

---

# SfePy Documentation

*Release 2019.2*

**Robert Cimrman and Contributors**

Jul 02, 2019



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Supported Platforms . . . . .	4
2.2	Requirements . . . . .	4
2.3	Installing SfePy . . . . .	5
2.4	Installing SfePy from Sources . . . . .	5
2.5	Testing Installation . . . . .	6
2.6	Debugging . . . . .	7
2.7	Using IPython . . . . .	7
2.8	Notes on Multi-platform Python Distributions . . . . .	8
2.9	Notes on Installing SfePy Dependencies on Various Platforms . . . . .	9
<b>3</b>	<b>Tutorial</b>	<b>11</b>
3.1	Basic <i>SfePy</i> Usage . . . . .	11
3.2	Basic Notions . . . . .	12
3.3	Running a Simulation . . . . .	13
3.4	Example Problem Description File . . . . .	14
3.5	Interactive Example: Linear Elasticity . . . . .	17
<b>4</b>	<b>User's Guide</b>	<b>25</b>
4.1	Running a Simulation . . . . .	26
4.2	Visualization of Results . . . . .	30
4.3	Problem Description File . . . . .	33
4.4	Building Equations in SfePy . . . . .	46
4.5	Term Evaluation . . . . .	47
4.6	Solution Postprocessing . . . . .	48
4.7	Probing . . . . .	49
4.8	Postprocessing filters . . . . .	49
4.9	Solvers . . . . .	50
4.10	Solving Problems in Parallel . . . . .	52
4.11	Isogeometric Analysis . . . . .	53
<b>5</b>	<b>Examples</b>	<b>55</b>
5.1	Primer . . . . .	55
5.2	Using Salome with SfePy . . . . .	78
5.3	Preprocessing: <i>FreeCAD/OpenSCAD + Gmsh</i> . . . . .	81
5.4	Material Identification . . . . .	90
5.5	Mesh parametrization . . . . .	94
5.6	Examples . . . . .	98

5.7	Example Applications . . . . .	477
<b>6</b>	<b>Theoretical Background</b>	<b>479</b>
6.1	Notes on solving PDEs by the Finite Element Method . . . . .	479
6.2	Implementation of Essential Boundary Conditions . . . . .	481
<b>7</b>	<b>Developer Guide</b>	<b>485</b>
7.1	Retrieving the Latest Code . . . . .	486
7.2	SfePy Directory Structure . . . . .	487
7.3	Exploring the Code . . . . .	488
7.4	How to Contribute . . . . .	488
7.5	How to Regenerate Documentation . . . . .	493
7.6	How to Implement a New Term . . . . .	494
7.7	How To Make a Release . . . . .	500
7.8	Module Index . . . . .	502
<b>8</b>	<b>Indices and tables</b>	<b>819</b>
<b>Bibliography</b>		<b>821</b>
<b>Python Module Index</b>		<b>823</b>
<b>Index</b>		<b>827</b>

---

**CHAPTER  
ONE**

---

## **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](#) and [Tutorial](#). Some more advanced features are discussed in [Primer](#).



---

CHAPTER  
TWO

---

## INSTALLATION

### Table of Contents

- *Supported Platforms*
- *Requirements*
  - *Notes on selecting Python Distribution*
- *Installing SfePy*
- *Installing SfePy from Sources*
  - *Compilation of C Extension Modules*
  - *Installation*
- *Testing Installation*
  - *Running Automated Test Suite*
  - *Running Failing Tests in Raise Mode*
- *Debugging*
- *Using IPython*
- *Notes on Multi-platform Python Distributions*
  - *Anaconda*
    - \* *Compilation of C Extension Modules on Windows*
- *Notes on Installing SfePy Dependencies on Various Platforms*
  - *Gentoo*
  - *Archlinux*
    - \* *Instructions*
  - *Debian*
  - *(K)Ubuntu*

## 2.1 Supported Platforms

*SfePy* is known to work on various flavors of recent Linux, Intel-based MacOS and Windows. It can be used with both Python 2.7 and Python 3.x.

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.

## 2.2 Requirements

Installation prerequisites, required to build *SfePy*:

- a C compiler suite,
- Python 2.7 or 3.x,
- NumPy,
- Cython.

Python packages required for using *SfePy*:

- Pyparsing,
- SciPy,
- scikit-umfpack for enabling UMFPACK solver for SciPy  $\geq 0.14.0$ ,
- Matplotlib for various plots, GTKAgg for live plotting via `log.py`,
- PyTables for storing results in HDF5 files,
- SymPy for some tests and functions,
- Mayavi for `postproc.py`,
- Pysparse for additional eigenvalue problem solvers (currently available for Python 2.7.x only),
- igakit for script/gen\_iga\_patch.py - simple IGA domain generator,
- petsc4py and mpi4py for running parallel examples and using parallel solvers from PETSc,
- pymetis for mesh partitioning using Metis,
- wxPython for better IPython integration.

Make sure the dependencies of those packages are also installed (e.g. `igakit` requires FORTRAN compiler, `scikit-umfpack` does not work without UMFPACK, `petsc4py` without PETSc etc.).

*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 `pexpect` and `gmsh` or `tetgen`.

- IPython is recommended over the regular Python shell to fluently follow some parts of primer/tutorial (see [Using IPython](#)).
- MUMPS library for using MUMPS linear direct solver (real and complex arithmetic, parallel factorization)

### 2.2.1 Notes on selecting Python Distribution

SfePy should work with any recent Python 2.7 or 3.x (in long-term view Python 3.6+ is recommended). It is only matter of taste to use either native OS Python installation or any other suitable distribution. We could recommend the following distributions to use:

- **Linux:** OS native installation (See [Notes on Installing SfePy Dependencies on Various Platforms](#) for further details.)
- **macOS:** multi-platform scientific Python distributions [Anaconda](#) (See [Notes on Multi-platform Python Distributions](#) for further details.)
- **Windows:** use free versions of commercial multi-platform scientific Python distributions [Anaconda](#) or [Enthought Canopy](#) (see [Notes on Multi-platform Python Distributions](#) for further details). In addition a completely free open-source portable distribution [WinPython](#) can be used.

On any supported platform we could recommend [Anaconda](#) distribution as easy-to-use, stable and up-to-date Python distribution with all the required dependencies (including pre-built `sfePy` package).

Note: all *SfePy* releases are regularly tested on recent Linux distributions (Debian and (K)Ubuntu) using OS Python installation and Anaconda, macOS 10.12+ using Anaconda and Windows 8.1+ using Anaconda.

## 2.3 Installing SfePy

For [Anaconda](#) and `.deb` based Linux distributions (Debian, (K)Ubuntu), pre-built *SfePy* packages are available. You may directly install them with:

- [Anaconda](#) distribution: install `sfePy` from `conda-forge` channel:

```
conda install -c conda-forge sfePy
```

- Debian/(K)Ubuntu: install `python-sfePy`:

```
sudo apt-get install python-sfePy
```

There are no further steps required to install/configure *SfePy* (see [Notes on Multi-platform Python Distributions](#) for additional notes).

## 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.

### 2.4.1 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. Compile the extension modules

- for in-place use:

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

- for installation:

```
python setup.py build
```

We recommend starting with the in-place build.

### 2.4.2 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):

```
python setup.py install
```

- local (requires write access to <installation prefix>):

```
python setup.py install --root=<installation prefix>
```

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

## 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.

### 2.5.1 Running Automated Test Suite

Depending on type of your build run following tests:

- in-place build:

```
python ./run_tests.py
```

or

```
python ./sfepy-run run_tests
```

or (on Unix-based systems) directly by

```
./sfepy-run run_tests
```

- installed (local or system-wide) build:

```
sfepy-run run_tests
```

(This command creates a directory called 'output' in the current directory as well as some other auxiliary files. Use the in-place build testing if you do not want to care about this.)

Please note this method is *not fully supported on Windows systems yet*. It can be currently used only with pre-compiled *sfepy* packages (see [Installing SfePy](#)).

## 2.5.2 Running Failing Tests in Raise Mode

If a particular test fails, run it in the raise mode

```
python sfepy-run run_tests --raise tests/<failing_test_name.py>
```

and, please, report the output to the [SfePy mailing list](#).

It is also possible to automatically start a debugger when/if an exception is raised by running a test in the debug mode:

```
python sfepy-run run_tests --debug tests/failing_test_name.py
```

## 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](#).

## 2.7 Using IPython

We generally recommend to use (a customized) [IPython](#) interactive shell over the regular Python interpreter when following [Tutorial](#) or [Primer](#) (or even for any regular interactive work with *SfePy*).

Install [IPython](#) (as a generic part of your selected distribution) and then customize it to your choice.

Depending on your IPython usage, you can customize your *default* profile or create a *SfePy* specific new one as follows:

1. Create a new *SfePy* profile:

```
ipython profile create sfepy
```

2. Open the `~/.ipython/profile_sfepy/ipython_config.py` file in a text editor and add/edit after the `c = get_config()` line:

```
exec_lines = [
    'import numpy as nm',
    'import matplotlib as mpl',
    'mpl.use("WXAgg")',
#
# Add your preferred SfePy customization here...
#
]

c.InteractiveShellApp.exec_lines = exec_lines
c.TerminalIPythonApp.gui = 'wx'
c.TerminalInteractiveShell.colors = 'Linux' # NoColor, Linux, or LightBG
```

Please note, that generally it is not recommended to use *star* (\*) imports here.

3. Run the customized IPython shell:

```
ipython --profile=sfepy
```

## 2.8 Notes on Multi-platform Python Distributions

### 2.8.1 Anaconda

We highly recommend this scientific-oriented Python distribution.

(Currently regularly tested by developers on *SfePy* releases with Python 2.7/3.6 64-bit on Ubuntu 16.04 LTS, Windows 8.1+ and macOS 10.12+.)

Download appropriate [Anaconda](#) Python 2.7 or 3.x installer package and follow install instructions. We recommend to choose *user-level* install option (no admin privileges required).

Anaconda can be used for:

1. installing the latest release of *SfePy* directly from the [conda-forge](#) channel (see [sfepy-feedstock](#)). In this case, follow the instructions in [Installing SfePy](#).

Installing/upgrading *SfePy* from the conda-forge channel can also be achieved by adding [conda-forge](#) to your channels with:

```
conda config --add channels conda-forge
```

Once the [conda-forge](#) channel has been enabled, *SfePy* can be installed with:

```
conda install sfepy
```

It is possible to list all of the versions of *SfePy* available on your platform with:

```
conda search sfepy --channel conda-forge
```

2. installing the *SfePy* dependencies only - then proceed with the [Installing SfePy from Sources](#) instructions.

In this case, install the missing/required packages using built-in *conda* package manager:

```
conda install mayavi wxpython
```

See *conda help* for further information.

Occasionally, you should check for distribution and/or installed packages updates (there is no built-in automatic update mechanism available):

```
conda update conda
conda update anaconda
conda update <package>
```

or try:

```
conda update --all
```

## Compilation of C Extension Modules on Windows

To build *SfePy* extension modules, included `mingw-w32/64` compiler tools should work fine. If you encounter any problems, we recommend to install and use Microsoft Visual C++ Build Tools instead (see [Anaconda FAQ](#)).

## 2.9 Notes on Installing SfePy Dependencies on Various Platforms

The following information has been provided by users of the listed platforms and may become obsolete over time. The generic installation instructions above should work in any case, provided the required dependencies are installed.

### 2.9.1 Gentoo

```
emerge -va pytables pyparsing numpy scipy matplotlib ipython mayavi
```

### 2.9.2 Archlinux

```
pacman -S python2-numpy python2-scipy python2-matplotlib ipython2 python2-sympy
yaourt -S python-pytables python2-mayavi
```

### Instructions

Edit Makefile and change all references from python to python2. Edit scripts and change shebangs to point to python2.

### 2.9.3 Debian

(Old instructions, check also [\*\(K\)Ubuntu\*](#) below.)

First, you have to install the dependencies packages:

```
apt-get install python-tables python-pyparsing python-matplotlib python-scipy
```

Than *SfePy* can be installed with:

```
apt-get install python-sfepy
```

## 2.9.4 (K)Ubuntu

(Tested on Kubuntu 16.04 LTS.)

First, you have to install the dependencies packages (if *apt-get* is not installed, install it or try *apt-get install* instead):

```
sudo apt-get install python-scipy python-matplotlib python-tables python-pyparsing  
  ↵ libbsuitesparse-dev python-setuptools mayavi2 python-dev ipython python-sympy cython  
  ↵ python-sparse
```

Than *SfePy* can be installed with:

```
apt-get install python-sfepy
```

### Table of Contents

- *Basic SfePy Usage*
  - *Invoking SfePy from the Command Line*
  - *Using SfePy Interactively*
- *Basic Notions*
  - *Sneak Peek: What is Going on Under the Hood*
- *Running a Simulation*
  - *Postprocessing the Results*
- *Example Problem Description File*
- *Interactive Example: Linear Elasticity*
  - *Complete Example as a Script*

## 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*.

### 3.1.1 Invoking SfePy from the Command Line

This section introduces the basics of running *SfePy* from the command line.

The script *simple.py* is the **most basic starting point** in *SfePy*. It can be invoked in many (similar) ways which depends on used OS, Python distribution and *SfePy* build method (see *Installing SfePy* for further info). All (working) alternatives described below are interchangeable, so don't panic and feel free to pick your preferred choice (see *Basic Usage* for further explanation and more usage examples).

Depending on selected build method and OS used we recommend for:

- In-place build

Use the top-level directory of *SfePy* source tree as your working directory and use:

```
./simple.py <problem_description_file>
```

or (particularly on Windows based systems)

```
python ./simple.py <problem_description_file>
```

- Installed (local or system-wide) build

Use any working directory including your *problem description file* and use:

```
python <path/to/installed/simple.py> <problem_description_file>
```

or simply (on Unix based systems)

```
<path/to/installed/simple.py> <problem_description_file>
```

You can also use the simple *SfePy command-wrapper* (ensure that *SfePy* installation *executable* directory is included in your PATH):

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

Please note, that improper mixing of *in-place* and *install* builds on single command line may result in strange runtime errors.

### 3.1.2 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. You can customize your *IPython* startup profile as described in [Using IPython](#).

## 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 [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...

### 3.2.1 Sneak Peek: What is Going on Under the Hood

1. A top-level script (usually *simple.py* as in this tutorial) 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.

## 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.

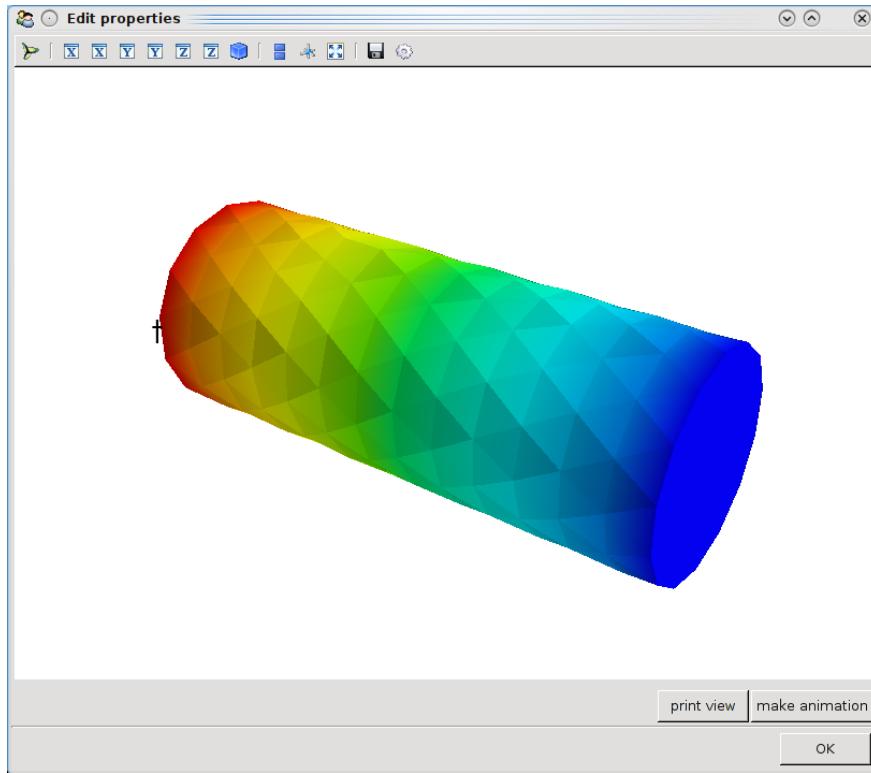
- Download examples/diffusion/poisson\_short\_syntax.py. It represents our sample *SfePy Problem Description File*, which defines the problem to be solved in terms *SfePy* can understand.
- Use the downloaded file in place of <*problem\_description\_file.py*> and run *simple.py* as *described above*. The successful execution of the command creates output file *cylinder.vtk* in the *SfePy* top-level directory.

### 3.3.1 Postprocessing the Results

- The *postproc.py* script can be used for quick postprocessing and visualization of the *SfePy* output files. It requires *mayavi* installed on your system.
- As a simple example, try:

```
./postproc.py 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.

## 3.4 Example Problem Description File

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

The problem at hand is the following:

$$c\Delta T = f \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma, \quad (3.1)$$

where  $\Gamma \subseteq \Omega$  is a subset of the domain  $\Omega$  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 (3.1) in *weak formulation* (6.5). The  $f = 0$ ,  $g = \frac{\partial T}{\partial n} = 0$ . So only one term in weak form (6.5) remains:

$$\int_{\Omega} c \nabla T \cdot \nabla s = 0, \quad \forall s \in V_0. \quad (3.2)$$

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 `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`', it's 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}),
}
```

```
'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 (3.2), 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](#).

## 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:

$$-\frac{\partial \sigma_{ij}(\underline{u})}{\partial x_j} + f_i = 0 \text{ in } \Omega, \quad \underline{u} = 0 \text{ on } \Gamma_1, \quad u_1 = \bar{u}_1 \text{ on } \Gamma_2, \quad (3.3)$$

where the stress is defined as  $\sigma_{ij} = 2\mu e_{ij} + \lambda e_{kk} \delta_{ij}$ ,  $\lambda, \mu$  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 (3.3) is

$$\int_{\Omega} D_{ijkl}e_{kl}(\underline{u})e_{ij}(\underline{v}) + \int_{\Omega} f_i v_i = 0, \quad (3.4)$$

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  $\Omega$ .

```
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  $\Omega$ , where the solution is sought, and  $\Gamma_1, \Gamma_2$ , 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  $\mathbf{u}$  and the test variable  $\mathbf{v}$ .

```
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  $\Gamma_1$ , and shift the right edge  $\Gamma_2$  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', gammal, {'u.all' : 0.0})
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.

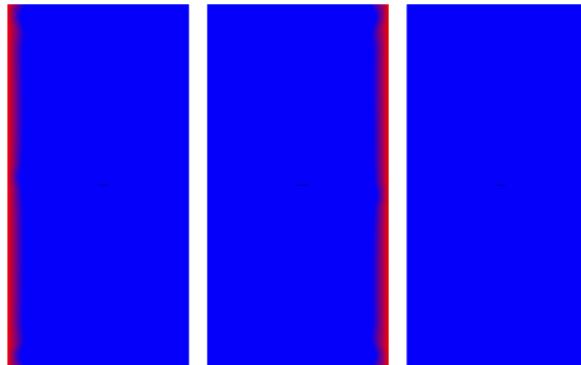
```
In [35]: pb.save_regions_as_groups('regions')
```

And view them.

```
In [36]: from sfepy.postprocess.viewer import Viewer

In [37]: view = Viewer('regions.vtk')
In [38]: view()
```

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(ebc_ids=Conditions([fix_u, shift_u]))
In [40]: pb.set_solver(nls)

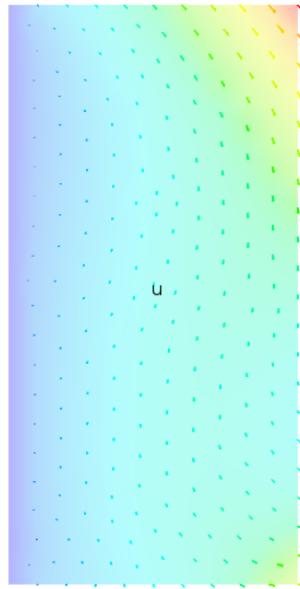
In [41]: status = IndexedStruct()
In [42]: vec = pb.solve(status=status)

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', vec)
In [46]: view = Viewer('linear_elasticity.vtk')
In [47]: view()
```

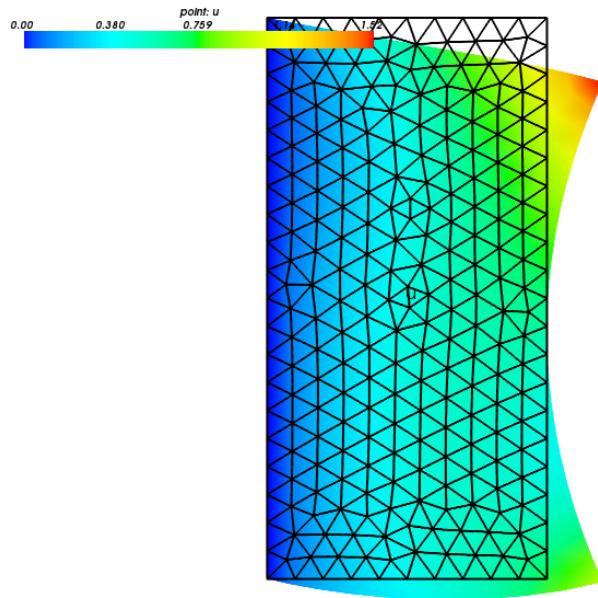
This is the resulting image:



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

```
In [48]: view(vector_mode='warp_norm', rel_scaling=2,
...:           is_scalar_bar=True, is_wireframe=True)
```

And the result is:



See the docstring of `view()` and play with its options.

### 3.5.1 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  from __future__ import print_function
3  from __future__ import absolute_import
4  from argparse import ArgumentParser
5  import numpy as nm
6
7  import sys
8  sys.path.append('..')
9
10 from sfepy.base.base import IndexedStruct
11 from sfepy.discrete import (FieldVariable, Material, Integral, Function,
12                            Equation, Equations, Problem)
13 from sfepy.discrete.fem import Mesh, FEDomain, Field
14 from sfepy.terms import Term
15 from sfepy.discrete.conditions import Conditions, EssentialBC
16 from sfepy.solvers.ls import ScipyDirect
17 from sfepy.solvers.nls import Newton
18 from sfepy.postprocess.viewer import Viewer
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):
23     """
24     Define a displacement depending on the y coordinate.
25     """
26     val = shift * coors[:, 1]**2
27

```

```
28     return val
29
30 helps = {
31     'show' : 'show the results figure',
32 }
33
34 def main():
35     from sfepy import data_dir
36
37     parser = ArgumentParser()
38     parser.add_argument('--version', action='version', version='%(prog)s')
39     parser.add_argument('-s', '--show',
40                         action="store_true", dest='show',
41                         default=False, help=helps['show'])
42     options = parser.parse_args()
43
44     mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
45     domain = FEDomain('domain', mesh)
46
47     min_x, max_x = domain.get_mesh_bounding_box()[:,0]
48     eps = 1e-8 * (max_x - min_x)
49     omega = domain.create_region('Omega', 'all')
50     gammal = domain.create_region('Gammal',
51                                   'vertices in x < %.10f' % (min_x + eps),
52                                   'facet')
53     gamma2 = domain.create_region('Gamma2',
54                                   'vertices in x > %.10f' % (max_x - eps),
55                                   'facet')
56
57     field = Field.from_args('fu', nm.float64, 'vector', omega,
58                             approx_order=2)
59
60     u = FieldVariable('u', 'unknown', field)
61     v = FieldVariable('v', 'test', field, primary_var_name='u')
62
63     m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
64     f = Material('f', val=[[0.02], [0.01]])
65
66     integral = Integral('i', order=3)
67
68     t1 = Term.new('dw_lin_elastic(m.D, v, u)',
69                  integral, omega, m=m, v=v, u=u)
70     t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
71     eq = Equation('balance', t1 + t2)
72     eqs = Equations([eq])
73
74     fix_u = EssentialBC('fix_u', gammal, {'u.all' : 0.0})
75
76     bc_fun = Function('shift_u_fun', shift_u_fun,
77                       extra_args={'shift' : 0.01})
78     shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})
79
80     ls = ScipyDirect({})
81
82     nls_status = IndexedStruct()
83     nls = Newton({}, lin_solver=ls, status=nls_status)
84
85     pb = Problem('elasticity', equations=eqs)
```

```
86 pb.save_regions_as_groups('regions')
87
88 pb.set_bcs(ebcsl=Conditions([fix_u, shift_u]))
89
90 pb.set_solver(nls)
91
92 status = IndexedStruct()
93 state = pb.solve(status=status)
94
95 print('Nonlinear solver status:\n', nls_status)
96 print('Stationary solver status:\n', status)
97
98 pb.save_state('linear_elasticity.vtk', state)
99
100 if options.show:
101     view = Viewer('linear_elasticity.vtk')
102     view(vector_mode='warp_norm', rel_scaling=2,
103           is_scalar_bar=True, is_wireframe=True)
104
105 if __name__ == '__main__':
106     main()
```



---

CHAPTER  
FOUR

---

USER'S GUIDE

**Table of Contents**

- *Running a Simulation*
  - *Basic Usage*
  - *Applications*
  - *Using Command Wrapper*
    - \* *Notes*
  - *Stand-Alone Examples*
  - *Running Tests*
    - \* *Common Tasks*
  - *Computations and Examples*
    - \* *Common Tasks*
- *Visualization of Results*
- *Problem Description File*
  - *Long Syntax*
  - *FE Mesh*
  - *Regions*
    - \* *Region Definition Syntax*
  - *Fields*
  - *Variables*
  - *Integrals*
  - *Essential Boundary Conditions and Constraints*
    - \* *Dirichlet Boundary Conditions*
    - \* *Periodic Boundary Conditions*
    - \* *Linear Combination Boundary Conditions*
  - *Initial Conditions*
  - *Materials*

- *Equations and Terms*
  - \* *Examples*
- *Configuring Solvers*
- *Functions*
  - \* *Defining Material Parameters*
  - \* *Examples*
- *Miscellaneous*
- *Building Equations in SfePy*
  - *Syntax of Terms in Equations*
- *Term Evaluation*
- *Solution Postprocessing*
- *Probing*
- *Postprocessing filters*
- *Solvers*
  - *Time-stepping solvers*
  - *Nonlinear Solvers*
  - *Linear Solvers*
  - *Virtual Linear Solvers with Automatic Selection*
  - *Eigenvalue Problem Solvers*
  - *Quadratic Eigenvalue Problem Solvers*
  - *Optimization Solvers*
- *Solving Problems in Parallel*
  - *Current Implementation Drawbacks*
  - *Examples*
- *Isogeometric Analysis*
  - *Current Implementation*
    - \* *Domain Description*
    - \* *Region Selection*
  - *Examples*

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

## 4.1 Running a Simulation

The following 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 info. The \$ indicates the command prompt of your terminal.

### 4.1.1 Basic Usage

- `$ ./simple.py examples/diffusion/poisson_short_syntax.py`
  - Creates cylinder.vtk
- `$ ./simple.py examples/navier_stokes/stokes.py`
  - Creates channels\_symm944t.vtk
- `$ ./run_tests.py`
  - See [Running Tests](#)

### 4.1.2 Applications

- Phononic Materials

```
– $ ./phonon.py -p examples/phononic/band_gaps.py
* see examples/phononic/output/
```

### 4.1.3 Using Command Wrapper

All top-level *SfePy* scripts (applications) can be run via single `sfePy-run` wrapper:

```
$ ./sfePy-run
usage: sfePy-run [command] [options]

Simple wrapper for main SfePy commands.

positional arguments:
(extractor,homogen,phonon,postproc,probe,run_tests,simple)
                    Available SfePy command(s).
options            Additional options passed directly to selected
                   [command].

optional arguments:
-h, --help          show this help message and exit
-v, --version       show program's version number and exit
-w, --window        use alternative (pythonw) interpreter
```

### Notes

- This is a “new” supported method. Any *SfePy* script can be still run as stand-alone (as mentioned above).
- Both “inplace” and “system-wide” installations are supported.

### 4.1.4 Stand-Alone Examples

- `$ python examples/rs_correctors.py`

- \$ python examples/compare\_elastic\_materials.py
- \$ python examples/live\_plot.py

### 4.1.5 Running Tests

The tests are run by the `run_tests.py` script. Run

```
$ ./run_tests.py -h
```

to get help.

#### Common Tasks

- Run all tests, filter output; result files related to the tests can be found in `output-tests` directory:

```
./run_tests.py  
./run_tests.py --filter-more  
./run_tests.py --filter-less
```

- Run a particular test file, filter output:

```
# Test if linear elasticity input file works.  
./run_tests.py tests/test_input_le.py
```

- Debug a failing test by automatically starting a debugger when/if an exception is raised

```
./run_tests.py tests/test_input_le.py --debug
```

- Raise silenced exceptions that could have occurred in a failing test:

```
./run_tests.py tests/test_input_le.py --raise
```

### 4.1.6 Computations and Examples

The example problems in the `examples` directory can be computed by the script `simple.py` which is in the top-level directory of the *SfePy* distribution. If it is run without arguments, a help message is printed:

```
$ ./simple.py  
Usage: simple.py [options] filename_in  
  
Solve partial differential equations given in a SfePy problem definition file.  
  
Example problem definition files can be found in ``examples/`` directory of the  
SfePy top-level directory. This script works with all the examples except those  
in ``examples/standalone/``.  
  
Both normal and parametric study runs are supported. A parametric study allows  
repeated runs for varying some of the simulation parameters - see  
``examples/diffusion/poisson_parametric_study.py`` file.  
  
Options:  
--version show program's version number and exit
```

```

-h, --help                  show this help message and exit
-c "key : value, ...", --conf="key : value, ..."
                           override problem description file items, written as
                           python dictionary without surrounding braces
-O "key : value, ...", --options="key : value, ..."
                           override options item of problem description, written
                           as python dictionary without surrounding braces
-d "key : value, ...", --define="key : value, ..."
                           pass given arguments written as python dictionary
                           without surrounding braces to define() function of
                           problem description file
-o filename                basename of output file(s) [default: <basename of
                           input file>]
--format=format             output file format, one of: {vtk, h5} [default: vtk]
--save-restart=mode         if given, save restart files according to the given
                           mode.
--load-restart=filename    if given, load the given restart file
--log=file                 log all messages to specified file (existing file will
                           be overwritten!)
-q, --quiet                do not print any messages to screen
--save-ebc                 save a zero solution with applied EBCs (Dirichlet
                           boundary conditions)
--save-ebc-nodes           save a zero solution with added non-zeros in EBC
                           (Dirichlet boundary conditions) nodes - scalar
                           variables are shown using colors, vector variables
                           using arrows with non-zero components corresponding to
                           constrained components
--save-regions             save problem regions as meshes
--save-regions-as-groups  save problem regions in a single mesh but mark them by
                           using different element/node group numbers
--save-field-meshes        save meshes of problem fields (with extra DOF nodes)
--solve-not                do not solve (use in connection with --save-*)
--list=what                list data, what can be one of: {terms, solvers}

```

Additional (stand-alone) examples are in the examples/ directory, e.g.:

```
$ python examples/compare_elastic_materials.py
```

Parametric study example:

```
$ ./simple.py examples/diffusion/poisson_parametric_study.py
```

## Common Tasks

- Run a simulation:

```

./simple.py examples/diffusion/poisson_short_syntax.py
./simple.py examples/diffusion/poisson_short_syntax.py -o some_results # ->
➥ produces some_results.vtk

```

- Print available terms:

```
./simple.py --list=terms
```

- Run a simulation and also save Dirichlet boundary conditions:

```
./simple.py --save-ebc 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:

```
./simple.py 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:

```
./simple.py 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:

```
./simple.py examples/large_deformation/balloon.py
```

and compare the residuals printed in the corresponding time steps.

## 4.2 Visualization of Results

The `postproc.py` script can be used for quick postprocessing and visualization of the *SfePy* results. It requires `mayavi2` installed on your system. Running `postproc.py` without arguments produces:

```
$ ./postproc.py
Usage: postproc.py [options] filename

This is a script for quick Mayavi-based visualizations of finite element
computations results.

Examples
-----
The examples assume that run_tests.py has been run successfully and the
resulting data files are present.

- view data in output-tests/test_navier_stokes.vtk

$ python postproc.py output-tests/test_navier_stokes.vtk
$ python postproc.py output-tests/test_navier_stokes.vtk --3d

- create animation (forces offscreen rendering) from
output-tests/test_time_poisson.*.vtk

$ python postproc.py output-tests/test_time_poisson.*.vtk -a mov
```

```

- create animation (forces offscreen rendering) from
  output-tests/test_hyperelastic.*.vtk

The range specification for the displacements 'u' is required, as
output-tests/test_hyperelastic.00.vtk contains only zero
displacements which leads to invisible glyph size.

$ python postproc.py output-tests/test_hyperelastic.*.vtk
↪ --ranges=u,0,0.02 -a mov

- same as above, but slower frame rate

$ python postproc.py output-tests/test_hyperelastic.*.vtk
↪ --ranges=u,0,0.02 -a mov --ffmpeg-options="-r 2 -sameq"

```

Options:

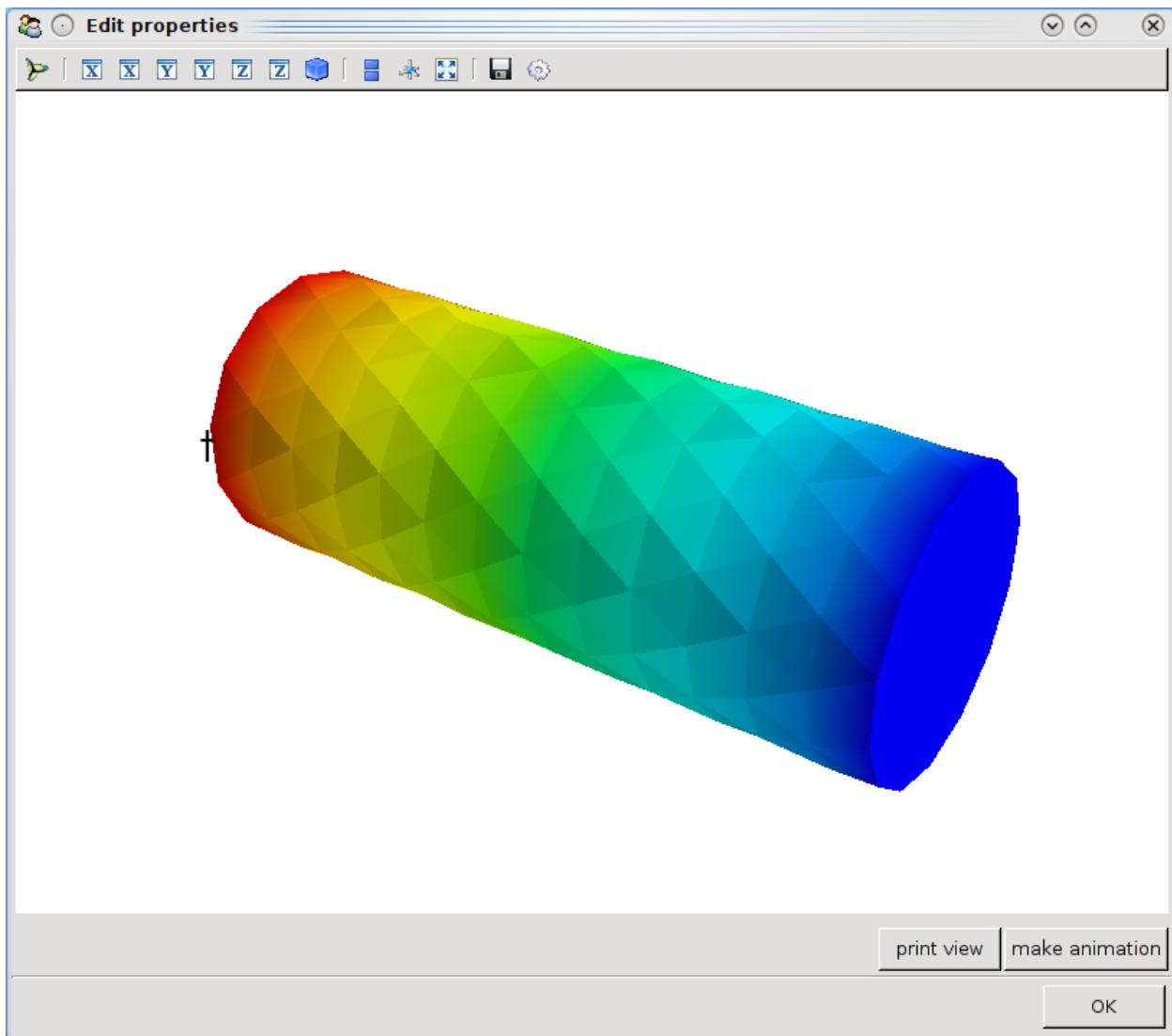
--version	show program's version number and exit
-h, --help	show this help message and exit
-l, --list-ranges	do not plot, only list names and ranges of all data
-n, --no-show	do not call mlab.show()
--no-offscreen	force no offscreen rendering for --no-show
--3d	3d plot mode
--view=angle,angle[,distance[,focal_point]]	camera azimuth, elevation angles, and optionally also distance and focal point coordinates (without []) as in `mlab.view()` [default: if --3d is True: "45,45", else: "0,0"]
--roll=angle	camera roll angle [default: 0.0]
--fgcolor=R,G,B	foreground color, that is the color of all text annotation labels (axes, orientation axes, scalar bar labels) [default: 0.0,0.0,0.0]
--bgcolor=R,G,B	background color [default: 1.0,1.0,1.0]
--layout=layout	layout for multi-field plots, one of: rowcol, colrow, row, col [default: rowcol]
--scalar-mode=mode	mode for plotting scalars with --3d, one of: cut_plane, iso_surface, both [default: iso_surface]
--vector-mode=mode	mode for plotting vectors, one of: arrows, norm, arrows_norm, warp_norm [default: arrows_norm]
-s scale, --scale-glyphs=scale	relative scaling of glyphs (vector field visualization) [default: 0.05]
--clamping	glyph clamping mode
--ranges=name1,min1,max1:name2,min2,max2:...	force data ranges [default: automatic from data]
-b, --scalar-bar	show scalar bar for each data
--wireframe	show wireframe of mesh surface for each data
--opacity=opacity	global surface and wireframe opacity in [0.0, 1.0] [default: 1.0]
--rel-text-width=width	relative text annotation width [default: 0.02]
-w, --watch	watch the results file for changes (single file mode only)
-o filename, --output=filename	view image file name [default: 'view.png']
--output-dir=directory	output directory for saving view images; ignored when

```
-o option is given, as the directory part of the
filename is taken instead [default: '.']
-a <ffmpeg-supported format>, --animation=<ffmpeg-supported format>
    if set to a ffmpeg-supported format (e.g. mov, avi,
    mpg), ffmpeg is installed and results of multiple time
    steps are given, an animation is created in the same
    directory as the view images
--ffmpeg-options="<>ffmpeg options"
    ffmpeg animation encoding options (enclose in "")
    [default: -r 10 -sameq]
-r resolution, --resolution=resolution
    image resolution in NxN format [default: shorter axis:
    600; depends on layout: for rowcol it is 800x600]
--all
    draw all data (normally, node_groups and mat_id are
    omitted)
--only-names=list of names
    draw only named data
--group-names=name1,...,nameN:...
    superimpose plots of data in each group
--subdomains=mat_id_name,threshold_limits,single_color
    superimpose surfaces of subdomains over each data;
    example value: mat_id,0,None,True
--step=step
    set the time step [default: 0]
--anti-aliasing=value
    value of anti-aliasing [default: mayavi2 default]
-d 'var_name0,function_name0,par0=val0,par1=val1,...:var_name1,...', --domain-
→specific='var_name0,function_name0,par0=val0,par1=val1,...:var_name1,...'
    domain specific drawing functions and configurations
```

As a simple example, try:

```
$ ./simple.py examples/diffusion/poisson_short_syntax.py
$ ./postproc.py cylinder.vtk
```

The following window should display:



The `-l` switch lists information contained in a results file, e.g.:

```
$ ./postproc.py -l cylinder.vtk
sfePy: 0: cylinder.vtk
point scalars
  "node_groups" (354,) range: 0 0 12_norm_range: 0.0 0.0
    "t" (354,) range: -2.0 2.0 12_norm_range: 0.0106091 2.0
  cell scalars
    "mat_id" (1348,) range: 6 6 12_norm_range: 6.0 6.0
```

## 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 `examples/` directory of SfePy.

### 4.3.1 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.

### 4.3.2 FE Mesh

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

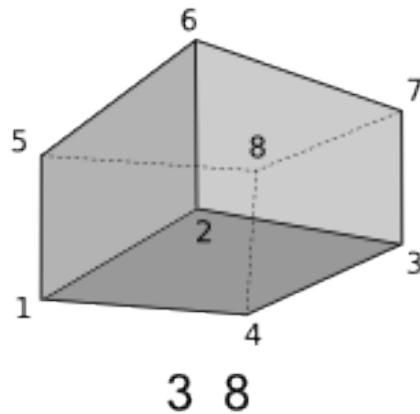
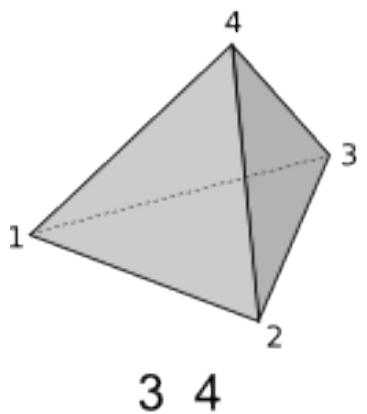
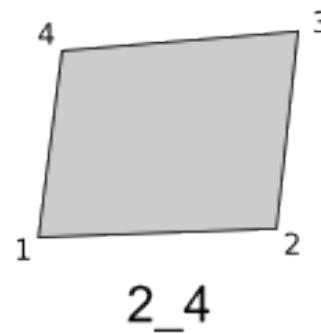
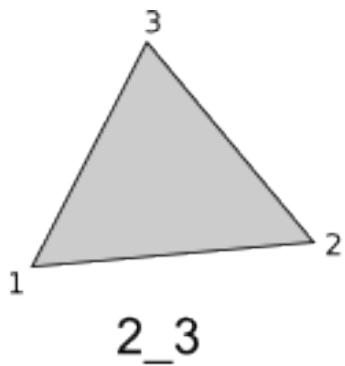
- legacy VTK (`.vtk`)
- custom HDF5 file (`.h5`)
- medit mesh file (`.mesh`)
- tetgen mesh files (`.node`, `.ele`)
- comsol text mesh file (`.txt`)
- abaqus text mesh file (`.inp`)
- avs-ucd text mesh file (`.inp`)
- hypermesh text mesh file (`.hmascii`)
- hermes3d mesh file (`.mesh3d`)
- nastran text 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:



### 4.3.3 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 [Log2012]):

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

## topological entity selection footnotes

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

<sup>2</sup> Only if mesh format supports reading boundary condition vertices as vertex sets.

<sup>3</sup> `<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.

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

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

## 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'),
}
```

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'}),
}
```

## 4.3.4 Fields

Fields correspond to FE spaces:

```
fields = {
    <name> : (<data_type>, <shape>, <region_name>, <approx_order>)
}
```

### 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 ‘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)

**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’.

### 4.3.5 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'),
}
```

### 4.3.6 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),
}
```

### 4.3.7 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}),
}
```

#### 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.

Only the 'shifted\_periodic' LCBC needs the second region and the DOF mapping function, see below.

Linear combination boundary conditions:

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

### 4.3.8 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}),
}
```

### 4.3.9 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' : {
        'Ym': stiffness_from_youngpoisson(dim, 7.0e9, 0.4),
        'Yc': stiffness_from_youngpoisson(dim, 70.0e9, 0.2) }
    } ,),
}
```

### 4.3.10 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""",
}
```

### 4.3.11 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](#).

### 4.3.12 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.

#### Defining Material Parameters

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  $(n\_coor, n\_row, n\_col)$ , where  $n\_coor$  is the number of quadrature points given by the `coors` argument,  $n\_coor = coors.shape[0]$ , and  $(n\_row, n\_col)$  is the shape of a material parameter in each quadrature point. For example, for scalar parameters, the shape is  $(n\_coor, 1, 1)$ .

#### Examples

See `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

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:

```
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 user-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!')),  
}  
  
# 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:

```

variable_1 = {
    'name' : 'p',
    'kind' : 'parameter field',
    'field' : 'temperature',
    'like' : None,
    'special' : {'setter' : 'get_load_variable'},
}

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

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

```

### 4.3.13 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' : 'h5',

    # 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>',

# 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,
}

```

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

## 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  $t \in [0, t_{\text{final}}]$ :

$$\frac{\partial T}{\partial t} + c\Delta T = 0 \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma. \quad (4.1)$$

The weak formulation of (4.1) is: Find  $T \in V$ , such that

$$\int_{\Omega} s \frac{\partial T}{\partial t} + \int_{\Omega} c \nabla T : \nabla s = 0, \quad \forall s \in V_0, \quad (4.2)$$

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

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

which directly corresponds to the discrete version of (4.2): Find  $\mathbf{T} \in V_h$ , 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  $u \approx \phi u$ ,  $\nabla u \approx G u$  for  $u \in \{s, T\}$ . The integrals over the discrete domain  $\Omega_h$  are approximated by a numerical quadrature, that is named `i` in our case.

#### 4.4.1 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 an 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.

### 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.

## 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, state, 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.
    state : State 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.
    """
    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`.

- 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_volume_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('d_volume.2.Omega(u)')
```

## 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` 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.

## 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)
```

## 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.

### 4.9.1 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.generalized_alpha`: Solve elastodynamics problems by the generalized  $\alpha$  method.
- `ts.newmark`: Solve elastodynamics problems by the Newmark method.
- `ts.simple`: Implicit time stepping solver with a fixed time step.
- `ts.stationary`: Solver for stationary problems without time stepping.
- `ts.velocity_verlet`: Solve elastodynamics problems by the velocity-Verlet method.

See [`sfePy.solvers.ts\_solvers`](#) for available *time-stepping* solvers and their options.

### 4.9.2 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_broyden_like`: Interface to Broyden and Anderson solvers from `scipy.optimize`.
- `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.

### 4.9.3 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.cm_pb`: Conjugate multiple problems.
- `ls.mumps`: Interface to MUMPS solver.
- `ls.mumps_par`: Interface to MUMPS parallel solver.
- `ls.petsc`: PETSc Krylov subspace solver.
- `ls.pyamg`: Interface to PyAMG solvers.
- `ls.pyamg_krylov`: Interface to PyAMG Krylov solvers.
- `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.

### 4.9.4 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.

## 4.9.5 Eigenvalue Problem Solvers

The following eigenvalue problem solvers are available:

- `eig.pysparse`: Pysparse-based eigenvalue solver for sparse symmetric problems.
- `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.

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

## 4.9.6 Quadratic Eigenvalue Problem Solvers

The following quadratic eigenvalue problem solvers are available:

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

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

## 4.9.7 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 `sfeypy.solvers.optimize` for available *optimization* solvers and their options.

## 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 `sfeypy.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 `sfeypy.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.

### 4.10.1 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.

## 4.10.2 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.

## 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 in<sup>1</sup>. 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.

### 4.11.1 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 volume terms work
- term integration over the whole domain as well as a volume subdomain
- simple linearization (output file generation) based on sampling the results with uniform parametric vectors
- basic domain generation with `script/gen_iga_patch.py` based on `igakit`

The following is not implemented yet:

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

---

<sup>1</sup> 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  $\mathbb{I}$  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.

### 4.11.2 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.

## EXAMPLES

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

### 5.1 Primer

#### Table of Contents

- *Introduction*
  - *Problem statement*
- *Meshing*
- *Problem description*
- *Running SfePy*
- *Post-processing*
  - *Running SfePy in interactive mode*
  - *Generating output at element nodes*
- *Probing*
- *Interactive Example*

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

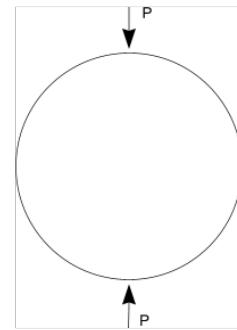
#### 5.1.1 Introduction

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:

$$\sigma_t = \frac{2P}{\pi t D} \quad (5.1)$$

$$\sigma_c = \frac{6P}{\pi t D} \quad (5.2)$$

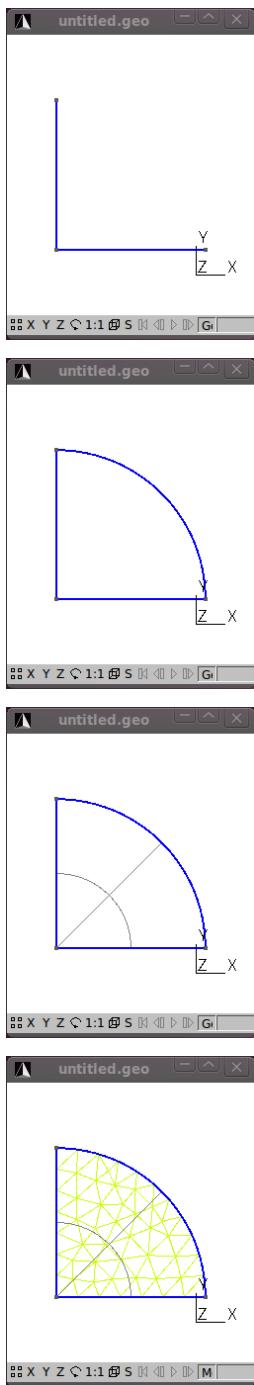
These solutions may be approximated using finite element methods. To solve this problem using *SfePy* the first step is meshing a suitable model.

## 5.1.2 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:

1. The ITS specimen has a diameter of 150 mm. Using *Gmsh* add three new points (geometry elementary entities) at the following coordinates: (0.00,0), (75.0,0.0) and (0.0,75.0).
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* add a ruled 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.

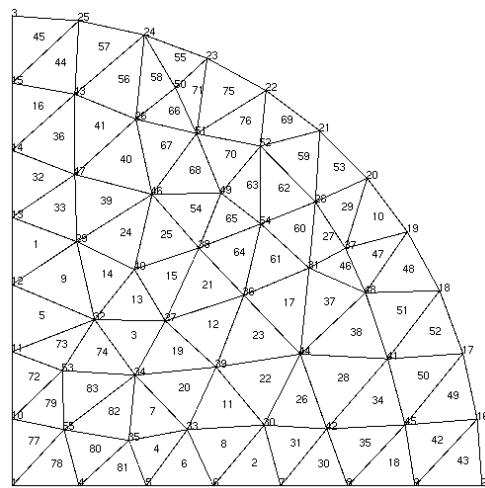


That's the meshing done. Save the mesh in a format that *SfePy* recognizes. For now use the *medit .mesh* format e.g. *its2D.mesh*.

**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.

### 5.1.3 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:`\mathbf{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \mathbf{e}_{ij}(\mathbf{u}) \mathbf{e}_{kl}(\mathbf{u})
    = 0
    ;, \quad \forall \mathbf{u}
    ;,
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 = 2000.0 # 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,
        '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'),  
    '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.O.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 con-

version. 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.

### 5.1.4 Running SfePy

One option to solve the problem is to run the `SfePy simple.py` script from the command line:

```
./simple.py its2D_1.py
```

**Note:** For the purpose of this tutorial it is assumed that the *problem description file* (`its2D_1.py`) is in the same directory as the `simple.py` script. 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`.

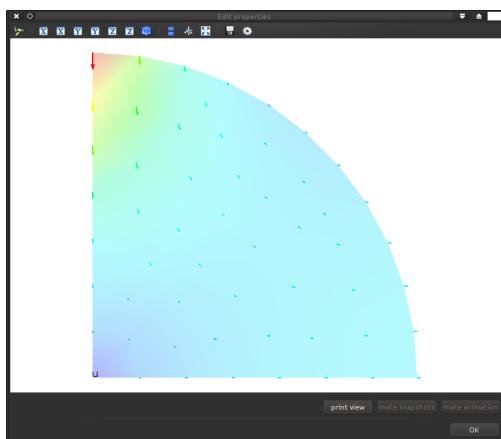
The VTK format is an ASCII format. Open the file using a text editor. You’ll notice that the output file includes separate sections:

- 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 (`postproc.py`) to quickly view the solution. To run this script you need to have `Mayavi` installed. From the command line issue the following (assuming the correct paths):

```
./postproc.py its2D.vtk
```

The `postproc.py` script 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.



## 5.1.5 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:`\nabla u` such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(u) \epsilon_{kl}(u)
    = 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 absolute_import
from 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

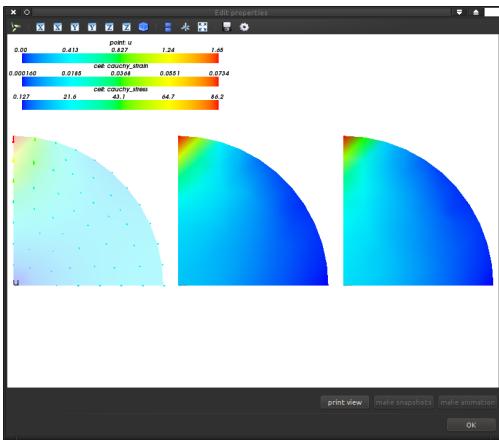
asphalt = materials['Asphalt'][0]
asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
options.update({'post_process_hook' : 'stress_strain',})
```

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):

```
./simple.py its2D_2.py
./postproc.py its2D.vtk -b
```

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 *simple.py* script 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, state = solve_pde('its2D_2.py')
```

The problem is solved and the problem definition and solution are provided in the *pb* and *state* 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 = state()
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],
       ...,
       [ 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 = state.get_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 `state()` and `state.get_parts()` return a view of the DOF vector, that is why in Out[8] the vector is reshaped according to Out[6].

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
```

```
(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 matrix, temporarily remove the imposed boundary conditions:

In [14]: pb.remove\_bcs()

Now we can calculate the force vector  $f$ :

In [15]: f = pb.evaluator.eval\_residual(u)

In [16]: f.shape

Out[16]: (110, )

In [17]: f

Out[17]:

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

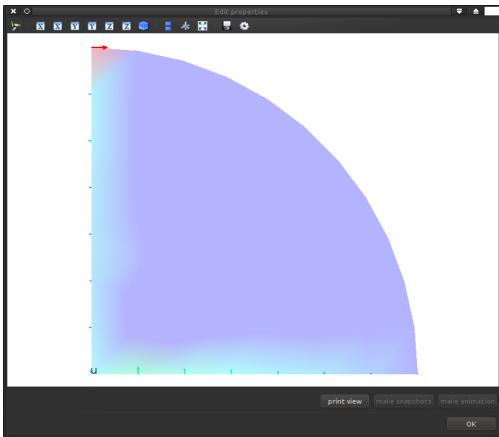
Remember to restore the original boundary conditions previously removed in step [14]:

```
In [18]: pb.time_update()
```

To view the residual force vector, we can save it to a VTK file. This requires creating a state and set its DOF vector to  $f$  as follows:

```
In [19]: state = pb.create_state()
In [20]: state.set_full(f)
In [21]: out = state.create_output_dict()
In [22]: pb.save_state('file.vtk', out=out)
```

Running the `postproc.py` script on `file.vtk` displays the average nodal forces as shown below:



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

```
In [23]: f.shape = (55, 2)
In [24]: array([ 6.20373272e+02, -1.13686838e-13])
```

Great, we have an almost zero residual vertical load or force apparent at node 2 i.e.  $-1.13686838e-13$  Newton. 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:`\mathbf{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) = 0
    \quad \forall \mathbf{u}
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 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')
    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
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 customized IPython shell:

```
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, state = solve_pde('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('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][0])
```

```

Horizontal tensile stress = 7.57220e+00 MPa/mm
In [35]: print 'Vertical compressive stress = %.5e MPa/mm' % (-svar()[0][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', None, '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.

### 5.1.6 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 :math:`\nabla u` such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\nabla u) e_{kl}(\nabla u) = 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 absolute_import
from 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]]
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_indices = [0, 4, 1]
    sym_labels = ['11', '22', '12']

    plt.subplot(312)
    pars, vals = probe(ip, 'cauchy_strain')
    for ii, ic in enumerate(sym_indices):
        plt.plot(pars, vals[:,ic], label=r'$e_{%s}$' % 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, ic in enumerate(sym_indices):
        plt.plot(pars, vals[:,ic], 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',
)

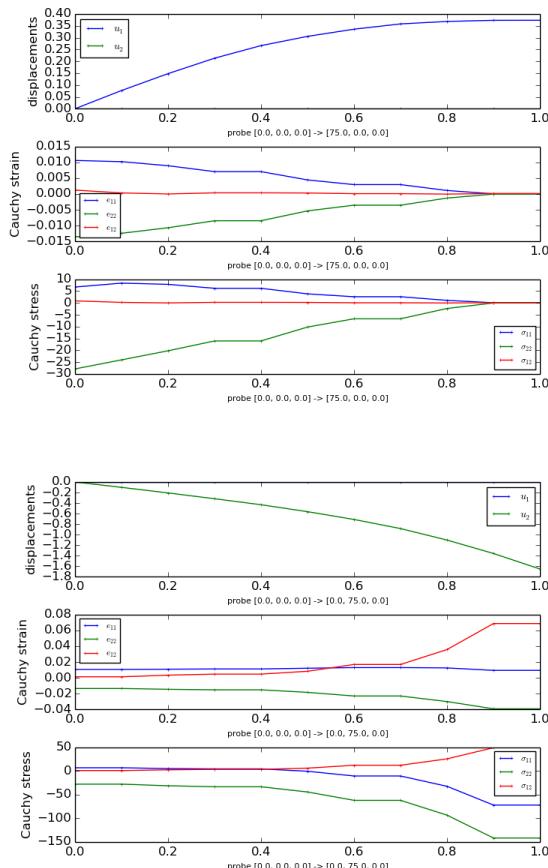
```

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.

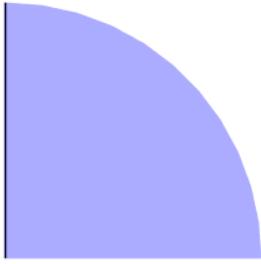
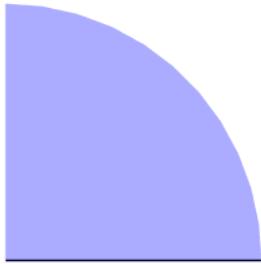
Run *SfePy* to solve the problem and apply the probes:

```
./simple.py 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.



### 5.1.7 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``, and how to display the results using Mayavi.  
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 examples/linear_elasticity/its2D_interactive.py -h
```

Notes

-----

```

The ``--probe`` and ``--show`` options work simultaneously only if Mayavi and
Matplotlib use the same backend type (for example wx).
"""

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.autoFallback import AutoDirect
from sfepy.solvers.ls import ScipyDirect, ScipySuperLU, ScipyUmfpack, MUMPSSolver
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 examples.linear_elasticity.its2D_2 import stress_strain
from 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)
    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'$e_{%s}$' % 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',
    'show' : 'show the results figure',
}

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('--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('-s', '--show',
                    action="store_true", dest='show',
                    default=False, help=helps['show'])
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')

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(ebc=Conditions([xsym, ysym]))

pb.set_solver(nls)

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

# Postprocess the solution.
out = state.create_output_dict()
out = stress_strain(out, pb, state, 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, state, 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 options.show:
    # Show the solution. If the approximation order is greater than 1, the
    # extra DOFs are simply thrown away.
    from sfepy.postprocess.viewer import Viewer

    view = Viewer('its2D_interactive.vtk')
    view(vector_mode='warp_norm', rel_scaling=1,
          is_scalar_bar=True, is_wireframe=True)

if __name__ == '__main__':
    main()

```

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

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

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

```
python 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.

## 5.2 Using Salome with SfePy

### 5.2.1 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:

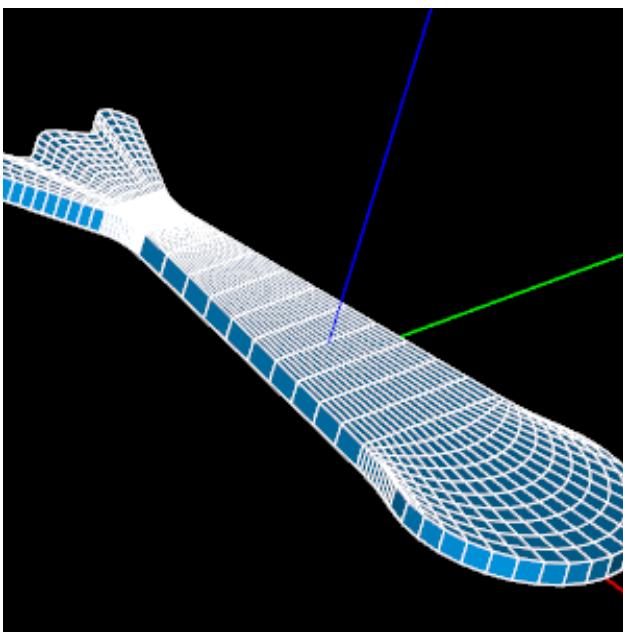
- <sfepy\_root>: the root directory of the *SfePy* source code
- <work\_dir>: the working directory where you plan to save your files

### 5.2.2 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:



### 5.2.3 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>`.

### 5.2.4 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>/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.

### 5.2.5 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

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

to

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

## 5.2.6 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:

```
./simple.py 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:

```
./postproc.py --wireframe --vector-mode=warp_norm -s 2 Mesh_Partition_Hexa.vtk
```

You should get the following plot. The undeformed mesh is displayed with a wireframe for comparison. 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*!

## 5.3 Preprocessing: FreeCAD/OpenSCAD + Gmsh

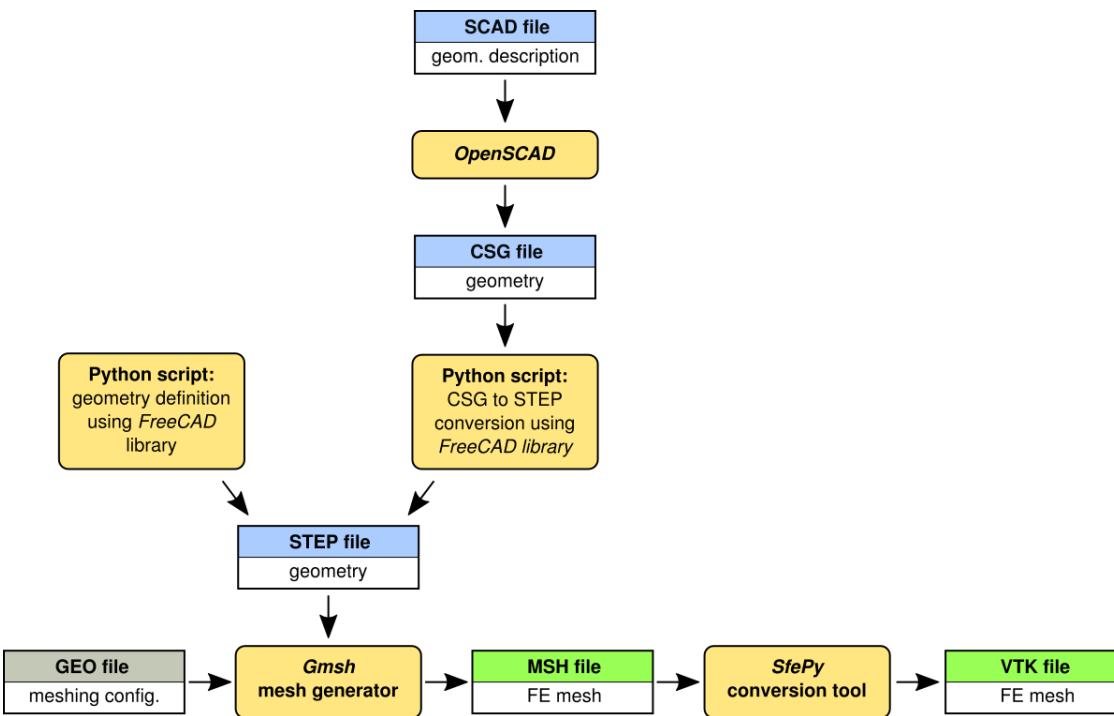
### Table of Contents

- *Introduction*
- *Creating geometry using FreeCAD*
- *Creating geometry using OpenSCAD*
- *Gmsh - generating finite element mesh*
  - *Note: planar 2D meshes*
- *Multipart models*

### 5.3.1 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.



### 5.3.2 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 FREECADCADPATH = '/usr/lib/freecad/lib/'
3 sys.path.append(FREECADCADPATH)
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")
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')

```

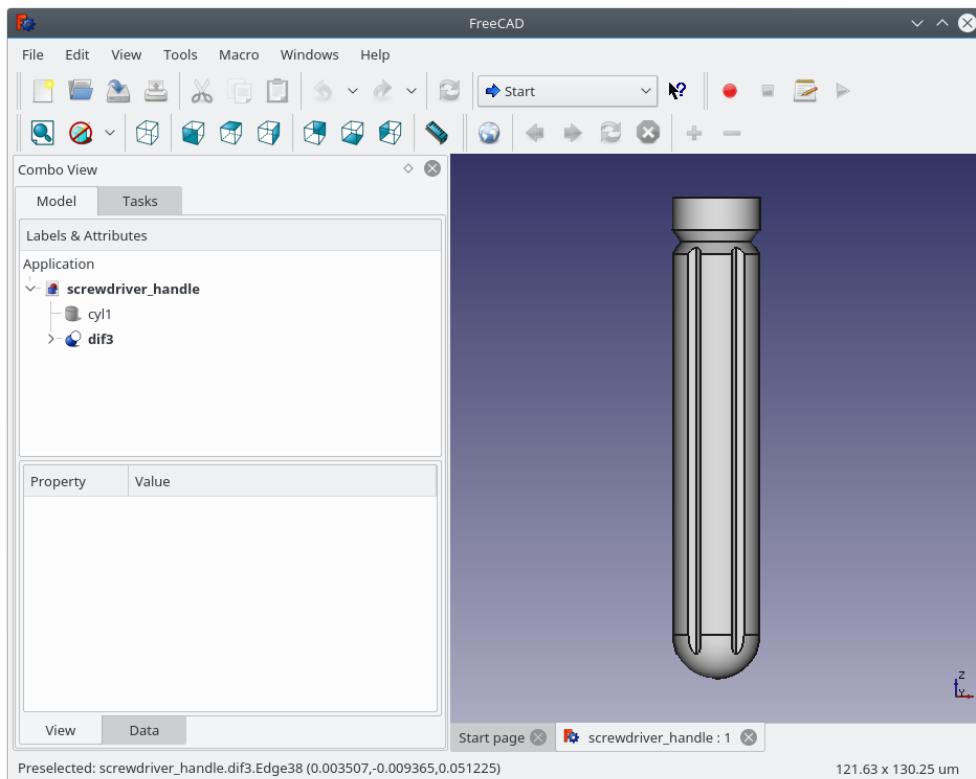
A finite element mesh can be generated directly in *FreeCAD* using *MeshPart* module:

```

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.

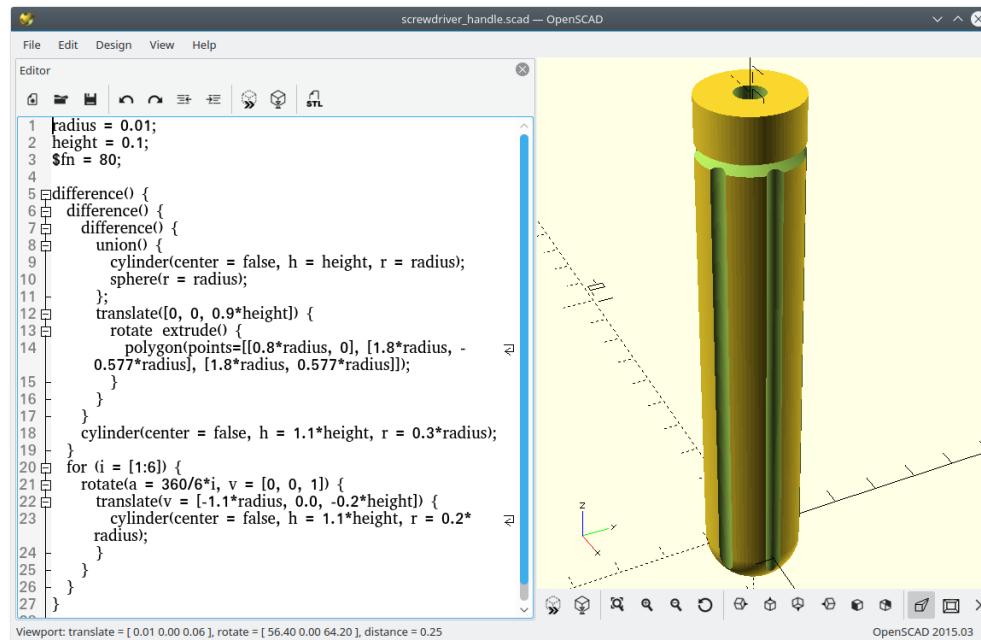
### 5.3.3 Creating geometry using *OpenSCAD*

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            };
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        }
24    }
25 }
```



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')
```

### 5.3.4 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;
```

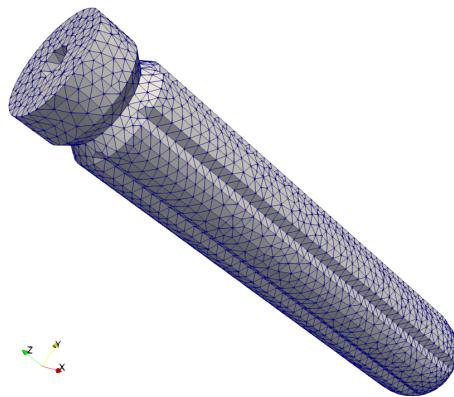
Now, run *Gmsh* generator and export the mesh into the *MSH* format in which all surface and volumetric elements are stored:

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

By converting the *MSH* file into the *VTK* format using *script/convert\_mesh.py*:

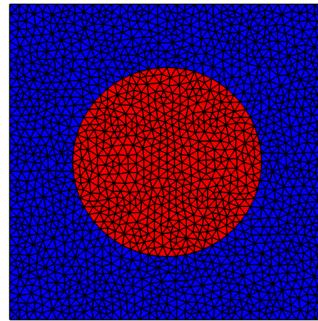
```
script/convert_mesh.py screwdriver_handle.msh screwdriver_handle.vtk
```

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:

```
script/convert_mesh.py --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
script/convert_mesh.py 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 `script/convert_mesh.py` is not needed in this case.

### 5.3.5 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:

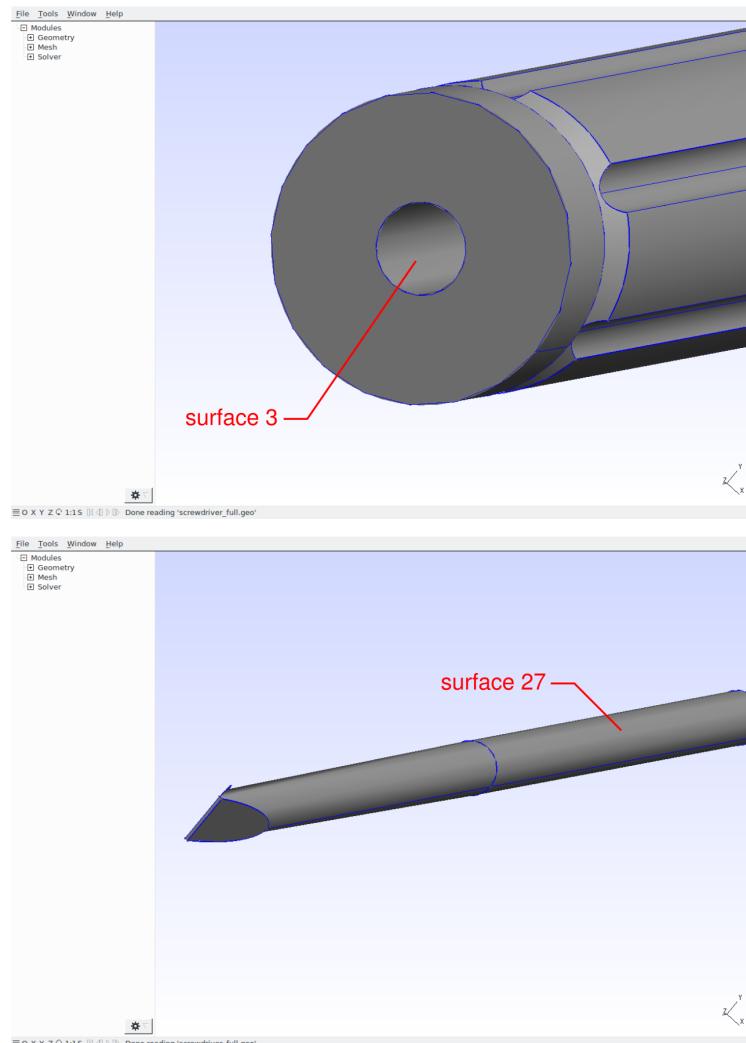
```

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 than 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};
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.

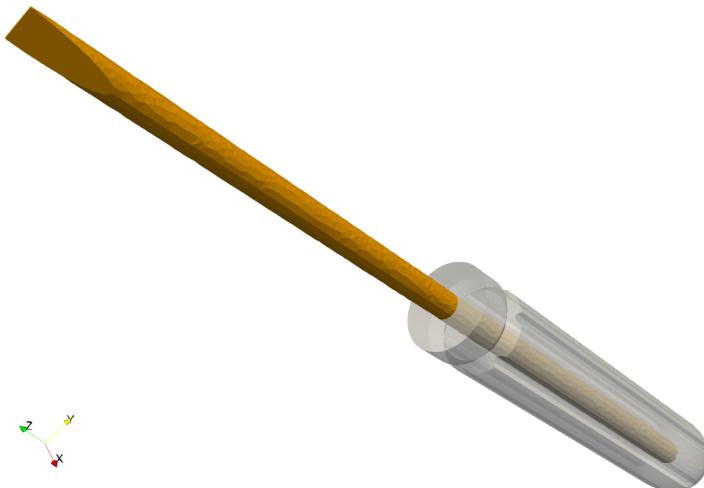
Using the above stated *GEO* file, *Gmsh* creates a mesh containing duplicate vertices on the handle/shank interface. These duplicate vertices can be removed during the conversion to the *VTK* format by giving `--merge` (or just `-m`) argument to `convert_mesh.py` script:

```
script/convert_mesh.py -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:

```
script/convert_mesh.py --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() {
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.
21 ↳577*radius]]);
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 importCSG.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.

## 5.4 Material Identification

## Table of Contents

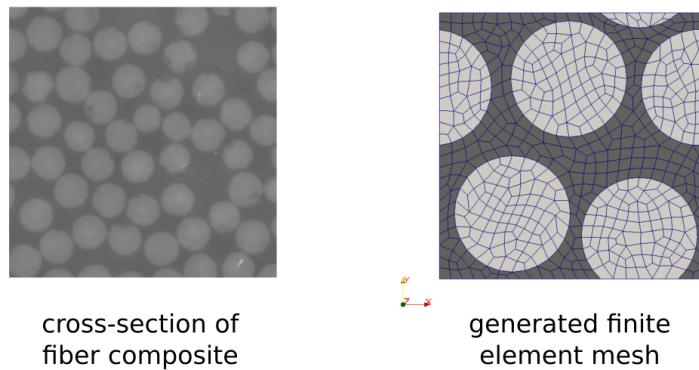
- *Introduction*
  - *Composite structure*
  - *Material testing*
- *Numerical simulation*
  - *Homogenized coefficients*
  - *Macroscopic simulation*
- *Identification procedure*
- *Running identification script*

### 5.4.1 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 in<sup>1</sup>):



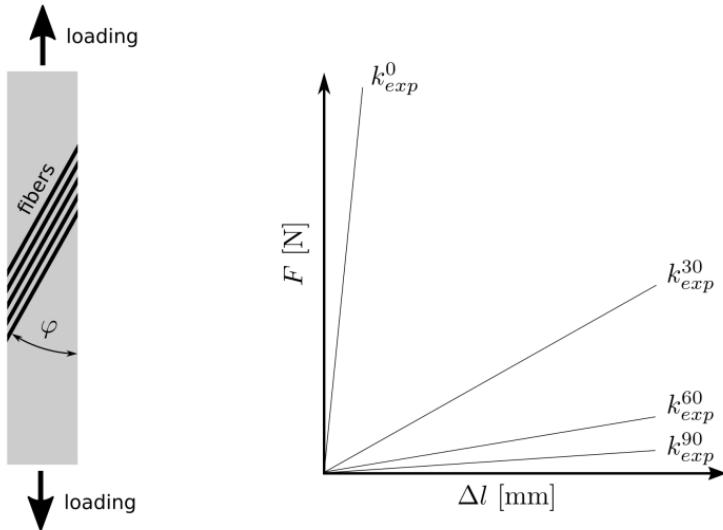
This RVE is used in the micromechanical FE analysis which is based on the two-scale homogenization method.

#### 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, see<sup>2</sup>. The slopes of the linearized dependencies were used in an objective function of the identification process.

<sup>1</sup> 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](http://arxiv.org/PS_cache/cond-mat/pdf/0503/0503627v2.pdf)

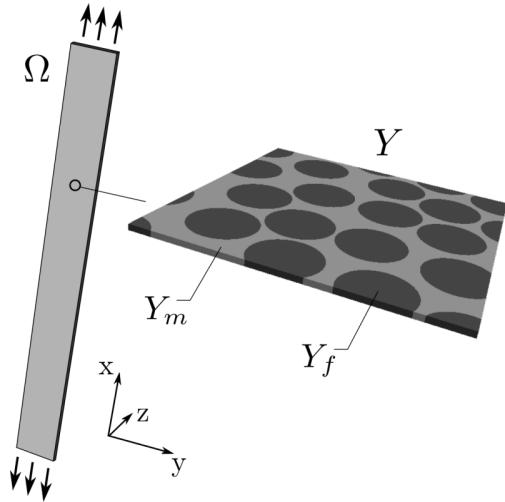
<sup>2</sup> Srbová H., Kroupa T., Zemčík R., Identification of the Material Parameters of a Unidirectional Fiber Composite Using a Micromodel, Materiali in Tehnologije, 46(5), 2012, 431-434.



## 5.4.2 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 problem<sup>3</sup>. The homogenization procedure results in the microscopic problem solved within the RVE and the macroscopic problem that involves the homogenized elastic coefficients.



### 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.

<sup>3</sup> 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.

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 modulus, 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.

#### 5.4.3 Identification procedure

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

$$\Phi(\mathbf{x}) = \sum_{i \in \{0, 30, 60, 90\}} \left( 1 - \frac{k_{comp}^i(\mathbf{x})}{k_{exp}^i} \right)^2, \quad (5.3)$$

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.

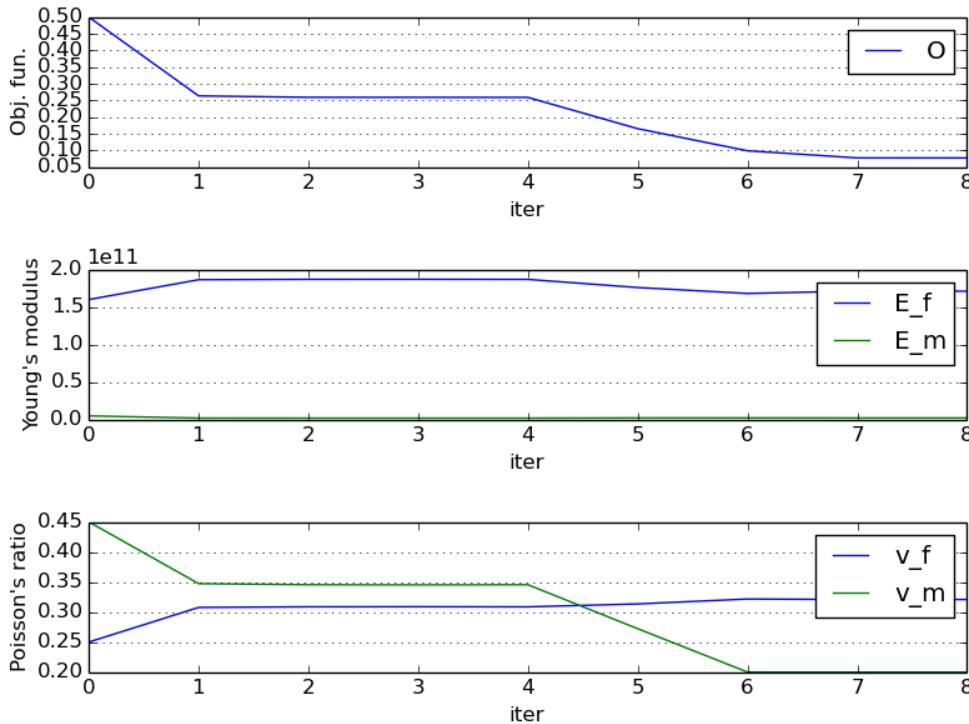
Python script for material identification: material\_opt\_src.

#### 5.4.4 Running identification script

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

```
$ python 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  $E_f$ ,  $\nu_f$ ,  $E_m$ ,  $\nu_m$  can be found at the end of the script output:

```

1 >>> material optimization FINISHED <<<
2 material_opt_micro: terminated
3 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.

## 5.5 Mesh parametrization

### Table of Contents

- [Introduction](#)
- [SplineBox](#)
  - [Example](#)
- [SplineRegion2D](#)
  - [Example](#)

## 5.5.1 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.

## 5.5.2 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:

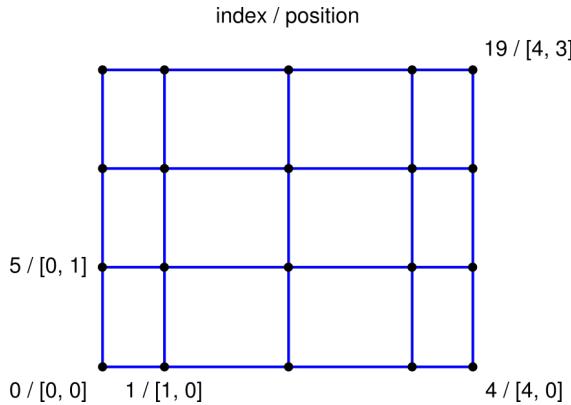
$$ncp_i = nsg + degree, \quad i = 1, 2(, 3) \quad (5.4)$$

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_tril.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()
```

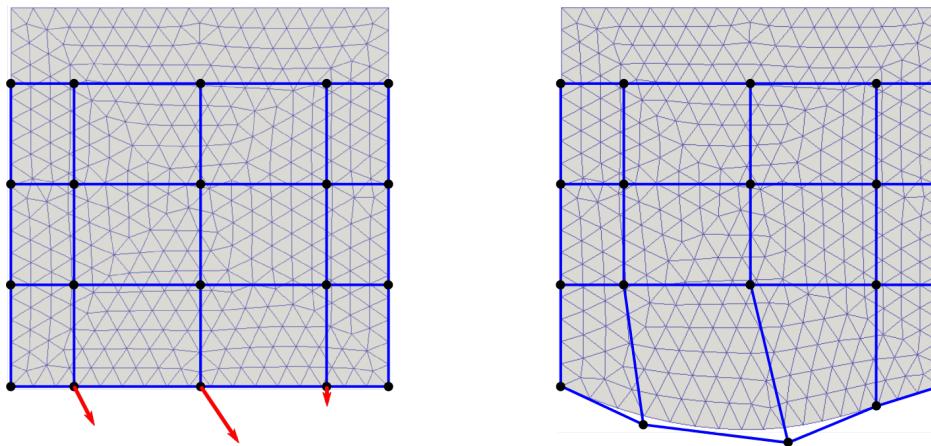
- Write the deformed mesh and the spline control net (the net of control points) into vtk files:

```

spb.write_control_net('square_tril_spbox.vtk')
mesh.write('square_tril_deform.vtk')

```

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



### 5.5.3 SplineRegion2D

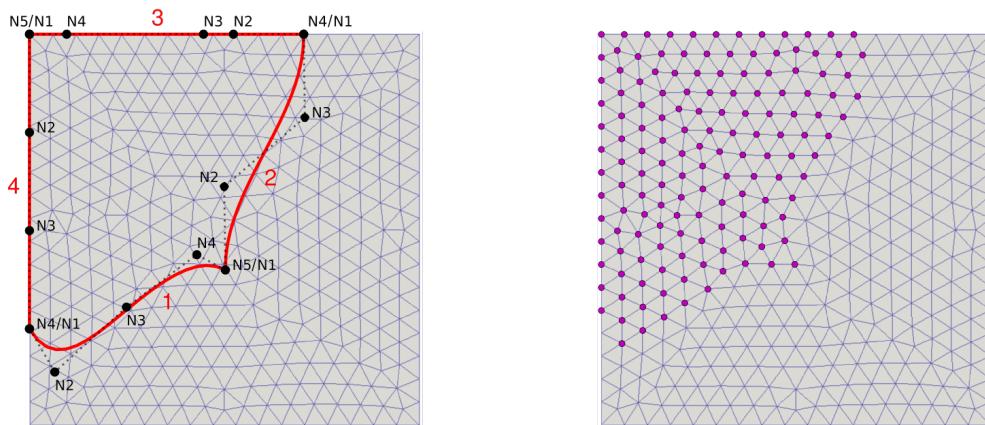
In this case, the region (only in 2D) of parametrization is defined by four B-spline curves:

```

1 from sfepy.mesh.splinebox 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:

```

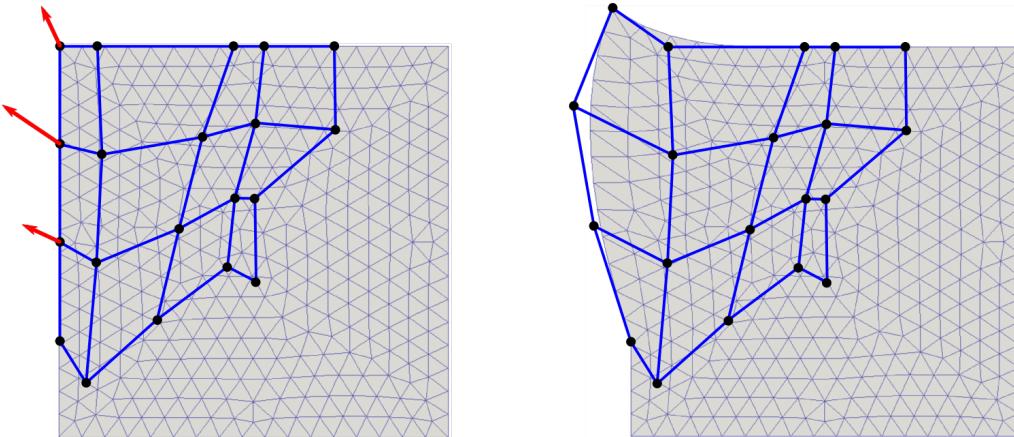
1 spb.move_control_point(5, [-.2, .1])
2 spb.move_control_point(10, [-.3, .2])
3 spb.move_control_point(15, [-.1, .2])

```

- Evaluate the new coordinates:

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

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



## 5.6 Examples

### 5.6.1 acoustics

`acoustics/acoustics.py`

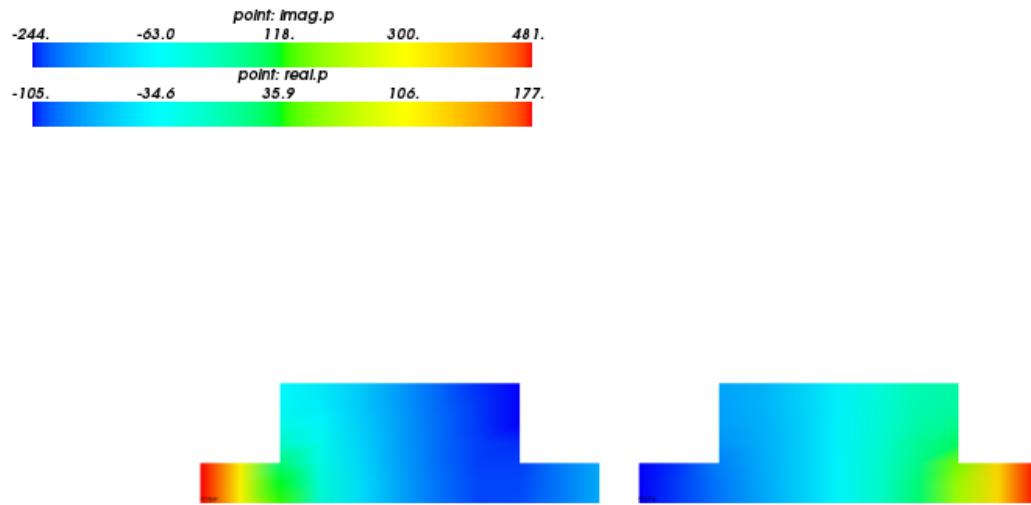
#### Description

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
'acoustic_pressure' field definition.

Find :math:`p` such that:

.. math::
    \begin{aligned}
    & 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
    \end{aligned}
"""

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
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_volume_dot.i.Omega( q, p )
        - %s * dw_surface_dot.i.Gamma_out( q, p )
        = %s * dw_surface_integrate.i.Gamma_in( q )"""
        % (c*c, w*w, 1j*w*c, 1j*w*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).
        '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

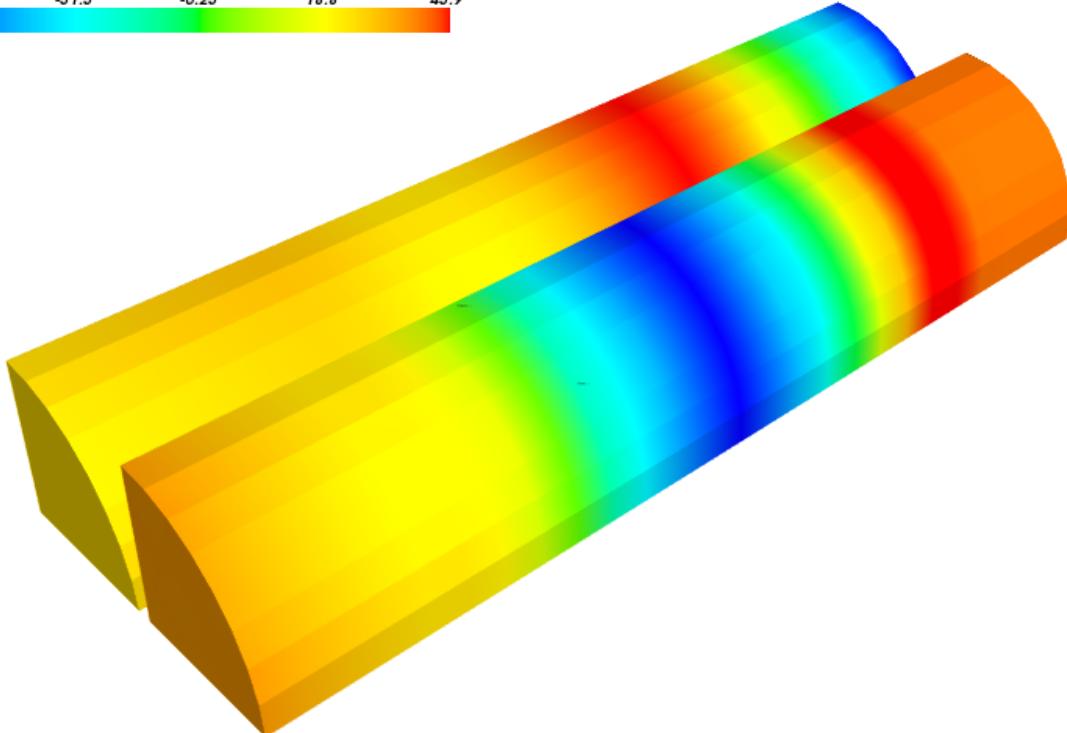
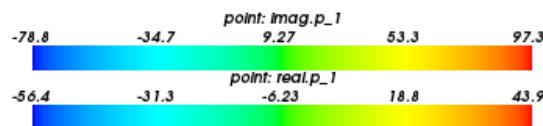
### Description

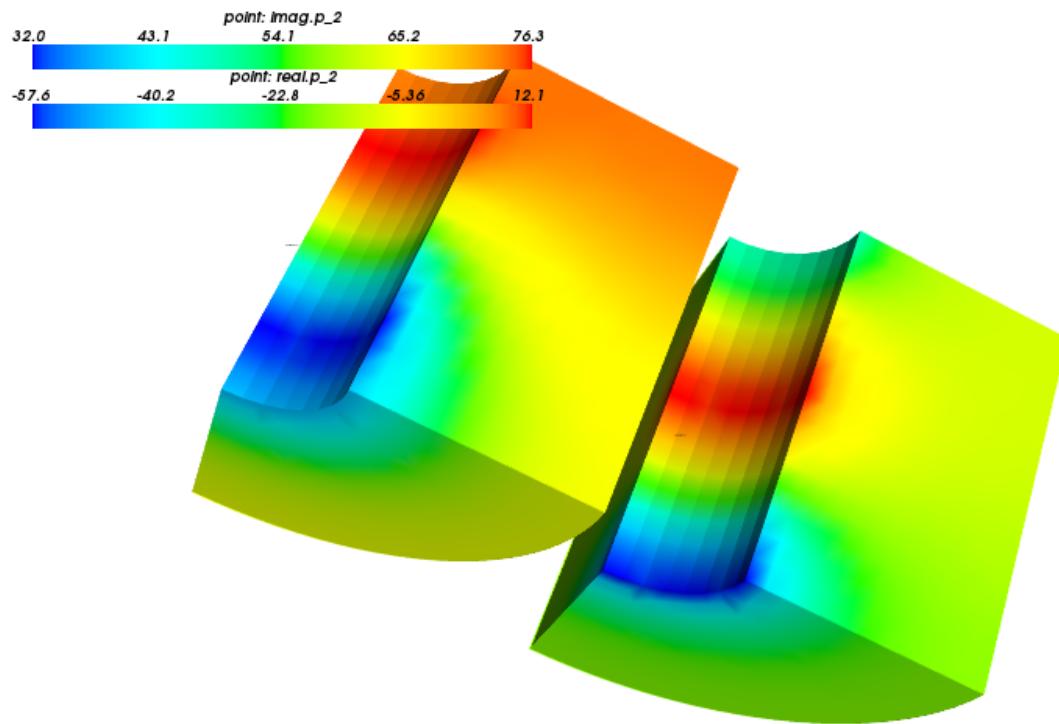
Acoustic pressure distribution in 3D.

Two Laplace equations, one in  $\Omega_1$ , 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:

$$\begin{aligned} \int_{\Omega} k^2 qp - \int_{\Omega} \nabla q \cdot \nabla p \\ -iw/c \int_{\Gamma_{out}} qp + iw\rho/Z \int_{\Gamma_2} q(p_2 - p_1) + iw\rho/Z \int_{\Gamma_1} q(p_1 - p_2) \\ = iw\rho \int_{\Gamma_{in}} v_n q , \quad \forall q . \end{aligned}$$





[source code](#)

```
r"""
Acoustic pressure distribution in 3D.

