector_format(*dim*)

read(*mesh*, *omit_facets=False*, ***kwargs*)

read_data(*step*, *filename=None*, *cache=None*)

read_last_step()

The default implementation: just return 0 as the last step.

read_times(*filename=None*)

Read true time step data from individual time steps.

Returns

steps

[array] The time steps.

times

[array] The times of the time steps.

nts

[array] The normalized times of the time steps, in [0, 1].

Notes

The default implementation returns empty arrays.

set_float_format(*format=None*)

write(*filename*, *mesh*, ***kwargs*)

class sfepy.discrete.fem.meshio.**MeshioLibIO**(*filename*, *file_format=None*, ***kwargs*)

```
cell_types = {('hexahedron', 3): '3_8', ('line', 1): '1_2', ('line', 2): '1_2',
              ('line', 3): '1_2', ('quad', 2): '2_4', ('quad', 3): '2_4', ('tetra', 3): '3_4',
              ('triangle', 2): '2_3', ('triangle', 3): '2_3', ('wedge', 3): '3_6'}
```

format = 'meshio'

read(*mesh*, *omit_facets=False*, ***kwargs*)

read_bounding_box(*ret_dim=False*)

read_data(*step*, *filename=None*, *cache=None*)

Renames cell resp. vertex data with name “*:ref” to mat_id resp. node_groups

Parameters

step: has no effect

filename

[string, optional] The file name to use instead of self.filename.

cache: has no effect

Returns

out

[dictionary] Data loaded from file, keys are names. values are Structs with name repeated, mode (‘vertex’ or ‘cell’) and the data itself.

```
read_dimension()

write(filename, mesh, out=None, ts=None, **kwargs)

class sfepy.discrete.fem.meshio.NEUMeshIO(filename, **kwargs)

    format = 'gambit'

    read(mesh, **kwargs)

    read_dimension(ret_fd=False)

    write(filename, mesh, out=None, **kwargs)

class sfepy.discrete.fem.meshio.UserMeshIO(filename, **kwargs)
    Special MeshIO subclass that enables reading and writing a mesh using a user-supplied function.

    format = 'function'

    get_filename_trunk()

    read(mesh, *args, **kwargs)

    write(filename, mesh, *args, **kwargs)

class sfepy.discrete.fem.meshio.XYZMeshIO(filename, **kwargs)
    Trivial XYZ format working only with coordinates (in a .XYZ file) and the connectivity stored in another file
    with the same base name and .IEN suffix.

    format = 'xyz'

    read(mesh, omit_facets=False, **kwargs)

    read_bounding_box(ret_fd=False, ret_dim=False)

    read_dimension(ret_fd=False)

    write(filename, mesh, out=None, **kwargs)

sfepy.discrete.fem.meshio.check_format_suffix(file_format, suffix)
    Check compatibility of a mesh file format and a mesh file suffix.

sfepy.discrete.fem.meshio.convert_complex_output(out_in)
    Convert complex values in the output dictionary out_in to pairs of real and imaginary parts.

sfepy.discrete.fem.meshio.mesh_from_groups(mesh, ids, coors, ngroups, tris, mat_tris, quads, mat_quads,
                                           tetras, mat_tetras, hexas, mat_hexas, remap=None)

sfepy.discrete.fem.meshio.output_mesh_formats(mode='r')

sfepy.discrete.fem.meshio.split_conns_mat_ids(conns_in)
    Split connectivities (columns except the last ones in conns_in) from cell groups (the last columns of conns_in).

sfepy.discrete.fem.meshio.update_supported_formats(formats)

sfepy.discrete.fem.meshio.var
    alias of XYZMeshIO
```

sfepy.discrete.fem.periodic module

```

sfepy.discrete.fem.periodic.get_grid_plane(idim)
sfepy.discrete.fem.periodic.match_coors(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_grid_line(coors1, coors2, which, get_saved=True)
    Match coordinates coors1 with coors2 along the axis which.
sfepy.discrete.fem.periodic.match_grid_plane(coors1, coors2, idim, get_saved=True)
sfepy.discrete.fem.periodic.match_plane_by_dir(coors1, coors2, direction, get_saved=True)
    Match coordinates coors1 with coors2 in a given direction.
sfepy.discrete.fem.periodic.match_x_line(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_x_plane(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_y_line(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_y_plane(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_z_line(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_z_plane(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.set_accuracy(eps)

```

sfepy.discrete.fem.poly_spaces module

```

class sfepy.discrete.fem.poly_spaces.BernsteinSimplexPolySpace(name, geometry, order)
    Bernstein polynomial space on simplex domains.

```

Notes

Naive proof-of-concept implementation, does not use recurrent formulas or Duffy transformation to obtain tensor product structure.

```
name = 'bernstein_simplex'
```

```

class sfepy.discrete.fem.poly_spaces.BernsteinTensorProductPolySpace(name, geometry, order)
    Bernstein polynomial space.

```

Each row of the *nodes* attribute defines indices of 1D Bernstein basis functions that need to be multiplied together to evaluate the corresponding shape function. This defines the ordering of basis functions on the reference element.

```
name = 'bernstein_tensor_product'
```

```

class sfepy.discrete.fem.poly_spaces.FEPolySpace(name, geometry, order)
    Base for FE polynomial space classes.

    describe_nodes()

    get_mtx_i()

```

```
class sfepy.discrete.fem.poly_spaces.LagrangeNodes(**kwargs)
    Helper class for defining nodes of Lagrange elements.

    static append_bubbles(nodes, nts, iseq, nt, order)

    static append_edges(nodes, nts, iseq, nt, edges, order)

    static append_faces(nodes, nts, iseq, nt, faces, order)

    static append_tp_bubbles(nodes, nts, iseq, nt, ao)

    static append_tp_edges(nodes, nts, iseq, nt, edges, ao)

    static append_tp_faces(nodes, nts, iseq, nt, faces, ao)

class sfepy.discrete.fem.poly_spaces.LagrangePolySpace(name, geometry, order)

    create_context(cmesh, eps, check_errors, i_max, newton_eps, tdim=None)

class sfepy.discrete.fem.poly_spaces.LagrangeSimplexBPolySpace(name, geometry, order,
                                                                init_context=True)

    Lagrange polynomial space with forced bubble function on a simplex domain.

    create_context(*args, **kwargs)

    name = 'lagrange_simplex_bubble'

class sfepy.discrete.fem.poly_spaces.LagrangeSimplexPolySpace(name, geometry, order,
                                                                init_context=True)

    Lagrange polynomial space on a simplex domain.

    name = 'lagrange_simplex'

class sfepy.discrete.fem.poly_spaces.LagrangeTensorProductPolySpace(name, geometry, order,
                                                                init_context=True)

    Lagrange polynomial space on a tensor product domain.

    get_mtx_i()

    name = 'lagrange_tensor_product'

class sfepy.discrete.fem.poly_spaces.LagrangeWedgePolySpace(name, geometry, order,
                                                                init_context=True)

    name = 'lagrange_wedge'

class sfepy.discrete.fem.poly_spaces.LobattoTensorProductPolySpace(name, geometry, order)

    Hierarchical polynomial space using Lobatto functions.

    Each row of the nodes attribute defines indices of Lobatto functions that need to be multiplied together to evaluate
    the corresponding shape function. This defines the ordering of basis functions on the reference element.

    name = 'lobatto_tensor_product'

class sfepy.discrete.fem.poly_spaces.NodeDescription(node_types, nodes)

    Describe FE nodes defined on different parts of a reference element.

    has_extra_nodes()

    Return True if the element has some edge, face or bubble nodes.
```

```
class sfepy.discrete.fem.poly_spaces.SEMTensorProductPolySpace(name, geometry, order,  
                                                                init_context=True)
```

Spectral element method polynomial space = Lagrange polynomial space with Legendre-Gauss-Lobatto nodes. The same nodes and corresponding weights should be used for numerical quadrature to obtain a diagonal mass matrix.

```
name = 'sem_tensor_product'
```

```
class sfepy.discrete.fem.poly_spaces.SerendipityTensorProductPolySpace(name, geometry, order)
```

Serendipity polynomial space using Lagrange functions.

Notes

- Orders ≥ 4 (with bubble functions) are not supported.
- Does not use CLagrangeContext, basis functions are hardcoded.
- *self.nodes*, *self.node_coors* are not used for basis evaluation and assembling.

```

all_bfs = {2: {1: [[x*(y - 1.0) - y + 1.0, x*(1.0 - y), x*y, -x*y + y], [[y - 1.0,
x - 1.0], [1.0 - y, -x], [y, x], [-y, 1.0 - x]]], 2: [[x*(x*(2.0 - 2.0*y) + y*(5.0
- 2.0*y) - 3.0) + y*(2.0*y - 3.0) + 1.0, x*(x*(2.0 - 2.0*y) + y*(2.0*y - 1.0) -
1.0), x*(2.0*x*y + y*(2.0*y - 3.0)), x*(2.0*x*y + y*(-2.0*y - 1.0)) + y*(2.0*y -
1.0), x*(x*(4*y - 4) - 4*y + 4), x*y*(4.0 - 4.0*y), x*(-4.0*x*y + 4.0*y), x*y*(4.0*y
- 4.0) + y*(4.0 - 4.0*y)], [[x*(4.0 - 4.0*y) + y*(5.0 - 2.0*y) - 3.0, x*(-2.0*x -
4.0*y + 5.0) + 4.0*y - 3.0], [x*(4.0 - 4.0*y) + y*(2.0*y - 1.0) - 1.0, x*(-2.0*x +
4.0*y - 1.0)], [4.0*x*y + y*(2.0*y - 3.0), x*(2.0*x + 4.0*y - 3.0)], [4.0*x*y +
y*(-2.0*y - 1.0), x*(2.0*x - 4.0*y - 1.0) + 4.0*y - 1.0], [x*(8*y - 8) - 4*y + 4,
x*(4*x - 4)], [y*(4.0 - 4.0*y), x*(4.0 - 8.0*y)], [-8.0*x*y + 4.0*y, x*(4.0 -
4.0*x)], [y*(4.0*y - 4.0), x*(8.0*y - 4.0) - 8.0*y + 4.0]]], 3: [[x*(x*(x*(4.5*y -
4.5) - 9.0*y + 9.0) + y*(y*(4.5*y - 9.0) + 10.0) - 5.5) + y*(y*(9.0 - 4.5*y) - 5.5)
+ 1.0, x*(x*(x*(4.5 - 4.5*y) + 4.5*y - 4.5) + y*(y*(9.0 - 4.5*y) - 5.5) + 1.0),
x*(x*(4.5*x*y - 4.5*y) + y*(y*(4.5*y - 4.5) + 1.0)), x*(x*(-4.5*x*y + 9.0*y) +
y*(y*(4.5 - 4.5*y) - 5.5)) + y*(y*(4.5*y - 4.5) + 1.0), x*(x*(x*(13.5 - 13.5*y) +
22.5*y - 22.5) - 9.0*y + 9.0), x*(x*(x*(13.5*y - 13.5) - 18.0*y + 18.0) + 4.5*y -
4.5), x*y*(y*(13.5*y - 22.5) + 9.0), x*y*(y*(18.0 - 13.5*y) - 4.5), x*(x*(-13.5*x*y
+ 18.0*y) - 4.5*y), x*(x*(13.5*x*y - 22.5*y) + 9.0*y), x*y*(y*(13.5*y - 18.0) + 4.5)
+ y*(y*(18.0 - 13.5*y) - 4.5), x*y*(y*(22.5 - 13.5*y) - 9.0) + y*(y*(13.5*y - 22.5)
+ 9.0)], [[x*(x*(13.5*y - 13.5) - 18.0*y + 18.0) + y*(y*(4.5*y - 9.0) + 10.0) - 5.5,
x*(x*(4.5*x - 9.0) + y*(13.5*y - 18.0) + 10.0) + y*(18.0 - 13.5*y) - 5.5],
[x*(x*(13.5 - 13.5*y) + 9.0*y - 9.0) + y*(y*(9.0 - 4.5*y) - 5.5) + 1.0, x*(x*(4.5 -
4.5*x) + y*(18.0 - 13.5*y) - 5.5)], [x*(13.5*x*y - 9.0*y) + y*(y*(4.5*y - 4.5) +
1.0), x*(x*(4.5*x - 4.5) + y*(13.5*y - 9.0) + 1.0)], [x*(-13.5*x*y + 18.0*y) +
y*(y*(4.5 - 4.5*y) - 5.5), x*(x*(9.0 - 4.5*x) + y*(9.0 - 13.5*y) - 5.5) + y*(13.5*y
- 9.0) + 1.0], [x*(x*(40.5 - 40.5*y) + 45.0*y - 45.0) - 9.0*y + 9.0, x*(x*(22.5 -
13.5*x) - 9.0)], [x*(x*(40.5*y - 40.5) - 36.0*y + 36.0) + 4.5*y - 4.5, x*(x*(13.5*x
- 18.0) + 4.5)], [y*(y*(13.5*y - 22.5) + 9.0), x*(y*(40.5*y - 45.0) + 9.0)],
[y*(y*(18.0 - 13.5*y) - 4.5), x*(y*(36.0 - 40.5*y) - 4.5)], [x*(-40.5*x*y + 36.0*y)
- 4.5*y, x*(x*(18.0 - 13.5*x) - 4.5)], [x*(40.5*x*y - 45.0*y) + 9.0*y, x*(x*(13.5*x
- 22.5) + 9.0)], [y*(y*(13.5*y - 18.0) + 4.5), x*(y*(40.5*y - 36.0) + 4.5) + y*(36.0
- 40.5*y) - 4.5], [y*(y*(22.5 - 13.5*y) - 9.0), x*(y*(45.0 - 40.5*y) - 9.0) +
y*(40.5*y - 45.0) + 9.0]]], 3: {1: [[x*(y*(1.0 - z) + z - 1.0) + y*(z - 1.0) - z
+ 1.0, x*(y*(z - 1.0) - z + 1.0), x*y*(1.0 - z), x*y*(z - 1.0) + y*(1.0 - z), x*(y*z
- z) - y*z + z, x*(-y*z + z), x*y*z, -x*y*z + y*z], [[y*(1.0 - z) + z - 1.0, x*(1.0
- z) + z - 1.0, x*(1.0 - y) + y - 1.0], [y*(z - 1.0) - z + 1.0, x*(z - 1.0), x*(y -
1.0)], [y*(1.0 - z), x*(1.0 - z), -x*y], [y*(z - 1.0), x*(z - 1.0) - z + 1.0, x*y -
y], [y*z - z, x*z - z, x*(y - 1.0) - y + 1.0], [-y*z + z, -x*z, x*(1.0 - y)], [y*z,
x*z, x*y], [-y*z, -x*z + z, -x*y + y]]], 2: [[x*(x*(y*(2.0*z - 2.0) - 2.0*z + 2.0)
+ y*(y*(2.0*z - 2.0) + z*(2.0*z - 7.0) + 5.0) + z*(5.0 - 2.0*z) - 3.0) + y*(y*(2.0 -
2.0*z) + z*(5.0 - 2.0*z) - 3.0) + z*(2.0*z - 3.0) + 1.0, x*(x*(y*(2.0*z - 2.0) -
2.0*z + 2.0) + y*(y*(2.0 - 2.0*z) + z*(3.0 - 2.0*z) - 1.0) + z*(2.0*z - 1.0) - 1.0),
x*(x*y*(2.0 - 2.0*z) + y*(y*(2.0 - 2.0*z) + z*(2.0*z + 1.0) - 3.0)), x*(x*y*(2.0 -
2.0*z) + y*(y*(2.0*z - 2.0) + z*(3.0 - 2.0*z) - 1.0)) + y*(y*(2.0 - 2.0*z) +
z*(2.0*z - 1.0) - 1.0), x*(x*(-2.0*y*z + 2.0*z) + y*(-2.0*y*z + z*(2.0*z + 3.0)) +
z*(-2.0*z - 1.0)) + y*(2.0*y*z + z*(-2.0*z - 1.0)) + z*(2.0*z - 1.0), x*(x*(-2.0*y*z
+ 2.0*z) + y*(2.0*y*z + z*(1.0 - 2.0*z)) + z*(2.0*z - 3.0)), x*(2.0*x*y*z +
y*(2.0*y*z + z*(2.0*z - 5.0))), x*(2.0*x*y*z + y*(-2.0*y*z + z*(1.0 - 2.0*z))) +
y*(2.0*y*z + z*(2.0*z - 3.0)), x*(x*(y*(4.0 - 4.0*z) + 4.0*z - 4.0) + y*(4.0*z -
4.0) - 4.0*z + 4.0), x*y*(y*(4.0*z - 4.0) - 4.0*z + 4.0), x*(x*y*(4.0*z - 4.0) +
y*(4.0 - 4.0*z)), x*y*(y*(4.0 - 4.0*z) + 4.0*z - 4.0) + y*(y*(4.0*z - 4.0) - 4.0*z +
4.0), x*(x*(4.0*y*z - 4.0*z) - 4.0*y*z + 4.0*z), x*y*(-4.0*y*z + 4.0*z),
x*(-4.0*x*y*z + 4.0*y*z), x*y*(4.0*y*z - 4.0*z) + y*(-4.0*y*z + 4.0*z), x*(y*z*(4.0
- 4.0*z) + z*(4.0*z - 4.0)) + y*z*(4.0*z - 4.0) + z*(4.0 - 4.0*z), x*(y*z*(4.0*z -
4.0) + z*(4.0 - 4.0*z)), x*y*z*(4.0 - 4.0*z), x*y*z*(4.0*z - 4.0) + y*z*(4.0 -
4.0*z)], [[x*(y*(4.0*z - 4.0) - 4.0*z + 4.0) + y*(y*(2.0*z - 2.0) + z*(2.0*z - 7.0)
+ 5.0) + z*(5.0 - 2.0*z) - 3.0, x*(x*(2.0*z - 2.0) + y*(4.0*z - 4.0) + y*(2.0*y
7.0) + 5.0) + y*(4.0 - 4.0*z) + z*(5.0 - 2.0*z) - 3.0, x*(x*(2.0*y - 2.0) + y*(2.0*y
+ 4.0*z - 7.0) - 4.0*z + 5.0) + y*(-2.0*y - 4.0*z + 5.0) + 4.0*z - 3.0],
[x*(y*(4.0*z - 4.0) - 4.0*z + 4.0) + y*(y*(2.0 - 2.0*z) + z*(3.0 - 2.0*z) - 1.0) +

```

```
create_context(cmesh, eps, check_errors, i_max, newton_eps, tdim=None)
```

```
name = 'serendipity_tensor_product'
```

```
supported_orders = {1, 2, 3}
```

```
sfepy.discrete.fem.poly_spaces.eval_lagrange1d_basis(coors, ncoors)
```

```
sfepy.discrete.fem.poly_spaces.get_lgl_nodes(p)
```

Compute the Legendre-Gauss-Lobatto nodes and weights.

sfepy.discrete.fem.refine module

Basic uniform mesh refinement functions.

```
sfepy.discrete.fem.refine.refine_1_2(mesh_in)
```

Refines 1D mesh by cutting each element in half

```
sfepy.discrete.fem.refine.refine_2_3(mesh_in)
```

Refines mesh out of triangles by cutting each edge in half and making 4 new finer triangles out of one coarser one.

```
sfepy.discrete.fem.refine.refine_2_4(mesh_in)
```

Refines mesh out of quadrilaterals by cutting each edge in half and making 4 new finer quadrilaterals out of one coarser one.

```
sfepy.discrete.fem.refine.refine_3_4(mesh_in)
```

Refines tetrahedra by cutting each edge in half and making 8 new finer tetrahedra out of one coarser one. Old nodal coordinates come first in *coors*, then the new ones. The new tetrahedra are similar to the old one, no degeneration is supposed to occur as at most 3 congruence classes of tetrahedra appear, even when re-applied iteratively (provided that *conns* are not modified between two applications - ordering of vertices in tetrahedra matters not only for positivity of volumes).

References:

- Juergen Bey: Simplicial grid refinement: on Freudenthal's algorithm and the optimal number of congruence classes, Numer.Math. 85 (2000), no. 1, 1–29, or
- Juergen Bey: Tetrahedral grid refinement, Computing 55 (1995), no. 4, 355–378, or <http://citeseer.ist.psu.edu/bey95tetrahedral.html>

```
sfepy.discrete.fem.refine.refine_3_8(mesh_in)
```

Refines hexahedral mesh by cutting each edge in half and making 8 new finer hexahedrons out of one coarser one.

```
sfepy.discrete.fem.refine.refine_reference(geometry, level)
```

Refine reference element given by *geometry*.

Notes

The error edges must be generated in the order of the connectivity of the previous (lower) level.

sfepy.discrete.fem.refine_hanging module

Functions for a mesh refinement with hanging nodes.

Notes

Using LCBCs with hanging nodes is not supported.

`sfepy.discrete.fem.refine_hanging.find_facet_substitutions(facets, cells, sub_cells, refine_facets)`

Find facet substitutions in connectivity.

sub = [coarse cell, coarse facet, fine1 cell, fine1 facet, fine2 cell, fine2 facet]

`sfepy.discrete.fem.refine_hanging.find_level_interface(domain, refine_flag)`

Find facets of the coarse mesh that are on the coarse-refined cell boundary.

ids w.r.t. current mesh: - facets: global, local w.r.t. cells[:, 0], local w.r.t. cells[:, 1]

- interface cells: - cells[:, 0] - cells to refine - cells[:, 1] - their facet sharing neighbors (w.r.t. both meshes) - cells[:, 2] - facet kind: 0 = face, 1 = edge

`sfepy.discrete.fem.refine_hanging.refine(domain0, refine, subs=None, ret_sub_cells=False)`

`sfepy.discrete.fem.refine_hanging.refine_region(domain0, region0, region1)`

Coarse cell sub_cells[ii, 0] in mesh0 is split into sub_cells[ii, 1:] in mesh1.

The new fine cells are interleaved among the original coarse cells so that the indices of the coarse cells do not change.

The cell groups are preserved. The vertex groups are preserved only in the coarse (non-refined) cells.

sfepy.discrete.fem._serendipity module

sfepy.discrete.fem.utils module

`sfepy.discrete.fem.utils.compute_nodal_edge_dirs(nodes, region, field, return_imap=False)`

Nodal edge directions are computed by simple averaging of direction vectors of edges a node is contained in. Edges are assumed to be straight and a node must be on a single edge (a border node) or shared by exactly two edges.

`sfepy.discrete.fem.utils.compute_nodal_normals(nodes, region, field, return_imap=False)`

Nodal normals are computed by simple averaging of element normals of elements every node is contained in.

`sfepy.discrete.fem.utils.extend_cell_data(data, domain, rname, val=None, is_surface=False, average_surface=True)`

Extend cell data defined in a region to the whole domain.

Parameters

data

[array] The data defined in the region.

domain

[FEDomain instance] The FE domain.

rname

[str] The region name.

val

[float, optional] The value for filling cells not covered by the region. If not given, the smallest value in data is used.

is_surface

[bool] If True, the data are defined on a surface region. In that case the values are averaged or summed into the cells containing the region surface faces (a cell can have several faces of the surface), see *average_surface*.

average_surface

[bool] If True, the data defined on a surface region are averaged, otherwise the data are summed.

Returns**edata**

[array] The data extended to all domain elements.

`sfepy.discrete.fem.utils.get_edge_paths(graph, mask)`

Get all edge paths in a graph with non-masked vertices. The mask is updated.

`sfepy.discrete.fem.utils.get_min_value(dofs)`

Get a reasonable minimal value of DOFs suitable for extending over a whole domain.

`sfepy.discrete.fem.utils.invert_remap(remap)`

Return the inverse of *remap*, i.e. a mapping from a sub-range indices to a full range, see [prepare_remap\(\)](#).

`sfepy.discrete.fem.utils.prepare_remap(indices, n_full)`

Prepare vector for remapping range $[0, n\_full]$ to its subset given by *indices*.

`sfepy.discrete.fem.utils.prepare_translate(old_indices, new_indices)`

Prepare vector for translating *old_indices* to *new_indices*.

Returns**translate**

[array] The translation vector. Then $new\_ar = translate[old\_ar]$.

`sfepy.discrete.fem.utils.refine_mesh(filename, level)`

Uniformly refine *level*-times a mesh given by *filename*.

The refined mesh is saved to a file with name constructed from base name of *filename* and *level*-times appended *'_r'* suffix.

Parameters**filename**

[str] The mesh file name.

level

[int] The refinement level.

sfepy.discrete.dg sub-package

sfepy.discrete.dg.dg_1D_vizualizer module

Module for animating solutions in 1D. Can also save them but requires ffmpeg package see `save_animation` method.

`sfepy.discrete.dg.dg_1D_vizualizer.animate1D_dgsol(Y, X, T, ax=None, fig=None, ylims=None, labs=None, plott=None, delay=None)`

Animates solution of 1D problem into current figure. Keep reference to returned animation object otherwise it is discarded

Parameters

- Y**
solution, array `[T] x [X] x n`, where `n` is dimension of the solution
- X**
space interval discretization
- T**
time interval discretization
- ax**
specify axes to plot to (Default value = None)
- fig**
specify figure to plot to (Default value = None)
- ylims**
limits for y axis, default are 10% offsets of Y extremes
- labs**
labels to use for parts of the solution (Default value = None)
- plott**
plot type - how to plot data: tested plot, step (Default value = None)
- delay**
(Default value = None)

Returns

- anim**
the animation object, keep it to see the animation, used for saving too

`sfepy.discrete.dg.dg_1D_vizualizer.animate_1D_DG_sol(coors, t0, t1, u, tn=None, dt=None, ic=<function <lambda>>, exact=<function <lambda>>, delay=None, polar=False)`

Animates solution to 1D problem produced by DG:

1. animates DOF values in elements as steps
2. animates reconstructed solution with discontinuities

Parameters

- coors**
coordinates of the mesh
- t0**
[float] starting time

t1
[float] final time

u
vectors of DOFs, for each order one, $\text{shape}(u) = (\text{order}, \text{nspace\_steps}, \text{ntime\_steps}, 1)$

ic
analytical initial condition, optional (Default value = $\lambda x: 0.0$)

tn
number of time steps to plot, starting at 0, if None and dt is not None run animation through all time steps, spaced dt within $[t_0, t_n]$ (Default value = None)

dt
time step size, if None and tn is not None computed as $(t_1 - t_0) / t_n$ otherwise set to 1 if dt and tn are both None, t0 and t1 are ignored and solution is animated as if in time 0 ... ntime_steps (Default value = None)

exact
(Default value = λx)

t: 0

delay
(Default value = None)

polar
(Default value = False)

Returns

anim_dofs
[animation object of DOFs,]

anim_recon
[animation object of reconstructed solution]

`sfepy.discrete.dg.dg_1D_vizualizer.head()`

Maybe get head of the list.

Parameters

l
[indexable]

Returns

head
[first element in l or None is l is empty]

`sfepy.discrete.dg.dg_1D_vizualizer.load_1D_vtks(fold, name)`

Reads series of .vtk files and crunches them into form suitable for plot10_DG_sol.

Attempts to read modal cell data for variable mod_data. i.e.

$?_{\text{modal}\{i\}}$, where i is number of modal DOF

Resulting solution data have shape: (order, nspace_steps, ntime_steps, 1)

Parameters

fold
folder where to look for files

name

used in `{name}.i.vtk`, `i = 0, 1, ... tns - 1`

Returns**coors**

[ndarray]

mod_data

[ndarray] solution data

`sfepy.discrete.dg.dg_1D_vizualizer.load_state_1D_vtk(name)`

Load one VTK file containing state in time

Parameters**name**

[str]

Returns**coors**

[ndarray]

u

[ndarray]

`sfepy.discrete.dg.dg_1D_vizualizer.plot1D_legendre_dofs(coors, dofss, fun=None)`

Plots values of DOFs as steps

Parameters**coors**

coordinates of nodes of the mesh

dofss

iterable of different projections' DOFs into legendre space

fun

analytical function to plot (Default value = None)

`sfepy.discrete.dg.dg_1D_vizualizer.plotsXT(Y1, Y2, YE, extent, lab1=None, lab2=None, lab3=None)`

Plots Y1 and Y2 to one axes and YE to the second axes, Y1 and Y2 are presumed to be two solutions and YE their error

Parameters**Y1**

solution 1, shape = (space nodes, time nodes)

Y2

solution 2, shape = (space nodes, time nodes)

YE

soulutio 1 - soulution 2||

extent

imshow extent

lab1

(Default value = None)

lab2

(Default value = None)

lab3

(Default value = None)

`sfepy.discrete.dg.dg_1D_vizualizer.reconstruct_legendre_dofs(coors, tn, u)`

Creates solution and coordinates vector which when plotted as

`plot(xx, ww)`

represent solution reconstructed from DOFs in Legendre poly space at cell borders.

Works only as linear interpolation between cell boundary points

Parameters**coors**

coors of nodes of the mesh

uvectors of DOFs, for each order one, `shape(u) = (order, nspace_steps, ntime_steps, 1)`**tn**

number of time steps to reconstruct, if None all steps are reconstructed

Returns**ww**[ndarray] solution values vector, shape is `(3 * nspace_steps - 1, ntime_steps, 1)`,**xx**[ndarray] corresponding coordinates vector, shape is `(3 * nspace_steps - 1, 1)``sfepy.discrete.dg.dg_1D_vizualizer.save_animation(anim, filename)`

Saves animation as .mp4, requires ffmpeg package

Parameters**anim**

animation object

filename

name of the file, without the .mp4 ending

`sfepy.discrete.dg.dg_1D_vizualizer.save_sol_snap(Y, X, T, t0=0.5, filename=None, name=None, ylims=None, labs=None, plott=None)`Wrapper for `sol_frame`, saves the frame to file specified.**Parameters****name**

name of the solution e.g. name of the solver used (Default value = None)

filename

name of the file, overrides automatic generation (Default value = None)

Ysolution, array `[T] x [X] x n`, where n is dimension of the solution**X**

space interval discetization

T

time interval discretization

t0

time to take snap at (Default value = .5)

ylims

limits for y axis, default are 10% offsets of Y extremes

labs

labels to use for parts of the solution (Default value = None)

plott

plot type - how to plot data: tested plot, step (Default value = None)

Returns**fig**

`sfepy.discrete.dg.dg_1D_vizualizer.setup_axis(X, Y, ax=None, fig=None, ylims=None)`

Setup axis, including timer for animation or snaps

Parameters**X**

space discretization to get limits

Y

solution to get limits

ax

ax where to put everything, if None current axes are used (Default value = None)

fig

fig where to put everything, if None current figure is used (Default value = None)

ylims

custom ylims, if None y axis limits are calculated from Y (Default value = None)

Returns**ax****fig****time_text**

object to fill in text

`sfepy.discrete.dg.dg_1D_vizualizer.setup_lines(ax, Yshape, labs, plott)`

Sets up artist for animation or solution snaps

Parameters**ax**

axes to use for artist

Yshape

[tuple] shape of the solution array

labs

[list] labels for the solution

plott

[str ("steps" or "plot")] type of plot to use

Returns**lines**

`sfepy.discrete.dg.dg_1D_vizualizer.sol_frame(Y, X, T, t0=0.5, ax=None, fig=None, ylims=None, labs=None, plott=None)`

Creates snap of solution at specified time frame `t0`, basically gets one frame from `animate1D_dgsol`, but colors wont be the same :-)

Parameters

- Y**
solution, array `[T] x [X] x n`, where `n` is dimension of the solution
- X**
space interval discetization
- T**
time interval discretization
- t0**
time to take snap at (Default value = .5)
- ax**
specify axes to plot to (Default value = None)
- fig**
specifiy figure to plot to (Default value = None)
- ylims**
limits for y axis, default are 10% offsets of Y extremes
- labs**
labels to use for parts of the solution (Default value = None)
- plott**
plot type - how to plot data: tested plot, step (Default value = None)

Returns

fig

`sfepy.discrete.dg.fields` module

Fields for Discontinous Galerkin method

```
class sfepy.discrete.dg.fields.DGField(name, dtype, shape, region, space='H1',
                                     poly_space_basis='legendre', approx_order=1, integral=None)
```

Discontinuous Galerkin method approximation with Legendre basis.

Class for usage with DG terms, provides functionality for Discontinous Galerkin method like neighbour look up, projection to discontinuous basis and correct DOF treatment.

```
clear_facet_neighbour_idx_cache(region=None)
```

If region is None clear all!

Parameters

region
[`sfepy.discrete.common.region.Region`] If None clear all.

```
clear_facet_qp_basis()
```

Clears `facet_qp_basis` cache

clear_facet_vols_cache(*region=None*)

Clears facet volume cache for given region or all regions.

Parameters**region**

[sfepy.discrete.common.region.Region] region to clear cache or None to clear all

clear_normals_cache(*region=None*)

Clears normals cache for given region or all regions.

Parameters**region**

[sfepy.discrete.common.region.Region] region to clear cache or None to clear all

create_mapping(*region, integral, integration, return_mapping=True*)

Creates and returns mapping

Parameters**region**

[sfepy.discrete.common.region.Region]

integral

[Integral]

integration

[str] 'volume' is only accepted option

return_mapping

[default True] (Default value = True)

Returns**mapping**

[FEMapping]

create_output(*dofs, var_name, dof_names=None, key=None, extend=True, fill_value=None, linearization=None*)

Converts the DOFs corresponding to the field to a dictionary of output data usable by Mesh.write().

For 1D puts DOFs into vairables `u_modal{0} ... u_modal{n}`, where `n = approx_order` and marks them for writing as cell data.

For 2+D puts dofs into `name_cell_nodes` and creates sturct with: `mode = "cell_nodes"`, data and interpolation scheme.

Also get node values and adds them to dictionary as `cell_nodes`

Parameters**dofs**

[ndarray, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

var_name

[str] The variable name corresponding to *dofs*.

dof_names

[tuple of str] The names of DOF components. (Default value = None)

key

[str, optional] The key to be used in the output dictionary instead of the variable name.
(Default value = None)

extend

[bool, not used] Extend the DOF values to cover the whole domain. (Default value = True)

fill_value

[float or complex, not used] The value used to fill the missing DOF values if *extend* is True.
(Default value = None)

linearization

[Struct or None, not used] The linearization configuration for higher order approximations.
(Default value = None)

Returns**out**

[dict]

family_name = 'volume_DG_legendre_discontinuous'

get_bc_facet_idx(*region*)

Caches results in self.boundary_facet_local_idx

Parameters**region**

[sfepy.discrete.common.region.Region] surface region defining BCs

Returns**bc2bfi**

[ndarray] index of cells on boundary along with corresponding facets

get_bc_facet_values(*fun*, *region*, *ret_coors=False*, *diff=0*)

Returns values of fun in facet QPs of the region

Parameters

diff: derivative 0 or 1 supported

fun: Function value or values to set qps values to

region

[sfepy.discrete.common.region.Region] boundary region

ret_coors: default False,

Return physical coors of qps in shape (n_cell, n_qp, dim).

Returns**vals**

[ndarray] In shape (n_cell,) + (self.dim,) * diff + (n_qp,)

get_both_facet_basis_vals(*state*, *region*, *derivative=None*)

Returns values of the basis function in quadrature points on facets broadcasted to all cells inner to the element as well as outer ones along with weights for the qps broadcasted and transformed to elements.

Contains quick fix to flip facet QPs for right integration order.

Parameters**state**

[used to get EPBC info]

region
[sfepy.discrete.common.region.Region for connectivity]

derivative
[if u need derivative] (Default value = None)

Returns

outer_facet_basis_vals:

inner_facet_basis_vals:

shape (n_cell, n_el_nod, n_el_facet, n_qp) or
(n_cell, n_el_nod, n_el_facet, dim, n_qp)

when derivative is True or 1

whs: shape (n_cell, n_el_facet, n_qp)

get_both_facet_state_vals(state, region, derivative=None, reduce_nod=True)

Computes values of the variable represented by dofs in quadrature points located at facets, returns both values - inner and outer, along with weights.

Parameters

state
[state variable containing BC info]

region
[sfepy.discrete.common.region.Region]

derivative
[compute derivative if truthy,] compute n-th derivative if a number (Default value = None)

reduce_nod
[if False DOES NOT sum nodes into values at QPs] (Default value = True)

Returns

inner_facet_values (n_cell, n_el_facets, n_qp),
outer facet values (n_cell, n_el_facets, n_qp), weights, if derivative is True:
inner_facet_values (n_cell, n_el_facets, dim, n_qp), outer_facet values (n_cell, n_el_facets, dim, n_qp)

get_cell_normals_per_facet(region)

Caches results, use clear_normals_cache to clear the cache.

Parameters

region: sfepy.discrete.common.region.Region
Main region, must contain cells.

Returns

normals: ndarray
normals of facets in array of shape (n_cell, n_el_facets, dim)

get_coor(nods=None)

Returns coors for matching nodes # TODO revise DG_EPBC and EPBC matching?

Parameters

nods
if None use all nodes (Default value = None)

Returns**coors**

[ndarray] coors on surface

get_data_shape(*integral*, *integration*='cell', *region_name*=None)

Returns data shape (n_nod, n_qp, self.gel.dim, self.n_el_nod)

Parameters**integral**

[integral used]

integration

'volume' is only supported value (Default value = 'volume')

region_name

[not used] (Default value = None)

Returns**data_shape**

[tuple]

get_dofs_in_region(*region*, *merge*=True)

Return indices of DOFs that belong to the given region.

Not Used in BC treatment

Parameters**region**

[sfepy.discrete.common.region.Region]

merge

[bool] merge dof tuple into one numpy array, default True

Returns**dofs**

[ndarray]

get_econn(*conn_type*, *region*, *trace_region*=None)

Getter for econn

Parameters**conn_type**

[tuple or string] ('cell', dim) or 'cell' is only supported

region

[sfepy.discrete.common.region.Region]

trace_region

[ignored] (Default value = None)

Returns**econn**

[ndarray] connectivity information

get_facet_basis(*derivative*=False, *basis_only*=False)

Returns values of basis in facets quadrature points, data shape is a bit crazy right now:

(number of qps, 1, n_el_facets, 1, n_el_nod)

end for derivatine:

(1, number of qps, (dim,) * derivative, n_el_facets, 1, n_el_nod)

Parameters

derivative: truthy or integer

basis_only: do not return weights

Returns

facet_bf

[ndarray] values of basis functions in facet qps

weights

[ndarray, optionally] weights of qps

get_facet_neighbor_idx(*region=None, eq_map=None*)

Returns index of cell neighbours sharing facet, along with local index of the facet within neighbour, also treats periodic boundary conditions i.e. plugs correct neighbours for cell on periodic boundary. Where there are no neighbours specified puts -1 instead of neighbour and facet id

Cashes neighbour index in self.facet_neighbours

Parameters

region

[sfepy.discrete.common.region.Region] Main region, must contain cells.

eq_map

eq_map from state variable containing information on EPBC and DG EPBC. (Default value = None)

Returns

facet_neighbours

[ndarray]

Shape is

(n_cell, n_el_facet, 2),

first value is index of the neighbouring cell, the second is index of the facet in said nb. cell.

get_facet_qp()

Returns quadrature points on all facets of the reference element in array of shape (n_qp, 1, n_el_facets, dim)

Returns

qps

[ndarray] quadrature points

weights

[ndarray] Still needs to be transformed to actual facets!

get_facet_vols(*region*)

Cashes results, use clear_facet_vols_cache to clear the cache

Parameters

region

[sfepy.discrete.common.region.Region]

Returns

vols_out: ndarray
volumes of the facets by cells shape (n_cell, n_el_facets, 1)

get_nodal_values(*dofs*, *region*, *ref_nodes=None*)

Computes nodal representation of the DOFs

Parameters

dofs

[array_like] dofs to transform to nodes

region : ignored

ref_nodes:

reference node to use instead of default qps

Parameters

dofs

[array_like]

region

[Region]

ref_nodes

[array_like] (Default value = None)

Returns

nodes

[ndarray]

nodal_vals

[ndarray]

static get_region_info(*region*)

Extracts information about region needed in various methods of DGField

Parameters

region

[sfepy.discrete.common.region.Region]

Returns

dim, n_cell, n_el_facets

is_surface = False

set_cell_dofs(*fun=0.0*, *region=None*, *dpn=None*, *warn=None*)

Compute projection of fun onto the basis, in main region, alternatively set DOFs directly to provided value or values

Parameters

fun

[callable, scallar or array corresponding to dofs] (Default value = 0.0)

region

[sfepy.discrete.common.region.Region] region to set DOFs on (Default value = None)

dpn

[number of dofs per element] (Default value = None)

warn

[not used] (Default value = None)

Returns**nods**

[ndarray]

vals

[ndarray]

set_dofs (*fun=0.0, region=None, dpn=None, warn=None*)

Compute projection of *fun* into the basis, alternatively set DOFs directly to provided value or values either in main volume region or in boundary region.

Parameters**fun**

[callable, scalar or array corresponding to dofs] (Default value = 0.0)

region

[sfepy.discrete.common.region.Region] region to set DOFs on (Default value = None)

dpn

[number of dofs per element] (Default value = None)

warn

(Default value = None)

Returns**nods**

[ndarray]

vals

[ndarray]

set_facet_dofs (*fun, region, dpn, warn*)

Compute projection of *fun* onto the basis on facets, alternatively set DOFs directly to provided value or values

Parameters**fun**

[callable, scalar or array corresponding to dofs]

region

[sfepy.discrete.common.region.Region] region to set DOFs on

dpn

[int] number of dofs per element

warn

not used

Returns**nods**

[ndarray]

vals
[ndarray]

setup_extra_data(*info*)

This is called in `create_adof_conns(conn_info, var_indx=None, active_only=True, verbose=True)`

for each variable but has no effect.

Parameters

info
set to self.info

`sfepy.discrete.dg.fields.get_gel(region)`

Parameters

region
[sfepy.discrete.common.region.Region]

Returns

gel
base geometry element of the region

`sfepy.discrete.dg.fields.get_raveler(n_el_nod, n_cell)`

Returns function for raveling i.e. packing dof data from two dimensional array of shape (n_cell, n_el_nod, 1) to (n_el_nod*n_cell, 1)

The raveler returns view into the input array.

Parameters

n_el_nod
param n_el_nod, n_cell: expected dimensions of dofs array

n_cell
[int]

Returns

ravel
[callable]

`sfepy.discrete.dg.fields.get_unraveler(n_el_nod, n_cell)`

Returns function for unraveling i.e. unpacking dof data from serialized array from shape (n_el_nod*n_cell, 1) to (n_cell, n_el_nod, 1).

The unraveler returns non-writeable view into the input array.

Parameters

n_el_nod
[int] expected dimensions of dofs array

n_cell
[int]

Returns

unravel
[callable]

sfepy.discrete.dg.poly_spaces module**class** sfepy.discrete.dg.poly_spaces.**LegendrePolySpace**(*name, geometry, order, extended*)

Legendre hierarchical polynomials basis, over [0, 1] domain.

get_interpol_scheme()

For dim > 1 returns F and P matrices according to gmsh basis specification [1]: Let us assume that the approximation of the view's value over an element is written as a linear combination of d basis functions $f_i, i = 0, \dots, n - 1$ (the coefficients being stored in list-of-values).

Defining

$$f_i = \sum_{j=0}^{d-1} F_{ij} \cdot p_j,$$

with

$p_j(u, v, w) = u^{P_j^{(0)}} \cdot v^{P_j^{(1)}} \cdot w^{P_j^{(2)}}$ (u, v and w being the coordinates in the element's parameter space), then val-coef-matrix denotes the $n \times n$ matrix F and val-exp-matrix denotes the $n \times 3$ matrix P where n is number of basis functions as calculated by `get_n_el_nod`.

Expects matrices to be saved in attributes coefM and expoM!

Returns**interp_scheme_struct**

[Struct] Struct with name of the scheme, geometry desc and P and F

get_nth_fun(*n*)

Uses shifted Legendre polynomials formula on interval [0, 1].

Convenience function for testing

Parameters**n**

[int]

Returns**fun**

[callable] n-th function of the legendre basis

get_nth_fun_der(*n, diff=1*)

Returns diff derivative of nth function. Uses shifted legendre polynomials formula on interval [0, 1].

Useful for testing.

Parameters**n**

[int]

diff

[int] (Default value = 1)

Returns**fun**

[callable] derivative of n-th function of the 1D legendre basis

gradjacobiP(*coors*, *alpha*, *beta*, *diff=1*)

diff derivative of the jacobi polynomials on interval $[-1, 1]$ up to `self.order + 1` at *coors*

Parameters

coors

alpha

[float]

beta

[float]

diff

[int] (Default value = 1)

Returns

values

[ndarray] output shape is `shape(coor) + (self.order + 1,)`

gradlegendreP(*coors*, *diff=1*)

Parameters

diff

[int] default 1

coors

[array_like] coordinates, preferably in interval $[-1, 1]$ for which this basis is intended

Returns

values

[ndarray] values at *coors* of all the legendre polynomials up to `self.order`

jacobiP(*coors*, *alpha*, *beta*)

Values of the jacobi polynomials on interval $[-1, 1]$ up to `self.order + 1` at *coors*

Parameters

coors

[array_like]

beta

[float]

alpha

[float]

Returns

values

[ndarray] output shape is `shape(coor) + (self.order + 1,)`

legendreP(*coors*)

Parameters

coors

[array_like] coordinates, preferably in interval $[-1, 1]$ for which this basis is intended

Returns

values

[ndarray] values at *coors* of all the legendre polynomials up to `self.order`

```
legendre_funs = [<function LegendrePolySpace.<lambda>>, <function
LegendrePolySpace.<lambda>>, <function LegendrePolySpace.<lambda>>, <function
LegendrePolySpace.<lambda>>, <function LegendrePolySpace.<lambda>>, <function
LegendrePolySpace.<lambda>>]
```

```
class sfepy.discrete.dg.poly_spaces.LegendreSimplexPolySpace(name, geometry, order,
                                                             extended=False)
```

```
    name = 'legendre_simplex'
```

```
class sfepy.discrete.dg.poly_spaces.LegendreTensorProductPolySpace(name, geometry, order)
```

```
    build_interpol_scheme()
```

Builds F and P matrices returned by self.get_interpol_scheme.

Note that this function returns coefficients according to gmsh parametrization of Quadrangle i.e. $[-1, 1] \times [-1, 1]$ and hence the form of basis function is not the same as exhibited by the LegendreTensorProductPolySpace object which acts on parametrization $[0, 1] \times [0, 1]$.

Returns

F

[ndarray] coefficient matrix

P

[ndarray] exponent matrix

```
    name = 'legendre_tensor_product'
```

```
sfepy.discrete.dg.poly_spaces.get_n_el_nod(order, dim, extended=False)
```

Number of nodes per element for discontinuous legendre basis, i.e. number of iterations yielded by iter_by_order

When extended is False

$$N_p = \frac{(n+1) \cdot (n+2) \cdot \dots \cdot (n+d)}{d!}$$

where n is the order and d the dimension. When extended is True

$$N_p = (n+1)^d$$

where n is the order and d the dimension.

Parameters

order

[int] desired order of multidimensional basis

dim

[int] dimension of the basis

extended

[bool] iterate over extended tensor product basis (Default value = False)

Returns

n_el_nod

[int] number of basis functions in basis

`sfepy.discrete.dg.poly_spaces.iter_by_order(order, dim, extended=False)`

Iterates over all combinations of basis functions indexes needed to create multidimensional basis in a way that creates hierarchical basis

Parameters

order

[int] desired order of multidimensional basis

dim

[int] dimension of the basis

extended

[bool] iterate over extended tensor product basis (Default value = False)

Yields

idx

[tuple] containing basis function indexes, used in `_combine_polyvals` and `_combine_polyvals_der`

sfepy.discrete.dg.limiters module

Limiters for high order DG methods

class `sfepy.discrete.dg.limiters.ComposedLimiter(fields, limiters, verbose=False)`

class `sfepy.discrete.dg.limiters.DGLimiter(field, verbose=False)`

name = 'abstract DG limiter'

class `sfepy.discrete.dg.limiters.IdentityLimiter(field, verbose=False)`

Neutral limiter returning unchanged solution.

name = 'identity'

class `sfepy.discrete.dg.limiters.MomentLimiter1D(field, verbose=False)`

Moment limiter for 1D based on [1]

name = 'moment_1D_limiter'

class `sfepy.discrete.dg.limiters.MomentLimiter2D(field, verbose=False)`

Moment limiter for 2D based on [R31316dc91f1d-1] .. [R31316dc91f1d-1] Krivodonova (2007):

Limiters for high-order discontinuous Galerkin methods

name = 'moment_limiter_2D'

`sfepy.discrete.dg.limiters.minmod(a, b, c)`

Minmod function of three variables, returns:

0 , where $\text{sign}(a) \neq \text{sign}(b) \neq \text{sign}(c)$

$\min(a,b,c)$, elsewhere

Parameters

a

[array_like]

c

[array_like]

b
[array_like]

Returns

out
[ndarray]

`sfepy.discrete.dg.limiters.minmod_seq(abc)`

Minmod function of n variables, returns:

0 , where $\text{sign}(a_1) \neq \text{sign}(a_2) \neq \dots \neq \text{sign}(a_n)$

$\min(a_1, a_2, a_3, \dots, a_n)$, elsewhere

Parameters

abc
[sequence of array_like]

Returns

out
[ndarray]

sfepy.solvers.ts_dg_solvers module

Explicit time stepping solvers for use with DG FEM

class `sfepy.solvers.ts_dg_solvers.DGMultiStageTSS`(*conf*, *nls=None*, *context=None*, ***kwargs*)

Explicit time stepping solver with multistage solve_step method

Kind: 'ts.multistaged'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

t0
[float (default: 0.0)] The initial time.

t1
[float (default: 1.0)] The final time.

dt
[float] The time step. Used if *n_step* is not given.

n_step
[int (default: 10)] The number of time steps. Has precedence over *dt*.

quasistatic
[bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver is invoked also for the initial time.

limiters
[dictionary] Limiters for DGFields, keys: field name, values: limiter class

name = 'ts.multistaged'

output_step_info(*ts*)

solve_step(*ts, nls, vec, prestep_fun=None, poststep_fun=None, status=None*)

solve_step0(*nls, vec0*)

class sfepy.solvers.ts_dg_solvers.**EulerStepSolver**(*conf, nls=None, context=None, **kwargs*)

Simple forward euler method

Kind: 'ts.euler'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'ts.euler'

solve_step(*ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None*)

class sfepy.solvers.ts_dg_solvers.**RK4StepSolver**(*conf, nls=None, context=None, **kwargs*)

Classical 4th order Runge-Kutta method, implemetantions is based on [1]

Kind: 'ts.runge_kutta_4'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'ts.runge_kutta_4'

solve_step(*ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None*)

stage_updates = (<function RK4StepSolver.<lambda>>, <function
RK4StepSolver.<lambda>>, <function RK4StepSolver.<lambda>>, <function
RK4StepSolver.<lambda>>)

class sfepy.solvers.ts_dg_solvers.**TVDRK3StepSolver**(*conf, nls=None, context=None, **kwargs*)

3rd order Total Variation Diminishing Runge-Kutta method based on [1]

$$\begin{aligned}\mathbf{p}^{(1)} &= \mathbf{p}^n - \Delta t \bar{\mathcal{L}}(\mathbf{p}^n), \\ \mathbf{p}^{(2)} &= \frac{3}{4}\mathbf{p}^n + \frac{1}{4}\mathbf{p}^{(1)} - \frac{1}{4}\Delta t \bar{\mathcal{L}}(\mathbf{p}^{(1)}), \\ \mathbf{p}^{(n+1)} &= \frac{1}{3}\mathbf{p}^n + \frac{2}{3}\mathbf{p}^{(2)} - \frac{2}{3}\Delta t \bar{\mathcal{L}}(\mathbf{p}^{(2)}).\end{aligned}$$

Kind: 'ts.tvd_runge_kutta_3'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'ts.tvd_runge_kutta_3'

solve_step(*ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None*)

sfepy.discrete.iga sub-package

sfepy.discrete.iga.domain module

Computational domain for isogeometric analysis.

class sfepy.discrete.iga.domain.**IGDomain**(*name, nurbs, bmesh, regions=None, **kwargs*)

Bezier extraction based NURBS domain for isogeometric analysis.

static from_data(*knots, degrees, cps, weights, cs, conn, bcps, bweights, bconn, regions, name='iga_domain_from_data'*)

Create the IGA domain from the given data.

static from_file(*filename*)

filename

[str] The name of the IGA domain file.

static read_domain_from_hdf5(*fd, group*)

Create a domain from the given hdf5 data group.

fd: tables.File

HDF5 file handle to read the mesh from.

group: tables.group.Group

HDF5 data group (of file fd) to read the mesh from.

write_domain_to_hdf5(*fd, group*)

Save the domain to a hdf5 file.

fd: tables.File

HDF5 file handle to write the mesh to.

group: tables.group.Group

HDF5 data group (of file fd) to write the mesh to.

class sfepy.discrete.iga.domain.**NurbsPatch**(*knots, degrees, cps, weights, cs, conn*)

Single NURBS patch data.

elevate(*times=0*)

Elevate the patch degrees several *times* by one.

Returns

nurbs

[NurbsPatch instance] Either *self* if *times* is zero, or a new instance.

evaluate(*field, u=None, v=None, w=None*)

Igakit-like interface for NURBS evaluation.

sfepy.discrete.iga.domain_generators module

IGA domain generators.

`sfepy.discrete.iga.domain_generators.create_from_igakit(inurbs, verbose=False)`

Create *IGDomain* data from a given igakit NURBS object.

Parameters**inurbs**

[igakit.nurbs.NURBS instance] The igakit NURBS object.

Returns**nurbs**

[NurbsPatch instance] The NURBS data. The igakit NURBS object is stored as *nurbs* attribute.

bmesh

[Struct instance] The Bezier mesh data.

regions

[dict] The patch surface regions.

`sfepy.discrete.iga.domain_generators.gen_patch_block_domain(dims, shape, centre, degrees,
continuity=None, cp_mode='greville',
name='block', verbose=True)`

Generate a single IGA patch block in 2D or 3D of given degrees and continuity using igakit.

Parameters**dims**

[array of D floats] Dimensions of the block.

shape

[array of D ints] Numbers of unique knot values along each axis.

centre

[array of D floats] Centre of the block.

degrees

[array of D floats] NURBS degrees along each axis.

continuity

[array of D ints, optional] NURBS continuity along each axis. If None, *degrees-1* is used.

cp_mode

['greville' or 'uniform'] The control points mode. The default 'greville' results in a uniform Bezier mesh, while the 'uniform' mode results in a uniform grid of control points a finer Bezier mesh inside the block and a coarser Bezier mesh near the block boundary.

name

[string] Domain name.

verbose

[bool] If True, report progress of the domain generation.

Returns**nurbs**

[NurbsPatch instance] The NURBS data. The igakit NURBS object is stored as *nurbs* attribute.

bmesh
[Struct instance] The Bezier mesh data.

regions
[dict] The patch surface regions.

sfepy.discrete.iga.extmods.igac module

class sfepy.discrete.iga.extmods.igac.CNURBSContext

R
bf
bfq
bufBN
cprint()
dR_dx
dR_dxi
e_coors_max
evaluate(*coors, diff=False, **kwargs*)
iel

sfepy.discrete.iga.extmods.igac.eval_bernstein_basis(*funs, ders, x, degree*)

sfepy.discrete.iga.extmods.igac.eval_in_tp_coors(*variable, indices, ref_coors, control_points, weights, degrees, cs, conn*)

Evaluate a field variable (if given) or the NURBS geometry in the given tensor-product reference coordinates. The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters

variable
[array] The DOF values of the variable with *n_c* components, shape $(:, n_c)$.

indices
[list of arrays] The indices of knot spans for each axis, defining the Bezier element numbers.

ref_coors
[list of arrays] The reference coordinates in $[0, 1]$ for each knot span for each axis, defining the reference coordinates in the Bezier elements given by *indices*.

control_points
[array] The NURBS control points.

weights
[array] The NURBS weights.

degrees
[sequence of ints or int] The basis degrees in each parametric dimension.

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn

[array] The connectivity of the global NURBS basis.

Returns**out**

[array] The field variable values or NURBS geometry coordinates for the given reference coordinates.

```
sfepy.discrete.iga.extmods.igac.eval_mapping_data_in_qp(qps, control_points, weights, degrees, cs,
                                                         conn, cells=None)
```

Evaluate data required for the isogeometric domain reference mapping in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree.

Parameters**qps**

[array] The quadrature points coordinates with components in [0, 1] reference element domain.

control_points

[array] The NURBS control points.

weights

[array] The NURBS weights.

degrees

[sequence of ints or int] The basis degrees in each parametric dimension.

cs

[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn

[array] The connectivity of the global NURBS basis.

cells

[array, optional] If given, use only the given Bezier elements.

Returns**bfs**

[array] The NURBS shape functions in the physical quadrature points of all elements.

bfgs

[array] The NURBS shape functions derivatives w.r.t. the physical coordinates in the physical quadrature points of all elements.

dets

[array] The Jacobians of the mapping to the unit reference element in the physical quadrature points of all elements.

```
sfepy.discrete.iga.extmods.igac.eval_variable_in_qp(variable, qps, control_points, weights, degrees,
                                                     cs, conn, cells=None)
```

Evaluate a field variable in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree. The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters**variable**[array] The DOF values of the variable with `n_c` components, shape `(:, n_c)`.

qps
[array] The quadrature points coordinates with components in [0, 1] reference element domain.

control_points
[array] The NURBS control points.

weights
[array] The NURBS weights.

degrees
[sequence of ints or int] The basis degrees in each parametric dimension.

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
[array] The connectivity of the global NURBS basis.

cells
[array, optional] If given, use only the given Bezier elements.

Returns

coors
[array] The physical coordinates of the quadrature points of all elements.

vals
[array] The field variable values in the physical quadrature points.

dets
[array] The Jacobians of the mapping to the unit reference element in the physical quadrature points.

`sfepy.discrete.iga.extmods.igac.is_nurbs(weights)`

Return True if some weights are not one.

sfepy.discrete.iga.fields module

Fields for isogeometric analysis.

class `sfepy.discrete.iga.fields.IGField(name, dtype, shape, region, approx_order=None, **kwargs)`

Bezier extraction based NURBS approximation for isogeometric analysis.

Notes

The field has to cover the whole IGA domain. The field's NURBS basis can have higher degree than the domain NURBS basis.

create_basis_context()

Create the context required for evaluating the field basis.

create_eval_mesh()

Create a mesh with the original NURBS connectivity for evaluating the field. The mesh coordinates are the NURBS control points.

create_mapping(region, integral, integration)

Create a new reference mapping.

create_mesh(*extra_nodes=True*)

Create a mesh corresponding to the field region. For IGA fields, this is directly the topological mesh. The *extra_nodes* argument is ignored.

create_output(*dofs*, *var_name*, *dof_names=None*, *key=None*, ***kwargs*)

Convert the DOFs corresponding to the field to a dictionary of output data usable by `Mesh.write()`.

Parameters

dofs

[array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

var_name

[str] The variable name corresponding to *dofs*.

dof_names

[tuple of str] The names of DOF components.

key

[str, optional] The key to be used in the output dictionary instead of the variable name.

Returns

out

[dict] The output dictionary.

family_name = 'volume_H1_iga'

get_data_shape(*integral*, *integration='cell'*, *region_name=None*)

Get element data dimensions.

Parameters

integral

[Integral instance] The integral describing used numerical quadrature.

integration

['cell'] The term integration type. Only 'cell' type is implemented.

region_name

[str] The name of the region of the integral.

Returns

data_shape

[4 ints] The (*n_el*, *n_qp*, *dim*, *n_en*) for volume shape kind.

Notes

- *n_el* = number of elements
- *n_qp* = number of quadrature points per element/facet
- *dim* = spatial dimension
- *n_en* = number of element nodes

get_dofs_in_region(*region*, *merge=True*)

Return indices of DOFs that belong to the given region and group.

Notes

merge is not used.

get_econn(*conn_type*, *region*, *trace_region=None*, *local=False*)

Get DOF connectivity of the given type in the given region.

get_surface_basis(*region*)

get_true_order()

is_higher_order()

Return True, if the field's approximation order is greater than one.

setup_extra_data(*info*)

`sfepy.discrete.iga.fields.parse_approx_order`(*approx_order*)

sfepy.discrete.iga.iga module

Isogeometric analysis utilities.

Notes

The functions `compute_bezier_extraction_1d()` and `eval_nurbs_basis_tp()` implement the algorithms described in [1].

[1] **Michael J. Borden, Michael A. Scott, John A. Evans, Thomas J. R. Hughes:**

Isogeometric finite element data structures based on Bezier extraction of NURBS, Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, Texas, March 2010.

`sfepy.discrete.iga.iga.combine_bezier_extraction`(*cs*)

For a nD B-spline parametric domain, combine the 1D element extraction operators in each parametric dimension into a single operator for each nD element.

Parameters

cs

[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

Returns

ccs

[list of 2D arrays] The combined element extraction operators.

`sfepy.discrete.iga.iga.compute_bezier_control`(*control_points*, *weights*, *ccs*, *conn*, *bconn*)

Compute the control points and weights of the Bezier mesh.

Parameters

control_points

[array] The NURBS control points.

weights

[array] The NURBS weights.

ccs

[list of 2D arrays] The combined element extraction operators.

conn
[array] The connectivity of the global NURBS basis.

bconn
[array] The connectivity of the Bezier basis.

Returns

bezier_control_points
[array] The control points of the Bezier mesh.

bezier_weights
[array] The weights of the Bezier mesh.

`sfepy.discrete.iga.iga.compute_bezier_extraction(knots, degrees)`

Compute local (element) Bezier extraction operators for a nD B-spline parametric domain.

Parameters

knots
[sequence of array or array] The knot vectors.

degrees
[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

`sfepy.discrete.iga.iga.compute_bezier_extraction_1d(knots, degree)`

Compute local (element) Bezier extraction operators for a 1D B-spline parametric domain.

Parameters

knots
[array] The knot vector.

degree
[int] The curve degree.

Returns

cs
[array of 2D arrays (3D array)] The element extraction operators.

`sfepy.discrete.iga.iga.create_boundary_qp(coors, dim)`

Create boundary quadrature points from the surface quadrature points.

Uses the Bezier element tensor product structure.

Parameters

coors
[array, shape (n_qp, d)] The coordinates of the surface quadrature points.

dim
[int] The topological dimension.

Returns

bcoors
[array, shape (n_qp, d + 1)] The coordinates of the boundary quadrature points.

`sfepy.discrete.iga.iga.create_connectivity(n_els, knots, degrees)`

Create connectivity arrays of nD Bezier elements.

Parameters

n_els

[sequence of ints] The number of elements in each parametric dimension.

knots

[sequence of array or array] The knot vectors.

degrees

[sequence of ints or int] The basis degrees in each parametric dimension.

Returns

conn

[array] The connectivity of the global NURBS basis.

bconn

[array] The connectivity of the Bezier basis.

`sfepy.discrete.iga.iga.create_connectivity_1d(n_el, knots, degree)`

Create connectivity arrays of 1D Bezier elements.

Parameters

n_el

[int] The number of elements.

knots

[array] The knot vector.

degree

[int] The basis degree.

Returns

conn

[array] The connectivity of the global NURBS basis.

bconn

[array] The connectivity of the Bezier basis.

`sfepy.discrete.iga.iga.eval_bernstein_basis(x, degree)`

Evaluate the Bernstein polynomial basis of the given *degree*, and its derivatives, in a point *x* in [0, 1].

Parameters

x

[float] The point in [0, 1].

degree

[int] The basis degree.

Returns

funs

[array] The *degree* + 1 values of the Bernstein polynomial basis.

ders

[array] The *degree* + 1 values of the Bernstein polynomial basis derivatives.

```
sfepy.discrete.iga.iga.eval_mapping_data_in_qp(qps, control_points, weights, degrees, cs, conn,  
                                              cells=None)
```

Evaluate data required for the isogeometric domain reference mapping in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree.

Parameters**qps**

[array] The quadrature points coordinates with components in [0, 1] reference element domain.

control_points

[array] The NURBS control points.

weights

[array] The NURBS weights.

degrees

[sequence of ints or int] The basis degrees in each parametric dimension.

cs

[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn

[array] The connectivity of the global NURBS basis.

cells

[array, optional] If given, use only the given Bezier elements.

Returns**bfs**

[array] The NURBS shape functions in the physical quadrature points of all elements.

bfgs

[array] The NURBS shape functions derivatives w.r.t. the physical coordinates in the physical quadrature points of all elements.

dets

[array] The Jacobians of the mapping to the unit reference element in the physical quadrature points of all elements.

```
sfepy.discrete.iga.iga.eval_nurbs_basis_tp(qp, ie, control_points, weights, degrees, cs, conn)
```

Evaluate the tensor-product NURBS shape functions in a quadrature point for a given Bezier element.

Parameters**qp**

[array] The quadrature point coordinates with components in [0, 1] reference element domain.

ie

[int] The Bezier element index.

control_points

[array] The NURBS control points.

weights

[array] The NURBS weights.

degrees

[sequence of ints or int] The basis degrees in each parametric dimension.

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
[array] The connectivity of the global NURBS basis.

Returns

R
[array] The NURBS shape functions.

dR_dx
[array] The NURBS shape functions derivatives w.r.t. the physical coordinates.

det
[array] The Jacobian of the mapping to the unit reference element.

`sfepy.discrete.iga.iga.eval_variable_in_qp(variable, qps, control_points, weights, degrees, cs, conn, cells=None)`

Evaluate a field variable in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree. The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters

variable
[array] The DOF values of the variable with `n_c` components, shape `(:, n_c)`.

qps
[array] The quadrature points coordinates with components in `[0, 1]` reference element domain.

control_points
[array] The NURBS control points.

weights
[array] The NURBS weights.

degrees
[sequence of ints or int] The basis degrees in each parametric dimension.

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
[array] The connectivity of the global NURBS basis.

cells
[array, optional] If given, use only the given Bezier elements.

Returns

coors
[array] The physical coordinates of the quadrature points of all elements.

vals
[array] The field variable values in the physical quadrature points.

dets
[array] The Jacobians of the mapping to the unit reference element in the physical quadrature points.

`sfepy.discrete.iga.iga.get_bezier_element_entities(degrees)`

Get faces and edges of a Bezier mesh element in terms of indices into the element's connectivity (reference Bezier element entities).

Parameters

degrees

[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

faces

[list of arrays] The indices for each face or None if not 3D.

edges

[list of arrays] The indices for each edge or None if not at least 2D.

vertices

[list of arrays] The indices for each vertex.

Notes

The ordering of faces and edges has to be the same as in `sfepy.discrete.fem.geometry_element.geometry_data`.

`sfepy.discrete.iga.iga.get_bezier_topology(bconn, degrees)`

Get a topology connectivity corresponding to the Bezier mesh connectivity.

In the referenced Bezier control points the Bezier mesh is interpolatory.

Parameters

bconn

[array] The connectivity of the Bezier basis.

degrees

[sequence of ints or int] The basis degrees in each parametric dimension.

Returns

tconn

[array] The topology connectivity (corner nodes, or vertices, of Bezier elements) with vertex ordering suitable for a FE mesh.

`sfepy.discrete.iga.iga.get_facet_axes(dim)`

For each reference Bezier element facet return the facet axes followed by the remaining (perpendicular) axis, as well as the remaining axis coordinate of the facet.

Parameters

dim

[int] The topological dimension.

Returns

axes

[array] The axes of the reference element facets.

coors

[array] The remaining coordinate of the reference element facets.

`sfepy.discrete.iga.iga.get_patch_box_regions(n_els, degrees)`

Get box regions of Bezier topological mesh in terms of element corner vertices of Bezier mesh.

Parameters

n_els

[sequence of ints] The number of elements in each parametric dimension.

degrees

[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

regions

[dict] The Bezier mesh vertices of box regions.

`sfepy.discrete.iga.iga.get_raveled_index(indices, shape)`

Get a global raveled index corresponding to nD indices into an array of the given shape.

`sfepy.discrete.iga.iga.get_surface_degrees(degrees)`

Get degrees of the NURBS patch surfaces.

Parameters

degrees

[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

sdegrees

[list of arrays] The degrees of the patch surfaces, in the order of the reference Bezier element facets.

`sfepy.discrete.iga.iga.get_unraveled_indices(index, shape)`

Get nD indices into an array of the given shape corresponding to a global raveled index.

`sfepy.discrete.iga.iga.tensor_product(a, b)`

Compute tensor product of two 2D arrays with possibly different shapes. The result has the form:

```
c = [[a00 b, a01 b, ...],
     [a10 b, a11 b, ...],
     ...
     ...]
```

sfepy.discrete.iga.io module

IO for NURBS and Bezier extraction data.

`sfepy.discrete.iga.io.read_iga_data(filename, group=None)`

Read IGA-related data from a HDF5 file using pytables.

filename: str or tables.File

File to read the hdf5 mesh to.

group: tables.group.Group or None

HDF5 file group to read the mesh from. If it's None, the root of file is used.

Returns

tuple

Data for restoring IGA domain.

`sfepy.discrete.iga.io.write_iga_data(filename, group, knots, degrees, control_points, weights, cs, conn, bezier_control_points, bezier_weights, bezier_conn, regions, name=None)`

Write IGA-related data into a HDF5 file using pytables.

filename: str or tables.File

File to read the hdf5 mesh to.

group: tables.group.Group, optional

HDF5 file group to read the data from. If None, the root of file is used.

Returns

tuple

Data for restoring IGA domain.

sfepy.discrete.iga.mappings module

Reference mappings for isogeometric analysis.

class `sfepy.discrete.iga.mappings.IGMapping(domain, cells, nurbs=None)`

Reference mapping for isogeometric analysis based on Bezier extraction.

Parameters

domain

[IGDomain instance] The mapping domain.

cells

[array] The mapping region cells. (All domain cells required.)

nurbs

[NurbsPatch instance, optional] If given, the *nurbs* is used instead of *domain.nurbs*. The *nurbs* has to be obtained by degree elevation of *domain.nurbs*.

get_geometry()

Return reference element geometry as a GeometryElement instance.

get_mapping(qp_coors, weights)

Get the mapping for given quadrature points and weights.

Returns

cmap

[CMapping instance] The reference mapping.

Notes

Does not set total volume of the C mapping structure!

get_physical_qps(*qp_coors*)

Get physical quadrature points corresponding to given reference Bezier element quadrature points.

Returns

qps

[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

sfepy.discrete.iga.plot_nurbs module

sfepy.discrete.iga.plot_nurbs.plot_bezier_mesh(*ax, control_points, conn, degrees, label=False*)

Plot the Bezier mesh of a NURBS given by its control points and connectivity.

sfepy.discrete.iga.plot_nurbs.plot_bezier_nurbs_basis_1d(*ax, control_points, weights, degrees, cs, conn, n_points=20*)

Plot a 1D NURBS basis using the Bezier extraction and local Bernstein basis.

sfepy.discrete.iga.plot_nurbs.plot_control_mesh(*ax, control_points, label=False*)

Plot the control mesh of a NURBS given by its control points.

sfepy.discrete.iga.plot_nurbs.plot_iso_lines(*ax, nurbs, color='b', n_points=100*)

Plot the NURBS object using iso-lines in Greville abscissae coordinates.

sfepy.discrete.iga.plot_nurbs.plot_nurbs_basis_1d(*ax, nurbs, n_points=100, x_axis='parametric', legend=False*)

Plot a 1D NURBS basis.

sfepy.discrete.iga.plot_nurbs.plot_parametric_mesh(*ax, knots*)

Plot the parametric mesh of a NURBS given by its knots.

sfepy.discrete.iga.utils module

Utility functions based on igakit.

sfepy.discrete.iga.utils.create_linear_fe_mesh(*nurbs, pars=None*)

Convert a NURBS object into a nD-linear tensor product FE mesh.

Parameters

nurbs

[igakit.nurbs.NURBS instance] The NURBS object.

pars

[sequence of array, optional] The values of parameters in each parametric dimension. If not given, the values are set so that the resulting mesh has the same number of vertices as the number of control points/basis functions of the NURBS object.

Returns

coors

[array] The coordinates of mesh vertices.

conn

[array] The vertex connectivity array.

desc

[str] The cell kind.

`sfepy.discrete.iga.utils.create_mesh_and_output(nurbs, pars=None, **kwargs)`

Create a nD-linear tensor product FE mesh using `create_linear_fe_mesh()`, evaluate field variables given as keyword arguments in the mesh vertices and create a dictionary of output data usable by `Mesh.write()`.

Parameters**nurbs**

[igakit.nurbs.NURBS instance] The NURBS object.

pars

[sequence of array, optional] The values of parameters in each parametric dimension. If not given, the values are set so that the resulting mesh has the same number of vertices as the number of control points/basis functions of the NURBS object.

****kwargs**

[kwargs] The field variables as keyword arguments. Their names serve as keys in the output dictionary.

Returns**mesh**

[Mesh instance] The finite element mesh.

out

[dict] The output dictionary.

`sfepy.discrete.iga.utils.save_basis(nurbs, pars)`

Save a NURBS object basis on a FE mesh corresponding to the given parametrization in VTK files.

Parameters**nurbs**

[igakit.nurbs.NURBS instance] The NURBS object.

pars

[sequence of array, optional] The values of parameters in each parametric dimension.

sfepy.discrete.structural sub-package**sfepy.discrete.structural.fields module**

Fields corresponding to structural elements.

class `sfepy.discrete.structural.fields.Shell10XField(name, dtype, shape, region, approx_order=1)`

The approximation for the shell10x element.

create_mapping(*region, integral, integration, return_mapping=True*)

Create a new reference mapping.

create_output(*dofs, var_name, dof_names=None, key=None, thickness=None, **kwargs*)

Convert the DOFs corresponding to the field to a dictionary of output data usable by `Mesh.write()`.

Parameters

dofs

[array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

var_name

[str] The variable name corresponding to *dofs*.

dof_names

[tuple of str] The names of DOF components.

key

[str, optional] The key to be used in the output dictionary instead of the variable name.

Returns**out**

[dict] The output dictionary.

```
family_name = 'volume_H1_shell10x'
```

sfepy.discrete.structural.mappings module

Finite element reference mappings for structural elements.

```
class sfepy.discrete.structural.mappings.Shell10XMapping(region, field)
```

The reference mapping for the shell10x element.

```
get_mapping(qp_coors, weights)
```

Get the mapping for given quadrature points and weights.

```
get_physical_qps(qp_coors)
```

Get physical quadrature points corresponding the given reference element quadrature points.

Returns**qps**

[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

sfepy.homogenization package

sfepy.homogenization.band_gaps_app module

```
class sfepy.homogenization.band_gaps_app.AcousticBandGapsApp(conf, options, output_prefix,  
                                                             **kwargs)
```

Application for computing acoustic band gaps.

```
call()
```

Construct and call the homogenization engine according to options.

```
plot_band_gaps(coefs)
```

```
plot_dispersion(coefs)
```

```
static process_options(options)
```

Application options setup. Sets default values for missing non-compulsory options.

static process_options_pv(*options*)

Application options setup for phase velocity computation. Sets default values for missing non-compulsory options.

setup_options()

sfepy.homogenization.band_gaps_app.plot_eigs(*ax, plot_rsc, plot_labels, valid, freq_range, plot_range*)

Plot resonance/eigen-frequencies.

valid must correspond to *freq_range*

resonances : red masked resonances: dotted red

sfepy.homogenization.band_gaps_app.plot_gap(*ax, ranges, kind, kind_desc, plot_range, plot_rsc*)

Plot single band gap frequency ranges as rectangles.

sfepy.homogenization.band_gaps_app.plot_gaps(*ax, plot_rsc, gaps, kinds, gap_ranges, freq_range, plot_range*)

Plot band gaps as rectangles.

sfepy.homogenization.band_gaps_app.plot_logs(*ax, plot_rsc, plot_labels, freqs, logs, valid, freq_range, plot_range, draw_eigs=True, show_legend=True*)

Plot logs of min/middle/max eigs of a mass matrix.

sfepy.homogenization.band_gaps_app.save_raw_bg_logs(*filename, logs*)

Save raw band gaps *logs* into the *filename* file.

sfepy.homogenization.band_gaps_app.transform_plot_data(*datas, plot_transform, conf*)

sfepy.homogenization.band_gaps_app.try_set_defaults(*obj, attr, defaults, recur=False*)

sfepy.homogenization.coefficients module

class sfepy.homogenization.coefficients.Coefficients(***kwargs*)

Class for storing (homogenized) material coefficients.

static from_file_hdf5(*filename*)

to_file_hdf5(*filename*)

to_file_latex(*filename, names, format='%e', cdot=False, filter=None, idx=None*)

Save the coefficients to a file in LaTeX format.

Parameters

filename

[str] The name of the output file.

names

[dict] Mapping of attribute names to LaTeX names.

format

[str] Format string for numbers.

cdot

[bool] For ‘%.e’ formats only. If True, replace ‘e’ by LaTeX ‘ $\cdot 10^{\text{exponent}}$ ’ format.

filter

[int] For ‘%.e’ formats only. Typeset as 0, if exponent is less than *filter*.

idx

[int] For multi-coefficients, set the coefficient index.

to_file_txt(*filename, names, format*)

to_latex(*attr_name, dim, style='table', format='%f', step=None*)

`sfePy.homogenization.coefficients.coef_arrays_to_dicts(idict, format='%s/%d')`

sfePy.homogenization.coefs_base module

class `sfePy.homogenization.coefs_base.CoeffDim`(*name, problem, kwargs*)

class `sfePy.homogenization.coefs_base.CoeffDimDim`(*name, problem, kwargs*)

class `sfePy.homogenization.coefs_base.CoeffDimSym`(*name, problem, kwargs*)

class `sfePy.homogenization.coefs_base.CoeffDummy`(*name, problem, kwargs*)

Dummy class serving for computing and returning its requirements.

class `sfePy.homogenization.coefs_base.CoeffEval`(*name, problem, kwargs*)

Evaluate expression.

class `sfePy.homogenization.coefs_base.CoeffExprPar`(*name, problem, kwargs*)

The coefficient which expression can be parametrized via 'expr_pars', the dimension is given by the number of parameters.

Example:

```
'expression': 'dw_surface_ndot.5.Ys(mat_norm.k%d, corr1)', 'expr_pars': [ii for ii in range(dim)],  
'class': cb.CoeffExprPar,
```

static **set_variables_default**(*variables, ir, set_var, data*)

class `sfePy.homogenization.coefs_base.CoeffMN`(*name, problem, kwargs*)

get_coef(*row, col, volume, problem, data*)

static **set_variables_default**(*variables, ir, ic, mode, set_var, data, dtype*)

class `sfePy.homogenization.coefs_base.CoeffN`(*name, problem, kwargs*)

get_coef(*row, volume, problem, data*)

static **set_variables_default**(*variables, ir, ic, mode, set_var, data, dtype*)

class `sfePy.homogenization.coefs_base.CoeffNonSym`(*name, problem, kwargs*)

is_sym = False

static **iter_sym**(*dim*)

class `sfePy.homogenization.coefs_base.CoeffNonSymNonSym`(*name, problem, kwargs*)

is_sym = False

static **iter_sym**(*dim*)

class `sfePy.homogenization.coefs_base.CoeffNone`(*name, problem, kwargs*)


```

class sfepy.homogenization.coefs_base.CoeffOne(name, problem, kwargs)
    static set_variables_default(variables, set_var, data, dtype)
class sfepy.homogenization.coefs_base.CoeffSum(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffSym(name, problem, kwargs)
    is_sym = True
    static iter_sym(dim)
class sfepy.homogenization.coefs_base.CoeffSymDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffSymSym(name, problem, kwargs)
    is_sym = True
    static iter_sym(dim)
class sfepy.homogenization.coefs_base.CorrDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CorrDimDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CorrEqPar(name, problem, kwargs)
    The corrector which equation can be parametrized via 'eq_pars', the dimension is given by the number of parameters.
    Example:
        'equations': 'dw_diffusion.5.Y(mat.k, q, p) =
                    dw_integrate.5.%s(q)',
        'eq_pars': ('bYMp', 'bYMm'), 'class': cb.CorrEqPar,
class sfepy.homogenization.coefs_base.CorrEval(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CorrMiniApp(name, problem, kwargs)
    get_output(corr_sol, is_dump=False, extend=True, variables=None, var_map=None)
    get_save_name(save_format='h5', stamp='')
    get_save_name_base()
    save(state, problem, variables=None, ts=None, var_map=None)
    setup_output(save_formats=None, post_process_hook=None, split_results_by=None)
        Instance attributes have precedence!
class sfepy.homogenization.coefs_base.CorrN(name, problem, kwargs)
    static set_variables_default(variables, ir, set_var, data)
class sfepy.homogenization.coefs_base.CorrNN(name, problem, kwargs)
    __init__() kwargs: {
        'ebcs' : [], 'epbcs' : [], 'equations' : {}, 'set_variables' : None,
    },
    static set_variables_default(variables, ir, ic, set_var, data)

```

```
class sfepy.homogenization.coefs_base.CorrOne(name, problem, kwargs)
    static set_variables_default(variables, set_var, data)

class sfepy.homogenization.coefs_base.CorrSetBCS(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CorrSolution(**kwargs)
    Class for holding solutions of corrector problems.
    get_output(is_dump=False, var_map=None)
    get_ts_val(step)
    iter_solutions()
    iter_time_steps()

class sfepy.homogenization.coefs_base.MinAppBase(name, problem, kwargs)
    static any_from_conf(name, problem, kwargs)
    init_solvers(problem)
        Setup solvers. Use local options if these are defined, otherwise use the global ones.
        For linear problems, assemble the matrix and try to presolve the linear system.
    process_options()
        Setup application-specific options.
        Subclasses should implement this method as needed.
    Returns
        app_options
            [Struct instance] The application options.

class sfepy.homogenization.coefs_base.OnesDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.ShapeDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.ShapeDimDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.TSTimes(name, problem, kwargs)
    Coefficient-like class, returns times of the time stepper.

class sfepy.homogenization.coefs_base.VolumeFractions(name, problem, kwargs)
    Coefficient-like class, returns volume fractions of given regions within the whole domain.
```

sfepy.homogenization.coefs_phononic module

```
class sfepy.homogenization.coefs_phononic.AcousticMassLiquidTensor(name, problem, kwargs)
    get_coefs(freq)
        Get frequency-dependent coefficients.
```

class sfepy.homogenization.coefs_phononic.**AcousticMassTensor**(*name, problem, kwargs*)

The acoustic mass tensor for a given frequency.

Returns

self

[AcousticMassTensor instance] This class instance whose *evaluate()* method computes for a given frequency the required tensor.

Notes

eigenmomenta, *eigs* should contain only valid resonances.

evaluate(*freq*)

get_coefs(*freq*)

Get frequency-dependent coefficients.

to_file_txt = None

class sfepy.homogenization.coefs_phononic.**AppliedLoadTensor**(*name, problem, kwargs*)

The applied load tensor for a given frequency.

Returns

self

[AppliedLoadTensor instance] This class instance whose *evaluate()* method computes for a given frequency the required tensor.

Notes

eigenmomenta, *ueigenmomenta*, *eigs* should contain only valid resonances.

evaluate(*freq*)

to_file_txt = None

class sfepy.homogenization.coefs_phononic.**BandGaps**(*name, problem, kwargs*)

Band gaps detection.

Parameters

eigensolver

[str] The name of the eigensolver for mass matrix eigenvalues.

eig_range

[(int, int)] The eigenvalues range (squared frequency) to consider.

freq_margins

[(float, float)] Margins in percents of initial frequency range given by *eig_range* by which the range is increased.

fixed_freq_range

[(float, float)] The frequency range to consider. Has precedence over *eig_range* and *freq_margins*.

freq_step

[float] The frequency step for tracing, in percent of the frequency range.

freq_eps

[float] The frequency difference smaller than *freq_eps* is considered zero.

zero_eps

[float] The tolerance for finding zeros of mass matrix eigenvalues.

detect_fun

[callable] The function for detecting the band gaps. Default is *detect_band_gaps()*.

log_save_name

[str] If not None, the band gaps log is to be saved under the given name.

raw_log_save_name

[str] If not None, the raw band gaps log is to be saved under the given name.

fix_eig_range(*n_eigs*)

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns

app_options

[Struct instance] The application options.

static save_log(*filename, float_format, bg*)

Save band gaps, valid flags and eigenfrequencies.

static to_file_txt(*fd, float_format, bg*)

class sfepy.homogenization.coefs_phononic.**ChristoffelAcousticTensor**(*name, problem, kwargs*)

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns

app_options

[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.**DensityVolumeInfo**(*name, problem, kwargs*)

Determine densities of regions specified in *region_to_material*, and compute average density based on region volumes.

static to_file_txt(*fd, float_format, dv_info*)

class sfepy.homogenization.coefs_phononic.**Eigenmomenta**(*name, problem, kwargs*)

Eigenmomenta corresponding to eigenvectors.

Parameters

var_name

[str] The name of the variable used in the integral.

threshold

[float] The threshold under which an eigenmomentum is considered zero.

threshold_is_relative

[bool] If True, the *threshold* is relative w.r.t. max. norm of eigenmomenta.

transform

[callable, optional] Optional function for transforming the eigenvectors before computing the eigenmomenta.

Returns**eigenmomenta**

[Struct] The resulting eigenmomenta. An eigenmomentum above threshold is marked by the attribute 'valid' set to True.

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns**app_options**

[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.**PhaseVelocity**(*name, problem, kwargs*)

Compute phase velocity.

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns**app_options**

[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.**PolarizationAngles**(*name, problem, kwargs*)

Compute polarization angles, i.e., angles between incident wave direction and wave vectors. Vector length does not matter - eigenvectors are used directly.

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns**app_options**

[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.**SchurEVP**(*name, problem, kwargs*)

Schur complement eigenvalue problem.

post_process(*eigs, mtx_s_phi, mtx_dib, problem*)

prepare_matrices(*problem*)

$A = K + B^T D^{-1} B$

class sfepy.homogenization.coefs_phononic.**SimpleEVP**(*name, problem, kwargs*)

Simple eigenvalue problem.

post_process(*eigs, mtx_s_phi, data, problem*)

prepare_matrices(*problem*)

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns**app_options**

[Struct instance] The application options.

save(eigs, mtx_phi, problem)

`sfepy.homogenization.coefs_phononic.compute_cat_dim_dim(coef, iw_dir)`

Christoffel acoustic tensor part of dielectric tensor dimension.

`sfepy.homogenization.coefs_phononic.compute_cat_dim_sym(coef, iw_dir)`

Christoffel acoustic tensor part of piezo-coupling tensor dimension.

`sfepy.homogenization.coefs_phononic.compute_cat_sym_sym(coef, iw_dir)`

Christoffel acoustic tensor (part) of elasticity tensor dimension.

`sfepy.homogenization.coefs_phononic.compute_eigenmomenta(em_equation, var_name, problem, eig_vectors, transform=None)`

Compute the eigenmomenta corresponding to given eigenvectors.

`sfepy.homogenization.coefs_phononic.cut_freq_range(freq_range, eigs, valid, freq_margins, eig_range, fixed_freq_range, freq_eps)`

Cut off masked resonance frequencies. Margins are preserved, like no resonances were cut.

Returns**freq_range**

[array] The new range of frequencies.

freq_range_margins

[array] The range of frequencies with prepended/appended margins equal to *fixed_freq_range* if it is not None.

`sfepy.homogenization.coefs_phononic.describe_gaps(gaps)`

`sfepy.homogenization.coefs_phononic.detect_band_gaps(mass, freq_info, opts, gap_kind='normal', mtx_b=None)`

Detect band gaps given solution to eigenproblem (eigs, eig_vectors). Only valid resonance frequencies (e.i. those for which corresponding eigenmomenta are above a given threshold) are taken into account.

Notes

- make `freq_eps` relative to `]f0, f1[` size?

`sfepy.homogenization.coefs_phononic.find_zero(f0, f1, callback, freq_eps, zero_eps, mode)`

For `f` in `]f0, f1[` find frequency `f` for which either the smallest (`mode = 0`) or the largest (`mode = 1`) eigenvalue of problem `P` given by `callback` is zero.

Returns**flag**

[0, 1, or 2] The flag, see Notes below.

frequency

[float] The found frequency.

eigenvalue

[float] The eigenvalue corresponding to the found frequency.

Notes

Meaning of the return value combinations:

mode	flag	meaning
0, 1	0	eigenvalue $\rightarrow 0$ for f in $]f_0, f_1[$
0	1	$f \rightarrow f_1$, smallest eigenvalue < 0
0	2	$f \rightarrow f_0$, smallest eigenvalue > 0 and $\rightarrow -\infty$
1	1	$f \rightarrow f_1$, largest eigenvalue < 0 and $\rightarrow +\infty$
1	2	$f \rightarrow f_0$, largest eigenvalue > 0

`sfepy.homogenization.coefs_phononic.get_callback(mass, solver_kind, mtx_b=None, mode='trace')`

Return callback to solve band gaps or dispersion eigenproblem P.

Notes**Find zero callbacks return:**

eigenvalues

Trace callbacks return:

(eigenvalues,)

or

(eigenvalues, eigenvectors) (in full (dispoersion) mode)

If `mtx_b` is `None`, the problem P is $M w = \lambda w$,**otherwise it is** $\omega^2 M w = \eta B w$

`sfepy.homogenization.coefs_phononic.get_gap_ranges(freq_range, gaps, kinds)`

For each (potential) band gap in *gaps*, return the frequency ranges of its parts according to *kinds*.

`sfepy.homogenization.coefs_phononic.get_log_freqs(f0, f1, df, freq_eps, n_point_min, n_point_max)`

Get logging frequencies.

The frequencies get denser towards the interval boundaries.

`sfepy.homogenization.coefs_phononic.get_ranges(freq_range, eigs)`

Get an eigenvalue range slice and a corresponding initial frequency range within a given frequency range.

`sfepy.homogenization.coefs_phononic.split_chunks(indx)`

Split index vector to chunks of consecutive numbers.

sfepy.homogenization.convolution module

```
class sfepy.homogenization.convolution.ConvolutionKernel(name, times, kernel, decay=None,  
                                                         exp_coefs=None, exp_decay=None)
```

The convolution kernel with exponential synchronous decay approximation approximating the original kernel represented by the array $c[i]$, $i = 0, 1, \dots$

$$c_0 \equiv c[0], c_{e0} \equiv c_0 c_0^e,$$
$$c(t) \approx c_0 d(t) \approx c_0 e(t) = c_{e0} e_n(t),$$

where $d(0) = e_n(0) = 1$, d is the synchronous decay and e its exponential approximation, $e = c_0^e \exp(-c_1^e t)$.

```
diff_dt(use_exp=False)
```

The derivative of the kernel w.r.t. time.

```
get_exp()
```

Get the exponential synchronous decay kernel approximation.

```
get_full()
```

Get the original (full) kernel.

```
int_dt(use_exp=False)
```

The integral of the kernel in time.

```
sfepy.homogenization.convolution.approximate_exponential(x, y)
```

Approximate $y = f(x)$ by $y_a = c_1 \exp(-c_2 x)$.

Initial guess is given by assuming y has already the required exponential form.

```
sfepy.homogenization.convolution.compute_mean_decay(coef)
```

Compute mean decay approximation of a non-scalar fading memory coefficient.

```
sfepy.homogenization.convolution.eval_exponential(coefs, x)
```

```
sfepy.homogenization.convolution.fit_exponential(x, y, return_coefs=False)
```

Evaluate $y = f(x)$ after approximating f by an exponential.

sfepy.homogenization.engine module

```
class sfepy.homogenization.engine.CoefVolume(name, problem, kwargs)
```

```
class sfepy.homogenization.engine.HomogenizationEngine(problem, options, app_options=None,  
                                                         volumes=None, output_prefix='he:',  
                                                         **kwargs)
```

```
call(ret_all=False, time_tag="")
```

```
static define_volume_coef(coef_info, volumes)
```

Define volume coefficients and make all other dependent on them.

Parameters**coef_info**

[dict] The coefficient definitions.

volumes

[dict] The definitions of volumes.

Returns**coef_info**

[dict] The coefficient definitions extended by the volume coefficients.

static process_options(options)

Application options setup. Sets default values for missing non-compulsory options.

set_micro_states(states)**setup_options(app_options=None)****class sfepy.homogenization.engine.HomogenizationWorker****static calculate(mini_app, problem, dependencies, dep_requires, save_names, micro_states, chunk_tab, mode, proc_id)****static calculate_req(problem, opts, post_process_hook, name, req_info, coef_info, save_names, dependencies, micro_states, time_tag='', chunk_tab=None, proc_id=0')**

Calculate a requirement, i.e. correctors or coefficients.

Parameters**problem**

[problem] The problem definition related to the microstructure.

opts

[struct] The options of the homogenization application.

post_process_hook

[function] The postprocessing hook.

name

[str] The name of the requirement.

req_info

[dict] The definition of correctors.

coef_info

[dict] The definition of homogenized coefficients.

save_names

[dict] The dictionary containing names of saved correctors.

dependencies

[dict] The dependencies required by the correctors/coefficients.

micro_states

[array] The configurations of multiple microstructures.

time_tag

[str] The label corresponding to the actual time step and iteration, used in the corrector file names.

chunk_tab

[list] In the case of multiprocessing the requirements are divided into several chunks that are solved in parallel.

proc_id

[int] The id number of the processor (core) which is solving the actual chunk.

Returns

val

[coefficient/corrector or list of coefficients/correctors] The resulting homogenized coefficients or correctors.

static get_sorted_dependencies(*req_info, coef_info, compute_only*)

Make corrs and coefs list sorted according to the dependencies.

class sfepy.homogenization.engine.**HomogenizationWorkerMulti**(*num_workers*)

static calculate_req_multi(*tasks, lock, remaining, numdeps, inverse_deps, problem, opts, post_process_hook, req_info, coef_info, save_names, dependencies, micro_states, time_tag, chunk_tab, proc_id*)

Calculate a requirement in parallel.

Parameters

tasks

[queue] The queue of requirements to be solved.

lock

[lock] The multiprocessing lock used to ensure save access to the global variables.

remaining

[int] The number of remaining requirements.

numdeps

[dict] The number of dependencies for the each requirement.

inverse_deps

[dict] The inverse dependencies - which requirements depend on a given one.

For the definition of other parameters see ‘calculate_req’.

static chunk_micro_tasks(*num_workers, num_micro, reqs, coefs, chunks_per_worker=1, store_micro_idxes=[]*)

Split multiple microproblems into several chunks that can be processed in parallel.

Parameters

num_workers

[int] The number of available CPUs.

num_micro

[int] The number of microstructures.

reqs

[dict] The requirement definitions.

coefs

[dict] The coefficient definitions.

chunks_per_worker

[int] The number of chunks per one worker.

store_micro_idxes

[list of int] The indices of microstructures whose results are to be stored.

Returns

micro_tab

[list of slices] The indices of microproblems contained in each chunk.

new_reqs
[dict] The new requirement definitions.

new_coefs
[dict] The new coefficient definitions.

static dechunk_reqs_coefs(*deps, num_chunks*)
Merge the results related to the multiple microproblems.

Parameters

deps
[dict] The calculated dependencies.

num_chunks
[int] The number of chunks.

Returns

new_deps
[dict] The merged dependencies.

static process_reqs_coefs(*old, num_workers, store_idxes=[]*)

class sfepy.homogenization.engine.**HomogenizationWorkerMultiMPI**(*num_workers*)

sfepy.homogenization.engine.**get_dict_idxval**(*dict_array, idx*)

sfepy.homogenization.engine.**insert_sub_reqs**(*reqs, levels, req_info*)
Recursively build all requirements in correct order.

sfepy.homogenization.homogen_app module

class sfepy.homogenization.homogen_app.**HomogenizationApp**(*conf, options, output_prefix, **kwargs*)

call(*verbose=False, ret_all=None, itime=None, iter=None*)
Call the homogenization engine and compute the homogenized coefficients.

Parameters

verbose
[bool] If True, print the computed coefficients.

ret_all
[bool or None] If not None, it can be used to override the ‘return_all’ option. If True, also the dependencies are returned.

time_tag: str
The time tag used in file names.

Returns

coefs
[Coefficients instance] The homogenized coefficients.

dependencies
[dict] The dependencies, if *ret_all* is True.

get_micro_cache_key(*key, icoor, itime*)

static process_options(*options*)

Application options setup. Sets default values for missing non-compulsory options.

setup_macro_data(*data*)

Setup macroscopic deformation gradient.

setup_options()

update_micro_states()

Update microstructures state according to the macroscopic data and corrector functions.

sfepy.homogenization.micmac module

sfepy.homogenization.micmac.get_correctors_from_file_hdf5(*coefs_filename*='coefs.h5',
dump_names=None)

sfepy.homogenization.micmac.get_homog_coefs_linear(*ts*, *coor*, *mode*, *micro_filename*=None,
regenerate=False, *coefs_filename*=None,
define_args=None, *output_dir*=None)

sfepy.homogenization.micmac.get_homog_coefs_nonlinear(*ts*, *coor*, *mode*, *macro_data*=None,
term=None, *problem*=None, *iteration*=None,
define_args=None, *output_dir*=None,
ret_corrs=False, ***kwargs*)

sfepy.homogenization.recovery module

sfepy.homogenization.recovery.add_strain_rs(*corrs_rs*, *strain*, *vu*, *dim*, *iel*, *out*=None)

sfepy.homogenization.recovery.add_stress_p(*out*, *pb*, *integral*, *region*, *vp*, *data*)

sfepy.homogenization.recovery.combine_scalar_grad(*corrs*, *grad*, *vn*, *ii*, *shift_coors*=None)

$$\eta_k \partial_k^x p$$

or

$$(y_k + \eta_k) \partial_k^x p$$

sfepy.homogenization.recovery.compute_mac_stress_part(*pb*, *integral*, *region*, *material*, *vu*, *mac_strain*)

sfepy.homogenization.recovery.compute_micro_u(*corrs*, *strain*, *vu*, *dim*, *out*=None)

Micro displacements.

$$\mathbf{u}^1 = \chi^{ij} e_{ij}^x(\mathbf{u}^0)$$

sfepy.homogenization.recovery.compute_p_corr_steady(*corrs_pressure*, *pressure*, *vp*, *iel*)

$$\tilde{\pi}^P p$$

`sfepy.homogenization.recovery.compute_p_corr_time`(*corrs_rs, dstrains, corrs_pressure, pressures, vdp, dim, iel, ts*)

$$\sum_{ij} \int_0^t \frac{d}{dt} \tilde{\pi}^{ij}(t-s) \frac{d}{ds} e_{ij}(\mathbf{u}(s)) ds + \int_0^t \frac{d}{dt} \tilde{\pi}^P(t-s) p(s) ds$$

`sfepy.homogenization.recovery.compute_p_from_macro`(*p_grad, coor, iel, centre=None, extdim=0*)
Macro-induced pressure.

$$\partial_j^x p(y_j - y_j^c)$$

`sfepy.homogenization.recovery.compute_stress_strain_u`(*pb, integral, region, material, vu, data*)

`sfepy.homogenization.recovery.compute_u_corr_steady`(*corrs_rs, strain, vu, dim, iel*)

$$\sum_{ij} \omega^{ij} e_{ij}(\mathbf{u})$$

Notes

- iel = element number

`sfepy.homogenization.recovery.compute_u_corr_time`(*corrs_rs, dstrains, corrs_pressure, pressures, vu, dim, iel, ts*)

$$\sum_{ij} \left[\int_0^t \omega^{ij}(t-s) \frac{d}{ds} e_{ij}(\mathbf{u}(s)) ds \right] + \int_0^t \tilde{\omega}^P(t-s) p(s) ds$$

`sfepy.homogenization.recovery.compute_u_from_macro`(*strain, coor, iel, centre=None*)
Macro-induced displacements.

$$e_{ij}^x(\mathbf{u})(y_j - y_j^c)$$

`sfepy.homogenization.recovery.convolve_field_scalar`(*fvars, pvars, iel, ts*)

$$\int_0^t f(t-s)p(s)ds$$

Notes

- t is given by step
- f: fvars scalar field variables, defined in a micro domain, have shape [step][fmf dims]
- p: pvars scalar point variables, a scalar in a point of macro-domain, FMField style have shape [n_step][var dims]

`sfepy.homogenization.recovery.convolve_field_sym_tensor`(*fvars, pvars, var_name, dim, iel, ts*)

$$\int_0^t f^{ij}(t-s)p_{ij}(s)ds$$

Notes

- *t* is given by step
- *f*: fvars field variables, defined in a micro domain, have shape [step][fmf dims]
- *p*: pvars sym. tensor point variables, a scalar in a point of macro-domain, FMField style, have shape [dim, dim][var_name][n_step][var dims]

`sfepy.homogenization.recovery.destroy_pool()`

`sfepy.homogenization.recovery.get_output_suffix(iel, ts, naming_scheme, format, output_format)`

`sfepy.homogenization.recovery.get_recovery_points(region, eps0)`

`sfepy.homogenization.recovery.recover_bones(problem, micro_problem, region, eps0, ts, strain, dstrains, p_grad, pressures, corrs_permeability, corrs_rs, corrs_time_rs, corrs_pressure, corrs_time_pressure, var_names, naming_scheme='step_iel')`

Notes

- note that

$$\tilde{\pi}^P$$

is in corrs_pressure -> from time correctors only 'u', 'dp' are needed.

`sfepy.homogenization.recovery.recover_micro_hook(micro_filename, region, macro, eps0, region_mode='el_centers', eval_mode='constant', eval_vars=None, corrs=None, recovery_file_tag="", define_args=None, output_dir=None, verbose=False)`

Parameters

micro_filename

[str] The definition file of the microproblem.

region

[Region or array] The macroscopic region to be recovered. If array, the centers of microscopic RVEs (Representative Volume Element) are assumed to be stored in it. If Region, the RVE centers are computed according to *region_mode*, see below.

macro

[dict of arrays or tuples] Either macroscopic values (if array) or the tuple (mode, eval_var, nodal_values) is expected. The tuple is used to evaluate the macroscopic values in given points of RVEs (see 'eval_mode'). *mode* can be 'val', 'grad', 'div', or 'cauchy_strain'.

eps0

[float] The size of the microstructures (RVE).

region_mode

[{'el_centers', 'tiled'}] If 'el_centers', the RVE centers are identical to the element centers of the macroscopic FE mesh. If 'tiled', the recovered region is tiled by rescaled RVEs.

eval_mode

[{'constant', 'continuous'}] If 'constant', the macroscopic fields are evaluated only at the RVE centers. If 'continuous', the fields are evaluated at all points of the RVE mesh.

eval_vars

[list of variables] The list of variables use to evaluate the macroscopic fields.

corrs

[dict of CorrSolution] The correctors for recovery.

recovery_file_tag

[str] The tag which is appended to the output file.

define_args

[dict] The define arguments for the microscopic problem.

output_dir

[str] The output directory.

verbose

[bool] The verbose terminal output.

```
sfepy.homogenization.recovery.recover_paraflow(problem, micro_problem, region, ts, strain, dstrains,
                                                pressures1, pressures2, corrs_rs, corrs_time_rs,
                                                corrs_alpha1, corrs_time_alpha1, corrs_alpha2,
                                                corrs_time_alpha2, var_names,
                                                naming_scheme='step_iel')
```

```
sfepy.homogenization.recovery.save_recovery_region(mac_pb, rname, filename=None)
```

sfepy.homogenization.utils module

```
sfepy.homogenization.utils.build_op_pi(var, ir, ic)
```

$Pi_i^{rs} = y_s \delta_{ir}$ for $r = ir, s = ic$.

```
sfepy.homogenization.utils.coor_to_sym(ir, ic, dim)
```

```
sfepy.homogenization.utils.create_pis(problem, var_name)
```

$Pi_i^{rs} = y_s \delta_{ir}$, $ul\{y\}$ in Y coordinates.

```
sfepy.homogenization.utils.create_scalar_pis(problem, var_name)
```

$Pi^k = y_k$, $ul\{y\}$ in Y coordinates.

```
sfepy.homogenization.utils.define_box_regions(dim, lbn, rtf=None, eps=0.001, kind='facet')
```

Define sides and corner regions for a box aligned with coordinate axes.

Parameters**dim**

[int] Space dimension

lbn

[tuple] Left bottom near point coordinates if rtf is not None. If rtf is None, lbn are the (positive) distances from the origin.

rtf

[tuple] Right top far point coordinates.

eps

[float] A parameter, that should be smaller than the smallest mesh node distance.

kind

[bool, optional] The region kind.