Two Laplace equations, one in :math:`\Omega_1`, other in
:math:`\Omega_2`, connected on the interface region :math:`\Gamma_{12}`
using traces of variables.

Find two complex acoustic pressures :math:`p_1`, :math:`p_2` such that:

.. math::

    \int_{\Omega} k^2 q p - \int_{\Omega} \nabla q \cdot \nabla p \\
    - 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) \\
    = i w \rho \int_{\Gamma_{in}} v_n q \\
    ;, \quad \forall q ;
"""

from __future__ import absolute_import
from sfepy import data_dir

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'),
    'q_2' : ('test field', 'acoustic_pressure_2', 'p_2'),
}

ebcs = {}

integrals = {
    'i' : 2,
}

```

```
equations = {
    'Acoustic pressure' :
    """%s * dw_volume_dot.i.Omega_1(q_1, p_1)
    + %s * dw_volume_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_surface_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_surface_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',
    'file_per_var': True,
}

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/vibro\_acoustic3d.py

### Description

Vibro-acoustic problem

3D acoustic domain with 2D perforated deforming interface.

*Master problem:* defined in 3D acoustic domain (`vibro_acoustic3d.py`)

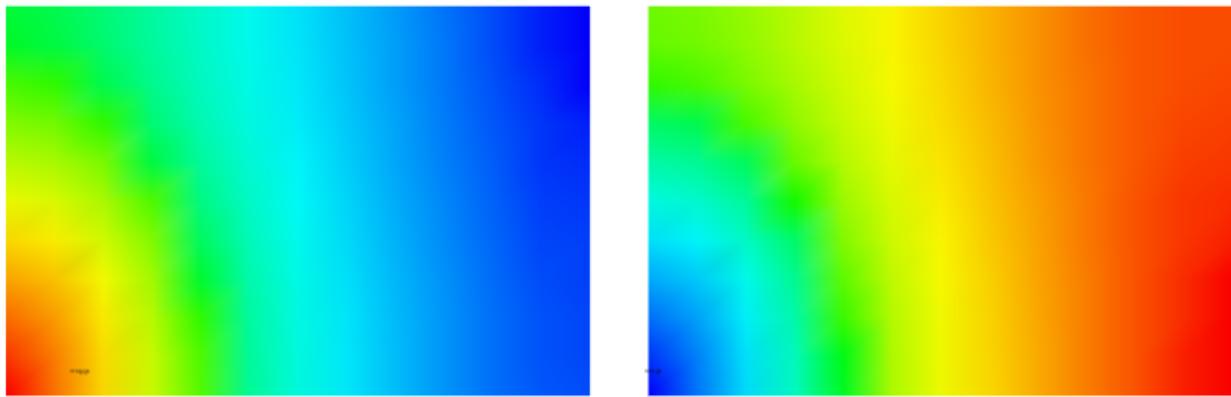
*Slave subproblem:* 2D perforated interface (`vibro_acoustic3d_mid.py`)

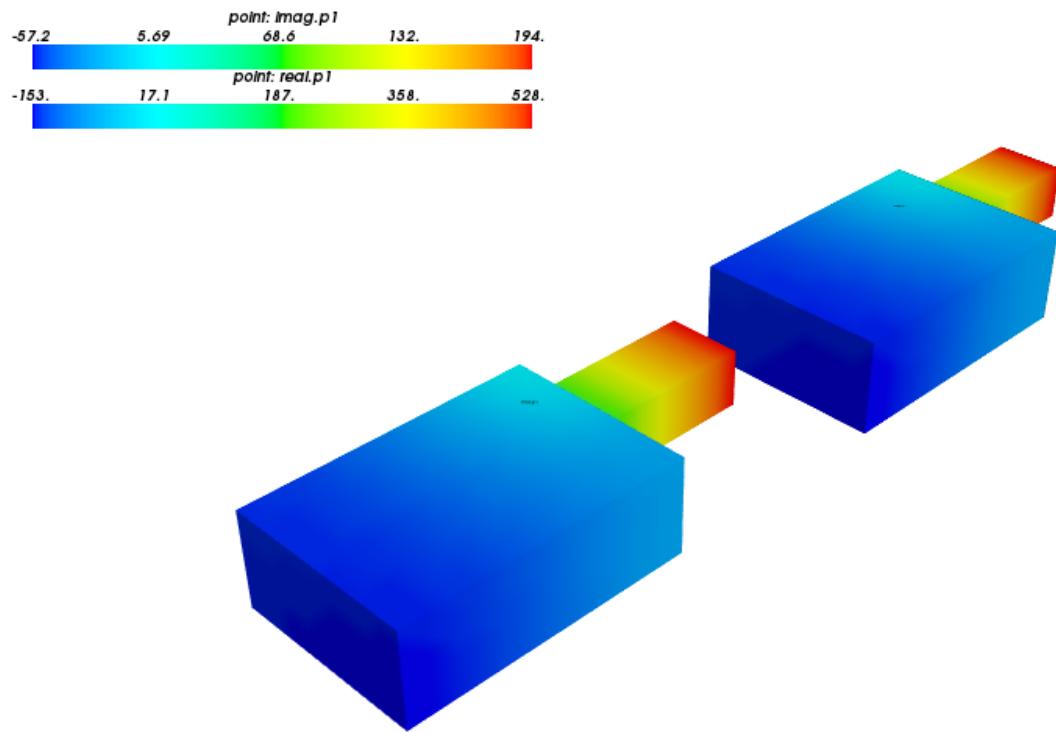
Master 3D problem - find  $p$  (acoustic pressure) and  $g$  (transversal acoustic velocity) such that:

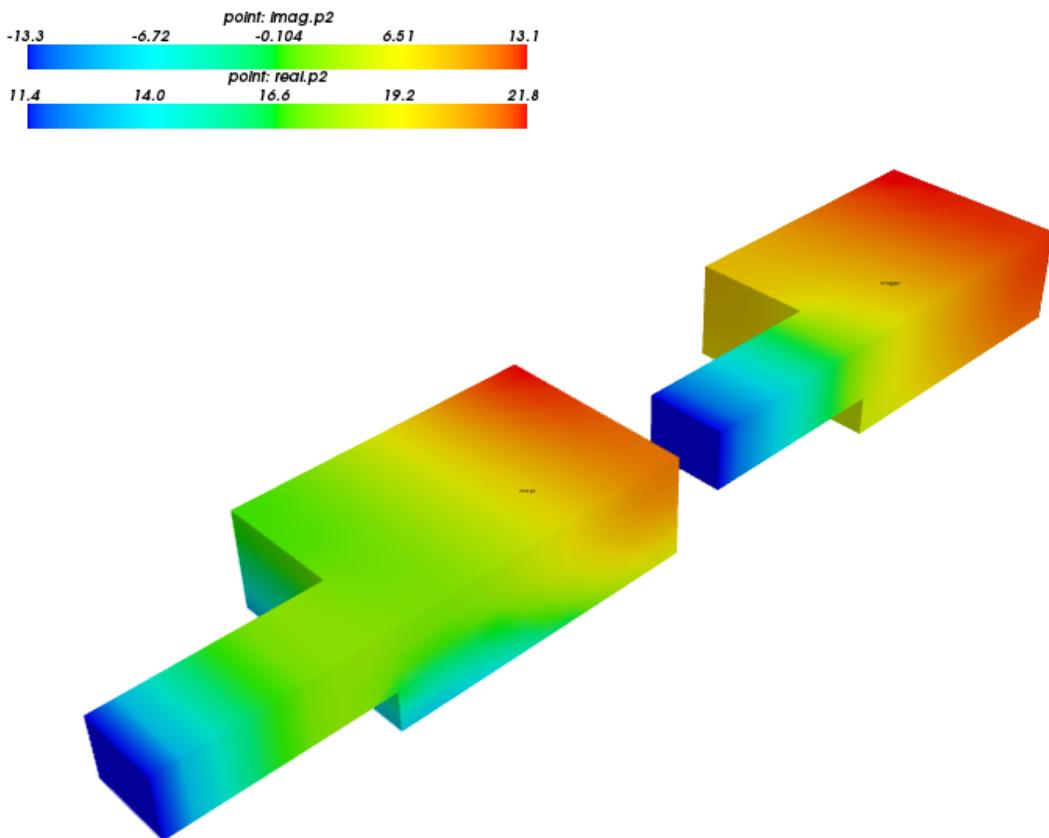
$$\begin{aligned} c^2 \int_{\Omega} \nabla q \cdot \nabla p - \omega^2 \int_{\Omega} qp + i\omega c \int_{\Gamma_{in}} qp + i\omega c \int_{\Gamma_{out}} qp - i\omega c^2 \int_{\Gamma_0} (q^+ - q^-)g &= 2i\omega c \int_{\Gamma_{in}} q\bar{p}, \quad \forall q, \\ -i\omega \int_{\Gamma_0} f(p^+ - p^-) - \omega^2 \int_{\Gamma_0} Ffg + \omega^2 \int_{\Gamma_0} Cf w &= 0, \quad \forall f, \end{aligned}$$

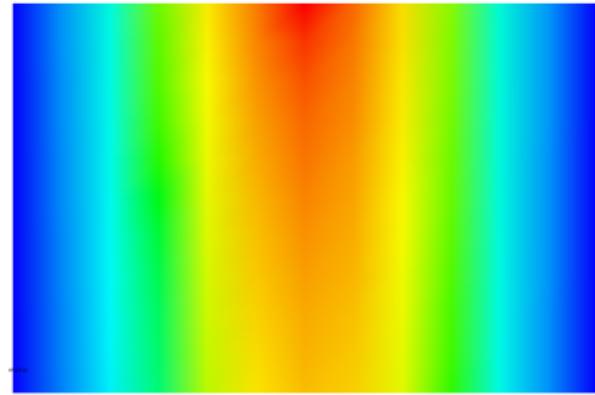
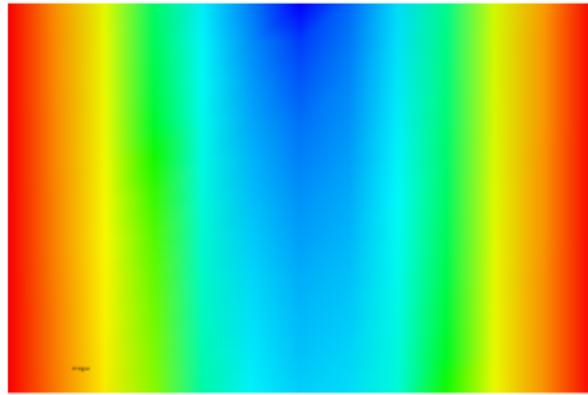
Slave 2D subproblem - find  $w$  (plate deflection) and  $\underline{\theta}$  (rotation) such that:

$$\begin{aligned} \omega^2 \int_{\Gamma_0} C z g - \omega^2 \int_{\Gamma_0} S z w + \int_{\Gamma_0} \nabla z \cdot \underline{\underline{G}} \cdot \nabla w - \int_{\Gamma_0} \underline{\theta} \cdot \underline{\underline{G}} \cdot \nabla z &= 0 , \quad \forall z , \\ -\omega^2 \int_{\Gamma_0} R \underline{\nu} \cdot \underline{\theta} + \int_{\Gamma_0} D_{ijkl} e_{ij}(\underline{\nu}) e_{kl}(\underline{\theta}) - \int_{\Gamma_0} \underline{\nu} \cdot \underline{\underline{G}} \cdot \nabla w + \int_{\Gamma_0} \underline{\nu} \cdot \underline{\underline{G}} \cdot \underline{\theta} &= 0 , \quad \forall \underline{\nu} , \end{aligned}$$









[source code](#)

```
r"""
Vibro-acoustic problem

3D acoustic domain with 2D perforated deforming interface.

*Master problem*: defined in 3D acoustic domain (`vibro_acoustic3d.py`)

*Slave subproblem*: 2D perforated interface (`vibro_acoustic3d_mid.py`)

Master 3D problem - find :math:`p` (acoustic pressure)
and :math:`g` (transversal acoustic velocity) such that:

.. math::

    \begin{aligned}
    & 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 \\
    & = 0
    \end{aligned}

```

```

\;, \quad \forall f \;,\;

Slave 2D subproblem - find :math:`w` (plate deflection)
and :math:`\ul{\theta}` (rotation) such that:

.. math::
    \begin{aligned}
    & \omega^2 \int_{\Gamma_0} C z g \\
    & - \omega^2 \int_{\Gamma_0} S z w \\
    & + \int_{\Gamma_0} \nabla z \cdot \nabla u_l G \cdot \nabla w \\
    & - \int_{\Gamma_0} \ul{\theta} \cdot \nabla u_l G \cdot \nabla z \\
    & = 0 \\
    \;, \quad \forall z \;,\;
    \\[1em]
    & - \omega^2 \int_{\Gamma_0} R, \ul{\nu} \cdot \nabla \ul{\theta} \\
    & + \int_{\Gamma_0} D_{ijkl} e_{ij}(\ul{\nu}) e_{kl}(\ul{\theta}) \\
    & - \int_{\Gamma_0} \ul{\nu} \cdot \nabla u_l G \cdot \nabla w \\
    & + \int_{\Gamma_0} \ul{\nu} \cdot \nabla u_l G \cdot \nabla \ul{\theta} \\
    & = 0 \\
    \;, \quad \forall \ul{\nu} \;,\;
    \end{aligned}
"""

from __future__ import absolute_import
from sfepy import data_dir
filename_mesh = data_dir + '/meshes/3d/acoustic_wg.vtk'

sound_speed = 343.0
wave_num = 5.5
p_inc = 300

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'),
    'aux_Left': ('vertices in (x < 0.001)', 'face'),
    'aux_Right': ('vertices in (x > 0.299)', 'face'),
    'Gamma0_1_Left': ('r.Gamma0_1 *v r.aux_Left', 'edge'),
    'Gamma0_1_Right': ('r.Gamma0_1 *v r.aux_Right', 'edge'),
}

fields = {
    'pressure1': ('complex', 'scalar', 'Omega1', 1),
    'pressure2': ('complex', 'scalar', 'Omega2', 1),
    'tvelocity': ('complex', 'scalar', 'Gamma0_1', 1),
    'deflection': ('complex', 'scalar', 'Gamma0_1', 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'),
}

ebcs = {
    'fixed_l': ('Gamma0_1_Left', {'w.0': 0.0}),
    'fixed_r': ('Gamma0_1_Right', {'w.0': 0.0}),
}

options = {
    'file_per_var': True,
}

functions = {}

materials = {
    'ac' : ({'F': -2.064e+00, 'c': -1.064e+00}, ),
}

equations = {
    'eq_1' : """
        %e * dw_laplace.5.Omega1(q1, p1)
        + %e * dw_laplace.5.Omega2(q2, p2)
        - %e * dw_volume_dot.5.Omega1(q1, p1)
        - %e * dw_volume_dot.5.Omega2(q2, p2)
        + %s * dw_surface_dot.5.GammaIn(q1, p1)
        + %s * dw_surface_dot.5.GammaOut(q2, p2)
        - %s * dw_surface_dot.5.Gamma0_1(q1, g0)
        + %s * dw_surface_dot.5.Gamma0_2(q2, tr(g0))
        = %s * dw_surface_integrate.5.GammaIn(q1)"""
        % (c2, c2, w2, w2,
           1j * wc, 1j * wc,
           1j * wc2, 1j * wc2,
           2j * wc * p_inc),
    'eq_2' : """
        - %s * dw_surface_dot.5.Gamma0_1(f0, p1)
        + %s * dw_surface_dot.5.Gamma0_1(f0, tr(p2))
        - %e * dw_surface_dot.5.Gamma0_1(ac.F, f0, g0)
        + %e * dw_surface_dot.5.Gamma0_1(ac.c, f0, w)
        = 0"""
        % (1j * w, 1j * w, w2, w2),
}

solvers = {
    'ls': ('ls.cm_pb',
           {'others': [data_dir
                      + '/examples/acoustics/vibro_acoustic3d_mid.py'],
            'coupling_variables': ['g0', 'w'],
            }),
    'nls': ('nls.newton',
            {'i_max' : 1,
             'eps_a' : 1e-6,
             })
}
```

```

        'eps_r' : 1e-6,
    }
}
```

## 5.6.2 diffusion

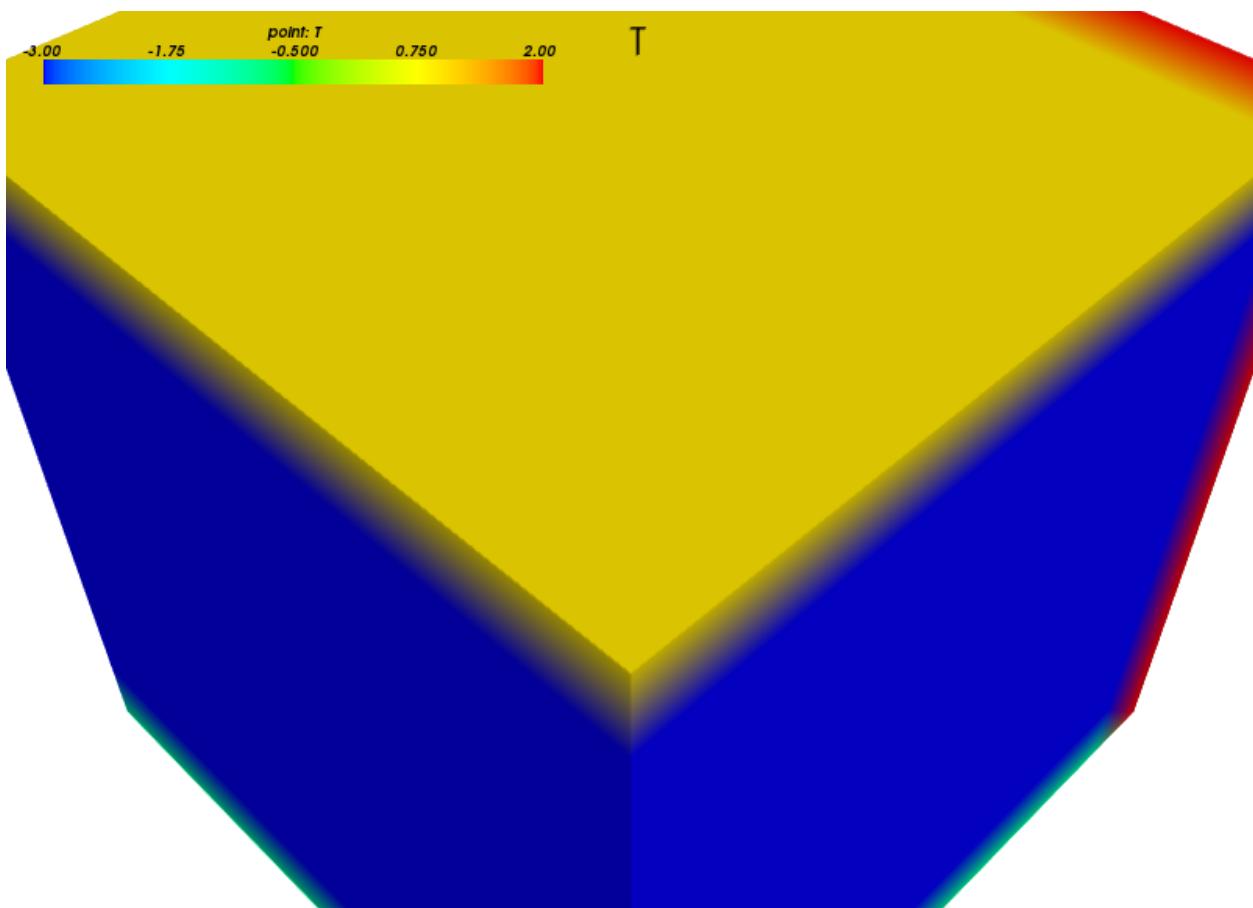
### diffusion/cube.py

#### 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 = {
    '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
}

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""",
}

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

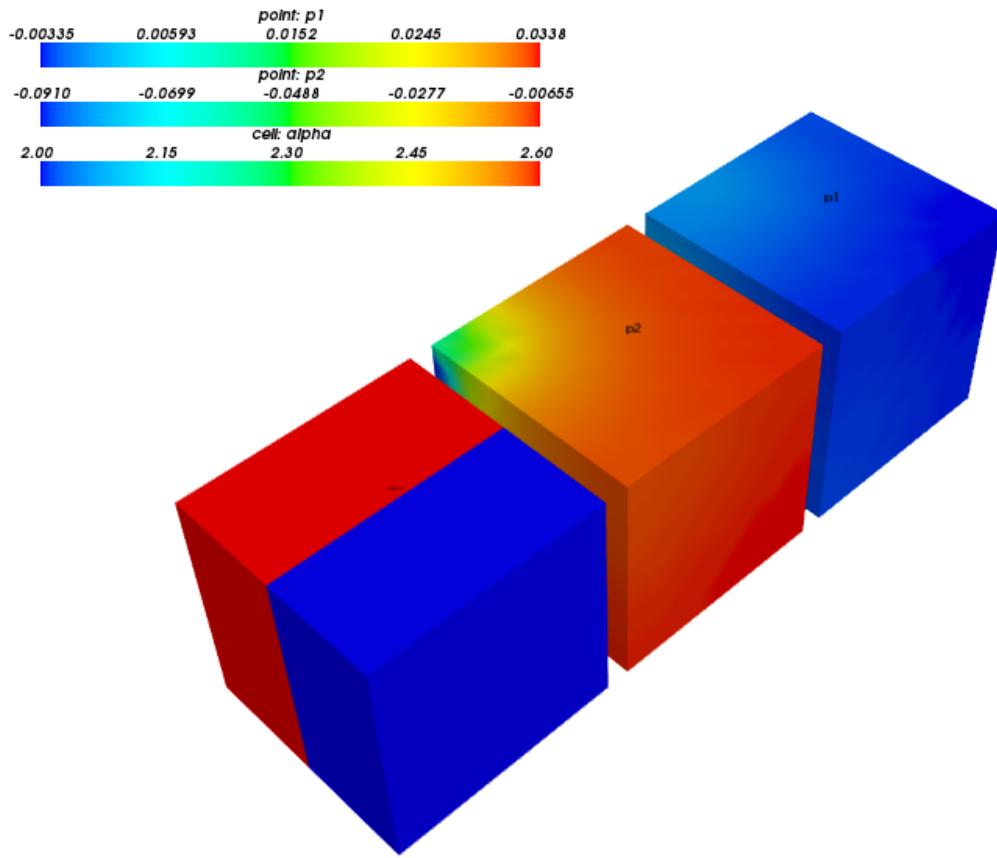
### Description

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  $\Omega_i$ .

$$\int_{\Omega_i} K^i \nabla p^i \cdot \nabla q^i + \int_{\Omega_i} \sum_j \bar{G} \alpha_k (p^i - p^j) q^i = \int_{\Omega_i} f^i q^i,$$

$$\forall q^i \in Q^i, \quad i, j = 1, 2 \quad \text{and} \quad i \neq j,$$

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 :math:`\Omega_i`.
```

```
.. math::
    \int_{\Omega_i} K^i \nabla p^i \cdot \nabla q^i + \int_{\Omega_i} \sum_j \bar{G} \alpha_{kj} (p^i - p^j) q^i
    = \int_{\Omega_i} f^i q^i,
.. math::
    \forall q^i \in Q^i, \quad i, j=1,2 \quad \text{and} \quad i \neq j,
```

where :math:`K^i` is the local permeability of the  $i$ -th compartment, :math:`\bar{G} \alpha\_{kj} = G^i\_{-j}` is the perfusion coefficient related to the compartments :math:`i` and :math:`j`, :math:`f^i` are sources or sinks which represent the external flow into the  $i$ -th compartment and :math:`p^i` is the pressure in the  $i$ -th compartment.

"""

```
from __future__ import absolute_import
from sfepy.base.base import Struct
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'),
    'Omegal': ('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.Omegal(mat.K, q1, p1)
               + dw_volume_dot.5.Omegal(mat.G_alfa, q1, p1)
               - dw_volume_dot.5.Omegal(mat.G_alfa, q1, p2)
               = dw_volume_integrate.5.Source(mat.f_1, q1)""",

    'komp2': """dw_diffusion.5.Omega2(mat.K, q2, p2)
               + dw_volume_dot.5.Omega2(mat.G_alfa, q2, p2)
               - dw_volume_dot.5.Omega2(mat.G_alfa, q2, p1)
               = dw_volume_integrate.5.Sink(mat.f_2, q2)"""
}

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

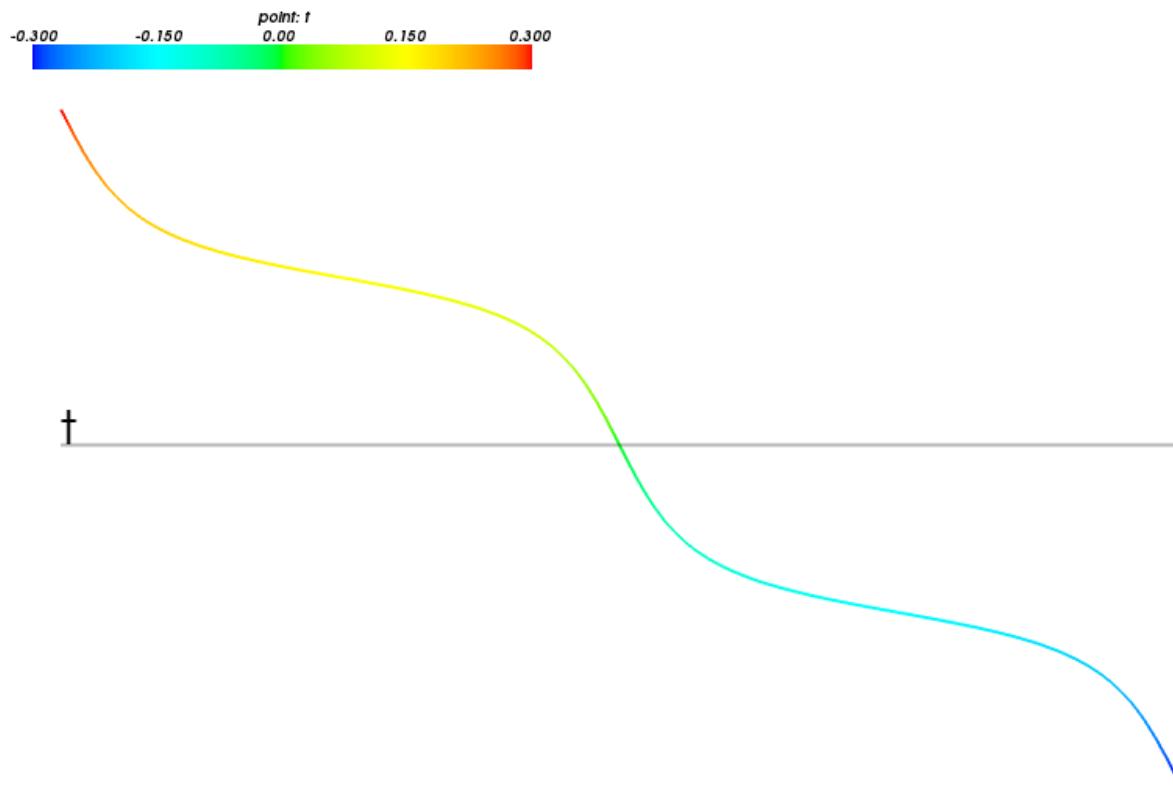
Find  $t$  such that:

$$\int_{\Omega} c(x) \frac{ds}{dx} \frac{dt}{dx} = 0, \quad \forall s,$$

where the coefficient  $c(x) = 0.1 + \sin(2\pi x)^2$  is computed in `get_coef()`.

View the results using:

```
$ ./postproc.py -b -d't,plot_warp_scalar,rel_scaling=1' --wireframe --view=-90,90,1.5,
--roll=0 laplace_1d.vtk
```



[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) \, \nabla t \cdot \nabla t = 0
    \quad \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::

$ ./postproc.py -b -d't,plot_warp_scalar,rel_scaling=1' --wireframe --view=-90,90,1.
--roll=0 laplace_1d.vtk
"""

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)
        desc = ['1_2']

        mesh = Mesh.from_data('laplace_1d', coors, None,
                              [conn], [mat_ids], desc)
        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),
}

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.

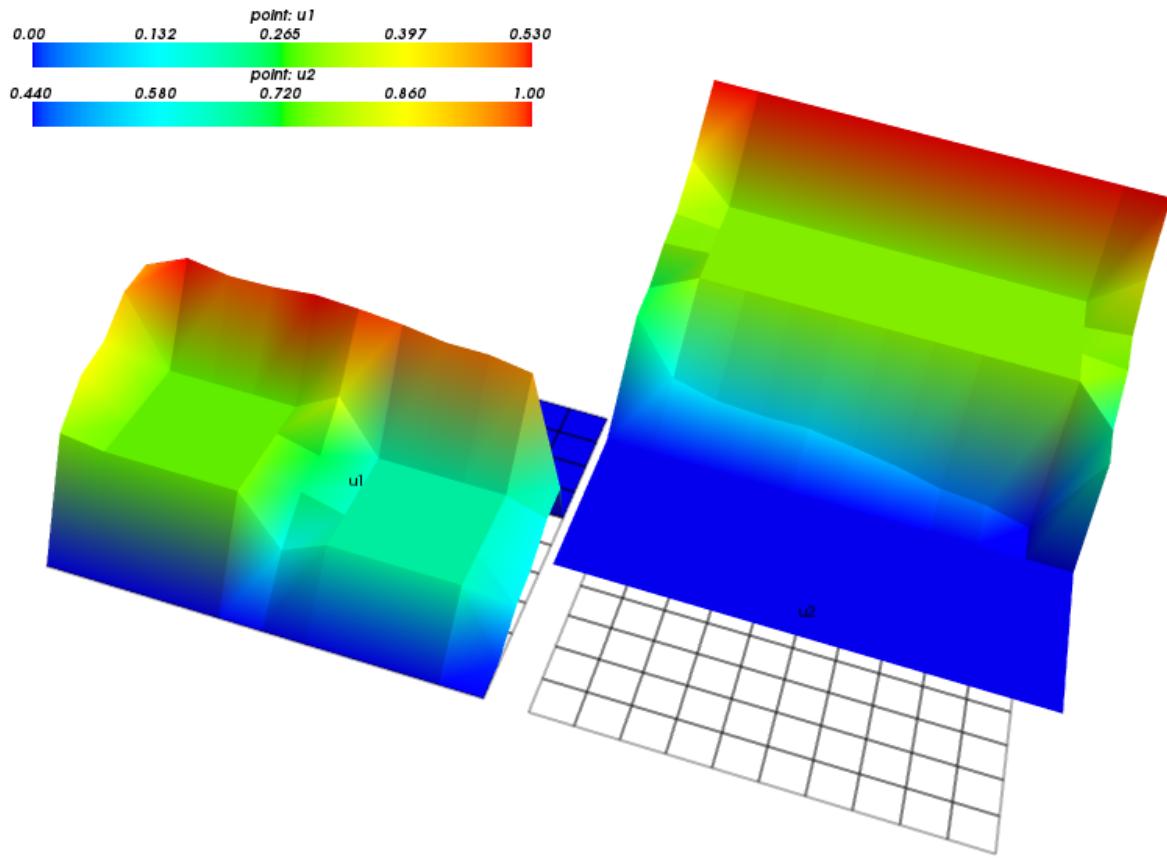
Find  $u$  such that:

$$\begin{aligned}
 & \int_{\Omega_1} \nabla v_1 \cdot \nabla u_1 = 0, \quad \forall v_1, \\
 & \int_{\Omega_2} \nabla v_2 \cdot \nabla u_2 = 0, \quad \forall v_2, \\
 & u_1 = 0 \text{ on } \Gamma_{bottom}, \\
 & u_2 = 1 \text{ on } \Gamma_{top}, \\
 & u_1(\underline{x}) = u_2(\underline{x}) + a(\underline{x}) \text{ for } \underline{x} \in \Gamma = \bar{\Omega}_1 \cap \bar{\Omega}_2 \\
 & u_1(\underline{x}) = u_1(\underline{y}) + b(\underline{y}) \text{ for } \underline{x} \in \Gamma_{left}, \underline{y} \in \Gamma_{right}, \underline{y} = P(\underline{x}), \\
 & u_1 = c_{11} \text{ in } \Omega_{m11} \subset \Omega_1, \\
 & u_1 = c_{12} \text{ in } \Omega_{m12} \subset \Omega_1, \\
 & u_2 = c_2 \text{ in } \Omega_{m2} \subset \Omega_2,
 \end{aligned}$$

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  $\Omega_{m11}$ ,  $\Omega_{m12}$  and  $\Omega_{m2}$  are replaced by the integral mean values.

View the results using:

```
$ ./postproc.py square_quad.vtk -b --wireframe -d'u1,plot_warp_scalar,rel_
˓→scaling=1:u2,plot_warp_scalar,rel_scaling=1'
```



[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 ;,

    \int_{\Omega_2} \nabla v_2 \cdot \nabla u_2
    = 0
    ;, \quad \forall v_2 ;,

    u_1 = 0 \text{ } \boxed{\text{on }} \Gamma_{bottom} ;
    u_2 = 1 \text{ } \boxed{\text{on }} \Gamma_{top} ;
    u_1(\boldsymbol{u}_1(\boldsymbol{x})) = u_2(\boldsymbol{u}_1(\boldsymbol{x})) + a(\boldsymbol{u}_1(\boldsymbol{x})) \text{ } \boxed{\text{for }} \\
    \boldsymbol{u}_1(\boldsymbol{x}) \in \bar{\Omega}_1 \cap \bar{\Omega}_2
    u_1(\boldsymbol{u}_1(\boldsymbol{y})) = u_1(\boldsymbol{u}_1(\boldsymbol{y})) + b(\boldsymbol{u}_1(\boldsymbol{y})) \text{ } \boxed{\text{for }}
```

```
\ul{x} \in \Gamma_{left}, \ul{y} \in \Gamma_{right}, \ul{y} = P(\ul{x}) \;;,  
u_1 = c_{11} \mbox{ in } \Omega_{m11} \subset \Omega_1 \;;,  
u_1 = c_{12} \mbox{ in } \Omega_{m12} \subset \Omega_1 \;;,  
u_2 = c_2 \mbox{ in } \Omega_{m2} \subset \Omega_2 \;;,  
  
where :math:`a(\ul{x})` , :math:`b(\ul{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_{m11}` , :math:`\Omega_{m12}` and :math:`\Omega_{m2}` are replaced  
by the integral mean values.
```

View the results using::

```
$ ./postproc.py square_quad.vtk -b --wireframe -d'u1,plot_warp_scalar,rel_  
scaling=1:u2,plot_warp_scalar,rel_scaling=1'  
"""  
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',  
    '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'),
}

ebcs = {
    'fix1' : ('Top', {'u2.all' : 1.0}),
    'fix2' : ('Bottom', {'u1.all' : 0.0}),
}

lcbc = {
    '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 = {
    'il' : 2,
}

equations = {
    'eq1' : """
        dw_laplace.il.Omega1(v1, u1) = 0
    """,
    'eq2' : """
        dw_laplace.il.Omega2(v2, u2) = 0
    """
}

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

```

```
'i_max'      : 1,
'eps_a'       : 1e-10,
} },
}
```

## 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:

$$\int_{\Omega} \nabla v \cdot \nabla u = 0 , \quad \forall s .$$

### 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:

```
$ python examples/diffusion/laplace_refine_interactive.py output
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'
```

Default options, 3D, storing results in ‘output’ directory:

```
$ python examples/diffusion/laplace_refine_interactive.py -3 output
$ python postproc.py output/hanging.vtk --wireframe -b --3d
```

Finer initial domain, 2D, storing results in ‘output’ directory:

```
$ python examples/diffusion/laplace_refine_interactive.py --shape=11,11 output
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'
```

Bi-quadratic approximation, 2D, storing results in ‘output’ directory:

```
$ python examples/diffusion/laplace_refine_interactive.py --order=2 output

# View solution with higher order DOFs removed.
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'

# View full solution on a mesh adapted for visualization.
$ python postproc.py output/hanging_u.vtk --wireframe -b -d'u,plot_warp_scalar'
```

[source code](#)

```
#!/usr/bin/env python
r"""
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 :math:`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::*

```
$ python examples/diffusion/laplace_refine_interactive.py output
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'
```

*Default options, 3D, storing results in 'output' directory::*

```
$ python examples/diffusion/laplace_refine_interactive.py -3 output
$ python postproc.py output/hanging.vtk --wireframe -b --3d
```

*Finer initial domain, 2D, storing results in 'output' directory::*

```
$ python examples/diffusion/laplace_refine_interactive.py --shape=11,11 output
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'

Bi-quadratic approximation, 2D, storing results in 'output' directory::

$ python examples/diffusion/laplace_refine_interactive.py --order=2 output

# View solution with higher order DOFs removed.
$ python postproc.py output/hanging.vtk --wireframe -b -d'u,plot_warp_scalar'

# View full solution on a mesh adapted for visualization.
$ python postproc.py output/hanging_u.vtk --wireframe -b -d'u,plot_warp_scalar'
"""

from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser

import os
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',
    '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('--3d', '--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),
                                  '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)

pb.set_bcs(ebcs=Conditions([fix1, fix2]))

pb.set_solver(nls)

state = pb.solve()

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:

```
./postproc.py laplace_shifted_periodic.vtk --wireframe -b -d'u,plot_warp_scalar,rel_
˓→scaling=1'
```

or use the `--show` option.

source code

```

#!/usr/bin/env python
"""
Laplace equation with shifted periodic BCs.

Display using::

    ./postproc.py laplace_shifted_periodic.vtk --wireframe -b -d'u,plot_warp_scalar,rel_
˓→scaling=1'

or use the --show option.
"""

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)
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):
    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)

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

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]',
'show' : 'show the results figure',
}

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('--show',
                        action="store_true", dest='show',
                        default=False, help=helps['show'])
    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 options.show:
        from sfepy.postprocess.viewer import Viewer
        from sfepy.postprocess.domain_specific import DomainSpecificPlot

        view = Viewer('laplace_shifted_periodic.vtk')
        view(rel_scaling=1,
              domain_specific={'u' : DomainSpecificPlot('plot_warp_scalar',
                                                        ['rel_scaling=1'])},
              is_scalar_bar=True, is_wireframe=True,
              opacity=0.3)

    if __name__ == '__main__':
        main()

```

## diffusion/laplace\_time\_ebcs.py

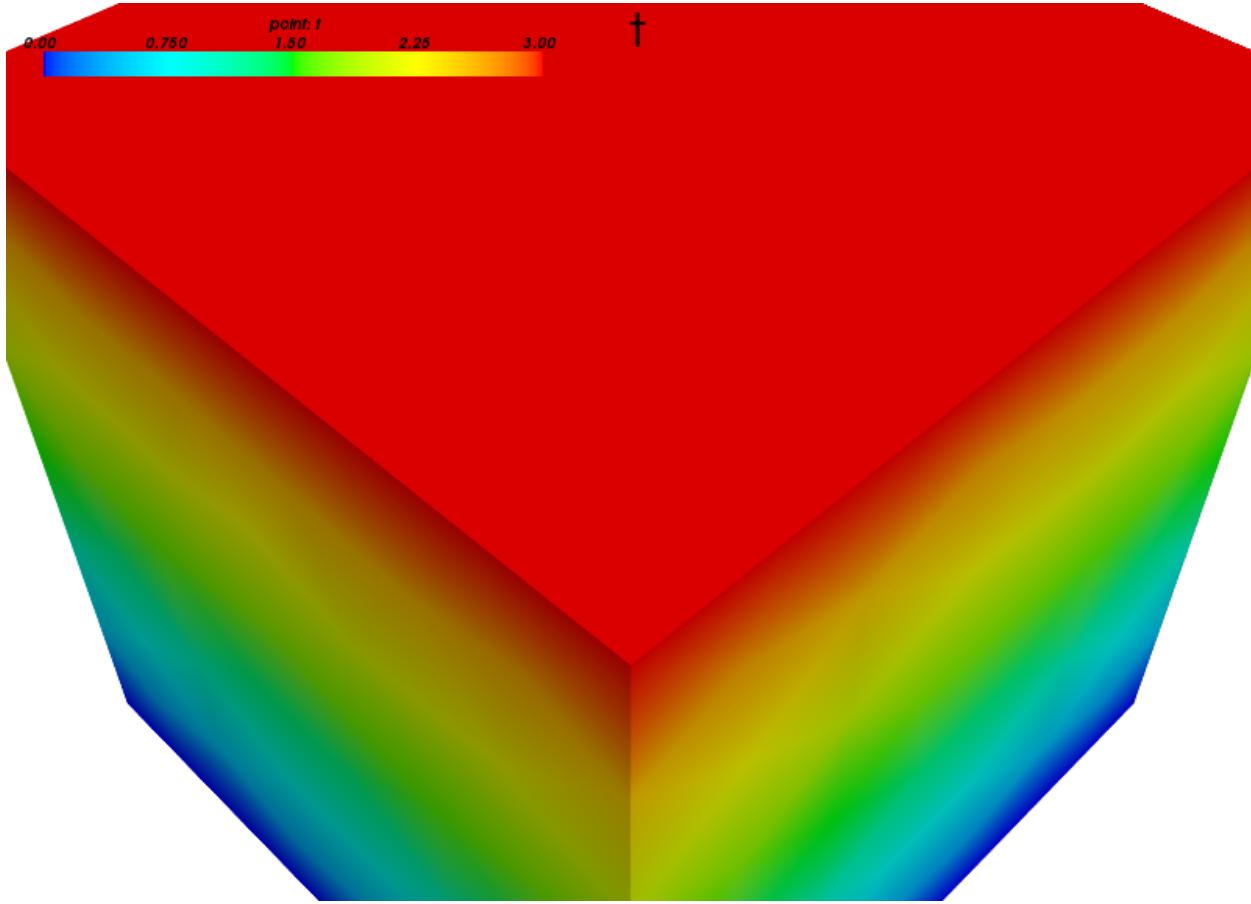
### Description

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.

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_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

    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

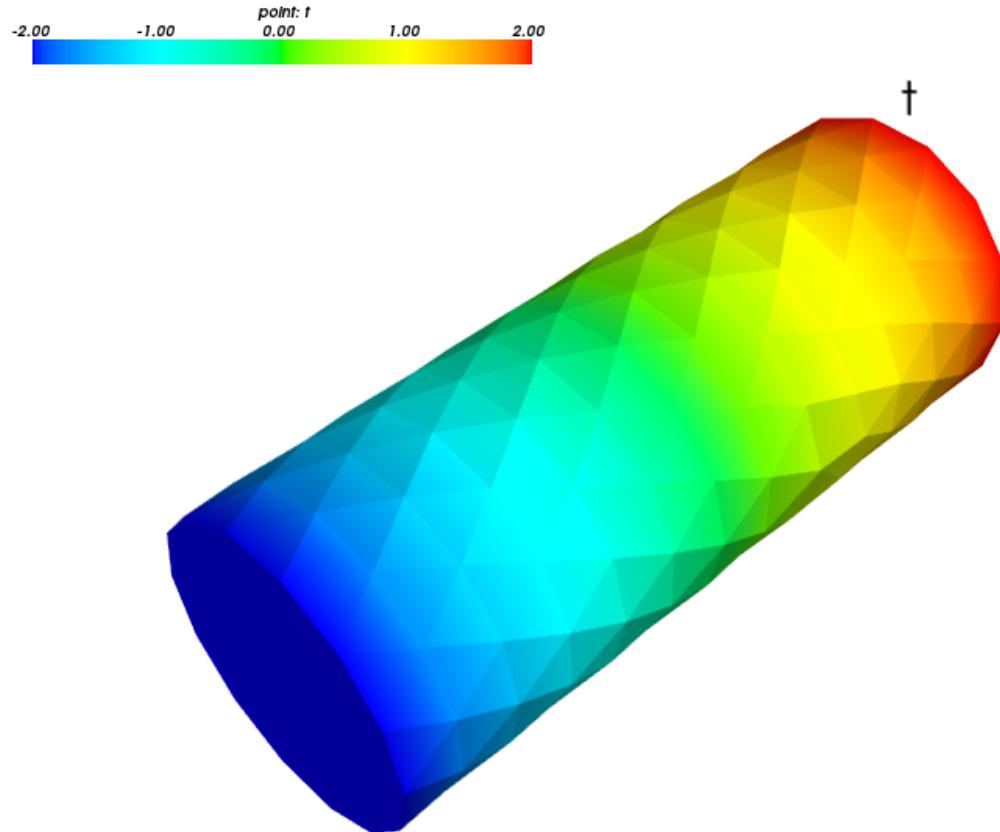
### Description

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',
    '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},
}

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

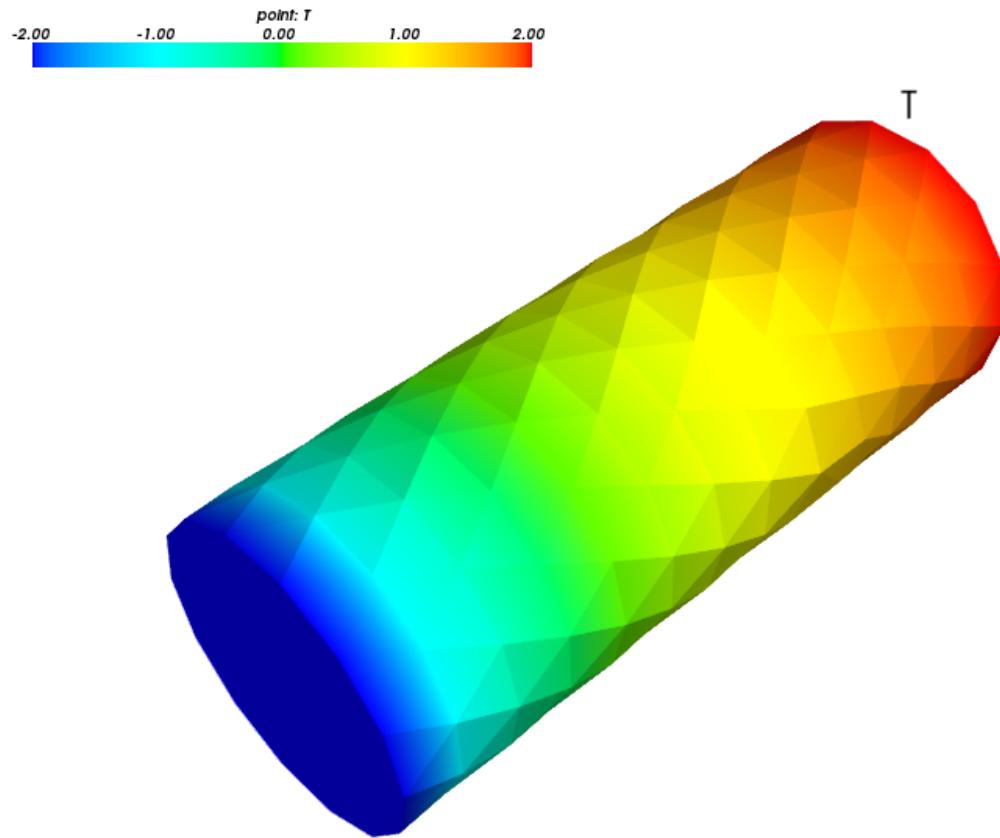
### Description

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_{\text{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
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_volume_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 = {
    '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:

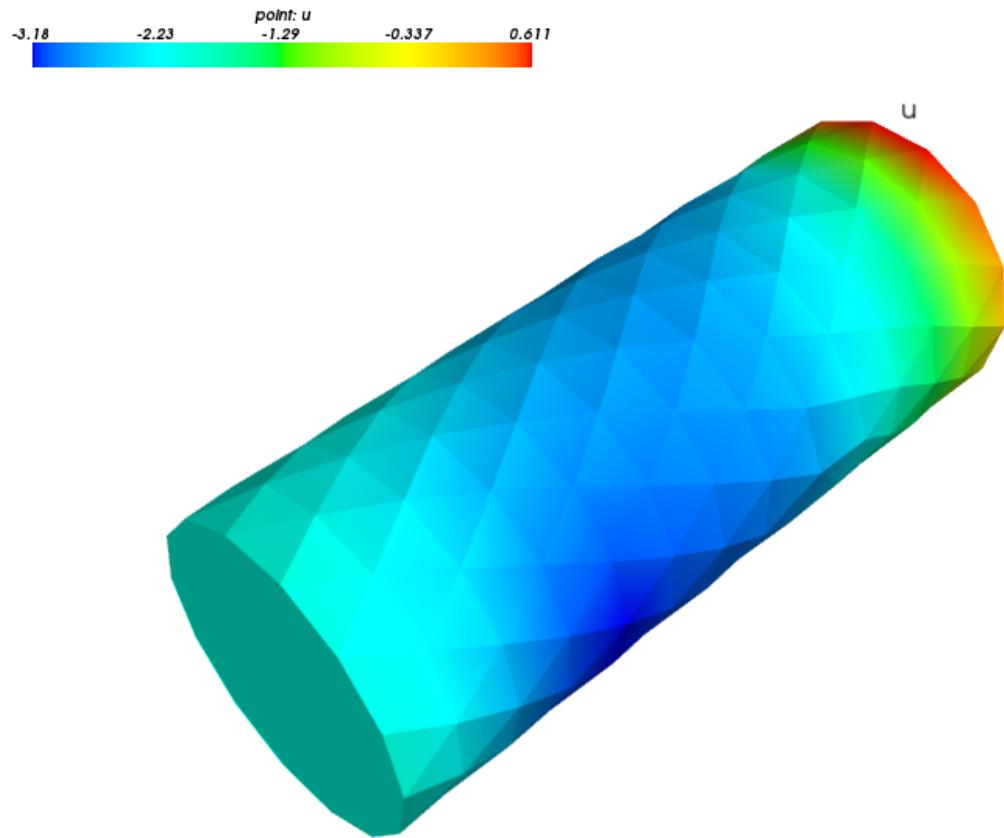
$$\int_{\Omega} c \nabla v \cdot \nabla u = - \int_{\Omega_L} b v = - \int_{\Omega_L} f v p, \quad \forall v,$$

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_volume_dot` should be replaced by `dw_volume_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 \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 `:func:`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 `:func:`get_load_variable()``.

In order to define the load  $b(x)$  directly, the term ``dw\_volume\_dot``

```

should be replaced by ``dw_volume_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,
}

equations = {
    'Laplace equation' :
        """dw_laplace.i.Omega( m.c, v, u )
        - dw_convect_v_grad_s.i.Omega( v, w, u )
        = - dw_volume_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 :math:`\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):
    """
    Define nodal values of 'p' in the nodal coordinates `coors`.

    """
    y = coors[:, 1]

    val = 5e5 * y
    return val

def get_convective_velocity(ts, coors, region=None):
    """
    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.

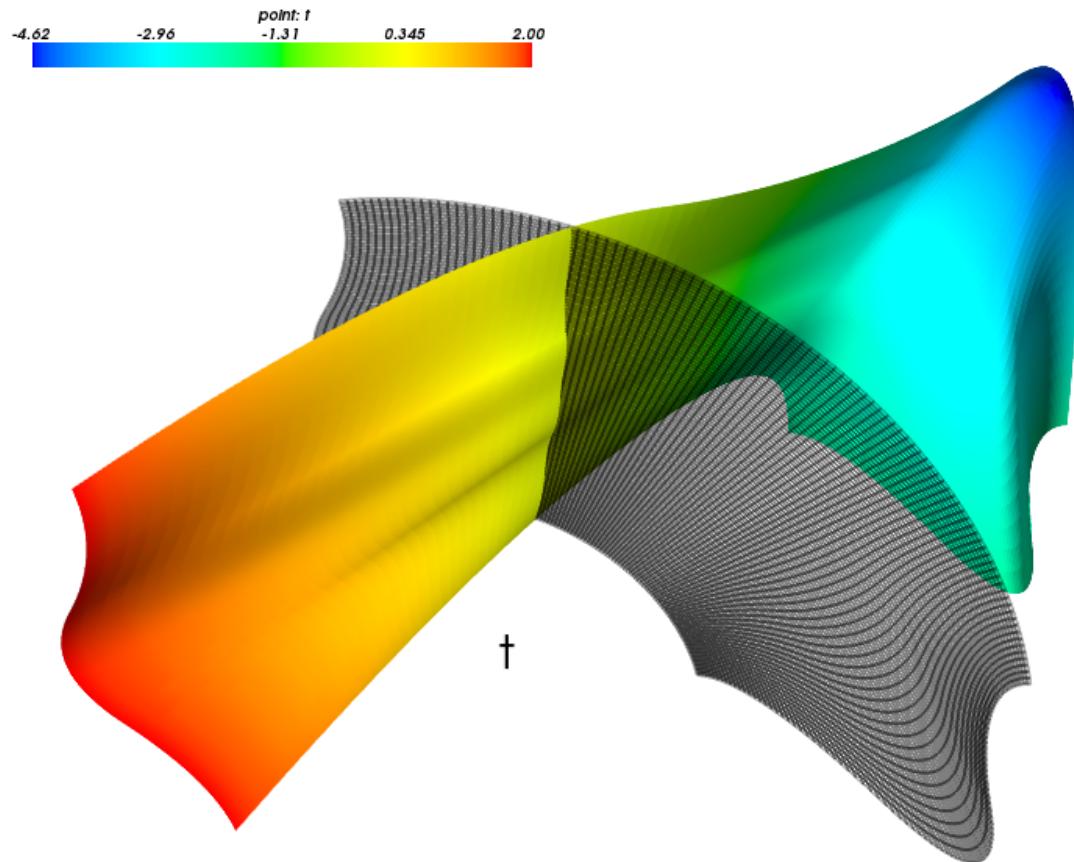
Find  $t$  such that:

$$\int_{\Omega} c \nabla s \cdot \nabla t = \int_{\Omega_0} f s, \quad \forall s.$$

Try setting the Dirichlet boundary condition (ebcs) on various sides of the domain ('Gamma1', ..., 'Gamma4').

View the results using:

```
$ ./postproc.py patch2d.vtk --wireframe -b
$ ./postproc.py patch2d.vtk --wireframe -b -d't,plot_warp_scalar,rel_scaling=1'
```



[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 \text{forall } s ;.
```

Try setting the Dirichlet boundary condition (ebcs) on various sides of the domain ('`'Gamma1'``', ..., ``'Gamma4'``').

View the results using::

```
$ ./postproc.py patch2d.vtk --wireframe -b
$ ./postproc.py patch2d.vtk --wireframe -b -d't,plot_warp_scalar,rel_scaling=1'
"""

from __future__ import absolute_import
from sfepy import data_dir

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.

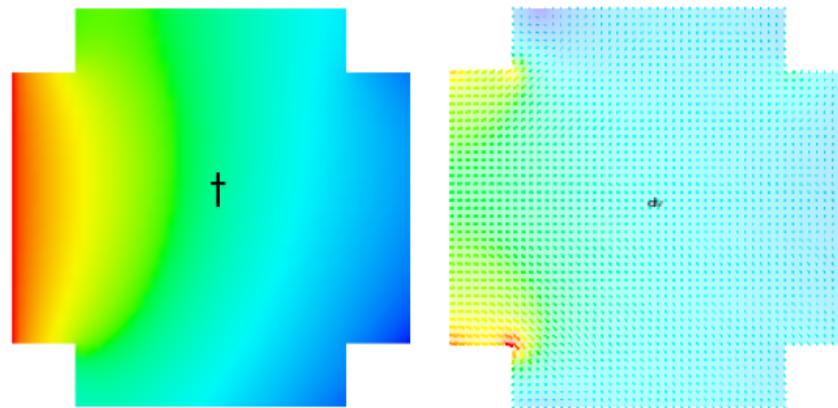
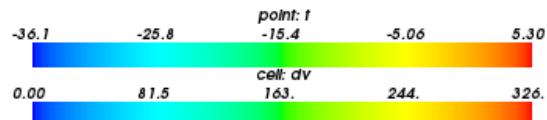
### Usage Examples

Run with the default settings (no refinement):

```
python simple.py examples/diffusion/poisson_neumann.py
```

Refine the mesh twice:

```
python simple.py 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,
```

where :math:`g` is the given flux, :math:`g = \bar{u}\_l \cdot K\_{ij} \nabla\_j p` on  $\Gamma_N$ , and :math:`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 ' $\Gamma_N$ ' (the Neumann condition boundary part), the flux/length should correspond to the given value :math:`g = -50`, while on ' $\Gamma_D$ ' 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)::

    python simple.py examples/diffusion/poisson_neumann.py

Refine the mesh twice::

    python simple.py 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('d_surface_flux.i.%s(m.K, t)' % gamma,
                            verbose=False)
        area = pb.evaluate('d_surface.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},),
    '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_surface_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\_parallel\_interactive.py

### Description

Parallel assembling and solving of a Poisson's equation, using commands for interactive use.

Find  $u$  such that:

$$\int_{\Omega} \nabla v \cdot \nabla u = \int_{\Omega} vf, \quad \forall s.$$

### 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:

```
$ python examples/diffusion/poisson_parallel_interactive.py -h
```

See PETSc options:

```
$ python examples/diffusion/poisson_parallel_interactive.py --help
```

Single process run useful for debugging with `debug()`:

```
$ python examples/diffusion/poisson_parallel_interactive.py output-parallel
```

Parallel runs:

```
$ mpixexec -n 3 python examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101
$ mpixexec -n 3 python examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --metis
$ mpixexec -n 5 python examples/diffusion/poisson_parallel_interactive.py output-parallel -2 --shape=101,101 --verify --metis -ksp_monitor -ksp_converged_reason
```

View the results using:

```
$ python postproc.py output-parallel/sol.h5 --wireframe -b -d'u,plot_warp_scalar'
```

source code

```
#!/usr/bin/env python
r"""
Parallel assembling and solving of a Poisson's equation, using commands for
interactive use.

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

```
.. math::  
    \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 `:func:`create_local_problem``.
- The command line options are saved into `<output_dir>/options.txt` file.

#### Usage Examples

---

##### See all options::

```
$ python examples/diffusion/poisson_parallel_interactive.py -h
```

##### See PETSc options::

```
$ python examples/diffusion/poisson_parallel_interactive.py -help
```

Single process run useful for debugging with `:func:`debug()``  
`<sfePy.base.base.debug>`::`:

```
$ python examples/diffusion/poisson_parallel_interactive.py output-parallel
```

##### Parallel runs::

```
$ mpixexec -n 3 python examples/diffusion/poisson_parallel_interactive.py output-  
parallel -2 --shape=101,101
```

```
$ mpixexec -n 3 python examples/diffusion/poisson_parallel_interactive.py output-  
parallel -2 --shape=101,101 --metis
```

```
$ mpixexec -n 5 python examples/diffusion/poisson_parallel_interactive.py output-  
parallel -2 --shape=101,101 --verify --metis -ksp_monitor -ksp_converged_reason
```

##### View the results using::

```
$ python postproc.py output-parallel/sol.h5 --wireframe -b -d'u,plot_warp_scalar'
```

```

"""
from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser
import os
import time
import glob
from itertools import chain

import numpy as nm
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.discrete.fem import Mesh, FEDomain, Field
from sfepy.discrete.common.region import Region
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                           Equation, Equations, Problem, State)
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('Gammal',
                                      '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)

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', gammal_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(ebc=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)

    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)

    mesh = Mesh.from_file(mesh_filename)

    if rank == 0:
        cell_tasks = pl.partition_mesh(mesh, size, use_metis=options.metis,
                                       verbose=True)

    else:
        cell_tasks = None

    output('creating global domain and field...')
    tt = time.clock()
    domain = FEDomain('domain', mesh)
    omega = domain.create_region('Omega', 'all')
    field = Field.from_args('fu', nm.float64, 1, omega, approx_order=order)
    output('...done in', time.clock() - tt)

    output('distributing field %s...' % field.name)
    tt = time.clock()

    distribute = pl.distribute_fields_dofs
    lfd, gfds = distribute([field], cell_tasks,
                           is_overlap=True,
                           save_inter_regions=options.save_inter_regions,
                           output_dir=options.output_dir,
                           comm=comm, verbose=True)
    lfd = lfd[0]

    output('...done in', time.clock() - tt)

    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...')
    tt = time.clock()

    omega_gi = Region.from_cells(lfd.cells, field.domain)

```

```
omega_gi.finalize()
omega_gi.update_shape()

pb = create_local_problem(omega_gi, order)

output('...done in', time.clock() - tt)

variables = pb.get_variables()
eqs = pb.equations

u_i = variables['u_i']
field_i = u_i.field

if options.plot:
    ppd.plot_local_dofs([None, None], field, field_i, omega_gi,
                         options.output_dir, rank)

output('allocating global system...')
tt = time.clock()

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,
                                             is_overlap=True, comm=comm,
                                             verbose=True)

output('...done in', time.clock() - tt)

output('evaluating local problem...')
tt = time.clock()

state = State(variables)
state.fill(0.0)
state.apply_ebc()

rhs_i = eqs.eval_residuals(state())
# This must be after pl.create_petsc_system() call!
mtx_i = eqs.eval_tangent_matrices(state(), pb.mtx_a)

output('...done in', time.clock() - tt)

output('assembling global system...')
tt = time.clock()

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)

output('...done in', time.clock() - tt)
```

```

output('creating solver...')
tt = time.clock()

conf = Struct(method='cg', precond='gamg', sub_precond='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)

output('...done in', time.clock() - tt)

output('solving...')
tt = time.clock()

psol = ls(prhs, psol)

psol_i = pl.create_local_petsc_vector(pdfs)
gather, scatter = pl.create_gather_scatter(pdfs, psol_i, psol, comm=comm)

scatter(psol_i, psol)

sol0_i = state() - psol_i[...]
psol_i[...] = sol0_i

gather(psol, psol_i)

output('...done in', time.clock() - tt)

output('saving solution...')
tt = time.clock()

u_i.set_data(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)

```

```
        output('...done in', time.clock() - tt)

    if options.show:
        plt.show()

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',
    'show' :
    'show partitioning plots (implies --plot)',
    '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('--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')

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))])

```

```

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)

mesh = gen_block_mesh(dims, shape, centre, name='block-fem',
                      verbose=True)
mesh.write(mesh_filename, io='auto')

comm.barrier()

output('field order:', options.order)

solve_problem(mesh_filename, options, 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  $\Omega$  is a square domain,  $\Omega_1 \in \Omega$  is a circular domain.

Now let's see what happens if  $\Omega_1$  diameter changes.

Run:

```
$ ./simple.py <this file>
```

and then look in ‘output/r\_omega1’ directory, try for example:

```
$ ./postproc.py output/r_omega1/circles_in_square*.vtk
```

Remark: this simple case could be achieved also by defining  $\Omega_1$  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 \mathbf{in} \Omega,
    t = 2 \mathbf{on} \Gamma_1 ;
    t = -2 \mathbf{on} \Gamma_2 ;
    f = 1 \mathbf{in} \Omega_1 ;
    f = 0 \mathbf{otherwise},
```

where  $\Omega$  is a square domain,  $\Omega_1 \in \Omega$  is a circular domain.

Now let's see what happens if  $\Omega_1$  diameter changes.

Run::

```
$ ./simple.py <this file>
```

and then look in 'output/r\_omegal' directory, try for example::

```
$ ./postproc.py output/r_omegal/circles_in_square*.vtk
```

Remark: this simple case could be achieved also by defining  $\Omega_1$  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_omegal_size',
    'output_dir' : 'output/r_omegal',
}
```

```
# 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.
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 = {
    '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_omegal_size( problem ):
    """Vary size of \Omega_\text{el}. 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_omegal_size:'

    diameters = nm.linspace( 0.1, 0.6, 7 ) + 0.001
    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.

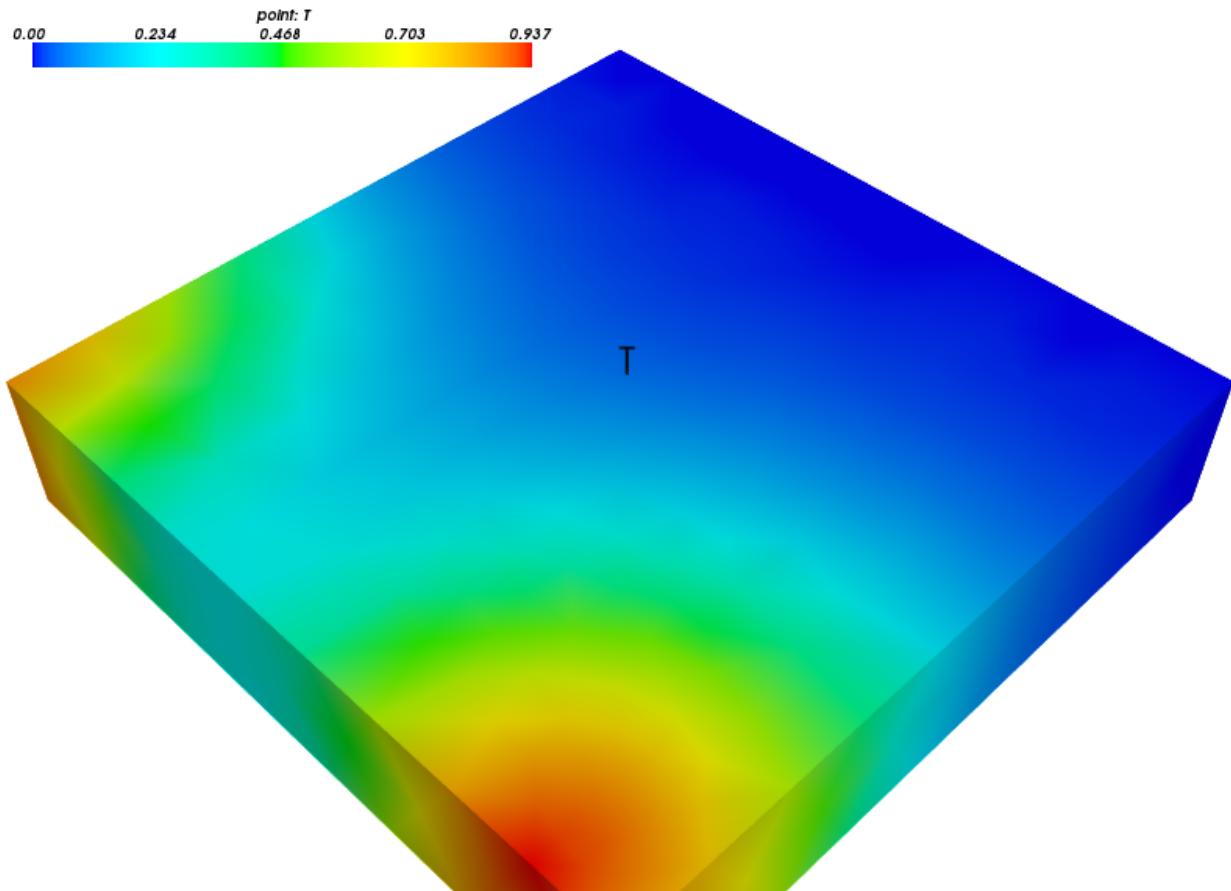
The mesh is suitable for periodic boundary conditions. It consists of a cylinder enclosed by a box in the x and y directions.

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:

$$\int_{\Omega} cs \frac{\partial T}{\partial t} + \int_{\Omega} \sigma_2 \nabla s \cdot \nabla T = \int_{\Omega_2} P_3 T, \quad \forall s.$$



[source code](#)

```
r"""

```

Transient Laplace equation with a localized power source and periodic boundary conditions.

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 :math:`t \in [0, t\_{\text{final}}]`.

Find :math:`T(t)` for :math:`t \in [0, t\_{\text{final}}]` such that:

```
.. math::
    \int_{\Omega} c s \ \operatorname{pdiff}\{T\}\{t\}
```

```
+ \int_{\Omega} \sigma_2 \nabla s \cdot \nabla T
= \int_{\Omega} \sigma_2 P_3 T
\;, \quad \forall s \;.
"""

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:
        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'),
}

ics = {
    'ic' : ('Omega', {'T.0' : 0.0}),
}

integrals = {
    'i' : 1,
}

equations = {
    'Temperature' :
    """dw_volume_dot.i.cylinder( cylinder.capacity, s, dT/dt )
       dw_volume_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_volume_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

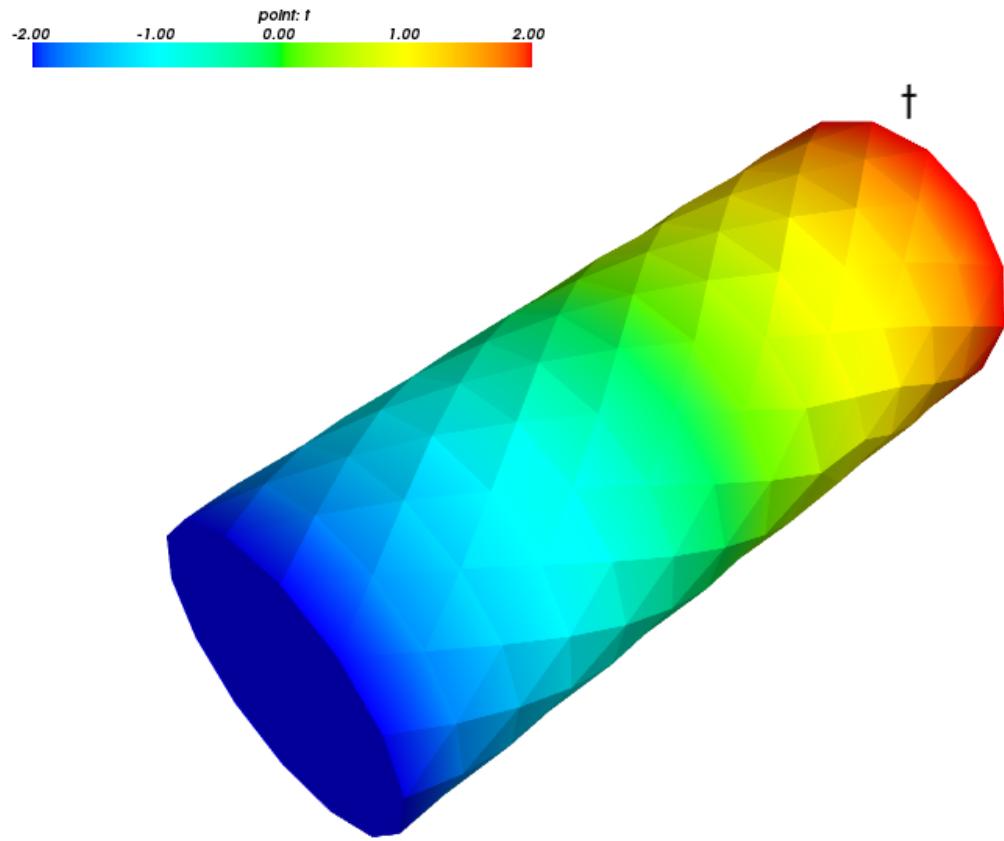
### Description

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} \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'

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.

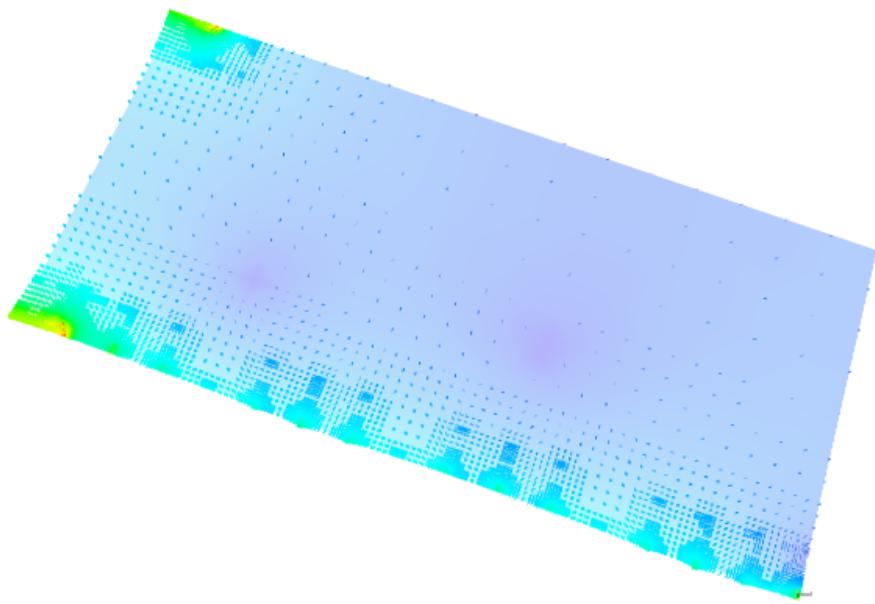
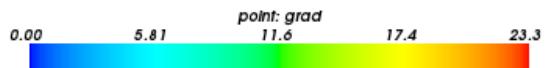
Find  $t$  such that:

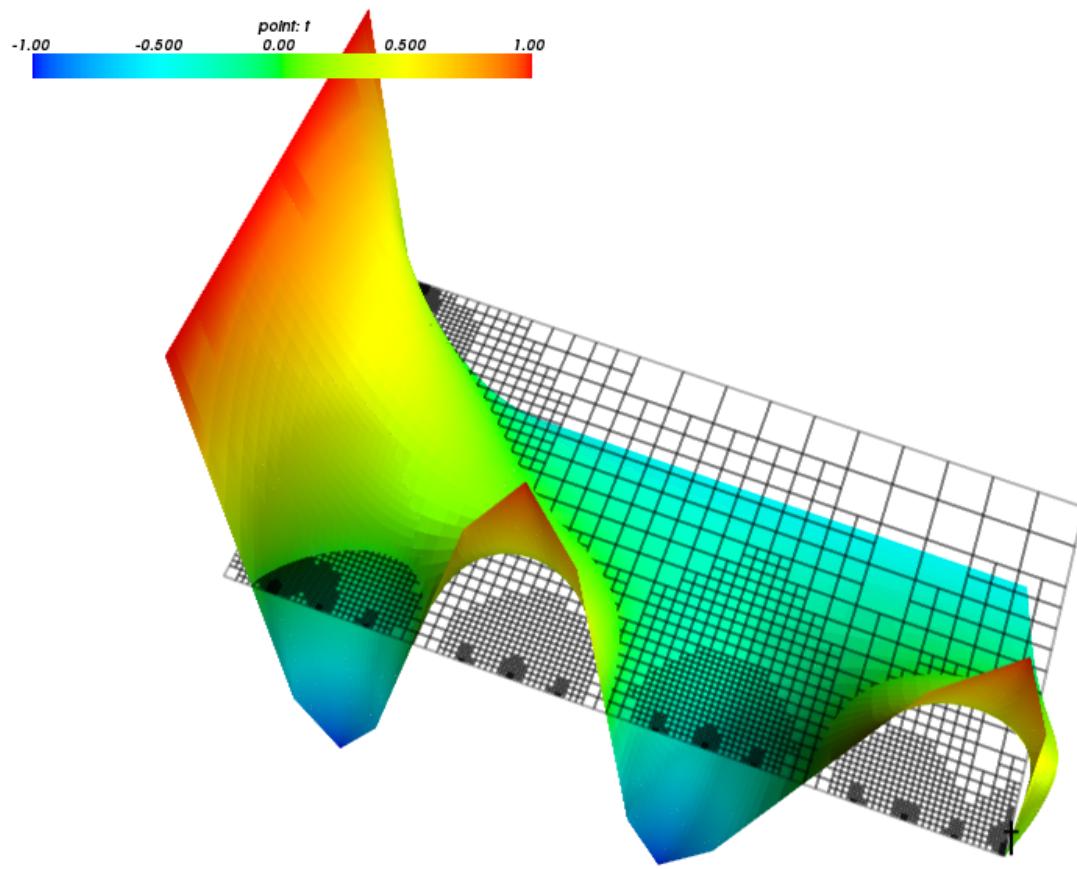
$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$

This example demonstrates how to use a hierarchical basis approximation - it uses the fifth order Lobatto polynomial space for the 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. Use the following commands to view the results (assuming default output directory and names):

```
$ ./postproc.py -b -d't,plot_warp_scalar,rel_scaling=1' 2_4_2_refined_t.vtk --
˓→wireframe
$ ./postproc.py -b 2_4_2_refined_grad.vtk
```

The `sfePy.discrete.fem.meshio.UserMeshIO` class is used to refine the original two-element mesh before the actual solution.





[source code](#)

```
r"""
Laplace equation with Dirichlet boundary conditions given by a sine function
and constants.
```

Find :math:`t` such that:

```
.. math:::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    ;, \quad \forall s ;.
```

This example demonstrates how to use a hierarchical basis approximation – it uses the fifth order Lobatto polynomial space for the 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. Use the following commands to view the results (assuming default output directory and names)::

```
$ ./postproc.py -b -d't,plot_warp_scalar,rel_scaling=1' 2_4_2_refined_t.vtk --
→wireframe
$ ./postproc.py -b 2_4_2_refined_grad.vtk
```

The `:class:`sfepy.discrete.fem.meshio.UserMeshIO`` class is used to refine the original two-element mesh before the actual solution.

"""

```

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.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):
    """
    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

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' : 0, # Min. refinement level to achieve everywhere.
    'max_level' : 5, # Max. refinement level.
    'eps' : 1e-3, # 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', 5, 'H1', 'lobatto'),
    # Compare with the Lagrange basis.
    ## 'temperature' : ('real', 1, 'Omega', 5, 'H1', 'lagrange'),
}

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

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,
    }),
}
```

## diffusion/time\_advection\_diffusion.py

### Description

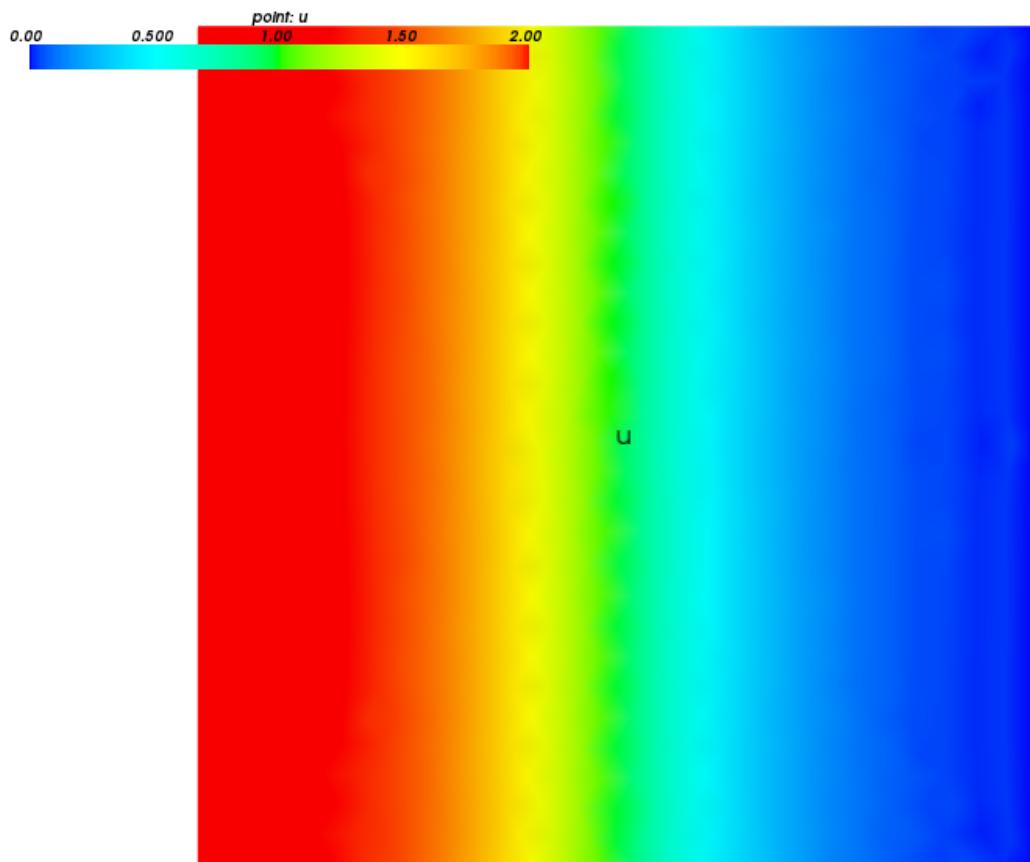
The transient advection-diffusion equation with a given divergence-free advection velocity.

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:

```
python postproc.py square_triz.*.vtk -b --wireframe
```



[source code](#)

```
r"""
The transient advection-diffusion equation with a given divergence-free
advection velocity.
```

Find :math:`u` such that:

```
.. math::
    \int_{\Omega} s \partial_t u + \int_{\Omega} s \nabla \cdot (\underline{v} u) + \int_{\Omega} D \nabla s \cdot \nabla u
    = 0
```

```
\;, \quad \forall s \;.\n\nView the results using::\n\n    python postproc.py square_tri2.*.vtk -b --wireframe\n\n"""\nfrom __future__ import absolute_import\nfrom sfepy import data_dir\n\nfilename_mesh = data_dir + '/meshes/2d/square_tri2.mesh'\n\nregions = {\n    'Omega' : 'all', # or 'cells of group 6'\n    'Gamma_Left' : ('vertices in (x < -0.99999)', 'facet'),\n    'Gamma_Right' : ('vertices in (x > 0.99999)', 'facet'),\n}\n\nfields = {\n    'concentration' : ('real', 1, 'Omega', 1),\n}\n\nvariables = {\n    'u' : ('unknown field', 'concentration', 0, 1),\n    's' : ('test field', 'concentration', 'u'),\n}\n\nebcs = {\n    'u1' : ('Gamma_Left', {'u.0' : 2.0}),\n    'u2' : ('Gamma_Right', {'u.0' : 0.0}),\n}\n\n# Units: D: 0.0001 m^2 / day, v: [0.1, 0] m / day -> time in days.\nmaterials = {\n    'm' : ({'D' : 0.0001, 'v' : [[0.1], [0.0]]}),,\n}\n\nintegrals = {\n    'i' : 2,\n}\n\nequations = {\n    'advection-diffusion' :\n        """\n            dw_volume_dot.i.Omega(s, du/dt)\n            + dw_advect_div_free.i.Omega(m.v, s, u)\n            + dw_laplace.i.Omega(m.D, s, u)\n            = 0\n        """\n}\n\nsolvers = {\n    'ts' : ('ts.simple', {\n        't0' : 0.0,\n        't1' : 10.0,\n        'dt' : None,\n        'n_step' : 11, # Has precedence over dt.\n        'verbose' : 1,\n    }),
```

```

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

options = {
    'ts' : 'ts',
    'nls' : 'newton',
    'ls' : 'ls',
    'save_times' : 'all',
}

```

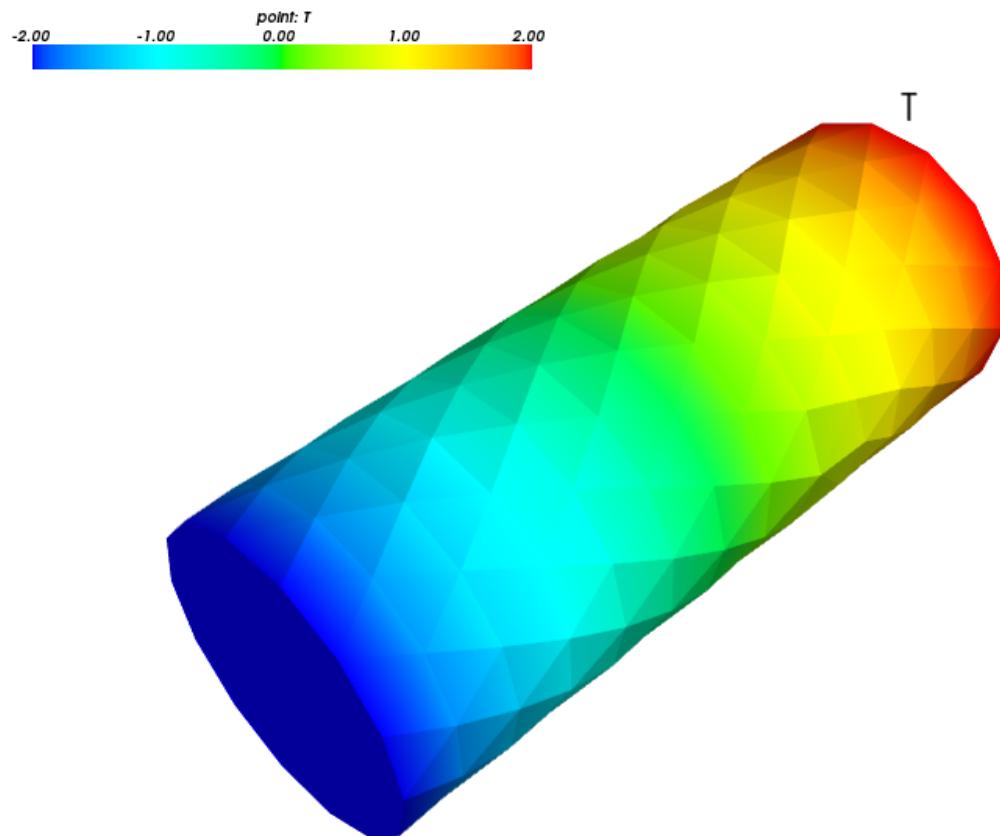
## diffusion/time\_poisson.py

### 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
function.

Find :math:`T(t)` for :math:`t \in [0, t_{\rm final}]` such that:

.. math::
    \int_{\Omega} s \, \operatorname{pdiff}\{T\}{t}
    + \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'

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 > 0.099999)', 'facet'),
}

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_volume_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,
    '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

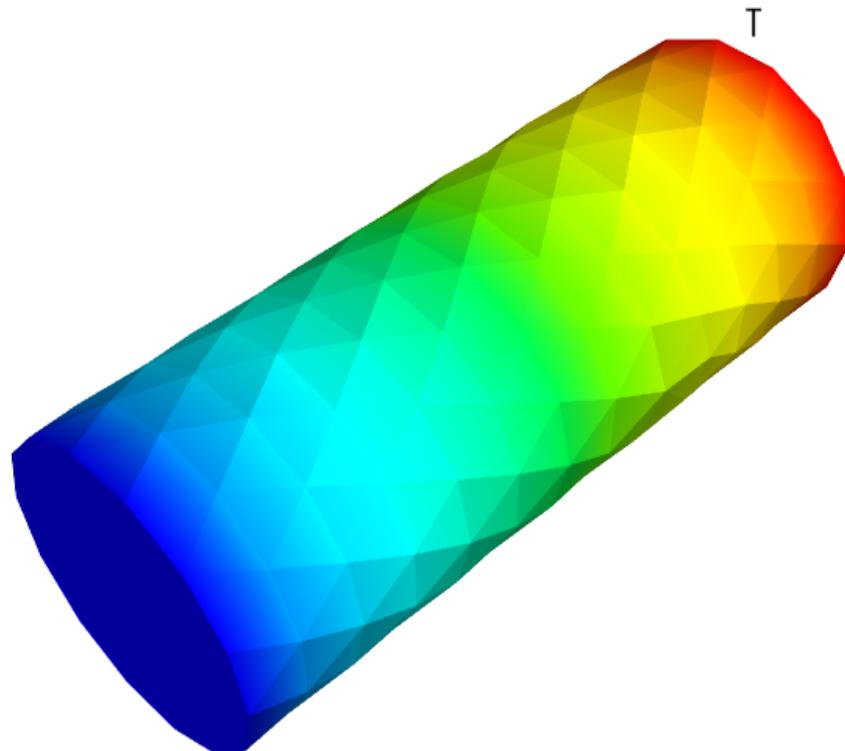
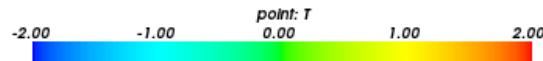
### Description

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 :math:`T(t)` for :math:`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 \forall s ;
"""

from __future__ import absolute_import
from sfepy import data_dir

from examples.diffusion.time_poisson import get_ic

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

materials = {
    'coef' : ({'val' : 0.01},),
}

regions = {
    'Omega' : 'all',
    '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, 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_volume_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,
        '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 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 examples/diffusion/time_poisson_interactive.py -h
"""
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, 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)

    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, 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'])
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_volume_dot(s, dT/dt)',
             integral, omega, s=s, T=T)
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]))

state0 = pb.get_initial_state()
init_fun, prestep_fun, _poststep_fun = pb.get_tss_functions(state0)

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):
        _poststep_fun(ts, vec)

        # Probe the solution.
        dvel_qp = ev('ev_diffusion_velocity.%d.Omega(m.diffusivity, T)' %
                     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_%s.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

    else:
        poststep_fun = _poststep_fun

pb.time_update(tss.ts)
state0.apply_ebc()

# This is required if {'is_linear': True} is passed to Newton.
mtx = prepare_matrix(pb, state0)
pb.try_presolve(mtx)

tss_status = IndexedStruct()
tss(state0.get_vec(pb.active_only),
     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()

```

### 5.6.3 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

    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)'),
}

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' : 'd_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, Pil, Pi2 )',
        'set_variables': [('Pil', ('pis', 'corrs_rs'), 'u'),
}

```

```
        ('Pi2', ('pis', 'corrs_rs'), 'u')],
    'class' : cb.CoeffSymSym,
},
'vol': {
    'regions': ['Ym', 'Yf'],
    'expression': 'd_volume.5.%s(u)',
    'class': cb.VolumeFractions,
},
'filenames' : {},
}

requirements = {
    'pis' : {
        'variables' : ['u'],
        'class' : cb.ShapeDimDim,
    },
    'corrs_rs' : {
        'requires' : ['pis'],
        'ebcs' : ['fixed_u'],
        'epbc' : all_periodic,
        'equations' : equation_corrs,
        'set_variables' : [('Pi', 'pis', 'u')],
        'class' : cb.CorrDimDim,
        'save_name' : 'corrs_le',
    },
}
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-4,
        'problem': 'linear',
    })
}

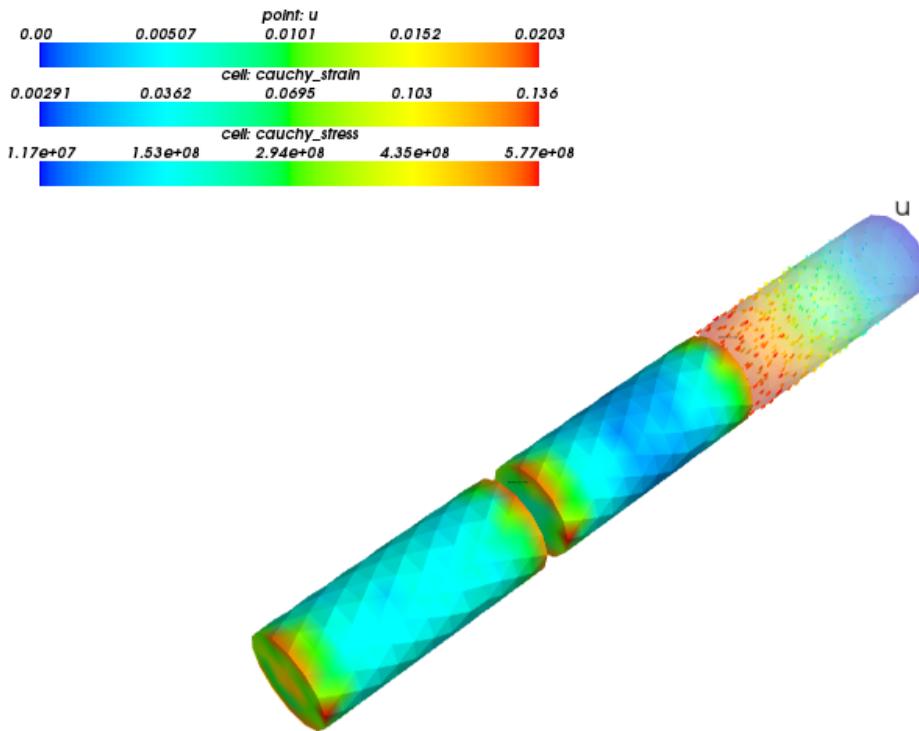
if is_opt:
    options.update({
        'parametric_hook': 'optimization_hook',
        'float_format': '%.16e',
    })

return locals()
```

## homogenization/linear\_elastic\_mM.py

### Description

missing description!



source code

```
from __future__ import absolute_import
import os
from sfepy import data_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')
        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);

    rstrain = pb.evaluate('ev_cauchy_strain.i.%s(u)' % rname,
                          mode='el_avg')

    recover_micro_hook(pb.conf.options.micro_filename,
                       region, {'strain' : rstrain})

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, **kwargs):
    global regenerate

    out = get_homog_coefs_linear(ts, coors, mode, regenerate=regenerate,
                                 micro_filename=options['micro_filename'])
    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'),
    '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 = data_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_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)

        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},),
        }

        filename_mesh.define(is_opt=is_opt, regions=regions,
                             functions=functions, materials=materials)
```

```

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'}),
}

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](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 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,\n    compute_stress_strain_u, compute_mac_stress_part\n\n\ndef recovery_le(pb, corrs, macro):\n    out = {}\n\n    dim = corrs['corrs_le']['u_00'].shape[1]\n    mic_u = - compute_micro_u(corrs['corrs_le'], macro['strain'], 'u', dim)\n\n    out['u_mic'] = Struct(name='output_data',\n                           mode='vertex', data=mic_u,\n                           var_name='u', dofs=None)\n\n    stress_Y, strain_Y = \
        compute_stress_strain_u(pb, 'i', 'Y', 'mat.D', 'u', mic_u)\n    stress_Y += \
        compute_mac_stress_part(pb, 'i', 'Y', 'mat.D', 'u', macro['strain'])\n\n    strain = macro['strain'] + strain_Y\n\n    out['cauchy_strain'] = Struct(name='output_data',\n                                   mode='cell', data=strain,\n                                   dofs=None)\n    out['cauchy_stress'] = Struct(name='output_data',\n                                   mode='cell', data=stress_Y,\n                                   dofs=None)\n\n    return out\n\nfilename_mesh = data_dir + '/meshes/3d/matrix_fiber.mesh'\ndim = 3\nregion_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),
                   '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
    'volume': {'expression': 'd_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.scipy_direct', {}),
    '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,
                           var_name='u', dofs=None)
    out['p_mic'] = Struct(name='output_data',
                           mode='cell', data=mic_p[:, nm.newaxis,
                           :, nm.newaxis],
                           var_name='p', dofs=None)

    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,
                               dofs=None)
out['cauchy_stress'] = Struct(name='output_data',
                               mode='cell', data=stress_Y,
                               dofs=None)
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'),
    'Pilu': ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pi2u': ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pilp': ('parameter field', 'corrector_p', '(set-to-None)'),
    'Pi2p': ('parameter field', 'corrector_p', '(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:
    epbc = {
        '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:
    epbc = {
        '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':
        """- dw_stokes.i.Y(u, q)
        - dw_volume_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, Pilu, Pi2u)',
        'set_variables': [('Pilu', ('pis', 'corrs_rs'), 'u'),
                          ('Pi2u', ('pis', 'corrs_rs'), 'u')],
        'class': cb.CoeffSymSym,
    },
    'elastic_p': {
        'requires': ['corrs_rs'],
        'expression': 'dw_volume_dot.i.Y(mat.gamma, Pilp, Pi2p)',
        'set_variables': [('Pilp', 'corrs_rs', 'p'),

```

```

        ('Pi2p', 'corrs_rs', 'p')],
    'class': cb.CoefSymSym,
},
'D': {
    'requires': ['c.elastic_u', 'c.elastic_p'],
    'class': cb.CoefSum,
},
'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_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

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,
                          save_field_meshes=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
        if hasattr(opts, 'parametric_hook'): # Parametric study.
            parametric_hook = conf.get_function(opts.parametric_hook)
            app.parameterize(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)

        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([['O'], ['E_f', 'E_m'], ['v_f', 'v_m']],
                      ylabels=['Obj. fun.', "Young's modulus", "Poisson's ratio"],
                      xlabel='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(fevel, 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():
    srkdir = 'examples/homogenization/'
    micro_filename = srkdir + 'homogenization_opt.py'
    macro_filename = srkdir + '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],
                           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_volume_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_volume_integrate_mat.3.Y(mat_he.E, u)',
                     mode='el_avg')

out['green_strain'] = Struct(name='output_data',
                               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('file_per_var', False)
pb.save_state(filename, out=out, file_per_var=fpv)

def def_mat(ts, mode, coors, term, pb):
    if not (mode == 'qp'):
        return

    if not hasattr(pb, 'family_data'):
        pb.family_data = HyperElasticULFFamilyData()

    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, term.integration)

```

```
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.integration,
                                                    term.region.name)

conf_mat = pb.conf.materials
solid_key = [key for key in conf_mat.keys() if 'solid' in key][0]
solid_mat = conf_mat[solid_key].values
mat = {}
for mat_key in ['mu', 'K']:
    if isinstance(solid_mat[mat_key], dict):
        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))
    else:
        mat[mat_key] = nm.ones((n_el, n_qp, 1, 1)) * solid_mat[mat_key]

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': 'd_volume.5.Y(u)'},
    'output_dir': './output',
    'coefs_filename': 'coefs_hyper_homog',
    'multiprocessing': True,
    'chunks_per_worker': 2,
    'mesh_update_variable': 'u',
    'mesh_update_corrector': 'corrs_rs',
    'recovery_hook': 'recovery_hook',
    'store_micro_idxs': [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': 1000,
               'mu': {'Ym': 100, 'Yc': 10},
               },),
}
}

variables = {
    'u': ('unknown field', 'displacement'),
    'v': ('test field', 'displacement', 'u'),
    'Pi': ('parameter field', 'displacement', 'u'),
    'Pilu': ('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 = {
}

```

```
'A': {
    'requires': ['pis', 'corrs_rs'],
    'expression': 'dw_nonsym_elastic.3.Y(mat_he.A, Pilu, Pi2u)',
    'set_variables': [('Pilu', ('pis', 'corrs_rs'), 'u'),
                      ('Pi2u', ('pis', 'corrs_rs'), 'u')],
    'class': cb.CoeffNonSymNonSym,
},
'S': {
    'expression': 'ev_volume_integrate_mat.3.Y(mat_he.S, u)',
    'set_variables': [],
    'class': cb.CoeffOne,
}
}

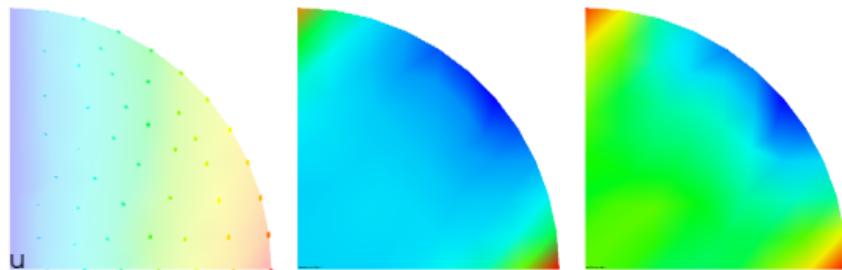
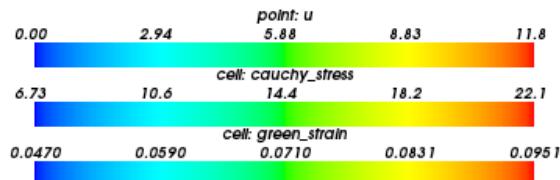
requirements = {
    'pis': {
        'variables': ['u'],
        'class': cb.ShapeDimDim,
    },
    'corrs_rs': {
        'requires': ['pis'],
        'ebcs': ['fixed_u'],
        'epbcs': ['periodic_ux', 'periodic_uy'],
        'equations': {
            'balance_of_forces':
                """dw_nonsym_elastic.3.Y(mat_he.A, 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.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-4,
        'problem': 'nonlinear',
    }),
}
```

## homogenization/nonlinear\_hyperelastic\_mM.py

### Description

missing description!



[source code](#)

```
import numpy as nm
import six

from sfepy import data_dir
from sfepy.base.base import Struct, output
from sfepy.terms.terms_hypereastic_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_volume_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_volume_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_idxs = 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_idxs:
                key = happ.get_micro_cache_key('coors', ii, pb.ts.step)
                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.integration)

    mtx_f = family_data.mtx_f.reshape((coors.shape[0],
                                        + family_data.mtx_f.shape[-2:]))

    out = get_homog_coefs_nonlinear(ts, coors, mode, mtx_f,
                                    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):
    vec = pb.equations.make_full_vec(vec)
    pb.equations.set_variables_from_state(vec)

    update_var = pb.conf.options.mesh_update_variables[0]
    state_u = pb.equations.variables[update_var]

    nods = state_u.field.get_dofs_in_region(state_u.field.region, merge=True)
    coors = pb.domain.get_mesh_coors().copy()
    coors[nods, :] += state_u().reshape(len(nods), state_u.n_components)

    if len(state_u.field.mappings0) == 0:
        state_u.field.save_mappings()

    state_u.field.clear_mappings()
    pb.set_mesh_coors(coors, update_fields=False, actual=True,
                      clear_all=False)

    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': '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'

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,
    })
}

```

## 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>

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.coeffs_base as cb
import numpy as nm
from sfepy import data_dir
import six
from six.moves import range

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 \
            six.iteritems(corrs['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(corrs):
            if ('_gamma_' + ch in key):
                kk = int(key[-1]) - 1
                press_mic += corrs[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'] += corrs['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)

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']):
    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': (['P1YM', 'PrYM'], {'pM.0': 'pM.0'}, 'match_x_plane')}
lcbs = {}

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'),
})

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'),
        'P1Y' + ch: ('r.P1 *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: ([['P1Y' + 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],
                           {'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}),
})

lcbcbs.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'])

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',
    'ls': 'ls', # linear solver to use
    'volumes': {
        'total': {
            'variables': ['vol_all'],
            'expression': """d_volume.iV.Y(vol_all)""",
        },
        'one': {
            'value': 1.0,
        }
    },
    'output_dir': './output',
    'file_per_var': True,
    'coefs_filename': 'coefs_perf_ ' + pb_def['name'],
    'coefs_info': {'eps0': eps0},
    'recovery_hook': 'recovery_perf',
}

for ipm in ['p', 'm']:
    options['volumes'].update({
        'bYM' + ipm: {
            'variables': ['pM'],
            'expression': "d_surface.iS.bYM%s(pM) % ipm",
        },
        'bY' + ipm: {
            'variables': ['vol_all'],
            'expression': "d_surface.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': "d_surface.iS.%s(p%s) % (ireg, ch)",
            }
        })

coefs = {
    'vol_bYMp': {
        'regions': ['bYMp', 'bYMM'],
        'expression': 'd_surface.iS.%s(pM)',
        'class': cb.VolumeFractions,
    },
    'filenames': {},
}

requirements = {
    'corrs_one_YM': {
        'variable': ['pM'],
```

```

        'ebcs': ['gamma_pm_YMmchs', 'gamma_pm_bYMchs'],
        'epbcs': [],
        'save_name': 'corrs_one_YM',
        'class': cb.CorrSetBCS,
    },
}

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_surface_integrate.iS.bYM%s(qM) """
                    % (1.0/param_h, ipm),
            },
            'class': cb.CorrOne,
            'save_name': 'corrs_%s_gamma_%s' % (pb_def['name'], ipm),
        },
    })

    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_surface_integrate.iS.bYM%s(corr_M)' % ipm2,
                'set_volume': 'bYp',
                'class': cb.CoefOne,
            },
        })
    }

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)
    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(kwags.keys()), 'corrs_gamma_')

pis = kwags['pis_' + ch]
corrs_gamma = kwags['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, **kwags):
    ch = get_channel(list(kwags.keys()), 'pis_')
    pis = kwags['pis_' + ch]
    corrs_pi = kwags['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_coppis,
            '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,
        },
        'corrs_one' + ch: {
            'variable': ['pM'],
            'ebcs': ebc_eta[ch],
            'epbcs': [],
            'save_name': 'corrs_%s_one%s' % (pb_def['name'], ch),
            'class': cb.CorrSetBCS,
        },
        'corrs_eta' + ch: {
            'ebcs': ebc_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),
    },
    'corrs_pi' + ch: {
        'requires': ['pis_' + ch],
        'set_variables': [('Pi_' + ch, 'pis_' + ch, 'p' + ch)],
        'ebcs': [],
        'epbcs': all_periodicY[ch],
        'lcbc': ['imv' + ch],
        'equations': {
            'eq_pi': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                        + dw_volume_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_volume_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_surface_integrate.iS.bYM%s(corr_M)' % ipm,
            'set_volume': 'bYp',
            'class': cb.CoefOne,
        },
        '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_surface_integrate.iS.bYM%s(corr1_M)"""
                            % (1.0/param_h, ipm),
            'class': cb.CoefOne,
        },
    })
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.CoefDim,
    },
    'P' + io: { # test+
        'requires': ['pis_' + ch, 'corrs_pi' + ch],
        'set_variables': set_corr_cc,
    })

```

```
'expression': 'ev_surface_integrate.iS.bY%s(corr1_%s)' \
    % (io, ch),
'set_volume': 'bYp',
'dim': pb_def['dim'] - 1,
'class': cb.CoefDim,
},
'S_test' + io: {
    'requires': ['corrs_pi' + ch],
    'set_variables': [('corr1_' + ch, 'corrs_pi' + ch, 'p' + ch)],
    'expression': '%e * ev_surface_integrate.iS.bY%s(corr1_%s)' \
        % (1.0 / param_h, io, ch),
    'dim': pb_def['dim'] - 1,
    'class': cb.CoefDim,
},
}),
})

requirements.update({
    'corrs_gamma_' + io: {
        'requires': [],
        'variables': ['p' + ch, 'q' + ch],
        'ebcs': [],
        'epbcs': all_periodicY[ch],
        'lcbc': ['imv' + ch],
        'equations': {
            'eq_gamma': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                + dw_volume_dot.iV.Y%s(q%s, ls%s)
                = %e * dw_surface_integrate.iS.bY%s(q%s)"""
                % ((ch,) * 7 + (1.0/param_h, io, ch)),
            'eq_imv': 'dw_volume_dot.iV.Y%s(lv%s, p%s) = 0'
                % ((ch,) * 3),
        },
        'class': cb.CorrOne,
        '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_surface_integrate.iS.bY%s(corr1_%s)' \
                % (io, ch),
            'set_volume': 'bYp',
            'class': cb.CoefOne,
        },
    })
}

solvers = {
    'ls': ('ls.scipy_direct', {}),
    '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.
    """
    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) - $Y$, $Y_i$, ...
# 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, Pil, Pi2
variables = {
    'uc'      : ('unknown field', 'displacement', 0),
    'vc'      : ('test field',     'displacement', 'uc'),
    'Pi'      : ('parameter field', 'displacement', 'uc'),
    'Pil'     : ('parameter field', 'displacement', None),
    'Pi2'     : ('parameter field', 'displacement', None),
}

##
# Periodic boundary conditions.
if dim == 3:
    epbc = {
        '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:
    epbc = {
        '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.
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'],
'ebc's' : ['fixed_u'],
'epbc's' : 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,  

},  

}  
  
##  
# Solvers.  
solvers = {  
    'ls' : ('ls.scipy_direct', {}),  
    '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""">>>>
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  $\mathbf{E}_{ijkl}$  is expressed
using  $\mathbf{\Pi}_i$  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'],
                          file_per_var=False)
    pis = mini_app()
    print(pis)
    spause(r""">>>>
...the  $\mathbf{\Pi}_i$  operators.
['q'/other key to quit/continue...]"")>

    spause(r""">>>>
Next,  $\mathbf{E}_{ijkl}$  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'],
                          file_per_var=False)
    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:

    python postproc.py %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('d_volume.i3.Y(uc)')

```

```

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()

```

## 5.6.4 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:

$$\int_{\Omega^{(0)}} \left( \underline{\underline{S}}^{\text{eff}}(\underline{u}) + K(J-1) \underline{\underline{J}} \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV = 0 , \quad \forall \underline{v} ,$$

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 $\underline{\underline{C}} = \underline{\underline{F}}^T \underline{\underline{F}}$
$\underline{\underline{E}}(\underline{u})$	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

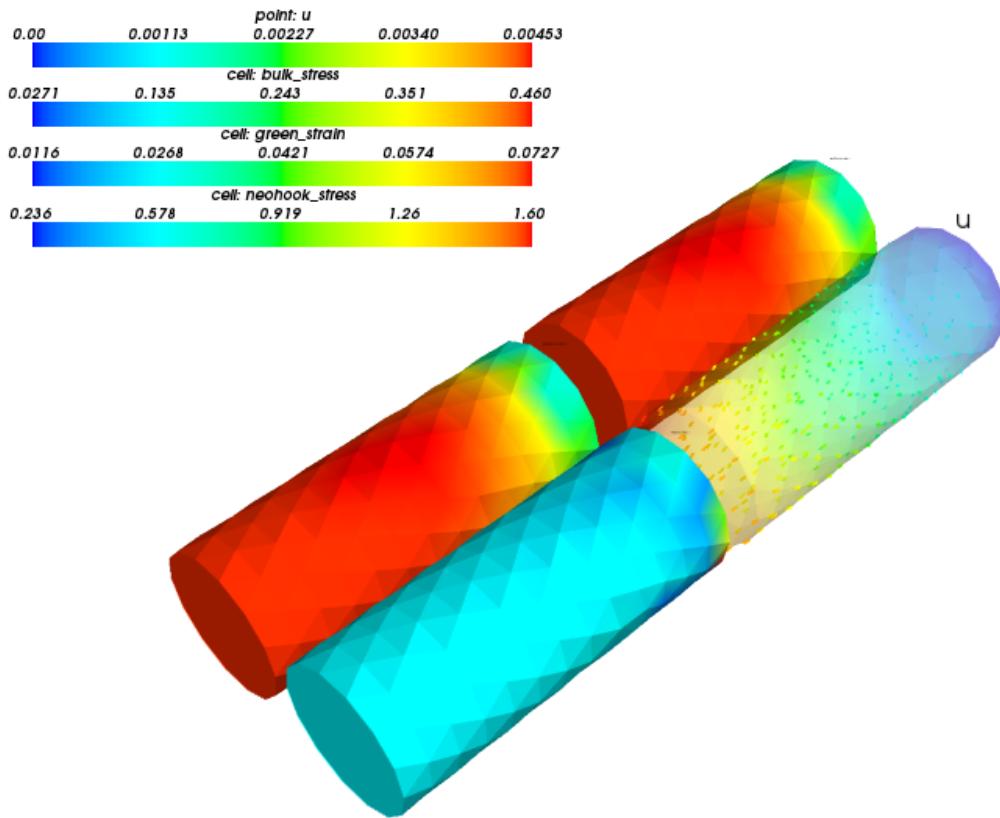
The effective stress  $\underline{\underline{S}}^{\text{eff}}(\underline{u})$  incorporates also the effects of the active fibres in two preferential directions:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} (\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1}) + \sum_{k=1}^2 \tau^k \underline{\underline{\omega}}^k .$$

The first term is the neo-Hookean term and the sum add contributions of the two fibre systems. The tensors  $\underline{\underline{\omega}}^k = \underline{d}^k \underline{d}^k$  are defined by the fibre system direction vectors  $\underline{d}^k$  (unit).

For the one-dimensional tensions  $\tau^k$  holds simply ( $k$  omitted):

$$\tau = A f_{\max} \exp \left\{ - \left( \frac{\epsilon - \varepsilon_{\text{opt}}}{s} \right)^2 \right\}, \quad \epsilon = \underline{\underline{E}} : \underline{\underline{\omega}}.$$



[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 :math:`\underline{\underline{u}} such that:

```
.. math::
    \int \Omega \underline{s} \underline{u} \cdot \left( \underline{\underline{S}} \underline{\underline{\epsilon}}(\underline{\underline{u}}) + K(J-1) \underline{\underline{C}}^{-1} \right) : \underline{\underline{\delta}} \underline{\underline{E}}(\underline{\underline{v}}) \, dV = 0
    \quad \forall \underline{\underline{v}}
```

where

```
.. list-table::
    :widths: 20 80
```

```
* - :math:`\nabla F`  
  - deformation gradient :math:`F_{ij} = \partial F_i / \partial X_j`  
* - :math:`J`  
  - :math:`\det(F)`  
* - :math:`C`  
  - right Cauchy-Green deformation tensor :math:`C = F^T F`  
* - :math:`E(\boldsymbol{u})`  
  - Green strain tensor :math:`E_{ij} = 1/2 (\partial u_i / \partial X_j + \partial u_j / \partial X_i) + (\partial u_m / \partial X_i) (\partial u_m / \partial X_j)`  
* - :math:`S(\boldsymbol{u})`  
  - effective second Piola-Kirchhoff stress tensor
```

The effective stress :math:`S(\boldsymbol{u})` incorporates also the effects of the active fibres in two preferential directions:

```
.. math::  
    S(\boldsymbol{u}) = \mu J^{(2)} (\mathbf{I} - \frac{1}{3} \text{tr}(C) \mathbf{I} + \sum_{k=1}^2 \tau^k \omega_k \omega_k)
```

The first term is the neo-Hookean term and the sum add contributions of the two fibre systems. The tensors :math:`\omega\_k` are defined by the fibre system direction vectors :math:`\mathbf{u}\_d` (unit).

For the one-dimensional tensions :math:`\tau^k` holds simply (:math:`^k` omitted):

```
.. math::  
    \tau = A f_{\max} \exp(-(\epsilon - \varepsilon_0)^2) \quad \epsilon = E : \omega_k  
    \tau = 0.5 \exp(-(\epsilon - 0.2)^2)  
  
"""  
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'  
  
vf_matrix = 0.5  
vf_fibres1 = 0.2  
vf_fibres2 = 0.3  
  
options = {  
    'nls' : 'newton',  
    'ls' : 'ls',  
    'ts' : 'ts',  
    'save_times' : 'all',  
    'post_process_hook' : 'stress_strain',  
}  
  
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',
}

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 shound be defined.
    mode : 'qp' or 'special'
        Call mode.
    which : int
        Fibre system id.
    vf : float
        Fibre system volume fraction.
    """
    if mode != 'qp': return

    fmax = 10.0
    eps_opt = 0.01
    s = 1.0

    tt = ts.nt * 2.0 * nm.pi

    if which == 0: # system 1
        fdir = nm.array([1.0, 0.0, 0.0], dtype=nm.float64)
        act = 0.5 * (1.0 + nm.sin(tt - (0.5 * nm.pi)))

    elif which == 1: # system 2
        fdir = nm.array([0.0, 1.0, 0.0], dtype=nm.float64)
        act = 0.5 * (1.0 + nm.sin(tt + (0.5 * nm.pi)))

    else:
        raise ValueError('unknown fibre system! (%d)' % which)

    fdir.shape = (3, 1)
    fdir /= nm.linalg.norm(fdir)

    print(act)

    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' : nm.tile(fdir, shape),
        'act' : nm.tile(act, shape),
    }

```

```
    return out

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'),
}

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}),
}

## # Balance of forces.
integral_1 = {
    'name' : 'i',
    'order' : 1,
}
equations = {
    'balance':
        """dw_tl_he_neohooke.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""",
}

def stress_strain(out, problem, state, extend=False):
    from sfepy.base.base import Struct, debug

    ev = problem.evaluate
    strain = ev('dw_tl_he_neohooke.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_neohooke.i.Omega( solid.mu, v, u )',
                mode='el_avg', term_mode='stress')
    out['neohooke_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

##

# Solvers etc.
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
}

solver_1 = {
    'name' : 'newton',
    'kind' : 'nls.newton',

    'i_max'      : 7,
    '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,
}

solver_2 = {
    'name' : 'ts',
    'kind' : 'ts.simple',

    't0'   : 0,
    't1'   : 1,
    'dt'   : None,
    'n_step' : 21, # has precedence over dt!
    'verbose' : 1,
}

```

## large\_deformation/balloon.py

### Description

Inflation of a Mooney-Rivlin hyperelastic balloon.

This example serves as a verification of the membrane term (`dw_tl_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 pres-

sure  $p$  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 by the 'integral\_mean\_value' linear combination condition.

Find  $\underline{u}(\underline{X})$  and a constant  $p$  such that:

- balance of forces:

$$\int_{\Omega^{(0)}} \left( \underline{\underline{S}}^{\text{eff}}(\underline{u}) - p \underline{\underline{J}} \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}; \underline{v}) \, dV + \int_{\Gamma^{(0)}} \underline{\underline{S}}^{\text{eff}}(\tilde{\underline{u}}) \delta \underline{\underline{E}}(\tilde{\underline{u}}; \tilde{\underline{v}}) h_0 \, dS = 0, \quad \forall \underline{v} \in [H_0^1(\Omega)]^3,$$

- volume conservation:

$$\int_{\Omega_0} [\omega(t) - J(u)] q \, dx = 0 \quad \forall q \in L^2(\Omega),$$

where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$\underline{J}$	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = \underline{\underline{F}}^T \underline{\underline{F}}$
$\underline{\underline{E}}(\underline{u})$	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  $\underline{\underline{S}}^{\text{eff}}(\underline{u})$  is given by:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} \left( \underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1} \right) + \kappa J^{-\frac{4}{3}} (\text{tr}(\underline{\underline{C}} \underline{\underline{I}} - \underline{\underline{C}}) - \frac{2}{6} ((\text{tr} \underline{\underline{C}})^2 - \text{tr}(\underline{\underline{C}}^2)) \underline{\underline{C}}^{-1}).$$

The  $\tilde{\underline{u}}$  and  $\tilde{\underline{v}}$  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):

```
python simple.py examples/large_deformation/balloon.py -d 'plot: True'
```

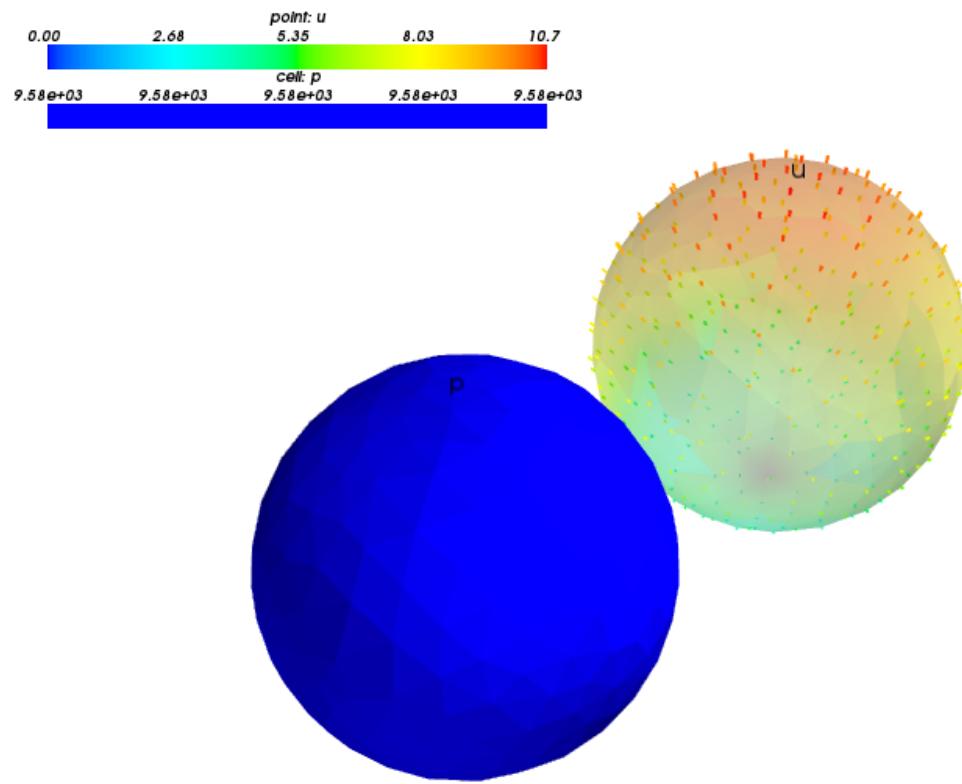
The agreement should be very good, even though the mesh is coarse.

View the results using:

```
python postproc.py unit_ball.h5 --wireframe -b -d 'u, plot_displacements, rel_scaling=1' ↵
--step=-1
```

This example uses the adaptive time-stepping solver ('ts.adaptive') with the default adaptivity function `adapt_time_step()`. Plot the used time steps by:

```
python script/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 <sfepy.terms.terms_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

$\dots$  math::

$$p = 4 \frac{h_0}{r_0} (\frac{1}{L} - \frac{1}{L^7}) (c_1 + c_2 L^2) \;,$$

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 by the `'''integral_mean_value'''` linear combination condition.

Find  $\mathbf{u}$  and a constant  $p$  such that:

- balance of forces:

```
.. math::
\int \Omega \mathbf{u} \cdot \nabla p + J \operatorname{tr}(\mathbf{C}^{-1}) : \mathbf{E} = \delta \mathbf{u} \cdot \mathbf{v} + \int \Gamma \tilde{\mathbf{u}} \cdot \mathbf{v} - h_0 \operatorname{div} \mathbf{v} = 0 ;, \quad \forall \mathbf{v} \in H^1_0(\Omega) ]^3 ;,
```

- volume conservation:

```
.. math::
\int \Omega \operatorname{div} \mathbf{u} = 0
\quad \forall q \in L^2(\Omega) ;,
```

where

```
.. list-table::
:widths: 20 80

* -  $\mathbf{F}$ 
- deformation gradient  $F_{ij} = \partial_i x_j$ 
* -  $J$ 
-  $\det(F)$ 
* -  $\mathbf{C}$ 
- right Cauchy-Green deformation tensor  $C = F^T F$ 
* -  $\mathbf{E}$ 
- Green strain tensor  $E_{ij} = \frac{1}{2} (\partial_i u_j + \partial_j u_i) + \partial_i u_j \partial_j u_i + \partial_i u_m \partial_m u_j$ 
* -  $\mathbf{S}$ 
- effective second Piola-Kirchhoff stress tensor
```

The effective stress  $\mathbf{S}$  is given by:

```
.. math::
\mathbf{S} = \mu J^{-\frac{2}{3}} (\mathbf{I} - \frac{1}{3} \operatorname{tr}(\mathbf{C}) \mathbf{C}^{-1}) + \kappa J^{-\frac{4}{3}} (\operatorname{tr}(\mathbf{C}) \mathbf{I} - \mathbf{C}) - \frac{2}{6} ((\operatorname{tr}(\mathbf{C}))^2 - \operatorname{tr}(\mathbf{C}^2)) \mathbf{C}^{-1} ;.
```

The  $\tilde{\mathbf{u}}$  and  $\tilde{\mathbf{v}}$  variables correspond to  $\mathbf{u}$ ,  $\mathbf{v}$ , 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)::

```
python simple.py examples/large_deformation/balloon.py -d 'plot: True'
```

The agreement should be very good, even though the mesh is coarse.

View the results using::

```
python postproc.py unit_ball.h5 --wireframe -b -d'u,plot_displacements,rel_scaling=1
→' --step=-1
```

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::

```
python script/plot_times.py unit_ball.h5
"""
from __future__ import absolute_import
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

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):
    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/V0 from relative stretch r/r0 of a ball.
    """
    return nm.power(rel_stretch, 3.0)

def get_rel_stretch(rel_volume):
    """
    Get relative stretch r/r0 from relative volume V/V0 of a ball.
    """

```

```
    return nm.power(rel_volume, 1.0/3.0)

def get_balloon_pressure(rel_stretch, h0, r0, c1, c2):
    """
    Rivlin 1952 + Dumais:

    P = 4*h0/r0 * (1/L-1/L^7).* (C1+L^2*C2)
    """
    L = rel_stretch
    p = 4.0 * h0 / r0 * (1.0/L - 1.0/L**7) * (c1 + c2 * L**2)

    return p

def plot_radius(problem, state):
    import matplotlib.pyplot as plt

    from sfepy.postprocess.time_history import extract_time_history

    ths, ts = extract_time_history('unit_ball.h5', 'p e 0')

    p = ths['p'][0]
    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['text.fontsize'] = 16

    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')

    fig = plt.gcf()
    fig.savefig('balloon_pressure_stretch.pdf')

    plt.show()

def define(plot=False):
    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',
        '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),
    'pressure': (nm.float64, 1, 'Omega', 0),
}

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'),
}
}

ebcs = {
    'fix1' : ('Ax1', {'u.all' : 0.0}),
    'fix2' : ('Ax2', {'u.[0, 1]' : 0.0}),
    'fix3' : ('Equator', {'u.1' : 0.0}),
}
}

lcbc = {
    '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_volume_dot.2.Omega(q, omega)""",
    }

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max'      : 6,
        'eps_a'       : 1e-4,
        'eps_r'       : 1e-8,
        'macheps'     : 1e-16,
        'lin_red'     : 1e-2,
        'ls_red'      : 0.5,
        'ls_red_warp': 0.1,
        'ls_on'       : 100.0,
        'ls_min'      : 1e-5,
        'check'       : 0,
        'delta'       : 1e-6,
        'is_plot'     : False,
        'problem'     : 'nonlinear',
    }),
    '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,

        '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):
        """
        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'),
    '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))
        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='%(prog)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)

# 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/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:

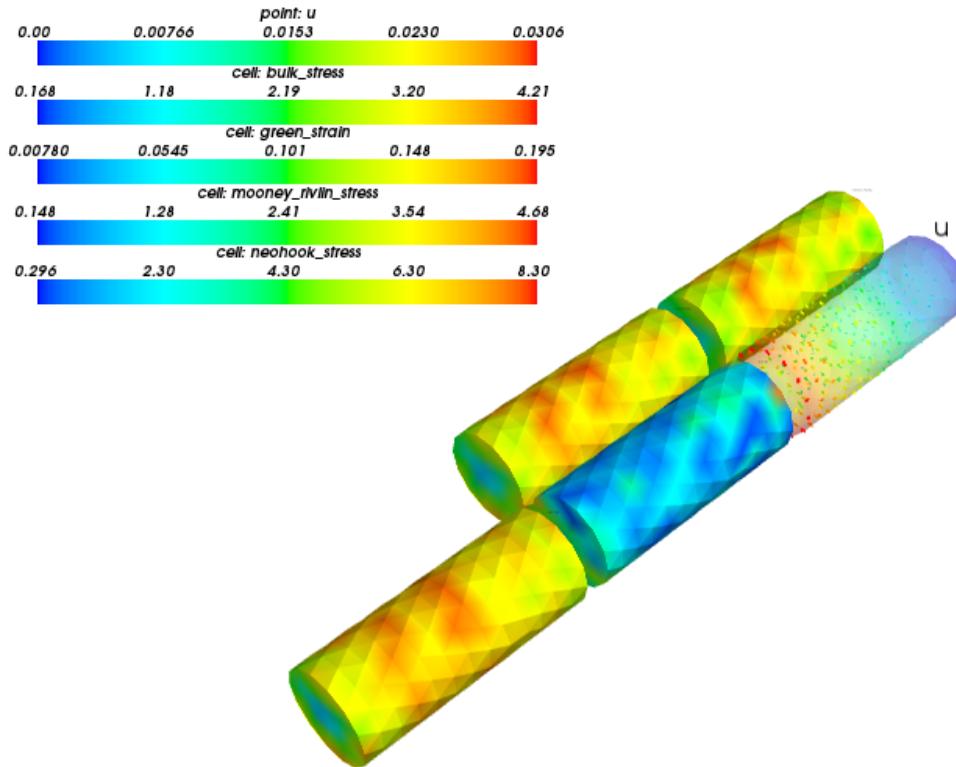
$$\int_{\Omega^{(0)}} \left( \underline{\underline{S}}^{\text{eff}}(\underline{u}) + K(J - 1) \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV = 0 , \quad \forall \underline{v} ,$$

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 $C = F^T F$
$\underline{\underline{E}}(\underline{u})$	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:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} (\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1}) + \kappa J^{-\frac{4}{3}} (\text{tr}(\underline{\underline{C}} \underline{\underline{I}} - \underline{\underline{C}}) - \frac{2}{6} ((\text{tr} \underline{\underline{C}})^2 - \text{tr}(\underline{\underline{C}}^2)) \underline{\underline{C}}^{-1}) .$$



[source code](#)

```
# -*- coding: utf-8 -*-
"""
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:`\nabla u` such that:

```
.. math::
   \int \Omega \nabla u : \nabla u - K(J-1) J \det(C)^{-1} = 0
   \quad \forall v;
```

where

```
.. list-table::
   :widths: 20 80

   * - :math:`\nabla u` - deformation gradient :math:`F_{ij} = \partial u_i / \partial X_j`
   * - :math:`J` - :math:`\det(F)`
   * - :math:`C` - right Cauchy-Green deformation tensor :math:`C = F^T F`
   * - :math:`\nabla E` - Green strain tensor :math:`E_{ij} = 1/2 (\partial u_i / \partial X_j + \partial u_j / \partial X_i) + (\partial u_m / \partial X_i) (\partial u_m / \partial X_j)`
   * - :math:`S` - effective second Piola-Kirchhoff stress tensor
```

The effective stress :math:`\nabla S` is given by:

```
.. math::
   \nabla S = \mu J^{-1/3} (\nabla I
   - \frac{3}{J} \text{tr}(C) C^{-1}
   + \kappa J^{-4/3} (\text{tr}(C) \nabla I - \nabla C
   - \frac{6}{J} (\text{tr}(C))^2 \nabla I) C^{-1})
   \quad .
```

```
"""
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
    }
}

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_neohookean.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_neohookean.i.Omega( solid.mu, v, u )',
            mode='el_avg', term_mode='stress')
out['neohookean_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_neohookean.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,
}

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:

$$\int_{\Omega^{(0)}} \underline{\underline{S}}^{\text{eff}}(\underline{u}, p) : \underline{\underline{E}}(\underline{v}) \, dV = 0 , \quad \forall \underline{v} ,$$

$$\int_{\Omega^{(0)}} q (J(\underline{u}) - 1) \, dV = 0 , \quad \forall q .$$

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  $\lambda$  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:

$$\mu = 2 C_{(10)} , \\ \kappa = 2 C_{(01)} .$$

## Usage Examples

Default options:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py -p
```

Using different mesh fineness:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \\\n--shape "5, 5, 5"
```

Different dimensions of the computational domain:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \\\n--dims "2, 1, 3"
```

Different length of time interval and/or number of time steps:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \\\n-t 0,15,21
```

Use higher approximation order (the `-t` option to decrease the time step is required for convergence here):

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \\\n--order 2 -t 0,2,21
```

Change material parameters:

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py -m 2,1
```

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)} \cdot \left( \overline{I}_1 - 3 \right) \\
        + C_{(01)} \cdot \left( \overline{I}_2 - 3 \right) ;,
```

where :math:`\overline{I}\_1` and :math:`\overline{I}\_2` are the first and second main invariants of the deviatoric part of the right

*Cauchy-Green deformation tensor :math:`\mathbf{C}`. The coefficients :math:`C\_{(10)}` and :math:`C\_{(01)}` are material parameters.*

*Components of the second Piola-Kirchhoff stress are in the case of an incompressible material*

```
.. math::
    S_{ij} = 2 \cdot \operatorname{pdiff}{W}{C_{ij}} - p \cdot F^{-1}_{ik} \cdot F^{-T}_{kj} ;,
```

*where :math:`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 :math:`\mathbf{u}` and pressure field :math:`p` such that:*

```
.. math::
    \int_{\Omega} \mathbf{suz} \cdot \mathbf{eff}(\mathbf{u}, p) : \mathbf{E}(\mathbf{v}) \operatorname{d}V = 0
    \quad \forall \mathbf{v} ;,
    \int_{\Omega} q \cdot (J(\mathbf{u}) - 1) \operatorname{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 \cdot \left( C_{(10)} + \frac{C_{(01)}}{\lambda} \right) \lambda +
    \left( \lambda^2 - \frac{1}{\lambda} \right) ;,
```

*where :math:`\lambda = l/l\_0` is the prescribed stretch (:math:`l\_0` and :math:`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 :math:`\lambda` 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 (`:class:`NeoHookeanTLTerm``, `<sfepy.terms.terms_hyperelastic_t1.NeoHookeanTLTerm>`, `:class:`MooneyRivlinTLTerm``, `<sfepy.terms.terms_hyperelastic_t1.MooneyRivlinTLTerm>`) and the ones used in this example is:*

```
.. math::
    \mu = 2 \cdot C_{(10)} ;
    \kappa = 2 \cdot C_{(01)} .
```

### Usage Examples

-----

Default options::

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py
```

To show a comparison of stress against the analytic formula::

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py -p
```

Using different mesh fineness::

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \
--shape "5, 5, 5"
```

Different dimensions of the computational domain::

```
$ python examples/large_deformation/hyperelastic_tl_up_interactive.py \
--dims "2, 1, 3"
```

Different length of time interval and/or number of time steps::

```
$ python 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 examples/large_deformation/hyperelastic_tl_up_interactive.py \
--order 2 -t 0,2,21
```

Change material parameters::

```
$ python 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
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').draggable()
    ax_stress.set_ylabel(r'true stress $\mathrm{[Pa]}$')
    ax_stress.grid()

    ax_difference.set_ylabel(r'difference in true stress $\mathrm{[Pa]}$')
    ax_difference.set_xlabel(r'stretch $\mathrm{[-]}$')
    ax_difference.grid()
    plt.show()

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_neohooke.%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_10 = problem.evaluate(
    'dw_tl_he_neohooke.%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_10 + stress_01 + stress_p

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

global_stress.append(stress[0, 0, 0])
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})
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_volume_integrate(q)',  
integral, omega, q=q)  
  
eq_balance = Equation('balance', term_neohookean+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(ebcos=ebcos)  
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:  
    plot_graphs(  
        material_parameters, axial_stress, axial_displacement,  
        undeformed_length=dims[0])  
  
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(
    '-p', '--plot', action='store_true', default=False,
    help='Whether to plot a comparison with analytical formula.')
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)

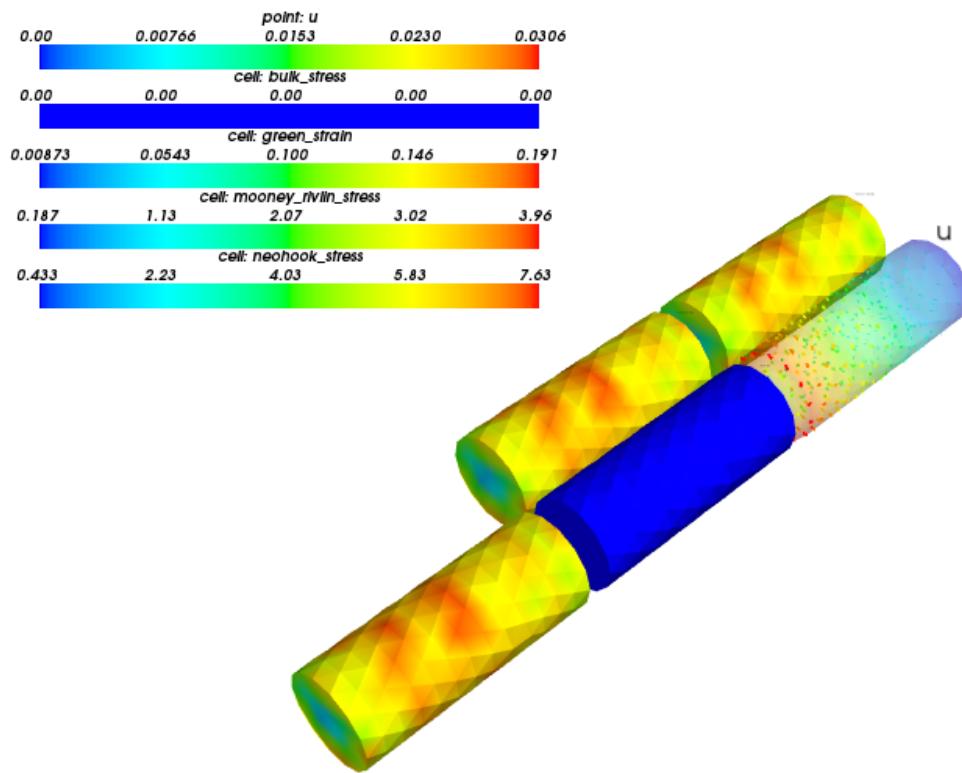
```

## 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'],
    'output_dir': 'output',
    'post_process_hook': 'stress_strain',
}
```

```

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

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_neochook.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""",
}

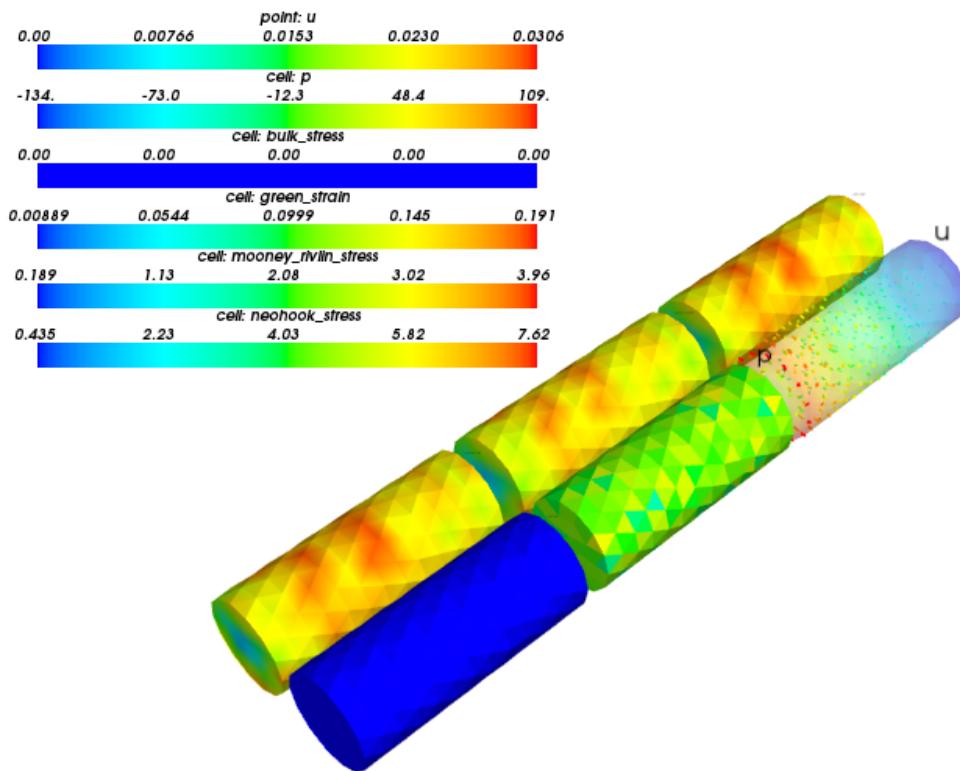
## 
# Solvers etc.
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 25,
        'eps_a': 1e-8,
        'eps_r': 1.0,
        'macheps': 1e-16,
        'lin_red': 1e-2, # Linear system error < (eps_a * lin_red).
        'ls_red': 0.1,
        'ls_red_warp': 0.001,
        'ls_on': 1.1,
        'ls_min': 1e-5,
        'check': 0,
        '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\_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,
    '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

    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_neohookean.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_neohookean.3.Omega( solid.mu, v, u )',
            mode='el_avg', term_mode='stress')
out['neohookean_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_neohookean.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,
        '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\_t1.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)

$$\int_{\Omega^{(0)}} \left( \underline{\underline{S}}^{\text{eff}} - p \underline{\underline{J}} \underline{\underline{C}}^{-1} \right) : \delta \underline{\underline{E}}(\underline{v}) \, dV + \int_{\Gamma_0^{(0)}} \underline{\underline{\nu}} \cdot \underline{\underline{F}}^{-1} \cdot \underline{\underline{\sigma}} \cdot \underline{v} J \, dS = 0 , \quad \forall \underline{v} ,$$

(mass balance equation (perfusion))

$$\int_{\Omega^{(0)}} q J(\underline{u}) + \int_{\Omega^{(0)}} \underline{\underline{K}}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial X} \frac{\partial p}{\partial X} = \int_{\Omega^{(0)}} q J(\underline{u}^{(n-1)}) , \quad \forall q ,$$

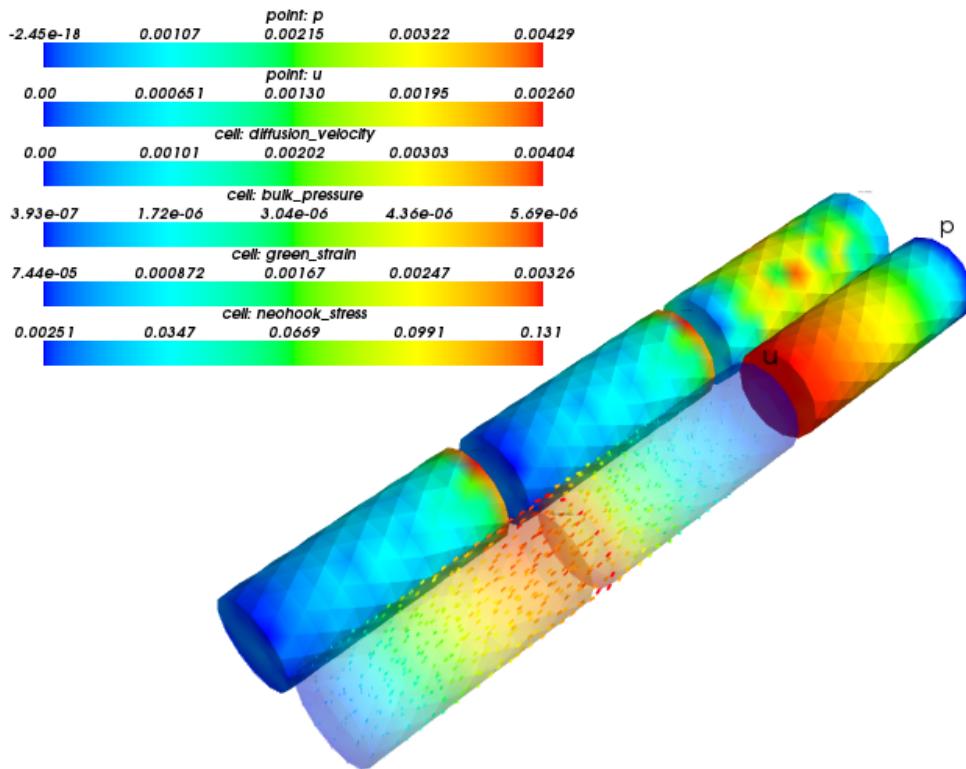
where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$\underline{\underline{J}}$	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = \underline{\underline{F}}^T \underline{\underline{F}}$
$\underline{\underline{E}}(\underline{u})$	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:

$$\underline{\underline{S}}^{\text{eff}}(\underline{u}) = \mu J^{-\frac{2}{3}} (\underline{\underline{I}} - \frac{1}{3} \text{tr}(\underline{\underline{C}}) \underline{\underline{C}}^{-1}) .$$

The linearized deformation-dependent permeability is defined as  $\underline{\underline{K}}(\underline{u}) = J \underline{\underline{F}}^{-1} \underline{k} f(J) \underline{\underline{F}}^{-T}$ , where  $\underline{u}$  relates to the previous time step ( $n - 1$ ) and  $f(J) = \max \left( 0, \left( 1 + \frac{(J-1)}{N_f} \right)^2 \right)$  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:`\mathbf{u}` such that:

(equilibrium equation with boundary tractions)

.. math::
    \int_{\Omega} (\mathbf{u}^T \mathbf{S} \mathbf{u} - p J \mathbf{u}^T \mathbf{C}^{-1} \mathbf{u})
    + \int_{\Gamma_0} \mathbf{u}^T \mathbf{\nu} \mathbf{\Gamma}^{-1} \mathbf{F}^{-1} \mathbf{\sigma} \mathbf{u} d\Gamma
    = 0
    ;, \quad \forall \mathbf{v} ;,

(mass balance equation (perfusion))

.. math::
    \int_{\Omega} q J(\mathbf{u}) + \int_{\Omega} K(\mathbf{u}) \mathbf{u}^T \mathbf{u} \mathbf{u}^T \mathbf{u} = p \partial_q q \mathbf{X} \partial_p p \mathbf{X}

```

```
= \int_1\{\Omega\backslash\text{suz}\}{} q\ J(\text{ul}\{u\}\backslash\text{sum})  
\;, \quad \forall q\ \;,
```

where

```
.. list-table::  
    :widths: 20 80  
  
    * - :math:`\text{ull}\{F\}`  
      - 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 (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)^2\right)` 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  
  
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,
}

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.il.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.il.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.il.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.il.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,  
    '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 shound 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

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,),
}

```

## 5.6.5 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 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 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 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 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 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
```

[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 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 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 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 examples/linear_elasticity/dispersion_analysis.py meshes/2d/square_2m.mesh --
→solver-conf="kind='eig.qepr', 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 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
"""
from __future__ import absolute_import
import os
import sys
sys.path.append('.')
import gc
import functools
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
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 TimeStepper
from sfepy.linalg.utils import output_array_stats, max_diff_csr

def apply_units(pars, unit_multipliers):
    new_pars = apply_unit_multipliers(pars,
                                      ['stress', 'one', 'density',
                                       'stress', 'one', 'density'],
                                      unit_multipliers)
    return new_pars

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 define(filename_mesh, pars, approx_order, refinement_level, solver_conf,
          plane='strain', post_process=False):
    io = MeshIO.any_from_filename(filename_mesh)
    bbox = io.read_bounding_box()
    dim = bbox.shape[1]
```

```

size = (bbox[1] - bbox[0]).max()

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),
}

young1, poisson1, density1, young2, poisson2, density2 = pars
materials = {
    'm' : ({
        'D' : {'Y1' : stiffness(dim, young=young1, poisson=poisson1,
                               plane=plane),
                'Y2' : stiffness(dim, young=young2, poisson=poisson2,
                               plane=plane)},
        'density' : {'Y1' : density1, 'Y2' : 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], 1e-8))

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:
    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(1e-8 * size)
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_volume_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_volume_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_volume_integrate_mat.2.Omega(m.density, u)',
                          mode='el_avg', copy_materials=False, verbose=False)

    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_volume_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_volume_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):
    stepper = TimeStepper(rng[0], rng[1], dt=None, n_step=rng[2])
    return stepper

def save_eigenvectors(filename, svecs, wmag, wdir, pb):
    if svecs is None: return

    variables = pb.get_variables()
    # Make full eigenvectors (add DOFs fixed by boundary conditions).
    vecs = nm.empty((variables.di.ptr[-1], svecs.shape[1]),
                    dtype=svecs.dtype)
    for ii in range(svecs.shape[1]):
        vecs[:, ii] = variables.make_full_vec(svecs[:, ii])

    # Save the eigenvectors.
    out = {}
    state = pb.create_state()

    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]):
        state.set_full(vecs[:, ii])
        aux = state.create_output_dict()
        aux2 = {}
        pp(aux2, pb, state, 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):
    """
    Assemble the blocks of dispersion eigenvalue problem matrices.
    """
    define_problem = functools.partial(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)

    conf = ProblemConf.from_dict(define_problem(), 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).
    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
    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 * scaling0)
    output('1. Brillouin zone size with applied unit multipliers:', bzone)

    pb.time_update()
    pb.update_materials()

    # Assemble the matrices.
    mtxs = {}
    for key, eq in pb.equations.iteritems():
        mtxs[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, mtxs
```

```

def setup_n_eigs(options, pb, mtxs):
    """
    Setup the numbers of eigenvalues based on options and numbers of DOFs.
    """
    solver_n_eigs = n_eigs = options.n_eigs
    n_dof = mtxs['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(mtxs, val, mode, pb):
    """
    Build the matrices of the dispersion eigenvalue problem.
    """
    if mode == 'omega':
        mtx_a = mtxs['K'] + val**2 * mtxs['S'] + val * mtxs['R']
        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, mode, wdir, bzone, pb, mtxs,
                         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 mode == 'omega':
        omegas = nm.sqrt(eigs)

        output('eigs, omegas:')
        for ii, om in enumerate(omegas):
            output('{:>3}. {:.10e}, {:.10e}'.format(ii, eigs[ii], om))

        out = tuple(eigs) + 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:
    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'
        '[default: %(default)s]',
    'conf' :
        'if given, an alternative problem description file with apply_units() and'
        'define() functions [default: %(default)s]',
    '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]',
    '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',
'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='young1,poisson1,density1'
                           ',young2,poisson2,density2',
                           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('--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('--range', metavar='start,stop,count',
                           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'),
                 combined=options.silent == False)

if options.conf is not None:
    mod = import_file(options.conf)

else:
    mod = sys.modules['__main__']

apply_units = mod.apply_units
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

options.pars = [float(ii) for ii in options.pars.split(',')]

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)

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(options.pars, options.unit_multipliers)
output('material parameters with applied unit multipliers:')
output(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:
    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) / (options.n_eigs - 1)))
plot_kw_args = {'color' : get_color(ii), 'ls' : '', 'marker' : 'o'}
for ii in range(options.n_eigs)

log_names = []
log_plot_kw_args = []
if options.log_std_waves:
    std_wave_fun, log_names, log_plot_kw_args = get_std_wave_fun(
        pb, options)

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)

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_kw_args=[plot_kw_args, plot_kw_args + log_plot_kw_args],
            formats=[['{:5e}'] * options.n_eigs,
                     ['{:5e}' * (options.n_eigs + len(log_names))],
            yscales=['linear', 'linear'],
            xlabel=[r'$\kappa$', r'$\lambda_i$'],
            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)

for iv, wmag in stepper:
    output('step %d: wave vector %s' % (iv, wmag * 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, options.mode,
        wdir, bzone, pb, mtxs, std_wave_fun=std_wave_fun
    )
    log(*out, x=[wmag, wmag])

    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'$\kappa_{\text{}}$' % ii for ii in range(options.n_eigs)]
               + log_names],
              plot_kw_args=[plot_kw_args + log_plot_kw_args],
              formats=[['{:e}' * (options.n_eigs + len(log_names))],
                       'linear'],
              xlabel=r'$\omega$',
              ylabel='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))

    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, options.mode,
        wdir, bzone, pb, mtxs, 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:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \sum_{i=1}^4 \int_{\Gamma_i} \underline{v} \cdot f^i(d(\underline{u})) \underline{n}^i = 0 ,$$

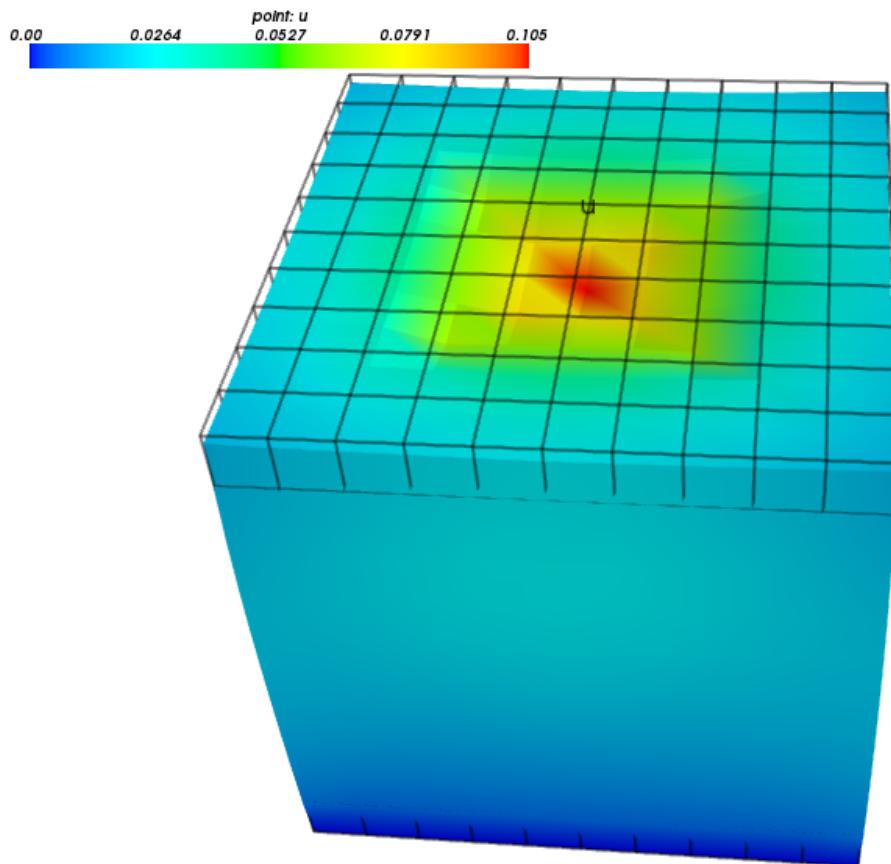
where

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

## 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 indentor.

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

```
.. math::
\int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) + \sum_{i=1}^4 \int_{\Gamma_i} \mathbf{u} \cdot \mathbf{f}^i(d(\mathbf{u})) \mathbf{n}^i = 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 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 __future__ import absolute_import
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),
        'dw_contact_plane': 1e-10
    })
}
```

```
},),
'cp0' : ({
    '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]],
},),
'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)
= 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

    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 la.combine(zip(*bb))]

    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()

if __name__ == '__main__':
    main()

```

## linear\_elasticity/elastic\_contact\_sphere.py

### Description

Elastic contact sphere simulating an indentation test.

Find  $\underline{u}$  such that:

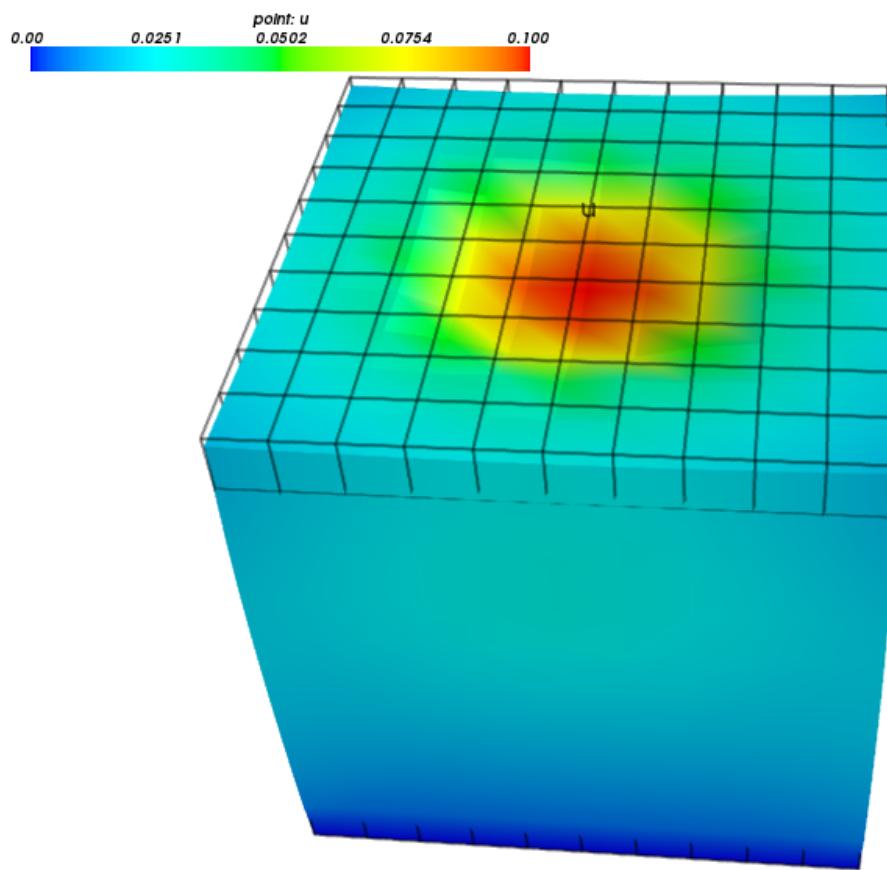
$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}(\underline{u}) = 0 ,$$

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.



[source code](#)

```
r"""
Elastic contact sphere simulating an indentation test.

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

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}(\underline{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],
        '.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

    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 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 la.combine(zip(*bb))]

    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:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = - \int_{\Gamma_{bottom}} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \quad \forall \underline{v},$$

$$\underline{u} = 0 \text{ on } \Gamma_{left},$$

$$u_1 = u_2 = 0 \text{ on } \Gamma_{right},$$

$$\underline{u}(\underline{x}) = \underline{u}(\underline{y}) \text{ for } \underline{x} \in \Gamma_{bottom}, \underline{y} \in \Gamma_{top}, y = P_1(\underline{x}),$$

$$\underline{u}(\underline{x}) = \underline{u}(\underline{y}) + a(\underline{y}) \text{ for } \underline{x} \in \Gamma_{near}, \underline{y} \in \Gamma_{far}, y = P_2(\underline{x}),$$

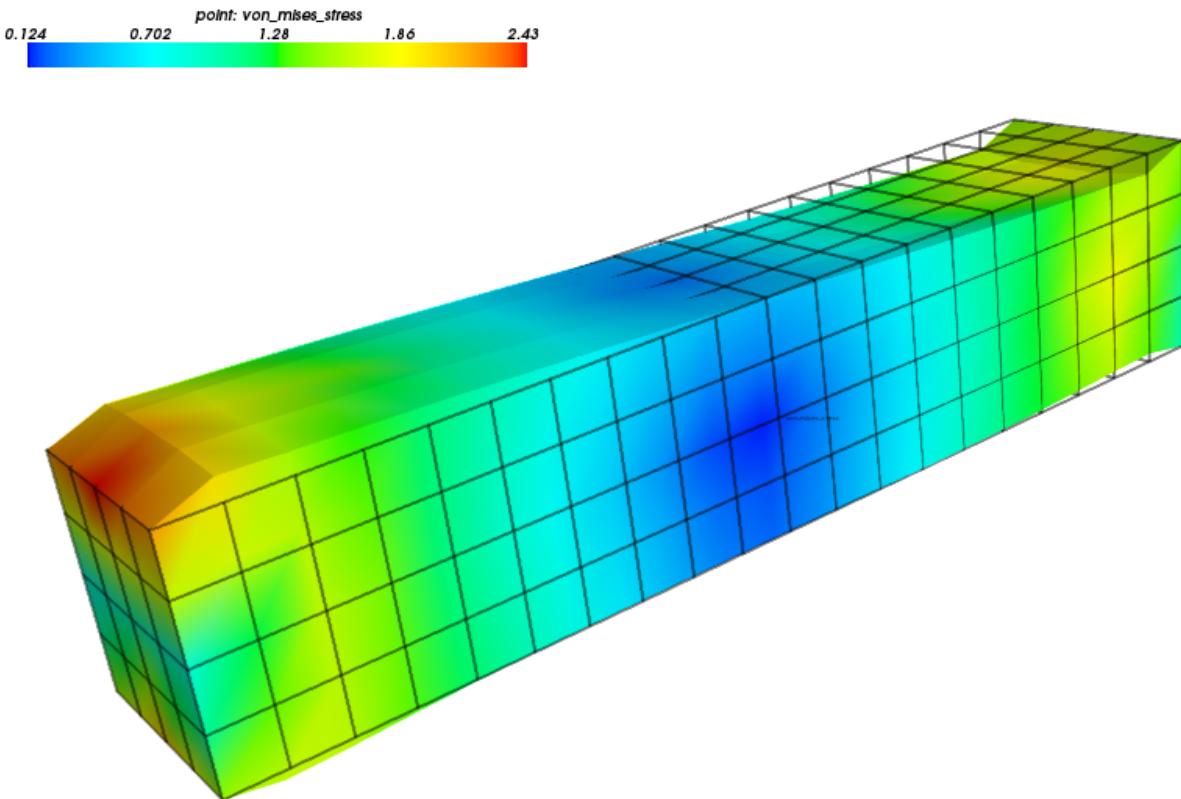
where

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl},$$

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:

```
$ ./postproc.py block.vtk --wireframe -b --only-names=u -d'u,plot_displacements,rel_
scaling=1,color_kind="scalars",color_name="von_mises_stress"
```



[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:`\underline{u}\{u\}` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\underline{u}\{v\}) e_{kl}(\underline{u}\{u\})
    = - \int_{\Gamma_{bottom}} \underline{u}\{v\} \cdot \nabla \underline{u}\{\sigma\} \cdot \underline{n}
    ;, \quad \forall \underline{u}\{v\} ;,
    \underline{u}\{u\} = 0 \text{ } \boxed{\text{on } \Gamma_{left}} ;,
    u_1 = u_2 = 0 \text{ } \boxed{\text{on } \Gamma_{right}} ;,
    \underline{u}\{u\}(\underline{u}\{x\}) = \underline{u}\{u\}(\underline{u}\{y\}) \boxed{\text{for } \underline{u}\{x\} \in \Gamma_{bottom}, \underline{u}\{y\} \in \Gamma_{top}},
    \underline{u}\{y\} = P_1(\underline{u}\{x\}) ;,
    \underline{u}\{u\}(\underline{u}\{x\}) = \underline{u}\{u\}(\underline{u}\{y\}) + a(\underline{u}\{y\}) \boxed{\text{for } \underline{u}\{x\} \in \Gamma_{near}, \underline{u}\{y\} \in \Gamma_{far}},
    \underline{u}\{y\} = P_2(\underline{u}\{x\}) ;,
```

where

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

and the traction :math:`\nabla \underline{u}\{\sigma\} \cdot \underline{n} = \bar{p} \nabla I` is given in terms of traction pressure :math:`\bar{p}`. The function :math:`a(\underline{u}\{y\})` is given (the shift), :math:`P\_1` and :math:`P\_2` are the periodic coordinate mappings.

View the results using::

```
$ ./postproc.py block.vtk --wireframe -b --only-names=u -d'u,plot_displacements,rel_
scaling=1,color_kind="scalars",color_name="von_mises_stress"
"""
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

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 = {
    '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:

$$\int_{\Omega} \rho \underline{v} \frac{\partial^2 \underline{u}}{\partial \underline{v}^2} + \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = 0 , \quad \forall \underline{v} ,$$

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.

## Usage Examples

Run with the default settings (the Newmark method, 3D problem, results stored in `output/ed/`):

```
python simple.py examples/linear_elasticity/elastodynamic.py
```

Solve using the Bathe method:

```
python simple.py examples/linear_elasticity/elastodynamic.py -O "ts='tsb'"
```

View the resulting deformation using:

- color by  $\underline{u}$ :

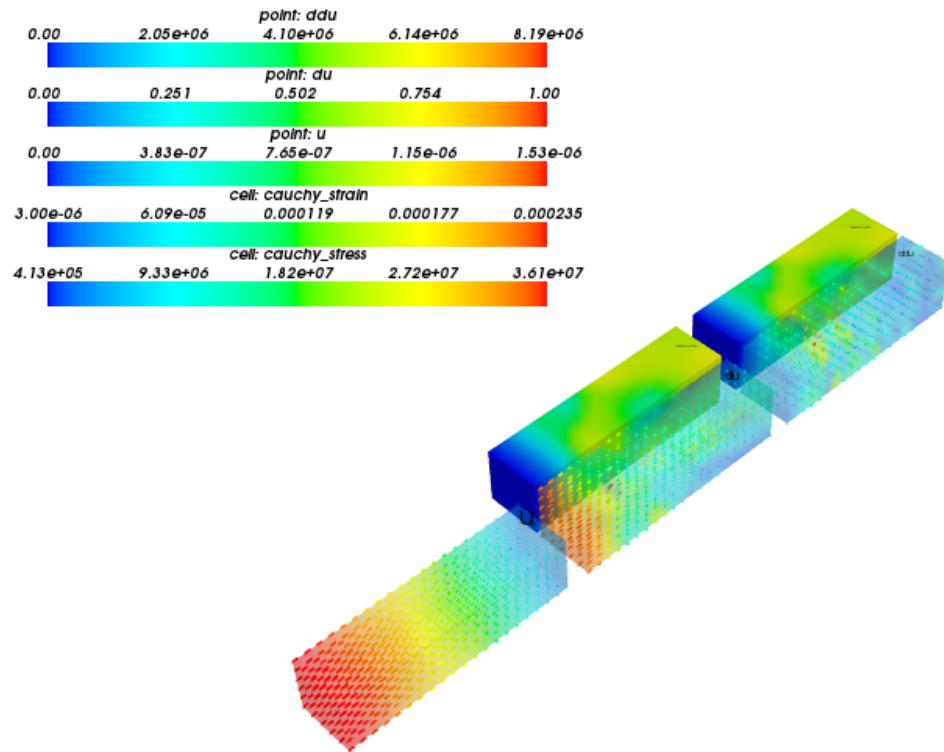
```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,  
plot_displacements,rel_scaling=1e3'
```

- color by  $e(\underline{u})$ :

```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,  
plot_displacements,rel_scaling=1e3,color_kind="tensors",color_name="cauchy_  
strain"'
```

- color by  $\underline{\underline{u}}$ :

```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,  
plot_displacements,rel_scaling=1e3,color_kind="tensors",color_name="cauchy_  
stress"'
```



[source code](#)

```
r"""
The linear elastodynamics solution of an iron plate impact problem.
```

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

```
.. math::
    \int_{\Omega} \rho \mathbf{u} \cdot \nabla \mathbf{u} - \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}
```

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.*

#### Usage Examples

---

*Run with the default settings (the Newmark method, 3D problem, results stored in ``output/ed/``)::*

```
python simple.py examples/linear_elasticity/elastodynamic.py
```

*Solve using the Bathe method::*

```
python simple.py examples/linear_elasticity/elastodynamic.py -O "ts='tsb'"
```

*View the resulting deformation using:*

*- color by :math:`\|u\|`::*

```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,
→plot_displacements,rel_scaling=1e3'
```

*- color by :math:`\|e\|(\|u\|)`::*

```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,
→plot_displacements,rel_scaling=1e3,color_kind="tensors",color_name="cauchy_strain"'
```

*- color by :math:`\|\sigma\|(\|u\|)`::*

```
python postproc.py output/ed/user_block.h5 -b --wireframe --only-names=u -d 'u,
→plot_displacements,rel_scaling=1e3,color_kind="tensors",color_name="cauchy_stress"'"
"""
```

```
from __future__ import absolute_import
```

```
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
```

```
plane = 'strain'
```

```
dim = 3
```

```
# Material parameters.
```

```
E = 200e9
```

```
nu = 0.3
```

```
rho = 7800.0
```

```
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)
```

```
# Initial velocity.
```

```
v0 = 1.0
```

```
# Mesh dimensions and discretization.
d = 2.5e-3
if dim == 3:
    L = 4 * d
    dims = [L, d, d]

    shape = [21, 6, 6]
    #shape = [101, 26, 26]

else:
    L = 2 * d
    dims = [L, 2 * d]

    shape = [61, 61]
    # shape = [361, 361]

# Element size.
H = L / (shape[0] - 1)

# Time-stepping parameters.
# Note: the Courant number C0 = dt * cl / H
dt = H / cl # C0 = 1

if dim == 3:
    t1 = 0.9 * L / cl

else:
    t1 = 1.5 * d / 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, dofs=None)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
                                   data=stress, dofs=None)
```

```

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

integrals = {
    'i' : 2,
}

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'),
}
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'}),,
}

equations = {
    'balance_of_forces' :
    """dw_volume_dot.i.Omega(solid.rho, ddv, ddu)
    + dw_zero.i.Omega(dv, du)
    + dw_lin_elastic.i.Omega(solid.D, v, u) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {
        'use_presolve' : True,
    }),
    'ls-i' : ('ls.petsc', {
        'method' : 'cg',
        'precond' : 'icc',
        'i_max' : 150,
        'eps_a' : 1e-32,
        'eps_r' : 1e-8,
        'verbose' : 2,
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-6,
        'eps_r' : 1e-6,
    }),
    'tsvv' : ('ts.velocity_verlet', {
        # Explicit method -> requires at least 10x smaller dt than the other
        # time-stepping solvers.
        '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,
    'verbose' : 1,
}),
},
}

options = {
    'ts' : 'tsn',
    # 'ts' : 'tsb',
    'nls' : 'newton',
    # 'ls' : 'ls-i',
    'ls' : 'ls',
    'save_times' : 20,
    'active_only' : False,
    'output_format' : 'h5',
    'output_dir' : 'output/ed',
    'post_process_hook' : 'post_process',
}
}

```

## linear\_elasticity/its2D\_1.py

### Description

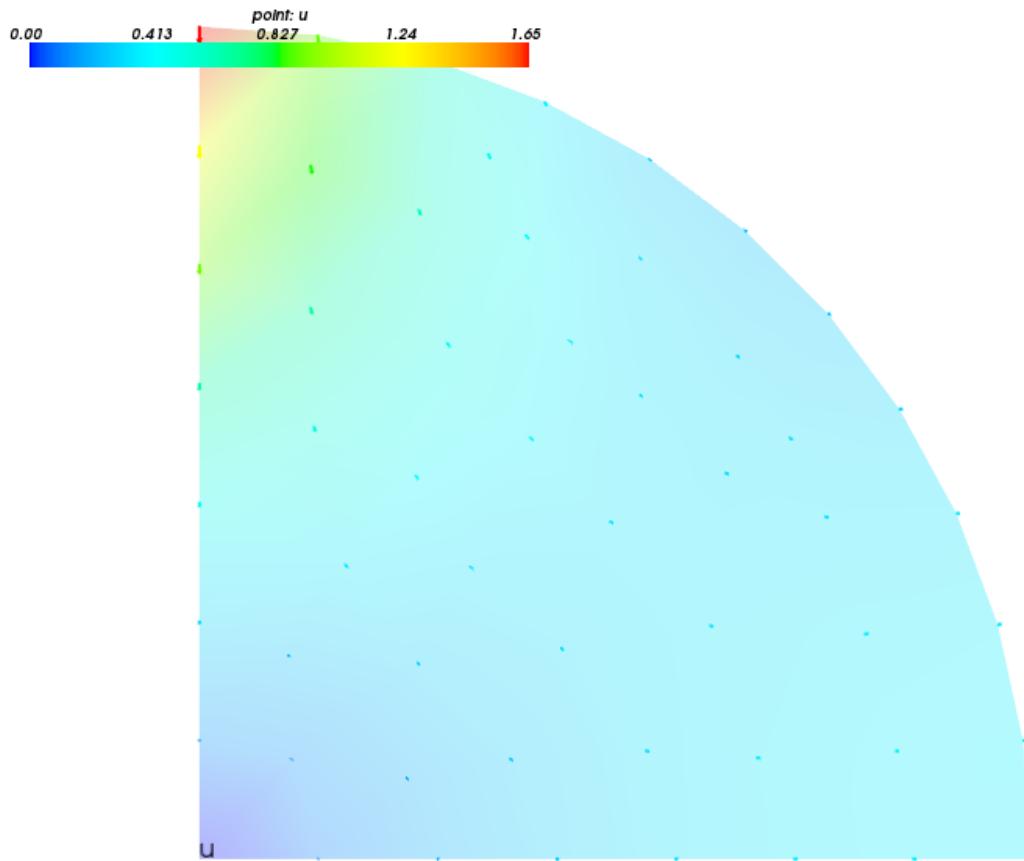
Diametrically point loaded 2-D disk. See *Primer*.

Find  $\underline{u}$  such that:

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

where

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



[source code](#)

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

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

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\underline{u}) e_{kl}(\underline{v}) = 0
    \quad \forall \underline{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 = 2000.0 # 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,
        'eps_a' : 1e-6,
    }),
}

```

**linear\_elasticity/its2D\_2.py****Description**

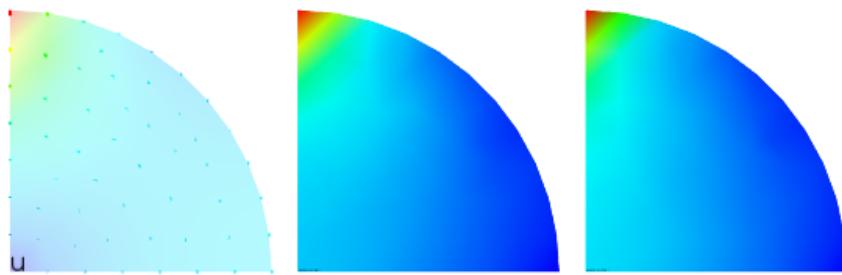
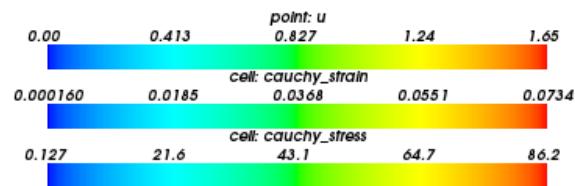
Diametrically point loaded 2-D disk with postprocessing. See [Primer](#).

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:`\underline{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \underline{e}_{ij}(\underline{v}) \underline{e}_{kl}(\underline{u}) = 0
    \quad \forall \underline{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 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

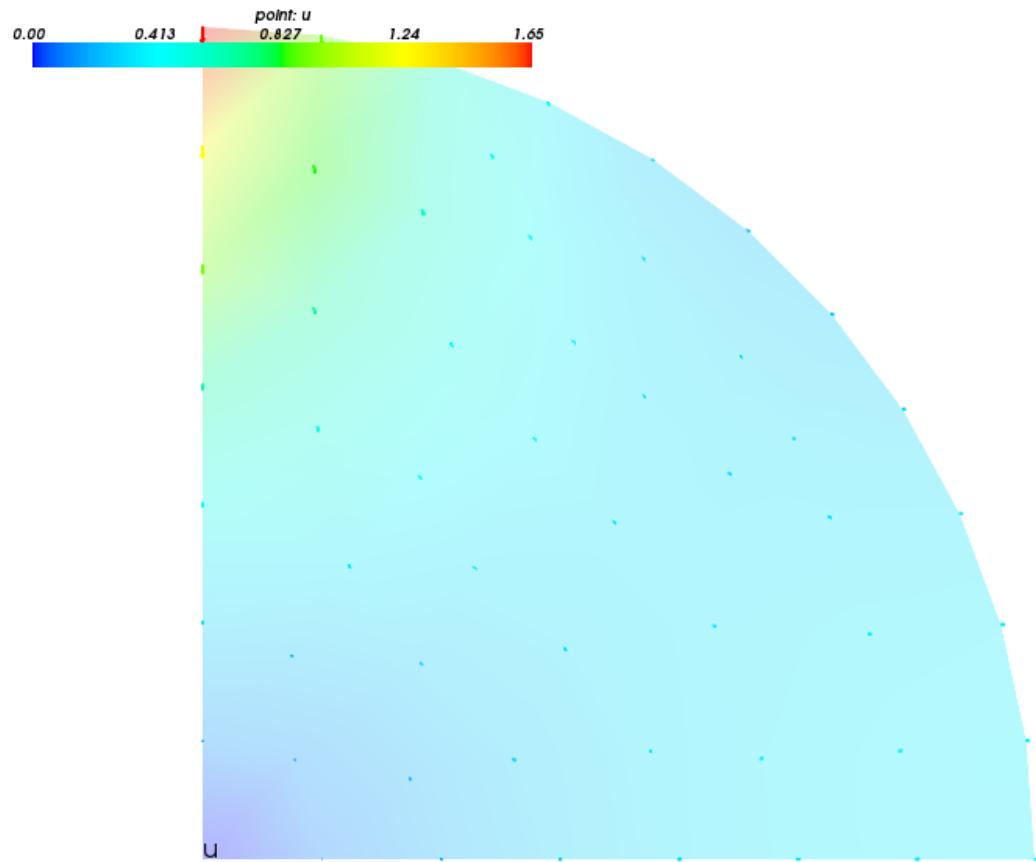
Diametrically point loaded 2-D disk with nodal stress calculation. See [Primer](#).

Find  $u$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(v) e_{kl}(u) = 0, \quad \forall v,$$

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:`\mathbf{u}` such that:

.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(v) \epsilon_{kl}(u) = 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 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')
    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](#).

Use it as follows (assumes running from the sfepy directory; on Windows, you may need to prefix all the commands with “python ” and remove “./”):

1. solve the problem:

```
./simple.py examples/linear_elasticity/its2D_4.py
```

2. optionally, view the results:

```
./postproc.py its2D.h5 -b
```

3. optionally, convert results to VTK, and view again:

```
./extractor.py -d its2D.h5  
./postproc.py its2D.vtk -b
```

4. probe the data:

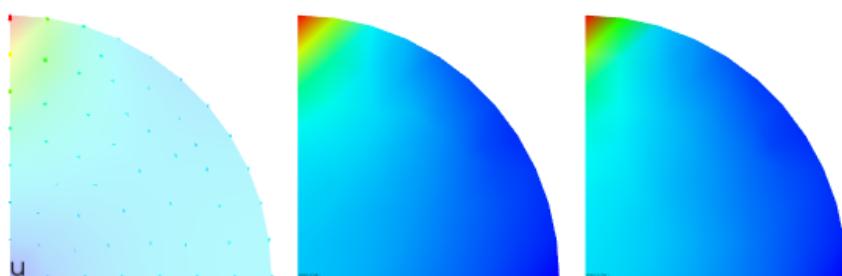
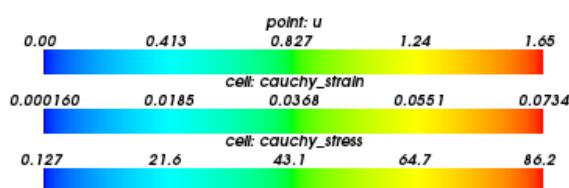
```
./probe.py examples/linear_elasticity/its2D_4.py its2D.h5
```

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`.
```

Use it as follows (assumes running from the `sfePy` directory; on Windows, you may need to prefix all the commands with "python " and remove "./"):

1. solve the problem::

```
./simple.py examples/linear_elasticity/its2D_4.py
```

2. optionally, view the results::

```
./postproc.py its2D.h5 -b
```

3. optionally, convert results to VTK, and view again::

```
./extractor.py -d its2D.h5
./postproc.py its2D.vtk -b
```

4. probe the data::

```
./probe.py examples/linear_elasticity/its2D_4.py its2D.h5
```

Find :math:`\mathbf{u}\_l(u)` such that:

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

where

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

```
"""
from __future__ import absolute_import
from 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))

    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'$e_{%s}$' % 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))

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\_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`, and how to display the results using Mayavi. 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 examples/linear_elasticity/its2D_interactive.py -h
```

## Notes

The --probe and --show options work simultaneously only if Mayavi and Matplotlib use the same backend type (for example wx).

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`, and how to display the results using Mayavi.
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 examples/linear_elasticity/its2D_interactive.py -h

Notes
-----
The ``--probe`` and ``--show`` options work simultaneously only if Mayavi and
Matplotlib use the same backend type (for example wx).
"""

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.ls import ScipyDirect, ScipySuperLU, ScipyUmfpack, MUMPSolver
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 examples.linear_elasticity.its2D_2 import stress_strain
from 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)
    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'$e_{\%s}$' % 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',
    'show' : 'show the results figure',
}

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('--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('-s', '--show',
                        action="store_true", dest='show',
                        default=False, help=helps['show'])
    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')

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(ebcos=Conditions([xsym, ysym]))

pb.set_solver(nls)

```

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

# Postprocess the solution.
out = state.create_output_dict()
out = stress_strain(out, pb, state, 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, state, 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 options.show:
    # Show the solution. If the approximation order is greater than 1, the
    # extra DOFs are simply thrown away.
    from sfepy.postprocess.viewer import Viewer

    view = Viewer('its2D_interactive.vtk')
    view(vector_mode='warp_norm', rel_scaling=1,
          is_scalar_bar=True, is_wireframe=True)

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

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

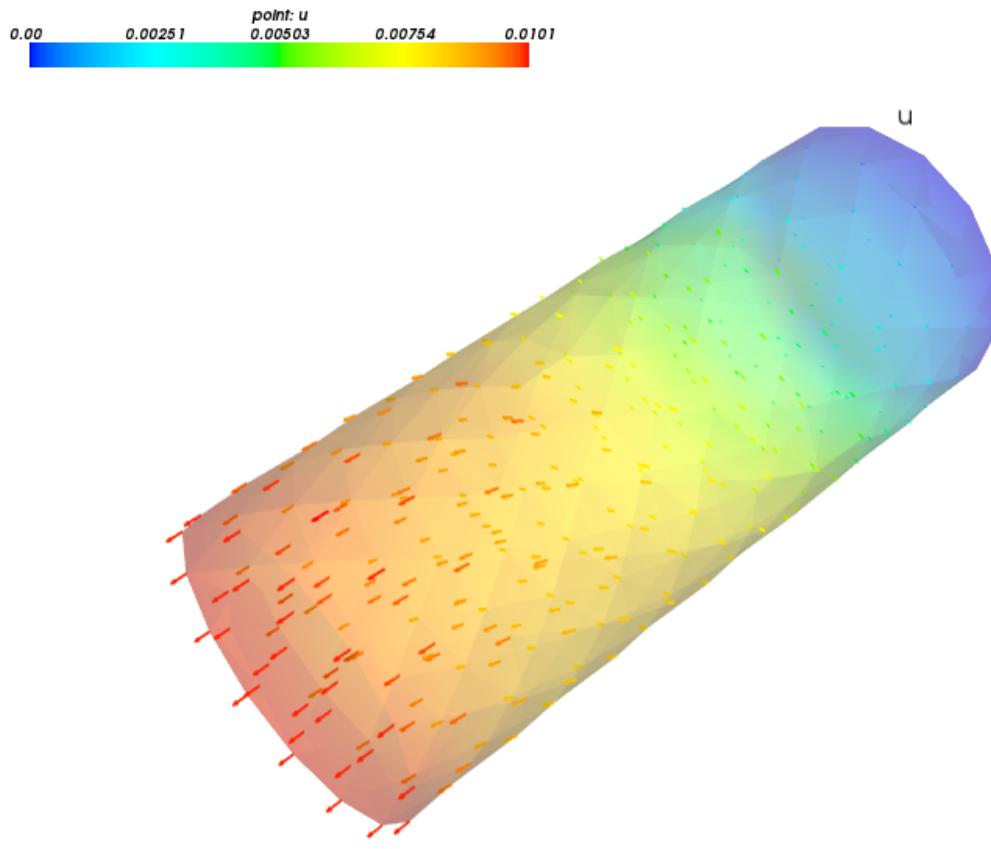
This example models a cylinder that is fixed at one end while the second end has a specified displacement of 0.01 in the x direction (this boundary condition is named 'Displaced'). There is also a specified displacement of 0.005 in the z direction for points in the region labeled 'SomewhereTop'. This 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  $\lambda$  and  $\mu$  (see [http://en.wikipedia.org/wiki/Lam%C3%A9\\_parameters](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:

```
$ ./postproc.py cylinder.vtk --wireframe -b --only-names=u -d'u,plot_displacements,
→rel_scaling=1'
```



[source code](#)

```
# -*- coding: utf-8 -*-
"""
Linear elasticity with given displacements.

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

.. math::
    \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}
    \quad .
```

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 :math:`\lambda` and :math:`\mu` (see [http://en.wikipedia.org/wiki/Lam%C3%A9\\_parameters](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::

```
$ ./postproc.py cylinder.vtk --wireframe -b --only-names=u -d'u,plot_displacements,
--rel_scaling=1'
"""

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}),
    '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**

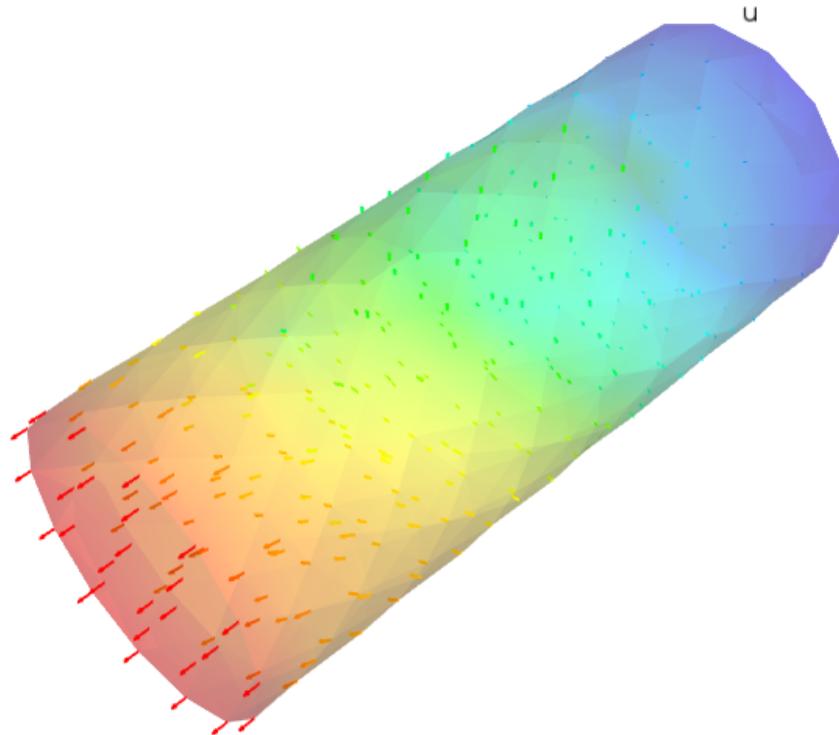
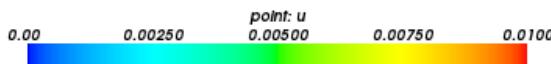
Time-dependent linear elasticity with a simple damping.

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:`\nabla u` such that:

.. math::
    \int_{\Omega} c \nabla v \cdot \nabla u \, dt
    + \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}
    ;.

"""

from __future__ import print_function
from __future__ import absolute_import
from copy import deepcopy

import numpy as nm
from examples.linear_elasticity.linear_elastic import \
    filename_mesh, materials, regions, fields, ebc, \
    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.
ebc = deepcopy(ebc)
ebc['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_volume_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,
        '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

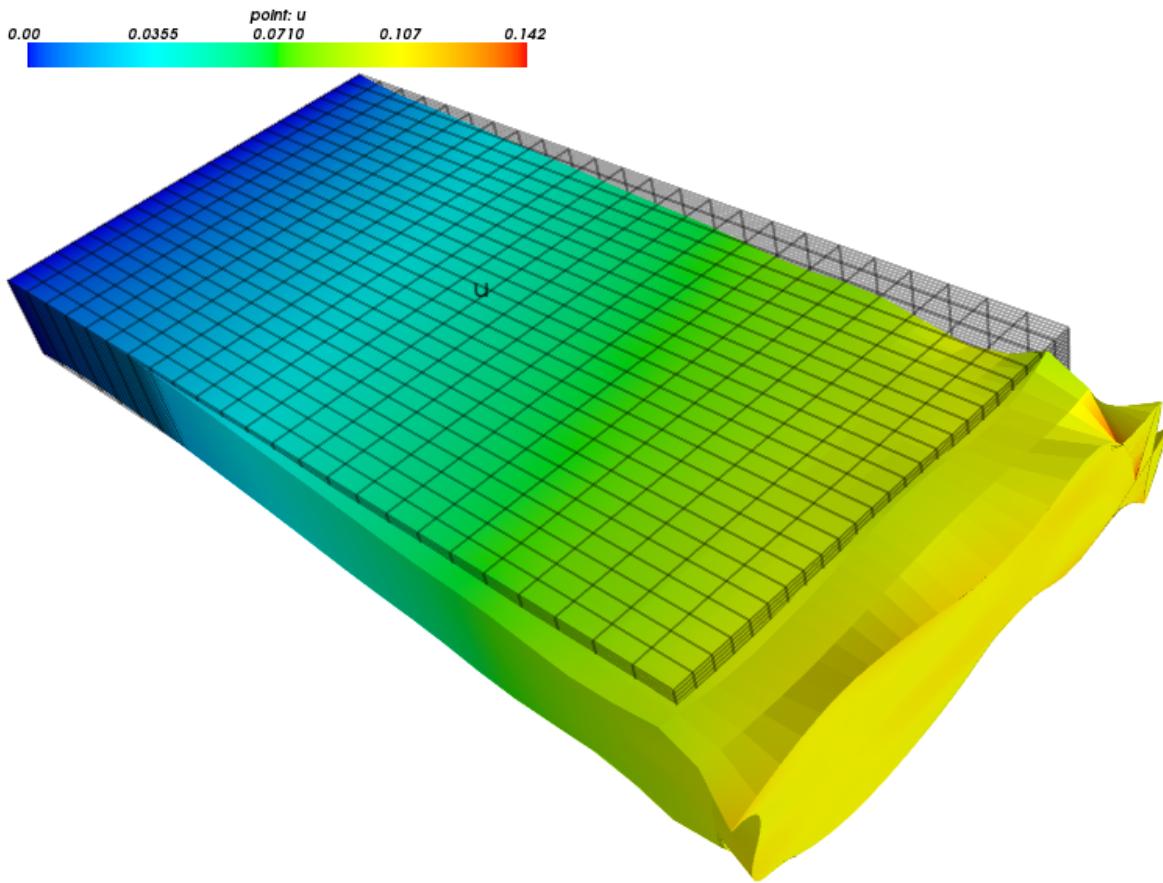
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} .$$

The domain geometry was created by:

```
$ ./script/gen_iga_patch.py -d [1,0.5,0.1] -s [11,5,3] --degrees [2,2,2] -o meshes/
˓→iga/block3d.iga
```

View the results using:

```
$ ./postproc.py block3d.vtk --wireframe -b
$ ./postproc.py block3d.vtk --wireframe -b -d 'u,plot_displacements,rel_scaling=1e0'
```



[source code](#)

```
r"""
Linear elasticity solved in a single patch NURBS domain using the isogeometric
analysis (IGA) approach.