Returns**regions**

[dict] The box regions.

`sfepy.homogenization.utils.get_box_volume(dim, lbn, rtf=None)`

Volume of a box aligned with coordinate axes.

Parameters:

dim

[int] Space dimension

lbn

[tuple] Left bottom near point coordinates if rtf is not None. If rtf is None, lbn are the (positive) distances from the origin.

rtf

[tuple] Right top far point coordinates.

Returns:

volume

[float] The box volume.

`sfepy.homogenization.utils.get_lattice_volume(axes)`

Volume of a periodic cell in a rectangular 3D (or 2D) lattice.

Parameters**axes**

[array] The array with the periodic cell axes $a_1, \dots, a_3$ as rows.

Returns**volume**

[float] The periodic cell volume $V = (a_1 \times a_2) \cdot a_3$. In 2D $V = |(a_1 \times a_2)|$ with zeros as the third components of vectors a_1, a_2 .

`sfepy.homogenization.utils.get_volume(problem, field_name, region_name, quad_order=1)`

Get volume of a given region using integration defined by a given field. Both the region and the field have to be defined in *problem*.

`sfepy.homogenization.utils.integrate_in_time(coef, ts, scheme='forward')`

Forward difference or trapezoidal rule. 'ts' can be anything with 'times' attribute.

`sfepy.homogenization.utils.interp_conv_mat(mat, ts, tdiff)`

`sfepy.homogenization.utils.iter_nonsym(dim)`

`sfepy.homogenization.utils.iter_sym(dim)`

`sfepy.homogenization.utils.rm_multi(s)`

`sfepy.homogenization.utils.set_nonlin_states(variables, nl_state, problem)`

Setup reference state for nonlinear homogenization

Parameters**variables**

[dict] All problem variables

nl_state
[reference state]

problem
[problem description]

sfepy.linalg package

sfepy.linalg.check_derivatives module

Utilities for checking derivatives of functions.

`sfepy.linalg.check_derivatives.check_fx(x0, fx, fx_args, dfx, dfx_args=None, delta=1e-05)`

Check derivatives of a (vectorized) scalar function of a scalar variable.

`sfepy.linalg.check_derivatives.check_vfvx(x0, fx, fx_args, dfx, dfx_args=None, delta=1e-05)`

Check derivatives of a (vectorized) vector or scalar function of a vector variable.

sfepy.linalg.eigen module

`sfepy.linalg.eigen.cg_eigs(mtx, rhs=None, precondition=None, i_max=None, eps_r=1e-10, shift=None, select_indices=None, verbose=False, report_step=10)`

Make several iterations of the conjugate gradients and estimate so the eigenvalues of a (sparse SPD) matrix (Lanczos algorithm).

Parameters

mtx
[spmatrix or array] The sparse matrix A .

precond
[spmatrix or array, optional] The preconditioner matrix. Any object that can be multiplied by vector can be passed.

i_max
[int] The maximum number of the Lanczos algorithm iterations.

eps_r
[float] The relative stopping tolerance.

shift
[float, optional] Eigenvalue shift for non-SPD matrices. If negative, the shift is computed as $|shift| ||A||_{\infty}$.

select_indices
[(min, max), optional] If given, computed only the eigenvalues with indices $min \leq i \leq max$.

verbose
[bool] Verbosity control.

report_step
[int] If *verbose* is True, report in every *report_step*-th step.

Returns

vec
[array] The approximate solution to the linear system.

n_it
[int] The number of CG iterations used.

norm_rs
[array] Convergence history of residual norms.

eigs
[array] The approximate eigenvalues sorted in ascending order.

`sfepy.linalg.eigen.sym_tri_eigen(diags, select_indices=None)`

Compute eigenvalues of a symmetric tridiagonal matrix using `scipy.linalg.eigvals_banded()`.

sfepy.linalg.geometry module

`sfepy.linalg.geometry.barycentric_coors(coors, s_coors)`

Get barycentric (area in 2D, volume in 3D) coordinates of points with coordinates *coors* w.r.t. the simplex given by *s_coors*.

Returns

bc
[array] The barycentric coordinates. Then reference element coordinates $xi = dot(bc, T_{ref\_coors})$.

`sfepy.linalg.geometry.flag_points_in_polygon2d(polygon, coors)`

Test if points are in a 2D polygon.

Parameters

polygon
[array, (:, 2)] The polygon coordinates.

coors: array, (:, 2)
The coordinates of points.

Returns

flag
[bool array] The flag that is True for points that are in the polygon.

Notes

This is a semi-vectorized version of [1].

[1] PNPOLY - Point Inclusion in Polygon Test, W. Randolph Franklin (WRF)

`sfepy.linalg.geometry.get_coors_in_ball(coors, centre, radius, radius2=None, inside=True)`

Return indices of coordinates inside or outside a ball given by centre and radius:

inside	radius	radius2	condition
True	r	None	$ x - c \leq r$
True	r	r2	$r2 \leq x - c \leq r$
False	r	None	$ x - c > r$
False	r	r2	$ x - c > r \mid x - c \leq r2$

Notes

All float comparisons are done using $\leq$ or $\geq$ operators, i.e. the points on the boundaries are taken into account.

`sfepy.linalg.geometry.get_coors_in_tube(coors, centre, axis, radius_in, radius_out, length,
inside_radii=True)`

Return indices of coordinates inside a tube given by centre, axis vector, inner and outer radii and length.

Parameters

inside_radii

[bool, optional] If False, select points outside the radii, but within the tube length.

Notes

All float comparisons are done using $\leq$ or $\geq$ operators, i.e. the points on the boundaries are taken into account.

`sfepy.linalg.geometry.get_face_areas(faces, coors)`

Get areas of planar convex faces in 2D and 3D.

Parameters

faces

[array, shape (n, m)] The indices of n faces with m vertices into *coors*.

coors

[array] The coordinates of face vertices.

Returns

areas

[array] The areas of the faces.

`sfepy.linalg.geometry.get_perpendiculars(vec)`

For a given vector, get a unit vector perpendicular to it in 2D, or get two mutually perpendicular unit vectors perpendicular to it in 3D.

`sfepy.linalg.geometry.get_simplex_circumcentres(coors, force_inside_eps=None)`

Compute the circumcentres of n_s simplices in 1D, 2D and 3D.

Parameters

coors

[array] The coordinates of the simplices with n_v vertices given in an array of shape (n_s , n_v , dim), where dim is the space dimension and $2 \leq n_v \leq (dim + 1)$.

force_inside_eps

[float, optional] If not None, move the circumcentres that are outside of their simplices or closer to their boundary then *force_inside_eps* so that they are inside the simplices at the distance given by *force_inside_eps*. It is ignored for edges.

Returns

centres

[array] The circumcentre coordinates as an array of shape (n_s , dim).

`sfepy.linalg.geometry.get_simplex_volumes(cells, coors)`

Get volumes of simplices in nD.

Parameters

cells

[array, shape (n, d)] The indices of n simplices with d vertices into *coors*.

coors

[array] The coordinates of simplex vertices.

Returns**volumes**

[array] The volumes of the simplices.

`sfepy.linalg.geometry.inverse_element_mapping(coors, e_coors, eval_basis, ref_coors, suppress_errors=False)`

Given spatial element coordinates, find the inverse mapping for points with coordinates $X = X(\mathbf{x}_i)$, i.e. $\mathbf{x}_i = \mathbf{x}_i(X)$.

Returns**xi**

[array] The reference element coordinates.

`sfepy.linalg.geometry.make_axis_rotation_matrix(direction, angle)`

Create a rotation matrix $\underline{\underline{R}}$ corresponding to the rotation around a general axis $\underline{d}$ by a specified angle α .

$$\underline{\underline{R}} = \underline{d}\underline{d}^T + \cos(\alpha)(I - \underline{d}\underline{d}^T) + \sin(\alpha) \text{skew}(\underline{d})$$

Parameters**direction**

[array] The rotation axis direction vector $\underline{d}$.

angle

[float] The rotation angle α .

Returns**mtx**

[array] The rotation matrix $\underline{\underline{R}}$.

Notes

The matrix follows the right hand rule: if the right hand thumb points along the axis vector $\underline{d}$ the fingers show the positive angle rotation direction.

Examples

Make transformation matrix for rotation of coordinate system by 90 degrees around 'z' axis.

```
>>> mtx = make_axis_rotation_matrix([0., 0., 1.], nm.pi/2)
>>> mtx
array([[ 0.,  1.,  0.],
       [-1.,  0.,  0.],
       [ 0.,  0.,  1.]])
```

Coordinates of vector $[1, 0, 0]^T$ w.r.t. the original system in the rotated system. (Or rotation of the vector by -90 degrees in the original system.)

```
>>> nm.dot(mtx, [1., 0., 0.])
>>> array([ 0., -1.,  0.])
```

Coordinates of vector $[1, 0, 0]^T$ w.r.t. the rotated system in the original system. (Or rotation of the vector by +90 degrees in the original system.)

```
>>> nm.dot(mtx.T, [1., 0., 0.])
>>> array([ 0.,  1.,  0.])
```

`sfepy.linalg.geometry.points_in_simplex(coors, s_coors, eps=1e-08)`

Test if points with coordinates *coors* are in the simplex given by *s_coors*.

`sfepy.linalg.geometry.rotation_matrix2d(angle)`

Construct a 2D (plane) rotation matrix corresponding to *angle*.

`sfepy.linalg.geometry.transform_bar_to_space_coors(bar_coors, coors)`

Transform barycentric coordinates *bar_coors* within simplices with vertex coordinates *coors* to space coordinates.

sfepy.linalg.sparse module

Some sparse matrix utilities missing in scipy.

`sfepy.linalg.sparse.compose_sparse(blocks, row_sizes=None, col_sizes=None)`

Compose sparse matrices into a global sparse matrix.

Parameters

blocks

[sequence of sequences] The sequence of sequences of equal lengths - the individual sparse matrix blocks. The integer 0 can be used to mark an all-zero block, if its size can be determined from the other blocks.

row_sizes

[sequence, optional] The required row sizes of the blocks. It can be either a sequence of non-negative integers, or a sequence of slices with non-negative limits. In any case the sizes have to be compatible with the true block sizes. This allows to extend the matrix shape as needed and to specify sizes of all-zero blocks.

col_sizes

[sequence, optional] The required column sizes of the blocks. See *row_sizes*.

Returns

mtx

[coo_matrix] The sparse matrix (COO format) composed from the given blocks.

Examples

Stokes-like problem matrix.

```
>>> import scipy.sparse as sp
>>> A = sp.csr_matrix([[1, 0], [0, 1]])
>>> B = sp.coo_matrix([[1, 1]])
>>> K = compose_sparse([[A, B.T], [B, 0]])
>>> print K.todense()
[[1 0 1]
 [0 1 1]
 [1 1 0]]
```

`sfepy.linalg.sparse.infinity_norm(mtx)`

Infinity norm of a sparse matrix (maximum absolute row sum).

Parameters

mtx

[spmatrix or array] The sparse matrix.

Returns

norm

[float] Infinity norm of the matrix.

 See also

`scipy.linalg.norm`
dense matrix norms

Notes

- This serves as an upper bound on spectral radius.
- CSR and CSC avoid copying *indices* and *indptr* arrays.
- inspired by PyAMG

`sfepy.linalg.sparse.insert_sparse_to_csr(mtx1, mtx2, irs, ics)`

Insert a sparse matrix *mtx2* into a CSR sparse matrix *mtx1* at rows *irs* and columns *ics*. The submatrix *mtx1[irs,ics]* must already be preallocated and have the same structure as *mtx2*.

`sfepy.linalg.sparse.save_sparse_txt(filename, mtx, fnt='%d %d %f\n')`

Save a CSR/CSC sparse matrix into a text file

`sfepy.linalg.sympy_operators` module

`sfepy.linalg.sympy_operators.boundary(f, variables)`

`sfepy.linalg.sympy_operators.default_space_variables(variables)`

`sfepy.linalg.sympy_operators.div(field, variables=None)`

`sfepy.linalg.sympy_operators.grad(f, variables=None)`

`sfepy.linalg.sympy_operators.grad_v(f, variables=None)`

`sfepy.linalg.sympy_operators.laplace(f, variables=None)`

`sfepy.linalg.sympy_operators.set_dim(dim)`

sfepy.linalg.utils module

sfepy.linalg.utils.apply_to_sequence(*seq*, *fun*, *ndim*, *out_item_shape*)

Applies function *fun*() to each item of the sequence *seq*. An item corresponds to the last *ndim* dimensions of *seq*.

Parameters

seq

[array] The sequence array with shape $(n_1, \dots, n_r, m_1, \dots, m_{\{ndim\}})$.

fun

[function] The function taking an array argument of shape of length *ndim*.

ndim

[int] The number of dimensions of an item in *seq*.

out_item_shape

[tuple] The shape an output item.

Returns

out

[array] The resulting array of shape $(n_1, \dots, n_r) + out\_item\_shape$. The *out_item_shape* must be compatible with the *fun*.

sfepy.linalg.utils.argsort_rows(*seq*)

Returns an index array that sorts the sequence *seq*. Works along rows if *seq* is two-dimensional.

sfepy.linalg.utils.assemble1d(*ar_out*, *indx*, *ar_in*)

Perform $ar\_out[indx] += ar\_in$, where items of *ar_in* corresponding to duplicate indices in *indx* are summed together.

sfepy.linalg.utils.chunk_arrays(*arrs*, *chunk_size*)

Yield consecutive views into arrays *arrs* of the same lengths and most *chunk_size* long.

sfepy.linalg.utils.cycle(*bounds*)

Cycles through all combinations of bounds, returns a generator.

More specifically, let *bounds*=[a, b, c, ...], so *cycle* returns all combinations of lists $[0 \leq i < a, 0 \leq j < b, 0 \leq k < c, \dots]$ for all *i,j,k,...*

Examples: In [9]: list(cycle([3, 2])) Out[9]: [[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]

In [14]: list(cycle([3, 4])) [[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1], [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]

sfepy.linalg.utils.dets_fast(*a*)

Fast determinant calculation of 3-dimensional array.

Parameters

a

[array] The input array with shape (m, n, n).

Returns

out

[array] The output array with shape (m,): $out[i] = \det(a[i, :, :])$.

sfepy.linalg.utils.dot_sequences(*mtx*, *vec*, *mode*='AB')

Computes dot product for each pair of items in the two sequences.

Equivalent to

```
>>> out = nm.empty((vec.shape[0], mtx.shape[1], vec.shape[2]),
>>>                  dtype=vec.dtype)
>>> for ir in range(mtx.shape[0]):
>>>     out[ir] = nm.dot(mtx[ir], vec[ir])
```

Parameters

mtx

[array] The array of matrices with shape (n_item, m, n) .

vec

[array] The array of vectors with shape (n_item, a) or matrices with shape (n_item, a, b) .

mode

[one of 'AB', 'ATB', 'ABT', 'ATBT'] The mode of the dot product - the corresponding axes are dotted together:

'AB' : $a = n$ 'ATB' : $a = m$ 'ABT' : $b = n$ (*) 'ATBT' : $b = m$ (*)

(*) The 'BT' part is ignored for the vector second argument.

Returns

out

[array] The resulting array.

Notes

Uses `numpy.matmul()` via the `@` operator.

`sfepy.linalg.utils.get_blocks_stats(blocks, *args)`

Return statistics of array/matrix *blocks* defined by indices in *args*.

Returns

stats: structured array

The array with 'shape', 'min', 'mean' and 'max' fields at positions of each matrix block.

Examples

```
>>> import numpy as nm
>>> from sfepy.linalg.utils import get_blocks_stats
>>>
>>> A = nm.eye(3)
>>> B = nm.full((3,2), 2)
>>> C = nm.full((1,3), 3)
>>> D = nm.full((1,2), 4)
>>> M = nm.block([A, B], [C, D])
>>>
>>> sr = [slice(0, 3), slice(3, 5)]
>>> sc = [slice(0, 3), slice(3, 4)]
>>> stats = get_blocks_stats(M, sr, sc)
>>>
>>> print(stats['shape'])
[[ (3, 3) (3, 1) ]
```

(continues on next page)

(continued from previous page)

```

[(1, 3) (1, 1)]
>>>
>>> print(stats['min'])
[[0. 2.]
 [3. 4.]]

```

`sfepy.linalg.utils.insert_strided_axis(ar, axis, length)`

Insert a new axis of given length into an array using numpy stride tricks, i.e. no copy is made.

Parameters

- ar**
[array] The input array.
- axis**
[int] The axis before which the new axis will be inserted.
- length**
[int] The length of the inserted axis.

Returns

- out**
[array] The output array sharing data with *ar*.

Examples

```

>>> import numpy as nm
>>> from sfepy.linalg import insert_strided_axis
>>> ar = nm.random.rand(2, 1, 2)
>>> ar
array([[[ 0.18905119,  0.44552425]],

```

```

      [[ 0.78593989, 0.71852473]])

```

```

>>> ar.shape
(2, 1, 2)
>>> ar2 = insert_strided_axis(ar, 1, 3)
>>> ar2
array([[[[ 0.18905119,  0.44552425]],

```

```

      [[ 0.18905119, 0.44552425]],
      [[ 0.18905119, 0.44552425]]],
      [[[ 0.78593989, 0.71852473]],
      [[ 0.78593989, 0.71852473]],
      [[ 0.78593989, 0.71852473]])

```

```

>>> ar2.shape
(2, 3, 1, 2)

```

`sfepy.linalg.utils.invs_fast(a, det=None)`

Fast inversion calculation of 4-dimensional array.

Parameters

a

[array] The input array with shape (c, q, n, n).

det: array

To speed up the calculation, enter the already calculated determinant.

Returns

out

[array] The output array with shape (c, q, n, n): `out[c, q] = inv(a[c, q, :, :])`.

`sfepy.linalg.utils.map_permutations(seq1, seq2, check_same_items=False)`

Returns an index array *imap* such that `seq1[imap] == seq2`, if both sequences have the same items - this is not checked by default!

In other words, finds the indices of items of *seq2* in *seq1*.

`sfepy.linalg.utils.max_diff_csr(mtx1, mtx2)`

`sfepy.linalg.utils.mini_newton(fun, x0, dfun, i_max=100, eps=1e-08)`

`sfepy.linalg.utils.norm_l2_along_axis(ar, axis=1, n_item=None, squared=False)`

Compute l2 norm of rows (axis=1) or columns (axis=0) of a 2D array.

n_item ... use only the first *n_item* columns/rows squared ... if True, return the norm squared

`sfepy.linalg.utils.normalize_vectors(vecs, eps=1e-08)`

Normalize an array of vectors in place.

Parameters

vecs

[array] The 2D array of vectors in rows.

eps

[float] The tolerance for considering a vector to have zero norm. Such vectors are left unchanged.

`sfepy.linalg.utils.output_array_stats(ar, name, verbose=True)`

`sfepy.linalg.utils.print_array_info(ar)`

Print array shape and other basic information.

`sfepy.linalg.utils.unique_rows(ar, return_index=False, return_inverse=False)`

Return unique rows of a two-dimensional array *ar*. The arguments follow *numpy.unique()*.

sfepy.mechanics package**sfepy.mechanics.contact_bodies module**

class sfepy.mechanics.contact_bodies.**ContactPlane**(*anchor, normal, bounds*)

get_distance(*points*)

mask_points(*points*)

class sfepy.mechanics.contact_bodies.**ContactSphere**(*centre, radius*)

get_distance(*points*)

Get the penetration distance and normals of points w.r.t. the sphere surface.

Returns

d

[array] The penetration distance.

normals

[array] The normals from the points to the sphere centre.

mask_points(*points, eps*)

sfepy.mechanics.contact_bodies.**plot_points**(*ax, points, marker, **kwargs*)

sfepy.mechanics.contact_bodies.**plot_polygon**(*ax, polygon*)

sfepy.mechanics.elastic_constants module**sfepy.mechanics.matcoefs module**

Conversion of material parameters and other utilities.

class sfepy.mechanics.matcoefs.**ElasticConstants**(*young=None, poisson=None, bulk=None, lam=None, mu=None, p_wave=None, _regenerate_relations=False*)

Conversion formulas for various groups of elastic constants. The elastic constants supported are:

- E : Young's modulus
- ν : Poisson's ratio
- K : bulk modulus
- λ : Lamé's first parameter
- μ, G : shear modulus, Lamé's second parameter
- M : P-wave modulus, longitudinal wave modulus

The elastic constants are referred to by the following keyword arguments: young, poisson, bulk, lam, mu, p_wave.

Exactly two of them must be provided to the `__init__()` method.

Examples

- basic usage:

```
>>> from sfepy.mechanics.matcoefs import ElasticConstants
>>> ec = ElasticConstants(lam=1.0, mu=1.5)
>>> ec.young
3.6000000000000001
>>> ec.poisson
0.20000000000000001
>>> ec.bulk
2.0
>>> ec.p_wave
4.0
>>> ec.get(['bulk', 'lam', 'mu', 'young', 'poisson', 'p_wave'])
[2.0, 1.0, 1.5, 3.6000000000000001, 0.20000000000000001, 4.0]
```

- reinitialize existing instance:

```
>>> ec.init(p_wave=4.0, bulk=2.0)
>>> ec.get(['bulk', 'lam', 'mu', 'young', 'poisson', 'p_wave'])
[2.0, 1.0, 1.5, 3.6000000000000001, 0.20000000000000001, 4.0]
```

get(*names*)

Get the named elastic constants.

init(*young=None, poisson=None, bulk=None, lam=None, mu=None, p_wave=None*)

Set exactly two of the elastic constants, and compute the remaining. (Re)-initializes the existing instance of ElasticConstants.

class sfepy.mechanics.matcoefs.TransformToPlane(*iplane=None*)

Transformations of constitutive law coefficients of 3D problems to 2D.

tensor_plane_stress(*c3=None, d3=None, b3=None*)

Transforms all coefficients of the piezoelectric constitutive law from 3D to plane stress problem in 2D: strain/stress ordering: 11 22 33 12 13 23. If *d3* is None, uses only the stiffness tensor *c3*.

Parameters

c3

[array] The stiffness tensor.

d3

[array] The dielectric tensor.

b3

[array] The piezoelectric coupling tensor.

sfepy.mechanics.matcoefs.bulk_from_lame(*lam, mu*)

Compute bulk modulus from Lamé parameters.

$$\gamma = \lambda + \frac{2}{3}\mu$$

sfepy.mechanics.matcoefs.bulk_from_youngpoisson(*young, poisson, plane='strain'*)

Compute bulk modulus corresponding to Young's modulus and Poisson's ratio.

`sfepy.mechanics.matcoefs.lame_from_stiffness(stiffness, plane='strain')`

Compute Lamé parameters from an isotropic stiffness tensor.

`sfepy.mechanics.matcoefs.lame_from_youngpoisson(young, poisson, plane='strain')`

Compute Lamé parameters from Young's modulus and Poisson's ratio.

The relationship between Lamé parameters and Young's modulus, Poisson's ratio (see [1],[2]):

$$\lambda = \frac{\nu E}{(1 + \nu)(1 - 2\nu)}, \quad \mu = \frac{E}{2(1 + \nu)}$$

The plain stress hypothesis:

$$\bar{\lambda} = \frac{2\lambda\mu}{\lambda + 2\mu}$$

[1] I.S. Sokolnikoff: Mathematical Theory of Elasticity. New York, 1956.

[2] T.J.R. Hughes: The Finite Element Method, Linear Static and Dynamic Finite Element Analysis. New Jersey, 1987.

`sfepy.mechanics.matcoefs.stiffness_from_lame(dim, lam, mu)`

Compute stiffness tensor corresponding to Lamé parameters.

$$\mathbf{D}_{(2D)} = \begin{bmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

$$\mathbf{D}_{(3D)} = \begin{bmatrix} \lambda + 2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda + 2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda + 2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}$$

`sfepy.mechanics.matcoefs.stiffness_from_lame_mixed(dim, lam, mu)`

Compute stiffness tensor corresponding to Lamé parameters for mixed formulation.

$$\mathbf{D}_{(2D)} = \begin{bmatrix} \tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 \\ \tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

$$\mathbf{D}_{(3D)} = \begin{bmatrix} \tilde{\lambda} + 2\mu & \tilde{\lambda} & \tilde{\lambda} & 0 & 0 & 0 \\ \tilde{\lambda} & \tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 & 0 & 0 \\ \tilde{\lambda} & \tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}$$

where

$$\tilde{\lambda} = -\frac{2}{3}\mu$$

`sfepy.mechanics.matcoefs.stiffness_from_youngpoisson(dim, young, poisson, plane='strain')`

Compute stiffness tensor corresponding to Young's modulus and Poisson's ratio.

`sfepy.mechanics.matcoefs.stiffness_from_youngpoisson_mixed(dim, young, poisson, plane='strain')`

Compute stiffness tensor corresponding to Young's modulus and Poisson's ratio for mixed formulation.

`sfepy.mechanics.matcoefs.stiffness_from_yps_ortho3(young, poisson, shear)`

Compute 3D stiffness tensor $\mathbf{D}$ of an orthotropic linear elastic material. Young's modulus ($[E_1, E_2, E_3]$), Poisson's ratio ($[\nu_{12}, \nu_{13}, \nu_{23}]$), and shear modulus ($[G_{12}, G_{13}, G_{23}]$) are given.

$$\mathbf{C}_{(3D)} = \begin{bmatrix} 1/E_1 & -\nu_{21}/E_2 & -\nu_{31}/E_3 & 0 & 0 & 0 \\ -\nu_{12}/E_1 & 1/E_2 & -\nu_{32}/E_3 & 0 & 0 & 0 \\ -\nu_{13}/E_1 & -\nu_{23}/E_2 & 1/E_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/G_{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/G_{13} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/G_{23} \end{bmatrix}$$

$$\mathbf{D}_{(3D)} = \text{inv}(\mathbf{C}_{(3D)})$$

$$\nu_{21} = \nu_{12} \frac{E_2}{E_1}, \quad \nu_{31} = \nu_{13} \frac{E_3}{E_1}, \quad \nu_{32} = \nu_{23} \frac{E_3}{E_2}$$

[1] R.M. Jones: Mechanics of composite materials. 1999.

`sfepy.mechanics.matcoefs.wave_speeds_from_youngpoisson(young, poisson, rho)`

Compute the P- and S-wave speeds from the Young's modulus E and Poisson's ratio ν in a homogeneous isotropic material.

$$v_p^2 = \frac{E(1-\nu)}{\rho(1+\nu)(1-2\nu)} = \frac{(\lambda + 2\mu)}{\rho}$$

$$v_s^2 = \frac{E}{2\rho(1+\nu)} = \frac{\mu}{\rho}$$

Parameters

young

[float or array] The Young's modulus.

poisson

[float or array] The Poisson's ratio.

rho

[float or array] The density.

Returns

vp

[float or array] The P-wave speed.

vs

[float or array] The S-wave speed.

`sfepy.mechanics.matcoefs.youngpoisson_from_stiffness(stiffness, plane='strain')`

Compute Young's modulus and Poisson's ratio from an isotropic stiffness tensor.

`sfepy.mechanics.matcoefs.youngpoisson_from_wave_speeds(vp, vs, rho)`

Compute the Young's modulus E and Poisson's ratio ν from the P- and S-wave speeds in a homogeneous isotropic material.

$$E = \frac{\rho v_s^2 (3v_p^2 - 4v_s^2)}{(v_p^2 - v_s^2)}$$

$$\nu = \frac{(v_p^2/2 - v_s^2)}{(v_p^2 - v_s^2)}$$

Parameters

- vp**
[float or array] The P-wave speed.
- vs**
[float or array] The S-wave speed.
- rho**
[float or array] The density.

Returns

- young**
[float or array] The Young's modulus.
- poisson**
[float or array] The Poisson's ratio.

sfepy.mechanics.membranes module

`sfepy.mechanics.membranes.create_mapping(coors, gel, order)`

Create mapping from transformed (in x - y plane) element faces to reference element faces.

Parameters

- coors**
[array] The transformed coordinates of element nodes, shape (n_{el}, n_{ep}, dim) . The function verifies that the all z components are zero.
- gel**
[GeometryElement instance] The geometry element corresponding to the faces.
- order**
[int] The polynomial order of the mapping.

Returns

- mapping**
[FEMapping instance] The reference element face mapping.

`sfepy.mechanics.membranes.create_transformation_matrix(coors)`

Create a transposed coordinate transformation matrix, that transforms 3D coordinates of element face nodes so that the transformed nodes are in the x - y plane. The rotation is performed w.r.t. the first node of each face.

Parameters

- coors**
[array] The coordinates of element nodes, shape (n_{el}, n_{ep}, dim) .

Returns

- mtx_t**
[array] The transposed transformation matrix T , i.e. $X_{inplane} = T^T X_{3D}$.

Notes

$T = [t_1, t_2, n]$, where t_1, t_2 , are unit in-plane (column) vectors and n is the unit normal vector, all mutually orthonormal.

`sfepy.mechanics.membranes.describe_deformation(el_disps, bfg)`

Describe deformation of a thin incompressible 2D membrane in 3D space, composed of flat finite element faces.

The coordinate system of each element (face), i.e. the membrane mid-surface, should coincide with the x, y axes of the x - y plane.

Parameters

el_disps

[array] The displacements of element nodes, shape (n_{el}, n_{ep}, dim) .

bfg

[array] The in-plane base function gradients, shape $(n_{el}, n_{qp}, dim-1, n_{ep})$.

Returns

mtx_c ; array

The in-plane right Cauchy-Green deformation tensor C_{ij} , $i, j = 1, 2$.

c33

[array] The component C_{33} computed from the incompressibility condition.

mtx_b

[array] The discrete Green strain variation operator.

`sfepy.mechanics.membranes.describe_geometry(field, region, integral)`

Describe membrane geometry in a given region.

Parameters

field

[Field instance] The field defining the FE approximation.

region

[Region instance] The surface region to describe.

integral

[Integral instance] The integral defining the quadrature points.

Returns

mtx_t

[array] The transposed transformation matrix T , see [create_transformation_matrix\(\)](#).

membrane_geo

[CMapping instance] The mapping from transformed elements to a reference elements.

`sfepy.mechanics.membranes.get_green_strain_sym3d(mtx_c, c33)`

Get the 3D Green strain tensor in symmetric storage.

Parameters

mtx_c ; array

The in-plane right Cauchy-Green deformation tensor C_{ij} , $i, j = 1, 2$, shape $(n_{el}, n_{qp}, dim-1, dim-1)$.

c33

[array] The component C_{33} computed from the incompressibility condition, shape (n_{el}, n_{qp}) .

Returns**mtx_e**

[array] The membrane Green strain $E_{ij} = \frac{1}{2}(C_{ij}) - \delta_{ij}$, symmetric storage: items (11, 22, 33, 12, 13, 23), shape $(n\_el, n\_qp, sym, 1)$.

`sfepy.mechanics.membranes.get_invariants(mtx_c, c33)`

Get the first and second invariants of the right Cauchy-Green deformation tensor describing deformation of an incompressible membrane.

Parameters**mtx_c ; array**

The in-plane right Cauchy-Green deformation tensor C_{ij} , $i, j = 1, 2$, shape $(n\_el, n\_qp, dim-1, dim-1)$.

c33

[array] The component C_{33} computed from the incompressibility condition, shape (n_el, n_qp) .

Returns**i1**

[array] The first invariant of C_{ij} .

i2

[array] The second invariant of C_{ij} .

`sfepy.mechanics.membranes.get_tangent_stress_matrix(stress, bfg)`

Get the tangent stress matrix of a thin incompressible 2D membrane in 3D space, given a stress.

Parameters**stress**

[array] The components 11, 22, 12 of the second Piola-Kirchhoff stress tensor, shape $(n\_el, n\_qp, 3, 1)$.

bfg

[array] The in-plane base function gradients, shape $(n\_el, n\_qp, dim-1, n\_ep)$.

Returns**mtx**

[array] The tangent stress matrix, shape $(n\_el, n\_qp, dim*n\_ep, dim*n\_ep)$.

`sfepy.mechanics.membranes.transform_asm_matrices(out, mtx_t)`

Transform matrix assembling contributions to global coordinate system, one node at a time.

Parameters**out**

[array] The array of matrices, transformed in-place.

mtx_t

[array] The transposed transformation matrix T , see [create_transformation_matrix\(\)](#).

`sfepy.mechanics.membranes.transform_asm_vectors(out, mtx_t)`

Transform vector assembling contributions to global coordinate system, one node at a time.

Parameters**out**

[array] The array of vectors, transformed in-place.

mtx_t

[array] The transposed transformation matrix T , see [create_transformation_matrix\(\)](#).

sfepy.mechanics.shell10x module

Functions implementing the shell10x element.

sfepy.mechanics.shell10x.add_eas_dofs(*mtx_b*, *qp_coors*, *det*, *det0*, *dxidx0*)

Add additional strain components [Andelfinger and Ramm] (7 parameters to be condensed out).

sfepy.mechanics.shell10x.create_drl_transform(*ebs*)

Create the transformation matrix for locking of the drilling rotations.

sfepy.mechanics.shell10x.create_elastic_tensor(*young*, *poisson*, *shear_correction=True*)

Create the elastic tensor with the applied shear correction (the default) for the shell10x element.

sfepy.mechanics.shell10x.create_local_bases(*coors*)

Create local orthonormal bases in each vertex of quadrilateral cells.

Parameters

coors

[array] The coordinates of cell vertices, shape (*n_el*, 4, 3).

Returns

ebs

[array] The local bases, shape (*n_el*, 4, 3, 3). The basis vectors are rows of the ($\dots$, 3, 3) blocks.

sfepy.mechanics.shell10x.create_rotation_ops(*ebs*)

Create operators associated to rotation DOFs.

Parameters

ebs

[array] The local bases, shape (*n_el*, 4, 3, 3).

Returns

rops

[array] The rotation operators, shape (*n_el*, 4, 3, 3).

sfepy.mechanics.shell10x.create_strain_matrix(*bfgm*, *dxidx*, *dsg*)

Create the strain operator matrix.

sfepy.mechanics.shell10x.create_strain_transform(*mtx_ts*)

Create strain tensor transformation matrices, given coordinate transformation matrices.

Notes

Expresses TET^T in terms of symmetrix storage as Qe , with the ordering of components: $e = [e_{11}, e_{22}, e_{33}, 2e_{12}, 2e_{13}, 2e_{23}]$.

`sfepy.mechanics.shell10x.create_transformation_matrix(coors)`

Create a transposed coordinate transformation matrix, that transforms 3D coordinates of quadrilateral cell vertices so that the transformed vertices of a plane cell are in the $x - y$ plane. The rotation is performed w.r.t. the centres of quadrilaterals.

Parameters

coors

[array] The coordinates of cell vertices, shape $(n_{el}, 4, 3)$.

Returns

mtx_t

[array] The transposed transformation matrix T , i.e. $X_{inplane} = T^T X_{3D}$.

Notes

$T = [t_1, t_2, n]$, where t_1, t_2 , are unit in-plane (column) vectors and n is the unit normal vector, all mutually orthonormal.

`sfepy.mechanics.shell10x.get_dsg_strain(coors_loc, qp_coors)`

Compute DSG strain components.

Returns

dsg

[array] The strain matrix components corresponding to e_{13}, e_{23} , shape $(n_{el}, n_{qp}, 2, 24)$.

Notes

Involves w, α, β DOFs.

`sfepy.mechanics.shell10x.get_mapping_data(ebs, rops, ps, coors_loc, qp_coors, qp_weights, special_dx3=False)`

Compute reference element mapping data for shell10x elements.

Notes

The code assumes that the quadrature points are w.r.t. (t = thickness of the shell) $[0, 1] \times [0, 1] \times [-t/2, t/2]$ reference cell and the quadrature weights are multiplied by t .

`sfepy.mechanics.shell10x.lock_drilling_rotations(mtx, ebs, coefs)`

Lock the drilling rotations in the stiffness matrix.

`sfepy.mechanics.shell10x.rotate_elastic_tensor(mtx_d, bfu, ebs)`

Rotate the elastic tensor into the local coordinate system of each cell. The local coordinate system results from interpolation of ebs with the bilinear basis.

`sfepy.mechanics.shell10x.transform_asm_matrices(out, mtx_t, blocks)`

Transform matrix assembling contributions to global coordinate system, one node at a time.

Parameters

out

[array] The array of matrices, transformed in-place.

mtx_t

[array] The array of transposed transformation matrices T , see [create_transformation_matrix\(\)](#).

blocks

[array] The DOF blocks that are

sfepy.mechanics.tensors module

Functions to compute some tensor-related quantities usual in continuum mechanics.

class `sfepy.mechanics.tensors.StressTransform(def_grad, jacobian=None)`

Encapsulates functions to convert various stress tensors in the symmetric storage given the deformation state.

get_1pk_from_2pk(stress_in)

Get the first Piola-Kirchhoff stress given the second Piola-Kirchhoff stress.

$$P_{ij} = F_{ik} S_{kj}$$

Parameters

stress_in

[array_like] The second Piola-Kirchhoff stress in vector symmetric storage with the indices ordered as [11, 22, 33, 12, 13, 23]

Returns

stress_out_full

[array] The first Piola-Kirchhoff stress in matrix storage.

get_cauchy_from_2pk(stress_in)

Get the Cauchy stress given the second Piola-Kirchhoff stress.

$$\sigma_{ij} = J^{-1} F_{ik} S_{kl} F_{jl}$$

Parameters

stress_in

[array_like] The second Piola-Kirchhoff stress in vector symmetric storage with the indices ordered as [11, 22, 33, 12, 13, 23]

Returns

stress_out

[array] The Cauchy stress in vector symmetric storage with the indices ordered as [11, 22, 33, 12, 13, 23].

`sfepy.mechanics.tensors.dim2sym(dim)`

Given the space dimension, return the symmetric storage size.

`sfepy.mechanics.tensors.get_cauchy_strain(grad)`

Given a gradient, return the corresponding Cauchy strain (symmetric gradient).

`sfepy.mechanics.tensors.get_deviator(tensor, sym_storage=True)`

The deviatoric part (deviator) of a tensor.

`sfepy.mechanics.tensors.get_full_indices(dim)`

The indices for converting the symmetric storage to the full storage.

`sfepy.mechanics.tensors.get_non_diagonal_indices(dim)`

The non_diagonal indices for the full vector storage.

`sfepy.mechanics.tensors.get_sym_indices(dim)`

The indices for converting the full storage to the symmetric storage.

`sfepy.mechanics.tensors.get_t4_from_t2s(t2s)`

Get the full 4D tensor with major/minor symmetries from its 2D matrix representation.

Parameters

t2s

[array] The symmetrically-stored tensor of shape (S, S), where S it the symmetric storage size.

Returns

t4

[array] The full 4D tensor of shape (D, D, D, D), where D is the space dimension.

`sfepy.mechanics.tensors.get_trace(tensor, sym_storage=True)`

The trace of a tensor.

`sfepy.mechanics.tensors.get_volumetric_tensor(tensor, sym_storage=True)`

The volumetric part of a tensor.

`sfepy.mechanics.tensors.get_von_mises_stress(stress, sym_storage=True)`

Given a symmetric stress tensor, compute the von Mises stress (also known as Equivalent tensile stress).

Notes

$$\sigma_V = \sqrt{\frac{(\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{33})^2 + (\sigma_{11} - \sigma_{33})^2 + 6(\sigma_{12}^2 + \sigma_{13}^2 + \sigma_{23}^2)}{2}}$$

`sfepy.mechanics.tensors.prepare_cylindrical_transform(coors, origin, mode='axes')`

Prepare matrices for transforming tensors into cylindrical coordinates with the axis 'z' in a given origin.

Parameters

coors

[array] The Cartesian coordinates.

origin

[array of length 3] The origin.

mode

['axes' or 'data'] In 'axes' (default) mode the matrix transforms data to different coordinate system, while in 'data' mode the matrix transforms the data in the same coordinate system and is transpose of the matrix in the 'axes' mode.

Returns**mtx**[array] The array of transformation matrices for each coordinate in *coors*.`sfepy.mechanics.tensors.sym2dim(sym)`

Given the symmetric storage size, return the space dimension.

Notes

This function works for any space dimension.

`sfepy.mechanics.tensors.transform_data(data, coors=None, mode='cylindrical', mtx=None)`Transform vector or tensor data components between orthogonal coordinate systems in 3D using transformation matrix M , that should express rotation of the original coordinate system to the new system denoted by $\bullet'$ below.

For vectors:

$$\underline{v}' = M \cdot \underline{v}$$

For second order tensors:

$$\underline{\underline{t}}' = M \cdot \underline{\underline{t}} \cdot M^T$$

or

$$t'_{ij} = M_{ip} M_{jq} t_{pq}$$

For fourth order tensors:

$$t'_{ijkl} = M_{ip} M_{jq} M_{kr} M_{ls} t_{pqrs}$$

Parameters**data**[array, shape (num, n_r) or (num, n_r, n_c)] The vectors (n_r is 3) or tensors (symmetric storage, n_r is 6, n_c , if available, is 1 or 6) to be transformed.**coors**[array] The Cartesian coordinates of the data. Not needed when *mtx* argument is given.**mode**[one of ['cylindrical']] The requested coordinate system. Not needed when *mtx* argument is given.**mtx**[array] The array of transformation matrices M for each data row.**Returns****new_data**

[array] The transformed data.

sfepy.mechanics.units module

Some utilities for work with units of physical quantities.

class sfepy.mechanics.units.Quantity(*name, unit_set*)

A physical quantity in a given set of basic units.

Examples

Construct the stress quantity:

```
>>> from sfepy.mechanics.units import Unit, Quantity
>>> units = ['m', 's', 'kg', 'C']
>>> unit_set = [Unit(key) for key in units]
>>> q1 = Quantity('stress', unit_set)
>>> q1()
'1.0 Pa'
```

Show its unit using various prefixes:

```
>>> q1('m')
'1000.0 mPa'
>>> q1('')
'1.0 Pa'
>>> q1('k')
'0.001 kPa'
>>> q1('M')
'1e-06 MPa'
```

Construct the stress quantity in another unit set:

```
>>> units = ['mm', 's', 'kg', 'C']
>>> unit_set = [Unit(key) for key in units]
>>> q2 = Quantity('stress', unit_set)
>>> q2()
'1.0 kPa'
```

Show its unit using various prefixes:

```
>>> q2('m')
'1000000.0 mPa'
>>> q2('')
'1000.0 Pa'
>>> q2('k')
'1.0 kPa'
>>> q2('M')
'0.001 MPa'
```

class sfepy.mechanics.units.Unit(*name*)

A unit of a physical quantity. The prefix and coefficient of the unit are determined from to its name.

Examples

Construct some units:

```
>>> from sfepy.mechanics.units import Unit
>>> unit = Unit('mm')
>>> print unit
Unit:mm
  coef:
    0.001
  name:
    mm
  prefix:
    m
  prefix_length:
    1
  unit:
    m
>>> unit = Unit('kg')
>>> print unit
Unit:kg
  coef:
    1000.0
  name:
    kg
  prefix:
    k
  prefix_length:
    1
  unit:
    g
```

Get prefixes for a coefficient:

```
>>> Unit.get_prefix(100.0)
('d', 10.0)
>>> Unit.get_prefix(100.0, omit=('d',))
('k', 0.10000000000000001)
```

static `get_prefix(coef, bias=0.1, omit=None)`

Get the prefix and numerical multiplier corresponding to a numerical coefficient, omitting prefixes in omit tuple.

`sfepy.mechanics.units.apply_unit_multipliers(values, unit_kinds, unit_multipliers)`

Apply time, length and mass unit multipliers to given values with units corresponding to unit kinds.

Returns

new_values

[list] The new values with applied unit multipliers

`sfepy.mechanics.units.apply_units_to_pars(pars, pars_kinds, unit_multipliers)`

Apply units in *unit_multipliers* to *pars* according to their kinds.

Parameters

pars

[dict] The input parameters given as *name : value* items.

pars_kinds

[dict] The kinds of the parameters given as *name : kind* items, with kinds defined in [apply_unit_multipliers\(\)](#).

unit_multipliers

[tuple] The time, length and mass unit multipliers.

Returns**new_pars**

[dict] The output parameters.

`sfepy.mechanics.units.get_consistent_unit_set`(*length=None, time=None, mass=None, temperature=None*)

Given a set of basic units, return a consistent set of derived units for quantities listed in the `units_of_quantities` dictionary.

sfepy.mechanics.extmods.ccontres module

`sfepy.mechanics.extmods.ccontres.assemble_contact_residual_and_stiffness`(*Gc, vals, rows, cols, GPs, ISN, IEN, X, Um, H, dH, gw, activeGPsOld, neq, npd, epss, keyContactDetection, keyAssembleKc*)

`sfepy.mechanics.extmods.ccontres.evaluate_contact_constraints`(*GPs, ISN, IEN, N, AABBBmin, AABBBmax, head, next, X, elementID, segmentID, npd, neq, longestEdge*)

`sfepy.mechanics.extmods.ccontres.get_AABB`(*X, longestEdge, IEN, ISN, elementID, segmentID, neq*)

`sfepy.mechanics.extmods.ccontres.get_longest_edge_and_gps`(*GPs, neq, elementID, segmentID, ISN, IEN, H, X*)

`sfepy.mechanics.extmods.ccontres.init_global_search`(*N, AABBBmin, AABBBmax, X*)

The linked list initialization. The head array contains, at the position *Ic*, the index of the first point that belongs to the cell *Ic*, the second point index is then `next[head[Ic]]`, the third point index is `next[next[head[Ic]]]` etc. - the next array points from the *i*-th point in each cell to the (*i*+1)-th point, until -1 is reached.

sfepy.mesh package**sfepy.mesh.bspline module**

class `sfepy.mesh.bspline.BSpline`(*degree=3, is_cyclic=False, ncp=0*)

B-spline curve representation

approximate(*coors, ncp=None, knot_type='clamped', knots=None, alpha=0.5, do_eval=False, do_param_correction=False*)

Approximate set of points by the B-spline curve.

Parameters**coors**

[array] The coordinates of the approximated points.

nep

[int] The number of control points.

knot_type

[str] The knot vector type.

knots

[array] The knot vector.

alpha

[float]

The parameter vector distribution:

1.0 = chordal 0.5 = centripetal

do_eval

[bool] Evaluate the curve coordinates?

do_param_correction

[bool] Perform parametric corrections to improve the approximation?

static basis_function_dg(*degree, t, knots, n*)

B-spline basis functions.

Parameters**degree**

[int] The degree of the spline function.

t

[array] The parametric vector.

knots

[array] The knot vector.

n

[int] The number of intervals.

Returns**bfun**

[array] The spline basis function evaluated for given values.

static basis_function_dg0(*t, knots, n*)

Basis function: degree = 0

Parameters**t**

[array] The parametric vector.

knots

[array] The knot vector.

n

[int] The number of intervals.

Returns

bfun

[array] The spline basis function evaluated for given values.

draw(*ret_ax=False, ax=None, color='r', cp_id=True*)

Draw B-spline curve.

Parameters**ret_ax**

[bool] Return an axes object?

ax

[axes object] The axes to which will be drawn.

color

[str] Line color.

cp_id

[bool] If True, label control points.

draw_basis()

Draw B-spline curve.

eval(*t=None, cp_coors=None*)

Evaluate the coordinates of the bpspline curve.

Parameters**t**

[array] The parameter vector of the B-spline.

cp_coors

[array] The coordinates of the control points.

eval_basis(*t=None, return_val=False*)

Evaluate the basis of the bpspline.

Parameters**t**

[array] The parameter vector of the B-spline.

get_control_points()

Get the B-spline control points.

Returns**coors**

[array] The coordinates of control points.

get_knot_vector()

Return the knot vector.

Returns**knots**

[array] The knot vector.

insert_knot(*new*)

Insert a new knot into the knot vector.

Parameters

new

[float] The new knot value.

make_knot_vector(*knot_type='clamped', knot_data=None, knot_range=(0.0, 1.0)*)

Create a knot vector of the requested type.

Parameters

knot_type

[str] The knot vector type: clamped/cyclic/userdef.

knot_data

The extra knot data.

set_approx_points(*coors*)

Set the coordinates of approximated points.

Parameters

coors

[array] The coordinates of approximated points.

set_control_points(*coors, cyclic_form=False*)

Set the B-spline control points.

Parameters

coors

[array] The coordinates of unique control points.

cyclic_form

[bool] Are the control points in the cyclic form?

set_knot_vector(*knots*)

Set the knot vector.

Parameters

knots

[array] The knot vector.

set_param(*t*)

Set the B-spline parametric vector.

Parameters

t

[array] The parameter vector of the B-spline.

set_param_n(*n=100, knot_range=(0.0, 1.0)*)

Generate the B-spline parametric vector using the number of steps.

Parameters

n

[array] The number of steps in the B-spline parametric vector.

class sfepy.mesh.bspline.**BSplineSurf**(*degree=(3, 3), is_cyclic=(False, False)*)

B-spline surface representation

approximate(*coors, ncp, do_eval=False*)

Approximate set of points by the B-spline surface.

Parameters

coors

[array] The coordinates of the approximated points.

nep

[tuple of int] The number of control points.

draw(*ret_ax=False, ax=None*)

Draw B-spline surface.

Parameters

ret_ax

[bool] Return an axes object?

ax

[axes object] The axes to which will be drawn.

eval(*t=(None, None), cp_coors=None*)

Evaluate the coordinates of the bpspline curve.

Parameters

t

[tuple of array] The parametric vector of the B-splines.

cp_coors

[array] The coordinates of the control points.

get_control_points()

Get the B-spline surface control points.

Returns

coors

[array] The coordinates of control points.

make_knot_vector(*knot_type=('clamped', 'clamped'), knot_data=(None, None)*)

Create a knot vector of the requested type.

Parameters

knot_type

[tuple of str] The knot vector types.

knot_data

[tuple of ANY] The extra knot data.

set_approx_points(*coors*)

Set the coordinates of approximated points.

Parameters

coors

[array] The coordinates of approximated points.

set_control_points(*coors, cyclic_form=False*)

Set the B-spline control points.

Parameters

coors

[array] The coordinates of unique control points.

cyclic_form

[bool] Are the control points in the cyclic form?

set_param_n(*n*=(100, 100))

Generate the B-spline parametric vector using the number of steps.

Parameters**n**

[tuple of array] The number of steps in the B-spline parametric vectors.

write_control_polygon_vtk(*filename*, *float_format*='%0.6f')

Write the control polygon to VTK file.

Parameters**filename: str**

Name of the VTK file.

float_format: str

Float formatting.

write_surface_vtk(*filename*, *float_format*='%0.6f')

Write the spline surface to VTK file.

Parameters**filename: str**

Name of the VTK file.

float_format: str

Float formatting.

sfepy.mesh.bspline.approximation_example()

The example of using BSplineSurf for approximation of the surface given by the set of points.

sfepy.mesh.bspline.get_2d_points(*is3d=False*)

Returns the set of points.

Parameters**is3d**

[bool] 3D coordinates?

sfepy.mesh.bspline.main(*argv*)

sfepy.mesh.bspline.simple_example()

The example of using B-spline class.

sfepy.mesh.bspline.to_ndarray(*a*)

sfePy.mesh.geom_tools module

class `sfePy.mesh.geom_tools.geometry(dim=3)`

The geometry is given by a sets of points (d0), lines (d1), surfaces (d2) and volumes (d3). A lines are constructed from 2 points, a surface from any number of lines, a volume from any number of surfaces.

Physical volumes are contruted from any number of volumes.

The `self.d0`, `self.d1`, `self.d2` and `self.d3` are dictionaries holding a map geometry element number -> instance of point,line,surface of volume

Examples

To get all the points which define a surface 5, use:

```
self.d2[5].getpoints()
```

This would give you a list [...] of `point()` instances.

addline(*n, l*)

`l=[p1,p2]`

addlines(*ls, off=1*)

`ls=[l1, l2, ...]`

addphysicalsurface(*n, surfacelist*)

`surfacelist=[s1,s2,s3,...]`

addphysicalvolume(*n, volumelist*)

`volumelist=[v1,v2,v3,...]`

addpoint(*n, p*)

`p=[x,y,z]`

addpoints(*ps, off=1*)

`ps=[p1, p2, ...]`

addsurface(*n, s, is_hole=False*)

`s=[l1,l2,l3,...]`

addsurfaces(*ss, off=1*)

`s=[s1,s2,s3,...]`

addvolume(*n, v*)

`v=[s1,s2,s3,...]`

addvolumes(*vs, off=1*)

`v=[v1,v2,v3,...]`

static from_gmsh_file(*filename*)

Import geometry - Gmsh geometry format.

Parameters

filename

[string] file name

Returns

```
        geo
            [geometry] geometry description
getBCnum(snum)
leaveonlyphysicalsurfaces()
leaveonlyphysicalvolumes()
printinfo(verbose=False)
splitlines(ls, n)
to_poly_file(filename)
    Export geometry to poly format (tetgen and triangle geometry format).

Parameters

    geo
        [geometry] geometry description
    filename
        [string] file name
class sfepy.mesh.geom_tools.geomobject

    getn()
class sfepy.mesh.geom_tools.line(g, n, l)

    getpoints()
class sfepy.mesh.geom_tools.physicalsurface(g, n, s)

    getsurfaces()
class sfepy.mesh.geom_tools.physicalvolume(g, n, v)

    getvolumes()
class sfepy.mesh.geom_tools.point(g, n, p)

    getstr()
    getxyz()
class sfepy.mesh.geom_tools.surface(g, n, s, is_hole=False)

    getcenterpoint()
    getholepoints()
    getinsidepoint()
    getlines()
    getpoints()
    separate(s)
class sfepy.mesh.geom_tools.volume(g, n, v)
```


getinsidepoint()

getsurfaces()

sfepy.mesh.mesh_generators module

sfepy.mesh.mesh_generators.gen_block_mesh(*dims, shape, centre, mat_id=0, name='block', coors=None, verbose=True*)

Generate a 2D or 3D block mesh. The dimension is determined by the lenght of the shape argument.

Parameters

dims

[array of 2 or 3 floats] Dimensions of the block.

shape

[array of 2 or 3 ints] Shape (counts of nodes in x, y, z) of the block mesh.

centre

[array of 2 or 3 floats] Centre of the block.

mat_id

[int, optional] The material id of all elements.

name

[string] Mesh name.

verbose

[bool] If True, show progress of the mesh generation.

Returns

mesh

[Mesh instance]

sfepy.mesh.mesh_generators.gen_cylinder_mesh(*dims, shape, centre, axis='x', force_hollow=False, is_open=False, open_angle=0.0, non_uniform=False, make_2d=False, name='cylinder', verbose=True*)

Generate a cylindrical mesh along an axis. Its cross-section can be ellipsoidal.

Parameters

dims

[array of 5 floats] Dimensions of the cylinder: inner surface semi-axes a1, b1, outer surface semi-axes a2, b2, length. The length can be zero, resulting in a planar annular topology.

shape

[array of 3 ints] Shape (counts of nodes in radial, circumferential and longitudinal directions) of the cylinder mesh.

centre

[array of 3 floats] Centre of the cylinder.

axis: one of 'x', 'y', 'z'

The axis of the cylinder.

force_hollow

[boolean] Force hollow mesh even if inner radii a1 = b1 = 0.

is_open

[boolean] Generate an open cylinder segment.

open_angle

[float] Opening angle in radians.

non_uniform

[boolean] If True, space the mesh nodes in radial direction so that the element volumes are (approximately) the same, making thus the elements towards the outer surface thinner.

make_2d

[boolean] If True, generate an annular mesh in the x-y plane. Sets length to 0.

name

[string] Mesh name.

verbose

[bool] If True, show progress of the mesh generation.

Returns**mesh**

[Mesh instance]

`sfepy.mesh.mesh_generators.gen_extended_block_mesh(b_dims, b_shape, e_dims, e_shape, centre, grading_fun=None, name=None)`

Generate a 3D mesh with a central block and (coarse) extending side meshes.

The resulting mesh is again a block. Each of the components has a different material id.

Parameters**b_dims**

[array of 3 floats] The dimensions of the central block.

b_shape

[array of 3 ints] The shape (counts of nodes in x, y, z) of the central block mesh.

e_dims

[array of 3 floats] The dimensions of the complete block (central block + extensions).

e_shape

[int] The count of nodes of extending blocks in the direction from the central block.

centre

[array of 3 floats] The centre of the mesh.

grading_fun

[callable, optional] A function of $x \in [0, 1]$ that can be used to shift nodes in the extension axis directions to allow smooth grading of element sizes from the centre. The default function is $x * * p$ with p determined so that the element sizes next to the central block have the size of the shortest edge of the central block.

name

[string, optional] The mesh name.

Returns**mesh**

[Mesh instance]

`sfepy.mesh.mesh_generators.gen_mesh_from_geom(geo, a=None, verbose=False, refine=False)`

Runs mesh generator - tetgen for 3D or triangle for 2D meshes.

Parameters

geo
[geometry] geometry description

a
[int, optional] a maximum area/volume constraint

verbose
[bool, optional] detailed information

refine
[bool, optional] refines mesh

Returns

mesh
[Mesh instance] triangular or tetrahedral mesh

`sfepy.mesh.mesh_generators.gen_mesh_from_string(mesh_name, mesh_dir)`

`sfepy.mesh.mesh_generators.gen_mesh_from_voxels(voxels, dims, etype='q')`

Generate FE mesh from voxels (volumetric data).

Parameters

voxels
[array] Voxel matrix, 1=material.

dims
[array] Size of one voxel.

etype
[integer, optional] 'q' - quadrilateral or hexahedral elements 't' - triangular or tetrahedral elements

Returns

mesh
[Mesh instance] Finite element mesh.

`sfepy.mesh.mesh_generators.gen_misc_mesh(mesh_dir, force_create, kind, args, suffix='.mesh', verbose=False)`

Create sphere or cube mesh according to *kind* in the given directory if it does not exist and return path to it.

`sfepy.mesh.mesh_generators.gen_tiled_mesh(mesh, grid=None, scale=1.0, eps=1e-06, ret_ndmap=False)`

Generate a new mesh by repeating a given periodic element along each axis.

Parameters

mesh
[Mesh instance] The input periodic FE mesh.

grid
[array] Number of repetition along each axis.

scale
[float, optional] Scaling factor.

eps
[float, optional] Tolerance for boundary detection.

ret_ndmap
[bool, optional] If True, return global node map.

Returns**mesh_out**

[Mesh instance] FE mesh.

ndmap

[array] Maps: actual node id → node id in the reference cell.

`sfepy.mesh.mesh_generators.get_tensor_product_conn(shape)`

Generate vertex connectivity for cells of a tensor-product mesh of the given shape.

Parameters**shape**

[array of 2 or 3 ints] Shape (counts of nodes in x, y, z) of the mesh.

Returns**conn**

[array] The vertex connectivity array.

desc

[str] The cell kind.

`sfepy.mesh.mesh_generators.main()`

`sfepy.mesh.mesh_generators.tiled_mesh1d(conn, coors, ngrps, idim, n_rep, bb, eps=1e-06, ndmap=False)`

sfepy.mesh.mesh_tools module

`sfepy.mesh.mesh_tools.elems_q2t(el)`

`sfepy.mesh.mesh_tools.expand2d(mesh2d, dist, rep)`

Expand 2D planar mesh into 3D volume, convert triangular/quad mesh to tetrahedrons/hexahedrons.

Parameters**mesh2d**

[Mesh] The 2D mesh.

dist

[float] The elements size in the 3rd direction.

rep

[int] The number of elements in the 3rd direction.

Returns**mesh3d**

[Mesh] The 3D mesh.

`sfepy.mesh.mesh_tools.extract_edges(mesh, eps=1e-16)`

Extract outline edges of a given mesh. The outline edge is an edge for which $\text{norm}(\text{nvec}_1 - \text{nvec}_2) < \text{eps}$, where nvec_1 and nvec_2 are the normal vectors of the incident facets.

Parameters**mesh**

[Mesh] The 3D or 2D mesh.

eps

[float] The tolerance parameter of the outline edge searching algorithm.

Returns**mesh_out**

[tuple] The data of the outline mesh, Mesh.from_data() format, i.e. (coors, ngroups, ed_conns, mat_ids, descs).

`sfepy.mesh.mesh_tools.get_surface_faces(domain)`

`sfepy.mesh.mesh_tools.merge_lines(mesh, eps=1e-18)`

Merge edges of an edge-only mesh that are in the same direction w.r.t. the tolerance *eps*.

`sfepy.mesh.mesh_tools.smooth_mesh(mesh, n_iter=4, lam=0.6307, mu=-0.6347, weights=None, bconstr=True, volume_corr=False)`

FE mesh smoothing.

Based on:

[1] Steven K. Boyd, Ralph Muller, Smooth surface meshing for automated finite element model generation from 3D image data, Journal of Biomechanics, Volume 39, Issue 7, 2006, Pages 1287-1295, ISSN 0021-9290, 10.1016/j.jbiomech.2005.03.006. (<http://www.sciencedirect.com/science/article/pii/S0021929005001442>)

Parameters**mesh**

[mesh] FE mesh.

n_iter

[integer, optional] Number of iteration steps.

lam

[float, optional] Smoothing factor, see [1].

mu

[float, optional] Unshrinking factor, see [1].

weights

[array, optional] Edge weights, see [1].

bconstr: logical, optional

Boundary constraints, if True only surface smoothing performed.

volume_corr: logical, optional

Correct volume after smoothing process.

Returns**coors**

[array] Coordinates of mesh nodes.

`sfepy.mesh.mesh_tools.surface_components(gr_s, surf_faces)`

Determine surface components given surface mesh connectivity graph.

`sfepy.mesh.mesh_tools.surface_graph(surf_faces, n_nod)`

`sfepy.mesh.mesh_tools.triangulate(mesh, verbose=False)`

Triangulate a 2D or 3D tensor product mesh: quadrilaterals->triangles, hexahedrons->tetrahedrons.

Parameters**mesh**

[Mesh] The input mesh.

Returns

mesh

[Mesh] The triangulated mesh.

sfePy.mesh.splinebox module

class `sfePy.mesh.splinebox.SplineBox(bbox, coors, nsg=None, field=None)`

B-spline geometry parametrization. The geometry can be modified by moving spline control points.

static `create_spb(bbox, coors, degree=3, nsg=None)`

evaluate(*cp_values=None, outside=True*)

Evaluate the new position of the mesh coordinates.

Parameters

cp_values

[array] The actual control point values. If None, use self.control_values.

outside

[bool] If True, return also the coordinates outside the spline box.

Returns

new_coors

[array] The new position of the mesh coordinates.

evaluate_derivative(*cpoint, dirvec*)

Evaluate derivative of the spline in a given control point and direction.

Parameters

cpoint

[int, list] The position (index or grid indicies) of the spline control point.

dirvec

[array] The directional vector.

Returns

diff

[array] The derivative field.

static `gen_cp_idx(ncp)`

`get_box_matrix()`

Returns:

mtx

[2D array] The matrix containing the coefficients of b-spline basis functions.

get_control_points(*init=False*)

Get the spline control points coordinates.

Returns

cpt_coors

[array] The coordinates of the spline control points.

init

[bool] If True, return the initial state.

get_coors_shape()

Get the shape of the coordinates.

move_control_point(*cpoint*, *val*)

Change shape of spline parametrization.

Parameters**cpoint**

[int, list] The position (index or grid indicies) of the spline control point.

val

[array] Displacement.

set_control_points(*cpt_coors*, *add=False*)

Set the spline control points position.

Parameters**cpt_coors**

[array] The coordinates of the spline control points.

add

[bool] If True, coors += cpt_coors

write_control_net(*filename*, *deform_by_values=True*)

Write the SplineBox shape to the VTK file.

Parameters**filename**

[str] The VTK file name.

class sfepy.mesh.splinebox.SplineRegion2D(*spl_bnd*, *coors*, *rho=1000.0*)

B-spline geometry parametrization. The boundary of the SplineRegion2D is defined by BSpline curves.

static create_spb(*spl_bnd*, *coors*, *rho=10*)

Initialize SplineBox knots, control points, base functions, ...

static define_control_points(*cp_bnd_coors*, *ncp*)

Find positions of “inner” control points depending on boundary splines.

find_ts(*coors*)

Function finds parameters (t, s) corresponding to given points (coors).

static points_in_poly(*points*, *poly*, *tol=1e-06*)

Find which points are located inside the polygon.

sfepy.parallel package**sfepy.parallel.evaluate module**

PETSc-related parallel evaluation of problem equations.

class sfepy.parallel.evaluate.PETScParallelEvaluator(*problem*, *pdofs*, *drange*, *is_overlap*, *psol*, *comm*, *matrix_hook=None*, *verbose=False*)

The parallel evaluator of the problem equations for [PETScNonlinearSolver](#).

Its methods can be used as the function and Jacobian callbacks of the PETSc SNES (Scalable Nonlinear Equations Solvers).

Notes

Assumes `problem.active_only == False`.

eval_residual(*snes, psol, prhs*)

eval_tangent_matrix(*snes, psol, pmtx, ppmtx*)

sfepy.parallel.parallel module

Functions for a high-level PETSc-based parallelization.

sfepy.parallel.parallel.assemble_mtx_to_petsc(*pmtx, mtx, pdofs, drange, is_overlap=True, comm=None, verbose=False*)

Assemble a local CSR matrix to a global PETSc matrix.

sfepy.parallel.parallel.assemble_rhs_to_petsc(*prhs, rhs, pdofs, drange, is_overlap=True, comm=None, verbose=False*)

Assemble a local right-hand side vector to a global PETSc vector.

sfepy.parallel.parallel.call_in_rank_order(*fun, comm=None*)

Call a function *fun* task by task in the task rank order.

sfepy.parallel.parallel.create_gather_scatter(*pdofs, pvec_i, pvec, comm=None*)

Create the `gather()` function for updating a global PETSc vector from local ones and the `scatter()` function for updating local PETSc vectors from the global one.

sfepy.parallel.parallel.create_gather_to_zero(*pvec*)

Create the `gather_to_zero()` function for collecting the global PETSc vector on the task of rank zero.

sfepy.parallel.parallel.create_local_petsc_vector(*pdofs*)

Create a local PETSc vector with the size corresponding to *pdofs*.

sfepy.parallel.parallel.create_petsc_matrix(*sizes, mtx_prealloc=None, comm=None*)

Create and allocate a PETSc matrix.

sfepy.parallel.parallel.create_petsc_system(*mtx, sizes, pdofs, drange, is_overlap=True, comm=None, verbose=False*)

Create and pre-allocate (if *is_overlap* is True) a PETSc matrix and related solution and right-hand side vectors.

sfepy.parallel.parallel.create_prealloc_data(*mtx, pdofs, drange, verbose=False*)

Create CSR preallocation data for a PETSc matrix based on the owned PETSc DOFs and a local matrix with EBCs not applied.

sfepy.parallel.parallel.create_task_dof_maps(*field, cell_tasks, inter_facets, is_overlap=True, use_expand_dofs=False, save_inter_regions=False, output_dir=None*)

For each task list its inner and interface DOFs of the given field and create PETSc numbering that is consecutive in each subdomain.

For each task, the DOF map has the following structure:

```
[inner,
 [own_inter1, own_inter2, ...],
 [overlap_cells1, overlap_cells2, ...],
 n_task_total, task_offset]
```


The overlapping cells are defined so that the system matrix corresponding to each task can be assembled independently, see [1]. TODO: Some “corner” cells may be added even if not needed - filter them out by using the PETSc DOFs range.

When debugging domain partitioning problems, it is advisable to set `save_inter_regions` to True to save the task interfaces as meshes as well as vertex-based markers - to be used only with moderate problems and small numbers of tasks.

[1] J. Sistek and F. Cirak. Parallel iterative solution of the incompressible Navier-Stokes equations with application to rotating wings. Submitted for publication, 2015

```
sfepy.parallel.parallel.distribute_field_dofs(field, gfd, use_expand_dofs=False, comm=None,
                                              verbose=False)
```

Distribute the owned cells and DOFs of the given field to all tasks.

The DOFs use the PETSc ordering and are in form of a connectivity, so that each task can easily identify them with the DOFs of the original global ordering or local ordering.

```
sfepy.parallel.parallel.distribute_fields_dofs(fields, cell_tasks, is_overlap=True,
                                              use_expand_dofs=False, save_inter_regions=False,
                                              output_dir=None, comm=None, verbose=False)
```

Distribute the owned cells and DOFs of the given field to all tasks.

Uses interleaved PETSc numbering in each task, i.e., the PETSc DOFs of each tasks are consecutive and correspond to the first field DOFs block followed by the second etc.

Expand DOFs to equations if `use_expand_dofs` is True.

```
sfepy.parallel.parallel.expand_dofs(dofs, n_components)
```

Expand DOFs to equation numbers.

```
sfepy.parallel.parallel.get_composite_sizes(lfds)
```

Get (local, total) sizes of a vector and local equation range for a composite matrix built from field blocks described by `lfds` local field distributions information.

```
sfepy.parallel.parallel.get_inter_facets(domain, cell_tasks)
```

For each couple of neighboring task subdomains get the common boundary (interface) facets.

```
sfepy.parallel.parallel.get_local_ordering(field_i, petsc_dofs_conn, use_expand_dofs=False)
```

Get PETSc DOFs in the order of local DOFs of the localized field `field_i`.

Expand DOFs to equations if `use_expand_dofs` is True.

```
sfepy.parallel.parallel.get_sizes(petsc_dofs_range, n_dof, n_components)
```

Get (local, total) sizes of a vector and local equation range.

```
sfepy.parallel.parallel.init_petsc_args()
```

```
sfepy.parallel.parallel.partition_mesh(mesh, n_parts, usemetis=True, verbose=False)
```

Partition the mesh cells into `n_parts` subdomains, using metis, if available.

```
sfepy.parallel.parallel.setup_composite_dofs(lfds, fields, local_variables, verbose=False)
```

Setup composite DOFs built from field blocks described by `lfds` local field distributions information.

Returns (local, total) sizes of a vector, local equation range for a composite matrix, and the local ordering of composite PETSc DOFs, corresponding to `local_variables` (must be in the order of `fields`!).

```
sfepy.parallel.parallel.verify_task_dof_maps(dof_maps, id_map, field, use_expand_dofs=False,
                                              verbose=False)
```

Verify the counts and values of DOFs in *dof_maps* and *id_map* corresponding to *field*.

Returns the vector with a task number for each DOF.

`sfepy.parallel.parallel.view_petsc_local(data, name='data', viewer=None, comm=None)`

View local PETSc *data* called *name*. The data object has to have *.view()* method.

sfepy.parallel.plot_parallel_dofs module

Functions to visualize the partitioning of a domain and a field DOFs.

`sfepy.parallel.plot_parallel_dofs.label_dofs(ax, coors, dofs, colors)`

Label DOFs using the given colors.

`sfepy.parallel.plot_parallel_dofs.mark_subdomains(ax, cmesh, cell_tasks, size=None, icolor=0, alpha=1.0, mask=False)`

Mark cells of subdomains corresponding to each task by a different color. Plots nothing in 3D.

`sfepy.parallel.plot_parallel_dofs.plot_local_dofs(axes, field, field_i, omega_gi, output_dir, rank)`

Plot the local and global field DOFs local to the subdomain on the task with the given *rank*.

`sfepy.parallel.plot_parallel_dofs.plot_partitioning(axes, field, cell_tasks, gfd, output_dir, size)`

Plot the partitioning of the domain and field DOFs.

sfepy.postprocess package

sfepy.postprocess.plot_cmesh module

Functions to visualize the CMesh geometry and topology.

`sfepy.postprocess.plot_cmesh.label_global_entities(ax, cmesh, edim, color='b', fontsize=10, **kwargs)`

Label mesh topology entities using global ids.

`sfepy.postprocess.plot_cmesh.label_local_entities(ax, cmesh, edim, color='b', fontsize=10, **kwargs)`

Label mesh topology entities using cell-local ids.

`sfepy.postprocess.plot_cmesh.plot_cmesh(ax, cmesh, wireframe_opts=None, entities_opts=None)`

Convenience function for plotting all entities of a finite element mesh.

Pass *plot()* arguments to *wireframe_opts* dict.

Pass '*color*', '*label_global*', '*label_global*' for *text()* color and font sizes arguments and '*size*' for *scatter()* to each dict for topological entities in *entities_opts* list.

Examples

```
>>> # 2D mesh.
>>> plot_cmsh(None, cmesh,
              wireframe_opts = {'color' : 'k', 'linewidth' : 2},
              entities_opts=[
                {'color' : 'k', 'label_local' : 8, 'size' : 20},
                {'color' : 'b', 'label_global' : 12, 'label_local' : 8, 'size' : 10},
                {'color' : 'r', 'label_global' : 12, 'size' : 20},
              ])

```

`sfepy.postprocess.plot_cmsh.plot_entities(ax, cmesh, edim, color='b', size=10, **kwargs)`

Plot mesh topology entities using scatter plot.

`sfepy.postprocess.plot_cmsh.plot_wireframe(ax, cmesh, color='k', **kwargs)`

Plot a finite element mesh as a wireframe using edges connectivity.

sfepy.postprocess.plot_dofs module

Functions to visualize the mesh connectivity with global and local DOF numberings.

`sfepy.postprocess.plot_dofs.plot_global_dofs(ax, coors, econn)`

Plot global DOF numbers given in an extended connectivity.

The DOF numbers are plotted for each element, so on common facets they are plotted several times - this can be used to check the consistency of the global DOF connectivity.

`sfepy.postprocess.plot_dofs.plot_local_dofs(ax, coors, econn)`

Plot local DOF numbers corresponding to an extended connectivity.

`sfepy.postprocess.plot_dofs.plot_mesh(ax, coors, conn, edges, color='k', **plot_kwargs)`

Plot a finite element mesh as a wireframe.

`sfepy.postprocess.plot_dofs.plot_nodes(ax, coors, econn, ref_nodes, dofs)`

Plot Lagrange reference element nodes corresponding to global DOF numbers given in an extended connectivity.

`sfepy.postprocess.plot_dofs.plot_points(ax, coors, vals=None, point_size=20, show_colorbar=False)`

Plot points with given coordinates, optionally colored using `vals` values.

sfepy.postprocess.plot_facets module

Functions to visualize the geometry elements and numbering and orientation of their facets (edges and faces).

The standard geometry elements can be plotted by running:

```
$ python sfepy/postprocess/plot_facets.py

```

`sfepy.postprocess.plot_facets.draw_arrow(ax, coors, angle=20.0, length=0.3, **kwargs)`

Draw a line ended with an arrow head, in 2D or 3D.

`sfepy.postprocess.plot_facets.plot_edges(ax, gel, length)`

Plot edges of a geometry element as numbered arrows.

`sfepy.postprocess.plot_facets.plot_faces(ax, gel, radius, n_point)`

Plot faces of a 3D geometry element as numbered oriented arcs. An arc centre corresponds to the first node of a face. It points from the first edge towards the last edge of the face.

`sfepy.postprocess.plot_facets.plot_geometry(ax, gel)`

Plot a geometry element as a wireframe.

sfepy.postprocess.plot_quadrature module

Functions to visualize quadrature points in reference elements.

`sfepy.postprocess.plot_quadrature.label_points(ax, coors)`

Label points with their indices.

`sfepy.postprocess.plot_quadrature.plot_quadrature(ax, geometry, order, boundary=False,
min_radius=10, max_radius=50,
show_colorbar=False, show_labels=False)`

Plot quadrature points for the given geometry and integration order.

The points are plotted as circles/spheres with radii given by quadrature weights - the weights are mapped to `[min_radius, max_radius]` interval.

`sfepy.postprocess.plot_quadrature.plot_weighted_points(ax, coors, weights, min_radius=10,
max_radius=50, show_colorbar=False)`

Plot points with given coordinates as circles/spheres with radii given by weights.

sfepy.postprocess.probes_vtk module

Classes for probing values of Variables, for example, along a line, using PyVTK library

class `sfepy.postprocess.probes_vtk.Probe(data, mesh, **kwargs)`

Probe class.

add_circle_probe(*name, centre, normal, radius, n_point*)

Create the ray (line) probe - VTK object.

Parameters

name

[str] The probe name.

centre

[array] The coordinates of the circle center point.

normal

[array] The normal vector perpendicular to the circle plane.

radius

[float] The radius of the circle.

n_point

[int] The number of probe points.

add_line_probe(*name, p0, p1, n_point*)

Create the line probe - VTK object.

Parameters

name
[str] The probe name.

p0
[array_like] The coordinates of the start point.

p1
[array_like] The coordinates of the end point.

n_point
[int] The number of probe points.

add_points_probe(*name, coors*)

Create the point probe - VTK object.

Parameters

name
[str] The probe name.

coors
[array] The coordinates of the probe points.

add_ray_probe(*name, p0, dirvec, p_fun, n_point*)

Create the ray (line) probe - VTK object.

Parameters

name
[str] The probe name.

p0
[array] The coordinates of the start point.

dirvec
[array] The probe direction vector.

p_fun
[function] The function returning the probe parametrization along the dirvec direction.

n_point
[int] The number of probe points.

gen_mesh_probe_png(*probe, png_filename*)

Generate PNG image of the FE mesh.

Parameters

probe
[VTK objectstr] The probe, VTKPolyData or VTKSource.

png_filename
[str] The name of the output PNG file.

new_vtk_polyline(*points, closed=False*)

Create the VTKPolyData object and store the line data.

Parameters

points
[array] The line points.

Returns

vtkpd

[VTK object] VTKPolyData with the polyline.

class sfepy.postprocess.probes_vtk.**ProbeFromFile**(filename, **kwargs)

Probe class - read a given VTK file.

sfepy.postprocess.time_history module

sfepy.postprocess.time_history.**average_vertex_var_in_cells**(ths_in)

Average histories in the element nodes for each nodal variable originally requested in elements.

sfepy.postprocess.time_history.**dump_to_vtk**(filename, output_filename_trunk=None, step0=0, steps=None, fields=None, linearization=None)

Dump a multi-time-step results file into a sequence of VTK files.

sfepy.postprocess.time_history.**extract_time_history**(filename, extract, verbose=True)

Extract time history of a variable from a multi-time-step results file.

Parameters**filename**

[str] The name of file to extract from.

extract

[str] The description of what to extract in a string of comma-separated description items. A description item consists of: name of the variable to extract, mode ('e' for elements, 'n' for nodes), ids of the nodes or elements (given by the mode). Example: 'u n 10 15, p e 0' means variable 'u' in nodes 10, 15 and variable 'p' in element 0.

verbose

[bool] Verbosity control.

Returns**ths**

[dict] The time histories in a dict with variable names as keys. If a nodal variable is requested in elements, its value is a dict of histories in the element nodes.

ts

[TimeStepper instance] The time stepping information.

sfepy.postprocess.time_history.**extract_times**(filename)

Read true time step data from individual time steps.

Returns**steps**

[array] The time steps.

times

[array] The times of the time steps.

nts

[array] The normalized times of the time steps, in [0, 1].

dts

[array] The true time deltas.

`sfepy.postprocess.time_history.guess_time_units(times)`

Given a vector of times in seconds, return suitable time units and new vector of times suitable for plotting.

Parameters

times

[array] The vector of times in seconds.

Returns

new_times

[array] The vector of times in *units*.

units

[str] The time units.

`sfepy.postprocess.time_history.save_time_history(ths, ts, filename_out)`

Save time history and time-stepping information in a HDF5 file.

sfepy.postprocess.utils_vtk module

Postprocessing utils based on VTK library

`sfepy.postprocess.utils_vtk.get_vtk_by_group(vtkdata, group_lower, group_upper=None)`

Get submesh by material group id.

Parameters

vtkdata

[VTK object] Mesh, scalar, vector and tensor data.

group_lower

[int] The lower material id.

group_upper

[int] The Upper material id.

Returns

sselection

[VTK object] Mesh, scalar, vector and tensor data.

`sfepy.postprocess.utils_vtk.get_vtk_edges(vtkdata)`

Get mesh edges.

Parameters

vtkdata

[VTK object] Mesh, scalar, vector and tensor data.

Returns

edges

[VTK object] Mesh, scalar, vector and tensor data.

`sfepy.postprocess.utils_vtk.get_vtk_from_file(filename)`

Read VTK file.

Parameters

filename

[str] Name of the VTK file.

Returns**vtkdata**

[VTK object] Mesh, scalar, vector and tensor data.

`sfepy.postprocess.utils_vtk.get_vtk_from_mesh(mesh, data, prefix="")`

`sfepy.postprocess.utils_vtk.get_vtk_surface(vtkdata)`

Get mesh surface.

Parameters**vtkdata**

[VTK object] Mesh, scalar, vector and tensor data.

Returns**surface**

[VTK object] Mesh, scalar, vector and tensor data.

`sfepy.postprocess.utils_vtk.tetrahedralize_vtk_mesh(vtkdata)`

3D cells are converted to tetrahedral meshes, 2D cells to triangles.

Parameters**vtkdata**

[VTK object] Mesh, scalar, vector and tensor data.

Returns**tetra**

[VTK object] Mesh, scalar, vector and tensor data.

`sfepy.postprocess.utils_vtk.write_vtk_to_file(filename, vtkdata)`

Write VTK file.

Parameters**filename**

[str] Name of the VTK file.

vtkdata

[VTK object] Mesh, scalar, vector and tensor data.

sfepy.solvers package

sfepy.solvers.auto_fallback module

class `sfepy.solvers.auto_fallback.AutoDirect(conf, **kwargs)`

The automatically selected linear direct solver.

The first available solver from the following list is used: `ls.mumps` <`sfepy.solvers.ls.MUMPSSolver`>, `ls.scipy_umfpack` <`sfepy.solvers.ls.ScipyUmfpack`> and `ls.scipy_superlu` <`sfepy.solvers.ls.ScipySuperLU`>.

Kind: 'ls.auto_direct'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters


```
name = 'ls.auto_direct'
```

```
class sfepy.solvers.auto_fallback.AutoFallbackSolver(conf, **kwargs)
```

Base class for virtual solvers with the automatic fallback.

```
class sfepy.solvers.auto_fallback.AutoIterative(conf, **kwargs)
```

The automatically selected linear iterative solver.

The first available solver from the following list is used: *ls.petsc* <*sfepy.solvers.ls.PETScKrylovSolver*> and *ls.scipy_iterative* <*sfepy.solvers.ls.ScipyIterative*>

Kind: 'ls.auto_iterative'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

```
name = 'ls.auto_iterative'
```

sfepy.solvers.eigen module

```
class sfepy.solvers.eigen.LOBPCGEigenvalueSolver(conf, **kwargs)
```

SciPy-based LOBPCG solver for sparse symmetric problems.

Kind: 'eig.scipy_lobpcg'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

i_max

[int (default: 20)] The maximum number of iterations.

eps_a

[float] The absolute tolerance for the convergence.

largest

[bool (default: True)] If True, solve for the largest eigenvalues, otherwise the smallest.

precond

[{dense matrix, sparse matrix, LinearOperator}] The preconditioner.

```
name = 'eig.scipy_lobpcg'
```

```
class sfepy.solvers.eigen.MatlabEigenvalueSolver(conf, comm=None, context=None, **kwargs)
```

Matlab eigenvalue problem solver.

Kind: 'eig.matlab'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[{'eig', 'eigs', None} (default: 'eigs')] The solution method. Note that eig() function cannot be used for all inputs. If *n_eigs* is not None, eigs() is used regardless of this parameter.

balance

[{'balance', 'nobalance'}] (default: 'balance') The balance option for eig().

algorithm

[{'chol', 'qz'}] (default: 'chol') The algorithm option for eig().

which

[{'lm', 'sm', 'la', 'sa', 'be', 'lr', 'sr', 'li', 'si', 'sigma'}] (default: 'lm') Which eigenvectors and eigenvalues to find with eigs().

*

[*] Additional parameters supported by eigs().

name = 'eig.matlab'

solver_call(mtx_filename, eigs_filename, which=None)

class sfepy.solvers.eigen.OctaveEigenvalueSolver(conf, comm=None, context=None, **kwargs)

Octave eigenvalue problem solver.

Kind: 'eig.octave'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'eig.octave'

solver_call(mtx_filename, eigs_filename, which)

class sfepy.solvers.eigen.PrimmeEigenvalueSolver(conf, comm=None, context=None, **kwargs)

PRIMME eigenvalue problem solver.

<https://github.com/primme/primme>

Installation: pip install primme

Kind: 'eig.primme'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**which**

[{'LM', 'SM', 'LA', 'SA', 'CLT', 'CGT'}] (default: 'LM') Which eigenvectors and eigenvalues to find.

sigma

[float] Find eigenvalues near sigma.

maxiter

[int] Maximum number of iterations.

tol

[float (default: 0)] Tolerance for eigenpairs (stopping criterion).

*

[*] Additional parameters supported by eigsh().

```
name = 'eig.primme'
```

```
class sfepy.solvers.eigen.SLEPcEigenvalueSolver(conf, comm=None, context=None, **kwargs)
```

General SLEPc eigenvalue problem solver.

Kind: 'eig.slepc'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[str (default: 'krylovschur')] The actual solver to use.

problem

[str (default: 'gnhep')] The problem type: Hermitian (hep), non-Hermitian (nhep), generalized Hermitian (ghep), generalized non-Hermitian (gnhep), generalized non-Hermitian with positive semi-definite B (pgnhep), and generalized Hermitian-indefinite (ghiep).

i_max

[int (default: 20)] The maximum number of iterations.

eps

[float] The convergence tolerance.

conv_test

[{"abs", "rel", "norm", "user"}, (default: 'abs')] The type of convergence test.

which

[{'largest_magnitude', 'smallest_magnitude', 'largest_real', 'smallest_real', 'largest_imaginary', 'smallest_imaginary', 'target_magnitude', 'target_real', 'target_imaginary', 'all', 'which_user'} (default: 'largest_magnitude')] Which eigenvectors and eigenvalues to find.

*

[*] Additional parameters supported by the method.

```
create_eps(options=None, comm=None)
```

```
create_petsc_matrix(mtx, comm=None)
```

```
name = 'eig.slepc'
```

```
class sfepy.solvers.eigen.ScipyEigenvalueSolver(conf, **kwargs)
```

SciPy-based solver for both dense and sparse problems.

The problem is considered sparse if *n_eigs* argument is not None.

Kind: 'eig.scipy'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[{'eig', 'eigh', 'eigs', 'eigsh'} (default: 'eigs')] The method for solving general or symmetric eigenvalue problems: for dense problems [eig\(\)](#) or [eigh\(\)](#) can be used, for sparse problems [eigs\(\)](#) or [eigsh\(\)](#) should be used.

which

['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI' (default: 'SM')] Which eigenvectors and eigenvalues to find, see `scipy.sparse.linalg.eigs()` or `scipy.sparse.linalg.eigsh()`. For dense problems, only 'LM' and 'SM' can be used

linear_solver

[({'ls.cholesky', 'ls.mumps', ... }, ls_conf)] The method used to construct an inverse linear operator. If None, the eigenvalue solver will solve the linear system internally.

*

[*] Additional parameters supported by the method.

name = 'eig.scipy'

class `sfepy.solvers.eigen.ScipySGEigenvalueSolver`(*conf*, ***kwargs*)

SciPy-based solver for dense symmetric problems.

Kind: 'eig.sgscipy'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'eig.sgscipy'

`sfepy.solvers.eigen.eig`(*mtx_a*, *mtx_b=None*, *n_eigs=None*, *eigenvectors=True*, *return_time=None*, *solver_kind='eig.scipy'*, ***kwargs*)

Utility function that constructs an eigenvalue solver given by *solver_kind*, calls it and returns its output.

`sfepy.solvers.eigen.init_slepc_args`()

`sfepy.solvers.eigen.standard_call`(*call*)

Decorator handling argument preparation and timing for eigensolvers.

sfepy.solvers.ls module

class `sfepy.solvers.ls.CholeskySolver`(*conf*, ***kwargs*)

Interface to scikit-sparse.cholesky solver.

Kind: 'ls.cholesky'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**use_presolve**

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, `.clear()` has to be called manually whenever the matrix changes - expert use only!

name = 'ls.cholesky'

presolve(*mtx*, *use_mtx_digest=True*)

class sfepy.solvers.ls.**MUMPSSolver**(*conf*, ***kwargs*)

Interface to MUMPS solver.

Kind: 'ls.mumps'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

use_presolve

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

memory_relaxation

[int (default: 20)] The percentage increase in the estimated working space.

clear()

static coo_is_symmetric(*mtx*, *tol=1e-09*)

Symmetry check of the sparse matrix.

name = 'ls.mumps'

presolve(*mtx*, *use_mtx_digest=True*, *factorize=True*)

class sfepy.solvers.ls.**MultiProblem**(*conf*, *context=None*, ***kwargs*)

Conjugate multiple problems.

Allows to define conjugate multiple problems.

Kind: 'ls.cm_pb'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[{'auto', 'umfpack', 'superlu'} (default: 'auto')] The actual solver to use.

use_presolve

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

others

[list] The list of auxiliary problem definition files.

coupling_variables

[list] The list of coupling variables.

init_subproblems(*conf*, ***kwargs*)

name = 'ls.cm_pb'

sparse_submat(*Ad, Ar, Ac, gr, gc, S*)

$A[gr,gc] = S$

class sfepy.solvers.ls.**PETScKrylovSolver**(*conf, comm=None, context=None, **kwargs*)

PETSc Krylov subspace solver.

The solver supports parallel use with a given MPI communicator (see *comm* argument of `PETScKrylovSolver.__init__()`) and allows passing in PETSc matrices and vectors. Returns a (global) PETSc solution vector instead of a (local) numpy array, when given a PETSc right-hand side vector.

The solver and preconditioner types are set upon the solver object creation. Tolerances can be overridden when called by passing a *conf* object.

Convergence is reached when $rnorm < \max(eps_r * rnorm_0, eps_a)$, where, in PETSc, *rnorm* is by default the norm of *preconditioned* residual.

Kind: 'ls.petsc'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[str (default: 'cg')] The actual solver to use.

setup_precond

[callable] User-supplied function for the preconditioner initialization/setup. It is called as `setup_precond(mtx, context)`, where *mtx* is the matrix, *context* is a user-supplied context, and should return an object with *setUp(self, pc)* and *apply(self, pc, x, y)* methods. Has precedence over the *precond/sub_precond* parameters.

precond

[str (default: 'icc')] The preconditioner.

sub_precond

[str (default: 'none')] The preconditioner for matrix blocks (in parallel runs).

precond_side

[['left', 'right', 'symmetric', None]] The preconditioner side.

create_matrix

[callable] User-defined function returning the linear system matrix and optionally a preconditioning matrix in a PETSc format. It is called as `create_matrix(self, mtx, context, comm)`, where *self* is the solver instance, *mtx* is the matrix passed to `self.__call__()`, *context* is a user-supplied context and *comm* the PETSc communicator.

block_size

[int] The number of degrees of freedom per node.

i_max

[int (default: 100)] The maximum number of iterations.

eps_a

[float (default: 1e-08)] The absolute tolerance for the residual.

eps_r

[float (default: 1e-08)] The relative tolerance for the residual.

eps_d

[float (default: 100000.0)] The divergence tolerance for the residual.

force_reuse

[bool (default: False)] If True, skip the check whether the KSP solver object corresponds to the *mtx* argument: it is always reused.

*

[*] Additional parameters supported by the method. Can be used to pass all PETSc options supported by `petsc.Options()`.

create_ksp(*options=None, comm=None*)

create_petsc_matrix(*mtx, comm=None*)

init_ksp(*conf=None, comm=None*)

name = 'ls.petsc'

set_field_split(*field_ranges, comm=None*)

Setup local PETSc ranges for fields to be used with 'fieldsplit' preconditioner.

This function must be called before solving the linear system.

class sfepy.solvers.ls.**PyAMGKrylovSolver**(*conf, context=None, **kwargs*)

Interface to PyAMG Krylov solvers.

Kind: 'ls.pyamg_krylov'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**method**

[str (default: 'cg')] The actual solver to use.

setup_precond

[callable (default: <function PyAMGKrylovSolver.<lambda> at 0x7db6dc730310>)] User-supplied function for the preconditioner initialization/setup. It is called as `setup_precond(mtx, context)`, where *mtx* is the matrix, *context* is a user-supplied context, and should return one of {sparse matrix, dense matrix, LinearOperator}.

callback

[callable] User-supplied function to call after each iteration. It is called as `callback(xk)`, where *xk* is the current solution vector, except the `gmres` method, where the argument is the residual norm.

i_max

[int (default: 100)] The maximum number of iterations.

eps_r

[float (default: 1e-08)] The relative tolerance for the residual.

*

[*] Additional parameters supported by the method.

name = 'ls.pyamg_krylov'

class sfepy.solvers.ls.**PyAMGSolver**(*conf, **kwargs*)

Interface to PyAMG solvers.

The *method* parameter can be one of: 'smoothed_aggregation_solver', 'ruge_stuben_solver'. The *accel* parameter specifies the Krylov solver name, that is used as an accelerator for the multigrid solver.

Kind: 'ls.pyamg'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[str (default: 'smoothed_aggregation_solver')] The actual solver to use.

accel

[str] The accelerator.

callback

[callable] User-supplied function to call after each iteration. It is called as `callback(xk)`, where `xk` is the current solution vector, except the gmres accelerator, where the argument is the residual norm.

i_max

[int (default: 100)] The maximum number of iterations.

eps_r

[float (default: 1e-08)] The relative tolerance for the residual.

force_reuse

[bool (default: False)] If True, skip the check whether the MG solver object corresponds to the `mtx` argument: it is always reused.

*

[*] Additional parameters supported by the method. Use the 'method:' prefix for arguments of the method construction function (e.g. 'method:max_levels' : 5), and the 'solve:' prefix for the subsequent solver call.

name = 'ls.pyamg'

class sfepy.solvers.ls.**PyPardisoSolver**(*conf*, *method=None*, ***kwargs*)

PyPardiso direct solver.

PyPardiso (<https://github.com/haasad/PyPardiso>) is a python package to solve large sparse linear systems of equations with the Intel oneAPI Math Kernel Library PARDISO solver, a shared-memory multiprocessing parallel direct sparse solver. PyPardiso is not a python interface to the PARDISO Solver Project (<https://panua.ch/pardiso>).

Kind: 'ls.pypardiso'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

use_presolve

[bool (default: False)] If True, pre-factorize the matrix. It is not needed for performance here, as it just calls `pypardiso.spsolve()` on zeros.

name = 'ls.pypardiso'

presolve(*mtx*, *use_mtx_digest=True*)

class sfepy.solvers.ls.**RMMSolver**(*conf*, *context=None*, ***kwargs*)

Special solver for explicit transient elastodynamics.

The solver uses the reciprocal mass matrix algorithm [1], [2] to directly construct a sparse inverse mass matrix. Instead of solving a linear system, calling the solver simply performs a sparse matrix multiplication.

Limitations:

- Assumes that the density is constant in time.
- Uses the direct EBC application, i.e., no EBC projection matrix.

Kind: 'ls.rmm'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

rmm_term

[str] The RMM term definition, see [MassTerm](#).

debug

[bool (default: False)] If True, run in debug mode.

init_rmm(mtx)

name = 'ls.rmm'

class sfepy.solvers.ls.**SchurMumps**(conf, **kwargs)

Mumps Schur complement solver.

Kind: 'ls.schur_mumps'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

use_presolve

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

memory_relaxation

[int (default: 20)] The percentage increase in the estimated working space.

schur_variables

[list] The list of Schur variables.

name = 'ls.schur_mumps'

class sfepy.solvers.ls.**ScipyDirect**(conf, method=None, **kwargs)

Direct sparse solver from SciPy.

Kind: 'ls.scipy_direct'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[{'auto', 'umfpack', 'superlu'} (default: 'auto')] The actual solver to use.

use_presolve

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

clear()

name = 'ls.scipy_direct'

presolve(mtx, use_mtx_digest=True)

class sfepy.solvers.ls.ScipyIterative(conf, context=None, **kwargs)

Interface to SciPy iterative solvers.

The *eps_r* tolerance is both absolute and relative - the solvers stop when either the relative or the absolute residual is below it.

Kind: 'ls.scipy_iterative'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[str (default: 'cg')] The actual solver to use.

setup_precond

[callable (default: <function ScipyIterative.<lambda> at 0x7db6dc70bc70>)] User-supplied function for the preconditioner initialization/setup. It is called as setup_precond(mtx, context), where mtx is the matrix, context is a user-supplied context, and should return one of {sparse matrix, dense matrix, LinearOperator}.

callback

[callable] User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector, except the gmres method, where the argument is the residual.

i_max

[int (default: 100)] The maximum number of iterations.

eps_a

[float (default: 1e-08)] The absolute tolerance for the residual.

eps_r

[float (default: 1e-08)] The relative tolerance for the residual.

*

[*] Additional parameters supported by the method.

name = 'ls.scipy_iterative'

class sfepy.solvers.ls.ScipySuperLU(conf, **kwargs)

SuperLU - direct sparse solver from SciPy.

Kind: 'ls.scipy_superlu'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

use_presolve

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest.
If False, .clear() has to be called manually whenever the matrix changes - expert use only!

name = 'ls.scipy_superlu'

class sfepy.solvers.ls.ScipyUmfpack(*conf*, ***kwargs*)

UMFPACK - direct sparse solver from SciPy.

Kind: 'ls.scipy_umfpack'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**use_presolve**

[bool (default: False)] If True, pre-factorize the matrix.

use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest.
If False, .clear() has to be called manually whenever the matrix changes - expert use only!

name = 'ls.scipy_umfpack'

sfepy.solvers.ls.petsc_call(*call*)

Decorator handling argument preparation and timing for PETSc-based linear solvers.

sfepy.solvers.ls.solve(*mtx*, *rhs*, *solver_class=None*, *solver_conf=None*)

Solve the linear system with the matrix *mtx* and the right-hand side *rhs*.

Convenience wrapper around the linear solver classes below.

sfepy.solvers.ls.standard_call(*call*)

Decorator handling argument preparation and timing for linear solvers.

sfepy.solvers.nls module

Nonlinear solvers.

class sfepy.solvers.nls.Newton(*conf*, ***kwargs*)

Solves a nonlinear system $f(x) = 0$ using the Newton method.

The solver uses a backtracking line-search on divergence.

Kind: 'nls.newton'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**i_max**

[int (default: 1)] The maximum number of iterations.

eps_a

[float (default: 1e-10)] The absolute tolerance for the residual, i.e. $||f(x^i)||$.

eps_r

[float (default: 1.0)] The relative tolerance for the residual, i.e. $\|f(x^i)\|/\|f(x^0)\|$.

eps_mode

['and' or 'or' (default: 'and')] The logical operator to use for combining the absolute and relative tolerances.

macheps

[float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

scale_system_fun

[function(mtx, rhs, x0, context)] User-defined function for scaling the linear system and initial guess in each iteration.

scale_solution_fun

[function(x, context)] User-defined function for scaling the solution in each iteration.

scaled_error

[bool (default: False)] If True, the error of the linear solver is calculated using the scaled values.

lin_red

[float or None (default: 1.0)] The linear system solution error should be smaller than ($eps_a * lin\_red$), otherwise a warning is printed. If None, the check is skipped.

lin_precision

[float or None] If not None, the linear system solution tolerances are set in each nonlinear iteration relative to the current residual norm by the *lin_precision* factor. Ignored for direct linear solvers.

step_red

[0.0 < float <= 1.0 (default: 1.0)] Step reduction factor. Equivalent to the mixing parameter a : $(1 - a)x + a(x + dx) = x + adx$

line_search_fun

[function(it, vec_x0, vec_dx0, err_last, conf, fun, timers, log=None) (default: <function apply_line_search_bt at 0x7db6dc70a290>)] The line search function.

ls_on

[float (default: 0.99999)] Start the backtracking line-search by reducing the step, if $\|f(x^i)\|/\|f(x^{i-1})\|$ is larger than *ls_on*.

ls_red

[0.0 < float < 1.0 (default: 0.1)] The step reduction factor in case of correct residual assembling.

ls_red_warp

[0.0 < float < 1.0 (default: 0.001)] The step reduction factor in case of failed residual assembling (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

ls_min

[0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.

give_up_warp

[bool (default: False)] If True, abort on the “warp violation” error.

check

[0, 1 or 2 (default: 0)] If >= 1, check the tangent matrix using finite differences. If 2, plot the resulting sparsity patterns.

delta

[float (default: 1e-06)] If *check* ≥ 1 , the finite difference matrix is taken as $A_{ij} = \frac{f_i(x_j + \delta) - f_i(x_j - \delta)}{2\delta}$.

log

[dict or None] If not None, log the convergence according to the configuration in the following form: {'text' : 'log.txt', 'plot' : 'log.pdf'}. Each of the dict items can be None.

log_vlines

['iteration' or 'solve' or both (default: ('solve'))] Put log vertical lines after each iteration and/or before the solve.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

```
__annotations__ = {}
```

```
__call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None,
        **kwargs)
```

Call self as a function.

```
__init__(conf, **kwargs)
```

```
__module__ = 'sfepy.solvers.nls'
```

```
name = 'nls.newton'
```

```
class sfepy.solvers.nls.PETScNonlinearSolver(conf, pmtx=None, prhs=None, comm=None, **kwargs)
```

Interface to PETSc SNES (Scalable Nonlinear Equations Solvers).

The solver supports parallel use with a given MPI communicator (see *comm* argument of [PETScNonlinearSolver.__init__\(\)](#)). Returns a (global) PETSc solution vector instead of a (local) numpy array, when given a PETSc initial guess vector.

For parallel use, the *fun* and *fun_grad* callbacks should be provided by [PETScParallelEvaluator](#).

Kind: 'nls.petsc'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**method**

[str (default: 'newtonls')] The SNES type.

i_max

[int (default: 10)] The maximum number of iterations.

if_max

[int (default: 100)] The maximum number of function evaluations.

eps_a

[float (default: 1e-10)] The absolute tolerance for the residual, i.e. $\|f(x^i)\|$.

eps_r

[float (default: 1.0)] The relative tolerance for the residual, i.e. $\|f(x^i)\|/\|f(x^0)\|$.

eps_s

[float (default: 0.0)] The convergence tolerance in terms of the norm of the change in the solution between steps, i.e. $\|\delta x\| < \text{epsilon}_s \|x\|$

```
__annotations__ = {}

__call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None,
         **kwargs)

    Call self as a function.

__init__(conf, pmtx=None, prhs=None, comm=None, **kwargs)

__module__ = 'sfepy.solvers.nls'

name = 'nls.petsc'
```

```
class sfepy.solvers.nls.ScipyRoot(conf, **kwargs)
```

Interface to `scipy.optimize.root()`.

Kind: 'nls.scipy_root'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

method

[str (default: 'anderson')] Type of solver supported in `scipy.optimize.root()`, one of: ['hybr', 'lm', 'broyden1', 'broyden2', 'anderson', 'linearmixing', 'diagbroyden', 'excitingmixing', 'krylov', 'df-sane']

use_jacobian

[bool (default: False)] If True, use the exact Jacobian.

tol

[float] Tolerance for termination. For detailed control, use solver-specific options.

callback

[callback(x, f)] Optional callback function. It is called on every iteration as with x the current solution and f the corresponding residual. For all methods but 'hybr' and 'lm'.

options

[dict] A dictionary of solver options. E.g., *xtol* or *maxiter*, see `scipy.optimize.show_options('root')` for details.

```
__annotations__ = {}

__call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None,
         **kwargs)

    Call self as a function.

__init__(conf, **kwargs)

__module__ = 'sfepy.solvers.nls'

methods = ['hybr', 'lm', 'broyden1', 'broyden2', 'anderson', 'linearmixing',
           'diagbroyden', 'excitingmixing', 'krylov', 'df-sane']

name = 'nls.scipy_root'
```

```
sfepy.solvers.nls.apply_line_search_bt(vec_x0, vec_dx0, it, err_last, conf, fun, timers, log=None,
                                       context=None)
```

Apply a backtracking line-search.

`sfepy.solvers.nls.check_tangent_matrix(conf, vec_x0, fun, fun_grad)`

Verify the correctness of the tangent matrix as computed by `fun_grad()` by comparing it with its finite difference approximation evaluated by repeatedly calling `fun()` with `vec_x0` items perturbed by a small delta.

`sfepy.solvers.nls.conv_test(conf, it, err, err0)`

Nonlinear solver convergence test.