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

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{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 .
```

The domain geometry was created by::

```
$ ./script/gen_iga_patch.py -d [1,0.5,0.1] -s [11,5,3] --degrees [2,2,2] -o meshes/
iga/block3d.iga
```

View the results using::

```
$ ./postproc.py block3d.vtk --wireframe -b
$ ./postproc.py block3d.vtk --wireframe -b -d 'u,plot_displacements,rel_scaling=1e0'
"""

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 = {
    '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

missing description!

source code

```

#!/usr/bin/env python
from __future__ import print_function
from __future__ import absolute_import
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.postprocess.viewer import Viewer
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

helps = {
    'show' : 'show the results figure',
}

def main():
    from sfepy import data_dir

    parser = ArgumentParser()
    parser.add_argument('--version', action='version', version='%(prog)s')
    parser.add_argument('-s', '--show',
                        action="store_true", dest='show',
                        default=False, help=helps['show'])
    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')
gammal = domain.create_region('Gammal',
                               '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)

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', gammal, {'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()
state = pb.solve(status=status)

print('Nonlinear solver status:\n', nls_status)
print('Stationary solver status:\n', status)

pb.save_state('linear_elasticity.vtk', state)

if options.show:
    view = Viewer('linear_elasticity.vtk')
```

```

    view(vector_mode='warp_norm', rel_scaling=2,
          is_scalar_bar=True, is_wireframe=True)

if __name__ == '__main__':
    main()

```

## linear\_elasticity/linear\_elastic\_tractions.py

### Description

Linear elasticity with pressure traction load on a surface and constrained to one-dimensional motion.

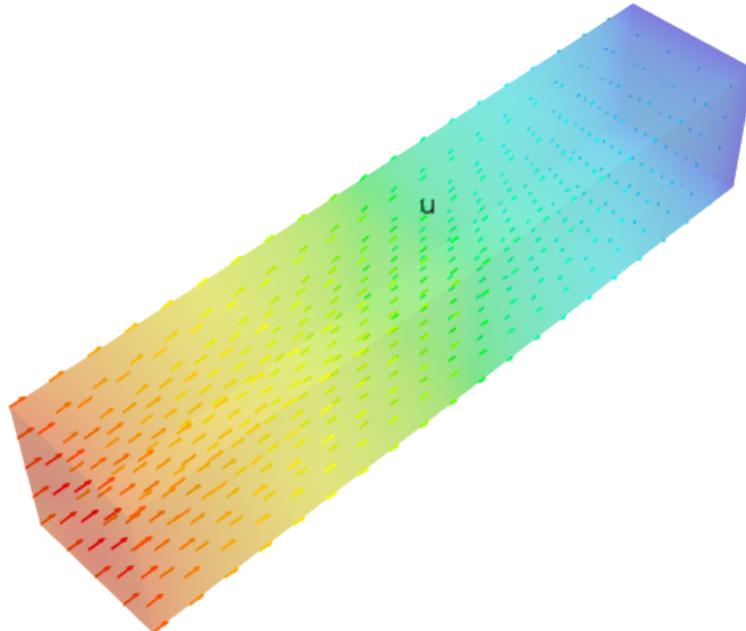
Find  $\underline{u}$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \quad \forall \underline{v},$$

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  $\bar{p}$ .



[source code](#)

```
r"""
Linear elasticity with pressure traction load on a surface and constrained to
one-dimensional motion.
```

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

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    = - \int_{\Gamma_{right}} \mathbf{u} \cdot \mathbf{n} \cdot \sigma \cdot \mathbf{n} \\
    ;, \quad \forall \mathbf{v} ;,
```

where

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

and :math:`\mathbf{\sigma} \cdot \mathbf{n} = p \mathbf{n}` with given traction pressure :math:`p`.

"""

```
from __future__ import absolute_import
import numpy as nm
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 define():
    """Define the problem to solve."""
    from sfepy import data_dir
```

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

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

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

```
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'),
}

ebcs = {
    'fixb' : ('Left', {'u.all' : 0.0}),
    'fixt' : ('Right', {'u.[1,2]' : 0.0}),
}

## 
# Balance of forces.
equations = {
    'elasticity' :
    """dw_lin_elastic.2.Omega( solid.D, v, u )
    = - dw_surface_ltr.2.Right( load.val, v )""",
}

## 
# Solvers etc.
solvers = {
    'ls' : ('ls.scipy_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()

```

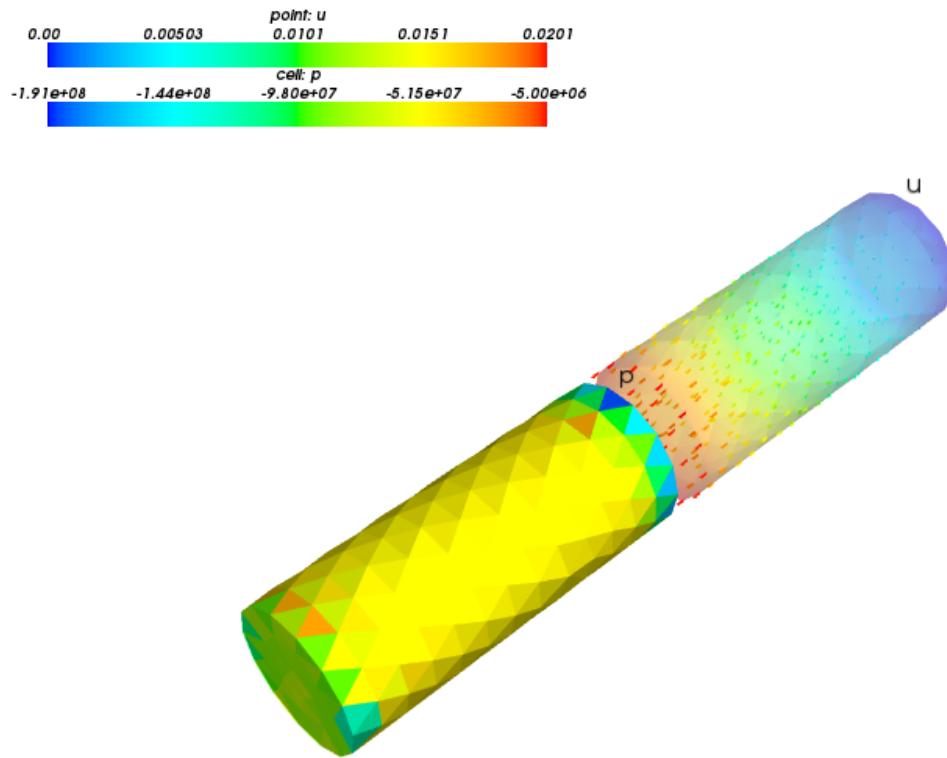
## linear\_elasticity/linear\_elastic\_up.py

### 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 :math:`\mathbf{u}`, :math:`p` such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    - \int_{\Omega} p \nabla \cdot \mathbf{u} \\
    = 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}

#! 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.
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_volume_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:

$$\begin{aligned} & \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \\ & + \int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}\left(\frac{\partial \underline{u}}{\partial \tau}(\tau)\right) d\tau \right] e_{ij}(\underline{v}) \\ & = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \quad \forall \underline{v}, \end{aligned}$$

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  $\underline{\sigma} \cdot \underline{n} = \bar{p} \underline{I} \cdot \underline{n}$  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 play with the following:

```
$ ./postproc.py block.h5 -b --only-names=u -d 'u,plot_displacements,rel_scaling=1e0,
→ opacity=1.0,color_name="viscous_stress",color_kind="tensors"' --wireframe
```

Use:

```
$ ./postproc.py -l block.h5
```

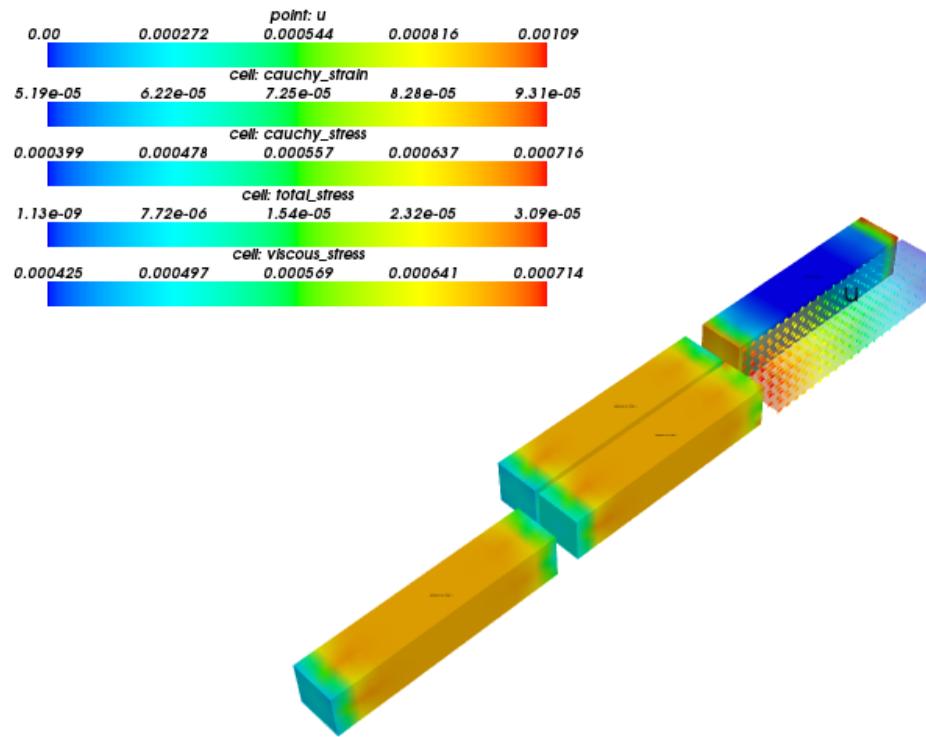
to see names and kinds of variables.

### Time history plots

Run the following:

```
$ python examples/linear_elasticity/linear_viscoelastic.py -h
$ python 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  
 $\mathcal{H}_{ijkl}(t) = \mathcal{H}_{ijkl}(0) e^{-d t}$  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  $\mathbf{u}$  such that:

```
.. math::
\int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
+ \int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial \tau}(\tau) d\tau \right] e_{ij}(\mathbf{v}) \\
= - \int_{\Gamma} \Gamma_{right} \mathbf{u} \cdot \mathbf{v} \sigma \cdot \mathbf{n} d\Gamma,
```

*where*

```
.. math::
  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  $\mathcal{U}^{\sigma} = \bar{p} I$  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 play with the following::

```
$ ./postproc.py block.h5 -b --only-names=u -d 'u,plot_displacements,rel_
˓→scaling=1e0,opacity=1.0,color_name="viscous_stress",color_kind="tensors"' --
˓→wireframe
```

*Use::*

```
$ ./postproc.py -l block.h5
```

to see names and kinds of variables.

*Time history plots*

^^^^^^^^^^^^^

Run the following::

```
$ python examples/linear_elasticity/linear_viscoelastic.py -h
$ python 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.

```
"""
from __future__ import absolute_import
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
import six

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]
        out['Hd'] = kernel[1, 0, 0] / kernel[0, 0, 0]

        for key, val in six.iteritems(out):
            out[key] = nm.tile(val, (coors.shape[0], 1, 1))

    return out

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)
output('fading memory length:', fading_memory_length)

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',
                                   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 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=vstress, 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=vstress, dofs=None)

    out['total_stress'] = Struct(name='output_data', mode='cell',
                                 data=estress + vstress, dofs=None)

    return out

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 )
        = - 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,
    }),
}
```

```

def main():
    """
    Plot the load, displacement, strain and stresses w.r.t. time.
    """
    from argparse import ArgumentParser, RawDescriptionHelpFormatter
    import matplotlib.pyplot as plt

    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'])
    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)

    normalized_kernel = kernel[:, 0, 0] / 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(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.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)
    plt.plot(ts.times, tensor, lw=3)
    plt.title('%s components, element %d' % (tensor_name, options.element))
    plt.xlabel('time')

plt.show()

if __name__ == '__main__':
    main()
```

## linear\_elasticity/material\_nonlinearity.py

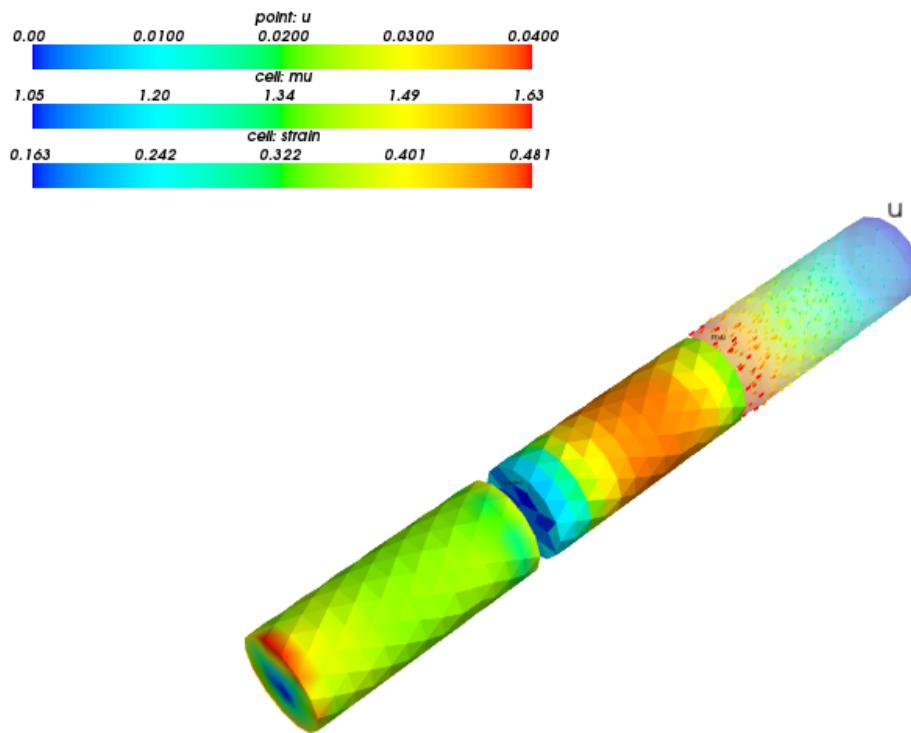
### Description

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 -*-
"""
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{u}_l(v)) \epsilon_{kl}(\mathbf{u}_l(u))
    = 0
    ;, \quad \forall \mathbf{u}_l(v) ;,
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.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_volume_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)

    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',
}

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/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 / m<sup>3</sup>

### Examples

- Run with the default arguments, show results (color = strain):

```
python examples/linear_elasticity/modal_analysis.py --show
```

- Fix bottom surface of the domain, show 9 eigen-shapes:

```
python examples/linear_elasticity/modal_analysis.py -b cantilever -n 9 --show
```

- Increase mesh resolution:

```
python examples/linear_elasticity/modal_analysis.py -s 31,31 -n 9 --show
```

- Use 3D domain:

```
python examples/linear_elasticity/modal_analysis.py -d 1,1,1 -c 0,0,0 -s 8,8,8 --  
    ↵show
```

- Change the eigenvalue problem solver to LOBPCG:

```
python examples/linear_elasticity/modal_analysis.py --solver="eig.scipy_lobpcg,i_  
    ↵max:100, largest:False" --show
```

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
```

```

- length: m
- mass: kg
- stiffness / stress: Pa
- density: kg / m^3

Examples
-----
- Run with the default arguments, show results (color = strain)::

    python examples/linear_elasticity/modal_analysis.py --show

- Fix bottom surface of the domain, show 9 eigen-shapes::

    python examples/linear_elasticity/modal_analysis.py -b cantilever -n 9 --show

- Increase mesh resolution::

    python examples/linear_elasticity/modal_analysis.py -s 31,31 -n 9 --show

- Use 3D domain::

    python examples/linear_elasticity/modal_analysis.py -d 1,1,1 -c 0,0,0 -s 8,8,8 --
    ↪show

- Change the eigenvalue problem solver to LOBPCG::

    python examples/linear_elasticity/modal_analysis.py --solver="eig.scipy_lobpcg,i_
    ↪max:100,largest:False" --show

    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
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]',
'show' : 'show the results figure',
}

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',
                        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('--show',
                    action="store_true", dest='show',
                    default=False, help=helps['show']))
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),
        "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')

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_volume_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(ebcsl=Conditions([fixed]))
    n_rbm = 0

elif options.bc_kind == 'fixed':
    fixed = EssentialBC('Fixed', bottom_top, {'u.all' : 0.0})
    pb.time_update(ebcsl=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)' %
           (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.get_variables()

vecs = nm.empty((variables.di.ptr[-1], 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 = {}
state = pb.create_state()
for ii in range(eigs.shape[0]):
    state.set_full(vecs[:, ii])
    aux = state.create_output_dict()
    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 len(eigs) and options.show:
    # Show the solution. If the approximation order is greater than 1, the
    # extra DOFs are simply thrown away.
    from sfepy.postprocess.viewer import Viewer
    from sfepy.postprocess.domain_specific import DomainSpecificPlot

    scaling = 0.05 * dims.max() / nm.abs(vecs).max()

    ds = {}
    for ii in range(eigs.shape[0]):
        pd = DomainSpecificPlot('plot_displacements',
                                ['rel_scaling=%s' % scaling,
                                 'color_kind="tensors"',
                                 'color_name="strain%03d"' % ii])
        ds['u%03d' % ii] = pd

    view = Viewer('eigenshapes.vtk')
    view(domain_specific=ds, only_names=sorted(ds.keys()),
          is_scalar_bar=False, is_wireframe=True)

if __name__ == '__main__':
    main()
```

## linear\_elasticity/nodal\_lcbc.py

### Description

Linear elasticity with nodal linear combination constraints.

Find  $\underline{u}$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \quad \forall \underline{v},$$

where

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

and  $\underline{\sigma} \cdot \underline{n} = \bar{p}\underline{I} \cdot \underline{n}$  with given traction pressure  $\bar{p}$ . The constraints are given in terms of coefficient matrices and right-hand sides, see the `lcbc` 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 `simple.py`. Try:

```
python simple.py examples/linear_elasticity/nodal_lcbc.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:

```
python simple.py examples/linear_elasticity/nodal_lcbc.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:

```
python simple.py examples/linear_elasticity/nodal_lcbc.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:

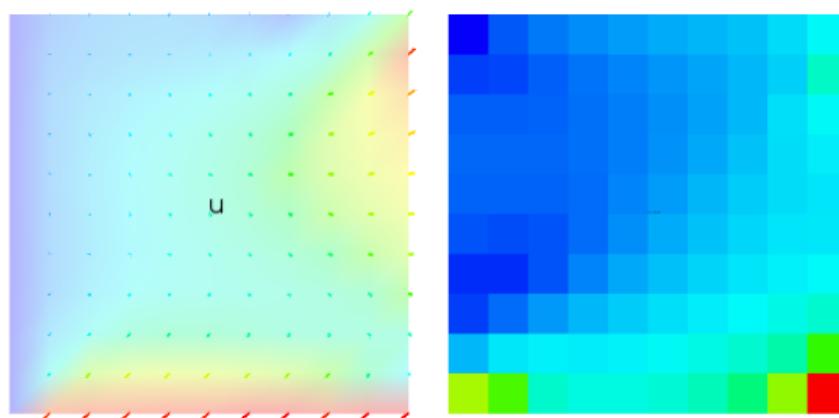
```
python postproc.py square_quad.vtk --wireframe -b

python postproc.py square_quad.vtk --wireframe -b --only-names=u -d'u,plot_
↪displacements,rel_scaling=1,color_kind="scalars",color_name="von_mises_stress"
```

View the 3D results using:

```
python postproc.py cube_medium_tetra.vtk --wireframe -b

python postproc.py cube_medium_tetra.vtk --wireframe -b --only-names=u -d'u,plot_
↪displacements,rel_scaling=1,color_kind="scalars",color_name="von_mises_stress"
```



[source code](#)

```
"""
Linear elasticity with nodal linear combination constraints.

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

```
.. math::
\int_{ijkl} \Omega_{ijkl} e_{ij}(\mathbf{u}_l(v)) e_{kl}(\mathbf{u}_l(u))
= - \int_{right} \mathbf{u}_l(v) \cdot \nabla \mathbf{u}_l(\sigma) \cdot \mathbf{u}_l(n)
;, \quad \text{forall } \mathbf{u}_l(v) ;,
```

where

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

and  $\nabla \cdot \mathbf{u}_l(\sigma) \cdot \mathbf{u}_l(n) = \bar{p} \cdot \mathbf{u}_l(n)$  with given traction pressure  $\bar{p}$ . The constraints are given in terms of coefficient matrices and right-hand sides, see the ``lcbs`` 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 ``simple.py``. Try::

```
python simple.py examples/linear_elasticity/nodal_lcbs.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::

```
python simple.py examples/linear_elasticity/nodal_lcbs.py --define='use_ebc: False
'.
```

to replace ``ebcs`` with the ``nlcbc4`` constraints. The results should be the same for the two cases. Both options can be combined::

```
python simple.py examples/linear_elasticity/nodal_lcbs.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::

```
python postproc.py square_quad.vtk --wireframe -b
```

```
python postproc.py square_quad.vtk --wireframe -b --only-names=u -d'u,plot_
displacements,rel_scaling=1,color_kind="scalars",color_name="von_mises_stress"'
```

View the 3D results using::

```
python postproc.py cube_medium_tetra.vtk --wireframe -b
```

```
python postproc.py cube_medium_tetra.vtk --wireframe -b --only-names=u -d'u,plot_
displacements,rel_scaling=1,color_kind="scalars",color_name="von_mises_stress"
'''
```

```

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.
    """
    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_ebc=True):
    assert_(dim in (2, 3))

    if dim == 2:
        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:
        lcbc = {
            'nlcbc1' : ('Top', {'u.all' : None}, None, 'nodal_combination',
                        ([[1.0, -1.0]], [0.0])),
            'nlcbc2' : ('Bottom', {'u.all' : None}, None, 'nodal_combination',
                        ([[1.0, 1.0]], [0.0]))
        }
    else:
        lcbc = {
            'nlcbc1' : ('Top', {'u.all' : None}, None, 'nodal_combination',
                        ([[1.0, 0.0], [0.0, 1.0]])),
            'nlcbc2' : ('Bottom', {'u.all' : None}, None, 'nodal_combination',
                        ([[1.0, 0.0], [0.0, 1.0]]))
        }
```

```

        ([[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_ebcbs:
    ebcbs = {
        'fix' : ('Left', {'u.all' : 0.0}),
    }

else:
    ebcbs = {}

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

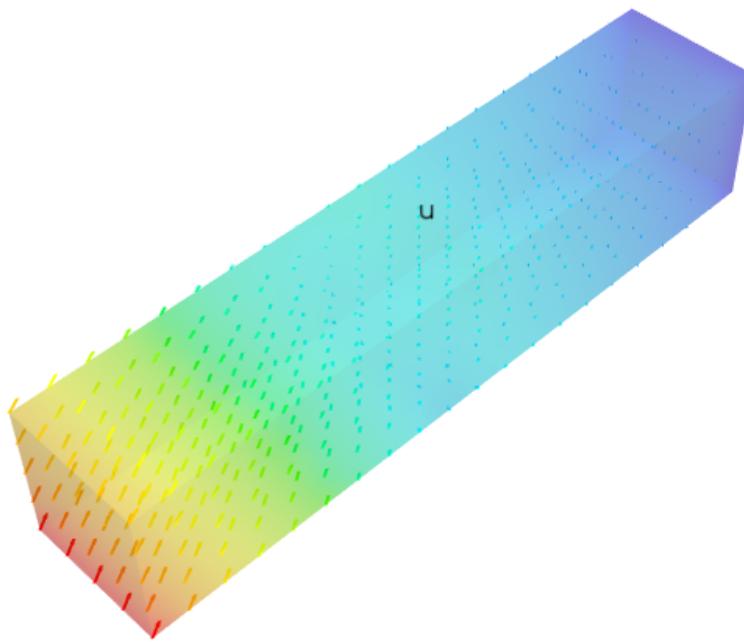
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

The stiffness of fibres  $D_{ijkl}^f$  is defined analogously,  $\underline{d}$  is the unit fibre direction vector and  $\sigma_{ij}$  is the prestress.

## Visualization

Use the following to see the deformed structure with 10x magnified displacements:

```
$ ./postproc.py block.vtk -b --vector-mode=warp_norm -s 1 --wireframe
```



[source code](#)

```
r"""
Linear elasticity with a given prestress in one subdomain and a (pre)strain
fibre reinforcement in the other.

```

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

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\underline{u}) e_{kl}(\underline{u})
    + \int_{\Omega_1} \sigma_{ij} e_{ij}(\underline{u})
    + \int_{\Omega_2} D^f_{ijkl} e_{ij}(\underline{u}) \left(d_k d_l\right)
    = 0
    ;, \quad \forall \underline{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:`\mathbf{u}\_{ld}` 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::

```
$ ./postproc.py block.vtk -b --vector-mode=warp_norm -s 1 --wireframe
"""
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),
        '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.Omegal( 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/shell10x\_cantilever.py

### Description

Bending of a long thin cantilever beam computed using the `dw_shell10x` term.

Find displacements of the central plane  $\underline{u}$ , and rotations  $\underline{\alpha}$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}, \underline{\beta}) e_{kl}(\underline{u}, \underline{\alpha}) = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{f}, \quad \forall \underline{v},$$

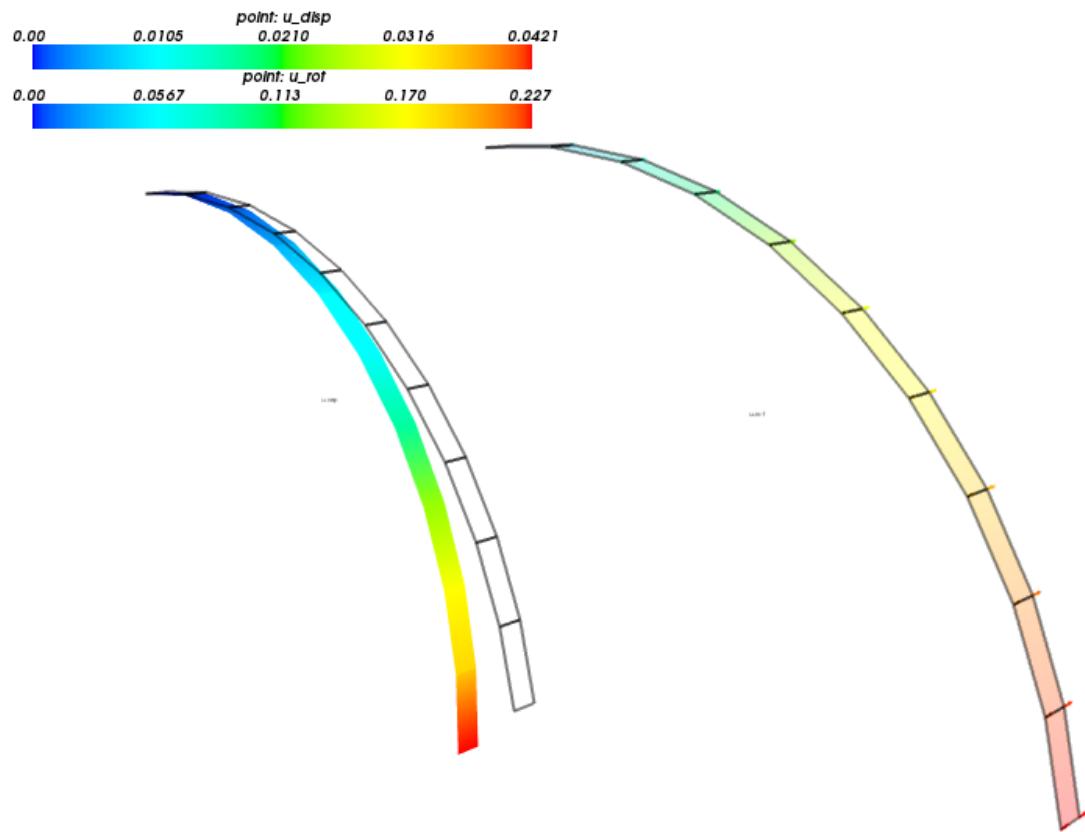
where  $D_{ijkl}$  is the isotropic elastic tensor, given using the Young's modulus  $E$  and the Poisson's ratio  $\nu$ .

The variable  $\underline{u}$  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:

```
python postproc.py shell10x.vtk -d 'u_disp,plot_displacements,rel_scaling=1.0' --
    opacity='wireframe=0.5' -b --wireframe
```



[source code](#)

```
r"""
Bending of a long thin cantilever beam computed using the
:class:`dw_shell10x <sfepy.terms.terms_shells.Shell10XTerm>` term.

Find displacements of the central plane :math:`\mathbf{u}`, and rotations
:math:`\mathbf{\alpha}` such that:

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}, \mathbf{u}) e_{kl}(\mathbf{u}, \mathbf{\alpha})
    = - \int_{\Gamma_{right}} \mathbf{v} \cdot \mathbf{f}
    ;, \quad \forall \mathbf{v} ;
where :math:`D_{ijkl}` is the isotropic elastic tensor, given using the Young's
modulus :math:`E` and the Poisson's ratio :math:`\nu`.

The variable ``u`` below holds both :math:`\mathbf{u}` and :math:`\mathbf{\alpha}` DOFs. For visualization, it is saved as two fields ``u_disp`` and ``u_rot``,
corresponding to :math:`\mathbf{u}` and :math:`\mathbf{\alpha}` respectively.

See also :ref:`linear_elasticity-shell10x_cantilever_interactive` example.

View the results using::

    python postproc.py shell10x.vtk -d 'u_disp,plot_displacements,rel_scaling=1.0' --
    <opacity='wireframe=0.5'> -b --wireframe

```

```
"""
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 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:')
    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'),
}

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

ebcs = {
    'fix' : ('Gamma1', {'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 computed using 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  $\nu$ .

The variable  $\underline{u}$  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:

```
python 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:

```
python examples/linear_elasticity/shell10x_cantilever_interactive.py output -t bend -p
```

Apply the twisting transformation to the beam domain coordinates, change number of cells, show the solution:

```
python examples/linear_elasticity/shell10x_cantilever_interactive.py output -t twist -
-n 2,51,3 -s
```

source code

```
#!/usr/bin/env python
r"""

Bending of a long thin cantilever beam computed using the
:class:`dw_shell10x <sfeypy.terms.terms_shells.Shell10XTerm>` term.
```

Find displacements of the central plane  $\mathbf{u}$ , and rotations  $\mathbf{\alpha}$  such that:

```
.. math::
\int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}, \mathbf{v}) e_{kl}(\mathbf{u}, \mathbf{\alpha})
= - \int_{\Gamma} \mathbf{f} \cdot \mathbf{v} \quad \forall \mathbf{v},
```

where  $D_{ijkl}$  is the isotropic elastic tensor, given using the Young's modulus  $E$  and the Poisson's ratio  $\nu$ .

The variable `` $\mathbf{u}$ '' below holds both  $\mathbf{u}$  and  $\mathbf{\alpha}$  DOFs. For visualization, it is saved as two fields `` $\mathbf{u}_\text{disp}$ '' and `` $\mathbf{u}_\text{rot}$ '', corresponding to  $\mathbf{u}$  and  $\mathbf{\alpha}$ , 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 :ref:`linear\_elasticity-shell10x\_cantilever` example.

#### Usage Examples

---

See all options::

```
python 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::

```
python examples/linear_elasticity/shell10x_cantilever_interactive.py output -t bend
→ -p
```

Apply the twisting transformation to the beam domain coordinates, change number of `cells`, show the solution::

```
python examples/linear_elasticity/shell10x_cantilever_interactive.py output -t
→ twist -n 2,51,3 -s
"""
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.solvers import use_first_available
from sfepy.solvers.ls import MUMPSolver, ScipyDirect
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)

    conn = [_mesh.get_conn(_mesh.descs[0])]

    mesh = Mesh.from_data(_mesh.name, coors, _mesh.cmesh.vertex_groups, conn,
                          [_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]

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)

    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_base='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 = use_first_available([(MUMPSolver, {}), (ScipyDirect, {})])
    nls_status = IndexedStruct()

```

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

pb = Problem('elasticity with shell10x', equations=eqs)
pb.set_bcs(ebcos=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',
    'scaling' : 'the displacement scaling, with --show [default: %(default)s]',
    'show' : 'show the results figure',
    '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'])
    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('--u-scaling', metavar='float', type=float,
                    action='store', dest='scaling',
                    default=1.0, help=helps['scaling'])
parser.add_argument('-s', '--show',
                    action="store_true", dest='show',
                    default=False, help=helps['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)

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)

    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))

    plt.show()

if options.show:
    from sfepy.postprocess.viewer import Viewer
    from sfepy.postprocess.domain_specific import DomainSpecificPlot

    ds = {'u_disp' :
          DomainSpecificPlot('plot_displacements',
                             ['rel_scaling=%f' % options.scaling])}
    view = Viewer(odir('shell10x_cantilever.vtk'))
    view(domain_specific=ds, is_scalar_bar=True, is_wireframe=True,
          opacity={'wireframe' : 0.5})

if __name__ == '__main__':
    main()

```

## 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:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) + \int_{\Gamma_c} \varepsilon_N \langle g_N(\underline{u}) \rangle n \underline{v} = 0, \quad \forall \underline{v},$$

where  $\varepsilon_N \langle g_N(\underline{u}) \rangle$  is the penalty function,  $\varepsilon_N$  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

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

Usage examples:

```

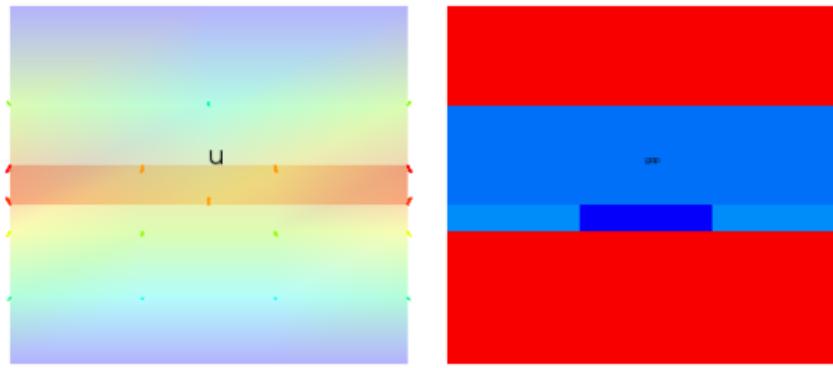
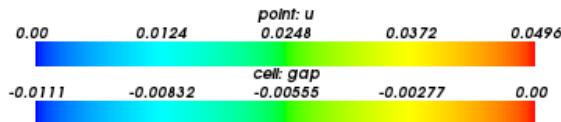
./simple.py examples/linear_elasticity/two_bodies_contact.py --save-regions-as-groups_
--save-ebc-nodes

./postproc.py two_bodies.mesh.vtk -b --wire
./postproc.py two_bodies.mesh.vtk -b --wire -d 'u,plot_displacements,rel_scaling=1.0'

```

```
./script/plot_logs.py log.txt

./postproc.py --wire two_bodies.mesh_efc_nodes.vtk
./postproc.py --wire two_bodies.mesh_regions.vtk
```



[source code](#)

```
r"""
Contact of two elastic bodies with a penalty function for enforcing the contact
constraints.
```

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

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    + \int_{\Gamma_c} \varepsilon_N \langle g_N(\mathbf{u}) \rangle_n \mathbf{u}_n \mathbf{u}_v \\
    = 0
    ;, \quad \forall \mathbf{v}
```

where :math:`\langle \varepsilon\_N(\mathbf{u}) \rangle\_n` is the penalty function, :math:`\varepsilon\_N` is the normal penalty parameter, :math:`\langle \cdot \rangle\_n` are the Macaulay's brackets of the gap function :math:`g\_N(\mathbf{u})` and

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

```
\;.

Usage examples::

./simple.py examples/linear_elasticity/two_bodies_contact.py --save-regions-as-
→groups --save-ebc-nodes

./postproc.py two_bodies.mesh.vtk -b --wire
./postproc.py two_bodies.mesh.vtk -b --wire -d 'u,plot_displacements,rel_scaling=1.0
→'

./script/plot_logs.py log.txt

./postproc.py --wire two_bodies.mesh_ebc_nodes.vtk
./postproc.py --wire two_bodies.mesh_regions.vtk
"""

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.discrete.fem.meshio import UserMeshIO

import numpy as nm

dim = 2

if dim == 2:
    dims0 = [1.0, 0.5]
    shape0 = [4, 4]
    centre0 = [0, -0.25]

    dims1 = [1.0, 0.5]
    shape1 = [3, 3]
    centre1 = [0, 0.25]

    shift1 = [0.0, -0.1]

else:
    dims0 = [1.0, 1.0, 0.5]
    shape0 = [2, 2, 2]
    centre0 = [0, 0, -0.25]

    dims1 = [1.0, 1.0, 0.5]
    shape1 = [2, 2, 2]
    centre1 = [0, 0, 0.25]

    shift1 = [0.0, 0.0, -0.1]

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 = Mesh.from_data(name, coors, ngroups, [conn], [mat_id], m0.descs)

mesh.write(name, io='auto')

return mesh

def mesh_hook(mesh, mode):
    if mode == 'read':
        return gen_two_bodies(dims0, shape0, centre0,
                              dims1, shape1, centre1, shift1)

    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)

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process',
}

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

materials = {
    'solid' : ({'D': stiffness_from_youngpoisson(dim,
                                                    young=1.0, poisson=0.3)},),
```

```

'contact' : ({'.epss' : 1e1},),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}
bbox0 = get_bbox(dims0, centre0, eps=1e-5)
bbox1 = get_bbox(dims1, nm.asarray(centre1) + nm.asarray(shift1), eps=1e-5)

if dim == 2:
    regions = {
        'Omega' : 'all',
        'Omega0' : 'cells of group 0',
        'Omega1' : 'cells of group 1',
        'Bottom' : ('vertices in (y < %f)' % bbox0[0, 1], 'facet'),
        'Top' : ('vertices in (y > %f)' % bbox1[1, 1], 'facet'),
        'Contact0' : ('(vertices in (y > %f) *v r.Omega0)' % bbox0[1, 1],
                      'facet'),
        'Contact1' : ('(vertices in (y < %f) *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 < %f)' % bbox0[0, 2], 'facet'),
        'Top' : ('vertices in (z > %f)' % bbox1[1, 2], 'facet'),
        'Contact0' : ('(vertices in (z > %f) *v r.Omega0)' % bbox0[1, 2],
                      'facet'),
        'Contact1' : ('(vertices in (z < %f) *v r.Omega1)' % bbox1[0, 2],
                      'facet'),
        'Contact' : ('r.Contact0 +s r.Contact1', 'facet')
    }

ebcs = {
    'fixb' : ('Bottom', {'u.all' : 0.0}),
    'fixt' : ('Top', {'u.all' : 0.0}),
}
integrals = {
    'i' : 10,
}
equations = {
    'elasticity' :
        """dw_lin_elastic.2.Omega(solid.D, v, u)
        + dw_contact.i.Contact(contact.epss, v, u)
        = 0""",
}
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
}

```

```
'i_max' : 5,
'eps_a' : 1e-6,
'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' : 100.1,
'ls_min' : 1e-5,
'check' : 0,
'delta' : 1e-8,
'log' : {'text' : 'log.txt', 'plot' : None},
})
}
```

## 5.6.6 miscellaneous

### miscellaneous/live\_plot.py

#### Description

missing description!

source code

```
from __future__ import print_function
from __future__ import absolute_import
import os
import sys
sys.path.append( '.' )

import numpy as nm

from sfepy.base.base import output, pause
from sfepy.base.log import Log

def main():
    cwd = os.path.split(os.path.join(os.getcwd(), __file__))[0]

    log = Log(([['sin(x) + i sin(x**2)', 'cos(x)', ['exp(x)']],
               yscales=['linear', 'log'],
               xlabel='angle', ylabel=None, a function',
               log_filename=os.path.join(cwd, 'live_plot.log')))

    log2 = Log([('x^3'],
               yscales=['linear'],
               xlabel='x', ylabel='a cubic function',
               aggregate=50, sleep=0.05,
               log_filename=os.path.join(cwd, 'live_plot2.log'),
               formats=[['{:5e}']])

    added = 0
    for x in nm.linspace(0, 4.0 * nm.pi, 200):
        output('x: ', x)

        if x < (2.0 * nm.pi):
            log(nm.sin(x)+1j*nm.sin(x**2), nm.cos(x), nm.exp(x), x=[x, None])
```

```

else:
    if added:
        log(nm.sin(x)+1j*nm.sin(x**2), nm.cos(x), nm.exp(x), x**2,
            x=[x, None, x])
    else:
        log.plot_vlines(color='r', linewidth=2)
        log.add_group(['x^2'], yscale='linear', xlabel='new x',
                      ylabel='square', formats=['%+g'])
        added += 1

if (added == 20) or (added == 50):
    log.plot_vlines([2], color='g', linewidth=2)

log2(x*x*x, x=[x])

print(log)
print(log2)
pause()

log(finished=True)
log2(finished=True)

if __name__ == '__main__':
    main()

```

## 5.6.7 multi\_physics

### multi\_physics/biot.py

#### Description

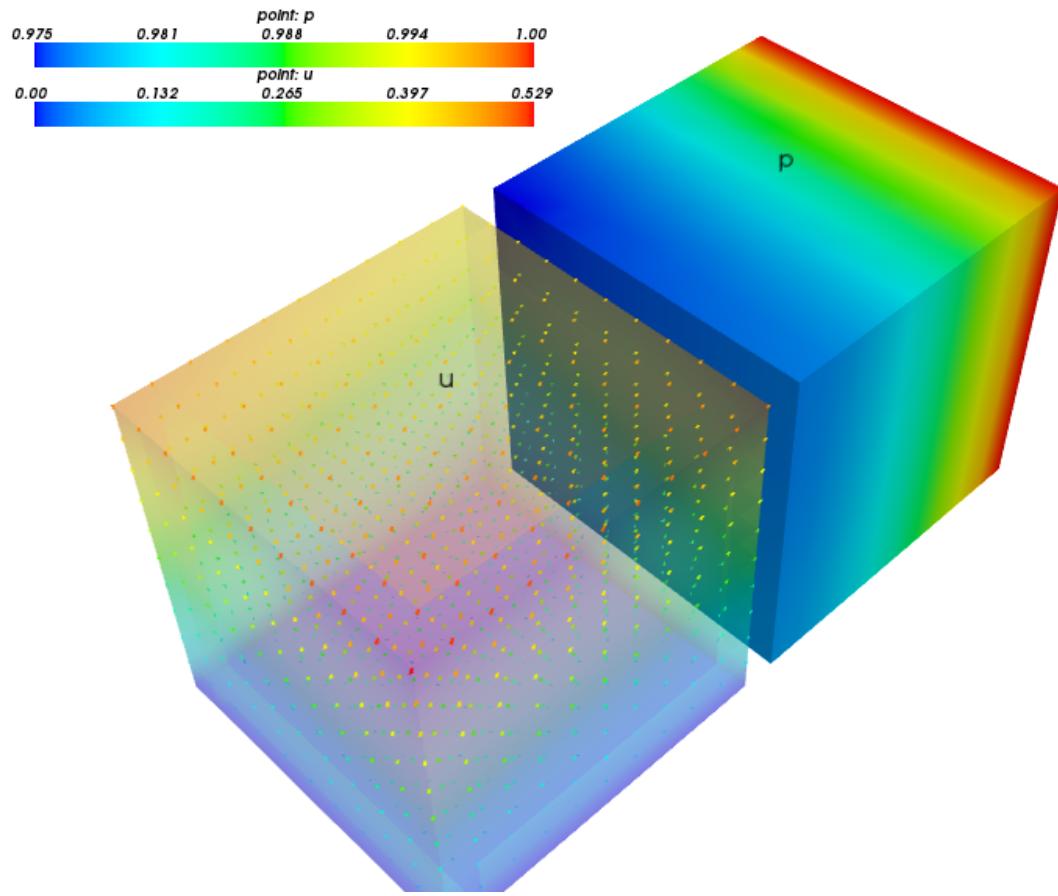
Biot problem - deformable porous medium.

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:`\mathbf{u}`, :math:`p` such that:

.. math::
    \int_{\Omega} D_{ijkl} \mathbf{e}_{ij}(\mathbf{u}) \mathbf{e}_{kl}(\mathbf{u}) \\
    - \int_{\Omega} p \alpha_{ij} \mathbf{e}_{ij}(\mathbf{u}) \\
    = 0
    \quad \forall \mathbf{u} \in V, \\

    \int_{\Omega} q \alpha_{ij} \mathbf{e}_{ij}(\mathbf{u}) \\
    + \int_{\Omega} K_{ij} \nabla_i q \cdot \nabla_j p \\
    = 0
    \quad \forall q \in 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}),
    '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,
    'ls_on' : 1.1,
    'ls_min' : 1e-5,
    'check' : 0,
    'delta' : 1e-6,
}

```

## multi\_physics/biot\_npbc.py

### Description

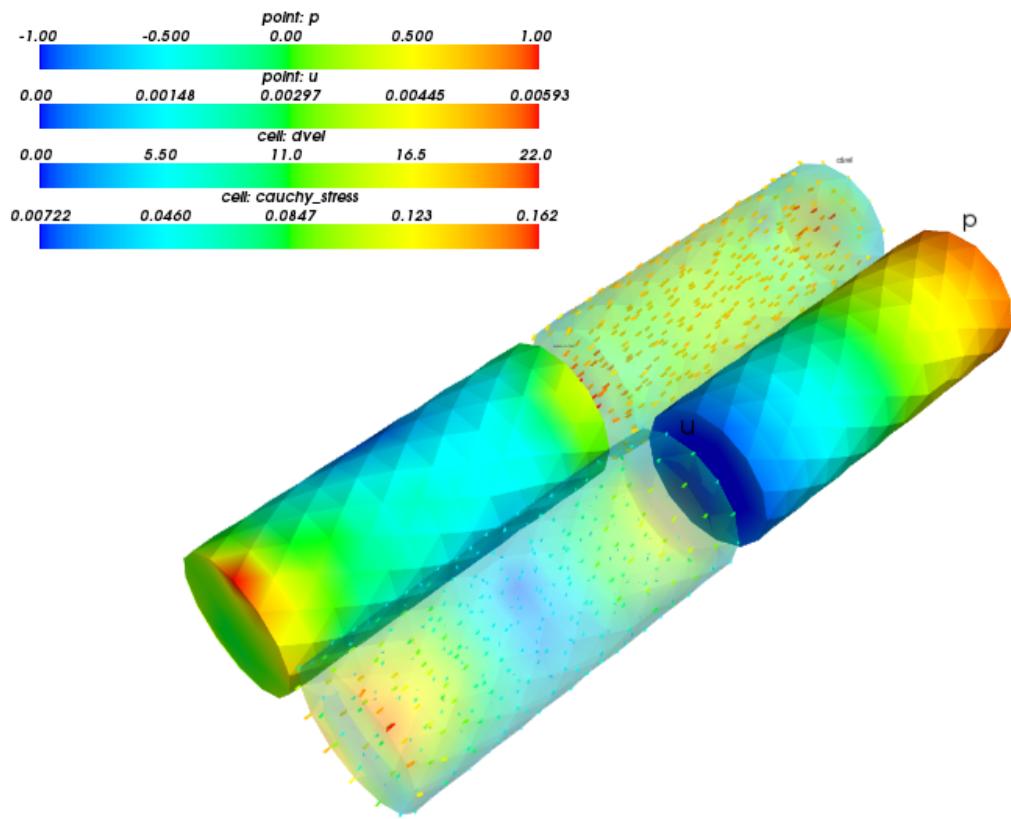
Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region.

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 :math:`\mathbf{u}`, :math:`p` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u})
    - \int_{\Omega} p \alpha_{ij} e_{ij}(\mathbf{u})
    = 0
    ;, \quad \forall \mathbf{v} ;,

    \int_{\Omega} q \alpha_{ij} e_{ij}(\mathbf{u})
    + \int_{\Omega} K_{ij} \nabla_i q \cdot \nabla_j p
    = 0
    ;, \quad \forall q ;,

    \mathbf{u} \cdot \mathbf{n} = 0 \text{ on } \Gamma_{walls} ;,
```

where

```
.. math::
    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.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
        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)

    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'),
}

ebcs = {
    'inlet' : ('Inlet', {'p.0' : 1.0, 'u.all' : 0.0}),
    'outlet' : ('Outlet', {'p.0' : -1.0}),
}

lcbc = {
    '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\_npbc\_lagrange.py

### Description

Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region enforced using Lagrange multipliers.

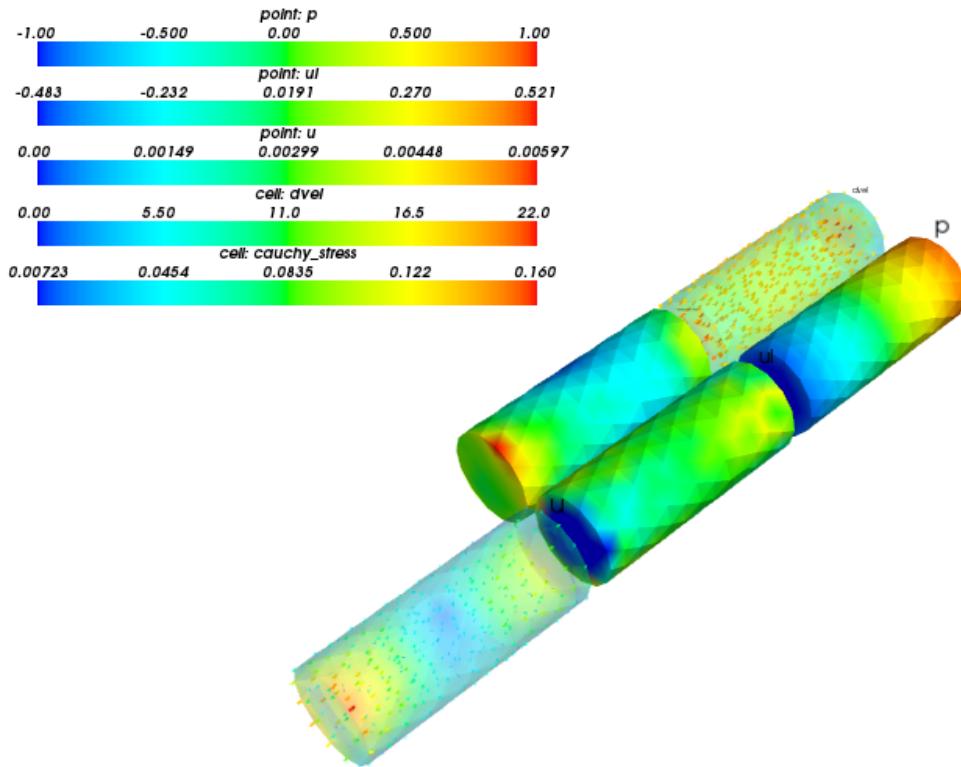
The non-penetration condition is enforced weakly using the Lagrange multiplier  $\lambda$ . There is also a rigid body movement constraint imposed on the  $\Gamma_{outlet}$  region using the linear combination boundary conditions.

Find  $\underline{u}$ ,  $p$  and  $\lambda$  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 :math:`\lambda`. There is also a rigid body movement constraint imposed on the :math:`\Gamma\_{outlet}` region using the linear combination boundary conditions.

Find :math:`\mathbf{u}`, :math:`\mathbf{p}` and :math:`\lambda` such that:

$$\begin{aligned} \text{.. math::} \\ & \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\ & - \int_{\Omega} p \alpha_{ij} e_{ij}(\mathbf{u}) \\ & + \int_{\Gamma_{walls}} \lambda u_l n_l \cdot \mathbf{u} \\ & = 0 \\ & \quad \forall \mathbf{v}, \\ & \int_{\Omega} q \alpha_{ij} e_{ij}(\mathbf{u}) \\ & + \int_{\Omega} K_{ij} \nabla_i q \cdot \nabla_j p \\ & = 0 \\ & \quad \forall q, \\ & \int_{\Gamma_{walls}} \hat{\lambda} u_l n_l \cdot \mathbf{u} \\ & = 0 \\ & \quad \forall \hat{\lambda}, \\ & \mathbf{u} \cdot \mathbf{n} = 0 \text{ on } \Gamma_{walls}, \end{aligned}$$

where

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

"""
from __future__ import absolute_import
from examples.multi_physics.biot_npbc 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',
                         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',
        ## 'file_per_var' : True,

        '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}),
    }

    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  $\alpha_{ij}$  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:

```
$ python examples/multi_physics/biot_parallel_interactive.py -h
```

See PETSc options:

```
$ python examples/multi_physics/biot_parallel_interactive.py --help
```

Single process run useful for debugging with `debug()`:

```
$ python examples/multi_physics/biot_parallel_interactive.py output-parallel
```

Parallel runs:

```
$ mpiexec -n 3 python examples/multi_physics/biot_parallel_interactive.py output-parallel -2 --shape=101,101
$ mpiexec -n 3 python examples/multi_physics/biot_parallel_interactive.py output-parallel -2 --shape=101,101 --metis
$ mpiexec -n 8 python 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:

```
$ mpiexec -n 2 python examples/multi_physics/biot_parallel_interactive.py output-parallel --shape=101,101 -snes_monitor -snes_converged_reason -ksp_monitor -pc_type_fieldsplit
$ mpiexec -n 8 python 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 (strip linearization or approximation orders one):

```
$ python postproc.py output-parallel/sol.h5 --wireframe -b -d'p,plot_warp_scalar:u,plot_displacements'
```

View the results using (adaptive linearization):

```
$ python postproc.py output-parallel/sol_u.h5 --wireframe -b -d'u,plot_displacements'
$ python postproc.py output-parallel/sol_p.h5 --wireframe -b -d'p,plot_warp_scalar'
```

source code

```
#!/usr/bin/env python
r"""
```

*Parallel assembling and solving of a Biot problem (deformable porous medium), using commands for interactive use.*

Find :math:`\mathbf{u}`, :math:`p` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    - \int_{\Omega} p \alpha_{ij} e_{ij}(\mathbf{u}) \\
    = 0
    \quad \forall \mathbf{u} \quad ;,
    \quad \forall q \quad ;,
    \quad \forall \mathbf{K}_{ij} \nabla_i q \nabla_j p \\
    = 0
    \quad \forall q \quad ;,
```

where

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

#### 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 :math:`\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::

```
$ python examples/multi_physics/biot_parallel_interactive.py -h
```

See PETSc options::

```
$ python examples/multi_physics/biot_parallel_interactive.py -help
```

```

Single process run useful for debugging with :func:`debug()
<sfeypy.base.base.debug>`::

$ python examples/multi_physics/biot_parallel_interactive.py output-parallel

Parallel runs::

$ mpiexec -n 3 python examples/multi_physics/biot_parallel_interactive.py output-
parallel -2 --shape=101,101

$ mpiexec -n 3 python examples/multi_physics/biot_parallel_interactive.py output-
parallel -2 --shape=101,101 --metis

$ mpiexec -n 8 python 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::

$ mpiexec -n 2 python examples/multi_physics/biot_parallel_interactive.py output-
parallel --shape=101,101 -snes_monitor -snes_converged_reason -ksp_monitor -pc_type_
--fieldsplit

$ mpiexec -n 8 python 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 (strip linearization or approximation orders one)::

$ python postproc.py output-parallel/sol.h5 --wireframe -b -d'p,plot_warp_scalar:u,
plot_displacements'

View the results using (adaptive linearization)::

$ python postproc.py output-parallel/sol_u.h5 --wireframe -b -d'u,plot_displacements'
"""

from __future__ import absolute_import
from argparse import RawDescriptionHelpFormatter, ArgumentParser
import os
import time

import numpy as nm

from sfeypy.base.base import output, Struct
from sfeypy.base.ioutils import ensure_path, remove_files_patterns, save_options
from sfeypy.discrete.fem import Mesh, FEDomain, Field
from sfeypy.discrete.common.region import Region
from sfeypy.discrete import (FieldVariable, Material, Integral, Function,
                             Equation, Equations, Problem, State)
from sfeypy.discrete.conditions import Conditions, EssentialBC
from sfeypy.terms import Term
from sfeypy.solvers.ls import PETScKrylovSolver
from sfeypy.solvers.nls import PETScNonlinearSolver
from sfeypy.mechanics.matcoefs import stiffness_from_lame

import sfeypy.parallel.parallel as pl

```

```
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]])

    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', gammal_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(ebc3=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)

    mesh = Mesh.from_file(mesh_filename)

    if rank == 0:
        cell_tasks = pl.partition_mesh(mesh, size, use_metis=options.metis,
                                       verbose=True)

    else:
        cell_tasks = None

    output('creating global domain and fields...')
    tt = time.clock()
    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]
    output('...done in', time.clock() - tt)

```

```
output('distributing fields...')
tt = time.clock()

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)

output('...done in', time.clock() - tt)

output('creating local problem...')
tt = time.clock()

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_variables()

state = State(variables)
state.fill(0.0)
state.apply_ebc()

output('...done in', time.clock() - tt)

output('allocating global system...')
tt = time.clock()

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)

output('...done in', time.clock() - tt)

output('creating solver...')
tt = time.clock()

conf = Struct(method='bcgsl', precond='jacobi', sub_precond='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)

output('...done in', time.clock() - tt)

output('solving...')
tt = time.clock()

state = pb.create_state()
state.apply_ebc()

ev.psol_i[...] = state()
ev.gather(psol, ev.psol_i)

psol = nls(psol)

ev.scatter(ev.psol_i, psol)
sol0_i = ev.psol_i[...]

output('...done in', time.clock() - tt)

output('saving solution...')
tt = time.clock()

state.set_full(sol0_i)
out = state.create_output_dict()

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)

    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)

output('...done in', time.clock() - tt)

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',
    '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('--2d', '--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('--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)
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')

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))])

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)

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)

solve_problem(mesh_filename, options, 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  $\alpha_{ij}$  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 `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:

```
python simple.py examples/multi_physics/biot_short_syntax.py -ksp_monitor
```

or simply run:

```
python simple.py 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`').

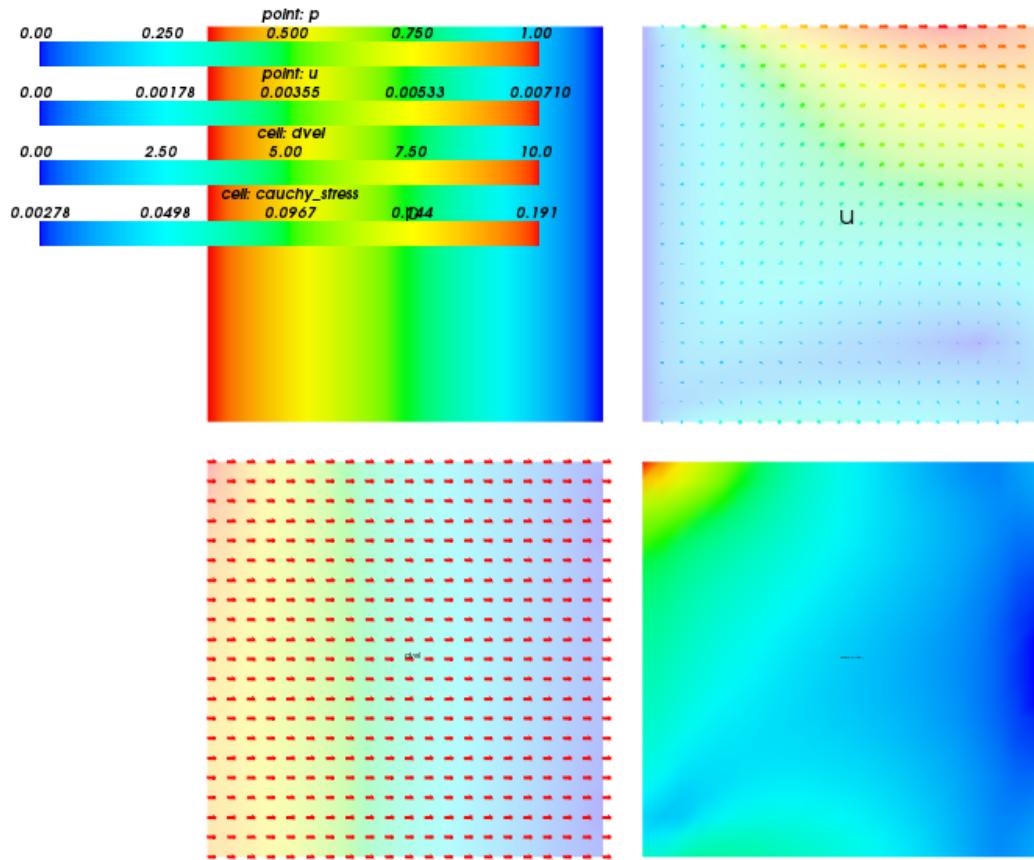
Find  $\underline{u}, p$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}) + \int_{\Gamma_{TB}} \varepsilon(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u}) = 0 , \quad \forall \underline{v} ,$$

$$- \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u}) - \int_{\Omega} K_{ij} \nabla_i q \nabla_j p = 0 , \quad \forall q ,$$

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 :math:`\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 example, set ``'ls' : 'iterative-p'`` in `options`, and run::

```
python simple.py examples/multi_physics/biot_short_syntax.py -ksp_monitor
```

or simply run::

```
python simple.py 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:`\nabla u = p` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\nabla v) e_{kl}(\nabla u)
    - \int_{\Omega} p \alpha_{ij} e_{ij}(\nabla v)
    + \int_{\Gamma_B} \nu \epsilon_{ij} (\nabla u)_i (\nabla v)_j = 0
    ;, \quad \forall v ;
    - \int_{\Omega} q \alpha_{ij} e_{ij}(\nabla u)
    - \int_{\Omega} K_{ij} \nabla_i q \cdot \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.
    """
    if mode == 'qp':
        n_nod, dim = coor.shape

        out = {}
        out['D'] = nm.tile(stiffness_from_lame(dim, lam=1.7, mu=0.3),
                            (coor.shape[0], 1, 1))

        alpha = [[0.132, 0.092],
                  [0.052, 0.132]]
        out['alpha'] = nm.tile(alpha, (coor.shape[0], 1, 1))

        perm = nm.eye(dim, dtype=nm.float64)
        out['K'] = nm.tile(perm, (coor.shape[0], 1, 1))

        out['np_eps'] = nm.tile(1e5, (coor.shape[0], 1, 1))

    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)

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""",
}
```

```

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]

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().
precond = Struct(name='precond', shape=mtx.shape, matvec=matvec_bj)
precond = spla.aslinearoperator(precond)

    return precond

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,
        # '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**

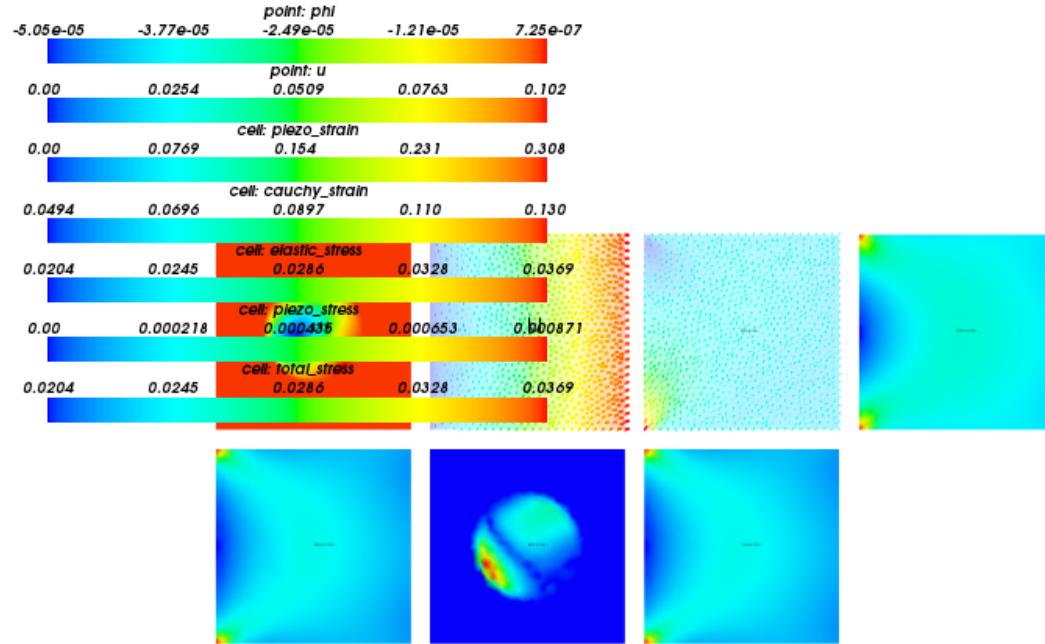
Piezo-elasticity problem - linear elastic material with piezoelectric effects.

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 :math:`\mathbf{u}`, :math:`\phi` such that:

.. math::
    \begin{aligned}
    & -\omega^2 \int_Y \rho \mathbf{u} \cdot \mathbf{v} \cdot \nabla \phi \\
    & + \int_Y D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    & - \int_{Y_2} g_{kij} e_{ij}(\mathbf{u}) \nabla_k \phi \\
    & = 0 \\
    & \quad \forall \mathbf{v}, \\
    & \int_{Y_2} g_{kij} e_{ij}(\mathbf{u}) \nabla_k \psi \\
    & + \int_Y K_{ij} \nabla_i \psi \nabla_j \phi \\
    & = 0 \\
    & \quad \forall \psi,
    \end{aligned}
    where
    .. math::
        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

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':
        n_nod, 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 = {
            # Lame 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' : 0.1142, # in 1e4 kg/m3
        }

        for key, val in six.iteritems(out):
            out[key] = nm.tile(val, (coor.shape[0], 1, 1))
        return out

materials = {
    'inclusion' : (None, 'get_inclusion_pars')
}
```

```

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

integrals = {
    'i' : 2,
}

equations = {
    '1' : """- %f * dw_volume_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,
                }),
}

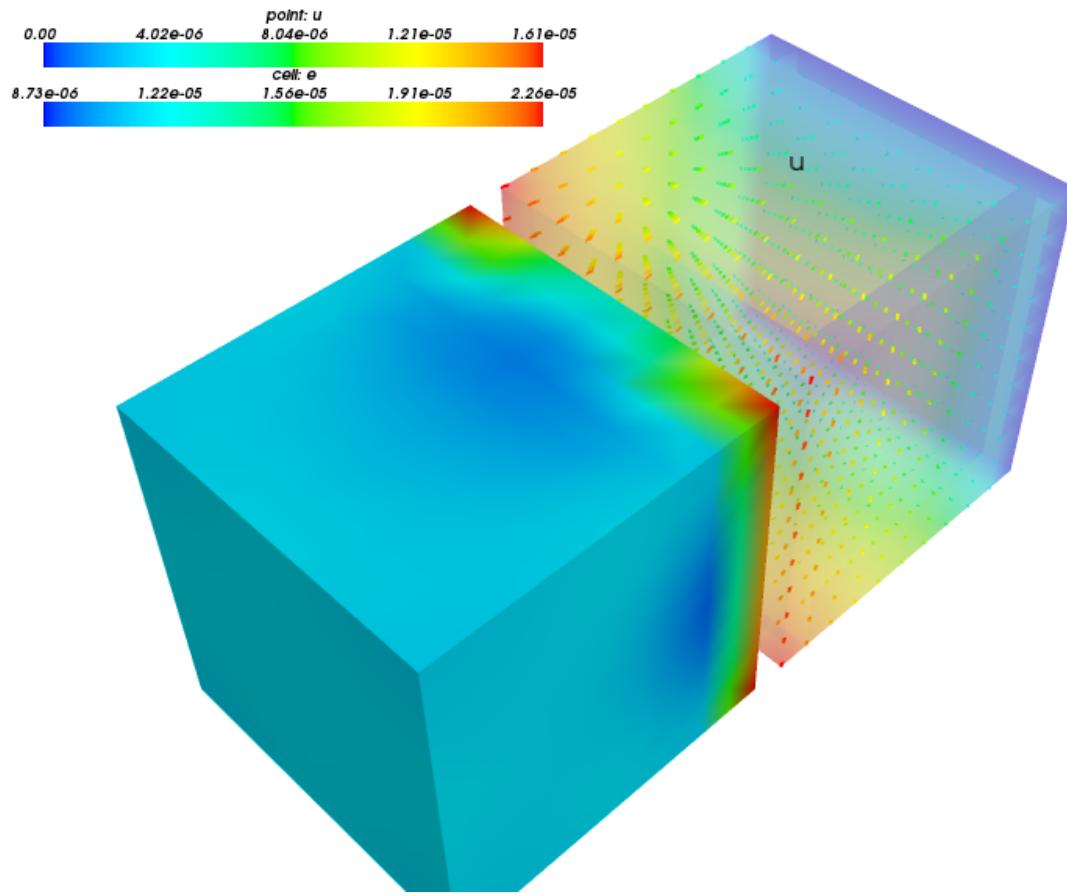
```

## 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](#)

```
r"""
Piezo-elasticity problem - homogenization of a piezoelectric linear elastic
matrix with embedded metalic 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
from sfepy.base.base import Struct
from sfepy.homogenization.micmac import get_homog_coefs_linear
import os.path as osp
from sfepy.homogenization.recovery import recover_micro_hook_eps
from sfepy.discrete.projections import make_l2_projection_data

def linear_projection(pb, data_qp):
    svar = pb.create_variables(['svar'])['svar']
    aux = []
    for ii in range(data_qp.shape[2]):
        make_l2_projection_data(svar, data_qp[..., ii, :].copy())
        aux.append(svar())
```

```

    return nm.ascontiguousarray(nm.array(aux).T)

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_parts()
    displ = state_dict['u']
    strain_qp = pb.evaluate('ev_cauchy_strain.i2.Omega(u)', mode='qp')

    nodal_data = {
        'u': displ.reshape((displ.shape[0] // dim, dim)), # displacement
        'strain': linear_projection(pb, strain_qp), # strain
    }
    const_data = {
        'phi': pb.conf.phi, # el. potentials
    }
    def_args = {
        'eps0': pb.conf.eps0,
        'filename_mesh': pb.conf.filename_mesh_micro,
    }
    pvar = pb.create_variables(['svar'])

    recover_micro_hook_eps(pb.conf.filename_micro, rreg,
                           pvar['svar'], nodal_data, const_data, pb.conf.eps0,
                           define_args=def_args)

    return out

def get_homog_fun(fname):
    return lambda ts, coors, mode=None, problem=None, **kwargs:\n        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', '.'),\n                             'coefs_piezo.h5')

    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():
    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 = data_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 = {
        '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'),
    }

    ebc_s = {
        '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 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 microscropic level
def recovery_micro(pb, corrs, macro):

```

```
eps0 = macro['eps0']
mesh = pb.domain.mesh
regions = pb.domain.regions
dim = mesh.dim
Ymc_map = regions['Ymc'].get_entities(0)
Ym_map = regions['Ym'].get_entities(0)
# 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]

for ii in range(dim):
    for jj in range(dim):
        kk = coor_to_sym(ii, jj, dim)
        phi += corrs['corrs_rs']['r%d%d' % (ii, jj)] \
            * nm.expand_dims(macro['strain'][Ym_map, kk], axis=1)
        u1 += corrs['corrs_rs']['u%d%d' % (ii, jj)] \
            * nm.expand_dims(macro['strain'][Ymc_map, kk], axis=1)

u = macro['u'][Ymc_map, :] + eps0 * u1
mvar = pb.create_variables(['u', 'r', 'svar'])
e_mac_Ymc = [None] * macro['strain'].shape[1]

for ii in range(dim):
    for jj in range(dim):
        kk = coor_to_sym(ii, jj, dim)
        mvar['svar'].set_data(macro['strain'][:, kk])
        mac_e_Ymc = pb.evaluate('ev_volume_integrate.i2.Ymc(svar)',
                                 mode='el_avg',
                                 var_dict={'svar': mvar['svar']})

        e_mac_Ymc[kk] = mac_e_Ymc.squeeze()

e_mac_Ymc = nm.vstack(e_mac_Ymc).T[:, nm.newaxis, :, nm.newaxis]

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 = {
    'u0': (macro['u'][Ymc_map, :], 'u', 'p'),
    '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)

    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'):

    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',
        'file_per_var': True,
        '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'),
    }

```

```
'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'),
}

epbc = {
    'p_ux': ([['Left', 'Right'], {'u.all': 'u.all'}, 'match_x_plane'],
              {'u.all': 'u.all'}, 'match_y_plane'),
    'p_uy': ([['Near', 'Far'], {'u.all': 'u.all'}, 'match_z_plane'],
              {'u.all': 'u.all'}, 'match_x_plane'),
    'p_uz': ([['Bottom', 'Top'], {'u.all': 'u.all'}, 'match_z_plane'],
              {'u.all': 'u.all'}, 'match_y_plane'),
    'p_rx': ([['Ym_left', 'Ym_right'], {'r.0': 'r.0'}, 'match_x_plane'],
              {'r.0': 'r.0'}, 'match_y_plane'),
    'p_ry': ([['Ym_near', 'Ym_far'], {'r.0': 'r.0'}, 'match_y_plane'],
              {'r.0': 'r.0'}, 'match_z_plane'),
    'p_rz': ([['Ym_bottom', 'Ym_top'], {'r.0': 'r.0'}, 'match_z_plane'],
              {'r.0': 'r.0'}, 'match_x_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, 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,),
    '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.scipy_direct', {}),
    '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.CoefSymSym,
    },
    '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.CoefSymSym,
    },
    'A': {
        'requires': ['c.A1', 'c.A2'],
        'expression': 'c.A1 + c.A2',
        'class': cb.CoefEval,
    },
    'vol': {
        'regions': ['Ym', 'Yc1', 'Yc2'],
        'expression': 'd_volume.i2.%s(svar)',
        'class': cb.VolumeFractions,
    },
    'eps0': {
        'requires': [],
        'expression': '%e' % eps0,
        'class': cb.CoefEval,
    },
    '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_i', 'nls': 'ns_ea6'},
},
}

# 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],
            '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'},
        },
    })

    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.CoefSym,
},
'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.CoefSym,
},
'V' + sk: {
    'requires': ['c.V1_' + sk, 'c.V2_' + sk],
    'expression': 'c.V1_%s - c.V2_%s' % (sk, sk),
    'class': cb.CoefEval,
},
})

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
```

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
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',
    'Omegal' : 'cells of group 1',
    'Omega2' : 'cells of group 2',
    'Omega2_Surface': ('r.Omegal *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}),
    '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_volume_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()

    # 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:

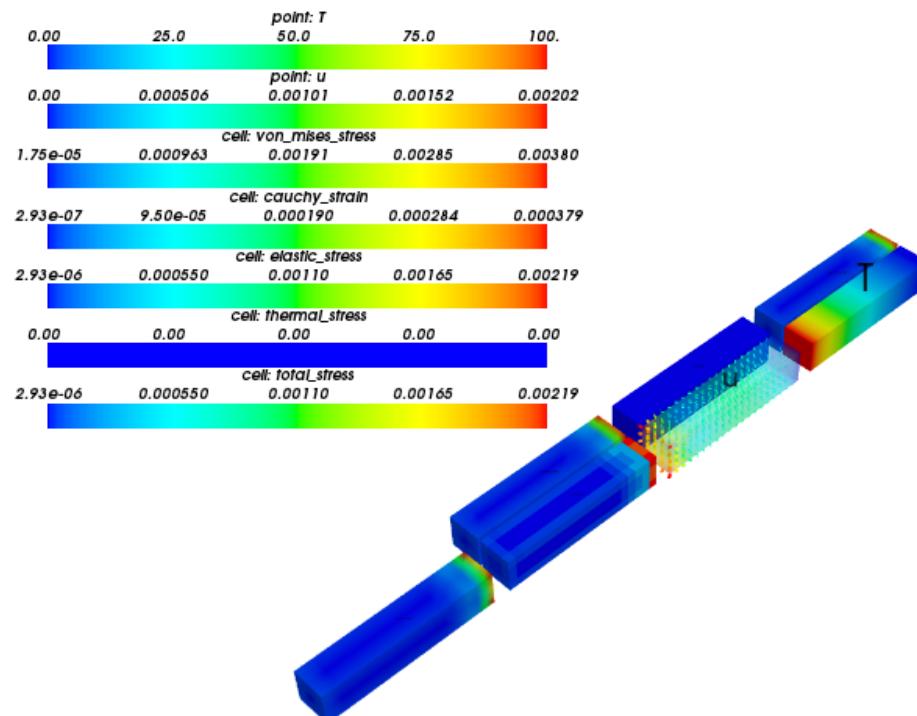
$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} (T - T_0) \alpha_{ij} e_{ij}(\underline{v}) = 0 , \quad \forall \underline{v} ,$$

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  $\alpha$  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}
    ;, \\

    \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):
    """
    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)

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]], 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""",
}

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

$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} ,$$

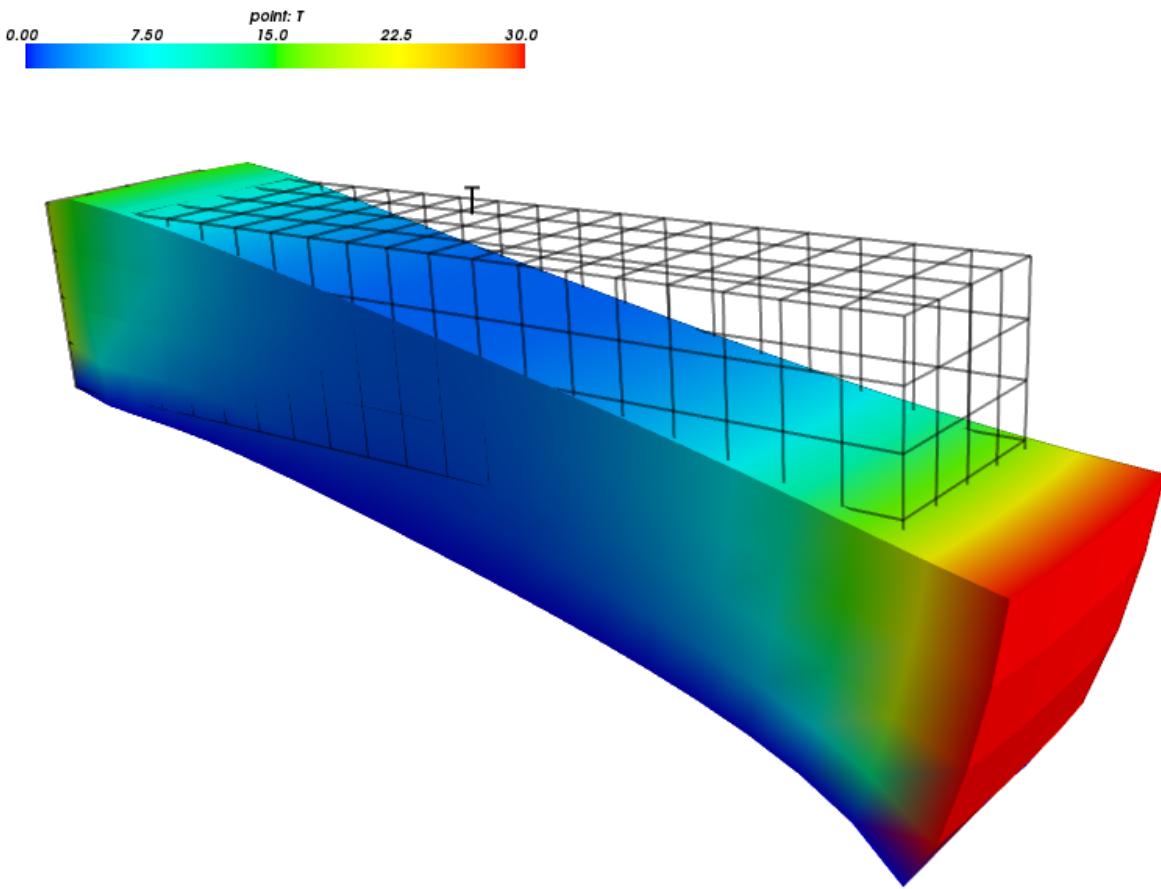
$$\alpha_{ij} = (3\lambda + 2\mu)\alpha\delta_{ij} ,$$

$T_0$  is the background temperature and  $\alpha$  is the thermal expansion coefficient.

### Notes

The gallery image was produced by (plus proper view settings):

```
./postproc.py block.vtk -d'u,plot_displacements,rel_scaling=1000,color_kind="scalars",
˓→color_name="T" --wireframe --only-names=u -b
```



[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:`\mathbf{u}`, :math:`T` such that:

```
.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \\
    - \int_{\Omega} (T - T_0) \alpha_{ij} e_{ij}(\mathbf{u}) \\
    = 0
    ;, \quad \forall \mathbf{v} ;,
    \int_{\Omega} \nabla s \cdot \nabla T \\
    = 0
    ;, \quad \forall s ;.
```

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} ;,
```

$T_0$  is the background temperature and  $\alpha$  is the thermal expansion coefficient.

Notes

-----

The gallery image was produced by (plus proper view settings)::

```
./postproc.py block.vtk -d'u,plot_displacements,rel_scaling=1000,color_kind=
→"scalars",color_name="T" --wireframe --only-names=u -b
"""
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),
}

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]], 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,
    }),
}
```

## 5.6.8 navier\_stokes

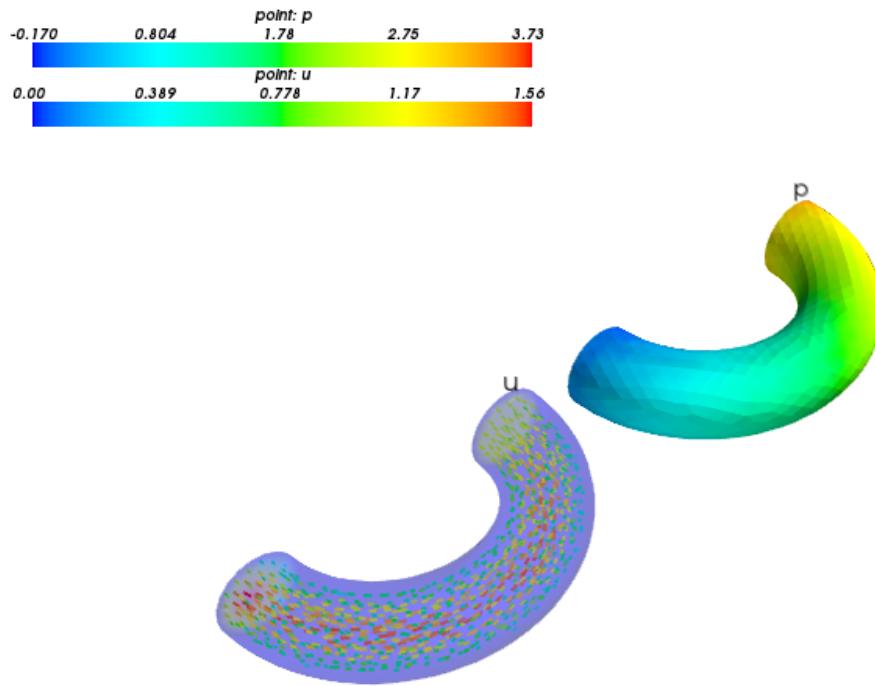
### navier\_stokes/navier\_stokes.py

#### 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:`\mathbf{u}`, :math:`p` such that:

.. math::
    \int_{\Omega} \nu \nabla \mathbf{u} : \nabla \mathbf{u} + (\mathbf{u} \cdot \nabla) \mathbf{u} \cdot \mathbf{v} - p \nabla \cdot \mathbf{u} \mathbf{v} = 0
    \quad \forall \mathbf{v},
    \int_{\Omega} q \nabla \cdot \mathbf{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',
}

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',
    '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.il.Omega( v, p ) = 0""",
'incompressibility' :
"""dw_stokes.il.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,
    'ls_on'        : 0.99999,
    'ls_min'       : 1e-5,
    'check'        : 0,
    'delta'        : 1e-6,
}

def verify_incompressibility( out, problem, state, 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_

    vv = problem.get_variables()
    one = nm.ones( (vv['p'].field.n_nod,), dtype = nm.float64 )
    vv['p'].set_data(one)
    zero = problem.evaluate('dw_stokes.il.Omega( u, p )', p=one, u=vv['u']())
    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 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:

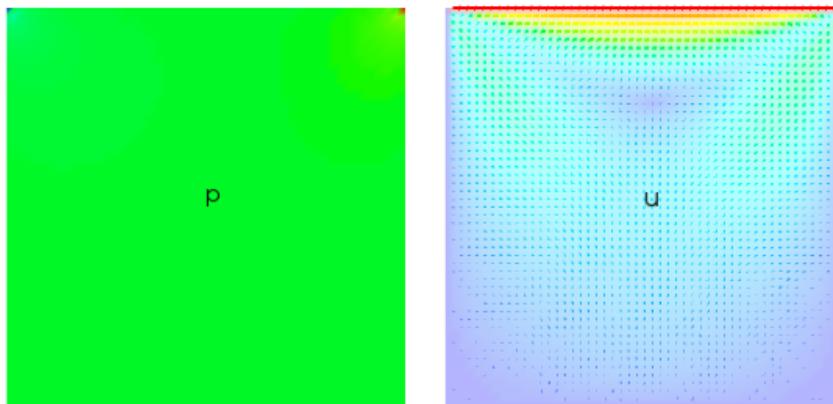
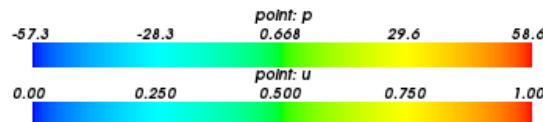
$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} = 0 , \quad \forall \underline{v} ,$$

$$\int_{\Omega} q \nabla \cdot \underline{u} = 0 , \quad \forall q .$$

The mesh is created by `gen_block_mesh()` function.

View the results using:

```
$ ./postproc.py user_block.vtk -b
```



[source code](#)

```
# -*- coding: utf-8 -*-
r"""
Navier-Stokes equations for incompressible fluid flow in 2D.
```

Find :math:`\nabla u` , :math:`p` such that:

```
.. math::
    \int_{\Omega} \nu \nabla u \cdot \nabla v : \nabla p = 0
    \int_{\Omega} ((\nabla u) \cdot \nabla v) \cdot \nabla p = 0
    \int_{\Omega} q \nabla u \cdot \nabla v = 0
```

The mesh is created by ``gen\_block\_mesh()`` function.

View the results using::

```
$ ./postproc.py user_block.vtk -b
"""
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'),
}

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:

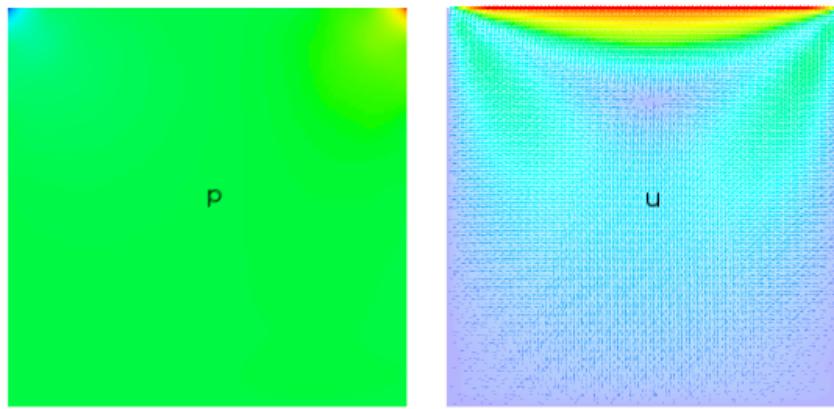
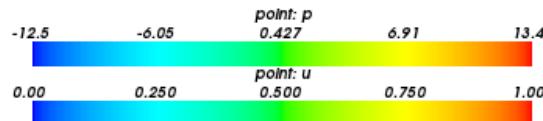
$$\begin{aligned} \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q. \end{aligned}$$

The domain geometry was created by:

```
$ ./script/gen_iga_patch.py -2 -d 0.1,0.1 -s 10,10 -o meshes/iga/block2d.iga
```

View the results using:

```
$ ./postproc.py block2d.vtk -b
```



[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 :math:`\mathbf{u}`, :math:`p` such that:

.. math::
    \int_{\Omega} \nu \nabla \cdot \mathbf{u} : \nabla \cdot \mathbf{u}
    + \int_{\Omega} ((\mathbf{u} \cdot \nabla) \mathbf{u}) \cdot \mathbf{u}
    - \int_{\Omega} p \nabla \cdot \mathbf{u} = 0
    \quad \forall \mathbf{v} ,
    \int_{\Omega} q \nabla \cdot \mathbf{u} = 0
    \quad \forall q .

```

The domain geometry was created by::

```
$ ./script/gen_iga_patch.py -2 -d 0.1,0.1 -s 10,10 -o meshes/iga/block2d.iga

View the results using::

$ ./postproc.py block2d.vtk -b
"""
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' :
        """dw_stokes.i.Omega(u, q) = 0""",
}

solvers = {
```

```
'ls' : ('ls.scipy_direct', {}),
'nls' : ('nls.newton', {
    'i_max'      : 15,
    'eps_a'       : 1e-10,
    'eps_r'       : 1.0,
}),
}
```

## navier\_stokes/stabilized\_navier\_stokes.py

### Description

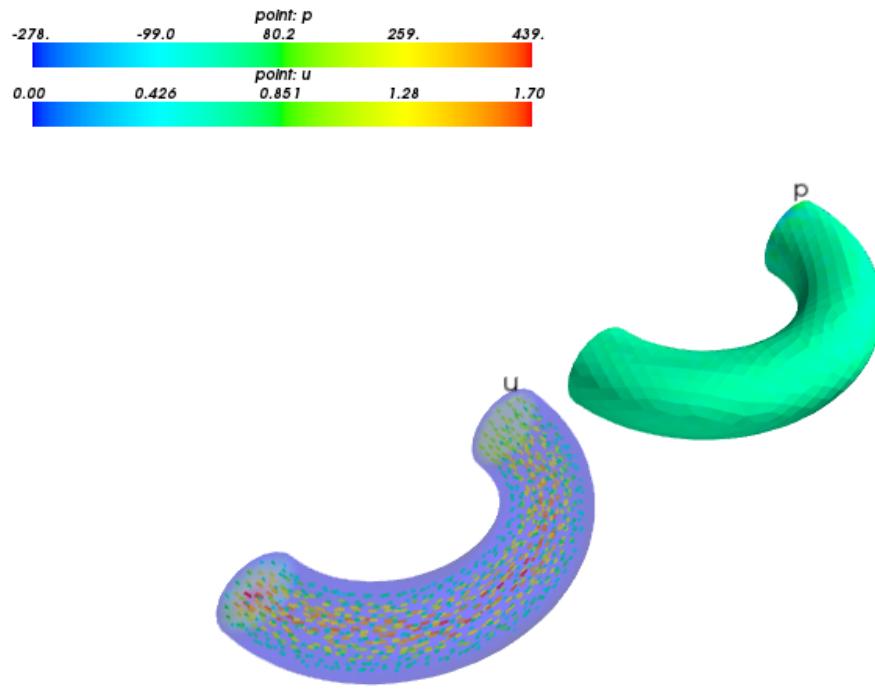
Stabilized Navier-Stokes problem with grad-div, SUPG and PSPG stabilization solved by a custom Oseen solver.

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:

$$\begin{aligned} & \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} \\ & + \gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v}) \\ & + \sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v}) \\ & + \sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v}) = 0, \quad \forall \underline{v}, \\ & \int_{\Omega} q \nabla \cdot \underline{u} \\ & + \sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q \\ & + \sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q = 0, \quad \forall q. \end{aligned}$$



[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 :math:`\mathbf{u}`, :math:`p` such that:

```
.. math::
\begin{array}{l}
\int_{\Omega} \nabla \cdot \mathbf{u} \cdot \nabla \mathbf{v} + \int_{\Omega} (\mathbf{u} \cdot \nabla) \mathbf{v} + \int_{\Omega} p \nabla \cdot \mathbf{v} \\
- \int_{\Omega} \mathbf{b} \cdot \nabla \mathbf{v} + \gamma \int_{\Omega} (\nabla \mathbf{u} \cdot \mathbf{v}) + \sum_K \int_{T_K} \delta_K ((\mathbf{u} \cdot \nabla) \mathbf{v}) \\
+ \sum_K \int_{T_K} \delta_K (\nabla p \cdot \mathbf{v}) = 0
\end{array}
```

```
\end{array}

\begin{array}{l}
\int_{\Omega} q \nabla \cdot \mathbf{u} \Omega \cdot \nabla q \\
+ \sum_K \int_{\Gamma_h} \int_{T_K} (\mathbf{u}_b \cdot \nabla) \nabla p \cdot \nabla q \\
= 0
\quad \text{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}),
}

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',
}

## 
# Functions.
import os.path as op
import sys

sys.path.append(data_dir) # Make installed example work.
import 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

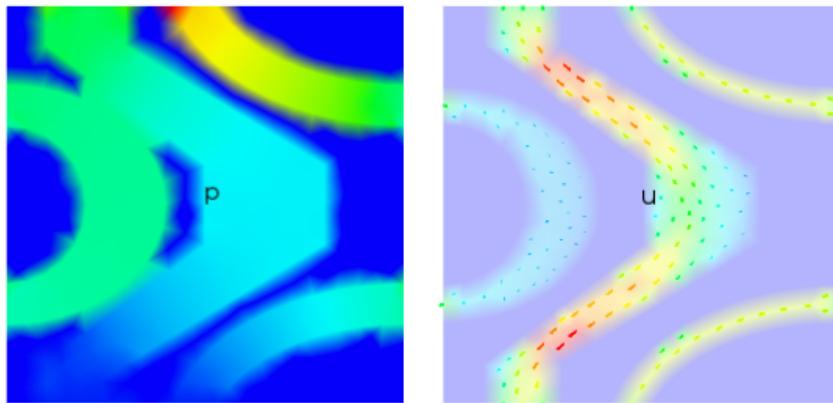
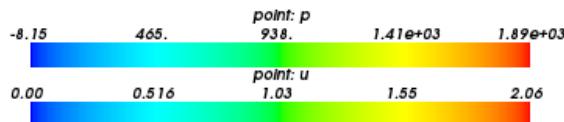
Stokes equations for incompressible fluid flow.