Parameters

conf

[Struct instance] The nonlinear solver configuration.

it

[int] The current iteration.

err

[float] The current iteration error.

err0

[float] The initial error.

Returns

status

[int] The convergence status: -1 = no convergence (yet), 0 = solver converged - tolerances were met, 1 = max. number of iterations reached.

`sfepy.solvers.nls.standard_nls_call(call)`

Decorator handling argument preparation and timing for nonlinear solvers.

sfepy.solvers.optimize module

`class sfepy.solvers.optimize.FMinSteepestDescent(conf, **kwargs)`

Steepest descent optimization solver.

Kind: 'opt.fmin_sd'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

i_max

[int (default: 10)] The maximum number of iterations.

eps_rd

[float (default: 1e-05)] The relative delta of the objective function.

eps_of

[float (default: 0.0001)] The tolerance for the objective function.

eps_ofg

[float (default: 1e-08)] The tolerance for the objective function gradient.

norm

[numpy norm (default: inf)] The norm to be used.

ls

[bool (default: True)] If True, use a line-search.

ls_method

[{'backtracking', 'full'} (default: 'backtracking')] The line-search method.

ls_on

[float (default: 0.99999)] Start the backtracking line-search by reducing the step, if $\|f(x^i)\|/\|f(x^{i-1})\|$ is larger than *ls_on*.

ls0

[0.0 < float < 1.0 (default: 1.0)] The initial step.

ls_red

[0.0 < float < 1.0 (default: 0.5)] The step reduction factor in case of correct residual assembling.

ls_red_warp

[0.0 < float < 1.0 (default: 0.1)] The step reduction factor in case of failed residual assembling (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

ls_min

[0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.

check

[0, 1 or 2 (default: 0)] If ≥ 1 , check the tangent matrix using finite differences. If 2, plot the resulting sparsity patterns.

delta

[float (default: 1e-06)] If *check* ≥ 1 , the finite difference matrix is taken as $A_{ij} = \frac{f_i(x_j + \delta) - f_i(x_j - \delta)}{2\delta}$.

output

[function] If given, use it instead of `output()` function.

yscales

[list of str (default: ['linear', 'log', 'log', 'linear'])] The list of four convergence log subplot scales.

log

[dict or None] If not None, log the convergence according to the configuration in the following form: {'text' : 'log.txt', 'plot' : 'log.pdf'}. Each of the dict items can be None.

name = 'opt.fmin_sd'

class sfepy.solvers.optimize.**ScipyFMinSolver**(*conf*, ***kwargs*)

Interface to SciPy optimization solvers `scipy.optimize.fmin_*`.

Kind: 'nls.scipy_fmin_like'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**method**

[{'fmin', 'fmin_bfgs', 'fmin_cg', 'fmin_cobyla', 'fmin_l_bfgs_b', 'fmin_ncg', 'fmin_powell', 'fmin_slsqp', 'fmin_tnc'} (default: 'fmin')] The actual optimization method to use.

i_max

[int (default: 10)] The maximum number of iterations.


```

        *
        [*] Additional parameters supported by the method.
    name = 'nls.scipy_fmin_like'

    set_method(conf)

sfepy.solvers.optimize.check_gradient(xit, aofg, fn_of, delta, check)

sfepy.solvers.optimize.conv_test(conf, it, of, of0, ofg_norm=None)

```

Returns

```

    flag
    [int]
    • -1 ... continue
    • 0 ... small OF -> stop
    • 1 ... i_max reached -> stop
    • 2 ... small OFG -> stop
    • 3 ... small relative decrease of OF

sfepy.solvers.optimize.wrap_function(function, args)

```

sfepy.solvers.oseen module

```
class sfepy.solvers.oseen.Oseen(conf, context=None, **kwargs)
```

The Oseen solver for Navier-Stokes equations.

Kind: 'nls.oseen'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

```

    stabil_mat
    [str] The name of stabilization material.

    adimensionalize
    [bool (default: False)] If True, adimensionalize the problem (not implemented!).

    check_navier_stokes_residual
    [bool (default: False)] If True, check the Navier-Stokes residual after the nonlinear loop.

    i_max
    [int (default: 1)] The maximum number of iterations.

    eps_a
    [float (default: 1e-10)] The absolute tolerance for the residual, i.e.  $\|f(x^i)\|$ .

    eps_r
    [float (default: 1.0)] The relative tolerance for the residual, i.e.  $\|f(x^i)\|/\|f(x^0)\|$ .

    macheps
    [float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

```

lin_red

[float (default: 1.0)] The linear system solution error should be smaller than ($\epsilon_{ps\_a} * \text{lin\_red}$), otherwise a warning is printed.

lin_precision

[float or None] If not None, the linear system solution tolerances are set in each nonlinear iteration relative to the current residual norm by the *lin_precision* factor. Ignored for direct linear solvers.

name = 'nls.oseen'

```
class sfepy.solvers.oseen.StabilizationFunction(name_map, gamma=None, delta=None, tau=None,
                                              tau_red=1.0, tau_mul=1.0, delta_mul=1.0,
                                              gamma_mul=1.0, diameter_mode='max')
```

Definition of stabilization material function for the Oseen solver.

Notes

- $\text{tau\_red} \leq 1.0$; if tau is None: $\text{tau} = \text{tau\_red} * \text{delta}$
- diameter mode: 'edge': longest edge 'volume': volume-based, 'max': max. of previous

get_maps()

Get the maps of names and indices of variables in state vector.

setup(problem)

Setup common problem-dependent data.

```
sfepy.solvers.oseen.are_close(a, b, rtol=0.2, atol=1e-08)
```

```
sfepy.solvers.oseen.scale_matrix(mtx, indx, factor)
```

sfepy.solvers.qeigen module

Quadratic eigenvalue problem solvers.

```
class sfepy.solvers.qeigen.LQuadraticEVPSolver(conf, mtx_m=None, mtx_d=None, mtx_k=None,
                                              n_eigs=None, eigenvectors=None, status=None,
                                              context=None, **kwargs)
```

Quadratic eigenvalue problem solver based on the problem linearization.

$(w^2 M + w D + K) x = 0$.

Kind: 'eig.qevp'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**method**

[{'companion', 'cholesky'}] (default: 'companion') The linearization method.

solver

[dict (default: {'kind': 'eig.scipy', 'method': 'eig'})] The configuration of an eigenvalue solver for the linearized problem $(A - w B) x = 0$.

mode

[{'normal', 'inverted'} (default: 'normal')] Solve either $A - w B$ (normal), or $B - 1/w A$ (inverted).

debug

[bool (default: False)] If True, print debugging information.

name = 'eig.qevp'

`sfepy.solvers.eigen.standard_call(call)`

Decorator handling argument preparation and timing for quadratic eigensolvers.

sfepy.solvers.semismooth_newton module

class sfepy.solvers.semismooth_newton.SemismoothNewton(conf, **kwargs)

The semi-smooth Newton method.

This method is suitable for solving problems of the following structure:

$$\begin{aligned} F(y) &= 0 \\ A(y) &\geq 0, \quad B(y) \geq 0, \quad \langle A(y), B(y) \rangle = 0 \end{aligned}$$

The function $F(y)$ represents the smooth part of the problem.

Regular step: $y \leftarrow y - J(y)^{-1} \Phi(y)$

Steepest descent step: $y \leftarrow y - \beta J(y) \Phi(y)$

Although `fun_smooth_grad()` computes the gradient of the smooth part only, it should return the global matrix, where the non-smooth part is uninitialized, but pre-allocated.

Kind: 'nls.semismooth_newton'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**semismooth**

[bool (default: True)] If True, use the semi-smooth algorithm. Otherwise a non-smooth equation is assumed (use a brute force).

i_max

[int (default: 1)] The maximum number of iterations.

eps_a

[float (default: 1e-10)] The absolute tolerance for the residual, i.e. $\|f(x^i)\|$.

eps_r

[float (default: 1.0)] The relative tolerance for the residual, i.e. $\|f(x^i)\|/\|f(x^0)\|$.

macheps

[float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

lin_red

[float (default: 1.0)] The linear system solution error should be smaller than $(eps_a * lin\_red)$, otherwise a warning is printed.

ls_on

[float (default: 0.99999)] Start the backtracking line-search by reducing the step, if $\|f(x^i)\|/\|f(x^{i-1})\|$ is larger than ls_on .

ls_red

[dict (default: { 'regular': 0.1, 'steepest_descent': 0.01 })] The step reduction factor in case of correct residual assembling for regular and steepest descent modes.

ls_red_warp

[0.0 < float < 1.0 (default: 0.001)] The step reduction factor in case of failed residual assembling (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

ls_min

[0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.

compute_jacobian(*vec_x*, *fun_smooth_grad*, *fun_a_grad*, *fun_b_grad*, *vec_smooth_r*, *vec_a_r*, *vec_b_r*)

name = 'nls.semismooth_newton'

sfePy.solvers.solvers module

Base (abstract) solver classes.

class `sfePy.solvers.solvers.EigenvalueSolver`(*conf*, *mtx_a*=None, *mtx_b*=None, *n_eigs*=None, *eigenvectors*=None, *status*=None, *context*=None, ***kwargs*)

Abstract eigenvalue solver class.

class `sfePy.solvers.solvers.LinearSolver`(*conf*, *mtx*=None, *status*=None, *context*=None, ***kwargs*)

Abstract linear solver class.

clear()

get_tolerance()

Return tuple (*eps_a*, *eps_r*) of absolute and relative tolerance settings. Either value can be *None*, meaning that the solver does not use that setting.

presolve(*mtx*)

class `sfePy.solvers.solvers.NonlinearSolver`(*conf*, *fun*=None, *fun_grad*=None, *lin_solver*=None, *iter_hook*=None, *status*=None, *context*=None, ***kwargs*)

Abstract nonlinear solver class.

class `sfePy.solvers.solvers.OptimizationSolver`(*conf*, *obj_fun*=None, *obj_fun_grad*=None, *status*=None, *obj_args*=None, *context*=None, ***kwargs*)

Abstract optimization solver class.

class `sfePy.solvers.solvers.QuadraticEVPSolver`(*conf*, *mtx_m*=None, *mtx_d*=None, *mtx_k*=None, *n_eigs*=None, *eigenvectors*=None, *status*=None, *context*=None, ***kwargs*)

Abstract quadratic eigenvalue problem solver class.

class `sfePy.solvers.solvers.Solver`(*conf*=None, *context*=None, ***kwargs*)

Base class for all solver kinds. Takes care of processing of common configuration options.

The factory method `any_from_conf()` can be used to create an instance of any subclass.

The subclasses have to reimplement `__init__()` and `__call__()`.

All solvers use the following configuration parameters:

Parameters**name**

[str] The name referred to in problem description options.

kind

[str] The solver kind, as given by the *name* class attribute of the Solver subclasses.

report_status

[bool (default: False)] If True, the solver can report the solution status.

verbose

[bool (default: False)] If True, the solver can print more information about the solution.

static any_from_conf(*conf*, ****kwargs**)

Create an instance of a solver class according to the configuration.

build_solver_kwargs(*conf*)

Build the *kwargs* dict for the underlying solver function using the extra options (marked by '*' in *_parameters*) in *conf*. The declared parameters are omitted.

classmethod process_conf(*conf*, *kwargs*)

Process configuration parameters.

set_field_split(*field_ranges*, ****kwargs**)

class sfepy.solvers.solvers.SolverMeta(*name*, *bases*, *ndict*)

Metaclass for solver classes that automatically adds configuration parameters to the solver class docstring from the *_parameters* class attribute.

class sfepy.solvers.solvers.TimeStepController(*conf*, ****kwargs**)

Abstract time step controller class.

get_initial_dt(*ts*, *vec*, ****kwargs**)

class sfepy.solvers.solvers.TimeSteppingSolver(*conf*, *nls=None*, *status=None*, *tsc=None*, *context=None*, ****kwargs**)

Abstract time stepping solver class.

set_dof_info(*di*)

sfepy.solvers.solvers.format_next(*text*, *new_text*, *pos*, *can_newline*, *width*, *ispaces*)

sfepy.solvers.solvers.make_get_conf(*conf*, *kwargs*)

sfepy.solvers.solvers.make_option_docstring(*name*, *kind*, *default*, *required*, *doc*)

sfepy.solvers.solvers.typeset_to_indent(*txt*, *indent*, *width*)

sfepy.solvers.solvers.use_first_available(*solver_list*, *context=None*, ****kwargs**)

Use the first available solver from *solver_list*.

Parameters**solver_list**

[list of str or Struct] The list of solver names or configuration objects.

context

[object, optional] An optional solver context to pass to the solver.

****kwargs**

[keyword arguments] Additional solver options, see the particular *__init__*() methods.

Returns**out**

[Solver] The first available solver.

sfePy.solvers.ts module**class** `sfePy.solvers.ts.TimeStepper`(*t0, t1, dt=None, n_step=None, step=None, is_quasistatic=False*)

Time stepper class.

advance()**static from_conf**(*conf*)**get_state**()**iter_from**(*step*)**normalize_time**()**restore_step_time**()**set_from_data**(*t0, t1, dt=None, n_step=None, step=None*)**set_from_ts**(*ts, step=None*)**set_state**(*step=0, **kwargs*)**set_step**(*step=0, nt=0.0*)**set_substep_time**(*sub_dt*)**class** `sfePy.solvers.ts.VariableTimeStepper`(*t0, t1, dt=None, n_step=None, step=None, is_quasistatic=False*)

Time stepper class with a variable time step.

advance()**static from_conf**(*conf*)**get_default_time_step**()**get_state**()**iter_from**(*step*)**iter_from_current**()*ts.step, ts.time* is consistent with *step*, time returned here *ts.nt* is normalized time in [0, 1].**set_from_data**(*t0, t1, dt=None, n_step=None, step=None*)**set_from_ts**(*ts, step=None*)**set_n_digit_from_min_dt**(*dt*)**set_state**(*step=0, dt=None, times=None, **kwargs*)**set_step**(*step=0, nt=0.0*)

```
set_time_step(dt, update_time=False)
```

```
sfepy.solvers.ts.get_print_info(n_step)
```

sfepy.solvers.ts_controllers module

Time step controllers.

```
class sfepy.solvers.ts_controllers.ElastodynamicsBasicTSC(conf, **kwargs)
```

Adaptive time step I-controller for elastodynamics.

The implementation is based on [1]. The default parameters correspond to the PID-Controller as implemented in `tsc.ed_pid` with `P=D=0, I=1`.

[1] Grafenhorst, Matthias, Joachim Rang, and Stefan Hartmann. “Time-Adaptive Finite Element Simulations of Dynamical Problems for Temperature-Dependent Materials.” *Journal of Mechanics of Materials and Structures* 12, no. 1 (November 26, 2016): 57–91. <https://doi.org/10.2140/jomms.2017.12.57>.

Kind: ‘tsc.ed_basic’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

eps_r

[list of floats or float] Relative tolerance(s).

eps_a

[list of floats or float] Absolute tolerance(s).

fmin

[float (default: 0.3)] Minimum step size change factor on step rejection.

fmax

[float (default: 2.5)] Maximum step size change factor on step acceptance.

fsafety

[float (default: 0.8)] Step size change safety factor.

error_order

[float (default: 2)] The order of the solver error estimate.

guess_dt0

[bool (default: False)] Guess a good initial step size from initial conditions.

```
get_initial_dt(ts, vec, unpack, **kwargs)
```

Adapted from [1] for second order ODEs.

[1] Hairer, Ernst, Gerhard Wanner, and Syvert P. Nørsett. *Solving Ordinary Differential Equations I: Nons-tiff Problems*. Vol. 8. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer, 1993. <https://doi.org/10.1007/978-3-540-78862-1>.

```
static get_scaled_errors(dt, vec0, vec1, eps_as, eps_rs, unpack)
```

```
name = 'tsc.ed_basic'
```

class sfepy.solvers.ts_controllers.**ElastodynamicsLinearTSC**(*conf*, ***kwargs*)

Adaptive time step controller for elastodynamics and linear problems.

Simple heuristics around *ElastodynamicsBasicTSC* that increases the step size only after a sufficient number of accepted iterations passed and the increase is large enough. In particular:

- Let *new_dt* be the step size proposed by *tsc.ed_basic* and *dt* the current step size.
- If the current step is rejected, the count attribute is reset to zero and *fred * new_dt* is returned.
- If the current step is accepted:
 - If the count is lower than *inc_wait*, it is incremented and *dt* is returned.
 - Otherwise, if $(\text{new\_dt} / \text{dt}) \geq \text{min\_finc}$ (≥ 1), the count is reset to zero and *new_dt* is returned.
 - Else, if $(\text{new\_dt} / \text{dt}) < \text{min\_finc}$, *dt* is returned.

Kind: 'tsc.ed_linear'

For common configuration parameters, see *Solver*.

Specific configuration parameters:

Parameters

eps_r

[list of floats or float] Relative tolerance(s).

eps_a

[list of floats or float] Absolute tolerance(s).

fmin

[float (default: 0.3)] Minimum step size change factor on step rejection.

fmax

[float (default: 2.5)] Maximum step size change factor on step acceptance.

fsafety

[float (default: 0.8)] Step size change safety factor.

error_order

[float (default: 2)] The order of the solver error estimate.

guess_dt0

[bool (default: False)] Guess a good initial step size from initial conditions.

fred

[float (default: 1.0)] Additional step size reduction factor w.r.t. *tsc.ed_basic*.

inc_wait

[int (default: 10)] The number of consecutive accepted steps to wait before increasing the step size.

min_finc

[float ≥ 1 (default: 1.5)] Minimum step size increase factor.

name = 'tsc.ed_linear'

class sfepy.solvers.ts_controllers.**ElastodynamicsPIDTSC**(*conf*, ***kwargs*)

Adaptive time step PID controller for elastodynamics.

The implementation is based on [1], [2] (PI Controller) and [3] (PID). The default parameters correspond to the I-Controller as implemented in *tsc.ed_basic*.

[1] Grafenhorst, Matthias, Joachim Rang, and Stefan Hartmann. “Time-Adaptive Finite Element Simulations of Dynamical Problems for Temperature-Dependent Materials.” *Journal of Mechanics of Materials and Structures* 12, no. 1 (November 26, 2016): 57–91. <https://doi.org/10.2140/jomms.2017.12.57>. [2] Hairer, Ernst, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Science & Business Media, 1993. [3] Söderlind, Gustaf. “Digital Filters in Adaptive Time-Stepping.” *ACM Transactions on Mathematical Software* 29, no. 1 (March 1, 2003): 1–26. <https://doi.org/10.1145/641876.641877>.

Kind: ‘tsc.ed_pid’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

eps_r

[list of floats or float] Relative tolerance(s).

eps_a

[list of floats or float] Absolute tolerance(s).

fmin

[float (default: 0.3)] Minimum step size change factor on step rejection.

fmax

[float (default: 2.5)] Maximum step size change factor on step acceptance.

fsafety

[float (default: 0.8)] Step size change safety factor.

error_order

[float (default: 2)] The order of the solver error estimate.

guess_dt0

[bool (default: False)] Guess a good initial step size from initial conditions.

pcoef

[float (default: 0.0)] Proportional (P) coefficient of the step size control.

icoef

[float (default: 1.0)] Integral (I) coefficient of the step size control.

dcoef

[float (default: 0.0)] Derivative (D) coefficient of the step size control.

name = ‘tsc.ed_pid’

class sfepy.solvers.ts_controllers.**FixedTSC**(*conf*, ***kwargs*)

Fixed (do-nothing) time step controller.

Kind: ‘tsc.fixed’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = ‘tsc.fixed’

class sfepy.solvers.ts_controllers.**TimesSequenceTSC**(*conf*, ***kwargs*)

Given times sequence time step controller.

Kind: 'tsc.time_sequence'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

times

[iterable (default: range(1, 6))] A sequence of times to generate.

get_initial_dt(*ts*, *vec*, ***kwargs*)

name = 'tsc.time_sequence'

sfepy.solvers.ts_controllers.**eval_scaled_norm**(*terr*, *eps_a*, *eps_r*)

sfepy.solvers.ts_solvers module

Time stepping solvers.

class sfepy.solvers.ts_solvers.**AdaptiveTimeSteppingSolver**(*conf*, *nls=None*, *context=None*, ***kwargs*)

Implicit time stepping solver with an adaptive time step.

Either the built-in or user supplied function can be used to adapt the time step.

Kind: 'ts.adaptive'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

t0

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

quasistatic

[bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver is invoked also for the initial time.

adapt_fun

[callable(*ts*, *status*, *adt*, *context*, *verbose*)] If given, use this function to set the time step in *ts*. The function return value is a bool - if True, the adaptivity loop should stop. The other parameters below are collected in *adt*, *status* is the nonlinear solver status, *context* is a user-defined context and *verbose* is a verbosity flag. Solvers created by [Problem](#) use the Problem instance as the context.

dt_red_factor

[float (default: 0.2)] The time step reduction factor.

dt_red_max

[float (default: 0.001)] The maximum time step reduction factor.

dt_inc_factor

[float (default: 1.25)] The time step increase factor.

dt_inc_on_iter

[int (default: 4)] Increase the time step if the nonlinear solver converged in less than this amount of iterations for *dt_inc_wait* consecutive time steps.

dt_inc_wait

[int (default: 5)] The number of consecutive time steps, see *dt_inc_on_iter*.

name = 'ts.adaptive'

output_step_info(*ts*)

solve_step(*ts, nls, vec, prestep_fun*)

Solve a single time step.

class sfepy.solvers.ts_solvers.**BatheTS**(*conf, nls=None, context=None, **kwargs*)

Solve elastodynamics problems by the Bathe method.

The method was introduced in [1].

[1] Klaus-Juergen Bathe, Conserving energy and momentum in nonlinear dynamics: A simple implicit time integration scheme, Computers & Structures, Volume 85, Issues 7-8, 2007, Pages 437-445, ISSN 0045-7949, <https://doi.org/10.1016/j.compstruc.2006.09.004>.

Kind: 'ts.bathe'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**t0**

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys 'u', 'du', 'ddu' and 'extra', and values corresponding to the names of the actual variables. See *var_names* returned from [transform_equations_ed\(\)](#)

```
clear_lin_solver(clear_constant_matrices=True)
```

```
create_nlst1(nls, dt, u0, v0, a0)
```

The first sub-step: the trapezoidal rule.

```
create_nlst2(nls, dt, u0, u1, v0, v1)
```

The second sub-step: the three-point Euler backward method.

```
name = 'ts.bathe'
```

```
step(ts, vec, nls, pack, unpack, prestep_fun)
```

Solve a single time step.

```
class sfepy.solvers.ts_solvers.CentralDifferenceTS(conf, nls=None, tsc=None, context=None,  
                                                    **kwargs)
```

Solve elastodynamics problems by the explicit central difference method.

It is the same method as obtained by using [NewmarkTS](#) with $\beta = 0$, $\gamma = 1/2$, but uses a simpler code.

It is also mathematically equivalent to the [VelocityVerletTS](#) method. The current implementation code is essentially the same.

This solver supports, when used with [RMMSolver](#), the reciprocal mass matrix algorithm, see [MassTerm](#).

Kind: 'ts.central_difference'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

t0

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys 'u', 'du', 'ddu' and 'extra', and values corresponding to the names of the actual variables. See *var_names* returned from [transform_equations_ed\(\)](#)

```
create_nlst(nls, dt, u0, v0, a0)
```

```
name = 'ts.central_difference'
```

step(*ts, vec, nls, pack, unpack, **kwargs*)

Solve a single time step.

class sfepy.solvers.ts_solvers.**ElastodynamicsBaseTS**(*conf, nls=None, tsc=None, context=None, **kwargs*)

Base class for elastodynamics solvers.

Assumes block-diagonal matrix in *u, v, a*.

clear_lin_solver(*clear_constant_matrices=True*)

get_a0(*nls, u0, e0, v0, unpack*)

get_initial_vec(*nls, vec0, init_fun, prestep_fun, poststep_fun*)

get_matrices(*nls, vec, unpack=None*)

class sfepy.solvers.ts_solvers.**GeneralizedAlphaTS**(*conf, nls=None, tsc=None, context=None, **kwargs*)

Solve elastodynamics problems by the generalized α method.

- The method was introduced in [1].
- The method is unconditionally stable provided $\alpha_m \leq \alpha_f \leq \frac{1}{2}$, $\beta \geq \frac{1}{4} + \frac{1}{2}(\alpha_f - \alpha_m)$.
- The method is second-order accurate provided $\gamma = \frac{1}{2} - \alpha_m + \alpha_f$. This is used when *gamma* is *None*.
- High frequency dissipation is maximized for $\beta = \frac{1}{4}(1 - \alpha_m + \alpha_f)^2$. This is used when *beta* is *None*.
- The default values of α_m , α_f (if *alpha_m* or *alpha_f* are *None*) are based on the user specified high-frequency dissipation parameter *rho_inf*.

Special settings:

- $\alpha_m = 0$ corresponds to the HHT- α method.
- $\alpha_f = 0$ corresponds to the WBZ- α method.
- $\alpha_m = 0$, $\alpha_f = 0$ produces the Newmark method.

[1] J. Chung, G.M.Hubert. “A Time Integration Algorithm for Structural Dynamics with Improved Numerical Dissipation: The Generalized- α Method” ASME Journal of Applied Mechanics, 60, 371:375, 1993.

Kind: ‘ts.generalized_alpha’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

rho_inf

[float (default: 0.5)] The spectral radius in the high frequency limit (user specified high-frequency dissipation) in [0, 1]: 1 = no dissipation, 0 = asymptotic annihilation.

alpha_m

[float] The parameter α_m .

alpha_f

[float] The parameter α_f .

beta

[float] The Newmark-like parameter β .

gamma

[float] The Newmark-like parameter γ .

t0

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys 'u', 'du', 'ddu' and 'extra', and values corresponding to the names of the actual variables. See *var_names* returned from [transform_equations_ed\(\)](#)

create_nlst(*nls*, *dt*, *alpha_m*, *alpha_f*, *gamma*, *beta*, *u0*, *v0*, *a0*)

name = 'ts.generalized_alpha'

step(*ts*, *vec*, *nls*, *pack*, *unpack*, ***kwargs*)

Solve a single time step.

class `sfepy.solvers.ts_solvers.NewmarkTS`(*conf*, *nls=None*, *tsc=None*, *context=None*, ***kwargs*)

Solve elastodynamics problems by the Newmark method.

The method was introduced in [1]. Common settings [2]:

name	kind	beta	gamma	Omega_crit
trapezoidal rule:	implicit	1/4	1/2	unconditional
linear acceleration:	implicit	1/6	1/2	$2\sqrt{3}$
Fox-Goodwin:	implicit	1/12	1/2	$\sqrt{6}$
central difference:	explicit	0	1/2	2

All of these methods are 2-order of accuracy.

[1] Newmark, N. M. (1959) A method of computation for structural dynamics. Journal of Engineering Mechanics, ASCE, 85 (EM3) 67-94.

[2] Arnaud Delaplace, David Ryckelynck: Solvers for Computational Mechanics

Kind: 'ts.newmark'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

beta

[float (default: 0.25)] The Newmark method parameter beta.

gamma

[float (default: 0.5)] The Newmark method parameter gamma.

t0

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys 'u', 'du', 'ddu' and 'extra', and values corresponding to the names of the actual variables. See *var_names* returned from [transform_equations_ed\(\)](#)

create_nlst(*nls*, *dt*, *gamma*, *beta*, *u0*, *e0*, *v0*, *a0*, *pack*, *unpack*)

extra_variables = True

name = 'ts.newmark'

step(*ts*, *vec*, *nls*, *pack*, *unpack*, ***kwargs*)

Solve a single time step.

class sfepy.solvers.ts_solvers.**SimpleTimeSteppingSolver**(*conf*, *nls=None*, *context=None*, ***kwargs*)

Implicit time stepping solver with a fixed time step.

Kind: 'ts.simple'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**t0**

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

quasistatic

[bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver is invoked also for the initial time.

name = 'ts.simple'

output_step_info(*ts*)

solve_step(*ts, nls, vec, prestep_fun=None*)

solve_step0(*nls, vec0*)

class sfepy.solvers.ts_solvers.**StationarySolver**(*conf, nls=None, context=None, **kwargs*)

Solver for stationary problems without time stepping.

This class is provided to have a unified interface of the time stepping solvers also for stationary problems.

Kind: 'ts.stationary'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters

name = 'ts.stationary'

class sfepy.solvers.ts_solvers.**VelocityVerletTS**(*conf, nls=None, tsc=None, context=None, **kwargs*)

Solve elastodynamics problems by the explicit velocity-Verlet method.

The algorithm can be found in [1].

It is mathematically equivalent to the [CentralDifferenceTS](#) method. The current implementation code is essentially the same, as the mid-time velocities are not used for anything other than computing the new time velocities.

Kind: 'ts.velocity_verlet'

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

Parameters**t0**

[float (default: 0.0)] The initial time.

t1

[float (default: 1.0)] The final time.

dt

[float] The time step. Used if *n_step* is not given.

n_step

[int (default: 10)] The number of time steps. Has precedence over *dt*.

is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys 'u', 'du', 'ddu' and 'extra', and values corresponding to the names of the actual variables. See `var_names` returned from `transform_equations_ed()`

create_nlst(*nls*, *dt*, *u0*, *v0*, *a0*)

name = 'ts.velocity_verlet'

step(*ts*, *vec*, *nls*, *pack*, *unpack*, ***kwargs*)

Solve a single time step.

`sfepy.solvers.ts_solvers.adapt_time_step`(*ts*, *status*, *adt*, *context=None*, *verbose=False*)

Adapt the time step of *ts* according to the exit status of the nonlinear solver.

The time step *dt* is reduced, if the nonlinear solver did not converge. If it converged in less then a specified number of iterations for several time steps, the time step is increased. This is governed by the following parameters:

- *red_factor* : time step reduction factor
- *red_max* : maximum time step reduction factor
- *inc_factor* : time step increase factor
- *inc_on_iter* : increase time step if the nonlinear solver converged in less than this amount of iterations...
- *inc_wait* : ...for this number of consecutive time steps

Parameters**ts**

[VariableTimeStepper instance] The time stepper.

status

[IndexedStruct instance] The nonlinear solver exit status.

adt

[Struct instance] The object with the adaptivity parameters of the time-stepping solver such as *red_factor* (see above) as attributes.

context

[object, optional] The context can be used in user-defined adaptivity functions. Not used here.

Returns**is_break**

[bool] If True, the adaptivity loop should stop.

`sfepy.solvers.ts_solvers.gen_multi_vec_packing`(*di*, *names*, *extra_variables=False*)

Return DOF vector (un)packing functions for nlst. For multiphysical problems (non-empty *ie* slice for extra variables) the *unpack()* function accepts an additional argument *mode* that can be set to 'full' or 'nls'.

The following DOF ordering must be obeyed:

- The full DOF vector:

```
---iue---|-iv-|-ia-
-iu-|-ie-|
```

`sfepy.solvers.ts_solvers.get_min_dt(adt)`

`sfepy.solvers.ts_solvers.standard_ts_call(call)`

Decorator handling argument preparation and timing for time-stepping solvers.

`sfepy.solvers.ts_solvers.transform_equations_ed(equations, materials)`

Transform equations and variables for *ElastodynamicsBaseTS*-based time stepping solvers. The displacement variable name is automatically detected by seeking the second time derivative, i.e. the ‘dd’ prefix in variable names.

sfepy.terms package

sfepy.terms.terms module

`class sfepy.terms.terms.ConnInfo(**kwargs)`

`get_region(can_trace=True)`

`get_region_name(can_trace=True)`

`class sfepy.terms.terms.Term(name, arg_str, integral, region, **kwargs)`

`advance(ts)`

Advance to the next time step. Implemented in subclasses.

`arg_geometry_types = {}`

`arg_shapes = {}`

`arg_types = {}`

`assemble_to(asm_obj, val, iels, mode='vector', diff_var=None)`

Assemble the results of term evaluation.

For standard terms, assemble the values in *val* corresponding to elements/cells *iels* into a vector or a CSR sparse matrix *asm_obj*, depending on *mode*.

For terms with a dynamic connectivity (e.g. contact terms), in ‘*matrix*’ mode, return the extra COO sparse matrix instead. The extra matrix has to be added to the global matrix by the caller. By default, this is done in *Equations.evaluate()*.

`assign_args(variables, materials, user=None)`

Check term argument existence in variables, materials, user data and assign the arguments to terms. Also check compatibility of field and term regions.

`call_function(out, fargs)`

`call_get_fargs(args, kwargs)`

`check_args()`

Common checking to all terms.

Check compatibility of field and term regions.

`check_shapes(*args, **kwargs)`

Check term argument shapes at run-time.

classify_args()

Classify types of the term arguments and find matching call signature.

A state variable can be in place of a parameter variable and vice versa.

diff_info = {}

eval_complex(*shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs*)

eval_real(*shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs*)

evaluate(*mode='eval', diff_var=None, standalone=True, ret_status=False, **kwargs*)

Evaluate the term.

Parameters**mode**

['eval' (default), or 'weak'] The term evaluation mode.

Returns**val**

[float or array] In 'eval' mode, the term returns a single value (the integral, it does not need to be a scalar), while in 'weak' mode it returns an array for each element.

status

[int, optional] The flag indicating evaluation success (0) or failure (nonzero). Only provided if *ret_status* is True.

iels

[array of ints, optional] The local elements indices in 'weak' mode. Only provided in non-'eval' modes.

static from_desc(*constructor, desc, region, integrals=None*)

geometries = ['1_2', '2_3', '2_4', '3_4', '3_8']

get(*variable, quantity_name, bf=None, integration=None, step=None, time_derivative=None*)

Get the named quantity related to the variable.

Notes

This is a convenience wrapper of `Variable.evaluate()` that initializes the arguments using the term data.

get_arg_name(*arg_type, full=False, join=None*)

Get the name of the argument specified by *arg_type*.

Parameters**arg_type**

[str] The argument type string.

full

[bool] If True, return the full name. For example, if the name of a variable argument is 'u' and its time derivative is requested, the full name is 'du/dt'.

join

[str, optional] Optionally, the material argument name tuple can be joined to a single string using the *join* string.

Returns

name

[str] The argument name.

get_args(*arg_types=None, **kwargs*)

Return arguments by type as specified in *arg_types* (or *self.ats*). Arguments in ****kwargs** can override the ones assigned at the term construction - this is useful for passing user data.

get_args_by_name(*arg_names*)

Return arguments by name.

get_assembling_cells(*shape=None*)

Return the assembling cell indices into a DOF connectivity.

get_conn_info()

get_conn_key()

The key to be used in DOF connectivity information.

get_data_shape(*variable*)

Get data shape information from variable.

Notes

This is a convenience wrapper of `FieldVariable.get_data_shape()` that initializes the arguments using the term data.

get_dof_conn_type(*var_name*)

get_kwargs(*keys, **kwargs*)

Extract arguments from ****kwargs** listed in *keys* (default is `None`).

get_mapping(*variable, get_saved=False, return_key=False*)

Get the reference mapping from a variable.

Notes

This is a convenience wrapper of `Field.get_mapping()` that initializes the arguments using the term data.

get_material_names(*part=0*)

get_materials(*join=False*)

get_parameter_names()

get_parameter_variables()

get_physical_qps()

Get physical quadrature points corresponding to the term region and integral.

get_qp_key()

Return a key identifying uniquely the term quadrature points.

get_region()

get_state_names()

If variables are given, return only true unknowns whose data are of the current time step (0).

get_state_variables(*unknown_only=False*)

get_str()

get_user_names()

get_variable_names()

get_variables(*as_list=True*)

get_vector(*variable*)

Get the vector stored in *variable* according to self.arg_steps and self.arg_derivatives. Supports only the backward difference w.r.t. time.

get_virtual_name()

get_virtual_variable()

integration = 'cell'

integration_order = None

name = ''

static new(*name_args, integral, region, **kwargs*)

Create a new Term instance.

Parameters

name_args

[str] The term name and arguments, e.g. 'dw_laplace(m.coef, v, u)'.

integral

[Integral instance] The integral defining the term quadrature.

region

[Region instance] The term region (domain of integration for most terms).

****kwargs**

[keyword arguments, optional] The actual term arguments. Their compatibility with *name_args* is checked when a term is added to an Equation instance, or when evaluated.

set_arg_types()

set_integral(*integral*)

Set the term integral.

setup(*allow_derivatives=False*)

setup_args(***kwargs*)

setup_formal_args(*allow_derivatives=False*)

setup_geometry_types()

setup_integration()

standalone_setup()

static tile_mat(*mat, nel*)

time_update(*ts*)

static translate_fargs_mapping(*function, fargs, force=False*)

class sfepy.terms.terms.**Terms**(*objs=None*)

append(*obj*)

assign_args(*variables, materials, user=None*)

Assign all term arguments.

static from_desc(*term_descs, regions, integrals=None*)

Create terms, assign each term its region.

get_material_names()

get_user_names()

get_variable_names()

insert(*ii, obj*)

setup(***kwargs*)

update_expression()

sfepy.terms.terms.**create_arg_parser**(*allow_derivatives=False*)

sfepy.terms.terms.**get_arg_kinds**(*arg_types*)

Translate *arg_types* of a Term to a canonical form.

Parameters

arg_types

[tuple of strings] The term argument types, as given in the *arg_types* attribute.

Returns

arg_kinds

[list of strings] The argument kinds - one of 'virtual_variable', 'state_variable', 'parameter_variable', 'opt_material', 'ts', 'user'.

sfepy.terms.terms.**get_shape_kind**(*integration*)

Get data shape kind for given integration type.

sfepy.terms.terms.**split_complex_args**(*args*)

Split complex arguments to real and imaginary parts.

Returns

newargs

[dictionary] Dictionary with lists corresponding to *args* such that each argument of numpy.complex128 data type is split to its real and imaginary part. The output depends on the number of complex arguments in 'args':

- 0: list (key 'r') identical to input one
- 1: two lists with keys 'r', 'i' corresponding to real and imaginary parts
- 2: output dictionary contains four lists:
 - 'r' - real(arg1), real(arg2)

- 'i' - $\text{imag}(\text{arg1}), \text{imag}(\text{arg2})$
- 'ri' - $\text{real}(\text{arg1}), \text{imag}(\text{arg2})$
- 'ir' - $\text{imag}(\text{arg1}), \text{real}(\text{arg2})$

`sfepy.terms.terms_adj_navier_stokes` module

class `sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term`(*name, arg_str, integral, region,*
***kwargs*)

The first adjoint term to nonlinear convective term $dw_{convect}$.

Definition

$$\int_{\Omega} ((\underline{v} \cdot \nabla) \underline{u}) \cdot \underline{w}$$

Call signature

dw_adj_convect1 (virtual, state, parameter)

Arguments

- virtual : $\underline{v}$
- state : $\underline{w}$
- parameter : $\underline{u}$

arg_shapes = {'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('virtual', 'state', 'parameter')

static function(*out, state_w, grad_u, cmap, is_diff*)

get_fargs(*virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs*)

name = 'dw_adj_convect1'

class `sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term`(*name, arg_str, integral, region,*
***kwargs*)

The second adjoint term to nonlinear convective term $dw_{convect}$.

Definition

$$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{v}) \cdot \underline{w}$$

Call signature

dw_adj_convect2 (virtual, state, parameter)

Arguments

- virtual : $\underline{v}$
- state : $\underline{w}$
- parameter : $\underline{u}$

```
arg_shapes = {'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('virtual', 'state', 'parameter')
static function(out, state_w, state_u, cmap, is_diff)
get_fargs(virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_convect2'

class sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm(name, arg_str, integral, region,
                                                         **kwargs)
```

Gateaux differential of $\Psi(\underline{u}) = \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}$ w.r.t. $\underline{u}$ in the direction $\underline{v}$ or adjoint term to *dw_div_grad*.

Definition

$$w \delta_u \Psi(\underline{u}) \circ \underline{v}$$

Call signature

dw_adj_div_grad (material_1, material_2, virtual, parameter)

Arguments

- material_1 : w (weight)
- material_2 : ν (viscosity)
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'parameter': 'D',
              'virtual': ('D', None)}
arg_types = ('material_1', 'material_2', 'virtual', 'parameter')
static function(out, grad, viscosity, cmap_v, cmap_s, is_diff)
get_fargs(mat1, mat2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_div_grad'

class sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm(name, arg_str, integral, region,
                                                         **kwargs)
```

Call signature

d_of_ns_min_grad (material_1, material_2, parameter)

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'parameter': 1}
arg_types = ('material_1', 'material_2', 'parameter')
static function(out, grad, viscosity, cmap)
get_eval_shape(weight, mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```



```
get_fargs(weight, mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'd_of_ns_min_grad'
```

```
class sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressDiffTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Gateaux differential of $\Psi(p)$ w.r.t. p in the direction q .

Definition

$$w\delta_p\Psi(p) \circ q$$

Call signature

```
dw_of_ns_surf_min_d_press_diff (material, virtual)
```

Arguments

- material : w (weight)
- virtual : q

```
arg_shapes = {'material': 1, 'virtual': (1, None)}
```

```
arg_types = ('material', 'virtual')
```

```
get_fargs(weight, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_of_ns_surf_min_d_press_diff'
```

```
class sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm(name, arg_str, integral, region,
                                                                **kwargs)
```

Sensitivity of $\Psi(p)$.

Definition

$$\delta\Psi(p) = \delta \left(\int_{\Gamma_{in}} p - \int_{\Gamma_{out}} b_{press} \right)$$

Call signature

```
ev_of_ns_surf_min_d_press (material_1, material_2, parameter)
```

Arguments

- material_1 : w (weight)
- material_2 : b_{press} (given pressure)
- parameter : p

```
arg_shapes = {'material_1': 1, 'material_2': 1, 'parameter': 1}
```

```
arg_types = ('material_1', 'material_2', 'parameter')
```

```
static function(out, pressure, weight, bpress, cmap, is_diff)
```

```
get_eval_shape(weight, bpress, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(weight, bpress, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'ev_of_ns_surf_min_d_press'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDConvectTerm(name, arg_str, integral, region, **kwargs)
```

Sensitivity (shape derivative) of convective term $dw_{convect}$.

Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition

$$\int_{\Omega} [u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \mathcal{V}) - u_k \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i]$$

Call signature

ev_sd_convect (parameter_u, parameter_w, parameter_mv)

Arguments

- parameter_u : $\underline{u}$
- parameter_w : $\underline{w}$
- parameter_mv : $\mathcal{V}$

```
arg_shapes = {'parameter_mv': 'D', 'parameter_u': 'D', 'parameter_w': 'D'}
```

```
arg_types = ('parameter_u', 'parameter_w', 'parameter_mv')
```

```
static function(out, state_u, grad_u, state_w, div_mv, grad_mv, cmap_u, mode)
```

```
get_eval_shape(par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_convect'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm(name, arg_str, integral, region, **kwargs)
```

Sensitivity (shape derivative) of diffusion term dw_{div_grad} .

Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition

$$\int_{\Omega} \hat{I} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \hat{I} \nabla \underline{v} : \nabla \underline{u}$$

$$\hat{I}_{ijkl} = \delta_{ik} \delta_{jl} \nabla \cdot \mathcal{V} - \delta_{ik} \delta_{js} \frac{\partial \mathcal{V}_l}{\partial x_s} - \delta_{is} \delta_{jl} \frac{\partial \mathcal{V}_k}{\partial x_s}$$

Call signature

ev_sd_div_grad (opt_material, parameter_u, parameter_w, parameter_mv)

Arguments

- material : ν (viscosity, optional)

- parameter_u : $\underline{u}$
- parameter_w : $\underline{w}$
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_mv': 'D', 'parameter_u': 'D',
'parameter_w': 'D'}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter_u', 'parameter_w', 'parameter_mv')
```

```
static function(out, grad_u, grad_w, div_mv, grad_mv, viscosity, cmap_u, mode)
```

```
get_eval_shape(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_div_grad'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDDivTerm(name, arg_str, integral, region, **kwargs)
```

Sensitivity (shape derivative) of Stokes term dw_{stokes} in ‘div’ mode.

Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition

$$\int_{\Omega} p[(\nabla \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}}) - \frac{\partial \mathcal{V}_k}{\partial x_i} \frac{\partial w_i}{\partial x_k}]$$

Call signature

```
ev_sd_div (parameter_u, parameter_p, parameter_mv)
```

Arguments

- parameter_u : $\underline{u}$
- parameter_p : p
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = {'parameter_mv': 'D', 'parameter_p': 1, 'parameter_u': 'D'}
```

```
arg_types = ('parameter_u', 'parameter_p', 'parameter_mv')
```

```
static function(out, div_u, grad_u, state_p, div_mv, grad_mv, cmap_u, mode)
```

```
get_eval_shape(par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_div'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDDotTerm(name, arg_str, integral, region, **kwargs)
```

Sensitivity (shape derivative) of dot product of scalars or vectors.

Definition

$$\int_{\Omega} pq(\nabla \cdot \underline{\mathcal{V}}), \int_{\Omega} (\underline{u} \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}})$$

Call signature

ev_sd_dot (parameter_1, parameter_2, parameter_mv)

Arguments

- parameter_1 : p or $\underline{u}$
- parameter_2 : q or $\underline{w}$
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = [{'parameter_1': 'D', 'parameter_2': 'D', 'parameter_mv': 'D'},
{'parameter_1': 1, 'parameter_2': 1}]
```

```
arg_types = ('parameter_1', 'parameter_2', 'parameter_mv')
```

```
static function(out, state_p, state_q, div_mv, cmap, mode)
```

```
get_eval_shape(par1, par2, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(par1, par2, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_dot'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Sensitivity (shape derivative) of stabilization term $dw_{st\_grad\_div}$.

Definition

$$\gamma \int_{\Omega} [(\nabla \cdot \underline{u})(\nabla \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot \underline{w}) - (\nabla \cdot \underline{u}) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i}]$$

Call signature

ev_sd_st_grad_div (material, parameter_u, parameter_w, parameter_mv)

Arguments

- material : γ
- parameter_u : $\underline{u}$
- parameter_w : $\underline{w}$
- parameter_mv : $\underline{\mathcal{V}}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_mv': 'D', 'parameter_u': 'D',
'parameter_w': 'D'}
```

```
arg_types = ('material', 'parameter_u', 'parameter_w', 'parameter_mv')
```

```
static function(out, div_u, grad_u, div_w, grad_w, div_mv, grad_mv, coef, cmap_u, mode)
```

```
get_eval_shape(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_st_grad_div'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Sensitivity (shape derivative) of stabilization terms $dw\_st\_supg\_p$ or $dw\_st\_pspg\_c$.

Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \left[\frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i) (\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k) \frac{\partial u_i}{\partial x_k} \right]$$

Call signature

```
ev_sd_st_pspg_c (material, parameter_b, parameter_u, parameter_r, parameter_mv)
```

Arguments

- material : δ_K
- parameter_b : $\underline{b}$
- parameter_u : $\underline{u}$
- parameter_r : r
- parameter_mv : $\mathcal{V}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_b': 'D', 'parameter_mv': 'D',
              'parameter_r': 1, 'parameter_u': 'D'}
```

```
arg_types = ('material', 'parameter_b', 'parameter_u', 'parameter_r',
             'parameter_mv')
```

```
static function(out, state_b, grad_u, grad_r, div_mv, grad_mv, coef, cmap_u, mode)
```

```
get_eval_shape(mat, par_b, par_u, par_r, par_mv, mode=None, term_mode=None, diff_var=None,
               **kwargs)
```

```
get_fargs(mat, par_b, par_u, par_r, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_st_pspg_c'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDPSPGPStabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Sensitivity (shape derivative) of stabilization term $dw\_st\_pspg\_p$.

Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \left[(\nabla r \cdot \nabla p) (\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} (\nabla \mathcal{V}_k \cdot \nabla p) - (\nabla r \cdot \nabla \mathcal{V}_k) \frac{\partial p}{\partial x_k} \right]$$

Call signature

```
ev_sd_st_pspg_p (material, parameter_r, parameter_p, parameter_mv)
```

Arguments

- material : τ_K
- parameter_r : r
- parameter_p : p
- parameter_mv : $\underline{\mathcal{V}}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_mv': 'D', 'parameter_p': 1,
              'parameter_r': 1}
```

```
arg_types = ('material', 'parameter_r', 'parameter_p', 'parameter_mv')
```

```
static function(out, grad_r, grad_p, div_mv, grad_mv, coef, cmap_p, mode)
```

```
get_eval_shape(mat, par_r, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, par_r, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_st_pspg_p'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Sensitivity (shape derivative) of stabilization term $dw_{st\_supg\_c}$.

Definition

$$\sum_{K \in \mathcal{T}_n} \int_{T_K} \delta_K [(\underline{b} \cdot \nabla u_k)(\underline{b} \cdot \nabla w_k)(\nabla \cdot \mathcal{V}) - (\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial u_k}{\partial x_i} (\underline{b} \cdot \nabla w_k) - (\underline{u} \cdot \nabla u_k)(\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial w_k}{\partial x_i}]$$

Call signature

ev_sd_st_supg_c (material, parameter_b, parameter_u, parameter_w, parameter_mv)

Arguments

- material : δ_K
- parameter_b : $\underline{b}$
- parameter_u : $\underline{u}$
- parameter_w : $\underline{w}$
- parameter_mv : $\underline{\mathcal{V}}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_b': 'D', 'parameter_mv': 'D',
              'parameter_u': 'D', 'parameter_w': 'D'}
```

```
arg_types = ('material', 'parameter_b', 'parameter_u', 'parameter_w',
              'parameter_mv')
```

```
static function(out, state_b, grad_u, grad_w, div_mv, grad_mv, coef, cmap_u, mode)
```

```
get_eval_shape(mat, par_b, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None,
               **kwargs)
```

```
get_fargs(mat, par_b, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_st_supg_c'
```

```
class sfepy.terms.terms_adj_navier_stokes.SUPGAdjStabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Adjoint term to SUPG stabilization term $dw_{st\_supg\_c}$.

Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [((\underline{v} \cdot \nabla) \underline{u})((\underline{u} \cdot \nabla) \underline{w}) + ((\underline{u} \cdot \nabla) \underline{u})((\underline{v} \cdot \nabla) \underline{w})]$$

Call signature

```
dw_st_adj_supg_c (material, virtual, parameter, state)
```

Arguments

- material : δ_K
- virtual : $\underline{v}$
- state : $\underline{w}$
- parameter : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'virtual':
('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function(out, state_w, state_u, grad_u, coef, cmap, conn, is_diff)
```

```
get_fargs(mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj_supg_c'
```

```
class sfepy.terms.terms_adj_navier_stokes.SUPGAdj1StabilizationTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

The first adjoint term to SUPG stabilization term $dw_{st\_supg\_p}$.

Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p(\underline{v} \cdot \nabla \underline{w})$$

Call signature

```
dw_st_adj1_supg_p (material, virtual, state, parameter)
```

Arguments

- material : δ_K
- virtual : $\underline{v}$
- state : $\underline{w}$

- `parameter :  $p$`

```
arg_shapes = {'material': '1, 1', 'parameter': 1, 'state': 'D', 'virtual': ('D',  
'state')}
```

```
arg_types = ('material', 'virtual', 'state', 'parameter')
```

```
static function(out, state_w, grad_p, coef, cmap_w, conn_w, is_diff)
```

```
get_fargs(mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj1_supg_p'
```

```
class sfepy.terms.terms_adj_navier_stokes.SUPGAdj2StabilizationTerm(name, arg_str, integral,  
                                                                    region, **kwargs)
```

The second adjoint term to SUPG stabilization term $dw\_st\_supg\_p$ as well as adjoint term to PSPG stabilization term $dw\_st\_pspg\_c$.

Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \nabla r(\underline{v} \cdot \nabla \underline{u})$$

Call signature

<code>dw_st_adj2_supg_p</code> (material, virtual, parameter, state)

Arguments

- `material :  $\tau_K$`
- `virtual :  $\underline{v}$`
- `parameter :  $\underline{u}$`
- `state :  $r$`

```
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 1, 'virtual': ('D',  
'state')}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function(out, grad_u, state_r, coef, cmap_u, cmap_r, conn_r, is_diff)
```

```
get_fargs(mat, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj2_supg_p'
```

```
sfepy.terms.terms_adj_navier_stokes.grad_as_vector(grad)
```


sfepy.terms.terms_basic module

class sfepy.terms.terms_basic.**IntegrateMatTerm**(*name, arg_str, integral, region, **kwargs*)

Evaluate material parameter m in a volume region.

Depending on evaluation mode, integrate a material parameter over a volume region ('eval'), average it in elements ('el_avg') or interpolate it into volume quadrature points ('qp').

Uses reference mapping of y variable.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$\int_{\mathcal{D}} c$$

Call signature

ev_integrate_mat (material, parameter)

Arguments

- material : c (can have up to two dimensions)
- parameter : y

arg_shapes = [{'material': 'N, N', 'parameter': 'N'}]

arg_types = ('material', 'parameter')

static function(*out, mat, geo, fmode*)

get_eval_shape(*mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs*)

get_fargs(*mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs*)

integration = ('cell', 'facet')

name = 'ev_integrate_mat'

class sfepy.terms.terms_basic.**IntegrateOperatorTerm**(*name, arg_str, integral, region, **kwargs*)

Integral of a test function weighted by a scalar function c .

Definition

$$\int_{\mathcal{D}} q \text{ or } \int_{\mathcal{D}} cq$$

Call signature

dw_integrate (opt_material, virtual)

Arguments

- material : c (optional)
- virtual : q

```
arg_shapes = [{'opt_material': '1, 1', 'virtual': (1, None)}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual')
```

```
static function(out, material, bf, geo)
```

```
get_fargs(material, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
name = 'dw_integrate'
```

```
class sfepy.terms.terms_basic.IntegrateTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate (weighted) variable in a region.

Depending on evaluation mode, integrate a variable over a region ('eval'), average it in elements ('el_avg') or interpolate it into quadrature points ('qp'). For a surface region and vector variables, setting *term_mode* to 'flux' leads to computing corresponding fluxes for the three modes instead.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$\int_{\mathcal{D}} y, \int_{\mathcal{D}} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n}$$
$$\int_{\mathcal{D}} cy, \int_{\mathcal{D}} c\underline{y}, \int_{\Gamma} c\underline{y} \cdot \underline{n} \text{ flux}$$

Call signature

ev_integrate (opt_material, parameter)

Arguments

- material : *c* (optional)
- parameter : *y* or *y*

```
arg_shapes = [{'opt_material': '1, 1', 'parameter': 'N'}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter')
```

```
static function(out, val_qp, vg, fmode)
```

```
get_eval_shape(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
name = 'ev_integrate'
```

```
class sfepy.terms.terms_basic.SumNodalValuesTerm(name, arg_str, integral, region, **kwargs)
```

Sum nodal values.

Call signature

ev_sum_vals (parameter)

Arguments

- `parameter` : p or $\underline{u}$

```
arg_shapes = {'parameter': 'N'}
```

```
arg_types = ('parameter',)
```

```
static function(out, vec)
```

```
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sum_vals'
```

```
class sfepy.terms.terms_basic.SurfaceMomentTerm(name, arg_str, integral, region, **kwargs)
```

Surface integral of the outer product of the unit outward normal $\underline{n}$ and the coordinate $\underline{x}$ shifted by $\underline{x}_0$

Definition

$$\int_{\Gamma} \underline{n}(\underline{x} - \underline{x}_0)$$

Call signature

ev_surface_moment (material, parameter)

Arguments

- `material` : $\underline{x}_0$ (special)
- `parameter` : any variable

```
arg_shapes = {'material': '.: D', 'parameter': 'N'}
```

```
arg_types = ('material', 'parameter')
```

```
static function(out, in_, cmap, conn)
```

```
get_eval_shape(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'ev_surface_moment'
```

```
class sfepy.terms.terms_basic.VolumeSurfaceTerm(name, arg_str, integral, region, **kwargs)
```

Volume of a D -dimensional domain, using a surface integral. Uses approximation of the parameter variable.

Definition

$$1/D \int_{\Gamma} \underline{x} \cdot \underline{n}$$

Call signature

ev_volume_surface (parameter)

Arguments

- parameter : any variable

```
arg_shapes = {'parameter': 'N'}
```

```
arg_types = ('parameter',)
```

```
static function(out, in_, cmap, conn)
```

```
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'ev_volume_surface'
```

```
class sfepy.terms.terms_basic.VolumeTerm(name, arg_str, integral, region, **kwargs)
```

Volume or surface of a domain. Uses approximation of the parameter variable.

Definition

$$\int_{\mathcal{D}} 1$$

Call signature

ev_volume (parameter)

Arguments

- parameter : any variable

```
arg_shapes = [{'parameter': 'N'}]
```

```
arg_types = ('parameter',)
```

```
static function(out, geo)
```

```
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
name = 'ev_volume'
```

```
class sfepy.terms.terms_basic.ZeroTerm(name, arg_str, integral, region, **kwargs)
```

A do-nothing term useful for introducing additional variables into the equations.

Definition

$$0$$

Call signature

dw_zero (virtual, state)

Arguments

- virtual : q or $\underline{v}$
- state : p or $\underline{u}$

```
arg_shapes = {'state': 'N', 'virtual': ('N', None)}
```

```
arg_types = ('virtual', 'state')
```

```
static function(out)
```

```
get_fargs(vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_zero'
```

sfepy.terms.terms_biot module

```
class sfepy.terms.terms_biot.BiotETHTerm(name, arg_str, integral, region, **kwargs)
```

This term has the same definition as `dw_biot_th`, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \alpha_{ij}(t - \tau) p(\tau) d\tau \right] e_{ij}(\underline{v}),$$

$$\int_{\Omega} \left[\int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$$

Call signature

dw_biot_eth	(ts, material_0, material_1, virtual, state)
	(ts, material_0, material_1, state, virtual)

Arguments 1

- ts : TimeStepper instance
- material_0 : $\alpha_{ij}(0)$
- material_1 : $\exp(-\lambda \Delta t)$ (decay at t_1)
- virtual : $\underline{v}$
- state : p

Arguments 2

- ts : TimeStepper instance
- material_0 : $\alpha_{ij}(0)$
- material_1 : $\exp(-\lambda \Delta t)$ (decay at t_1)
- state : $\underline{u}$
- virtual : q

```
arg_shapes = {'material_0': 'S, 1', 'material_1': '1, 1', 'state/div': 'D',
               'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None)}
```

```
arg_types = (('ts', 'material_0', 'material_1', 'virtual', 'state'), ('ts',  
'material_0', 'material_1', 'state', 'virtual'))
```

```
get_fargs(ts, mat0, mat1, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div')
```

```
name = 'dw_biot_eth'
```

```
class sfepy.terms.terms_biot.BiotStressTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate Biot stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$-\int_{\Omega} \alpha_{ij} p$$

Call signature

ev_biot_stress (material, parameter)

Arguments

- material : α_{ij}
- parameter : p

```
arg_shapes = {'material': 'S, 1', 'parameter': 1}
```

```
arg_types = ('material', 'parameter')
```

```
static function(out, val_qp, mat, vg, fmode)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'cell'
```

```
name = 'ev_biot_stress'
```

```
class sfepy.terms.terms_biot.BiotTHTerm(name, arg_str, integral, region, **kwargs)
```

Fading memory Biot term. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \alpha_{ij}(t - \tau) p(\tau) d\tau \right] e_{ij}(\underline{v}) , \\ \int_{\Omega} \left[\int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$$

Call signature

dw_biot_th (ts, material, virtual, state)
(ts, material, state, virtual)

Arguments 1

- `ts` : `TimeStepper` instance
- `material` : $\alpha_{ij}(\tau)$
- `virtual` : $\underline{v}$
- `state` : p

Arguments 2

- `ts` : `TimeStepper` instance
- `material` : $\alpha_{ij}(\tau)$
- `state` : $\underline{u}$
- `virtual` : q

```
arg_shapes = {'material': '.: N, S, 1', 'state/div': 'D', 'state/grad': 1,
'virtual/div': (1, None), 'virtual/grad': ('D', None)}
```

```
arg_types = (('ts', 'material', 'virtual', 'state'), ('ts', 'material', 'state',
'virtual'))
```

```
get_fargs(ts, mats, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div')
```

```
name = 'dw_biot_th'
```

```
class sfepy.terms.terms_biot.BiotTerm(name, arg_str, integral, region, **kwargs)
```

Biot coupling term with α_{ij} given in:

- vector form exploiting symmetry - in 3D it has the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has the indices ordered as [11, 22, 12],
- matrix form - non-symmetric coupling parameter.

Corresponds to weak forms of Biot gradient and divergence terms. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}), \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u})$$

Call signature

dw_biot	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_v, parameter_s)

Arguments 1

- `material` : α_{ij}
- `virtual` : $\underline{v}$
- `state` : p

Arguments 2

- `material` : α_{ij}
- `state` : $\underline{u}$

- virtual : q

Arguments 3

- material : α_{ij}
- parameter_v : $\underline{u}$
- parameter_s : p

```
arg_shapes = [{'material': 'S, 1', 'parameter_s': 1, 'parameter_v': 'D',  
'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad':  
(1, None)}, {'material': 'D, D'}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),  
(('material', 'parameter_v', 'parameter_s')))
```

```
get_eval_shape(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'dw_biot'
```

```
set_arg_types()
```

sfepy.terms.terms_compat module

```
class sfepy.terms.terms_compat.CauchyStrainTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

ev_cauchy_strain_s (parameter)

```
name = 'ev_cauchy_strain_s'
```

```
class sfepy.terms.terms_compat.DSumNodalValuesTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_sum_vals (parameter)

```
name = 'd_sum_vals'
```

```
class sfepy.terms.terms_compat.DSurfaceFluxTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_surface_flux (material, parameter)

```
name = 'd_surface_flux'
```



```
class sfepy.terms.terms_compat.DSurfaceMomentTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_surface_moment (material, parameter)

```
name = 'd_surface_moment'
```

```
class sfepy.terms.terms_compat.DVolumeSurfaceTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_volume_surface (parameter)

```
name = 'd_volume_surface'
```

```
class sfepy.terms.terms_compat.DotSurfaceProductTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

dw_surface_dot (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

```
name = 'dw_surface_dot'
```

```
class sfepy.terms.terms_compat.DotVolumeProductTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

dw_volume_dot (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

```
name = 'dw_volume_dot'
```

```
class sfepy.terms.terms_compat.IntegrateSurfaceMatTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

ev_surface_integrate_mat (material, parameter)

```
name = 'ev_surface_integrate_mat'
```

```
class sfepy.terms.terms_compat.IntegrateSurfaceOperatorTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

Call signature

dw_surface_integrate (opt_material, virtual)

```
name = 'dw_surface_integrate'
```

```
class sfepy.terms.terms_compat.IntegrateSurfaceTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

<code>ev_surface_integrate</code> (opt_material, parameter)

```
name = 'ev_surface_integrate'
```

```
class sfepy.terms.terms_compat.IntegrateVolumeMatTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

<code>ev_volume_integrate_mat</code> (material, parameter)

```
name = 'ev_volume_integrate_mat'
```

```
class sfepy.terms.terms_compat.IntegrateVolumeOperatorTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

Call signature

<code>dw_volume_integrate</code> (opt_material, virtual)

```
name = 'dw_volume_integrate'
```

```
class sfepy.terms.terms_compat.IntegrateVolumeTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

<code>ev_volume_integrate</code> (opt_material, parameter)

```
name = 'ev_volume_integrate'
```

```
class sfepy.terms.terms_compat.SDVolumeDotTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

<code>ev_sd_volume_dot</code> (parameter_1, parameter_2, parameter_mv)

```
name = 'ev_sd_volume_dot'
```

```
class sfepy.terms.terms_compat.SurfaceDivTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

<code>ev_surface_div</code> (opt_material, parameter)

```
name = 'ev_surface_div'
```

```
class sfepy.terms.terms_compat.SurfaceGradTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

ev_surface_grad (opt_material, parameter)

```
name = 'ev_surface_grad'
```

```
class sfepy.terms.terms_compat.SurfaceTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_surface (parameter)

```
name = 'd_surface'
```

```
class sfepy.terms.terms_compat.VolumeXTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

d_volume (parameter)

```
name = 'd_volume'
```

sfepy.terms.terms_constraints module

```
class sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm(name, arg_str, integral, region,
                                                                **kwargs)
```

Non-penetration condition in the weak sense using a penalty.

Definition

$$\int_{\Gamma} c(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u})$$

Call signature

dw_non_penetration_p (material, virtual, state)

Arguments

- material : c
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function(out, val_qp, ebf, mat, sg, diff_var)
```

```
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'dw_non_penetration_p'
```

```
class sfepy.terms.terms_constraints.NonPenetrationTerm(name, arg_str, integral, region, **kwargs)
```

Non-penetration condition in the weak sense.

Definition

$$\int_{\Gamma} c \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} c \hat{\lambda} \underline{n} \cdot \underline{u}$$
$$\int_{\Gamma} \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} \hat{\lambda} \underline{n} \cdot \underline{u}$$

Call signature

dw_non_penetration	(opt_material, virtual, state)
	(opt_material, state, virtual)

Arguments 1

- material : c (optional)
- virtual : $\underline{v}$
- state : λ

Arguments 2

- material : c (optional)
- state : $\underline{u}$
- virtual : $\hat{\lambda}$

```
arg_shapes = [{'opt_material': '1, 1', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None)}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state', 'virtual'))
```

```
static function(out, val_qp, ebf, bf, mat, sg, diff_var, mode)
```

ebf belongs to vector variable, *bf* to scalar variable.

```
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
modes = ('grad', 'div')
```

```
name = 'dw_non_penetration'
```

sfepy.terms.terms_contact module

class sfepy.terms.terms_contact.**ContactIPCTerm**(*args, **kwargs)

Contact term based on IPC Toolkit.

This term has a dynamic connectivity of DOFs in its region.

Definition

$$\int_{\Gamma_c} \sum_{k \in C} \nabla b(d_k, \underline{u})$$

Call signature

dw_contact_ipc (material_m, material_k, material_d, material_p, virtual, state)

Arguments

- material_m : m
- material_k : k
- material_d : $\hat{d}$
- material_p : PSDProjectionMethod
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_d': ':: 1', 'material_k': ':: 1', 'material_m': ':: 1', 'material_p': ':: str', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material_m', 'material_k', 'material_d', 'material_p', 'virtual', 'state')
```

```
call_function(out, fargs)
```

```
eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
```

```
static function(out, fun, *args)
```

```
static function_weak(out, out_cc)
```

```
geometries = ['3_4', '3_8']
```

```
get_contact_info(state)
```

```
get_fargs(avg_mass, stiffness, dhat, spd_projection, virtual, state, mode=None, term_mode=None, diff_var=None, ci=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'dw_contact_ipc'
```

class sfepy.terms.terms_contact.**ContactInfo**(region, integral, geo, state)

Various contact-related data of contact terms.

update(xx)

A dict-like update for Struct attributes.

class sfepy.terms.terms_contact.**ContactTerm**(*args, **kwargs)

Contact term with a penalty function.

The penalty function is defined as $\varepsilon_N \langle g_N(\underline{u}) \rangle$, where ε_N is the normal penalty parameter and $\langle g_N(\underline{u}) \rangle$ are the Macaulay's brackets of the gap function $g_N(\underline{u})$.

This term has a dynamic connectivity of DOFs in its region.

Definition

$$\int_{\Gamma_c} \varepsilon_N \langle g_N(\underline{u}) \rangle \underline{n} \underline{v}$$

Call signature

dw_contact (material, virtual, state)

Arguments

- material : ε_N
- virtual : $\underline{v}$
- state : $\underline{u}$

arg_shapes = {'material': '.: 1', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material', 'virtual', 'state')

call_function(out, fargs)

eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

static function(out, fun, *args)

static function_weak(out, out_cc)

get_contact_info(geo, state, init_gps=False)

get_eval_shape(epss, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(epss, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

static integrate(out, val_qp, geo, fmode)

integration = 'facet'

name = 'dw_contact'

sfePy.terms.terms_dg module

Discontinuous Galekrin method specific terms

Note

In `einsum` calls the following convention is used:

i represents iterating over all cells of a region;

n represents iterating over selected cells of a region, for example over cells on boundary;

b represents iterating over basis functions of state variable;

d represents iterating over basis functions of test variable;

k, l, m represent iterating over geometric dimensions, for example coordinates of velocity or facet normal vector or rows and columns of diffusion tensor;

q represents iterating over quadrature points;

f represents iterating over facets of cell;

class `sfePy.terms.terms_dg.AdvectionDGFluxTerm`(*name, arg_str, integral, region, **kwargs*)

Lax-Friedrichs flux term for advection of scalar quantity p with the advection velocity $\underline{a}$ given as a material parameter (a known function of space and time).

Definition

$$\int_{\partial T_K} \underline{n} \cdot \underline{f}^*(p_{in}, p_{out}) q$$

where

$$\underline{f}^*(p_{in}, p_{out}) = \underline{a} \frac{p_{in} + p_{out}}{2} + (1 - \alpha) \underline{n} C \frac{p_{in} - p_{out}}{2},$$

$\alpha \in [0, 1]$; $\alpha = 0$ for upwind scheme, $\alpha = 1$ for central scheme, and

$$C = \max_{p \in [?, ?]} |n_x a_1 + n_y a_2| = \max_{p \in [?, ?]} |\underline{n} \cdot \underline{a}|$$

the p_{in} resp. p_{out} is solution on the boundary of the element provided by element itself resp. its neighbor and $\underline{a}$ is advection velocity.

Call signature

```
dw_dg_advect_laxfrie_flux (opt_material, material_advelo, virtual, state)
```

Arguments 1

- material : $\underline{a}$
- virtual : q
- state : p

Arguments 3

- material : $\underline{a}$
- virtual : q

- state : p
- opt_material : α

```
alpha = 0
```

```
arg_shapes = [{'material_advelo': 'D, 1', 'opt_material': '.: 1', 'state': 1,
'virtual': (1, 'state')}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'material_advelo', 'virtual', 'state')
```

```
function(out, state, diff_var, field, region, advelo)
```

```
get_fargs(alpha, advelo, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'cell'
```

```
modes = ('weak',)
```

```
name = 'dw_dg_advect_laxfrie_flux'
```

```
symbolic = {'expression': 'div(a*p)*w', 'map': {'a': 'material', 'p': 'state',
'v': 'virtual'}}
```

```
class sfepy.terms.terms_dg.DGTerm(name, arg_str, integral, region, **kwargs)
```

Abstract base class for DG terms, provides alternative call_function and eval_real methods to accommodate returning iels and vals.

```
call_function(out, fargs)
```

```
eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
```

```
poly_space_basis = 'legendre'
```

```
class sfepy.terms.terms_dg.DiffusionDGFluxTerm(name, arg_str, integral, region, **kwargs)
```

Basic DG diffusion flux term for scalar quantity.

Definition

$$\int_{\partial T_K} D \langle \nabla p \rangle [q], \int_{\partial T_K} D \langle \nabla q \rangle [p]$$

where

$$\langle \nabla \phi \rangle = \frac{\nabla \phi_{in} + \nabla \phi_{out}}{2}$$

$$[\phi] = \phi_{in} - \phi_{out}$$

Math

The p_{in} resp. p_{out} is solution on the boundary of the element provided by element itself resp. its neighbour.

Call signature

dw_dg_diffusion_flux	(material, state, virtual)
	(material, virtual, state)

Arguments 1

- material : D

- state : p
- virtual : q

Arguments 2

- material : D
- virtual : q
- state : p

```
arg_shapes = [{'material': '1', '1', 'state/avg_state': 1, 'state/avg_virtual': 1,
'virtual/avg_state': (1, None), 'virtual/avg_virtual': (1, None)}]
```

```
arg_types = (('material', 'state', 'virtual'), ('material', 'virtual', 'state'))
```

```
function(out, state, diff_var, field, region, D)
```

```
get_fargs(diff_tensor, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'cell'
```

```
modes = ('avg_state', 'avg_virtual')
```

```
name = 'dw_dg_diffusion_flux'
```

class sfepy.terms.terms_dg.**DiffusionInteriorPenaltyTerm**(name, arg_str, integral, region, **kwargs)
 Penalty term used to counteract discontinuity arising when modeling diffusion using Discontinuous Galerkin schemes.

Definition

$$\int_{\partial T_K} \bar{D}C_w \frac{Ord^2}{d(\partial T_K)} [p][q]$$

where

$$[\phi] = \phi_{in} - \phi_{out}$$

Math

the p_{in} resp. p_{out} is solution on the boundary of the element provided by element itself resp. its neighbour.

Call signature

dw_dg_interior_penalty (material, material_Cw, virtual, state)

Arguments

- material : D
- material : C_w
- state : p
- virtual : q

```
arg_shapes = [{'material': '1', '1', 'material_Cw': '.: 1', 'state': 1, 'virtual':
(1, 'state')}]
```

```
arg_types = ('material', 'material_Cw', 'virtual', 'state')
```

```
function(out, state, diff_var, field, region, Cw, diff_tensor)
```

```
get_fargs(diff_tensor, Cw, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak',)
```

```
name = 'dw_dg_interior_penalty'
```

```
class sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm(name, arg_str, integral, region,
                                                         **kwargs)
```

Lax-Friedrichs flux term for nonlinear hyperbolic term of scalar quantity p with the vector function $\underline{f}$ given as a material parameter.

Definition

$$\int_{\partial T_K} \underline{n} \cdot \underline{f}^*(p_{in}, p_{out}) q$$

where

$$\underline{f}^*(p_{in}, p_{out}) = \frac{\underline{f}(p_{in}) + \underline{f}(p_{out})}{2} + (1 - \alpha) \underline{n} C \frac{p_{in} - p_{out}}{2},$$

$\alpha \in [0, 1]$; $\alpha = 0$ for upwind scheme, $\alpha = 1$ for central scheme, and

$$C = \max_{p \in [?, ?]} \left| n_x \frac{df_1}{dp} + n_y \frac{df_2}{dp} + \dots \right| = \max_{p \in [?, ?]} \left| \vec{n} \cdot \frac{d\underline{f}}{dp}(p) \right|$$

the p_{in} resp. p_{out} is solution on the boundary of the element provided by element itself resp. its neighbor.

Call signature

dw_dg_nonlinear_laxfrie_flux (opt_material, fun, fun_d, virtual, state)

Arguments 1

- material : $\underline{f}$
- material : $\frac{df}{dp}$
- virtual : q
- state : p

Arguments 3

- material : $\underline{f}$
- material : $\frac{df}{dp}$
- virtual : q
- state : p
- opt_material : α

```

alf = 0

arg_shapes = [{'material_fun': '.: 1', 'material_fun_d': '.: 1', 'opt_material':
'.: 1', 'state': 1, 'virtual': (1, 'state')}, {'opt_material': None}]

arg_types = ('opt_material', 'fun', 'fun_d', 'virtual', 'state')

function(out, state, field, region, f, df)

get_fargs(alpha, fun, dfun, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'cell'

modes = ('weak',)

name = 'dw_dg_nonlinear_laxfrie_flux'

symbolic = {'expression': 'div(f(p))*w', 'map': {'f': 'function', 'p': 'state',
'v': 'virtual'}}

```

```
class sfepy.terms.terms_dg.NonlinearScalarDotGradTerm(name, arg_str, integral, region, **kwargs)
```

Product of virtual and divergence of vector function of state or volume dot product of vector function of state and gradient of scalar virtual.

Definition

$$\int_{\Omega} q \cdot \nabla \cdot \underline{f}(p) = \int_{\Omega} q \cdot \operatorname{div} \underline{f}(p), \quad \int_{\Omega} \underline{f}(p) \cdot \nabla q$$

Call signature

dw_ns_dot_grad_s	(fun, fun_d, virtual, state)
	(fun, fun_d, state, virtual)

Arguments 1

- function : $\underline{f}$
- virtual : q
- state : p

Arguments 2

- function : $\underline{f}$
- state : p
- virtual : q

TODO maybe this term would fit better to terms_dot?

```

arg_shapes = [{'material_fun': '.: 1', 'material_fun_d': '.: 1',
'state/grad_state': 1, 'state/grad_virtual': 1, 'virtual/grad_state': (1, None),
'virtual/grad_virtual': (1, None)}]

arg_types = (('fun', 'fun_d', 'virtual', 'state'), ('fun', 'fun_d', 'state',
'virtual'))

```

```
static function(out, out_qp, geo, fmode)

get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad_state', 'grad_virtual')

name = 'dw_ns_dot_grad_s'
```

sfepy.terms.terms_diffusion module

class sfepy.terms.terms_diffusion.**AdvectDivFreeTerm**(name, arg_str, integral, region, **kwargs)

Advection of a scalar quantity p with the advection velocity $\underline{y}$ given as a material parameter (a known function of space and time).

The advection velocity has to be divergence-free!

Definition

$$\int_{\Omega} \nabla \cdot (\underline{y} p) q = \int_{\Omega} \underbrace{((\nabla \cdot \underline{y}) + \underline{y} \cdot \nabla) p}_{\equiv 0} q$$

Call signature

dw_advect_div_free (material, virtual, state)

Arguments

- material : $\underline{y}$
- virtual : q
- state : p

```
arg_shapes = {'material': 'D, 1', 'state': '1', 'virtual': ('1', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
mode = 'grad_state'
```

```
name = 'dw_advect_div_free'
```

class sfepy.terms.terms_diffusion.**ConvectVGradSTerm**(name, arg_str, integral, region, **kwargs)

Scalar gradient term with convective velocity.

Definition

$$\int_{\Omega} q(\underline{u} \cdot \nabla p)$$

Call signature

dw_convect_v_grad_s (virtual, state_v, state_s)

Arguments

- virtual : q

- `state_v` : u
- `state_s` : p

```
arg_shapes = [{'state_s': 1, 'state_v': 'D', 'virtual': (1, 'state_s')}]
```

```
arg_types = ('virtual', 'state_v', 'state_s')
```

```
static function(out, val_v, grad_s, cmap_v, cmap_s, is_diff)
```

```
get_fargs(virtual, state_v, state_s, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_convect_v_grad_s'
```

```
class sfepy.terms.terms_diffusion.DiffusionCoupling(name, arg_str, integral, region, **kwargs)
```

Diffusion coupling term with material parameter K_j .

Definition

$$\int_{\Omega} p K_j \nabla_j q, \int_{\Omega} q K_j \nabla_j p$$

Call signature

dw_diffusion_coupling	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_1, parameter_2)

Arguments

- `material` : K_j
- `virtual` : q
- `state` : p

```
arg_shapes = {'material': 'D, 1', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter_1', 'parameter_2'))
```

```
static d_fun(out, mat, val, grad, vg)
```

```
static dw_fun(out, val, mat, bf, vg, fmode)
```

```
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak0', 'weak1', 'eval')
```

```
name = 'dw_diffusion_coupling'
```

```
set_arg_types()
```

class sfepy.terms.terms_diffusion.**DiffusionRTerm**(*name, arg_str, integral, region, **kwargs*)

Diffusion-like term with material parameter K_j (to use on the right-hand side).

Definition

$$\int_{\Omega} K_j \nabla_j q$$

Call signature

dw_diffusion_r (material, virtual)

Arguments

- material : K_j
- virtual : q

arg_shapes = {'material': 'D, 1', 'virtual': (1, None)}

arg_types = ('material', 'virtual')

static function(*out, mtx_d, cmap*)

get_fargs(*mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs*)

name = 'dw_diffusion_r'

class sfepy.terms.terms_diffusion.**DiffusionTerm**(*name, arg_str, integral, region, **kwargs*)

General diffusion term with permeability K_{ij} . Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p$$

Call signature

dw_diffusion (material, virtual, state)
(material, parameter_1, parameter_2)

Arguments

- material: K_{ij}
- virtual/parameter_1: q
- state/parameter_2: p

arg_shapes = {'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}

arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))

get_eval_shape(*mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs*)

get_fargs(*mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs*)

```

modes = ('weak', 'eval')

name = 'dw_diffusion'

set_arg_types()

symbolic = {'expression': 'div( K * grad( u ) )', 'map': {'K': 'material', 'u':
'state'}}

```

class sfepy.terms.terms_diffusion.**DiffusionVelocityTerm**(*name, arg_str, integral, region, **kwargs*)

Evaluate diffusion velocity.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$- \int_D K_{ij} \nabla_j p$$

Call signature

ev_diffusion_velocity (material, parameter)

Arguments

- material : K_{ij}
- parameter : p

```
arg_shapes = {'material': 'D, D', 'parameter': 1}
```

```
arg_types = ('material', 'parameter')
```

```
static function(out, grad, mat, vg, fmode)
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet_extra')
```

```
name = 'ev_diffusion_velocity'
```

class sfepy.terms.terms_diffusion.**LaplaceTerm**(*name, arg_str, integral, region, **kwargs*)

Laplace term with c coefficient. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} c \nabla q \cdot \nabla p$$

Call signature

dw_laplace (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

Arguments 1

- material: c

- virtual/parameter_1: q
- state/parameter_2: p

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 1, 'parameter_2': 1,
               'state': 1, 'virtual': (1, 'state')}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1',
            'parameter_2'))

modes = ('weak', 'eval')

name = 'dw_laplace'

set_arg_types()

symbolic = {'expression': 'c * div( grad( u ) )', 'map': {'c': 'opt_material',
                'u': 'state'}}
```

```
class sfepy.terms.terms_diffusion.NonlinearDiffusionTerm(name, arg_str, integral, region,
                                                         **kwargs)
```

The diffusion term with a scalar coefficient given by a user supplied function of the state variable.

Definition

$$\int_{\Omega} \nabla q \cdot \nabla p f(p)$$

Call signature

dw_nl_diffusion (fun, dfun, virtual, state)

Arguments

- fun : $f(p)$
- dfun : $\partial f(p)/\partial p$
- virtual : q
- state : p

```
arg_shapes = {'dfun': <function NonlinearDiffusionTerm.<lambda>>, 'fun': <function
NonlinearDiffusionTerm.<lambda>>, 'state': 1, 'virtual': (1, 'state')}

arg_types = ('fun', 'dfun', 'virtual', 'state')

static function(out, out_qp, geo)

get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_nl_diffusion'
```

```
class sfepy.terms.terms_diffusion.SDDiffusionTerm(name, arg_str, integral, region, **kwargs)
```

Diffusion sensitivity analysis term.

Definition

$$\int_{\Omega} \hat{K}_{ij} \nabla_i q \nabla_j p$$

$$\hat{K}_{ij} = K_{ij} \left(\delta_{ik} \delta_{jl} \nabla \cdot \underline{\mathcal{V}} - \delta_{ik} \frac{\partial \mathcal{V}_j}{\partial x_l} - \delta_{jl} \frac{\partial \mathcal{V}_i}{\partial x_k} \right)$$

Call signature

ev_sd_diffusion (material, parameter_q, parameter_p, parameter_mv)

Arguments

- material: K_{ij}
- parameter_q: q
- parameter_p: p
- parameter_mv: $\underline{\mathcal{V}}$

```
arg_shapes = {'material': 'D, D', 'parameter_mv': 'D', 'parameter_p': 1,
              'parameter_q': 1}
```

```
arg_types = ('material', 'parameter_q', 'parameter_p', 'parameter_mv')
```

```
static function(out, grad_q, grad_p, grad_w, div_w, mtx_d, cmap)
```

```
get_eval_shape(mat, parameter_q, parameter_p, parameter_mv, mode=None, term_mode=None,
               diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter_q, parameter_p, parameter_mv, mode=None, term_mode=None, diff_var=None,
           **kwargs)
```

```
name = 'ev_sd_diffusion'
```

```
class sfepy.terms.terms_diffusion.SurfaceFlux0OperatorTerm(name, arg_str, integral, region,
                                                           **kwargs)
```

Surface flux operator term.

Definition

$$\int_{\Gamma} q \underline{n} \cdot \underline{K} \cdot \nabla p$$

Call signature

dw_surface_flux (opt_material, virtual, state)

Arguments

- material : $\underline{K}$
- virtual : q
- state : p

```
arg_shapes = [{'opt_material': 'D, D', 'state': 1, 'virtual': (1, 'state')},
               {'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual', 'state')
```

```
static function(out, grad, mat, bf, cmap, fis, mode)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
name = 'dw_surface_flux'
```

```
class sfepy.terms.terms_diffusion.SurfaceFluxTerm(name, arg_str, integral, region, **kwargs)
```

Surface flux term.

Supports 'eval', 'el_eval' and 'el_avg' evaluation modes.

Definition

$$\int_{\Gamma} \underline{n} \cdot K_{ij} \nabla_j p$$

Call signature

ev_surface_flux (material, parameter)

Arguments

- material: K
- parameter: p ,

```
arg_shapes = {'material': 'D, D', 'parameter': 1}
```

```
arg_types = ('material', 'parameter')
```

```
static function(out, grad, mtx_d, cmap, mode)
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
name = 'ev_surface_flux'
```

sfepy.terms.terms_dot module

```
class sfepy.terms.terms_dot.BCNewtonTerm(name, arg_str, integral, region, **kwargs)
```

Newton boundary condition term.

Definition

$$\int_{\Gamma} \alpha q (p - p_{\text{outer}})$$

Call signature

dw_bc_newton (material_1, material_2, virtual, state)

Arguments

- material_1 : α
- material_2 : p_{outer}
- virtual : q
- state : p

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'state': 1, 'virtual':
(1, 'state')}
```

```
arg_shapes_dict = None
```

```
arg_types = ('material_1', 'material_2', 'virtual', 'state')
```

```
get_fargs(alpha, p_outer, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
mode = 'weak'
```

```
name = 'dw_bc_newton'
```

```
class sfepy.terms.terms_dot.DotProductTerm(name, arg_str, integral, region, **kwargs)
```

Volume and surface $L^2()$ weighted dot product for both scalar and vector fields. If the region is a surface and either virtual or state variable is a vector, the orientation of the normal vectors is outwards to the parent region of the virtual variable. Can be evaluated. Can use derivatives.

Definition

$$\begin{aligned} & \int_{\mathcal{D}} qp, \int_{\mathcal{D}} \underline{v} \cdot \underline{u} \\ & \int_{\Gamma} \underline{v} \cdot \underline{np}, \int_{\Gamma} q\underline{n} \cdot \underline{u}, \\ & \int_{\mathcal{D}} cq\underline{p}, \int_{\mathcal{D}} c\underline{v} \cdot \underline{u}, \int_{\mathcal{D}} \underline{v} \cdot \underline{c} \cdot \underline{u} \end{aligned}$$

Call signature

dw_dot	(opt_material, virtual, state)
	(opt_material, parameter_1, parameter_2)

Arguments

- material: c or $\underline{c}$ (optional)
- virtual/parameter_1: q or $\underline{v}$
- state/parameter_2: p or $\underline{u}$

```
arg_shapes_dict = {'cell': [{'opt_material': '1, 1', 'parameter_1': 1,
'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}, {'opt_material': None},
{'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D',
'virtual': ('D', 'state')}, {'opt_material': 'D, D'}, {'opt_material': None}],
'facet': [{'opt_material': '1, 1', 'parameter_1': 1, 'parameter_2': 1, 'state':
1, 'virtual': (1, 'state')}, {'opt_material': None}, {'opt_material': '1, 1',
'state': 'D', 'virtual': (1, None)}, {'opt_material': None}, {'opt_material':
'1, 1', 'state': 1, 'virtual': ('D', None)}, {'opt_material': None},
{'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D',
'virtual': ('D', 'state')}, {'opt_material': 'D, D'}, {'opt_material': None}]}
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1',
'parameter_2'))
```

```
static d_dot(out, mat, val1_qp, val2_qp, geo)
```

```
static dw_dot(out, mat, val_qp, vgeo, sgeo, fun, fmode)
```

```
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
modes = ('weak', 'eval')
```

```
name = 'dw_dot'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_dot.DotSPProductVolumeOperatorWETHTerm(name, arg_str, integral, region,
**kwargs)
```

Fading memory volume $L^2(\Omega)$ weighted dot product for scalar fields. This term has the same definition as `dw_volume_dot_w_scalar_th`, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \mathcal{G}(t - \tau) p(\tau) \, d\tau \right] q$$

Call signature

<code>dw_volume_dot_w_scalar_eth</code> (ts, material_0, material_1, virtual, state)

Arguments

- ts : TimeStepper instance
- material_0 : $\mathcal{G}(0)$
- material_1 : $\exp(-\lambda \Delta t)$ (decay at t_1)
- virtual : q
- state : p

```
arg_shapes = {'material_0': '1, 1', 'material_1': '1, 1', 'state': 1, 'virtual':
(1, 'state')}
```

```
arg_types = ('ts', 'material_0', 'material_1', 'virtual', 'state')
```

```
static function(out, coef, val_qp, rmap, ccmap, is_diff)
```

```
get_fargs(ts, mat0, mat1, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_volume_dot_w_scalar_eth'
```

```
class sfepy.terms.terms_dot.DotSPProductVolumeOperatorWTHTerm(name, arg_str, integral, region,
                                                                **kwargs)
```

Fading memory volume $L^2(\Omega)$ weighted dot product for scalar fields. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \mathcal{G}(t - \tau) p(\tau) \, d\tau \right] q$$

Call signature

```
dw_volume_dot_w_scalar_th (ts, material, virtual, state)
```

Arguments

- `ts`: TimeStepper instance
- `material`: $\mathcal{G}(\tau)$
- `virtual`: q
- `state`: p

```
arg_shapes = {'material': '.: N, 1, 1', 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = ('ts', 'material', 'virtual', 'state')
```

```
static function(out, coef, val_qp, rmap, ccmap, is_diff)
```

```
get_fargs(ts, mats, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_volume_dot_w_scalar_th'
```

```
class sfepy.terms.terms_dot.ScalarDotGradIScalarTerm(name, arg_str, integral, region, **kwargs)
```

Dot product of a scalar and the i -th component of gradient of a scalar. The index should be given as a ‘special_constant’ material parameter.

Definition

$$Z^i = \int_{\Omega} q \nabla_i p$$

Call signature

```
dw_s_dot_grad_i_s (material, virtual, state)
```

Arguments

- material : i
- virtual : q
- state : p

```
arg_shapes = {'material': '.: 1, 1', 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static dw_fun(out, bf, vg, grad, idx, fmode)
```

```
get_fargs(material, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_s_dot_grad_i_s'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_dot.ScalarDotMGradScalarTerm(name, arg_str, integral, region, **kwargs)
```

Volume dot product of a scalar gradient dotted with a material vector with a scalar.

Definition

$$\int_{\Omega} \underline{qy} \cdot \nabla p, \int_{\Omega} \underline{py} \cdot \nabla q$$

Call signature

dw_s_dot_mgrad_s	(material, virtual, state)
	(material, state, virtual)

Arguments 1

- material : $\underline{y}$
- virtual : q
- state : p

Arguments 2

- material : $\underline{y}$
- state : p
- virtual : q

```
arg_shapes = [{'material': 'D, 1', 'state/grad_state': 1, 'state/grad_virtual': 1, 'virtual/grad_state': (1, None), 'virtual/grad_virtual': (1, None)}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'))
```

```
static function(out, out_qp, geo, fmode)
```

```
get_fargs(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad_state', 'grad_virtual')
```

```
name = 'dw_s_dot_mgrad_s'
```

class sfepy.terms.terms_dot.**VectorDotGradScalarTerm**(*name, arg_str, integral, region, **kwargs*)

Volume dot product of a vector and a gradient of scalar. Can be evaluated.

Definition

$$\begin{aligned} & \int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q \\ & \int_{\Omega} c \underline{v} \cdot \nabla p, \int_{\Omega} c \underline{u} \cdot \nabla q \\ & \int_{\Omega} \underline{v} \cdot (\underline{c} \nabla p), \int_{\Omega} \underline{u} \cdot (\underline{c} \nabla q) \end{aligned}$$

Call signature

dw_v_dot_grad_s	(opt_material, virtual, state)
	(opt_material, state, virtual)
	(opt_material, parameter_v, parameter_s)

Arguments 1

- material: *c* or $\underline{c}$ (optional)
- virtual/parameter_v: *v*
- state/parameter_s: *p*

Arguments 2

- material : *c* or $\underline{c}$ (optional)
- state : $\underline{u}$
- virtual : *q*

```
arg_shapes = [{ 'opt_material': '1, 1', 'parameter_s': 1, 'parameter_v': 'D',
'virtual/s_weak': 'D', 'state/v_weak': 1, 'virtual/s_weak': (1, None),
'virtual/v_weak': ('D', None)}, { 'opt_material': 'D, D'}, { 'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state',
'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))
```

```
get_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('v_weak', 's_weak', 'eval')
```

```
name = 'dw_v_dot_grad_s'
```

```
set_arg_types()
```

class sfepy.terms.terms_dot.**VectorDotScalarTerm**(*name, arg_str, integral, region, **kwargs*)

Volume dot product of a vector and a scalar. Can be evaluated.

Definition

$$\int_{\Omega} \underline{v} \cdot \underline{c} p, \int_{\Omega} \underline{u} \cdot \underline{c} q$$

Call signature

dw_vm_dot_s	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_v, parameter_s)

Arguments 1

- material : c
- virtual/parameter_v: v
- state/parameter_s: p

Arguments 2

- material : c
- state : u
- virtual : q

```
arg_shapes = [{'material': 'D', 1, 'parameter_s': 1, 'parameter_v': 'D',
'state/s_weak': 'D', 'state/v_weak': 1, 'virtual/s_weak': (1, None),
'virtual/v_weak': ('D', None)}]

arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),
('material', 'parameter_v', 'parameter_s'))

static d_dot(out, mat, val1_qp, val2_qp, geo)

static dw_dot(out, mat, val_qp, bfve, bfsc, geo, fmode)

get_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('v_weak', 's_weak', 'eval')

name = 'dw_vm_dot_s'

set_arg_types()
```

sfepy.terms.terms_elastic module

class sfepy.terms.terms_elastic.CauchyStrainTerm(name, arg_str, integral, region, **kwargs)

Evaluate Cauchy strain tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12]. The last three (non-diagonal) components are doubled so that it is energetically conjugate to the Cauchy stress tensor with the same storage.

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\mathcal{D}} \underline{\underline{\epsilon}}(\underline{w})$$

Call signature

ev_cauchy_strain (parameter)

Arguments

- parameter : $\underline{w}$

```
arg_shapes = {'parameter': 'D'}
```

```
arg_types = ('parameter',)
```

```
static function(out, strain, vg, fmode)
```

```
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet_extra')
```

```
name = 'ev_cauchy_strain'
```

```
class sfepy.terms.terms_elastic.CauchyStressETHTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate fading memory Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Assumes an exponential approximation of the convolution kernel resulting in much higher efficiency.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$$

Call signature

ev_cauchy_stress_eth (ts, material_0, material_1, parameter)

Arguments

- ts : TimeStepper instance
- material_0 : $\mathcal{H}_{ijkl}(0)$
- material_1 : $\exp(-\lambda \Delta t)$ (decay at t_1)
- parameter : $\underline{w}$

```
arg_shapes = {'material_0': 'S, S', 'material_1': '1, 1', 'parameter': 'D'}
```

```
arg_types = ('ts', 'material_0', 'material_1', 'parameter')
```

```
get_eval_shape(ts, mat0, mat1, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(ts, mat0, mat1, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_cauchy_stress_eth'
```

class sfepy.terms.terms_elastic.CauchyStressTHTerm(*name, arg_str, integral, region, **kwargs*)

Evaluate fading memory Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$$

Call signature

ev_cauchy_stress_th (ts, material, parameter)

Arguments

- ts : TimeStepper instance
- material : $\mathcal{H}_{ijkl}(\tau)$
- parameter : $\underline{w}$

arg_shapes = {'material': '.: N, S, S', 'parameter': 'D'}

arg_types = ('ts', 'material', 'parameter')

get_eval_shape(ts, mats, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(ts, mats, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_cauchy_stress_th'

class sfepy.terms.terms_elastic.CauchyStressTerm(*name, arg_str, integral, region, **kwargs*)

Evaluate Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\mathcal{D}} D_{ijkl} e_{kl}(\underline{w})$$

Call signature

ev_cauchy_stress (material, parameter)

Arguments

- material : D_{ijkl}
- parameter : $\underline{w}$

arg_shapes = {'material': 'S, S', 'parameter': 'D'}

```
arg_types = ('material', 'parameter')
```

```
static function(out, coef, strain, mat, vg, fmode)
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet_extra')
```

```
name = 'ev_cauchy_stress'
```

```
class sfepy.terms.terms_elastic.ElasticWaveCauchyTerm(name, arg_str, integral, region, **kwargs)
```

Elastic dispersion term involving the wave strain g_{ij} , $g_{ij}(\underline{u}) = \frac{1}{2}(u_i \kappa_j + \kappa_i u_j)$, with the wave vector $\underline{\kappa}$ and the elastic strain e_{ij} . D_{ijkl} is given in the usual matrix form exploiting symmetry: in 3D it is 6×6 with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is 3×3 with the indices ordered as [11, 22, 12].

Definition

$$\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) e_{kl}(\underline{u})$$

$$\int_{\Omega} D_{ijkl} g_{ij}(\underline{u}) e_{kl}(\underline{v})$$

Call signature

dw_elastic_wave_cauchy	(material_1, material_2, virtual, state)
	(material_1, material_2, state, virtual)

Arguments 1

- material_1 : D_{ijkl}
- material_2 : $\underline{\kappa}$
- virtual : $\underline{v}$
- state : $\underline{u}$

Arguments 2

- material_1 : D_{ijkl}
- material_2 : $\underline{\kappa}$
- state : $\underline{u}$
- virtual : $\underline{v}$

```
arg_shapes = {'material_1': 'S, S', 'material_2': '.: D', 'state': 'D',
'virtual': ('D', 'state')}
```

```
arg_types = (('material_1', 'material_2', 'virtual', 'state'), ('material_1',
'material_2', 'state', 'virtual'))
```

```
static function(out, out_qp, geo, fmode)
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_fargs(mat, kappa, gvar, evar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('ge', 'eg')
```

```
name = 'dw_elastic_wave_cauchy'
```

```
class sfepy.terms.terms_elastic.ElasticWaveTerm(name, arg_str, integral, region, **kwargs)
```

Elastic dispersion term involving the wave strain g_{ij} , $g_{ij}(\underline{u}) = \frac{1}{2}(u_i \kappa_j + \kappa_i u_j)$, with the wave vector $\underline{\kappa}$. D_{ijkl} is given in the usual matrix form exploiting symmetry: in 3D it is 6×6 with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is 3×3 with the indices ordered as [11, 22, 12].

Definition

$$\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) g_{kl}(\underline{u})$$

Call signature

```
dw_elastic_wave (material_1, material_2, virtual, state)
```

Arguments

- material_1 : D_{ijkl}
- material_2 : $\underline{\kappa}$
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_1': 'S, S', 'material_2': '.: D', 'state': 'D',
'virtual': ('D', 'state')}
```

```
arg_types = ('material_1', 'material_2', 'virtual', 'state')
```

```
static function(out, out_qp, geo, fmode)
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_fargs(mat, kappa, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_elastic_wave'
```

```
class sfepy.terms.terms_elastic.LinearDRotSpringTerm(name, arg_str, integral, region, **kwargs)
```

Linear spring element with the stiffness transformed into the element direction.

Definition

$$f_k^{(i)} = -f_k^{(j)} = K_{kl}(u_l^{(j)} - u_l^{(i)})$$

$$\forall \text{ elements } T_K^{i,j}$$

in a region connecting nodes i, j

Call signature

```
dw_lin_dspring_rot (opt_material, material, virtual, state)
```

Arguments

- opt_material : $\underline{d}$

- material : $\underline{k}$
- virtual: $\underline{v}$
- state: $\underline{u}$

Stiffness matrix $\underline{K} = T(\underline{d})^T K(\underline{k}) T(\underline{d})$ is defined by 6 components $\underline{k} = [k_{u1}, k_{u2}, k_{u3}, k_{r1}, k_{r2}, k_{r3}]$ in 3D and by 3 components $\underline{k} = [k_{u1}, k_{u2}, k_{r1}]$, where k_{ui} is the stiffness for the displacement DOF and k_{ri} is for the rotational DOF. Note that the components of $\underline{k}$ are in the local coordinates system specified by a given direction $\underline{d}$ or by the vector $\underline{d} = \underline{x}^{(j)} - \underline{x}^{(i)}$ for non-coincidental end nodes. The stiffness parameter $\underline{K}$ can also be defined as a 6x6 matrix in 3D or a 3x3 matrix in 2D.

```
arg_shapes = [{'material': 'S', 1, 'opt_material': 'D', 1, 'state': 'S',
'virtual': ('S', 'state')}, {'material': 'S', S}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'material', 'virtual', 'state')
```

```
name = 'dw_lin_dspring_rot'
```

```
class sfepy.terms.terms_elastic.LinearDSpringTerm(name, arg_str, integral, region, **kwargs)
```

Linear spring element with the stiffness transformed into the element direction.

Definition

$$f_k^{(i)} = -f_k^{(j)} = K_{kl}(u_l^{(j)} - u_l^{(i)})$$

$$\forall \text{ elements } T_K^{i,j}$$

in a region connecting nodes i, j

Call signature

```
dw_lin_dspring (opt_material, material, virtual, state)
```

Arguments

- opt_material : $\underline{d}$
- material : $\underline{k}$
- virtual: $\underline{v}$
- state: $\underline{u}$

Stiffness matrix $\underline{K} = T(\underline{d})^T K(\underline{k}) T(\underline{d})$ is defined by 6 components $\underline{k} = [k_{u1}, k_{u2}, k_{u3}, k_{r1}, k_{r2}, k_{r3}]$ in 3D and by 3 components $\underline{k} = [k_{u1}, k_{u2}, k_{r1}]$, where k_{ui} is the stiffness for the displacement DOF and k_{ri} is for the rotational DOF. Note that the components of $\underline{k}$ are in the local coordinates system specified by a given direction $\underline{d}$ or by the vector $\underline{d} = \underline{x}^{(j)} - \underline{x}^{(i)}$ for non-coincidental end nodes. The stiffness parameter $\underline{K}$ can also be defined as a 6x6 matrix in 3D or a 3x3 matrix in 2D.

```
arg_shapes = [{'material': 'D', 1, 'opt_material': 'D', 1, 'state': 'D',
'virtual': ('D', 'state')}, {'material': 'D', D}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'material', 'virtual', 'state')
```

```
static function(out, mat, vec, mtx_t, diff_var)
```

```
geometries = ['1_2', '2_1_2', '3_1_2']
```

```
get_fargs(dvec, mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration_order = 0
```

```
name = 'dw_lin_dspring'
```

```
class sfepy.terms.terms_elastic.LinearElasticETHTerm(name, arg_str, integral, region, **kwargs)
```

This term has the same definition as `dw_lin_elastic_th`, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{u}(\tau)) \, d\tau \right] e_{ij}(\underline{v})$$

Call signature

<code>dw_lin_elastic_eth</code> (ts, material_0, material_1, virtual, state)

Arguments

- ts : TimeStepper instance
- material_0 : $\mathcal{H}_{ijkl}(0)$
- material_1 : $\exp(-\lambda \Delta t)$ (decay at t_1)
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_0': 'S, S', 'material_1': '1, 1', 'state': 'D',
              'virtual': ('D', 'state')}
```

```
arg_types = ('ts', 'material_0', 'material_1', 'virtual', 'state')
```

```
static function(out, coef, strain, mtx_d, cmap, is_diff)
```

```
get_fargs(ts, mat0, mat1, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_elastic_eth'
```

```
class sfepy.terms.terms_elastic.LinearElasticIsotropicTerm(name, arg_str, integral, region,
                                                            **kwargs)
```

Isotropic linear elasticity term.

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

with

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$$

Call signature

<code>dw_lin_elastic_iso</code> (material_1, material_2, virtual, state)
(material_1, material_2, parameter_1, parameter_2)

Arguments

- material_1: λ
- material_2: μ
- virtual/parameter_1: v
- state/parameter_2: u

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'parameter_1': 'D',
              'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = (('material_1', 'material_2', 'virtual', 'state'), ('material_1',
                      'material_2', 'parameter_1', 'parameter_2'))
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_eval_shape(mat1, mat2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(lam, mu, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_elastic_iso'
```

```
class sfepy.terms.terms_elastic.LinearElasticTHTerm(name, arg_str, integral, region, **kwargs)
```

Fading memory linear elastic (viscous) term. Can use derivatives.

Definition

$$\int_{\Omega} \left[\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{u}(\tau)) \, d\tau \right] e_{ij}(\underline{v})$$

Call signature

```
dw_lin_elastic_th (ts, material, virtual, state)
```

Arguments

- ts : TimeStepper instance
- material : $\mathcal{H}_{ijkl}(\tau)$
- virtual : v
- state : u

```
arg_shapes = {'material': '.: N, S, S', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('ts', 'material', 'virtual', 'state')
```

```
static function(out, coef, strain, mtx_d, cmap, is_diff)
```

```
get_fargs(ts, mats, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_elastic_th'
```

```
class sfepy.terms.terms_elastic.LinearElasticTerm(name, arg_str, integral, region, **kwargs)
```

General linear elasticity term, with D_{ijkl} given in the usual matrix form exploiting symmetry: in 3D it is 6×6 with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is 3×3 with the indices ordered as [11, 22, 12]. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

Call signature

dw_lin_elastic	(material, virtual, state)
	(material, parameter_1, parameter_2)

Arguments 1

- material : D_{ijkl}
- virtual : $\underline{v}$
- state : $\underline{u}$

Arguments 2

- material : D_{ijkl}
- parameter_1 : $\underline{w}$
- parameter_2 : $\underline{u}$

```
arg_shapes = {'material': 'S, S', 'parameter_1': 'D', 'parameter_2': 'D',  
'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1',  
'parameter_2'))
```

```
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'dw_lin_elastic'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_elastic.LinearPrestressTerm(name, arg_str, integral, region, **kwargs)
```

Linear prestress term, with the prestress σ_{ij} given either in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12], or in the matrix (possibly non-symmetric) form. Can be evaluated.

Definition

$$\int_{\Omega} \sigma_{ij} e_{ij}(\underline{v})$$

Call signature

dw_lin_prestress	(material, virtual)
	(material, parameter)

Arguments 1

- material : σ_{ij}
- virtual : $\underline{v}$

Arguments 2

- material : σ_{ij}
- parameter : $\underline{u}$

```
arg_shapes = [{'material': 'S, 1', 'parameter': 'D', 'virtual': ('D', None)},
{'material': 'D, D'}]
```

```
arg_types = (('material', 'virtual'), ('material', 'parameter'))
```

```
d_lin_prestress(out, strain, mat, vg, fmode)
```

```
get_eval_shape(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'dw_lin_prestress'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_elastic.LinearSpringTerm(name, arg_str, integral, region, **kwargs)
```

Linear spring element.

Definition

$$\underline{f}^{(i)} = -\underline{f}^{(j)} = k(\underline{u}^{(j)} - \underline{u}^{(i)})$$

$$\forall \text{ elements } T_K^{i,j}$$

in a region connecting nodes i, j

Call signature

dw_lin_spring (material, virtual, state)

Arguments 1

- material : k
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function(out, stiffness, vec, diff_var)
```

```
geometries = ['1_2', '2_1_2', '3_1_2']
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration_order = 0
```

```
name = 'dw_lin_spring'
```

class sfepy.terms.terms_elastic.LinearStrainFiberTerm(*name, arg_str, integral, region, **kwargs*)

Linear (pre)strain fiber term with the unit direction vector $\underline{d}$.

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) (d_k d_l)$$

Call signature

dw_lin_strain_fib (material_1, material_2, virtual)

Arguments

- material_1 : D_{ijkl}
- material_2 : $\underline{d}$
- virtual : $\underline{v}$

arg_shapes = {'material_1': 'S, S', 'material_2': 'D, 1', 'virtual': ('D', None)}

arg_types = ('material_1', 'material_2', 'virtual')

static function(out, mtx_d, mat, cmap)

get_fargs(mat1, mat2, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_lin_strain_fib'

class sfepy.terms.terms_elastic.LinearTrussInternalForceTerm(*name, arg_str, integral, region, **kwargs*)

Evaluate internal force in the element direction. To be used with 'el_avg' or 'qp' evaluation modes which give the same results. The material parameter EA is equal to Young modulus times element cross-section. The internal force is given by $F^{(i)} = -F^{(j)} = EA/l(U^{(j)} - U^{(i)})$, where l is the element length and U, F are the nodal displacements and the nodal forces in the element direction.

Definition

$$F = EA/l(U^{(j)} - U^{(i)})$$

$$\forall \text{ elements } T_K^{i,j}$$

in a region connecting nodes i, j

Call signature

ev_lin_truss_force (material, parameter)

Arguments

- material : EA
- parameter : $\underline{u}$

arg_shapes = {'material': '1, 1', 'parameter': 'D'}

arg_types = ('material', 'parameter')

```

static function(out, mat, vec, mtx_t)

geometries = ['1_2', '2_1_2', '3_1_2']

get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration_order = 0

name = 'ev_lin_truss_force'

```

class sfepy.terms.terms_elastic.LinearTrussTerm(name, arg_str, integral, region, **kwargs)

Evaluate internal force in the element direction. To be used with ‘el_avg’ or ‘qp’ evaluation modes which give the same results. The material parameter EA is equal to Young modulus times element cross-section. The internal force is given by $F^{(i)} = -F^{(j)} = EA/l(U^{(j)} - U^{(i)})$, where l is the element length and U, F are the nodal displacements and the nodal forces in the element direction.

Definition

$$F^{(i)} = -F^{(j)} = EA/l(U^{(j)} - U^{(i)})$$

$\forall$ elements $T_K^{i,j}$
in a region connecting nodes i, j

Call signature

dw_lin_truss (material, virtual, state)

Arguments

- material : EA
- parameter : $\underline{u}$

```

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')

static function(out, mat, vec, mtx_t, length, diff_var)

geometries = ['1_2', '2_1_2', '3_1_2']

get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

static get_mtx_t_and_length(coors, dx=None)

integration_order = 0

name = 'dw_lin_truss'

```

class sfepy.terms.terms_elastic.NonsymElasticTerm(name, arg_str, integral, region, **kwargs)

Elasticity term with non-symmetric gradient. The indices of matrix D_{ijkl} are ordered as [11, 12, 13, 21, 22, 23, 31, 32, 33] in 3D and as [11, 12, 21, 22] in 2D.

Definition

$$\int_{\Omega} \underline{\underline{D}} \nabla \underline{u} : \nabla \underline{v}$$

Call signature

dw_nonsym_elastic	(material, virtual, state)
	(material, parameter_1, parameter_2)

Arguments 1

- material : $\underline{\underline{D}}$
- virtual : $\underline{v}$
- state : $\underline{u}$

Arguments 2

- material : $\underline{\underline{D}}$
- parameter_1 : $\underline{w}$
- parameter_2 : $\underline{u}$

```
arg_shapes = {'material': 'D2, D2', 'parameter_1': 'D', 'parameter_2': 'D',
              'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1',
        'parameter_2'))
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'dw_nonsym_elastic'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_elastic.SDLinearElasticTerm(name, arg_str, integral, region, **kwargs)
```

Sensitivity analysis of the linear elastic term.

Definition

$$\int_{\Omega} \hat{D}_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

$$\hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \underline{\mathcal{V}}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$$

Call signature

ev_sd_lin_elastic	(material, parameter_w, parameter_u, parameter_mv)

Arguments

- material : D_{ijkl}
- parameter_w : $\underline{w}$
- parameter_u : $\underline{u}$

- `parameter_mv` : $\mathcal{V}$

```
arg_shapes = {'material': 'S, S', 'parameter_mv': 'D', 'parameter_u': 'D',
              'parameter_w': 'D'}

arg_types = ('material', 'parameter_w', 'parameter_u', 'parameter_mv')

static function(out, coef, grad_v, grad_u, grad_w, mtx_d, cmap)

geometries = ['2_3', '2_4', '3_4', '3_8']

get_eval_shape(mat, par_w, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, par_w, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_lin_elastic'
```

`sfepy.terms.terms_electric` module

class `sfepy.terms.terms_electric.ElectricSourceTerm`(*name, arg_str, integral, region, **kwargs*)
 Electric source term.

Definition

$$\int_{\Omega} cs(\nabla\phi)^2$$

Call signature

<code>dw_electric_source</code> (material, virtual, parameter)

Arguments

- `material` : c (electric conductivity)
- `virtual` : s (test function)
- `parameter` : ϕ (given electric potential)

```
arg_shapes = {'material': '1, 1', 'parameter': 1, 'virtual': (1, None)}

arg_types = ('material', 'virtual', 'parameter')

static function(out, grad, coef, cmap)

get_fargs(mat, virtual, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_electric_source'
```

sfePy.terms.terms_fibres module**class** sfePy.terms.terms_fibres.**FibresActiveTLTerm**(*args, **kwargs)

Hyperelastic active fibres term. Effective stress $S_{ij} = Af_{\max} \exp \left\{ -\left(\frac{\epsilon - \epsilon_{\text{opt}}}{s}\right)^2 \right\} d_i d_j$, where $\epsilon = E_{ij} d_i d_j$ is the Green strain $\underline{E}$ projected to the fibre direction $\underline{d}$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_fib_a (material_1, material_2, material_3, material_4, material_5, virtual, state)

Arguments

- material_1 : $f_{\max}$
- material_2 : ϵ_{opt}
- material_3 : s
- material_4 : $\underline{d}$
- material_5 : A
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'material_3': '1, 1',  
'material_4': 'D, 1', 'material_5': '1, 1', 'state': 'D', 'virtual': ('D',  
'state')}
```

```
arg_types = ('material_1', 'material_2', 'material_3', 'material_4', 'material_5',  
'virtual', 'state')
```

```
family_data_names = ['green_strain']
```

```
get_eval_shape(mat1, mat2, mat3, mat4, mat5, virtual, state, mode=None, term_mode=None,  
diff_var=None, **kwargs)
```

```
get_fargs(mat1, mat2, mat3, mat4, mat5, virtual, state, mode=None, term_mode=None, diff_var=None,  
**kwargs)
```

```
name = 'dw_tl_fib_a'
```

```
static stress_function(out, pars, green_strain, fibre_data=None)
```

```
static tan_mod_function(out, pars, green_strain, fibre_data=None)
```

class sfePy.terms.terms_fibres.**FibresExponentialTLTerm**(*args, **kwargs)

Hyperelastic fibres term with an exponential response. Effective stress $S_{ij} = \max(0, \sigma [\exp \{k(\epsilon - \epsilon_0)\} - 1]) d_i d_j$, where $\epsilon = E_{ij} d_i d_j$ is the Green strain $\underline{E}$ projected to the fibre direction $\underline{d}$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

```
dw_tl_fib_e (material_1, material_2, material_3, material_4, virtual, state)
```

Arguments

- material_1 : σ
- material_3 : k
- material_3 : ϵ_0
- material_4 : $\underline{d}$
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'material_3': '1, 1',
'material_4': 'D, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material_1', 'material_2', 'material_3', 'material_4', 'virtual',
'state')
```

```
family_data_names = ['green_strain']
```

```
get_eval_shape(mat1, mat2, mat3, mat4, virtual, state, mode=None, term_mode=None, diff_var=None,
**kwargs)
```

```
get_fargs(mat1, mat2, mat3, mat4, virtual, state, mode=None, term_mode=None, diff_var=None,
**kwargs)
```

```
name = 'dw_tl_fib_e'
```

```
static stress_function(out, pars, green_strain, fibre_data=None)
```

```
static tan_mod_function(out, pars, green_strain, fibre_data=None)
```

```
sfepy.terms.terms_fibres.compute_fibre_strain(green_strain, omega)
```

Compute the Green strain projected to the fibre direction.

```
sfepy.terms.terms_fibres.create_omega(fdir)
```

Create the fibre direction tensor $\omega_{ij} = d_i d_j$.

sfepy.terms.terms_flexo module

Flexoelectricity related terms.

```
class sfepy.terms.terms_flexo.MixedFlexoCouplingTerm(*args, **kwargs)
```

Flexoelectric coupling term, mixed formulation.

Definition

$$\int_{\Omega} f_{ijkl} e_{jk,l}(\underline{\delta w}) \nabla_i p$$
$$\int_{\Omega} f_{ijkl} e_{jk,l}(\underline{w}) \nabla_i q$$

Call signature

de_m_flexo_coupling	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_t, parameter_s)

Arguments 1

- material: f_{ijkl}
- virtual/parameter_t: $\underline{\delta w}$
- state/parameter_s: p

Arguments 2

- material: f_{ijkl}
- state : $\underline{w}$
- virtual : q

```
arg_shapes = [{'material': 'D, SD', 'parameter_s': 1, 'parameter_t': 'D2',  
'state/dp-w': 'D2', 'state/dw-p': 1, 'virtual/dp-w': (1, None), 'virtual/dw-p':  
( 'D2', None)}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),  
( 'material', 'parameter_t', 'parameter_s'))
```

```
get_function(mat, tvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('dw-p', 'dp-w', 'eval')
```

```
name = 'de_m_flexo_coupling'
```

```
class sfepy.terms.terms_flexo.MixedFlexoTerm(*args, **kwargs)
```

Mixed formulation displacement gradient consistency term.

Definition

$$\int_{\Omega} v_{i,j} a_{ij}$$
$$\int_{\Omega} u_{i,j} \delta a_{ij}$$

Call signature

de_m_flexo	(virtual, state)
	(state, virtual)
	(parameter_v, parameter_t)

Arguments 1

- virtual/parameter_v: $\underline{v}$

- state/parameter_t: $\underline{a}$

Arguments 2

- state : $\underline{u}$
- virtual : $\delta \underline{a}$

```
arg_shapes = [{'parameter_t': 'D2', 'parameter_v': 'D', 'state/da-u': 'D',
'state/du-a': 'D2', 'virtual/da-u': ('D2', None), 'virtual/du-a': ('D', None)},
{'opt_material': None}]
```

```
arg_types = (('virtual', 'state'), ('state', 'virtual'), ('parameter_v',
'parameter_t'))
```

```
get_function(vvar, tvar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('du-a', 'da-u', 'eval')
```

```
name = 'de_m_flexo'
```

```
class sfepy.terms.terms_flexo.MixedStrainGradElasticTerm(*args, **kwargs)
```

Flexoelectric strain gradient elasticity term, mixed formulation.

Additional evaluation modes:

- 'strain' - compute strain from the displacement gradient (state) variable.

Definition

$$\int_{\Omega} a_{ijklmn} e_{ij,k}(\delta \underline{w}) e_{lm,n}(\underline{w})$$

Call signature

de_m_sg_elastic	(material, virtual, state)
	(material, parameter_1, parameter_2)

Arguments

- material: a_{ijklmn}
- virtual/parameter_1: $\delta \underline{w}$
- state/parameter_2: $\underline{w}$

```
arg_shapes = {'material': 'SD, SD', 'parameter_1': 'D2', 'parameter_2': 'D2',
'state': 'D2', 'virtual': ('D2', 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1',
'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_m_sg_elastic'
```

```
sfepy.terms.terms_flexo.make_grad2strain(dim)
```

sfepy.terms.terms_hyperelastic_base module

```
class sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm(name, arg_str, integral,
                                                                    region, **kwargs)
```

Deformation gradient $\underline{\underline{F}}$ in quadrature points for *term_mode*='def_grad' (default) or the jacobian *J* if *term_mode*='jacobian'.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$\underline{\underline{F}} = \frac{\partial \underline{x}}{\partial \underline{X}}|_{qp} = \underline{\underline{I}} + \frac{\partial \underline{u}}{\partial \underline{X}}|_{qp},$$
$$\underline{x} = \underline{X} + \underline{u}, J = \det(\underline{\underline{F}})$$

Call signature

ev_def_grad (parameter)

Arguments

- parameter : $\underline{u}$

```
arg_shapes = {'parameter': 'D'}
```

```
arg_types = ('parameter',)
```

```
static function(out, vec, vg, econn, term_mode, fmode)
```

```
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet_extra')
```

```
name = 'ev_def_grad'
```

```
class sfepy.terms.terms_hyperelastic_base.HyperElasticBase(*args, **kwargs)
```

Base class for all hyperelastic terms in TL/UL formulation.

HyperElasticBase.__call__() computes element contributions given either stress (-> residual) or tangent modulus (-> tangent stiffness matrix), i.e. constitutive relation type (CRT) related data. The CRT data are computed in subclasses implementing particular CRT (e.g. neo-Hookean material), in *self.compute_crt_data()*.

Modes:

- 0: total formulation
- 1: updated formulation

Notes

This is not a proper Term!

```
arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
compute_stress(mat, family_data, **kwargs)
```

```
compute_tan_mod(mat, family_data, **kwargs)
```

```
static function(out, fun, *args)
```

```
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
static integrate(out, val_qp, vg, fmode)
```

```
integration = ('cell', 'facet_extra')
```

```
class sfepy.terms.terms_hyperelastic_base.HyperElasticFamilyData(**kwargs)
```

Base class for hyperelastic family data.

The common (family) data are cached in the evaluate cache of state variable.

```
data_shapes = {'det_f': ('n_el', 'n_qp', 1, 1), 'green_strain': ('n_el', 'n_qp',
'sym', 1), 'in2_b': ('n_el', 'n_qp', 1, 1), 'in2_c': ('n_el', 'n_qp', 1, 1),
'inv_f': ('n_el', 'n_qp', 'dim', 'dim'), 'mtx_f': ('n_el', 'n_qp', 'dim', 'dim'),
'sym_b': ('n_el', 'n_qp', 'sym', 1), 'sym_c': ('n_el', 'n_qp', 'sym', 1),
'sym_inv_c': ('n_el', 'n_qp', 'sym', 1), 'tr_b': ('n_el', 'n_qp', 1, 1), 'tr_c':
('n_el', 'n_qp', 1, 1)}
```

```
init_data_struct(state_shape, name='family_data')
```

sfepy.terms.terms_hyperelastic_tl module

```
class sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm(*args, **kwargs)
```

Hyperelastic bulk active term. Stress $S_{ij} = AJC_{ij}^{-1}$, where A is the activation in $[0, F_{\max}]$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_bulk_active (material, virtual, state)

Arguments

- material : A
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'sym_inv_c']  
  
name = 'dw_tl_bulk_active'  
  
static stress_function(out, mat, det_f, vec_inv_cs)  
  
static tan_mod_function(out, mat, det_f, vec_inv_cs)
```

```
class sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm(*args, **kwargs)
```

Hyperelastic bulk penalty term. Stress $S_{ij} = K(J - 1) JC_{ij}^{-1}$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

<code>dw_tl_bulk_penalty</code> (material, virtual, state)

Arguments

- material : K
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'sym_inv_c']  
  
name = 'dw_tl_bulk_penalty'  
  
static stress_function(out, mat, det_f, vec_inv_cs)  
  
static tan_mod_function(out, mat, det_f, vec_inv_cs)
```

```
class sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm(*args, **kwargs)
```

Hyperelastic bulk pressure term. Stress $S_{ij} = -p JC_{ij}^{-1}$.

Definition

$$\int_{\Omega} S_{ij}(p) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

<code>dw_tl_bulk_pressure</code> (virtual, state, state_p)

Arguments

- virtual : $\underline{v}$
- state : $\underline{u}$
- state_p : p

```
arg_geometry_types = {('state_p', None): {'facet_extra': 'facet'}}
```

```

arg_shapes = {'state': 'D', 'state_p': 1, 'virtual': ('D', 'state')}
arg_types = ('virtual', 'state', 'state_p')
compute_data(family_data, mode, **kwargs)
family_data_names = ['det_f', 'sym_inv_c']
get_eval_shape(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_tl_bulk_pressure'

static stress_function(out, pressure_qp, det_f, vec_inv_cs)
static tan_mod_u_function(out, pressure_qp, det_f, vec_inv_cs)
static weak_dp_function(out, mtx_f, vec_inv_cs, det_f, cmap_s, cmap_v, transpose, mode)
static weak_function(out, stress, tan_mod, mtx_f, det_f, cmap, is_diff, mode_ul)

class sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm(*args, **kwargs)
    Diffusion term in the total Lagrangian formulation with linearized deformation-dependent permeability  $\underline{K}(\underline{u}) = J \underline{F}^{-1} \underline{k} f(J) \underline{F}^{-T}$ , where  $\underline{u}$  relates to the previous time step ( $n - 1$ ) and  $f(J) = \max\left(0, \left(1 + \frac{(J-1)}{N_f}\right)\right)^2$  expresses the dependence on volume compression/expansion.

```

Definition

$$\int_{\Omega} \underline{K}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial \underline{X}} \frac{\partial p}{\partial \underline{X}}$$

Call signature

dw_tl_diffusion (material_1, material_2, virtual, state, parameter)

Arguments

- material_1 : $\underline{k}$
- material_2 : N_f
- virtual : q
- state : p
- parameter : $\underline{u}^{(n-1)}$

```

arg_shapes = {'material_1': 'D, D', 'material_2': '1, 1', 'parameter': 'D',
'state': 1, 'virtual': (1, 'state')}
arg_types = ('material_1', 'material_2', 'virtual', 'state', 'parameter')
family_data_names = ['mtx_f', 'det_f']
static function(out, pressure_grad, mtx_d, ref_porosity, mtx_f, det_f, cmap, mode)
get_eval_shape(permeability, ref_porosity, virtual, state, parameter, mode=None, term_mode=None,
diff_var=None, **kwargs)

```

```
get_fargs(perm, ref_porosity, virtual, state, parameter, mode=None, term_mode=None, diff_var=None,  
          **kwargs)
```

```
name = 'dw_tl_diffusion'
```

```
class sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm(*args, **kwargs)
```

Hyperelastic generalized Yeoh term [1]. Effective stress $S_{ij} = 2pK(I_1 - 3)^{p-1}J^{-\frac{2}{3}}(\delta_{ij} - \frac{1}{3}C_{kk}C_{ij}^{-1})$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

<code>dw_tl_he_genyeoh</code> (material, virtual, state)

Arguments

- material : p, K
- virtual : $\underline{v}$
- state : $\underline{u}$

[1] Travis W. Hohenberger, Richard J. Windslow, Nicola M. Pugno, James J. C. Busfield. Aconstitutive Model For Both Lowand High Strain Nonlinearities In Highly Filled Elastomers And Implementation With User-Defined Material Subroutines In Abaqus. Rubber Chemistry And Technology, Vol. 92, No. 4, Pp. 653-686 (2019)

```
arg_shapes = {'material': '1, 2', 'state': 'D', 'virtual': ('D', 'state')}
```

```
family_data_names = ['det_f', 'tr_c', 'sym_inv_c']
```

```
geometries = ['3_4', '3_8']
```

```
name = 'dw_tl_he_genyeoh'
```

```
stress_function(out, mat, *fargs, **kwargs)
```

```
tan_mod_function(out, mat, *fargs, **kwargs)
```

```
class sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTLBase(*args, **kwargs)
```

Base class for all hyperelastic surface terms in TL formulation family.

```
get_family_data = HyperElasticSurfaceTLFamilyData
```

```
class sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTLFamilyData(**kwargs)
```

Family data for TL formulation applicable for surface terms.

```
cache_name = 'tl_surface_common'
```

```
data_names = ('mtx_f', 'det_f', 'inv_f')
```

```
static family_function(mtx_f, det_f, mtx_fi, state, cmap, fis, conn)
```

```
class sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase(*args, **kwargs)
```

Base class for all hyperelastic terms in TL formulation family.

The subclasses should have the following static method attributes: - *stress_function()* (the stress) - *tan_mod_function()* (the tangent modulus)

The common (family) data are cached in the evaluate cache of state variable.

```
get_family_data = HyperElasticTLFamilyData
```

```
hyperelastic_mode = 0
```

```
static weak_function(out, stress, tan_mod, mtx_f, det_f, cmap, is_diff, mode_ul)
```

```
class sfepy.terms.terms_hyperelastic_tl.HyperElasticTLFamilyData(**kwargs)
```

Family data for TL formulation.

```
cache_name = 'tl_common'
```

```
data_names = ('mtx_f', 'det_f', 'sym_c', 'tr_c', 'in2_c', 'sym_inv_c',
               'green_strain')
```

```
static family_function(mtx_f, det_f, vec_cs, tr_c, in_2c, vec_inv_cs, vec_es, state, cmap, conn)
```

```
class sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm(*args, **kwargs)
```

Hyperelastic Mooney-Rivlin term. Effective stress $S_{ij} = \kappa J^{-\frac{4}{3}} (C_{kk} \delta_{ij} - C_{ij} - \frac{2}{3} I_2 C_{ij}^{-1})$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_he_mooney_rivlin (material, virtual, state)

Arguments

- material : κ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'tr_c', 'sym_inv_c', 'sym_c', 'in2_c']
```

```
name = 'dw_tl_he_mooney_rivlin'
```

```
static stress_function(out, mat, det_f, tr_c, vec_inv_cs, vec_cs, in_2c)
```

```
static tan_mod_function(out, mat, det_f, tr_c, vec_inv_cs, vec_cs, in_2c)
```

```
class sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm(*args, **kwargs)
```

Hyperelastic neo-Hookean term. Effective stress $S_{ij} = \mu J^{-\frac{2}{3}} (\delta_{ij} - \frac{1}{3} C_{kk} C_{ij}^{-1})$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_he_neohook (material, virtual, state)

Arguments

- material : μ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'tr_c', 'sym_inv_c']
```

```
name = 'dw_tl_he_neohook'
```

```
static stress_function(out, mat, det_f, tr_c, vec_inv_cs)
```

```
static tan_mod_function(out, mat, det_f, tr_c, vec_inv_cs)
```

```
class sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm(*args, **kwargs)
```

Single term of the hyperelastic Ogden model [1] with the strain energy density

$$W = \frac{\mu}{\alpha} (\lambda_1^\alpha + \lambda_2^\alpha + \lambda_3^\alpha - 3) ,$$

where $\lambda_k, k = 1, 2, 3$ are the principal stretches, whose squares are the principal values of the right Cauchy-Green deformation tensor $\mathbf{C}$.

Effective stress (2nd Piola-Kirchhoff) is [2]

$$S_{ij} = 2 \frac{\partial W}{\partial C_{ij}} = \sum_{k=1}^3 S^{(k)} N_i^{(k)} N_j^{(k)} ,$$

where the principal stresses are

$$S^{(k)} = J^{-2/3} \left(\mu \bar{\lambda}^{\alpha-2} - \sum_{j=1}^3 \frac{\mu}{3} \frac{\lambda_j^\alpha}{\lambda_k^2} \right) , \quad k = 1, 2, 3 .$$

and $\mathbf{N}^{(k)}, k = 1, 2, 3$ are the eigenvectors of $\mathbf{C}$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_he_ogden (material, virtual, state)

Arguments

- material : p, K
- virtual : $\underline{v}$
- state : $\underline{u}$

[1] Ogden, R. W. Large deformation isotropic elasticity - on the correlation of theory and experiment for incompressible rubberlike solids. Proceedings of the Royal Society A, Vol. 326, No. 1567, Pp. 565-584 (1972), DOI [10.1098/rspa.1972.0026](https://doi.org/10.1098/rspa.1972.0026).

[2] Steinmann, P., Hossain, M., Possart, G. Hyperelastic models for rubber-like materials: Consistent tangent operators and suitability for Treloar's data. Archive of Applied Mechanics, Vol. 82, No. 9, Pp. 1183-1217 (2012), DOI [10.1007/s00419-012-0610-z](https://doi.org/10.1007/s00419-012-0610-z).

```
arg_shapes = {'material': '1, 2', 'state': 'D', 'virtual': ('D', 'state')}
```

```
family_data_names = ['det_f', 'sym_c', 'tr_c', 'sym_inv_c']
```

```
geometries = ['3_4', '3_8']
```

```
name = 'dw_tl_he_ogden'
```

```
stress_function(out, mat, *fargs, **kwargs)
```

```
tan_mod_function(out, mat, *fargs, **kwargs)
```

```
class sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm(*args, **kwargs)
```

Surface flux term in the total Lagrangian formulation, consistent with [DiffusionTLTerm](#).

Definition

$$\int_{\Gamma} \underline{\nu} \cdot \underline{K}(\underline{u}^{(n-1)}) \frac{\partial p}{\partial \underline{X}}$$

Call signature

```
ev_tl_surface_flux (material_1, material_2, parameter_1, parameter_2)
```

Arguments

- material_1 : $\underline{k}$
- material_2 : N_f
- parameter_1 : p
- parameter_2 : $\underline{u}^{(n-1)}$

```
arg_shapes = {'material_1': 'D, D', 'material_2': '1, 1', 'parameter_1': 1,
'parameter_2': 'D'}
```

```
arg_types = ('material_1', 'material_2', 'parameter_1', 'parameter_2')
```

```
family_data_names = ['det_f', 'inv_f']
```

```
static function(out, pressure_grad, mtx_d, ref_porosity, mtx_fi, det_f, cmap, mode)
```

```
get_eval_shape(perm, ref_porosity, pressure, displacement, mode=None, term_mode=None,
diff_var=None, **kwargs)
```

```
get_fargs(perm, ref_porosity, pressure, displacement, mode=None, term_mode=None, diff_var=None,
**kwargs)
```

```
integration = 'facet_extra'
```

```
name = 'ev_tl_surface_flux'
```

```
class sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm(*args, **kwargs)
```

Surface traction term in the total Lagrangian formulation, expressed using $\underline{\nu}$, the outward unit normal vector w.r.t. the undeformed surface, $\underline{F}(\underline{u})$, the deformation gradient, $J = \det(\underline{F})$, and $\underline{\underline{\sigma}}$ a given traction, often equal to a given pressure, i.e. $\underline{\underline{\sigma}} = \pi \underline{\underline{I}}$.

Definition

$$\int_{\Gamma} \underline{\nu} \cdot \underline{F}^{-1} \cdot \underline{\underline{\sigma}} \cdot \underline{\nu} J$$

Call signature

<code>dw_tl_surface_traction</code> (opt_material, virtual, state)

Arguments

- material : $\underline{\underline{\sigma}}$
- virtual : $\underline{\nu}$
- state : $\underline{u}$

```
arg_shapes = [{'opt_material': 'D', 'D', 'state': 'D', 'virtual': ('D', 'state')},  
{ 'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual', 'state')
```

```
family_data_names = ['det_f', 'inv_f']
```

```
static function(out, traction, det_f, mtx_fi, bf, cmap, fis, mode)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
name = 'dw_tl_surface_traction'
```

```
class sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm(*args, **kwargs)
```

Volume of a D -dimensional domain, using a surface integral in the total Lagrangian formulation, expressed using $\underline{\nu}$, the outward unit normal vector w.r.t. the undeformed surface, $\underline{F}(\underline{u})$, the deformation gradient, and $J = \det(\underline{F})$. Uses the approximation of $\underline{u}$ for the deformed surface coordinates $\underline{x}$.

Definition

$$1/D \int_{\Gamma} \underline{\nu} \cdot \underline{F}^{-1} \cdot \underline{x} J$$

Call signature

<code>ev_tl_volume_surface</code> (parameter)

Arguments

- parameter : $\underline{u}$

```

arg_shapes = {'parameter': 'D'}
arg_types = ('parameter',)
family_data_names = ['det_f', 'inv_f']
static function(out, coors, det_f, mtx_fi, bf, cmap, conn)
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet_extra'
name = 'ev_tl_volume_surface'

```

```
class sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm(*args, **kwargs)
```

Volume term (weak form) in the total Lagrangian formulation.

Definition

$\int_{\Omega} q J(\underline{u})$
 volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$
 rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$

Call signature

dw_tl_volume	(virtual, state)

Arguments

- virtual : q
- state : $\underline{u}$

```

arg_geometry_types = {('virtual', None): {'facet_extra': 'facet'}}
arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f', 'sym_inv_c']
static function(out, mtx_f, vec_inv_cs, det_f, cmap_s, cmap_v, transpose, mode)
get_eval_shape(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_tl_volume'

```

sfepy.terms.terms_hyperelastic_ul module**class** sfepy.terms.terms_hyperelastic_ul.**BulkPenaltyULTerm**(*args, **kwargs)Hyperelastic bulk penalty term. Stress $\tau_{ij} = K(J - 1) J \delta_{ij}$.**Definition**

$$\int_{\Omega} \mathcal{L} \tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$$

Call signature

dw_ul_bulk_penalty (material, virtual, state)

Arguments

- material : K
- virtual : $\underline{v}$
- state : $\underline{u}$

family_data_names = ['det_f']**name** = 'dw_ul_bulk_penalty'**static stress_function**(out, mat, det_f)**static tan_mod_function**(out, mat, det_f)**class** sfepy.terms.terms_hyperelastic_ul.**BulkPressureULTerm**(*args, **kwargs)Hyperelastic bulk pressure term. Stress $S_{ij} = -p J \delta_{ij}$.**Definition**

$$\int_{\Omega} \mathcal{L} \tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$$

Call signature

dw_ul_bulk_pressure (virtual, state, state_p)

Arguments

- virtual : $\underline{v}$
- state : $\underline{u}$
- state_p : p

arg_geometry_types = {('state_p', None): {'facet_extra': 'facet'}}**arg_shapes** = {'state': 'D', 'state_p': 1, 'virtual': ('D', 'state')}**arg_types** = ('virtual', 'state', 'state_p')**compute_data**(family_data, mode, **kwargs)

```

family_data_names = ['det_f', 'sym_b']

static family_function(mtx_f, det_f, vec_bs, tr_b, in_2b, vec_es, state, cmap, conn)

get_eval_shape(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_ul_bulk_pressure'

static stress_function(out, pressure_qp, det_f)

static tan_mod_u_function(out, pressure_qp, det_f)

static weak_dp_function(out, det_f, cmap_s, cmap_v, transpose, mode)

static weak_function(out, stress, tan_mod, mtx_f, det_f, cmap, is_diff, mode_ul)

class sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm(*args, **kwargs)
Compressibility term for the updated Lagrangian formulation

```

Definition

$$\frac{\int_{\Omega} 1}{\gamma p q}$$

Call signature

dw_ul_compressible (material, virtual, state, parameter_u)

Arguments

- material : γ
- virtual : q
- state : p
- parameter_u : (u)

```

arg_geometry_types = {('state', None): {'facet_extra': 'facet'}, ('virtual',
None): {'facet_extra': 'facet'}}

arg_shapes = {'material': '1, 1', 'parameter_u': 'D', 'state': 1, 'virtual': (1,
'state')}

arg_types = ('material', 'virtual', 'state', 'parameter_u')

family_data_names = ['mtx_f', 'det_f']

static function(out, coef, val_qp, rcmap, ccmap, is_diff)

get_fargs(bulk, virtual, state, parameter_u, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_ul_compressible'

```

```
class sfepy.terms.terms_hyperelastic_ul.HyperElasticULBase(*args, **kwargs)
```

Base class for all hyperelastic terms in UL formulation family.

The subclasses should have the following static method attributes: - *stress_function()* (the stress) - *tan_mod_function()* (the tangent modulus)

```
get_family_data = HyperElasticULFamilyData
```

```
hyperelastic_mode = 1
```

```
static weak_function(out, stress, tan_mod, mtx_f, det_f, cmap, is_diff, mode_ul)
```

```
class sfepy.terms.terms_hyperelastic_ul.HyperElasticULFamilyData(**kwargs)
```

Family data for UL formulation.

```
cache_name = 'ul_common'
```

```
data_names = ('mtx_f', 'det_f', 'sym_b', 'tr_b', 'in2_b', 'green_strain')
```

```
static family_function(mtx_f, det_f, vec_bs, tr_b, in_2b, vec_es, state, cmap, conn)
```

```
class sfepy.terms.terms_hyperelastic_ul.HyperelasticByFunULTerm(*args, **kwargs)
```

General hyperelastic term: the tangent modulus and the stress tensor are calculated by a user defined function *fun*.

Definition

$$\int_{\Omega} \mathcal{L} \tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$$

Call signature

dw_ul_he_by_fun (fun, virtual, state)

Arguments

- material : *fun*
- virtual : *v*
- state : *u*

```
arg_shapes = [{'fun': <function HyperelasticByFunULTerm.<lambda>>, 'state': 'D',  
'virtual': ('D', 'state')}]
```

```
arg_types = ('fun', 'virtual', 'state')
```

```
family_data_names = ['mtx_f', 'det_f', 'sym_b', 'tr_b', 'in2_b', 'green_strain']
```

```
get_fargs(mat_fun, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_ul_he_by_fun'
```

```
class sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm(*args, **kwargs)
```

Hyperelastic Mooney-Rivlin term.

Definition

$$\int_{\Omega} \mathcal{L} \tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$$

Call signature

dw_ul_he_mooney_rivlin (material, virtual, state)

Arguments

- material : κ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'tr_b', 'sym_b', 'in2_b']
```

```
name = 'dw_ul_he_mooney_rivlin'
```

```
static stress_function(out, mat, det_f, tr_b, vec_bs, in_2b)
```

```
static tan_mod_function(out, mat, det_f, tr_b, vec_bs, in_2b)
```

```
class sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm(*args, **kwargs)
```

Hyperelastic neo-Hookean term. Effective stress $\tau_{ij} = \mu J^{-\frac{2}{3}} (b_{ij} - \frac{1}{3} b_{kk} \delta_{ij})$.

Definition

$$\int_{\Omega} \mathcal{L} \tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$$

Call signature

dw_ul_he_neohook (material, virtual, state)

Arguments

- material : μ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
family_data_names = ['det_f', 'tr_b', 'sym_b']
```

```
name = 'dw_ul_he_neohook'
```

```
static stress_function(out, mat, det_f, tr_b, vec_bs)
```

```
static tan_mod_function(out, mat, det_f, tr_b, vec_bs)
```

```
class sfepy.terms.terms_hyperelastic_ul.VolumeULTerm(*args, **kwargs)
```

Volume term (weak form) in the updated Lagrangian formulation.

Definition

$$\int_{\Omega} q J(\underline{u})$$

volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$
rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$

Call signature

dw_ul_volume (virtual, state)

Arguments

- virtual : q
- state : $\underline{u}$

```
arg_geometry_types = {('virtual', None): {'facet_extra': 'facet'}}
arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f']
static function(out, det_f, cmap_s, cmap_v, transpose, mode)
get_eval_shape(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_ul_volume'
```

sfepy.terms.terms_jax module

Proof-of-concept JAX-based terms supporting automatic differentiation.

class sfepy.terms.terms_jax.**LinearElasticLADTerm**(name, arg_str, integral, region, **kwargs)

Homogeneous isotropic linear elasticity term differentiable w.r.t. material parameters λ, μ (Lamé's parameters).

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

with

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$$

Call signature

dw_lin_elastic_l_ad (material_1, material_2, virtual, state)

Arguments

- material_1: λ (Lamé's first parameter)
- material_2: μ (Lamé's second parameter, shear modulus)
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'state': 'D',
'virtual': ('D', 'state')}
```



```

arg_types = (('material_1', 'material_2', 'virtual', 'state'),)

diff_info = {'material_1': 1, 'material_2': 1}

static function(out, fun, fargs)

get_fargs(material1, material2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)

name = 'dw_lin_elastic_l_ad'

```

```
class sfepy.terms.terms_jax.LinearElasticYPADTerm(name, arg_str, integral, region, **kwargs)
```

Homogeneous isotropic linear elasticity term differentiable w.r.t. material parameters E (Young's modulus), ν (Poisson's ratio).

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

with

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda\delta_{ij}\delta_{kl},$$

where

$$\lambda = E\nu/((1+\nu)(1-2\nu)),$$

$$\mu = E/2(1+\nu)$$

Call signature

dw_lin_elastic_yp_ad (material_1, material_2, virtual, state)

Arguments

- material_1: E
- material_2: ν
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

```

arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'state': 'D',
'virtual': ('D', 'state')}

arg_types = (('material_1', 'material_2', 'virtual', 'state'),)

diff_info = {'material_1': 1, 'material_2': 1}

static function(out, fun, fargs)

get_fargs(material1, material2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)

name = 'dw_lin_elastic_yp_ad'

```

class sfepy.terms.terms_jax.**MassADTerm**(*name, arg_str, integral, region, **kwargs*)

Homogeneous mass term differentiable w.r.t. the material parameter.

Definition

$$\int_{\mathcal{D}} \rho \underline{v} \cdot \underline{u}$$

Call signature

dw_mass_ad (material, virtual, state)

Arguments

- material_1: ρ
- virtual: $\underline{v}$
- state: $\underline{u}$

```
arg_shapes = [{ 'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state') },
               { 'state': '1', 'virtual': (1, 'state') }]
```

```
arg_types = (('material', 'virtual', 'state'),)
```

```
diff_info = { 'material': 1 }
```

```
static function(out, fun, fargs)
```

```
get_fargs(material_density, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
modes = ('weak',)
```

```
name = 'dw_mass_ad'
```

class sfepy.terms.terms_jax.**NeoHookeanTLADTerm**(*name, arg_str, integral, region, **kwargs*)

Homogeneous Hyperelastic neo-Hookean term differentiable w.r.t. the material parameter. Effective stress $S_{ij} = \mu J^{-\frac{2}{3}} (\delta_{ij} - \frac{1}{3} C_{kk} C_{ij}^{-1})$.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_he_neohook_ad (material, virtual, state)

Arguments

- material : μ
- virtual : $\underline{v}$
- state : $\underline{u}$

```

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')
diff_info = {'material': 1}

static function(out, fun, fargs)

geometries = ['2_3', '2_4', '3_4', '3_8']

get_fargs(material, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)

name = 'dw_tl_he_neohook_ad'

```

class sfepy.terms.terms_jax.OgdenTLADTerm(name, arg_str, integral, region, **kwargs)

Homogeneous hyperelastic Ogden model term differentiable w.r.t. the material parameters, with the strain energy density

$$W = \frac{\mu}{\alpha} (\bar{\lambda}_1^\alpha + \bar{\lambda}_2^\alpha + \bar{\lambda}_3^\alpha - 3) ,$$

where $\lambda_k, k = 1, 2, 3$ are the principal stretches, whose squares are the principal values of the right Cauchy-Green deformation tensor $\mathbf{C}$. For more details see [OgdenTLTerm](#).

WARNING: The current implementation fails to compute the tangent matrix when $\mathbf{C}$ has multiple eigenvalues (e.g. zero deformation). In that case nans are returned, as a result of dividing by zero. See [1], Section 11.2.3, page 385.

[1] Borst, R. de, Crisfield, M.A., Remmers, J.J.C., Verhoosel, C.V., 2012. Nonlinear Finite Element Analysis of Solids and Structures, 2nd edition. ed. Wiley, Hoboken, NJ.

Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

dw_tl_he_ogden_ad (material_mu, material_alpha, virtual, state)

Arguments

- material_1 : μ
- material_2 : α
- virtual : $\underline{v}$
- state : $\underline{u}$

```

arg_shapes = {'material_alpha': '1, 1', 'material_mu': '1, 1', 'state': 'D',
'virtual': ('D', 'state')}

arg_types = ('material_mu', 'material_alpha', 'virtual', 'state')

diff_info = {'material_alpha': 1, 'material_mu': 1}

static function(out, fun, fargs)

```

```
geometries = ['3_4', '3_8']
```

```
get_fargs(material_mu, material_alpha, virtual, state, mode=None, term_mode=None, diff_var=None,
          **kwargs)
```

```
modes = ('weak',)
```

```
name = 'dw_tl_he_ogden_ad'
```

```
sfepy.terms.terms_jax.ceval_elasticity_l(lam, mu, vbfg, ubfg, det, cu)
```

```
sfepy.terms.terms_jax.ceval_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)
```

```
sfepy.terms.terms_jax.ceval_mass(density, vbf, ubf, det, cu)
```

```
sfepy.terms.terms_jax.ceval_neohook(mu, vbfg, ubfg, det, cu)
```

```
sfepy.terms.terms_jax.ceval_neohook0(mu, vbfg, ubfg, det, cu)
```

```
sfepy.terms.terms_jax.ceval_ogden(mu, alpha, vbfg, ubfg, det, cu)
```

```
sfepy.terms.terms_jax.eval_alpha_ogden(mu, alpha, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

Original documentation:

Jacobian of `ceval_ogden` with respect to positional argument(s) 1. Takes the same arguments as `ceval_ogden` but returns the jacobian of the output with respect to the arguments at positions 1.

```
sfepy.terms.terms_jax.eval_density_mass(density, vbf, ubf, det, cu)
```

Vectorized version of `ceval_mass`. Takes similar arguments as `ceval_mass` but with additional array axes over which `ceval_mass` is mapped.

Original documentation:

Jacobian of `ceval_mass` with respect to positional argument(s) 0. Takes the same arguments as `ceval_mass` but returns the jacobian of the output with respect to the arguments at positions 0.

```
sfepy.terms.terms_jax.eval_elasticity_l(lam, mu, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

```
sfepy.terms.terms_jax.eval_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

```
sfepy.terms.terms_jax.eval_jac_elasticity_l(lam, mu, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) -1. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions -1.

```
sfepy.terms.terms_jax.eval_jac_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) -1. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions -1.

`sfepy.terms.terms_jax.eval_jac_mass(density, vbf, ubf, det, cu)`

Vectorized version of `ceval_mass`. Takes similar arguments as `ceval_mass` but with additional array axes over which `ceval_mass` is mapped.

Original documentation:

Jacobian of `ceval_mass` with respect to positional argument(s) -1. Takes the same arguments as `ceval_mass` but returns the jacobian of the output with respect to the arguments at positions -1.

`sfepy.terms.terms_jax.eval_jac_neohook(mu, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

Original documentation:

Jacobian of `ceval_neohook` with respect to positional argument(s) -1. Takes the same arguments as `ceval_neohook` but returns the jacobian of the output with respect to the arguments at positions -1.

`sfepy.terms.terms_jax.eval_jac_ogden(mu, alpha, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

Original documentation:

Jacobian of `ceval_ogden` with respect to positional argument(s) -1. Takes the same arguments as `ceval_ogden` but returns the jacobian of the output with respect to the arguments at positions -1.

`sfepy.terms.terms_jax.eval_lam_elasticity_l(lam, mu, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) 0. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions 0.

`sfepy.terms.terms_jax.eval_mass(density, vbf, ubf, det, cu)`

Vectorized version of `ceval_mass`. Takes similar arguments as `ceval_mass` but with additional array axes over which `ceval_mass` is mapped.

`sfepy.terms.terms_jax.eval_mu_elasticity_l(lam, mu, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) 1. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions 1.

`sfepy.terms.terms_jax.eval_mu_neohook(mu, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

Original documentation:

Jacobian of `ceval_neohook` with respect to positional argument(s) 0. Takes the same arguments as `ceval_neohook` but returns the jacobian of the output with respect to the arguments at positions 0.

`sfepy.terms.terms_jax.eval_mu_ogden(mu, alpha, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

Original documentation:

Jacobian of `ceval_ogden` with respect to positional argument(s) 0. Takes the same arguments as `ceval_ogden` but returns the jacobian of the output with respect to the arguments at positions 0.

`sfepy.terms.terms_jax.eval_neohook(mu, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

`sfepy.terms.terms_jax.eval_ogden(mu, alpha, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

`sfepy.terms.terms_jax.eval_poisson_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) 1. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions 1.

`sfepy.terms.terms_jax.eval_young_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)`

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) 0. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions 0.

`sfepy.terms.terms_jax.get_neohook_strain_energy(mu, C)`

`sfepy.terms.terms_jax.get_neohook_strain_energy_f(mu, F)`

`sfepy.terms.terms_jax.get_neohook_stress_1pk(mu, gu)`

`sfepy.terms.terms_jax.get_neohook_stress_2pk(mu, gu)`

`sfepy.terms.terms_jax.get_ogden_strain_energy_f(mu, alpha, F)`

`sfepy.terms.terms_jax.get_ogden_stress_1pk(mu, alpha, gu)`

`sfepy.terms.terms_jax.get_state_per_cell(term, state)`

`sfepy.terms.terms_jax.get_strain(gu)`

`sfepy.terms.terms_jax.get_stress(lam, mu, gu)`

sfepy.terms.terms_mass module

class sfepy.terms.terms_mass.**MassTerm**(*args, **kwargs)

Mass term with lumping and RMM support [1].

The lumping parameter can be ‘row_sum’, ‘hrz’ or ‘none’ (default). It applies for $\beta > 0$:

- $\beta = 0$ corresponds to the consistent mass matrix M^C ;
- $0 < \beta < 1$ corresponds to the averaged mass matrix M^A .
- $\beta = 1$ corresponds to the lumped mass matrix M^L ;

term_mode can be None (default), ‘DPM’ (diagonal projection matrix A), or ‘RMM’ (reciprocal mass matrix C).

Definition

$$M^C = \int_{\mathcal{D}} \rho \underline{v} \cdot \underline{u}$$

$$M^L = \text{lumping}(M^C)$$

$$M^A = (1 - \beta)M^C + \beta M^L$$

$$A = \sum_e A_e$$

$$C = \sum_e A_e^T (M_e^A)^{-1} A_e$$

Call signature

de_mass (material_rho, material_lumping, material_beta, virtual, state)

Arguments

- material: ρ
- material: lumping
- material: β
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

```
arg_shapes = {'material_beta': '.: 1', 'material_lumping': '.: str',
              'material_rho': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material_rho', 'material_lumping', 'material_beta', 'virtual',
             'state')
```

```
get_function(rho, lumping, beta, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_mass'
```

sfepy.terms.terms_membrane module

class sfepy.terms.terms_membrane.TLMembraneTerm(*args, **kwargs)

Mooney-Rivlin membrane with plain stress assumption.

The membrane has a uniform initial thickness h_0 and obeys a hyperelastic material law with strain energy by Mooney-Rivlin: $\Psi = a_1(I_1 - 3) + a_2(I_2 - 3)$.

Call signature

dw_tl_membrane (material_a1, material_a2, material_h0, virtual, state)

Arguments

- material_a1 : a_1
- material_a2 : a_2
- material_h0 : h_0
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_a1': '1, 1', 'material_a2': '1, 1', 'material_h0': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material_a1', 'material_a2', 'material_h0', 'virtual', 'state')
```

```
static eval_function(out, a1, a2, h0, mtx_c, c33, mtx_b, mtx_t, geo, term_mode, fmode)
```

```
static function(out, fun, *args)
```

Notes

fun is either *weak_function* or *eval_function* according to evaluation mode.

```
geometries = ['3_4', '3_8']
```

```
get_eval_shape(a1, a2, h0, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(a1, a2, h0, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'dw_tl_membrane'
```

```
static weak_function(out, a1, a2, h0, mtx_c, c33, mtx_b, mtx_t, bfg, geo, fmode)
```

sfepy.terms.terms_membrane.eval_membrane_mooney_rivlin(a1, a2, mtx_c, c33, mode)

Evaluate stress or tangent stiffness of the Mooney-Rivlin membrane.

[1] Baoguo Wu, Xingwen Du and Huifeng Tan: A three-dimensional FE nonlinear analysis of membranes, Computers & Structures 59 (1996), no. 4, 601–605.

sfepy.terms.terms_multilinear module

class sfepy.terms.terms_multilinear.**ECauchyStressTerm**(*args, **kwargs)

Evaluate Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Definition

$$\int_{\Omega} D_{ijkl} e_{kl}(\underline{w})$$

Call signature

de_cauchy_stress (material, parameter)

Arguments

- material : D_{ijkl}
- parameter : $\underline{w}$

arg_shapes = {'material': 'S, S', 'parameter': 'D'}

arg_types = ('material', 'parameter')

get_function(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'de_cauchy_stress'

class sfepy.terms.terms_multilinear.**EConvectTerm**(*args, **kwargs)

Nonlinear convective term.

Definition

$$\int_{\Omega} c((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$$

Call signature

de_convect (opt_material, virtual, state) (opt_material, parameter_1, parameter_2)

Arguments

- material: c (optional)
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))

```
get_function(coef, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_convect'
```

```
class sfepy.terms.terms_multilinear.EDiffusionTerm(*args, **kwargs)
```

General diffusion term.

Definition

$$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p$$

Call signature

de_diffusion	(material, virtual, state)
	(material, parameter_1, parameter_2)

Arguments

- material: K_{ij}
- virtual/parameter_1: q
- state/parameter_2: p

```
arg_shapes = {'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_diffusion'
```

```
class sfepy.terms.terms_multilinear.EDivGradTerm(*args, **kwargs)
```

Vector field diffusion term.

Definition

$$\int_{\Omega} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}$$

Call signature

de_div_grad	(opt_material, virtual, state)
	(opt_material, parameter_1, parameter_2)

Arguments

- material: ν (viscosity, optional)
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

```
arg_shapes = [{'opt_material': '1', '1', 'parameter_1': 'D', 'parameter_2': 'D',
'virtual': ('D', 'state')}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1',
'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_div_grad'
```

```
class sfepy.terms.terms_multilinear.EDivTerm(*args, **kwargs)
```

Weighted divergence term.

Definition

$$\int_{\Omega} \nabla \cdot \underline{v}, \int_{\Omega} c \nabla \cdot \underline{v}$$

Call signature

de_div	(opt_material, virtual)
	(opt_material, parameter)

Arguments

- material: c (optional)
- virtual/parameter: $\underline{v}$

```
arg_shapes = [{'opt_material': '1', '1', 'parameter': 'D', 'virtual': ('D', None)},
{'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual'), ('opt_material', 'parameter'))
```

```
get_function(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_div'
```

```
class sfepy.terms.terms_multilinear.EDotTerm(*args, **kwargs)
```

Volume and surface $L^2(\Omega)$ weighted dot product for both scalar and vector fields. Can be evaluated. Can use derivatives.

Definition

$$\begin{aligned} \int_{\mathcal{D}} qp, \int_{\mathcal{D}} \underline{v} \cdot \underline{u} \\ \int_{\mathcal{D}} cqp, \int_{\mathcal{D}} c\underline{v} \cdot \underline{u} \\ \int_{\mathcal{D}} \underline{v} \cdot (\underline{c}\underline{u}) \end{aligned}$$

Call signature

de_dot (opt_material, virtual, state) (opt_material, parameter_1, parameter_2)

Arguments

- material: c or $\underline{c}$ (optional)
- virtual/parameter_1: q or $\underline{v}$
- state/parameter_2: p or $\underline{u}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 1, 'parameter_2': 1,
'state': 1, 'virtual': (1, 'state')}, {'opt_material': None}, {'opt_material':
'1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D',
'state')}, {'opt_material': 'D, D'}, {'opt_material': None}, {'opt_material': '1,
1', 'parameter_1': 'N', 'parameter_2': 'N', 'state': 'N', 'virtual': ('N',
'state')}, {'opt_material': 'N, N'}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1',
'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
modes = ('weak', 'eval')
```

```
name = 'de_dot'
```

```
class sfepy.terms.terms_multilinear.EGradTerm(*args, **kwargs)
```

Weighted gradient term.

Definition

$$\int_{\Omega} \nabla \underline{v}, \int_{\Omega} c \nabla \underline{v}, \int_{\Omega} \underline{c} \cdot \nabla \underline{v}, \int_{\Omega} \underline{\underline{c}} \cdot \nabla \underline{v}$$

Call signature

de_grad (opt_material, parameter)

Arguments

- material: c (optional)
- virtual/parameter: $\underline{v}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter': 'N'}, {'opt_material': 'N,
1'}, {'opt_material': 'N, N'}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter')
```

```
get_function(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'de_grad'
```

class sfepy.terms.terms_multilinear.**EIntegrateOperatorTerm**(*args, **kwargs)

Volume and surface integral of a test function weighted by a scalar function c .

Definition

$$\int_{\mathcal{D}} q \text{ or } \int_{\mathcal{D}} cq$$

Call signature

de_integrate (opt_material, virtual)

Arguments

- material : c (optional)
- virtual : q

```
arg_shapes = [{'opt_material': '1, 1', 'virtual': (1, None)}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual')
```

```
get_function(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
```

```
name = 'de_integrate'
```

class sfepy.terms.terms_multilinear.**ELaplaceTerm**(*args, **kwargs)

Laplace term with c coefficient. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} \nabla q \cdot \nabla p, \int_{\Omega} c \nabla q \cdot \nabla p$$

Call signature

de_laplace (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

Arguments

- material: c
- virtual/parameter_1: q
- state/parameter_2: p

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_laplace'
```

```
class sfepy.terms.terms_multilinear.ELinearConvectTerm(*args, **kwargs)
```

Linearized convective term.

Definition

$$\int_{\Omega} ((\underline{w} \cdot \nabla) \underline{u}) \cdot \underline{v}$$

Call signature

de_lin_convect	(virtual, parameter, state)
	(parameter_1, parameter_2, parameter_3)

Arguments

- virtual/parameter_1: $\underline{v}$
- parameter/parameter_2: $\underline{w}$
- state/parameter_3: $\underline{u}$

```
arg_shapes = {'parameter': 'D', 'parameter_1': 'D', 'parameter_2': 'D',  
'parameter_3': 'D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = (('virtual', 'parameter', 'state'), ('parameter_1', 'parameter_2',  
'parameter_3'))
```

```
get_function(virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_lin_convect'
```

```
class sfepy.terms.terms_multilinear.ELinearElasticTerm(*args, **kwargs)
```

General linear elasticity term, with D_{ijkl} given in the usual matrix form exploiting symmetry: in 3D it is 6×6 with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is 3×3 with the indices ordered as [11, 22, 12].

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

Call signature

de_lin_elastic	(material, virtual, state)
	(material, parameter_1, parameter_2)

Arguments

- material: D_{ijkl}
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

```

arg_shapes = {'material': 'S, S', 'parameter_1': 'D', 'parameter_2': 'D',
'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1',
'parameter_2'))

get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_lin_elastic'

```

class sfepy.terms.terms_multilinear.**ELinearTractionTerm**(*args, **kwargs)

Linear traction term. The material parameter can have one of the following shapes:

- 1 or (1, 1) - a given scalar pressure
- (D, 1) - a traction vector
- (S, 1) or (D, D) - a given stress in symmetric or non-symmetric tensor storage (in symmetric storage indicies are order as follows: 2D: [11, 22, 12], 3D: [11, 22, 33, 12, 13, 23])

Definition

$$\int_{\Gamma} \underline{v} \cdot \underline{n}, \int_{\Gamma} c \underline{v} \cdot \underline{n}$$

$$\int_{\Gamma} \underline{v} \cdot (\underline{\sigma} \underline{n}), \int_{\Gamma} \underline{v} \cdot \underline{f}$$

Call signature

de_surface_ltr	(opt_material, virtual)
	(opt_material, parameter)

Arguments

- material: c , $\underline{f}$, $\underline{\sigma}$ or $\underline{\underline{\sigma}}$
- virtual/parameter: $\underline{v}$

```

arg_shapes = [{'opt_material': 'S, 1', 'parameter': 'D', 'virtual': ('D', None)},
{'opt_material': None}, {'opt_material': '1, 1'}, {'opt_material': 'D, 1'},
{'opt_material': 'D, D'}]

arg_types = (('opt_material', 'virtual'), ('opt_material', 'parameter'))

get_function(traction, vvar, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'de_surface_ltr'

```

class sfepy.terms.terms_multilinear.**ENonPenetrationPenaltyTerm**(*args, **kwargs)

Non-penetration condition in the weak sense using a penalty.

Definition

$$\int_{\Gamma} c(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u})$$

Call signature

de_non_penetration_p (material, virtual, state)

Arguments

- material : c
- virtual : $\underline{v}$
- state : $\underline{u}$

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material', 'virtual', 'state')

get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'de_non_penetration_p'

class sfepy.terms.terms_multilinear.**ENonSymElasticTerm**(*args, **kwargs)

Elasticity term with non-symmetric gradient. The indices of matrix D_{ijkl} are ordered as [11, 12, 13, 21, 22, 23, 31, 32, 33] in 3D and as [11, 12, 21, 22] in 2D.

Definition

$$\int_{\Omega} \underline{\underline{D}} \nabla \underline{v} : \nabla \underline{u}$$

Call signature

de_nonsym_elastic (material, virtual, state)
(material, parameter_1, parameter_2)

Arguments

- material: $\underline{\underline{D}}$
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

arg_shapes = {'material': 'D2, D2', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))


```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_nonsym_elastic'
```

```
class sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm(*args, **kwargs)
```

Volume dot product of a scalar gradient dotted with a material vector with a scalar.

Definition

$$\int_{\Omega} q \underline{y} \cdot \nabla p, \int_{\Omega} p \underline{y} \cdot \nabla q$$

Call signature

de_s_dot_mgrad_s	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_1, parameter_2)

Arguments 1

- material : $\underline{y}$
- virtual : q
- state : p

Arguments 2

- material : $\underline{y}$
- state : p
- virtual : q

```
arg_shapes = [{'material': 'D, 1', 'parameter_1': 1, 'parameter_2': 1,
'state/grad_state': 1, 'state/grad_virtual': 1, 'virtual/grad_state': (1, None),
'virtual/grad_virtual': (1, None)}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),
('material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad_state', 'grad_virtual', 'eval')
```

```
name = 'de_s_dot_mgrad_s'
```

```
class sfepy.terms.terms_multilinear.EStokesTerm(*args, **kwargs)
```

Stokes problem coupling term. Corresponds to weak forms of gradient and divergence terms.

Definition

$$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u}$$

$$\int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$$

Call signature

de_stokes	(opt_material, virtual, state)
	(opt_material, state, virtual)
	(opt_material, parameter_v, parameter_s)

Arguments 1

- material: c (optional)
- virtual/parameter_v: v
- state/parameter_s: p

Arguments 2

- material : c (optional)
- state : u
- virtual : q

```
arg_shapes = [{'opt_material': '1', '1', 'parameter_s': 1, 'parameter_v': 'D',
'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad':
('D', None)}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state',
'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))
```

```
get_function(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'de_stokes'
```

```
class sfepy.terms.terms_multilinear.ETermBase(*args, **kwargs)
```

Reserved letters:

c .. cells q .. quadrature points d-h .. DOFs axes r-z .. auxiliary axes

Layout specification letters:

c .. cells q .. quadrature points v .. variable component - matrix form (v, d) -> vector v*d g .. gradient component

d .. local DOF (basis, node) 0 .. all material axes

```
build_expression(expr, *eargs, mode=None, diff_var=None)
```

```

can_backend = {'dask_single': <module 'dask.array' from
'/home/eldaran/.local/lib/python3.10/site-packages/dask/array/__init__.py'>,
'dask_threads': <module 'dask.array' from
'/home/eldaran/.local/lib/python3.10/site-packages/dask/array/__init__.py'>, 'jax':
<module 'jax.numpy' from
'/home/eldaran/.local/lib/python3.10/site-packages/jax/numpy/__init__.py'>,
'jax_vmap': <module 'jax.numpy' from
'/home/eldaran/.local/lib/python3.10/site-packages/jax/numpy/__init__.py'>, 'numpy':
<module 'numpy' from
'/home/eldaran/.local/lib/python3.10/site-packages/numpy/__init__.py'>,
'numpy_loop': <module 'numpy' from
'/home/eldaran/.local/lib/python3.10/site-packages/numpy/__init__.py'>,
'numpy_qloop': <module 'numpy' from
'/home/eldaran/.local/lib/python3.10/site-packages/numpy/__init__.py'>,
'opt_einsum': <module 'opt_einsum' from
'/home/eldaran/.local/lib/python3.10/site-packages/opt_einsum/__init__.py'>,
'opt_einsum_dask_single': <module 'dask.array' from
'/home/eldaran/.local/lib/python3.10/site-packages/dask/array/__init__.py'>,
'opt_einsum_dask_threads': <module 'dask.array' from
'/home/eldaran/.local/lib/python3.10/site-packages/dask/array/__init__.py'>,
'opt_einsum_loop': <module 'opt_einsum' from
'/home/eldaran/.local/lib/python3.10/site-packages/opt_einsum/__init__.py'>,
'opt_einsum_qloop': <module 'opt_einsum' from
'/home/eldaran/.local/lib/python3.10/site-packages/opt_einsum/__init__.py'>}}

clear_cache()

eval_complex(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

static function_silent(out, eval_einsum, *args)

static function_timer(out, eval_einsum, *args)

get_eval_shape(*args, **kwargs)

get_fargs(*args, **kwargs)

get_normals(arg)

get_operands(diff_var)

get_paths(expressions, operands)

layout_letters = 'cqqvvd0'

make_function(expr, *args, mode=None, diff_var=None)

print_expressions(diff_var=None)

set_backend(backend='numpy', optimize=True, layout=None, **kwargs)

set_verbosity(verbosity=None)

verbosity = 0

```

```
class sfepy.terms.terms_multilinear.ExpressionArg(**kwargs)
    static from_term_arg(arg, term)
    get_bf(expr_cache, cell_dependent=False)
    get_dofs(cache, expr_cache, oname)

class sfepy.terms.terms_multilinear.ExpressionBuilder(n_add, cache)
    add_arg_dofs(iin, ein, name, n_components, iia=None)
    add_bf(iin, ein, name, cell_dependent=False)
    add_bfg(iin, ein, name)
    add_cell_scalar(name, cname)
    add_eye(iic, ein, name, iia=None)
    add_material_arg(arg, ii, ein)
    add_psg(iic, ein, name, iia=None)
    add_pvg(iic, ein, name, iia=None)
    add_qp_scalar(name, cname)
    add_state_arg(arg, ii, ein, modifier, diff_var)
    add_virtual_arg(arg, ii, ein, modifier)
    apply_layout(layout, operands, defaults=None, verbosity=0)
    build(texpr, *args, mode=None, diff_var=None)
    get_expressions(subscripts=None)
    static join_subscripts(subscripts, out_subscripts)
    letters = 'defgh'
    make_eye(size)
    make_psg(dim)
    make_pvg(dim)
    print_shapes(subscripts, operands)
    transform(subscripts, operands, transformation='loop', **kwargs)

class sfepy.terms.terms_multilinear.StokesTractionTerm(*args, **kwargs)
    Surface traction term for Stokes flow problem.
```

Definition

$$\int_{\Gamma} \underline{\nu} \underline{\nu} \cdot (\nabla \underline{u} \cdot \underline{n})$$

Call signature

de_stokes_traction	(opt_material, virtual, state)
	(opt_material, parameter_1, parameter_2)

Arguments

- material: ν (viscosity, optional)
- virtual/parameter_1: v
- state/parameter_2: u

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D',
'state': 'D', 'virtual': ('D', 'state')}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1',
'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
modes = ('weak', 'eval')
```

```
name = 'de_stokes_traction'
```

```
class sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm(*args, **kwargs)
```

Surface flux operator term.

Definition

$$\int_{\Gamma} q \underline{n} \cdot \underline{K} \cdot \nabla p, \int_{\Gamma} p \underline{n} \cdot \underline{K} \cdot \nabla q$$

Call signature

de_surface_flux	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_1, parameter_2)

Arguments 1

- material : $\underline{K}$
- virtual : q
- state : p

Arguments 2

- material : $\underline{K}$
- state : p
- virtual : q

```
arg_shapes = [{'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1,
'state/grad_state': 1, 'state/grad_virtual': 1, 'virtual/grad_state': (1, None),
'virtual/grad_virtual': (1, None)}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),
('material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
modes = ('grad_state', 'grad_virtual', 'eval')
```

```
name = 'de_surface_flux'
```

```
class sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm(*args, **kwargs)
```

Surface piezoelectric flux operator term.

Corresponds to the electric flux due to mechanically induced electrical displacement.

Definition

$$\int_{\Gamma} q g_{kij} e_{ij}(\underline{u}) n_k, \int_{\Gamma} p g_{kij} e_{ij}(\underline{v}) n_k$$

Call signature

de_surface_piezo_flux	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_1, parameter_2)

Arguments 1

- material : g_{kij}
- virtual/parameter_1 : q
- state/parameter_2 : $\underline{u}$

Arguments 2

- material : g_{kij}
- state : p
- virtual : $\underline{v}$

```
arg_shapes = [{'material': 'D, S', 'parameter_1': 1, 'parameter_2': 'D',
'state/grad_state': 'D', 'state/grad_virtual': 1, 'virtual/grad_state': (1,
None), 'virtual/grad_virtual': ('D', None)}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),
('material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet_extra'
```

```
modes = ('grad_state', 'grad_virtual', 'eval')
```

```
name = 'de_surface_piezo_flux'
```

class sfepy.terms.terms_multilinear.**SurfacePiezoFluxTerm**(*args, **kwargs)

Surface piezoelectric flux term.

Corresponds to the electric flux due to mechanically induced electrical displacement.

Definition

$$\int_{\Gamma} g_{kij} e_{ij}(\underline{u}) n_k$$

Call signature

ev_surface_piezo_flux (material, parameter)

Arguments 1

- material : g_{kij}
- parameter : $\underline{u}$

arg_shapes = {'material': 'D, S', 'parameter': 'D'}

arg_types = ('material', 'parameter')

get_function(mat, var1, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet_extra'

modes = ('eval',)

name = 'ev_surface_piezo_flux'

sfepy.terms.terms_multilinear.**append_all**(seqs, item, ii=None)

sfepy.terms.terms_multilinear.**collect_modifiers**(modifiers)

sfepy.terms.terms_multilinear.**find_free_indices**(indices)

sfepy.terms.terms_multilinear.**get_einsum_ops**(eargs, ebuilder, expr_cache)

sfepy.terms.terms_multilinear.**get_loop_indices**(subs, loop_index)

sfepy.terms.terms_multilinear.**get_output_shape**(out_subscripts, subscripts, operands)

sfepy.terms.terms_multilinear.**get_sizes**(indices, operands)

sfepy.terms.terms_multilinear.**get_slice_ops**(subs, ops, loop_index)

sfepy.terms.terms_multilinear.**parse_term_expression**(texpr)

sfepy.terms.terms_multilinear.**sym2nonsym**(sym_obj, axes=[3])

sfepy.terms.terms_navier_stokes module**class** sfepy.terms.terms_navier_stokes.**ConvectTerm**(*name, arg_str, integral, region, **kwargs*)

Nonlinear convective term.

Definition

$$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$$

Call signature

dw_convect (virtual, state)

Arguments

- virtual : $\underline{v}$
- state : $\underline{u}$

arg_shapes = {'state': 'D', 'virtual': ('D', 'state')}**arg_types** = ('virtual', 'state')**static function**(*out, grad, state, cmap, is_diff*)**get_fargs**(*virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs*)**name** = 'dw_convect'**class** sfepy.terms.terms_navier_stokes.**DivGradTerm**(*name, arg_str, integral, region, **kwargs*)

Diffusion term.

Definition

$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nabla \underline{v} : \nabla \underline{u}$$

Call signature

dw_div_grad (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

Arguments

- material: ν (viscosity, optional)
- virtualparameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$

arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}, {'opt_material': None}]**arg_types** = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))


```

d_div_grad(out, grad1, grad2, mat, vg, fmode)

static function(out, grad, viscosity, cmap_v, cmap_s, is_diff)

get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'dw_div_grad'

set_arg_types()

```

```
class sfepy.terms.terms_navier_stokes.DivOperatorTerm(name, arg_str, integral, region, **kwargs)
```

Weighted divergence term of a test function.

Definition

$$\int_{\Omega} \nabla \cdot \underline{v} \text{ or } \int_{\Omega} c \nabla \cdot \underline{v}$$

Call signature

dw_div (opt_material, virtual)

Arguments

- material : c (optional)
- virtual : $\underline{v}$

```
arg_shapes = [{'opt_material': '1, 1', 'virtual': ('D', None)}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual')
```

```
static function(out, mat, vg)
```

```
get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_div'
```

```
class sfepy.terms.terms_navier_stokes.DivTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate divergence of a vector field.

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\mathcal{D}} \nabla \cdot \underline{u}, \int_{\mathcal{D}} c \nabla \cdot \underline{u}$$

Call signature

ev_div (opt_material, parameter)

Arguments

- parameter : $\underline{u}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter': 'D'}, {'opt_material': None}]
arg_types = ('opt_material', 'parameter')
static function(out, mat, div, vg, fmode)
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = ('cell', 'facet_extra')
name = 'ev_div'
```

```
class sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm(name, arg_str, integral, region,
                                                                **kwargs)
```

Grad-div stabilization term (γ is a global stabilization parameter).

Definition

$$\gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v})$$

Call signature

dw_st_grad_div (material, virtual, state)

Arguments

- material : γ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')
static function(out, div, coef, cmap, is_diff)
get_fargs(gamma, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_st_grad_div'
```

```
class sfepy.terms.terms_navier_stokes.GradTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate gradient of a scalar or vector field.

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\mathcal{D}} \nabla p \text{ or } \int_{\mathcal{D}} \nabla \underline{u}$$
$$\int_{\mathcal{D}} c \nabla p \text{ or } \int_{\mathcal{D}} c \nabla \underline{u}$$

Call signature

ev_grad (opt_material, parameter)

Arguments

- parameter : p or $\underline{u}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter': 'N'}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter')
```

```
static function(out, mat, grad, vg, fmode)
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet_extra')
```

```
name = 'ev_grad'
```

```
class sfepy.terms.terms_navier_stokes.LinearConvect2Term(name, arg_str, integral, region,
                                                         **kwargs)
```

Linearized convective term with the convection velocity given as a material parameter.

Definition

$$\int_{\Omega} ((\underline{c} \cdot \nabla) \underline{u}) \cdot \underline{v} \\ ((\underline{c} \cdot \nabla) \underline{u})|_{qp}$$

Call signature

```
dw_lin_convect2 (material, virtual, state)
```

Arguments

- material : $\underline{c}$
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': 'D, 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function(out, grad, state_b, cmap, is_diff)
```

```
get_fargs(material, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_convect2'
```

```
class sfepy.terms.terms_navier_stokes.LinearConvectTerm(name, arg_str, integral, region, **kwargs)
```

Linearized convective term.

Definition

$$\int_{\Omega} ((\underline{w} \cdot \nabla) \underline{u}) \cdot \underline{v} \\ ((\underline{w} \cdot \nabla) \underline{u})|_{qp}$$

Call signature

dw_lin_convect (virtual, parameter, state)

Arguments

- virtual : $\underline{v}$
- parameter : $\underline{u}$
- state : $\underline{u}$

```
arg_shapes = {'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('virtual', 'parameter', 'state')
```

```
static function(out, grad, state_b, cmap, is_diff)
```

```
get_fargs(virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_convect'
```

```
class sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

PSPG stabilization term, convective part (τ is a local stabilization parameter).

Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$$

Call signature

dw_st_pspg_c (material, virtual, parameter, state)

Arguments

- material : τ_K
- virtual : q
- parameter : $\underline{b}$
- state : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'virtual': (1, None)}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function(out, state_b, state_u, coef, cmap_p, cmap_u, conn, is_diff)
```

```
get_fargs(tau, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_pspg_c'
```

```
class sfepy.terms.terms_navier_stokes.PSPGPStabilizationTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

PSPG stabilization term, pressure part (τ is a local stabilization parameter), alias to Laplace term `dw_laplace`.

Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q$$

Call signature

dw_st_pspg_p (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

Arguments

- material : τ_K
- virtual : q
- state : p

```
name = 'dw_st_pspg_p'
```

```
class sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

SUPG stabilization term, convective part (δ is a local stabilization parameter).

Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v})$$

Call signature

dw_st_supg_c (material, virtual, parameter, state)

Arguments

- material : δ_K
- virtual : $\underline{v}$
- parameter : $\underline{b}$
- state : $\underline{u}$

```
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'virtual':
('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function(out, state_b, state_u, coef, cmap, conn, is_diff)
```

```
get_fargs(delta, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_supg_c'
```

```
class sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm(name, arg_str, integral, region,
                                                             **kwargs)
```

SUPG stabilization term, pressure part (δ is a local stabilization parameter).

Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v})$$

Call signature

dw_st_supg_p (material, virtual, parameter, state)

Arguments

- material : δ_K
- virtual : $\underline{v}$
- parameter : $\underline{b}$
- state : p

```
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 1, 'virtual': ('D',
None)}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function(out, state_b, grad_p, coef, cmap_u, cmap_p, is_diff)
```

```
get_fargs(delta, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_supg_p'
```

```
class sfepy.terms.terms_navier_stokes.StokesTerm(name, arg_str, integral, region, **kwargs)
```

Stokes problem coupling term. Corresponds to weak forms of gradient and divergence terms. Can be evaluated.

Definition

$$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u}$$

or

$$\int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$$

Call signature

dw_stokes (opt_material, virtual, state)
(opt_material, state, virtual)
(opt_material, parameter_v, parameter_s)

Arguments 1

- material: c (optional)
- virtual/parameter_v: $\underline{v}$
- state/parameter_s: p

Arguments 2

- material : c (optional)
- state : $\underline{u}$
- virtual : q

```
arg_shapes = [{'opt_material': '1', '1', 'parameter_s': 1, 'parameter_v': 'D',
'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad':
('D', None)}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state',
'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))
```

```
static d_eval(out, coef, vec_qp, div, vvg)
```

```
get_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'dw_stokes'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_navier_stokes.StokesWaveDivTerm(name, arg_str, integral, region, **kwargs)
```

Stokes dispersion term with the wave vector $\underline{\kappa}$ and the divergence operator.

Definition

$$\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\nabla \cdot \underline{u}) , \int_{\Omega} (\underline{\kappa} \cdot \underline{u})(\nabla \cdot \underline{v})$$

Call signature

dw_stokes_wave_div	(material, virtual, state)
	(material, state, virtual)

Arguments 1

- material : $\underline{\kappa}$
- virtual : $\underline{v}$
- state : $\underline{u}$

Arguments 2

- material : $\underline{\kappa}$
- state : $\underline{u}$
- virtual : $\underline{v}$

```
arg_shapes = {'material': '.: D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'))
```

```
static function(out, out_qp, geo, fmode)
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_fargs(kappa, kvar, dvar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('kd', 'dk')
```

```
name = 'dw_stokes_wave_div'
```

```
class sfepy.terms.terms_navier_stokes.StokesWaveTerm(name, arg_str, integral, region, **kwargs)
```

Stokes dispersion term with the wave vector $\underline{\kappa}$.

Definition

$$\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\underline{\kappa} \cdot \underline{u})$$

Call signature

dw_stokes_wave (material, virtual, state)

Arguments

- material : $\underline{\kappa}$
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': 'D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function(out, out_qp, geo, fmode)
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_fargs(kappa, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_stokes_wave'
```

sfepy.terms.terms_piezo module

```
class sfepy.terms.terms_piezo.PiezoCouplingTerm(name, arg_str, integral, region, **kwargs)
```

Piezoelectric coupling term. Can be evaluated.

Definition

$$\int_{\Omega} g_{kij} e_{ij}(\underline{v}) \nabla_k p$$
$$\int_{\Omega} g_{kij} e_{ij}(\underline{u}) \nabla_k q$$

Call signature

dw_piezo_coupling (material, virtual, state)
(material, state, virtual)
(material, parameter_v, parameter_s)

Arguments 1

- material: g_{kij}
- virtual/parameter_v: $\underline{v}$
- state/parameter_s: p

Arguments 2

- material : g_{kij}
- state : $\underline{u}$
- virtual : q

```
arg_shapes = {'material': 'D, S', 'parameter_s': 1, 'parameter_v': 'D',
              'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad':
              ('D', None)}
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'),
             ('material', 'parameter_v', 'parameter_s'))
```

```
get_eval_shape(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'dw_piezo_coupling'
```

```
set_arg_types()
```

```
class sfepy.terms.terms_piezo.PiezoStrainTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate piezoelectric strain tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

$$\int_{\Omega} g_{kij} e_{ij}(\underline{u})$$

Call signature

ev_piezo_strain (material, parameter)

Arguments

- material : g_{kij}
- parameter : $\underline{u}$

```
arg_shapes = {'material': 'D, S', 'parameter': 'D'}
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_piezo_strain'
```

```
class sfepy.terms.terms_piezo.PiezoStressTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate piezoelectric stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

$$\int_{\Omega} g_{kij} \nabla_k p$$

Call signature

ev_piezo_stress (material, parameter)

Arguments

- material : g_{kij}
- parameter : p

```
arg_shapes = {'material': 'D, S', 'parameter': '1'}
```

```
arg_types = ('material', 'parameter')
```

```
static function(out, val_qp, vg, fmode)
```

```
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_piezo_stress'
```

```
class sfepy.terms.terms_piezo.SDPiezoCouplingTerm(*args, **kwargs)
```

Sensitivity (shape derivative) of the piezoelectric coupling term.

Definition

$$\int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{u}) \nabla_k p$$
$$\hat{g}_{kij} = g_{kij} (\nabla \cdot \underline{\mathcal{V}}) - g_{kil} \frac{\partial \mathcal{V}_j}{\partial x_l} - g_{lij} \frac{\partial \mathcal{V}_k}{\partial x_l}$$

Call signature

ev_sd_piezo_coupling (material, parameter_u, parameter_p, parameter_mv)

Arguments

- material : g_{kij}
- parameter_u : $\underline{u}$
- parameter_p : p
- parameter_mv : $\underline{\mathcal{V}}$

```

arg_shapes = {'material': 'D, S', 'parameter_mv': 'D', 'parameter_p': 1,
'parameter_u': 'D'}

arg_types = ('material', 'parameter_u', 'parameter_p', 'parameter_mv')

geometries = ['2_3', '2_4', '3_4', '3_8']

get_function(mat, par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_piezo_coupling'

```

sfepy.terms.terms_point module

class sfepy.terms.terms_point.ConcentratedPointLoadTerm(name, arg_str, integral, region, **kwargs)
 Concentrated point load term.

The load value must be given in form of a special material parameter (name prefixed with '.'), e.g. (in 2D):

```
'load' : ({'.val' : [0.0, 1.0]},)
```

This term should be used with special care, as it bypasses the usual evaluation in quadrature points. It should only be used with nodal FE basis. The number of rows of the load must be equal to the number of nodes in the region and the number of columns equal to the field dimension.

Definition

$$\underline{f}^i = \bar{f}^i \quad \forall \text{ FE node } i \text{ in a region}$$

Call signature

```
dw_point_load (material, virtual)
```

Arguments

- material : $\underline{f}^i$
- virtual : $\underline{v}$,

```

arg_shapes = {'material': '.: N', 'virtual': ('N', None)}

arg_types = ('material', 'virtual')

static function(out, mat)

get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'point'

name = 'dw_point_load'

```

class sfepy.terms.terms_point.LinearPointSpringTerm(name, arg_str, integral, region, **kwargs)
 Linear springs constraining movement of FE nodes in a region; to use as a relaxed Dirichlet boundary conditions.

Definition

$$\underline{f}^i = -k\underline{u}^i \quad \forall \text{ FE node } i \text{ in a region}$$

Call signature

dw_point_lspring (material, virtual, state)

Arguments

- material : k
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material': '.: 1', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function(out, stiffness, vec, diff_var)
```

```
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'point'
```

```
name = 'dw_point_lspring'
```

sfepy.terms.terms_sensitivity module

```
class sfepy.terms.terms_sensitivity.ESDDiffusionTerm(*args, **kwargs)
```

Diffusion sensitivity analysis term.

Definition

$$\int_{\Omega} \hat{K}_{ij} \nabla_i q \nabla_j p$$
$$\hat{K}_{ij} = K_{ij} \left(\delta_{ik} \delta_{jl} \nabla \cdot \underline{\mathcal{V}} - \delta_{ik} \frac{\partial \mathcal{V}_j}{\partial x_l} - \delta_{jl} \frac{\partial \mathcal{V}_i}{\partial x_k} \right)$$

Call signature

de_sd_diffusion (material, virtual, state, parameter_mv)
(material, parameter_1, parameter_2, parameter_mv)

Arguments

- material: K_{ij}
- virtual/parameter_1: q
- state/parameter_2: p
- parameter_mv: $\underline{\mathcal{V}}$

```
arg_shapes = {'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1,  
'parameter_mv': 'D', 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = (('material', 'virtual', 'state', 'parameter_mv'), ('material',  
'parameter_1', 'parameter_2', 'parameter_mv'))
```

```

get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_sd_diffusion'

```

```
class sfepy.terms.terms_sensitivity.ESDDivGradTerm(*args, **kwargs)
```

Sensitivity (shape derivative) of diffusion term *de_div_grad*.

Definition

$$\int_{\Omega} \hat{I} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \hat{I} \nabla \underline{v} : \nabla \underline{u}$$

$$\hat{I}_{ijkl} = \delta_{ik} \delta_{jl} \nabla \cdot \underline{\mathcal{V}} - \delta_{ik} \delta_{js} \frac{\partial \mathcal{V}_l}{\partial x_s} - \delta_{is} \delta_{jl} \frac{\partial \mathcal{V}_k}{\partial x_s}$$

Call signature

de_sd_div_grad	(opt_material, virtual, state, parameter_mv)
	(opt_material, parameter_1, parameter_2, parameter_mv)

Arguments

- material: ν (viscosity, optional)
- virtual/parameter_1: $\underline{v}$
- state/parameter_2: $\underline{u}$
- parameter_mv: $\underline{\mathcal{V}}$

```

arg_shapes = [{'opt_material': '1', '1': '1', 'parameter_1': 'D', 'parameter_2': 'D',
'parameter_mv': 'D', 'state': 'D', 'virtual': ('D', 'state')}, {'opt_material':
None}]

```

```

arg_types = (('opt_material', 'virtual', 'state', 'parameter_mv'), ('opt_material',
'parameter_1', 'parameter_2', 'parameter_mv'))

```

```
get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_sd_div_grad'
```

```
class sfepy.terms.terms_sensitivity.ESDDotTerm(*args, **kwargs)
```

Sensitivity (shape derivative) of dot product of scalars or vectors.

Definition

$$\int_{\Omega} qp(\nabla \cdot \underline{\mathcal{V}}), \int_{\Omega} (\underline{v} \cdot \underline{u})(\nabla \cdot \underline{\mathcal{V}})$$

$$\int_{\Omega} cqp(\nabla \cdot \underline{\mathcal{V}}), \int_{\Omega} c(\underline{v} \cdot \underline{u})(\nabla \cdot \underline{\mathcal{V}})$$

$$\int_{\Omega} \underline{v} \cdot (\underline{M} \underline{u})(\nabla \cdot \underline{\mathcal{V}})$$

Call signature

de_sd_dot	(opt_material, virtual, state, parameter_mv)
	(opt_material, parameter_1, parameter_2, parameter_mv)

Arguments

- material: c or $\underline{\underline{M}}$ (optional)
- virtual/parameter_1: q or $\underline{v}$
- state/parameter_2: p or $\underline{u}$
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_1': 1, 'parameter_2': 1,
'parameter_mv': 'D', 'state': 1, 'virtual': (1, 'state')}, {'opt_material':
None}, {'opt_material': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D',
'parameter_mv': 'D', 'state': 'D', 'virtual': ('D', 'state')}, {'opt_material':
'D, D'}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state', 'parameter_mv'), ('opt_material',
'parameter_1', 'parameter_2', 'parameter_mv'))
```

```
get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'de_sd_dot'
```

```
class sfepy.terms.terms_sensitivity.ESDLinearElasticTerm(*args, **kwargs)
```

Sensitivity analysis of the linear elastic term. D_{ijkl} can be given in symmetric or non-symmetric form.

Definition

$$\int_{\Omega} \hat{D}_{ijkl} \frac{\partial v_i}{\partial x_j} \frac{\partial u_k}{\partial x_l}$$

$$\hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \underline{\mathcal{V}}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$$

Call signature

de_sd_lin_elastic	(material, virtual, state, parameter_mv)
	(material, parameter_1, parameter_2, parameter_mv)

Arguments 1

- material : $\underline{\underline{D}}$
- virtual/parameter_v : $\underline{v}$
- state/parameter_s : $\underline{u}$
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = [{'material': 'S, S', 'parameter_1': 'D', 'parameter_2': 'D',
'parameter_mv': 'D', 'state': 'D', 'virtual': ('D', 'state')}, {'material': 'D2,
D2'}]
```

```

arg_types = (('material', 'virtual', 'state', 'parameter_mv'), ('material',
'parameter_1', 'parameter_2', 'parameter_mv'))

geometries = ['2_3', '2_4', '3_4', '3_8']

get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_sd_lin_elastic'

```

```
class sfepy.terms.terms_sensitivity.ESDLinearTractionTerm(*args, **kwargs)
```

Sensitivity of the linear traction term.

Definition

$$\int_{\Gamma} \underline{v} \cdot [(\hat{\underline{\sigma}} \nabla \cdot \underline{\mathcal{V}} - \hat{\underline{\sigma}} \nabla \underline{\mathcal{V}}) \underline{n}]$$

$$\hat{\underline{\sigma}} = \underline{I}, \hat{\underline{\sigma}} = c \underline{I} \text{ or } \hat{\underline{\sigma}} = \underline{\sigma}$$

Call signature

de_sd_surface_ltr	(opt_material, virtual, parameter_mv)
	(opt_material, parameter, parameter_mv)

Arguments

- material: $c, \underline{\sigma}, \underline{\sigma}$
- virtual/parameter: $\underline{v}$
- parameter_mv: $\underline{\mathcal{V}}$

```

arg_shapes = [{'opt_material': 'S', 1, 'parameter': 'D', 'parameter_mv': 'D',
'virtual': ('D', None)}, {'opt_material': None}, {'opt_material': '1', 1},
{'opt_material': 'D', D}]

```

```

arg_types = (('opt_material', 'virtual', 'parameter_mv'), ('opt_material',
'parameter', 'parameter_mv'))

```

```
get_function(traction, vvar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
modes = ('weak', 'eval')
```

```
name = 'de_sd_surface_ltr'
```

```
class sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm(*args, **kwargs)
```

Sensitivity (shape derivative) of the piezoelectric coupling term.

Definition

$$\int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{v}) \nabla_k p, \int_{\Omega} \hat{g}_{kij} e_{ij}(\underline{u}) \nabla_k q$$

$$\hat{g}_{kij} = g_{kij} (\nabla \cdot \underline{\mathcal{V}}) - g_{kil} \frac{\partial \mathcal{V}_j}{\partial x_l} - g_{lij} \frac{\partial \mathcal{V}_k}{\partial x_l}$$

Call signature

de_sd_piezo_coupling	(material, virtual, state, parameter_mv)
	(material, state, virtual, parameter_mv)
	(material, parameter_v, parameter_s, parameter_mv)

Arguments 1

- material : g_{kij}
- virtual/parameter_v : $\underline{v}$
- state/parameter_s : p
- parameter_mv : $\underline{\mathcal{V}}$

Arguments 2

- material : g_{kij}
- state : $\underline{u}$
- virtual : q
- parameter_mv : $\underline{\mathcal{V}}$

```
arg_shapes = {'material': 'D, S', 'parameter_mv': 'D', 'parameter_s': 1,
              'parameter_v': 'D', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None),
              'virtual/grad': ('D', None)}
```

```
arg_types = (('material', 'virtual', 'state', 'parameter_mv'), ('material', 'state',
                       'virtual', 'parameter_mv'), ('material', 'parameter_v', 'parameter_s',
                       'parameter_mv'))
```

```
geometries = ['2_3', '2_4', '3_4', '3_8']
```

```
get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'de_sd_piezo_coupling'
```

```
class sfepy.terms.terms_sensitivity.ESDStokesTerm(*args, **kwargs)
```

Stokes problem coupling term. Corresponds to weak forms of gradient and divergence terms.

Definition

$$\int_{\Omega} p \hat{I}_{ij} \frac{\partial v_i}{\partial x_j}, \int_{\Omega} q \hat{I}_{ij} \frac{\partial u_i}{\partial x_j}$$

$$\hat{I}_{ij} = \delta_{ij} \nabla \cdot \underline{\mathcal{V}} - \frac{\partial \mathcal{V}_j}{\partial x_i}$$

Call signature

de_sd_stokes	(opt_material, virtual, state, parameter_mv)
	(opt_material, state, virtual, parameter_mv)
	(opt_material, parameter_v, parameter_s, parameter_mv)

Arguments 1

- virtual/parameter_v: $\underline{v}$

- state/parameter_s: p
- parameter_mv: $\mathcal{V}$

Arguments 2

- state : $\underline{u}$
- virtual : q
- parameter_mv: $\mathcal{V}$

```
arg_shapes = [{'opt_material': '1, 1', 'parameter_mv': 'D', 'parameter_s': 1,
'parameter_v': 'D', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None),
'virtual/grad': ('D', None)}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state', 'parameter_mv'), ('opt_material',
'state', 'virtual', 'parameter_mv'), ('opt_material', 'parameter_v', 'parameter_s',
'parameter_mv'))
```

```
get_function(coef, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'de_sd_stokes'
```

```
texpr = 'ij,i.j,0'
```

```
class sfepy.terms.terms_sensitivity.ESDVectorDotGradScalarTerm(*args, **kwargs)
```

Sensitivity of volume dot product of a vector and a gradient of scalar.

Definition

$$\int_{\Omega} \hat{I}_{ij} \frac{\partial p}{\partial x_j} v_i, \int_{\Omega} \hat{I}_{ij} \frac{\partial q}{\partial x_j} u_i$$

$$\hat{I}_{ij} = \delta_{ij} \nabla \cdot \mathcal{V} - \frac{\partial \mathcal{V}_j}{\partial x_i}$$

Call signature

de_sd_v_dot_grad_s	(opt_material, virtual, state, parameter_mv)
	(opt_material, state, virtual, parameter_mv)
	(opt_material, parameter_v, parameter_s, parameter_mv)

Arguments 1

- virtual/parameter_v: $\underline{v}$
- state/parameter_s: p
- parameter_mv: $\mathcal{V}$

Arguments 2

- state : $\underline{u}$
- virtual : q
- parameter_mv: $\mathcal{V}$

```
name = 'de_sd_v_dot_grad_s'
```

```
texpr = 'ij,i,0.j'
```

```
sfepy.terms.terms_sensitivity.get_nonsym_grad_op(sgrad)
```

sfepy.terms.terms_shells module

Terms implementing shell elements.

class sfepy.terms.terms_shells.**Shell10XTerm**(name, arg_str, integral, region, **kwargs)

The shell10x element term based on the Reissner-Mindlin theory [1], [2], corresponding to a shell of thickness t .

The term requires a custom 3D quadrature, where the z components of quadrature point coordinates are transformed from $[0, 1]$ to $[-t/2, t/2]$, and the quadrature weights are multiplied by t . The variables $\underline{v}$ and $\underline{u}$ have to use [Shell10XField](#) and have six components. The reference element mapping is implemented by [Shell10XMapping](#). The term does not implement the piezo-electric components of the shell10x element yet.

The term has to be used with quadrilateral cells in 3D and should behave as the linear elastic term, but with fewer degrees of freedom for the same accuracy for shell-like structures. The shell has six degrees of freedom in each of the four nodes: $\mathbf{u}_i = [u_i, v_i, w_i, \alpha_i, \beta_i, \gamma_i]^T$, $i = 1, 2, 3, 4$. The strain and stress vectors are calculated in a local (co-rotational) coordinate system given by basis vectors $\mathbf{e}'_1$, $\mathbf{e}'_2$ and $\mathbf{e}'_3$. It holds that

$$[u'_i, v'_i, w'_i, \alpha'_i, \beta'_i, \gamma'_i]^T = \hat{\mathbf{H}}^T \mathbf{u}_i$$

where

$$\hat{\mathbf{H}} = \begin{bmatrix} \mathbf{H} & \\ & \mathbf{H} \end{bmatrix} \quad \text{and} \quad \mathbf{H} = [\mathbf{e}'_1 \mathbf{e}'_2 \mathbf{e}'_3]$$

is a nodal DOF transformation matrix.

The local displacements u' , v' and w' at any point in the layer characterized by the isoparametric coordinates ξ , η and ζ ($\xi, \eta, \zeta \in \langle -1, 1 \rangle$) are interpolated from the nodal displacement and rotation values (i.e. both membrane and bending components) using standard isoparametric approximation functions for a quadrilateral, hence

$$\begin{aligned} u'(\xi, \eta, \zeta) &= \sum_{i=1}^4 N_i(\xi, \eta) \cdot (u'_i + \bar{u}_i), \\ v'(\xi, \eta, \zeta) &= \sum_{i=1}^4 N_i(\xi, \eta) \cdot (v'_i + \bar{v}_i), \\ w'(\xi, \eta, \zeta) &= \sum_{i=1}^4 N_i(\xi, \eta) \cdot (w'_i + \bar{w}_i) \end{aligned}$$

where $\bar{u}_i$, $\bar{v}_i$ and $\bar{w}_i$ are the bending components of displacements calculated from displacements due to rotations $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ about local nodal axes $\tilde{\mathbf{e}}_i$ as

$$\begin{bmatrix} \bar{u} \\ \bar{v} \\ \bar{w} \end{bmatrix}_i = \tilde{\zeta} \begin{bmatrix} \tilde{\mathbf{e}}_1 & -\tilde{\mathbf{e}}_2 \end{bmatrix}_i \begin{bmatrix} \tilde{\mathbf{e}}_2^T \\ \tilde{\mathbf{e}}_1^T \end{bmatrix}_i \begin{bmatrix} \alpha' \\ \beta' \\ \gamma' \end{bmatrix}_i$$

where $\tilde{\zeta} = (t/2)\zeta$. The local nodal axes $\tilde{\mathbf{e}}_i$ are constructed in order to describe the behavior of warped (non-planar) elements adequately.

The term employs three shell element enhancements:

- DSG method

- EAS method
- drilling rotations lock (parameter χ - a good value is about 10^{-7})

For detailed theoretical information see the references.

High-Performance 4-Node Shell Element with Piezoelectric Coupling Mechanics of Advanced Materials and Structures Vol. 13, Iss. 5, doi:10.1080/15376490600777657

High-performance four-node shell element with piezoelectric coupling for the analysis of smart laminated structures. Int. J. Numer. Meth. Engng., 70: 934–961. doi:10.1002/nme.1909

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$$

Call signature

```
dw_shell10x (material_d, material_drill, virtual, state)
```

Arguments

- material_d : D
- material_drill : χ
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_d': '6, 6', 'material_drill': '.: 1', 'state': 6,
'virtual': (6, 'state')}
```

```
arg_types = ('material_d', 'material_drill', 'virtual', 'state')
```

```
static function(out, mtx_k, el_u, fmode)
```

```
geometries = ['3_2_4']
```

```
get_fargs(mtx_d, drill, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_physical_qps()
```

Get physical quadrature points corresponding to the term region and integral.

```
integration = 'custom'
```

```
name = 'dw_shell10x'
```

```
poly_space_basis = 'shell10x'
```

```
set_integral(integral)
```

Set the term integral.

sfePy.terms.terms_surface module**class** `sfePy.terms.terms_surface.ContactPlaneTerm(*args, **kwargs)`

Small deformation elastic contact plane term with penetration penalty.

The plane is given by an anchor point $\underline{A}$ and a normal $\underline{n}$. The contact occurs in points that orthogonally project onto the plane into a polygon given by orthogonal projections of boundary points $\{\underline{B}_i\}$, $i = 1, \dots, N_B$ on the plane. In such points, a penetration distance $d(\underline{u}) = (\underline{X} + \underline{u} - \underline{A}, \underline{n})$ is computed, and a force $f(d(\underline{u}))\underline{n}$ is applied. The force depends on the non-negative parameters k (stiffness) and f_0 (force at zero penetration):

- If $f_0 = 0$:

$$\begin{aligned} f(d) &= 0 \text{ for } d \leq 0, \\ f(d) &= kd \text{ for } d > 0. \end{aligned}$$

- If $f_0 > 0$:

$$\begin{aligned} f(d) &= 0 \text{ for } d \leq -\frac{2r_0}{k}, \\ f(d) &= \frac{k^2}{4r_0}d^2 + kd + r_0 \text{ for } -\frac{2r_0}{k} < d \leq 0, \\ f(d) &= kd + f_0 \text{ for } d > 0. \end{aligned}$$

In this case the dependence $f(d)$ is smooth, and a (small) force is applied even for (small) negative penetrations: $-\frac{2r_0}{k} < d \leq 0$.

Definition

$$\int_{\Gamma} \underline{v} \cdot f(d(\underline{u}))\underline{n}$$

Call signature

dw_contact_plane (material_f, material_n, material_a, material_b, virtual, state)

Arguments

- material_f : $[k, f_0]$
- material_n : $\underline{n}$ (special)
- material_a : $\underline{A}$ (special)
- material_b : $\{\underline{B}_i\}$, $i = 1, \dots, N_B$ (special)
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_a': '.: D', 'material_b': '.: N, D', 'material_f': '1, 2', 'material_n': '.: D', 'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material_f', 'material_n', 'material_a', 'material_b', 'virtual', 'state')
```

```
static function(out, force, normal, geo, fmode)
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs(force_pars, normal, anchor, bounds, virtual, state, mode=None, term_mode=None,
          diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'dw_contact_plane'
```

```
static smooth_f(d, k, f0, a, eps, diff)
```

```
class sfepy.terms.terms_surface.ContactSphereTerm(*args, **kwargs)
```

Small deformation elastic contact sphere term with penetration penalty.

The sphere is given by a centre point $\underline{C}$ and a radius R . The contact occurs in points that are closer to $\underline{C}$ than R . In such points, a penetration distance $d(\underline{u}) = R - \|\underline{X} + \underline{u} - \underline{C}\|$ is computed, and a force $f(d(\underline{u}))\underline{n}(\underline{u})$ is applied, where $\underline{n}(\underline{u}) = (\underline{X} + \underline{u} - \underline{C})/\|\underline{X} + \underline{u} - \underline{C}\|$. The force depends on the non-negative parameters k (stiffness) and f_0 (force at zero penetration):

- If $f_0 = 0$:

$$\begin{aligned} f(d) &= 0 \text{ for } d \leq 0, \\ f(d) &= kd \text{ for } d > 0. \end{aligned}$$

- If $f_0 > 0$:

$$\begin{aligned} f(d) &= 0 \text{ for } d \leq -\frac{2r_0}{k}, \\ f(d) &= \frac{k^2}{4r_0}d^2 + kd + r_0 \text{ for } -\frac{2r_0}{k} < d \leq 0, \\ f(d) &= kd + f_0 \text{ for } d > 0. \end{aligned}$$

In this case the dependence $f(d)$ is smooth, and a (small) force is applied even for (small) negative penetrations: $-\frac{2r_0}{k} < d \leq 0$.

Definition

$$\int_{\Gamma} \underline{v} \cdot f(d(\underline{u}))\underline{n}(\underline{u})$$

Call signature

```
dw_contact_sphere (material_f, material_c, material_r, virtual, state)
```

Arguments

- material_f : $[k, f_0]$
- material_c : $\underline{C}$ (special)
- material_r : R (special)
- virtual : $\underline{v}$
- state : $\underline{u}$

```
arg_shapes = {'material_c': '.: D', 'material_f': '1, 2', 'material_r': '.: 1',
              'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material_f', 'material_c', 'material_r', 'virtual', 'state')

static function(out, force, normals, fd, geo, fmode)

geometries = ['3_4', '3_8']

get_fargs(force_pars, centre, radius, virtual, state, mode=None, term_mode=None, diff_var=None,
          **kwargs)

integration = 'facet'

name = 'dw_contact_sphere'
```

```
class sfepy.terms.terms_surface.LinearTractionTerm(name, arg_str, integral, region, **kwargs)
```

Linear traction forces, where, depending on dimension of ‘material’ argument, $\underline{\underline{\sigma}} \cdot \underline{n}$ is $\bar{p}\underline{\underline{I}} \cdot \underline{n}$ for a given scalar pressure, $\underline{f}$ for a traction vector, and itself for a stress tensor.

The material parameter can have one of the following shapes: 1 or (1, 1), (D, 1), (S, 1) in all modes, or (D, D) in the *eval* mode only. The symmetric tensor storage (S, 1) is as follows: in 3D S = 6 and the indices ordered as [11, 22, 33, 12, 13, 23], in 2D S = 3 and the indices ordered as [11, 22, 12].

Definition

$$\int_{\Gamma} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \int_{\Gamma} \underline{v} \cdot \underline{n},$$

Call signature

dw_surface_ltr	(opt_material, virtual)
	(opt_material, parameter)

Arguments

- material : $\underline{\underline{\sigma}}$
- virtual : $\underline{v}$

```
arg_shapes = [{'opt_material': 'S, 1', 'parameter': 'D', 'virtual': ('D', None)},
              {'opt_material': 'D, 1'}, {'opt_material': '1, 1'}, {'opt_material': 'D, D'},
              {'opt_material': None}]

arg_types = (('opt_material', 'virtual'), ('opt_material', 'parameter'))

static d_fun(out, traction, val, sg)

get_eval_shape(traction, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(traction, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'dw_surface_ltr'
```

set_arg_types()

class sfepy.terms.terms_surface.**SDLinearTractionTerm**(*name, arg_str, integral, region, **kwargs*)

Sensitivity of the linear traction term.