This example demonstrates fields defined on subdomains as well as use of periodic boundary conditions.

Find  $\underline{u}, p$  such that:

$$\int_{Y_1 \cup Y_2} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{Y_1 \cup Y_2} p \nabla \cdot \underline{v} = 0 , \quad \forall \underline{v} ,$$

$$\int_{Y_1 \cup Y_2} q \nabla \cdot \underline{u} = 0 , \quad \forall q .$$



[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 :math:`\nabla u` and :math:p` such that:

.. math::
    \int_{Y_1 \cup Y_2} \nu \nabla \cdot \nabla u : \nabla \cdot \nabla u
    - \int_{Y_1 \cup Y_2} p \nabla \cdot \nabla \cdot \nabla u = 0
    ;, \quad \forall v
    , \quad \forall q

    \int_{Y_1 \cup Y_2} q \nabla \cdot \nabla \cdot \nabla u = 0
    ;, \quad \forall q
    .

"""

from __future__ import absolute_import
from sfepy import data_dir
from sfepy.discrete.fem.periodic import match_y_line

filename_mesh = data_dir + '/meshes/2d/special/channels_symm944t.mesh'

if filename_mesh.find('symm') >= 0:
    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',
    '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,
}
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_r1',
    'region' : ['Left', 'Right'],
    'dofs' : {'u.all' : 'u.all', 'p.0' : 'p.0'},
}

```

```
'match' : 'match_y_line',
}

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* boundary conditions as well as *edge direction* boundary conditions together with Navier or slip boundary conditions.

Find  $\underline{u}, p$  such that:

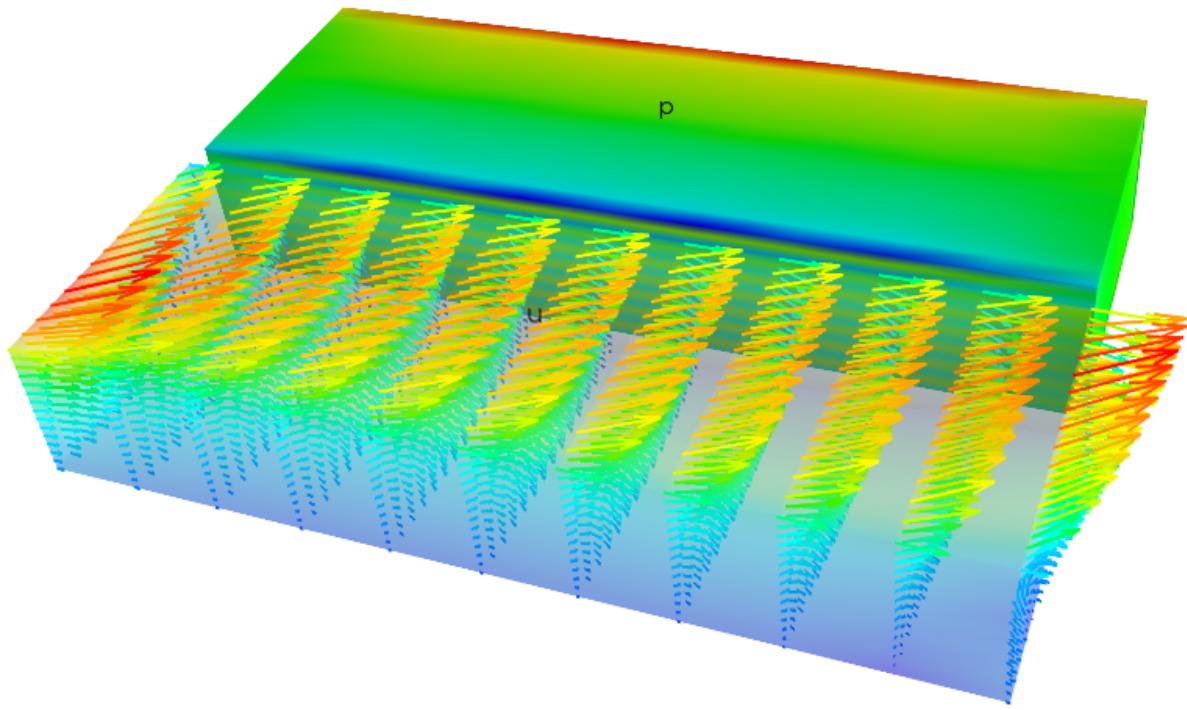
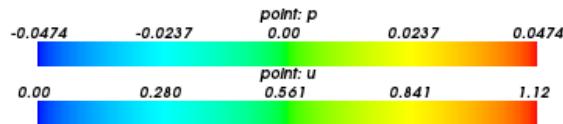
$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{\Omega} p \nabla \cdot \underline{v} + \int_{\Gamma_1} \beta \underline{v} \cdot (\underline{u} - \underline{u}_d) + \int_{\Gamma_2} \beta \underline{v} \cdot \underline{u} = 0, \quad \forall \underline{v},$$
$$\int_{\Omega} \mu \nabla q \cdot \nabla p + \int_{\Omega} q \nabla \cdot \underline{u} = 0, \quad \forall q,$$

where  $\nu$  is the fluid viscosity,  $\beta$  is the slip coefficient,  $\mu$  is the (small) numerical diffusion coefficient,  $\Gamma_1$  is the top wall that moves with the given driving velocity  $\underline{u}_d$  and  $\Gamma_2$  are the remaining walls. The Navier conditions are in effect on both  $\Gamma_1, \Gamma_2$  and are expressed by the corresponding integrals in the equations above.

The *no-penetration* boundary conditions are applied on  $\Gamma_1, \Gamma_2$ , except the vertices of the block edges, where the *edge direction* boundary conditions are applied. Optionally, Dirichlet boundary conditions can be applied on the inlet, see the code below.

The mesh is created by `gen_block_mesh()` function - try different mesh dimensions and resolutions below. For large meshes use the '`ls_i`' linear solver - PETSc + petsc4py is needed in that case.

See also [navier\\_stokes/stokes\\_slip\\_bc\\_penalty.py](#).



[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’ boundary conditions as well as ‘edge direction’ boundary conditions together with Navier or slip boundary conditions.

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

```
.. math::
    \int_{\Omega} \nu \nabla \cdot \mathbf{u} : \nabla \cdot \mathbf{u}
    - \int_{\Omega} p \nabla \cdot \mathbf{u}
    + \int_{\Gamma_1} \beta \mathbf{u} \cdot \mathbf{n} (\mathbf{u} - \mathbf{u}_d)
    + \int_{\Gamma_2} \beta \mathbf{u} \cdot \mathbf{n} \mathbf{u}
    = 0
    ;, \quad \forall \mathbf{v} ;,
    \int_{\Omega} \mu \nabla q \cdot \nabla p
    + \int_{\Omega} q \nabla \cdot \mathbf{u}
    = 0
    ;, \quad \forall q ;,
```

where :math:`\nu` is the fluid viscosity, :math:`\beta` is the slip

coefficient, :math:`\mu` is the (small) numerical diffusion coefficient, :math:`\Gamma\_1` is the top wall that moves with the given driving velocity :math:`u\_d` and :math:`\Gamma\_2` are the remaining walls. The Navier conditions are in effect on both :math:`\Gamma\_1` and :math:`\Gamma\_2` and are expressed by the corresponding integrals in the equations above.

The `no-penetration` boundary conditions are applied on :math:`\Gamma\_1` and :math:`\Gamma\_2`, except the vertices of the block edges, where the `edge direction` boundary conditions are applied. Optionally, Dirichlet boundary conditions can be applied on the inlet, see the code below.

The mesh is created by ``gen\_block\_mesh()`` function - try different mesh dimensions and resolutions below. For large meshes use the ``'ls\_i'`` linear solver - PETSc + petsc4py is needed in that case.

See also :ref:`navier\_stokes-stokes\_slip\_bc\_penalty`.

```
"""
from __future__ import absolute_import
import numpy as nm

from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.homogenization.utils import define_box_regions

# Mesh dimensions.
dims = nm.array([3, 1, 0.5])

# Mesh resolution: increase to improve accuracy.
shape = [11, 15, 15]

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'),
    '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', 1),
    'pressure' : ('real', 1, 'Omega', 1),
}

def get_u_d(ts, coors, region=None):
    """
    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'),
}
# Try setting the inlet velocity by un-commenting the 'inlet' ebc.
ebcs = {
    ## 'inlet' : ('Inlet_f', {'u.0' : 1.0, 'u.[1, 2]' : 0.0}),
}

lcbcs = {
    'walls' : ('Gamma_v', {'u.all' : None}, None, 'no_penetration',
               'normals_Gamma.vtk'),
    'edges' : ('Edges_v', [(-0.5, 1.5)], {'u.all' : None}, None,
               'edge_direction', 'edges_Edges.vtk'),
}
materials = {
    'm' : ({
        'nu' : 1e-3,
        'beta' : 1e-2,
        'mu' : 1e-10,
    },),
}
equations = {
    'balance' :
    """dw_div_grad.5.Omega(m.nu, v, u)
    - dw_stokes.5.Omega(v, p)
    + dw_surface_dot.5.Gamma1_f(m.beta, v, u)
    + dw_surface_dot.5.Gamma2_f(m.beta, v, u)
    =
    + dw_surface_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""",
}

```

```
}

solvers = {
    'ls_d' : ('ls.scipy_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' : 2500, # maxits
    }),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
    }),
}
}

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

## navier\_stokes/stokes\_slip\_bc\_penalty.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 a weak application of *no-penetration* boundary conditions using the penalty term `dw_non_penetration_p`.

Find  $\underline{u}, p$  such that:

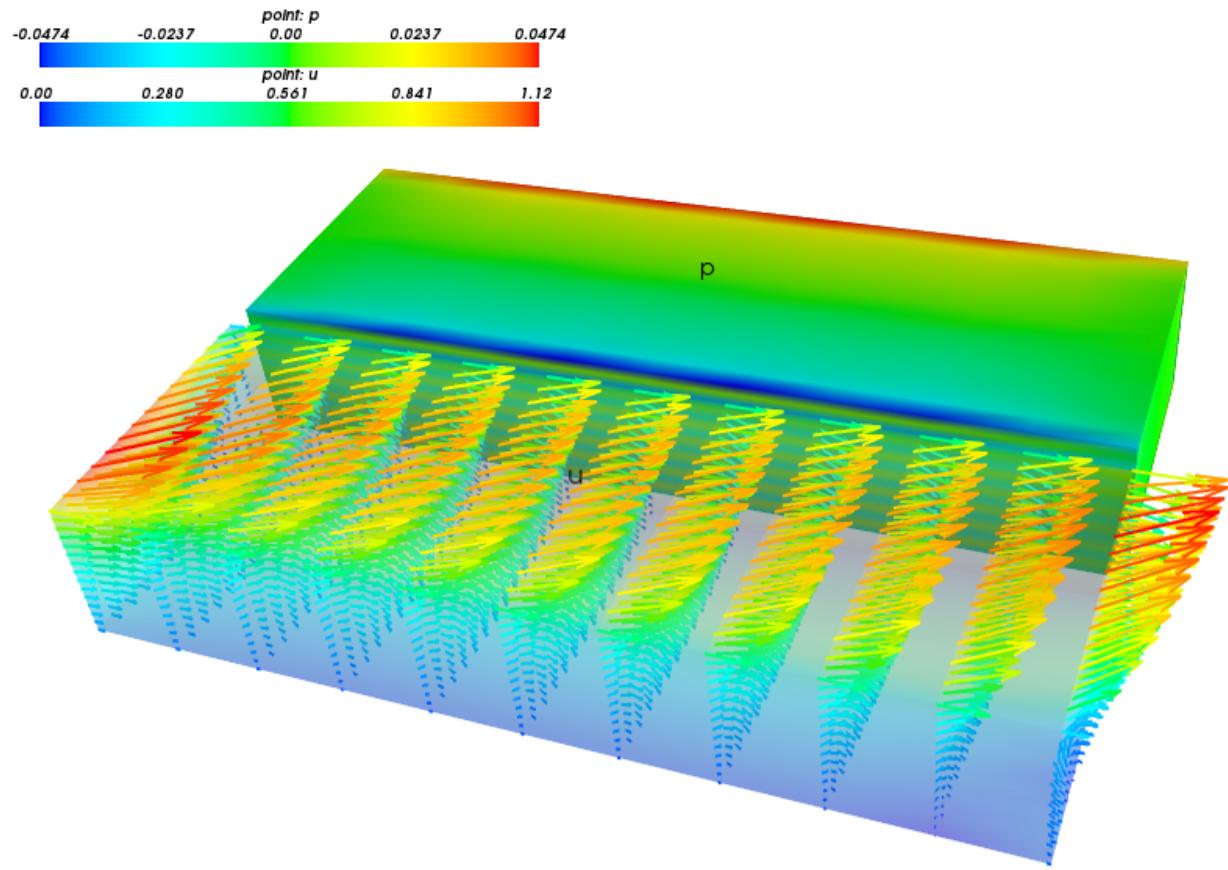
$$\begin{aligned} \int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} - \int_{\Omega} p \nabla \cdot \underline{v} + \int_{\Gamma_1} \beta \underline{v} \cdot (\underline{u} - \underline{u}_d) + \int_{\Gamma_2} \beta \underline{v} \cdot \underline{u} + \int_{\Gamma_1 \cup \Gamma_2} \epsilon (\underline{n} \cdot \underline{v}) (\underline{n} \cdot \underline{u}) &= 0, \quad \forall \underline{v}, \\ \int_{\Omega} \mu \nabla q \cdot \nabla p + \int_{\Omega} q \nabla \cdot \underline{u} &= 0, \quad \forall q, \end{aligned}$$

where  $\nu$  is the fluid viscosity,  $\beta$  is the slip coefficient,  $\mu$  is the (small) numerical diffusion coefficient,  $\epsilon$  is the penalty coefficient (sufficiently large),  $\Gamma_1$  is the top wall that moves with the given driving velocity  $\underline{u}_d$  and  $\Gamma_2$  are the remaining walls. The Navier conditions are in effect on both  $\Gamma_1$ ,  $\Gamma_2$  and are expressed by the corresponding integrals in the equations above.

The *no-penetration* boundary conditions are applied on  $\Gamma_1$ ,  $\Gamma_2$ . Optionally, Dirichlet boundary conditions can be applied on the inlet, see the code below.

The mesh is created by `gen_block_mesh()` function - try different mesh dimensions and resolutions below. For large meshes use the '`ls_i`' linear solver - PETSc + petsc4py is needed in that case.

See also [navier\\_stokes/stokes\\_slip\\_bc.py](#).



[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 a weak application of `no-penetration` boundary
conditions using the penalty term ``dw_non_penetration_p``.
```

Find  $\mathbf{u}$ ,  $p$  such that:

```
.. math::

\int_{\Omega} \nu \nabla \cdot \mathbf{u} : \nabla \cdot \mathbf{u}
- \int_{\Omega} p \nabla \cdot \mathbf{u}
+ \int_{\Gamma_1} \beta \mathbf{u} \cdot \mathbf{n} (\mathbf{u} - \mathbf{u}_d)
+ \int_{\Gamma_2} \beta \mathbf{u} \cdot \mathbf{n} \mathbf{u}
+ \int_{\Gamma_1 \cup \Gamma_2} \epsilon (\mathbf{n} \cdot \mathbf{u}) (\mathbf{n} \cdot \mathbf{u})
= 0
;,
\quad \forall \mathbf{v} ;,

\int_{\Omega} \mu \nabla q \cdot \nabla p
+ \int_{\Omega} q \nabla \cdot \mathbf{u}
= 0
;,
\quad \forall q ;,
```

where  $\nu$  is the fluid viscosity,  $\beta$  is the slip

```
coefficient, :math:`\mu` is the (small) numerical diffusion coefficient,  
:math:`\epsilon` is the penalty coefficient (sufficiently large),  
:math:`\Gamma_1` is the top wall that moves with the given driving velocity  
:math:`\Gamma_{u_d}` and :math:`\Gamma_2` are the remaining walls. The Navier  
conditions are in effect on both :math:`\Gamma_1`, :math:`\Gamma_2` and are  
expressed by the corresponding integrals in the equations above.
```

The ‘no-penetration’ boundary conditions are applied on :math:`\Gamma\_1` ,  
:math:`\Gamma\_2` . Optionally, Dirichlet boundary conditions can be applied on  
the inlet, see the code below.

The mesh is created by ``gen\_block\_mesh()`` function - try different mesh  
dimensions and resolutions below. For large meshes use the `'''ls\_i'''` linear  
solver - PETSc + petsc4py is needed in that case.

See also :ref:`navier\_stokes-stokes\_slip\_bc` .

```
"""  
from __future__ import absolute_import  
import numpy as nm  
  
from sfepy.discrete.fem.meshio import UserMeshIO  
from sfepy.mesh.mesh_generators import gen_block_mesh  
from sfepy.homogenization.utils import define_box_regions  
  
# Mesh dimensions.  
dims = nm.array([3, 1, 0.5])  
  
# Mesh resolution: increase to improve accuracy.  
shape = [11, 15, 15]  
  
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',  
    'Gamma1_f' : ('copy r.Top', 'face'),  
    'Gamma2_f' : ('r.Near +v r.Bottom +v r.Far', 'face'),  
    'Gamma_f' : ('r.Gamma1_f +v r.Gamma2_f', 'face'),  
    'Inlet_f' : ('r.Left -v r.Gamma_f', 'face'),  
})  
  
fields = {  
    'velocity' : ('real', 3, 'Omega', 1),  
    'pressure' : ('real', 1, 'Omega', 1),  
}
```

```

def get_u_d(ts, coors, region=None) :
    """
    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'),
}
}

# Try setting the inlet velocity by un-commenting the 'inlet' ebc.
ebcs = {
    ## 'inlet' : ('Inlet_f', {'u.0' : 1.0, 'u.[1, 2]' : 0.0}),
}

materials = {
    'm' : ({
        'nu' : 1e-3,
        'beta' : 1e-2,
        'mu' : 1e-10,
        'np_eps' : 1e3,
    },),
}

equations = {
    'balance' :
        """dw_div_grad.5.Omega(m.nu, v, u)
        - dw_stokes.5.Omega(v, p)
        + dw_surface_dot.5.Gamma1_f(m.beta, v, u)
        + dw_surface_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_surface_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""",
}
}

solvers = {
    'ls_d' : ('ls.scipy_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' : 1000, # maxits
}),
'newton' : ('nls.newton', {
    'i_max' : 1,
    'eps_a' : 1e-10,
}),
}

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

## 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)
```

## 5.6.9 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.ioutils import InDir
from sfepy.homogenization.coefficients import Coefficients

from 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'

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',
                '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',
                 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()
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.scipy_direct', {}),
        '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),
    }

```

```
        '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):
    ebc = {
        'fixed_corners' : ('Corners', {'u_m.all' : 0.0}),
        'fixed_gamma_mc' : ('Gamma_mc', {'u_c.all' : 0.0}),
    }

    epbc = {}
    all_periodic = []
    for vn in ['u_m']:
        val = {'%s.all' % vn : '%s.all' % vn}

        epbc.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])

    lcbc = {}

    return ebc, epbc, lcbc, 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 = {
        '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_volume_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' : """d_volume.i.%s(aux)""",
    'ema' : """ev_volume_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'})

    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,
            '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.CoefSymSym,
    },
    'Gamma' : {
        'requires' : ['c.D'],
        'options' : {
            'mode' : 'simple',
            'incident_wave_dir' : None,
        },
        'class' : cp.ChristoffelAcousticTensor,
    },
    'dispersion' : {
        'requires' : ['evp', 'c.eigenmomenta', 'c.M', 'c.Gamma'],
        'options' : self.band_gaps_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

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,
    },
}
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,
})
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}),
}
lcbs = {
    'rigid' : ('Y_r', {'u.all' : None}, None, 'rigid'),
}

```

```
    return ebcsl, {}, 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_volume_dot.i.Y_c( m.density_c, v, u )
                  + dw_volume_dot.i.Y_r( m.density_r, v, u )""",
    }

    expr_coefs = {
        'VF' : """d_volume.i.%s(aux)""",
        'ema' : """ev_volume_integrate.i.Y_c( m.density_c, u )
                  + ev_volume_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'],
            'epbcs' : None,
            'lcbc' : ['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 :math:`Y\_m` and the soft inclusion :math:`Y\_c` enclosing the rigid heavy sub-inclusion*

```
:math:`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 examples.phononic.band_gaps_conf import (BandGapsRigidConf, get_pars,  
                                                normalize)  
  
normalize # 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,
}

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' : ('normalize', (-2, 2)),

    '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):
    var = problem.get_variables()['u']

    for key in out.keys():
        ii = int(key[1:])
        vec = mtx_phi[:,ii].copy()
        var.set_data(vec)

```

```
strain = problem.evaluate('ev_cauchy_strain.i.Y_c(u)', u=var,
                           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,
                                  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()
```

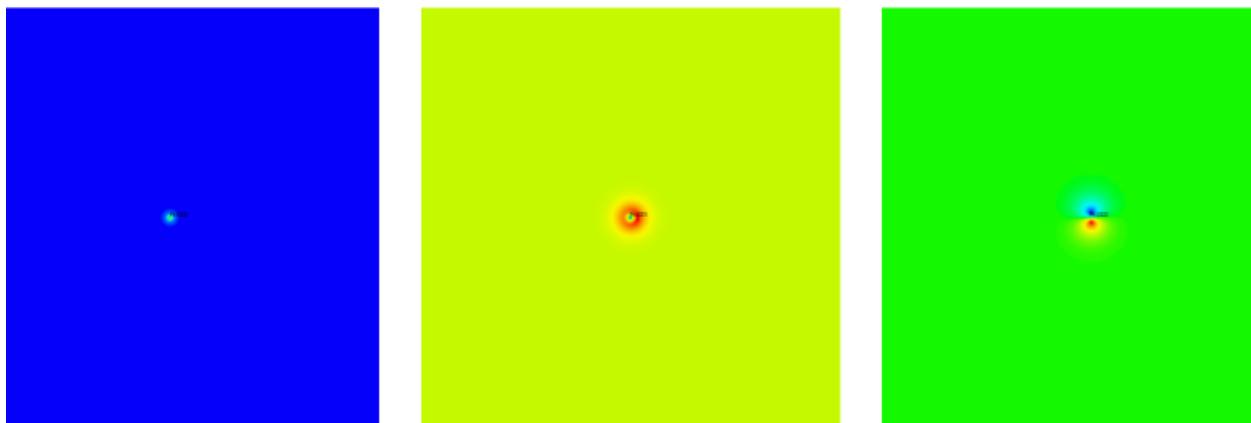
## 5.6.10 quantum

### 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 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

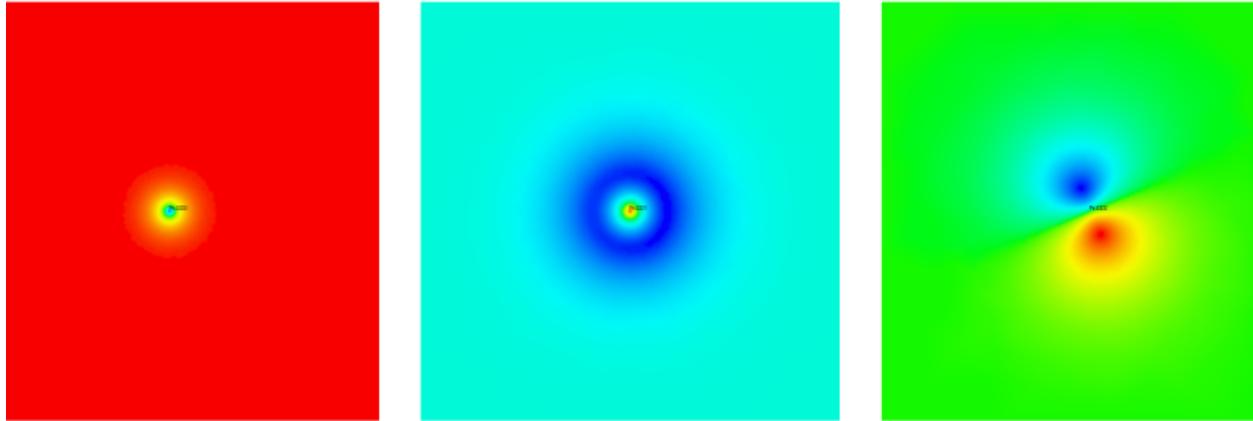
```

## 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_12_along_axis

from 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

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

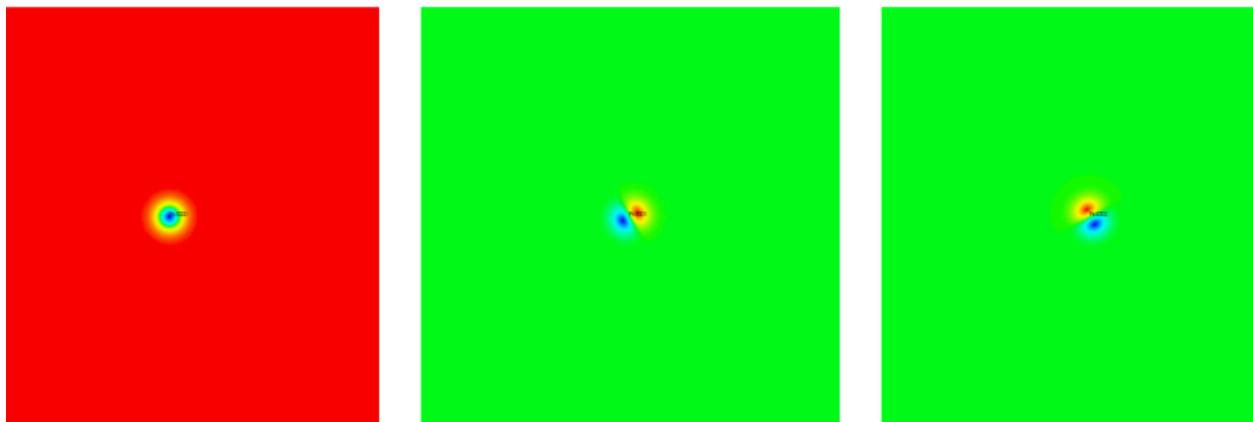
```

## 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 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

    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 the `simple.py` script:

```
python simple.py examples/quantum/boron.py
python simple.py examples/quantum/hydrogen.py
python simple.py examples/quantum/oscillator.py
python simple.py 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 the `simple.py` script::

```
python simple.py examples/quantum/boron.py
python simple.py examples/quantum/hydrogen.py
python simple.py examples/quantum/oscillator.py
python simple.py 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)

    output('n      exact      FEM      error')
    for ie, eig in enumerate(epvp.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(epvp.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}),
}
```

```

}

integrals = {
    'i' : 2 * approx_order,
}

equations = {
    'lhs' : """dw_laplace.i.Omega(m.val, v, Psi)
              + dw_volume_dot.i.Omega(mat_v.V, v, Psi)""",
    'rhs' : """dw_volume_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()

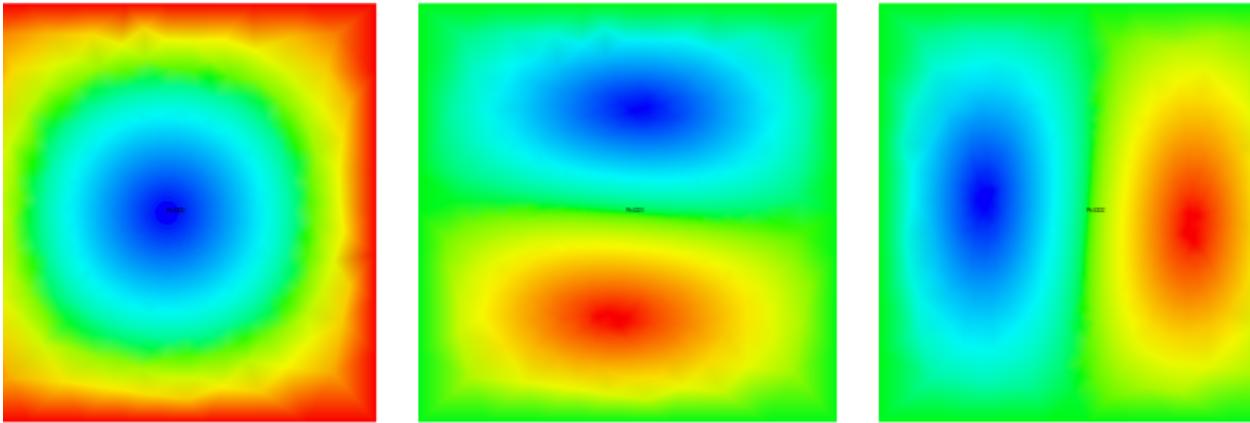
```

## 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 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]]


    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
```

## 5.7 Example Applications

(Note: the examples below were created around 2010 and do not reflect the current state of SfePy).

- Fish heart model: [https://sfepy.org/fish\\_heart](https://sfepy.org/fish_heart)
- Phononic materials: <https://sfepy.org/phononic>



---

CHAPTER  
SIX

---

## 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:

### 6.1 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.

#### 6.1.1 Strong form of Poisson's equation and its integration

Let us start our discussion about FEM with the strong form of Poisson's equation

$$\Delta T = f(x), \quad x \in \Omega, \quad (6.1)$$

$$T = u(x), \quad x \in \Gamma_D, \quad (6.2)$$

$$\nabla T \cdot \mathbf{n} = g(x), \quad x \in \Gamma_N, \quad (6.3)$$

where  $\Omega \subset \mathbb{R}^n$  is the solution domain with the boundary  $\partial\Omega$ ,  $\Gamma_D$  is the part of the boundary where Dirichlet boundary conditions are given,  $\Gamma_N$  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 (6.1) is

$$\int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = 0,$$

where  $s$  is a **test** function. Integrating this equation by parts

$$0 = \int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = \int_{\Omega} \nabla \cdot (\nabla T) \cdot s \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega =$$

$$= - \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega + \int_{\Omega} \nabla \cdot (\nabla T \cdot s) \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega$$

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

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_D} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma + \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega. \quad (6.4)$$

The equation (6.4) is the initial weak form of the Poisson's problem (6.1)–(6.3). 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  $\Gamma_D$  in equation (6.4). 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

$$V(\Omega) = \{f(x) \in H^1(\Omega)\},$$

$$V_0(\Omega) = \{f(x) \in H^1(\Omega); f(x) = 0, x \in \Gamma_D\}.$$

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  $\Gamma_D$ .

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

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega,$$

$$T(x) = u(x), \quad x \in \Gamma_D.$$

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 (6.4) 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.

### 6.1.2 The weak form of the Poisson's equation

Now we can write the resulting weak form for the Poisson's problem (6.1)–(6.3). For any test function  $s \in V_0(\Omega)$  find  $T \in V(\Omega)$  such that

$$\begin{aligned} \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega &= \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \quad \text{and} \\ T(x) &= u(x), \quad x \in \Gamma_D. \end{aligned} \tag{6.5}$$

### 6.1.3 Discussion of discretization and meshing

It is planned to have an example of the discretization based on the Poisson's equation weak form (6.5). For now, please refer to the wikipedia page [Finite Element Method](#) for a basic description of the discretization and meshing.

### 6.1.4 Numerical solution of the problem

To solve numerically given problem based on the weak form (6.5) we have to go through 5 steps:

1. Define geometry of the domain  $\Omega$  and surfaces  $\Gamma_D$  and  $\Gamma_N$ .
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  $\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega$ ,  $\int_{\Gamma_N} g \cdot s \, d\Gamma$ ,  $\int_{\Omega} f \cdot s \, d\Omega$ .

## 6.2 Implementation of Essential Boundary Conditions

The essential boundary conditions can be applied in several ways. Here we describe the implementation used in SfePy.

### 6.2.1 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:

$$T^T A T x_r = T^T (b - A x_f)$$

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

$$T^T A T x_r = T^T b ,$$

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.

## 6.2.2 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  $\Delta u$  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

$$f(u) \equiv Au - b = 0 ,$$

$$\frac{df}{du} = A ,$$

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  $u_r$ .



## DEVELOPER GUIDE

### Table of Contents

- *Retrieving the Latest Code*
- *SfePy Directory Structure*
- *Exploring the Code*
- *How to Contribute*
  - *Reporting problems*
  - *Contributing changes*
    - \* *Using git and github*
    - \* *Notes on commits and patches*
    - \* *Without using git*
  - *Coding style*
    - \* *Docstring standard*
- *How to Regenerate Documentation*
- *How to Implement a New Term*
  - *Notes on terminology*
  - *Introduction*
  - *Evaluation modes*
  - *Basic attributes*
    - \* *Argument types*
    - \* *Integration kinds*
    - \* *function()*
    - \* *get\_fargs()*
  - *Additional attributes*
    - \* *Argument shapes*
    - \* *Geometries*
  - *Examples*

- *Concluding remarks*
- *How To Make a Release*
- *Module Index*
  - *Main scripts*
  - *Utility scripts*
  - *sfePy package*
  - *sfePy.applications package*
  - *sfePy.base package*
  - *sfePy.discrete package*
    - \* *sfePy.discrete.common sub-package*
    - \* *sfePy.discrete.fem sub-package*
    - \* *sfePy.discrete.iga sub-package*
    - \* *sfePy.discrete.structural sub-package*
  - *sfePy.homogenization package*
  - *sfePy.linalg package*
  - *sfePy.mechanics package*
  - *sfePy.mesh package*
  - *sfePy.parallel package*
  - *sfePy.postprocess package*
  - *sfePy.solvers package*
  - *sfePy.terms package*

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.

## 7.1 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.

## 7.2 SfePy Directory Structure

Here we list and describe the directories that are in the main sfepy directory.

Table 7.1: Top directory structure.

name	description
<i>build/</i>	directory created by the build process (generated)
<i>doc/</i>	source files of this documentation
<i>examples/</i>	example problem description files
<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>output-tests/</i>	output directory for tests
<i>script/</i>	various small scripts (simple mesh generators, mesh format converters etc.)
<i>sfepy/</i>	the source code
<i>tests/</i>	the tests run by <i>run_tests.py</i>
<i>tmp/</i>	directory for temporary files (generated)

New users/developers (after going through the *Tutorial*) should explore the *examples/* directory. For developers, the principal directory is *sfepy/*, which has the following contents:

Table 7.2: *sfepy/* directory structure.

name	description	field-specific
<i>application/</i>	top level application classes (e.g. PDESolverApp that implements all that <i>simple.py</i> script does)	
<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: <ul style="list-style-type: none"><li>• <i>common/</i> - common parent classes for discretization-specific classes</li><li>• <i>fem/</i> - finite element specific classes</li><li>• <i>iga/</i> - isogeometric analysis specific classes</li></ul>	
<i>mesh/</i>	some utilities to interface with tetgen and triangle mesh generators	
<i>homogenization/</i>	Homogenization engine and supporting modules - highly specialized code, one of the reasons of SfePy 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>optimize/</i>	modules for shape optimization based on free-form deformation	•
<i>parallel/</i>	modules supporting parallel assembling and solution of problems	
<i>postprocess/</i>	Mayavi-based post-processing modules ( <i>postproc.py</i> )	
<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 <i>Term Overview</i>	

The directories in the “field-specific” column are mostly interesting for specialists working in the respective fields.

The *fem/* is the heart of the code, while the *terms/* contains the particular integral forms usable to build equations - new term writers should look there.

## 7.3 Exploring the Code

It is convenient to install IPython (see also [Using 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.
5 ↵mesh)...
6 sfepy: ...done in 0.00 s
7
8 In [3]: print mesh
9 Mesh:meshes/2d/rectangle_tri
10 cmesh:
11     CMesh: n_coor: 258, dim 2, tdim: 2, n_el 454
12     descs:
13         list: ['2_3']
14     dim:
15         2
16     dims:
17         list: [2]
18     io:
19         None
20     n_el:
21         454
22     n_nod:
23         258
24     name:
25         meshes/2d/rectangle_tri
26     nodal_bcs:
27         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.

## 7.4 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

### 7.4.1 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.

### 7.4.2 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
```

8. 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. 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:

1. Fetch the upstream changes:

```
git fetch upstream
```

Never start with `git pull upstream!`

2. 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`.
3. 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.

4. 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 [tbaggy.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](#).

### 7.4.3 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 L<sup>A</sup>T<sub>E</sub>X in a docstring, use a raw string:

```

1 def function():
2     r"""
3     This is a function with :math:`\boxed{\text{\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.

## 7.5 How to Regenerate Documentation

The following steps summarize how to regenerate this documentation.

1. Install `sphinx` and `numpydoc`. 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<sup>A</sup>T<sub>E</sub>X 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

```

## 7.6 How to Implement a New Term

*tentative documentation*

**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.

### 7.6.1 Notes on terminology

*Volume* refers to the whole domain (in space of dimension  $d$ ), while *surface* to a subdomain of dimension  $d - 1$ , for example a part of the domain boundary. So in 3D problems volume = volume, surface = surface, while in 2D volume = area, surface = curve.

### 7.6.2 Introduction

A term in SfePy usually corresponds to a single integral term in (weak) integral formulation of an equation. Both volume and surface 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 [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 `sfePy.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.

### 7.6.3 Evaluation modes

A term can support several evaluation modes, as described in [Term Evaluation](#).

### 7.6.4 Basic attributes

A term class should inherit from `sfePy.terms.terms.Term` base class. The simplest possible term with volume 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 ‘volume’ (the default, if not given), ‘surface’ or ‘surface\_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:

- ‘volume’ for volume integral over a region that contains elements; uses volume element connectivity for assembling;
- ‘surface’ for surface integral over a region that contains faces; uses surface face connectivity for assembling;
- ‘surface\_extra’ for surface integral over a region that contains faces; uses volume element 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*, *l*, *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\_neohookean* 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()*.

## 7.6.5 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.

### 7.6.6 Examples

Let us now discuss the implementation of a simple weak term `dw_volume_integrate` defined as  $\int_{\Omega} cq$ , where  $c$  is a weight (material parameter) and  $q$  is a virtual variable. This term is implemented as follows:

```

1  class IntegrateVolumeOperatorTerm(Term):
2      r"""
3          Volume integral of a test function weighted by a scalar function
4          :math:`c`.
5
6          :Definition:
7
8          .. math::
9              \int_{\Omega} q \, \mathrm{d}\Omega \quad \text{or} \quad \int_{\Omega} c \, q \, \mathrm{d}\Omega
10
11         :Arguments:
12             - material : :math:`c` (optional)
13             - virtual : :math:`q`
14
15         """
16         name = 'dw_volume_integrate'
17         arg_types = ('opt_material', 'virtual')
18         arg_shapes = [{ 'opt_material' : (1, 1), 'virtual' : (1, None) },
19                         { 'opt_material' : None }]
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
- lines 20-28: 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 22: transpose the base function and tile it so that it has the correct shape - it is repeated for each element;
  - line 23: ensure C contiguous order;
  - lines 24-27: perform numerical integration in C - *geo.integrate()* requires the C contiguous order;
  - line 28: return the status.
- lines 30-35: prepare arguments for the function above:
    - line 32: verify that the variable is scalar, as our implementation does not support vectors;
    - line 33: get reference element mapping corresponding to the virtual variable;
    - line 35: 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 *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 \ \cdot \ \nabla q
10            \int_{\Omega} p \underline{y} \cdot \nabla q
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
23     name = 'dw_s_dot_mgrad_s'
24     arg_types = (('material', 'virtual', 'state'),
25                  ('material', 'state', 'virtual'))
26     arg_shapes = [{('material') : 'D, 1',
27                   'virtual/grad_state' : (1, None),
28                   'state/grad_state' : 1,
29                   'virtual/grad_virtual' : (1, None),
30                   'state/grad_virtual' : 1}]
31     modes = ('grad_state', 'grad_virtual')
```

```

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')
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

## 7.6.7 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.

## 7.7 How To Make a Release

### 7.7.1 Release Tasks

A few notes on what to do during a release.

#### Things to check before a release

1. synchronize module documentation (dry run):

```
$ ./script/sync_module_docs.py doc/src/ . -n
```

2. regenerate gallery page and examples:

```
$ script/gen_gallery.py -l ../doc-devel
$ rm -rf doc/examples/
$ cp -a gallery/examples/ doc/
```

3. create temporary/testing tarball:

```
$ python setup.py sdist
```

4. check in-place build:

```
$ # unpack the tarball
$ # cd into

$ python setup.py build_ext --inplace
$ ./test_install.py
```

5. check that documentation can be built:

```
$ # copy site_cfg.py
$ python setup.py htmldocs
$ firefox doc/_build/html/index.html
```

or use:

```
$ cd doc/
$ make html
$ firefox _build/html/index.html
```

try also:

```
$ # copy gallery/images
$ python setup.py pdfdocs
```

6. check installed build:

```
$ pip install . --user
$ cd
$ sfepy-run run_tests
$ rm -r output/
```

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:

```
$ python script/gen_release_notes.py 2018.2
```

- 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:

```
$ python setup.py pdfdocs
```

- create tarball:

```
$ python 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 tarball to <https://sfepy.org/doc-devel/downloads.html>
- 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, scipy-dev@python.org, scipy-user@python.org, numpy-discussion@python.org, python-announce-list@python.org

## 7.8 Module Index

### 7.8.1 Main scripts

#### extractor.py script

Extract information from a SfePy multi-time-step results file (HDF5 format) and/or linearize results with stored higher order DOFs.

For the linearization, the original input (problem description) file must be specified as the first argument. Use the option `--linearization` below to override linearization parameters defined in the input file. The linearization forces `--dump` option, i.e., output to VTK files.

#### Examples

```
$ ./extractor.py -e "p e 0 1999" bone.h5 $ ./extractor.py -e "p e 0 1999" bone.h5 -a $ ./extractor.py -e "p e 0 1999"
bone.h5 -o extracted.h5 $ ./extractor.py -e "p e 0 1999" bone.h5 -o extracted.h5 -a
extractor.create_problem(filename)
extractor.main()
extractor.parse_linearization(linearization)
```

#### homogen.py script

```
homogen.main()
```

#### phonon.py script

```
phonon.main()
```

#### postproc.py script

This is a script for quick Mayavi-based visualizations of finite element computations results.

#### Examples

The examples assume that `run_tests.py` has been run successfully and the resulting data files are present.

- view data in `output-tests/test_navier_stokes.vtk`

```
$ python postproc.py output-tests/test_navier_stokes.vtk $ python postproc.py output-
tests/test_navier_stokes.vtk -3d
```

- save a snapshot image and exit

```
$ python postproc.py output-tests/test_poisson.vtk -o image.png -n
```

- save a snapshot image without off-screen rendering and exit

```
$ python postproc.py output-tests/test_poisson.vtk -o image.png -n --no-offscreen
```

- create animation (forces offscreen rendering) from output-tests/test\_time\_poisson.\*.vtk  
\$ python postproc.py output-tests/test\_time\_poisson.\*.vtk -a mov
  - create animation (forces offscreen rendering) from output-tests/test\_hyperelastic.\*.vtk  
The range specification for the displacements ‘u’ is required, as output-tests/test\_hyperelastic.00.vtk contains only zero displacements which leads to invisible glyph size.  
\$ python postproc.py output-tests/test\_hyperelastic.\*.vtk -ranges=u,0,0.02 -a mov
  - same as above, but slower frame rate  
\$ python postproc.py output-tests/test\_hyperelastic\_TL.\*.vtk -ranges=u,0,0.02 -a mov -ffmpeg-options="--r 2 -sameq"
- ```
class postproc.ParseDomainSpecific(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseGroupNames(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseOpacity(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseRanges(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseResolution(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseSubdomains(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
class postproc.ParseView(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
postproc.main()
postproc.view_file(filename, filter_names, options, view=None)
```

## probe.py script

Probe finite element solutions in points defined by various geometrical probes.

### Generation mode

python probe.py [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

python probe.py [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.

```
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.

```
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  $4\pi x^2$ .

```
probe.main()
```

```
probe.postprocess(filename_input, filename_results, options)
```

Postprocess probe data files - replot, integrate data.

## run\_tests.py script

### Notes on writing new test files:

A test file can contain anything, but usually it is similar to a regular input file (defining a test problem), with a mandatory Test class. This class holds all the test\_\* functions, as well as the from\_conf(), which serves to initialize the test (conf is in fact the test file itself, options are command-line options).

All variables defined in a test file are collected in ‘conf’ variable passed to a Test.\_\_init\_\_(). For example, ‘input\_name’ in test\_input\_\*.py files is accessible as ‘conf.input\_name’. This is useful if the test class is defined outside the test file, as the classes in tests\_basic.py are.

The test\_\* functions are collected automatically by run\_tests.py, with one exception: if a certain order of their evaluation is required, a class attribute ‘test’ of the Test class with a list of the test function names should be defined (example: test\_meshio.py).

```
class run_tests.OutputFilter(allowed_lines)
```

```
    flush()
```

```
start()
stop()
write(msg)

run_tests.get_dir(default)
run_tests.main()
run_tests.run_test(conf_name, options, ifile)
run_tests.wrap_run_tests(options)
```

### simple.py script

Solve partial differential equations given in a SfePy problem definition file.

Example problem definition files can be found in examples/ directory of the SfePy top-level directory. This script works with all the examples except those in examples/standalone/.

Both normal and parametric study runs are supported. A parametric study allows repeated runs for varying some of the simulation parameters - see examples/diffusion/poisson\_parametric\_study.py file.

```
simple.main()
simple.print_solvers()
simple.print_terms()
```

### simple\_homog\_mpi.py script

## 7.8.2 Utility scripts

### build\_helpers.py script

Build helpers for setup.py.

Includes package dependency checks and monkey-patch to numpy.distutils to work with Cython.

### Notes

The original version of this file was adapted from NiPy project [1].

[1] <http://nipy.sourceforge.net/>

```
class build_helpers.Clean(dist)
    Distutils Command class to clean, enhanced to clean also files generated during python setup.py build_ext -in-place.
    run()

class build_helpers.DoxygenDocs(dist)

    description = 'generate docs by Doxygen'
    run()

class build_helpers.NoOptionsDocs(dist)
```

```
finalize_options()
initialize_options()
user_options = [('None', None, 'this command has no options')]

class build_helpers.SphinxHTMLDocs(dist)

    description = 'generate html docs by Sphinx'
    run()

class build_helpers.SphinxPDFDocs(dist)

    description = 'generate pdf docs by Sphinx'
    run()

build_helpers.generate_a_pyrex_source(self, base, ext_name, source, extension)
Monkey patch for numpy build_src.build_src method

    Uses Cython instead of Pyrex.

build_helpers.get_sphinx_make_command()
build_helpers.have_good_cython()

build_helpers.package_check(pkg_name, version=None, optional=False, checker=<class distutils.version.LooseVersion>, 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 `distutils.version.LooseVersion`.

**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.

### test\_install.py script

Simple script for testing various SfePy functionality, examples not covered by tests, and running the tests.

The script just runs the commands specified in its main() using the *subprocess* module, captures the output and compares one or more key words to the expected ones.

The output of failed commands is saved to ‘test\_install.log’ file.

`test_install.check_output(cmd)`  
Run the specified command and capture its outputs.

#### Returns

**out** [tuple] The (stdout, stderr) output tuple.

`test_install.main()`

`test_install.report(out, name, line, item, value, eps=None, return_item=False)`  
Check that *item* at *line* of the output string *out* is equal to *value*. If not, print the output.

`test_install.report2(out, name, items, return_item=False)`  
Check that *items* are in the output string *out*. If not, print the output.

`test_install.report_tests(out, return_item=False)`  
Check that all tests in the output string *out* passed. If not, print the output.

### script/blockgen.py script

Block mesh generator.

`blockgen.main()`

### script/convert\_mesh.py script

Convert a mesh file from one SfePy-supported format to another.

Examples:

```
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new.vtk
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new.vtk -s2.5
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new.vtk -s0.5,2,1
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new.vtk -s0.5,2,1 -c 0
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new.mesh --remesh='q2/0 ale-8 09/7
˓→V'
$ ./script/convert_mesh.py meshes/3d/cylinder.mesh new2.mesh --remesh='rq2/0 ale-8 09/
˓→7 V'
```

`convert_mesh.main()`

### script/cylindergen.py script

Cylinder mesh generator.

`cylindergen.main()`

### script/edit\_identifiers.py script

Convert mixedCase identifiers to under\_scores.

```
edit_identifiers.cw2us (x)
edit_identifiers.edit (line)
edit_identifiers.main ()
edit_identifiers.match_candidate ()
    match(string[, pos[, endpos]]) -> match object or None. Matches zero or more characters at the beginning of
    the string
edit_identifiers.mc2us (x)
edit_identifiers.split_on (token, chars)
edit_identifiers.us2cw (x)
edit_identifiers.us2mc (x)
```

### script/eval\_ns\_forms.py script

Operators present in the FE discretization of (adjoint) Navier-Stokes terms.

```
eval_ns_forms.create_scalar (name, n_ep)
eval_ns_forms.create_scalar_base (name, n_ep)
eval_ns_forms.create_scalar_base_grad (name, phic, dim)
eval_ns_forms.create_scalar_var_data (name, phi, g, u)
eval_ns_forms.create_u_operator (u, transpose=False)
eval_ns_forms.create_vector (name, n_ep, dim)
    ordering is DOF-by-DOF
eval_ns_forms.create_vector_base (name, phic, dim)
eval_ns_forms.create_vector_base_grad (name, gc, transpose=False)
eval_ns_forms.create_vector_var_data (name, phi, vindx, g, gt, vgindx, u)
eval_ns_forms.grad_vector_to_matrix (name, gv)
eval_ns_forms.main ()
eval_ns_forms.substitute_continuous (expr, names, u, phi)
```

### script/eval\_tl\_forms.py script

Operators present in the FE discretization of hyperelastic terms in the total Lagrangian formulation.

```
eval_tl_forms.main ()
```

### script/extract\_edges.py script

Extract outline edges of a given mesh and save them into ‘<original path>/edge\_<original mesh file name>.vtk’ or into a user defined output file. The outline edge is an edge for which  $\text{norm}(\text{nvec1} - \text{nvec2}) < \text{eps}$ , where nvec1 and nvec2 are the normal vectors of the incident facets.

**extract\_edges.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, desc).

**extract\_edges.main()****extract\_edges.merge\_lines**(*mesh*, *eps*=*1e-18*)**script/extract\_surface.py script**

Given a mesh file, this script extracts its surface and prints it to stdout in form of a list where each row is [element, face, component]. A component corresponds to a contiguous surface region - for example, a cubical mesh with a spherical hole has two surface components. Two surface faces sharing a single node belong to one component.

With ‘-m’ option, a mesh of the surface is created and saved in ‘<original path>/surf\_<original mesh file name>.mesh’.

**extract\_surface.get\_surface\_faces**(*domain*)**extract\_surface.main()****extract\_surface.surface\_components**(*gr\_s*, *surf\_faces*)

Determine surface components given surface mesh connectivity graph.

**extract\_surface.surface\_graph**(*surf\_faces*, *n\_nod*)**script/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. Generate the files:

- for sfepy.org deployment:

```
$ ./script/gen_gallery.py -l ../doc-devel
```

- for local test build run from ./:

```
$ ./script/gen_gallery.py -l doc/_build/html/
```

2. remove doc/examples/:

```
$ rm -rf doc/examples/
```

3. copy gallery/examples/ to doc/:

```
$ cp -a gallery/examples/ doc/
```

4. regenerate the documentation:

```
$ python setup.py htmldocs
```

Additional steps for sfepy.org deployment:

- copy doc/\_build/html/ to <sfepy.org>/doc-devel/
- copy gallery/index.html and gallery/images/ to <sfepy.org>/

gen\_gallery.**generate\_gallery\_html** (*examples\_dir*, *output\_filename*, *gallery\_dir*, *rst\_dir*, *thumbnails\_dir*, *dir\_map*, *link\_prefix*)

Generate the gallery html file with thumbnail images and links to examples.

### Parameters

**output\_filename** [str] The output html file name.  
**gallery\_dir** [str] The top level directory of gallery files.  
**rst\_dir** [str] The full path to rst files of examples within *gallery\_dir*.  
**thumbnails\_dir** [str] The full path to thumbnail images within *gallery\_dir*.  
**dir\_map** [dict] The directory mapping returned by *generate\_rst\_files()*  
**link\_prefix** [str, optional] The prefix to prepend to links to individual pages of examples.

gen\_gallery.**generate\_images** (*images\_dir*, *examples\_dir*)

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*)

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**()

## script/gen\_iga\_patch.py script

Generate a single IGA patch block in 2D or 3D of given degrees and continuity using igakit.

The grid has equally-spaced knot vectors.

gen\_iga\_patch.**main**()

## script/gen\_lobatto1d\_c.py script

Generate lobatto1d.c and lobatto1h.c files.

gen\_lobatto1d\_c.**append\_declarations** (*out*, *copolys*, *comment*, *cvar\_name*, *shift*=0)

gen\_lobatto1d\_c.**append\_lists** (*out*, *names*, *length*)

gen\_lobatto1d\_c.**append\_polys** (*out*, *copolys*, *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')
```

### script/gen\_mesh\_prev.py script

Mesh Preview Generator.

#### Examples

```
$ ./script/gen_mesh_prev.py meshes/2d/  
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.

```
gen_mesh_prev.main()
```

### script/gen\_release\_notes.py script

Generate release notes using git log starting from the given version.

```
gen_release_notes.main()
```

### script/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.
```

### script/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.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_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)
```

### script/plot\_condition\_numbers.py script

Plot conditions numbers w.r.t. polynomial approximation order of reference element matrices for various FE polynomial spaces (bases).

```
plot_condition_numbers.main()
```

### script/plot\_logs.py script

Plot logs of variables saved in a text file by sfeypy.base.log.Log class.

The plot should be almost the same as the plot that would be generated by the Log directly.

```
class plot_logs.ParseRc(option_strings, dest, nargs=None, const=None, default=None, type=None,
                         choices=None, required=False, help=None, metavar=None)

plot_logs.main()
```

### script/plot\_mesh.py script

Plot mesh connectivities, facet orientations, global and local DOF ids etc.

To switch off plotting some mesh entities, set the corresponding color to *None*.

```
plot_mesh.main()
```

### script/plot\_quadratures.py script

Plot quadrature points for the given geometry and integration order.

```
plot_quadratures.main()
```

### script/plot\_times.py script

Plot time steps, times of time steps and time deltas in a HDF5 results file.

```
plot_times.main()
```

### script/save\_basis.py script

Save polynomial basis on reference elements or on a mesh for visualization into a given output directory.

```
save_basis.get_dofs(dofs, n_total)
save_basis.main()
save_basis.save_basis_on_mesh(mesh, options, output_dir, lin, permutations=None, suffix="")
```

### script/show\_authors.py script

```
show_authors.main()
```

### script/show\_mesh\_info.py script

Print various information about a mesh.

```
show_mesh_info.main()
```

### script/show\_terms\_use.py script

Show terms use in problem description files in the given directory.

```
show_terms_use.main()
```

### script/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()
```

### script/tile\_periodic\_mesh.py script

The program scales a periodic input mesh (a rectangle or box) in `filename_in` by a scale factor and generates a new mesh by repeating the scaled original mesh in a regular grid (`scale x scale [x scale]`) if repeat option is None, or in a grid `nx x ny x nz` for repeat ‘`nx,ny,nz`’, producing again a periodic rectangle or box mesh.

```
class tile_periodic_mesh.ParseRepeat(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
tile_periodic_mesh.main()
```

### 7.8.3 sfepy package

#### sfepy.config module

```
class sfepy.config.Config

    compile_flags()
    debug_flags()
    is_release()
    link_flags()
    numpydoc_path()
    python_include()
    python_version()
    system()
    tetgen_path()

sfepy.config.has_attr(obj, attr)
```

#### sfepy.version module

```
sfepy.version.get_basic_info(version='2019.2')
    Return SfePy installation directory information. Append current git commit hash to version.
```

### 7.8.4 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, 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.

**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.

## 7.8.5 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*)

`class sfepy.base.base.IndexedStruct(**kwargs)`

`class sfepy.base.base.OneTypeList(item_class, seq=None)`

```
find(name, ret_idx=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()**

**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(filenames, classes, package_name=None, ignore_errors=False, name_attr='name')`

For each filename in filenames, load all subclasses of classes listed.

`sfepy.base.base.mark_time(times, msg=None)`

Time measurement utility.

Measures times of execution between subsequent calls using time.clock(). The time is printed if the msg argument is not None.

### Examples

```
>>> times = []
>>> mark_time(times)
... do something
>>> mark_time(times, 'elapsed')
elapsed 0.1
... do something else
>>> mark_time(times, 'elapsed again')
elapsed again 0.05
>>> times
[0.1000000000000001, 0.0500000000000003]
```

`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.

---

```
sfeypy.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().

```
sfeypy.base.base.structify(obj)
```

Convert a (nested) dict *obj* into a (nested) Struct.

```
sfeypy.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.

```
sfeypy.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.

```
sfeypy.base.base.use_method_with_name(instance, method, new_name)
```

## sfeypy.base.compat module

This module contains functions that have different names or behavior depending on NumPy and Scipy versions.

```
sfeypy.base.compat.in1d(ar1, ar2, assume_unique=False, invert=False)
```

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))`.

New 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.

New 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], dtype=bool)
>>> test[mask]
array([0, 2, 0])
>>> mask = np.in1d(test, states, invert=True)
>>> mask
array([False,  True, False,  True, False], dtype=bool)
>>> test[mask]
array([1, 5])
```

`sfepy.base.compat.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None)`

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, and 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*. .. versionadded:: 1.9.0

**axis** [int or None, optional] The axis to operate on. If None, *ar* will be flattened beforehand. Otherwise, duplicate items will be removed along the provided axis, with all the other axes belonging to the each of the unique elements. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. .. versionadded:: 1.13.0

### 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. .. versionadded:: 1.9.0

### See also:

**numpy.lib.arraysetops** Module with a number of other functions for performing set operations on arrays.

## 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='|S1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'],
      dtype='|S1')
```

Reconstruct the input array from the unique values:

```
>>> 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])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

## 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, over-
                  ride=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`)

`sfeypy.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.

`sfeypy.base.conf.dict_from_string` (`string`)

Parse `string` and return a dictionary that can be used to construct/override a `ProblemConf` instance.

`sfeypy.base.conf.get_standard_keywords` ()

`sfeypy.base.conf.transform_conditions` (`adict, prefix`)

`sfeypy.base.conf.transform_ebc` (`adict`)

`sfeypy.base.conf.transform_epbc` (`adict`)

`sfeypy.base.conf.transform_fields` (`adict`)

`sfeypy.base.conf.transform_functions` (`adict`)

`sfeypy.base.conf.transform_ics` (`adict`)

`sfeypy.base.conf.transform_integrals` (`adict`)

```
sfePy.base.conf.transform_lcbc (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.

    default = False
    key = 'check_term_finiteness'
    keys()
        Return sorted list of keys.
    validate = {'verbose': <function validate_bool>, 'check_term_finiteness': <function validate_bool>}
    validator(val)
        Convert b to a boolean or raise a ValueError.
    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()**

**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()**

**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 writed by self.write_data()}`

### Parameters

**fd: tables.File** The hdf5 file handle the data should be writed 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 writed 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)
```

```
(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.

**sfeypy.base.log module**

```
class sfeypy.base.log.Log (data_names=None, plot_kwarg=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_kwarg=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 ()
```

```
iter_names (igs=None)
```

```
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 ()
```

```
sfeypy.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.

```
sfeypy.base.log.name_to_key (name, ii)
```

```
sfeypy.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.

    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.

```
sfeypy.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).

## sfeypy.base.multiproc module

Multiprocessing functions.

```
sfeypy.base.multiproc.get_multiproc(mpi=False)
```

```
sfeypy.base.multiproc.get_num_workers()
```

Get the number of slave nodes.

```
sfeypy.base.multiproc.is_remote_dict(d)
```

## sfeypy.base.multiproc\_mpi module

### sfeypy.base.multiproc\_proc module

Multiprocessing functions - using multiprocessing (process based) module.

```
class sfeypy.base.multiproc_proc.MyQueue
```

```
    get()
```

```
    put(value)
```

```
sfeypy.base.multiproc_proc.get_dict(name, clear=False, **kwargs)
```

Get the remote dictionary.

```
sfeypy.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:(ABCD...)}  [{Supress:{“{“Forward: ... } Supress:{“}”}} Forward: ... ]})
```

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:(ABCD...)} [{{{{Suppress:(“{”) Forward: ...} } Suppress:(“}”)} Forward: ...}])
```

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

**Id** [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.TestCommon(**kwargs)`

`static assert_equal(a, b, msg='assertion of equality failed!')`

`static compare_vectors(vec1, vec2, allowed_error=1e-08, label1='vec1', label2='vec2', norm=None)`

`static eval_coor_expression(expression, coor)`

`get_number()`

`static report(*args)`

All tests should print via this function.

`run(debug=False, ifile=None)`

`static xfail(test_method)`

Decorator that allows a test to fail.

## 7.8.6 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.

**canonize\_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.

**canonize\_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.EssentialBC` (*name*, *region*, *dofs*, *key=*", *times=None*)

Essential boundary condition.

#### 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.

```
canonize_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 condidion.

#### 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.

**canonize\_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)****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, asm\_obj=None)****Parameters**

**mode** [one of ‘eval’, ‘el\_eval’, ‘el\_avg’, ‘qp’, ‘weak’] The evaluation mode.

**static from\_desc (name, desc, variables, regions, materials, integrals, user=None)****class sfepy.discrete.equations.Equations (equations)****add\_equation (equation)**

Add a new equation.

**Parameters**

**equation** [Equation instance] The new equation.

**advance (ts)****apply\_ebc (vec, force\_values=None)**

Apply essential (Dirichlet) boundary conditions to a state vector.

**apply\_ic (vec, force\_values=None)**

Apply initial conditions to a state 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, rdcs=None, cdcs=None, shape=None, active_only=True, 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 volume DOF connectivities are used, with the exception of trace surface DOF connectivities. If True, any kind of DOF connectivities is allowed.

**rdcs, cdcs** [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.

**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_state_vector()
```

```
create_stripped_state_vector()
```

```
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.

```
eval_residuals(state, by_blocks=False, names=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.

#### Returns

**out** [array or dict of array] The assembled residual vector. If *by\_blocks* is True, a dictionary is returned instead, with keys given *block\_name* part of the individual equation names.

```
eval_tangent_matrices(state, tangent_matrix, by_blocks=False, names=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.

**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*, *asm\_obj=None*)  
Evaluate the equations.

**Parameters**

**mode** [one of ‘eval’, ‘el\_avg’, ‘qp’, ‘weak’] The evaluation mode.

**names** [str or sequence of str, optional] Evaluate only equations of the given name(s).

**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*, *verbose=True*)

**get\_domain()**

**get\_graph\_conn** (*any\_dof\_conn=False*, *rdfs=None*, *cdcs=None*, *active\_only=True*)  
Get DOF connectivities needed for creating tangent matrix graph.

**Parameters**

**any\_dof\_conn** [bool] By default, only volume DOF connectivities are used, with the exception of trace surface DOF connectivities. If True, any kind of DOF connectivities is allowed.

**rdfs, cdcs** [arrays, optional] Additional row and column DOF connectivities, corresponding to the variables used in the equations.

**active\_only** [bool] If True, the active DOF connectivities have reduced size and are created with the reduced (active DOFs only) numbering.

**Returns**

**rdfs, cdcs** [arrays] The row and column DOF connectivities defining the matrix graph blocks.

**get\_lcbc\_operator()**

**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\_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\_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.

**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** [array] 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\_variables\_from\_state(vec, step=0)**

Set data (vectors of DOF values) of variables.

**Parameters**

**data** [array] The state vector.

**step** [int] The time history step, 0 (default) = current.

**setup\_initial\_conditions(ics, functions=None)**

**state\_to\_output(vec, fill\_value=None, var\_info=None, extend=True)**

**strip\_state\_vector(vec, follow\_epbc=False)**

Strip a full vector by removing EBC dofs.

## 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.

```
time_update(ts, ebcs=None, epbcs=None, lcbcs=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.

**lcbcs** [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.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)
```

This class provides the functions required by a nonlinear solver for a given problem.

```
eval_residual(vec, is_full=False)
```

```
eval_tangent_matrix(vec, mtx=None, is_full=False)
```

```
make_full_vec(vec)
```

```
new_ulf_iteration(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.

```
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, lcbcs=None, ts=None, functions=None, auto_init=False, mode='eval', extra_args=None, active_only=True, 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.

**lcbcs** [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.

**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, 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.

**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)
```

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.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.

**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**

**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.

**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.

**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.

**refine\_points** (*variable, points, cache*)  
No refinement for this probe.

**report** ()  
Report the probe parameters.

sfeypy.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.

sfeypy.discrete.probes.**read\_header** (*fd*)  
Read the probe data header from file descriptor fd.

**Returns**

**header** [Struct instance] The probe data header.

sfeypy.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.

sfeypy.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 `-I - <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*, *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*, *lcbc=None*,  
*ts=None*, *functions=None*, *mode='eval'*, *var\_dict=None*, *strip\_variables=True*,  
*extra\_args=None*, *active\_only=True*, *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.

**lcbc** [Conditions instance, optional] The linear combination boundary conditions for ‘weak’ mode. If not given, *self.lcbc* 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.

**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_volume_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`, `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`, `lcbc=None`, `ts=None`,  
`functions=None`, `mode='eval'`, `dw_mode='vector'`, `term_mode=None`, `var_dict=None`,  
`strip_variables=True`, `ret_variables=False`, `active_only=True`, `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()**

Create a zero state vector 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()****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**(*state0, 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**

**state0** [State] The state holding the problem variables.

**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*, *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.

**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*, *state=None*, *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.

**state** [State instance, optional] The state instance. If not given, a new state is created using the variables in problem equations. Otherwise, its variables are modified in place.

**ts** [TimeStepper instance, optional] The time stepper. If not given, a default one is created. Otherwise, it is modified in place.

#### Returns

**new\_state** [State instance] The loaded state.

**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\_field\_meshes** (*filename\_trunk*)

**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*, *state=None*, *ts=None*)

Save the current state and time step to a restart file.

### Parameters

**filename** [str] The restart file name.

**state** [State instance, optional] The state instance. If not given, a new state is created using the variables in problem equations.

**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, *file\_per\_var*=False, \*\**kwargs*)

### Parameters

**file\_per\_var** [bool or None] If True, data of each 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’, *file\_per\_var* is assumed True.

**select\_bcs** (*ebc\_names*=None, *epbc\_names*=None, *lcbc\_names*=None, *create\_matrix*=False)

**select\_materials** (*material\_names*, *only\_conf*=False)

**select\_variables** (*variable\_names*, *only\_conf*=False)

**set\_bcs** (*ebcs*=None, *epbcs*=None, *lcbcs*=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* or *ts*. If such solver names do not exist, use the first of each required solver kind listed in *conf\_solvers*.

**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.

#### 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.

**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\_ics** (*ics*=None, *functions*=None)  
Setup the initial conditions for use.

**setup\_output** (*output\_filename\_trunk*=None, *output\_dir*=None, *output\_format*=None, *float\_format*=None, *file\_per\_var*=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, *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** [State or array, optional] If given, the initial state satisfying the initial conditions. By default, it 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.

### Returns

**state** [State] The final state.

**time\_update** (*ts=None*, *ebcs=None*, *epbcs=None*, *lcbc=None*, *functions=None*, *create\_matrix=False*, *is\_matrix=True*)  
**try\_presolve** (*mtx*)  
**update\_equations** (*ts=None*, *ebcs=None*, *epbcs=None*, *lcbc=None*, *functions=None*, *create\_matrix=False*, *is\_matrix=True*)  
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.

**lcbc** [Conditions instance, optional] The linear combination boundary conditions. If not given, *self.lcbc* 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*.

**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 reshapable to the shape required by `order`.

**order** [int, optional] The quadrature order. If not given, it is set to  $2 * target.field.approx_order$ .

```
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.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://mathematician.de/software/hedge>) 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  $[0, 1] \times \text{dim}$ . 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(geometries, order)
    Return the actual integration order for given geometry.
```

#### Parameters

**geometries** [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.state module

Module for handling state variables.

```
class sfePy.discrete.state.State(variables, vec=None, preserve_caches=False)
```

Class holding/manipulating the state variables and corresponding DOF vectors.

Manipulating the state class changes the underlying variables, and hence also the corresponding equations/terms (if any).

## Notes

This class allows working with LCBC conditions in time-dependent problems, as it keeps track of the reduced DOF vector that cannot be reconstructed from the full DOF vector by using the usual `variables.strip_state_vector()`.

**apply\_ebc** (`force_values=None`)

Apply essential (Dirichlet) boundary conditions to the state.

**apply\_ic** (`force_values=None`)

Apply initial conditions to the state.

**copy** (`deep=False, preserve_caches=False`)

Copy the state. By default, the new state contains the same variables, and creates new DOF vectors. If `deep` is True, also the DOF vectors are copied.

### Parameters

**deep** [bool] If True, make a copy of the DOF vectors.

**preserve\_caches** [bool] If True, do not invalidate evaluate caches of variables.

**create\_output\_dict** (`fill_value=None, var_info=None, extend=True, linearization=None`)

Transforms state to an output dictionary, 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`.

## Examples

```
>>> out = state.create_output_dict()
>>> problem.save_state('file.vtk', out=out)
```

**fill** (`value`)

Fill the DOF vector with given value.

**static from\_variables** (`variables`)

Create a State instance for the given variables.

The DOF vector is created using the DOF data in `variables`.

### Parameters

**variables** [Variables instance] The variables.

**get\_parts** ()

Return parts of the DOF vector corresponding to individual state variables.

### Returns

**out** [dict] The dictionary of the DOF vector parts.

**get\_reduced** (`follow_epbc=False`)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

**get\_vec** (`active_only`)

**get\_weighted\_norm** (`vec, weights=None, return_weights=False`)

Return the weighted norm of DOF vector `vec`.

By default, each component of `vec` is weighted by the 1/norm of the corresponding state part, or 1 if the norm is zero. Alternatively, the weights can be provided explicitly using `weights` argument.

## Parameters

**vec** [array] The DOF vector corresponding to the variables.  
**weights** [dict, optional] If given, the weights are used instead of the norms of the state parts.  
Keys of the dictionary must be equal to the names of variables comprising the DOF vector.  
**return\_weights: bool** If True, return also the used weights.

## Returns

**norm** [float] The weighted norm.  
**weights** [dict, optional] If *return\_weights* is True, the used weights.

## Examples

```
>>> err = state0.get_weighted_norm(state() - state0())
```

### **has\_ebc()**

Test whether the essential (Dirichlet) boundary conditions have been applied to the DOF vector.

### **init\_history()**

Initialize variables with history.

### **set\_full(vec, var\_name=None, force=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\_parts(parts, force=False)**

Set parts of the DOF vector corresponding to individual state variables.

## Parameters

**parts** [dict] The dictionary of the DOF vector parts.

### **set\_reduced(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\_vec(vec, active\_only)**

## sfePy.discrete.variables module

Classes of variables for equations/terms.

```
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, *is\_trace*=False, *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** [‘volume’, ‘surface’, ‘surface\_extra’, or ‘point’] The term integration type. If None, it is derived from *integral*.

**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.

**is\_trace** [bool, default False] Indicate evaluation 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='volume'*, *region\_name=None*)  
Get element data dimensions for given approximation.

#### Parameters

**integral** [Integral instance] The integral describing used numerical quadrature.

**integration** ['volume', 'surface', 'surface\_extra', 'point' or 'custom'] The term integration type.

**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** (*dc\_type*, *is\_trace=False*, *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.fem.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\_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 L<sup>2</sup> 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()
static reset()

set_constant(val)
    Set the variable to a constant value.

set_data(data=None, idx=None, step=0, preserve_caches=False)
    Set data (vector of DOF values) of the variable.

```

**Parameters**

**data** [array] The vector of DOF values.  
**idx** [int, optional] If given, *data*[*idx*] 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, force_values=None)
    Apply essential (Dirichlet) and periodic boundary conditions defined for the state variables to vector vec.  

apply_ic(vec, force_values=None)
    Apply initial conditions defined for the state variables to vector vec.

```

```

check_vector_size(vec, stripped=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.  
**stripped** [bool] If True, the size of the DOF vector should be reduced, i.e. without DOFs fixed by boundary conditions.

```

create_state_vector()

```

```

create_stripped_state_vector()

```

```

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.

**static from\_conf** (*conf, fields*)

This method resets the variable counters for automatic order!

**get\_dual\_names()**

Get names of pairs of dual variables.

#### Returns

**duals** [dict] The dual names as virtual name : state name pairs.

**get\_indx** (*var\_name, stripped=False, allow\_dual=False*)

**get\_lcbc\_operator()**

**get\_matrix\_shape()**

**get\_state\_part\_view** (*state, var\_name, stripped=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.

**has\_ebc** (*vec, force\_values=None*)

**has\_virtuals()**

**init\_history()**

**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.

**set\_adof\_conn** (*adof\_conn*)  
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\_from\_state** (*var\_names*, *state*, *var\_names\_state*)  
Set variables with names in *var\_names* from state variables with names in *var\_names\_state* using DOF values in the state vector *state*.

**set\_state\_part** (*state*, *part*, *var\_name*, *stripped*=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.

**state\_to\_output** (*vec*, *fill\_value*=None, *var\_info*=None, *extend*=True, *linearization*=None)  
Convert a state vector to a dictionary of output data usable by Mesh.write().

**strip\_state\_vector** (*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.

**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_conn(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, is\_trace flag).

## 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)`

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.

**update** (*name, n\_dof*)  
Set the number of DOFs of the given variable.

**Parameters**

- name** [str] The name of variable the DOFs correspond to.
- n\_dof** [int] The number of DOFs.

**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_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)
```

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()
```

## sfePy.discrete.common.extmods.assemble module

Low level finite element assembling functions.

```
sfePy.discrete.common.extmods.assemble.assemble_matrix()
sfePy.discrete.common.extmods.assemble.assemble_matrix_complex()
sfePy.discrete.common.extmods.assemble.assemble_vector()
sfePy.discrete.common.extmods.assemble.assemble_vector_complex()
```

## 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
```

```
cell_groups
cell_types
conns
coors
cprint()
create_new()
```

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 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()**

**from\_data()**  
Fill CMesh data using Python data.

**get\_cell\_conn()**

**get\_centroids()**  
Return the coordinates of centroids of mesh entities with dimension *dim*.

**get\_complete()**  
Get entities of dimension *dim* that are completely given by entities of dimension *dent* listed in *entities*.

**get\_conn()**

**get\_conn\_as\_graph()**  
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()**  
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()**  
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()**

**get\_local\_ids()**  
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()**  
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()**  
Return the volumes of mesh entities with dimension *dim* > 0.

**key\_to\_index**

---

```

n_coor
n_el
num
set_local_entities()
setup_connectivity()
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.create_mesh_graph()`

Create sparse (CSR) graph corresponding to given row and column connectivities.

#### **Parameters**

**n\_row** [int] The number of row connectivity nodes.

**n\_col** [int] The number of column connectivity nodes.

**n\_gr** [int] The number of element groups.

**rconns** [list of arrays] The list of length  $n\_gr$  of row connectivities.

**cconns** [list of arrays] The list of length  $n\_gr$  of column connectivities.

#### **Returns**

**nnz** [int] The number of graph nonzeros.

**prow** [array] The array of CSR row pointers.

**icol** [array] The array of CSR column indices.

`sfePy.discrete.common.extmods.cmesh.get_cmem_usage()`

`sfePy.discrete.common.extmods.cmesh.graph_components()`

Determine connected components of a compressed sparse graph.

#### **Returns**

**n\_comp** [int] The number of components.

**flag** [array] The flag marking for each node its component.

`sfePy.discrete.common.extmods.cmesh.orient_elements()`

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()`

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()
sfePy.discrete.common.extmods.crefcoors.find_ref_coors_convex()
```

## sfePy.discrete.common.extmods.mappings module

Low level reference mapping functionality.

```
class sfePy.discrete.common.extmods.mappings.CMapping
```

```
alloc_extra_data()
bf
bfg
cprint()
describe()
    Describe the element geometry - compute the reference element mapping.
det
dim
evaluate_bfbgm()
    Evaluate volume base function gradients in surface quadrature points.
get_element_diameters()
    Compute diameters of selected elements.
integral
integrate()
    Integrate arr over the domain of the mapping into out.
mode
mtx_t
n_el
n_ep
n_qp
normal
ps
qp
shape
volume
```

## 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’}, optional] The evaluation mode: the field value (default) or the field value gradient.

**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) 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_base='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\_base** [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** [CMapping instance] The reference mapping.

**mapping** [VolumeMapping or SurfaceMapping instance] The mapping.

**key** [tuple] The key of the mapping in *mappings* or *mappings0*.

```
save_mappings ()
```

Save current reference mappings to *mappings0* attribute.

```
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,          verb-
  bose=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  $x_i$  such that  $X(x_i) = P$ :

1. make inverse connectivity - for each vertex have cells it is in.
2. find the closest vertex  $V$ .
3. choose initial cell:  $i_0 = \text{first from cells incident to } V$ .
4. while not  $P$  in  $C_{-i}$ , change  $C_{-i}$  towards  $P$ , check if  $P$  in new  $C_{-i}$ .

```
sfePy.discrete.common.global_interp.get_ref_coors_general(field,           coors,
   close_limit=0.1,
   get_cells_fun=None,
   cache=None,          verb-
  bose=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** [VolumeMapping or SurfaceMapping 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.

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.region module`

**class** `sfePy.discrete.common.region.Region(name, definition, domain, parse_def, kind='cell', parent=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

### `cells`

**contains** (*other*)

Return True in the region contains the *other* region.

The check is performed using entities corresponding to the other region kind.

**copy** ()

Vertices-based copy.

**delete\_zero\_faces** (*eps=1e-14*)

### `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*)

### `faces`

### `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.

#### Notes

If the number of unique values in *cells* is smaller or equal to the number of cells in the region, all *cells* has to be also the region cells (*self* is a superset of *cells*). The region cells are considered depending on *true\_cells\_only*.

Otherwise, indices of all cells in *self* that are in *cells* are returned.

**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)`

### `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.

#### `vertices`

### `sfeypy.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.

### `sfeypy.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.SurfaceMapping*.

```
fix_element_orientation()
```

Ensure element vertices ordering giving positive cell volumes.

```
get_conn(ret_gel=False)
```

Get the cell-vertex connectivity and, if *ret\_gel* is True, also the corresponding reference geometry element.

```
get_diameter()
```

Return the diameter of the domain.

#### Notes

The diameter corresponds to the Friedrichs constant.

```
get_element_diameters(cells, vg, 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

    base1d
    cprint()
    e_coors_max
    evaluate()
    geo_ctxt
    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()
```

Evaluate 1D Lobatto functions of the given order in given points.

```
sfePy.discrete.fem.extmods.lobatto_bases.eval_lobatto_tensor_product()
```

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  $\geq 1$ ) 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  $\leftrightarrow$  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

`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.FESurface(name, region, efaces, volume_econn, volume_region=None)`

Description of a surface of a finite element domain.

`get_connectivity(local=False, is_trace=False)`

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.

**is\_trace** [bool] If True, return mirror connectivity according to *local*.

**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

- interps and hence node\_descs are per region (must have single geometry!)

Field shape information:

- *shape* - the shape of the base 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 base functions) in a point

**clear\_qp\_base()**

Remove cached quadrature points and base functions.

**create\_bqp** (*region\_name, integral*)

**create\_mapping** (*region, integral, integration, return\_mapping=True*)  
Create a new reference mapping.

Compute jacobians, element volumes and base function derivatives for Volume-type geometries (volume mappings), and jacobians, normals and base 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.

**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\_base** (*key*, *derivative*, *integral*, *iels=None*, *from\_geometry=False*, *base\_only=True*)

**get\_connectivity** (*region*, *integration*, *is\_trace=False*)

Convenience alias to *Field.get\_econn()*, that is used in some terms.

**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='volume'*, *region\_name=None*)

Get element data dimensions.

#### Parameters

**integral** [Integral instance] The integral describing used numerical quadrature.

**integration** ['volume', 'surface', 'surface\_extra', 'point' or 'custom'] The term integration type.

**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

- *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 and group.

**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’ or ‘s#’, where # is the number of face vertices.

**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.replace_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.get_base()` 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\_coors** ()

Setup coordinates of field nodes.

#### **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.

### **class sfepy.discrete.fem.fields\_base.SurfaceField** (*name*, *dtype*, *shape*, *region*, *approx\_order=1*)

Finite element field base class over surface (element dimension is one less than space dimension).

#### **average\_qp\_to\_vertices** (*data\_qp*, *integral*)

Average data given in quadrature points in region elements into region vertices.

$$u_n = \sum_e (u_{e,avg} * area_e) / \sum_e area_e = \sum_e \int_{area_e} u / \sum_e area_e$$

#### **get\_econn** (*conn\_type*, *region*, *is\_trace=False*, *integration=None*)

Get extended connectivity of the given type in the given region.

#### **setup\_extra\_data** (*geometry*, *info*, *is\_trace*)

### **class sfepy.discrete.fem.fields\_base.VolumeField** (*name*, *dtype*, *shape*, *region*, *approx\_order=1*)

Finite element field base class over volume elements (element dimension equals space dimension).

---

**average\_qp\_to\_vertices** (*data\_qp, integral*)  
Average data given in quadrature points in region elements into region vertices.

$$u_n = \sum_e (u_{e,avg} * volume_e) / \sum_e volume_e = \sum_e \int_{volume_e} u / \sum_e volume_e$$

**get\_econn** (*conn\_type, region, is\_trace=False, integration=None*)  
Get extended connectivity of the given type in the given region.

**setup\_extra\_data** (*geometry, info, is\_trace*)

**setup\_point\_data** (*field, region*)

**setup\_surface\_data** (*region, is\_trace=False, trace\_region=None*)  
nodes[leconn] == econn

```
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)
```

**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 given DOFs using a function of space coordinates or value *fun*.

## **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.H1DiscontinuousField(name, dtype, shape, region, approx_order=1)
```

**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 base (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)
    family_name = 'volume_H1_lagrange'

    interp_v_vals_to_n_vals (vec)
        Interpolate a function defined by vertex DOF values using the FE geometry base (P1 or Q1) into the extra
        nodes, i.e. define the extra DOF values.
```

## 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.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:

$$[u_1, u_2, u_3]^T = [\underline{d}_1, \underline{d}_2, \underline{d}_3]^T w,$$

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\_pbc** (*dofs*, *master*)

Treat dofs with periodic BC.

```
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.NodalLLCOperator(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.RigidOperator(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.

**get\_base (coors, diff=False)**

Get base functions or their gradient evaluated in given coordinates.

**get\_geometry ()**

Return reference element geometry as a GeometryElement instance.

**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`).

**class sfePy.discrete.fem.mappings.SurfaceMapping (coors, conn, poly\_space=None, gel=None, order=1)**

Mapping from reference domain to physical domain of the space dimension higher by one.

**get\_base (coors, diff=False)**

Get base functions or their gradient evaluated in given coordinates.

**get\_mapping (qp\_coors, weights, poly\_space=None, mode='surface')**

Get the mapping for given quadrature points, weights, and polynomial space.

### Returns

**cmap [CMapping instance]** The surface mapping.

**set\_basis\_indices (indices)**

Set indices to cell-based basis that give the facet-based basis.

```
class sfepy.discrete.fem.mappings.VolumeMapping (coors, conn, poly_space=None,
  gel=None, order=1)
    Mapping from reference domain to physical domain of the same space dimension.

    get_mapping (qp_coors, weights, poly_space=None, ori=None, transform=None)
        Get the mapping for given quadrature points, weights, and polynomial space.
```

#### Returns

**cmap** [CMapping instance] The volume mapping.

## 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.

#### 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, descbs=None)
```

Create a mesh from mesh IO data.

```
static from_file (filename=None, io='auto', prefix_dir=None, omit_facets=False)
```

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)
```

Create a mesh corresponding to cells, or, if *is\_surface* is True, to facets, of a given region.

```
get_bounding_box ()
```

```
get_conn (desc, ret_cells=False)
```

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*, *\*\*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, conn)`

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, conn, mesh_in)`

Create a mesh reusing mat\_ids and desc 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.ANSYS_CDB_MeshIO(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.AVSUCDMeshIO(filename, **kwargs)

    format = 'avs_ucd'
    static guess(filename)
    read(mesh, **kwargs)
    read_dimension()
    write(filename, mesh, out=None, **kwargs)

class sfepy.discrete.fem.meshio.AbaqusMeshIO(filename, **kwargs)

    format = 'abaqus'
    static guess(filename)
    read(mesh, **kwargs)
    read_dimension()
    write(filename, mesh, out=None, **kwargs)

class sfepy.discrete.fem.meshio.BDFMeshIO(filename, **kwargs)

    format = 'nastran'
    static format_str(str, idx, n=8)
    read(mesh, **kwargs)
    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.HDF5MeshIO(filename, **kwargs)

    ch = '\xfe'
    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)
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, idx, 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/python2.7/string.pyc’>
write(filename, mesh, out=None, ts=None, cache=None, **kwargs)
static write_mesh_to_hdf5(filename, group, mesh)
```

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.

```
class sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO(filename, **kwargs)
```

```
format = ‘hmascii’
read(mesh, **kwargs)
read_dimension()
```

```
write(filename, mesh, out=None, **kwargs)
```

```
class sfepy.discrete.fem.meshio.MEDMeshIO(filename, **kwargs)
```

```
format = ‘med’
read(mesh, **kwargs)
```

```
class sfepy.discrete.fem.meshio.MeditMeshIO(filename, **kwargs)
```

```
    format = 'medit'  
    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)
```

```
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 = ...  
    descbs = ...  
    mesh._set_io_data(nodes, ngroups, conns, mat_ids, descbs)  
    return mesh
```

See the Mesh class' docstring how the nodes, ngroups, conns, mat\_ids and descbs 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 descbs) to construct your specific format.

The methods read\_dimension(), read\_bounding\_box() should be implemented in subclasses, as it is often possible to get that kind of information without reading the whole mesh file.

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 od 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)
```

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!’

**static for\_format** (*filename*, *format*=*None*, *writable*=*False*, *prefix\_dir*=*None*)

Create a MeshIO instance for file *filename* with forced *format*.

#### Parameters

**filename** [str] The name of the mesh file.

**format** [str] One of supported formats. If *None*, *MeshIO.any\_from\_filename()* is called instead.

**writable** [bool] If *True*, verify that the mesh format is writable.

**prefix\_dir** [str] The directory name to prepend to *filename*.

#### Returns

**io** [MeshIO subclass instance] The MeshIO subclass instance corresponding to the *format*.

**format** = *None*

**get\_filename\_trunk()**

**get\_vector\_format** (*dim*)

**read** (*mesh*, *omit\_facets*=*False*, *\*\*kwargs*)

**read\_bounding\_box** (*ret\_fd*=*False*, *ret\_dim*=*False*)

**read\_data** (*step*, *filename*=*None*, *cache*=*None*)

**read\_dimension** (*ret\_fd*=*False*)

**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.**Msh2MeshIO** (*filename*, *\*\*kwargs*)

**format** = ‘msh\_v2’

**msh\_cells** = {1: (2, 2), 2: (2, 3), 3: (2, 4), 4: (3, 4), 5: (3, 8), 6: (3, 6)}

**prism2hexa** = array([0, 1, 2, 2, 3, 4, 5, 5])

```
    read(mesh, omit_facets=True, **kwargs)
    read_bounding_box(ret_fd=False, ret_dim=False)
    read_dimension(ret_fd=True)

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.TetgenMeshIO(filename, **kwargs)
```

**format** = 'tetgen'

**static getele**(fele)

Reads t.1.ele, returns a list of elements.

Example:

```
>>> elements, regions = self.getele("t.1.ele")
>>> elements
[(20, 154, 122, 258), (86, 186, 134, 238), (15, 309, 170, 310), (146,
229, 145, 285), (206, 207, 125, 211), (99, 193, 39, 194), (185, 197,
158, 225), (53, 76, 74, 6), (19, 138, 129, 313), (23, 60, 47, 96),
(119, 321, 1, 329), (188, 296, 122, 322), (30, 255, 177, 256), ...]
>>> regions
[100: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 7, ...],
...]
```

**static getnodes**(fnods)

Reads t.1.nodes, returns a list of nodes.

Example:

```
>>> self.getnodes("t.1.node")
[(0.0, 0.0, 0.0), (4.0, 0.0, 0.0), (0.0, 4.0, 0.0), (-4.0, 0.0, 0.0),
(0.0, 0.0, 4.0), (0.0, -4.0, 0.0), (0.0, -0.0, -4.0), (-2.0, 0.0,
-2.0), (-2.0, 2.0, 0.0), (0.0, 2.0, -2.0), (0.0, -2.0, -2.0), (2.0,
0.0, -2.0), (2.0, 2.0, 0.0), ... ]
```

**read**(mesh, \*\*kwargs)

**read\_bounding\_box**()

**read\_dimension**()

**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.VTKMeshIO(filename, **kwargs)

    format = 'vtk'
    get_dimension(coors)
    read(mesh, **kwargs)
    read_bounding_box(ret_fd=False, ret_dim=False)
    read_coors(ret_fd=False)
    read_data(step, filename=None, cache=None)
    read_dimension(ret_fd=False)
    write(filename, mesh, out=None, ts=None, **kwargs)

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.guess_format(filename, ext, formats, io_table)
    Guess the format of filename, candidates are in formats.

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.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.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,           geom-
   etry,           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,           geom-
   etry,           order,
   init_context=True)
    Lagrange polynomial space on a simplex domain.

    name = 'lagrange_simplex'

class sfePy.discrete.fem.poly_spaces.LagrangeTensorProductPolySpace (name,           geom-
   etry,           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.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.PolySpace (name, geometry, order)
    Abstract polynomial space class.

static any_from_args (name, geometry, order, base='lagrange', force_bubble=False)
    Construct a particular polynomial space classes according to the arguments passed in.

describe_nodes()

eval_base (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_base() 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

**base** [array] The basis (shape (n\_coor, 1, n\_base)) or its first derivative (shape (n\_coor, dim,
n\_base)) or its second derivative (shape (n\_coor, dim, dim, n\_base)) 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 base 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.

```
get_mtx_i()

keys = {(1, 2): 'simplex', (3, 4): 'simplex', (3, 8): 'tensor_product', (2, 3): 'simplex', (2, 4): 'tensor_product'}

static suggest_name (geometry, order, base='lagrange', force_bubble=False)
    Suggest the polynomial space name given its constructor parameters.

sfepy.discrete.fem.poly_spaces.transform_basis (transform, bf)
    Transform a basis bf using transform array of matrices.
```

## sfepy.discrete.fem.refine module

Basic uniform mesh refinement functions.

`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.utils module

```
sfePy.discrete.fem.utils.compute_nodal_edge_dirs(nodes, region, field, re-
  turn_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, re-
  turn_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.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**

**bfg**

```
bufBN
cprint()
dR_dx
dR_dxi
e_coors_max
evaluate()
iel

sfepy.discrete.iga.extmods.igac.eval_bernstein_basis()
sfepy.discrete.iga.extmods.igac.eval_in_tp_coors()
```

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()
```

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.

`sfeypy.discrete.iga.extmods.igac.eval_variable_in_qp()`

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.

`sfeypy.discrete.iga.extmods.igac.is_nurbs()`

Return True if some weights are not one.

## sfeypy.discrete.iga.fields module

Fields for isogeometric analysis.

**class** `sfeypy.discrete.iga.fields.IGField(name, dtype, shape, region, approx_order=None, **kwargs)`

Bezier extraction based NURBS field 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=‘volume’, region\_name=None*)

Get element data dimensions.

#### Parameters

**integral** [Integral instance] The integral describing used numerical quadrature.

**integration** [‘volume’] The term integration type. Only ‘volume’ 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, is\_trace=False, integration=None*)

Get DOF connectivity of the given type in the given region.

**get\_true\_order** ()

**is\_higher\_order()**

Return True, if the field's approximation order is greater than one.

**set\_dofs (fun=0.0, region=None, dpn=None, warn=None)**

Set the values of DOFs given by the *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 DOF-by-DOF, i.e. a single one-dimensional vector with all values of the first component, then of the second one etc. concatenated together.