Definition

$$\int_{\Gamma} \underline{v} \cdot (\underline{\sigma} \underline{n}), \int_{\Gamma} \underline{v} \cdot \underline{n},$$

Call signature

ev_sd_surface_ltr (opt_material, parameter, parameter_mv)

Arguments

- material : $\underline{\sigma}$
- parameter : $\underline{v}$

```
arg_shapes = [{'opt_material': 'S, 1', 'parameter': 'D', 'parameter_mv': 'D'},
{'opt_material': '1, 1'}, {'opt_material': 'D, 1'}, {'opt_material': 'D, D'},
{'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter', 'parameter_mv')
```

```
static d_fun(out, traction, val, grad_mv, div_mv, sg)
```

```
get_eval_shape(traction, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(traction, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'ev_sd_surface_ltr'
```

```
set_arg_types()
```

class sfepy.terms.terms_surface.**SDSurfaceIntegrateTerm**(*name, arg_str, integral, region, **kwargs*)

Sensitivity of scalar traction.

Definition

$$\int_{\Gamma} p \nabla \cdot \underline{v}$$

Call signature

ev_sd_surface_integrate (parameter, parameter_mv)

Arguments

- parameter : p
- parameter_mv : $\underline{v}$

```
arg_shapes = {'parameter': 1, 'parameter_mv': 'D'}
```

```
arg_types = ('parameter', 'parameter_mv')

static function(out, val_p, div_v, sg)

get_eval_shape(par, par_v, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(par, par_v, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'ev_sd_surface_integrate'
```

```
class sfepy.terms.terms_surface.SurfaceNormalDotTerm(name, arg_str, integral, region, **kwargs)
    “Scalar traction” term, (weak form).
```

Definition

$$\int_{\Gamma} q \underline{c} \cdot \underline{n}$$

Call signature

dw_surface_ndot	(material, virtual)
	(material, parameter)

Arguments

- material : $\underline{c}$
- virtual : q

```
arg_shapes = {'material': 'D, 1', 'parameter': 1, 'virtual': (1, None)}
arg_types = (('material', 'virtual'), ('material', 'parameter'))

static d_fun(out, material, val, sg)

static dw_fun(out, material, bf, sg)

get_eval_shape(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'dw_surface_ndot'

set_arg_types()
```

```
class sfepy.terms.terms_surface.SurfaceJumpTerm(name, arg_str, integral, region, **kwargs)
    Interface jump condition.
```

Definition

$$\int_{\Gamma} c q (p_1 - p_2)$$

Call signature

dw_jump (opt_material, virtual, state_1, state_2)

Arguments

- material : c
- virtual : q
- state_1 : p_1
- state_2 : p_2

```
arg_shapes = [{'opt_material': '1, 1', 'state_1': 1, 'state_2': 1, 'virtual':
(1, None)}, {'opt_material': None}]
```

```
arg_types = ('opt_material', 'virtual', 'state_1', 'state_2')
```

```
static function(out, jump, mul, bf1, bf2, sg, fmode)
```

```
get_fargs(coef, virtual, state1, state2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'dw_jump'
```

sfepy.terms.terms_th module

```
class sfepy.terms.terms_th.ETHTerm(name, arg_str, integral, region, **kwargs)
```

Base class for terms depending on time history with exponential convolution kernel (fading memory terms).

```
advance_eth_data(ts, data)
```

```
get_eth_data(key, state, decay, values)
```

```
class sfepy.terms.terms_th.THTerm(name, arg_str, integral, region, **kwargs)
```

Base class for terms depending on time history (fading memory terms).

```
eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
```

sfepy.terms.terms_volume module

```
class sfepy.terms.terms_volume.LinearVolumeForceTerm(name, arg_str, integral, region, **kwargs)
```

Vector or scalar linear volume forces (weak form) — a right-hand side source term.

Definition

$$\int_{\Omega} \underline{f} \cdot \underline{v} \text{ or } \int_{\Omega} f q$$

Call signature

dw_volume_lvf (material, virtual)

Arguments

- material : $\underline{f}$ or f
- virtual : $\underline{v}$ or q

```
arg_shapes = [{'material': 'D', 1, 'virtual': ('D', None)}, {'material': '1', 1, 'virtual': (1, None)}]
```

```
arg_types = ('material', 'virtual')
```

```
static function(out, force_qp, cmap)
```

```
get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_volume_lvf'
```

```
class sfepy.terms.terms_volume.NonlinearVolumeForceTerm(name, arg_str, integral, region, **kwargs)
```

The volume force term with the force given by a user supplied function of the state variable.

Definition

$$\int_{\Omega} qf(p)$$

Call signature

<code>dw_volume_nvf</code> (fun, dfun, virtual, state)

Arguments

- fun : $f(p)$
- dfun : $\partial f(p)/\partial p$
- virtual : q
- state : p

```
arg_shapes = {'dfun': <function NonlinearVolumeForceTerm.<lambda>>, 'fun':  
<function NonlinearVolumeForceTerm.<lambda>>, 'state': 1, 'virtual': (1, 'state')}
```

```
arg_types = ('fun', 'dfun', 'virtual', 'state')
```

```
static function(out, out_qp, geo)
```

```
get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_volume_nvf'
```

sfepy.terms.utils module

```
sfepy.terms.utils.check_finiteness(data, info)
```

```
sfepy.terms.utils.get_range_indices(num)
```

Return indices and slices in given range.

Returns

indx

[list of tuples] The list of $(ii, slice(ii, ii + 1))$ of the indices. The first item is the index itself, the second item is a convenience slice to index components of material parameters.

sfepy.terms.extmods.terms module

Low level term evaluation functions.

```

sfepy.terms.extmods.terms.actBfT(out, bf, A)

sfepy.terms.extmods.terms.d_biot_div(out, coef, state, strain, mtx_d, cmap)

sfepy.terms.extmods.terms.d_diffusion(out, grad_p1, grad_p2, mtx_d, cmap)

sfepy.terms.extmods.terms.d_laplace(out, grad_p1, grad_p2, coef, cmap)

sfepy.terms.extmods.terms.d_lin_elastic(out, coef, strain_v, strain_u, mtx_d, cmap)

sfepy.terms.extmods.terms.d_of_nsMinGrad(out, grad, viscosity, cmap)

sfepy.terms.extmods.terms.d_of_nsSurfMinDPress(out, pressure, weight, bpress, cmap, is_diff)

sfepy.terms.extmods.terms.d_piezo_coupling(out, strain, charge_grad, mtx_g, cmap)

sfepy.terms.extmods.terms.d_sd_convect(out, state_u, grad_u, state_w, div_mv, grad_mv, cmap_u, mode)

sfepy.terms.extmods.terms.d_sd_diffusion(out, grad_q, grad_p, grad_w, div_w, mtx_d, cmap)

sfepy.terms.extmods.terms.d_sd_div(out, div_u, grad_u, state_p, div_mv, grad_mv, cmap_u, mode)

sfepy.terms.extmods.terms.d_sd_div_grad(out, grad_u, grad_w, div_mv, grad_mv, viscosity, cmap_u,
                                         mode)

sfepy.terms.extmods.terms.d_sd_lin_elastic(out, coef, grad_v, grad_u, grad_w, mtx_d, cmap)

sfepy.terms.extmods.terms.d_sd_st_grad_div(out, div_u, grad_u, div_w, grad_w, div_mv, grad_mv, coef,
                                             cmap_u, mode)

sfepy.terms.extmods.terms.d_sd_st_pspg_c(out, state_b, grad_u, grad_r, div_mv, grad_mv, coef, cmap_u,
                                           mode)

sfepy.terms.extmods.terms.d_sd_st_pspg_p(out, grad_r, grad_p, div_mv, grad_mv, coef, cmap_p, mode)

sfepy.terms.extmods.terms.d_sd_st_supg_c(out, state_b, grad_u, grad_w, div_mv, grad_mv, coef, cmap_u,
                                           mode)

sfepy.terms.extmods.terms.d_sd_volume_dot(out, state_p, state_q, div_mv, cmap, mode)

sfepy.terms.extmods.terms.d_surface_flux(out, grad, mtx_d, cmap, mode)

sfepy.terms.extmods.terms.d_tl_surface_flux(out, pressure_grad, mtx_d, ref_porosity, mtx_fi, det_f,
                                             cmap, mode)

sfepy.terms.extmods.terms.d_tl_volume_surface(out, coors, det_f, mtx_fi, bf, cmap, conn)

sfepy.terms.extmods.terms.d_volume_surface(out, in_, cmap, conn)

sfepy.terms.extmods.terms.de_cauchy_strain(out, strain, cmap, mode)

sfepy.terms.extmods.terms.de_cauchy_stress(out, strain, mtx_d, cmap, mode)

sfepy.terms.extmods.terms.de_he_rtm(out, stress, det_f, cmap, el_list, mode_ul)

```

```
sfepy.terms.extmods.terms.di_surface_moment(out, in_, cmap, conn)
sfepy.terms.extmods.terms.dq_cauchy_strain(out, state, cmap, conn)
sfepy.terms.extmods.terms.dq_def_grad(out, state, cmap, conn, mode)
sfepy.terms.extmods.terms.dq_div_vector(out, state, cmap, conn)
sfepy.terms.extmods.terms.dq_finite_strain_tl(mtx_f, det_f, vec_cs, tr_c, in_2c, vec_inv_cs, vec_es,
                                              state, cmap, conn)
sfepy.terms.extmods.terms.dq_finite_strain_ul(mtx_f, det_f, vec_bs, tr_b, in_2b, vec_es, state, cmap,
                                              conn)
sfepy.terms.extmods.terms.dq_grad(out, state, cmap, conn)
sfepy.terms.extmods.terms.dq_state_in_qp(out, state, bf, conn)
sfepy.terms.extmods.terms.dq_tl_finite_strain_surface(mtx_f, det_f, mtx_fi, state, cmap, fis, conn)
sfepy.terms.extmods.terms.dq_tl_he_stress_bulk(out, mat, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_he_stress_bulk_active(out, mat, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_he_stress_mooney_rivlin(out, mat, det_f, tr_c, vec_inv_cs, vec_cs,
                                                         in_2c)
sfepy.terms.extmods.terms.dq_tl_he_stress_neohook(out, mat, det_f, tr_c, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_bulk(out, mat, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_bulk_active(out, mat, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_mooney_rivlin(out, mat, det_f, tr_c, vec_inv_cs, vec_cs,
                                                         in_2c)
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_neohook(out, mat, det_f, tr_c, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_stress_bulk_pressure(out, pressure_qp, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_tl_tan_mod_bulk_pressure_u(out, pressure_qp, det_f, vec_inv_cs)
sfepy.terms.extmods.terms.dq_ul_he_stress_bulk(out, mat, det_f)
sfepy.terms.extmods.terms.dq_ul_he_stress_mooney_rivlin(out, mat, det_f, tr_b, vec_bs, in_2b)
sfepy.terms.extmods.terms.dq_ul_he_stress_neohook(out, mat, det_f, tr_b, vec_bs)
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_bulk(out, mat, det_f)
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_mooney_rivlin(out, mat, det_f, tr_b, vec_bs, in_2b)
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_neohook(out, mat, det_f, tr_b, vec_bs)
sfepy.terms.extmods.terms.dq_ul_stress_bulk_pressure(out, pressure_qp, det_f)
sfepy.terms.extmods.terms.dq_ul_tan_mod_bulk_pressure_u(out, pressure_qp, det_f)
sfepy.terms.extmods.terms.dw_adj_convect1(out, state_w, grad_u, cmap, is_diff)
```

```

sfepy.terms.extmods.terms.dw_adj_convect2(out, state_w, state_u, cmap, is_diff)
sfepy.terms.extmods.terms.dw_biot_div(out, coef, strain, mtx_d, cmap_s, cmap_v, is_diff)
sfepy.terms.extmods.terms.dw_biot_grad(out, coef, pressure_qp, mtx_d, cmap_s, cmap_v, is_diff)
sfepy.terms.extmods.terms.dw_convect_v_grad_s(out, val_v, grad_s, cmap_v, cmap_s, is_diff)
sfepy.terms.extmods.terms.dw_diffusion(out, grad, mtx_d, cmap, is_diff)
sfepy.terms.extmods.terms.dw_diffusion_r(out, mtx_d, cmap)
sfepy.terms.extmods.terms.dw_div(out, coef, div, cmap_s, cmap_v, is_diff)
sfepy.terms.extmods.terms.dw_electric_source(out, grad, coef, cmap)
sfepy.terms.extmods.terms.dw_grad(out, coef, state, cmap_s, cmap_v, is_diff)
sfepy.terms.extmods.terms.dw_he_rtm(out, stress, tan_mod, mtx_f, det_f, cmap, is_diff, mode_ul)
sfepy.terms.extmods.terms.dw_laplace(out, grad, coef, cmap, is_diff)
sfepy.terms.extmods.terms.dw_lin_convect(out, grad, state_b, cmap, is_diff)
sfepy.terms.extmods.terms.dw_lin_elastic(out, coef, strain, mtx_d, cmap, is_diff)
sfepy.terms.extmods.terms.dw_lin_prestress(out, stress, cmap)
sfepy.terms.extmods.terms.dw_lin_strain_fib(out, mtx_d, mat, cmap)
sfepy.terms.extmods.terms.dw_nonsym_elastic(out, grad, mtx_d, cmap, is_diff)
sfepy.terms.extmods.terms.dw_piezo_coupling(out, strain, charge_grad, mtx_g, cmap, mode)
sfepy.terms.extmods.terms.dw_st_adj1_supg_p(out, state_w, grad_p, coef, cmap_w, conn_w, is_diff)
sfepy.terms.extmods.terms.dw_st_adj2_supg_p(out, grad_u, state_r, coef, cmap_u, cmap_r, conn_r,
                                             is_diff)
sfepy.terms.extmods.terms.dw_st_adj_supg_c(out, state_w, state_u, grad_u, coef, cmap, conn, is_diff)
sfepy.terms.extmods.terms.dw_st_grad_div(out, div, coef, cmap, is_diff)
sfepy.terms.extmods.terms.dw_st_pspg_c(out, state_b, state_u, coef, cmap_p, cmap_u, conn, is_diff)
sfepy.terms.extmods.terms.dw_st_supg_c(out, state_b, state_u, coef, cmap, conn, is_diff)
sfepy.terms.extmods.terms.dw_st_supg_p(out, state_b, grad_p, coef, cmap_u, cmap_p, is_diff)
sfepy.terms.extmods.terms.dw_surface_flux(out, grad, mat, bf, cmap, fis, mode)
sfepy.terms.extmods.terms.dw_surface_ltr(out, traction, cmap)
sfepy.terms.extmods.terms.dw_surface_s_v_dot_n(out, coef, val_qp, rcmap, ccmap, is_diff)
sfepy.terms.extmods.terms.dw_surface_v_dot_n_s(out, coef, val_qp, rcmap, ccmap, is_diff)
sfepy.terms.extmods.terms.dw_tl_diffusion(out, pressure_grad, mtx_d, ref_porosity, mtx_f, det_f, cmap,
                                             mode)

```

```
sfepy.terms.extmods.terms.dw_tl_surface_traction(out, traction, det_f, mtx_fi, bf, cmap, fis, mode)
sfepy.terms.extmods.terms.dw_tl_volume(out, mtx_f, vec_inv_cs, det_f, cmap_s, cmap_v, transpose, mode)
sfepy.terms.extmods.terms.dw_ul_volume(out, det_f, cmap_s, cmap_v, transpose, mode)
sfepy.terms.extmods.terms.dw_v_dot_grad_s_sw(out, coef, val_qp, cmap_v, cmap_s, is_diff)
sfepy.terms.extmods.terms.dw_v_dot_grad_s_vw(out, coef, grad, cmap_v, cmap_s, is_diff)
sfepy.terms.extmods.terms.dw_volume_dot_scalar(out, coef, val_qp, rcmap, ccmap, is_diff)
sfepy.terms.extmods.terms.dw_volume_dot_vector(out, coef, val_qp, rcmap, ccmap, is_diff)
sfepy.terms.extmods.terms.dw_volume_lvf(out, force_qp, cmap)
sfepy.terms.extmods.terms.errclear()
sfepy.terms.extmods.terms.he_eval_from_mtx(out, mtx_d, state_v, state_u, conn, el_list)
sfepy.terms.extmods.terms.he_residuum_from_mtx(out, mtx_d, state, conn, el_list)
sfepy.terms.extmods.terms.mulAB_integrate(out, A, B, cmap, mode)
sfepy.terms.extmods.terms.sym2nonsym(out, A)
sfepy.terms.extmods.terms.term_ns_asm_convect(out, grad, state, cmap, is_diff)
sfepy.terms.extmods.terms.term_ns_asm_div_grad(out, grad, viscosity, cmap_v, cmap_s, is_diff)
```

Scripts

sfepy.scripts.blockgen module

Block mesh generator.

```
sfepy.scripts.blockgen.add_args(parser)
sfepy.scripts.blockgen.gen_block(options)
sfepy.scripts.blockgen.main()
```

sfepy.scripts.convert_mesh module

Convert a mesh file from one SfePy-supported format to another.

```
sfepy.scripts.convert_mesh.main()
```

sfepy.scripts.cylindergen module

Cylinder mesh generator.

```
sfepy.scripts.cylindergen.add_args(parser)
sfepy.scripts.cylindergen.gen_cylinder(options)
sfepy.scripts.cylindergen.main()
```

sfepy.scripts.gen_iga_patch module

Generate a single IGA patch block in 2D or 3D of given degrees and continuity using igakit.

The grid has equally-spaced knot vectors.

```
sfepy.scripts.gen_iga_patch.add_args(parser)
sfepy.scripts.gen_iga_patch.gen_iga_patch(options)
sfepy.scripts.gen_iga_patch.main()
```

sfepy.scripts.gen_mesh module

Simple mesh generators and statistics.

```
sfepy.scripts.gen_mesh.main()
```

sfepy.scripts.gen_mesh_prev module

Mesh Preview Generator.

Examples

```
$ ./script/gen_mesh_prev.py meshes/2d/
```

```
sfepy.scripts.gen_mesh_prev.gen_shot(vtk_filename, png_filename)
```

Generate PNG image of the FE mesh.

Parameters

vtk_filename

[str] The input mesh filename (file in VTK format).

png_filename

[str] The name of the output PNG file.

```
sfepy.scripts.gen_mesh_prev.main()
```

sfepy.scripts.plot_condition_numbers module

Plot conditions numbers w.r.t. polynomial approximation order of reference element matrices for various FE polynomial spaces (bases).

`sfepy.scripts.plot_condition_numbers.main()`

sfepy.scripts.plot_logs module

Plot logs of variables saved in a text file by `sfepy.base.log.Log` class.

The plot should be almost the same as the plot that would be generated by the Log directly.

```
class sfepy.scripts.plot_logs.ParseRc(option_strings, dest, nargs=None, const=None, default=None,  
                                     type=None, choices=None, required=False, help=None,  
                                     metavar=None)
```

`sfepy.scripts.plot_logs.main()`

sfepy.scripts.plot_mesh module

Plot mesh connectivities, facet orientations, global and local DOF ids etc.

To switch off plotting some mesh entities, set the corresponding color to *None*.

`sfepy.scripts.plot_mesh.main()`

sfepy.scripts.plot_quadratures module

Plot quadrature points for the given geometry and integration order.

`sfepy.scripts.plot_quadratures.main()`

sfepy.scripts.plot_times module

Plot time steps, times of time steps and time deltas in a HDF5 results file.

`sfepy.scripts.plot_times.main()`

sfepy.scripts.probe module

Probe finite element solutions in points defined by various geometrical probes.

In the examples below it is supposed that sfepy is installed. When using the in-place build, replace `sfepy-probe` by `python3 sfepy/scripts/probe.py`.

Generation mode

`sfepy-probe` [generation options] <input file> <results file>

Probe the data in the results file corresponding to the problem defined in the input file. The input file options must contain 'gen_probes' and 'probe_hook' keys, pointing to proper functions accessible from the input file scope.

For each probe returned by `gen_probes()` a data plot figure and a text file with the data plotted are saved, see the options below.

Generation options

`-o`, `-auto-dir`, `-same-dir`, `-f`, `-only-names`, `-s`

Postprocessing mode

`sfepy-probe` [postprocessing options] <probe file> <figure file>

Read a previously probed data from the probe text file, re-plot them, and integrate them along the probe.

Postprocessing options

`-postprocess`, `-radial`, `-only-names`

Notes

For extremely thin hexahedral elements the Newton's iteration for finding the reference element coordinates might converge to a spurious solution outside of the element. To obtain some values even in this case, try increasing the `-close-limit` option value.

`sfepy.scripts.probe.generate_probes(filename_input, filename_results, options, conf=None,`
`problem=None, probes=None, labels=None, probe_hooks=None)`

Generate probe figures and data files.

`sfepy.scripts.probe.integrate_along_line(x, y, is_radial=False)`

Integrate numerically (trapezoidal rule) a function $y = y(x)$.

If `is_radial` is `True`, multiply each y by $4\pi x^2$.

`sfepy.scripts.probe.main()`

`sfepy.scripts.probe.postprocess(filename_input, filename_results, options)`

Postprocess probe data files - replot, integrate data.

sfepy.scripts.resview module

This is a script for quick VTK-based visualizations of finite element computations results.

In the examples below it is supposed that sfepy is installed. When using the in-place build, replace sfepy-view by python3 sfepy/scripts/resview.py.

Examples

The examples assume that `python -c "import sfepy; sfepy.test('--output-dir=output-tests')"` has been run successfully and the resulting data files are present.

- View data in output-tests/test_navier_stokes.vtk:

```
sfepy-view output-tests/navier_stokes-navier_stokes.vtk
```

- Customize the above output: plot0: field “p”, switch on edges, plot1: field “u”, surface with opacity 0.4, glyphs scaled by factor 2e-2:

```
sfepy-view output-tests/navier_stokes-navier_stokes.vtk -f p:e:p0 u:o.4:p1 u:g:f2e-  
↪2:p1
```

- As above, but glyphs are scaled by the factor determined automatically as 20% of the minimum bounding box size:

```
sfepy-view output-tests/navier_stokes-navier_stokes.vtk -f p:e:p0 u:o.4:p1 u:g:f10  
↪:%:p1
```

- View data and take a screenshot:

```
sfepy-view output-tests/diffusion-poisson.vtk -o image.png
```

- Take a screenshot without a window popping up:

```
sfepy-view output-tests/diffusion-poisson.vtk -o image.png --off-screen
```

- Create animation from output-tests/diffusion-time_poisson.*.vtk:

```
sfepy-view output-tests/diffusion-time_poisson.*.vtk -a mov.mp4
```

- Create animation from output-tests/test_hyperelastic.*.vtk, set frame rate to 3, plot displacements and mooney_rivlin_stress:

```
sfepy-view output-tests/test_hyperelastic_TL.*.vtk -f u:wu:e:p0 mooney_rivlin_  
↪stress:p1 -a mov.mp4 -r 3
```

```
class sfepy.scripts.resview.FieldOptsToListAction(option_strings, dest, nargs=None, const=None,  
                                                    default=None, type=None, choices=None,  
                                                    required=False, help=None, metavar=None)
```

```
    separator = ':'
```

```
class sfepy.scripts.resview.OptsToListAction(option_strings, dest, nargs=None, const=None,  
                                              default=None, type=None, choices=None,  
                                              required=False, help=None, metavar=None)
```

```

separator = '='

class sfepy.scripts.resview.StoreNumberAction(option_strings, dest, nargs=None, const=None,
                                              default=None, type=None, choices=None,
                                              required=False, help=None, metavar=None)

sfepy.scripts.resview.add_mat_id_to_grid(grid, cell_groups)

sfepy.scripts.resview.ensure3d(arr)

sfepy.scripts.resview.get_camera_position(bounds, azimuth, elevation, distance=None, zoom=1.0)

sfepy.scripts.resview.get_nonzero_norm(arr, **kwargs)

sfepy.scripts.resview.main()

sfepy.scripts.resview.make_cells_from_conn(conns, convert_to_vtk_type)

sfepy.scripts.resview.make_grid_from_mesh(mesh, add_mat_id=False)

sfepy.scripts.resview.make_title(filenamees)

sfepy.scripts.resview.parse_options(opts, separator=':')

sfepy.scripts.resview.print_camera_position(plotter)

sfepy.scripts.resview.pv_plot(filenamees, options, plotter=None, step=None, annotations=None,
                              scalar_bar_limits=None, ret_scalar_bar_limits=False, step_inc=None,
                              use_cache=True)

sfepy.scripts.resview.read_mesh(filenamees, step=None, print_info=True, ret_n_steps=False,
                                use_cache=True)

sfepy.scripts.resview.read_probes_as_annotations(filenamees, add_label=True)

```

sfepy.scripts.run_tests module

Run SfePy tests. All arguments are passed to pytest.

```
sfepy.scripts.run_tests.main()
```

sfepy.scripts.simple module

Solve partial differential equations given in a SfePy problem definition file.

Example problem definition files can be found in `sfepy/examples/` directory of the SfePy top-level directory.

In the examples below it is supposed that sfepy is installed. When using the in-place build, replace `sfepy-run` by `python3 sfepy/scripts/simple.py`.

The supported application kinds (`-app` option) are:

- `bvp` - boundary value problem. Example:

```
sfepy-run sfepy/examples/diffusion/poisson.py
```

- `homogen` - calculation of local microscopic problems (correctors) and homogenized coefficients. Example:

```
sfepy-run sfepy/examples/homogenization/perfusion_micro.py
```

- bvp-mM - micro-macro boundary value problem. Solve a coupled two-scale problem in parallel using MPI. One computational node is solving a macroscopic equation while the others are solving local microscopic problems and homogenized coefficients. The `--app` option is required in this case. Example:

```
mpiexec -n 4 sfepy-run --app=bvp-mM --debug-mpi sfepy/examples/homogenization/  
↪nonlinear_hyperelastic_mM.py
```

- evp - eigenvalue problem. Example:

```
sfepy-run sfepy/examples/quantum/well.py
```

- phonon - phononic band gaps. Example:

```
sfepy-run sfepy/examples/phononic/band_gaps.py --phonon-plot
```

Both normal and parametric study runs are supported. A parametric study allows repeated runs for varying some of the simulation parameters - see `sfepy/examples/diffusion/poisson_parametric_study.py` file.

```
sfepy.scripts.simple.main()
```

```
sfepy.scripts.simple.print_solvers()
```

```
sfepy.scripts.simple.print_terms()
```

Tests

`sfepy.tests.conftest` module

```
sfepy.tests.conftest.output_dir(request, tmpdir_factory)
```

Output directory for tests.

```
sfepy.tests.conftest.pytest_addoption(parser)
```

```
sfepy.tests.conftest.pytest_configure(config)
```

`sfepy.tests.test_assembling` module

```
sfepy.tests.test_assembling.data()
```

```
sfepy.tests.test_assembling.test_assemble_matrix(data)
```

```
sfepy.tests.test_assembling.test_assemble_matrix_complex(data)
```

```
sfepy.tests.test_assembling.test_assemble_vector(data)
```

```
sfepy.tests.test_assembling.test_assemble_vector_complex(data)
```

sfepy.tests.test_base module

```
sfepy.tests.test_base.test_container_add()  
sfepy.tests.test_base.test_parse_conf()  
sfepy.tests.test_base.test_resolve_deps()  
sfepy.tests.test_base.test_struct_add()  
sfepy.tests.test_base.test_struct_i_add()  
sfepy.tests.test_base.test_verbose_output()
```

sfepy.tests.test_cmesh module

```
sfepy.tests.test_cmesh.filename_mesher()  
sfepy.tests.test_cmesh.test_cmesh_counts(filename_mesher)  
sfepy.tests.test_cmesh.test_entity_volumes()
```

sfepy.tests.test_conditions module

```
sfepy.tests.test_conditions.check_vec(vec, ii, ok, conds, variables)  
sfepy.tests.test_conditions.data()  
sfepy.tests.test_conditions.init_vec(variables)  
sfepy.tests.test_conditions.test_ebcs(data)  
sfepy.tests.test_conditions.test_epbcs(data)  
sfepy.tests.test_conditions.test_ics(data)  
sfepy.tests.test_conditions.test_save_ebc(data, output_dir)
```

sfepy.tests.test_declarative_examples module

```
sfepy.tests.test_declarative_examples.inedir(filename)  
sfepy.tests.test_declarative_examples.test_examples(ex_filename, output_dir)  
sfepy.tests.test_declarative_examples.test_examples_dg(ex_filename, output_dir)
```

`sfepy.tests.test_dg_field` module

```
class sfepy.tests.test_dg_field.TestDGField
    test_create_output1D()
    test_create_output2D()
    test_get_bc_facet_values_1D()
    test_get_bc_facet_values_2D()
    test_get_bc_facet_values_2D_const()
    test_get_facet_idx1D()
    test_get_facet_idx2D()
    test_get_facet_neighbor_idx_1d()
    test_get_facet_neighbor_idx_2d()
    test_set_dofs_1D()
    test_set_dofs_2D()

sfepy.tests.test_dg_field.prepare_dgfield(approx_order, mesh)
sfepy.tests.test_dg_field.prepare_dgfield_1D(approx_order)
sfepy.tests.test_dg_field.prepare_field_2D(approx_order)
```

`sfepy.tests.test_dg_terms_calls` module

Test all terms in `terms_dg`. Performs numerical test on simple mesh.

```
class sfepy.tests.test_dg_terms_calls.DGTermTestEnvornment(dim, approx_order, **kwargs)
```

Class for easy creation of all the data needed for testing terms.

```
burg_fun(u)
```

```
burg_fun_d(u)
```

```
prepare_materials(field, velo=1.0, diffusion=0.1, penalty=100)
```

Crates material objects with data attribute, containing properly shaped data to pass to terms

Parameters

- **field** – DGField
- **velo** – optional values for velocity $\mathbf{a}$
- **diffusion** – optional value for diffusion tensor $\mathbf{D}$
- **penalty** – optional value for diffusion penalty C_w

Returns

$\mathbf{a}$, $\mathbf{D}$, C_w

prepare_variables(*field*)

Prepares state and test variables, adds empty eq_map to state variable

Parameters

field

Returns

state, test

class sfepy.tests.test_dg_terms_calls.**TestAdvectDGFluxTerm**

test_function_explicit_1D(*dg_test_env*)

test_function_implicit_1D(*dg_test_env*)

class sfepy.tests.test_dg_terms_calls.**TestDiffusionDGFluxTerm**

test_function_explicit_left_1D(*dg_test_env*)

test_function_explicit_right_1D(*dg_test_env*)

test_function_implicit_left_1D(*dg_test_env*)

test_function_implicit_right_1D(*dg_test_env*)

class sfepy.tests.test_dg_terms_calls.**TestDiffusionInteriorPenaltyTerm**

test_function_explicit_1D(*dg_test_env*)

test_function_implicit_1D(*dg_test_env*)

class sfepy.tests.test_dg_terms_calls.**TestNonlinScalarDotGradTerm**

test_function_explicit_1D(*dg_test_env*)

class sfepy.tests.test_dg_terms_calls.**TestNonlinearHyperDGFluxTerm**

test_function_explicit_1D(*dg_test_env*)

sfepy.tests.test_dg_terms_calls.**dg_test_env**(*request*)

sfepy.tests.test_domain module

sfepy.tests.test_domain.**compare_mesh**(*geo_name, coors, conn*)

sfepy.tests.test_domain.**domain**()

sfepy.tests.test_domain.**refine**(*domain, out_dir, level=3*)

sfepy.tests.test_domain.**test_facets**(*domain*)

sfepy.tests.test_domain.**test_refine_2_3**(*output_dir*)

sfepy.tests.test_domain.**test_refine_2_4**(*output_dir*)

sfepy.tests.test_domain.**test_refine_3_4**(*output_dir*)

sfepy.tests.test_domain.**test_refine_3_8**(*output_dir*)

sfepy.tests.test_domain.**test_refine_hexa**(*output_dir*)

sfepy.tests.test_domain.**test_refine_tetra**(*domain, output_dir*)

sfepy.tests.test_ed_solvers module

```
sfepy.tests.test_ed_solvers.define(t1=1.5e-05, dt=1e-06, dims=(0.1, 0.02, 0.005), shape=(11, 3, 3),  
                                     young=70000000000.0, poisson=0.3, density=2700,  
                                     mass_lumping='row_sum', mass_beta=0.2)
```

```
sfepy.tests.test_ed_solvers.problem()
```

```
sfepy.tests.test_ed_solvers.test_active_only(output_dir)
```

Note: with tsc the results would differ, as `eval_scaled_norm()` depends on the vector length.

```
sfepy.tests.test_ed_solvers.test_ed_solvers(problem, output_dir)
```

```
sfepy.tests.test_ed_solvers.test_rmm_solver(problem, output_dir)
```

sfepy.tests.test_eigenvalue_solvers module

```
sfepy.tests.test_eigenvalue_solvers.data()
```

```
sfepy.tests.test_eigenvalue_solvers.mesh_hook(mesh, mode)
```

Generate the block mesh.

```
sfepy.tests.test_eigenvalue_solvers.test_eigenvalue_solvers(data)
```

sfepy.tests.test_elasticity_small_strain module

```
sfepy.tests.test_elasticity_small_strain.get_pars(dim, full=False)
```

```
sfepy.tests.test_elasticity_small_strain.solutions(output_dir)
```

```
sfepy.tests.test_elasticity_small_strain.test_converged(solutions)
```

```
sfepy.tests.test_elasticity_small_strain.test_linear_terms(solutions)
```

sfepy.tests.test_fem module

```
sfepy.tests.test_fem.gels()
```

```
sfepy.tests.test_fem.test_basis_functions_delta(gels)
```

Test δ property of basis functions evaluated in the reference element nodes.

```
sfepy.tests.test_fem.test_basis_functions_values(gels)
```

Compare basis function values and their gradients with correct data. Also test that sum of values over all element nodes gives one.

sfepy.tests.test_functions module

```
sfepy.tests.test_functions.get_circle(coors, domain=None)
sfepy.tests.test_functions.get_p_edge(ts, coors, bc=None, **kwargs)
sfepy.tests.test_functions.get_pars(ts, coors, mode=None, extra_arg=None, equations=None,
                                     term=None, problem=None, **kwargs)
sfepy.tests.test_functions.get_u_edge(ts, coors, bc=None, **kwargs)
sfepy.tests.test_functions.problem()
sfepy.tests.test_functions.test_ebc_functions(problem, output_dir)
sfepy.tests.test_functions.test_material_functions(problem)
sfepy.tests.test_functions.test_region_functions(problem, output_dir)
```

sfepy.tests.test_high_level module

```
sfepy.tests.test_high_level.data()
sfepy.tests.test_high_level.fix_u_fun(ts, coors, bc=None, problem=None, extra_arg=None)
sfepy.tests.test_high_level.test_solving(data, output_dir)
sfepy.tests.test_high_level.test_term_arithmetics(data)
sfepy.tests.test_high_level.test_term_evaluation(data)
sfepy.tests.test_high_level.test_variables(data)
```

sfepy.tests.test_homogenization_engine module

```
sfepy.tests.test_homogenization_engine.test_chunk_micro()
sfepy.tests.test_homogenization_engine.test_dependencies()
```

sfepy.tests.test_homogenization_perfusion module

```
sfepy.tests.test_homogenization_perfusion.compare_scalars(s1, s2, l1='s1', l2='s2',
                                                           allowed_error=1e-08)
sfepy.tests.test_homogenization_perfusion.test_solution(output_dir)
```

sfepy.tests.test_hyperelastic_tlul module

`sfepy.tests.test_hyperelastic_tlul.test_solution(output_dir)`

sfepy.tests.test_io module

`sfepy.tests.test_io.test_recursive_dict_hdf5(output_dir)`

`sfepy.tests.test_io.test_sparse_matrix_hdf5(output_dir)`

sfepy.tests.test_laplace_unit_disk module

`sfepy.tests.test_laplace_unit_disk.data()`

`sfepy.tests.test_laplace_unit_disk.test_boundary_fluxes(data)`

sfepy.tests.test_laplace_unit_square module

`sfepy.tests.test_laplace_unit_square.data()`

`sfepy.tests.test_laplace_unit_square.linear(bc, ts, coor, which)`

`sfepy.tests.test_laplace_unit_square.linear_x(bc, ts, coor)`

`sfepy.tests.test_laplace_unit_square.linear_y(bc, ts, coor)`

`sfepy.tests.test_laplace_unit_square.linear_z(bc, ts, coor)`

`sfepy.tests.test_laplace_unit_square.test_boundary_fluxes(data, output_dir)`

`sfepy.tests.test_laplace_unit_square.test_solution(data)`

sfepy.tests.test_lcbcs module

`sfepy.tests.test_lcbcs.test_elasticity_rigid(mesh_filename, output_dir)`

`sfepy.tests.test_lcbcs.test_laplace_shifted_periodic(output_dir)`

`sfepy.tests.test_lcbcs.test_stokes_slip_bc(output_dir)`

sfepy.tests.test_linalg module

`sfepy.tests.test_linalg.test_assemble1d()`

`sfepy.tests.test_linalg.test_geometry()`

`sfepy.tests.test_linalg.test_get_blocks_stats()`

`sfepy.tests.test_linalg.test_tensors()`

`sfepy.tests.test_linalg.test_unique_rows()`

sfepy.tests.test_linear_solvers module

class sfepy.tests.test_linear_solvers.**DiagPC**

Diagonal (Jacobi) preconditioner.

Equivalent to setting '*precond*' : '*jacobi*'.

apply(*pc*, *x*, *y*)

setUp(*pc*)

sfepy.tests.test_linear_solvers.**problem**()

sfepy.tests.test_linear_solvers.**setup_petsc_precond**(*mtx*, *problem*)

sfepy.tests.test_linear_solvers.**test_ls_reuse**(*problem*)

sfepy.tests.test_linear_solvers.**test_solvers**(*problem*, *output_dir*)

sfepy.tests.test_linearization module

sfepy.tests.test_linearization.**test_linearization**(*output_dir*)

sfepy.tests.test_log module

sfepy.tests.test_log.**log**(*log_filename*)

sfepy.tests.test_log.**log_filename**(*output_dir*)

sfepy.tests.test_log.**test_log_rw**(*log_filename*, *log*, *output_dir*)

sfepy.tests.test_matcoefs module

sfepy.tests.test_matcoefs.**test_conversion_functions**()

sfepy.tests.test_matcoefs.**test_elastic_constants**()

sfepy.tests.test_matcoefs.**test_stiffness_tensors**()

sfepy.tests.test_matcoefs.**test_wave_speeds**()

sfepy.tests.test_mesh_expand module

sfepy.tests.test_mesh_expand.**test_mesh_expand**()

sfePy.tests.test_mesh_generators module

`sfePy.tests.test_mesh_generators.test_gen_block_mesh(output_dir)`
`sfePy.tests.test_mesh_generators.test_gen_cylinder_mesh(output_dir)`
`sfePy.tests.test_mesh_generators.test_gen_extended_block_mesh(output_dir)`
`sfePy.tests.test_mesh_generators.test_gen_mesh_from_geom(output_dir)`
`sfePy.tests.test_mesh_generators.test_gen_mesh_from_voxels(output_dir)`
`sfePy.tests.test_mesh_generators.test_gen_tiled_mesh(output_dir)`

sfePy.tests.test_mesh_interp module

`sfePy.tests.test_mesh_interp.do_interpolation(m2, m1, data, field_name, force=False)`
Interpolate data from m1 to m2.
`sfePy.tests.test_mesh_interp.gen_datas(meshes)`
`sfePy.tests.test_mesh_interp.in_dir(adir)`
`sfePy.tests.test_mesh_interp.prepare_variable(filename, n_components)`
`sfePy.tests.test_mesh_interp.test_evaluate_at()`
`sfePy.tests.test_mesh_interp.test_field_gradient()`
`sfePy.tests.test_mesh_interp.test_interpolation(output_dir)`
`sfePy.tests.test_mesh_interp.test_interpolation_two_meshes(output_dir)`
`sfePy.tests.test_mesh_interp.test_invariance()`
`sfePy.tests.test_mesh_interp.test_invariance_qp()`

sfePy.tests.test_mesh_smoothing module

`sfePy.tests.test_mesh_smoothing.get_volume(el, nd)`
`sfePy.tests.test_mesh_smoothing.test_mesh_smoothing(output_dir)`

sfePy.tests.test_meshio module

`sfePy.tests.test_meshio.mesh_hook(mesh, mode)`
Define a mesh programmatically.
`sfePy.tests.test_meshio.test_compare_same_meshes()`
Compare same meshes in various formats.
`sfePy.tests.test_meshio.test_hdf5_meshio()`
`sfePy.tests.test_meshio.test_read_dimension()`

`sfepy.tests.test_meshio.test_read_meshes()`

Try to read all listed meshes.

`sfepy.tests.test_meshio.test_write_read_meshes(output_dir)`

Try to write and then read all supported formats.

sfepy.tests.test_msm_laplace module

`sfepy.tests.test_msm_laplace.ebc(ts, coor, **kwargs)`

`sfepy.tests.test_msm_laplace.problem()`

`sfepy.tests.test_msm_laplace.rhs(ts, coor, mode=None, expression=None, **kwargs)`

`sfepy.tests.test_msm_laplace.test_msm_laplace(problem, output_dir)`

sfepy.tests.test_msm_symbolic module

`sfepy.tests.test_msm_symbolic.ebc(ts, coor, solution=None)`

`sfepy.tests.test_msm_symbolic.problem()`

`sfepy.tests.test_msm_symbolic.rhs(ts, coor, mode=None, expression=None, **kwargs)`

`sfepy.tests.test_msm_symbolic.test_msm_symbolic_diffusion(problem, output_dir)`

`sfepy.tests.test_msm_symbolic.test_msm_symbolic_laplace(problem, output_dir)`

sfepy.tests.test_normals module

`sfepy.tests.test_normals.test_normals()`

Check orientations of surface normals on the reference elements.

sfepy.tests.test_parsing module

`sfepy.tests.test_parsing.test_parse_equations()`

`sfepy.tests.test_parsing.test_parse_regions()`

sfepy.tests.test_poly_spaces module

Test continuity of polynomial basis and its gradients along an edge on y line (2D) or on a face in x - y plane (3D) between two elements aligned with the coordinate system, stack one on top of the other. The evaluation occurs in several points shifted by a very small amount from the boundary between the elements into the top and the bottom element.

For H1 space, the basis should be continuous. The components of its gradient parallel to the edge/face should be continuous as well, while the perpendicular component should have the same absolute value, but different sign in the top and the bottom element.

All connectivity permutations of the two elements are tested.

The serendipity basis implementation is a pure python proof-of-concept. Its order in continuity tests is limited to 2 on 3_8 elements to decrease the tests run time.

```
sfepy.tests.test_poly_spaces.gels()
sfepy.tests.test_poly_spaces.test_continuity(gels)
sfepy.tests.test_poly_spaces.test_gradients(gels)
sfepy.tests.test_poly_spaces.test_hessians(gels)
    Test the second partial derivatives of basis functions using finite differences.
sfepy.tests.test_poly_spaces.test_partition_of_unity(gels)
```

sfepy.tests.test_projections module

```
sfepy.tests.test_projections.data()
sfepy.tests.test_projections.test_mass_matrix(data)
sfepy.tests.test_projections.test_project_tensors(data)
sfepy.tests.test_projections.test_projection_iga_fem()
sfepy.tests.test_projections.test_projection_tri_quad(data, output_dir)
```

sfepy.tests.test_quadratures module

```
sfepy.tests.test_quadratures.get_poly(order, dim, is_simplex=False)
    Construct a polynomial of given order in space dimension dim, and integrate it symbolically over a rectangular
    or simplex domain for coordinates in [0, 1].
sfepy.tests.test_quadratures.symarray(prefix, shape)
    Copied from SymPy so that the tests pass for its different versions.
sfepy.tests.test_quadratures.test_quadratures()
    Test if the quadratures have orders they claim to have, using symbolic integration by sympy.
sfepy.tests.test_quadratures.test_weight_consistency()
    Test if integral of 1 (= sum of weights) gives the domain volume.
```

sfepy.tests.test_ref_coors module

```
sfepy.tests.test_ref_coors.test_ref_coors_fem()
sfepy.tests.test_ref_coors.test_ref_coors_iga()
```

sfepy.tests.test_refine_hanging module

Test continuity along a boundary with hanging nodes due to a mesh refinement.

`sfepy.tests.test_refine_hanging.eval_fun(ts, coors, mode, **kwargs)`

`sfepy.tests.test_refine_hanging.gels()`

`sfepy.tests.test_refine_hanging.test_continuity(gels, output_dir)`

`sfepy.tests.test_refine_hanging.test_preserve_coarse_entities(output_dir)`

sfepy.tests.test_regions module

`sfepy.tests.test_regions.data()`

`sfepy.tests.test_regions.get_cells(coors, domain=None)`

`sfepy.tests.test_regions.get_vertices(coors, domain=None)`

`sfepy.tests.test_regions.test_operators(data)`

Test operators in region selectors.

`sfepy.tests.test_regions.test_selectors(data)`

Test basic region selectors.

sfepy.tests.test_semismooth_newton module

`sfepy.tests.test_semismooth_newton.convert_to_csr(m_in)`

`sfepy.tests.test_semismooth_newton.define_matrices()`

`sfepy.tests.test_semismooth_newton.eval_matrix(mtx, **kwargs)`

`sfepy.tests.test_semismooth_newton.test_semismooth_newton()`

sfepy.tests.test_sparse module

`sfepy.tests.test_sparse.test_compose_sparse()`

sfepy.tests.test_splinebox module

`sfepy.tests.test_splinebox.test_spbox_2d()`

Check position of a given vertex in the deformed mesh.

`sfepy.tests.test_splinebox.test_spbox_3d()`

Check volume change of the mesh which is deformed using the SplineBox functions.

`sfepy.tests.test_splinebox.test_spbox_field()`

‘Field’ vs. ‘coors’.

`sfepy.tests.test_splinebox.test_spreigion2d()`

Check position of a given vertex in the deformed mesh.

`sfepy.tests.test_splinebox.tetravolume(cells, vertices)`

sfepy.tests.test_tensors module

`sfepy.tests.test_tensors.get_ortho_d(phi1, phi2)`

`sfepy.tests.test_tensors.test_stress_transform()`

`sfepy.tests.test_tensors.test_tensors()`

`sfepy.tests.test_tensors.test_transform_data()`

`sfepy.tests.test_tensors.test_transform_data4()`

sfepy.tests.test_term_call_modes module

`sfepy.tests.test_term_call_modes.data()`

`sfepy.tests.test_term_call_modes.make_term_args(arg_shapes, arg_kinds, arg_types, ats_mode, domain, material_value=None, poly_space_basis=None)`

`sfepy.tests.test_term_call_modes.test_term_call_modes(data)`

sfepy.tests.test_term_consistency module

`sfepy.tests.test_term_consistency.get_pars(ts, coor, mode=None, term=None, **kwargs)`

`sfepy.tests.test_term_consistency.problem()`

`sfepy.tests.test_term_consistency.test_consistency_d_dw(problem)`

`sfepy.tests.test_term_consistency.test_ev_div(problem)`

`sfepy.tests.test_term_consistency.test_ev_grad(problem)`

`sfepy.tests.test_term_consistency.test_eval_matrix(problem)`

`sfepy.tests.test_term_consistency.test_surface_evaluate(problem)`

`sfepy.tests.test_term_consistency.test_vector_matrix(problem)`

sfepy.tests.test_term_sensitivity module

`sfepy.tests.test_term_sensitivity.modify_mesh(val, spbox, dv_mode, cp_pos)`

`sfepy.tests.test_term_sensitivity.problem()`

`sfepy.tests.test_term_sensitivity.test_sensitivity(problem)`

sfepy.tests.test_units module

`sfepy.tests.test_units.test_consistent_sets()`

`sfepy.tests.test_units.test_units()`

sfepy.tests.test_volume module

Test computing volumes by volume or surface integrals.

`sfepy.tests.test_volume.problem()`

`sfepy.tests.test_volume.test_volume(problem)`

`sfepy.tests.test_volume.test_volume_t1(problem)`

Tools

tools/build_helpers.py script

Build helpers for setup.py.

Notes

The original version of this file was adapted from NiPy project [1].

[1] <http://nipy.sourceforge.net/>

class build_helpers.Clean(*dist: Distribution, **kw*)

Command class to clean, enhanced to clean also files generated during *python setup.py build_ext -inplace*.

run()

After calling the super class implementation, this function removes the directories specific to scikit-build.

class build_helpers.DoxygenDocs(*dist: Distribution, **kw*)

description = 'generate docs by Doxygen'

run()

A command's raison d'être: carry out the action it exists to perform, controlled by the options initialized in 'initialize_options()', customized by other commands, the setup script, the command-line, and config files, and finalized in 'finalize_options()'. All terminal output and filesystem interaction should be done by 'run()'.

This method must be implemented by all command classes.

class build_helpers.NoOptionsDocs(*dist: Distribution, **kw*)

finalize_options()

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if 'foo' depends on 'bar', then it is safe to set 'foo' from 'bar' as long as 'foo' still has the same value it was assigned in 'initialize_options()'.

This method must be implemented by all command classes.

initialize_options()

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, ‘initialize_options()’ implementations are just a bunch of “self.foo = None” assignments.

This method must be implemented by all command classes.

```
user_options = [('None', None, 'this command has no options')]
```

```
class build_helpers.SphinxHTMLDocs(dist: Distribution, **kw)
```

```
description = 'generate html docs by Sphinx'
```

run()

A command’s raison d’etre: carry out the action it exists to perform, controlled by the options initialized in ‘initialize_options()’, customized by other commands, the setup script, the command-line, and config files, and finalized in ‘finalize_options()’. All terminal output and filesystem interaction should be done by ‘run()’.

This method must be implemented by all command classes.

```
class build_helpers.SphinxPDFDocs(dist: Distribution, **kw)
```

```
description = 'generate pdf docs by Sphinx'
```

run()

A command’s raison d’etre: carry out the action it exists to perform, controlled by the options initialized in ‘initialize_options()’, customized by other commands, the setup script, the command-line, and config files, and finalized in ‘finalize_options()’. All terminal output and filesystem interaction should be done by ‘run()’.

This method must be implemented by all command classes.

```
build_helpers.get_sphinx_make_command()
```

```
build_helpers.have_good_cython()
```

```
build_helpers.package_check(pkg_name, version=None, optional=False, checker=<function parse>, version_getter=None, messages=None, show_only=False)
```

Check if package *pkg_name* is present, and in correct version.

Parameters**pkg_name**

[str or sequence of str] The name of the package as imported into python. Alternative names (e.g. for different versions) may be given in a list.

version

[str, optional] The minimum version of the package that is required. If not given, the version is not checked.

optional

[bool, optional] If False, raise error for absent package or wrong version; otherwise warn

checker

[callable, optional] If given, the callable with which to return a comparable thing from a version string. The default is `packaging.version.parse()`.

version_getter

[callable, optional:] If given, the callable that takes *pkg_name* as argument, and returns the package version string - as in:

```
``version = version_getter(pkg_name)``
```

The default is equivalent to:

```
mod = __import__(pkg_name); version = mod.__version__``
```

messages

[dict, optional] If given, the dictionary providing (some of) output messages.

show_only

[bool] If True, do not raise exceptions, only show the package name and version information.

`build_helpers.recursive_glob(top_dir, pattern)`

Utility function working like `glob.glob()`, but working recursively and returning generator.

Parameters**topdir**

[str] The top-level directory.

pattern

[str or list of str] The pattern or list of patterns to match.

tools/gen_field_table.py script

Generate available fields table for ReST documentation.

`gen_field_table.gen_field_table(app)`

`gen_field_table.main()`

`gen_field_table.setup(app)`

`gen_field_table.typeset(fd)`

Utility function called by Sphinx.

`gen_field_table.typeset_field_table(fd, field_table)`

tools/gen_gallery.py script

Generate the images and rst files for gallery of SfePy examples.

The following steps need to be made to regenerate the documentation with the updated example files:

1. remove doc/examples/*:

```
$ rm -rf doc/examples/*
```

2. generate the files:

```
$ ./tools/gen_gallery.py
```

3. regenerate the documentation:

```
$ python setup.py htmldocs
```

```
gen_gallery.apply_view_options(views, default)
```

```
gen_gallery.ebase2fbase(ebase)
```

```
gen_gallery.generate_gallery(examples_dir, output_filename, doc_dir, rst_dir, thumbnails_dir, dir_map,  
                             n_col=3)
```

Generate the gallery rst file with thumbnail images and links to examples.

Parameters

output_filename

[str] The output rst file name.

doc_dir

[str] The top level directory of gallery files.

rst_dir

[str] The full path to rst files of examples within *doc_dir*.

thumbnails_dir

[str] The full path to thumbnail images within *doc_dir*.

dir_map

[dict] The directory mapping returned by *generate_rst_files()*

n_col

[int] The number of columns in the gallery table.

```
gen_gallery.generate_images(images_dir, examples_dir, pattern='*.py')
```

Generate images from results of running examples found in *examples_dir* directory.

The generated images are stored to *images_dir*,

```
gen_gallery.generate_rst_files(rst_dir, examples_dir, images_dir, pattern='*.py')
```

Generate Sphinx rst files for examples in *examples_dir* with images in *images_dir* and put them into *rst_dir*.

Returns

dir_map

[dict] The directory mapping of examples and corresponding rst files.

```
gen_gallery.generate_thumbnails(thumbnails_dir, images_dir, scale=0.3)
```

Generate thumbnails into *thumbnails_dir* corresponding to images in *images_dir*.

```
gen_gallery.main()
```

```
gen_gallery.resview_plot(filename, filename_out, options)
```

```
gen_gallery.run_resview_plot(*args)
```

A fix for the problem that calling *resview_plot()* directly often terminates the program.

tools/gen_legendre_simplex_base.py script

Generate simplex legendre 2D basis coefficients and exponents matrices and save them to legendre2D_simplex_coefs.txt and legendre2D_simplex_expos.txt

```
gen_legendre_simplex_base.main()
```

tools/gen_lobatto1d_c.py script

Generate lobatto1d.c and lobatto1h.c files.

```
gen_lobatto1d_c.append_declarations(out, cpolys, comment, cvar_name, shift=0)
```

```
gen_lobatto1d_c.append_lists(out, names, length)
```

```
gen_lobatto1d_c.append_polys(out, cpolys, comment, cvar_name, var_name='x', shift=0)
```

```
gen_lobatto1d_c.gen_lobatto(max_order)
```

```
gen_lobatto1d_c.main()
```

```
gen_lobatto1d_c.plot_polys(fig, polys, var_name='x')
```

tools/gen_release_notes.py script

Generate release notes using git log starting from the given version.

```
gen_release_notes.main()
```

tools/gen_serendipity_basis.py script

```
python3 tools/gen_serendipity_basis.py > sfepy/discrete/fem/_serendipity.py
```

```
gen_serendipity_basis.main()
```

tools/gen_solver_table.py script

Generate available solvers table for ReST documentation.

```
gen_solver_table.gen_solver_table(app)
```

```
gen_solver_table.main()
```

```
gen_solver_table.setup(app)
```

```
gen_solver_table.trim(docstring)
```

Trim and split (doc)string.

```
gen_solver_table.typeset(fd)
```

Utility function called by Sphinx.

```
gen_solver_table.typeset_solvers_table(fd, solver_table)
```

Generate solvers table ReST output.

tools/gen_term_table.py script

Generate the table of all terms for the sphinx documentation.

`gen_term_table.create_parser(slist, current_section)`

`gen_term_table.format_next(text, new_text, pos, can_newline, width, ispaces)`

`gen_term_table.gen_term_table(app)`

`gen_term_table.get_examples(table)`

`gen_term_table.main()`

`gen_term_table.set_section(sec)`

`gen_term_table.setup(app)`

`gen_term_table.to_list(slist, sec)`

`gen_term_table.typeset(filename)`

Utility function called by sphinx.

`gen_term_table.typeset_examples(term_class, term_use)`

`gen_term_table.typeset_term_syntax(term_class)`

`gen_term_table.typeset_term_table(fd, keys, table, title)`

Terms are sorted by name without the d*_ prefix.

`gen_term_table.typeset_term_tables(fd, table)`

Generate tables: basic, sensitivity, special.

`gen_term_table.typeset_to_indent(txt, indent0, indent, width)`

tools/install_data.py script

`class install_data.install_data(dist: Distribution, **kw)`

`finalize_options()`

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if 'foo' depends on 'bar', then it is safe to set 'foo' from 'bar' as long as 'foo' still has the same value it was assigned in 'initialize_options()'.
This method must be implemented by all command classes.

tools/show_authors.py script

`show_authors.main()`

tools/show_terms_use.py script

Show terms use in problem description files in the given directory.

`show_terms_use.main()`

tools/sync_module_docs.py script

Synchronize the documentation files in a given directory `doc_dir` with the actual state of the SfePy sources in `top_dir`. Missing files are created, files with no corresponding source file are removed, other files are left untouched.

Notes

The developer guide needs to be edited manually to reflect the changes.

`sync_module_docs.main()`

BIBLIOGRAPHY

- [Logg2012] A. Logg: Efficient Representation of Computational Meshes. 2012
- [1] Remacle, J.-F., Chevaugeon, N., Marchandise, E., & Geuzaine, C. (2007). Efficient visualization of high-order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4), 750-771. <https://doi.org/10.1002/nme.1787>
- [1] Krivodonova (2007): Limiters for high-order discontinuous Galerkin methods
- [1] Hesthaven, J. S., & Warburton, T. (2008). Nodal Discontinuous Galerkin Methods. *Journal of Physics A: Mathematical and Theoretical* (Vol. 54). New York, NY: Springer New York. <http://doi.org/10.1007/978-0-387-72067-8>, p. 63
- [1] Gottlieb, S., & Shu, C.-W. (2002). Total variation diminishing Runge-Kutta schemes. *Mathematics of Computation of the American Mathematical Society*, 67(221), 73-85. <https://doi.org/10.1090/s0025-5718-98-00913-2>
- [1] González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018. Inverse mass matrix via the method of localized Lagrange multipliers. *International Journal for Numerical Methods in Engineering* 113, 277-295. <https://doi.org/10.1002/nme.5613>
- [2] González, J.A., Kopačka, J., Kolman, R., Cho, S.S., Park, K.C., 2019. Inverse mass matrix for isogeometric explicit transient analysis via the method of localized Lagrange multipliers. *International Journal for Numerical Methods in Engineering* 117, 939-966. <https://doi.org/10.1002/nme.5986>
- [1] Swope, William C.; H. C. Andersen; P. H. Berens; K. R. Wilson (1 January 1982). "A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters". *The Journal of Chemical Physics*. 76 (1): 648 (Appendix). doi:10.1063/1.442716
- [1] González, J.A., Kolman, R., Cho, S.S., Felippa, C.A., Park, K.C., 2018. Inverse mass matrix via the method of localized Lagrange multipliers. *International Journal for Numerical Methods in Engineering* 113, 277-295. <https://doi.org/10.1002/nme.5613>
- [1] Zemčík, R., Rolfes, R., Rose, M. and Tessmer, J. (2006),
- [2] Zemčík, R., Rolfes, R., Rose, M. and Teßmer, J. (2007),

PYTHON MODULE INDEX

b

build_helpers, 1201

g

gen_field_table, 1203
gen_gallery, 1203
gen_legendre_simplex_base, 1205
gen_lobattoid_c, 1205
gen_release_notes, 1205
gen_serendipity_basis, 1205
gen_solver_table, 1205
gen_term_table, 1206

i

install_data, 1206

s

sfepy.applications.application, 778
sfepy.applications.evp_solver_app, 778
sfepy.applications.pde_solver_app, 779
sfepy.base.base, 780
sfepy.base.compat, 786
sfepy.base.conf, 790
sfepy.base.getch, 792
sfepy.base.goptions, 793
sfepy.base.ioutils, 793
sfepy.base.log, 798
sfepy.base.log_plotter, 800
sfepy.base.mem_usage, 800
sfepy.base.multiproc, 801
sfepy.base.multiproc_mpi, 801
sfepy.base.multiproc_proc, 804
sfepy.base.parse_conf, 805
sfepy.base.plotutils, 806
sfepy.base.reader, 806
sfepy.base.resolve_deps, 806
sfepy.base.testing, 807
sfepy.base.timing, 807
sfepy.config, 777
sfepy.discrete.common.dof_info, 857
sfepy.discrete.common.domain, 860
sfepy.discrete.common.extmods._fmfield, 861

sfepy.discrete.common.extmods._geommech, 861
sfepy.discrete.common.extmods.assemble, 861
sfepy.discrete.common.extmods.cmapping, 861
sfepy.discrete.common.extmods.cmesh, 861
sfepy.discrete.common.extmods.crefcoors, 864
sfepy.discrete.common.fields, 864
sfepy.discrete.common.global_interp, 867
sfepy.discrete.common.mappings, 870
sfepy.discrete.common.poly_spaces, 872
sfepy.discrete.common.region, 873
sfepy.discrete.conditions, 808
sfepy.discrete.dg.dg_1D_vizualizer, 914
sfepy.discrete.dg.fields, 919
sfepy.discrete.dg.limiters, 931
sfepy.discrete.dg.poly_spaces, 928
sfepy.discrete.equations, 811
sfepy.discrete.evaluate, 818
sfepy.discrete.evaluate_variable, 822
sfepy.discrete.fem._serendipity, 912
sfepy.discrete.fem.domain, 877
sfepy.discrete.fem.extmods.bases, 878
sfepy.discrete.fem.extmods.lobatto_bases, 878
sfepy.discrete.fem.facets, 879
sfepy.discrete.fem.fe_surface, 880
sfepy.discrete.fem.fields_base, 881
sfepy.discrete.fem.fields_hierarchic, 887
sfepy.discrete.fem.fields_l2, 887
sfepy.discrete.fem.fields_nodal, 889
sfepy.discrete.fem.fields_positive, 890
sfepy.discrete.fem.geometry_element, 891
sfepy.discrete.fem.history, 892
sfepy.discrete.fem.lcbc_operators, 892
sfepy.discrete.fem.linearizer, 895
sfepy.discrete.fem.mappings, 896
sfepy.discrete.fem.mesh, 898
sfepy.discrete.fem.meshio, 900
sfepy.discrete.fem.periodic, 907
sfepy.discrete.fem.poly_spaces, 907
sfepy.discrete.fem.refine, 911
sfepy.discrete.fem.refine_hanging, 912
sfepy.discrete.fem.utils, 912
sfepy.discrete.functions, 822

`sfepy.discrete.iga.domain`, 934
`sfepy.discrete.iga.domain_generators`, 935
`sfepy.discrete.iga.extmods.igac`, 936
`sfepy.discrete.iga.fields`, 938
`sfepy.discrete.iga.iga`, 940
`sfepy.discrete.iga.io`, 946
`sfepy.discrete.iga.mappings`, 947
`sfepy.discrete.iga.plot_nurbs`, 948
`sfepy.discrete.iga.utils`, 948
`sfepy.discrete.integrals`, 823
`sfepy.discrete.materials`, 824
`sfepy.discrete.parse_equations`, 827
`sfepy.discrete.parse_regions`, 827
`sfepy.discrete.probes`, 828
`sfepy.discrete.problem`, 832
`sfepy.discrete.projections`, 844
`sfepy.discrete.quadratures`, 845
`sfepy.discrete.simplex_cubature`, 847
`sfepy.discrete.structural.fields`, 949
`sfepy.discrete.structural.mappings`, 950
`sfepy.discrete.variables`, 847
`sfepy.homogenization.band_gaps_app`, 950
`sfepy.homogenization.coefficients`, 951
`sfepy.homogenization.coefs_base`, 952
`sfepy.homogenization.coefs_phononic`, 954
`sfepy.homogenization.convolution`, 960
`sfepy.homogenization.engine`, 960
`sfepy.homogenization.homogen_app`, 963
`sfepy.homogenization.micmac`, 964
`sfepy.homogenization.recovery`, 964
`sfepy.homogenization.utils`, 967
`sfepy.linalg.check_derivatives`, 969
`sfepy.linalg.eigen`, 969
`sfepy.linalg.geometry`, 970
`sfepy.linalg.sparse`, 973
`sfepy.linalg.sympy_operators`, 974
`sfepy.linalg.utils`, 975
`sfepy.mechanics.contact_bodies`, 979
`sfepy.mechanics.elastic_constants`, 979
`sfepy.mechanics.extmods.ccontres`, 993
`sfepy.mechanics.matcoefs`, 979
`sfepy.mechanics.membranes`, 983
`sfepy.mechanics.shell10x`, 986
`sfepy.mechanics.tensors`, 988
`sfepy.mechanics.units`, 991
`sfepy.mesh.bspline`, 993
`sfepy.mesh.geom_tools`, 999
`sfepy.mesh.mesh_generators`, 1001
`sfepy.mesh.mesh_tools`, 1004
`sfepy.mesh.splinebox`, 1006
`sfepy.parallel.evaluate`, 1007
`sfepy.parallel.parallel`, 1008
`sfepy.parallel.plot_parallel_dofs`, 1010
`sfepy.postprocess.plot_cmesh`, 1010
`sfepy.postprocess.plot_dofs`, 1011
`sfepy.postprocess.plot_facets`, 1011
`sfepy.postprocess.plot_quadrature`, 1012
`sfepy.postprocess.probes_vtk`, 1012
`sfepy.postprocess.time_history`, 1014
`sfepy.postprocess.utils_vtk`, 1015
`sfepy.scripts.blockgen`, 1182
`sfepy.scripts.convert_mesh`, 1182
`sfepy.scripts.cylindergen`, 1183
`sfepy.scripts.gen_iga_patch`, 1183
`sfepy.scripts.gen_mesh`, 1183
`sfepy.scripts.gen_mesh_prev`, 1183
`sfepy.scripts.plot_condition_numbers`, 1184
`sfepy.scripts.plot_logs`, 1184
`sfepy.scripts.plot_mesh`, 1184
`sfepy.scripts.plot_quadratures`, 1184
`sfepy.scripts.plot_times`, 1184
`sfepy.scripts.probe`, 1184
`sfepy.scripts.resview`, 1186
`sfepy.scripts.run_tests`, 1187
`sfepy.scripts.simple`, 1187
`sfepy.solvers.auto_fallback`, 1016
`sfepy.solvers.eigen`, 1017
`sfepy.solvers.ls`, 1020
`sfepy.solvers.nls`, 1027
`sfepy.solvers.optimize`, 1031
`sfepy.solvers.oseen`, 1033
`sfepy.solvers.qeigen`, 1034
`sfepy.solvers.semismooth_newton`, 1035
`sfepy.solvers.solvers`, 1036
`sfepy.solvers.ts`, 1038
`sfepy.solvers.ts_controllers`, 1039
`sfepy.solvers.ts_dg_solvers`, 932
`sfepy.solvers.ts_solvers`, 1042
`sfepy.terms.extmods.terms`, 1179
`sfepy.terms.terms`, 1050
`sfepy.terms.terms_adj_navier_stokes`, 1055
`sfepy.terms.terms_basic`, 1065
`sfepy.terms.terms_biot`, 1069
`sfepy.terms.terms_compat`, 1072
`sfepy.terms.terms_constraints`, 1075
`sfepy.terms.terms_contact`, 1077
`sfepy.terms.terms_dg`, 1079
`sfepy.terms.terms_diffusion`, 1084
`sfepy.terms.terms_dot`, 1090
`sfepy.terms.terms_elastic`, 1096
`sfepy.terms.terms_electric`, 1109
`sfepy.terms.terms_fibres`, 1110
`sfepy.terms.terms_flexo`, 1111
`sfepy.terms.terms_hyperelastic_base`, 1114
`sfepy.terms.terms_hyperelastic_tl`, 1115
`sfepy.terms.terms_hyperelastic_ul`, 1124
`sfepy.terms.terms_jax`, 1128
`sfepy.terms.terms_mass`, 1135

`sfepy.terms.terms_membrane`, 1136
`sfepy.terms.terms_multilinear`, 1137
`sfepy.terms.terms_navier_stokes`, 1152
`sfepy.terms.terms_piezo`, 1160
`sfepy.terms.terms_point`, 1163
`sfepy.terms.terms_sensitivity`, 1164
`sfepy.terms.terms_shells`, 1170
`sfepy.terms.terms_surface`, 1172
`sfepy.terms.terms_th`, 1177
`sfepy.terms.terms_volume`, 1177
`sfepy.terms.utils`, 1178
`sfepy.tests.conftest`, 1188
`sfepy.tests.test_assembling`, 1188
`sfepy.tests.test_base`, 1189
`sfepy.tests.test_cmesh`, 1189
`sfepy.tests.test_conditions`, 1189
`sfepy.tests.test_declarative_examples`, 1189
`sfepy.tests.test_dg_field`, 1190
`sfepy.tests.test_dg_terms_calls`, 1190
`sfepy.tests.test_domain`, 1191
`sfepy.tests.test_ed_solvers`, 1192
`sfepy.tests.test_eigenvalue_solvers`, 1192
`sfepy.tests.test_elasticity_small_strain`, 1192
`sfepy.tests.test_fem`, 1192
`sfepy.tests.test_functions`, 1193
`sfepy.tests.test_high_level`, 1193
`sfepy.tests.test_homogenization_engine`, 1193
`sfepy.tests.test_homogenization_perfusion`, 1193
`sfepy.tests.test_hyperelastic_tlul`, 1194
`sfepy.tests.test_io`, 1194
`sfepy.tests.test_laplace_unit_disk`, 1194
`sfepy.tests.test_laplace_unit_square`, 1194
`sfepy.tests.test_lcbcs`, 1194
`sfepy.tests.test_linalg`, 1194
`sfepy.tests.test_linear_solvers`, 1195
`sfepy.tests.test_linearization`, 1195
`sfepy.tests.test_log`, 1195
`sfepy.tests.test_matcoefs`, 1195
`sfepy.tests.test_mesh_expand`, 1195
`sfepy.tests.test_mesh_generators`, 1196
`sfepy.tests.test_mesh_interp`, 1196
`sfepy.tests.test_mesh_smoothing`, 1196
`sfepy.tests.test_meshio`, 1196
`sfepy.tests.test_msm_laplace`, 1197
`sfepy.tests.test_msm_symbolic`, 1197
`sfepy.tests.test_normals`, 1197
`sfepy.tests.test_parsing`, 1197
`sfepy.tests.test_poly_spaces`, 1197
`sfepy.tests.test_projections`, 1198
`sfepy.tests.test_quadratures`, 1198
`sfepy.tests.test_ref_coors`, 1198
`sfepy.tests.test_refine_hanging`, 1199
`sfepy.tests.test_regions`, 1199
`sfepy.tests.test_semismooth_newton`, 1199
`sfepy.tests.test_sparse`, 1199
`sfepy.tests.test_splinebox`, 1199
`sfepy.tests.test_tensors`, 1200
`sfepy.tests.test_term_call_modes`, 1200
`sfepy.tests.test_term_consistency`, 1200
`sfepy.tests.test_term_sensitivity`, 1200
`sfepy.tests.test_units`, 1201
`sfepy.tests.test_volume`, 1201
`sfepy.version`, 778
`show_authors`, 1206
`show_terms_use`, 1207
`sync_module_docs`, 1207

Symbols

- `__annotations__` (*sfepy.solvers.nls.Newton* attribute), 1029
- `__annotations__` (*sfepy.solvers.nls.PETScNonlinearSolver* attribute), 1029
- `__annotations__` (*sfepy.solvers.nls.ScipyRoot* attribute), 1030
- `__call__()` (*sfepy.solvers.nls.Newton* method), 1029
- `__call__()` (*sfepy.solvers.nls.PETScNonlinearSolver* method), 1030
- `__call__()` (*sfepy.solvers.nls.ScipyRoot* method), 1030
- `__init__()` (*sfepy.discrete.materials.Material* method), 824
- `__init__()` (*sfepy.solvers.nls.Newton* method), 1029
- `__init__()` (*sfepy.solvers.nls.PETScNonlinearSolver* method), 1030
- `__init__()` (*sfepy.solvers.nls.ScipyRoot* method), 1030
- `__module__` (*sfepy.solvers.nls.Newton* attribute), 1029
- `__module__` (*sfepy.solvers.nls.PETScNonlinearSolver* attribute), 1030
- `__module__` (*sfepy.solvers.nls.ScipyRoot* attribute), 1030
- A**
- `AcousticBandGapsApp` (class in *sfepy.homogenization.band_gaps_app*), 950
- `AcousticMassLiquidTensor` (class in *sfepy.homogenization.coefs_phononic*), 954
- `AcousticMassTensor` (class in *sfepy.homogenization.coefs_phononic*), 954
- `acquire()` (*sfepy.base.multiproc_mpi.RemoteLock* method), 802
- `actBfT()` (in module *sfepy.terms.extmods.terms*), 1179
- `adapt_time_step()` (in module *sfepy.solvers.ts_solvers*), 1049
- `AdaptiveTimeSteppingSolver` (class in *sfepy.solvers.ts_solvers*), 1042
- `add()` (*sfepy.base.timing.Timer* method), 807
- `add_arg_dofs()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_args()` (in module *sfepy.scripts.blockgen*), 1182
- `add_args()` (in module *sfepy.scripts.cylindergen*), 1183
- `add_args()` (in module *sfepy.scripts.gen_iga_patch*), 1183
- `add_bf()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_bfg()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_cell_scalar()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_circle_probe()` (*sfepy.postprocess.probes_vtk.Probe* method), 1012
- `add_eas_dofs()` (in module *sfepy.mechanics.shell10x*), 986
- `add_equation()` (*sfepy.discrete.equations.Equations* method), 811
- `add_eye()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_from_bc()` (*sfepy.discrete.fem.lcbc_operators.LCBCOperators* method), 893
- `add_group()` (*sfepy.base.log.Log* method), 798
- `add_line_probe()` (*sfepy.postprocess.probes_vtk.Probe* method), 1012
- `add_mat_id_to_grid()` (in module *sfepy.scripts.resview*), 1187
- `add_material_arg()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_missing()` (*sfepy.base.conf.ProblemConf* method), 790
- `add_points_probe()` (*sfepy.postprocess.probes_vtk.Probe* method), 1013
- `add_psg()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_pvg()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_qp_scalar()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_ray_probe()` (*sfepy.postprocess.probes_vtk.Probe* method), 1013
- `add_state_arg()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `add_strain_rs()` (in module *sfepy.homogenization.recovery*), 964
- `add_stress_p()` (in module

- sfepy.homogenization.recovery*), 964
- `add_virtual_arg()` (*sfepy.terms.terms_multilinear.Expression* attribute), 1148
- `addline()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addlines()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addphysicalsurface()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addphysicalvolume()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addpoint()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addpoints()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addsurface()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addsurfaces()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addvolume()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `addvolumes()` (*sfepy.mesh.geom_tools.geometry* method), 999
- `AdjConvect1Term` (class in *sfepy.terms.terms_adj_navier_stokes*), 1055
- `AdjConvect2Term` (class in *sfepy.terms.terms_adj_navier_stokes*), 1055
- `AdjDivGradTerm` (class in *sfepy.terms.terms_adj_navier_stokes*), 1056
- `advance()` (*sfepy.discrete.equations.Equations* method), 812
- `advance()` (*sfepy.discrete.problem.Problem* method), 833
- `advance()` (*sfepy.discrete.variables.Variable* method), 852
- `advance()` (*sfepy.discrete.variables.Variables* method), 853
- `advance()` (*sfepy.solvers.ts.TimeStepper* method), 1038
- `advance()` (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
- `advance()` (*sfepy.terms.terms.Term* method), 1050
- `advance_eth_data()` (*sfepy.terms.terms_th.ETHTerm* method), 1177
- `AdvectDivFreeTerm` (class in *sfepy.terms.terms_diffusion*), 1084
- `AdvectionDGFluxTerm` (class in *sfepy.terms.terms_dg*), 1079
- `alf` (*sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm* attribute), 1082
- `all_bfs` (*sfepy.discrete.fem.poly_spaces.SerendipityTensorProduct* attribute), 909
- `alpha` (*sfepy.terms.terms_dg.AdvectionDGFluxTerm* attribute), 1080
- `animate_1D_dg_sol()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 914
- `animate_1D_DG_sol()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 914
- `ANSYSCDBMeshIO` (class in *sfepy.discrete.fem.meshio*), 900
- `any_from_args()` (*sfepy.discrete.common.poly_spaces.PolySpace* static method), 872
- `any_from_conf()` (*sfepy.homogenization.coefs_base.MinAppBase* static method), 954
- `any_from_conf()` (*sfepy.solvers.solvers.Solver* static method), 1037
- `any_from_filename()` (*sfepy.discrete.fem.meshio.MeshIO* static method), 904
- `append()` (*sfepy.base.base.Container* method), 780
- `append()` (*sfepy.discrete.fem.history.History* method), 892
- `append()` (*sfepy.discrete.fem.lcbc_operators.LCBCOperators* method), 893
- `append()` (*sfepy.terms.terms.Terms* method), 1054
- `append_all()` (in module *sfepy.terms.terms_multilinear*), 1151
- `append_bubbles()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_declarations()` (in module *gen_lobatto1d_c*), 1205
- `append_edges()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_faces()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_lists()` (in module *gen_lobatto1d_c*), 1205
- `append_polys()` (in module *gen_lobatto1d_c*), 1205
- `append_raw()` (*sfepy.discrete.common.dof_info.DofInfo* method), 857
- `append_tp_bubbles()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_tp_edges()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_tp_faces()` (*sfepy.discrete.fem.poly_spaces.LagrangeNodes* static method), 908
- `append_variable()` (*sfepy.discrete.common.dof_info.DofInfo* method), 858
- `Application` (class in *sfepy.applications.application*), 778
- `AppliedLoadTensor` (class in *sfepy.homogenization.coefs_phononic*), 955
- `apply()` (*sfepy.tests.test_linear_solvers.DiagPC* method), 1195
- `ApplyPolySands()` (*sfepy.base.log_plotter.LogPlotter* method), 800
- `apply_ebc()` (*sfepy.discrete.equations.Equations*

- method), 812
- apply_ebc() (sfepy.discrete.variables.DGFieldVariable method), 847
- apply_ebc() (sfepy.discrete.variables.FieldVariable method), 848
- apply_ebc() (sfepy.discrete.variables.Variables method), 853
- apply_ebc_to_matrix() (in module sfepy.discrete.evaluate), 818
- apply_ic() (sfepy.discrete.equations.Equations method), 812
- apply_ic() (sfepy.discrete.variables.FieldVariable method), 848
- apply_ic() (sfepy.discrete.variables.Variables method), 853
- apply_layout() (sfepy.terms.terms_multilinear.ExpressionBuilder attribute), 1148
- apply_line_search_bt() (in module sfepy.solvers.nls), 1030
- apply_to_sequence() (in module sfepy.linalg.utils), 975
- apply_unit_multipliers() (in module sfepy.mechanics.units), 992
- apply_units_to_pars() (in module sfepy.mechanics.units), 992
- apply_view_options() (in module gen_gallery), 1204
- approximate() (sfepy.mesh.bspline.BSpline method), 993
- approximate() (sfepy.mesh.bspline.BSplineSurf method), 996
- approximate_exponential() (in module sfepy.homogenization.convolution), 960
- approximation_example() (in module sfepy.mesh.bspline), 998
- are_close() (in module sfepy.solvers.oseen), 1034
- are_disjoint() (in module sfepy.discrete.common.region), 876
- arg_geometry_types (sfepy.terms.terms.Term attribute), 1050
- arg_geometry_types (sfepy.terms.terms_hyperelastic_tl.BulkPressureTerm attribute), 1116
- arg_geometry_types (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm attribute), 1123
- arg_geometry_types (sfepy.terms.terms_hyperelastic_ul.BulkPressureTerm attribute), 1124
- arg_geometry_types (sfepy.terms.terms_hyperelastic_ul.CompressibilityTerm attribute), 1125
- arg_geometry_types (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm attribute), 1128
- arg_shapes (sfepy.terms.terms.Term attribute), 1050
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.AdjConvectTerm attribute), 1055
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.AdjConvectTerm attribute), 1056
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm attribute), 1056
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm attribute), 1056
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm attribute), 1057
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm attribute), 1057
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDConvectTerm attribute), 1058
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm attribute), 1059
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDDivTerm attribute), 1059
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDDotTerm attribute), 1060
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm attribute), 1060
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm attribute), 1061
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDPSPGPSStabilizationTerm attribute), 1062
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm attribute), 1062
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SUPGCAdjStabilizationTerm attribute), 1063
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SUPGPAAdj1StabilizationTerm attribute), 1064
- arg_shapes (sfepy.terms.terms_adj_navier_stokes.SUPGPAAdj2StabilizationTerm attribute), 1064
- arg_shapes (sfepy.terms.terms_basic.IntegrateMatTerm attribute), 1065
- arg_shapes (sfepy.terms.terms_basic.IntegrateOperatorTerm attribute), 1065
- arg_shapes (sfepy.terms.terms_basic.IntegrateTerm attribute), 1066
- arg_shapes (sfepy.terms.terms_basic.SumNodalValuesTerm attribute), 1067
- arg_shapes (sfepy.terms.terms_basic.SurfaceMomentTerm attribute), 1067
- arg_shapes (sfepy.terms.terms_basic.VolumeSurfaceTerm attribute), 1068
- arg_shapes (sfepy.terms.terms_basic.VolumeTerm attribute), 1068
- arg_shapes (sfepy.terms.terms_basic.ZeroTerm attribute), 1068
- arg_shapes (sfepy.terms.terms_biot.BiotETHTerm attribute), 1069
- arg_shapes (sfepy.terms.terms_biot.BiotETHTerm attribute), 1069
- arg_shapes (sfepy.terms.terms_biot.BiotStressTerm attribute), 1070
- arg_shapes (sfepy.terms.terms_biot.BiotTerm attribute), 1070
- arg_shapes (sfepy.terms.terms_biot.BiotTHTerm attribute), 1071

`arg_shapes (sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1075`
`arg_shapes (sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1076`
`arg_shapes (sfepy.terms.terms_contact.ContactIPTerm attribute), 1077`
`arg_shapes (sfepy.terms.terms_contact.ContactTerm attribute), 1078`
`arg_shapes (sfepy.terms.terms_dg.AdvectionDGFluxTerm attribute), 1080`
`arg_shapes (sfepy.terms.terms_dg.DiffusionDGFluxTerm attribute), 1081`
`arg_shapes (sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm attribute), 1081`
`arg_shapes (sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm attribute), 1083`
`arg_shapes (sfepy.terms.terms_dg.NonlinearScalarDotGradTerm attribute), 1083`
`arg_shapes (sfepy.terms.terms_diffusion.AdvectDivFreeTerm attribute), 1084`
`arg_shapes (sfepy.terms.terms_diffusion.ConvectVGradSTerm attribute), 1085`
`arg_shapes (sfepy.terms.terms_diffusion.DiffusionCouplingTerm attribute), 1085`
`arg_shapes (sfepy.terms.terms_diffusion.DiffusionRTerm attribute), 1086`
`arg_shapes (sfepy.terms.terms_diffusion.DiffusionTerm attribute), 1086`
`arg_shapes (sfepy.terms.terms_diffusion.DiffusionVelocityTerm attribute), 1087`
`arg_shapes (sfepy.terms.terms_diffusion.LaplaceTerm attribute), 1088`
`arg_shapes (sfepy.terms.terms_diffusion.NonlinearDiffusionTerm attribute), 1088`
`arg_shapes (sfepy.terms.terms_diffusion.SDDiffusionTerm attribute), 1089`
`arg_shapes (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm attribute), 1089`
`arg_shapes (sfepy.terms.terms_diffusion.SurfaceFluxTerm attribute), 1090`
`arg_shapes (sfepy.terms.terms_dot.BCNewtonTerm attribute), 1091`
`arg_shapes (sfepy.terms.terms_dot.DotSPProductVolumeOperatorTerm attribute), 1092`
`arg_shapes (sfepy.terms.terms_dot.DotSPProductVolumeOperatorTerm attribute), 1093`
`arg_shapes (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm attribute), 1094`
`arg_shapes (sfepy.terms.terms_dot.ScalarDotMGradScalarTerm attribute), 1094`
`arg_shapes (sfepy.terms.terms_dot.VectorDotGradScalarTerm attribute), 1095`
`arg_shapes (sfepy.terms.terms_dot.VectorDotScalarTerm attribute), 1096`
`arg_shapes (sfepy.terms.terms_elastic.CauchyStrainTerm attribute), 1097`
`arg_shapes (sfepy.terms.terms_elastic.CauchyStressETHTerm attribute), 1097`
`arg_shapes (sfepy.terms.terms_elastic.CauchyStressTerm attribute), 1098`
`arg_shapes (sfepy.terms.terms_elastic.CauchyStressTHTerm attribute), 1098`
`arg_shapes (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm attribute), 1099`
`arg_shapes (sfepy.terms.terms_elastic.ElasticWaveTerm attribute), 1100`
`arg_shapes (sfepy.terms.terms_elastic.LinearDRotSpringTerm attribute), 1101`
`arg_shapes (sfepy.terms.terms_elastic.LinearDSpringTerm attribute), 1101`
`arg_shapes (sfepy.terms.terms_elastic.LinearElasticETHTerm attribute), 1102`
`arg_shapes (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm attribute), 1103`
`arg_shapes (sfepy.terms.terms_elastic.LinearElasticTerm attribute), 1104`
`arg_shapes (sfepy.terms.terms_elastic.LinearElasticTHTerm attribute), 1103`
`arg_shapes (sfepy.terms.terms_elastic.LinearPrestressTerm attribute), 1105`
`arg_shapes (sfepy.terms.terms_elastic.LinearSpringTerm attribute), 1105`
`arg_shapes (sfepy.terms.terms_elastic.LinearStrainFiberTerm attribute), 1106`
`arg_shapes (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm attribute), 1106`
`arg_shapes (sfepy.terms.terms_elastic.LinearTrussTerm attribute), 1107`
`arg_shapes (sfepy.terms.terms_elastic.NonsymElasticTerm attribute), 1108`
`arg_shapes (sfepy.terms.terms_elastic.SDLinearElasticTerm attribute), 1109`
`arg_shapes (sfepy.terms.terms_electric.ElectricSourceTerm attribute), 1109`
`arg_shapes (sfepy.terms.terms_fibres.FibresActiveTLTerm attribute), 1110`
`arg_shapes (sfepy.terms.terms_fibres.FibresExponentialTLTerm attribute), 1111`
`arg_shapes (sfepy.terms.terms_flexo.MixedFlexoCouplingTerm attribute), 1112`
`arg_shapes (sfepy.terms.terms_flexo.MixedFlexoTerm attribute), 1113`
`arg_shapes (sfepy.terms.terms_flexo.MixedStrainGradElasticTerm attribute), 1113`
`arg_shapes (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm attribute), 1114`
`arg_shapes (sfepy.terms.terms_hyperelastic_base.HyperElasticBase attribute), 1115`

`arg_shapes (sfepy.terms.terms_hyperelastic_tl.BulkPressureTerm attribute), 1116`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm attribute), 1117`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm attribute), 1118`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm attribute), 1121`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm attribute), 1121`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm attribute), 1122`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTensionTLTerm attribute), 1122`
`arg_shapes (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm attribute), 1123`
`arg_shapes (sfepy.terms.terms_hyperelastic_ul.BulkPressureTerm attribute), 1124`
`arg_shapes (sfepy.terms.terms_hyperelastic_ul.CompressibilityTerm attribute), 1125`
`arg_shapes (sfepy.terms.terms_hyperelastic_ul.HyperelasticityTerm attribute), 1126`
`arg_shapes (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm attribute), 1128`
`arg_shapes (sfepy.terms.terms_jax.LinearElasticLADTerm attribute), 1128`
`arg_shapes (sfepy.terms.terms_jax.LinearElasticYPADTerm attribute), 1129`
`arg_shapes (sfepy.terms.terms_jax.MassADTerm attribute), 1130`
`arg_shapes (sfepy.terms.terms_jax.NeoHookeanTLADTerm attribute), 1130`
`arg_shapes (sfepy.terms.terms_jax.OgdenTLADTerm attribute), 1131`
`arg_shapes (sfepy.terms.terms_mass.MassTerm attribute), 1135`
`arg_shapes (sfepy.terms.terms_membrane.TLMembraneTerm attribute), 1136`
`arg_shapes (sfepy.terms.terms_multilinear.ECauchyStressTerm attribute), 1137`
`arg_shapes (sfepy.terms.terms_multilinear.EConvectTerm attribute), 1137`
`arg_shapes (sfepy.terms.terms_multilinear.EDiffusionTerm attribute), 1138`
`arg_shapes (sfepy.terms.terms_multilinear.EDivGradTerm attribute), 1138`
`arg_shapes (sfepy.terms.terms_multilinear.EDivTerm attribute), 1139`
`arg_shapes (sfepy.terms.terms_multilinear.EDotTerm attribute), 1140`
`arg_shapes (sfepy.terms.terms_multilinear.EGradTerm attribute), 1140`
`arg_shapes (sfepy.terms.terms_multilinear.EIntegrateOperatorTerm attribute), 1141`
`arg_shapes (sfepy.terms.terms_multilinear.ELaplaceTerm attribute), 1141`
`arg_shapes (sfepy.terms.terms_multilinear.ELinearConvectTerm attribute), 1142`
`arg_shapes (sfepy.terms.terms_multilinear.ELinearElasticTerm attribute), 1142`
`arg_shapes (sfepy.terms.terms_multilinear.ELinearTractionTerm attribute), 1143`
`arg_shapes (sfepy.terms.terms_multilinear.ENonPenetrationPenaltyTerm attribute), 1144`
`arg_shapes (sfepy.terms.terms_multilinear.ENonSymElasticTerm attribute), 1144`
`arg_shapes (sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm attribute), 1145`
`arg_shapes (sfepy.terms.terms_multilinear.EStokesTerm attribute), 1146`
`arg_shapes (sfepy.terms.terms_multilinear.StokesTractionTerm attribute), 1149`
`arg_shapes (sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm attribute), 1149`
`arg_shapes (sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm attribute), 1150`
`arg_shapes (sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm attribute), 1151`
`arg_shapes (sfepy.terms.terms_navier_stokes.ConvectTerm attribute), 1152`
`arg_shapes (sfepy.terms.terms_navier_stokes.DivGradTerm attribute), 1152`
`arg_shapes (sfepy.terms.terms_navier_stokes.DivOperatorTerm attribute), 1153`
`arg_shapes (sfepy.terms.terms_navier_stokes.DivTerm attribute), 1154`
`arg_shapes (sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm attribute), 1154`
`arg_shapes (sfepy.terms.terms_navier_stokes.GradTerm attribute), 1155`
`arg_shapes (sfepy.terms.terms_navier_stokes.LinearConvect2Term attribute), 1155`
`arg_shapes (sfepy.terms.terms_navier_stokes.LinearConvectTerm attribute), 1156`
`arg_shapes (sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm attribute), 1156`
`arg_shapes (sfepy.terms.terms_navier_stokes.StokesTerm attribute), 1159`
`arg_shapes (sfepy.terms.terms_navier_stokes.StokesWaveDivTerm attribute), 1159`
`arg_shapes (sfepy.terms.terms_navier_stokes.StokesWaveTerm attribute), 1160`
`arg_shapes (sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm attribute), 1157`
`arg_shapes (sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm attribute), 1158`
`arg_shapes (sfepy.terms.terms_piezo.PiezoCouplingTerm attribute), 1161`

`arg_shapes` (`sfepy.terms.terms_piezo.PiezoStrainTerm` attribute), 1056

`arg_shapes` (`sfepy.terms.terms_piezo.PiezoStressTerm` attribute), 1056

`arg_shapes` (`sfepy.terms.terms_piezo.SDPiezoCouplingTerm` attribute), 1057

`arg_shapes` (`sfepy.terms.terms_point.ConcentratedPointLoadTerm` attribute), 1057

`arg_shapes` (`sfepy.terms.terms_point.LinearPointSpringTerm` attribute), 1058

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDDiffusionTerm` attribute), 1059

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDDivGradTerm` attribute), 1059

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDDotTerm` attribute), 1060

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDLinearElasticTerm` attribute), 1060

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDLinearTractionTerm` attribute), 1061

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm` attribute), 1062

`arg_shapes` (`sfepy.terms.terms_sensitivity.ESDStokesTerm` attribute), 1062

`arg_shapes` (`sfepy.terms.terms_shells.Shell10XTerm` attribute), 1063

`arg_shapes` (`sfepy.terms.terms_surface.ContactPlaneTerm` attribute), 1064

`arg_shapes` (`sfepy.terms.terms_surface.ContactSphereTerm` attribute), 1064

`arg_shapes` (`sfepy.terms.terms_surface.LinearTractionTerm` attribute), 1065

`arg_shapes` (`sfepy.terms.terms_surface.SDLinearTractionTerm` attribute), 1066

`arg_shapes` (`sfepy.terms.terms_surface.SDSurfaceIntegrateTerm` attribute), 1066

`arg_shapes` (`sfepy.terms.terms_surface.SurfaceNormalDotTerm` attribute), 1067

`arg_shapes` (`sfepy.terms.terms_surface.SurfaceJumpTerm` attribute), 1067

`arg_shapes` (`sfepy.terms.terms_volume.LinearVolumeForceTerm` attribute), 1068

`arg_shapes` (`sfepy.terms.terms_volume.NonlinearVolumeForceTerm` attribute), 1068

`arg_shapes_dict` (`sfepy.terms.terms_dot.BCNewtonTerm` attribute), 1069

`arg_shapes_dict` (`sfepy.terms.terms_dot.DotProductTerm` attribute), 1069

`arg_types` (`sfepy.terms.terms.Term` attribute), 1070

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.AdjConvectionTerm` attribute), 1072

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.AdjConvectionTerm` attribute), 1072

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm` attribute), 1071

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm` attribute), 1056

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressDiffTerm` attribute), 1057

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm` attribute), 1057

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDConvectTerm` attribute), 1058

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm` attribute), 1059

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDDivTerm` attribute), 1059

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDDotTerm` attribute), 1060

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm` attribute), 1060

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm` attribute), 1061

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDPSPGPStabilizationTerm` attribute), 1062

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm` attribute), 1062

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SUPGCAdjStabilizationTerm` attribute), 1063

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SUPGPAdj1StabilizationTerm` attribute), 1064

`arg_types` (`sfepy.terms.terms_adj_navier_stokes.SUPGPAdj2StabilizationTerm` attribute), 1064

`arg_types` (`sfepy.terms.terms_basic.IntegrateMatTerm` attribute), 1065

`arg_types` (`sfepy.terms.terms_basic.IntegrateOperatorTerm` attribute), 1066

`arg_types` (`sfepy.terms.terms_basic.IntegrateTerm` attribute), 1066

`arg_types` (`sfepy.terms.terms_basic.SumNodalValuesTerm` attribute), 1067

`arg_types` (`sfepy.terms.terms_basic.SurfaceMomentTerm` attribute), 1067

`arg_types` (`sfepy.terms.terms_basic.VolumeSurfaceTerm` attribute), 1068

`arg_types` (`sfepy.terms.terms_basic.VolumeTerm` attribute), 1068

`arg_types` (`sfepy.terms.terms_basic.ZeroTerm` attribute), 1069

`arg_types` (`sfepy.terms.terms_biot.BiotETHTerm` attribute), 1070

`arg_types` (`sfepy.terms.terms_biot.BiotStressTerm` attribute), 1070

`arg_types` (`sfepy.terms.terms_biot.BiotTerm` attribute), 1072

`arg_types` (`sfepy.terms.terms_biot.BiotTHTerm` attribute), 1071

`arg_types` (`sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm` attribute), 1071

attribute), 1075
 arg_types (sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1076
 arg_types (sfepy.terms.terms_contact.ContactIPCTerm attribute), 1077
 arg_types (sfepy.terms.terms_contact.ContactTerm attribute), 1078
 arg_types (sfepy.terms.terms_dg.AdvectionDGFluxTerm attribute), 1080
 arg_types (sfepy.terms.terms_dg.DiffusionDGFluxTerm attribute), 1081
 arg_types (sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm attribute), 1081
 arg_types (sfepy.terms.terms_dg.NonlinearHyperbolicDGEqTerm attribute), 1083
 arg_types (sfepy.terms.terms_dg.NonlinearScalarDotGradientTerm attribute), 1083
 arg_types (sfepy.terms.terms_diffusion.AdvectDivFreeTerm attribute), 1084
 arg_types (sfepy.terms.terms_diffusion.ConvectVGradSTerm attribute), 1085
 arg_types (sfepy.terms.terms_diffusion.DiffusionCouplingTerm attribute), 1085
 arg_types (sfepy.terms.terms_diffusion.DiffusionRTerm attribute), 1086
 arg_types (sfepy.terms.terms_diffusion.DiffusionTerm attribute), 1086
 arg_types (sfepy.terms.terms_diffusion.DiffusionVelocityTerm attribute), 1087
 arg_types (sfepy.terms.terms_diffusion.LaplaceTerm attribute), 1088
 arg_types (sfepy.terms.terms_diffusion.NonlinearDiffusionTerm attribute), 1088
 arg_types (sfepy.terms.terms_diffusion.SDDiffusionTerm attribute), 1089
 arg_types (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm attribute), 1089
 arg_types (sfepy.terms.terms_diffusion.SurfaceFluxTerm attribute), 1090
 arg_types (sfepy.terms.terms_dot.BCNewtonTerm attribute), 1091
 arg_types (sfepy.terms.terms_dot.DotProductTerm attribute), 1092
 arg_types (sfepy.terms.terms_dot.DotSProductVolumeOperatorTerm attribute), 1093
 arg_types (sfepy.terms.terms_dot.DotSProductVolumeOperatorTerm attribute), 1093
 arg_types (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm attribute), 1094
 arg_types (sfepy.terms.terms_dot.ScalarDotMGradScalarTerm attribute), 1094
 arg_types (sfepy.terms.terms_dot.VectorDotGradScalarTerm attribute), 1095
 arg_types (sfepy.terms.terms_dot.VectorDotScalarTerm attribute), 1096
 arg_types (sfepy.terms.terms_elastic.CauchyStrainTerm attribute), 1097
 arg_types (sfepy.terms.terms_elastic.CauchyStressETHTerm attribute), 1097
 arg_types (sfepy.terms.terms_elastic.CauchyStressTerm attribute), 1098
 arg_types (sfepy.terms.terms_elastic.CauchyStressTHTerm attribute), 1098
 arg_types (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm attribute), 1099
 arg_types (sfepy.terms.terms_elastic.ElasticWaveTerm attribute), 1100
 arg_types (sfepy.terms.terms_elastic.LinearDRotSpringTerm attribute), 1101
 arg_types (sfepy.terms.terms_elastic.LinearDSpringTerm attribute), 1101
 arg_types (sfepy.terms.terms_elastic.LinearElasticETHTerm attribute), 1102
 arg_types (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm attribute), 1103
 arg_types (sfepy.terms.terms_elastic.LinearElasticTerm attribute), 1104
 arg_types (sfepy.terms.terms_elastic.LinearElasticTHTerm attribute), 1103
 arg_types (sfepy.terms.terms_elastic.LinearPrestressTerm attribute), 1105
 arg_types (sfepy.terms.terms_elastic.LinearSpringTerm attribute), 1105
 arg_types (sfepy.terms.terms_elastic.LinearStrainFiberTerm attribute), 1106
 arg_types (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm attribute), 1106
 arg_types (sfepy.terms.terms_elastic.LinearTrussTerm attribute), 1107
 arg_types (sfepy.terms.terms_elastic.NonsymElasticTerm attribute), 1108
 arg_types (sfepy.terms.terms_elastic.SDLinearElasticTerm attribute), 1109
 arg_types (sfepy.terms.terms_electric.ElectricSourceTerm attribute), 1109
 arg_types (sfepy.terms.terms_fibres.FibresActiveTLTerm attribute), 1110
 arg_types (sfepy.terms.terms_fibres.FibresExponentialTLTerm attribute), 1111
 arg_types (sfepy.terms.terms_flexo.MixedFlexoCouplingTerm attribute), 1112
 arg_types (sfepy.terms.terms_flexo.MixedFlexoTerm attribute), 1113
 arg_types (sfepy.terms.terms_flexo.MixedStrainGradElasticTerm attribute), 1113
 arg_types (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm attribute), 1114
 arg_types (sfepy.terms.terms_hyperelastic_base.HyperElasticBase

attribute), 1115

`arg_types(sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm, attribute), 1117`

`arg_types(sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm, attribute), 1117`

`arg_types(sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm, attribute), 1121`

`arg_types(sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm, attribute), 1122`

`arg_types(sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm, attribute), 1123`

`arg_types(sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm, attribute), 1123`

`arg_types(sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm, attribute), 1124`

`arg_types(sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm, attribute), 1125`

`arg_types(sfepy.terms.terms_hyperelastic_ul.HyperelasticityULTerm, attribute), 1126`

`arg_types(sfepy.terms.terms_hyperelastic_ul.VolumeULTerm, attribute), 1128`

`arg_types(sfepy.terms.terms_jax.LinearElasticLADTerm, attribute), 1128`

`arg_types(sfepy.terms.terms_jax.LinearElasticYPADTerm, attribute), 1129`

`arg_types(sfepy.terms.terms_jax.MassADTerm, attribute), 1130`

`arg_types(sfepy.terms.terms_jax.NeoHookeanTLADTerm, attribute), 1131`

`arg_types(sfepy.terms.terms_jax.OgdenTLADTerm, attribute), 1131`

`arg_types(sfepy.terms.terms_mass.MassTerm, attribute), 1135`

`arg_types(sfepy.terms.terms_membrane.TLMembraneTerm, attribute), 1136`

`arg_types(sfepy.terms.terms_multilinear.ECauchyStressTerm, attribute), 1137`

`arg_types(sfepy.terms.terms_multilinear.EConvectTerm, attribute), 1137`

`arg_types(sfepy.terms.terms_multilinear.EDiffusionTerm, attribute), 1138`

`arg_types(sfepy.terms.terms_multilinear.EDivGradTerm, attribute), 1139`

`arg_types(sfepy.terms.terms_multilinear.EDivTerm, attribute), 1139`

`arg_types(sfepy.terms.terms_multilinear.EDotTerm, attribute), 1140`

`arg_types(sfepy.terms.terms_multilinear.EGradTerm, attribute), 1140`

`arg_types(sfepy.terms.terms_multilinear.EIntegrateOperatorTerm, attribute), 1141`

`arg_types(sfepy.terms.terms_multilinear.ELaplaceTerm, attribute), 1141`

`arg_types(sfepy.terms.terms_multilinear.ELinearConvectTerm, attribute), 1142`

`arg_types(sfepy.terms.terms_multilinear.ELinearElasticTerm, attribute), 1143`

`arg_types(sfepy.terms.terms_multilinear.ELinearTractionTerm, attribute), 1143`

`arg_types(sfepy.terms.terms_multilinear.ENonPenetrationPenaltyTerm, attribute), 1144`

`arg_types(sfepy.terms.terms_multilinear.ENonSymElasticTerm, attribute), 1144`

`arg_types(sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm, attribute), 1145`

`arg_types(sfepy.terms.terms_multilinear.EStokesTerm, attribute), 1146`

`arg_types(sfepy.terms.terms_multilinear.StokesTractionTerm, attribute), 1149`

`arg_types(sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm, attribute), 1149`

`arg_types(sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm, attribute), 1150`

`arg_types(sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm, attribute), 1151`

`arg_types(sfepy.terms.terms_navier_stokes.ConvectTerm, attribute), 1152`

`arg_types(sfepy.terms.terms_navier_stokes.DivGradTerm, attribute), 1152`

`arg_types(sfepy.terms.terms_navier_stokes.DivOperatorTerm, attribute), 1153`

`arg_types(sfepy.terms.terms_navier_stokes.DivTerm, attribute), 1154`

`arg_types(sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm, attribute), 1154`

`arg_types(sfepy.terms.terms_navier_stokes.GradTerm, attribute), 1155`

`arg_types(sfepy.terms.terms_navier_stokes.LinearConvect2Term, attribute), 1155`

`arg_types(sfepy.terms.terms_navier_stokes.LinearConvectTerm, attribute), 1156`

`arg_types(sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm, attribute), 1156`