**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 (dpn, n\_dof)] The values of the DOFs, DOF-by-DOF when raveled in C (row-major) order.

**setup\_extra\_data (geometry, info, is\_trace)**

```
sfeypy.discrete.iga.fields.parse_approx_order(approx_order)
```

**sfeypy.discrete.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.

**sfeypy.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.

```
sfeypy.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

**fun** [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 field 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).

## 7.8.7 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 accoring 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(fig\_num, plot\_rsc, plot\_labels, valid, freq\_range, plot\_range, show=False, clear=False, new\_axes=False)**  
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(fig_num, plot_rsc, gaps, kinds, gap_ranges,
  freq_range, plot_range, show=False,
  clear=False, new_axes=False)
```

Plot band gaps as rectangles.

```
sfePy.homogenization.band_gaps_app.plot_logs(fig_num, plot_rsc, plot_labels,
  freqs, logs, valid, freq_range,
  plot_range, draw_eigs=True,
  show_legend=True, show=False,
  clear=False, new_axes=False)
```

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='%.2e', 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^{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.CoefDim(name, problem, kwargs)
```

```
class sfePy.homogenization.coefs_base.CoefDimDim(name, problem, kwargs)
```

```

class sfepy.homogenization.coefs_base.CoefDimSym(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoefDummy(name, problem, kwargs)
    Dummy class serving for computing and returning its requirements.

class sfepy.homogenization.coefs_base.CoefEval(name, problem, kwargs)
    Evaluate expression.

class sfepy.homogenization.coefs_base.CoefExprPar(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.CoefExprPar,
static set_variables_default(variables, ir, set_var, data)

class sfepy.homogenization.coefs_base.CoefN(name, problem, kwargs)

get_coef(row, volume, problem, data)
static set_variables_default(variables, ir, ic, mode, set_var, data)

class sfepy.homogenization.coefs_base.CoefNN(name, problem, kwargs)

get_coef(row, col, volume, problem, data)
static set_variables_default(variables, ir, ic, mode, set_var, data)

class sfepy.homogenization.coefs_base.CoefNonSym(name, problem, kwargs)

is_sym=False
static iter_sym(dim)

class sfepy.homogenization.coefs_base.CoefNonSymNonSym(name, problem, kwargs)

is_sym=False
static iter_sym(dim)

class sfepy.homogenization.coefs_base.CoefNone(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoefOne(name, problem, kwargs)

static set_variables_default(variables, set_var, data)

class sfepy.homogenization.coefs_base.CoefSum(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoefSym(name, problem, kwargs)

is_sym=True
static iter_sym(dim)

class sfepy.homogenization.coefs_base.CoefSymSym(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_surface_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, file_per_var=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_ts_val(step)
    iter_solutions()
    iter_time_steps()

class sfepy.homogenization.coefs_base.MiniAppBase(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.**PressureEigenvalueProblem**(*name, problem, kwargs*)

Pressure eigenvalue problem solver for time-dependent correctors.

**presolve**(*mtx*)

Prepare  $A^{-1} B^T$  for the Schur complement.

**solve\_pressure\_eigenproblem**(*mtx, eig\_problem=None, n\_eigs=0, check=False*)

$G = B^* A^{-1} B^T$  or  $B^* A^{-1} B^T + D$

**class** sfepy.homogenization.coefs\_base.**ShapeDim**(*name, problem, kwargs*)

**class** sfepy.homogenization.coefs\_base.**ShapeDimDim**(*name, problem, kwargs*)

**class** sfepy.homogenization.coefs\_base.**TCorrectorsViaPressureEVP**(*name, problem, kwargs*)

Time correctors via the pressure eigenvalue problem.

**compute\_correctors**(*evp, sign, state0, ts, problem=None, vec\_g=None*)**save**(*corrs, problem, ts*)**setup\_equations**(*equations, problem=None*)

Set equations, update boundary conditions and materials.

**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\_base.**create\_ts\_coef**(*cls*)

Define a new class with modified call method which accepts time dependent data (correctors).

**sfepy.homogenization.coefs\_elastic module**

**class** sfepy.homogenization.coefs\_elastic.**PressureRHSVector**(*name, problem, kwargs*)

**class** sfepy.homogenization.coefs\_elastic.**TCorrectorsPressureViaPressureEVP**(*name, problem, kwargs*)

**class** sfepy.homogenization.coefs\_elastic.**TCorrectorsRSViaPressureEVP**(*name, problem, kwargs*)

## sfePy.homogenization.coefs\_perfusion module

```
class sfePy.homogenization.coefs_perfusion.CoeffRegion(name, problem, kwargs)
```

```
    get_variables(problem, ir, data)
```

```
class sfePy.homogenization.coefs_perfusion.CorrRegion(name, problem, kwargs)
```

```
    get_variables(ir, data)
```

## 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()
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()

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()
```

```
class sfepy.homogenization.coefs_phononic.PhaseVelocity(name, problem, kwargs)
    Compute phase velocity.

    process_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()

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()

    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?

`sfeypy.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 -> 0 for f in ]f0, f1[              |
| 0    | 1    | f -> f1, smallest eigenvalue < 0               |
| 0    | 2    | f -> f0, smallest eigenvalue > 0 and -> -infty |
| 1    | 1    | f -> f1, largest eigenvalue < 0 and -> +infty  |
| 1    | 2    | f -> f0, largest eigenvalue > 0                |

`sfeypy.homogenization.coefs_phononic.get_callback(mass, method, 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

`sfeypy.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*.

`sfeypy.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.

`sfeypy.homogenization.coefs_phononic.get_ranges(freq_range, eigs)`

Get an eigenvalue range slice and a corresponding initial frequency range within a given frequency range.

`sfeypy.homogenization.coefs_phononic.split_chunks(indx)`

Split index vector to chunks of consecutive numbers.

## sfePy.homogenization.convolutions module

```
class sfePy.homogenization.convolutions.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  $c[i]$ ,  $i = 0, 1, \dots$

$$c_0 \equiv c[0], c_{e0} \equiv c_0 e^e, \\ c(t) \approx c_0 d(t) \approx c_0 e(t) = c_{e0} e_n(t),$$

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.convolutions.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.convolutions.compute_mean_decay(coef)
```

Compute mean decay approximation of a non-scalar fading memory coefficient.

```
sfePy.homogenization.convolutions.eval_exponential(coefs, x)
```

```
sfePy.homogenization.convolutions.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.CoeffVolume(name, problem, kwargs)
```

```
class sfePy.homogenization.engine.HomogenizationEngine(problem, options, app_options=None, volumes=None, out_put_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)
    set_micro_coors (ncoors)
    setup_options (app_options=None)

class sfepy.homogenization.engine.HomogenizationWorker

static calculate (mini_app, problem, dependencies, dep_requires, save_names, micro_coors,
                  chunk_tab, mode, proc_id)
static calculate_req (problem, opts, post_process_hook, name, req_info, coef_info, save_names, de-
                  pendencies, micro_coors, 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\_coors** [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_coors, 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 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\_coors** (*num\_workers*, *num\_micro*, *reqs*, *coefs*, *chunks\_per\_worker=1*,  
*store\_micro\_idxs=[]*)

Split multiple microp problems 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\_idxs** [list of int] The indices of microstructures whose results are to be stored.

#### Returns

**micro\_tab** [list of slices] The indices of microp problems 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 microp problems.

#### 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\_idxs=[]*)

**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.engine.**rm\_multi** (*s*)

## **sfepy.homogenization.homogen\_app module**

**class** sfepy.homogenization.homogen\_app.**HomogenizationApp** (*conf*, *options*, *output\_prefix*,  
*\*\*kwargs*)

**call** (*verbose=False*, *ret\_all=None*, *itime=None*, *iiter=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\_deformation** (*mtx\_F*)

Setup macroscopic deformation gradient.

**setup\_options** ()

**update\_micro\_coors** (*ret\_val=False*)

Update microstructures coordinates according to the deformation gradient 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,      mi-
   cro_filename=None,      regenerate=False,      coefs_filename=None,
   define_args=None)
sfePy.homogenization.micmac.get_homog_coefs_nonlinear(ts,    coor,    mode,    mtx_f=None,
   term=None,    problem=None,
   iteration=None, **kwargs)
```

## sfePy.homogenization.recovery module

```
sfePy.homogenization.recovery.add_strain_rs(corrss_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(corrss,      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(corrss, 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(corrsp, pressure, vp, iel)
```

$$\tilde{\pi}^P p$$

```
sfePy.homogenization.recovery.compute_p_corr_time(corrsp, dstrains, corrsp, 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(corrsp, strain, vu, dim, iel)
```

$$\sum_{ij} \omega^{ij} e_{ij}(\mathbf{u})$$

## Notes

- iel = element number

```
sfePy.homogenization.recovery.compute_u_corr_time(corrsp, dstrains, corrsp, 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.

$$e_{ij}(\mathbf{u}) (y_j - y_j^c)$$

```
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.get_output_suffix(iel, ts, naming_scheme, format, output_format)
```

```
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,
naming_scheme='step_iel', recovery_file_tag='', define_args=None,
verbose=False)
```

```
sfePy.homogenization.recovery.recover_micro_hook_eps(micro_filename, region,
eval_var, nodal_values,
const_values, eps0, recovery_file_tag='', define_args=None, verbose=False)
```

```
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^r{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^r{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.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]

## 7.8.8 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.arpack_eigs (mtx, nev=1, which='SM')
    Calculate several eigenvalues and corresponding eigenvectors of a matrix using ARPACK from SciPy. The eigenvalues are sorted in ascending order.

sfePy.linalg.eigen.cg_eigs (mtx, rhs=None, precond=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  $xi = \text{dot}(bc.T, ref\_coors)$ .

`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, inside=True)`

Return indices of coordinates inside or outside a ball given by centre and radius.

## 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 `<=` or `>=` 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 than *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_base, ref_coors, suppress_errors=False)`

Given spatial element coordinates, find the inverse mapping for points with coordinates  $X = X(xi)$ , i.e.  $xi = xi(X)$ .

### Returns

**xi** [array] The reference element coordinates.

`sfePy.linalg.geometry.make_axis_rotation_matrix(direction, angle)`

Create a rotation matrix  $\underline{R}$  corresponding to the rotation around a general axis  $\underline{d}$  by a specified angle  $\alpha$ .

$$\underline{R} = \underline{d}\underline{d}^T + \cos(\alpha)(I - \underline{d}\underline{d}^T) + \sin(\alpha) \text{skew}(\underline{d})$$

### Parameters

**direction** [array] The rotation axis direction vector  $\underline{d}$ .

**angle** [float] The rotation angle  $\alpha$ .

#### 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, fmt='%.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

```
class sfePy.linalg.utils.MatrixAction(**kwargs)
```

```
    static from_array(arr)
    static from_function(fun, expected_shape, dtype)
    to_array()
```

```
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  $(n_1, \dots, n_r) + \text{out\_item\_shape}$ . 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.assembleId(ar_out, idx, ar_in)
```

Perform *ar\_out*[*idx*] += *ar\_in*, where items of *ar\_in* corresponding to duplicate indices in *idx* are summed together.

```
sfePy.linalg.utils.combine(seqs)
```

Same as cycle, but with general sequences.

Example:

```
In [19]: c = combine([['a', 'x'], ['b', 'c'], ['dd']])
```

```
In [20]: list(c) Out[20]: [['a', 'b', 'dd'], ['a', 'c', 'dd'], ['x', 'b', 'dd'], ['x', 'c', 'dd']]
```

```
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<=i<a, 0<=j<b, 0<=k<c, ...] 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.core.umath_tests.matrix_multiply()` if available, which is much faster than the default implementation.

The default implementation uses `numpy.sum()` and element-wise multiplication. For r-D arrays (*n<sub>1</sub>*, ..., *n<sub>r</sub>*; ?, ?) the arrays are first reshaped to (*n<sub>1</sub>* \* ... \* *n<sub>r</sub>*; ?, ?), then the dot is performed, and finally the shape is restored to (*n<sub>1</sub>*, ..., *n<sub>r</sub>*, ?, ?).

`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]],,
[[[ 0.78593989, 0.71852473]],,
[[ 0.78593989, 0.71852473]],,
[[ 0.78593989, 0.71852473]]])
```

```
>>> ar2.shape
(2, 3, 1, 2)
```

`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_12_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.permutations (seq)
sfepy.linalg.utils.print_array_info (ar)
    Print array shape and other basic information.

sfepy.linalg.utils.split_range (n_item, step)
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().

```

## 7.8.9 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
- $\nu$  : Poisson's ratio
- $K$  : bulk modulus
- $\lambda$  : Lamé's first parameter
- $\mu, G$  : 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.youngpoisson_from_stiffness(stiffness, plane='strain')
```

Compute Young's modulus and Poisson's ratio from an isotropic stiffness tensor.

## 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` [VolumeMapping 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.

`sfeypy.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  $w, \alpha, \beta$  DOFs.

`sfeypy.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  $t$ .

`sfeypy.mechanics.shell10x.lock_drilling_rotations(mtx, ebs, coefs)`

Lock the drilling rotations in the stiffness matrix.

`sfeypy.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.

`sfeypy.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

## sfeypy.mechanics.tensors module

Functions to compute some tensor-related quantities usual in continuum mechanics.

`class sfeypy.mechanics.tensors.StressTransform(def_grad, jacobian=None)`

Encapsulates functions to convert various stress tensors in the symmetric storage given the deformation state.

**get\_cauchy\_from\_2pk**(*stress\_in*)

Get the Cauchy stress given the second Piola-Kirchhoff stress.

$$\sigma_{ij} = J^{-1} F_{ik} S_{kl} F_{jl}$$

**sfePy.mechanics.tensors.dim2sym**(*dim*)

Given the space dimension, return the symmetric storage size.

**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', 0.1000000000000001)

```

**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.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()`

`sfePy.mechanics.extmods.ccontres.evaluate_contact_constraints()`

`sfePy.mechanics.extmods.ccontres.get_AABB()`

`sfePy.mechanics.extmods.ccontres.get_longest_edge_and_gps()`

`sfePy.mechanics.extmods.ccontres.init_global_search()`

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.

## 7.8.10 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.

**ncp** [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 bpsline 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 bpsline.

**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.

**ncp** [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 bpsline 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='%.6f'*)  
Write the control polygon to VTK file.

**Parameters****filename:** str Name of the VTK file.**float\_format:** str Float formating.

**write\_surface\_vtk** (*filename*, *float\_format='%.6f'*)  
Write the spline surface to VTK file.

**Parameters****filename:** str Name of the VTK file.**float\_format:** str Float formating.

`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 or 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)
    ss=[s1,s2,s3,...]
```

```
addvolume (n, v)
    v=[s1,s2,s3,...]
```

```
addvolumes (vs, off=1)
    vs=[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,
  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.  
**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.  
**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.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] Unshrinkin 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.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 indices) of the spline control point.

**dirvec** [array] The directional vector.

#### Returns

**diff** [array] The derivative field.

`static gen_cp_idxs(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 indices) 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.

## 7.8.11 sfepy.parallel package

### sfepy.parallel.evaluate module

### sfepy.parallel.parallel module

### 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 (ax, 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(axs, field, cell_tasks, gfd, output_dir, size)
```

Plot the partitioning of the domain and field DOFs.

## 7.8.12 sfePy.postprocess package

### sfePy.postprocess.domain\_specific module

Domain-specific plot functions.

All the plot functions accept the following parameters:

- *source* : Mayavi source
- *ctp* : Mayavi cell-to-point filter
- *position* :  $(x, y, z)$
- *family* : ‘point’ or ‘cell’
- *kind* : ‘scalars’, ‘vectors’ or ‘tensors’
- *name* : name of a variable

All the plot functions return: - *kind* : ‘scalars’, ‘vectors’ or ‘tensors’ - *name* : name of a variable - *active* : Mayavi module

```
class sfePy.postprocess.domain_specific.DomainSpecificPlot(fun_name, args)
```

Class holding domain-specific plot function and its parameters.

```
sfePy.postprocess.domain_specific.plot_displacements(source, ctp, bbox, position, family, kind, name, rel_scaling=1.0, color_kind=None, color_name=None, opacity=1.0)
```

Show displacements by displaying a colormap given by quantity *color\_name* on the deformed mesh.

#### Parameters

- rel\_scaling** [float] The relative scaling of displacements.
- color\_kind** [str, optional] The kind of data determining the colormap.
- color\_name** [str, optional] The name of data determining the colormap.
- opacity** [float] The surface plot opacity.

```
sfePy.postprocess.domain_specific.plot_velocity(source, ctp, bbox, position, family, kind, name, seed=‘sphere’, type=‘ribbon’, integration_direction=‘both’, seed_scale=1.0, seed_resolution=20, widget_enabled=True, color_kind=None, color_name=None, opacity=1.0, **kwargs)
```

Show velocity field by displaying streamlines and optionally a surface plot given by quantity *color\_name*.

#### Parameters

- seed** [one of (‘sphere’, ‘point’, ‘line’, ‘plane’)] The streamline seed name.
- type** [one of (‘line’, ‘ribbon’, ‘tube’)] The streamline seed line type.

**integration\_direction** [one of ('forward', 'backward', 'both')] The stream tracer integration direction.

**seed\_scale** [float] The seed size scale.

**seed\_resolution** [int] The number of seed points in a direction (depends on *seed*).

**widget\_enabled** [bool] If True, the seed widget is visible and can be interacted with.

**color\_kind** [str, optional] The kind of data determining the colormap.

**color\_name** [str, optional] The name of data determining the colormap.

**opacity** [float] The surface plot opacity.

**\*\*kwargs** [dict] Additional keyword arguments for attributes of *streamline.seed.widget*.

```
sfePy.postprocess.domain_specific.plot_warp_scalar(source, ctp, bbox, position, family,
  kind, name, rel_scaling=1.0,
  color_kind=None,
  color_name=None, opacity=1.0)
```

Show a 2D scalar field by displaying a colormap given by quantity *color\_name* on the deformed mesh deformed by the scalar in the third dimension.

#### Parameters

**rel\_scaling** [float] The relative scaling of scalar warp.

**color\_kind** [str, optional] The kind of data determining the colormap.

**color\_name** [str, optional] The name of data determining the colormap.

**opacity** [float] The surface plot opacity.

## sfePy.postprocess.plot\_cmsh module

Functions to visualize the CMesh geometry and topology.

```
sfePy.postprocess.plot_cmsh.label_global_entities(ax, cmesh, edim, color='b', font-
  size=10, **kwargs)
```

Label mesh topology entities using global ids.

```
sfePy.postprocess.plot_cmsh.label_local_entities(ax, cmesh, edim, color='b', font-
  size=10, **kwargs)
```

Label mesh topology entities using cell-local ids.

```
sfePy.postprocess.plot_cmsh.plot_cmsh(ax, cmesh, wireframe_opts=None, enti-
   ties_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\_local' 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)
```

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,
  or-
  der,           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** `sfeypy.postprocess.probes_vtk.ProbeFromFile` (*filename*, `**kwargs`)

Probe class - read a given VTK file.

## `sfeypy.postprocess.sources` module

**class** `sfeypy.postprocess.sources.FileSource` (*filename*, *watch=False*, *offscreen=True*)

General file source.

**file\_changed()**

**get\_mat\_id** (*mat\_id\_name='mat\_id'*)

Get material ID numbers of the underlying mesh elements.

**get\_step\_time** (*step=None*, *time=None*)

Set current step and time to the values closest greater or equal to either step or time. Return the found values.

**get\_ts\_info()**

**poll\_file()**

Check the source file's time stamp and notify the `self.notify_obj` in case it changed. Subclasses should implement the `file_changed()` method.

**reset()**

Reset.

**setup\_mat\_id** (*mat\_id\_name='mat\_id'*, *single\_color=False*)

**setup\_notification**(*obj, attr*)

The attribute ‘attr’ of the object ‘obj’ will be set to True when the source file is watched and changes.

**class** sfepy.postprocess.sources.**GenericFileSource**(\*args, \*\*kwargs)

File source usable with any format supported by MeshIO classes.

**add\_data\_to\_dataset**(*dataset, data*)

Add point and cell data to the dataset.

**create\_dataset**()

Create a tvtk.UnstructuredGrid dataset from the Mesh instance of the file source.

**create\_source**()

Create a VTK source from data in a SfePy-supported file.

**Notes**

All data need to be set here, otherwise time stepping will not work properly - data added by user later will be thrown away on time step change.

**file\_changed**()**get\_bounding\_box**()**get\_mat\_id**(*mat\_id\_name='mat\_id'*)

Get material ID numbers of the underlying mesh elements.

**read\_common**(*filename*)**set\_filename**(*filename, vis\_source*)**class** sfepy.postprocess.sources.**GenericSequenceFileSource**(\*args, \*\*kwargs)

File source usable with any format supported by MeshIO classes, with exception of HDF5 (.h5), for file sequences.

**file\_changed**()**read\_common**(*filename*)**set\_filename**(*filename, vis\_source*)**class** sfepy.postprocess.sources.**VTKFileSource**(*filename, watch=False, offscreen=True*)

A thin wrapper around mlab.pipeline.open().

**create\_source**()

Create a VTK file source

**get\_bounding\_box**()**set\_filename**(*filename, vis\_source*)**class** sfepy.postprocess.sources.**VTKSequenceFileSource**(\*args, \*\*kwargs)

A thin wrapper around mlab.pipeline.open() for VTK file sequences.

**create\_source**()

Create a VTK file source

**set\_filename**(*filename, vis\_source*)**sfepy.postprocess.sources.create\_file\_source**(*filename, watch=False, offscreen=True*)

Factory function to create a file source corresponding to the given file format.

## sfeypy.postprocess.time\_history module

```
sfeypy.postprocess.time_history.**average_vertex_var_in_cells** (ths_in)
    Average histories in the element nodes for each nodal variable originally requested in elements.

sfeypy.postprocess.time_history.**dump_to_vtk** (filename,           output_filename_trunk=None,
  step0=0, steps=None, fields=None, lineariza-
  tion=None)
    Dump a multi-time-step results file into a sequence of VTK files.

sfeypy.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.

```
sfeypy.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.

```
sfeypy.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.

```
sfeypy.postprocess.time_history.**save_time_history** (ths, ts, filename_out)
    Save time history and time-stepping information in a HDF5 file.
```

## sfeypy.postprocess.utils module

```
sfeypy.postprocess.utils.**get_data_ranges** (obj, return_only=False, use_names=None, fil-
  ter_names=None)
    Collect and print information on ranges of data in a dataset.
```

### Parameters

**obj** [a mayavi pipeline object] The object to probe for data.  
**return\_only** [boolean] If True, do not print the information, just return it to the caller.  
**use\_names** [list of strings] Consider only data with names in the list.  
**filter\_names** [list of strings] Consider only data with names not in the list.

### Returns

**ranges** [dict] The requested data ranges.

## 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

**selection** [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.postprocess.viewer module

```
class sfePy.postprocess.viewer.ClosingHandler
```

```
object_button_quit_changed(info)
```

```
class sfePy.postprocess.viewer.ReloadSource
```

```
class sfePy.postprocess.viewer.SetStep
```

```
init_seq_selection(name, new)
```

```
is_adjust = False
```

```
step = None
```

```
time = None
```

```
class sfePy.postprocess.viewer.Viewer(filename, watch=False, ffmpeg_options=None, output_dir='.', offscreen=False)
```

Class to automate visualization of various data using Mayavi.

It can use any format that mlab.pipeline.open() handles, e.g. a VTK format. After opening a data file, all data (point, cell, scalars, vectors, tensors) are plotted in a grid layout.

Parameters:

**watch** [bool] If True, watch the file for changes and update the mayavi pipeline automatically.

**ffmpeg\_options** [str] The ffmpeg animation encoding options.

**output\_dir** [str] The output directory, where view snapshots will be saved.

Examples:

```
>>> view = Viewer('file.vtk')
>>> view() # view with default parameters
>>> view(layout='col') # use column layout
```

```
build_mlab_pipeline(file_source=None,           is_3d=False,           layout='rowcol',
                     scalar_mode='iso_surface',      vector_mode='arrows_norm',
                     rel_scaling=None, clamping=False, ranges=None, is_scalar_bar=False,
                     is_wireframe=False,          opacity=None,       subdomains_args=None,
                     rel_text_width=None,         filter_names=None, group_names=None,
                     only_names=None, domain_specific=None, **kwargs)
Sets self.source, self.is_3d_data
```

```
call_empty(*args, **kwargs)
```

```
call_mlab(scene=None,    show=True,    is_3d=False,    view=None,    roll=None,    parallel_projection=False,
          fgcolor=(0.0, 0.0, 0.0),   bgcolor=(1.0, 1.0, 1.0),
          colormap='blue-red',      layout='rowcol',    scalar_mode='iso_surface',    vector_mode='arrows_norm',
          rel_scaling=None,         clamping=False,   ranges=None,   is_scalar_bar=False,
          is_wireframe=False,       opacity=None,     subdomains_args=None,
          rel_text_width=None,      filter_names=None, group_names=None,
          only_names=None, domain_specific=None)
```

By default, all data (point, cell, scalars, vectors, tensors) are plotted in a grid layout, except data named ‘node\_groups’, ‘mat\_id’ which are usually not interesting.

### Parameters

**show** [bool] Call mlab.show().

**is\_3d** [bool] If True, use scalar cut planes instead of surface for certain datasets. Also sets 3D view mode.

**view** [tuple] Azimuth, elevation angles, distance and focal point as in *mlab.view()*.

**roll** [float] Roll angle tuple as in mlab.roll().

**parallel\_projection: bool** If True, use parallel projection.

**fgcolor** [tuple of floats (R, G, B)] The foreground color, that is the color of all text annotation labels (axes, orientation axes, scalar bar labels).

**bgcolor** [tuple of floats (R, G, B)] The background color.

**colormap** [str] The colormap name.

**layout** [str] Grid layout for placing the datasets. Possible values are: ‘row’, ‘col’, ‘rowcol’, ‘colrow’.

**scalar\_mode** [str] Mode for plotting scalars and tensor magnitudes, one of ‘cut\_plane’, ‘iso\_surface’, ‘both’.

**vector\_mode** [str] Mode for plotting vectors, one of ‘arrows’, ‘norm’, ‘arrows\_norm’, ‘warp\_norm’.

**rel\_scaling** [float] Relative scaling of glyphs for vector datasets.

**clamping** [bool] Clamping for vector datasets.

**ranges** [dict] List of data ranges in the form {name : (min, max), ...}.

**is\_scalar\_bar** [bool] If True, show a scalar bar for each data.

**is\_wireframe** [bool] If True, show a wireframe of mesh surface bar for each data.

**opacity** [float] Global surface and wireframe opacity setting in [0.0, 1.0],

**subdomains\_args** [tuple] Tuple of (mat\_id\_name, threshold\_limits, single\_color), see [add\\_subdomains\\_surface\(\)](#), or None.

**rel\_text\_width** [float] Relative text width.

**fig\_filename** [str] File name for saving the resulting scene figure.

**resolution** [tuple] Scene and figure resolution. If None, it is set automatically according to the layout.

**filter\_names** [list of strings] Omit the listed datasets. If None, it is initialized to ['node\_groups', 'mat\_id']. Pass [] if you need no filtering.

**only\_names** [list of strings] Draw only the listed datasets. If None, it is initialized all names besides those in filter\_names.

**group\_names** [list of tuples] List of data names in the form [(name1, ..., nameN), (...)]. Plots of data named in each group are superimposed. Repetitions of names are possible.

**step** [int, optional] If not None, the time step to display. The closest higher step is used if the desired one is not available. Has precedence over *time*.

**time** [float, optional] If not None, the time of the time step to display. The closest higher time is used if the desired one is not available.

**anti\_aliasing** [int] Value of anti-aliasing.

**domain\_specific** [dict] Domain-specific drawing functions and configurations.

**encode\_animation** (filename, format, ffmpeg\_options=None)

**get\_animation\_info** (filename, add\_output\_dir=True, last\_step=None)

**get\_data\_names** (source=None, detailed=False)

**get\_size\_hint** (layout, resolution=None)

**render\_scene** (scene, options)  
Render the scene, preferably after it has been activated.

**reset\_view()**

**save\_animation** (filename, steps=None, times=None)  
Animate the current scene view for the selected time steps or times and save a snapshot of each view.

**save\_image** (filename)  
Save a snapshot of the current scene.

**set\_source\_filename** (filename)

**show\_scalar\_bars** (scalar\_bars)

---

**class** sfepy.postprocess.viewer.**ViewerGUI** (fgcolor=(0.0, 0.0, 0.0), bgcolor=(1.0, 1.0, 1.0), \*\*traits)

sfepy.postprocess.viewer.**add\_glyphs** (obj, position, bbox, rel\_scaling=None, scale\_factor='auto', clamping=False, color=None)

sfepy.postprocess.viewer.**add\_iso\_surface** (obj, position, contours=10, opacity=1.0)

sfepy.postprocess.viewer.**add\_scalar\_cut\_plane** (obj, position, normal, opacity=1.0)

sfepy.postprocess.viewer.**add\_subdomains\_surface** (obj, position, mat\_id\_name='mat\_id', threshold\_limits=(None, None), \*\*kwargs)

sfepy.postprocess.viewer.**add\_surf** (obj, position, opacity=1.0)

sfepy.postprocess.viewer.**add\_text** (obj, position, text, width=None, color=(0, 0, 0))

```
sfepy.postprocess.viewer.add_vector_cut_plane(obj, position, normal, bbox,
   rel_scaling=None, scale_factor='auto',
   clamping=False, opacity=1.0)

sfepy.postprocess.viewer.get_glyphs_scale_factor(rng, rel_scaling, bbox)

sfepy.postprocess.viewer.get_opacities(opacity)
    Provide defaults for all supported opacity settings.

sfepy.postprocess.viewer.get_position_counts(n_data, layout)

sfepy.postprocess.viewer.make_animation(filename, view, roll, anim_format, options,
   steps=None, times=None, reuse_viewer=None)

sfepy.postprocess.viewer.set_colormap(source, colormap_name)
    Set the given colormap to all look-up tables depending on the given source.
```

## 7.8.13 sfepy.solvers package

### 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.PysparseEigenvalueSolver(conf, **kwargs)
```

Pysparse-based eigenvalue solver for sparse symmetric problems.

Kind: ‘eig.pysparse’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

#### Parameters

**i\_max** [int (default: 100)] The maximum number of iterations.

**eps\_a** [float (default: 1e-05)] The absolute tolerance for the convergence.

**tau** [float (default: 0.0)] The target value.

**method** [{‘cgs’, ‘qmrs’} (default: ‘qmrs’)] The linear iterative solver supported by pysparse.

**verbosity** [int (default: 0)] The pysparse verbosity level.

**strategy** [{0, 1} (default: 1)] The shifting and sorting strategy of JDSYM: strategy=0 enables the default JDSYM algorithm, strategy=1 enables JDSYM to avoid convergence to eigenvalues smaller than *tau*.

\* [\*] Additional parameters supported by the solver.

**name = ‘eig.pysparse’**

**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
- \* [\*] 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:

**name = ‘eig.sgscipy’**

`sfepy.solvers.eigen.eig(mtx_a, mtx_b=None, n_eigs=None, eigenvectors=True, return_time=None, method='eig.scipy', **kwargs)`  
Utility function that constructs an eigenvalue solver given by *method*, calls it and returns solution.

`sfepy.solvers.eigen.standard_call(call)`  
Decorator handling argument preparation and timing for eigensolvers.

## sfepy.solvers.ls module

**class sfepy.solvers.ls.MUMPSParallelSolver (conf, \*\*kwargs)**  
Interface to MUMPS parallel solver.

Kind: ‘ls.mumps\_par’

For common configuration parameters, see [Solver](#).

Specific configuration parameters:

**Parameters**

- memory\_relaxation** [int (default: 20)] The percentage increase in the estimated working space.

**name = ‘ls.mumps\_par’**

```
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.

**memory\_relaxation** [int (default: 20)] The percentage increase in the estimated working space.

**name** = ‘ls.mumps’

**presolve** (*mtx*, *presolve\_flag=False*)

```
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.

**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.

**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*)

**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 <lambda> at 0x2b7ffc69d6e0>)] 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\_staben\_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.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.  
**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.

```

name = 'ls.scipy_direct'

presolve (mtx)

```

**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 <lambd> at 0x2b7ffc69d230>)] 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.

**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.

**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.ls\_mumps module

```
class sfePy.solvers.ls_mumps.MumpsSolver(is_sym=False, mpi_comm=None, system='real',
   silent=True, mem_relax=20)
    MUMPS object.
```

```
expand_schur(x2)
    Expand the Schur local solution on the complete solution.
```

### Parameters

**x2** [array] The local Schur solution.

### Returns

**x** [array] The global solution.

```
get_schur(schur_list)
```

Get the Schur matrix and the condensed right-hand side vector.

### Parameters

**schur\_list** [array] The list of the Schur DOFs (indexing starts with 1).

### Returns

**schur\_arr** [array] The Schur matrix of order ‘schur\_size’.

**schur\_rhs** [array] The reduced right-hand side vector.

```
set_mtx_centralized(mtx)
```

Set the sparse matrix.

### Parameters

**mtx** [scipy sparse martix] The sparse matrix in COO format.

```
set_rcd_centralized(ir, ic, data, n)
```

Set the matrix by row and column indicies and data vector. The matrix shape is determined by the maximal values of row and column indicies. The indices start with 1.

### Parameters

**ir** [array] The row idicies.

**ic** [array] The column idicies.

**data** [array] The matrix entries.

**n** [int] The matrix dimension.

```
set_rhs(rhs)
```

Set the right hand side of the linear system.

```
set_silent()
```

```
set_verbose()  
sfepy.solvers.ls_mumps.coo_is_symmetric(mtx, tol=1e-06)  
sfepy.solvers.ls_mumps.load_library(libname)  
    Load shared library in a system dependent way.  
sfepy.solvers.ls_mumps.load_mumps_libraries()  
sfepy.solvers.ls_mumps.mumps_pcomplex  
    alias of LP_c_double  
sfepy.solvers.ls_mumps.mumps_preal  
    alias of LP_c_double  
class sfepy.solvers.ls_mumps.mumps_struct_c_4  
  
    a  
        Structure/Union member  
    a_elt  
        Structure/Union member  
    a_loc  
        Structure/Union member  
    cntl  
        Structure/Union member  
    colsca  
        Structure/Union member  
    comm_fortran  
        Structure/Union member  
    deficiency  
        Structure/Union member  
    eltptr  
        Structure/Union member  
    eltvar  
        Structure/Union member  
    icntl  
        Structure/Union member  
    info  
        Structure/Union member  
    infog  
        Structure/Union member  
    instance_number  
        Structure/Union member  
    irhs_ptr  
        Structure/Union member  
    irhs_sparse  
        Structure/Union member  
    irn  
        Structure/Union member
```

**irn\_loc**  
Structure/Union member

**isol\_loc**  
Structure/Union member

**jcn**  
Structure/Union member

**jcn\_loc**  
Structure/Union member

**job**  
Structure/Union member

**listvar\_schur**  
Structure/Union member

**lredrhs**  
Structure/Union member

**lrhs**  
Structure/Union member

**lwk\_user**  
Structure/Union member

**mapping**  
Structure/Union member

**mblock**  
Structure/Union member

**n**  
Structure/Union member

**nblock**  
Structure/Union member

**nelt**  
Structure/Union member

**npcol**  
Structure/Union member

**nprow**  
Structure/Union member

**nrhs**  
Structure/Union member

**nz**  
Structure/Union member

**nz\_alloc**  
Structure/Union member

**nz\_loc**  
Structure/Union member

**nz\_rhs**  
Structure/Union member

**ooc\_prefix**  
Structure/Union member

**ooc\_tmpdir**  
Structure/Union member

**par**  
Structure/Union member

**perm\_in**  
Structure/Union member

**pivnul\_list**  
Structure/Union member

**redrhs**  
Structure/Union member

**rhs**  
Structure/Union member

**rhs\_sparse**  
Structure/Union member

**rinfo**  
Structure/Union member

**rinfog**  
Structure/Union member

**rowsca**  
Structure/Union member

**schur**  
Structure/Union member

**schur\_lld**  
Structure/Union member

**schur\_mloc**  
Structure/Union member

**schur\_nloc**  
Structure/Union member

**size\_schur**  
Structure/Union member

**sol\_loc**  
Structure/Union member

**sym**  
Structure/Union member

**sym\_perm**  
Structure/Union member

**uns\_perm**  
Structure/Union member

**version\_number**  
Structure/Union member

```
    wk_user
        Structure/Union member

    write_problem
        Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struct_c_5_0

    a
        Structure/Union member

    a_elt
        Structure/Union member

    a_loc
        Structure/Union member

    cntl
        Structure/Union member

    colsca
        Structure/Union member

    colsca_from_mumps
        Structure/Union member

    comm_fortran
        Structure/Union member

    deficiency
        Structure/Union member

    dkeep
        Structure/Union member

    eltptr
        Structure/Union member

    eltvar
        Structure/Union member

    icntl
        Structure/Union member

    info
        Structure/Union member

    infog
        Structure/Union member

    instance_number
        Structure/Union member

    irhs_ptr
        Structure/Union member

    irhs_sparse
        Structure/Union member

    irn
        Structure/Union member
```

**irn\_loc**  
Structure/Union member

**isol\_loc**  
Structure/Union member

**jcn**  
Structure/Union member

**jcn\_loc**  
Structure/Union member

**job**  
Structure/Union member

**keep**  
Structure/Union member

**keep8**  
Structure/Union member

**listvar\_schur**  
Structure/Union member

**lredrhs**  
Structure/Union member

**lrhs**  
Structure/Union member

**lwk\_user**  
Structure/Union member

**mapping**  
Structure/Union member

**mblock**  
Structure/Union member

**n**  
Structure/Union member

**nblock**  
Structure/Union member

**nelt**  
Structure/Union member

**npcol**  
Structure/Union member

**nprow**  
Structure/Union member

**nrhs**  
Structure/Union member

**nz**  
Structure/Union member

**nz\_alloc**  
Structure/Union member

**nz\_loc**  
Structure/Union member

**nz\_rhs**  
Structure/Union member

**ooc\_prefix**  
Structure/Union member

**ooc\_tmpdir**  
Structure/Union member

**par**  
Structure/Union member

**perm\_in**  
Structure/Union member

**pivnul\_list**  
Structure/Union member

**redrhs**  
Structure/Union member

**rhs**  
Structure/Union member

**rhs\_sparse**  
Structure/Union member

**rinfo**  
Structure/Union member

**rinfog**  
Structure/Union member

**rowsca**  
Structure/Union member

**rowsca\_from\_mumps**  
Structure/Union member

**schur**  
Structure/Union member

**schur\_lld**  
Structure/Union member

**schur\_mloc**  
Structure/Union member

**schur\_nloc**  
Structure/Union member

**size\_schur**  
Structure/Union member

**sol\_loc**  
Structure/Union member

**sym**  
Structure/Union member

```
sym_perm
    Structure/Union member

uns_perm
    Structure/Union member

version_number
    Structure/Union member

wk_user
    Structure/Union member

write_problem
    Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struct_c_5_1

    a
        Structure/Union member

    a_elt
        Structure/Union member

    a_loc
        Structure/Union member

    cntl
        Structure/Union member

    colsca
        Structure/Union member

    colsca_from_mumps
        Structure/Union member

    comm_fortran
        Structure/Union member

    deficiency
        Structure/Union member

    dkeep
        Structure/Union member

    eltptr
        Structure/Union member

    eltvar
        Structure/Union member

    icntl
        Structure/Union member

    info
        Structure/Union member

    infog
        Structure/Union member

    instance_number
        Structure/Union member
```

**irhs\_ptr**  
Structure/Union member

**irhs\_sparse**  
Structure/Union member

**irn**  
Structure/Union member

**irn\_loc**  
Structure/Union member

**isol\_loc**  
Structure/Union member

**jcn**  
Structure/Union member

**jcn\_loc**  
Structure/Union member

**job**  
Structure/Union member

**keep**  
Structure/Union member

**keep8**  
Structure/Union member

**listvar\_schur**  
Structure/Union member

**lredrhs**  
Structure/Union member

**lrhs**  
Structure/Union member

**lwk\_user**  
Structure/Union member

**mapping**  
Structure/Union member

**mblock**  
Structure/Union member

**n**  
Structure/Union member

**nblock**  
Structure/Union member

**nelt**  
Structure/Union member

**nnz**  
Structure/Union member

**nnz\_loc**  
Structure/Union member

**npcol**  
Structure/Union member

**nprow**  
Structure/Union member

**nrhs**  
Structure/Union member

**nz**  
Structure/Union member

**nz\_alloc**  
Structure/Union member

**nz\_loc**  
Structure/Union member

**nz\_rhs**  
Structure/Union member

**ooc\_prefix**  
Structure/Union member

**ooc\_tmpdir**  
Structure/Union member

**par**  
Structure/Union member

**perm\_in**  
Structure/Union member

**pivnul\_list**  
Structure/Union member

**redrhs**  
Structure/Union member

**rhs**  
Structure/Union member

**rhs\_sparse**  
Structure/Union member

**rinfo**  
Structure/Union member

**rinfog**  
Structure/Union member

**rowsca**  
Structure/Union member

**rowsca\_from\_mumps**  
Structure/Union member

**save\_dir**  
Structure/Union member

**save\_prefix**  
Structure/Union member

```
schur
    Structure/Union member

schur_lld
    Structure/Union member

schur_mloc
    Structure/Union member

schur_nloc
    Structure/Union member

size_schur
    Structure/Union member

sol_loc
    Structure/Union member

sym
    Structure/Union member

sym_perm
    Structure/Union member

uns_perm
    Structure/Union member

version_number
    Structure/Union member

wk_user
    Structure/Union member

write_problem
    Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struct_c_x

aux
    Structure/Union member

comm_fortran
    Structure/Union member

icntl
    Structure/Union member

job
    Structure/Union member

par
    Structure/Union member

sym
    Structure/Union member

sfepy.solvers.ls_mumps_parallel module

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: 2.2204460492503131e-16)] The float considered to be machine “zero”.

**lin\_red** [float (default: 1.0)] The linear system solution error should be smaller than  $(\text{eps\_a} * \text{lin\_red})$ , 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.

**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  $\geq 1$ , check the tangent matrix using finite differences. If 2, plot the resulting sparsity patterns.

**delta** [float (default: 1e-06)] If *check*  $\geq 1$ , 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.

**is\_linear** [bool (default: False)] If True, the problem is considered to be linear.

**\_\_call\_\_** (*vec\_x0*, *conf=None*, *fun=None*, *fun\_grad=None*, *lin\_solver=None*, *iter\_hook=None*, *status=None*)  
Nonlinear system solver call.

Solves a nonlinear system  $f(x) = 0$  using the Newton method with backtracking line-search, starting with an initial guess  $x^0$ .

#### Parameters

**vec\_x0** [array] The initial guess vector  $x_0$ .  
**conf** [Struct instance, optional] The solver configuration parameters,  
**fun** [function, optional] The function  $f(x)$  whose zero is sought - the residual.  
**fun\_grad** [function, optional] The gradient of  $f(x)$  - the tangent matrix.  
**lin\_solver** [LinearSolver instance, optional] The linear solver for each nonlinear iteration.  
**iter\_hook** [function, optional] User-supplied function to call before each iteration.  
**status** [dict-like, optional] The user-supplied object to hold convergence statistics.

## Notes

- The optional parameters except *iter\_hook* and *status* need to be given either here or upon *Newton* construction.
- Setting *conf.is\_linear == True* means a pre-assembled and possibly pre-solved matrix. This is mostly useful for linear time-dependent problems.

```
__init__(conf, **kwargs)  
__metaclass__  
    alias of SolverMeta  
__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.  $\|\delta x\| < \text{epsilon}_s \|x\|$   
**call** (vec\_x0, conf=None, fun=None, fun\_grad=None, lin\_solver=None, iter\_hook=None, status=None, pmtx=None, prhs=None, comm=None)  
**init** (conf, pmtx=None, prhs=None, comm=None, \*\*kwargs)

```

__metaclass__
    alias of SolverMeta

__module__ = 'sfepy.solvers.nls'
name = 'nls.petsc'

class sfepy.solvers.nls.ScipyBroyden(conf, **kwargs)
    Interface to Broyden and Anderson solvers from scipy.optimize.

    Kind: 'nls.scipy_broyden_like'

    For common configuration parameters, see Solver.

    Specific configuration parameters:

        Parameters

            method [str (default: 'anderson')] The name of the solver in scipy.optimize.

            i_max [int (default: 10)] The maximum number of iterations.

            alpha [float (default: 0.9)] See scipy.optimize.

            M [float (default: 5)] See scipy.optimize.

            f_tol [float (default: 1e-06)] See scipy.optimize.

            w0 [float (default: 0.1)] See scipy.optimize.

            call (vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None)

            init (conf, **kwargs)

            __metaclass__
                alias of SolverMeta

            __module__ = 'sfepy.solvers.nls'
            name = 'nls.scipy_broyden_like'

            set_method(conf)

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.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  $\geq 1$ , check the tangent matrix using finite differences. If 2, plot the resulting sparsity patterns.  
**delta** [float (default: 1e-06)] If *check*  $\geq 1$ , 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: 2.2204460492503131e-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.

**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

- tau\_red <= 1.0; if tau is None: tau = tau\_red \* 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.qeigen.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: 2.2204460492503131e-16)] The float considered to be machine “zero”.
lin_red [float (default: 1.0)] The linear system solution error should be smaller than  $(\text{eps\_a} * \text{lin\_red})$ , 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 [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.
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,    sta-
  tus=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.

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,    sta-
  tus=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,    con-
  text=None, **kwargs)

Abstract optimization solver class.

class sfepy.solvers.solvers.QuadraticEVPSolver (conf,      mtx_m=None,      mtx_d=None,
   mtx_k=None,     n_eigs=None,     eigenvec-
   tors=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.

**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**

Metaclass for solver classes that automatically adds configuration parameters to the solver class docstring from the *\_parameters* class attribute.

```
class sfepy.solvers.solvers.TimeSteppingSolver(conf, nls=None, status=None, context=None, **kwargs)
    Abstract time stepping solver class.

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)
    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.

**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, dts=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\_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 nonlinear 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.

```
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'

class sfepy.solvers.ts_solvers.ElastodynamicsBaseTS(conf, nls=None, context=None, **kwargs)
    Base class for elastodynamics solvers.

    Assumes block-diagonal matrix in u, v, a.
```

- get\_a0**(nls, u0, v0)
- get\_initial\_vec**(nls, vec0, init\_fun, prestep\_fun, poststep\_fun)
- get\_matrices**(nls, vec)

```
class sfepy.solvers.ts_solvers.GeneralizedAlphaTS(conf, nls=None, context=None, **kwargs)
    Solve elastodynamics problems by the generalized  $\alpha$  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  $\alpha_m$ ,  $\alpha_f$  (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- $\alpha$  method.

- $\alpha_f = 0$  corresponds to the WBZ- $\alpha$  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- $\alpha$  Method” ASME Journal of Applied Mechanics, 60, 371:375, 1993.

Kind: ‘ts.generalized\_alpha’

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.  
**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  $\alpha_m$ .  
**alpha\_f** [float] The parameter  $\alpha_f$ .  
**beta** [float] The Newmark-like parameter  $\beta$ .  
**gamma** [float] The Newmark-like parameter  $\gamma$ .  
**create\_nlst** (*nls*, *dt*, *alpha\_m*, *alpha\_f*, *gamma*, *beta*, *u0*, *v0*, *a0*)  
**name = ‘ts.generalized\_alpha’**

**class** `sfePy.solvers.ts_solvers.NewmarkTS` (*conf*, *nls=None*, *context=None*, *\*\*kwargs*)  
Solve elastodynamics problems by the Newmark method.

The method was introduced in [1]. Common settings [2]:

| <b>name</b>          | <b>kind</b> | <b>beta</b> | <b>gamma</b> | <b>Omega_crit</b> |
|----------------------|-------------|-------------|--------------|-------------------|
| 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

**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.

**beta** [float (default: 0.25)] The Newmark method parameter beta.

**gamma** [float (default: 0.5)] The Newmark method parameter gamma.

**create\_nlst** (*nls*, *dt*, *gamma*, *beta*, *u0*, *v0*, *a0*)

**name = ‘ts.newmark’**

```
class sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver(conf,      nls=None,      con-
   text=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:

**name = ‘ts.stationary’**

```
class sfepy.solvers.ts_solvers.VelocityVerletTS(conf,      nls=None,      context=None,
  **kwargs)
```

Solve elastodynamics problems by the velocity-Verlet method.

The algorithm can be found in [1].

[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

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.

**create\_nlst** (*nls*, *dt*, *u0*, *v0*, *a0*)

**name = ‘ts.velocity\_verlet’**

`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 than 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(size, num)`

`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.

## 7.8.14 `sfePy.terms` package

### Term Overview

#### Term Syntax

In general, the syntax of a term call is:

```
<term name>.<i>.<r>(<arg1>, <arg2>, ... ),
```

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 7.3: Notation.

| symbol                                     | meaning                                                                                                                                                                                 |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\Omega$                                   | volume (sub)domain                                                                                                                                                                      |
| $\Gamma$                                   | surface (sub)domain                                                                                                                                                                     |
| $d$                                        | dimension of space                                                                                                                                                                      |
| $t$                                        | time                                                                                                                                                                                    |
| $y$                                        | any function                                                                                                                                                                            |
| $\underline{y}$                            | any vector function                                                                                                                                                                     |
| $\underline{n}$                            | unit outward normal                                                                                                                                                                     |
| $q, s$                                     | scalar test function                                                                                                                                                                    |
| $p, r$                                     | scalar unknown or parameter function                                                                                                                                                    |
| $\bar{p}$                                  | scalar parameter function                                                                                                                                                               |
| $\underline{v}$                            | vector test function                                                                                                                                                                    |
| $\underline{w}, \underline{u}$             | vector unknown or parameter function                                                                                                                                                    |
| $\underline{b}$                            | vector parameter function                                                                                                                                                               |
| $\underline{\underline{e}}(\underline{u})$ | Cauchy strain tensor ( $\frac{1}{2}((\nabla u) + (\nabla u)^T)$ )                                                                                                                       |
| $\underline{\underline{F}}$                | 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 $E_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_m}{\partial X_i} \frac{\partial u_m}{\partial X_j})$ |
| $\underline{S}$                            | second Piola-Kirchhoff stress tensor                                                                                                                                                    |
| $\underline{f}$                            | vector volume forces                                                                                                                                                                    |
| $f$                                        | scalar volume force (source)                                                                                                                                                            |
| $\rho$                                     | density                                                                                                                                                                                 |
| $\nu$                                      | kinematic viscosity                                                                                                                                                                     |
| $c$                                        | any constant                                                                                                                                                                            |
| $\delta_{ij}, \underline{\underline{I}}$   | Kronecker delta, identity matrix                                                                                                                                                        |
| $\text{tr } \bullet$                       | trace of a second order tensor ( $\sum_{i=1}^d \bullet_{ii}$ )                                                                                                                          |
| $\text{dev } \bullet$                      | deviator of a second order tensor ( $\bullet - \frac{1}{d} \text{tr } \bullet \underline{\underline{I}}$ )                                                                              |
| $T_K \in \mathcal{T}_h$                    | $K$ -th element of triangulation (= mesh) $\mathcal{T}_h$ of domain $\Omega$                                                                                                            |
| $K \leftarrow \mathcal{I}_h$               | $K$ 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 7.4: Term name prefixes.

| prefix | meaning       | evaluation modes                  | meaning                                                                                                                                  |
|--------|---------------|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| dw     | discrete weak | 'weak'                            | terms having a virtual (test) argument and zero or more unknown arguments, used for FE assembling                                        |
| d      | discrete      | 'eval', 'el_eval'                 | terms having all arguments known, the result is the value of the term integral evaluation                                                |
| ev     | evaluate      | 'eval', 'el_eval', 'el_avg', 'qp' | terms having all arguments known and supporting all evaluation modes except 'weak' (no virtual variables in arguments, no FE assembling) |

## 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 devided 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*

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 7.5: Basic terms

| name/class                                                  | arguments                                                                                                                      | definition                                                                                                                                                                                                                   |
|-------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_advect_div_free</code><br><i>AdvectDivFreeTerm</i> | <material>,<br><virtual>,<br><state>                                                                                           | $\int_{\Omega} \nabla \cdot (\underline{y} p) q = \int_{\Omega} \left( (\nabla \cdot \underline{y}) + \underline{y} \cdot \nabla \right) p q$                                                                                |
| <code>dw_bc_newton</code><br><i>BCNewtonTerm</i>            | <material_1>,<br><material_2>,<br><virtual>,<br><state>                                                                        | $\int_{\Gamma} \alpha q (p - p_{\text{outer}})$                                                                                                                                                                              |
| <code>dw_biot</code><br><i>BiotTerm</i>                     | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual><br><material>,<br><parameter_v>,<br><parameter_s> | $\int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}), \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u})$                                                                                                                       |
| <code>ev_biot_stress</code><br><i>BiotStressTerm</i>        | <material>,<br><parameter>                                                                                                     | $-\int_{\Omega} \alpha_{ij} \bar{p}$<br>vector for $K \leftarrow \mathcal{I}_h : -\int_{T_K} \alpha_{ij} \bar{p} / \int_{T_K} 1$<br>$-\alpha_{ij} \bar{p} _{qp}$                                                             |
| <code>ev_cauchy_strain</code><br><i>CauchyStrainTerm</i>    | <parameter>                                                                                                                    | $\int_{\Omega} \underline{\underline{e}}(\underline{w})$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$<br>$\underline{\underline{e}}(\underline{w}) _{qp}$ |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                   | arguments                                                                                 | definition                                                                                                                                                                                                                               |
|--------------------------------------------------------------|-------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ev_cauchy_strain_s</code><br><i>CauchyStrainSTerm</i>  | <parameter>                                                                               | $\int_{\Gamma} \underline{\underline{e}}(\underline{w})$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1</math></p> $\underline{\underline{e}}(\underline{w}) _{qp}$ |
| <code>ev_cauchy_stress</code><br><i>CauchyStressTerm</i>     | <material>,<br><parameter>                                                                | $\int_{\Omega} D_{ijkl} e_{kl}(\underline{w})$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} D_{ijkl} e_{kl}(\underline{w}) / \int_{T_K} 1</math></p> $D_{ijkl} e_{kl}(\underline{w}) _{qp}$                               |
| <code>dw_contact</code><br><i>ContactTerm</i>                | <material>,<br><virtual>,<br><state>                                                      | $\int_{\Gamma_c} \varepsilon_N \langle g_N(u) \rangle n \underline{v}$                                                                                                                                                                   |
| <code>dw_contact_plane</code><br><i>ContactPlaneTerm</i>     | <material_f>,<br><material_n>,<br><material_a>,<br><material_b>,<br><virtual>,<br><state> | $\int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) n$                                                                                                                                                                                |
| <code>dw_contact_sphere</code><br><i>ContactSphereTerm</i>   | <material_f>,<br><material_c>,<br><material_r>,<br><virtual>,<br><state>                  | $\int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}(\underline{u})$                                                                                                                                                     |
| <code>dw_convect</code><br><i>ConvectTerm</i>                | <virtual>,<br><state>                                                                     | $\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$                                                                                                                                                         |
| <code>dw_convect_v_grad_s</code><br><i>ConvectVGradSTerm</i> | <virtual>,<br><state_v>,<br><state_s>                                                     | $\int_{\Omega} q(\underline{u} \cdot \nabla p)$                                                                                                                                                                                          |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                            | arguments                                                                                                                      | definition                                                                                                                                                                                                                                                                |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ev_def_grad<br><i>DeformationGradientTerm</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_diffusion<br><i>DiffusionTerm</i>                  | <material>,<br><virtual>,<br><state><br><material>,<br><parameter_1>,<br><parameter_2>                                         | $\int_{\Omega} K_{ij} \nabla_i q \nabla_j p, \int_{\Omega} K_{ij} \nabla_i \bar{p} \nabla_j r$                                                                                                                                                                            |
| dw_diffusion_coupling<br><i>DiffusionCoupling</i>     | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual><br><material>,<br><parameter_1>,<br><parameter_2> | $\int_{\Omega} p K_j \nabla_j q, \int_{\Omega} q K_j \nabla_j p$                                                                                                                                                                                                          |
| dw_diffusion_r<br><i>DiffusionRTerm</i>               | <material>,<br><virtual>                                                                                                       | $\int_{\Omega} K_j \nabla_j q$                                                                                                                                                                                                                                            |
| ev_diffusion_velocity<br><i>DiffusionVelocityTerm</i> | <material>,<br><parameter>                                                                                                     | $-\int_{\Omega} K_{ij} \nabla_j \bar{p}$ <p>vector for <math>K \leftarrow \mathcal{I}_h : -\int_{T_K} K_{ij} \nabla_j \bar{p} / \int_{T_K} 1</math></p> $-K_{ij} \nabla_j \bar{p}$                                                                                        |
| dw_div<br><i>DivOperatorTerm</i>                      | <opt_material>,<br><virtual>                                                                                                   | $\int_{\Omega} \nabla \cdot \underline{v} \text{ or } \int_{\Omega} c \nabla \cdot \underline{v}$                                                                                                                                                                         |
| ev_div<br><i>DivTerm</i>                              | <parameter>                                                                                                                    | $\int_{\Omega} \nabla \cdot \underline{u}$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla \cdot \underline{u} / \int_{T_K} 1</math></p> $(\nabla \cdot \underline{u}) _{qp}$                                                                          |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                          | arguments                                                                                                                                                                  | definition                                                                                                                                                                                                                                                                      |
|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_div_grad</code><br><i>DivGradTerm</i>                      | <code>&lt;opt_material&gt;, &lt;virtual&gt;, &lt;state&gt;</code><br><code>&lt;opt_material&gt;, &lt;parameter_1&gt;, &lt;parameter_2&gt;</code>                           | $\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \nabla \underline{u} : \nabla \underline{w}$<br>$\int_{\Omega} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nabla \underline{u} : \nabla \underline{w}$                        |
| <code>dw_elastic_wave</code><br><i>ElasticWaveTerm</i>              | <code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                                                                        | $\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) g_{kl}(\underline{u})$                                                                                                                                                                                                            |
| <code>dw_elastic_wave_cauchy</code><br><i>ElasticWaveCauchyTerm</i> | <code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;virtual&gt;, &lt;state&gt;</code><br><code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;state&gt;, &lt;virtual&gt;</code> | $\int_{\Omega} D_{ijkl} g_{ij}(\underline{v}) e_{kl}(\underline{u}), \int_{\Omega} D_{ijkl} g_{ij}(\underline{u}) e_{kl}(\underline{v})$                                                                                                                                        |
| <code>dw_electric_source</code><br><i>ElectricSourceTerm</i>        | <code>&lt;material&gt;, &lt;virtual&gt;, &lt;parameter&gt;</code>                                                                                                          | $\int_{\Omega} cs(\nabla \phi)^2$                                                                                                                                                                                                                                               |
| <code>ev_grad</code><br><i>GradTerm</i>                             | <code>&lt;parameter&gt;</code>                                                                                                                                             | $\int_{\Omega} \nabla p \text{ or } \int_{\Omega} \nabla \underline{w}$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla p / \int_{T_K} 1 \text{ or } \int_{T_K} \nabla \underline{w} / \int_{T_K} 1$<br>$(\nabla p) _{qp} \text{ or } \nabla \underline{w} _{qp}$ |
| <code>dw_jump</code><br><i>SurfaceJumpTerm</i>                      | <code>&lt;opt_material&gt;, &lt;virtual&gt;, &lt;state_1&gt;, &lt;state_2&gt;</code>                                                                                       | $\int_{\Gamma} c q(p_1 - p_2)$                                                                                                                                                                                                                                                  |
| <code>dw_laplace</code><br><i>LaplaceTerm</i>                       | <code>&lt;opt_material&gt;, &lt;virtual&gt;, &lt;state&gt;</code><br><code>&lt;opt_material&gt;, &lt;parameter_1&gt;, &lt;parameter_2&gt;</code>                           | $\int_{\Omega} c \nabla q \cdot \nabla p, \int_{\Omega} c \nabla \bar{p} \cdot \nabla r$                                                                                                                                                                                        |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                            | arguments                                                                                                                    | definition                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_lin_convect</code><br><i>LinearConvectTerm</i>               | <virtual>,<br><parameter>,<br><state>                                                                                        | $\int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v}$<br>$((\underline{b} \cdot \nabla) \underline{u}) _{qp}$                                                                                                                 |
| <code>dw_lin_convect2</code><br><i>LinearConvect2Term</i>             | <material>,<br><virtual>,<br><state>                                                                                         | $\int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v}$<br>$((\underline{b} \cdot \nabla) \underline{u}) _{qp}$                                                                                                                 |
| <code>dw_lin_elastic</code><br><i>LinearElasticTerm</i>               | <material>,<br><virtual>,<br><state><br><material>,<br><parameter_1>,<br><parameter_2>                                       | $\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$                                                                                                                                                                                     |
| <code>dw_lin_elastic_iso</code><br><i>LinearElasticIsotropyTerm</i>   | <material_1>,<br><material_2>,<br><virtual>,<br><state><br><material_1>,<br><material_2>,<br><parameter_1>,<br><parameter_2> | $\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \text{ with } D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$                                                                      |
| <code>dw_lin_prestress</code><br><i>LinearPrestressTerm</i>           | <material>,<br><virtual><br><material>,<br><parameter>                                                                       | $\int_{\Omega} \sigma_{ij} e_{ij}(\underline{v})$                                                                                                                                                                                                        |
| <code>dw_lin_strain_fib</code><br><i>LinearStrainFiberTerm</i>        | <material_1>,<br><material_2>,<br><virtual>                                                                                  | $\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) (d_k d_l)$                                                                                                                                                                                                 |
| <code>dw_non_penetration</code><br><i>NonPenetrationTerm</i>          | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><state>,<br><virtual>                                         | $\int_{\Gamma} c \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} c \hat{\lambda} \underline{n} \cdot \underline{u}$<br>$\int_{\Gamma} \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} \hat{\lambda} \underline{n} \cdot \underline{u}$ |
| <code>dw_non_penetration_p</code><br><i>NonPenetrationPenaltyTerm</i> | <material>,<br><virtual>,<br><state>                                                                                         | $\int_{\Gamma} c(\underline{n} \cdot \underline{v})(\underline{n} \cdot \underline{u})$                                                                                                                                                                  |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                   | arguments                                                                                                                      | definition                                                                                                       |
|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>dw_nonsym_elastic</code><br><i>NonsymElasticTerm</i>   | <material>,<br><virtual>,<br><state><br><material>,<br><parameter_1>,<br><parameter_2>                                         | $\int_{\Omega} \underline{\underline{D}} \nabla \underline{u} : \nabla \underline{v}$                            |
| <code>dw_piezo_coupling</code><br><i>PiezoCouplingTerm</i>   | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual><br><material>,<br><parameter_v>,<br><parameter_s> | $\int_{\Omega} g_{kij} e_{ij}(\underline{v}) \nabla_k p, \int_{\Omega} g_{kij} e_{ij}(\underline{u}) \nabla_k q$ |
| <code>ev_piezo_strain</code><br><i>PiezoStrainTerm</i>       | <material>,<br><parameter>                                                                                                     | $\int_{\Omega} g_{kij} e_{ij}(\underline{u})$                                                                    |
| <code>ev_piezo_stress</code><br><i>PiezoStressTerm</i>       | <material>,<br><parameter>                                                                                                     | $\int_{\Omega} g_{kij} \nabla_k p$                                                                               |
| <code>dw_point_load</code><br><i>ConcentratedPointLoad</i>   | <material>,<br><virtual>                                                                                                       | $\underline{f}^i = \bar{f}^i \quad \forall \text{ FE node } i \text{ in a region}$                               |
| <code>dw_point_lspring</code><br><i>LinearPointSpring</i>    | <material>,<br><virtual>,<br><state>                                                                                           | $\underline{f}^i = -k \underline{u}^i \quad \forall \text{ FE node } i \text{ in a region}$                      |
| <code>dw_s_dot_grad_i_s</code><br><i>ScalarDotGradIScale</i> | <material>,<br><virtual>,<br><state>                                                                                           | $Z^i = \int_{\Omega} q \nabla_i p$                                                                               |
| <code>dw_s_dot_mgrad_s</code><br><i>ScalarDotMGradScale</i>  | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual>                                                   | $\int_{\Omega} q \underline{y} \cdot \nabla p, \int_{\Omega} p \underline{y} \cdot \nabla q$                     |
| <code>dw_shell10x</code><br><i>Shell10XTerm</i>              | <material_d>,<br><material_drill>,<br><virtual>,<br><state>                                                                    | $\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$                                             |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                  | arguments                                                                                                                                  | definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_stokes</code><br><i>StokesTerm</i>                 | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><state>,<br><virtual><br><opt_material>,<br><parameter_v>,<br><parameter_s> | $\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u}$ or $\int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$                                                                                                                                                                                                                                                                                                                                     |
| <code>dw_stokes_wave</code><br><i>StokesWaveTerm</i>        | <material>,<br><virtual>,<br><state>                                                                                                       | $\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\underline{\kappa} \cdot \underline{u})$                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>dw_stokes_wave_div</code><br><i>StokesWaveDivTerm</i> | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual>                                                               | $\int_{\Omega} (\underline{\kappa} \cdot \underline{v})(\nabla \cdot \underline{u}), \int_{\Omega} (\underline{\kappa} \cdot \underline{u})(\nabla \cdot \underline{v})$                                                                                                                                                                                                                                                                                                                                                     |
| <code>d_sum_vals</code><br><i>SumNodalValuesTerm</i>        | <parameter>                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>d_surface</code><br><i>SurfaceTerm</i>                | <parameter>                                                                                                                                | $\int_{\Gamma} 1$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>ev_surface_div</code><br><i>SurfaceDivTerm</i>        | <parameter>                                                                                                                                | $\int_{\Gamma} \nabla \cdot \underline{u}$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla \cdot \underline{u} / \int_{T_K} 1$<br>$(\nabla \cdot \underline{u}) _{qp}$                                                                                                                                                                                                                                                                                                                                         |
| <code>dw_surface_dot</code><br><i>DotProductSurfaceTerm</i> | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><parameter_1>,<br><parameter_2>                                             | $\int_{\Gamma} qp, \int_{\Gamma} \underline{v} \cdot \underline{u}, \int_{\Gamma} \underline{v} \cdot np, \int_{\Gamma} qn \cdot \underline{u}, \int_{\Gamma} pr, \int_{\Gamma} \underline{u} \cdot \underline{w}, \int_{\Gamma} \underline{w} \cdot np$<br>$\int_{\Gamma} cqp, \int_{\Gamma} cv \cdot \underline{u}, \int_{\Gamma} cpr, \int_{\Gamma} cu \cdot \underline{w}$<br>$\int_{\Gamma} \underline{v} \cdot \underline{M} \cdot \underline{u}, \int_{\Gamma} \underline{u} \cdot \underline{M} \cdot \underline{w}$ |
| <code>dw_surface_flux</code><br><i>SurfaceFluxOperator</i>  | <opt_material>,<br><virtual>,<br><state>                                                                                                   | $\int_{\Gamma} qn \cdot \underline{K} \cdot \nabla p$                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                                        | arguments                      | definition                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------------------------------|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| d_surface_flux<br><i>SurfaceFluxTerm</i>                          | <material>,<br><parameter>     | $\int_{\Gamma} \underline{n} \cdot K_{ij} \nabla_j \bar{p}$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p} / \int_{T_K} 1</math></p> <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p}</math></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| ev_surface_grad<br><i>SurfaceGradTerm</i>                         | <parameter>                    | $\int_{\Gamma} \nabla p \text{ or } \int_{\Gamma} \nabla \underline{w}$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla p / \int_{T_K} 1 \text{ or } \int_{T_K} \nabla \underline{w} / \int_{T_K} 1</math></p> <p><math>(\nabla p) _{qp} \text{ or } \nabla \underline{w} _{qp}</math></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| dw_surface_integrate<br><i>IntegrateSurfaceOp</i><br>virtual term | <opt_material>,<br><parameter> | $\int_{\Gamma} q \text{ or } \int_{\Gamma} cq$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| ev_surface_integrate<br><i>IntegrateSurfaceTerm</i>               | <opt_material>,<br><parameter> | $\int_{\Gamma} y, \int_{\Gamma} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n}$ $\int_{\Gamma} cy, \int_{\Gamma} c\underline{y}, \int_{\Gamma} c\underline{y} \cdot \underline{n}$ <p>flux</p> <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1, \int_{T_K} (\underline{y} \cdot \underline{n}) / \int_{T_K} 1</math></p> <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} c\underline{y} / \int_{T_K} 1, \int_{T_K} (c\underline{y} \cdot \underline{n}) / \int_{T_K} 1</math></p> <p><math>y _{qp}, \underline{y} _{qp}, (\underline{y} \cdot \underline{n}) _{qp}</math> flux</p> <p><math>cy _{qp}, c\underline{y} _{qp}, (c\underline{y} \cdot \underline{n}) _{qp}</math> flux</p> |
| ev_surface_integrate_mat<br><i>IntegrateSurfaceMatTerm</i>        | <material>,<br><parameter>     | $\int_{\Gamma} m$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} m / \int_{T_K} 1</math></p> <p><math>m _{qp}</math></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                        | arguments                                                                                                                                  | definition                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| dw_surface_ltr<br><i>LinearTractionTerm</i>       | <opt_material>,<br><virtual><br><opt_material>,<br><parameter>                                                                             | $\int_{\Gamma} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \int_{\Gamma} \underline{v} \cdot \underline{n},$                                                                                                                                                                                                                                                                                                                   |
| d_surface_moment<br><i>SurfaceMomentTerm</i>      | <material>,<br><parameter>                                                                                                                 | $\int_{\Gamma} \underline{n}(\underline{x} - \underline{x}_0)$                                                                                                                                                                                                                                                                                                                                                                                             |
| dw_surface_ndot<br><i>SurfaceNormalDotTerm</i>    | <material>,<br><virtual><br><material>,<br><parameter>                                                                                     | $\int_{\Gamma} q\underline{c} \cdot \underline{n}$                                                                                                                                                                                                                                                                                                                                                                                                         |
| dw_v_dot_grad_s<br><i>VectorDotGradScalarTerm</i> | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><state>,<br><virtual><br><opt_material>,<br><parameter_v>,<br><parameter_s> | $\int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q$<br>$\int_{\Omega} c\underline{v} \cdot \nabla p, \int_{\Omega} c\underline{u} \cdot \nabla q$<br>$\int_{\Omega} \underline{v} \cdot (\underline{\underline{M}} \nabla p), \int_{\Omega} \underline{u} \cdot (\underline{\underline{M}} \nabla q)$                                                                                                                 |
| dw_vm_dot_s<br><i>VectorDotScalarTerm</i>         | <material>,<br><virtual>,<br><state><br><material>,<br><state>,<br><virtual><br><material>,<br><parameter_v>,<br><parameter_s>             | $\int_{\Omega} \underline{v} \cdot \underline{m} p, \int_{\Omega} \underline{u} \cdot \underline{m} q$                                                                                                                                                                                                                                                                                                                                                     |
| d_volume<br><i>VolumeTerm</i>                     | <parameter>                                                                                                                                | $\int_{\Omega} 1$                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| dw_volume_dot<br><i>DotProductVolumeTerm</i>      | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><parameter_1>,<br><parameter_2>                                             | $\int_{\Omega} q p, \int_{\Omega} \underline{v} \cdot \underline{u}, \int_{\Omega} p r, \int_{\Omega} \underline{u} \cdot \underline{w}$<br>$\int_{\Omega} c q p, \int_{\Omega} c \underline{v} \cdot \underline{u}, \int_{\Omega} c p r, \int_{\Omega} c \underline{u} \cdot \underline{w}$<br>$\int_{\Omega} \underline{v} \cdot (\underline{\underline{M}} \underline{u}), \int_{\Omega} \underline{u} \cdot (\underline{\underline{M}} \underline{w})$ |

Continued on next page

Table 7.5 – continued from previous page

| name/class                                           | arguments                      | definition                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------------------------------|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ev_volume_integrate<br><i>IntegrateVolumeTerm</i>    | <opt_material>,<br><parameter> | $\int_{\Omega} y, \int_{\Omega} \underline{y}$<br>$\int_{\Omega} cy, \int_{\Omega} c\underline{y}$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} c\underline{y} / \int_{T_K} 1$<br>$y _{qp}, \underline{y} _{qp}$<br>$cy _{qp}, c\underline{y} _{qp}$ |
| dw_volume_integrate<br><i>IntegrateVolumeOp</i>      | <opt_material>,<br><virtual>   | $\int_{\Omega} q \text{ or } \int_{\Omega} cq$                                                                                                                                                                                                                                                                                                                                                                 |
| ev_volume_integrate_mat<br><i>IntegrateVolumeMat</i> | <material>,<br><parameter>     | $\int_{\Omega} m$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} m / \int_{T_K} 1$<br>$m _{qp}$                                                                                                                                                                                                                                                                                                        |
| dw_volume_lvf<br><i>LinearVolumeForce</i>            | <material>,<br><virtual>       | $\int_{\Omega} f \cdot \underline{v} \text{ or } \int_{\Omega} fq$                                                                                                                                                                                                                                                                                                                                             |
| d_volume_surface<br><i>VolumeSurfaceTerm</i>         | <parameter>                    | $1/D \int_{\Gamma} \underline{x} \cdot \underline{n}$                                                                                                                                                                                                                                                                                                                                                          |
| dw_zero<br><i>ZeroTerm</i>                           | <virtual>,<br><state>          | 0                                                                                                                                                                                                                                                                                                                                                                                                              |

## Table of sensitivity terms

Table 7.6: Sensitivity terms

| name/class                                                           | arguments                                                                                       | definition                                                                                                                                                                                                                                                                                                                                                 |
|----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_adj_convect1</code><br><i>AdjConvect1Term</i>               | <virtual>,<br><state>,<br><parameter>                                                           | $\int_{\Omega} ((\underline{v} \cdot \nabla) \underline{u}) \cdot \underline{w}$                                                                                                                                                                                                                                                                           |
| <code>dw_adj_convect2</code><br><i>AdjConvect2Term</i>               | <virtual>,<br><state>,<br><parameter>                                                           | $\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{v}) \cdot \underline{w}$                                                                                                                                                                                                                                                                           |
| <code>dw_adj_div_grad</code><br><i>AdjDivGradTerm</i>                | <material_1>,<br><material_2>,<br><virtual>,<br><parameter>                                     | $w \delta_u \Psi(\underline{u}) \circ \underline{v}$                                                                                                                                                                                                                                                                                                       |
| <code>d_sd_convect</code><br><i>SDConvectTerm</i>                    | <parameter_u>,<br><parameter_w>,<br><parameter_mesh_velocity>                                   | $\int_{\Omega} [u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \mathcal{V}) - u_k \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i]$                                                                                                                                                                             |
| <code>d_sd_diffusion</code><br><i>SDDiffusionTerm</i>                | <material>,<br><parameter_q>,<br><parameter_p>,<br><parameter_mesh_velocity>                    | $\int_{\Omega} [(\operatorname{div} \mathcal{V}) K_{ij} \nabla_i q \nabla_j p - K_{ij} (\nabla_j \mathcal{V} \nabla q) \nabla_i p - K_{ij} \nabla_j q (\nabla_i \mathcal{V} \nabla p)]$                                                                                                                                                                    |
| <code>d_sd_div</code><br><i>SDDivTerm</i>                            | <parameter_u>,<br><parameter_p>,<br><parameter_mesh_velocity>                                   | $\int_{\Omega} p [(\nabla \cdot \underline{w}) (\nabla \cdot \mathcal{V}) - \frac{\partial \mathcal{V}_k}{\partial x_i} \frac{\partial w_i}{\partial x_k}]$                                                                                                                                                                                                |
| <code>d_sd_div_grad</code><br><i>SDDivGradTerm</i>                   | <material_1>,<br><material_2>,<br><parameter_u>,<br><parameter_w>,<br><parameter_mesh_velocity> | $w \nu \int_{\Omega} [\frac{\partial u_i}{\partial x_k} \frac{\partial w_i}{\partial x_k} (\nabla \cdot \mathcal{V}) - \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} \frac{\partial w_i}{\partial x_k} - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_l}{\partial x_k} \frac{\partial w_i}{\partial x_l}]$ |
| <code>d_sd_lin_elastic</code><br><i>SDLinearElasticTerm</i>          | <material>,<br><parameter_w>,<br><parameter_u>,<br><parameter_mesh_velocity>                    | $\int_{\Omega} \hat{D}_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$<br>$\hat{D}_{ijkl} = D_{ijkl} (\nabla \cdot \mathcal{V}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$                                                                                                         |
| <code>d_sd_surface_integrate</code><br><i>SDSurfaceIntegrateTerm</i> | <parameter>,<br><parameter_mesh_velocity>                                                       | $\int_{\Gamma} p \nabla \cdot \mathcal{V}$                                                                                                                                                                                                                                                                                                                 |

Continued on next page

Table 7.6 – continued from previous page

| name/class                                | arguments                                                     | definition                                                                                                                                         |
|-------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| d_sd_volume_dot<br><i>SDDotVolumeTerm</i> | <parameter_1>,<br><parameter_2>,<br><parameter_mesh_velocity> | $\int_{\Omega} pq(\nabla \cdot \underline{\mathcal{V}}) , \int_{\Omega} (\underline{u} \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}})$ |

## Table of large deformation terms

Table 7.7: Large deformation terms

| name/class                                                       | arguments                                                                                                                                       | definition                                                                                                                                             |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_tl_bulk_active</code><br><i>BulkActiveTLTerm</i>        | <code>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                                                                   | $\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$                                                                      |
| <code>dw_tl_bulk_penalty</code><br><i>BulkPenaltyTLTerm</i>      | <code>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                                                                   | $\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$                                                                      |
| <code>dw_tl_bulk_pressure</code><br><i>BulkPressureTLTerm</i>    | <code>&lt;virtual&gt;, &lt;state&gt;, &lt;state_p&gt;</code>                                                                                    | $\int_{\Omega} S_{ij}(p) \delta E_{ij}(\underline{u}; \underline{v})$                                                                                  |
| <code>dw_tl_diffusion</code><br><i>DiffusionTLTerm</i>           | <code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;virtual&gt;, &lt;state&gt;, &lt;parameter&gt;</code>                                          | $\int_{\Omega} \underline{\underline{K}}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial \underline{X}} \frac{\partial p}{\partial \underline{X}}$ |
| <code>dw_tl_fib_a</code><br><i>FibresActiveTLTerm</i>            | <code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;material_3&gt;, &lt;material_4&gt;, &lt;material_5&gt;, &lt;virtual&gt;, &lt;state&gt;</code> | $\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$                                                                      |
| <code>dw_tl_he_mooney_rivlin</code><br><i>MooneyRivlinTLTerm</i> | <code>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                                                                   | $\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$                                                                      |
| <code>dw_tl_he_neohook</code><br><i>NeoHookeanTLTerm</i>         | <code>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                                                                   | $\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$                                                                      |
| <code>dw_tl_membrane</code><br><i>TLMembraneTerm</i>             | <code>&lt;material_a1&gt;, &lt;material_a2&gt;, &lt;material_h0&gt;, &lt;virtual&gt;, &lt;state&gt;</code>                                      |                                                                                                                                                        |
| <code>d_tl_surface_flux</code><br><i>SurfaceFluxTLTerm</i>       | <code>&lt;material_1&gt;, &lt;material_2&gt;, &lt;parameter_1&gt;, &lt;parameter_2&gt;</code>                                                   | $\int_{\Gamma} \nu \cdot \underline{\underline{K}}(\underline{u}^{(n-1)}) \frac{\partial p}{\partial \underline{X}}$                                   |

Continued on next page

Table 7.7 – continued from previous page

| name/class                                                                  | arguments                                              | definition                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------|--------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_tl_surface_traction</code><br><i>SurfaceTrac<sub>T</sub>TLTerm</i> | <opt_material>,<br><virtual>,<br><state>               | $\int_{\Gamma} \underline{\nu} \cdot \underline{\underline{F}}^{-1} \cdot \underline{\underline{\sigma}} \cdot \underline{\underline{\nu}} J$                                                                                    |
| <code>dw_tl_volume</code><br><i>VolumeTLTerm</i>                            | <virtual>,<br><state>                                  | $\int_{\Omega} qJ(\underline{u})$<br>volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$<br>rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$ |
| <code>d_tl_volume_surface</code><br><i>VolumeSurfaceTLTerm</i>              | <parameter>                                            | $1/D \int_{\Gamma} \underline{\nu} \cdot \underline{\underline{F}}^{-1} \cdot \underline{x} J$                                                                                                                                   |
| <code>dw_ul_bulk_penalty</code><br><i>BulkPenaltyULTerm</i>                 | <material>,<br><virtual>,<br><state>                   | $\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$                                                                                                                                                 |
| <code>dw_ul_bulk_pressure</code><br><i>BulkPressureULTerm</i>               | <virtual>,<br><state>,<br><state_p>                    | $\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$                                                                                                                                                 |
| <code>dw_ul_compressible</code><br><i>CompressibilityULTerm</i>             | <material>,<br><virtual>,<br><state>,<br><parameter_u> | $\frac{\int_{\Omega} 1}{\gamma p q}$                                                                                                                                                                                             |
| <code>dw_ul_he_mooney_rivlin</code><br><i>MooneyRivlinULTerm</i>            | <material>,<br><virtual>,<br><state>                   | $\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$                                                                                                                                                 |
| <code>dw_ul_he_neohook</code><br><i>NeoHookeanULTerm</i>                    | <material>,<br><virtual>,<br><state>                   | $\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u})e_{ij}(\delta\underline{v})/J$                                                                                                                                                 |
| <code>dw_ul_volume</code><br><i>VolumeULTerm</i>                            | <virtual>,<br><state>                                  | $\int_{\Omega} qJ(\underline{u})$<br>volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$<br>rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$ |

## Table of special terms

Table 7.8: Special terms

| name/class                                                      | arguments                                                                                                                            | definition                                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_biot_eth</code><br><i>BiotETHTerm</i>                  | <ts>,<br><material_0>,<br><material_1>,<br><virtual>,<br><state><br><ts>,<br><material_0>,<br><material_1>,<br><state>,<br><virtual> | $\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) p(\tau) d\tau \right] e_{ij}(\underline{v}) ,$<br>$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$                                                                                                                |
| <code>dw_biot_th</code><br><i>BiotTHTerm</i>                    | <ts>,<br><material>,<br><virtual>,<br><state><br><ts>,<br><material>,<br><state>,<br><virtual>                                       | $\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) p(\tau) d\tau \right] e_{ij}(\underline{v}) ,$<br>$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$                                                                                                                |
| <code>ev_cauchy_stress_eth</code><br><i>CauchyStressETHTerm</i> | <ts>,<br><material_0>,<br><material_1>,<br><parameter>                                                                               | $\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau / \int_{T_K} 1$<br>$\int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau _{qp}$ |
| <code>ev_cauchy_stress_th</code><br><i>CauchyStressTHTerm</i>   | <ts>,<br><material>,<br><parameter>                                                                                                  | $\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau$<br>vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau / \int_{T_K} 1$<br>$\int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{w}(\tau)) d\tau _{qp}$ |
| <code>dw_lin_elastic_eth</code><br><i>LinearElasticETHTerm</i>  | <ts>,<br><material_0>,<br><material_1>,<br><virtual>,<br><state>                                                                     | $\int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$                                                                                                                                                                                          |

Continued on next page

Table 7.8 – continued from previous page

| name/class                                                        | arguments                                                                                      | definition                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------------------------------|------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_lin_elastic_th</code><br><i>LinearElasticTHTer</i>       | <ts>,<br><material>,<br><virtual>,<br><state>                                                  | $\int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t-\tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$                                                                                                                                                                                                                                                    |
| <code>d_of_ns_surf_min_d_press</code><br><i>NSOSurfMinDPress</i>  | <material_1>,<br><material_2>,<br><parameter>                                                  | $\delta\Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} bpress \right)$                                                                                                                                                                                                                                                                                     |
| <code>dw_of_ns_surf_min_d_press</code><br><i>NSOSurfMinDPress</i> | <material>,<br><virtual>                                                                       | $w\delta_p\Psi(p) \circ q$                                                                                                                                                                                                                                                                                                                                                    |
| <code>d_sd_st_grad_div</code><br><i>SDGradDivStabilizat</i>       | <material>,<br><parameter_u>,<br><parameter_w>,<br><parameter_mesh_velocity>                   | $\gamma \int_{\Omega} [(\nabla \cdot \underline{u})(\nabla \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot \underline{w}) - (\nabla \cdot \underline{u}) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i}]$                             |
| <code>d_sd_st_pspg_c</code><br><i>SDPSPGCstabilizat</i>           | <material>,<br><parameter_b>,<br><parameter_u>,<br><parameter_r>,<br><parameter_mesh_velocity> | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [\frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i)(\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k) \frac{\partial u_i}{\partial x_k}]$   |
| <code>d_sd_st_pspg_p</code><br><i>SDPSPGPstabilizat</i>           | <material>,<br><parameter_r>,<br><parameter_p>,<br><parameter_mesh_velocity>                   | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K [(\nabla r \cdot \nabla p)(\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} (\nabla \mathcal{V}_k \cdot \nabla p) - (\nabla r \cdot \nabla \mathcal{V}_k) \frac{\partial p}{\partial x_k}]$                                                                                                                          |
| <code>d_sd_st_supg_c</code><br><i>SDSUPGCstabilizat</i>           | <material>,<br><parameter_b>,<br><parameter_u>,<br><parameter_w>,<br><parameter_mesh_velocity> | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [(\underline{b} \cdot \nabla u_k)(\underline{b} \cdot \nabla w_k)(\nabla \cdot \mathcal{V}) - (\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial u_k}{\partial x_i} (\underline{b} \cdot \nabla w_k) - (\underline{u} \cdot \nabla u_k) \frac{\partial w_k}{\partial x_i} (\underline{b} \cdot \nabla \mathcal{V}_i)]$ |
| <code>dw_st_adj1_supg_p</code><br><i>SUPGPAdj1Stabilizat</i>      | <virtual>,<br><state>,<br><parameter>                                                          | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p(\underline{v} \cdot \nabla \underline{w})$                                                                                                                                                                                                                                                                           |
| <code>dw_st_adj2_supg_p</code><br><i>SUPGPAdj2Stabilizat</i>      | <material>,<br><virtual>,<br><parameter>,<br><state>                                           | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla r(\underline{v} \cdot \nabla \underline{u})$                                                                                                                                                                                                                                                                             |

Continued on next page

Table 7.8 – continued from previous page

| name/class                                                                   | arguments                                                                                      | definition                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>dw_st_adj_supg_c</code><br><i>SUPGAdjStabilization</i>                 | <material>,<br><virtual>,<br><parameter>,<br><state>                                           | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [((\underline{v} \cdot \nabla) \underline{u}) ((\underline{u} \cdot \nabla) \underline{w}) + ((\underline{u} \cdot \nabla) \underline{u}) ((\underline{v} \cdot \nabla) \underline{w})]$ |
| <code>dw_st_grad_div</code><br><i>GradDivStabilization</i>                   | <material>,<br><virtual>,<br><state>                                                           | $\gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v})$                                                                                                                                                   |
| <code>dw_st_pspg_c</code><br><i>PSPGCStabilization</i>                       | <material>,<br><virtual>,<br><parameter>,<br><state>                                           | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$                                                                                                                               |
| <code>dw_st_pspg_p</code><br><i>PSPGPStabilization</i>                       | <opt_material>,<br><virtual>,<br><state><br><opt_material>,<br><parameter_1>,<br><parameter_2> | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q$                                                                                                                                                                   |
| <code>dw_st_supg_c</code><br><i>SUPGStabilization</i>                        | <material>,<br><virtual>,<br><parameter>,<br><state>                                           | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v})$                                                                                         |
| <code>dw_st_supg_p</code><br><i>SUPGPStabilization</i>                       | <material>,<br><virtual>,<br><parameter>,<br><state>                                           | $\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v})$                                                                                                                             |
| <code>dw_volume_dot_w_scalar</code><br><i>DotSProductVolumeMaterialTerm</i>  | <ths>,<br><material>,<br><virtual>,<br><state>                                                 | $\int_{\Omega} \left[ \int_0^t \mathcal{G}(t - \tau) p(\tau) d\tau \right] q$                                                                                                                                                            |
| <code>dw_volume_dot_w_scalar</code><br><i>DotSProductVolumeMaterialHTerm</i> | <ths>,<br><material>,<br><virtual>,<br><state>                                                 | $\int_{\Omega} \left[ \int_0^t \mathcal{G}(t - \tau) p(\tau) d\tau \right] q$                                                                                                                                                            |

## 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_shapes = {}
    arg_types = 0

    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.

    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 ()
get_geometry_types ()
```

### Returns

**out** [dict] The required geometry types for each variable argument.

**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** ()

**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 = ‘volume’**

**name = ”**

**static new** (*name*, *integral*, *region*, *\*\*kwargs*)

**set\_arg\_types** ()

**set\_integral** (*integral*)

Set the term integral.

**setup** ()

**setup\_args** (*\*\*kwargs*)

**setup\_formal\_args** ()

```

setup_integration()
standalone_setup()
time_update(ts)

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()
    update_expression()
sfepy.terms.terms.create_arg_parser()
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’ - imag(arg1), imag(arg2)
  - ‘ri’ - real(arg1), imag(arg2)
  - ‘ir’ - imag(arg1), real(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**

|                        |                             |
|------------------------|-----------------------------|
| <b>dw_adj_convect1</b> | (virtual, state, parameter) |
|------------------------|-----------------------------|

**Arguments**

- virtual :  $\underline{v}$
- state :  $\underline{w}$
- parameter :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'parameter': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('virtual', 'state', 'parameter')
```

```
static function()
```

```
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**

|                        |                             |
|------------------------|-----------------------------|
| <b>dw_adj_convect2</b> | (virtual, state, parameter) |
|------------------------|-----------------------------|

**Arguments**

- virtual :  $\underline{v}$
- state :  $\underline{w}$
- parameter :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'parameter': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('virtual', 'state', 'parameter')
```

```
static function()
```

```
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**

|                        |                                              |
|------------------------|----------------------------------------------|
| <b>dw_adj_div_grad</b> | (material_1, material_2, virtual, parameter) |
|------------------------|----------------------------------------------|

**Arguments**

- material\_1 :  $w$  (weight)
- material\_2 :  $\nu$  (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()
```

```
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**

|                         |                                     |
|-------------------------|-------------------------------------|
| <b>d_of_ns_min_grad</b> | (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()
```

```
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**

|                                       |                     |
|---------------------------------------|---------------------|
| <b>dw_of_ns_surf_min_d_press_diff</b> | (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.NSOSurfMinDPressTerm(name, arg_str,
   integral, region,
   **kwargs)
```

Sensitivity of  $\Psi(p)$ .

### Definition

$$\delta\Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} bpress \right)$$

### Call signature

|                                 |                                     |
|---------------------------------|-------------------------------------|
| <b>d_of_ns_surf_min_d_press</b> | (material_1, material_2, parameter) |
|---------------------------------|-------------------------------------|

### Arguments

- material\_1 :  $w$  (weight)
- material\_2 :  $bpress$  (given pressure)
- parameter :  $p$

```
arg_shapes = {'material_1': 1, 'material_2': 1, 'parameter': 1}
arg_types = ('material_1', 'material_2', 'parameter')
static function()

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 = 'surface'
name = 'd_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

|                     |                                                     |
|---------------------|-----------------------------------------------------|
| <b>d_sd_convect</b> | (parameter_u, parameter_w, parameter_mesh_velocity) |
|---------------------|-----------------------------------------------------|

### Arguments

- parameter\_u :  $u$
- parameter\_w :  $w$
- parameter\_mesh\_velocity :  $\mathcal{V}$

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter_w': 'D', 'parameter_u': 'D'}
arg_types = ('parameter_u', 'parameter_w', 'parameter_mesh_velocity')
static function()
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 = 'd_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

$$w\nu \int_{\Omega} \left[ \frac{\partial u_i}{\partial x_k} \frac{\partial w_i}{\partial x_k} (\nabla \cdot \mathcal{V}) - \frac{\partial V_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} \frac{\partial w_i}{\partial x_k} - \frac{\partial u_i}{\partial x_k} \frac{\partial V_l}{\partial x_k} \frac{\partial w_i}{\partial x_k} \right]$$

### Call signature

|                                                                         |                          |
|-------------------------------------------------------------------------|--------------------------|
| <b>d_sd_div_grad</b> (material_1, material_2, parameter_u, parameter_w, | parameter_mesh_velocity) |
|-------------------------------------------------------------------------|--------------------------|

### Arguments

- material\_1 :  $w$  (weight)
- material\_2 :  $\nu$  (viscosity)
- parameter\_u :  $u$
- parameter\_w :  $w$
- parameter\_mesh\_velocity :  $\mathcal{V}$

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'material_1': '1, 1', 'material_2': '1, 1', 'parameter_w': 'D', 'parameter_u': 'D'}
```

```
arg_types = ('material_1', 'material_2', 'parameter_u', 'parameter_w', 'parameter_mesh_velocity')
```

static function()

```
get_eval_shape (mat1, mat2, par_u, par_w, par_mv, mode=None, term_mode=None,
               diff_var=None, **kwargs)
```

```
get_fargs (mat1, mat2, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None,
           **kwargs)
```

```
name = 'd_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

|                 |                                                     |
|-----------------|-----------------------------------------------------|
| <b>d_sd_div</b> | (parameter_u, parameter_p, parameter_mesh_velocity) |
|-----------------|-----------------------------------------------------|

#### Arguments

- parameter\_u :  $\underline{u}$
- parameter\_p :  $p$
- parameter\_mesh\_velocity :  $\underline{\mathcal{V}}$

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter_p': 1, 'parameter_u': 'D'}
```

```
arg_types = ('parameter_u', 'parameter_p', 'parameter_mesh_velocity')
```

```
static function()
```

```
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 = 'd_sd_div'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDDotVolumeTerm(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

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <b>d_sd_volume_dot</b> | (parameter_1, parameter_2, parameter_mesh_velocity) |
|------------------------|-----------------------------------------------------|

#### Arguments

- parameter\_1 :  $p$  or  $\underline{u}$
- parameter\_2 :  $q$  or  $\underline{w}$
- parameter\_mesh\_velocity :  $\underline{\mathcal{V}}$

```
arg_shapes = [{‘parameter_mesh_velocity’: ‘D’, ‘parameter_2’: ‘D’, ‘parameter_1’: ‘D’}, {‘parameter_2’: 1, ‘parameter_1’: ‘D’}]
```

```
arg_types = ('parameter_1', 'parameter_2', 'parameter_mesh_velocity')
```

```
static function()
```

```
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 = 'd_sd_volume_dot'
class sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm(name,
   arg_str,
   integral,
   region,
   **kwargs)
```

Sensitivity (shape derivative) of stabilization term  $d_{w\_st\_grad\_div}$ .