`arg_types(sfepy.terms.terms_navier_stokes.StokesTerm, attribute), 1159`

`arg_types(sfepy.terms.terms_navier_stokes.StokesWaveDivTerm, attribute), 1159`

`arg_types(sfepy.terms.terms_navier_stokes.StokesWaveTerm, attribute), 1160`

`arg_types(sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm, attribute), 1157`

`arg_types(sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm, attribute), 1158`

`arg_types(sfepy.terms.terms_piezo.PiezoCouplingTerm, attribute), 1161`

`arg_types(sfepy.terms.terms_piezo.PiezoStressTerm, attribute), 1162`

`arg_types(sfepy.terms.terms_piezo.SDPiezoCouplingTerm, attribute), 1162`

- [attribute](#)), 1163
[arg_types](#) ([sfepy.terms.terms_point.ConcentratedPointLoadTerm](#)
[attribute](#)), 1163
[arg_types](#) ([sfepy.terms.terms_point.LinearPointSpringTerm](#)
[attribute](#)), 1164
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDDiffusionTerm](#)
[attribute](#)), 1164
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDDivGradTerm](#)
[attribute](#)), 1165
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDDotTerm](#)
[attribute](#)), 1166
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDLinearElasticTerm](#)
[attribute](#)), 1166
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDLinearTractionTerm](#)
[attribute](#)), 1167
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm](#)
[attribute](#)), 1168
[arg_types](#) ([sfepy.terms.terms_sensitivity.ESDStokesTerm](#)
[attribute](#)), 1169
[arg_types](#) ([sfepy.terms.terms_shells.Shell10XTerm](#) [attribute](#)), 1171
[arg_types](#) ([sfepy.terms.terms_surface.ContactPlaneTerm](#)
[attribute](#)), 1172
[arg_types](#) ([sfepy.terms.terms_surface.ContactSphereTerm](#)
[attribute](#)), 1174
[arg_types](#) ([sfepy.terms.terms_surface.LinearTractionTerm](#)
[attribute](#)), 1174
[arg_types](#) ([sfepy.terms.terms_surface.SDLinearTractionTerm](#)
[attribute](#)), 1175
[arg_types](#) ([sfepy.terms.terms_surface.SDSurfaceIntegrateTerm](#)
[attribute](#)), 1175
[arg_types](#) ([sfepy.terms.terms_surface.SurfaceNormalDotTerm](#)
[attribute](#)), 1176
[arg_types](#) ([sfepy.terms.terms_surface.SurfaceJumpTerm](#)
[attribute](#)), 1177
[arg_types](#) ([sfepy.terms.terms_volume.LinearVolumeForceTerm](#)
[attribute](#)), 1178
[arg_types](#) ([sfepy.terms.terms_volume.NonlinearVolumeForceTerm](#)
[attribute](#)), 1178
[argsort_rows\(\)](#) (in module [sfepy.linalg.utils](#)), 975
[as_dict\(\)](#) ([sfepy.base.base.Container](#) method), 780
[as_float_or_complex\(\)](#) (in module [sfepy.base.base](#)),
782
[assemble1d\(\)](#) (in module [sfepy.linalg.utils](#)), 975
[assemble_by_blocks\(\)](#) (in module
[sfepy.discrete.evaluate](#)), 818
[assemble_contact_residual_and_stiffness\(\)](#) (in
module [sfepy.mechanics.extmods.ccontres](#)), 993
[assemble_matrix\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assemble_matrix_b\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assemble_matrix_complex\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assemble_matrix_complex_b\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assemble_mtx_to_petsc\(\)](#) (in module
[sfepy.parallel.parallel](#)), 1008
[assemble_rhs_to_petsc\(\)](#) (in module
[sfepy.parallel.parallel](#)), 1008
[assemble_to\(\)](#) ([sfepy.terms.terms.Term](#) method), 1050
[assemble_vector\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assemble_vector_complex\(\)](#) (in module
[sfepy.discrete.common.extmods.assemble](#)),
861
[assert_\(\)](#) (in module [sfepy.base.base](#)), 782
[assert_equal\(\)](#) (in module [sfepy.base.testing](#)), 807
[assign_args\(\)](#) ([sfepy.terms.terms.Term](#) method), 1050
[assign_args\(\)](#) ([sfepy.terms.terms.Terms](#) method), 1054
[assign_standard_hooks\(\)](#) (in module
[sfepy.applications.pde_solver_app](#)), 779
[AutoDirect](#) (class in [sfepy.solvers.auto_fallback](#)), 1016
[AutoFallbackSolver](#) (class in
[sfepy.solvers.auto_fallback](#)), 1017
[AutoIterative](#) (class in [sfepy.solvers.auto_fallback](#)),
1017
[average_qp_to_vertices\(\)](#)
([sfepy.discrete.fem.fields_base.FEField](#)
method), 881
[average_to_vertices\(\)](#)
([sfepy.discrete.fem.fields_nodal.H1DiscontinuousField](#)
method), 889
[average_vertex_var_in_cells\(\)](#) (in module
[sfepy.postprocess.time_history](#)), 1014
[AverageForceOperator](#) (class in
[sfepy.discrete.fem.lcbc_operators](#)), 892
- ## B
- [BandGaps](#) (class in [sfepy.homogenization.coefs_phononic](#)),
955
[barycentric_coors\(\)](#) (in module
[sfepy.linalg.geometry](#)), 970
[base1d](#) ([sfepy.discrete.fem.extmods.bases.CLagrangeContext](#)
[attribute](#)), 878
[basis_function_dg\(\)](#) ([sfepy.mesh.bspline.BSpline](#)
static method), 994
[basis_function_dg0\(\)](#) ([sfepy.mesh.bspline.BSpline](#)
static method), 994
[BatheTS](#) (class in [sfepy.solvers.ts_solvers](#)), 1043
[BCNewtonTerm](#) (class in [sfepy.terms.terms_dot](#)), 1090
[BernsteinSimplexPolySpace](#) (class in
[sfepy.discrete.fem.poly_spaces](#)), 907

BernsteinTensorProductPolySpace (class in `sfepy.discrete.fem.poly_spaces`), 907
 bf (`sfepy.discrete.iga.extmods.igac.CNURBSContext` attribute), 936
 bfg (`sfepy.discrete.iga.extmods.igac.CNURBSContext` attribute), 936
 BiotETHTerm (class in `sfepy.terms.terms_biot`), 1069
 BiotStressTerm (class in `sfepy.terms.terms_biot`), 1070
 BiotTerm (class in `sfepy.terms.terms_biot`), 1071
 BiotTHTerm (class in `sfepy.terms.terms_biot`), 1070
 block_solve() (`sfepy.discrete.problem.Problem` method), 833
 boundary() (in module `sfepy.linalg.sympy_operators`), 974
 BSpline (class in `sfepy.mesh.bspline`), 993
 BSplineSurf (class in `sfepy.mesh.bspline`), 996
 bufBN (`sfepy.discrete.iga.extmods.igac.CNURBSContext` attribute), 936
 build() (`sfepy.terms.terms_multilinear.ExpressionBuilder` method), 1148
 build_expression() (`sfepy.terms.terms_multilinear.ETermBase` method), 1146
 build_helpers module, 1201
 build_interpol_scheme() (`sfepy.discrete.dg.poly_spaces.LegendreTensorProductPolySpace` method), 930
 build_op_pi() (in module `sfepy.homogenization.utils`), 967
 build_orientation_map() (in module `sfepy.discrete.fem.facets`), 879
 build_solver_kwargs() (`sfepy.solvers.solvers.Solver` method), 1037
 bulk_from_lame() (in module `sfepy.mechanics.matcoefs`), 980
 bulk_from_youngpoisson() (in module `sfepy.mechanics.matcoefs`), 980
 BulkActiveTLTerm (class in `sfepy.terms.terms_hyperelastic_tl`), 1115
 BulkPenaltyTLTerm (class in `sfepy.terms.terms_hyperelastic_tl`), 1116
 BulkPenaltyULTerm (class in `sfepy.terms.terms_hyperelastic_ul`), 1124
 BulkPressureTLTerm (class in `sfepy.terms.terms_hyperelastic_tl`), 1116
 BulkPressureULTerm (class in `sfepy.terms.terms_hyperelastic_ul`), 1124
 burg_fun() (`sfepy.tests.test_dg_terms_calls.DGTermTestEnvironment` method), 1190
 burg_fun_d() (`sfepy.tests.test_dg_terms_calls.DGTermTestEnvironment` method), 1190
 C
 cache (`sfepy.discrete.probes.Probe` attribute), 829
 cache_name (`sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTLFamilyData` attribute), 1118
 cache_name (`sfepy.terms.terms_hyperelastic_tl.HyperElasticTLFamilyData` attribute), 1119
 cache_name (`sfepy.terms.terms_hyperelastic_ul.HyperElasticULFamilyData` attribute), 1126
 Cached (class in `sfepy.base.ioutils`), 793
 calculate() (`sfepy.homogenization.engine.HomogenizationWorker` static method), 961
 calculate_req() (`sfepy.homogenization.engine.HomogenizationWorker` static method), 961
 calculate_req_multi() (`sfepy.homogenization.engine.HomogenizationWorkerMulti` static method), 962
 call() (`sfepy.applications.evp_solver_app.EVPSolverApp` method), 778
 call() (`sfepy.applications.pde_solver_app.PDESolverApp` method), 779
 call() (`sfepy.homogenization.band_gaps_app.AcousticBandGapsApp` method), 950
 call() (`sfepy.homogenization.engine.HomogenizationEngine` method), 960
 call() (`sfepy.homogenization.homogen_app.HomogenizationApp` method), 963
 call_basic() (`sfepy.applications.application.Application` method), 778
 call_function() (`sfepy.terms.terms.Term` method), 1050
 call_function() (`sfepy.terms.terms_contact.ContactIPCTerm` method), 1077
 call_function() (`sfepy.terms.terms_contact.ContactTerm` method), 1078
 call_function() (`sfepy.terms.terms_dg.DGTerm` method), 1080
 call_get_fargs() (`sfepy.terms.terms.Term` method), 1050
 call_in_rank_order() (in module `sfepy.parallel.parallel`), 1008
 call_msg (`sfepy.discrete.fem.meshio.MeshIO` attribute), 904
 call_parametrized() (`sfepy.applications.application.Application` method), 778
 can_backend (`sfepy.terms.terms_multilinear.ETermBase` attribute), 1146
 canonize_dof_names() (`sfepy.discrete.conditions.Condition` method), 808
 canonize_dof_names() (`sfepy.discrete.conditions.Conditions` method), 808
 canonize_dof_names() (`sfepy.discrete.conditions.LinearCombinationBC` method), 810

`canonicalize_dof_names()` (*sfepy.discrete.conditions.PeriodicBC* method), 810
`CauchyStrainSTerm` (class in *sfepy.terms.terms_compat*), 1072
`CauchyStrainTerm` (class in *sfepy.terms.terms_elastic*), 1096
`CauchyStressETHTerm` (class in *sfepy.terms.terms_elastic*), 1097
`CauchyStressTerm` (class in *sfepy.terms.terms_elastic*), 1098
`CauchyStressTHTerm` (class in *sfepy.terms.terms_elastic*), 1097
`CBasisContext` (class in *sfepy.discrete.common.extmods.crefcoors*), 864
`CConnectivity` (class in *sfepy.discrete.common.extmods.cmesh*), 861
`cell_groups` (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
`cell_types` (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
`cell_types` (*sfepy.discrete.fem.meshio.MeshioLibIO* attribute), 905
`cells` (*sfepy.discrete.common.region.Region* property), 874
`CentralDifferenceTS` (class in *sfepy.solvers.ts_solvers*), 1044
`ceval_elasticity_l()` (in module *sfepy.terms.terms_jax*), 1132
`ceval_elasticity_yp()` (in module *sfepy.terms.terms_jax*), 1132
`ceval_mass()` (in module *sfepy.terms.terms_jax*), 1132
`ceval_neohook()` (in module *sfepy.terms.terms_jax*), 1132
`ceval_neohook0()` (in module *sfepy.terms.terms_jax*), 1132
`ceval_ogden()` (in module *sfepy.terms.terms_jax*), 1132
`cg_eigs()` (in module *sfepy.linalg.eigen*), 969
`check_args()` (*sfepy.terms.terms.Term* method), 1050
`check_conditions()` (in module *sfepy.base.testing*), 807
`check_finiteness()` (in module *sfepy.terms.utils*), 1178
`check_format_suffix()` (in module *sfepy.discrete.fem.meshio*), 906
`check_fx()` (in module *sfepy.linalg.check_derivatives*), 969
`check_gradient()` (in module *sfepy.solvers.optimize*), 1033
`check_names()` (in module *sfepy.base.base*), 782
`check_shapes()` (*sfepy.terms.terms.Term* method), 1050
`check_tangent_matrix()` (in module *sfepy.solvers.nls*), 1030
`check_vec()` (in module *sfepy.tests.test_conditions*), 1189
`check_vec_size()` (*sfepy.discrete.variables.Variables* method), 853
`check_vfvx()` (in module *sfepy.linalg.check_derivatives*), 969
`CholeskySolver` (class in *sfepy.solvers.ls*), 1020
`ChristoffelAcousticTensor` (class in *sfepy.homogenization.coefs_phononic*), 956
`chunk_arrays()` (in module *sfepy.linalg.utils*), 975
`chunk_micro_tasks()` (*sfepy.homogenization.engine.HomogenizationWorkerMulti* static method), 962
`CircleProbe` (class in *sfepy.discrete.probes*), 828
`CLagrangeContext` (class in *sfepy.discrete.fem.extmods.bases*), 878
`classify_args()` (*sfepy.terms.terms.Term* method), 1050
`Clean` (class in *build_helpers*), 1201
`clean()` (*sfepy.base.multiproc_mpi.RemoteQueueMaster* method), 803
`clear()` (*sfepy.solvers.ls.MUMPSSolver* method), 1021
`clear()` (*sfepy.solvers.ls.ScipyDirect* method), 1026
`clear()` (*sfepy.solvers.solvers.LinearSolver* method), 1036
`clear_cache()` (*sfepy.terms.terms_multilinear.ETermBase* method), 1147
`clear_equations()` (*sfepy.discrete.problem.Problem* method), 833
`clear_evaluate_cache()` (*sfepy.discrete.variables.FieldVariable* method), 848
`clear_facet_neighbour_idx_cache()` (*sfepy.discrete.dg.fields.DGField* method), 919
`clear_facet_qp_basis()` (*sfepy.discrete.dg.fields.DGField* method), 919
`clear_facet_vols_cache()` (*sfepy.discrete.dg.fields.DGField* method), 919
`clear_lin_solver()` (*sfepy.solvers.ts_solvers.BatheTS* method), 1044
`clear_lin_solver()` (*sfepy.solvers.ts_solvers.ElastodynamicsBaseTS* method), 1045
`clear_mappings()` (*sfepy.discrete.common.fields.Field* method), 864
`clear_normals_cache()` (*sfepy.discrete.dg.fields.DGField* method), 920
`clear_qp_basis()` (*sfepy.discrete.fem.fields_base.FEField* method), 881
`clear_surface_groups()`

`(sfepy.discrete.fem.domain.FEDomain method)`, 877
`close()` (`sfepy.base.multiproc_mpi.MPIFileHandler method`), 801
`close()` (`sfepy.base.multiproc_mpi.MPILogFile method`), 802
`CMapping` (class in `sfepy.discrete.common.extmods.cmapping`), 861
`cmem_statistics()` (in module `sfepy.discrete.common.extmods.cmesh`), 863
`CMesh` (class in `sfepy.discrete.common.extmods.cmesh`), 862
`CNURBSContext` (class in `sfepy.discrete.iga.extmods.igac`), 936
`coef_arrays_to_dicts()` (in module `sfepy.homogenization.coefficients`), 952
`CoefDim` (class in `sfepy.homogenization.coefs_base`), 952
`CoefDimDim` (class in `sfepy.homogenization.coefs_base`), 952
`CoefDimSym` (class in `sfepy.homogenization.coefs_base`), 952
`CoefDummy` (class in `sfepy.homogenization.coefs_base`), 952
`CoefEval` (class in `sfepy.homogenization.coefs_base`), 952
`CoefExprPar` (class in `sfepy.homogenization.coefs_base`), 952
`Coefficients` (class in `sfepy.homogenization.coefficients`), 951
`CoefMN` (class in `sfepy.homogenization.coefs_base`), 952
`CoefN` (class in `sfepy.homogenization.coefs_base`), 952
`CoefNone` (class in `sfepy.homogenization.coefs_base`), 952
`CoefNonSym` (class in `sfepy.homogenization.coefs_base`), 952
`CoefNonSymNonSym` (class in `sfepy.homogenization.coefs_base`), 952
`CoefOne` (class in `sfepy.homogenization.coefs_base`), 952
`CoefSum` (class in `sfepy.homogenization.coefs_base`), 953
`CoefSym` (class in `sfepy.homogenization.coefs_base`), 953
`CoefSymDim` (class in `sfepy.homogenization.coefs_base`), 953
`CoefSymSym` (class in `sfepy.homogenization.coefs_base`), 953
`CoefVolume` (class in `sfepy.homogenization.engine`), 960
`collect_conn_info()` (`sfepy.discrete.equations.Equation method`), 811
`collect_conn_info()` (`sfepy.discrete.equations.Equations method`), 812
`collect_materials()` (`sfepy.discrete.equations.Equation method`), 811
`collect_materials()` (`sfepy.discrete.equations.Equations method`), 812
`collect_modifiers()` (in module `sfepy.terms.terms_multilinear`), 1151
`collect_term()` (in module `sfepy.discrete.parse_equations`), 827
`collect_variables()` (`sfepy.discrete.equations.Equation method`), 811
`collect_variables()` (`sfepy.discrete.equations.Equations method`), 812
`combine_bezier_extraction()` (in module `sfepy.discrete.iga.iga`), 940
`combine_scalar_grad()` (in module `sfepy.homogenization.recovery`), 964
`compare_mesh()` (in module `sfepy.tests.test_domain`), 1191
`compare_scalars()` (in module `sfepy.tests.test_homogenization_perfusion`), 1193
`compare_vectors()` (in module `sfepy.base.testing`), 807
`compile_flags()` (`sfepy.config.Config method`), 777
`compose_sparse()` (in module `sfepy.linalg.sparse`), 973
`compose_system_compile_flags()` (in module `sfepy.config`), 778
`ComposedLimiter` (class in `sfepy.discrete.dg.limiters`), 931
`CompressibilityULTerm` (class in `sfepy.terms.terms_hyperelastic_ul`), 1125
`compute_bezier_control()` (in module `sfepy.discrete.iga.iga`), 940
`compute_bezier_extraction()` (in module `sfepy.discrete.iga.iga`), 941
`compute_bezier_extraction_ld()` (in module `sfepy.discrete.iga.iga`), 941
`compute_cat_dim_dim()` (in module `sfepy.homogenization.coefs_phononic`), 958
`compute_cat_dim_sym()` (in module `sfepy.homogenization.coefs_phononic`), 958
`compute_cat_sym_sym()` (in module `sfepy.homogenization.coefs_phononic`), 958
`compute_data()` (`sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm method`), 1117
`compute_data()` (`sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm method`), 1124
`compute_eigenmomenta()` (in module `sfepy.homogenization.coefs_phononic`), 958
`compute_fibre_strain()` (in module `sfepy.terms.terms_fibres`), 1111
`compute_jacobian()` (`sfepy.solvers.semismooth_newton.SemismoothNewton method`), 1036
`compute_mac_stress_part()` (in module

- sfepy.homogenization.recovery*), 964
- `compute_mean_decay()` (in module *sfepy.homogenization.convolution*), 960
- `compute_micro_u()` (in module *sfepy.homogenization.recovery*), 964
- `compute_nodal_edge_dirs()` (in module *sfepy.discrete.fem.utils*), 912
- `compute_nodal_normals()` (in module *sfepy.discrete.fem.utils*), 912
- `compute_p_corr_steady()` (in module *sfepy.homogenization.recovery*), 964
- `compute_p_corr_time()` (in module *sfepy.homogenization.recovery*), 964
- `compute_p_from_macro()` (in module *sfepy.homogenization.recovery*), 965
- `compute_stress()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBase* method), 1115
- `compute_stress_strain_u()` (in module *sfepy.homogenization.recovery*), 965
- `compute_tan_mod()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBase* method), 1115
- `compute_u_corr_steady()` (in module *sfepy.homogenization.recovery*), 965
- `compute_u_corr_time()` (in module *sfepy.homogenization.recovery*), 965
- `compute_u_from_macro()` (in module *sfepy.homogenization.recovery*), 965
- `ComsolMeshIO` (class in *sfepy.discrete.fem.meshio*), 900
- `ConcentratedPointLoadTerm` (class in *sfepy.terms.terms_point*), 1163
- `Condition` (class in *sfepy.discrete.conditions*), 808
- `Conditions` (class in *sfepy.discrete.conditions*), 808
- `Config` (class in *sfepy.config*), 777
- `configure_output()` (in module *sfepy.base.base*), 782
- `ConnInfo` (class in *sfepy.terms.terms*), 1050
- `conns` (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
- `ConstantFunction` (class in *sfepy.discrete.functions*), 822
- `ConstantFunctionByRegion` (class in *sfepy.discrete.functions*), 822
- `ContactInfo` (class in *sfepy.terms.terms_contact*), 1077
- `ContactIPCTerm` (class in *sfepy.terms.terms_contact*), 1077
- `ContactPlane` (class in *sfepy.mechanics.contact_bodies*), 979
- `ContactPlaneTerm` (class in *sfepy.terms.terms_surface*), 1172
- `ContactSphere` (class in *sfepy.mechanics.contact_bodies*), 979
- `ContactSphereTerm` (class in *sfepy.terms.terms_surface*), 1173
- `ContactTerm` (class in *sfepy.terms.terms_contact*), 1078
- `Container` (class in *sfepy.base.base*), 780
- `contains()` (*sfepy.discrete.common.region.Region* method), 874
- `conv_test()` (in module *sfepy.solvers.nls*), 1031
- `conv_test()` (in module *sfepy.solvers.optimize*), 1033
- `ConvectTerm` (class in *sfepy.terms.terms_navier_stokes*), 1152
- `ConvectVGradSTerm` (class in *sfepy.terms.terms_diffusion*), 1084
- `convert_complex_output()` (in module *sfepy.discrete.fem.meshio*), 906
- `convert_to_csr()` (in module *sfepy.tests.test_semismooth_newton*), 1199
- `ConvolutionKernel` (class in *sfepy.homogenization.convolution*), 960
- `convolve_field_scalar()` (in module *sfepy.homogenization.recovery*), 965
- `convolve_field_sym_tensor()` (in module *sfepy.homogenization.recovery*), 965
- `coo_is_symmetric()` (*sfepy.solvers.ls.MUMPSSolver* method), 1021
- `coord_to_sym()` (in module *sfepy.homogenization.utils*), 967
- `coors` (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
- `coors` (*sfepy.discrete.fem.mesh.Mesh* property), 898
- `copy()` (*sfepy.base.base.Struct* method), 782
- `copy()` (*sfepy.discrete.common.region.Region* method), 874
- `copy()` (*sfepy.discrete.fem.mesh.Mesh* method), 898
- `copy()` (*sfepy.discrete.problem.Problem* method), 833
- `CorrDim` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrDimDim` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrEqPar` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrEval` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrMiniApp` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrN` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrNN` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrOne` (class in *sfepy.homogenization.coefs_base*), 953
- `CorrSetBCS` (class in *sfepy.homogenization.coefs_base*), 954
- `CorrSolution` (class in *sfepy.homogenization.coefs_base*), 954
- `count` (*sfepy.base.log.Log* attribute), 798
- `cprint()` (*sfepy.discrete.common.extmods.cmesh.CConnectivity* method), 861
- `cprint()` (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 862
- `cprint()` (*sfepy.discrete.fem.extmods.bases.CLagrangeContext* method), 878
- `cprint()` (*sfepy.discrete.iga.extmods.igac.CNURBSContext*

- method*), 936
- `cpu_count()` (in module *sfepy.base.multiproc_mpi*), 803
- `create()` (*sfepy.base.timing.Timers* method), 807
- `create_adof_conn()` (in module *sfepy.discrete.variables*), 857
- `create_adof_conns()` (in module *sfepy.discrete.variables*), 857
- `create_arg_parser()` (in module *sfepy.terms.terms*), 1054
- `create_basis_context()` (*sfepy.discrete.fem.fields_hierarchic.H1HierarchicalField* method), 887
- `create_basis_context()` (*sfepy.discrete.fem.fields_nodal.H1NodalMixIn* method), 889
- `create_basis_context()` (*sfepy.discrete.fem.fields_nodal.H1SEMVVolumeField* method), 890
- `create_basis_context()` (*sfepy.discrete.fem.fields_nodal.H1SNodalVolumeField* method), 890
- `create_basis_context()` (*sfepy.discrete.fem.fields_positive.H1BernsteinVolumeField* method), 890
- `create_basis_context()` (*sfepy.discrete.iga.fields.IGField* method), 938
- `create_bnf()` (in module *sfepy.base.parse_conf*), 805
- `create_bnf()` (in module *sfepy.discrete.parse_equations*), 827
- `create_bnf()` (in module *sfepy.discrete.parse_regions*), 827
- `create_boundary_qp()` (in module *sfepy.discrete.iga.iga*), 941
- `create_bqp()` (*sfepy.discrete.fem.fields_base.FEField* method), 881
- `create_bqp_key()` (*sfepy.discrete.fem.fields_base.FEField* method), 881
- `create_conn_graph()` (*sfepy.discrete.fem.mesh.Mesh* method), 898
- `create_connectivity()` (in module *sfepy.discrete.iga.iga*), 941
- `create_connectivity_1d()` (in module *sfepy.discrete.iga.iga*), 942
- `create_context()` (*sfepy.discrete.fem.poly_spaces.LagrangePolySpace* method), 908
- `create_context()` (*sfepy.discrete.fem.poly_spaces.LagrangeSimplexPolySpace* method), 908
- `create_context()` (*sfepy.discrete.fem.poly_spaces.SerendipityTensorProductPolySpace* method), 911
- `create_dof_graph()` (in module *sfepy.discrete.equations*), 817
- `create_drl_transform()` (in module *sfepy.mechanics.shell10x*), 986
- `create_elastic_tensor()` (in module *sfepy.mechanics.shell10x*), 986
- `create_eps()` (*sfepy.solvers.eigen.SLEPcEigenvalueSolver* method), 1019
- `create_eval_mesh()` (*sfepy.discrete.common.fields.Field* method), 864
- `create_eval_mesh()` (*sfepy.discrete.iga.fields.IGField* method), 938
- `create_evaluable()` (in module *sfepy.discrete.evaluate*), 819
- `create_evaluable()` (*sfepy.discrete.problem.Problem* method), 833
- `create_expression_output()` (in module *sfepy.discrete.fem.fields_base*), 885
- `create_extra_tdim_region()` (*sfepy.discrete.common.domain.Domain* method), 860
- `create_from_igakit()` (in module *sfepy.discrete.iga.domain_generators*), 935
- `create_gather_scatter()` (in module *sfepy.parallel.parallel*), 1008
- `create_gather_to_zero()` (in module *sfepy.parallel.parallel*), 1008
- `create_geometry_elements()` (in module *sfepy.discrete.fem.geometry_element*), 891
- `create_ksp()` (*sfepy.solvers.ls.PETScKrylovSolver* method), 1023
- `create_linear_fe_mesh()` (in module *sfepy.discrete.iga.utils*), 948
- `create_local_bases()` (in module *sfepy.mechanics.shell10x*), 986
- `create_local_petsc_vector()` (in module *sfepy.parallel.parallel*), 1008
- `create_mapping()` (in module *sfepy.mechanics.membranes*), 983
- `create_mapping()` (*sfepy.discrete.dg.fields.DGField* method), 920
- `create_mapping()` (*sfepy.discrete.fem.fields_base.FEField* method), 881
- `create_mapping()` (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* method), 887
- `create_mapping()` (*sfepy.discrete.iga.fields.IGField* method), 938
- `create_mapping()` (*sfepy.discrete.structural.fields.Shell10XField* method), 949
- `create_mass_matrix()` (in module *sfepy.discrete.projections*), 844
- `create_materials()` (*sfepy.discrete.problem.Problem* method), 805
- `create_matrix_graph()` (in module *sfepy.discrete.equations*), 817
- `create_matrix_graph()` (*sfepy.discrete.equations.Equations* method), 812

[create_mesh\(\)](#) (*sfepy.discrete.fem.fields_base.FEField method*), 882
[create_mesh\(\)](#) (*sfepy.discrete.iga.fields.IGField method*), 938
[create_mesh_and_output\(\)](#) (*in module sfepy.discrete.iga.utils*), 949
[create_new\(\)](#) (*sfepy.discrete.common.extmods.cmesh.CMesh method*), 862
[create_nlst\(\)](#) (*sfepy.solvers.ts_solvers.CentralDifferenceTS method*), 1044
[create_nlst\(\)](#) (*sfepy.solvers.ts_solvers.GeneralizedAlphaTS method*), 1046
[create_nlst\(\)](#) (*sfepy.solvers.ts_solvers.NewmarkTS method*), 1047
[create_nlst\(\)](#) (*sfepy.solvers.ts_solvers.VelocityVerletTS method*), 1049
[create_nlst1\(\)](#) (*sfepy.solvers.ts_solvers.BatheTS method*), 1044
[create_nlst2\(\)](#) (*sfepy.solvers.ts_solvers.BatheTS method*), 1044
[create_omega\(\)](#) (*in module sfepy.terms.terms_fibres*), 1111
[create_output\(\)](#) (*in module sfepy.discrete.fem.linearizer*), 895
[create_output\(\)](#) (*sfepy.discrete.dg.fields.DGField method*), 920
[create_output\(\)](#) (*sfepy.discrete.fem.fields_base.FEField method*), 882
[create_output\(\)](#) (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField method*), 887
[create_output\(\)](#) (*sfepy.discrete.iga.fields.IGField method*), 939
[create_output\(\)](#) (*sfepy.discrete.structural.fields.Shell10xField method*), 949
[create_output\(\)](#) (*sfepy.discrete.variables.FieldVariable method*), 848
[create_output\(\)](#) (*sfepy.discrete.variables.Variables method*), 853
[create_parser\(\)](#) (*in module gen_term_table*), 1206
[create_petsc_matrix\(\)](#) (*in module sfepy.parallel.parallel*), 1008
[create_petsc_matrix\(\)](#) (*sfepy.solvers.eigen.SLEPcEigenvalueSolver method*), 1019
[create_petsc_matrix\(\)](#) (*sfepy.solvers.ls.PETScKrylovSolver method*), 1023
[create_petsc_system\(\)](#) (*in module sfepy.parallel.parallel*), 1008
[create_pis\(\)](#) (*in module sfepy.homogenization.utils*), 967
[create_prealloc_data\(\)](#) (*in module sfepy.parallel.parallel*), 1008
[create_reduced_vec\(\)](#) (*sfepy.discrete.equations.Equations method*), 812
[create_reduced_vec\(\)](#) (*sfepy.discrete.variables.Variables method*), 854
[create_region\(\)](#) (*sfepy.discrete.common.domain.Domain method*), 860
[create_regions\(\)](#) (*sfepy.discrete.common.domain.Domain method*), 860
[create_rotation_ops\(\)](#) (*in module sfepy.mechanics.shell10x*), 986
[create_scalar_pis\(\)](#) (*in module sfepy.homogenization.utils*), 967
[create_spb\(\)](#) (*sfepy.mesh.splinebox.SplineBox static method*), 1006
[create_spb\(\)](#) (*sfepy.mesh.splinebox.SplineRegion2D static method*), 1007
[create_state\(\)](#) (*sfepy.discrete.problem.Problem method*), 835
[create_strain_matrix\(\)](#) (*in module sfepy.mechanics.shell10x*), 986
[create_strain_transform\(\)](#) (*in module sfepy.mechanics.shell10x*), 986
[create_subequations\(\)](#) (*sfepy.discrete.equations.Equations method*), 813
[create_subproblem\(\)](#) (*sfepy.discrete.problem.Problem method*), 835
[create_surface_facet\(\)](#) (*sfepy.discrete.fem.geometry_element.GeometryElement method*), 891
[create_surface_group\(\)](#) (*sfepy.discrete.fem.domain.FEDomain method*), 877
[create_task_dof_maps\(\)](#) (*in module sfepy.parallel.parallel*), 1008
[create_transformation_matrix\(\)](#) (*in module sfepy.mechanics.membranes*), 983
[create_transformation_matrix\(\)](#) (*in module sfepy.mechanics.shell10x*), 987
[create_variables\(\)](#) (*sfepy.discrete.problem.Problem method*), 835
[create_vec\(\)](#) (*sfepy.discrete.equations.Equations method*), 813
[create_vec\(\)](#) (*sfepy.discrete.variables.Variables method*), 854
[cut_freq_range\(\)](#) (*in module sfepy.homogenization.coefs_phononic*), 958
[cvt_array_index\(\)](#) (*in module sfepy.base.parse_conf*), 805
[cvt_cmplx\(\)](#) (*in module sfepy.base.parse_conf*), 805
[cvt_int\(\)](#) (*in module sfepy.base.parse_conf*), 805
[cvt_none\(\)](#) (*in module sfepy.base.parse_conf*), 805

cvt_real() (in module *sfepy.base.parse_conf*), 805
 cycle() (in module *sfepy.linalg.utils*), 975

D

d_biot_div() (in module *sfepy.terms.extmods.terms*), 1179
 d_diffusion() (in module *sfepy.terms.extmods.terms*), 1179
 d_div_grad() (*sfepy.terms.terms_navier_stokes.DivGradTerm* static method), 1152
 d_dot() (*sfepy.terms.terms_dot.DotProductTerm* static method), 1092
 d_dot() (*sfepy.terms.terms_dot.VectorDotScalarTerm* static method), 1096
 d_eval() (*sfepy.terms.terms_navier_stokes.StokesTerm* static method), 1159
 d_fun() (*sfepy.terms.terms_diffusion.DiffusionCoupling* static method), 1085
 d_fun() (*sfepy.terms.terms_surface.LinearTractionTerm* static method), 1174
 d_fun() (*sfepy.terms.terms_surface.SDLinearTractionTerm* static method), 1175
 d_fun() (*sfepy.terms.terms_surface.SurfaceNormalDotTerm* static method), 1176
 d_laplace() (in module *sfepy.terms.extmods.terms*), 1179
 d_lin_elastic() (in module *sfepy.terms.extmods.terms*), 1179
 d_lin_prestress() (*sfepy.terms.terms_elastic.LinearPrestressTerm* static method), 1105
 d_of_nsMinGrad() (in module *sfepy.terms.extmods.terms*), 1179
 d_of_nsSurfMinDPress() (in module *sfepy.terms.extmods.terms*), 1179
 d_piezo_coupling() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_convect() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_diffusion() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_div() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_div_grad() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_lin_elastic() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_st_grad_div() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_st_pspg_c() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_st_pspg_p() (in module *sfepy.terms.extmods.terms*), 1179
 d_sd_st_supg_c() (in module *sfepy.terms.extmods.terms*), 1179

d_sd_volume_dot() (in module *sfepy.terms.extmods.terms*), 1179
 d_surface_flux() (in module *sfepy.terms.extmods.terms*), 1179
 d_tl_surface_flux() (in module *sfepy.terms.extmods.terms*), 1179
 d_tl_volume_surface() (in module *sfepy.terms.extmods.terms*), 1179
 d_volume_surface() (in module *sfepy.terms.extmods.terms*), 1179
 data() (in module *sfepy.tests.test_assembling*), 1188
 data() (in module *sfepy.tests.test_conditions*), 1189
 data() (in module *sfepy.tests.test_eigenvalue_solvers*), 1192
 data() (in module *sfepy.tests.test_high_level*), 1193
 data() (in module *sfepy.tests.test_laplace_unit_disk*), 1194
 data() (in module *sfepy.tests.test_laplace_unit_square*), 1194
 data() (in module *sfepy.tests.test_projections*), 1198
 data() (in module *sfepy.tests.test_regions*), 1199
 data() (in module *sfepy.tests.test_term_call_modes*), 1200
 data_names (*sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTLFamilyData* attribute), 1118
 data_names (*sfepy.terms.terms_hyperelastic_tl.HyperElasticTLFamilyData* attribute), 1119
 data_names (*sfepy.terms.terms_hyperelastic_ul.HyperElasticULFamilyData* attribute), 1126
 data_shapes (*sfepy.terms.terms_hyperelastic_base.HyperElasticFamilyData* attribute), 1115
 DataMarker (class in *sfepy.base.ioutils*), 793
 DataSoftLink (class in *sfepy.base.ioutils*), 793
 de_cauchy_strain() (in module *sfepy.terms.extmods.terms*), 1179
 de_cauchy_stress() (in module *sfepy.terms.extmods.terms*), 1179
 de_he_rtm() (in module *sfepy.terms.extmods.terms*), 1179
 debug() (in module *sfepy.base.base*), 780, 782
 debug_flags() (*sfepy.config.Config* method), 777
 debug_on_error() (in module *sfepy.base.base*), 782
 debug_warped_cells() (*sfepy.config.Config* method), 777
 dec() (in module *sfepy.base.ioutils*), 795
 dechunk_reqs_coefs() (*sfepy.homogenization.engine.HomogenizationWorkerMulti* static method), 963
 default_space_variables() (in module *sfepy.linalg.sympy_operators*), 974
 define() (in module *sfepy.tests.test_ed_solvers*), 1192
 define_box_regions() (in module *sfepy.homogenization.utils*), 967
 define_control_points()

(*sfepy.mesh.splinebox.SplineRegion2D* static method), 1007
 define_matrices() (in module *sfepy.tests.test_semismooth_newton*), 1199
 define_volume_coef() (*sfepy.homogenization.engine.HomogenizationEngine* static method), 960
 DeformationGradientTerm (class in *sfepy.terms.terms_hyperelastic_base*), 1114
 delete_zero_faces() (*sfepy.discrete.common.region.Region* method), 874
 DensityVolumeInfo (class in *sfepy.homogenization.coefs_phononic*), 956
 describe_deformation() (in module *sfepy.mechanics.membranes*), 984
 describe_gaps() (in module *sfepy.homogenization.coefs_phononic*), 958
 describe_geometry() (in module *sfepy.mechanics.membranes*), 984
 describe_nodes() (*sfepy.discrete.fem.poly_spaces.FEPolySpace* method), 907
 description (*build_helpers.DoxygenDocs* attribute), 1201
 description (*build_helpers.SphinxHTMLDocs* attribute), 1202
 description (*build_helpers.SphinxPDFDocs* attribute), 1202
 destroy_pool() (in module *sfepy.homogenization.recovery*), 966
 detect_band_gaps() (in module *sfepy.homogenization.coefs_phononic*), 958
 dets_fast() (in module *sfepy.linalg.utils*), 975
 dg_test_env() (in module *sfepy.tests.test_dg_terms_calls*), 1191
 DGEssentialBC (class in *sfepy.discrete.conditions*), 808
 DGField (class in *sfepy.discrete.dg.fields*), 919
 DGFieldVariable (class in *sfepy.discrete.variables*), 847
 DGLimiter (class in *sfepy.discrete.dg.limiters*), 931
 DGMultiStageTSS (class in *sfepy.solvers.ts_dg_solvers*), 932
 DGPeriodicBC (class in *sfepy.discrete.conditions*), 808
 DGTerm (class in *sfepy.terms.terms_dg*), 1080
 DGTermTestEnvironment (class in *sfepy.tests.test_dg_terms_calls*), 1190
 di_surface_moment() (in module *sfepy.terms.extmods.terms*), 1179
 DiagPC (class in *sfepy.tests.test_linear_solvers*), 1195
 dict_extend() (in module *sfepy.base.base*), 782
 dict_from_keys_init() (in module *sfepy.base.base*), 783
 dict_from_options() (in module *sfepy.base.conf*), 791
 dict_from_string() (in module *sfepy.base.conf*), 792
 dict_to_array() (in module *sfepy.base.base*), 783
 dict_to_struct() (in module *sfepy.base.base*), 783
 diff_dt() (*sfepy.homogenization.convolution.ConvolutionKernel* method), 960
 diff_info (*sfepy.terms.terms.Term* attribute), 1051
 diff_info (*sfepy.terms.terms_jax.LinearElasticLADTerm* attribute), 1129
 diff_info (*sfepy.terms.terms_jax.LinearElasticYPADTerm* attribute), 1129
 diff_info (*sfepy.terms.terms_jax.MassADTerm* attribute), 1130
 diff_info (*sfepy.terms.terms_jax.NeoHookeanTLADTerm* attribute), 1131
 diff_info (*sfepy.terms.terms_jax.OgdenTLADTerm* attribute), 1131
 DiffusionCoupling (class in *sfepy.terms.terms_diffusion*), 1085
 DiffusionDGFluxTerm (class in *sfepy.terms.terms_dg*), 1080
 DiffusionInteriorPenaltyTerm (class in *sfepy.terms.terms_dg*), 1081
 DiffusionRTerm (class in *sfepy.terms.terms_diffusion*), 1085
 DiffusionTerm (class in *sfepy.terms.terms_diffusion*), 1086
 DiffusionTLTerm (class in *sfepy.terms.terms_hyperelastic_tl*), 1117
 DiffusionVelocityTerm (class in *sfepy.terms.terms_diffusion*), 1087
 dim (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
 dim2sym() (in module *sfepy.mechanics.tensors*), 988
 distribute_field_dofs() (in module *sfepy.parallel.parallel*), 1009
 distribute_fields_dofs() (in module *sfepy.parallel.parallel*), 1009
 div() (in module *sfepy.linalg.sympy_operators*), 974
 DivGradTerm (class in *sfepy.terms.terms_navier_stokes*), 1152
 DivOperatorTerm (class in *sfepy.terms.terms_navier_stokes*), 1153
 DivTerm (class in *sfepy.terms.terms_navier_stokes*), 1153
 do_interpolation() (in module *sfepy.tests.test_mesh_interp*), 1196
 DofInfo (class in *sfepy.discrete.common.dof_info*), 857
 Domain (class in *sfepy.discrete.common.domain*), 860
 domain() (in module *sfepy.tests.test_domain*), 1191
 dot_sequences() (in module *sfepy.linalg.utils*), 975
 DotProductTerm (class in *sfepy.terms.terms_dot*), 1091
 DotSPProductVolumeOperatorWETHTerm (class in *sfepy.terms.terms_dot*), 1092
 DotSPProductVolumeOperatorWTHTerm (class in

[sfepy.terms.terms_dot](#)), 1093
 DotSurfaceProductTerm (class in [sfepy.terms.terms_compat](#)), 1073
 DotVolumeProductTerm (class in [sfepy.terms.terms_compat](#)), 1073
 DoxygenDocs (class in [build_helpers](#)), 1201
 dq_cauchy_strain() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_def_grad() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_div_vector() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_finite_strain_tl() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_finite_strain_ul() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_grad() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_state_in_qp() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_finite_strain_surface() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_stress_bulk() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_stress_bulk_active() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_stress_mooney_rivlin() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_stress_neohook() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_tan_mod_bulk() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_tan_mod_bulk_active() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_tan_mod_mooney_rivlin() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_he_tan_mod_neohook() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_stress_bulk_pressure() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_tl_tan_mod_bulk_pressure_u() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_stress_bulk() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_stress_mooney_rivlin() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_stress_neohook() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_tan_mod_bulk() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_tan_mod_mooney_rivlin() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_he_tan_mod_neohook() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_stress_bulk_pressure() (in module [sfepy.terms.extmods.terms](#)), 1180
 dq_ul_tan_mod_bulk_pressure_u() (in module [sfepy.terms.extmods.terms](#)), 1180
 dR_dx ([sfepy.discrete.iga.extmods.igac.CNURBSContext](#) attribute), 936
 dR_dxi ([sfepy.discrete.iga.extmods.igac.CNURBSContext](#) attribute), 936
 draw() ([sfepy.mesh.bspline.BSpline](#) method), 995
 draw() ([sfepy.mesh.bspline.BSplineSurf](#) method), 997
 draw_arrow() (in module [sfepy.postprocess.plot_facets](#)), 1011
 draw_basis() ([sfepy.mesh.bspline.BSpline](#) method), 995
 draw_data() (in module [sfepy.base.log_plotter](#)), 800
 DSumNodalValuesTerm (class in [sfepy.terms.terms_compat](#)), 1072
 DSurfaceFluxTerm (class in [sfepy.terms.terms_compat](#)), 1072
 DSurfaceMomentTerm (class in [sfepy.terms.terms_compat](#)), 1072
 dump_to_vtk() (in module [sfepy.postprocess.time_history](#)), 1014
 DVolumeSurfaceTerm (class in [sfepy.terms.terms_compat](#)), 1073
 dw_adj_convect1() (in module [sfepy.terms.extmods.terms](#)), 1180
 dw_adj_convect2() (in module [sfepy.terms.extmods.terms](#)), 1180
 dw_biot_div() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_biot_grad() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_convect_v_grad_s() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_diffusion() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_diffusion_r() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_div() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_dot() ([sfepy.terms.terms_dot.DotProductTerm](#) static method), 1092
 dw_dot() ([sfepy.terms.terms_dot.VectorDotScalarTerm](#) static method), 1096
 dw_electric_source() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_fun() ([sfepy.terms.terms_diffusion.DiffusionCoupling](#) static method), 1085
 dw_fun() ([sfepy.terms.terms_dot.ScalarDotGradIScalarTerm](#) static method), 1094
 dw_fun() ([sfepy.terms.terms_surface.SufaceNormalDotTerm](#) static method), 1176
 dw_grad() (in module [sfepy.terms.extmods.terms](#)), 1181
 dw_he_rtm() (in module [sfepy.terms.extmods.terms](#)), 1181

<code>dw_laplace()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	
<code>dw_lin_convect()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_lin_elastic()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_lin_prestress()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_lin_strain_fib()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_nonsym_elastic()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_piezo_coupling()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_st_adj1_supg_p()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_st_adj2_supg_p()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_st_adj_supg_c()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_st_grad_div()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_st_pspg_c()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	
<code>dw_st_supg_c()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	
<code>dw_st_supg_p()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	
<code>dw_surface_flux()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_surface_ltr()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_surface_s_v_dot_n()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_surface_v_dot_n_s()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_tl_diffusion()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_tl_surface_traction()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1181	module
<code>dw_tl_volume()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	
<code>dw_ul_volume()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	
<code>dw_v_dot_grad_s_sw()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	module
<code>dw_v_dot_grad_s_vw()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	module
<code>dw_volume_dot_scalar()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	module
<code>dw_volume_dot_vector()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	module
<code>dw_volume_lvf()</code> (in module <code>sfepy.terms.extmods.terms</code>), 1182	module
E	
<code>e_coors_max</code> (<code>sfepy.discrete.fem.extmods.bases.CLagrangeContext</code> attribute), 878	
<code>e_coors_max</code> (<code>sfepy.discrete.iga.extmods.igac.CNURBSContext</code> attribute), 936	
<code>ebase2fbase()</code> (in module <code>gen_gallery</code>), 1204	
<code>ebc()</code> (in module <code>sfepy.tests.test_msm_laplace</code>), 1197	
<code>ebc()</code> (in module <code>sfepy.tests.test_msm_symbolic</code>), 1197	
<code>ECauchyStressTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1137	in
<code>EConvectTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1137	
<code>edge_oris</code> (<code>sfepy.discrete.common.extmods.cmesh.CMesh</code> attribute), 862	
<code>EdgeDirectionOperator</code> (class in <code>sfepy.discrete.fem.lcbc_operators</code>), 892	in
<code>edges</code> (<code>sfepy.discrete.common.region.Region</code> property), 874	
<code>EDiffusionTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1138	in
<code>edit()</code> (<code>sfepy.base.conf.ProblemConf</code> method), 790	
<code>edit_dict_strings()</code> (in module <code>sfepy.base.base</code>), 783	
<code>edit_filename()</code> (in module <code>sfepy.base.ioutils</code>), 795	
<code>edit_tuple_strings()</code> (in module <code>sfepy.base.base</code>), 783	
<code>EDivGradTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1138	
<code>EDivTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1139	
<code>EDotTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1139	
<code>EGradTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1140	
<code>eig()</code> (in module <code>sfepy.solvers.eigen</code>), 1020	
<code>Eigenmomenta</code> (class in <code>sfepy.homogenization.coefs_phononic</code>), 956	in
<code>EigenvalueSolver</code> (class in <code>sfepy.solvers.solvers</code>), 1036	
<code>EIntegrateOperatorTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1140	in
<code>ELaplaceTerm</code> (class in <code>sfepy.terms.terms_multilinear</code>), 1141	
<code>ElasticConstants</code> (class in <code>sfepy.mechanics.matcoefs</code>), 979	
<code>ElasticWaveCauchyTerm</code> (class in <code>sfepy.terms.terms_elastic</code>), 1099	in
<code>ElasticWaveTerm</code> (class in <code>sfepy.terms.terms_elastic</code>), 1100	
<code>ElastodynamicsBaseTS</code> (class in <code>sfepy.solvers.ts_solvers</code>), 1045	in
<code>ElastodynamicsBasicTSC</code> (class in <code>sfepy.solvers.ts_controllers</code>), 1039	in
<code>ElastodynamicsLinearTSC</code> (class in <code>sfepy.solvers.ts_controllers</code>), 1039	in
<code>ElastodynamicsPIDTSC</code> (class in <code>sfepy.solvers.ts_controllers</code>), 1040	in

ElectricSourceTerm (class in *sfepy.terms.terms_electric*), 1109

elems_q2t() (in module *sfepy.mesh.mesh_tools*), 1004

elevate() (*sfepy.discrete.iga.domain.NurbsPatch* method), 934

ELinearConvectTerm (class in *sfepy.terms.terms_multilinear*), 1142

ELinearElasticTerm (class in *sfepy.terms.terms_multilinear*), 1142

ELinearTractionTerm (class in *sfepy.terms.terms_multilinear*), 1143

emit() (*sfepy.base.multiproc_mpi.MPIFileHandler* method), 802

enc() (in module *sfepy.base.ioutils*), 795

ENonPenetrationPenaltyTerm (class in *sfepy.terms.terms_multilinear*), 1143

ENonSymElasticTerm (class in *sfepy.terms.terms_multilinear*), 1144

ensure3d() (in module *sfepy.scripts.resview*), 1187

ensure_path() (in module *sfepy.base.ioutils*), 795

entities (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862

enum() (in module *sfepy.base.multiproc_mpi*), 803

Equation (class in *sfepy.discrete.equations*), 811

equation_mapping() (*sfepy.discrete.variables.FieldVariable* method), 848

equation_mapping() (*sfepy.discrete.variables.Variables* method), 854

EquationMap (class in *sfepy.discrete.common.dof_info*), 858

Equations (class in *sfepy.discrete.equations*), 811

errclear() (in module *sfepy.terms.extmods.terms*), 1182

EScalarDotMGradScalarTerm (class in *sfepy.terms.terms_multilinear*), 1145

ESDDiffusionTerm (class in *sfepy.terms.terms_sensitivity*), 1164

ESDDivGradTerm (class in *sfepy.terms.terms_sensitivity*), 1165

ESDDotTerm (class in *sfepy.terms.terms_sensitivity*), 1165

ESDLinearElasticTerm (class in *sfepy.terms.terms_sensitivity*), 1166

ESDLinearTractionTerm (class in *sfepy.terms.terms_sensitivity*), 1167

ESDPiezoCouplingTerm (class in *sfepy.terms.terms_sensitivity*), 1167

ESDStokesTerm (class in *sfepy.terms.terms_sensitivity*), 1168

ESDVectorDotGradScalarTerm (class in *sfepy.terms.terms_sensitivity*), 1169

EssentialBC (class in *sfepy.discrete.conditions*), 808

EStokesTerm (class in *sfepy.terms.terms_multilinear*), 1145

ETermBase (class in *sfepy.terms.terms_multilinear*), 1146

ETHTerm (class in *sfepy.terms.terms_th*), 1177

EulerStepSolver (class in *sfepy.solvers.ts_dg_solvers*), 933

eval() (*sfepy.mesh.bspline.BSpline* method), 995

eval() (*sfepy.mesh.bspline.BSplineSurf* method), 997

eval_alpha_ogden() (in module *sfepy.terms.terms_jax*), 1132

eval_basis() (*sfepy.discrete.common.poly_spaces.PolySpace* method), 872

eval_basis() (*sfepy.discrete.fem.fields_base.FEField* method), 882

eval_basis() (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* method), 887

eval_basis() (*sfepy.discrete.fem.mappings.FEMapping* method), 896

eval_basis() (*sfepy.mesh.bspline.BSpline* method), 995

eval_bernstein_basis() (in module *sfepy.discrete.iga.extmods.igac*), 936

eval_bernstein_basis() (in module *sfepy.discrete.iga.iga*), 942

eval_complex() (in module *sfepy.discrete.evaluate_variable*), 822

eval_complex() (*sfepy.terms.terms.Term* method), 1051

eval_complex() (*sfepy.terms.terms_multilinear.ETermBase* method), 1147

eval_coor_expression() (in module *sfepy.base.testing*), 807

eval_density_mass() (in module *sfepy.terms.terms_jax*), 1132

eval_elasticity_l() (in module *sfepy.terms.terms_jax*), 1132

eval_elasticity_yp() (in module *sfepy.terms.terms_jax*), 1132

eval_equations() (in module *sfepy.discrete.evaluate*), 820

eval_equations() (*sfepy.discrete.problem.Problem* method), 835

eval_exponential() (in module *sfepy.homogenization.convolution*), 960

eval_fun() (in module *sfepy.tests.test_refine_hanging*), 1199

eval_function() (*sfepy.terms.terms_membrane.TLMembraneTerm* static method), 1136

eval_in_els_and_qp() (in module *sfepy.discrete.evaluate*), 821

eval_in_tp_coors() (in module *sfepy.discrete.iga.extmods.igac*), 936

eval_jac_elasticity_l() (in module *sfepy.terms.terms_jax*), 1132

eval_jac_elasticity_yp() (in module *sfepy.terms.terms_jax*), 1132

[sfepy.terms.terms_jax](#)), 1132
[eval_jac_mass\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_jac_neohook\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_jac_ogden\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_lagrangeld_basis\(\)](#) (in module [sfepy.discrete.fem.poly_spaces](#)), 911
[eval_lam_elasticity_l\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_lobattold\(\)](#) (in module [sfepy.discrete.fem.extmods.lobatto_bases](#)), 878
[eval_lobatto_tensor_product\(\)](#) (in module [sfepy.discrete.fem.extmods.lobatto_bases](#)), 878
[eval_mapping_data_in_qp\(\)](#) (in module [sfepy.discrete.fem.mappings](#)), 897
[eval_mapping_data_in_qp\(\)](#) (in module [sfepy.discrete.iga.extmods.igac](#)), 937
[eval_mapping_data_in_qp\(\)](#) (in module [sfepy.discrete.iga.iga](#)), 942
[eval_mass\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_matrix\(\)](#) (in module [sfepy.tests.test_semismooth_newton](#)), 1199
[eval_membrane_mooney_rivlin\(\)](#) (in module [sfepy.terms.terms_membrane](#)), 1136
[eval_mu_elasticity_l\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_mu_neohook\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_mu_ogden\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1133
[eval_neohook\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1134
[eval_nodal_coors\(\)](#) (in module [sfepy.discrete.fem.fields_base](#)), 886
[eval_nurbs_basis_tp\(\)](#) (in module [sfepy.discrete.iga.iga](#)), 943
[eval_ogden\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1134
[eval_op_cells\(\)](#) ([sfepy.discrete.common.region.Region](#) method), 874
[eval_op_edges\(\)](#) ([sfepy.discrete.common.region.Region](#) method), 874
[eval_op_faces\(\)](#) ([sfepy.discrete.common.region.Region](#) method), 874
[eval_op_facets\(\)](#) ([sfepy.discrete.common.region.Region](#) method), 874
[eval_op_vertices\(\)](#) ([sfepy.discrete.common.region.Region](#) method), 874
[eval_poisson_elasticity_yp\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1134
[eval_real\(\)](#) (in module [sfepy.discrete.evaluate_variable](#)), 822
[eval_real\(\)](#) ([sfepy.terms.terms.Term](#) method), 1051
[eval_real\(\)](#) ([sfepy.terms.terms_contact.ContactIPTerm](#) method), 1077
[eval_real\(\)](#) ([sfepy.terms.terms_contact.ContactTerm](#) method), 1078
[eval_real\(\)](#) ([sfepy.terms.terms_dg.DGTerm](#) method), 1080
[eval_real\(\)](#) ([sfepy.terms.terms_multilinear.ETermBase](#) method), 1147
[eval_real\(\)](#) ([sfepy.terms.terms_th.THTerm](#) method), 1177
[eval_residual\(\)](#) ([sfepy.discrete.evaluate.Evaluator](#) method), 818
[eval_residual\(\)](#) ([sfepy.parallel.evaluate.PETScParallelEvaluator](#) method), 1008
[eval_residuals\(\)](#) ([sfepy.discrete.equations.Equations](#) method), 813
[eval_scaled_norm\(\)](#) (in module [sfepy.solvers.ts_controllers](#)), 1042
[eval_tangent_matrices\(\)](#) ([sfepy.discrete.equations.Equations](#) method), 813
[eval_tangent_matrix\(\)](#) ([sfepy.discrete.evaluate.Evaluator](#) method), 818
[eval_tangent_matrix\(\)](#) ([sfepy.parallel.evaluate.PETScParallelEvaluator](#) method), 1008
[eval_variable_in_qp\(\)](#) (in module [sfepy.discrete.iga.extmods.igac](#)), 937
[eval_variable_in_qp\(\)](#) (in module [sfepy.discrete.iga.iga](#)), 944
[eval_young_elasticity_yp\(\)](#) (in module [sfepy.terms.terms_jax](#)), 1134
[evaluate\(\)](#) ([sfepy.discrete.equations.Equation](#) method), 811
[evaluate\(\)](#) ([sfepy.discrete.equations.Equations](#) method), 814
[evaluate\(\)](#) ([sfepy.discrete.fem.extmods.bases.CLagrangeContext](#) method), 878
[evaluate\(\)](#) ([sfepy.discrete.iga.domain.NurbsPatch](#) method), 934
[evaluate\(\)](#) ([sfepy.discrete.iga.extmods.igac.CNURBSContext](#) method), 936
[evaluate\(\)](#) ([sfepy.discrete.problem.Problem](#) method), 836
[evaluate\(\)](#) ([sfepy.discrete.variables.FieldVariable](#) method), 848
[evaluate\(\)](#) ([sfepy.homogenization.coefs_phononic.AcousticMassTensor](#) method), 955
[evaluate\(\)](#) ([sfepy.homogenization.coefs_phononic.AppliedLoadTensor](#) method), 955
[evaluate\(\)](#) ([sfepy.mesh.splinebox.SplineBox](#) method), 1006

evaluate() (*sfepy.terms.terms.Term* method), 1051
 evaluate_at() (*sfepy.discrete.common.fields.Field* method), 864
 evaluate_at() (*sfepy.discrete.variables.FieldVariable* method), 849
 evaluate_contact_constraints() (in module *sfepy.mechanics.extmods.ccontres*), 993
 evaluate_derivative() (*sfepy.mesh.splinebox.SplineBox* method), 1006
 evaluate_in_rc() (in module *sfepy.discrete.common.extmods.crefcoors*), 864
 Evaluator (class in *sfepy.discrete.evaluate*), 818
 EVPSolverApp (class in *sfepy.applications.evp_solver_app*), 778
 expand2d() (in module *sfepy.mesh.mesh_tools*), 1004
 expand_basis() (in module *sfepy.discrete.variables*), 857
 expand_dofs() (in module *sfepy.parallel.parallel*), 1009
 expand_nodes_to_dofs() (in module *sfepy.discrete.common.dof_info*), 859
 expand_nodes_to_equations() (in module *sfepy.discrete.common.dof_info*), 859
 ExpressionArg (class in *sfepy.terms.terms_multilinear*), 1147
 ExpressionBuilder (class in *sfepy.terms.terms_multilinear*), 1148
 extend() (*sfepy.base.base.Container* method), 780
 extend_cell_data() (in module *sfepy.discrete.fem.utils*), 912
 extend_dofs() (*sfepy.discrete.fem.fields_base.FEField* method), 882
 extend_dofs() (*sfepy.discrete.fem.fields_nodal.H1DiscontinuousField* method), 889
 extra_variables (*sfepy.solvers.ts_solvers.NewmarkTS* attribute), 1047
 extract_edges() (in module *sfepy.mesh.mesh_tools*), 1004
 extract_time_history() (in module *sfepy.postprocess.time_history*), 1014
 extract_times() (in module *sfepy.postprocess.time_history*), 1014

F

face_oris (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
 faces (*sfepy.discrete.common.region.Region* property), 874
 facet_oris (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 862
 facets (*sfepy.discrete.common.region.Region* property), 874

factorial() (in module *sfepy.discrete.simplex_cubature*), 847
 family_data_names (*sfepy.terms.terms_fibres.FibresActiveTLTerm* attribute), 1110
 family_data_names (*sfepy.terms.terms_fibres.FibresExponentialTLTerm* attribute), 1111
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm* attribute), 1115
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm* attribute), 1116
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm* attribute), 1117
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm* attribute), 1117
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm* attribute), 1118
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm* attribute), 1119
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm* attribute), 1120
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm* attribute), 1121
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm* attribute), 1121
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm* attribute), 1122
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm* attribute), 1123
 family_data_names (*sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm* attribute), 1123
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm* attribute), 1124
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm* attribute), 1124
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm* attribute), 1125
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.HyperelasticByFurukawaULTerm* attribute), 1126
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm* attribute), 1127
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm* attribute), 1127
 family_data_names (*sfepy.terms.terms_hyperelastic_ul.VolumeULTerm* attribute), 1128
 family_function() (*sfepy.terms.terms_hyperelastic_tl.HyperElasticSurface* static method), 1118
 family_function() (*sfepy.terms.terms_hyperelastic_tl.HyperElasticTLTerm* static method), 1119
 family_function() (*sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm* static method), 1125
 family_function() (*sfepy.terms.terms_hyperelastic_ul.HyperElasticULTerm* static method), 1126
 family_name (*sfepy.discrete.dg.fields.DGField* attribute), 921

family_name(*sfepy.discrete.fem.fields_hierarchic.H1HierarchicalField* attribute), 887
 family_name(*sfepy.discrete.fem.fields_l2.L2ConstantSurfaceField* attribute), 887
 family_name(*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* attribute), 887
 family_name(*sfepy.discrete.fem.fields_nodal.H1DiscontinuousField* attribute), 889
 family_name(*sfepy.discrete.fem.fields_nodal.H1NodalSurfaceField* attribute), 889
 family_name(*sfepy.discrete.fem.fields_nodal.H1NodalVolumeField* attribute), 889
 family_name(*sfepy.discrete.fem.fields_nodal.H1SEMSurfaceField* attribute), 890
 family_name(*sfepy.discrete.fem.fields_nodal.H1SEMVolumeField* attribute), 890
 family_name(*sfepy.discrete.fem.fields_nodal.H1SNodalSurfaceFields* attribute), 890
 family_name(*sfepy.discrete.fem.fields_nodal.H1SNodalVolumeFields* attribute), 890
 family_name(*sfepy.discrete.fem.fields_positive.H1BernsteinSurfaceField* attribute), 890
 family_name(*sfepy.discrete.fem.fields_positive.H1BernsteinVolumeField* attribute), 890
 family_name(*sfepy.discrete.iga.fields.IGField* attribute), 939
 family_name(*sfepy.discrete.structural.fields.Shell10XField* attribute), 950
 FEDomain (class in *sfepy.discrete.fem.domain*), 877
 FEField (class in *sfepy.discrete.fem.fields_base*), 881
 FEMapping (class in *sfepy.discrete.fem.mappings*), 896
 FEPhantomSurface (class in *sfepy.discrete.fem.fe_surface*), 880
 FEPolySpace (class in *sfepy.discrete.fem.poly_spaces*), 907
 FESurface (class in *sfepy.discrete.fem.fe_surface*), 880
 FibresActiveTLTerm (class in *sfepy.terms.terms_fibres*), 1110
 FibresExponentialTLTerm (class in *sfepy.terms.terms_fibres*), 1110
 Field (class in *sfepy.discrete.common.fields*), 864
 FieldOptsToListAction (class in *sfepy.scripts.resview*), 1186
 fields_from_conf() (in module *sfepy.discrete.common.fields*), 867
 FieldVariable (class in *sfepy.discrete.variables*), 848
 filename_meshes() (in module *sfepy.tests.test_cmesh*), 1189
 fill_state() (*sfepy.discrete.variables.Variables* method), 854
 finalize() (*sfepy.discrete.common.region.Region* method), 874
 finalize() (*sfepy.discrete.fem.lcbc_operators.LCBCOperator* method), 893
 finalizers() (*sfepy.base.base.OneTypeList* method), 781
 find_facet_substitutions() (in module *sfepy.discrete.fem.refine_hanging*), 912
 find_free_indices() (in module *sfepy.terms.terms_multilinear*), 1151
 find_level_interface() (in module *sfepy.discrete.fem.refine_hanging*), 912
 find_map() (in module *sfepy.discrete.fem.mesh*), 899
 find_ref_coors() (in module *sfepy.discrete.common.extmods.crefcoors*), 864
 find_ref_coors_convex() (in module *sfepy.discrete.common.extmods.crefcoors*), 864
 find_subclasses() (in module *sfepy.base.base*), 783
 find_ts() (*sfepy.mesh.splinebox.SplineRegion2D* method), 1007
 find_zero() (in module *sfepy.homogenization.coefs_phononic*), 958
 fit_exponential() (in module *sfepy.homogenization.convolution*), 960
 fix_double_nodes() (in module *sfepy.discrete.fem.mesh*), 899
 fix_eig_range() (*sfepy.homogenization.coefs_phononic.BandGaps* method), 956
 fix_element_orientation() (*sfepy.discrete.fem.domain.FEDomain* method), 877
 fix_u_fun() (in module *sfepy.tests.test_high_level*), 1193
 FixedTSC (class in *sfepy.solvers.ts_controllers*), 1041
 flag_points_in_polygon2d() (in module *sfepy.linalg.geometry*), 970
 FMinSteepestDescent (class in *sfepy.solvers.optimize*), 1031
 font_size() (in module *sfepy.base.plotutils*), 806
 format (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* attribute), 900
 format (*sfepy.discrete.fem.meshio.ComsolMeshIO* attribute), 900
 format (*sfepy.discrete.fem.meshio.GmshIO* attribute), 900
 format (*sfepy.discrete.fem.meshio.HDF5MeshIO* attribute), 902
 format (*sfepy.discrete.fem.meshio.HDF5XdmfMeshIO* attribute), 903
 format (*sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO* attribute), 903
 format (*sfepy.discrete.fem.meshio.Mesh3DMeshIO* attribute), 904

<code>format</code> (<code>sfepy.discrete.fem.meshio.MeshIO</code> attribute), 904	<code>from_desc()</code> (<code>sfepy.discrete.equations.Equation</code> static method), 811
<code>format</code> (<code>sfepy.discrete.fem.meshio.MeshioLibIO</code> attribute), 905	<code>from_desc()</code> (<code>sfepy.terms.terms.Term</code> static method), 1051
<code>format</code> (<code>sfepy.discrete.fem.meshio.NEUMeshIO</code> attribute), 906	<code>from_desc()</code> (<code>sfepy.terms.terms.Terms</code> static method), 1054
<code>format</code> (<code>sfepy.discrete.fem.meshio.UserMeshIO</code> attribute), 906	<code>from_dict()</code> (<code>sfepy.base.conf.ProblemConf</code> static method), 790
<code>format</code> (<code>sfepy.discrete.fem.meshio.XYZMeshIO</code> attribute), 906	<code>from_facets()</code> (<code>sfepy.discrete.common.region.Region</code> static method), 875
<code>format_next()</code> (in module <code>gen_term_table</code>), 1206	<code>from_file()</code> (<code>sfepy.base.conf.ProblemConf</code> static method), 790
<code>format_next()</code> (in module <code>sfepy.solvers.solvers</code>), 1037	<code>from_file()</code> (<code>sfepy.discrete.fem.mesh.Mesh</code> static method), 898
<code>free_connectivity()</code> (<code>sfepy.discrete.common.extmods.cmesh.CMesh</code> method), 862	<code>from_file()</code> (<code>sfepy.discrete.iga.domain.IGDomain</code> static method), 934
<code>from_args()</code> (<code>sfepy.discrete.common.fields.Field</code> static method), 865	<code>from_file_and_options()</code> (<code>sfepy.base.conf.ProblemConf</code> static method), 791
<code>from_args()</code> (<code>sfepy.discrete.common.mappings.Mapping</code> static method), 870	<code>from_file_hdf5()</code> (<code>sfepy.discrete.fem.history.Histories</code> static method), 892
<code>from_cells()</code> (<code>sfepy.discrete.common.region.Region</code> static method), 874	<code>from_file_hdf5()</code> (<code>sfepy.homogenization.coefficients.Coefficients</code> static method), 951
<code>from_conf()</code> (<code>sfepy.base.log.Log</code> static method), 798	<code>from_gmsh_file()</code> (<code>sfepy.mesh.geom_tools.geometry</code> static method), 999
<code>from_conf()</code> (<code>sfepy.discrete.common.fields.Field</code> static method), 866	<code>from_module()</code> (<code>sfepy.base.conf.ProblemConf</code> static method), 791
<code>from_conf()</code> (<code>sfepy.discrete.conditions.Conditions</code> static method), 808	<code>from_region()</code> (<code>sfepy.discrete.fem.fe_surface.FESurface</code> static method), 880
<code>from_conf()</code> (<code>sfepy.discrete.equations.Equations</code> static method), 815	<code>from_region()</code> (<code>sfepy.discrete.fem.mesh.Mesh</code> static method), 898
<code>from_conf()</code> (<code>sfepy.discrete.functions.Functions</code> static method), 822	<code>from_sequence()</code> (<code>sfepy.discrete.fem.history.History</code> static method), 892
<code>from_conf()</code> (<code>sfepy.discrete.integrals.Integrals</code> static method), 823	<code>from_table()</code> (<code>sfepy.discrete.quadratures.QuadraturePoints</code> static method), 846
<code>from_conf()</code> (<code>sfepy.discrete.materials.Material</code> static method), 824	<code>from_term_arg()</code> (<code>sfepy.terms.terms_multilinear.ExpressionArg</code> static method), 1148
<code>from_conf()</code> (<code>sfepy.discrete.materials.Materials</code> static method), 826	<code>from_vertices()</code> (<code>sfepy.discrete.common.region.Region</code> static method), 875
<code>from_conf()</code> (<code>sfepy.discrete.problem.Problem</code> static method), 836	<code>Function</code> (class in <code>sfepy.discrete.functions</code>), 822
<code>from_conf()</code> (<code>sfepy.discrete.variables.Variable</code> static method), 852	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term</code> static method), 1055
<code>from_conf()</code> (<code>sfepy.discrete.variables.Variables</code> static method), 854	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term</code> static method), 1056
<code>from_conf()</code> (<code>sfepy.solvers.ts.TimeStepper</code> static method), 1038	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm</code> static method), 1056
<code>from_conf()</code> (<code>sfepy.solvers.ts.VariableTimeStepper</code> static method), 1038	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm</code> static method), 1056
<code>from_conf_file()</code> (<code>sfepy.discrete.problem.Problem</code> static method), 836	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPresTe</code> static method), 1057
<code>from_data()</code> (<code>sfepy.discrete.common.extmods.cmesh.CMesh</code> class method), 862	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.SDConvectTerm</code> static method), 1058
<code>from_data()</code> (<code>sfepy.discrete.fem.mesh.Mesh</code> static method), 898	<code>function()</code> (<code>sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm</code> static method), 1059
<code>from_data()</code> (<code>sfepy.discrete.iga.domain.IGDomain</code> static method), 934	

function() (sfepy.terms.terms_adj_navier_stokes.SDDivTerm static method), 1059	function() (sfepy.terms.terms_diffusion.ConvectVGradSTerm static method), 1085
function() (sfepy.terms.terms_adj_navier_stokes.SDDotTerm static method), 1060	function() (sfepy.terms.terms_diffusion.DiffusionRTerm static method), 1086
function() (sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizerTerm static method), 1060	function() (sfepy.terms.terms_diffusion.DiffusionVelocityTerm static method), 1087
function() (sfepy.terms.terms_adj_navier_stokes.SDPSPCGStabilizerTerm static method), 1061	function() (sfepy.terms.terms_diffusion.NonlinearDiffusionTerm static method), 1088
function() (sfepy.terms.terms_adj_navier_stokes.SDPSPCGStabilizerTerm static method), 1062	function() (sfepy.terms.terms_diffusion.SDDiffusionTerm static method), 1089
function() (sfepy.terms.terms_adj_navier_stokes.SDSUPCGStabilizerTerm static method), 1062	function() (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm static method), 1090
function() (sfepy.terms.terms_adj_navier_stokes.SUPGCAStabilizerTerm static method), 1063	function() (sfepy.terms.terms_diffusion.SurfaceFluxTerm static method), 1090
function() (sfepy.terms.terms_adj_navier_stokes.SUPGPAStabilizerTerm static method), 1064	function() (sfepy.terms.terms_dot.DotSPProductVolumeOperatorWETHTerm static method), 1093
function() (sfepy.terms.terms_adj_navier_stokes.SUPGPAStabilizerTerm static method), 1064	function() (sfepy.terms.terms_dot.DotSPProductVolumeOperatorWTHTerm static method), 1093
function() (sfepy.terms.terms_basic.IntegrateMatTerm static method), 1065	function() (sfepy.terms.terms_dot.ScalarDotMGradScalarTerm static method), 1094
function() (sfepy.terms.terms_basic.IntegrateOperatorTerm static method), 1066	function() (sfepy.terms.terms_elastic.CauchyStrainTerm static method), 1097
function() (sfepy.terms.terms_basic.IntegrateTerm static method), 1066	function() (sfepy.terms.terms_elastic.CauchyStressTerm static method), 1099
function() (sfepy.terms.terms_basic.SumNodalValuesTerm static method), 1067	function() (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm static method), 1099
function() (sfepy.terms.terms_basic.SurfaceMomentTerm static method), 1067	function() (sfepy.terms.terms_elastic.ElasticWaveTerm static method), 1100
function() (sfepy.terms.terms_basic.VolumeSurfaceTerm static method), 1068	function() (sfepy.terms.terms_elastic.LinearDSpringTerm static method), 1101
function() (sfepy.terms.terms_basic.VolumeTerm static method), 1068	function() (sfepy.terms.terms_elastic.LinearElasticETHTerm static method), 1102
function() (sfepy.terms.terms_basic.ZeroTerm static method), 1069	function() (sfepy.terms.terms_elastic.LinearElasticTHTerm static method), 1103
function() (sfepy.terms.terms_biot.BiotStressTerm static method), 1070	function() (sfepy.terms.terms_elastic.LinearSpringTerm static method), 1105
function() (sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm static method), 1075	function() (sfepy.terms.terms_elastic.LinearStrainFiberTerm static method), 1106
function() (sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm static method), 1076	function() (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm static method), 1106
function() (sfepy.terms.terms_contact.ContactIPCTerm static method), 1077	function() (sfepy.terms.terms_elastic.LinearTrussTerm static method), 1107
function() (sfepy.terms.terms_contact.ContactTerm static method), 1078	function() (sfepy.terms.terms_elastic.SDLinearElasticTerm static method), 1109
function() (sfepy.terms.terms_dg.AdvectionDGFluxTerm static method), 1080	function() (sfepy.terms.terms_electric.ElectricSourceTerm static method), 1109
function() (sfepy.terms.terms_dg.DiffusionDGFluxTerm static method), 1081	function() (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm static method), 1114
function() (sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm static method), 1082	function() (sfepy.terms.terms_hyperelastic_base.HyperElasticBase static method), 1115
function() (sfepy.terms.terms_dg.NonlinearHyperbolicDGEffractionTerm static method), 1083	function() (sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm static method), 1117
function() (sfepy.terms.terms_dg.NonlinearScalarDotGradientTerm static method), 1083	function() (sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm static method), 1121

[function\(\)](#) ([sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTerm](#) static method), 1122
[function\(\)](#) ([sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTractionTerm](#) static method), 1123
[function\(\)](#) ([sfepy.terms.terms_hyperelastic_tl.VolumeTLTractionTerm](#) static method), 1123
[function\(\)](#) ([sfepy.terms.terms_hyperelastic_ul.CompressibleVolumeTerm](#) static method), 1125
[function\(\)](#) ([sfepy.terms.terms_hyperelastic_ul.VolumeULTerm](#) static method), 1128
[function\(\)](#) ([sfepy.terms.terms_jax.LinearElasticLADTerm](#) static method), 1129
[function\(\)](#) ([sfepy.terms.terms_jax.LinearElasticYPADTerm](#) static method), 1129
[function\(\)](#) ([sfepy.terms.terms_jax.MassADTerm](#) static method), 1130
[function\(\)](#) ([sfepy.terms.terms_jax.NeoHookeanTLADTerm](#) static method), 1131
[function\(\)](#) ([sfepy.terms.terms_jax.OgdenTLADTerm](#) static method), 1131
[function\(\)](#) ([sfepy.terms.terms_membrane.TLMembraneTerm](#) static method), 1136
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.ConvectTerm](#) static method), 1152
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.DivGradTerm](#) static method), 1153
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.DivOperatorTerm](#) static method), 1153
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.DivTerm](#) static method), 1154
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm](#) static method), 1154
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.GradTerm](#) static method), 1155
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.LinearConvect2Term](#) static method), 1155
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.LinearConvectTerm](#) static method), 1156
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm](#) static method), 1156
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.StokesWaveDivTerm](#) static method), 1159
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.StokesWaveTerm](#) static method), 1160
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm](#) static method), 1157
[function\(\)](#) ([sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm](#) static method), 1158
[function\(\)](#) ([sfepy.terms.terms_piezo.PiezoStressTerm](#) static method), 1162
[function\(\)](#) ([sfepy.terms.terms_point.ConcentratedPointLoadTerm](#) static method), 1163
[function\(\)](#) ([sfepy.terms.terms_point.LinearPointSpringTerm](#) static method), 1164
[function\(\)](#) ([sfepy.terms.terms_shells.Shell10XTerm](#) static method), 1171
[function\(\)](#) ([sfepy.terms.terms_surface.ContactPlaneTerm](#) static method), 1172
[function\(\)](#) ([sfepy.terms.terms_surface.ContactSphereTerm](#) static method), 1174
[function\(\)](#) ([sfepy.terms.terms_surface.SDSurfaceIntegrateTerm](#) static method), 1176
[function\(\)](#) ([sfepy.terms.terms_surface.SurfaceJumpTerm](#) static method), 1177
[function\(\)](#) ([sfepy.terms.terms_volume.LinearVolumeForceTerm](#) static method), 1178
[function\(\)](#) ([sfepy.terms.terms_volume.NonlinearVolumeForceTerm](#) static method), 1178
[function_silent\(\)](#) ([sfepy.terms.terms_multilinear.ETermBase](#) static method), 1147
[function_timer\(\)](#) ([sfepy.terms.terms_multilinear.ETermBase](#) static method), 1147
[function_weak\(\)](#) ([sfepy.terms.terms_contact.ContactIPCTerm](#) static method), 1077
[function_weak\(\)](#) ([sfepy.terms.terms_contact.ContactTerm](#) static method), 1078
[Functions](#) (class in [sfepy.discrete.functions](#)), 822
G
[gels\(\)](#) (in module [sfepy.tests.test_fem](#)), 1192
[gels\(\)](#) (in module [sfepy.tests.test_poly_spaces](#)), 1197
[gels\(\)](#) (in module [sfepy.tests.test_refine_hanging](#)), 1199
[game_mulaVS3py\(\)](#) (in module [sfepy.discrete.common.extmods._geommech](#)), 821
[gen_block\(\)](#) (in module [sfepy.scripts.blockgen](#)), 1182
[gen_block_mesh\(\)](#) (in module [sfepy.mesh.mesh_generators](#)), 1001
[gen_cp_idxs\(\)](#) ([sfepy.mesh.splinebox.SplineBox](#) static method), 1006
[gen_cylinder\(\)](#) (in module [sfepy.scripts.cylindergen](#)), 1183
[gen_cylinder_mesh\(\)](#) (in module [sfepy.mesh.mesh_generators](#)), 1001
[gen_data\(\)](#) (in module [sfepy.tests.test_mesh_interp](#)), 1196
[gen_extended_block_mesh\(\)](#) (in module [sfepy.mesh.mesh_generators](#)), 1002
[gen_field_table](#) module, 1203
[gen_field_table\(\)](#) (in module [gen_field_table](#)), 1203
[gen_gallery](#) module, 1203
[gen_iga_patch\(\)](#) (in module [sfepy.scripts.gen_iga_patch](#)), 1183
[gen_legendre_simplex_base](#) module, 1205
[gen_lobatto\(\)](#) (in module [gen_lobatto1d_c](#)), 1205

gen_lobattold_c
 module, 1205
 gen_mesh_from_geom() (in module
 sfepy.mesh.mesh_generators), 1002
 gen_mesh_from_string() (in module
 sfepy.mesh.mesh_generators), 1003
 gen_mesh_from_voxels() (in module
 sfepy.mesh.mesh_generators), 1003
 gen_mesh_probe_png()
 (sfepy.postprocess.probes_vtk.Probe method),
 1013
 gen_misc_mesh() (in module
 sfepy.mesh.mesh_generators), 1003
 gen_multi_vec_packing() (in module
 sfepy.solvers.ts_solvers), 1049
 gen_patch_block_domain() (in module
 sfepy.discrete.iga.domain_generators), 935
 gen_points() (sfepy.discrete.probes.RayProbe
 method), 830
 gen_release_notes
 module, 1205
 gen_serendipity_basis
 module, 1205
 gen_shot() (in module sfepy.scripts.gen_mesh_prev),
 1183
 gen_solver_table
 module, 1205
 gen_solver_table() (in module gen_solver_table),
 1205
 gen_term_table
 module, 1206
 gen_term_table() (in module gen_term_table), 1206
 gen_tiled_mesh() (in module
 sfepy.mesh.mesh_generators), 1003
 GeneralizedAlphaTS (class in sfepy.solvers.ts_solvers),
 1045
 generate_decreasing_nonnegative_tuples_summing_to()
 (in module sfepy.discrete.simplex_cubature),
 847
 generate_gallery() (in module gen_gallery), 1204
 generate_images() (in module gen_gallery), 1204
 generate_permutations() (in module
 sfepy.discrete.simplex_cubature), 847
 generate_probes() (in module sfepy.scripts.probe),
 1185
 generate_rst_files() (in module gen_gallery), 1204
 generate_thumbnails() (in module gen_gallery),
 1204
 generate_unique_permutations() (in module
 sfepy.discrete.simplex_cubature), 847
 GenYeohTLTerm (class in
 sfepy.terms.terms_hyperelastic_tl), 1118
 geo_ctx (sfepy.discrete.fem.extmods.bases.CLagrangeContour
 attribute), 878
 geometries (sfepy.terms.terms.Term attribute), 1051
 geometries (sfepy.terms.terms_contact.ContactIPCTerm
 attribute), 1077
 geometries (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm
 attribute), 1099
 geometries (sfepy.terms.terms_elastic.ElasticWaveTerm
 attribute), 1100
 geometries (sfepy.terms.terms_elastic.LinearDSpringTerm
 attribute), 1101
 geometries (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm
 attribute), 1103
 geometries (sfepy.terms.terms_elastic.LinearSpringTerm
 attribute), 1105
 geometries (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm
 attribute), 1107
 geometries (sfepy.terms.terms_elastic.LinearTrussTerm
 attribute), 1107
 geometries (sfepy.terms.terms_elastic.NonsymElasticTerm
 attribute), 1108
 geometries (sfepy.terms.terms_elastic.SDLinearElasticTerm
 attribute), 1109
 geometries (sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm
 attribute), 1118
 geometries (sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm
 attribute), 1121
 geometries (sfepy.terms.terms_jax.NeoHookeanTLADTerm
 attribute), 1131
 geometries (sfepy.terms.terms_jax.OgdenTLADTerm
 attribute), 1131
 geometries (sfepy.terms.terms_membrane.TLMembraneTerm
 attribute), 1136
 geometries (sfepy.terms.terms_navier_stokes.StokesWaveDivTerm
 attribute), 1159
 geometries (sfepy.terms.terms_navier_stokes.StokesWaveTerm
 attribute), 1160
 geometries (sfepy.terms.terms_piezo.SDPiezoCouplingTerm
 attribute), 1163
 geometries (sfepy.terms.terms_sensitivity.ESDLinearElasticTerm
 attribute), 1167
 geometries (sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm
 attribute), 1168
 geometries (sfepy.terms.terms_shells.Shell10XTerm at-
 tribute), 1171
 geometries (sfepy.terms.terms_surface.ContactPlaneTerm
 attribute), 1172
 geometries (sfepy.terms.terms_surface.ContactSphereTerm
 attribute), 1174
 geometry (class in sfepy.mesh.geom_tools), 999
 GeometryElement (class in
 sfepy.discrete.fem.geometry_element), 891
 geomobject (class in sfepy.mesh.geom_tools), 1000
 get() (sfepy.base.base.Container method), 780
 get() (sfepy.base.base.Struct method), 782
 get() (sfepy.base.multiproc_mpi.RemoteDict method),

802
get() (*sfepy.base.multiproc_mpi.RemoteQueue* method), 802
get() (*sfepy.base.multiproc_mpi.RemoteQueueMaster* method), 803
get() (*sfepy.base.multiproc_proc.MyQueue* method), 804
get() (*sfepy.discrete.integrals.Integrals* method), 823
get() (*sfepy.mechanics.matcoefs.ElasticConstants* method), 980
get() (*sfepy.terms.terms.Term* method), 1051
get_1pk_from_2pk() (*sfepy.mechanics.tensors.StressTransform* method), 988
get_2d_points() (in module *sfepy.mesh.bspline*), 998
get_a0() (*sfepy.solvers.ts_solvers.ElastodynamicsBaseTS* method), 1045
get_AABB() (in module *sfepy.mechanics.extmods.ccontres*), 993
get_actual_cache() (*sfepy.discrete.probes.Probe* method), 829
get_actual_order() (in module *sfepy.discrete.quadratures*), 846
get_arg_kinds() (in module *sfepy.terms.terms*), 1054
get_arg_name() (*sfepy.terms.terms.Term* method), 1051
get_args() (*sfepy.terms.terms.Term* method), 1052
get_args_by_name() (*sfepy.terms.terms.Term* method), 1052
get_arguments() (in module *sfepy.base.base*), 784
get_assembling_cells() (*sfepy.terms.terms.Term* method), 1052
get_bc_facet_idx() (*sfepy.discrete.dg.fields.DGField* method), 921
get_bc_facet_values() (*sfepy.discrete.dg.fields.DGField* method), 921
get_bezier_element_entities() (in module *sfepy.discrete.iga.iga*), 944
get_bezier_topology() (in module *sfepy.discrete.iga.iga*), 945
get_bf() (*sfepy.terms.terms_multilinear.ExpressionArg* method), 1148
get_blocks_stats() (in module *sfepy.linalg.utils*), 976
get_both_facet_basis_vals() (*sfepy.discrete.dg.fields.DGField* method), 921
get_both_facet_state_vals() (*sfepy.discrete.dg.fields.DGField* method), 922
get_bounding_box() (*sfepy.discrete.fem.mesh.Mesh* method), 898
get_box_matrix() (*sfepy.mesh.splinebox.SplineBox* method), 1006
get_box_volume() (in module *sfepy.homogenization.utils*), 968
get_callback() (in module *sfepy.homogenization.coefs_phononic*), 959
get_camera_position() (in module *sfepy.scripts.resview*), 1187
get_cauchy_from_2pk() (*sfepy.mechanics.tensors.StressTransform* method), 988
get_cauchy_strain() (in module *sfepy.mechanics.tensors*), 988
get_cell_conn() (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 862
get_cell_indices() (*sfepy.discrete.common.region.Region* method), 875
get_cell_normals_per_facet() (*sfepy.discrete.dg.fields.DGField* method), 922
get_cells() (in module *sfepy.tests.test_regions*), 1199
get_cells() (*sfepy.discrete.common.region.Region* method), 875
get_centroids() (*sfepy.discrete.common.domain.Domain* method), 860
get_centroids() (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 862
get_charfun() (*sfepy.discrete.common.region.Region* method), 875
get_circle() (in module *sfepy.tests.test_functions*), 1193
get_cmem_usage() (in module *sfepy.discrete.common.extmods.cmesh*), 863
get_cmesh() (*sfepy.discrete.fem.mesh.Mesh* method), 899
get_coef() (*sfepy.homogenization.coefs_base.CoeffMN* method), 952
get_coef() (*sfepy.homogenization.coefs_base.CoeffN* method), 952
get_coefs() (*sfepy.homogenization.coefs_phononic.AcousticMassLiquidT* method), 954
get_coefs() (*sfepy.homogenization.coefs_phononic.AcousticMassTensor* method), 955
get_complete() (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 862
get_composite_sizes() (in module *sfepy.parallel.parallel*), 1009
get_condition_value() (in module *sfepy.discrete.conditions*), 810
get_conn() (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 862
get_conn() (*sfepy.discrete.fem.domain.FEDomain* method), 877
get_conn() (*sfepy.discrete.fem.mesh.Mesh* method), 899
get_conn_as_graph() (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 863

[get_conn_info\(\)](#) (*sfepy.terms.terms.Term* method), 1052
[get_conn_key\(\)](#) (*sfepy.terms.terms.Term* method), 1052
[get_conn_permutations\(\)](#) (*sfepy.discrete.fem.geometry_element.GeometryElement* method), 891
[get_connectivity\(\)](#) (*sfepy.discrete.fem.fe_surface.FESurface* method), 880
[get_consistent_unit_set\(\)](#) (in module *sfepy.mechanics.units*), 993
[get_constant_data\(\)](#) (*sfepy.discrete.materials.Material* method), 824
[get_contact_info\(\)](#) (*sfepy.terms.terms_contact.ContactTerm* method), 1077
[get_contact_info\(\)](#) (*sfepy.terms.terms_contact.ContactTerm* method), 1078
[get_control_points\(\)](#) (*sfepy.mesh.bspline.BSpline* method), 995
[get_control_points\(\)](#) (*sfepy.mesh.bspline.BSplineSurf* method), 997
[get_control_points\(\)](#) (*sfepy.mesh.splinebox.SplineBox* method), 1006
[get_coor\(\)](#) (*sfepy.discrete.dg.fields.DGField* method), 922
[get_coor\(\)](#) (*sfepy.discrete.fem.fields_base.FEField* method), 882
[get_coor\(\)](#) (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* method), 887
[get_coors_in_ball\(\)](#) (in module *sfepy.linalg.geometry*), 970
[get_coors_in_tube\(\)](#) (in module *sfepy.linalg.geometry*), 971
[get_coors_shape\(\)](#) (*sfepy.mesh.splinebox.SplineBox* method), 1006
[get_correctors_from_file_hdf5\(\)](#) (in module *sfepy.homogenization.micmac*), 964
[get_data\(\)](#) (*sfepy.discrete.materials.Material* method), 824
[get_data_name\(\)](#) (in module *sfepy.discrete.probes*), 831
[get_data_shape\(\)](#) (*sfepy.discrete.dg.fields.DGField* method), 923
[get_data_shape\(\)](#) (*sfepy.discrete.fem.fields_base.FEField* method), 882
[get_data_shape\(\)](#) (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* method), 887
[get_data_shape\(\)](#) (*sfepy.discrete.iga.fields.IGField* method), 939
[get_data_shape\(\)](#) (*sfepy.discrete.variables.FieldVariable* method), 849
[get_data_shape\(\)](#) (*sfepy.terms.terms.Term* method), 1052
[get_debug\(\)](#) (in module *sfepy.base.base*), 784
[get_default\(\)](#) (in module *sfepy.base.base*), 784
[get_default_attr\(\)](#) (in module *sfepy.base.base*), 784
[get_default_time_step\(\)](#) (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
[get_default_ts\(\)](#) (*sfepy.discrete.problem.Problem* method), 836
[get_dependency_graph\(\)](#) (in module *sfepy.discrete.common.region*), 876
[get_deviator\(\)](#) (in module *sfepy.mechanics.tensors*), 989
[get_diameter\(\)](#) (*sfepy.discrete.fem.domain.FEDomain* method), 877
[get_dict\(\)](#) (in module *sfepy.base.multiproc_mpi*), 803
[get_dict\(\)](#) (in module *sfepy.base.multiproc_proc*), 804
[get_dict_idxval\(\)](#) (in module *sfepy.homogenization.engine*), 963
[get_dim\(\)](#) (*sfepy.discrete.problem.Problem* method), 836
[get_distance\(\)](#) (*sfepy.mechanics.contact_bodies.ContactPlane* method), 979
[get_distance\(\)](#) (*sfepy.mechanics.contact_bodies.ContactSphere* method), 979
[get_dof_conn\(\)](#) (*sfepy.discrete.variables.FieldVariable* method), 850
[get_dof_conn_type\(\)](#) (*sfepy.terms.terms.Term* method), 1052
[get_dof_info\(\)](#) (*sfepy.discrete.variables.FieldVariable* method), 850
[get_dofs\(\)](#) (*sfepy.terms.terms_multilinear.ExpressionArg* method), 1148
[get_dofs_in_region\(\)](#) (*sfepy.discrete.dg.fields.DGField* method), 923
[get_dofs_in_region\(\)](#) (*sfepy.discrete.fem.fields_base.FEField* method), 883
[get_dofs_in_region\(\)](#) (*sfepy.discrete.fem.fields_l2.L2ConstantVolumeField* method), 888
[get_dofs_in_region\(\)](#) (*sfepy.discrete.iga.fields.IGField* method), 939
[get_domain\(\)](#) (*sfepy.discrete.equations.Equations* method), 815
[get_domain_size\(\)](#) (in module *sfepy.mechanics.shell10x*), 987
[get_dts\(\)](#) (*sfepy.base.timing.Timers* method), 807
[get_dual\(\)](#) (*sfepy.discrete.variables.Variable* method), 852
[get_dual_names\(\)](#) (*sfepy.discrete.variables.Variables* method), 849

method), 854
get_ebc_indices() (sfepy.discrete.problem.Problem method), 836
get_econn() (sfepy.discrete.dg.fields.DGField method), 923
get_econn() (sfepy.discrete.fem.fields_base.FEField method), 883
get_econn() (sfepy.discrete.fem.fields_l2.L2ConstantVolumeField method), 888
get_econn() (sfepy.discrete.iga.fields.IGField method), 940
get_edge_graph() (sfepy.discrete.common.region.Region method), 876
get_edge_paths() (in module sfepy.discrete.fem.utils), 913
get_edges_per_face() (sfepy.discrete.fem.geometry_element.GeometryElement method), 891
get_einsum_ops() (in module sfepy.terms.terms_multilinear), 1151
get_element_diameters() (sfepy.discrete.fem.domain.FEDomain method), 877
get_element_diameters() (sfepy.discrete.variables.FieldVariable method), 850
get_entities() (sfepy.discrete.common.region.Region method), 876
get_eth_data() (sfepy.terms.terms_th.ETHTerm method), 1177
get_eval_coors() (in module sfepy.discrete.fem.linearizer), 895
get_eval_dofs() (in module sfepy.discrete.fem.linearizer), 895
get_eval_expression() (in module sfepy.discrete.fem.fields_base), 887
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1056
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1057
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1058
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1059
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1059
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1060
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1060
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1061
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1062
get_eval_shape() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStab method), 1062
get_eval_shape() (sfepy.terms.terms_basic.IntegrateMatTerm method), 1065
get_eval_shape() (sfepy.terms.terms_basic.IntegrateTerm method), 1066
get_eval_shape() (sfepy.terms.terms_basic.SumNodalValuesTerm method), 1067
get_eval_shape() (sfepy.terms.terms_basic.SurfaceMomentTerm method), 1067
get_eval_shape() (sfepy.terms.terms_basic.VolumeSurfaceTerm method), 1068
get_eval_shape() (sfepy.terms.terms_basic.VolumeTerm method), 1068
get_eval_shape() (sfepy.terms.terms_biot.BiotTerm method), 1072
get_eval_shape() (sfepy.terms.terms_contact.ContactTerm method), 1078
get_eval_shape() (sfepy.terms.terms_diffusion.DiffusionCoupling method), 1085
get_eval_shape() (sfepy.terms.terms_diffusion.DiffusionTerm method), 1086
get_eval_shape() (sfepy.terms.terms_diffusion.DiffusionVelocityTerm method), 1087
get_eval_shape() (sfepy.terms.terms_diffusion.SDDiffusionTerm method), 1089
get_eval_shape() (sfepy.terms.terms_diffusion.SurfaceFluxTerm method), 1090
get_eval_shape() (sfepy.terms.terms_dot.DotProductTerm method), 1092
get_eval_shape() (sfepy.terms.terms_dot.VectorDotGradScalarTerm method), 1095
get_eval_shape() (sfepy.terms.terms_dot.VectorDotScalarTerm method), 1096
get_eval_shape() (sfepy.terms.terms_elastic.CauchyStrainTerm method), 1097
get_eval_shape() (sfepy.terms.terms_elastic.CauchyStressETHTerm method), 1097
get_eval_shape() (sfepy.terms.terms_elastic.CauchyStressTerm method), 1099
get_eval_shape() (sfepy.terms.terms_elastic.CauchyStressTHTerm method), 1098
get_eval_shape() (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm method), 1103
get_eval_shape() (sfepy.terms.terms_elastic.LinearElasticTerm method), 1104
get_eval_shape() (sfepy.terms.terms_elastic.LinearPrestressTerm method), 1105
get_eval_shape() (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm method), 1107
get_eval_shape() (sfepy.terms.terms_elastic.NonsymElasticTerm method), 1108
get_eval_shape() (sfepy.terms.terms_elastic.SDLinearElasticTerm method), 1109

`get_eval_shape()` (*sfepy.terms.terms_fibres.FibresActiveTLTerm* method), 836
`method`), 1110 `get_examples()` (in module *gen_term_table*), 1206
`get_eval_shape()` (*sfepy.terms.terms_fibres.FibresExponentialTerm* method), 1111 `get_examples()` (*sfepy.homogenization.convolution.ConvolutionKernel* method), 960
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBaseTerm* method), 1114 `get_examples()` (in module *sfepy.discrete.equations*), 818
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBaseTerm* method), 1115 `get_examples()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_tl.BulkPressureTerm* method), 1117 `get_examples()` (in module *sfepy.linalg.geometry*), 971
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_tl.DiffusionTerm* method), 1117 `get_examples()` (in module *sfepy.discrete.iga.iga*), 945
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTerm* method), 1121 `get_examples()` (*sfepy.discrete.dg.fields.DGField* method), 923
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_tl.VolumeTractionTerm* method), 1123 `get_examples()` (in module *sfepy.discrete.fem.facets*), 879
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_tl.VolumeTractionTerm* method), 1123 `get_examples()` (*sfepy.discrete.common.region.Region* method), 876
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_ul.BulkPressureTerm* method), 1125 `get_examples()` (*sfepy.discrete.dg.fields.DGField* method), 924
`get_eval_shape()` (*sfepy.terms.terms_hyperelastic_ul.VolumeTractionTerm* method), 1128 `get_examples()` (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 863
`get_eval_shape()` (*sfepy.terms.terms_membrane.TLMembraneTerm* method), 1136 `get_examples()` (*sfepy.discrete.dg.fields.DGField* method), 924
`get_eval_shape()` (*sfepy.terms.terms_multilinear.ETermBase* method), 1147 `get_examples()` (*sfepy.discrete.dg.fields.DGField* method), 924
`get_eval_shape()` (*sfepy.terms.terms_navier_stokes.DivGradTerm* method), 1153 `get_examples()` (*sfepy.discrete.dg.fields.DGField* method), 924
`get_eval_shape()` (*sfepy.terms.terms_navier_stokes.DivTerm* method), 1154 `get_examples()` (*sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTerm* attribute), 1118
`get_eval_shape()` (*sfepy.terms.terms_navier_stokes.GradTerm* attribute), 1155 `get_examples()` (*sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBaseTerm* attribute), 1119
`get_eval_shape()` (*sfepy.terms.terms_navier_stokes.StokesTerm* attribute), 1159 `get_examples()` (*sfepy.terms.terms_hyperelastic_ul.HyperElasticULBaseTerm* attribute), 1126
`get_eval_shape()` (*sfepy.terms.terms_piezo.PiezoCouplingTerm* attribute), 1161 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term* method), 1055
`get_eval_shape()` (*sfepy.terms.terms_piezo.PiezoStrainTerm* method), 1161 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term* method), 1056
`get_eval_shape()` (*sfepy.terms.terms_piezo.PiezoStressTerm* method), 1162 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm* method), 1056
`get_eval_shape()` (*sfepy.terms.terms_surface.LinearTractionTerm* method), 1174 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm* method), 1056
`get_eval_shape()` (*sfepy.terms.terms_surface.SDLinearTractionTerm* method), 1175 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm* method), 1057
`get_eval_shape()` (*sfepy.terms.terms_surface.SDSurfaceIntegrateTerm* method), 1176 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm* method), 1057
`get_eval_shape()` (*sfepy.terms.terms_surface.SurfaceNormalDotTerm* method), 1176 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.SDConvectTerm* method), 1058
`get_evaluate_cache()` (*sfepy.discrete.fem.fields_base.FEField* method), 883 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm* method), 1059
`get_evaluate_cache()` (*sfepy.discrete.probes.Probe* method), 829 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.SDDivTerm* method), 1059
`get_evaluator()` (*sfepy.discrete.problem.Problem* method), 829 `get_examples()` (*sfepy.terms.terms_adj_navier_stokes.SDDotTerm* method), 1059

method), 1060

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SDGradDivTerm method), 1060

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SDPSPGetFargs() method), 1061

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SDPSPGetFargs() method), 1062

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SDSUPGetFargs() method), 1062

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SUPGetFargs() method), 1063

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SUPGetFargs() method), 1064

get_fargs() (sfepy.terms.terms_adj_navier_stokes.SUPGetFargs() method), 1064

get_fargs() (sfepy.terms.terms_basic.IntegrateMatTerm method), 1065

get_fargs() (sfepy.terms.terms_basic.IntegrateOperatorTerm method), 1066

get_fargs() (sfepy.terms.terms_basic.IntegrateTerm method), 1066

get_fargs() (sfepy.terms.terms_basic.SumNodalValuesTerm method), 1067

get_fargs() (sfepy.terms.terms_basic.SurfaceMomentTerm method), 1067

get_fargs() (sfepy.terms.terms_basic.VolumeSurfaceTerm method), 1068

get_fargs() (sfepy.terms.terms_basic.VolumeTerm method), 1068

get_fargs() (sfepy.terms.terms_basic.ZeroTerm method), 1069

get_fargs() (sfepy.terms.terms_biot.BiotETHTerm method), 1070

get_fargs() (sfepy.terms.terms_biot.BiotStressTerm method), 1070

get_fargs() (sfepy.terms.terms_biot.BiotTerm method), 1072

get_fargs() (sfepy.terms.terms_biot.BiotTHTerm method), 1071

get_fargs() (sfepy.terms.terms_constraints.NonPenetrationTerm method), 1075

get_fargs() (sfepy.terms.terms_constraints.NonPenetrationTerm method), 1076

get_fargs() (sfepy.terms.terms_contact.ContactIPCTerm method), 1077

get_fargs() (sfepy.terms.terms_contact.ContactTerm method), 1078

get_fargs() (sfepy.terms.terms_dg.AdvectionDGFluxTerm method), 1080

get_fargs() (sfepy.terms.terms_dg.DiffusionDGFluxTerm method), 1081

get_fargs() (sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm method), 1082

get_fargs() (sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm method), 1083

get_fargs() (sfepy.terms.terms_dg.NonlinearScalarDotGradTerm method), 1084

get_fargs() (sfepy.terms.terms_diffusion.ConvectVGradTerm method), 1085

get_fargs() (sfepy.terms.terms_diffusion.DiffusionCoupling method), 1085

get_fargs() (sfepy.terms.terms_diffusion.DiffusionRTerm method), 1086

get_fargs() (sfepy.terms.terms_diffusion.DiffusionTerm method), 1086

get_fargs() (sfepy.terms.terms_diffusion.DiffusionVelocityTerm method), 1087

get_fargs() (sfepy.terms.terms_diffusion.NonlinearDiffusionTerm method), 1088

get_fargs() (sfepy.terms.terms_diffusion.SDDiffusionTerm method), 1089

get_fargs() (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm method), 1090

get_fargs() (sfepy.terms.terms_diffusion.SurfaceFluxTerm method), 1090

get_fargs() (sfepy.terms.terms_dot.BCNewtonTerm method), 1091

get_fargs() (sfepy.terms.terms_dot.DotProductTerm method), 1092

get_fargs() (sfepy.terms.terms_dot.DotSPProductVolumeOperatorWETHTerm method), 1093

get_fargs() (sfepy.terms.terms_dot.DotSPProductVolumeOperatorWHTTerm method), 1093

get_fargs() (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm method), 1094

get_fargs() (sfepy.terms.terms_dot.ScalarDotMGradScalarTerm method), 1094

get_fargs() (sfepy.terms.terms_dot.VectorDotGradScalarTerm method), 1095

get_fargs() (sfepy.terms.terms_dot.VectorDotScalarTerm method), 1096

get_fargs() (sfepy.terms.terms_elastic.CauchyStrainTerm method), 1097

get_fargs() (sfepy.terms.terms_elastic.CauchyStressETHTerm method), 1097

get_fargs() (sfepy.terms.terms_elastic.CauchyStressTerm method), 1099

get_fargs() (sfepy.terms.terms_elastic.CauchyStressTHTerm method), 1098

get_fargs() (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm method), 1099

get_fargs() (sfepy.terms.terms_elastic.ElasticWaveTerm method), 1100

get_fargs() (sfepy.terms.terms_elastic.LinearDSpringTerm method), 1101

get_fargs() (sfepy.terms.terms_elastic.LinearElasticETHTerm method), 1102

get_fargs() (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm method), 1102

method), 1103

get_fargs() (sfepy.terms.terms_elastic.LinearElasticTerm method), 1104

get_fargs() (sfepy.terms.terms_elastic.LinearElasticTHTerm method), 1103

get_fargs() (sfepy.terms.terms_elastic.LinearPrestressTerm method), 1105

get_fargs() (sfepy.terms.terms_elastic.LinearSpringTerm method), 1105

get_fargs() (sfepy.terms.terms_elastic.LinearStrainFiberTerm method), 1106

get_fargs() (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm method), 1107

get_fargs() (sfepy.terms.terms_elastic.LinearTrussTerm method), 1107

get_fargs() (sfepy.terms.terms_elastic.NonsymElasticTerm method), 1108

get_fargs() (sfepy.terms.terms_elastic.SDLinearElasticTerm method), 1109

get_fargs() (sfepy.terms.terms_electric.ElectricSourceTerm method), 1109

get_fargs() (sfepy.terms.terms_fibres.FibresActiveTLTerm method), 1110

get_fargs() (sfepy.terms.terms_fibres.FibresExponentialTLTerm method), 1111

get_fargs() (sfepy.terms.terms_hyperelastic_base.DeformationFiberTerm method), 1114

get_fargs() (sfepy.terms.terms_hyperelastic_base.HyperElasticTerm method), 1115

get_fargs() (sfepy.terms.terms_hyperelastic_tl.BulkPressureTerm method), 1117

get_fargs() (sfepy.terms.terms_hyperelastic_tl.DiffusionTerm method), 1117

get_fargs() (sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTerm method), 1121

get_fargs() (sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTerm method), 1122

get_fargs() (sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTerm method), 1123

get_fargs() (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm method), 1123

get_fargs() (sfepy.terms.terms_hyperelastic_ul.BulkPressureTerm method), 1125

get_fargs() (sfepy.terms.terms_hyperelastic_ul.CompressionTerm method), 1125

get_fargs() (sfepy.terms.terms_hyperelastic_ul.HyperelasticByFiberTerm method), 1126

get_fargs() (sfepy.terms.terms_hyperelastic_ul.VolumeUllTerm method), 1128

get_fargs() (sfepy.terms.terms_jax.LinearElasticLADTerm method), 1129

get_fargs() (sfepy.terms.terms_jax.LinearElasticYPADTerm method), 1129

get_fargs() (sfepy.terms.terms_jax.MassADTerm method), 1130

get_fargs() (sfepy.terms.terms_jax.NeoHookeanTLADTerm method), 1131

get_fargs() (sfepy.terms.terms_jax.OgdenTLADTerm method), 1132

get_fargs() (sfepy.terms.terms_membrane.TLMembraneTerm method), 1136

get_fargs() (sfepy.terms.terms_multilinear.ETermBase method), 1147

get_fargs() (sfepy.terms.terms_navier_stokes.ConvectTerm method), 1152

get_fargs() (sfepy.terms.terms_navier_stokes.DivGradTerm method), 1153

get_fargs() (sfepy.terms.terms_navier_stokes.DivOperatorTerm method), 1153

get_fargs() (sfepy.terms.terms_navier_stokes.DivTerm method), 1154

get_fargs() (sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm method), 1154

get_fargs() (sfepy.terms.terms_navier_stokes.GradTerm method), 1155

get_fargs() (sfepy.terms.terms_navier_stokes.LinearConvect2Term method), 1155

get_fargs() (sfepy.terms.terms_navier_stokes.LinearConvectTerm method), 1156

get_fargs() (sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm method), 1156

get_fargs() (sfepy.terms.terms_navier_stokes.StokesTerm method), 1159

get_fargs() (sfepy.terms.terms_navier_stokes.StokesWaveDivTerm method), 1160

get_fargs() (sfepy.terms.terms_navier_stokes.StokesWaveTerm method), 1160

get_fargs() (sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm method), 1157

get_fargs() (sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm method), 1158

get_fargs() (sfepy.terms.terms_piezo.PiezoCouplingTerm method), 1161

get_fargs() (sfepy.terms.terms_piezo.PiezoStrainTerm method), 1161

get_fargs() (sfepy.terms.terms_piezo.PiezoStressTerm method), 1162

get_fargs() (sfepy.terms.terms_point.ConcentratedPointLoadTerm method), 1163

get_fargs() (sfepy.terms.terms_point.LinearPointSpringTerm method), 1164

get_fargs() (sfepy.terms.terms_shells.Shell10XTerm method), 1171

get_fargs() (sfepy.terms.terms_surface.ContactPlaneTerm method), 1173

get_fargs() (sfepy.terms.terms_surface.ContactSphereTerm method), 1174

get_fargs() (sfepy.terms.terms_surface.LinearTractionTerm method), 1174

method), 1174

get_fargs() (sfepy.terms.terms_surface.SDLinearTractionTerm method), 1175

get_fargs() (sfepy.terms.terms_surface.SDSurfaceIntegralTerm method), 1176

get_fargs() (sfepy.terms.terms_surface.SurfaceNormalDotTerm method), 1176

get_fargs() (sfepy.terms.terms_surface.SurfaceJumpTerm method), 1177

get_fargs() (sfepy.terms.terms_volume.LinearVolumeForceTerm method), 1178

get_fargs() (sfepy.terms.terms_volume.NonlinearVolumeForceTerm method), 1178

get_field() (sfepy.discrete.variables.FieldVariable method), 850

get_filename_trunk() (sfepy.discrete.fem.meshio.MeshIO method), 904

get_filename_trunk() (sfepy.discrete.fem.meshio.UserMeshIO method), 906

get_full() (sfepy.discrete.variables.DGFieldVariable method), 847

get_full() (sfepy.discrete.variables.FieldVariable method), 850

get_full() (sfepy.homogenization.convolution.ConvolutionKernel method), 960

get_full_indices() (in module sfepy.mechanics.tensors), 989

get_function() (sfepy.base.conf.ProblemConf method), 791

get_function() (sfepy.terms.terms_flexo.MixedFlexoCouplingTerm method), 1112

get_function() (sfepy.terms.terms_flexo.MixedFlexoTerm method), 1113

get_function() (sfepy.terms.terms_flexo.MixedStrainGradientElasticTerm method), 1113

get_function() (sfepy.terms.terms_mass.MassTerm method), 1135

get_function() (sfepy.terms.terms_multilinear.ECauchyStressTerm method), 1137

get_function() (sfepy.terms.terms_multilinear.EConvectTerm method), 1137

get_function() (sfepy.terms.terms_multilinear.EDiffusionTerm method), 1138

get_function() (sfepy.terms.terms_multilinear.EDivGradientTerm method), 1139

get_function() (sfepy.terms.terms_multilinear.EDivTerm method), 1139

get_function() (sfepy.terms.terms_multilinear.EDotTerm method), 1140

get_function() (sfepy.terms.terms_multilinear.EGradTerm method), 1140

get_function() (sfepy.terms.terms_multilinear.EIntegrateOperator method), 1141

get_function() (sfepy.terms.terms_multilinear.ELaplaceTerm method), 1141

get_function() (sfepy.terms.terms_multilinear.ELinearConvectTerm method), 1142

get_function() (sfepy.terms.terms_multilinear.ELinearElasticTerm method), 1143

get_function() (sfepy.terms.terms_multilinear.ELinearTractionTerm method), 1143

get_function() (sfepy.terms.terms_multilinear.ENonPenetrationPenaltyTerm method), 1144

get_function() (sfepy.terms.terms_multilinear.ENonSymElasticTerm method), 1144

get_function() (sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm method), 1145

get_function() (sfepy.terms.terms_multilinear.EStokesTerm method), 1146

get_function() (sfepy.terms.terms_multilinear.StokesTractionTerm method), 1149

get_function() (sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm method), 1150

get_function() (sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm method), 1150

get_function() (sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm method), 1151

get_function() (sfepy.terms.terms_piezo.SDPiezoCouplingTerm method), 1163

get_function() (sfepy.terms.terms_sensitivity.ESDDiffusionTerm method), 1164

get_function() (sfepy.terms.terms_sensitivity.ESDDivGradTerm method), 1165

get_function() (sfepy.terms.terms_sensitivity.ESDDotTerm method), 1166

get_function() (sfepy.terms.terms_sensitivity.ESDLinearElasticTerm method), 1167

get_function() (sfepy.terms.terms_sensitivity.ESDLinearTractionTerm method), 1167

get_function() (sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm method), 1168

get_function() (sfepy.terms.terms_sensitivity.ESDStokesTerm method), 1169

get_gap_ranges() (in module sfepy.homogenization.coefs_phononic), 959

get_predict_key() (sfepy.base.multiproc_mpi.RemoteQueueMaster static method), 803

get_rangel() (in module sfepy.discrete.dg.fields), 927

get_geometry() (sfepy.discrete.fem.mappings.FEMapping method), 896

get_geometry() (sfepy.discrete.iga.mappings.IGMapping method), 947

get_graph_conns() (sfepy.discrete.equations.Equations method), 815

get_green_strain_sym3d() (in module sfepy.mechanics.membranes), 984

[get_grid\(\)](#) (*sfepy.discrete.fem.geometry_element.GeometryElement method*), 891
[get_grid_plane\(\)](#) (in module *sfepy.discrete.fem.periodic*), 907
[get_homog_coefs_linear\(\)](#) (in module *sfepy.homogenization.micmac*), 964
[get_homog_coefs_nonlinear\(\)](#) (in module *sfepy.homogenization.micmac*), 964
[get_incident\(\)](#) (*sfepy.discrete.common.extmods.cmesh.CMesh method*), 863
[get_indx\(\)](#) (*sfepy.discrete.variables.Variables method*), 854
[get_info\(\)](#) (*sfepy.discrete.common.dof_info.DofInfo method*), 858
[get_initial_condition\(\)](#) (*sfepy.discrete.variables.Variable method*), 852
[get_initial_dt\(\)](#) (*sfepy.solvers.solvers.TimeStepController method*), 1037
[get_initial_dt\(\)](#) (*sfepy.solvers.ts_controllers.ElastodynamicTS method*), 1039
[get_initial_dt\(\)](#) (*sfepy.solvers.ts_controllers.TimesSequenceTSC method*), 1042
[get_initial_state\(\)](#) (*sfepy.discrete.problem.Problem method*), 837
[get_initial_vec\(\)](#) (*sfepy.solvers.ts_solvers.ElastodynamicTS method*), 1045
[get_int_value\(\)](#) (in module *sfepy.base.multiproc_mpi*), 803
[get_int_value\(\)](#) (in module *sfepy.base.multiproc_proc*), 804
[get_integrals\(\)](#) (*sfepy.discrete.problem.Problem method*), 837
[get_inter_facets\(\)](#) (in module *sfepy.parallel.parallel*), 1009
[get_interp_coors\(\)](#) (*sfepy.discrete.variables.FieldVariable method*), 850
[get_interpol_scheme\(\)](#) (*sfepy.discrete.dg.poly_spaces.LegendrePolySpace method*), 928
[get_interpolation_name\(\)](#) (*sfepy.discrete.fem.geometry_element.GeometryElement method*), 891
[get_invariants\(\)](#) (in module *sfepy.mechanics.membranes*), 985
[get_item_by_name\(\)](#) (*sfepy.base.conf.ProblemConf method*), 791
[get_jacobian\(\)](#) (in module *sfepy.discrete.common.mappings*), 871
[get_keys\(\)](#) (*sfepy.discrete.materials.Material method*), 824
[get_knot_vector\(\)](#) (*sfepy.mesh.bspline.BSpline method*), 995
[get_kwargs\(\)](#) (*sfepy.terms.terms.Term method*), 1052
[get_lattice_volume\(\)](#) (in module *sfepy.homogenization.utils*), 968
[get_lcbc_operator\(\)](#) (*sfepy.discrete.equations.Equations method*), 815
[get_lcbc_operator\(\)](#) (*sfepy.discrete.variables.Variables method*), 854
[get_lgl_nodes\(\)](#) (in module *sfepy.discrete.fem.poly_spaces*), 911
[get_list\(\)](#) (in module *sfepy.base.multiproc_proc*), 804
[get_local_entities\(\)](#) (*sfepy.discrete.common.extmods.cmesh.CMesh method*), 863
[get_local_ids\(\)](#) (*sfepy.discrete.common.extmods.cmesh.CMesh method*), 863
[get_local_ordering\(\)](#) (in module *sfepy.parallel.parallel*), 1009
[get_log_freqs\(\)](#) (in module *sfepy.base.multiproc_proc*), 804
[get_log_freqs\(\)](#) (in module *sfepy.homogenization.coefs_phononic*), 959
[get_log_name\(\)](#) (*sfepy.base.log.Log method*), 798
[get_logger\(\)](#) (in module *sfepy.base.multiproc_mpi*), 803
[get_logging_conf\(\)](#) (in module *sfepy.base.log*), 799
[get_lowest_edge_and_gps\(\)](#) (in module *sfepy.mechanics.extmods.ccontres*), 993
[get_loop_indices\(\)](#) (in module *sfepy.terms.terms_multilinear*), 1151
[get_ls\(\)](#) (*sfepy.discrete.problem.Problem method*), 837
[get_manager\(\)](#) (in module *sfepy.base.multiproc_proc*), 804
[get_mapping\(\)](#) (*sfepy.discrete.common.fields.Field method*), 866
[get_mapping\(\)](#) (*sfepy.discrete.fem.mappings.FEMapping method*), 896
[get_mapping\(\)](#) (*sfepy.discrete.iga.mappings.IGMapping method*), 947
[get_mapping\(\)](#) (*sfepy.discrete.structural.mappings.Shell10XMapping method*), 950
[get_mapping\(\)](#) (*sfepy.discrete.variables.FieldVariable method*), 850
[get_mapping\(\)](#) (*sfepy.terms.terms.Term method*), 1052
[get_mapping_data\(\)](#) (in module *sfepy.discrete.common.mappings*), 871
[get_mapping_data\(\)](#) (in module *sfepy.mechanics.shell10x*), 987
[get_maps\(\)](#) (*sfepy.solvers.oseen.StabilizationFunction method*), 1034
[get_material_names\(\)](#) (*sfepy.terms.terms.Term method*), 1052
[get_material_names\(\)](#) (*sfepy.terms.terms.Terms method*), 1054

`get_materials()` (*sfepy.discrete.problem.Problem* method), 837
`get_materials()` (*sfepy.terms.terms.Term* method), 1052
`get_matrices()` (*sfepy.solvers.ts_solvers.ElastodynamicsByTensors* method), 1045
`get_matrix_shape()` (*sfepy.discrete.variables.Variables* method), 854
`get_mem_usage()` (in module *sfepy.base.mem_usage*), 800
`get_mesh_bounding_box()` (*sfepy.discrete.fem.domain.FEDomain* method), 877
`get_mesh_coors()` (*sfepy.discrete.fem.domain.FEDomain* method), 877
`get_mesh_coors()` (*sfepy.discrete.problem.Problem* method), 837
`get_meshes_from_region()` (*sfepy.discrete.problem.Problem* method), 837
`get_micro_cache_key()` (*sfepy.homogenization.homogen_app.HomogenizationApp* method), 963
`get_min_dt()` (in module *sfepy.solvers.ts_solvers*), 1049
`get_min_value()` (in module *sfepy.discrete.fem.utils*), 913
`get_min_vertex_distance()` (in module *sfepy.discrete.fem.mesh*), 899
`get_min_vertex_distance_naive()` (in module *sfepy.discrete.fem.mesh*), 899
`get_mirror_region()` (*sfepy.discrete.common.region.Region* method), 876
`get_mpdict_value()` (in module *sfepy.base.multiproc_proc*), 804
`get_mtx_i()` (*sfepy.discrete.fem.poly_spaces.FEPolySpace* method), 907
`get_mtx_i()` (*sfepy.discrete.fem.poly_spaces.LagrangeTensorProductPolySpace* method), 908
`get_mtx_t_and_length()` (*sfepy.terms.terms_elastic.LinearTrussTerm* static method), 1107
`get_multiproc()` (in module *sfepy.base.multiproc*), 801
`get_n_cells()` (*sfepy.discrete.common.region.Region* method), 876
`get_n_dof_total()` (*sfepy.discrete.common.dof_info.DofInfo* method), 858
`get_n_el_nod()` (in module *sfepy.discrete.dg.poly_spaces*), 930
`get_names()` (*sfepy.base.base.Container* method), 780
`get_names()` (*sfepy.base.base.OneTypeList* method), 781
`get_neohook_strain_energy()` (in module *sfepy.terms.terms_jax*), 1134
`get_neohook_strain_energy_f()` (in module *sfepy.terms.terms_jax*), 1134
`get_neohook_stress_1pk()` (in module *sfepy.terms.terms_jax*), 1134
`get_neohook_stress_2pk()` (in module *sfepy.terms.terms_jax*), 1134
`get_nls()` (*sfepy.discrete.problem.Problem* method), 837
`get_nls_functions()` (*sfepy.discrete.problem.Problem* method), 837
`get_nodal_values()` (*sfepy.discrete.dg.fields.DGField* method), 925
`get_non_diagonal_indices()` (in module *sfepy.mechanics.tensors*), 989
`get_nonsym_grad_op()` (in module *sfepy.terms.terms_sensitivity*), 1170
`get_nonzero_norm()` (in module *sfepy.scripts.resview*), 1187
`get_normals()` (in module *sfepy.discrete.common.mappings*), 872
`get_normals()` (*sfepy.terms.terms_multilinear.ETermBase* method), 1147
`get_nth_fun()` (*sfepy.discrete.dg.poly_spaces.LegendrePolySpace* method), 928
`get_nth_fun_der()` (*sfepy.discrete.dg.poly_spaces.LegendrePolySpace* method), 928
`get_num_workers()` (in module *sfepy.base.multiproc*), 801
`get_nums()` (in module *sfepy.base.resolve_deps*), 806
`get_ogden_strain_energy_f()` (in module *sfepy.terms.terms_jax*), 1134
`get_ogden_stress_1pk()` (in module *sfepy.terms.terms_jax*), 1134
`get_operands()` (*sfepy.terms.terms_multilinear.ETermBase* method), 1147
`get_operator()` (*sfepy.discrete.common.dof_info.EquationMap* method), 858
`get_or_create_hdf5_group()` (in module *sfepy.base.ioutils*), 795
`get_orientations()` (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 863
`get_ortho_d()` (in module *sfepy.tests.test_tensors*), 1200
`get_output()` (*sfepy.homogenization.coefs_base.CorrMiniApp* method), 953
`get_output()` (*sfepy.homogenization.coefs_base.CorrSolution* method), 954
`get_output_approx_order()` (*sfepy.discrete.fem.fields_base.FEField* method), 884
`get_output_function()` (*sfepy.base.base.Output* method), 781
`get_output_name()` (*sfepy.discrete.problem.Problem*

`method`), 837
`get_output_prefix()` (*sfepy.base.base.Output method*), 781
`get_output_shape()` (in module *sfepy.terms.terms_multilinear*), 1151
`get_output_suffix()` (in module *sfepy.homogenization.recovery*), 966
`get_p_edge()` (in module *sfepy.tests.test_functions*), 1193
`get_parameter_names()` (*sfepy.terms.terms.Term method*), 1052
`get_parameter_variables()` (*sfepy.terms.terms.Term method*), 1052
`get_parents()` (in module *sfepy.discrete.common.region*), 876
`get_pars()` (in module *sfepy.tests.test_elasticity_small_strain*), 1192
`get_pars()` (in module *sfepy.tests.test_functions*), 1193
`get_pars()` (in module *sfepy.tests.test_term_consistency*), 1200
`get_patch_box_regions()` (in module *sfepy.discrete.iga.iga*), 945
`get_paths()` (*sfepy.terms.terms_multilinear.ETermBase method*), 1147
`get_perpendiculars()` (in module *sfepy.linalg.geometry*), 971
`get_physical_qps()` (in module *sfepy.discrete.common.mappings*), 872
`get_physical_qps()` (*sfepy.discrete.fem.mappings.FEMappings method*), 896
`get_physical_qps()` (*sfepy.discrete.iga.mappings.IGMappings method*), 948
`get_physical_qps()` (*sfepy.discrete.structural.mappings.ShellMappings method*), 950
`get_physical_qps()` (*sfepy.terms.terms.Term method*), 1052
`get_physical_qps()` (*sfepy.terms.terms_shells.Shell10XTerm method*), 1171
`get_points()` (*sfepy.discrete.probes.CircleProbe method*), 828
`get_points()` (*sfepy.discrete.probes.LineProbe method*), 828
`get_points()` (*sfepy.discrete.probes.PointsProbe method*), 828
`get_points()` (*sfepy.discrete.probes.RayProbe method*), 830
`get_poly()` (in module *sfepy.tests.test_quadratures*), 1198
`get_potential_cells()` (in module *sfepy.discrete.common.global_interp*), 867
`get_prefix()` (*sfepy.mechanics.units.Unit static method*), 992
`get_primary()` (*sfepy.discrete.variables.Variable method*), 852
`get_primary_name()` (*sfepy.discrete.variables.Variable method*), 852
`get_print_info()` (in module *sfepy.base.ioutils*), 795
`get_print_info()` (in module *sfepy.solvers.ts*), 1039
`get_qp()` (*sfepy.discrete.fem.fields_base.FEField method*), 884
`get_qp()` (*sfepy.discrete.integrals.Integral method*), 823
`get_qp_key()` (*sfepy.terms.terms.Term method*), 1052
`get_queue()` (in module *sfepy.base.multiproc_mpi*), 803
`get_queue()` (in module *sfepy.base.multiproc_proc*), 804
`get_range_indices()` (in module *sfepy.terms.utils*), 1178
`get_ranges()` (in module *sfepy.homogenization.coefs_phononic*), 959
`get_raveled_index()` (in module *sfepy.discrete.iga.iga*), 946
`get_raveler()` (in module *sfepy.discrete.dg.fields*), 927
`get_raw()` (*sfepy.base.conf.ProblemConf method*), 791
`get_recovery_points()` (in module *sfepy.homogenization.recovery*), 966
`get_reduced()` (*sfepy.discrete.variables.FieldVariable method*), 850
`get_reduced_state()` (*sfepy.discrete.variables.Variables method*), 854
`get_ref_coors()` (in module *sfepy.discrete.common.global_interp*), 868
`get_ref_coors_convex()` (in module *sfepy.discrete.common.global_interp*), 868
`get_ref_coors_general()` (in module *sfepy.discrete.common.global_interp*), 869
`get_region()` (*sfepy.terms.terms.ConnInfo method*), 1050
`get_region()` (*sfepy.terms.terms.Term method*), 1052
`get_region_info()` (*sfepy.discrete.dg.fields.DGField static method*), 925
`get_region_name()` (*sfepy.terms.terms.ConnInfo method*), 1050
`get_restart_filename()` (*sfepy.discrete.problem.Problem method*), 837
`get_save_name()` (*sfepy.homogenization.coefs_base.CorrMiniApp method*), 953
`get_save_name_base()` (*sfepy.homogenization.coefs_base.CorrMiniApp method*), 953
`get_scaled_errors()` (*sfepy.solvers.ts_controllers.ElastodynamicsBasicTSC static method*), 1039
`get_shape()` (*sfepy.discrete.common.mappings.PhysicalQPs method*), 870
`get_shape_kind()` (in module *sfepy.terms.terms*), 1054
`get_simplex_circumcentres()` (in module

- `sfepy.linalg.geometry`), 971
- `get_simplex_cubature()` (in module `sfepy.discrete.simplex_cubature`), 847
- `get_simplex_volumes()` (in module `sfepy.linalg.geometry`), 971
- `get_sizes()` (in module `sfepy.parallel.parallel`), 1009
- `get_sizes()` (in module `sfepy.terms.terms_multilinear`), 1151
- `get_slaves()` (in module `sfepy.base.multiproc_mpi`), 803
- `get_slice_ops()` (in module `sfepy.terms.terms_multilinear`), 1151
- `get_solver()` (`sfepy.discrete.problem.Problem` method), 837
- `get_solver_conf()` (`sfepy.discrete.problem.Problem` method), 837
- `get_sorted_dependencies()` (`sfepy.homogenization.engine.HomogenizationWorker` static method), 962
- `get_sphinx_make_command()` (in module `build_helpers`), 1202
- `get_standard_keywords()` (in module `sfepy.base.conf`), 792
- `get_standard_type_defs()` (in module `sfepy.base.parse_conf`), 805
- `get_state()` (`sfepy.discrete.variables.Variables` method), 854
- `get_state()` (`sfepy.solvers.ts.TimeStepper` method), 1038
- `get_state()` (`sfepy.solvers.ts.VariableTimeStepper` method), 1038
- `get_state_in_region()` (`sfepy.discrete.variables.FieldVariable` method), 851
- `get_state_names()` (`sfepy.terms.terms.Term` method), 1052
- `get_state_parts()` (`sfepy.discrete.variables.Variables` method), 854
- `get_state_per_cell()` (in module `sfepy.terms.terms_jax`), 1134
- `get_state_variables()` (`sfepy.terms.terms.Term` method), 1052
- `get_str()` (`sfepy.terms.terms.Term` method), 1053
- `get_strain()` (in module `sfepy.terms.terms_jax`), 1134
- `get_stress()` (in module `sfepy.terms.terms_jax`), 1134
- `get_subdict()` (in module `sfepy.base.base`), 784
- `get_subset_info()` (`sfepy.discrete.common.dof_info.DofInfo` method), 858
- `get_surface_basis()` (`sfepy.discrete.fem.fields_nodal.GlobalNodalLike` method), 889
- `get_surface_basis()` (`sfepy.discrete.iga.fields.IGField` method), 940
- `get_surface_degrees()` (in module `sfepy.discrete.iga.iga`), 946
- `get_surface_entities()` (`sfepy.discrete.fem.geometry_element.GeometryElement` method), 891
- `get_surface_faces()` (in module `sfepy.mesh.mesh_tools`), 1005
- `get_surface_facets()` (`sfepy.discrete.common.extmods.cmesh.CMesh` method), 863
- `get_sym_indices()` (in module `sfepy.mechanics.tensors`), 989
- `get_t4_from_t2s()` (in module `sfepy.mechanics.tensors`), 989
- `get_tangent_stress_matrix()` (in module `sfepy.mechanics.membranes`), 985
- `get_tensor_product_conn()` (in module `sfepy.mesh.mesh_generators`), 1004
- `get_timestepper()` (`sfepy.discrete.problem.Problem` method), 837
- `get_tolerance()` (`sfepy.solvers.solvers.LinearSolver` method), 1036
- `get_top_dir()` (in module `sfepy.config`), 778
- `get_totals()` (`sfepy.base.timing.Timers` method), 807
- `get_trace()` (in module `sfepy.mechanics.tensors`), 989
- `get_true_order()` (`sfepy.discrete.fem.fields_base.FEField` method), 884
- `get_true_order()` (`sfepy.discrete.iga.fields.IGField` method), 940
- `get_trunk()` (in module `sfepy.base.ioutils`), 796
- `get_ts_val()` (`sfepy.homogenization.coefs_base.CorrSolution` method), 954
- `get_tss()` (`sfepy.discrete.problem.Problem` method), 837
- `get_tss_functions()` (`sfepy.discrete.problem.Problem` method), 837
- `get_type()` (`sfepy.base.ioutils.DataSoftLink` method), 793
- `get_u_edge()` (in module `sfepy.tests.test_functions`), 1193
- `get_unraveled_indices()` (in module `sfepy.discrete.iga.iga`), 946
- `get_unraveler()` (in module `sfepy.discrete.dg.fields`), 927
- `get_user_names()` (`sfepy.terms.terms.Term` method), 1053
- `get_user_names()` (`sfepy.terms.terms.Terms` method), 1054
- `get_var_names()` (`sfepy.discrete.conditions.LinearCombinationBC` method), 810
- `get_variable()` (`sfepy.discrete.equations.Equations` method), 815
- `get_variable_dependencies()`

- (*sfepy.discrete.equations.Equations* method), 815
- get_variable_names()* (*sfepy.discrete.equations.Equations* method), 816
- get_variable_names()* (*sfepy.terms.terms.Term* method), 1053
- get_variable_names()* (*sfepy.terms.terms.Terms* method), 1054
- get_variables()* (*sfepy.discrete.problem.Problem* method), 838
- get_variables()* (*sfepy.terms.terms.Term* method), 1053
- get_vec_part()* (*sfepy.discrete.variables.Variables* method), 855
- get_vector()* (*sfepy.terms.terms.Term* method), 1053
- get_vector_format()* (*sfepy.discrete.fem.meshio.MeshIO* method), 904
- get_vectors()* (*sfepy.discrete.fem.lcbc_operators.EdgeDirectional* method), 892
- get_vectors()* (*sfepy.discrete.fem.lcbc_operators.NormalDirectional* method), 894
- get_version()* (in module *sfepy.version*), 778
- get_vertices()* (in module *sfepy.tests.test_regions*), 1199
- get_vertices()* (*sfepy.discrete.fem.fields_base.FEField* method), 884
- get_virtual_name()* (*sfepy.terms.terms.Term* method), 1053
- get_virtual_variable()* (*sfepy.terms.terms.Term* method), 1053
- get_volume()* (in module *sfepy.homogenization.utils*), 968
- get_volume()* (in module *sfepy.tests.test_mesh_smoothing*), 1196
- get_volumes()* (*sfepy.discrete.common.extmods.cmesh.CMesh* method), 863
- get_volumetric_tensor()* (in module *sfepy.mechanics.tensors*), 989
- get_von_mises_stress()* (in module *sfepy.mechanics.tensors*), 989
- get_vtk_by_group()* (in module *sfepy.postprocess.utils_vtk*), 1015
- get_vtk_edges()* (in module *sfepy.postprocess.utils_vtk*), 1015
- get_vtk_from_file()* (in module *sfepy.postprocess.utils_vtk*), 1015
- get_vtk_from_mesh()* (in module *sfepy.postprocess.utils_vtk*), 1016
- get_vtk_surface()* (in module *sfepy.postprocess.utils_vtk*), 1016
- getBCnum()* (*sfepy.mesh.geom_tools.geometry* method), 1000
- getcenterpoint()* (*sfepy.mesh.geom_tools.surface* method), 1000
- getholepoints()* (*sfepy.mesh.geom_tools.surface* method), 1000
- getinsidepoint()* (*sfepy.mesh.geom_tools.surface* method), 1000
- getinsidepoint()* (*sfepy.mesh.geom_tools.volume* method), 1000
- getlines()* (*sfepy.mesh.geom_tools.surface* method), 1000
- getn()* (*sfepy.mesh.geom_tools.geomobject* method), 1000
- getpoints()* (*sfepy.mesh.geom_tools.line* method), 1000
- getpoints()* (*sfepy.mesh.geom_tools.surface* method), 1000
- getstr()* (*sfepy.mesh.geom_tools.point* method), 1000
- getsurfaces()* (*sfepy.mesh.geom_tools.physicalsurface* method), 1000
- getsurfaces()* (*sfepy.mesh.geom_tools.volume* method), 1001
- getvolume()* (*sfepy.mesh.geom_tools.physicalvolume* method), 1000
- getxyz()* (*sfepy.mesh.geom_tools.point* method), 1000
- GlobalNodalLikeBasis* (class in *sfepy.discrete.fem.fields_nodal*), 889
- GmshIO* (class in *sfepy.discrete.fem.meshio*), 900
- grad()* (in module *sfepy.linalg.sympy_operators*), 974
- grad_as_vector()* (in module *sfepy.terms.terms_adj_navier_stokes*), 1064
- grad_v()* (in module *sfepy.linalg.sympy_operators*), 974
- GradDivStabilizationTerm* (class in *sfepy.terms.terms_navier_stokes*), 1154
- gradjacobiP()* (*sfepy.discrete.dg.poly_spaces.LegendrePolySpace* method), 928
- gradlegendreP()* (*sfepy.discrete.dg.poly_spaces.LegendrePolySpace* method), 929
- GradTerm* (class in *sfepy.terms.terms_navier_stokes*), 1154
- group_by_variables()* (*sfepy.discrete.conditions.Conditions* method), 808
- group_chains()* (in module *sfepy.discrete.common.dof_info*), 859
- guess()* (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* static method), 900
- guess_time_units()* (in module *sfepy.postprocess.time_history*), 1014
- ## H
- H1BernsteinSurfaceField* (class in *sfepy.discrete.fem.fields_positive*), 890
- H1BernsteinVolumeField* (class in *sfepy.discrete.fem.fields_positive*), 890

H1DiscontinuousField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 889	in	HomogenizationWorker (class in <i>sfepy.homogenization.engine</i>), 961	in
H1HierarchicVolumeField (class in <i>sfepy.discrete.fem.fields_hierarchic</i>), 887	in	HomogenizationWorkerMulti (class in <i>sfepy.homogenization.engine</i>), 962	in
H1Mixin (class in <i>sfepy.discrete.fem.fields_base</i>), 885		HomogenizationWorkerMultiMPI (class in <i>sfepy.homogenization.engine</i>), 963	in
H1NodalMixin (class in <i>sfepy.discrete.fem.fields_nodal</i>), 889		hyperelastic_mode (<i>sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase</i> attribute), 1119	
H1NodalSurfaceField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 889	in	hyperelastic_mode (<i>sfepy.terms.terms_hyperelastic_ul.HyperElasticULBase</i> attribute), 1126	
H1NodalVolumeField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 889	in	HyperElasticBase (class in <i>sfepy.terms.terms_hyperelastic_base</i>), 1114	in
H1SEMSurfaceField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 890	in	HyperelasticByFunULTerm (class in <i>sfepy.terms.terms_hyperelastic_ul</i>), 1126	in
H1SEMVolumeField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 890	in	HyperElasticFamilyData (class in <i>sfepy.terms.terms_hyperelastic_base</i>), 1115	in
H1SNodalSurfaceField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 890	in	HyperElasticSurfaceTLBase (class in <i>sfepy.terms.terms_hyperelastic_tl</i>), 1118	in
H1SNodalVolumeField (class in <i>sfepy.discrete.fem.fields_nodal</i>), 890	in	HyperElasticSurfaceTLFamilyData (class in <i>sfepy.terms.terms_hyperelastic_tl</i>), 1118	in
has_attr() (in module <i>sfepy.config</i>), 778		HyperElasticTLBase (class in <i>sfepy.terms.terms_hyperelastic_tl</i>), 1118	in
has_cells() (<i>sfepy.discrete.common.region.Region</i> method), 876		HyperElasticTLFamilyData (class in <i>sfepy.terms.terms_hyperelastic_tl</i>), 1119	in
has_ebc() (<i>sfepy.discrete.variables.FieldVariable</i> method), 851		HyperElasticULBase (class in <i>sfepy.terms.terms_hyperelastic_ul</i>), 1125	in
has_ebc() (<i>sfepy.discrete.variables.Variables</i> method), 855		HyperElasticULFamilyData (class in <i>sfepy.terms.terms_hyperelastic_ul</i>), 1126	in
has_extra_nodes() (<i>sfepy.discrete.fem.poly_spaces.NodeHyperElastic</i> method), 908		HypermeshAsciiMeshIO (class in <i>sfepy.discrete.fem.meshio</i>), 903	in
has_faces() (<i>sfepy.discrete.common.domain.Domain</i> method), 860			
has_key() (<i>sfepy.base.base.Container</i> method), 780			
has_same_mesh() (<i>sfepy.discrete.variables.FieldVariable</i> method), 851		IdentityLimiter (class in <i>sfepy.discrete.dg.limiters</i>), 931	
has_virtuals() (<i>sfepy.discrete.variables.Variables</i> method), 855		iel (<i>sfepy.discrete.fem.extmods.bases.CLagrangeContext</i> attribute), 878	
have_good_cython() (in module <i>build_helpers</i>), 1202		iel (<i>sfepy.discrete.iga.extmods.igac.CNURBSContext</i> attribute), 936	
HDF5BaseData (class in <i>sfepy.base.ioutils</i>), 794		IGDomain (class in <i>sfepy.discrete.iga.domain</i>), 934	
HDF5ContextManager (class in <i>sfepy.base.ioutils</i>), 794		IGField (class in <i>sfepy.discrete.iga.fields</i>), 938	
HDF5Data (class in <i>sfepy.base.ioutils</i>), 794		IGMapping (class in <i>sfepy.discrete.iga.mappings</i>), 947	
HDF5MeshIO (class in <i>sfepy.discrete.fem.meshio</i>), 902		import_file() (in module <i>sfepy.base.base</i>), 784	
HDF5XdmfMeshIO (class in <i>sfepy.discrete.fem.meshio</i>), 903		in1d() (in module <i>sfepy.base.compat</i>), 786	
he_eval_from_mtx() (in module <i>sfepy.terms.extmods.terms</i>), 1182	in	in_dir() (in module <i>sfepy.tests.test_mesh_interp</i>), 1196	
he_residuum_from_mtx() (in module <i>sfepy.terms.extmods.terms</i>), 1182	in	IndexedStruct (class in <i>sfepy.base.base</i>), 781	
head() (in module <i>sfepy.discrete.dg.dg_ID_vizualizer</i>), 915		indices (<i>sfepy.discrete.common.extmods.cmesh.CCConnectivity</i> attribute), 861	
Histories (class in <i>sfepy.discrete.fem.history</i>), 892		InDir (class in <i>sfepy.base.ioutils</i>), 794	
History (class in <i>sfepy.discrete.fem.history</i>), 892		inedir() (in module <i>sfepy.tests.test_declarative_examples</i>), 1189	
HomogenizationApp (class in <i>sfepy.homogenization.homogen_app</i>), 963	in	infinity_norm() (in module <i>sfepy.linalg.sparse</i>), 973	
HomogenizationEngine (class in <i>sfepy.homogenization.engine</i>), 960	in	init() (<i>sfepy.mechanics.matcoefs.ElasticConstants</i> method), 980	

`init_data()` (*sfepy.discrete.variables.Variable* method), 852
`init_data_struct()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBase* method), 1115
`init_global_search()` (in module *sfepy.mechanics.extmods.ccontres*), 993
`init_history()` (*sfepy.discrete.variables.Variable* method), 852
`init_history()` (*sfepy.discrete.variables.Variables* method), 855
`init_ksp()` (*sfepy.solvers.ls.PETScKrylovSolver* method), 1023
`init_petsc_args()` (in module *sfepy.parallel.parallel*), 1009
`init_rmm()` (*sfepy.solvers.ls.RMMSolver* method), 1025
`init_slepc_args()` (in module *sfepy.solvers.eigen*), 1020
`init_solvers()` (*sfepy.discrete.problem.Problem* method), 838
`init_solvers()` (*sfepy.homogenization.coefs_base.MiniAppBase* method), 954
`init_state()` (*sfepy.discrete.equations.Equations* method), 816
`init_state()` (*sfepy.discrete.variables.Variables* method), 855
`init_subproblems()` (*sfepy.solvers.ls.MultiProblem* method), 1021
`init_time()` (*sfepy.discrete.equations.Equations* method), 816
`init_time()` (*sfepy.discrete.problem.Problem* method), 838
`init_vec()` (in module *sfepy.tests.test_conditions*), 1189
`InitialCondition` (class in *sfepy.discrete.conditions*), 809
`initialize_options()` (*build_helpers.NoOptionsDocs* method), 1201
`insert()` (*sfepy.base.base.Container* method), 780
`insert()` (*sfepy.terms.terms.Terms* method), 1054
`insert_as_static_method()` (in module *sfepy.base.base*), 784
`insert_knot()` (*sfepy.mesh.bspline.BSpline* method), 995
`insert_method()` (in module *sfepy.base.base*), 784
`insert_sparse_to_csr()` (in module *sfepy.linalg.sparse*), 974
`insert_static_method()` (in module *sfepy.base.base*), 784
`insert_strided_axis()` (in module *sfepy.linalg.utils*), 977
`insert_sub_reqs()` (in module *sfepy.homogenization.engine*), 963
`install_data` module, 1206
`install_data` (class in *install_data*), 1206
`int_dt()` (*sfepy.homogenization.convolution.ConvolutionKernel* method), 1150
`Integral` (class in *sfepy.discrete.integrals*), 823
`IntegralMeanValueOperator` (class in *sfepy.discrete.fem.lcbc_operators*), 892
`IntegralProbe` (class in *sfepy.discrete.probes*), 828
`Integrals` (class in *sfepy.discrete.integrals*), 823
`integrate()` (*sfepy.discrete.common.mappings.PyCMapping* method), 871
`integrate()` (*sfepy.discrete.integrals.Integral* method), 823
`integrate()` (*sfepy.terms.terms_contact.ContactTerm* static method), 1078
`integrate()` (*sfepy.terms.terms_hyperelastic_base.HyperElasticBase* static method), 1115
`integrate_along_line()` (in module *sfepy.scripts.probe*), 1185
`integrate_in_time()` (in module *sfepy.homogenization.utils*), 968
`IntegrateMatTerm` (class in *sfepy.terms.terms_basic*), 1065
`IntegrateOperatorTerm` (class in *sfepy.terms.terms_basic*), 1065
`IntegrateSurfaceMatTerm` (class in *sfepy.terms.terms_compat*), 1073
`IntegrateSurfaceOperatorTerm` (class in *sfepy.terms.terms_compat*), 1073
`IntegrateSurfaceTerm` (class in *sfepy.terms.terms_compat*), 1073
`IntegrateTerm` (class in *sfepy.terms.terms_basic*), 1066
`IntegrateVolumeMatTerm` (class in *sfepy.terms.terms_compat*), 1074
`IntegrateVolumeOperatorTerm` (class in *sfepy.terms.terms_compat*), 1074
`IntegrateVolumeTerm` (class in *sfepy.terms.terms_compat*), 1074
`integration` (*sfepy.terms.terms.Term* attribute), 1053
`integration` (*sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPress* attribute), 1058
`integration` (*sfepy.terms.terms_basic.IntegrateMatTerm* attribute), 1065
`integration` (*sfepy.terms.terms_basic.IntegrateOperatorTerm* attribute), 1066
`integration` (*sfepy.terms.terms_basic.IntegrateTerm* attribute), 1066
`integration` (*sfepy.terms.terms_basic.SurfaceMomentTerm* attribute), 1067
`integration` (*sfepy.terms.terms_basic.VolumeSurfaceTerm* attribute), 1068
`integration` (*sfepy.terms.terms_basic.VolumeTerm* attribute), 1068
`integration` (*sfepy.terms.terms_biot.BiotStressTerm* attribute), 1070

`integration(sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1076`
`integration(sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1076`
`integration(sfepy.terms.terms_contact.ContactIPTerm attribute), 1077`
`integration(sfepy.terms.terms_contact.ContactTerm attribute), 1078`
`integration(sfepy.terms.terms_dg.AdvectionDGFluxTerm attribute), 1080`
`integration(sfepy.terms.terms_dg.DiffusionDGFluxTerm attribute), 1081`
`integration(sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm attribute), 1083`
`integration(sfepy.terms.terms_diffusion.DiffusionVelocityTerm attribute), 1087`
`integration(sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm attribute), 1090`
`integration(sfepy.terms.terms_diffusion.SurfaceFluxTerm attribute), 1090`
`integration(sfepy.terms.terms_dot.BCNewtonTerm attribute), 1091`
`integration(sfepy.terms.terms_dot.DotProductTerm attribute), 1092`
`integration(sfepy.terms.terms_elastic.CauchyStrainTerm attribute), 1097`
`integration(sfepy.terms.terms_elastic.CauchyStressTerm attribute), 1099`
`integration(sfepy.terms.terms_hyperelastic_base.DeformationTerm attribute), 1114`
`integration(sfepy.terms.terms_hyperelastic_base.HyperElasticTerm attribute), 1115`
`integration(sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTerm attribute), 1121`
`integration(sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTerm attribute), 1122`
`integration(sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTerm attribute), 1123`
`integration(sfepy.terms.terms_jax.MassADTerm attribute), 1130`
`integration(sfepy.terms.terms_membrane.TLMembraneTerm attribute), 1136`
`integration(sfepy.terms.terms_multilinear.EDotTerm attribute), 1140`
`integration(sfepy.terms.terms_multilinear.EIntegrateOperatorTerm attribute), 1141`
`integration(sfepy.terms.terms_multilinear.ELinearTractionTerm attribute), 1143`
`integration(sfepy.terms.terms_multilinear.ENonPenetrationPenaltyTerm attribute), 1144`
`integration(sfepy.terms.terms_multilinear.StokesTractionTerm attribute), 1149`
`integration(sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm attribute), 1150`
`integration(sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm attribute), 1150`
`integration(sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm attribute), 1151`
`integration(sfepy.terms.terms_navier_stokes.DivTerm attribute), 1154`
`integration(sfepy.terms.terms_navier_stokes.GradTerm attribute), 1155`
`integration(sfepy.terms.terms_point.ConcentratedPointLoadTerm attribute), 1163`
`integration(sfepy.terms.terms_point.LinearPointSpringTerm attribute), 1164`
`integration(sfepy.terms.terms_sensitivity.ESDLinearTractionTerm attribute), 1167`
`integration(sfepy.terms.terms_shells.Shell10XTerm attribute), 1171`
`integration(sfepy.terms.terms_surface.ContactPlaneTerm attribute), 1173`
`integration(sfepy.terms.terms_surface.ContactSphereTerm attribute), 1174`
`integration(sfepy.terms.terms_surface.LinearTractionTerm attribute), 1174`
`integration(sfepy.terms.terms_surface.SDLinearTractionTerm attribute), 1175`
`integration(sfepy.terms.terms_surface.SDSurfaceIntegrateTerm attribute), 1176`
`integration(sfepy.terms.terms_surface.SurfaceNormalDotTerm attribute), 1176`
`integration(sfepy.terms.terms_surface.SurfaceJumpTerm attribute), 1177`
`integration_order(sfepy.terms.terms.Term attribute), 1053`
`integration_order(sfepy.terms.terms_elastic.LinearDSpringTerm attribute), 1101`
`integration_order(sfepy.terms.terms_elastic.LinearSpringTerm attribute), 1105`
`integration_order(sfepy.terms.terms_elastic.LinearTrussInternalForceTerm attribute), 1107`
`integration_order(sfepy.terms.terms_elastic.LinearTrussTerm attribute), 1107`
`interp_conv_mat()` (in module `sfepy.homogenization.utils`), 968
`interp_to_qp()` (`sfepy.discrete.fem.fields_base.FEField` method), 884
`interp_n_vals_to_n_vals()` (`sfepy.discrete.fem.fields_nodal.H1NodalSurfaceField` method), 889
`interp_v_vals_to_n_vals()` (`sfepy.discrete.fem.fields_nodal.H1NodalVolumeField` method), 890
`invalidate_evaluate_cache()` (`sfepy.discrete.variables.FieldVariable` method), 851
`invalidate_evaluate_caches()`

(*sfepy.discrete.variables.Variables* method), 855
 invalidate_term_caches() (*sfepy.discrete.equations.Equations* method), 816
 inverse_element_mapping() (in module *sfepy.linalg.geometry*), 972
 invert_dict() (in module *sfepy.base.base*), 784
 invert_remap() (in module *sfepy.discrete.fem.utils*), 913
 invs_fast() (in module *sfepy.linalg.utils*), 977
 iplot() (in module *sfepy.base.plotutils*), 806
 ipython_shell() (in module *sfepy.base.base*), 784
 is_active_bc() (in module *sfepy.discrete.common.dof_info*), 859
 is_bubble(*sfepy.discrete.fem.extmods.bases.CLagrangeContinuity* attribute), 878
 is_complex() (*sfepy.discrete.variables.Variable* method), 852
 is_cyclic(*sfepy.discrete.probes.CircleProbe* attribute), 828
 is_cyclic(*sfepy.discrete.probes.Probe* attribute), 829
 is_derived_class() (in module *sfepy.base.base*), 784
 is_finite() (*sfepy.discrete.variables.Variable* method), 853
 is_higher_order() (*sfepy.discrete.fem.fields_base.FEField* method), 884
 is_higher_order() (*sfepy.discrete.iga.fields.IGField* method), 940
 is_integer() (in module *sfepy.base.base*), 784
 is_kind() (*sfepy.discrete.variables.Variable* method), 853
 is_linear() (*sfepy.discrete.problem.Problem* method), 838
 is_nurbs() (in module *sfepy.discrete.iga.extmods.igac*), 938
 is_parameter() (*sfepy.discrete.variables.Variable* method), 853
 is_real() (*sfepy.discrete.variables.Variable* method), 853
 is_release() (*sfepy.config.Config* method), 777
 is_remote_dict() (in module *sfepy.base.multiproc*), 801
 is_remote_dict() (in module *sfepy.base.multiproc_mpi*), 803
 is_remote_dict() (in module *sfepy.base.multiproc_proc*), 804
 is_sequence() (in module *sfepy.base.base*), 784
 is_state() (*sfepy.discrete.variables.Variable* method), 853
 is_state_or_parameter() (*sfepy.discrete.variables.Variable* method), 853
 is_string() (in module *sfepy.base.base*), 785
 is_surface(*sfepy.discrete.dg.fields.DGField* attribute), 925
 is_sym (*sfepy.homogenization.coefs_base.CoeffNonSym* attribute), 952
 is_sym(*sfepy.homogenization.coefs_base.CoeffNonSymNonSym* attribute), 952
 is_sym (*sfepy.homogenization.coefs_base.CoeffSym* attribute), 953
 is_sym (*sfepy.homogenization.coefs_base.CoeffSymSym* attribute), 953
 is_virtual() (*sfepy.discrete.variables.Variable* method), 853
 iter0() (in module *sfepy.discrete.fem.facets*), 879
 iter01() (in module *sfepy.discrete.fem.facets*), 879
 iter01x01y() (in module *sfepy.discrete.fem.facets*), 879
 iter01x10y() (in module *sfepy.discrete.fem.facets*), 879
 iter01y01x() (in module *sfepy.discrete.fem.facets*), 880
 iter01y10x() (in module *sfepy.discrete.fem.facets*), 880
 iter02() (in module *sfepy.discrete.fem.facets*), 880
 iter1() (in module *sfepy.discrete.fem.facets*), 880
 iter10() (in module *sfepy.discrete.fem.facets*), 880
 iter10x01y() (in module *sfepy.discrete.fem.facets*), 880
 iter10x10y() (in module *sfepy.discrete.fem.facets*), 880
 iter10y01x() (in module *sfepy.discrete.fem.facets*), 880
 iter10y10x() (in module *sfepy.discrete.fem.facets*), 880
 iter12() (in module *sfepy.discrete.fem.facets*), 880
 iter20() (in module *sfepy.discrete.fem.facets*), 880
 iter21() (in module *sfepy.discrete.fem.facets*), 880
 iter_by_order() (in module *sfepy.discrete.dg.poly_spaces*), 930
 iter_dict_of_lists() (in module *sfepy.base.base*), 785
 iter_from() (*sfepy.solvers.ts.TimeStepper* method), 1038
 iter_from() (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
 iter_from_current() (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
 iter_names() (in module *sfepy.base.log*), 799
 iter_nonsym() (in module *sfepy.homogenization.utils*), 968
 iter_single() (*sfepy.discrete.conditions.Condition* method), 808
 iter_solutions() (*sfepy.homogenization.coefs_base.CorrSolution* method), 954
 iter_state() (*sfepy.discrete.variables.Variables* method), 855
 iter_sym() (in module *sfepy.homogenization.utils*), 968
 iter_sym() (*sfepy.homogenization.coefs_base.CoeffNonSym* static method), 952
 iter_sym() (*sfepy.homogenization.coefs_base.CoeffNonSymNonSym* static method), 952
 iter_sym() (*sfepy.homogenization.coefs_base.CoeffSym*

- static method), 953
- `iter_sym()` (`sfe.py.homogenization.coefs_base.CoeffSymSym` static method), 953
- `iter_terms()` (`sfe.py.discrete.materials.Material` method), 825
- `iter_time_steps()` (`sfe.py.homogenization.coefs_base.CorrSolution` method), 954
- `iteritems()` (`sfe.py.base.base.Container` method), 780
- `iterkeys()` (`sfe.py.base.base.Container` method), 780
- `itervalues()` (`sfe.py.base.base.Container` method), 780
- ## J
- `jacobiP()` (`sfe.py.discrete.dg.poly_spaces.LegendrePolySpace` method), 929
- `join_subscripts()` (`sfe.py.terms.terms_multilinear.ExpressionBuilder` static method), 1148
- `join_tokens()` (in module `sfe.py.discrete.parse_regions`), 827
- ## K
- `key_to_index` (`sfe.py.discrete.common.extmods.cmesh.CMesh` attribute), 863
- `keys` (`sfe.py.discrete.common.poly_spaces.PolySpace` attribute), 873
- `keys()` (`sfe.py.base.goptions.ValidatedDict` method), 793
- `keys()` (`sfe.py.base.multiproc_mpi.RemoteDict` method), 802
- `kind` (`sfe.py.discrete.fem.lcbc_operators.AverageForceOperator` attribute), 892
- `kind` (`sfe.py.discrete.fem.lcbc_operators.EdgeDirectionOperator` attribute), 892
- `kind` (`sfe.py.discrete.fem.lcbc_operators.IntegralMeanValueOperator` attribute), 892
- `kind` (`sfe.py.discrete.fem.lcbc_operators.MatchDOFsOperator` attribute), 894
- `kind` (`sfe.py.discrete.fem.lcbc_operators.MultiNodeLCOperator` attribute), 894
- `kind` (`sfe.py.discrete.fem.lcbc_operators.NodalLCOperator` attribute), 894
- `kind` (`sfe.py.discrete.fem.lcbc_operators.NoPenetrationOperator` attribute), 894
- `kind` (`sfe.py.discrete.fem.lcbc_operators.NormalDirectionOperator` attribute), 895
- `kind` (`sfe.py.discrete.fem.lcbc_operators.Rigid2Operator` attribute), 895
- `kind` (`sfe.py.discrete.fem.lcbc_operators.RigidOperator` attribute), 895
- `kind` (`sfe.py.discrete.fem.lcbc_operators.ShiftedPeriodicOperator` attribute), 895
- `L2ConstantSurfaceField` (class in `sfe.py.discrete.fem.fields_l2`), 887
- `L2ConstantVolumeField` (class in `sfe.py.discrete.fem.fields_l2`), 887
- `label_dofs()` (in module `sfe.py.parallel.plot_parallel_dofs`), 1010
- `label_global_entities()` (in module `sfe.py.postprocess.plot_cmesh`), 1010
- `label_local_entities()` (in module `sfe.py.postprocess.plot_cmesh`), 1010
- `label_points()` (in module `sfe.py.postprocess.plot_quadrature`), 1012
- `LagrangeNodes` (class in `sfe.py.discrete.fem.poly_spaces`), 907
- `LagrangePolySpace` (class in `sfe.py.discrete.fem.poly_spaces`), 908
- `LagrangeSimplexBPolySpace` (class in `sfe.py.discrete.fem.poly_spaces`), 908
- `LagrangeSimplexPolySpace` (class in `sfe.py.discrete.fem.poly_spaces`), 908
- `LagrangeTensorProductPolySpace` (class in `sfe.py.discrete.fem.poly_spaces`), 908
- `LagrangeWedgePolySpace` (class in `sfe.py.discrete.fem.poly_spaces`), 908
- `lame_from_stiffness()` (in module `sfe.py.mechanics.matcoefs`), 980
- `lame_from_youngpoisson()` (in module `sfe.py.mechanics.matcoefs`), 981
- `laplace()` (in module `sfe.py.linalg.sympy_operators`), 974
- `LaplaceTerm` (class in `sfe.py.terms.terms_diffusion`), 1087
- `layout_letters` (`sfe.py.terms.terms_multilinear.ETermBase` attribute), 1147
- `LCBCOperator` (class in `sfe.py.discrete.fem.lcbc_operators`), 892
- `LCBCOperators` (class in `sfe.py.discrete.fem.lcbc_operators`), 893
- `leaveonlyphysicalsurfaces()` (`sfe.py.mesh.geom_tools.geometry` method), 1000
- `leaveonlyphysicalvolumes()` (`sfe.py.mesh.geom_tools.geometry` method), 1000
- `legendre_funs` (`sfe.py.discrete.dg.poly_spaces.LegendrePolySpace` attribute), 929
- `legendreP()` (`sfe.py.discrete.dg.poly_spaces.LegendrePolySpace` method), 929
- `LegendrePolySpace` (class in `sfe.py.discrete.dg.poly_spaces`), 928
- `LegendreSimplexPolySpace` (class in `sfe.py.discrete.dg.poly_spaces`), 930
- `LegendreTensorProductPolySpace` (class in `sfe.py.discrete.dg.poly_spaces`), 930
- `letters` (`sfe.py.terms.terms_multilinear.ExpressionBuilder` attribute), 1148

`light_copy()` (*sfepy.discrete.common.region.Region* method), 876
`line` (class in *sfepy.mesh.geom_tools*), 1000
`linear()` (in module *sfepy.tests.test_laplace_unit_square*), 1194
`linear_x()` (in module *sfepy.tests.test_laplace_unit_square*), 1194
`linear_y()` (in module *sfepy.tests.test_laplace_unit_square*), 1194
`linear_z()` (in module *sfepy.tests.test_laplace_unit_square*), 1194
`LinearCombinationBC` (class in *sfepy.discrete.conditions*), 809
`LinearConvect2Term` (class in *sfepy.terms.terms_navier_stokes*), 1155
`LinearConvectTerm` (class in *sfepy.terms.terms_navier_stokes*), 1155
`LinearDRotSpringTerm` (class in *sfepy.terms.terms_elastic*), 1100
`LinearDSpringTerm` (class in *sfepy.terms.terms_elastic*), 1101
`LinearElasticETHTerm` (class in *sfepy.terms.terms_elastic*), 1102
`LinearElasticIsotropicTerm` (class in *sfepy.terms.terms_elastic*), 1102
`LinearElasticLADTerm` (class in *sfepy.terms.terms_jax*), 1128
`LinearElasticTerm` (class in *sfepy.terms.terms_elastic*), 1103
`LinearElasticTHTerm` (class in *sfepy.terms.terms_elastic*), 1103
`LinearElasticYPADTerm` (class in *sfepy.terms.terms_jax*), 1129
`linearize()` (*sfepy.discrete.fem.fields_base.FEField* method), 884
`LinearPointSpringTerm` (class in *sfepy.terms.terms_point*), 1163
`LinearPrestressTerm` (class in *sfepy.terms.terms_elastic*), 1104
`LinearSolver` (class in *sfepy.solvers.solvers*), 1036
`LinearSpringTerm` (class in *sfepy.terms.terms_elastic*), 1105
`LinearStrainFiberTerm` (class in *sfepy.terms.terms_elastic*), 1105
`LinearTractionTerm` (class in *sfepy.terms.terms_surface*), 1174
`LinearTrussInternalForceTerm` (class in *sfepy.terms.terms_elastic*), 1106
`LinearTrussTerm` (class in *sfepy.terms.terms_elastic*), 1107
`LinearVolumeForceTerm` (class in *sfepy.terms.terms_volume*), 1177
`LineProbe` (class in *sfepy.discrete.probes*), 828
`link_duals()` (*sfepy.discrete.variables.Variables* method), 855
`list_dict()` (in module *sfepy.base.parse_conf*), 805
`list_of()` (in module *sfepy.base.parse_conf*), 805
`load_1D_vtk()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 915
`load_classes()` (in module *sfepy.base.base*), 785
`load_dict()` (*sfepy.applications.pde_solver_app.PDESolverApp* method), 779
`load_restart()` (*sfepy.discrete.problem.Problem* method), 839
`load_slices` (*sfepy.discrete.fem.meshio.GmshIO* attribute), 900
`load_state_1D_vtk()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 916
`LobattoTensorProductPolySpace` (class in *sfepy.discrete.fem.poly_spaces*), 908
`LOBPCGEigenvalueSolver` (class in *sfepy.solvers.eigen*), 1017
`locate_files()` (in module *sfepy.base.ioutils*), 796
`lock_drilling_rotations()` (in module *sfepy.mechanics.shell10x*), 987
`Log` (class in *sfepy.base.log*), 798
`log()` (in module *sfepy.tests.test_log*), 1195
`log_filename()` (in module *sfepy.tests.test_log*), 1195
`LogPlotter` (class in *sfepy.base.log_plotter*), 800
`look_ahead_line()` (in module *sfepy.base.ioutils*), 796
`LQuadraticEVPSolver` (class in *sfepy.solvers.qeigen*), 1034

M

`main()` (in module *gen_field_table*), 1203
`main()` (in module *gen_gallery*), 1204
`main()` (in module *gen_legendre_simplex_base*), 1205
`main()` (in module *gen_lobatto1d_c*), 1205
`main()` (in module *gen_release_notes*), 1205
`main()` (in module *gen_serendipity_basis*), 1205
`main()` (in module *gen_solver_table*), 1205
`main()` (in module *gen_term_table*), 1206
`main()` (in module *sfepy.mesh.bspline*), 998
`main()` (in module *sfepy.mesh.mesh_generators*), 1004
`main()` (in module *sfepy.scripts.blockgen*), 1182
`main()` (in module *sfepy.scripts.convert_mesh*), 1182
`main()` (in module *sfepy.scripts.cylindergen*), 1183
`main()` (in module *sfepy.scripts.gen_iga_patch*), 1183
`main()` (in module *sfepy.scripts.gen_mesh*), 1183
`main()` (in module *sfepy.scripts.gen_mesh_prev*), 1183
`main()` (in module *sfepy.scripts.plot_condition_numbers*), 1184
`main()` (in module *sfepy.scripts.plot_logs*), 1184
`main()` (in module *sfepy.scripts.plot_mesh*), 1184
`main()` (in module *sfepy.scripts.plot_quadratures*), 1184
`main()` (in module *sfepy.scripts.plot_times*), 1184
`main()` (in module *sfepy.scripts.probe*), 1185
`main()` (in module *sfepy.scripts.resview*), 1187

main() (in module *sfepy.scripts.run_tests*), 1187
 main() (in module *sfepy.scripts.simple*), 1188
 main() (in module *show_authors*), 1206
 main() (in module *show_terms_use*), 1207
 main() (in module *sync_module_docs*), 1207
 make_axes() (*sfepy.base.log_plotter.LogPlotter* method), 800
 make_axis_rotation_matrix() (in module *sfepy.linalg.geometry*), 972
 make_cells_from_conn() (in module *sfepy.scripts.resview*), 1187
 make_eye() (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
 make_format() (*sfepy.discrete.fem.meshio.ANSYSCDBMesh* static method), 900
 make_full() (*sfepy.applications.evp_solver_app.EVPSolverApp* method), 778
 make_full_vec() (*sfepy.discrete.equations.Equations* method), 816
 make_full_vec() (*sfepy.discrete.evaluate.Evaluator* method), 818
 make_full_vec() (*sfepy.discrete.variables.Variables* method), 855
 make_function() (*sfepy.terms.terms_multilinear.ETermBase* method), 1147
 make_get_conf() (in module *sfepy.solvers.solvers*), 1037
 make_global_operator() (*sfepy.discrete.fem.lcbc_operators.LCBCOperator* method), 893
 make_grad2strain() (in module *sfepy.terms.terms_flexo*), 1113
 make_grid_from_mesh() (in module *sfepy.scripts.resview*), 1187
 make_h1_projection_data() (in module *sfepy.discrete.projections*), 844
 make_is_save() (in module *sfepy.discrete.problem*), 844
 make_knot_vector() (*sfepy.mesh.bspline.BSpline* method), 996
 make_knot_vector() (*sfepy.mesh.bspline.BSplineSurf* method), 997
 make_l2_projection() (in module *sfepy.discrete.projections*), 844
 make_l2_projection_data() (in module *sfepy.discrete.projections*), 844
 make_line_matrix() (in module *sfepy.discrete.fem.facets*), 880
 make_mesh() (in module *sfepy.discrete.fem.mesh*), 899
 make_option_docstring() (in module *sfepy.solvers.solvers*), 1037
 make_psg() (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
 make_pvg() (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
 make_sfepy_function() (in module *sfepy.discrete.functions*), 822
 make_square_matrix() (in module *sfepy.discrete.fem.facets*), 880
 make_term_args() (in module *sfepy.tests.test_term_call_modes*), 1200
 make_title() (in module *sfepy.scripts.resview*), 1187
 make_triangle_matrix() (in module *sfepy.discrete.fem.facets*), 880
 map_equations() (*sfepy.discrete.common.dof_info.EquationMap* method), 858
 map_permutations() (in module *sfepy.linalg.utils*), 978
 Mapping (class in *sfepy.discrete.common.mappings*), 870
 mark_subdomains() (in module *sfepy.parallel.plot_parallel_dofs*), 1010
 mask_points() (*sfepy.mechanics.contact_bodies.ContactPlane* method), 979
 mask_points() (*sfepy.mechanics.contact_bodies.ContactSphere* method), 979
 MassADTerm (class in *sfepy.terms.terms_jax*), 1129
 MassTerm (class in *sfepy.terms.terms_mass*), 1135
 master_loop() (in module *sfepy.base.multiproc_mpi*), 803
 master_send_continue() (in module *sfepy.base.multiproc_mpi*), 803
 master_send_task() (in module *sfepy.base.multiproc_mpi*), 803
 match_coors() (in module *sfepy.discrete.fem.periodic*), 907
 match_grid_line() (in module *sfepy.discrete.fem.periodic*), 907
 match_grid_plane() (in module *sfepy.discrete.fem.periodic*), 907
 match_plane_by_dir() (in module *sfepy.discrete.fem.periodic*), 907
 match_x_line() (in module *sfepy.discrete.fem.periodic*), 907
 match_x_plane() (in module *sfepy.discrete.fem.periodic*), 907
 match_y_line() (in module *sfepy.discrete.fem.periodic*), 907
 match_y_plane() (in module *sfepy.discrete.fem.periodic*), 907
 match_z_line() (in module *sfepy.discrete.fem.periodic*), 907
 match_z_plane() (in module *sfepy.discrete.fem.periodic*), 907
 MatchDOFsOperator (class in *sfepy.discrete.fem.lcbc_operators*), 893
 Material (class in *sfepy.discrete.materials*), 824
 Materials (class in *sfepy.discrete.materials*), 826
 MatlabEigenvalueSolver (class in *sfepy.solvers.eigen*), 1017

`max_diff_csr()` (in module `sfepy.linalg.utils`), 978
`mbfg` (`sfepy.discrete.fem.extmods.bases.CLagrangeContext` attribute), 878
`merge_lines()` (in module `sfepy.mesh.mesh_tools`), 1005
`merge_mesh()` (in module `sfepy.discrete.fem.mesh`), 899
`Mesh` (class in `sfepy.discrete.fem.mesh`), 898
`Mesh3DMeshIO` (class in `sfepy.discrete.fem.meshio`), 904
`mesh_conn` (`sfepy.discrete.fem.extmods.bases.CLagrangeContext` attribute), 878
`mesh_coors` (`sfepy.discrete.fem.extmods.bases.CLagrangeContext` attribute), 878
`mesh_from_groups()` (in module `sfepy.discrete.fem.meshio`), 906
`mesh_hook()` (in module `sfepy.tests.test_eigenvalue_solvers`), 1192
`mesh_hook()` (in module `sfepy.tests.test_meshio`), 1196
`MeshIO` (class in `sfepy.discrete.fem.meshio`), 904
`MeshioLibIO` (class in `sfepy.discrete.fem.meshio`), 905
`methods` (`sfepy.solvers.nls.ScipyRoot` attribute), 1030
`mini_newton()` (in module `sfepy.linalg.utils`), 978
`MiniAppBase` (class in `sfepy.homogenization.coefs_base`), 954
`minmod()` (in module `sfepy.discrete.dg.limiters`), 931
`minmod_seq()` (in module `sfepy.discrete.dg.limiters`), 932
`MixedFlexoCouplingTerm` (class in `sfepy.terms.terms_flexo`), 1111
`MixedFlexoTerm` (class in `sfepy.terms.terms_flexo`), 1112
`MixedStrainGradElasticTerm` (class in `sfepy.terms.terms_flexo`), 1113
`mode` (`sfepy.terms.terms_diffusion.AdvectDivFreeTerm` attribute), 1084
`mode` (`sfepy.terms.terms_dot.BCNewtonTerm` attribute), 1091
`modes` (`sfepy.terms.terms_biot.BiotETHTerm` attribute), 1070
`modes` (`sfepy.terms.terms_biot.BiotTerm` attribute), 1072
`modes` (`sfepy.terms.terms_biot.BiotTHTerm` attribute), 1071
`modes` (`sfepy.terms.terms_constraints.NonPenetrationTerm` attribute), 1076
`modes` (`sfepy.terms.terms_dg.AdvectionDGFluxTerm` attribute), 1080
`modes` (`sfepy.terms.terms_dg.DiffusionDGFluxTerm` attribute), 1081
`modes` (`sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm` attribute), 1082
`modes` (`sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm` attribute), 1083
`modes` (`sfepy.terms.terms_dg.NonlinearScalarDotGradTerm` attribute), 1084
`modes` (`sfepy.terms.terms_diffusion.DiffusionCoupling` attribute), 1085
`modes` (`sfepy.terms.terms_diffusion.DiffusionTerm` attribute), 1086
`modes` (`sfepy.terms.terms_diffusion.LaplaceTerm` attribute), 1088
`modes` (`sfepy.terms.terms_dot.DotProductTerm` attribute), 1092
`modes` (`sfepy.terms.terms_dot.ScalarDotMGradScalarTerm` attribute), 1094
`modes` (`sfepy.terms.terms_dot.VectorDotGradScalarTerm` attribute), 1095
`modes` (`sfepy.terms.terms_dot.VectorDotScalarTerm` attribute), 1096
`modes` (`sfepy.terms.terms_elastic.ElasticWaveCauchyTerm` attribute), 1099
`modes` (`sfepy.terms.terms_elastic.LinearElasticTerm` attribute), 1104
`modes` (`sfepy.terms.terms_elastic.LinearPrestressTerm` attribute), 1105
`modes` (`sfepy.terms.terms_elastic.NonsymElasticTerm` attribute), 1108
`modes` (`sfepy.terms.terms_flexo.MixedFlexoCouplingTerm` attribute), 1112
`modes` (`sfepy.terms.terms_flexo.MixedFlexoTerm` attribute), 1113
`modes` (`sfepy.terms.terms_flexo.MixedStrainGradElasticTerm` attribute), 1113
`modes` (`sfepy.terms.terms_jax.LinearElasticLADTerm` attribute), 1129
`modes` (`sfepy.terms.terms_jax.LinearElasticYPADTerm` attribute), 1129
`modes` (`sfepy.terms.terms_jax.MassADTerm` attribute), 1130
`modes` (`sfepy.terms.terms_jax.NeoHookeanTLADTerm` attribute), 1131
`modes` (`sfepy.terms.terms_jax.OgdenTLADTerm` attribute), 1132
`modes` (`sfepy.terms.terms_mass.MassTerm` attribute), 1135
`modes` (`sfepy.terms.terms_multilinear.EConvectTerm` attribute), 1138
`modes` (`sfepy.terms.terms_multilinear.EDiffusionTerm` attribute), 1138
`modes` (`sfepy.terms.terms_multilinear.EDivGradTerm` attribute), 1139
`modes` (`sfepy.terms.terms_multilinear.EDivTerm` attribute), 1139
`modes` (`sfepy.terms.terms_multilinear.EDotTerm` attribute), 1140
`modes` (`sfepy.terms.terms_multilinear.ELaplaceTerm` attribute), 1141
`modes` (`sfepy.terms.terms_multilinear.ELinearConvectTerm` attribute), 1142
`modes` (`sfepy.terms.terms_multilinear.ELinearElasticTerm`

- attribute*), 1143
- modes* (*sfepy.terms.terms_multilinear.ELinearTractionTerm attribute*), 1143
- modes* (*sfepy.terms.terms_multilinear.ESymElasticTerm attribute*), 1145
- modes* (*sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm attribute*), 1145
- modes* (*sfepy.terms.terms_multilinear.EStokesTerm attribute*), 1146
- modes* (*sfepy.terms.terms_multilinear.StokesTractionTerm attribute*), 1149
- modes* (*sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm attribute*), 1150
- modes* (*sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm attribute*), 1150
- modes* (*sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm attribute*), 1151
- modes* (*sfepy.terms.terms_navier_stokes.DivGradTerm attribute*), 1153
- modes* (*sfepy.terms.terms_navier_stokes.StokesTerm attribute*), 1159
- modes* (*sfepy.terms.terms_navier_stokes.StokesWaveDivTerm attribute*), 1160
- modes* (*sfepy.terms.terms_piezo.PiezoCouplingTerm attribute*), 1161
- modes* (*sfepy.terms.terms_sensitivity.ESDDiffusionTerm attribute*), 1165
- modes* (*sfepy.terms.terms_sensitivity.ESDDivGradTerm attribute*), 1165
- modes* (*sfepy.terms.terms_sensitivity.ESDDotTerm attribute*), 1166
- modes* (*sfepy.terms.terms_sensitivity.ESDLinearElasticTerm attribute*), 1167
- modes* (*sfepy.terms.terms_sensitivity.ESDLinearTractionTerm attribute*), 1167
- modes* (*sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm attribute*), 1168
- modes* (*sfepy.terms.terms_sensitivity.ESDStokesTerm attribute*), 1169
- modes* (*sfepy.terms.terms_surface.LinearTractionTerm attribute*), 1174
- modes* (*sfepy.terms.terms_surface.SurfaceNormalDotTerm attribute*), 1176
- modify_mesh()* (in *module sfepy.tests.test_term_sensitivity*), 1200
- module*
 - build_helpers*, 1201
 - gen_field_table*, 1203
 - gen_gallery*, 1203
 - gen_legendre_simplex_base*, 1205
 - gen_lobatto1d_c*, 1205
 - gen_release_notes*, 1205
 - gen_serendipity_basis*, 1205
 - gen_solver_table*, 1205
 - gen_term_table*, 1206
 - install_data*, 1206
 - sfepy.applications.application*, 778
 - sfepy.applications.evp_solver_app*, 778
 - sfepy.applications.pde_solver_app*, 779
 - sfepy.base.base*, 780
 - sfepy.base.compat*, 786
 - sfepy.base.conf*, 790
 - sfepy.base.getch*, 792
 - sfepy.base.goptions*, 793
 - sfepy.base.ioutils*, 793
 - sfepy.base.log*, 798
 - sfepy.base.log_plotter*, 800
 - sfepy.base.mem_usage*, 800
 - sfepy.base.multiproc*, 801
 - sfepy.base.multiproc_mpi*, 801
 - sfepy.base.multiproc_proc*, 804
 - sfepy.base.parse_conf*, 805
 - sfepy.base.plotutils*, 806
 - sfepy.base.reader*, 806
 - sfepy.base.resolve_deps*, 806
 - sfepy.base.testing*, 807
 - sfepy.base.timing*, 807
 - sfepy.config*, 777
 - sfepy.discrete.common.dof_info*, 857
 - sfepy.discrete.common.domain*, 860
 - sfepy.discrete.common.extmods._fmfield*, 861
 - sfepy.discrete.common.extmods._geommech*, 861
 - sfepy.discrete.common.extmods.assemble*, 861
 - sfepy.discrete.common.extmods.cmapping*, 861
 - sfepy.discrete.common.extmods.cmesh*, 861
 - sfepy.discrete.common.extmods.crefcoors*, 864
 - sfepy.discrete.common.fields*, 864
 - sfepy.discrete.common.global_interp*, 867
 - sfepy.discrete.common.mappings*, 870
 - sfepy.discrete.common.poly_spaces*, 872
 - sfepy.discrete.common.region*, 873
 - sfepy.discrete.conditions*, 808
 - sfepy.discrete.dg.dg_1D_vizualizer*, 914
 - sfepy.discrete.dg.fields*, 919
 - sfepy.discrete.dg.limiters*, 931
 - sfepy.discrete.dg.poly_spaces*, 928
 - sfepy.discrete.equations*, 811
 - sfepy.discrete.evaluate*, 818
 - sfepy.discrete.evaluate_variable*, 822
 - sfepy.discrete.fem._serendipity*, 912
 - sfepy.discrete.fem.domain*, 877
 - sfepy.discrete.fem.extmods.bases*, 878

sfepy.discrete.fem.extmods.lobatto_bases, 878
 sfepy.discrete.fem.facets, 879
 sfepy.discrete.fem.fe_surface, 880
 sfepy.discrete.fem.fields_base, 881
 sfepy.discrete.fem.fields_hierarchic, 887
 sfepy.discrete.fem.fields_l2, 887
 sfepy.discrete.fem.fields_nodal, 889
 sfepy.discrete.fem.fields_positive, 890
 sfepy.discrete.fem.geometry_element, 891
 sfepy.discrete.fem.history, 892
 sfepy.discrete.fem.lcbc_operators, 892
 sfepy.discrete.fem.linearizer, 895
 sfepy.discrete.fem.mappings, 896
 sfepy.discrete.fem.mesh, 898
 sfepy.discrete.fem.meshio, 900
 sfepy.discrete.fem.periodic, 907
 sfepy.discrete.fem.poly_spaces, 907
 sfepy.discrete.fem.refine, 911
 sfepy.discrete.fem.refine_hanging, 912
 sfepy.discrete.fem.utils, 912
 sfepy.discrete.functions, 822
 sfepy.discrete.iga.domain, 934
 sfepy.discrete.iga.domain_generators, 935
 sfepy.discrete.iga.extmods.igac, 936
 sfepy.discrete.iga.fields, 938
 sfepy.discrete.iga.iga, 940
 sfepy.discrete.iga.io, 946
 sfepy.discrete.iga.mappings, 947
 sfepy.discrete.iga.plot_nurbs, 948
 sfepy.discrete.iga.utils, 948
 sfepy.discrete.integrals, 823
 sfepy.discrete.materials, 824
 sfepy.discrete.parse_equations, 827
 sfepy.discrete.parse_regions, 827
 sfepy.discrete.probes, 828
 sfepy.discrete.problem, 832
 sfepy.discrete.projections, 844
 sfepy.discrete.quadratures, 845
 sfepy.discrete.simplex_cubature, 847
 sfepy.discrete.structural.fields, 949
 sfepy.discrete.structural.mappings, 950
 sfepy.discrete.variables, 847
 sfepy.homogenization.band_gaps_app, 950
 sfepy.homogenization.coefficients, 951
 sfepy.homogenization.coefs_base, 952
 sfepy.homogenization.coefs_phononic, 954
 sfepy.homogenization.convolution, 960
 sfepy.homogenization.engine, 960
 sfepy.homogenization.homogen_app, 963
 sfepy.homogenization.micmac, 964
 sfepy.homogenization.recovery, 964
 sfepy.homogenization.utils, 967
 sfepy.linalg.check_derivatives, 969
 sfepy.linalg.eigen, 969
 sfepy.linalg.geometry, 970
 sfepy.linalg.sparse, 973
 sfepy.linalg.sympy_operators, 974
 sfepy.linalg.utils, 975
 sfepy.mechanics.contact_bodies, 979
 sfepy.mechanics.elastic_constants, 979
 sfepy.mechanics.extmods.ccontres, 993
 sfepy.mechanics.matcoefs, 979
 sfepy.mechanics.membranes, 983
 sfepy.mechanics.shell10x, 986
 sfepy.mechanics.tensors, 988
 sfepy.mechanics.units, 991
 sfepy.mesh.bspline, 993
 sfepy.mesh.geom_tools, 999
 sfepy.mesh.mesh_generators, 1001
 sfepy.mesh.mesh_tools, 1004
 sfepy.mesh.splinbox, 1006
 sfepy.parallel.evaluate, 1007
 sfepy.parallel.parallel, 1008
 sfepy.parallel.plot_parallel_dofs, 1010
 sfepy.postprocess.plot_cmsh, 1010
 sfepy.postprocess.plot_dofs, 1011
 sfepy.postprocess.plot_facets, 1011
 sfepy.postprocess.plot_quadrature, 1012
 sfepy.postprocess.probes_vtk, 1012
 sfepy.postprocess.time_history, 1014
 sfepy.postprocess.utils_vtk, 1015
 sfepy.scripts.blockgen, 1182
 sfepy.scripts.convert_mesh, 1182
 sfepy.scripts.cylindergen, 1183
 sfepy.scripts.gen_iga_patch, 1183
 sfepy.scripts.gen_mesh, 1183
 sfepy.scripts.gen_mesh_prev, 1183
 sfepy.scripts.plot_condition_numbers, 1184
 sfepy.scripts.plot_logs, 1184
 sfepy.scripts.plot_mesh, 1184
 sfepy.scripts.plot_quadratures, 1184
 sfepy.scripts.plot_times, 1184
 sfepy.scripts.probe, 1184
 sfepy.scripts.resview, 1186
 sfepy.scripts.run_tests, 1187
 sfepy.scripts.simple, 1187
 sfepy.solvers.auto_fallback, 1016
 sfepy.solvers.eigen, 1017
 sfepy.solvers.ls, 1020
 sfepy.solvers.nls, 1027
 sfepy.solvers.optimize, 1031
 sfepy.solvers.oseen, 1033
 sfepy.solvers.qeigen, 1034
 sfepy.solvers.semismooth_newton, 1035
 sfepy.solvers.solvers, 1036
 sfepy.solvers.ts, 1038

`sfepy.solvers.ts_controllers`, 1039
`sfepy.solvers.ts_dg_solvers`, 932
`sfepy.solvers.ts_solvers`, 1042
`sfepy.terms.extmods.terms`, 1179
`sfepy.terms.terms`, 1050
`sfepy.terms.terms_adj_navier_stokes`, 1055
`sfepy.terms.terms_basic`, 1065
`sfepy.terms.terms_biot`, 1069
`sfepy.terms.terms_compat`, 1072
`sfepy.terms.terms_constraints`, 1075
`sfepy.terms.terms_contact`, 1077
`sfepy.terms.terms_dg`, 1079
`sfepy.terms.terms_diffusion`, 1084
`sfepy.terms.terms_dot`, 1090
`sfepy.terms.terms_elastic`, 1096
`sfepy.terms.terms_electric`, 1109
`sfepy.terms.terms_fibres`, 1110
`sfepy.terms.terms_flexo`, 1111
`sfepy.terms.terms_hyperelastic_base`, 1114
`sfepy.terms.terms_hyperelastic_tl`, 1115
`sfepy.terms.terms_hyperelastic_ul`, 1124
`sfepy.terms.terms_jax`, 1128
`sfepy.terms.terms_mass`, 1135
`sfepy.terms.terms_membrane`, 1136
`sfepy.terms.terms_multilinear`, 1137
`sfepy.terms.terms_navier_stokes`, 1152
`sfepy.terms.terms_piezo`, 1160
`sfepy.terms.terms_point`, 1163
`sfepy.terms.terms_sensitivity`, 1164
`sfepy.terms.terms_shells`, 1170
`sfepy.terms.terms_surface`, 1172
`sfepy.terms.terms_th`, 1177
`sfepy.terms.terms_volume`, 1177
`sfepy.terms.utils`, 1178
`sfepy.tests.conftest`, 1188
`sfepy.tests.test_assembling`, 1188
`sfepy.tests.test_base`, 1189
`sfepy.tests.test_cmesh`, 1189
`sfepy.tests.test_conditions`, 1189
`sfepy.tests.test_declarative_examples`, 1189
`sfepy.tests.test_dg_field`, 1190
`sfepy.tests.test_dg_terms_calls`, 1190
`sfepy.tests.test_domain`, 1191
`sfepy.tests.test_ed_solvers`, 1192
`sfepy.tests.test_eigenvalue_solvers`, 1192
`sfepy.tests.test_elasticity_small_strain`, 1192
`sfepy.tests.test_fem`, 1192
`sfepy.tests.test_functions`, 1193
`sfepy.tests.test_high_level`, 1193
`sfepy.tests.test_homogenization_engine`, 1193
`sfepy.tests.test_homogenization_perfusion`, 1193
`sfepy.tests.test_hyperelastic_tlul`, 1194
`sfepy.tests.test_io`, 1194
`sfepy.tests.test_laplace_unit_disk`, 1194
`sfepy.tests.test_laplace_unit_square`, 1194
`sfepy.tests.test_lcbcs`, 1194
`sfepy.tests.test_linalg`, 1194
`sfepy.tests.test_linear_solvers`, 1195
`sfepy.tests.test_linearization`, 1195
`sfepy.tests.test_log`, 1195
`sfepy.tests.test_matcoefs`, 1195
`sfepy.tests.test_mesh_expand`, 1195
`sfepy.tests.test_mesh_generators`, 1196
`sfepy.tests.test_mesh_interp`, 1196
`sfepy.tests.test_mesh_smoothing`, 1196
`sfepy.tests.test_meshio`, 1196
`sfepy.tests.test_msm_laplace`, 1197
`sfepy.tests.test_msm_symbolic`, 1197
`sfepy.tests.test_normals`, 1197
`sfepy.tests.test_parsing`, 1197
`sfepy.tests.test_poly_spaces`, 1197
`sfepy.tests.test_projections`, 1198
`sfepy.tests.test_quadratures`, 1198
`sfepy.tests.test_ref_coors`, 1198
`sfepy.tests.test_refine_hanging`, 1199
`sfepy.tests.test_regions`, 1199
`sfepy.tests.test_semismooth_newton`, 1199
`sfepy.tests.test_sparse`, 1199
`sfepy.tests.test_splinebox`, 1199
`sfepy.tests.test_tensors`, 1200
`sfepy.tests.test_term_call_modes`, 1200
`sfepy.tests.test_term_consistency`, 1200
`sfepy.tests.test_term_sensitivity`, 1200
`sfepy.tests.test_units`, 1201
`sfepy.tests.test_volume`, 1201
`sfepy.version`, 778
`show_authors`, 1206
`show_terms_use`, 1207
`sync_module_docs`, 1207
`MomentLimiter1D` (*class in sfepy.discrete.dg.limiters*), 931
`MomentLimiter2D` (*class in sfepy.discrete.dg.limiters*), 931
`MooneyRivlinTLTerm` (*class in sfepy.terms.terms_hyperelastic_tl*), 1119
`MooneyRivlinULTerm` (*class in sfepy.terms.terms_hyperelastic_ul*), 1126
`move_control_point()` (*sfepy.mesh.splinebox.SplineBox method*), 1007
`MPIFileHandler` (*class in sfepy.base.multiproc_mpi*), 801

MPILogFile (class in <i>sfepy.base.multiproc_mpi</i>), 802	name (sfepy.solvers.eigen.MatlabEigenvalueSolver attribute), 1018
MRLCBCOperator (class in <i>sfepy.discrete.fem.lcbc_operators</i>), 893	name (sfepy.solvers.eigen.OctaveEigenvalueSolver attribute), 1018
mulAB_integrate() (in module <i>sfepy.terms.extmods.terms</i>), 1182	name (sfepy.solvers.eigen.PrimmeEigenvalueSolver attribute), 1018
MultiNodeLCOperator (class in <i>sfepy.discrete.fem.lcbc_operators</i>), 894	name (sfepy.solvers.eigen.ScipyEigenvalueSolver attribute), 1020
MultiProblem (class in <i>sfepy.solvers.ls</i>), 1021	name (sfepy.solvers.eigen.ScipySGEEigenvalueSolver attribute), 1020
MUMPSSolver (class in <i>sfepy.solvers.ls</i>), 1020	name (sfepy.solvers.eigen.SLEPcEigenvalueSolver attribute), 1019
MyQueue (class in <i>sfepy.base.multiproc_proc</i>), 804	name (sfepy.solvers.ls.CholeskySolver attribute), 1020
N	name (sfepy.solvers.ls.MultiProblem attribute), 1021
n_coor (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 863	name (sfepy.solvers.ls.MUMPSSolver attribute), 1021
n_el (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 863	name (sfepy.solvers.ls.PETScKrylovSolver attribute), 1023
n_incident (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 861	name (sfepy.solvers.ls.PyAMGKrylovSolver attribute), 1023
name (sfepy.discrete.dg.limiters.DGLimiter attribute), 931	name (sfepy.solvers.ls.PyAMGSolver attribute), 1024
name (sfepy.discrete.dg.limiters.IdentityLimiter attribute), 931	name (sfepy.solvers.ls.PyPardisoSolver attribute), 1024
name (sfepy.discrete.dg.limiters.MomentLimiter1D attribute), 931	name (sfepy.solvers.ls.RMMSolver attribute), 1025
name (sfepy.discrete.dg.limiters.MomentLimiter2D attribute), 931	name (sfepy.solvers.ls.SchurMumps attribute), 1025
name (sfepy.discrete.dg.poly_spaces.LegendreSimplexPolySpace attribute), 930	name (sfepy.solvers.ls.ScipyDirect attribute), 1026
name (sfepy.discrete.dg.poly_spaces.LegendreTensorProductPolySpace attribute), 930	name (sfepy.solvers.ls.ScipyIterative attribute), 1026
name (sfepy.discrete.fem.poly_spaces.BernsteinSimplexPolySpace attribute), 907	name (sfepy.solvers.ls.ScipySuperLU attribute), 1027
name (sfepy.discrete.fem.poly_spaces.BernsteinTensorProductPolySpace attribute), 907	name (sfepy.solvers.ls.ScipyUmfpack attribute), 1027
name (sfepy.discrete.fem.poly_spaces.LagrangeSimplexBPolySpace attribute), 908	name (sfepy.solvers.nls.Newton attribute), 1029
name (sfepy.discrete.fem.poly_spaces.LagrangeSimplexPolySpace attribute), 908	name (sfepy.solvers.nls.PETScNonlinearSolver attribute), 1030
name (sfepy.discrete.fem.poly_spaces.LagrangeTensorProductPolySpace attribute), 908	name (sfepy.solvers.nls.ScipyRoot attribute), 1030
name (sfepy.discrete.fem.poly_spaces.LagrangeWedgePolySpace attribute), 908	name (sfepy.solvers.optimize.FMinSteepestDescent attribute), 1032
name (sfepy.discrete.fem.poly_spaces.LobattoTensorProductPolySpace attribute), 908	name (sfepy.solvers.optimize.ScipyFMinSolver attribute), 1033
name (sfepy.discrete.fem.poly_spaces.SEMTensorProductPolySpace attribute), 909	name (sfepy.solvers.oseen.Oseen attribute), 1034
name (sfepy.discrete.fem.poly_spaces.SerendipityTensorProductPolySpace attribute), 911	name (sfepy.solvers.qeigen.LQuadraticEVPSolver attribute), 1035
name (sfepy.solvers.auto_fallback.AutoDirect attribute), 1016	name (sfepy.solvers.semismooth_newton.SemismoothNewton attribute), 1036
name (sfepy.solvers.auto_fallback.AutoIterative attribute), 1017	name (sfepy.solvers.ts_controllers.ElastodynamicsBasicTSC attribute), 1039
name (sfepy.solvers.eigen.LOBPCGEigenvalueSolver attribute), 1017	name (sfepy.solvers.ts_controllers.ElastodynamicsLinearTSC attribute), 1040
	name (sfepy.solvers.ts_controllers.ElastodynamicsPIDTSC attribute), 1041
	name (sfepy.solvers.ts_controllers.FixedTSC attribute), 1041
	name (sfepy.solvers.ts_controllers.TimesSequenceTSC attribute), 1042
	name (sfepy.solvers.ts_dg_solvers.DGMultiStageTSS attribute), 932
	name (sfepy.solvers.ts_dg_solvers.EulerStepSolver attribute), 933

name (sfepy.solvers.ts_dg_solvers.RK4StepSolver attribute), 933	name (sfepy.terms.terms_basic.IntegrateMatTerm attribute), 1065
name (sfepy.solvers.ts_dg_solvers.TVDRK3StepSolver attribute), 933	name (sfepy.terms.terms_basic.IntegrateOperatorTerm attribute), 1066
name (sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver attribute), 1043	name (sfepy.terms.terms_basic.IntegrateTerm attribute), 1066
name (sfepy.solvers.ts_solvers.BatheTS attribute), 1044	name (sfepy.terms.terms_basic.SumNodalValuesTerm attribute), 1067
name (sfepy.solvers.ts_solvers.CentralDifferenceTS attribute), 1044	name (sfepy.terms.terms_basic.SurfaceMomentTerm attribute), 1067
name (sfepy.solvers.ts_solvers.GeneralizedAlphaTS attribute), 1046	name (sfepy.terms.terms_basic.VolumeSurfaceTerm attribute), 1068
name (sfepy.solvers.ts_solvers.NewmarkTS attribute), 1047	name (sfepy.terms.terms_basic.VolumeTerm attribute), 1068
name (sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver attribute), 1048	name (sfepy.terms.terms_basic.ZeroTerm attribute), 1069
name (sfepy.solvers.ts_solvers.StationarySolver attribute), 1048	name (sfepy.terms.terms_biot.BiotETHTerm attribute), 1070
name (sfepy.solvers.ts_solvers.VelocityVerletTS attribute), 1049	name (sfepy.terms.terms_biot.BiotStressTerm attribute), 1070
name (sfepy.terms.terms.Term attribute), 1053	name (sfepy.terms.terms_biot.BiotTerm attribute), 1072
name (sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term attribute), 1055	name (sfepy.terms.terms_biot.BiotTHTerm attribute), 1071
name (sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term attribute), 1056	name (sfepy.terms.terms_compat.CauchyStrainSTerm attribute), 1072
name (sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm attribute), 1056	name (sfepy.terms.terms_compat.DotSurfaceProductTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm attribute), 1057	name (sfepy.terms.terms_compat.DotVolumeProductTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDiffTerm attribute), 1057	name (sfepy.terms.terms_compat.DSumNodalValuesTerm attribute), 1072
name (sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDiffTerm attribute), 1058	name (sfepy.terms.terms_compat.DSurfaceFluxTerm attribute), 1072
name (sfepy.terms.terms_adj_navier_stokes.SDConvectTerm attribute), 1058	name (sfepy.terms.terms_compat.DSurfaceMomentTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm attribute), 1059	name (sfepy.terms.terms_compat.DVolumeSurfaceTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.SDDivTerm attribute), 1059	name (sfepy.terms.terms_compat.IntegrateSurfaceMatTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.SDDotTerm attribute), 1060	name (sfepy.terms.terms_compat.IntegrateSurfaceOperatorTerm attribute), 1073
name (sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm attribute), 1060	name (sfepy.terms.terms_compat.IntegrateSurfaceTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm attribute), 1061	name (sfepy.terms.terms_compat.IntegrateVolumeMatTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SDPSPGPGStabilizationTerm attribute), 1062	name (sfepy.terms.terms_compat.IntegrateVolumeOperatorTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm attribute), 1063	name (sfepy.terms.terms_compat.IntegrateVolumeTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SUPGCAdjStabilizationTerm attribute), 1063	name (sfepy.terms.terms_compat.SDVolumeDotTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SUPGPAdj1StabilizationTerm attribute), 1064	name (sfepy.terms.terms_compat.SurfaceDivTerm attribute), 1074
name (sfepy.terms.terms_adj_navier_stokes.SUPGPAdj2StabilizationTerm attribute), 1064	name (sfepy.terms.terms_compat.SurfaceGradTerm attribute), 1075

name (sfepy.terms.terms_compat.SurfaceTerm attribute), 1075	name (sfepy.terms.terms_dot.ScalarDotMGradScalarTerm attribute), 1094
name (sfepy.terms.terms_compat.VolumeXTerm attribute), 1075	name (sfepy.terms.terms_dot.VectorDotGradScalarTerm attribute), 1095
name (sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm attribute), 1076	name (sfepy.terms.terms_dot.VectorDotScalarTerm attribute), 1096
name (sfepy.terms.terms_constraints.NonPenetrationTerm attribute), 1076	name (sfepy.terms.terms_elastic.CauchyStrainTerm attribute), 1097
name (sfepy.terms.terms_contact.ContactIPCTerm attribute), 1077	name (sfepy.terms.terms_elastic.CauchyStressETHTerm attribute), 1097
name (sfepy.terms.terms_contact.ContactTerm attribute), 1078	name (sfepy.terms.terms_elastic.CauchyStressTerm attribute), 1099
name (sfepy.terms.terms_dg.AdvectionDGFluxTerm attribute), 1080	name (sfepy.terms.terms_elastic.CauchyStressTHTerm attribute), 1098
name (sfepy.terms.terms_dg.DiffusionDGFluxTerm attribute), 1081	name (sfepy.terms.terms_elastic.ElasticWaveCauchyTerm attribute), 1100
name (sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm attribute), 1082	name (sfepy.terms.terms_elastic.ElasticWaveTerm attribute), 1100
name (sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm attribute), 1083	name (sfepy.terms.terms_elastic.LinearDRotSpringTerm attribute), 1101
name (sfepy.terms.terms_dg.NonlinearScalarDotGradTerm attribute), 1084	name (sfepy.terms.terms_elastic.LinearDSpringTerm attribute), 1102
name (sfepy.terms.terms_diffusion.AdvectDivFreeTerm attribute), 1084	name (sfepy.terms.terms_elastic.LinearElasticETHTerm attribute), 1102
name (sfepy.terms.terms_diffusion.ConvectVGradSTerm attribute), 1085	name (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm attribute), 1103
name (sfepy.terms.terms_diffusion.DiffusionCoupling attribute), 1085	name (sfepy.terms.terms_elastic.LinearElasticTerm attribute), 1104
name (sfepy.terms.terms_diffusion.DiffusionRTerm attribute), 1086	name (sfepy.terms.terms_elastic.LinearElasticTHTerm attribute), 1103
name (sfepy.terms.terms_diffusion.DiffusionTerm attribute), 1087	name (sfepy.terms.terms_elastic.LinearPrestressTerm attribute), 1105
name (sfepy.terms.terms_diffusion.DiffusionVelocityTerm attribute), 1087	name (sfepy.terms.terms_elastic.LinearSpringTerm attribute), 1105
name (sfepy.terms.terms_diffusion.LaplaceTerm attribute), 1088	name (sfepy.terms.terms_elastic.LinearStrainFiberTerm attribute), 1106
name (sfepy.terms.terms_diffusion.NonlinearDiffusionTerm attribute), 1088	name (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm attribute), 1107
name (sfepy.terms.terms_diffusion.SDDiffusionTerm attribute), 1089	name (sfepy.terms.terms_elastic.LinearTrussTerm attribute), 1107
name (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm attribute), 1090	name (sfepy.terms.terms_elastic.NonsymElasticTerm attribute), 1108
name (sfepy.terms.terms_diffusion.SurfaceFluxTerm attribute), 1090	name (sfepy.terms.terms_elastic.SDLinearElasticTerm attribute), 1109
name (sfepy.terms.terms_dot.BCNewtonTerm attribute), 1091	name (sfepy.terms.terms_electric.ElectricSourceTerm attribute), 1109
name (sfepy.terms.terms_dot.DotProductTerm attribute), 1092	name (sfepy.terms.terms_fibres.FibresActiveTLTerm attribute), 1110
name (sfepy.terms.terms_dot.DotSPProductVolumeOperatorVolumeTerm attribute), 1093	name (sfepy.terms.terms_fibres.FibresExponentialTLTerm attribute), 1111
name (sfepy.terms.terms_dot.DotSPProductVolumeOperatorVolumeTerm attribute), 1093	name (sfepy.terms.terms_flexo.MixedFlexoCouplingTerm attribute), 1112
name (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm attribute), 1094	name (sfepy.terms.terms_flexo.MixedFlexoTerm attribute), 1113

name (sfepy.terms.terms_flexo.MixedStrainGradElasticTerm attribute), 1136
name (sfepy.terms.terms_flexo.MixedStrainGradElasticTerm attribute), 1113
name (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm attribute), 1137
name (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm attribute), 1114
name (sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm attribute), 1116
name (sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm attribute), 1116
name (sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm attribute), 1117
name (sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm attribute), 1118
name (sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm attribute), 1118
name (sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm attribute), 1119
name (sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm attribute), 1120
name (sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm attribute), 1121
name (sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm attribute), 1121
name (sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm attribute), 1122
name (sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm attribute), 1123
name (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm attribute), 1123
name (sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm attribute), 1124
name (sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm attribute), 1125
name (sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm attribute), 1125
name (sfepy.terms.terms_hyperelastic_ul.HyperelasticByFunULTerm attribute), 1126
name (sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm attribute), 1127
name (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm attribute), 1127
name (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm attribute), 1128
name (sfepy.terms.terms_jax.LinearElasticLADTerm attribute), 1129
name (sfepy.terms.terms_jax.LinearElasticYPADTerm attribute), 1129
name (sfepy.terms.terms_jax.MassADTerm attribute), 1130
name (sfepy.terms.terms_jax.NeoHookeanTLADTerm attribute), 1131
name (sfepy.terms.terms_jax.OgdenTLADTerm attribute), 1132
name (sfepy.terms.terms_mass.MassTerm attribute), 1135
name (sfepy.terms.terms_membrane.TLMembraneTerm attribute), 1136
name (sfepy.terms.terms_multilinear.ECauchyStressTerm attribute), 1137
name (sfepy.terms.terms_multilinear.EConvectTerm attribute), 1138
name (sfepy.terms.terms_multilinear.EDiffusionTerm attribute), 1138
name (sfepy.terms.terms_multilinear.EDivGradTerm attribute), 1139
name (sfepy.terms.terms_multilinear.EDivTerm attribute), 1139
name (sfepy.terms.terms_multilinear.EDotTerm attribute), 1140
name (sfepy.terms.terms_multilinear.EGradTerm attribute), 1140
name (sfepy.terms.terms_multilinear.EIntegrateOperatorTerm attribute), 1141
name (sfepy.terms.terms_multilinear.ELaplaceTerm attribute), 1142
name (sfepy.terms.terms_multilinear.ELinearConvectTerm attribute), 1142
name (sfepy.terms.terms_multilinear.ELinearElasticTerm attribute), 1143
name (sfepy.terms.terms_multilinear.ELinearTractionTerm attribute), 1143
name (sfepy.terms.terms_multilinear.ENonPenetrationPenaltyTerm attribute), 1144
name (sfepy.terms.terms_multilinear.ENonSymElasticTerm attribute), 1145
name (sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm attribute), 1145
name (sfepy.terms.terms_multilinear.EStokesTerm attribute), 1146
name (sfepy.terms.terms_multilinear.StokesTractionTerm attribute), 1149
name (sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm attribute), 1150
name (sfepy.terms.terms_multilinear.SurfacePiezoFluxOperatorTerm attribute), 1150
name (sfepy.terms.terms_multilinear.SurfacePiezoFluxTerm attribute), 1151
name (sfepy.terms.terms_navier_stokes.ConvectTerm attribute), 1152
name (sfepy.terms.terms_navier_stokes.DivGradTerm attribute), 1153
name (sfepy.terms.terms_navier_stokes.DivOperatorTerm attribute), 1153
name (sfepy.terms.terms_navier_stokes.DivTerm attribute), 1154
name (sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm attribute), 1154
name (sfepy.terms.terms_navier_stokes.GradTerm attribute), 1155
name (sfepy.terms.terms_navier_stokes.LinearConvect2Term attribute), 1155

attribute), 1155	attribute), 1175
name (sfepy.terms.terms_navier_stokes.LinearConvectTerm attribute), 1156	name (sfepy.terms.terms_surface.SDSufaceIntegrateTerm attribute), 1176
name (sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm attribute), 1156	name (sfepy.terms.terms_surface.SufaceNormalDotTerm attribute), 1176
name (sfepy.terms.terms_navier_stokes.PSPGPStabilizationTerm attribute), 1157	name (sfepy.terms.terms_surface.SurfaceJumpTerm attribute), 1177
name (sfepy.terms.terms_navier_stokes.StokesTerm attribute), 1159	name (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 1178
name (sfepy.terms.terms_navier_stokes.StokesWaveDivTerm attribute), 1160	name (sfepy.terms.terms_volume.NonlinearVolumeForceTerm attribute), 1178
name (sfepy.terms.terms_navier_stokes.StokesWaveTerm attribute), 1160	NeoHookeanTLADTerm (class in sfepy.terms.terms_jax), 1130
name (sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm attribute), 1157	NeoHookeanTLTerm (class in sfepy.terms.terms_hyperelastic_tl), 1119
name (sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm attribute), 1158	NeoHookeanULTerm (class in sfepy.terms.terms_hyperelastic_ul), 1127
name (sfepy.terms.terms_piezo.PiezoCouplingTerm attribute), 1161	NEUMeshIO (class in sfepy.discrete.fem.meshio), 906
name (sfepy.terms.terms_piezo.PiezoStrainTerm attribute), 1161	new() (sfepy.terms.terms.Term static method), 1053
name (sfepy.terms.terms_piezo.PiezoStressTerm attribute), 1162	new_ulf_iteration() (sfepy.discrete.evaluate.Evaluator static method), 818
name (sfepy.terms.terms_piezo.SDPiezoCouplingTerm attribute), 1163	new_vtk_polyline() (sfepy.postprocess.probes_vtk.Probe method), 1013
name (sfepy.terms.terms_point.ConcentratedPointLoadTerm attribute), 1163	NewmarkTS (class in sfepy.solvers.ts_solvers), 1046
name (sfepy.terms.terms_point.LinearPointSpringTerm attribute), 1164	Newton (class in sfepy.solvers.nls), 1027
name (sfepy.terms.terms_sensitivity.ESDDiffusionTerm attribute), 1165	NLSStatus (class in sfepy.base.testing), 807
name (sfepy.terms.terms_sensitivity.ESDDivGradTerm attribute), 1165	NodalLCOperator (class in sfepy.discrete.fem.lcbc_operators), 894
name (sfepy.terms.terms_sensitivity.ESDDotTerm attribute), 1166	NodeDescription (class in sfepy.discrete.fem.poly_spaces), 908
name (sfepy.terms.terms_sensitivity.ESDLinearElasticTerm attribute), 1167	NonlinearDiffusionTerm (class in sfepy.terms.terms_diffusion), 1088
name (sfepy.terms.terms_sensitivity.ESDLinearTractionTerm attribute), 1167	NonlinearHyperbolicDGFluxTerm (class in sfepy.terms.terms_dg), 1082
name (sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm attribute), 1168	NonlinearScalarDotGradTerm (class in sfepy.terms.terms_dg), 1083
name (sfepy.terms.terms_sensitivity.ESDStokesTerm attribute), 1169	NonlinearSolver (class in sfepy.solvers.solvers), 1036
name (sfepy.terms.terms_sensitivity.ESDVectorDotGradScalarTerm attribute), 1169	NonlinearVolumeForceTerm (class in sfepy.terms.terms_volume), 1178
name (sfepy.terms.terms_shells.Shell10XTerm attribute), 1171	NonPenetrationPenaltyTerm (class in sfepy.terms.terms_constraints), 1075
name (sfepy.terms.terms_surface.ContactPlaneTerm attribute), 1173	NonPenetrationTerm (class in sfepy.terms.terms_constraints), 1076
name (sfepy.terms.terms_surface.ContactSphereTerm attribute), 1174	NonsymElasticTerm (class in sfepy.terms.terms_elastic), 1107
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 1174	NoOptionsDocs (class in build_helpers), 1201
name (sfepy.terms.terms_surface.SDLinearTractionTerm attribute), 1175	NoPenetrationOperator (class in sfepy.discrete.fem.lcbc_operators), 894
	norm_l2_along_axis() (in module sfepy.linalg.utils), 978
	NormalDirectionOperator (class in sfepy.discrete.fem.lcbc_operators), 894
	normalize_time() (sfepy.solvers.ts.TimeStepper