#### 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

|                         |                                                               |
|-------------------------|---------------------------------------------------------------|
| <b>d_sd_st_grad_div</b> | (material, parameter_u, parameter_w, parameter_mesh_velocity) |
|-------------------------|---------------------------------------------------------------|

#### Arguments

- material :  $\gamma$
- parameter\_u :  $\underline{u}$
- parameter\_w :  $\underline{w}$
- parameter\_mesh\_velocity :  $\underline{\mathcal{V}}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'material': '1, 1', 'parameter_w': 'D', 'parameter_u': 'D'}
arg_types = ('material', 'parameter_u', 'parameter_w', 'parameter_mesh_velocity')
static function ()
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 = 'd_sd_st_grad_div'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm(name,
   arg_str, inte-
   gral, region,
   **kwargs)
```

Sensitivity (shape derivative) of stabilization terms  $d_{w\_st\_supg\_p}$  or  $d_{w\_st\_pspg\_c}$ .

#### Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [\frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i) (\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k) \frac{\partial u_i}{\partial x_k}]$$

#### Call signature

|                       |                                                                                            |
|-----------------------|--------------------------------------------------------------------------------------------|
| <b>d_sd_st_pspg_c</b> | (material, parameter_b, parameter_u, parameter_r,                 parameter_mesh_velocity) |
|-----------------------|--------------------------------------------------------------------------------------------|

#### Arguments

- material :  $\delta_K$
- parameter\_b :  $\underline{b}$

- parameter\_u :  $u$
- parameter\_r :  $r$
- parameter\_mesh\_velocity :  $\mathcal{V}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter_b': 'D', 'material': '1, 1', 'parameter_u': 'D', 'parameter_r': 'D'}
arg_types = ('material', 'parameter_b', 'parameter_u', 'parameter_r', 'parameter_mesh_velocity')
static function()
    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 = 'd_sd_st_pspg_c'

class sfepy.terms.terms_adj_navier_stokes.SDPSPGPStabilizationTerm(name,
   arg_str, integral, region,
   **kwargs)
```

Sensitivity (shape derivative) of stabilization term  $d_{w_{st}} p_{spg\_p}$ .

#### Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K [(\nabla r \cdot \nabla p)(\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} (\nabla \mathcal{V}_k \cdot \nabla p) - (\nabla r \cdot \nabla \mathcal{V}_k) \frac{\partial p}{\partial x_k}]$$

#### Call signature

|                             |                                                               |
|-----------------------------|---------------------------------------------------------------|
| <code>d_sd_st_pspg_p</code> | (material, parameter_r, parameter_p, parameter_mesh_velocity) |
|-----------------------------|---------------------------------------------------------------|

#### Arguments

- material :  $\tau_K$
- parameter\_r :  $r$
- parameter\_p :  $p$
- parameter\_mesh\_velocity :  $\mathcal{V}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter_r': 1, 'material': '1, 1', 'parameter_p': 1}
arg_types = ('material', 'parameter_r', 'parameter_p', 'parameter_mesh_velocity')
static function()
    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 = 'd_sd_st_pspg_p'
```

```
class sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm(name,
    arg_str, integral,
    region,
    **kwargs)
```

Sensitivity (shape derivative) of stabilization term  $d_{w_{st\_supg\_c}}$ .

#### Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [(\underline{b} \cdot \nabla u_k)(\underline{b} \cdot \nabla w_k)(\nabla \cdot \mathcal{V}) - (\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial u_k}{\partial x_i} (\underline{b} \cdot \nabla w_k) - (\underline{u} \cdot \nabla u_k)(\underline{b} \cdot \nabla \mathcal{V}_i) \frac{\partial w_k}{\partial x_i}]$$

#### Call signature

|                       |                                                                               |
|-----------------------|-------------------------------------------------------------------------------|
| <b>d_sd_st_supg_c</b> | (material, parameter_b, parameter_u, parameter_w,<br>parameter_mesh_velocity) |
|-----------------------|-------------------------------------------------------------------------------|

#### Arguments

- material :  $\delta_K$
- parameter\_b :  $\underline{b}$
- parameter\_u :  $\underline{u}$
- parameter\_w :  $\underline{w}$
- parameter\_mesh\_velocity :  $\mathcal{V}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter_b': 'D', 'material': '1, 1', 'parameter_w': 'D', 'parameter_u': 'D'}
```

```
arg_types = ('material', 'parameter_b', 'parameter_u', 'parameter_w', 'parameter_mesh_velocity')
```

```
static function ()
```

```
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 = 'd_sd_st_supg_c'
```

```
class sfepy.terms.terms_adj_navier_stokes.SUPGCAadjStabilizationTerm(name,
    arg_str,
    integral,
    region,
    **kwargs)
```

Adjoint term to SUPG stabilization term  $d_{w_{st\_supg\_c}}$ .

#### 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

|                         |                                       |
|-------------------------|---------------------------------------|
| <b>dw_st_adj_supg_c</b> | (material, virtual, parameter, state) |
|-------------------------|---------------------------------------|

#### Arguments

- material :  $\delta_K$

- virtual :  $\underline{v}$
- state :  $\underline{w}$
- parameter :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'parameter': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
static function ()
```

```
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.SUPGPAdj1StabilizationTerm(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

|                                |                                       |
|--------------------------------|---------------------------------------|
| <code>dw_st_adj1_supg_p</code> | (material, virtual, state, parameter) |
|--------------------------------|---------------------------------------|

#### Arguments

- material :  $\delta_K$
- virtual :  $\underline{v}$
- state :  $\underline{w}$
- parameter :  $p$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'parameter': 1, 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state', 'parameter')
```

```
static function ()
```

```
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.SUPGPAdj2StabilizationTerm(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

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla r (\underline{v} \cdot \nabla \underline{u})$$

**Call signature**

|                          |                                       |
|--------------------------|---------------------------------------|
| <b>dw_st_adj2_supg_p</b> | (material, virtual, parameter, state) |
|--------------------------|---------------------------------------|

**Arguments**

- material :  $\tau_K$
- virtual :  $\underline{v}$
- parameter :  $\underline{u}$
- state :  $r$

```
arg_shapes = {'state': 1, 'material': '1, 1', 'parameter': 'D', 'virtual': ('D', 'state')}

arg_types = ('material', 'virtual', 'parameter', 'state')

static function()

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.IntegrateSurfaceMatTerm (name, arg_str, integral, region,
**kwargs)
```

Evaluate material parameter  $m$  in a surface region.

Depending on evaluation mode, integrate a material parameter over a surface region ('eval'), average it in faces ('el\_avg') or interpolate it into surface quadrature points ('qp').

Uses reference mapping of  $y$  variable.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

**Definition**

$$\int_{\Gamma} m$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} m / \int_{T_K} 1$

$$m|_{qp}$$

**Call signature**

|                                 |                       |
|---------------------------------|-----------------------|
| <b>ev_surface_integrate_mat</b> | (material, parameter) |
|---------------------------------|-----------------------|

**Arguments**

- material :  $m$  (can have up to two dimensions)
- parameter :  $y$

```
arg_shapes = [{‘material’: ‘N, N’, ‘parameter’: ‘N’}]
```

```
arg_types = ('material', 'parameter')
```

```
integration = 'surface'
name = 'ev_surface_integrate_mat'

class sfepy.terms.terms_basic.IntegrateSurfaceOperatorTerm(name, arg_str, integral,
   region, **kwargs)
    Surface integral of a test function weighted by a scalar function  $c$ .
```

**Definition**

$$\int_{\Gamma} q \text{ or } \int_{\Gamma} cq$$

**Call signature**

|                             |                         |
|-----------------------------|-------------------------|
| <b>dw_surface_integrate</b> | (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’)
```

```
integration = ‘surface’
```

```
name = ‘dw_surface_integrate’
```

```
class sfepy.terms.terms_basic.IntegrateSurfaceTerm(name, arg_str, integral, region,
   **kwargs)
```

Evaluate (weighted) variable in a surface region.

Depending on evaluation mode, integrate a variable over a surface region ('eval'), average it in element faces ('el\_avg') or interpolate it into surface quadrature points ('qp'). For 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**

$$\begin{aligned} & \int_{\Gamma} y, \int_{\Gamma} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n} \\ & \int_{\Gamma} cy, \int_{\Gamma} c\underline{y}, \int_{\Gamma} c\underline{y} \cdot \underline{n} \text{ flux} \\ & \text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1, \int_{T_K} (\underline{y} \cdot \underline{n}) / \int_{T_K} 1 \\ & \text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} c\underline{y} / \int_{T_K} 1, \int_{T_K} (c\underline{y} \cdot \underline{n}) / \int_{T_K} 1 \\ & \quad y|_{qp}, \underline{y}|_{qp}, (\underline{y} \cdot \underline{n})|_{qp} \text{ flux} \\ & \quad cy|_{qp}, c\underline{y}|_{qp}, (c\underline{y} \cdot \underline{n})|_{qp} \text{ flux} \end{aligned}$$

**Call signature**

|                             |                           |
|-----------------------------|---------------------------|
| <b>ev_surface_integrate</b> | (opt_material, parameter) |
|-----------------------------|---------------------------|

**Arguments**

- material :  $c$  (optional)

- parameter :  $y$  or  $\underline{y}$

```
arg_shapes = [{‘opt_material’: ‘1, 1’, ‘parameter’: ‘N’}, {‘opt_material’: None}]

arg_types = (‘opt_material’, ‘parameter’)

static function (out, val_qp, sg, 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 = ‘surface’
    name = ‘ev_surface_integrate’

class sfepy.terms.terms_basic.IntegrateVolumeMatTerm (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

$$\int_{\Omega} m$$

vector for  $K \leftarrow \mathcal{I}_h$  :  $\int_{T_K} m / \int_{T_K} 1$

$$m|_{qp}$$

#### Call signature

|                                |                       |
|--------------------------------|-----------------------|
| <b>ev_volume_integrate_mat</b> | (material, parameter) |
|--------------------------------|-----------------------|

#### Arguments

- material :  $m$  (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)
    name = ‘ev_volume_integrate_mat’
```

```
class sfepy.terms.terms_basic.IntegrateVolumeOperatorTerm (name, arg_str, integral, region, **kwargs)
```

Volume integral of a test function weighted by a scalar function  $c$ .

#### Definition

$$\int_{\Omega} q \text{ or } \int_{\Omega} cq$$

### Call signature

|                            |                         |
|----------------------------|-------------------------|
| <b>dw_volume_integrate</b> | (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)  
name = ‘dw_volume_integrate’  
class sfepy.terms.terms_basic.IntegrateVolumeTerm(name, arg_str, integral, region,  
**kwargs)
```

Evaluate (weighted) variable in a volume region.

Depending on evaluation mode, integrate a variable over a volume region (‘eval’), average it in elements (‘el\_avg’) or interpolate it into volume quadrature points (‘qp’).

Supports ‘eval’, ‘el\_avg’ and ‘qp’ evaluation modes.

### Definition

$$\begin{aligned} & \int_{\Omega} y, \int_{\Omega} \underline{y} \\ & \int_{\Omega} cy, \int_{\Omega} \underline{cy} \\ & \text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1 \\ & \text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} \underline{cy} / \int_{T_K} 1 \\ & y|_{qp}, \underline{y}|_{qp} \\ & cy|_{qp}, \underline{cy}|_{qp} \end{aligned}$$

### Call signature

|                            |                           |
|----------------------------|---------------------------|
| <b>ev_volume_integrate</b> | (opt_material, parameter) |
|----------------------------|---------------------------|

### Arguments

- material :  $c$  (optional)
- parameter :  $y$  or  $\underline{y}$

```
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)
name = 'ev_volume_integrate'

class sfepy.terms.terms_basic.SumNodalValuesTerm(name, arg_str, integral, region,
  **kwargs)
Sum nodal values.
```

**Call signature**

|                   |             |
|-------------------|-------------|
| <b>d_sum_vals</b> | (parameter) |
|-------------------|-------------|

**Arguments**

- parameter :  $p$  or  $u$

```
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 = 'd_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**

|                         |                       |
|-------------------------|-----------------------|
| <b>d_surface_moment</b> | (material, parameter) |
|-------------------------|-----------------------|

**Arguments**

- material :  $\underline{x}_0$  (special)
- parameter : any variable

```
arg_shapes = {'material': 'D', 'parameter': 'N'}
arg_types = ('material', 'parameter')
static function ()
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 = 'surface'
name = 'd_surface_moment'

class sfepy.terms.terms_basic.SurfaceTerm(name, arg_str, integral, region, **kwargs)
Surface of a domain. Uses approximation of the parameter variable.
```

**Definition**

$$\int_{\Gamma} 1$$

**Call signature**

|                  |             |
|------------------|-------------|
| <b>d_surface</b> | (parameter) |
|------------------|-------------|

**Arguments**

- parameter : any variable

```
arg_shapes = {'parameter': 'N'}
```

```
arg_types = ('parameter',)
```

```
integration = 'surface'
```

```
name = 'd_surface'
```

```
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**

|                         |             |
|-------------------------|-------------|
| <b>d_volume_surface</b> | (parameter) |
|-------------------------|-------------|

**Arguments**

- parameter : any variable

```
arg_shapes = {'parameter': 'N'}
```

```
arg_types = ('parameter',)
```

```
static function()
```

```
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 = 'surface'
```

```
name = 'd_volume_surface'
```

```
class sfepy.terms.terms_basic.VolumeTerm(name, arg_str, integral, region, **kwargs)
Volume of a domain. Uses approximation of the parameter variable.
```

**Definition**

$$\int_{\Omega} 1$$

**Call signature**

|                 |             |
|-----------------|-------------|
| <b>d_volume</b> | (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)
name = ‘d_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**

|                |                  |
|----------------|------------------|
| <b>dw_zero</b> | (virtual, state) |
|----------------|------------------|

**Arguments**

- virtual : *q* or *v*
- state : *p* or *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**

$$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) p(\tau) \right] e_{ij}(\underline{v}) ,$$

$$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) \right] q$$

**Call signature**

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| <b>dw_biot_eth</b> | ( <i>ts</i> , material_0, material_1, virtual, state) |
|                    | ( <i>ts</i> , 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 = {'virtual/grad': ('D', None), 'state/div': 'D', 'state/grad': 1, 'material_0': 'S, 1', 'material_1': '1, 1', 'virt'
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

$$\begin{aligned} & - \int_{\Omega} \alpha_{ij} \bar{p} \\ \text{vector for } K \leftarrow \mathcal{I}_h : & - \int_{T_K} \alpha_{ij} \bar{p} / \int_{T_K} 1 \\ & - \alpha_{ij} \bar{p}|_{qp} \end{aligned}$$

### Call signature

|                |                       |
|----------------|-----------------------|
| ev_biot_stress | (material, parameter) |
|----------------|-----------------------|

### Arguments

- material :  $\alpha_{ij}$
- parameter :  $\bar{p}$

```
arg_shapes = {'material': 'S, 1', 'parameter': 1}
arg_types = ('material', 'parameter')
static function (out, val_qp, mat, vg, fnode)
get_fargs (mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
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**

|                   |                                |
|-------------------|--------------------------------|
| <b>dw_biot_th</b> | (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 = {'virtual/grad': ('D', None), 'state/grad': 1, 'state/div': 'D', 'material': ':: N, S, 1', 'virtual/div': (1, 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  $\alpha_{ij}$  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**

|                |                                      |
|----------------|--------------------------------------|
| <b>dw_biot</b> | (material, virtual, state)           |
|                | (material, state, virtual)           |
|                | (material, parameter_v, parameter_s) |

**Arguments 1**

- material :  $\alpha_{ij}$

- virtual :  $v$
  - state :  $p$

## Arguments 2

- material :  $\alpha_{ij}$
  - state :  $\underline{u}$
  - virtual :  $q$

Arguments 3

- material :  $\alpha_{ij}$
  - parameter\_v :  $u$
  - parameter\_s :  $p$

## **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 = {‘state’: ‘D’, ‘material’: ‘1, 1’, ‘virtual’: (‘D’, ‘state’)}  
arg_shapes
```

```
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)
```

```

integ = 'surface'
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**

|                           |                                |
|---------------------------|--------------------------------|
| <b>dw_non_penetration</b> | (opt_material, virtual, state) |
|                           | (opt_material, state, virtual) |

**Arguments 1**

- material :  $c$  (optional)
- virtual :  $\underline{v}$
- state :  $\lambda$

**Arguments 2**

- material :  $c$  (optional)
- state :  $\underline{u}$
- virtual :  $\hat{\lambda}$

```

arg_shapes = [{‘virtual/grad’: (‘D’, None), ‘opt_material’: ‘1, 1’, ‘state/grad’: 1, ‘state/div’: ‘D’, ‘virtual/div’: (1, 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 = ‘surface’

modes = (‘grad’, ‘div’)

name = ‘dw_non_penetration’

```

**sfepy.terms.terms\_contact module**

```

class sfepy.terms.terms_contact.ContactInfo(region, integral, geo, state)

```

Various contact-related data of contact terms.

**update** (xx)

```

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  $\varepsilon_N$  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

$$\int_{\Gamma_c} \varepsilon_N \langle g_N(u) \rangle \underline{nv}$$

#### Call signature

|                   |                            |
|-------------------|----------------------------|
| <b>dw_contact</b> | (material, virtual, state) |
|-------------------|----------------------------|

#### Arguments

- material :  $\varepsilon_N$
- virtual :  $\underline{v}$
- state :  $u$

```
arg_shapes = {'state': 'D', 'material': '.. 1', '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 = 'surface'

name = 'dw_contact'
```

### 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

|                           |                            |
|---------------------------|----------------------------|
| <b>dw_advect_div_free</b> | (material, virtual, state) |
|---------------------------|----------------------------|

#### Arguments

- material :  $y$
- virtual :  $q$
- state :  $p$

```
arg_shapes = {'state': '1', 'material': 'D, 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

|                            |                             |
|----------------------------|-----------------------------|
| <b>dw_convect_v_grad_s</b> | (virtual, state_v, state_s) |
|----------------------------|-----------------------------|

#### Arguments

- virtual :  $q$
- state\_v :  $\underline{u}$
- state\_s :  $p$

```
arg_shapes = [{‘state_s’: 1, ‘virtual’: (1, ‘state_s’), ‘state_v’: ‘D’}]
arg_types = ('virtual', 'state_v', 'state_s')
function()
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 copupling 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

|                              |                                      |
|------------------------------|--------------------------------------|
| <b>dw_diffusion_coupling</b> | (material, virtual, state)           |
|                              | (material, state, virtual)           |
|                              | (material, parameter_1, parameter_2) |

#### Arguments

- material :  $K_j$
- virtual :  $q$

- state :  $p$

```
arg_shapes = {'parameter_2': 1, 'state': 1, 'material': 'D, 1', 'parameter_1': 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

|                       |                     |
|-----------------------|---------------------|
| <b>dw_diffusion_r</b> | (material, virtual) |
|-----------------------|---------------------|

#### Arguments

- material :  $K_j$
- virtual :  $q$

```
arg_shapes = {'material': 'D, 1', 'virtual': (1, None)}
arg_types = ('material', 'virtual')
static function ()
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

$$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p, \int_{\Omega} K_{ij} \nabla_i \bar{p} \nabla_j r$$

#### Call signature

|                     |                                      |
|---------------------|--------------------------------------|
| <b>dw_diffusion</b> | (material, virtual, state)           |
|                     | (material, parameter_1, parameter_2) |

#### Arguments 1

- material :  $K_{ij}$

- virtual :  $q$
- state :  $p$

### Arguments 2

- material :  $K_{ij}$
- parameter\_1 :  $\bar{p}$
- parameter\_2 :  $r$

```
arg_shapes = {'parameter_2': 1, 'state': 1, 'material': 'D, D', 'parameter_1': 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 = {'map': {'K': 'material', 'u': 'state'}, 'expression': 'div( K * grad( u ) )'}
class sfepy.terms.terms_diffusion.DiffusionVelocityTerm (name, arg_str, integral, region, **kwargs)
Evaluate diffusion velocity.
```

Supports ‘eval’, ‘el\_avg’ and ‘qp’ evaluation modes.

### Definition

$$\begin{aligned} & - \int_{\Omega} K_{ij} \nabla_j \bar{p} \\ \text{vector for } K \leftarrow \mathcal{I}_h : & - \int_{T_K} K_{ij} \nabla_j \bar{p} / \int_{T_K} 1 \\ & - K_{ij} \nabla_j \bar{p} \end{aligned}$$

### Call signature

|                              |                       |
|------------------------------|-----------------------|
| <b>ev_diffusion_velocity</b> | (material, parameter) |
|------------------------------|-----------------------|

### Arguments

- material :  $K_{ij}$
- parameter :  $\bar{p}$

```
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)
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

$$\int_{\Omega} c \nabla q \cdot \nabla p, \int_{\Omega} c \nabla \bar{p} \cdot \nabla r$$

#### Call signature

|                   |                                          |
|-------------------|------------------------------------------|
| <b>dw_laplace</b> | (opt_material, virtual, state)           |
|                   | (opt_material, parameter_1, parameter_2) |

#### Arguments 1

- material :  $c$
- virtual :  $q$
- state :  $p$

#### Arguments 2

- material :  $c$
- parameter\_1 :  $\bar{p}$
- parameter\_2 :  $r$

```
arg_shapes = [{‘opt_material’: ‘1, 1’, ‘state’: 1, ‘parameter_1’: 1, ‘virtual’: (1, ‘state’), ‘parameter_2’: 1}, {‘opt_material’: ‘1, 1’, ‘state’: 1, ‘parameter_1’: 1, ‘virtual’: (1, ‘state’), ‘parameter_2’: 1}]
arg_types = ((‘opt_material’, ‘virtual’, ‘state’), (‘opt_material’, ‘parameter_1’, ‘parameter_2’))
modes = (‘weak’, ‘eval’)
name = ‘dw_laplace’
set_arg_types()
symbolic = {‘map’: {‘c’: ‘opt_material’, ‘u’: ‘state’}, ‘expression’: ‘c * div( grad( u ) )’}
```

```
class sfepy.terms.terms_diffusion.SDDiffusionTerm(name, arg_str, integral, region,
**kwargs)
```

Diffusion sensitivity analysis term.

#### Definition

$$\int_{\Omega} [(\operatorname{div} \underline{\mathcal{V}}) K_{ij} \nabla_i q \nabla_j p - K_{ij} (\nabla_j \underline{\mathcal{V}} \nabla q) \nabla_i p - K_{ij} \nabla_j q (\nabla_i \underline{\mathcal{V}} \nabla p)]$$

#### Call signature

|                       |                                                               |
|-----------------------|---------------------------------------------------------------|
| <b>d_sd_diffusion</b> | (material, parameter_q, parameter_p, parameter_mesh_velocity) |
|-----------------------|---------------------------------------------------------------|

#### Arguments

- material:  $K_{ij}$
- parameter\_q:  $q$
- parameter\_p:  $p$
- parameter\_mesh\_velocity:  $\underline{\mathcal{V}}$

```
arg_shapes = {‘parameter_mesh_velocity’: ‘D’, ‘parameter_q’: 1, ‘material’: ‘D, D’, ‘parameter_p’: 1}
```

```

arg_types = ('material', 'parameter_q', 'parameter_p', 'parameter_mesh_velocity')
static function():

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 = 'd_sd_diffusion'

class sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm(name, arg_str, integral, region, **kwargs)
    Surface flux operator term.

```

**Definition**

$$\int_{\Gamma} \underline{q}\underline{n} \cdot \underline{\underline{K}} \cdot \nabla p$$

**Call signature**

|                        |                                                         |
|------------------------|---------------------------------------------------------|
| <b>dw_surface_flux</b> | ( <i>opt_material</i> , <i>virtual</i> , <i>state</i> ) |
|------------------------|---------------------------------------------------------|

**Arguments**

- material :  $\underline{\underline{K}}$
- virtual :  $\underline{q}$
- state :  $\underline{p}$

```

arg_shapes = [{'opt_material': 'D, D', 'state': 1, 'virtual': (1, 'state')}, {'opt_material': None}]
arg_types = ('opt_material', 'virtual', 'state')

function():

get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'surface_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**

$$\int_{\Gamma} \underline{n} \cdot K_{ij} \nabla_j \bar{p}$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p} / \int_{T_K} 1$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p}$

**Call signature**

|                       |                                        |
|-----------------------|----------------------------------------|
| <b>d_surface_flux</b> | ( <i>material</i> , <i>parameter</i> ) |
|-----------------------|----------------------------------------|

## Arguments

- material:  $K$
- parameter:  $\bar{p}$ ,

```
arg_shapes = {'material': 'D, D', 'parameter': 1}
arg_types = ('material', 'parameter')
static function()
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 = 'surface_extra'
name = 'd_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

|                     |                                          |
|---------------------|------------------------------------------|
| <b>dw_bc_newton</b> | (material_1, material_2, virtual, state) |
|---------------------|------------------------------------------|

## Arguments

- material\_1 :  $\alpha$
- material\_2 :  $p_{\text{outer}}$
- virtual :  $q$
- state :  $p$

```
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'state': 1, 'virtual': (1, 'state')}
arg_types = ('material_1', 'material_2', 'virtual', 'state')
get_fargs (alpha, p_outer, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
mode = 'weak'
name = 'dw_bc_newton'
```

```
class sfePy.terms.terms_dot.DotProductSurfaceTerm(name, arg_str, integral, region,
**kwargs)
Surface  $L^2(\Gamma)$  dot product for both scalar and vector fields.
```

### Definition

$$\int_{\Gamma} qp, \int_{\Gamma} \underline{v} \cdot \underline{u}, \int_{\Gamma} \underline{v} \cdot np, \int_{\Gamma} qn \cdot \underline{u}, \int_{\Gamma} pr, \int_{\Gamma} \underline{u} \cdot \underline{w}, \int_{\Gamma} \underline{w} \cdot np \\ \int_{\Gamma} cqp, \int_{\Gamma} cv \cdot \underline{u}, \int_{\Gamma} cpr, \int_{\Gamma} cu \cdot \underline{w} \\ \int_{\Gamma} \underline{v} \cdot \underline{\underline{M}} \cdot \underline{u}, \int_{\Gamma} \underline{u} \cdot \underline{\underline{M}} \cdot \underline{w}$$

## Call signature

|                       |                                          |
|-----------------------|------------------------------------------|
| <b>dw_surface_dot</b> | (opt_material, virtual, state)           |
|                       | (opt_material, parameter_1, parameter_2) |

## Arguments 1

- material :  $c$  or  $M$  (optional)
  - virtual :  $q$  or  $v$
  - state :  $p$  or  $u$

## Arguments 2

- material :  $c$  or  $M$  (optional)
  - parameter\_1 :  $p$  or  $u$
  - parameter\_2 :  $r$  or  $w$

```

arg_shapes = [{‘opt_material’: ‘1, 1’, ‘state’: 1, ‘parameter_1’: 1, ‘virtual’: (1, ‘state’), ‘parameter_2’: 1}, {‘opt_mater
arg_types = ((‘opt_material’, ‘virtual’, ‘state’), (‘opt_material’, ‘parameter_1’, ‘parameter_2’))
integration = ‘surface’
modes = (‘weak’, ‘eval’)
name = ‘dw_surface_dot’

class sfepy.terms.terms_dot.DotProductVolumeTerm(name, arg_str, integral, region,
  **kwargs)
    Volume  $L^2(\Omega)$  weighted dot product for both scalar and vector fields. Can be evaluated. Can use derivatives.

```

## Definition

$$\int_{\Omega} qp, \int_{\Omega} \underline{v} \cdot \underline{u}, \int_{\Omega} pr, \int_{\Omega} \underline{u} \cdot \underline{w}$$

$$\int_{\Omega} cqp, \int_{\Omega} c\underline{v} \cdot \underline{u}, \int_{\Omega} cpr, \int_{\Omega} c\underline{u} \cdot \underline{w}$$

$$\int_{\Omega} \underline{v} \cdot (\underline{M}u), \int_{\Omega} \underline{u} \cdot (\underline{M}w)$$

## Call signature

|                      |                                          |
|----------------------|------------------------------------------|
| <b>dw_volume_dot</b> | (opt_material, virtual, state)           |
|                      | (opt_material, parameter_1, parameter_2) |

## Arguments 1

- material :  $c$  or  $\underline{\underline{M}}$  (optional)
  - virtual :  $q$  or  $v$

- state :  $p$  or  $\underline{u}$

### Arguments 2

- material :  $c$  or  $\underline{\underline{M}}$  (optional)
- parameter\_1 :  $p$  or  $\underline{u}$
- parameter\_2 :  $r$  or  $\underline{w}$

```
arg_shapes = [{‘opt_material’: ‘1, 1’, ‘state’: 1, ‘parameter_1’: 1, ‘virtual’: (1, ‘state’), ‘parameter_2’: 1}, {‘opt_material’: ‘1, 1’, ‘state’: 1, ‘parameter_1’: 1, ‘virtual’: (1, ‘state’)}]
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)
modes = (‘weak’, ‘eval’)
name = ‘dw_volume_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> | ( <code>ts</code> , <code>material_0</code> , <code>material_1</code> , <code>virtual</code> , <code>state</code> ) |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------|

### 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 ()
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

|                                        |                                |
|----------------------------------------|--------------------------------|
| <code>dw_volume_dot_w_scalar_th</code> | (ts, material, virtual, state) |
|----------------------------------------|--------------------------------|

#### Arguments

- ts : TimeStepper instance
- material :  $\mathcal{G}(\tau)$
- virtual :  $q$
- state :  $p$

```
arg_shapes = {'state': 1, 'material': '.: N, 1, 1', 'virtual': (1, 'state')}
arg_types = ('ts', 'material', 'virtual', 'state')

static function()
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

|                                |                            |
|--------------------------------|----------------------------|
| <code>dw_s_dot_grad_i_s</code> | (material, virtual, state) |
|--------------------------------|----------------------------|

#### Arguments

- material :  $i$
- virtual :  $q$
- state :  $p$

```
arg_shapes = {'state': 1, 'material': '.: 1, 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

$$\int_{\Omega} q\underline{y} \cdot \nabla p, \int_{\Omega} p\underline{y} \cdot \nabla q$$

#### Call signature

|                         |                            |
|-------------------------|----------------------------|
| <b>dw_s_dot_mgrad_s</b> | (material, virtual, state) |
|                         | (material, state, virtual) |

#### Arguments 1

- material :  $\underline{y}$
- virtual :  $\underline{q}$
- state :  $p$

#### Arguments 2

- material :  $\underline{y}$
- state :  $p$
- virtual :  $\underline{q}$

```
arg_shapes = [{‘state/grad_virtual’: 1, ‘state/grad_state’: 1, ‘material’: ‘D’, 1}, {‘virtual/grad_state’: (1, None), ‘virtual/grad_virtual’: 1}]
```

```
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

$$\begin{aligned} & \int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q \\ & \int_{\Omega} c\underline{v} \cdot \nabla p, \int_{\Omega} cu \cdot \nabla q \\ & \int_{\Omega} \underline{v} \cdot (\underline{M} \nabla p), \int_{\Omega} \underline{u} \cdot (\underline{M} \nabla q) \end{aligned}$$

#### Call signature

|                        |                                          |
|------------------------|------------------------------------------|
| <b>dw_v_dot_grad_s</b> | (opt_material, virtual, state)           |
|                        | (opt_material, state, virtual)           |
|                        | (opt_material, parameter_v, parameter_s) |

#### Arguments 1

- material :  $c$  or  $M$  (optional)
  - virtual :  $v$
  - state :  $p$

## Arguments 2

- material :  $c$  or  $M$  (optional)
  - state :  $u$
  - virtual :  $q$

## Arguments 3

- material :  $c$  or  $M$  (optional)
  - parameter\_v :  $u$
  - parameter\_s :  $p$

```

arg_shapes = [{‘opt_material’: ‘1, 1’, ‘state/s_weak’: ‘D’, ‘parameter_s’: 1, ‘virtual/v_weak’: (‘D’, None), ‘virtual/s_w
arg_types = ((‘opt_material’, ‘virtual’, ‘state’), (‘opt_material’, ‘state’, ‘virtual’), (‘opt_material’, ‘parameter_v’, ‘para
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

$$\int_{\Omega} \underline{v} \cdot \underline{m} p, \quad \int_{\Omega} \underline{u} \cdot \underline{m} q$$

## Call signature

|                    |                                      |
|--------------------|--------------------------------------|
| <b>dw_vm_dot_s</b> | (material, virtual, state)           |
|                    | (material, state, virtual)           |
|                    | (material, parameter v, parameter s) |

## Arguments 1

- material :  $m$
  - virtual :  $v$
  - state :  $p$

Arguments 2

- material :  $m$
  - state :  $u$
  - virtual :  $q$

## Arguments 3

- material : m
  - parameter\_v : u
  - parameter\_s : p

```
arg_shapes = [{'state/s_weak': 'D', 'material': 'D,1', 'parameter_s': 1, 'virtual/v_weak': ('D, None), 'virtual/s_weak': None}, {'state/s_weak': 'D,1', 'material': 'D', 'parameter_s': 1, 'virtual/v_weak': ('D, None), 'virtual/s_weak': 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.CauchyStrainSTerm(name, arg_str, integral, region,  
**kwargs)
```

Evaluate Cauchy strain tensor on a surface region.

See *CauchyStrainTerm*.

Supports ‘eval’, ‘el\_avg’ and ‘qp’ evaluation modes.

## Definition

$$\int_{\Gamma} \underline{\underline{e}}(\underline{w})$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$

$$\underline{\underline{e}}(\underline{w})|_{qp}$$

## Call signature

**ev\_cauchy\_strain\_s** (parameter)

## Arguments

- parameter : w

```
arg_types = ('parameter',)
```

```
integration = 'surface_extra'
```

```
name = 'ev_cauchy_strain_s'
```

```
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

$$\int_{\Omega} \underline{\underline{e}}(\underline{w})$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$

$$\underline{\underline{e}}(\underline{w})|_{qp}$$

#### Call signature

|                               |             |
|-------------------------------|-------------|
| <code>ev_cauchy_strain</code> | (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)
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

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau / \int_{T_K} 1$

$$\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau|_{qp}$$

#### Call signature

|                                   |                                         |
|-----------------------------------|-----------------------------------------|
| <code>ev_cauchy_stress_eth</code> | (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

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau$$

vector for  $K \leftarrow \mathcal{I}_h$  :  $\int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau / \int_{T_K} 1$

$$\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) d\tau|_{qp}$$

#### 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

$$\int_{\Omega} D_{ijkl} e_{kl}(\underline{w})$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} D_{ijkl} e_{kl}(\underline{w}) / \int_{T_K} 1$

$$D_{ijkl} e_{kl}(\underline{w})|_{qp}$$

#### Call signature

|                         |                       |
|-------------------------|-----------------------|
| <b>ev_cauchy_stress</b> | (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*)

**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  $6 \times 6$  with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is  $3 \times 3$  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

|                               |                                          |
|-------------------------------|------------------------------------------|
| <b>dw_elastic_wave_cauchy</b> | (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  $6 \times 6$  with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is  $3 \times 3$  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.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

|                           |                                              |
|---------------------------|----------------------------------------------|
| <b>dw_lin_elastic_eth</b> | (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 ()
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**

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \text{ with } D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$$

**Call signature**

|                           |                                                    |
|---------------------------|----------------------------------------------------|
| <b>dw_lin_elastic_iso</b> | (material_1, material_2, virtual, state)           |
|                           | (material_1, material_2, parameter_1, parameter_2) |

**Arguments**

- material\_1 :  $\lambda$
- material\_2 :  $\mu$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

**Arguments 2**

- material :  $D_{ijkl}$
- parameter\_1 :  $\underline{w}$
- parameter\_2 :  $\underline{u}$

```

arg_shapes = {'virtual': ('D', 'state'), 'state': 'D', 'parameter_2': 'D', 'material_1': '1, 1', 'material_2': '1, 1', 'parameter_1': '1, 1'}
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

|                          |                                |
|--------------------------|--------------------------------|
| <b>dw_lin_elastic_th</b> | (ts, material, virtual, state) |
|--------------------------|--------------------------------|

#### Arguments

- ts : TimeStepper instance
- material :  $\mathcal{H}_{ijkl}(\tau)$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': '.. N, S, S', 'virtual': ('D', 'state')}
```

```
arg_types = ('ts', 'material', 'virtual', 'state')
```

```
static function()
```

```
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  $6 \times 6$  with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it is  $3 \times 3$  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

|                       |                                      |
|-----------------------|--------------------------------------|
| <b>dw_lin_elastic</b> | (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 = {'parameter_2': 'D', 'state': 'D', 'material': 'S, S', 'parameter_1': '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  $\sigma_{ij}$  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

|                         |                       |
|-------------------------|-----------------------|
| <b>dw_lin_prestress</b> | (material, virtual)   |
|                         | (material, parameter) |

#### Arguments 1

- material :  $\sigma_{ij}$
- virtual :  $\underline{v}$

#### Arguments 2

- material :  $\sigma_{ij}$
- 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, fnode)
```

```
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.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**

|                          |                                   |
|--------------------------|-----------------------------------|
| <b>dw_lin_strain_fib</b> | (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()
get_fargs (mat1, mat2, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_lin_strain_fib'
```

```
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**

|                          |                                      |
|--------------------------|--------------------------------------|
| <b>dw_nonsym_elastic</b> | (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 = {'parameter_2': 'D', 'state': 'D', 'material': 'D2,D2', 'parameter_1': '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**

|                         |                                                                                                |
|-------------------------|------------------------------------------------------------------------------------------------|
| <b>d_sd_lin_elastic</b> | ( <i>material</i> , <i>parameter_w</i> , <i>parameter_u</i> , <i>parameter_mesh_velocity</i> ) |
|-------------------------|------------------------------------------------------------------------------------------------|

**Arguments**

- *material* :  $D_{ijkl}$
- *parameter\_w* :  $\underline{w}$
- *parameter\_u* :  $\underline{u}$
- *parameter\_mesh\_velocity* :  $\underline{\mathcal{V}}$

```

arg_shapes = {'parameter_mesh_velocity': 'D', 'material': 'S, S', 'parameter_w': 'D', 'parameter_u': 'D'}
arg_types = {'material', 'parameter_w', 'parameter_u', 'parameter_mesh_velocity'}
function()
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 = 'd_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**

|                           |                                                         |
|---------------------------|---------------------------------------------------------|
| <b>dw_electric_source</b> | ( <i>material</i> , <i>virtual</i> , <i>parameter</i> ) |
|---------------------------|---------------------------------------------------------|

**Arguments**

- material :  $c$  (electric conductivity)
- virtual :  $s$  (test function)
- parameter :  $\phi$  (given electric potential)

```
arg_shapes = {'material': '1, 1', 'parameter': 1, 'virtual': (1, None)}
arg_types = ('material', 'virtual', 'parameter')
static function ()
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  $S_{ij} = A f_{\max} \exp \left\{ -\left( \frac{\epsilon - \epsilon_{\text{opt}}}{s} \right)^2 \right\} d_i d_j$ , where  $\epsilon = E_{ij} d_i d_j$  is the Green strain  $E$  projected to the fibre direction  $d$ .
```

### Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

### Call signature

|                                                                                                          |
|----------------------------------------------------------------------------------------------------------|
| <code>dw_tl_fib_a</code> (material_1, material_2, material_3, material_4, material_5,<br>virtual, state) |
|----------------------------------------------------------------------------------------------------------|

### Arguments

- material\_1 :  $f_{\max}$
- material\_2 :  $\epsilon_{\text{opt}}$
- material\_3 :  $s$
- material\_4 :  $d$
- material\_5 :  $A$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
arg_shapes = {'virtual': ('D', 'state'), 'state': 'D', 'material_1': '1, 1', 'material_2': '1, 1', 'material_3': '1, 1', 'material_4': '1, 1', 'material_5': '1, 1'}
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)
```

```
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\_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

$$\begin{aligned}\underline{\underline{F}} &= \frac{\partial \underline{x}}{\partial \underline{X}}|_{qp} = \underline{\underline{I}} + \frac{\partial \underline{u}}{\partial \underline{X}}|_{qp}, \\ \underline{x} &= \underline{X} + \underline{u}, J = \det(\underline{\underline{F}})\end{aligned}$$

### 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)
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 = {'state': 'D', 'material': '1, 1', '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)
```

```
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 = {'sym_inv_c': ('n_el', 'n_qp', 'sym', 1)}  
init_data_struct(state_shape, name='family_data')
```

## `sfepy.terms.terms_hyperelastic_tl` module

**class** `sfePy.terms.terms_hyperelastic_t1.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()
static tan_mod_function()
```

**class** `sfePy.terms.terms_hypereastic_tl.BulkPenaltyTLTerm(*args, **kwargs)`  
Hyperelastic bulk penalty term. Stress  $S_{ij} = K(J - 1) J C_{ij}^{-1}$ .

## Definition

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

|                           |                            |
|---------------------------|----------------------------|
| <b>dw_tl_bulk_penalty</b> | (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()
static tan_mod_function()

class sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm(*args, **kwargs)
    Hyperelastic bulk pressure term. Stress  $S_{ij} = -pJC_{ij}^{-1}$ .
```

Definition

$$\int_{\Omega} S_{ij}(p) \delta E_{ij}(\underline{u}; \underline{v})$$

Call signature

|                            |                           |
|----------------------------|---------------------------|
| <b>dw_tl_bulk_pressure</b> | (virtual, state, state_p) |
|----------------------------|---------------------------|

Arguments

- virtual :  $\underline{v}$
- state :  $\underline{u}$
- state\_p :  $p$

```
arg_shapes = {'state_p': 1, 'state': 'D', '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()
static tan_mod_u_function()
static weak_dp_function()
static weak_function()
```

```
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

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <b>dw_tl_diffusion</b> | (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 ()
```

```
get_eval_shape (perm, 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.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 ()
```

```
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()

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()

class sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm (*args, **kwargs)
    Hyperelastic Mooney-Rivlin term. Effective stress  $S_{ij} = \kappa J^{-\frac{4}{3}}(C_{kk}\delta_{ij} - C_{ij} - \frac{2}{3}I_2C_{ij}^{-1})$ .

```

**Definition**

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

**Call signature**

|                               |                            |
|-------------------------------|----------------------------|
| <b>dw_tl_he_mooney_rivlin</b> | (material, virtual, state) |
|-------------------------------|----------------------------|

**Arguments**

- material :  $\kappa$
- 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()
static tan_mod_function()

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**

|                         |                            |
|-------------------------|----------------------------|
| <b>dw_tl_he_neohook</b> | (material, virtual, state) |
|-------------------------|----------------------------|

**Arguments**

- material :  $\mu$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```

family_data_names = ['det_f', 'tr_c', 'sym_inv_c']
name = 'dw_tl_he_neohook'
static stress_function()

```

```
static tan_mod_function()

class sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm(*args, **kwargs)
    Surface flux term in the total Lagrangian formulation, consistent with DiffusionTLTerm.
```

#### Definition

$$\int_{\Gamma} \underline{\nu} \cdot \underline{\underline{K}}(\underline{u}^{(n-1)}) \frac{\partial p}{\partial \underline{X}}$$

#### Call signature

|                          |                                                    |
|--------------------------|----------------------------------------------------|
| <b>d_tl_surface_flux</b> | (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()

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 = 'surface_extra'

name = 'd_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,  $\underline{\underline{F}}(\underline{u})$ , the deformation gradient,  $J = \det(\underline{\underline{F}})$ , and  $\underline{\underline{\sigma}}$  a given traction, often equal to a given pressure, i.e.  $\underline{\underline{\sigma}} = \pi \underline{\underline{I}}$ .
```

#### Definition

$$\int_{\Gamma} \underline{\nu} \cdot \underline{\underline{F}}^{-1} \cdot \underline{\underline{\sigma}} \cdot \underline{v} J$$

#### Call signature

|                               |                                |
|-------------------------------|--------------------------------|
| <b>dw_tl_surface_traction</b> | (opt_material, virtual, state) |
|-------------------------------|--------------------------------|

#### Arguments

- material :  $\underline{\underline{\sigma}}$
- virtual :  $\underline{v}$
- 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 ()
get_fargs (mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'surface_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  $\underline{v}$ , the outward unit normal vector w.r.t. the undeformed surface,  $\underline{\underline{F}}(\underline{u})$ , the deformation gradient, and  $J = \det(\underline{\underline{F}})$ . Uses the approximation of  $\underline{u}$  for the deformed surface coordinates  $\underline{x}$ .
```

**Definition**

$$1/D \int_{\Gamma} \underline{v} \cdot \underline{\underline{F}}^{-1} \cdot \underline{x} J$$

**Call signature**

|                            |             |
|----------------------------|-------------|
| <b>d_tl_volume_surface</b> | (parameter) |
|----------------------------|-------------|

**Arguments**

- parameter :  $\underline{u}$

```
arg_shapes = {'parameter': 'D'}
arg_types = ('parameter',)
family_data_names = ['det_f', 'inv_f']
static function ()
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 = 'surface_extra'
name = 'd_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**

|                     |                  |
|---------------------|------------------|
| <b>dw_tl_volume</b> | (virtual, state) |
|---------------------|------------------|

**Arguments**

- virtual :  $q$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f', 'sym_inv_c']
static function()
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

|                           |                            |
|---------------------------|----------------------------|
| <b>dw_ul_bulk_penalty</b> | (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()
static tan_mod_function()
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

|                            |                           |
|----------------------------|---------------------------|
| <b>dw_ul_bulk_pressure</b> | (virtual, state, state_p) |
|----------------------------|---------------------------|

### Arguments

- virtual :  $\underline{v}$
- state :  $\underline{u}$
- state\_p :  $p$

```

arg_shapes = {'state_p': 1, 'state': 'D', '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()
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()
static tan_mod_u_function()
static weak_dp_function()
static weak_function()

class sfepy.terms.terms_hypereastic_ul.CompressibilityULTerm(*args, **kwargs)
    Compressibility term for the updated Lagrangian formulation

```

#### Definition

$$\frac{\int_{\Omega} 1}{\gamma p q}$$

#### Call signature

|                           |                                         |
|---------------------------|-----------------------------------------|
| <b>dw_ul_compressible</b> | (material, virtual, state, parameter_u) |
|---------------------------|-----------------------------------------|

#### Arguments

- material :  $\gamma$
- virtual :  $q$
- state :  $p$
- parameter\_u :  $(u)$

```

arg_shapes = {'state': 1, 'material': '1, 1', 'parameter_u': 'D', 'virtual': (1, 'state')}
arg_types = ('material', 'virtual', 'state', 'parameter_u')
family_data_names = ['mtx_f', 'det_f']
static function()
get_fargs(bulk, virtual, state, parameter_u, mode=None, term_mode=None, diff_var=None,
          **kwargs)
name = 'dw_ul_compressible'

```

```

class sfepy.terms.terms_hypereastic_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()

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()

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

|                                     |                            |
|-------------------------------------|----------------------------|
| <code>dw_ul_he_mooney_rivlin</code> | (material, virtual, state) |
|-------------------------------------|----------------------------|

#### Arguments

- material :  $\kappa$
- 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()

static tan_mod_function()

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

|                               |                            |
|-------------------------------|----------------------------|
| <code>dw_ul_he_neohook</code> | (material, virtual, state) |
|-------------------------------|----------------------------|

#### Arguments

- material :  $\mu$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
family_data_names = ['det_f', 'tr_b', 'sym_b']

name = 'dw_ul_he_neohook'

static stress_function()

static tan_mod_function()
```

---

**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**

|                           |                  |
|---------------------------|------------------|
| <code>dw_ul_volume</code> | (virtual, state) |
|---------------------------|------------------|

**Arguments**

- virtual :  $q$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f']
static function ()
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\_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**

|                             |                                                         |
|-----------------------------|---------------------------------------------------------|
| <code>dw_tl_membrane</code> | (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_a2': '1, 1', 'state': 'D', 'material_a1': '1, 1', 'material_h0': '1, 1', '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 = 'surface'
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\_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

|                   |                  |
|-------------------|------------------|
| <b>dw_convect</b> | (virtual, state) |
|-------------------|------------------|

### Arguments

- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'virtual': ('D', 'state')}
```

```
arg_types = ('virtual', 'state')
```

```
static function ()
```

```
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

$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \nabla \underline{u} : \nabla \underline{w}$$

$$\int_{\Omega} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nabla \underline{u} : \nabla \underline{w}$$

**Call signature**

|                    |                                          |
|--------------------|------------------------------------------|
| <b>dw_div_grad</b> | (opt_material, virtual, state)           |
|                    | (opt_material, parameter_1, parameter_2) |

**Arguments 1**

- material :  $\nu$  (viscosity, optional)
- virtual :  $\underline{v}$
- state :  $\underline{u}$

**Arguments 2**

- material :  $\nu$  (viscosity, optional)
- parameter\_1 :  $\underline{u}$
- parameter\_2 :  $\underline{w}$

```

arg_shapes = [{"opt_material": "1,1", "state": "D", "parameter_1": "D", "virtual": ("D", "state"), "parameter_2": "D"}, {"opt_material": "D", "parameter_1": "D", "parameter_2": "D"}]
arg_types = (({"opt_material": "virtual", "state": "state"}, {"opt_material": "parameter_1", "parameter_2": "parameter_2"}))

d_div_grad(out, grad1, grad2, mat, vg, fmode)
static function()

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**

|               |                         |
|---------------|-------------------------|
| <b>dw_div</b> | (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

$$\int_{\Omega} \nabla \cdot \underline{u}$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla \cdot \underline{u} / \int_{T_K} 1$   
 $(\nabla \cdot \underline{u})|_{qp}$

#### Call signature

|        |             |
|--------|-------------|
| ev_div | (parameter) |
|--------|-------------|

#### Arguments

- parameter :  $\underline{u}$

```
arg_shapes = {'parameter': 'D'}
arg_types = ('parameter',)

static function (out, div, vg, fnode)
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_div'

class sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm(name,      arg_str,
   integral,      region,
   **kwargs)
Grad-div stabilization term ( $\gamma$  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 :  $\gamma$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'virtual': ('D', 'state')}
```

```
arg_types = ('material', 'virtual', 'state')
```

```
static function ()
```

```
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**

$$\int_{\Omega} \nabla p \text{ or } \int_{\Omega} \nabla \underline{w}$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla p / \int_{T_K} 1 \text{ or } \int_{T_K} \nabla \underline{w} / \int_{T_K} 1$

$$(\nabla p)|_{qp} \text{ or } \nabla \underline{w}|_{qp}$$

**Call signature**

|                |             |
|----------------|-------------|
| <b>ev_grad</b> | (parameter) |
|----------------|-------------|

**Arguments**

- parameter :  $p$  or  $\underline{w}$

```
arg_shapes = {'parameter': 'N'}
```

```
arg_types = ('parameter',)
```

```
static function (out, grad, 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)
```

```
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**

$$\int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v}$$

$$((\underline{b} \cdot \nabla) \underline{u})|_{qp}$$

**Call signature**

|                        |                            |
|------------------------|----------------------------|
| <b>dw_lin_convect2</b> | (material, virtual, state) |
|------------------------|----------------------------|

**Arguments**

- material :  $\underline{b}$
- virtual :  $\underline{v}$

- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': 'D, 1', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')
static function()

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

$$\int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v}$$
$$((\underline{b} \cdot \nabla) \underline{u})|_{qp}$$

#### Call signature

|                       |                             |
|-----------------------|-----------------------------|
| <b>dw_lin_convect</b> | (virtual, parameter, state) |
|-----------------------|-----------------------------|

#### Arguments

- virtual :  $\underline{v}$
- parameter :  $\underline{b}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'parameter': 'D', 'virtual': ('D', 'state')}
arg_types = ('virtual', 'parameter', 'state')
static function()

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 ( $\tau$  is a local stabilization parameter).
```

#### Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$$

#### Call signature

|                     |                                       |
|---------------------|---------------------------------------|
| <b>dw_st_pspg_c</b> | (material, virtual, parameter, state) |
|---------------------|---------------------------------------|

#### Arguments

- material :  $\tau_K$
- virtual :  $q$

- parameter :  $b$
- state :  $u$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'parameter': 'D', 'virtual': (1, None)}
arg_types = ('material', 'virtual', 'parameter', 'state')
static function ()
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 (  $\tau$  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**

|                     |                                          |
|---------------------|------------------------------------------|
| <b>dw_st_pspg_p</b> | (opt_material, virtual, state)           |
|                     | (opt_material, parameter_1, parameter_2) |

**Arguments**

- material :  $\tau_K$
- 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 (  $\delta$  is a local stabilization parameter).
```

**Definition**

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((b \cdot \nabla) u) \cdot ((b \cdot \nabla) v)$$

**Call signature**

|                     |                                       |
|---------------------|---------------------------------------|
| <b>dw_st_supg_c</b> | (material, virtual, parameter, state) |
|---------------------|---------------------------------------|

**Arguments**

- material :  $\delta_K$
- virtual :  $v$
- parameter :  $b$
- state :  $u$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'parameter': 'D', 'virtual': ('D', 'state')}

arg_types = ('material', 'virtual', 'parameter', 'state')

static function():

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 (  $\delta$  is a local stabilization parameter).

#### Definition

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v})$$

#### Call signature

|                     |                                       |
|---------------------|---------------------------------------|
| <b>dw_st_supg_p</b> | (material, virtual, parameter, state) |
|---------------------|---------------------------------------|

#### Arguments

- material :  $\delta_K$
- virtual :  $\underline{v}$
- parameter :  $\underline{b}$
- state :  $p$

```
arg_shapes = {'state': 1, 'material': '1, 1', 'parameter': 'D', 'virtual': ('D', None)}

arg_types = ('material', 'virtual', 'parameter', 'state')

static function():

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

$$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u} \text{ or } \int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$$

#### Call signature

|                  |                                          |
|------------------|------------------------------------------|
| <b>dw_stokes</b> | (opt_material, virtual, state)           |
|                  | (opt_material, state, virtual)           |
|                  | (opt_material, parameter_v, parameter_s) |

#### Arguments 1

- material :  $c$  (optional)

- virtual :  $\underline{v}$
- state :  $p$

**Arguments 2**

- material :  $c$  (optional)
- state :  $\underline{u}$
- virtual :  $q$

**Arguments 3**

- material :  $c$  (optional)
- parameter\_v :  $\underline{u}$
- parameter\_s :  $p$

```
arg_shapes = [{‘opt_material’: ‘1, 1’, ‘state/grad’: 1, ‘state/div’: ‘D’, ‘virtual/grad’: (‘D’, None), ‘parameter_s’: 1, ‘par-  

arg_types = ((‘opt_material’, ‘virtual’, ‘state’), (‘opt_material’, ‘state’, ‘virtual’), (‘opt_material’, ‘parameter_v’, ‘para-  

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*, *re-*  
*gion*, *\*\*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**

|                           |                            |
|---------------------------|----------------------------|
| <b>dw_stokes_wave_div</b> | (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 = {‘state’: ‘D’, ‘material’: ‘:: 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**

|                       |                            |
|-----------------------|----------------------------|
| <b>dw_stokes_wave</b> | (material, virtual, state) |
|-----------------------|----------------------------|

**Arguments**

- material :  $\underline{\kappa}$
- virtual :  $\underline{v}$
- statee :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': '.: 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'

class sfepy.terms.terms_navier_stokes.SurfaceDivTerm(name, arg_str, integral, region,
   **kwargs)
```

Evaluate divergence of a vector field.

Supports ‘eval’, ‘el\_avg’ and ‘qp’ evaluation modes.

**Definition**

$$\int_{\Gamma} \nabla \cdot \underline{u}$$

vector for  $K \leftarrow \mathcal{I}_h$  :  $\int_{T_K} \nabla \cdot \underline{u} / \int_{T_K} 1$   
 $(\nabla \cdot \underline{u})|_{qp}$

**Call signature**

|                       |             |
|-----------------------|-------------|
| <b>ev_surface_div</b> | (parameter) |
|-----------------------|-------------|

**Arguments**

- parameter :  $\underline{u}$

```
integration = 'surface_extra'
name = 'ev_surface_div'

class sfepy.terms.terms_navier_stokes.SurfaceGradTerm(name, arg_str, integral, region,
**kwargs)
```

Evaluate gradient of a scalar or vector field.

Supports ‘eval’, ‘el\_avg’ and ‘qp’ evaluation modes.

#### Definition

$$\int_{\Gamma} \nabla p \text{ or } \int_{\Gamma} \nabla \underline{w}$$

vector for  $K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla p / \int_{T_K} 1 \text{ or } \int_{T_K} \nabla \underline{w} / \int_{T_K} 1$

$$(\nabla p)|_{qp} \text{ or } \nabla \underline{w}|_{qp}$$

#### Call signature

|                        |             |
|------------------------|-------------|
| <b>ev_surface_grad</b> | (parameter) |
|------------------------|-------------|

#### Arguments

- parameter :  $p$  or  $\underline{w}$

```
integration = 'surface_extra'
name = 'ev_surface_grad'
```

### **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

|                          |                                      |
|--------------------------|--------------------------------------|
| <b>dw_piezo_coupling</b> | (material, virtual, state)           |
|                          | (material, state, virtual)           |
|                          | (material, parameter_v, parameter_s) |

#### Arguments 1

- material :  $g_{kij}$
- virtual :  $\underline{v}$
- state :  $p$

#### Arguments 2

- material :  $g_{kij}$

- state :  $\underline{u}$
- virtual :  $q$

### Arguments 3

- material :  $g_{kij}$
- parameter\_v :  $\underline{u}$
- parameter\_s :  $p$

```
arg_shapes = {'state/grad': 1, 'state/div': 'D', 'material': 'D, S', 'virtual/grad': ('D', None), 'parameter_s': 1, 'parameter_v': 1}
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**

|                              |                       |
|------------------------------|-----------------------|
| <code>ev_piezo_stress</code> | (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'
```

**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**

|                            |                     |
|----------------------------|---------------------|
| <code>dw_point_load</code> | (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 = {'state': 'D', 'material': '.: 1', '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\_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  $\xi$ ,  $\eta$  and  $\zeta$  ( $\xi, \eta, \zeta \in (-1, 1)$ ) 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  $\chi$  - 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

|                    |                                              |
|--------------------|----------------------------------------------|
| <b>dw_shell10x</b> | (material_d, material_drill, virtual, state) |
|--------------------|----------------------------------------------|

### Arguments

- material\_d :  $D$
- material\_drill :  $\chi$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 6, 'material_d': '6, 6', 'virtual': (6, 'state'), 'material_drill': '.. 1'}
```

```
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_base = '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

|                         |                                                                  |
|-------------------------|------------------------------------------------------------------|
| <b>dw_contact_plane</b> | (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_n': '.: D', 'material_a': '.: D', 'material_b': '.: N, D', 'material_f': '1, 2', 'state': 'D', 'virtual': 'D'}
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 = 'surface'
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

|                          |                                                      |
|--------------------------|------------------------------------------------------|
| <b>dw_contact_sphere</b> | (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_r': '.: 1', 'material_c': '.: D', 'virtual': ('D', 'state'), 'material_f': '1, 2', 'state': 'D'}
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 = 'surface'
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,  $\underline{\sigma} \cdot \underline{n}$  is  $\bar{p}\underline{I} \cdot \underline{n}$  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). The symmetric tensor storage is used in the last case: 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

$$\int_{\Gamma} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}, \int_{\Gamma} \underline{v} \cdot \underline{n},$$

#### Call signature

|                             |                           |
|-----------------------------|---------------------------|
| <code>dw_surface_ltr</code> | (opt_material, virtual)   |
|                             | (opt_material, parameter) |

#### Arguments

- material :  $\underline{\sigma}$
- virtual :  $\underline{v}$

```
arg_shapes = [{‘opt_material’: ‘S, 1’, ‘parameter’: ‘D’, ‘virtual’: (‘D’, None)}, {‘opt_material’: ‘D, 1’}, {‘opt_material’
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 = 'surface'
modes = ('weak', 'eval')
name = 'dw_surface_ltr'
set_arg_types ()

class sfepy.terms.terms_surface.SDSurfaceIntegrateTerm (name, arg_str, integral, region,
   **kwargs)
```

Sensitivity of scalar traction.

**Definition**

$$\int_{\Gamma} p \nabla \cdot \underline{\mathcal{V}}$$

**Call signature**

|                               |                                      |
|-------------------------------|--------------------------------------|
| <b>d_sd_surface_integrate</b> | (parameter, parameter_mesh_velocity) |
|-------------------------------|--------------------------------------|

**Arguments**

- parameter :  $p$
- parameter\_mesh\_velocity :  $\underline{\mathcal{V}}$

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'parameter': 1}
arg_types = ('parameter', 'parameter_mesh_velocity')
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 = 'surface'
name = 'd_sd_surface_integrate'

class sfepy.terms.terms_surface.SufaceNormalDotTerm(name, arg_str, integral, region,
   **kwargs)
“Scalar traction” term, (weak form).
```

**Definition**

$$\int_{\Gamma} q \underline{c} \cdot \underline{n}$$

**Call signature**

|                        |                       |
|------------------------|-----------------------|
| <b>dw_surface_ndot</b> | (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 = 'surface'
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

|                |                                           |
|----------------|-------------------------------------------|
| <b>dw_jump</b> | (opt_material, virtual, state_1, state_2) |
|----------------|-------------------------------------------|

#### Arguments

- material :  $c$
- virtual :  $q$
- state\_1 :  $p_1$
- state\_2 :  $p_2$

```
arg_shapes = [{‘state_1’: 1, ‘opt_material’: ‘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 = ‘surface’  
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

|                      |                     |
|----------------------|---------------------|
| <b>dw_volume_lvf</b> | (material, virtual) |
|----------------------|---------------------|

### Arguments

- material :  $f$  or  $f$
- virtual :  $v$  or  $q$

```
arg_shapes = [{‘material’: ‘D’, 1}, {‘virtual’: (‘D’, None)}, {‘material’: ‘1’, 1}, {‘virtual’: (1, None)}]
arg_types = (‘material’, ‘virtual’)
static function ()
get_fargs (mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
name = ‘dw_volume_lvf’
```

## 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()
sfePy.terms.extmods.terms.d_biot_div()
sfePy.terms.extmods.terms.d_diffusion()
sfePy.terms.extmods.terms.d_laplace()
sfePy.terms.extmods.terms.d_lin_elastic()
sfePy.terms.extmods.terms.d_of_nsMinGrad()
sfePy.terms.extmods.terms.d_of_nsSurfMinDPress()
sfePy.terms.extmods.terms.d_piezo_coupling()
sfePy.terms.extmods.terms.d_sd_convect()
sfePy.terms.extmods.terms.d_sd_diffusion()
sfePy.terms.extmods.terms.d_sd_div()
sfePy.terms.extmods.terms.d_sd_div_grad()
sfePy.terms.extmods.terms.d_sd_lin_elastic()
sfePy.terms.extmods.terms.d_sd_st_grad_div()
sfePy.terms.extmods.terms.d_sd_st_pspg_c()
```

```
sfepy.terms.extmods.terms.d_sd_st_pspg_p()  
sfepy.terms.extmods.terms.d_sd_st_supg_c()  
sfepy.terms.extmods.terms.d_sd_volume_dot()  
sfepy.terms.extmods.terms.d_surface_flux()  
sfepy.terms.extmods.terms.d_tl_surface_flux()  
sfepy.terms.extmods.terms.d_tl_volume_surface()  
sfepy.terms.extmods.terms.d_volume_surface()  
sfepy.terms.extmods.terms.de_cauchy_strain()  
sfepy.terms.extmods.terms.de_cauchy_stress()  
sfepy.terms.extmods.terms.de_he_rtm()  
sfepy.terms.extmods.terms.di_surface_moment()  
sfepy.terms.extmods.terms.dq_cauchy_strain()  
sfepy.terms.extmods.terms.dq_def_grad()  
sfepy.terms.extmods.terms.dq_div_vector()  
sfepy.terms.extmods.terms.dq_finite_strain_t1()  
sfepy.terms.extmods.terms.dq_finite_strain_u1()  
sfepy.terms.extmods.terms.dq_grad()  
sfepy.terms.extmods.terms.dq_state_in_qp()  
sfepy.terms.extmods.terms.dq_t1_finite_strain_surface()  
sfepy.terms.extmods.terms.dq_t1_he_stress_bulk()  
sfepy.terms.extmods.terms.dq_t1_he_stress_bulk_active()  
sfepy.terms.extmods.terms.dq_t1_he_stress_mooney_rivlin()  
sfepy.terms.extmods.terms.dq_t1_he_stress_neohook()  
sfepy.terms.extmods.terms.dq_t1_he_tan_mod_bulk()  
sfepy.terms.extmods.terms.dq_t1_he_tan_mod_bulk_active()  
sfepy.terms.extmods.terms.dq_t1_he_tan_mod_mooney_rivlin()  
sfepy.terms.extmods.terms.dq_t1_he_tan_mod_neohook()  
sfepy.terms.extmods.terms.dq_t1_stress_bulk_pressure()  
sfepy.terms.extmods.terms.dq_t1_tan_mod_bulk_pressure_u()  
sfepy.terms.extmods.terms.dq_ul_he_stress_bulk()  
sfepy.terms.extmods.terms.dq_ul_he_stress_mooney_rivlin()  
sfepy.terms.extmods.terms.dq_ul_he_stress_neohook()  
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_bulk()  
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_mooney_rivlin()  
sfepy.terms.extmods.terms.dq_ul_he_tan_mod_neohook()  
sfepy.terms.extmods.terms.dq_ul_stress_bulk_pressure()
```

```
sfepy.terms.extmods.terms.dq_ul_tan_mod_bulk_pressure_u()
sfepy.terms.extmods.terms.dw_adj_convect1()
sfepy.terms.extmods.terms.dw_adj_convect2()
sfepy.terms.extmods.terms.dw_biot_div()
sfepy.terms.extmods.terms.dw_biot_grad()
sfepy.terms.extmods.terms.dw_convect_v_grad_s()
sfepy.terms.extmods.terms.dw_diffusion()
sfepy.terms.extmods.terms.dw_diffusion_r()
sfepy.terms.extmods.terms.dw_div()
sfepy.terms.extmods.terms.dw_electric_source()
sfepy.terms.extmods.terms.dw_grad()
sfepy.terms.extmods.terms.dw_he_rtm()
sfepy.terms.extmods.terms.dw_laplace()
sfepy.terms.extmods.terms.dw_lin_convect()
sfepy.terms.extmods.terms.dw_lin_elastic()
sfepy.terms.extmods.terms.dw_lin_prestress()
sfepy.terms.extmods.terms.dw_lin_strain_fib()
sfepy.terms.extmods.terms.dw_nonsym_elastic()
sfepy.terms.extmods.terms.dw_piezo_coupling()
sfepy.terms.extmods.terms.dw_st_adj1_supg_p()
sfepy.terms.extmods.terms.dw_st_adj2_supg_p()
sfepy.terms.extmods.terms.dw_st_adj_supg_c()
sfepy.terms.extmods.terms.dw_st_grad_div()
sfepy.terms.extmods.terms.dw_st_pspg_c()
sfepy.terms.extmods.terms.dw_st_supg_c()
sfepy.terms.extmods.terms.dw_st_supg_p()
sfepy.terms.extmods.terms.dw_surface_flux()
sfepy.terms.extmods.terms.dw_surface_ltr()
sfepy.terms.extmods.terms.dw_surface_s_v_dot_n()
sfepy.terms.extmods.terms.dw_surface_v_dot_n_s()
sfepy.terms.extmods.terms.dw_t1_diffusion()
sfepy.terms.extmods.terms.dw_t1_surface_traction()
sfepy.terms.extmods.terms.dw_t1_volume()
sfepy.terms.extmods.terms.dw_ul_volume()
sfepy.terms.extmods.terms.dw_v_dot_grad_s_sw()
sfepy.terms.extmods.terms.dw_v_dot_grad_s_vw()
```

```
sfePy.terms.extmods.terms.dw_volume_dot_scalar()
sfePy.terms.extmods.terms.dw_volume_dot_vector()
sfePy.terms.extmods.terms.dw_volume_lvf()
sfePy.terms.extmods.terms.errclear()
sfePy.terms.extmods.terms.he_eval_from_mtx()
sfePy.terms.extmods.terms.he_residuum_from_mtx()
sfePy.terms.extmods.terms.mulAB_integrate()
sfePy.terms.extmods.terms.sym2nonsym()
sfePy.terms.extmods.terms.term_ns_asm_convect()
sfePy.terms.extmods.terms.term_ns_asm_div_grad()
```

PDF version of the documentation: [sfepy\\_manual.pdf](#)

---

**CHAPTER  
EIGHT**

---

**INDICES AND TABLES**

- genindex
- modindex
- search



## BIBLIOGRAPHY

[Logg2012] 1. Logg: Efficient Representation of Computational Meshes. 2012

[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

blockgen, 507  
build\_helpers, 505

### c

convert\_mesh, 507  
cylindergen, 507

### e

edit\_identifiers, 508  
eval\_ns\_forms, 508  
eval\_t1\_forms, 508  
extract\_edges, 508  
extract\_surface, 509  
extractor, 502

### g

gen\_gallery, 509  
gen\_iga\_patch, 510  
gen\_lobatto1d\_c, 510  
gen\_mesh\_prev, 511  
gen\_release\_notes, 511  
gen\_solver\_table, 511  
gen\_term\_table, 511

### h

homogen, 502

### p

phonon, 502  
plot\_condition\_numbers, 512  
plot\_logs, 512  
plot\_mesh, 512  
plot\_quadratures, 512  
plot\_times, 512  
postproc, 502  
probe, 503

### r

run\_tests, 504

### s

save\_basis, 513  
sfepy.applications.application, 514  
sfepy.applications.evp\_solver\_app, 515  
sfepy.applications.pde\_solver\_app, 515  
sfepy.base.base, 516  
sfepy.base.compat, 521  
sfepy.base.conf, 524  
sfepy.base.getch, 526  
sfepy.base.gooptions, 526  
sfepy.base.ioutils, 527  
sfepy.base.log, 531  
sfepy.base.log\_plotter, 532  
sfepy.base.mem\_usage, 532  
sfepy.base.multiproc, 533  
sfepy.base.multiproc\_proc, 533  
sfepy.base.parse\_conf, 534  
sfepy.base.plotutils, 535  
sfepy.base.reader, 535  
sfepy.base.resolve\_deps, 536  
sfepy.base.testing, 536  
sfepy.config, 514  
sfepy.discrete.common.dof\_info, 576  
sfepy.discrete.common.domain, 578  
sfepy.discrete.common.extmods.\_fmfield,  
    578  
sfepy.discrete.common.extmods.\_geommech,  
    578  
sfepy.discrete.common.extmods.assemble,  
    579  
sfepy.discrete.common.extmods.cmesh, 579  
sfepy.discrete.common.extmods.crefcoors,  
    581  
sfepy.discrete.common.extmods.mappings,  
    582  
sfepy.discrete.common.fields, 582  
sfepy.discrete.common.global\_interp, 585  
sfepy.discrete.common.mappings, 587  
sfepy.discrete.common.region, 589  
sfepy.discrete.conditions, 537  
sfepy.discrete.equations, 539  
sfepy.discrete.evaluate, 544