- method), 1038
- normalize_vectors() (in module *sfepy.linalg.utils*), 978
- NSOFMinGradTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1056
- NSOFSurfMinDPresDiffTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1057
- NSOFSurfMinDPresTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1057
- num (*sfepy.discrete.common.extmods.cmesh.CConnectivity* attribute), 861
- num (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 863
- numpydoc_path() (*sfepy.config.Config* method), 777
- NurbsPatch (class in *sfepy.discrete.iga.domain*), 934
- ## O
- OctaveEigenvalueSolver (class in *sfepy.solvers.eigen*), 1018
- offset (*sfepy.discrete.common.extmods.cmesh.CConnectivity* attribute), 861
- offsets (*sfepy.discrete.common.extmods.cmesh.CConnectivity* attribute), 862
- OgdenTLADTerm (class in *sfepy.terms.terms_jax*), 1131
- OgdenTLTerm (class in *sfepy.terms.terms_hyperelastic_tl*), 1120
- OnesDim (class in *sfepy.homogenization.coefs_base*), 954
- OneTypeList (class in *sfepy.base.base*), 781
- OptimizationSolver (class in *sfepy.solvers.solvers*), 1036
- OptsToListAction (class in *sfepy.scripts.resview*), 1186
- ordered_iteritems() (in module *sfepy.base.base*), 785
- orient_elements() (in module *sfepy.discrete.common.extmods.cmesh*), 863
- Oseen (class in *sfepy.solvers.oseen*), 1033
- Output (class in *sfepy.base.base*), 781
- output (*sfepy.base.log_plotter.LogPlotter* attribute), 800
- output_array_stats() (in module *sfepy.linalg.utils*), 978
- output_dir() (in module *sfepy.tests.conftest*), 1188
- output_mesh_formats() (in module *sfepy.discrete.fem.meshio*), 906
- output_step_info() (*sfepy.solvers.ts_dg_solvers.DGMultiStageTSSolver* method), 932
- output_step_info() (*sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver* method), 1043
- output_step_info() (*sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver* method), 1048
- ## P
- package_check() (in module *build_helpers*), 1202
- parametrize() (*sfepy.applications.application.Application* method), 778
- parse_approx_order() (in module *sfepy.discrete.common.fields*), 867
- parse_approx_order() (in module *sfepy.discrete.iga.fields*), 940
- parse_definition() (in module *sfepy.discrete.equations*), 818
- parse_options() (in module *sfepy.scripts.resview*), 1187
- parse_report() (*sfepy.discrete.probes.CircleProbe* static method), 828
- parse_report() (*sfepy.discrete.probes.LineProbe* static method), 828
- parse_report() (*sfepy.discrete.probes.PointsProbe* static method), 829
- parse_report() (*sfepy.discrete.probes.RayProbe* static method), 830
- parse_scalar() (in module *sfepy.discrete.probes*), 831
- parse_shape() (in module *sfepy.discrete.common.fields*), 867
- parse_term_expression() (in module *sfepy.terms.terms_multilinear*), 1151
- parse_vector() (in module *sfepy.discrete.probes*), 831
- ParseRc (class in *sfepy.scripts.plot_logs*), 1184
- partition_mesh() (in module *sfepy.parallel.parallel*), 1009
- path_of_hdf5_group() (in module *sfepy.base.ioutils*), 796
- pause() (in module *sfepy.base.base*), 785
- PDESolverApp (class in *sfepy.applications.pde_solver_app*), 779
- PeriodicBC (class in *sfepy.discrete.conditions*), 810
- petsc_call() (in module *sfepy.solvers.ls*), 1027
- PETScKrylovSolver (class in *sfepy.solvers.ls*), 1022
- PETScNonlinearSolver (class in *sfepy.solvers.nls*), 1029
- PETScParallelEvaluator (class in *sfepy.parallel.evaluate*), 1007
- PhaseVelocity (class in *sfepy.homogenization.coefs_phononic*), 957
- PhysicalQPs (class in *sfepy.discrete.common.mappings*), 870
- physicalsurface (class in *sfepy.mesh.geom_tools*), 1000
- physicalvolume (class in *sfepy.mesh.geom_tools*), 1000
- PiezoeCouplingTerm (class in *sfepy.terms.terms_piezo*), 1160
- PiezoeStrainTerm (class in *sfepy.terms.terms_piezo*), 1161
- PiezoStressTerm (class in *sfepy.terms.terms_piezo*), 1162
- plot1D_legendre_dofs() (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 916
- plot_band_gaps() (*sfepy.homogenization.band_gaps_app.AcousticBandCalculation* method), 950

<code>plot_bezier_mesh()</code> (in <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>plot_polys()</code> (in module <code>gen_lobatto1d_c</code>), 1205
<code>plot_bezier_nurbs_basis_1d()</code> (in <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>plot_quadrature()</code> (in module <code>sfepy.postprocess.plot_quadrature</code>), 1012
<code>plot_cmesh()</code> (in module <code>sfepy.postprocess.plot_cmesh</code>), 1010	<code>plot_vlines()</code> (<code>sfepy.base.log.Log</code> method), 799
<code>plot_control_mesh()</code> (in module <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>plot_weighted_points()</code> (in module <code>sfepy.postprocess.plot_quadrature</code>), 1012
<code>plot_data()</code> (<code>sfepy.base.log.Log</code> method), 799	<code>plot_wireframe()</code> (in module <code>sfepy.postprocess.plot_cmesh</code>), 1011
<code>plot_dispersion()</code> (<code>sfepy.homogenization.band_gaps_app.AcousticBandGapsApp.dg_1D_vizualizer</code>), 916	<code>plotsXT()</code> (in module <code>sfepy.postprocess.plot_cmesh</code>), 1011
<code>method</code>), 950	<code>point</code> (class in <code>sfepy.mesh.geom_tools</code>), 1000
<code>plot_edges()</code> (in module <code>sfepy.postprocess.plot_facets</code>), 1011	<code>points_in_poly()</code> (<code>sfepy.mesh.splinbox.SplineRegion2D</code> static method), 1007
<code>plot_eigs()</code> (in module <code>sfepy.homogenization.band_gaps_app</code>), 951	<code>points_in_simplex()</code> (in module <code>sfepy.linalg.geometry</code>), 973
<code>plot_entities()</code> (in module <code>sfepy.postprocess.plot_cmesh</code>), 1011	<code>PointsProbe</code> (class in <code>sfepy.discrete.probes</code>), 828
<code>plot_faces()</code> (in module <code>sfepy.postprocess.plot_facets</code>), 1011	<code>PolarizationAngles</code> (class in <code>sfepy.homogenization.coefs_phononic</code>), 957
<code>plot_gap()</code> (in module <code>sfepy.homogenization.band_gaps_app</code>), 951	<code>poll_draw()</code> (<code>sfepy.base.log_plotter.LogPlotter</code> method), 800
<code>plot_gaps()</code> (in module <code>sfepy.homogenization.band_gaps_app</code>), 951	<code>poly_space_basis</code> (<code>sfepy.terms.terms_dg.DGTerm</code> attribute), 1080
<code>plot_geometry()</code> (in module <code>sfepy.postprocess.plot_facets</code>), 1012	<code>poly_space_basis</code> (<code>sfepy.terms.terms_shells.Shell10XTerm</code> attribute), 1171
<code>plot_global_dofs()</code> (in module <code>sfepy.postprocess.plot_dofs</code>), 1011	<code>PolySpace</code> (class in <code>sfepy.discrete.common.poly_spaces</code>), 872
<code>plot_iso_lines()</code> (in module <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>post_process()</code> (<code>sfepy.homogenization.coefs_phononic.SchurEVP</code> method), 957
<code>plot_local_dofs()</code> (in module <code>sfepy.parallel.plot_parallel_dofs</code>), 1010	<code>post_process()</code> (<code>sfepy.homogenization.coefs_phononic.SimpleEVP</code> method), 957
<code>plot_local_dofs()</code> (in module <code>sfepy.postprocess.plot_dofs</code>), 1011	<code>postprocess()</code> (in module <code>sfepy.scripts.probe</code>), 1185
<code>plot_log()</code> (in module <code>sfepy.base.log</code>), 799	<code>prefix</code> (<code>sfepy.base.base.Output</code> property), 781
<code>plot_logs()</code> (in module <code>sfepy.homogenization.band_gaps_app</code>), 951	<code>prepare_cylindrical_transform()</code> (in module <code>sfepy.mechanics.tensors</code>), 989
<code>plot_matrix_diff()</code> (in module <code>sfepy.base.plotutils</code>), 806	<code>prepare_dgfield()</code> (in module <code>sfepy.tests.test_dg_field</code>), 1190
<code>plot_mesh()</code> (in module <code>sfepy.postprocess.plot_dofs</code>), 1011	<code>prepare_dgfield_1D()</code> (in module <code>sfepy.tests.test_dg_field</code>), 1190
<code>plot_nodes()</code> (in module <code>sfepy.postprocess.plot_dofs</code>), 1011	<code>prepare_field_2D()</code> (in module <code>sfepy.tests.test_dg_field</code>), 1190
<code>plot_nurbs_basis_1d()</code> (in module <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>prepare_materials()</code> (<code>sfepy.tests.test_dg_terms_calls.DGTermTestEnvornment</code> method), 1190
<code>plot_parametric_mesh()</code> (in module <code>sfepy.discrete.iga.plot_nurbs</code>), 948	<code>prepare_matrices()</code> (<code>sfepy.homogenization.coefs_phononic.SchurEVP</code> method), 957
<code>plot_partitioning()</code> (in module <code>sfepy.parallel.plot_parallel_dofs</code>), 1010	<code>prepare_matrices()</code> (<code>sfepy.homogenization.coefs_phononic.SimpleEVP</code> method), 957
<code>plot_points()</code> (in module <code>sfepy.mechanics.contact_bodies</code>), 979	<code>prepare_matrix()</code> (in module <code>sfepy.discrete.problem</code>), 844
<code>plot_points()</code> (in module <code>sfepy.postprocess.plot_dofs</code>), 1011	<code>prepare_remap()</code> (in module <code>sfepy.discrete.fem.utils</code>), 913
<code>plot_polygon()</code> (in module <code>sfepy.mechanics.contact_bodies</code>), 979	<code>prepare_translate()</code> (in module <code>sfepy.discrete.fem.utils</code>), 913
	<code>prepare_variable()</code> (in module <code>sfepy.mechanics.contact_bodies</code>), 979

`sfepy.tests.test_mesh_interp`), 1196
`prepare_variables()`
 (`sfepy.tests.test_dg_terms_calls.DGTermTestEnv` `problem` `method`), 1190
`presolve()` (`sfepy.solvers.ls.CholeskySolver` `method`), 1020
`presolve()` (`sfepy.solvers.ls.MUMPSSolver` `method`), 1021
`presolve()` (`sfepy.solvers.ls.PyPardisoSolver` `method`), 1024
`presolve()` (`sfepy.solvers.ls.ScipyDirect` `method`), 1026
`presolve()` (`sfepy.solvers.solvers.LinearSolver` `method`), 1036
`PrimeEigenvalueSolver` (`class` `in` `sfepy.solvers.eigen`), 1018
`print_array_info()` (`in` `module` `sfepy.linalg.utils`), 978
`print_camera_position()` (`in` `module` `sfepy.scripts.resview`), 1187
`print_expressions()`
 (`sfepy.terms.terms_multilinear.ETermBase` `method`), 1147
`print_leaf()` (`in` `module` `sfepy.discrete.parse_regions`), 827
`print_matrix_diff()` (`in` `module` `sfepy.base.plotutils`), 806
`print_mem_usage()` (`in` `module` `sfepy.base.mem_usage`), 801
`print_names()` (`sfepy.base.base.Container` `method`), 780
`print_names()` (`sfepy.base.base.OneTypeList` `method`), 781
`print_op()` (`in` `module` `sfepy.discrete.parse_regions`), 827
`print_shapes()` (`sfepy.terms.terms_multilinear.ExpressionBuilder` `method`), 1148
`print_solvers()` (`in` `module` `sfepy.scripts.simple`), 1188
`print_stack()` (`in` `module` `sfepy.discrete.parse_regions`), 827
`print_structs()` (`in` `module` `sfepy.base.base`), 785
`print_terms()` (`in` `module` `sfepy.scripts.simple`), 1188
`print_terms()` (`sfepy.discrete.equations.Equations` `method`), 816
`printinfo()` (`sfepy.mesh.geom_tools.geometry` `method`), 1000
`Probe` (`class` `in` `sfepy.discrete.probes`), 829
`Probe` (`class` `in` `sfepy.postprocess.probes_vtk`), 1012
`probe()` (`sfepy.discrete.probes.Probe` `method`), 829
`ProbeFromFile` (`class` `in` `sfepy.postprocess.probes_vtk`), 1014
`Problem` (`class` `in` `sfepy.discrete.problem`), 832
`problem()` (`in` `module` `sfepy.tests.test_ed_solvers`), 1192
`problem()` (`in` `module` `sfepy.tests.test_functions`), 1193
`problem()` (`in` `module` `sfepy.tests.test_linear_solvers`), 1195
`problem()` (`in` `module` `sfepy.tests.test_msm_laplace`), 1197
`problem()` (`in` `module` `sfepy.tests.test_msm_symbolic`), 1197
`problem()` (`in` `module` `sfepy.tests.test_term_consistency`), 1200
`problem()` (`in` `module` `sfepy.tests.test_term_sensitivity`), 1200
`problem()` (`in` `module` `sfepy.tests.test_volume`), 1201
`ProblemConf` (`class` `in` `sfepy.base.conf`), 790
`process_command()` (`sfepy.base.log_plotter.LogPlotter` `method`), 800
`process_conf()` (`sfepy.solvers.solvers.Solver` `class` `method`), 1037
`process_options()` (`sfepy.applications.evp_solver_app.EVPSolverApp` `static` `method`), 778
`process_options()` (`sfepy.applications.pde_solver_app.PDESolverApp` `static` `method`), 779
`process_options()` (`sfepy.homogenization.band_gaps_app.AcousticBandGapsApp` `static` `method`), 950
`process_options()` (`sfepy.homogenization.coefs_base.MiniAppBase` `method`), 954
`process_options()` (`sfepy.homogenization.coefs_phononic.BandGaps` `method`), 956
`process_options()` (`sfepy.homogenization.coefs_phononic.ChristoffelAcoustic` `method`), 956
`process_options()` (`sfepy.homogenization.coefs_phononic.Eigenmomentum` `method`), 957
`process_options()` (`sfepy.homogenization.coefs_phononic.PhaseVelocity` `method`), 957
`process_options()` (`sfepy.homogenization.coefs_phononic.PolarizationA` `method`), 957
`process_options()` (`sfepy.homogenization.coefs_phononic.SimpleEVP` `method`), 957
`process_options()` (`sfepy.homogenization.engine.HomogenizationEngine` `static` `method`), 961
`process_options()` (`sfepy.homogenization.homogen_app.HomogenizationApp` `static` `method`), 963
`process_options_pv()`
 (`sfepy.homogenization.band_gaps_app.AcousticBandGapsApp` `static` `method`), 950
`process_reqs_coefs()`
 (`sfepy.homogenization.engine.HomogenizationWorkerMulti` `static` `method`), 963
`project_by_component()` (`in` `module` `sfepy.discrete.projections`), 845
`project_to_facets()` (`in` `module` `sfepy.discrete.projections`), 845
`PSPGCStabilizationTerm` (`class` `in` `sfepy.terms.terms_navier_stokes`), 1156
`PSPGPStabilizationTerm` (`class` `in` `sfepy.terms.terms_navier_stokes`), 1156
`put()` (`sfepy.base.multiproc_mpi.RemoteQueue` `method`), 802

- put() (*sfepy.base.multiproc_mpi.RemoteQueueMaster* method), 803
 put() (*sfepy.base.multiproc_proc.MyQueue* method), 804
 pv_plot() (in module *sfepy.scripts.resview*), 1187
 PyAMGKrylovSolver (class in *sfepy.solvers.ls*), 1023
 PyAMGSolver (class in *sfepy.solvers.ls*), 1023
 PyCMapping (class in *sfepy.discrete.common.mappings*), 870
 PyPardisoSolver (class in *sfepy.solvers.ls*), 1024
 pytest_addoption() (in module *sfepy.tests.conftest*), 1188
 pytest_configure() (in module *sfepy.tests.conftest*), 1188
 python_include() (*sfepy.config.Config* method), 777
 python_shell() (in module *sfepy.base.base*), 785
 python_version() (*sfepy.config.Config* method), 777
- ## Q
- QuadraticEVPSolver (class in *sfepy.solvers.solvers*), 1036
 QuadraturePoints (class in *sfepy.discrete.quadratures*), 846
 Quantity (class in *sfepy.mechanics.units*), 991
- ## R
- R (*sfepy.discrete.iga.extmods.igac.CNURBSContext* attribute), 936
 raise_if_too_large() (in module *sfepy.base.mem_usage*), 801
 RayProbe (class in *sfepy.discrete.probes*), 830
 read() (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* method), 900
 read() (*sfepy.discrete.fem.meshio.ComsolMeshIO* method), 900
 read() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read() (*sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO* method), 903
 read() (*sfepy.discrete.fem.meshio.Mesh3DMeshIO* method), 904
 read() (*sfepy.discrete.fem.meshio.MeshIO* method), 905
 read() (*sfepy.discrete.fem.meshio.MeshioLibIO* method), 905
 read() (*sfepy.discrete.fem.meshio.NEUMeshIO* method), 906
 read() (*sfepy.discrete.fem.meshio.UserMeshIO* method), 906
 read() (*sfepy.discrete.fem.meshio.XYZMeshIO* method), 906
 read_array() (in module *sfepy.base.ioutils*), 796
 read_bounding_box() (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* method), 900
 read_bounding_box() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read_bounding_box() (*sfepy.discrete.fem.meshio.MeshioLibIO* method), 905
 read_bounding_box() (*sfepy.discrete.fem.meshio.XYZMeshIO* method), 906
 read_data() (*sfepy.discrete.fem.meshio.GmshIO* method), 900
 read_data() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read_data() (*sfepy.discrete.fem.meshio.MeshIO* method), 905
 read_data() (*sfepy.discrete.fem.meshio.MeshioLibIO* method), 905
 read_data_header() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read_dict_hdf5() (in module *sfepy.base.ioutils*), 796
 read_dimension() (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* method), 900
 read_dimension() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read_dimension() (*sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO* method), 903
 read_dimension() (*sfepy.discrete.fem.meshio.Mesh3DMeshIO* method), 904
 read_dimension() (*sfepy.discrete.fem.meshio.MeshioLibIO* method), 905
 read_dimension() (*sfepy.discrete.fem.meshio.NEUMeshIO* method), 906
 read_dimension() (*sfepy.discrete.fem.meshio.XYZMeshIO* method), 906
 read_domain_from_hdf5() (*sfepy.discrete.iga.domain.IGDomain* static method), 934
 read_from_hdf5() (in module *sfepy.base.ioutils*), 796
 read_header() (in module *sfepy.discrete.probes*), 831
 read_iga_data() (in module *sfepy.discrete.iga.io*), 946
 read_last_step() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 902
 read_last_step() (*sfepy.discrete.fem.meshio.MeshIO* method), 905
 read_list() (in module *sfepy.base.ioutils*), 796
 read_log() (in module *sfepy.base.log*), 799
 read_mesh() (in module *sfepy.scripts.resview*), 1187
 read_mesh_from_hdf5() (*sfepy.discrete.fem.meshio.HDF5MeshIO* static method), 902
 read_probes_as_annotations() (in module *sfepy.scripts.resview*), 1187
 read_results() (in module *sfepy.discrete.probes*), 831
 read_sparse_matrix_from_hdf5() (in module

sfepy.base.ioutils), 797
read_sparse_matrix_hdf5() (in module *sfepy.base.ioutils*), 797
read_time_history() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 903
read_time_stepper() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 903
read_times() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 903
read_times() (*sfepy.discrete.fem.meshio.MeshIO* method), 905
read_token() (in module *sfepy.base.ioutils*), 797
read_variables_time_history() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 903
Reader (class in *sfepy.base.reader*), 806
reconstruct_legendre_dofs() (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 917
recover_bones() (in module *sfepy.homogenization.recovery*), 966
recover_micro_hook() (in module *sfepy.homogenization.recovery*), 966
recover_paraflow() (in module *sfepy.homogenization.recovery*), 967
recursive_glob() (in module *build_helpers*), 1203
reduce_on_datas() (*sfepy.discrete.materials.Material* method), 825
reduce_vec() (*sfepy.discrete.equations.Equations* method), 816
reduce_vec() (*sfepy.discrete.variables.Variables* method), 855
refine() (in module *sfepy.discrete.fem.refine_hanging*), 912
refine() (in module *sfepy.tests.test_domain*), 1191
refine() (*sfepy.discrete.fem.domain.FEDomain* method), 877
refine_1_2() (in module *sfepy.discrete.fem.refine*), 911
refine_2_3() (in module *sfepy.discrete.fem.refine*), 911
refine_2_4() (in module *sfepy.discrete.fem.refine*), 911
refine_3_4() (in module *sfepy.discrete.fem.refine*), 911
refine_3_8() (in module *sfepy.discrete.fem.refine*), 911
refine_mesh() (in module *sfepy.discrete.fem.utils*), 913
refine_pars() (*sfepy.discrete.probes.Probe* static method), 829
refine_points() (*sfepy.discrete.probes.PointsProbe* method), 829
refine_points() (*sfepy.discrete.probes.Probe* method), 830
refine_points() (*sfepy.discrete.probes.RayProbe* method), 830
refine_reference() (in module *sfepy.discrete.fem.refine*), 911
refine_region() (in module *sfepy.discrete.fem.refine_hanging*), 912
refine_uniformly() (*sfepy.discrete.problem.Problem* method), 839
refmap_memory_factor() (*sfepy.config.Config* method), 777
Region (class in *sfepy.discrete.common.region*), 873
region_leaf() (in module *sfepy.discrete.common.domain*), 860
region_op() (in module *sfepy.discrete.common.domain*), 860
register_poly_space() (in module *sfepy.discrete.common.poly_spaces*), 873
release() (*sfepy.base.multiproc_mpi.RemoteLock* method), 802
remap_dict() (in module *sfepy.base.base*), 785
remote_get() (*sfepy.base.multiproc_mpi.RemoteDictMaster* method), 802
remote_get() (*sfepy.base.multiproc_mpi.RemoteQueueMaster* method), 803
remote_get_in() (*sfepy.base.multiproc_mpi.RemoteDictMaster* method), 802
remote_get_keys() (*sfepy.base.multiproc_mpi.RemoteDictMaster* method), 802
remote_get_len() (*sfepy.base.multiproc_mpi.RemoteDictMaster* method), 802
remote_put() (*sfepy.base.multiproc_mpi.RemoteQueueMaster* method), 803
remote_set() (*sfepy.base.multiproc_mpi.RemoteDictMaster* method), 802
RemoteDict (class in *sfepy.base.multiproc_mpi*), 802
RemoteDictMaster (class in *sfepy.base.multiproc_mpi*), 802
RemoteInt (class in *sfepy.base.multiproc_mpi*), 802
RemoteInt.IntDesc (class in *sfepy.base.multiproc_mpi*), 802
RemoteLock (class in *sfepy.base.multiproc_mpi*), 802
RemoteQueue (class in *sfepy.base.multiproc_mpi*), 802
RemoteQueueMaster (class in *sfepy.base.multiproc_mpi*), 802
remove_bcs() (*sfepy.discrete.problem.Problem* method), 839
remove_extra_dofs() (*sfepy.discrete.fem.fields_base.FEField* method), 885
remove_extra_dofs() (*sfepy.discrete.fem.fields_nodal.H1DiscontinuousField* method), 889
remove_files() (in module *sfepy.base.ioutils*), 797
remove_files_patterns() (in module *sfepy.base.ioutils*), 797
remove_known() (in module *sfepy.base.resolve_deps*), 806
remove_name() (*sfepy.base.base.Container* method),

- 780
- `replace()` (in module *sfepy.discrete.parse_regions*), 827
- `replace_with_region()` (in module *sfepy.discrete.parse_regions*), 827
- `report()` (in module *sfepy.base.testing*), 807
- `report()` (*sfepy.discrete.probes.CircleProbe* method), 828
- `report()` (*sfepy.discrete.probes.LineProbe* method), 828
- `report()` (*sfepy.discrete.probes.PointsProbe* method), 829
- `report()` (*sfepy.discrete.probes.Probe* method), 830
- `report()` (*sfepy.discrete.probes.RayProbe* method), 831
- `reset()` (*sfepy.base.timing.Timer* method), 807
- `reset()` (*sfepy.discrete.materials.Material* method), 825
- `reset()` (*sfepy.discrete.materials.Materials* method), 826
- `reset()` (*sfepy.discrete.problem.Problem* method), 839
- `reset_materials()` (*sfepy.discrete.equations.Equations* method), 816
- `reset_refinement()` (*sfepy.discrete.probes.Probe* method), 830
- `reset_regions()` (*sfepy.discrete.common.domain.Domain* method), 860
- `resolve()` (in module *sfepy.base.resolve_deps*), 806
- `resolve_chains()` (in module *sfepy.discrete.common.dof_info*), 859
- `restore()` (*sfepy.applications.application.Application* method), 778
- `restore_dofs()` (*sfepy.discrete.fem.fields_base.FEField* method), 885
- `restore_step_time()` (*sfepy.solvers.ts.TimeStepper* method), 1038
- `restore_substituted()` (*sfepy.discrete.fem.fields_base.FEField* method), 885
- `resview_plot()` (in module *gen_gallery*), 1204
- `rhs()` (in module *sfepy.discrete.parse_equations*), 827
- `rhs()` (in module *sfepy.tests.test_msm_laplace*), 1197
- `rhs()` (in module *sfepy.tests.test_msm_symbolic*), 1197
- `Rigid20operator` (class in *sfepy.discrete.fem.lcbc_operators*), 895
- `Rigid0operator` (class in *sfepy.discrete.fem.lcbc_operators*), 895
- `RK4StepSolver` (class in *sfepy.solvers.ts_dg_solvers*), 933
- `rm_multi()` (in module *sfepy.homogenization.utils*), 968
- `RMMSolver` (class in *sfepy.solvers.ls*), 1024
- `rotate_elastic_tensor()` (in module *sfepy.mechanics.shell10x*), 987
- `rotation_matrix2d()` (in module *sfepy.linalg.geometry*), 973
- `run()` (*build_helpers.Clean* method), 1201
- `run()` (*build_helpers.DoxygenDocs* method), 1201
- `run()` (*build_helpers.SphinxHTMLDocs* method), 1202
- `run()` (*build_helpers.SphinxPDFDocs* method), 1202
- `run_declarative_example()` (in module *sfepy.base.testing*), 807
- `run_resview_plot()` (in module *gen_gallery*), 1204
- ## S
- `save()` (*sfepy.homogenization.coefs_base.CorrMiniApp* method), 953
- `save()` (*sfepy.homogenization.coefs_phononic.SimpleEVP* method), 958
- `save_animation()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 917
- `save_as_mesh()` (*sfepy.discrete.variables.FieldVariable* method), 851
- `save_basis()` (in module *sfepy.discrete.iga.utils*), 949
- `save_dict()` (*sfepy.applications.pde_solver_app.PDESolverApp* method), 779
- `save_ebc()` (*sfepy.discrete.problem.Problem* method), 839
- `save_log()` (*sfepy.homogenization.coefs_phononic.BandGaps* static method), 956
- `save_mappings()` (*sfepy.discrete.common.fields.Field* method), 866
- `save_only()` (in module *sfepy.applications.pde_solver_app*), 779
- `save_options()` (in module *sfepy.base.ioutils*), 797
- `save_raw_bg_logs()` (in module *sfepy.homogenization.band_gaps_app*), 951
- `save_recovery_region()` (in module *sfepy.homogenization.recovery*), 967
- `save_regions()` (*sfepy.discrete.common.domain.Domain* method), 860
- `save_regions()` (*sfepy.discrete.problem.Problem* method), 839
- `save_regions_as_groups()` (*sfepy.discrete.common.domain.Domain* method), 860
- `save_regions_as_groups()` (*sfepy.discrete.problem.Problem* method), 840
- `save_restart()` (*sfepy.discrete.problem.Problem* method), 840
- `save_results()` (*sfepy.applications.evp_solver_app.EVPSolverApp* method), 779
- `save_sol_snap()` (in module *sfepy.discrete.dg.dg_1D_vizualizer*), 917
- `save_sparse_txt()` (in module *sfepy.linalg.sparse*), 974
- `save_state()` (*sfepy.discrete.problem.Problem* method), 840
- `save_time_history()` (in module *sfepy.postprocess.time_history*), 1015
- `ScalarDotGradIScalarTerm` (class in *sfepy.terms.terms_dot*), 1093

ScalarDotMGradScalarTerm (class in *sfepy.discrete.fem.poly_spaces*), 908
sfepy.terms.terms_dot), 1094
 scale_matrix() (in module *sfepy.solvers.oseen*), 1034
 SchurEVP (class in *sfepy.homogenization.coefs_phononic*), 957
 SchurMumps (class in *sfepy.solvers.ls*), 1025
 ScipyDirect (class in *sfepy.solvers.ls*), 1025
 ScipyEigenvalueSolver (class in *sfepy.solvers.eigen*), 1019
 ScipyFMinSolver (class in *sfepy.solvers.optimize*), 1032
 ScipyIterative (class in *sfepy.solvers.ls*), 1026
 ScipyRoot (class in *sfepy.solvers.nls*), 1030
 ScipySGEigenvalueSolver (class in *sfepy.solvers.eigen*), 1020
 ScipySuperLU (class in *sfepy.solvers.ls*), 1026
 ScipyUmfpack (class in *sfepy.solvers.ls*), 1027
 SDConvectTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1058
 SDDiffusionTerm (class in *sfepy.terms.terms_diffusion*), 1088
 SDDivGradTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1058
 SDDivTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1059
 SDDotTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1059
 SDGradDivStabilizationTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1060
 SDLinearElasticTerm (class in *sfepy.terms.terms_elastic*), 1108
 SDLinearTractionTerm (class in *sfepy.terms.terms_surface*), 1175
 SDPiezoCouplingTerm (class in *sfepy.terms.terms_piezo*), 1162
 SDPSPGCStabilizationTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1061
 SDPSPGPStabilizationTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1061
 SDSufaceIntegrateTerm (class in *sfepy.terms.terms_surface*), 1175
 SDSUPGCStabilizationTerm (class in *sfepy.terms.terms_adj_navier_stokes*), 1062
 SDVolumeDotTerm (class in *sfepy.terms.terms_compat*), 1074
 select_bcs() (*sfepy.discrete.problem.Problem* method), 840
 select_by_names() (in module *sfepy.base.base*), 785
 select_materials() (*sfepy.discrete.problem.Problem* method), 840
 select_variables() (*sfepy.discrete.problem.Problem* method), 840
 SemismoothNewton (class in *sfepy.solvers.semismooth_newton*), 1035
 SEMTensorProductPolySpace (class in *sfepy.discrete.fem.poly_spaces*), 908
 separate() (*sfepy.mesh.geom_tools.surface* method), 1000
 separator (*sfepy.scripts.resview.FieldOptsToListAction* attribute), 1186
 separator (*sfepy.scripts.resview.OptsToListAction* attribute), 1186
 SerendipityTensorProductPolySpace (class in *sfepy.discrete.fem.poly_spaces*), 909
 set_accuracy() (in module *sfepy.discrete.fem.mesh*), 900
 set_accuracy() (in module *sfepy.discrete.fem.periodic*), 907
 set_adof_conns() (*sfepy.discrete.variables.Variables* method), 856
 set_all_data() (*sfepy.discrete.materials.Material* method), 825
 set_approx_points() (*sfepy.mesh.bspline.BSpline* method), 996
 set_approx_points() (*sfepy.mesh.bspline.BSplineSurf* method), 997
 set_arg_types() (*sfepy.terms.terms.Term* method), 1053
 set_arg_types() (*sfepy.terms.terms_biot.BiotTerm* method), 1072
 set_arg_types() (*sfepy.terms.terms_diffusion.DiffusionCoupling* method), 1085
 set_arg_types() (*sfepy.terms.terms_diffusion.DiffusionTerm* method), 1087
 set_arg_types() (*sfepy.terms.terms_diffusion.LaplaceTerm* method), 1088
 set_arg_types() (*sfepy.terms.terms_dot.DotProductTerm* method), 1092
 set_arg_types() (*sfepy.terms.terms_dot.ScalarDotGradIScalarTerm* method), 1094
 set_arg_types() (*sfepy.terms.terms_dot.VectorDotGradScalarTerm* method), 1095
 set_arg_types() (*sfepy.terms.terms_dot.VectorDotScalarTerm* method), 1096
 set_arg_types() (*sfepy.terms.terms_elastic.LinearElasticTerm* method), 1104
 set_arg_types() (*sfepy.terms.terms_elastic.LinearPrestressTerm* method), 1105
 set_arg_types() (*sfepy.terms.terms_elastic.NonsymElasticTerm* method), 1108
 set_arg_types() (*sfepy.terms.terms_navier_stokes.DivGradTerm* method), 1153
 set_arg_types() (*sfepy.terms.terms_navier_stokes.StokesTerm* method), 1159
 set_arg_types() (*sfepy.terms.terms_piezo.PiezoCouplingTerm* method), 1161
 set_arg_types() (*sfepy.terms.terms_surface.LinearTractionTerm* method), 1174
 set_arg_types() (*sfepy.terms.terms_surface.SDLinearTractionTerm*

`method`), 1175
`set_arg_types()` (*sfepy.terms.terms_surface.SurfaceNormalDotTerm* method), 889
`method`), 1176
`set_axes_font_size()` (in module *sfepy.base.plotutils*), 806
`set_backend()` (*sfepy.terms.terms_multilinear.ETermBase* method), 1147
`set_basis_indices()` (*sfepy.discrete.fem.mappings.FEMapping* method), 897
`set_basis_transform()` (*sfepy.discrete.fem.fields_base.FEField* method), 885
`set_bcs()` (*sfepy.discrete.problem.Problem* method), 840
`set_cell_dofs()` (*sfepy.discrete.dg.fields.DGField* method), 925
`set_conf_solvers()` (*sfepy.discrete.problem.Problem* method), 840
`set_constant()` (*sfepy.discrete.variables.Variable* method), 853
`set_control_points()` (*sfepy.mesh.bspline.BSpline* method), 996
`set_control_points()` (*sfepy.mesh.bspline.BSplineSurf* method), 997
`set_control_points()` (*sfepy.mesh.splinebox.SplineBox* method), 1007
`set_coors()` (*sfepy.discrete.fem.fields_base.FEField* method), 885
`set_data()` (*sfepy.discrete.equations.Equations* method), 816
`set_data()` (*sfepy.discrete.materials.Material* method), 825
`set_data()` (*sfepy.discrete.variables.Variable* method), 853
`set_data()` (*sfepy.discrete.variables.Variables* method), 856
`set_default()` (*sfepy.base.base.Struct* method), 782
`set_default_state()` (*sfepy.discrete.problem.Problem* method), 841
`set_defaults()` (in module *sfepy.base.base*), 785
`set_dim()` (in module *sfepy.linalg.sympy_operators*), 974
`set_dof_info()` (*sfepy.solvers.solvers.TimeSteppingSolver* method), 1037
`set_dofs()` (*sfepy.discrete.common.fields.Field* method), 866
`set_dofs()` (*sfepy.discrete.dg.fields.DGField* method), 926
`set_dofs()` (*sfepy.discrete.fem.fields_hierarchic.H1HierarchicVolumeField* method), 887
`set_dofs()` (*sfepy.discrete.fem.fields_nodal.H1NodalMixin* method), 887
`set_equations()` (*sfepy.discrete.problem.Problem* method), 841
`set_equations_instance()` (*sfepy.discrete.problem.Problem* method), 841
`set_extra_args()` (*sfepy.discrete.functions.Function* method), 822
`set_extra_args()` (*sfepy.discrete.materials.Material* method), 825
`set_facet_dofs()` (*sfepy.discrete.dg.fields.DGField* method), 926
`set_field_split()` (*sfepy.solvers.ls.PETScKrylovSolver* method), 1023
`set_field_split()` (*sfepy.solvers.solvers.Solver* method), 1037
`set_fields()` (*sfepy.discrete.problem.Problem* method), 841
`set_float_format()` (*sfepy.discrete.fem.meshio.MeshIO* method), 905
`set_from_data()` (*sfepy.solvers.ts.TimeStepper* method), 1038
`set_from_data()` (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
`set_from_function()` (*sfepy.discrete.variables.FieldVariable* method), 851
`set_from_mesh_vertices()` (*sfepy.discrete.variables.FieldVariable* method), 851
`set_from_other()` (*sfepy.discrete.variables.FieldVariable* method), 851
`set_from_qp()` (*sfepy.discrete.variables.FieldVariable* method), 852
`set_from_ts()` (*sfepy.solvers.ts.TimeStepper* method), 1038
`set_from_ts()` (*sfepy.solvers.ts.VariableTimeStepper* method), 1038
`set_full_state()` (*sfepy.discrete.variables.Variables* method), 856
`set_function()` (*sfepy.discrete.functions.Function* method), 822
`set_function()` (*sfepy.discrete.materials.Material* method), 825
`set_ics()` (*sfepy.discrete.problem.Problem* method), 841
`set_integral()` (*sfepy.terms.terms.Term* method), 1053
`set_integral()` (*sfepy.terms.terms_shells.Shell10XTerm* method), 1171
`set_kind()` (*sfepy.discrete.common.region.Region* method), 876
`set_kind_tdim()` (*sfepy.discrete.common.region.Region* method), 876

method), 876
set_knot_vector() (*sfepy.mesh.bspline.BSpline method*), 996
set_linear() (*sfepy.discrete.problem.Problem method*), 841
set_local_entities() (*sfepy.discrete.common.extmods.cmesh.CMesh method*), 863
set_logging_level() (*in module sfepy.base.multiproc_mpi*), 803
set_materials() (*sfepy.discrete.problem.Problem method*), 841
set_mesh_coors() (*in module sfepy.discrete.fem.fields_base*), 887
set_mesh_coors() (*sfepy.discrete.problem.Problem method*), 841
set_method() (*sfepy.solvers.optimize.ScipyFMinSolver method*), 1033
set_micro_states() (*sfepy.homogenization.engine.HomogenizationEngine method*), 961
set_n_digit_from_min_dt() (*sfepy.solvers.ts.VariableTimeStepper method*), 1038
set_n_point() (*sfepy.discrete.probes.Probe method*), 830
set_nonlin_states() (*in module sfepy.homogenization.utils*), 968
set_options() (*sfepy.discrete.probes.Probe method*), 830
set_orientation_map() (*sfepy.discrete.fem.fe_surface.FESurface method*), 880
set_output() (*sfepy.base.base.Output method*), 781
set_output_dir() (*sfepy.discrete.problem.Problem method*), 841
set_output_prefix() (*sfepy.base.base.Output method*), 781
set_param() (*sfepy.mesh.bspline.BSpline method*), 996
set_param_n() (*sfepy.mesh.bspline.BSpline method*), 996
set_param_n() (*sfepy.mesh.bspline.BSplineSurf method*), 998
set_reduced_state() (*sfepy.discrete.variables.Variables method*), 856
set_regions() (*sfepy.discrete.problem.Problem method*), 841
set_section() (*in module gen_term_table*), 1206
set_solver() (*sfepy.discrete.problem.Problem method*), 841
set_state() (*sfepy.discrete.equations.Equations method*), 816
set_state() (*sfepy.discrete.variables.Variables method*), 856
set_state() (*sfepy.solvers.ts.TimeStepper method*), 1038
set_state() (*sfepy.solvers.ts.VariableTimeStepper method*), 1038
set_state_parts() (*sfepy.discrete.variables.Variables method*), 856
set_step() (*sfepy.solvers.ts.TimeStepper method*), 1038
set_step() (*sfepy.solvers.ts.VariableTimeStepper method*), 1038
set_substep_time() (*sfepy.solvers.ts.TimeStepper method*), 1038
set_time_step() (*sfepy.solvers.ts.VariableTimeStepper method*), 1038
set_variables() (*sfepy.discrete.problem.Problem method*), 842
set_variables_default() (*sfepy.homogenization.coefs_base.CoeffExprPar static method*), 952
set_variables_default() (*sfepy.homogenization.coefs_base.CoeffMN static method*), 952
set_variables_default() (*sfepy.homogenization.coefs_base.CoeffN static method*), 952
set_variables_default() (*sfepy.homogenization.coefs_base.CoeffOne static method*), 953
set_variables_default() (*sfepy.homogenization.coefs_base.CorrN static method*), 953
set_variables_default() (*sfepy.homogenization.coefs_base.CorrNN static method*), 953
set_variables_default() (*sfepy.homogenization.coefs_base.CorrOne static method*), 954
set_vec_part() (*sfepy.discrete.variables.Variables method*), 856
set_verbosity() (*sfepy.terms.terms_multilinear.ETermBase method*), 1147
setdefault() (*sfepy.base.base.IndexedStruct method*), 781
setup() (*in module gen_field_table*), 1203
setup() (*in module gen_solver_table*), 1205
setup() (*in module gen_term_table*), 1206
setup() (*sfepy.base.conf.ProblemConf method*), 791
setup() (*sfepy.discrete.fem.lcbc_operators.LCBCOperator method*), 893
setup() (*sfepy.discrete.fem.lcbc_operators.MRLCBCOperator method*), 893
setup() (*sfepy.solvers.oseen.StabilizationFunction method*), 1034
setup() (*sfepy.terms.terms.Term method*), 1053
setup() (*sfepy.terms.terms.Terms method*), 1054

<code>setUp()</code> (<i>sfepy.tests.test_linear_solvers.DiagPC method</i>), 1195	<code>setup_integration()</code> (<i>sfepy.terms.terms.Term method</i>), 1053
<code>setup_args()</code> (<i>sfepy.terms.terms.Term method</i>), 1053	<code>setup_lcbc_operators()</code> (<i>sfepy.discrete.variables.Variables method</i>), 857
<code>setup_axis()</code> (<i>in sfepy.discrete.dg.dg_1D_vizualizer module</i>), 918	<code>setup_lines()</code> (<i>in sfepy.discrete.dg.dg_1D_vizualizer module</i>), 918
<code>setup_bar_data()</code> (<i>sfepy.discrete.fem.fields_base.FEField method</i>), 885	<code>setup_macro_data()</code> (<i>sfepy.homogenization.homogen_app.Homogenization method</i>), 964
<code>setup_composite_dofs()</code> (<i>in sfepy.parallel.parallel module</i>), 1009	<code>setup_mirror_connectivity()</code> (<i>sfepy.discrete.fem.fe_surface.FESurface method</i>), 880
<code>setup_connectivity()</code> (<i>sfepy.discrete.common.extmods.cmesh.CMesh method</i>), 863	<code>setup_mirror_region()</code> (<i>sfepy.discrete.common.region.Region method</i>), 876
<code>setup_coors()</code> (<i>sfepy.discrete.fem.fields_base.FEField method</i>), 885	<code>setup_options()</code> (<i>sfepy.applications.application.Application method</i>), 778
<code>setup_default_output()</code> (<i>sfepy.discrete.problem.Problem method</i>), 842	<code>setup_options()</code> (<i>sfepy.applications.evp_solver_app.EVPSolverApp method</i>), 779
<code>setup_dof_info()</code> (<i>sfepy.discrete.variables.Variables method</i>), 856	<code>setup_options()</code> (<i>sfepy.applications.pde_solver_app.PDESolverApp method</i>), 779
<code>setup_dtype()</code> (<i>sfepy.discrete.variables.Variables method</i>), 856	<code>setup_options()</code> (<i>sfepy.homogenization.band_gaps_app.AcousticBandGapsApp method</i>), 951
<code>setup_entities()</code> (<i>sfepy.discrete.common.extmods.cmesh.CMesh method</i>), 863	<code>setup_options()</code> (<i>sfepy.homogenization.engine.HomogenizationEngine method</i>), 961
<code>setup_extra_data()</code> (<i>in sfepy.discrete.common.fields module</i>), 867	<code>setup_options()</code> (<i>sfepy.homogenization.homogen_app.HomogenizationApp method</i>), 964
<code>setup_extra_data()</code> (<i>sfepy.discrete.dg.fields.DGField method</i>), 927	<code>setup_ordering()</code> (<i>sfepy.discrete.variables.Variables method</i>), 857
<code>setup_extra_data()</code> (<i>sfepy.discrete.fem.fields_base.FEField method</i>), 885	<code>setup_parallelization()</code> (<i>in sfepy.discrete.fem.geometry_element module</i>), 891
<code>setup_extra_data()</code> (<i>sfepy.discrete.fem.fields_l2.L2Consistent method</i>), 888	<code>setup_output()</code> (<i>sfepy.applications.evp_solver_app.EVPSolverApp method</i>), 779
<code>setup_extra_data()</code> (<i>sfepy.discrete.iga.fields.IGField method</i>), 940	<code>setup_output()</code> (<i>sfepy.discrete.problem.Problem method</i>), 842
<code>setup_formal_args()</code> (<i>sfepy.terms.terms.Term method</i>), 1053	<code>setup_output()</code> (<i>sfepy.homogenization.coefs_base.CorrMiniApp method</i>), 953
<code>setup_from_highest()</code> (<i>sfepy.discrete.common.region.Region method</i>), 876	<code>setup_output_info()</code> (<i>sfepy.applications.pde_solver_app.PDESolverApp method</i>), 779
<code>setup_from_vertices()</code> (<i>sfepy.discrete.common.region.Region method</i>), 876	<code>setup_petsc_precond()</code> (<i>in sfepy.tests.test_linear_solvers module</i>), 1195
<code>setup_geometry_types()</code> (<i>sfepy.terms.terms.Term method</i>), 1053	<code>setup_point_data()</code> (<i>sfepy.discrete.fem.fields_base.FEField method</i>), 885
<code>setup_hooks()</code> (<i>sfepy.discrete.problem.Problem method</i>), 842	<code>setup_surface_data()</code> (<i>sfepy.discrete.fem.fields_base.FEField method</i>), 885
<code>setup_initial_conditions()</code> (<i>sfepy.discrete.equations.Equations method</i>), 816	<code>sfepy.applications.application module</code> , 778
<code>setup_initial_conditions()</code> (<i>sfepy.discrete.variables.FieldVariable method</i>), 852	<code>sfepy.applications.evp_solver_app module</code> , 778
<code>setup_initial_conditions()</code> (<i>sfepy.discrete.variables.Variables method</i>), 857	<code>sfepy.applications.pde_solver_app module</code> , 779
	<code>sfepy.base.base</code>

- module, 780
- sfepy.base.compat
 - module, 786
- sfepy.base.conf
 - module, 790
- sfepy.base.getch
 - module, 792
- sfepy.base.goptions
 - module, 793
- sfepy.base.ioutils
 - module, 793
- sfepy.base.log
 - module, 798
- sfepy.base.log_plotter
 - module, 800
- sfepy.base.mem_usage
 - module, 800
- sfepy.base.multiproc
 - module, 801
- sfepy.base.multiproc_mpi
 - module, 801
- sfepy.base.multiproc_proc
 - module, 804
- sfepy.base.parse_conf
 - module, 805
- sfepy.base.plotutils
 - module, 806
- sfepy.base.reader
 - module, 806
- sfepy.base.resolve_deps
 - module, 806
- sfepy.base.testing
 - module, 807
- sfepy.base.timing
 - module, 807
- sfepy.config
 - module, 777
- sfepy.discrete.common.dof_info
 - module, 857
- sfepy.discrete.common.domain
 - module, 860
- sfepy.discrete.common.extmods._fmfield
 - module, 861
- sfepy.discrete.common.extmods._geommech
 - module, 861
- sfepy.discrete.common.extmods.assemble
 - module, 861
- sfepy.discrete.common.extmods.cmapping
 - module, 861
- sfepy.discrete.common.extmods.cmesh
 - module, 861
- sfepy.discrete.common.extmods.crefcoors
 - module, 864
- sfepy.discrete.common.fields
 - module, 864
- sfepy.discrete.common.global_interp
 - module, 867
- sfepy.discrete.common.mappings
 - module, 870
- sfepy.discrete.common.poly_spaces
 - module, 872
- sfepy.discrete.common.region
 - module, 873
- sfepy.discrete.conditions
 - module, 808
- sfepy.discrete.dg.dg_1D_vizualizer
 - module, 914
- sfepy.discrete.dg.fields
 - module, 919
- sfepy.discrete.dg.limiters
 - module, 931
- sfepy.discrete.dg.poly_spaces
 - module, 928
- sfepy.discrete.equations
 - module, 811
- sfepy.discrete.evaluate
 - module, 818
- sfepy.discrete.evaluate_variable
 - module, 822
- sfepy.discrete.fem._serendipity
 - module, 912
- sfepy.discrete.fem.domain
 - module, 877
- sfepy.discrete.fem.extmods.bases
 - module, 878
- sfepy.discrete.fem.extmods.lobatto_bases
 - module, 878
- sfepy.discrete.fem.facets
 - module, 879
- sfepy.discrete.fem.fe_surface
 - module, 880
- sfepy.discrete.fem.fields_base
 - module, 881
- sfepy.discrete.fem.fields_hierarchic
 - module, 887
- sfepy.discrete.fem.fields_l2
 - module, 887
- sfepy.discrete.fem.fields_nodal
 - module, 889
- sfepy.discrete.fem.fields_positive
 - module, 890
- sfepy.discrete.fem.geometry_element
 - module, 891
- sfepy.discrete.fem.history
 - module, 892
- sfepy.discrete.fem.lcbc_operators
 - module, 892
- sfepy.discrete.fem.linearizer

module, 895	module, 847
sfepy.discrete.fem.mappings	sfepy.discrete.structural.fields
module, 896	module, 949
sfepy.discrete.fem.mesh	sfepy.discrete.structural.mappings
module, 898	module, 950
sfepy.discrete.fem.meshio	sfepy.discrete.variables
module, 900	module, 847
sfepy.discrete.fem.periodic	sfepy.homogenization.band_gaps_app
module, 907	module, 950
sfepy.discrete.fem.poly_spaces	sfepy.homogenization.coefficients
module, 907	module, 951
sfepy.discrete.fem.refine	sfepy.homogenization.coefs_base
module, 911	module, 952
sfepy.discrete.fem.refine_hanging	sfepy.homogenization.coefs_phononic
module, 912	module, 954
sfepy.discrete.fem.utils	sfepy.homogenization.convolution
module, 912	module, 960
sfepy.discrete.functions	sfepy.homogenization.engine
module, 822	module, 960
sfepy.discrete.iga.domain	sfepy.homogenization.homogen_app
module, 934	module, 963
sfepy.discrete.iga.domain_generators	sfepy.homogenization.micmac
module, 935	module, 964
sfepy.discrete.iga.extmods.igac	sfepy.homogenization.recovery
module, 936	module, 964
sfepy.discrete.iga.fields	sfepy.homogenization.utils
module, 938	module, 967
sfepy.discrete.iga.iga	sfepy.linalg.check_derivatives
module, 940	module, 969
sfepy.discrete.iga.io	sfepy.linalg.eigen
module, 946	module, 969
sfepy.discrete.iga.mappings	sfepy.linalg.geometry
module, 947	module, 970
sfepy.discrete.iga.plot_nurbs	sfepy.linalg.sparse
module, 948	module, 973
sfepy.discrete.iga.utils	sfepy.linalg.sympy_operators
module, 948	module, 974
sfepy.discrete.integrals	sfepy.linalg.utils
module, 823	module, 975
sfepy.discrete.materials	sfepy.mechanics.contact_bodies
module, 824	module, 979
sfepy.discrete.parse_equations	sfepy.mechanics.elastic_constants
module, 827	module, 979
sfepy.discrete.parse_regions	sfepy.mechanics.extmods.ccontres
module, 827	module, 993
sfepy.discrete.probes	sfepy.mechanics.matcoefs
module, 828	module, 979
sfepy.discrete.problem	sfepy.mechanics.membranes
module, 832	module, 983
sfepy.discrete.projections	sfepy.mechanics.shell10x
module, 844	module, 986
sfepy.discrete.quadratures	sfepy.mechanics.tensors
module, 845	module, 988
sfepy.discrete.simplex_cubature	sfepy.mechanics.units

- module, 991
- sfepy.mesh.bspline
 - module, 993
- sfepy.mesh.geom_tools
 - module, 999
- sfepy.mesh.mesh_generators
 - module, 1001
- sfepy.mesh.mesh_tools
 - module, 1004
- sfepy.mesh.splinebox
 - module, 1006
- sfepy.parallel.evaluate
 - module, 1007
- sfepy.parallel.parallel
 - module, 1008
- sfepy.parallel.plot_parallel_dofs
 - module, 1010
- sfepy.postprocess.plot_cmesh
 - module, 1010
- sfepy.postprocess.plot_dofs
 - module, 1011
- sfepy.postprocess.plot_facets
 - module, 1011
- sfepy.postprocess.plot_quadrature
 - module, 1012
- sfepy.postprocess.probes_vtk
 - module, 1012
- sfepy.postprocess.time_history
 - module, 1014
- sfepy.postprocess.utils_vtk
 - module, 1015
- sfepy.scripts.blockgen
 - module, 1182
- sfepy.scripts.convert_mesh
 - module, 1182
- sfepy.scripts.cylindergen
 - module, 1183
- sfepy.scripts.gen_iga_patch
 - module, 1183
- sfepy.scripts.gen_mesh
 - module, 1183
- sfepy.scripts.gen_mesh_prev
 - module, 1183
- sfepy.scripts.plot_condition_numbers
 - module, 1184
- sfepy.scripts.plot_logs
 - module, 1184
- sfepy.scripts.plot_mesh
 - module, 1184
- sfepy.scripts.plot_quadratures
 - module, 1184
- sfepy.scripts.plot_times
 - module, 1184
- sfepy.scripts.probe
 - module, 1184
- sfepy.scripts.resview
 - module, 1186
- sfepy.scripts.run_tests
 - module, 1187
- sfepy.scripts.simple
 - module, 1187
- sfepy.solvers.auto_fallback
 - module, 1016
- sfepy.solvers.eigen
 - module, 1017
- sfepy.solvers.ls
 - module, 1020
- sfepy.solvers.nls
 - module, 1027
- sfepy.solvers.optimize
 - module, 1031
- sfepy.solvers.oseen
 - module, 1033
- sfepy.solvers.qeigen
 - module, 1034
- sfepy.solvers.semismooth_newton
 - module, 1035
- sfepy.solvers.solvers
 - module, 1036
- sfepy.solvers.ts
 - module, 1038
- sfepy.solvers.ts_controllers
 - module, 1039
- sfepy.solvers.ts_dg_solvers
 - module, 932
- sfepy.solvers.ts_solvers
 - module, 1042
- sfepy.terms.extmods.terms
 - module, 1179
- sfepy.terms.terms
 - module, 1050
- sfepy.terms.terms_adj_navier_stokes
 - module, 1055
- sfepy.terms.terms_basic
 - module, 1065
- sfepy.terms.terms_biot
 - module, 1069
- sfepy.terms.terms_compat
 - module, 1072
- sfepy.terms.terms_constraints
 - module, 1075
- sfepy.terms.terms_contact
 - module, 1077
- sfepy.terms.terms_dg
 - module, 1079
- sfepy.terms.terms_diffusion
 - module, 1084
- sfepy.terms.terms_dot

module, 1090	module, 1190
sfepy.terms.terms_elastic	sfepy.tests.test_dg_terms_calls
module, 1096	module, 1190
sfepy.terms.terms_electric	sfepy.tests.test_domain
module, 1109	module, 1191
sfepy.terms.terms_fibres	sfepy.tests.test_ed_solvers
module, 1110	module, 1192
sfepy.terms.terms_flexo	sfepy.tests.test_eigenvalue_solvers
module, 1111	module, 1192
sfepy.terms.terms_hyperelastic_base	sfepy.tests.test_elasticity_small_strain
module, 1114	module, 1192
sfepy.terms.terms_hyperelastic_tl	sfepy.tests.test_fem
module, 1115	module, 1192
sfepy.terms.terms_hyperelastic_ul	sfepy.tests.test_functions
module, 1124	module, 1193
sfepy.terms.terms_jax	sfepy.tests.test_high_level
module, 1128	module, 1193
sfepy.terms.terms_mass	sfepy.tests.test_homogenization_engine
module, 1135	module, 1193
sfepy.terms.terms_membrane	sfepy.tests.test_homogenization_perfusion
module, 1136	module, 1193
sfepy.terms.terms_multilinear	sfepy.tests.test_hyperelastic_tlul
module, 1137	module, 1194
sfepy.terms.terms_navier_stokes	sfepy.tests.test_io
module, 1152	module, 1194
sfepy.terms.terms_piezo	sfepy.tests.test_laplace_unit_disk
module, 1160	module, 1194
sfepy.terms.terms_point	sfepy.tests.test_laplace_unit_square
module, 1163	module, 1194
sfepy.terms.terms_sensitivity	sfepy.tests.test_lcbcs
module, 1164	module, 1194
sfepy.terms.terms_shells	sfepy.tests.test_linalg
module, 1170	module, 1194
sfepy.terms.terms_surface	sfepy.tests.test_linear_solvers
module, 1172	module, 1195
sfepy.terms.terms_th	sfepy.tests.test_linearization
module, 1177	module, 1195
sfepy.terms.terms_volume	sfepy.tests.test_log
module, 1177	module, 1195
sfepy.terms.utils	sfepy.tests.test_matcoefs
module, 1178	module, 1195
sfepy.tests.conftest	sfepy.tests.test_mesh_expand
module, 1188	module, 1195
sfepy.tests.test_assembling	sfepy.tests.test_mesh_generators
module, 1188	module, 1196
sfepy.tests.test_base	sfepy.tests.test_mesh_interp
module, 1189	module, 1196
sfepy.tests.test_cmesh	sfepy.tests.test_mesh_smoothing
module, 1189	module, 1196
sfepy.tests.test_conditions	sfepy.tests.test_meshio
module, 1189	module, 1196
sfepy.tests.test_declarative_examples	sfepy.tests.test_msm_laplace
module, 1189	module, 1197
sfepy.tests.test_dg_field	sfepy.tests.test_msm_symbolic

module, 1197
sfepy.tests.test_normals
 module, 1197
sfepy.tests.test_parsing
 module, 1197
sfepy.tests.test_poly_spaces
 module, 1197
sfepy.tests.test_projections
 module, 1198
sfepy.tests.test_quadratures
 module, 1198
sfepy.tests.test_ref_coors
 module, 1198
sfepy.tests.test_refine_hanging
 module, 1199
sfepy.tests.test_regions
 module, 1199
sfepy.tests.test_semismooth_newton
 module, 1199
sfepy.tests.test_sparse
 module, 1199
sfepy.tests.test_splinebox
 module, 1199
sfepy.tests.test_tensors
 module, 1200
sfepy.tests.test_term_call_modes
 module, 1200
sfepy.tests.test_term_consistency
 module, 1200
sfepy.tests.test_term_sensitivity
 module, 1200
sfepy.tests.test_units
 module, 1201
sfepy.tests.test_volume
 module, 1201
sfepy.version
 module, 778
ShapeDim (class in sfepy.homogenization.coefs_base), 954
ShapeDimDim (class in sfepy.homogenization.coefs_base), 954
shell() (in module sfepy.base.base), 785
Shell10XField (class in sfepy.discrete.structural.fields), 949
Shell10XMapping (class in sfepy.discrete.structural.mappings), 950
Shell10XTerm (class in sfepy.terms.terms_shells), 1170
ShiftedPeriodicOperator (class in sfepy.discrete.fem.lcbc_operators), 895
show_authors
 module, 1206
show_terms_use
 module, 1207
simple_example() (in module sfepy.mesh.bspline), 998
SimpleEVP (class in sfepy.homogenization.coefs_phononic), 957
SimpleTimeSteppingSolver (class in sfepy.solvers.ts_solvers), 1047
skip_read_line() (in module sfepy.base.ioutils), 797
slave_get_task() (in module sfepy.base.multiproc_mpi), 803
slave_task_done() (in module sfepy.base.multiproc_mpi), 803
SLEPcEigenvalueSolver (class in sfepy.solvers.eigen), 1019
smooth_f() (sfepy.terms.terms_surface.ContactPlaneTerm static method), 1173
smooth_mesh() (in module sfepy.mesh.mesh_tools), 1005
SoftLink (class in sfepy.base.ioutils), 795
sol_frame() (in module sfepy.discrete.dg.dg_1D_vizualizer), 918
solutions() (in module sfepy.tests.test_elasticity_small_strain), 1192
solvable() (in module sfepy.base.resolve_deps), 807
solve() (in module sfepy.solvers.ls), 1027
solve() (sfepy.discrete.problem.Problem method), 842
solve_eigen_problem() (sfepy.applications.evp_solver_app.EVPSolverApp method), 779
solve_pde() (in module sfepy.applications.pde_solver_app), 779
solve_step() (sfepy.solvers.ts_dg_solvers.DGMultiStageTSS method), 932
solve_step() (sfepy.solvers.ts_dg_solvers.EulerStepSolver method), 933
solve_step() (sfepy.solvers.ts_dg_solvers.RK4StepSolver method), 933
solve_step() (sfepy.solvers.ts_dg_solvers.TVDRK3StepSolver method), 933
solve_step() (sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver method), 1043
solve_step() (sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver method), 1048
solve_step0() (sfepy.solvers.ts_dg_solvers.DGMultiStageTSS method), 933
solve_step0() (sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver method), 1048
Solver (class in sfepy.solvers.solvers), 1036
solver_call() (sfepy.solvers.eigen.MatlabEigenvalueSolver method), 1018
solver_call() (sfepy.solvers.eigen.OctaveEigenvalueSolver method), 1018
SolverMeta (class in sfepy.solvers.solvers), 1037
sort() (sfepy.discrete.conditions.Conditions method), 808
sort_by_dependency() (in module sfepy.discrete.common.region), 877

- `sparse_submat()` (*sfepy.solvers.ls.MultiProblem method*), 1021
- `spause()` (*in module sfepy.base.base*), 785
- `SphinxHTMLDocs` (*class in build_helpers*), 1202
- `SphinxPDFDocs` (*class in build_helpers*), 1202
- `SplineBox` (*class in sfepy.mesh.splinebox*), 1006
- `SplineRegion2D` (*class in sfepy.mesh.splinebox*), 1007
- `split_chunks()` (*in module sfepy.homogenization.coefs_phononic*), 959
- `split_complex_args()` (*in module sfepy.terms.terms*), 1054
- `split_conns_mat_ids()` (*in module sfepy.discrete.fem.meshio*), 906
- `splitlines()` (*sfepy.mesh.geom_tools.geometry method*), 1000
- `spy()` (*in module sfepy.base.plotutils*), 806
- `spy_and_show()` (*in module sfepy.base.plotutils*), 806
- `StabilizationFunction` (*class in sfepy.solvers.oseen*), 1034
- `stage_updates` (*sfepy.solvers.ts_dg_solvers.RK4StepSolver attribute*), 933
- `standalone_setup()` (*sfepy.terms.terms.Term method*), 1053
- `standard_call()` (*in module sfepy.solvers.eigen*), 1020
- `standard_call()` (*in module sfepy.solvers.ls*), 1027
- `standard_call()` (*in module sfepy.solvers.qeigen*), 1035
- `standard_nls_call()` (*in module sfepy.solvers.nls*), 1031
- `standard_ts_call()` (*in module sfepy.solvers.ts_solvers*), 1050
- `start()` (*sfepy.base.timing.Timer method*), 807
- `start()` (*sfepy.base.timing.Timers method*), 808
- `StationarySolver` (*class in sfepy.solvers.ts_solvers*), 1048
- `step()` (*sfepy.solvers.ts_solvers.BatheTS method*), 1044
- `step()` (*sfepy.solvers.ts_solvers.CentralDifferenceTS method*), 1044
- `step()` (*sfepy.solvers.ts_solvers.GeneralizedAlphaTS method*), 1046
- `step()` (*sfepy.solvers.ts_solvers.NewmarkTS method*), 1047
- `step()` (*sfepy.solvers.ts_solvers.VelocityVerletTS method*), 1049
- `stiffness_from_lame()` (*in module sfepy.mechanics.matcoefs*), 981
- `stiffness_from_lame_mixed()` (*in module sfepy.mechanics.matcoefs*), 981
- `stiffness_from_youngpoisson()` (*in module sfepy.mechanics.matcoefs*), 981
- `stiffness_from_youngpoisson_mixed()` (*in module sfepy.mechanics.matcoefs*), 981
- `stiffness_from_yps_ortho3()` (*in module sfepy.mechanics.matcoefs*), 982
- `StokesTerm` (*class in sfepy.terms.terms_navier_stokes*), 1158
- `StokesTractionTerm` (*class in sfepy.terms.terms_multilinear*), 1148
- `StokesWaveDivTerm` (*class in sfepy.terms.terms_navier_stokes*), 1159
- `StokesWaveTerm` (*class in sfepy.terms.terms_navier_stokes*), 1160
- `stop()` (*sfepy.base.timing.Timer method*), 807
- `stop()` (*sfepy.base.timing.Timers method*), 808
- `StoreNumberAction` (*class in sfepy.scripts.resview*), 1187
- `str_all()` (*sfepy.base.base.Struct method*), 782
- `str_class()` (*sfepy.base.base.Struct method*), 782
- `stress_function()` (*sfepy.terms.terms_fibres.FibresActiveTLTerm static method*), 1110
- `stress_function()` (*sfepy.terms.terms_fibres.FibresExponentialTLTerm static method*), 1111
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm static method*), 1116
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm static method*), 1116
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm static method*), 1117
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm method*), 1118
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm static method*), 1119
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm static method*), 1120
- `stress_function()` (*sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm method*), 1121
- `stress_function()` (*sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm static method*), 1124
- `stress_function()` (*sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm static method*), 1125
- `stress_function()` (*sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm static method*), 1127
- `stress_function()` (*sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm static method*), 1127
- `StressTransform` (*class in sfepy.mechanics.tensors*), 988
- `string` (*sfepy.discrete.fem.meshio.HDF5MeshIO attribute*), 903
- `Struct` (*class in sfepy.base.base*), 782
- `structify()` (*in module sfepy.base.base*), 785
- `substitute_dofs()` (*sfepy.discrete.fem.fields_base.FEField method*), 885
- `SufaceNormalDotTerm` (*class in sfepy.terms.terms_surface*), 1176
- `suggest_name()` (*sfepy.discrete.common.poly_spaces.PolySpace static method*), 873
- `SumNodalValuesTerm` (*class in sfepy.terms.terms_basic*), 1066

SUPGCAdjStabilizationTerm (class in `sfepy.terms.terms_adj_navier_stokes`), 1063
 SUPGCStabilizationTerm (class in `sfepy.terms.terms_navier_stokes`), 1157
 SUPGPAdj1StabilizationTerm (class in `sfepy.terms.terms_adj_navier_stokes`), 1063
 SUPGPAdj2StabilizationTerm (class in `sfepy.terms.terms_adj_navier_stokes`), 1064
 SUPGPStabilizationTerm (class in `sfepy.terms.terms_navier_stokes`), 1157
 supported_orders (`sfepy.discrete.fem.poly_spaces.SerendipityTensorProductPolySpace` attribute), 911
 surface (class in `sfepy.mesh.geom_tools`), 1000
 surface_components() (in module `sfepy.mesh.mesh_tools`), 1005
 surface_graph() (in module `sfepy.mesh.mesh_tools`), 1005
 SurfaceDivTerm (class in `sfepy.terms.terms_compat`), 1074
 SurfaceFluxOperatorTerm (class in `sfepy.terms.terms_diffusion`), 1089
 SurfaceFluxOperatorTerm (class in `sfepy.terms.terms_multilinear`), 1149
 SurfaceFluxTerm (class in `sfepy.terms.terms_diffusion`), 1090
 SurfaceFluxTLTerm (class in `sfepy.terms.terms_hyperelastic_tl`), 1121
 SurfaceGradTerm (class in `sfepy.terms.terms_compat`), 1074
 SurfaceJumpTerm (class in `sfepy.terms.terms_surface`), 1176
 SurfaceMomentTerm (class in `sfepy.terms.terms_basic`), 1067
 SurfacePiezoFluxOperatorTerm (class in `sfepy.terms.terms_multilinear`), 1150
 SurfacePiezoFluxTerm (class in `sfepy.terms.terms_multilinear`), 1150
 SurfaceTerm (class in `sfepy.terms.terms_compat`), 1075
 SurfaceTractionTLTerm (class in `sfepy.terms.terms_hyperelastic_tl`), 1122
 sym2dim() (in module `sfepy.mechanics.tensors`), 990
 sym2nonsym() (in module `sfepy.terms.extmods.terms`), 1182
 sym2nonsym() (in module `sfepy.terms.terms_multilinear`), 1151
 sym_tri_eigen() (in module `sfepy.linalg.eigen`), 970
 symarray() (in module `sfepy.tests.test_quadratures`), 1198
 symbolic (`sfepy.terms.terms_dg.AdvectionDGFluxTerm` attribute), 1080
 symbolic (`sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm` attribute), 1083
 symbolic (`sfepy.terms.terms_diffusion.DiffusionTerm` attribute), 1087
 symbolic (`sfepy.terms.terms_diffusion.LaplaceTerm` attribute), 1088
 sync() (`sfepy.base.multiproc_mpi.MPILogFile` method), 802
 sync_module_docs module, 1207
 system() (`sfepy.config.Config` method), 778
T
 tags (in module `sfepy.base.multiproc_mpi`), 803
 tan_mod_function() (`sfepy.terms.terms_fibres.FibresActiveTLTerm` static method), 1110
 tan_mod_function() (`sfepy.terms.terms_fibres.FibresExponentialTLTerm` static method), 1111
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm` static method), 1116
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm` static method), 1116
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm` method), 1118
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm` static method), 1119
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm` static method), 1120
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm` method), 1121
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm` static method), 1124
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm` static method), 1127
 tan_mod_function() (`sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm` static method), 1127
 tan_mod_u_function() (`sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm` static method), 1117
 tan_mod_u_function() (`sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm` static method), 1125
 tdim (`sfepy.discrete.common.extmods.cmesh.CMesh` attribute), 863
 tensor_plane_stress() (`sfepy.mechanics.matcoefs.TransformToPlane` method), 980
 tensor_product() (in module `sfepy.discrete.iga.iga`), 946
 Term (class in `sfepy.terms.terms`), 1050
 term_ns_asm_convect() (in module `sfepy.terms.extmods.terms`), 1182
 term_ns_asm_div_grad() (in module `sfepy.terms.extmods.terms`), 1182
 terminate() (`sfepy.base.log.Log` method), 799
 terminate() (`sfepy.base.log_plotter.LogPlotter` method), 800
 TermParse (class in `sfepy.discrete.parse_equations`), 827

Terms (class in *sfepy.terms.terms*), 1054
 test_active_only() (in module *sfepy.tests.test_ed_solvers*), 1192
 test_assemble1d() (in module *sfepy.tests.test_linalg*), 1194
 test_assemble_matrix() (in module *sfepy.tests.test_assembling*), 1188
 test_assemble_matrix_complex() (in module *sfepy.tests.test_assembling*), 1188
 test_assemble_vector() (in module *sfepy.tests.test_assembling*), 1188
 test_assemble_vector_complex() (in module *sfepy.tests.test_assembling*), 1188
 test_basis_functions_delta() (in module *sfepy.tests.test_fem*), 1192
 test_basis_functions_values() (in module *sfepy.tests.test_fem*), 1192
 test_boundary_fluxes() (in module *sfepy.tests.test_laplace_unit_disk*), 1194
 test_boundary_fluxes() (in module *sfepy.tests.test_laplace_unit_square*), 1194
 test_chunk_micro() (in module *sfepy.tests.test_homogenization_engine*), 1193
 test_cmesh_counts() (in module *sfepy.tests.test_cmesh*), 1189
 test_compare_same_mesheres() (in module *sfepy.tests.test_meshio*), 1196
 test_compose_sparse() (in module *sfepy.tests.test_sparse*), 1199
 test_consistency_d_dw() (in module *sfepy.tests.test_term_consistency*), 1200
 test_consistent_sets() (in module *sfepy.tests.test_units*), 1201
 test_container_add() (in module *sfepy.tests.test_base*), 1189
 test_continuity() (in module *sfepy.tests.test_poly_spaces*), 1198
 test_continuity() (in module *sfepy.tests.test_refine_hanging*), 1199
 test_converged() (in module *sfepy.tests.test_elasticity_small_strain*), 1192
 test_conversion_functions() (in module *sfepy.tests.test_matcoefs*), 1195
 test_create_output1D() (*sfepy.tests.test_dg_field.TestDGField* method), 1190
 test_create_output2D() (*sfepy.tests.test_dg_field.TestDGField* method), 1190
 test_dependencies() (in module *sfepy.tests.test_homogenization_engine*), 1193
 test_ebc_functions() (in module *sfepy.tests.test_functions*), 1193
 test_ebcs() (in module *sfepy.tests.test_conditions*), 1189
 test_ed_solvers() (in module *sfepy.tests.test_ed_solvers*), 1192
 test_eigenvalue_solvers() (in module *sfepy.tests.test_eigenvalue_solvers*), 1192
 test_elastic_constants() (in module *sfepy.tests.test_matcoefs*), 1195
 test_elasticity_rigid() (in module *sfepy.tests.test_lcbcs*), 1194
 test_entity_volumes() (in module *sfepy.tests.test_cmesh*), 1189
 test_epbcs() (in module *sfepy.tests.test_conditions*), 1189
 test_ev_div() (in module *sfepy.tests.test_term_consistency*), 1200
 test_ev_grad() (in module *sfepy.tests.test_term_consistency*), 1200
 test_eval_matrix() (in module *sfepy.tests.test_term_consistency*), 1200
 test_evaluate_at() (in module *sfepy.tests.test_mesh_interp*), 1196
 test_examples() (in module *sfepy.tests.test_declarative_examples*), 1189
 test_examples_dg() (in module *sfepy.tests.test_declarative_examples*), 1189
 test_facets() (in module *sfepy.tests.test_domain*), 1191
 test_field_gradient() (in module *sfepy.tests.test_mesh_interp*), 1196
 test_function_explicit_1D() (*sfepy.tests.test_dg_terms_calls.TestAdvectDGFluxTerm* method), 1191
 test_function_explicit_1D() (*sfepy.tests.test_dg_terms_calls.TestDiffusionInteriorPenaltyTerm* method), 1191
 test_function_explicit_1D() (*sfepy.tests.test_dg_terms_calls.TestNonlinearHyperDGFluxTerm* method), 1191
 test_function_explicit_1D() (*sfepy.tests.test_dg_terms_calls.TestNonlinScalarDotGradTerm* method), 1191
 test_function_explicit_left_1D() (*sfepy.tests.test_dg_terms_calls.TestDiffusionDGFluxTerm* method), 1191
 test_function_explicit_right_1D() (*sfepy.tests.test_dg_terms_calls.TestDiffusionDGFluxTerm* method), 1191
 test_function_implicit_1D() (*sfepy.tests.test_dg_terms_calls.TestAdvectDGFluxTerm* method), 1191
 test_function_implicit_1D() (*sfepy.tests.test_dg_terms_calls.TestDiffusionInteriorPenaltyTerm* method), 1191

`method`), 1191
`test_function_implicit_left_1D()`
 (*sfepy.tests.test_dg_terms_calls.TestDiffusionDGFluxTerm*
 method), 1191
`test_function_implicit_right_1D()`
 (*sfepy.tests.test_dg_terms_calls.TestDiffusionDGFluxTerm*
 method), 1191
`test_gen_block_mesh()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_gen_cylinder_mesh()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_gen_extended_block_mesh()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_gen_mesh_from_geom()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_gen_mesh_from_voxels()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_gen_tiled_mesh()` (in module *sfepy.tests.test_mesh_generators*), 1196
`test_geometry()` (in module *sfepy.tests.test_linalg*), 1194
`test_get_bc_facet_values_1D()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_bc_facet_values_2D()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_bc_facet_values_2D_const()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_blocks_stats()` (in module *sfepy.tests.test_linalg*), 1194
`test_get_facet_idx1D()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_facet_idx2D()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_facet_neighbor_idx_1d()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_get_facet_neighbor_idx_2d()`
 (*sfepy.tests.test_dg_field.TestDGField* *method*), 1190
`test_gradients()` (in module *sfepy.tests.test_poly_spaces*), 1198
`test_hdf5_meshio()` (in module *sfepy.tests.test_meshio*), 1196
`test_hessians()` (in module *sfepy.tests.test_poly_spaces*), 1198
`test_ics()` (in module *sfepy.tests.test_conditions*), 1189
`test_interpolation()` (in module *sfepy.tests.test_mesh_interp*), 1196
`test_interpolation_two_meshes()` (in module *sfepy.tests.test_mesh_interp*), 1196
`test_invariance()` (in module *sfepy.tests.test_mesh_interp*), 1196
`test_invariance_qp()` (in module *sfepy.tests.test_mesh_interp*), 1196
`test_laplace_shifted_periodic()` (in module *sfepy.tests.test_lcbcs*), 1194
`test_linear_terms()` (in module *sfepy.tests.test_elasticity_small_strain*), 1192
`test_linearization()` (in module *sfepy.tests.test_linearization*), 1195
`test_log_rw()` (in module *sfepy.tests.test_log*), 1195
`test_ls_reuse()` (in module *sfepy.tests.test_linear_solvers*), 1195
`test_mass_matrix()` (in module *sfepy.tests.test_projections*), 1198
`test_material_functions()` (in module *sfepy.tests.test_functions*), 1193
`test_mesh_expand()` (in module *sfepy.tests.test_mesh_expand*), 1195
`test_mesh_smoothing()` (in module *sfepy.tests.test_mesh_smoothing*), 1196
`test_msm_laplace()` (in module *sfepy.tests.test_msm_laplace*), 1197
`test_msm_symbolic_diffusion()` (in module *sfepy.tests.test_msm_symbolic*), 1197
`test_msm_symbolic_laplace()` (in module *sfepy.tests.test_msm_symbolic*), 1197
`test_normals()` (in module *sfepy.tests.test_normals*), 1197
`test_operators()` (in module *sfepy.tests.test_regions*), 1199
`test_parse_conf()` (in module *sfepy.tests.test_base*), 1189
`test_parse_equations()` (in module *sfepy.tests.test_parsing*), 1197
`test_parse_regions()` (in module *sfepy.tests.test_parsing*), 1197
`test_partition_of_unity()` (in module *sfepy.tests.test_poly_spaces*), 1198
`test_preserve_coarse_entities()` (in module *sfepy.tests.test_refine_hanging*), 1199
`test_project_tensors()` (in module *sfepy.tests.test_projections*), 1198
`test_projection_iga_fem()` (in module *sfepy.tests.test_projections*), 1198
`test_projection_tri_quad()` (in module *sfepy.tests.test_projections*), 1198
`test_quadratures()` (in module *sfepy.tests.test_quadratures*), 1198
`test_read_dimension()` (in module *sfepy.tests.test_meshio*), 1196
`test_read_meshes()` (in module *sfepy.tests.test_meshio*), 1196

<code>test_recursive_dict_hdf5()</code>	(in module <i>sfepy.tests.test_io</i>), 1194	<code>test_spreigion2d()</code>	(in module <i>sfepy.tests.test_splinebox</i>), 1199
<code>test_ref_coors_fem()</code>	(in module <i>sfepy.tests.test_ref_coors</i>), 1198	<code>test_stiffness_tensors()</code>	(in module <i>sfepy.tests.test_matcoefs</i>), 1195
<code>test_ref_coors_iga()</code>	(in module <i>sfepy.tests.test_ref_coors</i>), 1198	<code>test_stokes_slip_bc()</code>	(in module <i>sfepy.tests.test_lcbcs</i>), 1194
<code>test_refine_2_3()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_stress_transform()</code>	(in module <i>sfepy.tests.test_tensors</i>), 1200
<code>test_refine_2_4()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_struct_add()</code>	(in module <i>sfepy.tests.test_base</i>), 1189
<code>test_refine_3_4()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_struct_i_add()</code>	(in module <i>sfepy.tests.test_base</i>), 1189
<code>test_refine_3_8()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_surface_evaluate()</code>	(in module <i>sfepy.tests.test_term_consistency</i>), 1200
<code>test_refine_hexa()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_tensors()</code>	(in module <i>sfepy.tests.test_linalg</i>), 1194
<code>test_refine_tetra()</code>	(in module <i>sfepy.tests.test_domain</i>), 1191	<code>test_tensors()</code>	(in module <i>sfepy.tests.test_tensors</i>), 1200
<code>test_region_functions()</code>	(in module <i>sfepy.tests.test_functions</i>), 1193	<code>test_term_arithmetics()</code>	(in module <i>sfepy.tests.test_high_level</i>), 1193
<code>test_resolve_deps()</code>	(in module <i>sfepy.tests.test_base</i>), 1189	<code>test_term_call_modes()</code>	(in module <i>sfepy.tests.test_term_call_modes</i>), 1200
<code>test_rmm_solver()</code>	(in module <i>sfepy.tests.test_ed_solvers</i>), 1192	<code>test_term_evaluation()</code>	(in module <i>sfepy.tests.test_high_level</i>), 1193
<code>test_save_ebc()</code>	(in module <i>sfepy.tests.test_conditions</i>), 1189	<code>test_transform_data()</code>	(in module <i>sfepy.tests.test_tensors</i>), 1200
<code>test_selectors()</code>	(in module <i>sfepy.tests.test_regions</i>), 1199	<code>test_transform_data4()</code>	(in module <i>sfepy.tests.test_tensors</i>), 1200
<code>test_semismooth_newton()</code>	(in module <i>sfepy.tests.test_semismooth_newton</i>), 1199	<code>test_unique_rows()</code>	(in module <i>sfepy.tests.test_linalg</i>), 1194
<code>test_sensitivity()</code>	(in module <i>sfepy.tests.test_term_sensitivity</i>), 1200	<code>test_units()</code>	(in module <i>sfepy.tests.test_units</i>), 1201
<code>test_set_dofs_1D()</code>	(<i>sfepy.tests.test_dg_field.TestDGField</i> method), 1190	<code>test_variables()</code>	(in module <i>sfepy.tests.test_high_level</i>), 1193
<code>test_set_dofs_2D()</code>	(<i>sfepy.tests.test_dg_field.TestDGField</i> method), 1190	<code>test_vector_matrix()</code>	(in module <i>sfepy.tests.test_term_consistency</i>), 1200
<code>test_solution()</code>	(in module <i>sfepy.tests.test_homogenization_perfusion</i>), 1193	<code>test_verbose_output()</code>	(in module <i>sfepy.tests.test_base</i>), 1189
<code>test_solution()</code>	(in module <i>sfepy.tests.test_hyperelastic_tlul</i>), 1194	<code>test_volume()</code>	(in module <i>sfepy.tests.test_volume</i>), 1201
<code>test_solution()</code>	(in module <i>sfepy.tests.test_laplace_unit_square</i>), 1194	<code>test_volume_t1()</code>	(in module <i>sfepy.tests.test_volume</i>), 1201
<code>test_solvers()</code>	(in module <i>sfepy.tests.test_linear_solvers</i>), 1195	<code>test_wave_speeds()</code>	(in module <i>sfepy.tests.test_matcoefs</i>), 1195
<code>test_solving()</code>	(in module <i>sfepy.tests.test_high_level</i>), 1193	<code>test_weight_consistency()</code>	(in module <i>sfepy.tests.test_quadratures</i>), 1198
<code>test_sparse_matrix_hdf5()</code>	(in module <i>sfepy.tests.test_io</i>), 1194	<code>test_write_read_meshes()</code>	(in module <i>sfepy.tests.test_meshio</i>), 1197
<code>test_spbox_2d()</code>	(in module <i>sfepy.tests.test_splinebox</i>), 1199	<code>TestAdvectDGFluxTerm</code>	(class in <i>sfepy.tests.test_dg_terms_calls</i>), 1191
<code>test_spbox_3d()</code>	(in module <i>sfepy.tests.test_splinebox</i>), 1199	<code>TestDGField</code>	(class in <i>sfepy.tests.test_dg_field</i>), 1190
<code>test_spbox_field()</code>	(in module <i>sfepy.tests.test_splinebox</i>), 1199	<code>TestDiffusionDGFluxTerm</code>	(class in <i>sfepy.tests.test_dg_terms_calls</i>), 1191
		<code>TestDiffusionInteriorPenaltyTerm</code>	(class in <i>sfepy.tests.test_dg_terms_calls</i>), 1191

- sfepy.tests.test_dg_terms_calls*), 1191
- `TestNonlinearHyperDGFluxTerm` (class in *sfepy.tests.test_dg_terms_calls*), 1191
- `TestNonlinScalarDotGradTerm` (class in *sfepy.tests.test_dg_terms_calls*), 1191
- `tetrahedralize_vtk_mesh()` (in module *sfepy.postprocess.utils_vtk*), 1016
- `tetrvolume()` (in module *sfepy.tests.test_splinebox*), 1200
- `texpr` (*sfepy.terms.terms_sensitivity.ESDStokesTerm* attribute), 1169
- `texpr` (*sfepy.terms.terms_sensitivity.ESDVectorDotGradScalarTerm* attribute), 1170
- `THTerm` (class in *sfepy.terms.terms_th*), 1177
- `tile_mat()` (*sfepy.terms.terms.Term* static method), 1053
- `tilted_mesh1d()` (in module *sfepy.mesh.mesh_generators*), 1004
- `time_update()` (*sfepy.discrete.equations.Equations* method), 816
- `time_update()` (*sfepy.discrete.materials.Material* method), 825
- `time_update()` (*sfepy.discrete.materials.Materials* method), 826
- `time_update()` (*sfepy.discrete.problem.Problem* method), 843
- `time_update()` (*sfepy.discrete.variables.FieldVariable* method), 852
- `time_update()` (*sfepy.discrete.variables.Variable* method), 853
- `time_update()` (*sfepy.discrete.variables.Variables* method), 857
- `time_update()` (*sfepy.terms.terms.Term* method), 1053
- `time_update_materials()` (*sfepy.discrete.equations.Equations* method), 817
- `Timer` (class in *sfepy.base.timing*), 807
- `Timers` (class in *sfepy.base.timing*), 807
- `TimesSequenceTSC` (class in *sfepy.solvers.ts_controllers*), 1041
- `TimeStepController` (class in *sfepy.solvers.solvers*), 1037
- `TimeStepper` (class in *sfepy.solvers.ts*), 1038
- `TimeSteppingSolver` (class in *sfepy.solvers.solvers*), 1037
- `TLMembraneTerm` (class in *sfepy.terms.terms_membrane*), 1136
- `to_dict()` (*sfepy.base.base.Struct* method), 782
- `to_file_hdf5()` (*sfepy.homogenization.coefficients.Coefficients* method), 951
- `to_file_latex()` (*sfepy.homogenization.coefficients.Coefficients* method), 951
- `to_file_txt` (*sfepy.homogenization.coefs_phononic.AcousticMassTensor* attribute), 955
- `to_file_txt` (*sfepy.homogenization.coefs_phononic.AppliedLoadTensor* attribute), 955
- `to_file_txt()` (*sfepy.homogenization.coefficients.Coefficients* method), 952
- `to_file_txt()` (*sfepy.homogenization.coefs_phononic.BandGaps* static method), 956
- `to_file_txt()` (*sfepy.homogenization.coefs_phononic.DensityVolumeInfo* static method), 956
- `to_latex()` (*sfepy.homogenization.coefficients.Coefficients* method), 952
- `to_list()` (in module *gen_term_table*), 1206
- `to_numpy_array()` (in module *sfepy.mesh.bspline*), 998
- `to_poly_file()` (*sfepy.mesh.geom_tools.geometry* method), 1000
- `to_stack()` (in module *sfepy.discrete.parse_regions*), 827
- `transform_coors_to_lower_dim()` (in module *sfepy.discrete.fem.mappings*), 897
- `transform()` (*sfepy.terms.terms_multilinear.ExpressionBuilder* method), 1148
- `transform_asm_matrices()` (in module *sfepy.mechanics.membranes*), 985
- `transform_asm_matrices()` (in module *sfepy.mechanics.shell10x*), 987
- `transform_asm_vectors()` (in module *sfepy.mechanics.membranes*), 985
- `transform_bar_to_space_coors()` (in module *sfepy.linalg.geometry*), 973
- `transform_basis()` (in module *sfepy.discrete.common.poly_spaces*), 873
- `transform_conditions()` (in module *sfepy.base.conf*), 792
- `transform_coors()` (*sfepy.discrete.fem.mesh.Mesh* method), 899
- `transform_data()` (in module *sfepy.mechanics.tensors*), 990
- `transform_dgebcs()` (in module *sfepy.base.conf*), 792
- `transform_dgepbcs()` (in module *sfepy.base.conf*), 792
- `transform_ebcs()` (in module *sfepy.base.conf*), 792
- `transform_epbcs()` (in module *sfepy.base.conf*), 792
- `transform_equations_ed()` (in module *sfepy.solvers.ts_solvers*), 1050
- `transform_fields()` (in module *sfepy.base.conf*), 792
- `transform_functions()` (in module *sfepy.base.conf*), 792
- `transform_ics()` (in module *sfepy.base.conf*), 792
- `transform_input()` (*sfepy.base.conf.ProblemConf* method), 791
- `transform_input_trivial()` (*sfepy.base.conf.ProblemConf* method), 791
- `transform_integrals()` (in module *sfepy.base.conf*), 792
- `transform_lcbcs()` (in module *sfepy.base.conf*), 792

- transform_materials() (in module *sfepy.base.conf*), 792
- transform_plot_data() (in module *sfepy.homogenization.band_gaps_app*), 951
- transform_regions() (in module *sfepy.base.conf*), 792
- transform_solvers() (in module *sfepy.base.conf*), 792
- transform_to_i_struct_1() (in module *sfepy.base.conf*), 792
- transform_to_struct_01() (in module *sfepy.base.conf*), 792
- transform_to_struct_1() (in module *sfepy.base.conf*), 792
- transform_to_struct_10() (in module *sfepy.base.conf*), 792
- transform_variables() (in module *sfepy.base.conf*), 792
- TransformToPlane (class in *sfepy.mechanics.matcoefs*), 980
- translate_fargs_mapping() (*sfepy.terms.terms.Term* static method), 1054
- treat_pbc() (*sfepy.discrete.fem.lcbc_operators.MRLCBCOperator* method), 893
- triangulate() (in module *sfepy.mesh.mesh_tools*), 1005
- trim() (in module *gen_solver_table*), 1205
- try_block() (in module *sfepy.base.resolve_deps*), 807
- try_imports() (in module *sfepy.base.base*), 785
- try_presolve() (*sfepy.discrete.problem.Problem* method), 843
- try_set_defaults() (in module *sfepy.homogenization.band_gaps_app*), 951
- TSTimes (class in *sfepy.homogenization.coefs_base*), 954
- tuple_to_conf() (in module *sfepy.base.conf*), 792
- TVDRK3StepSolver (class in *sfepy.solvers.ts_dg_solvers*), 933
- typeset() (in module *gen_field_table*), 1203
- typeset() (in module *gen_solver_table*), 1205
- typeset() (in module *gen_term_table*), 1206
- typeset_examples() (in module *gen_term_table*), 1206
- typeset_field_table() (in module *gen_field_table*), 1203
- typeset_solvers_table() (in module *gen_solver_table*), 1205
- typeset_term_syntax() (in module *gen_term_table*), 1206
- typeset_term_table() (in module *gen_term_table*), 1206
- typeset_term_tables() (in module *gen_term_table*), 1206
- typeset_to_indent() (in module *gen_term_table*), 1206
- typeset_to_indent() (in module *sfepy.solvers.solvers*), 1037
- ## U
- Uncached (class in *sfepy.base.ioutils*), 795
- unique() (in module *sfepy.base.compat*), 787
- unique_rows() (in module *sfepy.linalg.utils*), 978
- Unit (class in *sfepy.mechanics.units*), 991
- unpack_data() (*sfepy.base.ioutils.DataMarker* method), 793
- unpack_data() (*sfepy.base.ioutils.DataSoftLink* method), 793
- unpack_data() (*sfepy.base.ioutils.HDF5BaseData* method), 794
- update() (*sfepy.base.base.Container* method), 780
- update() (*sfepy.base.base.Struct* method), 782
- update() (*sfepy.base.multiproc_mpi.RemoteDict* method), 802
- update() (*sfepy.terms.terms_contact.ContactInfo* method), 1077
- update_conf() (*sfepy.base.conf.ProblemConf* method), 791
- update_data() (*sfepy.discrete.materials.Material* method), 825
- update_dict_recursively() (in module *sfepy.base.base*), 785
- update_equations() (*sfepy.discrete.problem.Problem* method), 843
- update_expression() (*sfepy.terms.terms.Terms* method), 1054
- update_materials() (*sfepy.discrete.problem.Problem* method), 844
- update_micro_states() (*sfepy.homogenization.homogen_app.HomogenizationApp* method), 964
- update_shape() (*sfepy.discrete.common.region.Region* method), 876
- update_special_constant_data() (*sfepy.discrete.materials.Material* method), 826
- update_special_data() (*sfepy.discrete.materials.Material* method), 826
- update_supported_formats() (in module *sfepy.discrete.fem.meshio*), 906
- update_time_stepper() (*sfepy.discrete.problem.Problem* method), 844
- use_first_available() (in module *sfepy.solvers.solvers*), 1037
- use_method_with_name() (in module *sfepy.base.base*), 786
- user_options (*build_helpers.NoOptionsDocs* attribute), 1202

UserMeshIO (class in *sfepy.discrete.fem.meshio*), 906

V

validate (*sfepy.base.goptions.ValidatedDict* attribute), 793

validate() (*sfepy.base.conf.ProblemConf* method), 791

validate_bool() (in module *sfepy.base.goptions*), 793

ValidatedDict (class in *sfepy.base.goptions*), 793

value (*sfepy.base.multiproc_mpi.RemoteInt* attribute), 802

values() (*sfepy.base.goptions.ValidatedDict* method), 793

var (in module *sfepy.discrete.fem.meshio*), 906

Variable (class in *sfepy.discrete.variables*), 852

Variables (class in *sfepy.discrete.variables*), 853

VariableTimeStepper (class in *sfepy.solvers.ts*), 1038

VectorDotGradScalarTerm (class in *sfepy.terms.terms_dot*), 1094

VectorDotScalarTerm (class in *sfepy.terms.terms_dot*), 1095

VelocityVerletTS (class in *sfepy.solvers.ts_solvers*), 1048

verbosity (*sfepy.terms.terms_multilinear.ETermBase* attribute), 1147

verify_task_dof_maps() (in module *sfepy.parallel.parallel*), 1009

vertex_groups (*sfepy.discrete.common.extmods.cmesh.CMesh* attribute), 863

vertices (*sfepy.discrete.common.region.Region* property), 876

view_petsc_local() (in module *sfepy.parallel.parallel*), 1010

visit_stack() (in module *sfepy.discrete.parse_regions*), 827

volume (class in *sfepy.mesh.geom_tools*), 1000

VolumeFractions (class in *sfepy.homogenization.coefs_base*), 954

VolumeSurfaceTerm (class in *sfepy.terms.terms_basic*), 1067

VolumeSurfaceTLTerm (class in *sfepy.terms.terms_hyperelastic_tl*), 1122

VolumeTerm (class in *sfepy.terms.terms_basic*), 1068

VolumeTLTerm (class in *sfepy.terms.terms_hyperelastic_tl*), 1123

VolumeULTerm (class in *sfepy.terms.terms_hyperelastic_ul*), 1127

VolumeXTerm (class in *sfepy.terms.terms_compat*), 1075

W

wait_for_tag() (in module *sfepy.base.multiproc_mpi*), 803

wandering_element() (in module *sfepy.discrete.simplex_cubature*), 847

wave_speeds_from_youngpoisson() (in module *sfepy.mechanics.matcoefs*), 982

weak_dp_function() (*sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm* static method), 1117

weak_dp_function() (*sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm* static method), 1125

weak_function() (*sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm* static method), 1117

weak_function() (*sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBaseTerm* static method), 1119

weak_function() (*sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm* static method), 1125

weak_function() (*sfepy.terms.terms_hyperelastic_ul.HyperElasticULBaseTerm* static method), 1126

weak_function() (*sfepy.terms.terms_membrane.TLMembraneTerm* static method), 1136

wrap_function() (in module *sfepy.solvers.optimize*), 1033

write() (*sfepy.base.ioutils.HDF5Data* method), 794

write() (*sfepy.base.ioutils.SoftLink* method), 795

write() (*sfepy.base.multiproc_mpi.MPILogFile* method), 802

write() (*sfepy.discrete.fem.mesh.Mesh* method), 899

write() (*sfepy.discrete.fem.meshio.ANSYSCDBMeshIO* method), 900

write() (*sfepy.discrete.fem.meshio.ComsolMeshIO* method), 900

write() (*sfepy.discrete.fem.meshio.GmshIO* method), 901

write() (*sfepy.discrete.fem.meshio.HDF5MeshIO* method), 903

write() (*sfepy.discrete.fem.meshio.HDF5XdmfMeshIO* method), 903

write() (*sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO* method), 903

write() (*sfepy.discrete.fem.meshio.MeshIO* method), 905

write() (*sfepy.discrete.fem.meshio.MeshioLibIO* method), 906

write() (*sfepy.discrete.fem.meshio.NEUMeshIO* method), 906

write() (*sfepy.discrete.fem.meshio.UserMeshIO* method), 906

write() (*sfepy.discrete.fem.meshio.XYZMeshIO* method), 906

write_control_net() (*sfepy.mesh.splinebox.SplineBox* method), 1007

write_control_polygon_vtk() (*sfepy.mesh.bspline.BSplineSurf* method), 998

write_data() (*sfepy.base.ioutils.DataSoftLink* method), 794

write_data() (*sfepy.base.ioutils.HDF5Data* method),

794
 write_dict_hdf5() (in module *sfepy.base.ioutils*), 797
 write_domain_to_hdf5()
 (*sfepy.discrete.iga.domain.IGDomain* method),
 934
 write_iga_data() (in module *sfepy.discrete.iga.io*),
 947
 write_log() (in module *sfepy.base.log*), 800
 write_mesh_to_hdf5()
 (*sfepy.discrete.fem.meshio.HDF5MeshIO*
 static method), 903
 write_results() (in module *sfepy.discrete.probes*),
 831
 write_sparse_matrix_hdf5() (in module
 sfepy.base.ioutils), 797
 write_sparse_matrix_to_hdf5() (in module
 sfepy.base.ioutils), 797
 write_surface_vtk() (*sfepy.mesh.bspline.BSplineSurf*
 method), 998
 write_to_hdf5() (in module *sfepy.base.ioutils*), 798
 write_vtk_to_file() (in module
 sfepy.postprocess.utils_vtk), 1016
 write_xdmf_file() (*sfepy.discrete.fem.meshio.HDF5MeshIO*
 static method), 903

X

XYZMeshIO (class in *sfepy.discrete.fem.meshio*), 906

Y

youngpoisson_from_stiffness() (in module
 sfepy.mechanics.matcoefs), 982
 youngpoisson_from_wave_speeds() (in module
 sfepy.mechanics.matcoefs), 982

Z

zero_dofs() (*sfepy.discrete.conditions.Conditions*
 method), 808
 zero_dofs() (*sfepy.discrete.conditions.EssentialBC*
 method), 809
 ZeroTerm (class in *sfepy.terms.terms_basic*), 1068