```
sfePy.discrete.evaluate_variable, 546
sfePy.discrete.fem.domain, 592
sfePy.discrete.fem.extmods.bases, 593
sfePy.discrete.fem.extmods.lobatto_bases, 593
sfePy.discrete.fem.facets, 593
sfePy.discrete.fem.fe_surface, 594
sfePy.discrete.fem.fields_base, 595
sfePy.discrete.fem.fields_hierarchic, 600
sfePy.discrete.fem.fields_nodal, 600
sfePy.discrete.fem.geometry_element, 601
sfePy.discrete.fem.history, 602
sfePy.discrete.fem.lcbc_operators, 602
sfePy.discrete.fem.linearizer, 605
sfePy.discrete.fem.mappings, 605
sfePy.discrete.fem.mesh, 606
sfePy.discrete.fem.meshio, 607
sfePy.discrete.fem.periodic, 613
sfePy.discrete.fem.poly_spaces, 614
sfePy.discrete.fem.refine, 615
sfePy.discrete.fem.refine_hanging, 616
sfePy.discrete.fem.utils, 617
sfePy.discrete.functions, 546
sfePy.discrete.iga.domain, 618
sfePy.discrete.iga.domain_generators, 618
sfePy.discrete.iga.extmods.igac, 619
sfePy.discrete.iga.fields, 621
sfePy.discrete.iga.iga, 623
sfePy.discrete.iga.io, 628
sfePy.discrete.iga.mappings, 628
sfePy.discrete.iga.plot_nurbs, 629
sfePy.discrete.iga.utils, 629
sfePy.discrete.integrals, 547
sfePy.discrete.materials, 548
sfePy.discrete.parse_equations, 550
sfePy.discrete.parse_regions, 550
sfePy.discrete.probes, 550
sfePy.discrete.problem, 554
sfePy.discrete.projections, 564
sfePy.discrete.quadratures, 564
sfePy.discrete.simplex_cubature, 566
sfePy.discrete.state, 566
sfePy.discrete.structural.fields, 630
sfePy.discrete.structural.mappings, 631
sfePy.discrete.variables, 568
sfePy.homogenization.band_gaps_app, 631
sfePy.homogenization.coefficients, 632
sfePy.homogenization.coefs_base, 632
sfePy.homogenization.coefs_elastic, 635
sfePy.homogenization.coefs_perfusion, 636
sfePy.homogenization.coefs_phononic, 636
sfePy.homogenization.convolutions, 640
sfePy.homogenization.engine, 640
sfePy.homogenization.homogen_app, 642
sfePy.homogenization.micmac, 643
sfePy.homogenization.recovery, 643
sfePy.homogenization.utils, 646
sfePy.linalg.check_derivatives, 647
sfePy.linalg.eigen, 647
sfePy.linalg.geometry, 648
sfePy.linalg.sparse, 650
sfePy.linalg.sympy_operators, 652
sfePy.linalg.utils, 652
sfePy.mechanics.contact_bodies, 655
sfePy.mechanics.elastic_constants, 655
sfePy.mechanics.extmods.ccontres, 665
sfePy.mechanics.matcoefs, 655
sfePy.mechanics.membranes, 658
sfePy.mechanics.shell110x, 660
sfePy.mechanics.tensors, 661
sfePy.mechanics.units, 663
sfePy.mesh.bspline, 666
sfePy.mesh.geom_tools, 669
sfePy.mesh.mesh_generators, 671
sfePy.mesh.mesh_tools, 674
sfePy.mesh.splinebox, 675
sfePy.parallel.plot_parallel_dofs, 676
sfePy.postprocess.domain_specific, 677
sfePy.postprocess.plot_cmesh, 678
sfePy.postprocess.plot_dofs, 679
sfePy.postprocess.plot_facets, 679
sfePy.postprocess.plot_quadrature, 680
sfePy.postprocess.probes_vtk, 680
sfePy.postprocess.sources, 681
sfePy.postprocess.time_history, 683
sfePy.postprocess.utils, 683
sfePy.postprocess.utils_vtk, 684
sfePy.postprocess.viewer, 685
sfePy.solvers.eigen, 688
sfePy.solvers.ls, 689
sfePy.solvers.ls_mumps, 694
sfePy.solvers.nls, 704
sfePy.solvers.optimize, 708
sfePy.solvers.oseen, 709
sfePy.solvers.qeigen, 710
sfePy.solvers.semismooth_newton, 711
sfePy.solvers.solvers, 711
sfePy.solvers.ts, 713
sfePy.solvers.ts_solvers, 714
sfePy.terms.extmods.terms, 815
sfePy.terms.terms, 738
sfePy.terms.terms_adj_navier_stokes, 742
sfePy.terms.terms_basic, 751
sfePy.terms.terms_biot, 757
sfePy.terms.terms_constraints, 760
```

`sfepy.terms.terms_contact`, 761  
`sfepy.terms.terms_diffusion`, 762  
`sfepy.terms.terms_dot`, 768  
`sfepy.terms.terms_elastic`, 774  
`sfepy.terms.terms_electric`, 783  
`sfepy.terms.terms_fibres`, 784  
`sfepy.terms.terms_hyperelastic_base`, 785  
`sfepy.terms.terms_hyperelastic_tl`, 786  
`sfepy.terms.terms_hyperelastic_ul`, 792  
`sfepy.terms.terms_membrane`, 795  
`sfepy.terms.terms_navier_stokes`, 796  
`sfepy.terms.terms_piezo`, 805  
`sfepy.terms.terms_point`, 807  
`sfepy.terms.terms_shells`, 808  
`sfepy.terms.terms_surface`, 810  
`sfepy.terms.terms_th`, 814  
`sfepy.terms.terms_volume`, 814  
`sfepy.terms.utils`, 815  
`sfepy.version`, 514  
`show_authors`, 513  
`show_mesh_info`, 513  
`show_terms_use`, 513  
`simple`, 505  
`sync_module_docs`, 513

## t

`test_install`, 507  
`tile_periodic_mesh`, 513



# INDEX

## Symbols

\_\_call\_\_() (sfepy.solvers.nls.Newton method), 705  
\_\_call\_\_() (sfepy.solvers.nls.PETScNonlinearSolver method), 706  
\_\_call\_\_() (sfepy.solvers.nls.ScipyBroyden method), 707  
\_\_init\_\_() (sfepy.solvers.nls.Newton method), 706  
\_\_init\_\_() (sfepy.solvers.nls.PETScNonlinearSolver method), 706  
\_\_init\_\_() (sfepy.solvers.nls.ScipyBroyden method), 707  
\_\_metaclass\_\_ (sfepy.solvers.nls.Newton attribute), 706  
\_\_metaclass\_\_ (sfepy.solvers.nls.PETScNonlinearSolver attribute), 706  
\_\_metaclass\_\_ (sfepy.solvers.nls.ScipyBroyden attribute), 707  
\_\_module\_\_ (sfepy.solvers.nls.Newton attribute), 706  
\_\_module\_\_ (sfepy.solvers.nls.PETScNonlinearSolver attribute), 707  
\_\_module\_\_ (sfepy.solvers.nls.ScipyBroyden attribute), 707

## A

a (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 695  
a (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 698  
a (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 701  
a\_elt (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 695  
a\_elt (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 698  
a\_elt (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 701  
a\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 695  
a\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 698  
a\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 701  
AbaqusMeshIO (class in sfepy.discrete.fem.meshio), 608  
AcousticBandGapsApp (class in sfepy.homogenization.band\_gaps\_app), 631

AcousticMassLiquidTensor (class in sfepy.homogenization.coefs\_phononic), 636  
AcousticMassTensor (class in sfepy.homogenization.coefs\_phononic), 636  
actBfT() (in module sfepy.terms.extmods.terms), 815  
adapt\_time\_step() (in module sfepy.solvers.ts\_solvers), 718  
AdaptiveTimeSteppingSolver (class in sfepy.solvers.ts\_solvers), 714  
add\_circle\_probe() (sfepy.postprocess.probes\_vtk.Probe method), 680  
add\_data\_to\_dataset() (sfepy.postprocess.sources.GenericFileSource method), 682  
add\_eas\_dofs() (in module sfepy.mechanics.shell10x), 660  
add\_equation() (sfepy.discrete.equations.Equations method), 539  
add\_from\_bc() (sfepy.discrete.fem.lcbc\_operators.LCBOperators method), 603  
add\_glyphs() (in module sfepy.postprocess.viewer), 687  
add\_group() (sfepy.base.log.Log method), 531  
add\_iso\_surface() (in module sfepy.postprocess.viewer), 687  
add\_line\_probe() (sfepy.postprocess.probes\_vtk.Probe method), 680  
add\_missing() (sfepy.base.conf.ProblemConf method), 524  
add\_points\_probe() (sfepy.postprocess.probes\_vtk.Probe method), 680  
add\_ray\_probe() (sfepy.postprocess.probes\_vtk.Probe method), 681  
add\_scalar\_cut\_plane() (in module sfepy.postprocess.viewer), 687  
add\_strain\_rs() (in module sfepy.homogenization.recovery), 643  
add\_stress\_p() (in module sfepy.homogenization.recovery), 643  
add\_subdomains\_surface() (in module sfepy.postprocess.viewer), 687  
add\_surf() (in module sfepy.postprocess.viewer), 687  
add\_text() (in module sfepy.postprocess.viewer), 687

add\_vector\_cut\_plane() (in module sfepy.postprocess.viewer), 687  
addline() (sfepy.mesh.geom\_tools.geometry method), 670  
addlines() (sfepy.mesh.geom\_tools.geometry method), 670  
addphysicalsurface() (sfepy.mesh.geom\_tools.geometry method), 670  
addphysicalvolume() (sfepy.mesh.geom\_tools.geometry method), 670  
addpoint() (sfepy.mesh.geom\_tools.geometry method), 670  
addpoints() (sfepy.mesh.geom\_tools.geometry method), 670  
addsurface() (sfepy.mesh.geom\_tools.geometry method), 670  
addsurfaces() (sfepy.mesh.geom\_tools.geometry method), 670  
addvolume() (sfepy.mesh.geom\_tools.geometry method), 670  
addvolumes() (sfepy.mesh.geom\_tools.geometry method), 670  
AdjConvect1Term (class in sfepy.terms.terms\_adj\_navier\_stokes), 742  
AdjConvect2Term (class in sfepy.terms.terms\_adj\_navier\_stokes), 742  
AdjDivGradTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 742  
advance() (sfepy.discrete.equations.Equations method), 539  
advance() (sfepy.discrete.problem.Problem method), 554  
advance() (sfepy.discrete.variables.Variable method), 572  
advance() (sfepy.discrete.variables.Variables method), 573  
advance() (sfepy.solvers.ts.TimeStepper method), 713  
advance() (sfepy.solvers.ts.VariableTimeStepper method), 713  
advance() (sfepy.terms.terms.Term method), 738  
advance\_eth\_data() (sfepy.terms.terms\_th.ETHTerm method), 814  
AdvectDivFreeTerm (class in sfepy.terms.terms\_diffusion), 762  
alloc\_extra\_data() (sfepy.discrete.common.extmods.mapping.Mapping method), 582  
ANSYSCDBMeshIO (class in sfepy.discrete.fem.meshio), 607  
any\_from\_args() (sfepy.discrete.fem.poly\_spaces.PolySpace static method), 615  
any\_from\_conf() (sfepy.homogenization.coefs\_base.MiniAppBase static method), 634  
any\_from\_conf() (sfepy.solvers.solvers.Solver static method), 712  
any\_from\_filename() (sfepy.discrete.fem.meshio.MeshIO static method), 610  
append() (sfepy.base.base.Container method), 516  
append() (sfepy.discrete.fem.history.History method), 602  
append() (sfepy.discrete.fem.lcbc\_operators.LCBOperators method), 603  
append() (sfepy.terms.terms.Terms method), 741  
append\_bubbles() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_declarations() (in module gen\_lobatto1d\_c), 510  
append\_edges() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_faces() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_lists() (in module gen\_lobatto1d\_c), 510  
append\_polys() (in module gen\_lobatto1d\_c), 510  
append\_raw() (sfepy.discrete.common.dof\_info.DofInfo method), 576  
append\_tp\_bubbles() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_tp\_edges() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_tp\_faces() (sfepy.discrete.fem.poly\_spaces.LagrangeNodes static method), 614  
append\_variable() (sfepy.discrete.common.dof\_info.DofInfo method), 576  
Application (class in sfepy.applications.application), 514  
AppliedLoadTensor (class in sfepy.homogenization.coefs\_phononic), 636  
apply\_ebc() (sfepy.discrete.equations.Equations method), 539  
apply\_ebc() (sfepy.discrete.state.State method), 567  
apply\_ebc() (sfepy.discrete.variables.FieldVariable method), 568  
apply\_ebc() (sfepy.discrete.variables.Variables method), 573  
apply\_ebc\_to\_matrix() (in module sfepy.discrete.evaluate), 544  
apply\_ic() (sfepy.discrete.equations.Equations method), 539  
apply\_ic() (sfepy.discrete.state.State method), 567  
apply\_ic() (sfepy.discrete.variables.FieldVariable method), 568  
apply\_mappings() (sfepy.discrete.variables.Variables method), 573  
apply\_to\_sequence() (in module sfepy.linalg.utils), 652  
apply\_unit\_multipliers() (in module sfepy.mechanics.units), 665  
approximate() (sfepy.mesh.bspline.BSpline method), 666  
approximate() (sfepy.mesh.bspline.BSplineSurf method), 668  
approximate\_exponential() (in module sfepy.homogenization.convolutions), 640  
approximation\_example() (in module sfepy.mesh.bspline), 669

are\_close() (in module sfepy.solvers.oseen), 710  
 are\_disjoint() (in module sfepy.discrete.common.region), 591  
 arg\_shapes (sfepy.terms.terms.Term attribute), 738  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvectionTerm attribute), 742  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvectionTerm attribute), 742  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.AdjDivGradTerm attribute), 743  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMinGaussTerm attribute), 743  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMinGaussTerm attribute), 744  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMinGaussTerm attribute), 744  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDConvectionTerm attribute), 745  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDDivGradTerm attribute), 745  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDDivTerm attribute), 746  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDDotVolumeTerm attribute), 746  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDGradDivStabilizationTerm attribute), 747  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGCSimulationTerm attribute), 748  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGPStabilizationTerm attribute), 748  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SDSUPGCSimulationTerm attribute), 749  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SUPGCAjStabilizationTerm attribute), 750  
 arg\_shapes (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdjsStabilizationTerm attribute), 751  
 arg\_shapes (sfepy.terms.terms\_basic.IntegrateSurfaceMatTerm attribute), 751  
 arg\_shapes (sfepy.terms.terms\_basic.IntegrateSurfaceOperatorTerm attribute), 752  
 arg\_shapes (sfepy.terms.terms\_basic.IntegrateSurfaceTerm attribute), 753  
 arg\_shapes (sfepy.terms.terms\_basic.IntegrateVolumeMatTerm attribute), 753  
 arg\_shapes (sfepy.terms.terms\_basic.IntegrateVolumeOperatorTerm attribute), 754  
 arg\_shapes (sfepy.terms.terms\_basic.SumNodalValuesTerm attribute), 755  
 arg\_shapes (sfepy.terms.terms\_basic.SurfaceMomentTerm attribute), 755  
 arg\_shapes (sfepy.terms.terms\_basic.SurfaceTerm attribute), 756  
 arg\_shapes (sfepy.terms.terms\_basic.VolumeSurfaceTerm attribute), 756  
 arg\_shapes (sfepy.terms.terms\_basic.VolumeTerm attribute), 757  
 arg\_shapes (sfepy.terms.terms\_basic.ZeroTerm attribute), 757  
 arg\_shapes (sfepy.terms.terms\_biot.BiotETHTerm attribute), 758  
 arg\_shapes (sfepy.terms.terms\_biot.BiotStressTerm attribute), 758  
 arg\_shapes (sfepy.terms.terms\_biot.BiotTerm attribute), 760  
 arg\_shapes (sfepy.terms.terms\_biot.BiotTHTTerm attribute), 759  
 arg\_shapes (sfepy.terms.terms\_constraints.NonPenetrationPenaltyTerm attribute), 760  
 arg\_shapes (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 761  
 arg\_shapes (sfepy.terms.terms\_contact.ContactTerm attribute), 762  
 arg\_shapes (sfepy.terms.terms\_diffusion.AdvectDivFreeTerm attribute), 763  
 arg\_shapes (sfepy.terms.terms\_diffusion.ConvectVGradSTerm attribute), 763  
 arg\_shapes (sfepy.terms.terms\_diffusion.DiffusionCouplingTerm attribute), 764  
 arg\_shapes (sfepy.terms.terms\_diffusion.DiffusionRTerm attribute), 764  
 arg\_shapes (sfepy.terms.terms\_diffusion.DiffusionTerm attribute), 765  
 arg\_shapes (sfepy.terms.terms\_diffusion.DiffusionVelocityTerm attribute), 765  
 arg\_shapes (sfepy.terms.terms\_diffusion.LaplaceTerm attribute), 766  
 arg\_shapes (sfepy.terms.terms\_diffusion.SDDiffusionTerm attribute), 766  
 arg\_shapes (sfepy.terms.terms\_diffusion.SurfaceFluxOperatorTerm attribute), 767  
 arg\_shapes (sfepy.terms.terms\_diffusion.SurfaceFluxTerm attribute), 768  
 arg\_shapes (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 768  
 arg\_shapes (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 769  
 arg\_shapes (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 770  
 arg\_shapes (sfepy.terms.terms\_dot.DotSProductVolumeOperatorWETHTerm attribute), 770  
 arg\_shapes (sfepy.terms.terms\_dot.DotSProductVolumeTerm attribute), 771  
 arg\_shapes (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm attribute), 771

arg\_shapes (sfepy.terms.terms\_dot.ScalarDotMGradScalarTermarg\_shapes (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm  
     attribute), 772  
 arg\_shapes (sfepy.terms.terms\_dot.VectorDotGradScalarTermarg\_shapes (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm  
     attribute), 773  
 arg\_shapes (sfepy.terms.terms\_dot.VectorDotScalarTermarg\_shapes (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm  
     attribute), 774  
 arg\_shapes (sfepy.terms.terms\_elastic.CauchyStrainTermarg\_shapes (sfepy.terms.terms\_membrane.TLMembraneTerm  
     attribute), 775  
 arg\_shapes (sfepy.terms.terms\_elastic.CauchyStressETHTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.ConvectTerm  
     attribute), 776  
 arg\_shapes (sfepy.terms.terms\_elastic.CauchyStressTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.DivGradTerm  
     attribute), 777  
 arg\_shapes (sfepy.terms.terms\_elastic.CauchyStressTHTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.DivOperatorTerm  
     attribute), 776  
 arg\_shapes (sfepy.terms.terms\_elastic.ElasticWaveCauchyTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.DivTerm  
     attribute), 778  
 arg\_shapes (sfepy.terms.terms\_elastic.ElasticWaveTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.GradDivStabilizationTerm  
     attribute), 778  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearElasticETHTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.GradTerm  
     attribute), 779  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearElasticIsotropicTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.LinearConvect2Term  
     attribute), 779  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearElasticTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.LinearConvectTerm  
     attribute), 781  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearElasticTHTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.PSPGCStabilizationTerm  
     attribute), 780  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearPrestressTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.StokesTerm  
     attribute), 781  
 arg\_shapes (sfepy.terms.terms\_elastic.LinearStrainFiberTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm  
     attribute), 782  
 arg\_shapes (sfepy.terms.terms\_elastic.NonsymElasticTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.StokesWaveTerm  
     attribute), 782  
 arg\_shapes (sfepy.terms.terms\_elastic.SDLinearElasticTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.SUPGCStabilizationTerm  
     attribute), 783  
 arg\_shapes (sfepy.terms.terms\_electric.ElectricSourceTermarg\_shapes (sfepy.terms.terms\_navier\_stokes.SUPGPStabilizationTerm  
     attribute), 784  
 arg\_shapes (sfepy.terms.terms\_fibres.FibresActiveTLTermarg\_shapes (sfepy.terms.terms\_piezo.PiezoCouplingTerm  
     attribute), 784  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientsTermarg\_shapes (sfepy.terms.terms\_piezo.PiezoStrainTerm  
     attribute), 785  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBabushkaTermarg\_shapes (sfepy.terms.terms\_piezo.PiezoStressTerm  
     attribute), 786  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTILTTermarg\_shapes (sfepy.terms.terms\_point.ConcentratedPointLoadTerm  
     attribute), 787  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTermarg\_shapes (sfepy.terms.terms\_point.LinearPointSpringTerm  
     attribute), 788  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTILTTermarg\_shapes (sfepy.terms.terms\_shells.Shell10XTerm  
     attribute), 790  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTILTTermarg\_shapes (sfepy.terms.terms\_surface.ContactPlaneTerm  
     attribute), 790  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTILTTermarg\_shapes (sfepy.terms.terms\_surface.ContactSphereTerm  
     attribute), 791  
 arg\_shapes (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTermarg\_shapes (sfepy.terms.terms\_surface.LinearTractionTerm  
     attribute), 791

arg\_shapes (sfepy.terms.terms\_surface.SDSufaceIntegrateTerm\_types (sfepy.terms.terms\_basic.IntegrateVolumeTerm attribute), 813  
 arg\_shapes (sfepy.terms.terms\_surface.SufaceNormalDotTerm\_types (sfepy.terms.terms\_basic.SumNodalValuesTerm attribute), 755  
 arg\_shapes (sfepy.terms.terms\_surface.SurfaceJumpTerm arg\_types (sfepy.terms.terms\_basic.SurfaceMomentTerm attribute), 814  
 arg\_shapes (sfepy.terms.terms\_volume.LinearVolumeForceTerm\_types (sfepy.terms.terms\_basic.SurfaceTerm attribute), 815  
 arg\_types (sfepy.terms.terms.Term attribute), 738  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvect1Term attribute), 756  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvect2Term attribute), 742  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.AdjDivGradTerm attribute), 743  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMinGradTerm attribute), 758  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDPressDiffTerm attribute), 758  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDPressT0m attribute), 744  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDPressT0m attribute), 744  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDConvectTerm attribute), 745  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDDivGradTerm attribute), 745  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDDivTerm attribute), 746  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDDotVolumeTerm attribute), 746  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDGradDivStabilizationTerm attribute), 747  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGCStabilizationTerm attribute), 748  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGPStabilizationTerm attribute), 748  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SDSUPGCStabilizationTerm attribute), 749  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SUPGCAdjStabilizationTerm attribute), 750  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj1StabilizationTerm attribute), 750  
 arg\_types (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj2StabilizationTerm attribute), 751  
 arg\_types (sfepy.terms.terms\_basic.IntegrateSurfaceMatTerm attribute), 751  
 arg\_types (sfepy.terms.terms\_basic.IntegrateSurfaceOperatorTerm attribute), 752  
 arg\_types (sfepy.terms.terms\_basic.IntegrateSurfaceTerm attribute), 753  
 arg\_types (sfepy.terms.terms\_basic.IntegrateVolumeMatTerm attribute), 753  
 arg\_types (sfepy.terms.terms\_basic.IntegrateVolumeOperatorTerm attribute), 754  
 arg\_types (sfepy.terms.terms\_basic.SurfaceMomentTerm attribute), 755  
 arg\_types (sfepy.terms.terms\_basic.SurfaceTerm attribute), 756  
 arg\_types (sfepy.terms.terms\_basic.VolumeSurfaceTerm attribute), 756  
 arg\_types (sfepy.terms.terms\_basic.VolumeTerm attribute), 756  
 arg\_types (sfepy.terms.terms\_basic.ZeroTerm attribute), 757  
 arg\_types (sfepy.terms.terms\_biot.BiotETHTerm attribute), 758  
 arg\_types (sfepy.terms.terms\_biot.BiotStressTerm attribute), 758  
 arg\_types (sfepy.terms.terms\_biot.BiotTerm attribute), 758  
 arg\_types (sfepy.terms.terms\_biot.BiotTHTerm attribute), 759  
 arg\_types (sfepy.terms.terms\_constraints.NonPenetrationPenaltyTerm attribute), 760  
 arg\_types (sfepy.terms.terms\_contact.ContactTerm attribute), 761  
 arg\_types (sfepy.terms.terms\_diffusion.AdvectDivFreeTerm attribute), 762  
 arg\_types (sfepy.terms.terms\_diffusion.ConvectVGradSTerm attribute), 763  
 arg\_types (sfepy.terms.terms\_diffusion.DiffusionCouplingTerm attribute), 763  
 arg\_types (sfepy.terms.terms\_diffusion.DiffusionRTerm attribute), 764  
 arg\_types (sfepy.terms.terms\_diffusion.DiffusionTerm attribute), 764  
 arg\_types (sfepy.terms.terms\_diffusion.DiffusionVelocityTerm attribute), 765  
 arg\_types (sfepy.terms.terms\_diffusion.LaplaceTerm attribute), 765  
 arg\_types (sfepy.terms.terms\_diffusion.SDDiffusionTerm attribute), 766  
 arg\_types (sfepy.terms.terms\_diffusion.SurfaceFluxOperatorTerm attribute), 766  
 arg\_types (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 767  
 arg\_types (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 768  
 arg\_types (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 769  
 arg\_types (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 770

arg\_types (sfepy.terms.terms\_dot.DotSProductVolumeOperator.arg\_types (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm attribute), 770  
attribute), 790

arg\_types (sfepy.terms.terms\_dot.DotSProductVolumeOperator.arg\_types (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 771  
attribute), 790

arg\_types (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm.arg\_types (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm attribute), 771  
attribute), 791

arg\_types (sfepy.terms.terms\_dot.ScalarDotMGradScalarTerm.arg\_types (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 772  
attribute), 792

arg\_types (sfepy.terms.terms\_dot.VectorDotGradScalarTerm.arg\_types (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 773  
attribute), 793

arg\_types (sfepy.terms.terms\_dot.VectorDotScalarTerm.arg\_types (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 774  
attribute), 793

arg\_types (sfepy.terms.terms\_elastic.CauchyStrainSTerm.arg\_types (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 774  
attribute), 795

arg\_types (sfepy.terms.terms\_elastic.CauchyStrainTerm.arg\_types (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 775  
attribute), 795

arg\_types (sfepy.terms.terms\_elastic.CauchyStressETHTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.ConvectTerm attribute), 776  
attribute), 796

arg\_types (sfepy.terms.terms\_elastic.CauchyStressTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.DivGradTerm attribute), 777  
attribute), 797

arg\_types (sfepy.terms.terms\_elastic.CauchyStressTHTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.DivOperatorTerm attribute), 776  
attribute), 797

arg\_types (sfepy.terms.terms\_elastic.ElasticWaveCauchyTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.DivTerm attribute), 778  
attribute), 798

arg\_types (sfepy.terms.terms\_elastic.ElasticWaveTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.GradDivStabilizationTerm attribute), 778  
attribute), 799

arg\_types (sfepy.terms.terms\_elastic.LinearElasticETHTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.GradTerm attribute), 779  
attribute), 799

arg\_types (sfepy.terms.terms\_elastic.LinearElasticIsotropicTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.LinearConvect2Term attribute), 779  
attribute), 800

arg\_types (sfepy.terms.terms\_elastic.LinearElasticTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.LinearConvectTerm attribute), 781  
attribute), 800

arg\_types (sfepy.terms.terms\_elastic.LinearElasticTHTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.PSPGCStabilizationTerm attribute), 780  
attribute), 801

arg\_types (sfepy.terms.terms\_elastic.LinearPrestressTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.StokesTerm attribute), 781  
attribute), 803

arg\_types (sfepy.terms.terms\_elastic.LinearStrainFiberTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm attribute), 782  
attribute), 803

arg\_types (sfepy.terms.terms\_elastic.NonsymElasticTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.StokesWaveTerm attribute), 782  
attribute), 804

arg\_types (sfepy.terms.terms\_elastic.SDLinElasticTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.SUPGCStabilizationTerm attribute), 783  
attribute), 802

arg\_types (sfepy.terms.terms\_electric.ElectricSourceTerm.arg\_types (sfepy.terms.terms\_navier\_stokes.SUPGPStabilizationTerm attribute), 784  
attribute), 802

arg\_types (sfepy.terms.terms\_fibres.FibresActiveTLTerm.arg\_types (sfepy.terms.terms\_piezo.PiezoCouplingTerm attribute), 784  
attribute), 806

arg\_types (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientTerm.arg\_types (sfepy.terms.terms\_piezo.PiezoStressTerm attribute), 785  
attribute), 807

arg\_types (sfepy.terms.terms\_hyperelastic\_base.HyperElasticTerm.arg\_types (sfepy.terms.terms\_point.ConcentratedPointLoadTerm attribute), 786  
attribute), 807

arg\_types (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm.arg\_types (sfepy.terms.terms\_point.LinearPointSpringTerm attribute), 787  
attribute), 808

arg\_types (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm.arg\_types (sfepy.terms.terms\_shells.Shell10XTerm attribute), 788  
attribute), 809

arg\_types (sfepy.terms.terms\_surface.ContactPlaneTerm attribute), 811  
 arg\_types (sfepy.terms.terms\_surface.ContactSphereTerm attribute), 812  
 arg\_types (sfepy.terms.terms\_surface.LinearTractionTerm attribute), 812  
 arg\_types (sfepy.terms.terms\_surface.SDSufaceIntegrateTerm attribute), 813  
 arg\_types (sfepy.terms.terms\_surface.SurfaceNormalDotTerm attribute), 813  
 arg\_types (sfepy.terms.terms\_surface.SurfaceJumpTerm attribute), 814  
 arg\_types (sfepy.terms.terms\_volume.LinearVolumeForceTerm attribute), 815  
 argsort\_rows() (in module sfepy.linalg.utils), 652  
 arpack\_eigs() (in module sfepy.linalg.eigen), 647  
 as\_dict() (sfepy.base.base.Container method), 516  
 as\_float\_or\_complex() (in module sfepy.base.base), 518  
 assemble1d() (in module sfepy.linalg.utils), 652  
 assemble\_by\_blocks() (in module sfepy.discrete.evaluate), 544  
 assemble\_contact\_residual\_and\_stiffness() (in module sfepy.mechanics.extmods.ccontres), 665  
 assemble\_matrix() (in module sfepy.discrete.common.extmods.assemble), 579  
 assemble\_matrix\_complex() (in module sfepy.discrete.common.extmods.assemble), 579  
 assemble\_to() (sfepy.terms.terms.Term method), 738  
 assemble\_vector() (in module sfepy.discrete.common.extmods.assemble), 579  
 assemble\_vector\_complex() (in module sfepy.discrete.common.extmods.assemble), 579  
 assert\_() (in module sfepy.base.base), 518  
 assert\_equal() (sfepy.base.testing.TestCommon static method), 536  
 assign\_args() (sfepy.terms.terms.Term method), 738  
 assign\_args() (sfepy.terms.terms.Terms method), 741  
 assign\_standard\_hooks() (in module sfepy.applications.pde\_solver\_app), 515  
 aux (sfepy.solvers.ls\_mumps.mumps\_struct\_x attribute), 704  
 average\_qp\_to\_vertices() (sfepy.discrete.fem.fields\_base.SurfaceField method), 598  
 average\_qp\_to\_vertices() (sfepy.discrete.fem.fields\_base.VolumeField method), 598  
 average\_to\_vertices() (sfepy.discrete.fem.fields\_nodal.H1Dibulk\_qp\_to\_vertices), 600  
 average\_vertex\_var\_in\_cells() (in module sfepy.postprocess.time\_history), 683  
 AVSUCDMeshIO (class in sfepy.discrete.fem.meshio), 608  
**B**  
 BandGaps (class in sfepy.homogenization.coefs\_phononic), 636  
 barycentric\_coors() (in module sfepy.linalg.geometry), 648  
 base1d (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), 593  
 basis\_function\_dg() (sfepy.mesh.bspline.BSpline static method), 666  
 basis\_function\_dg0() (sfepy.mesh.bspline.BSpline static method), 666  
 BatheTS (class in sfepy.solvers.ts\_solvers), 715  
 BCNewtonTerm (class in sfepy.terms.terms\_dot), 768  
 BDFMeshIO (class in sfepy.discrete.fem.meshio), 608  
 bf (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
 bf (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 619  
 bfg (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
 bfg (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 619  
 BiotETHTerm (class in sfepy.terms.terms\_biot), 757  
 BiotStressTerm (class in sfepy.terms.terms\_biot), 758  
 BiotTerm (class in sfepy.terms.terms\_biot), 759  
 BiotTHTerm (class in sfepy.terms.terms\_biot), 758  
 block\_solve() (sfepy.discrete.problem.Problem method), 554  
 blockgen (module), 507  
 boundary() (in module sfepy.linalg.sympy\_operators), 652  
 BSpline (class in sfepy.mesh.bspline), 666  
 BSplineSurf (class in sfepy.mesh.bspline), 668  
 bufBN (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 619  
 build\_helpers (module), 505  
 build\_mlab\_pipeline() (sfepy.postprocess.viewer.Viewer method), 685  
 build\_op\_pi() (in module sfepy.homogenization.utils), 646  
 build\_orientation\_map() (in module sfepy.discrete.fem.facets), 594  
 build\_solver\_kwargs() (sfepy.solvers.solvers.Solver method), 712  
 bulk\_from\_lame() (in module sfepy.mechanics.matcoefs), 656  
 bulk\_mumford\_poisson() (in module sfepy.mechanics.matcoefs), 656

BulkActiveTLTerm (class sfepy.terms.terms\_hyperelastic\_tl), 786  
BulkPenaltyTLTerm (class sfepy.terms.terms\_hyperelastic\_tl), 786  
BulkPenaltyULTerm (class sfepy.terms.terms\_hyperelastic\_ul), 792  
BulkPressureTLTerm (class sfepy.terms.terms\_hyperelastic\_tl), 787  
BulkPressureULTerm (class sfepy.terms.terms\_hyperelastic\_ul), 792

**C**

cache (sfepy.discrete.probes.Probe attribute), 551  
cache\_name (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticSurfaceTLElementData attribute), 788  
cache\_name (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLFamilyData attribute), 789  
cache\_name (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticCPElementData attribute), 794  
Cached (class in sfepy.base.ioutils), 527  
calculate() (sfepy.homogenization.engine.HomogenizationWorker static method), 641  
calculate\_req() (sfepy.homogenization.engine.HomogenizationWorker static method), 641  
calculate\_req\_multi() (sfepy.homogenization.engine.HomogenizationWorker static method), 641  
call() (sfepy.applications.evp\_solver\_app.EVPSolverApp method), 515  
call() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 515  
call() (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp method), 631  
call() (sfepy.homogenization.engine.HomogenizationEngine method), 640  
call() (sfepy.homogenization.homogen\_app.HomogenizationApp method), 642  
call\_basic() (sfepy.applications.application.Application method), 514  
call\_empty() (sfepy.postprocess.viewer.Viewer method), 686  
call\_function() (sfepy.terms.terms.Term method), 738  
call\_function() (sfepy.terms.terms\_contact.ContactTerm method), 762  
call\_get\_fargs() (sfepy.terms.terms.Term method), 738  
call\_mlab() (sfepy.postprocess.viewer.Viewer method), 686  
call\_msg (sfepy.discrete.fem.meshio.MeshIO attribute), 611  
call\_parametrized() (sfepy.applications.application.Application method), 514  
canonize\_dof\_names() (sfepy.discrete.conditions.Condition method), 537  
canonize\_dof\_names() (sfepy.discrete.conditions.Conditions method), 537

in canonize\_dof\_names() (sfepy.discrete.conditions.LinearCombinationBC method), 538  
in canonize\_dof\_names() (sfepy.discrete.conditions.PeriodicBC method), 538  
in CauchyStrainSTerm (class in sfepy.terms.terms\_elastic), 774  
in CauchyStrainTerm (class in sfepy.terms.terms\_elastic), 774  
in CauchyStressETHTerm (class in sfepy.terms.terms\_elastic), 775  
CauchyStressTerm (class in sfepy.terms.terms\_elastic), 776  
CauchyStressTHTTerm (class in sfepy.terms.terms\_elastic), 776  
CBasisContext (class in sfepy.discrete.common.extmods.crefcoors), 581  
CGConnectivity (class in sfepy.discrete.common.extmods.cmesh), 579  
cell\_groups (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 579  
celltypes (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 579  
cells (sfepy.discrete.common.region.Region attribute), 589  
cg\_eigs() (in module sfepy.linalg.eigen), 647  
ch (sfepy.discrete.fem.meshio.HDF5MeshIO attribute), 608  
check\_args() (sfepy.terms.terms.Term method), 738  
check\_finiteness() (in module sfepy.terms.utils), 815  
check\_fx() (in module sfepy.linalg.check\_derivatives), 647  
check\_gradient() (in module sfepy.solvers.optimize), 709  
check\_names() (in module sfepy.base.base), 518  
check\_output() (in module test\_install), 507  
check\_shapes() (sfepy.terms.terms.Term method), 738  
check\_tangent\_matrix() (in module sfepy.solvers.nls), 707  
check\_vector\_size() (sfepy.discrete.variables.Variables method), 573  
check\_vfvx() (in module sfepy.linalg.check\_derivatives), 647  
ChristoffelAcousticTensor (class in sfepy.homogenization.coefs\_phononic), 637  
chunk\_micro\_coors() (sfepy.homogenization.engine.HomogenizationWorker static method), 642  
CircleProbe (class in sfepy.discrete.probes), 550  
GLagrangeContext (class in sfepy.discrete.fem.extmods.bases), 593  
classify\_args() (sfepy.terms.terms.Term method), 738  
Clean (class in build\_helpers), 505  
clear\_equations() (sfepy.discrete.problem.Problem method), 555

|                                                                       |                                                         |            |                                                                          |            |
|-----------------------------------------------------------------------|---------------------------------------------------------|------------|--------------------------------------------------------------------------|------------|
| clear_evaluate_cache() (sfepy.discrete.variables.FieldVariable)       | (class in sfepy.homogenization.coefs_base), method),    | <b>569</b> | CoefSym (class in sfepy.homogenization.coefs_base),                      | 633        |
| clear_mappings() (sfepy.discrete.common.fields.Field)                 | (method),                                               | <b>582</b> | CoefSym (class in sfepy.homogenization.coefs_base),                      | 633        |
| clear_qp_base() (sfepy.discrete.fem.fields_base.FEField)              | (method),                                               | <b>595</b> | CoefVolume (class in sfepy.homogenization.engine),                       | <b>640</b> |
| clear_surface_groups() (sfepy.discrete.fem.domain.FEDomain)           | (method),                                               | <b>592</b> | collect_conn_info() (sfepy.discrete.equations.Equation)                  |            |
| ClosingHandler (class in sfepy.postprocess.viewer),                   |                                                         | <b>685</b> | collect_conn_info() (sfepy.discrete.equations.Equations)                 |            |
| CMapping (class in sfepy.discrete.common.extmods.mapping)             | collect_materials() (sfepy.discrete.equations.Equation) |            | method),                                                                 | <b>539</b> |
| 582                                                                   | method),                                                |            | method),                                                                 | <b>539</b> |
| cmem_statistics() (in module sfepy.discrete.common.extmods.cmesh),    |                                                         | <b>581</b> | method),                                                                 | <b>539</b> |
| CMesh (class in sfepy.discrete.common.extmods.cmesh),                 |                                                         | <b>579</b> | collect_term() (in module sfepy.discrete.parse_equations),               | <b>550</b> |
| cntl (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute),             |                                                         | <b>695</b> | collect_variables() (sfepy.discrete.equations.Equation)                  |            |
| cntl (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),           |                                                         | <b>698</b> | method),                                                                 | <b>539</b> |
| cntl (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),           |                                                         | <b>701</b> | collect_variables() (sfepy.discrete.equations.Equations)                 |            |
| CNURBSContext (class in sfepy.discrete.iga.extmods.igac),             |                                                         | <b>619</b> | method),                                                                 | <b>539</b> |
| coef_arrays_to_dicts() (in module sfepy.homogenization.coefficients), |                                                         | <b>632</b> | colsca (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute),              | <b>695</b> |
| CoefDim (class in sfepy.homogenization.coefs_base),                   |                                                         | <b>632</b> | colsca (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),            | <b>698</b> |
| CoefDimDim (class in sfepy.homogenization.coefs_base),                |                                                         | <b>632</b> | colsca (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),            | <b>701</b> |
| CoefDimSym (class in sfepy.homogenization.coefs_base),                |                                                         | <b>632</b> | colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute), | <b>698</b> |
| CoefDummy (class in sfepy.homogenization.coefs_base),                 |                                                         | <b>633</b> | colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute), | <b>701</b> |
| CoefEval (class in sfepy.homogenization.coefs_base),                  |                                                         | <b>633</b> | combine() (in module sfepy.linalg.utils),                                | <b>652</b> |
| CoefExprPar (class in sfepy.homogenization.coefs_base),               |                                                         | <b>633</b> | combine_bezier_extraction() (in module sfepy.discrete.iga.iga),          | <b>623</b> |
| Coefficients (class in sfepy.homogenization.coefficients),            |                                                         | <b>632</b> | combine_scalar_grad() (in module sfepy.homogenization.recovery),         | <b>643</b> |
| CoefN (class in sfepy.homogenization.coefs_base),                     |                                                         | <b>633</b> | comm_fortran (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute),        | <b>695</b> |
| CoefNN (class in sfepy.homogenization.coefs_base),                    |                                                         | <b>633</b> | comm_fortran (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),      | <b>698</b> |
| CoefNone (class in sfepy.homogenization.coefs_base),                  |                                                         | <b>633</b> | comm_fortran (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),      | <b>701</b> |
| CoefNonSym (class in sfepy.homogenization.coefs_base),                |                                                         | <b>633</b> | comm_fortran (sfepy.solvers.ls_mumps.mumps_struct_c_x attribute),        | <b>704</b> |
| CoefNonSymNonSym (class in sfepy.homogenization.coefs_base),          |                                                         | <b>633</b> | compare_vectors() (sfepy.base.testing.TestCommon static method),         | <b>536</b> |
| CoefOne (class in sfepy.homogenization.coefs_base),                   |                                                         | <b>633</b> | compile_flags() (sfepy.config.Config method),                            | <b>514</b> |
| CoefRegion (class in sfepy.homogenization.coefs_perfusion),           |                                                         | <b>636</b> | compose_sparse() (in module sfepy.linalg.sparse),                        | <b>650</b> |
| CoefSum (class in sfepy.homogenization.coefs_base),                   |                                                         | <b>633</b> | CompressibilityULTerm (class in sfepy.terms.terms_hypereastic_ul),       | <b>793</b> |
|                                                                       |                                                         |            | compute_bezier_control() (in module sfepy.discrete.iga.iga),             | <b>623</b> |
|                                                                       |                                                         |            | compute_bezier_extraction() (in module sfepy.discrete.iga.iga),          | <b>624</b> |
|                                                                       |                                                         |            | compute_bezier_extraction_1d() (in module sfepy.discrete.iga.iga),       | <b>624</b> |

compute\_cat\_dim\_dim() (in module sfepy.homogenization.coefs\_phononic), 638  
compute\_cat\_dim\_sym() (in module sfepy.homogenization.coefs\_phononic), 638  
compute\_cat\_sym\_sym() (in module sfepy.homogenization.coefs\_phononic), 638  
compute\_correctors() (sfepy.homogenization.coefs\_base.TCMethod), 655  
compute\_data() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressure), 787  
compute\_data() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressure), 793  
compute\_eigenmomenta() (in module sfepy.homogenization.coefs\_phononic), 638  
compute\_fibre\_strain() (in module sfepy.terms.terms\_fibres), 785  
compute\_jacobian() (sfepy.solvers.semismooth\_newton.SemiNewton), 589  
compute\_mac\_stress\_part() (in module sfepy.homogenization.recovery), 643  
compute\_mean\_decay() (in module sfepy.homogenization.convolutions), 640  
compute\_micro\_u() (in module sfepy.homogenization.recovery), 643  
compute\_nodal\_edge\_dirs() (in module sfepy.discrete.fem.utils), 617  
compute\_nodal\_normals() (in module sfepy.discrete.fem.utils), 617  
compute\_p\_corr\_steady() (in module sfepy.homogenization.recovery), 644  
compute\_p\_corr\_time() (in module sfepy.homogenization.recovery), 644  
compute\_p\_from\_macro() (in module sfepy.homogenization.recovery), 644  
compute\_stress() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase), 786  
compute\_stress\_strain\_u() (in module sfepy.homogenization.recovery), 644  
compute\_tan\_mod() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase), 579  
compute\_u\_corr\_steady() (in module sfepy.homogenization.recovery), 644  
compute\_u\_corr\_time() (in module sfepy.homogenization.recovery), 644  
compute\_u\_from\_macro() (in module sfepy.homogenization.recovery), 644  
ComsolMeshIO (class in sfepy.discrete.fem.meshio), 608  
ConcentratedPointLoadTerm (class in sfepy.terms.terms\_point), 807  
Condition (class in sfepy.discrete.conditions), 537  
Conditions (class in sfepy.discrete.conditions), 537  
Config (class in sfepy.config), 514  
configure\_output() (in module sfepy.base.base), 518  
ConnInfo (class in sfepy.terms.terms), 738  
conn (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 579  
ConstantFunction (class in sfepy.discrete.functions), 546  
ConstantFunctionByRegion (class in sfepy.discrete.functions), 546  
ContactInfo (class in sfepy.terms.terms\_contact), 761  
ContactPlaneTerm (class in sfepy.terms.terms\_surface), 810  
ContactSphere (class in sfepy.mechanics.contact\_bodies), 655  
ContactSphereTerm (class in sfepy.terms.terms\_surface), 811  
ContactTerm (class in sfepy.terms.terms\_contact), 761  
Container (class in sfepy.base.base), 516  
ContactantsNewton (sfepy.discrete.common.region.Region method), 589  
conv\_test() (in module sfepy.solvers.nls), 707  
conv\_test() (in module sfepy.solvers.optimize), 709  
ConvectTerm (class in sfepy.terms.terms\_navier\_stokes), 796  
ConvectVGradSTerm (class in sfepy.terms.terms\_diffusion), 763  
convert\_complex\_output() (in module sfepy.discrete.fem.meshio), 613  
convert\_mesh (module), 507  
ConvolutionKernel (class in sfepy.homogenization.convolutions), 640  
convolve\_field\_scalar() (in module sfepy.homogenization.recovery), 644  
convolve\_field\_sym\_tensor() (in module sfepy.homogenization.recovery), 644  
coo\_is\_symmetric() (in module sfepy.solvers.ls\_mumps), 595  
coor\_to\_sym() (in module sfepy.homogenization.utils), 646  
coors (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 579  
coors (sfepy.discrete.fem.mesh.Mesh attribute), 606  
copy() (sfepy.base.base.Struct method), 517  
copy() (sfepy.discrete.common.region.Region method), 589  
copy() (sfepy.discrete.fem.mesh.Mesh method), 606  
copy() (sfepy.discrete.problem.Problem method), 555  
copy() (sfepy.discrete.state.State method), 567  
CorrDim (class in sfepy.homogenization.coefs\_base), 634  
CorrDimDim (class in sfepy.homogenization.coefs\_base), 634  
CorrEqPar (class in sfepy.homogenization.coefs\_base), 634  
CorrEval (class in sfepy.homogenization.coefs\_base), 634  
CorrMiniApp (class in sfepy.homogenization.coefs\_base), 634

CorrN (class in sfepy.homogenization.coefs\_base), 634  
 CorrNN (class in sfepy.homogenization.coefs\_base), 634  
 CorrOne (class in sfepy.homogenization.coefs\_base), 634  
 CorrRegion (class in sfepy.homogenization.coefs\_perfusion),  
     636  
 CorrSetBCS (class in sfepy.homogenization.coefs\_base),  
     634  
 CorrSolution (class in sfepy.homogenization.coefs\_base),  
     634  
 count (sfepy.base.log.Log attribute), 531  
 cprint() (sfepy.discrete.common.extmods.cmesh.CConnectivity  
     method), 579  
 cprint() (sfepy.discrete.common.extmods.cmesh.CMesh  
     method), 579  
 cprint() (sfepy.discrete.common.extmods.mappings.CMapping  
     method), 582  
 cprint() (sfepy.discrete.fem.extmods.bases.CLagrangeContext  
     method), 593  
 cprint() (sfepy.discrete.iga.extmods.igac.CNURBSContext  
     method), 620  
 create\_adof\_conn() (in module sfepy.discrete.variables),  
     575  
 create\_adof\_connns() (in module sfepy.discrete.variables),  
     575  
 create\_arg\_parser() (in module sfepy.terms.terms), 741  
 create\_basis\_context() (sfepy.discrete.fem.fields\_hierarchic.H1HierarchicalFields  
     method), 600  
 create\_basis\_context() (sfepy.discrete.fem.fields\_nodal.H1NodalMiximethod), 595  
     method), 601  
 create\_basis\_context() (sfepy.discrete.iga.fields.IGField  
     method), 621  
 create\_bnf() (in module sfepy.base.parse\_conf), 534  
 create\_bnf() (in module sfepy.discrete.parse\_equations),  
     550  
 create\_bnf() (in module sfepy.discrete.parse\_regions),  
     550  
 create\_boundary\_qp() (in module sfepy.discrete.iga.iga),  
     624  
 create\_bqp() (sfepy.discrete.fem.fields\_base.FEField  
     method), 595  
 create\_conn\_graph() (sfepy.discrete.fem.mesh.Mesh  
     method), 606  
 create\_connectivity() (in module sfepy.discrete.iga.iga),  
     624  
 create\_connectivity\_1d() (in module  
     sfepy.discrete.iga.iga), 625  
 create\_context() (sfepy.discrete.fem.poly\_spaces.LagrangePolySpace  
     method), 614  
 create\_context() (sfepy.discrete.fem.poly\_spaces.LagrangeSimplexBPolySpace  
     method), 614  
 create\_dataset() (sfepy.postprocess.sources.GenericFileSource  
     method), 682  
 create\_drl\_transform() (in module  
     sfepy.mechanics.shell10x), 660  
 create\_elastic\_tensor() (in module  
     sfepy.mechanics.shell10x), 660  
 create\_eval\_mesh() (sfepy.discrete.common.fields.Field  
     method), 583  
 create\_eval\_mesh() (sfepy.discrete.iga.fields.IGField  
     method), 621  
 create\_evaluable() (in module sfepy.discrete.evaluate),  
     544  
 create\_evaluable() (sfepy.discrete.problem.Problem  
     method), 555  
 create\_expression\_output() (in module  
     sfepy.discrete.fem.fields\_base), 599  
 create\_file\_source() (in module  
     sfepy.postprocess.sources), 682  
 create\_from\_igakit() (in module  
     sfepy.discrete.iga.domain\_generators), 618  
 create\_geometry\_elements() (in module  
     sfepy.discrete.fem.geometry\_element), 602  
 create\_ksp() (sfepy.solvers.ls.PETScKrylovSolver  
     method), 691  
 create\_linear\_fe\_mesh() (in module  
     sfepy.discrete.iga.utils), 629  
 create\_local\_bases() (in module  
     sfepy.mechanics.shell10x), 660  
 create\_mapping() (in module  
     sfepy.mechanics.shell10x), 660  
 create\_mapping() (sfepy.discrete.iga.fields.IGField  
     method), 621  
 create\_mapping() (sfepy.discrete.structural.fields.Shell10XField  
     method), 630  
 create\_mass\_matrix() (in module  
     sfepy.discrete.projections), 564  
 create\_materials() (sfepy.discrete.problem.Problem  
     method), 556  
 create\_matrix\_graph() (sfepy.discrete.equations.Equations  
     method), 539  
 create\_mesh() (sfepy.discrete.fem.fields\_base.FEField  
     method), 595  
 create\_mesh() (sfepy.discrete.iga.fields.IGField  
     method), 622  
 create\_mesh\_and\_output() (in module  
     sfepy.discrete.iga.utils), 630  
 create\_mesh\_graph() (in module  
     sfepy.discrete.common.extmods.cmesh),  
         create\_new() (sfepy.discrete.common.extmods.cmesh.CMesh  
             method), 579  
         create\_nlst() (sfepy.solvers.ts\_solvers.GeneralizedAlphaTS  
             method), 716  
         create\_nlst() (sfepy.solvers.ts\_solvers.NewmarkTS  
             method), 717

create\_nlst() (sfepy.solvers.ts\_solvers.VelocityVerletTS method), 718  
create\_nlst1() (sfepy.solvers.ts\_solvers.BatheTS method), 715  
create\_nlst2() (sfepy.solvers.ts\_solvers.BatheTS method), 715  
create\_omega() (in module sfepy.terms.terms\_fibres), 785  
create\_output() (in module sfepy.discrete.fem.linearizer), 605  
create\_output() (sfepy.discrete.fem.fields\_base.FEField method), 596  
create\_output() (sfepy.discrete.iga.fields.IGField method), 622  
create\_output() (sfepy.discrete.structural.fields.Shell10XField method), 630  
create\_output() (sfepy.discrete.variables.FieldVariable method), 569  
create\_output\_dict() (sfepy.discrete.state.State method), 567  
create\_parser() (in module gen\_term\_table), 511  
create\_petsc\_matrix() (sfepy.solvers.ls.PETScKrylovSolver method), 691  
create\_pis() (in module sfepy.homogenization.utils), 646  
create\_problem() (in module extractor), 502  
create\_region() (sfepy.discrete.common.domain.Domain method), 578  
create\_regions() (sfepy.discrete.common.domain.Domain method), 578  
create\_rotation\_ops() (in module sfepy.mechanics.shell10x), 660  
create\_scalar() (in module eval\_ns\_forms), 508  
create\_scalar\_base() (in module eval\_ns\_forms), 508  
create\_scalar\_base\_grad() (in module eval\_ns\_forms), 508  
create\_scalar\_pis() (in module sfepy.homogenization.utils), 646  
create\_scalar\_var\_data() (in module eval\_ns\_forms), 508  
create\_source() (sfepy.postprocess.sources.GenericFileSource method), 682  
create\_source() (sfepy.postprocess.sources.VTKFileSource method), 682  
create\_source() (sfepy.postprocess.sources.VTKSequenceFileSource method), 682  
create\_spb() (sfepy.mesh.splinebox.SplineBox static method), 675  
create\_spb() (sfepy.mesh.splinebox.SplineRegion2D static method), 676  
create\_state() (sfepy.discrete.problem.Problem method), 556  
create\_state\_vector() (sfepy.discrete.equations.Equations method), 540  
create\_state\_vector() (sfepy.discrete.variables.Variables method), 573  
create\_strain\_matrix() (in module sfepy.mechanics.shell10x), 660  
create\_strain\_transform() (in module sfepy.mechanics.shell10x), 660  
create\_stripped\_state\_vector() (sfepy.discrete.equations.Equations method), 540  
create\_stripped\_state\_vector() (sfepy.discrete.variables.Variables method), 573  
create\_subequations() (sfepy.discrete.equations.Equations method), 540  
create\_subproblem() (sfepy.discrete.problem.Problem method), 556  
create\_surface\_facet() (sfepy.discrete.fem.geometry\_element.GeometryElement method), 601  
create\_surface\_group() (sfepy.discrete.fem.domain.FEDomain method), 592  
create\_transformation\_matrix() (in module sfepy.mechanics.membranes), 658  
create\_transformation\_matrix() (in module sfepy.mechanics.shell10x), 660  
create\_ts\_coef() (in module sfepy.homogenization.coefs\_base), 635  
create\_u\_operator() (in module eval\_ns\_forms), 508  
create\_variables() (sfepy.discrete.problem.Problem method), 556  
create\_vector() (in module eval\_ns\_forms), 508  
create\_vector\_base() (in module eval\_ns\_forms), 508  
create\_vector\_base\_grad() (in module eval\_ns\_forms), 508  
create\_vector\_var\_data() (in module eval\_ns\_forms), 508  
cut\_freq\_range() (in module sfepy.homogenization.coefs\_phononic), 638  
cvt\_array\_index() (in module sfepy.base.parse\_conf), 534  
cvt\_cmplx() (in module sfepy.base.parse\_conf), 534  
cvt\_int() (in module sfepy.base.parse\_conf), 534  
cvt\_none() (in module sfepy.base.parse\_conf), 534  
cvt\_real() (in module sfepy.base.parse\_conf), 534  
cw2us() (in module edit\_identifiers), 508  
cycle() (in module sfepy.linalg.utils), 652  
sintragen (module), 507

## D

d\_biot\_div() (in module sfepy.terms.extmods.terms), 815  
d\_diffusion() (in module sfepy.terms.extmods.terms), 815  
d\_div\_grad() (sfepy.terms.terms\_navier\_stokes.DivGradTerm method), 797  
d\_dot() (sfepy.terms.terms\_dot.DotProductVolumeTerm static method), 770  
d\_dot() (sfepy.terms.terms\_dot.VectorDotScalarTerm static method), 774  
d\_eval() (sfepy.terms.terms\_navier\_stokes.StokesTerm static method), 803

|                                                                                               |                                                                                            |        |
|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|--------|
| d_fun() (sfepy.terms.terms_diffusion.DiffusionCoupling static method), 764                    | de_cauchy_strain() (in module sfepy.terms.extmods.terms), 816                              | module |
| d_fun() (sfepy.terms.terms_surface.LinearTractionTerm static method), 812                     | de_cauchy_stress() (in module sfepy.terms.extmods.terms), 816                              | module |
| d_fun() (sfepy.terms.terms_surface.SurfaceNormalDotTerm static method), 813                   | de_he_rtm() (in module sfepy.terms.extmods.terms), 816                                     | module |
| d_laplace() (in module sfepy.terms.extmods.terms), 815                                        | debug() (in module sfepy.base.base), 516, 518                                              | module |
| d_lin_elastic() (in module sfepy.terms.extmods.terms), 815                                    | debug_flags() (sfepy.config.Config method), 514                                            | module |
| d_lin_prestress() (sfepy.terms.terms_elastic.LinearPrestress method), 781                     | debug_on_error() (in module sfepy.base.base), 518                                          | module |
| d_of_nsMinGrad() (in module sfepy.terms.extmods.terms), 815                                   | dec() (in module sfepy.base.ioutils), 528                                                  | module |
| d_of_nsSurfMinDPress() (in module sfepy.terms.extmods.terms), 815                             | Textchunk_reqs_coefs() (sfepy.homogenization.engine.HomogenizationWork static method), 642 | module |
| d_piezo_coupling() (in module sfepy.terms.extmods.terms), 815                                 | default (sfepy.base.goptions.ValidatedDict attribute), 526                                 | module |
| d_sd_convect() (in module sfepy.terms.extmods.terms), 815                                     | default_space_variables() (in module sfepy.linalg.sympy_operators), 652                    | module |
| d_sd_diffusion() (in module sfepy.terms.extmods.terms), 815                                   | deficiency (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute), 695                        | module |
| d_sd_div() (in module sfepy.terms.extmods.terms), 815                                         | deficiency (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute), 698                      | module |
| d_sd_div_grad() (in module sfepy.terms.extmods.terms), 815                                    | deficiency (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute), 701                      | module |
| d_sd_lin_elastic() (in module sfepy.terms.extmods.terms), 815                                 | define_box_regions() (in module sfepy.homogenization.utils), 646                           | module |
| d_sd_st_grad_div() (in module sfepy.terms.extmods.terms), 815                                 | define_control_points() (sfepy.mesh.splinebox.SplineRegion2D static method), 676           | module |
| d_sd_st_pspg_c() (in module sfepy.terms.extmods.terms), 815                                   | define_volume_coef() (sfepy.homogenization.engine.HomogenizationEngine static method), 640 | module |
| d_sd_st_pspg_p() (in module sfepy.terms.extmods.terms), 815                                   | DeformationGradientTerm (class in sfepy.terms.hyperelastic_base), 785                      | module |
| d_sd_st_supg_c() (in module sfepy.terms.extmods.terms), 816                                   | delete_zero_faces() (sfepy.discrete.common.region.Region method), 589                      | module |
| d_sd_volume_dot() (in module sfepy.terms.extmods.terms), 816                                  | DensityVolumeInfo (class in sfepy.homogenization.coefs_phononic), 637                      | module |
| d_surface_flux() (in module sfepy.terms.extmods.terms), 816                                   | describe() (sfepy.discrete.common.extmods.mappings.CMapping method), 582                   | module |
| d_tl_surface_flux() (in module sfepy.terms.extmods.terms), 816                                | describe_deformation() (in module sfepy.mechanics.membranes), 658                          | module |
| d_tl_volume_surface() (in module sfepy.terms.extmods.terms), 816                              | describe_gaps() (in module sfepy.homogenization.coefs_phononic), 638                       | module |
| d_volume_surface() (in module sfepy.terms.extmods.terms), 816                                 | describe_geometry() (in module sfepy.mechanics.membranes), 658                             | module |
| data_names (sfepy.terms.terms_hypreelastic_tl.HyperElasticSurfaceTLFamilyData attribute), 788 | describe_nodes() (sfepy.discrete.fem.poly_spaces.PolySpace method), 615                    | module |
| data_names (sfepy.terms.terms_hypreelastic_tl.HyperElasticTLFamilyData attribute), 789        | description (build_helpers.DoxygenDocs attribute), 505                                     | module |
| data_names (sfepy.terms.terms_hypreelastic_ul.HyperElasticULFamilyData attribute), 794        | description (build_helpers.SphinxHTMLDocs attribute), 505                                  | module |
| data_shapes (sfepy.terms.terms_hypreelastic_base.HyperElasticFamilyData attribute), 786       | det (sfepy.discrete.common.extmods.mappings.CMapping method), 582                          | module |
| DataMarker (class in sfepy.base.ioutils), 527                                                 | detect_band_gaps() (in module sfepy.linalg.utils), 653                                     | module |
| DataSoftLink (class in sfepy.base.ioutils), 527                                               | di_surface_moment() (in module sfepy.terms.extmods.terms), 816                             | module |

|                                                                         |     |                                                                   |                                        |     |
|-------------------------------------------------------------------------|-----|-------------------------------------------------------------------|----------------------------------------|-----|
| dict_extend() (in module sfepy.base.base),                              | 518 | dq_finite_strain_ul()                                             | (in module sfepy.terms.extmods.terms), | 816 |
| dict_from_keys_init() (in module sfepy.base.base),                      | 518 | dq_grad()                                                         | (in module sfepy.terms.extmods.terms), | 816 |
| dict_from_options() (in module sfepy.base.conf),                        | 525 | dq_state_in_qp()                                                  | (in module sfepy.terms.extmods.terms), | 816 |
| dict_from_string() (in module sfepy.base.conf),                         | 525 | dq_tl_finite_strain_surface()                                     | (in module sfepy.terms.extmods.terms), | 816 |
| dict_to_array() (in module sfepy.base.base),                            | 518 | dq_tl_he_stress_bulk()                                            | (in module sfepy.terms.extmods.terms), | 816 |
| dict_to_struct() (in module sfepy.base.base),                           | 518 | dq_tl_he_stress_bulk_active()                                     | (in module sfepy.terms.extmods.terms), | 816 |
| diff_dt() (sfepy.homogenization.convolutions.ConvolutionKernel method), | 640 | dq_tl_he_stress_mooney_rivlin()                                   | (in module sfepy.terms.extmods.terms), | 816 |
| DiffusionCoupling (class in sfepy.terms.terms_diffusion),               | 763 | dq_tl_he_stress_neohookean()                                      | (in module sfepy.terms.extmods.terms), | 816 |
| DiffusionRTerm (class in sfepy.terms.terms_diffusion),                  | 764 | dq_tl_he_tan_mod_bulk()                                           | (in module sfepy.terms.extmods.terms), | 816 |
| DiffusionTerm (class in sfepy.terms.terms_diffusion),                   | 764 | dq_tl_he_tan_mod_bulk_active()                                    | (in module sfepy.terms.extmods.terms), | 816 |
| DiffusionTLTerm (class in sfepy.terms.terms_hyperelastic_tl),           | 787 | dq_tl_he_tan_mod_mooney_rivlin()                                  | (in module sfepy.terms.extmods.terms), | 816 |
| DiffusionVelocityTerm (class in sfepy.terms.terms_diffusion),           | 765 | dq_tl_he_tan_mod_neohookean()                                     | (in module sfepy.terms.extmods.terms), | 816 |
| dim (sfepy.discrete.common.extmods.cmesh.CMesh attribute),              | 580 | dq_tl_stress_bulk_pressure()                                      | (in module sfepy.terms.extmods.terms), | 816 |
| dim (sfepy.discrete.common.extmods.mappings.CMapping attribute),        | 582 | dq_tl_tan_mod_bulk_pressure_u()                                   | (in module sfepy.terms.extmods.terms), | 816 |
| dim2sym() (in module sfepy.mechanics.tensors),                          | 662 | dq_ul_he_stress_bulk()                                            | (in module sfepy.terms.extmods.terms), | 816 |
| div() (in module sfepy.linalg.sympy_operators),                         | 652 | dq_ul_he_stress_mooney_rivlin()                                   | (in module sfepy.terms.extmods.terms), | 816 |
| DivGradTerm (class in sfepy.terms.terms_navier_stokes),                 | 796 | dq_ul_he_stress_neohookean()                                      | (in module sfepy.terms.extmods.terms), | 816 |
| DivOperatorTerm (class in sfepy.terms.terms_navier_stokes),             | 797 | dq_ul_he_tan_mod_bulk()                                           | (in module sfepy.terms.extmods.terms), | 816 |
| DivTerm (class in sfepy.terms.terms_navier_stokes),                     | 798 | dq_ul_he_tan_mod_mooney_rivlin()                                  | (in module sfepy.terms.extmods.terms), | 816 |
| dkeep (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),            | 698 | dq_ul_he_tan_mod_neohookean()                                     | (in module sfepy.terms.extmods.terms), | 816 |
| dkeep (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),            | 701 | dq_ul_stress_bulk_pressure()                                      | (in module sfepy.terms.extmods.terms), | 816 |
| DofInfo (class in sfepy.discrete.common.dof_info),                      | 576 | dq_ul_tan_mod_bulk_pressure_u()                                   | (in module sfepy.terms.extmods.terms), | 816 |
| Domain (class in sfepy.discrete.common.domain),                         | 578 | dR_dx (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute),  | 620                                    |     |
| DomainSpecificPlot (class in sfepy.postprocess.domain_specific),        | 677 | dR_dxi (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), | 620                                    |     |
| dot_sequences() (in module sfepy.linalg.utils),                         | 653 | draw() (sfepy.mesh.bspline.BSpline method),                       | 666                                    |     |
| DotProductSurfaceTerm (class in sfepy.terms.terms_dot),                 | 768 | draw() (sfepy.mesh.bspline.BSplineSurf method),                   | 668                                    |     |
| DotProductVolumeTerm (class in sfepy.terms.terms_dot),                  | 769 | draw_arrow() (in module sfepy.postprocess.plot_facets),           | 679                                    |     |
| DotSProductVolumeOperatorWETHTerm (class in sfepy.terms.terms_dot),     | 770 | draw_basis() (sfepy.mesh.bspline.BSpline method),                 | 667                                    |     |
| DotSProductVolumeOperatorWTHTerm (class in sfepy.terms.terms_dot),      | 770 | draw_data() (in module sfepy.base.log_plotter),                   | 532                                    |     |
| DoxygenDocs (class in build_helpers),                                   | 505 |                                                                   |                                        |     |
| dq_cauchy_strain() (in module sfepy.terms.extmods.terms),               | 816 |                                                                   |                                        |     |
| dq_def_grad() (in module sfepy.terms.extmods.terms),                    | 816 |                                                                   |                                        |     |
| dq_div_vector() (in module sfepy.terms.extmods.terms),                  | 816 |                                                                   |                                        |     |
| dq_finite_strain_tl() (in module sfepy.terms.extmods.terms),            | 816 |                                                                   |                                        |     |

|                       |                                                                 |     |                          |                                        |                                                        |     |
|-----------------------|-----------------------------------------------------------------|-----|--------------------------|----------------------------------------|--------------------------------------------------------|-----|
| dump_to_vtk()         | (in module sfepy.postprocess.time_history),                     | 683 | module                   | dw_st_spg_c()                          | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_adj_convect1()     | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | dw_st_spg_p()                          | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_adj_convect2()     | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | dw_surface_flux()                      | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_biot_div()         | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | dw_surface_ltr()                       | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_biot_grad()        | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | dw_surface_s_v_dot_n()                 | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_convect_v_grad_s() | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | dw_surface_v_dot_n_s()                 | (in module sfepy.terms.extmods.terms),                 | 817 |
| dw_diffusion()        | (in module sfepy.terms.extmods.terms),                          | 817 | dw_tl_diffusion()        | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_diffusion_r()      | (in module sfepy.terms.extmods.terms),                          | 817 | dw_tl_surface_traction() | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_div()              | (in module sfepy.terms.extmods.terms),                          | 817 | dw_tl_volume()           | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_dot()              | (sfepy.terms.terms_dot.DotProductVolumeTerm static method),     | 770 | dw_ul_volume()           | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_dot()              | (sfepy.terms.terms_dot.VectorDotScalarTerm static method),      | 774 | dw_v_dot_grad_s_sw()     | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_electric_source()  | (in module sfepy.terms.extmods.terms),                          | 817 | dw_v_dot_grad_s_vw()     | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_fun()              | (sfepy.terms.terms_diffusion.DiffusionCoupling static method),  | 764 | dw_volume_dot_scalar()   | (in module sfepy.terms.extmods.terms), | 817                                                    |     |
| dw_fun()              | (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm static method), | 771 | dw_volume_dot_vector()   | (in module sfepy.terms.extmods.terms), | 818                                                    |     |
| dw_fun()              | (sfepy.terms.terms_surface.SurfaceNormalDotTerm static method), | 813 | dw_volume_lvf()          | (in module sfepy.terms.extmods.terms), | 818                                                    |     |
| dw_grad()             | (in module sfepy.terms.extmods.terms),                          | 817 |                          |                                        |                                                        |     |
| dw_he_rtm()           | (in module sfepy.terms.extmods.terms),                          | 817 |                          |                                        |                                                        |     |
| dw_laplace()          | (in module sfepy.terms.extmods.terms),                          | 817 |                          |                                        |                                                        |     |
| dw_lin_convect()      | (in module sfepy.terms.extmods.terms),                          | 817 |                          |                                        |                                                        |     |
| dw_lin_elastic()      | (in module sfepy.terms.extmods.terms),                          | 817 |                          |                                        |                                                        |     |
| dw_lin_prestress()    | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edge_oirs                              | (sfepy.discrete.common.extmods.cmesh.CMesh attribute), | 580 |
| dw_lin_strain_fib()   | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | EdgeDirectionOperator                  | (class in sfepy.discrete.fem.lcbc_operators),          | 602 |
| dw_nonsym_elastic()   | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edges                                  | (sfepy.discrete.common.region.Region attribute),       | 589 |
| dw_piezo_coupling()   | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit()                                 | (in module edit_identifiers),                          | 508 |
| dw_st_adj1_spg_p()    | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit()                                 | (sfepy.base.conf.ProblemConf method),                  | 524 |
| dw_st_adj2_spg_p()    | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit_dict_strings()                    | (in module sfepy.base.base),                           | 518 |
| dw_st_adj_spg_c()     | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit_filename()                        | (in module sfepy.base.ioutils),                        | 528 |
| dw_st_grad_div()      | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit_identifiers                       | (module),                                              | 508 |
| dw_st_pspg_c()        | (in module sfepy.terms.extmods.terms),                          | 817 | module                   | edit_tuple_strings()                   | (in module sfepy.base.base),                           | 519 |
|                       |                                                                 |     | module                   | eig()                                  | (in module sfepy.solvers.eigen),                       | 689 |
|                       |                                                                 |     | module                   | Eigenmomenta                           | (class in sfepy.homogenization.coefs_phononic),        | 637 |
|                       |                                                                 |     | module                   | EigenvalueSolver                       | (class in sfepy.solvers.solvers),                      | 711 |
|                       |                                                                 |     | module                   | ElasticConstants                       | (class in sfepy.mechanics.matcoefs),                   | 655 |

## E

|                       |                                                                |     |
|-----------------------|----------------------------------------------------------------|-----|
| e_coors_max           | (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), | 593 |
| e_coors_max           | (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute),     | 620 |
| edge_oirs             | (sfepy.discrete.common.extmods.cmesh.CMesh attribute),         | 580 |
| EdgeDirectionOperator | (class in sfepy.discrete.fem.lcbc_operators),                  | 602 |
| edges                 | (sfepy.discrete.common.region.Region attribute),               | 589 |
| edit()                | (in module edit_identifiers),                                  | 508 |
| edit()                | (sfepy.base.conf.ProblemConf method),                          | 524 |
| edit_dict_strings()   | (in module sfepy.base.base),                                   | 518 |
| edit_filename()       | (in module sfepy.base.ioutils),                                | 528 |
| edit_identifiers      | (module),                                                      | 508 |
| edit_tuple_strings()  | (in module sfepy.base.base),                                   | 519 |
| eig()                 | (in module sfepy.solvers.eigen),                               | 689 |
| Eigenmomenta          | (class in sfepy.homogenization.coefs_phononic),                | 637 |
| EigenvalueSolver      | (class in sfepy.solvers.solvers),                              | 711 |
| ElasticConstants      | (class in sfepy.mechanics.matcoefs),                           | 655 |

ElasticWaveCauchyTerm (class in sfepy.terms.terms\_elastic), 777

ElasticWaveTerm (class in sfepy.terms.terms\_elastic), 778

ElastodynamicsBaseTS (class in sfepy.solvers.ts\_solvers), 715

ElectricSourceTerm (class in sfepy.terms.terms\_electric), 783

elems\_q2t() (in module sfepy.mesh.mesh\_tools), 674

elevate() (sfepy.discrete.iga.domain.NurbsPatch method), 618

elptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695

elptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698

elptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701

eltvar (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695

eltvar (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698

eltvar (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701

enc() (in module sfepy.base.ioutils), 528

encode\_animation() (sfepy.postprocess.viewer.Viewer method), 687

ensure\_path() (in module sfepy.base.ioutils), 528

entities (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 580

Equation (class in sfepy.discrete.equations), 539

equation\_mapping() (sfepy.discrete.variables.FieldVariable method), 569

equation\_mapping() (sfepy.discrete.variables.Variables method), 573

EquationMap (class in sfepy.discrete.common.dof\_info), 577

Equations (class in sfepy.discrete.equations), 539

errclear() (in module sfepy.terms.extmods.terms), 818

EssentialBC (class in sfepy.discrete.conditions), 537

ETHTerm (class in sfepy.terms.terms\_th), 814

eval() (sfepy.mesh.bspline.BSpline method), 667

eval() (sfepy.mesh.bspline.BSplineSurf method), 668

eval\_base() (sfepy.discrete.fem.poly\_spaces.PolySpace method), 615

eval\_basis() (sfepy.mesh.bspline.BSpline method), 667

eval\_bernstein\_basis() (in module sfepy.discrete.iga.iga), 620

eval\_bernstein\_basis() (in module sfepy.discrete.iga.iga), 625

eval\_complex() (in module sfepy.discrete.evaluate\_variable), 546

eval\_complex() (sfepy.terms.terms.Term method), 738

eval\_coor\_expression() (sfepy.base.testing.TestCommon static method), 536

in eval\_equations() (in module sfepy.discrete.evaluate), 545

eval\_equations() (sfepy.discrete.problem.Problem method), 557

eval\_exponential() (in module sfepy.homogenization.convolutions), 640

eval\_function() (sfepy.terms.terms\_membrane.TLMembraneTerm static method), 795

eval\_in\_els\_and\_qp() (in module sfepy.discrete.evaluate), 545

eval\_in\_tp\_coors() (in module sfepy.discrete.iga.extmods.igac), 620

eval\_lobatto1d() (in module sfepy.discrete.fem.extmods.lobatto\_bases), 593

eval\_lobatto\_tensor\_product() (in module sfepy.discrete.fem.extmods.lobatto\_bases), 593

eval\_mapping\_data\_in\_qp() (in module sfepy.discrete.iga.extmods.igac), 620

eval\_mapping\_data\_in\_qp() (in module sfepy.discrete.iga.iga), 625

eval\_membrane\_mooney\_rivlin() (in module sfepy.terms.terms\_membrane), 796

eval\_nodal\_coors() (in module sfepy.discrete.fem.fields\_base), 600

eval\_ns\_forms (module), 508

eval\_nurbs\_basis\_tp() (in module sfepy.discrete.iga.iga), 625

eval\_op\_cells() (sfepy.discrete.common.region.Region method), 589

eval\_op\_edges() (sfepy.discrete.common.region.Region method), 589

eval\_op\_faces() (sfepy.discrete.common.region.Region method), 589

eval\_op\_facets() (sfepy.discrete.common.region.Region method), 589

eval\_op\_vertices() (sfepy.discrete.common.region.Region method), 589

eval\_real() (in module sfepy.discrete.evaluate\_variable), 546

eval\_real() (sfepy.terms.terms.Term method), 738

eval\_real() (sfepy.terms.terms\_contact.ContactTerm method), 762

eval\_real() (sfepy.terms.terms\_th.THTerm method), 814

eval\_residual() (sfepy.discrete.evaluate.Evaluator method), 544

eval\_residuals() (sfepy.discrete.equations.Equations method), 540

eval\_tangent\_matrices() (sfepy.discrete.equations.Equations method), 540

eval\_tangent\_matrix() (sfepy.discrete.evaluate.Evaluator method), 544

eval\_tl\_forms (module), 508

eval\_variable\_in\_qp() (in module sfepy.discrete.iga.extmods.igac), 621  
 eval\_variable\_in\_qp() (in module sfepy.discrete.iga.iga), 626  
 evaluate() (sfepy.discrete.equations.Equation method), 539  
 evaluate() (sfepy.discrete.equations.Equations method), 541  
 evaluate() (sfepy.discrete.fem.extmods.bases.CLagrangeContext method), 593  
 evaluate() (sfepy.discrete.iga.domain.NurbsPatch method), 618  
 evaluate() (sfepy.discrete.iga.extmods.igac.CNURBSContext method), 620  
 evaluate() (sfepy.discrete.problem.Problem method), 557  
 evaluate() (sfepy.discrete.variables.FieldVariable method), 569  
 evaluate() (sfepy.homogenization.coefs\_phononic.AcousticMassTensor method), 636  
 evaluate() (sfepy.homogenization.coefs\_phononic.AppliedLoadTensor method), 636  
 evaluate() (sfepy.mesh.splinebox.SplineBox method), 675  
 evaluate() (sfepy.terms.terms.Term method), 738  
 evaluate\_at() (sfepy.discrete.common.fields.Field method), 583  
 evaluate\_at() (sfepy.discrete.variables.FieldVariable method), 570  
 evaluate\_bfbgm() (sfepy.discrete.common.extmods.mappings.Mapping method), 582  
 evaluate\_contact\_constraints() (in module sfepy.mechanics.extmods.ccontres), 665  
 evaluate\_derivative() (sfepy.mesh.splinebox.SplineBox method), 675  
 evaluate\_in\_rc() (in module sfepy.discrete.common.extmods.crefcoors), 581  
 Evaluator (class in sfepy.discrete.evaluate), 544  
 EVPSolverApp (class in sfepy.applications.evp\_solver\_app), 515  
 expand2d() (in module sfepy.mesh.mesh\_tools), 674  
 expand\_basis() (in module sfepy.discrete.variables), 576  
 expand\_nodes\_to\_dofs() (in module sfepy.discrete.common.dof\_info), 577  
 expand\_nodes\_to\_equations() (in module sfepy.discrete.common.dof\_info), 577  
 expand\_schur() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
 extend() (sfepy.base.base.Container method), 516  
 extend\_cell\_data() (in module sfepy.discrete.fem.utils), 617  
 extend\_dofs() (sfepy.discrete.fem.fields\_base.FEField method), 596  
 extend\_dofs() (sfepy.discrete.fem.fields\_nodal.H1DiscontinuousField method), 600  
 extract\_edges (module), 508  
 extract\_edges() (in module extract\_edges), 508  
 extract\_surface (module), 509  
 extract\_time\_history() (in module sfepy.postprocess.time\_history), 683  
 extract\_times() (in module sfepy.postprocess.time\_history), 683  
 extractor (module), 502

**F**

face\_oris (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 580  
 faces (sfepy.discrete.common.region.Region attribute), 589  
 facet\_oris (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 580  
 facets (sfepy.discrete.common.region.Region attribute), 589  
 factorial() (in module sfepy.discrete.simplex\_cubature), 566  
 family\_data\_names (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 784  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm attribute), 786  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm attribute), 787  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 787  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 788  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm attribute), 789  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm attribute), 789  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm attribute), 790  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 791  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm attribute), 791  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 792  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm attribute), 792  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 793  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 793  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm attribute), 794  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm attribute), 794

family\_data\_names (sfepy.terms.terms\_hyperelastic.ul.VolumeULTerm attribute), 795  
 family\_function() (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticSurfaceTerm family Data module static method), 788  
 family\_function() (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticVolumeTerm family Data module static method), 789  
 family\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm family Data module static method), 793  
 family\_function() (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULTerm family Data module static method), 794  
 family\_name (sfepy.discrete.fem.fields\_hierarchic.H1HierarchicVolumeField family Homogenization module attribute), 600  
 family\_name (sfepy.discrete.fem.fields\_nodal.H1DiscontinuousField attribute), 600  
 family\_name (sfepy.discrete.fem.fields\_nodal.H1NodalSurfaceField attribute), 601  
 family\_name (sfepy.discrete.fem.fields\_nodal.H1NodalVolumeField attribute), 601  
 family\_name (sfepy.discrete.iga.fields.IGField attribute), 622  
 family\_name (sfepy.discrete.structural.fields.Shell10XField attribute), 631  
 FEDomain (class in sfepy.discrete.fem.domain), 592  
 FEField (class in sfepy.discrete.fem.fields\_base), 595  
 FEMapping (class in sfepy.discrete.fem.mappings), 605  
 FESurface (class in sfepy.discrete.fem.fe\_surface), 594  
 FibresActiveTLTerm (class in sfepy.terms.terms\_fibres), 784  
 Field (class in sfepy.discrete.common.fields), 582  
 fields\_from\_conf() (in module sfepy.discrete.common.fields), 584  
 FieldVariable (class in sfepy.discrete.variables), 568  
 file\_changed() (sfepy.postprocess.sources.FileSource method), 681  
 file\_changed() (sfepy.postprocess.sources.GenericFileSource method), 682  
 file\_changed() (sfepy.postprocess.sources.GenericSequenceFileSource method), 682  
 FileSource (class in sfepy.postprocess.sources), 681  
 fill() (sfepy.discrete.state.State method), 567  
 finalize() (sfepy.discrete.common.region.Region method), 589  
 finalize() (sfepy.discrete.fem.lcbc\_operators.LCBOperator method), 603  
 finalize\_options() (build\_helpers.NoOptionsDocs method), 505  
 find() (sfepy.base.base.OneTypeList method), 516  
 find\_facet\_substitutions() (in module sfepy.discrete.fem.refine\_hanging), 616  
 find\_level\_interface() (in module sfepy.discrete.fem.refine\_hanging), 616  
 find\_map() (in module sfepy.discrete.fem.mesh), 607  
 find\_ref\_coors() (in module sfepy.discrete.common.extmods.crefcoors),
 find\_ref\_coors\_convex() (in module sfepy.TlFamilyData mon.extmods.crefcoors), 582  
 find\_ts() (sfepy.mesh.splinebox.SplineRegion2D module), 519  
 find\_zero() (in module sfepy.Homogenization.coefs\_phononic), 639  
 fit\_exponential() (in module sfepy.Homogenization.convolutions), 640  
 fix\_double\_nodes() (in module sfepy.discrete.fem.mesh), 607  
 fix\_eig\_range() (sfepy.homogenization.coefs\_phononic.BandGaps module), 648  
 fix\_element\_orientation() (sfepy.discrete.fem.domain.FEDomain method), 592  
 flag\_points\_in\_polygon2d() (in module sfepy.linalg.geometry), 648  
 flush() (run\_tests.OutputFilter method), 504  
 FMinSteepestDescent (class in sfepy.solvers.optimize), 708  
 font\_size() (in module sfepy.base.plotutils), 535  
 for\_format() (sfepy.discrete.fem.meshio.MeshIO static method), 611  
 format (sfepy.discrete.fem.meshio.AbaqusMeshIO attribute), 608  
 format (sfepy.discrete.fem.meshio.ANSYSDBMeshIO attribute), 607  
 format (sfepy.discrete.fem.meshio.AVSUCDMeshIO attribute), 608  
 format (sfepy.discrete.fem.meshio.BDFMeshIO attribute), 608  
 format (sfepy.discrete.fem.meshio.ComsolMeshIO attribute), 608  
 format (sfepy.discrete.fem.meshio.HDF5MeshIO attribute), 608  
 format (sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO attribute), 609  
 format (sfepy.discrete.fem.meshio.MeditMeshIO attribute), 610  
 format (sfepy.discrete.fem.meshio.MEDMeshIO attribute), 609  
 format (sfepy.discrete.fem.meshio.Mesh3DMeshIO attribute), 610  
 format (sfepy.discrete.fem.meshio.MeshIO attribute), 611  
 format (sfepy.discrete.fem.meshio.Msh2MeshIO attribute), 611  
 format (sfepy.discrete.fem.meshio.NEUMeshIO attribute), 612  
 format (sfepy.discrete.fem.meshio.TetgenMeshIO attribute), 612

|                     |                                                                |     |                                                                                                 |
|---------------------|----------------------------------------------------------------|-----|-------------------------------------------------------------------------------------------------|
| format              | (sfepy.discrete.fem.meshio.UserMeshIO<br>tribute), 612         | at- | from_desc() (sfepy.terms.terms.Terms static method), 741                                        |
| format              | (sfepy.discrete.fem.meshio.VTKMeshIO<br>tribute), 613          | at- | from_dict() (sfepy.base.conf.ProblemConf static<br>method), 524                                 |
| format_next()       | (in module gen_term_table), 511                                |     | from_facets() (sfepy.discrete.common.region.Region<br>static method), 590                       |
| format_next()       | (in module sfepy.solvers.solvers), 713                         |     | from_file() (sfepy.base.conf.ProblemConf static method),<br>524                                 |
| format_str()        | (sfepy.discrete.fem.meshio.BDFMeshIO<br>static method), 608    |     | from_file() (sfepy.discrete.fem.mesh.Mesh static<br>method), 606                                |
| free_connectivity() | (sfepy.discrete.common.extmods.cmesh<br>method), 580           |     | from_file() (sfepy.discrete.iga.domain.IGDomain static<br>method), 618                          |
| from_args()         | (sfepy.discrete.common.fields.Field static<br>method), 584     |     | from_file_and_options() (sfepy.base.conf.ProblemConf<br>static method), 525                     |
| from_args()         | (sfepy.discrete.common.mappings.Mapping<br>static method), 587 |     | from_file_hdf5() (sfepy.discrete.fem.history.Histories<br>static method), 602                   |
| from_array()        | (sfepy.linalg.utils.MatrixAction static<br>method), 652        |     | from_file_hdf5() (sfepy.homogenization.coefficients.Coefficients<br>static method), 632         |
| from_cells()        | (sfepy.discrete.common.region.Region static<br>method), 589    |     | from_function() (sfepy.linalg.utils.MatrixAction static<br>method), 652                         |
| from_conf()         | (sfepy.base.log.Log static method), 531                        |     | from_gmsh_file() (sfepy.mesh.geom_tools.geometry<br>static method), 670                         |
| from_conf()         | (sfepy.discrete.common.fields.Field static<br>method), 584     |     | from_module() (sfepy.base.conf.ProblemConf static<br>method), 525                               |
| from_conf()         | (sfepy.discrete.conditions.Conditions static<br>method), 537   |     | from_region() (sfepy.discrete.fem.mesh.Mesh static<br>method), 606                              |
| from_conf()         | (sfepy.discrete.equations.Equations static<br>method), 541     |     | from_sequence() (sfepy.discrete.fem.history.History<br>static method), 602                      |
| from_conf()         | (sfepy.discrete.functions.Functions static<br>method), 547     |     | from_table() (sfepy.discrete.quadratures.QuadraturePoints<br>static method), 565                |
| from_conf()         | (sfepy.discrete.integrals.Integrals static<br>method), 547     |     | from_variables() (sfepy.discrete.state.State static<br>method), 567                             |
| from_conf()         | (sfepy.discrete.materials.Material static<br>method), 548      |     | from_vertices() (sfepy.discrete.common.region.Region<br>static method), 590                     |
| from_conf()         | (sfepy.discrete.materials.Materials static<br>method), 549     |     | Function (class in sfepy.discrete.functions), 546                                               |
| from_conf()         | (sfepy.discrete.problem.Problem static<br>method), 557         |     | function() (sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term<br>static method), 742          |
| from_conf()         | (sfepy.discrete.variables.Variable static<br>method), 572      |     | function() (sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term<br>static method), 742          |
| from_conf()         | (sfepy.discrete.variables.Variables static<br>method), 574     |     | function() (sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm<br>static method), 743           |
| from_conf()         | (sfepy.solvers.ts.TimeStepper static method),<br>713           |     | function() (sfepy.terms.terms_adj_navier_stokes.NSOFMinGradTerm<br>static method), 743          |
| from_conf()         | (sfepy.solvers.ts.VariableTimeStepper static<br>method), 713   |     | function() (sfepy.terms.terms_adj_navier_stokes.NSOSurfMinDPressTerm<br>static method), 744     |
| from_conf_file()    | (sfepy.discrete.problem.Problem static<br>method), 557         |     | function() (sfepy.terms.terms_adj_navier_stokes.SDConvectTerm<br>static method), 745            |
| from_data()         | (sfepy.discrete.common.extmods.cmesh.CMesh<br>method), 580     |     | function() (sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm<br>static method), 745            |
| from_data()         | (sfepy.discrete.fem.mesh.Mesh static<br>method), 606           |     | function() (sfepy.terms.terms_adj_navier_stokes.SDDivTerm<br>static method), 746                |
| from_data()         | (sfepy.discrete.iga.domain.IGDomain static<br>method), 618     |     | function() (sfepy.terms.terms_adj_navier_stokes.SDDotVolumeTerm<br>static method), 746          |
| from_desc()         | (sfepy.discrete.equations.Equation static<br>method), 539      |     | function() (sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTe<br>static method), 747 |
| from_desc()         | (sfepy.terms.terms.Term static method), 739                    |     |                                                                                                 |

```
function() (sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilization(sfepy.terms.terms_dot.ScalarDotMGradScalarTerm  
    static method), 748  
function() (sfepy.terms.terms_adj_navier_stokes.SDPSPGPStabilization(sfepy.terms.terms_elastic.CauchyStrainTerm  
    static method), 748  
function() (sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilization(sfepy.terms.terms_elastic.CauchyStressTerm  
    static method), 749  
function() (sfepy.terms.terms_adj_navier_stokes.SUPGCAdjStabilization(sfepy.terms.terms_elastic.ElasticWaveCauchyTerm  
    static method), 750  
function() (sfepy.terms.terms_adj_navier_stokes.SUPGPAdjStabilization(sfepy.terms.terms_elastic.ElasticWaveTerm  
    static method), 750  
function() (sfepy.terms.terms_adj_navier_stokes.SUPGPAdjStabilization(sfepy.terms.terms_elastic.LinearElasticETHTerm  
    static method), 751  
function() (sfepy.terms.terms_basic.IntegrateSurfaceTerm function() (sfepy.terms.terms_elastic.LinearElasticTHTerm  
    static method), 753  
function() (sfepy.terms.terms_basic.IntegrateVolumeMatTerfun function() (sfepy.terms.terms_elastic.LinearStrainFiberTerm  
    static method), 753  
function() (sfepy.terms.terms_basic.IntegrateVolumeOperatfun function() (sfepy.terms.terms_elastic.SDLinElasticTerm  
    static method), 754  
function() (sfepy.terms.terms_basic.IntegrateVolumeTerm function() (sfepy.terms.terms_electric.ElectricSourceTerm  
    static method), 754  
function() (sfepy.terms.terms_basic.SumNodalValuesTerm function() (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm  
    static method), 755  
function() (sfepy.terms.terms_basic.SurfaceMomentTerm function() (sfepy.terms.terms_hyperelastic_base.HyperElasticBase  
    static method), 755  
function() (sfepy.terms.terms_basic.VolumeSurfaceTerm function() (sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm  
    static method), 756  
function() (sfepy.terms.terms_basic.VolumeTerm static function() (sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm  
    method), 757  
function() (sfepy.terms.terms_basic.ZeroTerm static function() (sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm  
    method), 757  
function() (sfepy.terms.terms_biot.BiotStressTerm static function() (sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm  
    method), 758  
function() (sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm) (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm  
    static method), 760  
function() (sfepy.terms.terms_constraints.NonPenetrationTefunction() (sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm  
    static method), 761  
function() (sfepy.terms.terms_contact.ContactTerm static function() (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm  
    method), 762  
function() (sfepy.terms.terms_diffusion.ConvectVGradSTerfunction() (sfepy.terms.terms_membrane.TLMembraneTerm  
    static method), 763  
function() (sfepy.terms.terms_diffusion.DiffusionRTerm function() (sfepy.terms.terms_navier_stokes.ConvectTerm  
    static method), 764  
function() (sfepy.terms.terms_diffusion.DiffusionVelocityTefunction() (sfepy.terms.terms_navier_stokes.DivGradTerm  
    static method), 765  
function() (sfepy.terms.terms_diffusion.SDDiffusionTerm function() (sfepy.terms.terms_navier_stokes.DivOperatorTerm  
    static method), 767  
function() (sfepy.terms.terms_diffusion.SurfaceFluxOperatfun function() (sfepy.terms.terms_navier_stokes.DivTerm  
    static method), 767  
function() (sfepy.terms.terms_diffusion.SurfaceFluxTerm function() (sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm  
    static method), 768  
function() (sfepy.terms.terms_dot.DotSProductVolumeOperatfun WETHTe(sfepy.terms.terms_navier_stokes.GradTerm  
    static method), 770  
function() (sfepy.terms.terms_dot.DotSProductVolumeOperatfun WHTTe(sfepy.terms.terms_navier_stokes.LinearConvect2Term  
    static method), 771  
function() (sfepy.terms.terms_dot.DotSProductVolumeOperatfun WHTTe(sfepy.terms.terms_navier_stokes.LinearConvect2Term  
    static method), 800
```

function() (sfepy.terms.terms\_navier\_stokes.LinearConvectTerm mesh\_from\_voxels() (in module sfepy.mesh.mesh\_generators), 673  
 static method), 800

function() (sfepy.terms.terms\_navier\_stokes.PSPGCStabilizationTerm mesh\_prev (module), 511  
 static method), 801

function() (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm method), 681  
 static method), 804

function() (sfepy.terms.terms\_navier\_stokes.StokesWaveTerm gen\_mesh\_probe\_png() (sfepy.postprocess.probes\_vtk.Probe method), 673  
 static method), 804

function() (sfepy.terms.terms\_navier\_stokes.SUPGCStabilizationTerm gen\_misc\_mesh() (in module sfepy.mesh.mesh\_generators), 673  
 static method), 802

function() (sfepy.terms.terms\_navier\_stokes.SUPGPStabilizationTerm gen\_multi\_vec\_packing() (in module sfepy.solvers.ts\_solvers), 718  
 static method), 802

function() (sfepy.terms.terms\_piezo.PiezoStressTerm gen\_patch\_block\_domain() (in module sfepy.solvers.ts\_solvers), 619  
 static method), 802

function() (sfepy.terms.terms\_piezo.PiezoStressTerm gen\_points() (sfepy.discrete.probes.RayProbe method), 552  
 static method), 807

function() (sfepy.terms.terms\_point.ConcentratedPointLoad gen\_release\_notes (module), 511  
 static method), 807

function() (sfepy.terms.terms\_point.LinearPointSpringTerm gen\_shot() (in module gen\_mesh\_prev), 511  
 static method), 808

function() (sfepy.terms.terms\_shells.Shell10XTerm gen\_solver\_table (module), 511  
 static method), 809

function() (sfepy.terms.terms\_surface.ContactPlaneTerm gen\_solver\_table() (in module gen\_solver\_table), 511  
 static method), 811

function() (sfepy.terms.terms\_surface.ContactSphereTerm gen\_term\_table (module), 511  
 static method), 812

function() (sfepy.terms.terms\_surface.SDSufaceIntegrateTerm gen\_term\_table() (in module gen\_term\_table), 511  
 static method), 813

function() (sfepy.terms.terms\_surface.SurfaceJumpTerm gen\_tiled\_mesh() (in module sfepy.mesh.mesh\_generators), 673  
 static method), 814

function() (sfepy.terms.terms\_volume.LinearVolumeForceTerm generate\_gallery\_html() (in module gen\_gallery), 510  
 static method), 815

function\_weak() (sfepy.terms.terms\_contact.ContactTerm generate\_images() (in module gen\_gallery), 510  
 static method), 762

Functions (class in sfepy.discrete.functions), 546

## G

geme\_mulAVSB3py() (in module sfepy.discrete.common.extmods.\_geommech), 578

gen\_block\_mesh() (in module sfepy.mesh.mesh\_generators), 671

gen\_cp\_idxes() (sfepy.mesh.splinebox.SplineBox static method), 675

gen\_cylinder\_mesh() (in module sfepy.mesh.mesh\_generators), 672

gen\_extended\_block\_mesh() (in module sfepy.mesh.mesh\_generators), 672

gen\_gallery (module), 509

gen\_iga\_patch (module), 510

gen\_lobatto() (in module gen\_lobatto1d\_c), 510

gen\_lobatto1d\_c (module), 510

gen\_mesh\_from\_geom() (in module sfepy.mesh.mesh\_generators), 673

gen\_mesh\_from\_string() (in module sfepy.mesh.mesh\_generators), 673

generate\_a\_pyrex\_source() (in module build\_helpers), 506

generate\_decreasing\_nonnegative\_tuples\_summing\_to() (in module sfepy.discrete.simplex\_cubature), 566

GeneralizedAlphaTS (class in sfepy.solvers.ts\_solvers), 715

GenericFileSource (class in sfepy.postprocess.sources), 682

GenericSequenceFileSource (class in sfepy.postprocess.sources), 682

geo\_ctx (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), 593

geometries (sfepy.terms.terms.Term attribute), 739

geometries (sfepy.terms.terms\_elastic.ElasticWaveCauchyTerm attribute), 778

geometries (sfepy.terms.terms\_elastic.ElasticWaveTerm attribute), 778

geometries (sfepy.terms.terms\_elastic.LinearElasticIsotropicTerm attribute), 779

geometries (sfepy.terms.terms\_elastic.NonsymElasticTerm attribute), 782

geometries (sfepy.terms.terms\_elastic.SDLinearElasticTerm attribute), 783

geometries (sfepy.terms.terms\_membrane.TLMembraneTerm get\_bounding\_box() (sfepy.discrete.fem.mesh.Mesh method), 796  
attribute), 796  
geometries (sfepy.terms.terms\_navier\_stokes.StokesWaveDig get\_bounding\_box() (sfepy.postprocess.sources.GenericFileSource method), 682  
attribute), 804  
geometries (sfepy.terms.terms\_navier\_stokes.StokesWaveTeg get\_bounding\_box() (sfepy.postprocess.sources.VTKFileSource method), 682  
attribute), 804  
geometries (sfepy.terms.terms\_shells.Shell10XTerm at get\_box\_matrix() (sfepy.mesh.splinebox.SplineBox method), 675  
tribute), 809  
geometries (sfepy.terms.terms\_surface.ContactPlaneTerm get\_box\_volume() (in module sfepy.homogenization.utils), 646  
attribute), 811  
geometries (sfepy.terms.terms\_surface.ContactSphereTerm get\_callback() (in module sfepy.homogenization.coefs\_phononic), 639  
attribute), 812  
geometry (class in sfepy.mesh.geom\_tools), 669  
GeometryElement (class in sfepy.discrete.fem.geometry\_element), 601  
geomobject (class in sfepy.mesh.geom\_tools), 671  
get() (sfepy.base.base.Container method), 516  
get() (sfepy.base.base.Struct method), 517  
get() (sfepy.base.multiproc\_proc.MyQueue method), 533  
get() (sfepy.discrete.integrals.Integrals method), 547  
get() (sfepy.mechanics.matcoefs.ElasticConstants method), 656  
get() (sfepy.terms.terms.Term method), 739  
get\_2d\_points() (in module sfepy.mesh.bspline), 669  
get\_a0() (sfepy.solvers.ts\_solvers.ElastodynamicsBaseTS method), 715  
get\_AABB() (in module sfepy.mechanics.extmods.ccontres), 665  
get\_actual\_cache() (sfepy.discrete.probes.Probe method), 551  
get\_actual\_order() (in module sfepy.discrete.quadratures), 566  
get\_animation\_info() (sfepy.postprocess.viewer.Viewer method), 687  
get\_arg\_kinds() (in module sfepy.terms.terms), 741  
get\_arg\_name() (sfepy.terms.terms.Term method), 739  
get\_args() (sfepy.terms.terms.Term method), 739  
get\_args\_by\_name() (sfepy.terms.terms.Term method), 739  
get\_arguments() (in module sfepy.base.base), 519  
get\_assembling\_cells() (sfepy.terms.terms.Term method), 739  
get\_base() (sfepy.discrete.fem.fields\_base.FEField method), 596  
get\_base() (sfepy.discrete.fem.mappings.FEMapping method), 605  
get\_base() (sfepy.discrete.fem.mappings.SurfaceMapping method), 605  
get\_basic\_info() (in module sfepy.version), 514  
get\_bezier\_element\_entities() (in module sfepy.discrete.iga.iga), 626  
get bezier\_topology() (in module sfepy.discrete.iga.iga), 627  
get\_bounding\_box() (sfepy.discrete.fem.mesh.Mesh method), 606  
get\_box\_matrix() (sfepy.mesh.splinebox.SplineBox method), 675  
get\_box\_volume() (in module sfepy.homogenization.utils), 646  
get\_callback() (in module sfepy.homogenization.coefs\_phononic), 639  
get\_cauchy\_from\_2pk() (sfepy.mechanics.tensors.StressTransform method), 661  
get\_cell\_conn() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_cell\_indices() (sfepy.discrete.common.region.Region method), 590  
get\_cells() (sfepy.discrete.common.region.Region method), 590  
get\_centroids() (sfepy.discrete.common.domain.Domain method), 578  
get\_centroids() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_charfun() (sfepy.discrete.common.region.Region method), 591  
get\_cmem\_usage() (in module sfepy.discrete.common.extmods.cmesh), 581  
get\_coef() (sfepy.homogenization.coefs\_base.CoefN method), 633  
get\_coef() (sfepy.homogenization.coefs\_base.CoefNN method), 633  
get\_coefs() (sfepy.homogenization.coefs\_phononic.AcousticMassLiquidTer method), 636  
get\_coefs() (sfepy.homogenization.coefs\_phononic.AcousticMassTensor method), 636  
get\_complete() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_condition\_value() (in module sfepy.discrete.conditions), 539  
get\_conn() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_conn() (sfepy.discrete.fem.domain.FEDomain method), 592  
get\_conn() (sfepy.discrete.fem.mesh.Mesh method), 606  
get\_conn\_as\_graph() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_conn\_info() (sfepy.terms.terms.Term method), 739  
get\_conn\_key() (sfepy.terms.terms.Term method), 739  
get\_conn\_permutations() (sfepy.discrete.fem.geometry\_element.GeometryElement method), 601

get\_connectivity() (sfepy.discrete.fem.fe\_surface.FESurface.get\_dict\_idxval() (in module sfepy.homogenization.engine), 642  
method), 594  
get\_connectivity() (sfepy.discrete.fem.fields\_base.FEField.get\_dim() (sfepy.discrete.problem.Problem method), 557  
method), 596  
get\_consistent\_unit\_set() (in module sfepy.mechanics.units), 665  
get\_dimension() (sfepy.discrete.fem.meshio.VTKMeshIO.get\_dim() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613  
method), 613  
get\_dir() (in module run\_tests), 505  
get\_distance() (sfepy.mechanics.contact\_bodies.ContactPlane.get\_distance() (sfepy.mechanics.contact\_bodies.ContactSphere method), 655  
method), 655  
get\_contact\_info() (sfepy.terms.terms\_contact.ContactTerm.get\_distance() (sfepy.mechanics.contact\_bodies.ContactSphere method), 655  
method), 762  
get\_control\_points() (sfepy.mesh.bspline.BSpline.get\_dof\_conn() (sfepy.discrete.variables.FieldVariable method), 570  
method), 667  
get\_control\_points() (sfepy.mesh.bspline.BSplineSurf.get\_dof\_conn\_type() (sfepy.terms.terms.Term method), 739  
method), 668  
get\_control\_points() (sfepy.mesh.splinebox.SplineBox.get\_dof\_info() (sfepy.discrete.variables.FieldVariable method), 570  
method), 675  
get\_coor() (sfepy.discrete.fem.fields\_base.FEField.get\_dofs() (in module save\_basis), 513  
method), 596  
get\_coors\_in\_ball() (in module sfepy.linalg.geometry), 648  
get\_coors\_in\_tube() (in module sfepy.linalg.geometry), 648  
get\_coors\_shape() (sfepy.mesh.splinebox.SplineBox.get\_dofs\_in\_region() (sfepy.discrete.fem.fields\_base.FEField method), 597  
method), 675  
get\_correctors\_from\_file\_hdf5() (in module sfepy.homogenization.micmac), 643  
get\_data() (sfepy.discrete.materials.Material.get\_dofs\_in\_region() (sfepy.discrete.iga.fields.IGField method), 622  
method), 548  
get\_data\_name() (in module sfepy.discrete.probes), 553  
get\_data\_names() (sfepy.postprocess.viewer.Viewer.get\_domain() (sfepy.discrete.equations.Equations method), 541  
method), 687  
get\_data\_ranges() (in module sfepy.postprocess.utils), 683  
get\_data\_shape() (sfepy.discrete.fem.fields\_base.FEField.get\_dsg\_strain() (in module sfepy.mechanics.shell10x), 661  
method), 596  
get\_ebc\_indices() (sfepy.discrete.problem.Problem.get\_dual() (sfepy.discrete.variables.Variable method), 572  
method), 558  
get\_econn() (sfepy.discrete.fem.fields\_base.SurfaceField.get\_dual\_names() (sfepy.discrete.variables.Variables method), 574  
method), 598  
get\_econn() (sfepy.discrete.fem.fields\_base.VolumeField.get\_econn() (sfepy.discrete.iga.fields.IGField method), 622  
method), 599  
get\_edge\_graph() (sfepy.discrete.common.region.Region.get\_edges\_per\_face() (sfepy.discrete.fem.geometry\_element.GeometryElement method), 591  
method), 601  
get\_edge\_paths() (in module sfepy.discrete.fem.utils), 617  
get\_element\_diameters() (sfepy.discrete.common.extmods.mappings.CMap.get\_element\_diameters() (sfepy.discrete.fem.domain.FEDomain method), 582  
method), 592  
get\_element\_diameters() (sfepy.discrete.variables.FieldVariable.get\_element\_diameters() (sfepy.discrete.variables.FieldVariable method), 570  
method), 590  
get\_entities() (sfepy.discrete.common.region.Region.get\_eval\_coors() (in module sfepy.discrete.fem.linearizer), 605  
method), 591  
get\_deviator() (in module sfepy.mechanics.tensors), 662  
get\_diameter() (sfepy.discrete.fem.domain.FEDomain.get\_ewh\_data() (sfepy.terms.terms\_th.ETHTerm method), 814  
method), 592  
get\_dict() (in module sfepy.base.multiproc\_proc), 533

get\_eval\_dofs() (in module sfepy.discrete.fem.linearizer), [605](#)

get\_eval\_expression() (in module sfepy.discrete.fem.fields\_base), [600](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.NSO<sub>EMinGradTerm</sub>) (sfepy.terms.terms\_elastic.CauchyStrainTerm method), [775](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.NSO<sub>ESurfMinDisp</sub>) (sfepy.terms.terms\_elastic.CauchyStressETHTerm method), [776](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDG<sub>getevalshape</sub>) (sfepy.terms.terms\_elastic.CauchyStressTerm method), [777](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDD<sub>getGradShape</sub>) (sfepy.terms.terms\_elastic.CauchyStressTHTerm method), [776](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDD<sub>getTmlShape</sub>) (sfepy.terms.terms\_elastic.LinearElasticIsotropicTerm method), [779](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDD<sub>getValuesTml</sub>) (sfepy.terms.terms\_elastic.LinearElasticTerm method), [781](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDG<sub>getDivShape</sub>) (sfepy.terms.terms\_elastic.LinearPrestressTerm method), [781](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDPG<sub>getPGStabilize</sub>) (sfepy.terms.terms\_elastic.NonsymElasticTerm method), [782](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDPG<sub>getPGStabilize</sub>) (sfepy.terms.terms\_elastic.SDLinearElasticTerm method), [783](#)

get\_eval\_shape() (sfepy.terms.terms\_adj\_navier\_stokes.SDS<sub>getPGStabilize</sub>) (sfepy.terms.terms\_fibres.FibresActiveTLTerm method), [784](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.IntegrateSurfaceTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_base.DeformationGradient method), [785](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.IntegrateVolumeTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase method), [786](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.IntegrateVolumeTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), [787](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.SumNodalValuesTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm method), [788](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.SurfaceMomentTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm method), [790](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.VolumeSurfaceTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm method), [791](#)

get\_eval\_shape() (sfepy.terms.terms\_basic.VolumeTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm method), [792](#)

get\_eval\_shape() (sfepy.terms.terms\_biot.BiotTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), [793](#)

get\_eval\_shape() (sfepy.terms.terms\_contact.ContactTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm method), [795](#)

get\_eval\_shape() (sfepy.terms.terms\_diffusion.DiffusionCouplingTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_membrane.TLMembraneTerm method), [796](#)

get\_eval\_shape() (sfepy.terms.terms\_diffusion.DiffusionTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_navier\_stokes.DivGradTerm method), [797](#)

get\_eval\_shape() (sfepy.terms.terms\_diffusion.DiffusionVelocityTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_navier\_stokes.DivTerm method), [798](#)

get\_eval\_shape() (sfepy.terms.terms\_diffusion.SDDiffusionTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_navier\_stokes.GradTerm method), [799](#)

get\_eval\_shape() (sfepy.terms.terms\_diffusion.SurfaceFluxTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_navier\_stokes.StokesTerm method), [803](#)

get\_eval\_shape() (sfepy.terms.terms\_dot.DotProductVolumeTerm<sub>getevalshape</sub>) (sfepy.terms.terms\_piezo.PiezoCouplingTerm method), [806](#)

get\_eval\_shape() (sfepy.terms.terms\_piezo.PiezoStrainTerm.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDGradDivStabilizationTerm.get\_fargs(),  
     method), 806  
     method), 747

get\_eval\_shape() (sfepy.terms.terms\_piezo.PiezoStressTerm.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGCStabilizationTerm.get\_fargs(),  
     method), 807  
     method), 748

get\_eval\_shape() (sfepy.terms.terms\_surface.LinearTractionTerm.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGPStabilizationTerm.get\_fargs(),  
     method), 812  
     method), 748

get\_eval\_shape() (sfepy.terms.terms\_surface.SDSurfaceIntegrateTerm.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDSUPGCStabilizationTerm.get\_fargs(),  
     method), 813  
     method), 749

get\_eval\_shape() (sfepy.terms.terms\_surface.SurfaceNormalTerm.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SUPGCAjStabilizationTerm.get\_fargs(),  
     method), 813  
     method), 750

get\_evaluate\_cache() (sfepy.discrete.fem.fields\_base.FEField.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj1StabilizationTerm.get\_fargs(),  
     method), 597  
     method), 750

get\_evaluate\_cache() (sfepy.discrete.probes.Probe.get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj2StabilizationTerm.get\_fargs(),  
     method), 551  
     method), 751

get\_evaluator() (sfepy.discrete.problem.Problem.get\_fargs() (sfepy.terms.terms\_basic.IntegrateSurfaceTerm.get\_fargs(),  
     method), 558  
     method), 753

get\_exp() (sfepy.homogenization.convolutions.Convolution.get\_fargs() (sfepy.terms.terms\_basic.IntegrateVolumeMatTerm.get\_fargs(),  
     method), 640  
     method), 753

get\_expression\_arg\_names() (in module sfepy.discrete.equations). 543

get\_face\_areas() (in module sfepy.linalg.geometry), 649

get\_facet\_axes() (in module sfepy.discrete.iga.iga), 627

get\_facet\_dof\_permutations() (in module sfepy.discrete.fem.facets), 594

get\_facet\_indices() (sfepy.discrete.common.region.Region.get\_fargs() (sfepy.terms.terms\_basic.SurfaceMomentTerm.get\_fargs(),  
     method), 591  
     method), 755

get\_facet\_normals() (sfepy.discrete.common.extmods.cmeshg.CMeshg.get\_fargs() (sfepy.terms.terms\_basic.VolumeSurfaceTerm.get\_fargs(),  
     method), 580  
     method), 756

get\_family\_data (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTerm.get\_fargs() (sfepy.terms.terms\_basic.VolumeTerm.get\_fargs(),  
     attribute), 788  
     method), 757

get\_family\_data (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTerm.get\_fargs() (sfepy.terms.terms\_basic.ZeroTerm.get\_fargs(),  
     attribute), 788  
     method), 757

get\_family\_data (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULTerm.get\_fargs() (sfepy.terms.terms\_biot.BiotETHTerm.get\_fargs(),  
     attribute), 793  
     method), 758

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvectionTerm.get\_fargs() (sfepy.terms.terms\_biot.BiotStressTerm.get\_fargs(),  
     method), 742  
     method), 758

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvectionTerm.get\_fargs() (sfepy.terms.terms\_biot.BiotTerm.get\_fargs(),  
     method), 742  
     method), 760

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.AdjDivGradTerm.get\_fargs() (sfepy.terms.terms\_biot.BiotTHTerm.get\_fargs(),  
     method), 743  
     method), 759

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMingeTerm.get\_fargs() (sfepy.terms.terms\_constraints.NonPenetrationPenaltyTerm.get\_fargs(),  
     method), 743  
     method), 760

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDiffPenaltyTerm.get\_fargs() (sfepy.terms.terms\_constraints.NonPenetrationTerm.get\_fargs(),  
     method), 744  
     method), 761

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDiffPenaltyTerm.get\_fargs() (sfepy.terms.terms\_contact.ContactTerm.get\_fargs(),  
     method), 744  
     method), 762

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDConvectionTerm.get\_fargs() (sfepy.terms.terms\_diffusion.ConvectVGradSTerm.get\_fargs(),  
     method), 745  
     method), 763

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDDivGradTerm.get\_fargs() (sfepy.terms.terms\_diffusion.DiffusionCouplingTerm.get\_fargs(),  
     method), 745  
     method), 764

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDDivTerm.get\_fargs() (sfepy.terms.terms\_diffusion.DiffusionRTerm.get\_fargs(),  
     method), 746  
     method), 764

get\_fargs() (sfepy.terms.terms\_adj\_navier\_stokes.SDDotVolTerm.get\_fargs() (sfepy.terms.terms\_diffusion.DiffusionTerm.get\_fargs(),  
     method), 746  
     method), 765

get\_fargs() (sfepy.terms.terms\_diffusion.DiffusionVelocityTerm get\_fargs() (sfepy.terms.terms\_fibres.FibresActiveTLTerm method), 765  
get\_fargs() (sfepy.terms.terms\_diffusion.SDDiffusionTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientTerm method), 767  
get\_fargs() (sfepy.terms.terms\_diffusion.SurfaceFluxOperator get\_fargs() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase method), 768  
get\_fargs() (sfepy.terms.terms\_diffusion.SurfaceFluxTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 768  
get\_fargs() (sfepy.terms.terms\_dot.BCNewtonTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm method), 768  
get\_fargs() (sfepy.terms.terms\_dot.DotProductVolumeTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm method), 770  
get\_fargs() (sfepy.terms.terms\_dot.DotSProductVolumeOperator get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm method), 770  
get\_fargs() (sfepy.terms.terms\_dot.DotSProductVolumeOperator get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm method), 771  
get\_fargs() (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm method), 771  
get\_fargs() (sfepy.terms.terms\_dot.ScalarDotMGradScalarTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 772  
get\_fargs() (sfepy.terms.terms\_dot.VectorDotGradScalarTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm method), 773  
get\_fargs() (sfepy.terms.terms\_dot.VectorDotScalarTerm get\_fargs() (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm method), 774  
get\_fargs() (sfepy.terms.terms\_elastic.CauchyStrainTerm get\_fargs() (sfepy.terms.terms\_membrane.TLMembraneTerm method), 775  
get\_fargs() (sfepy.terms.terms\_elastic.CauchyStressETHTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.ConvectTerm method), 776  
get\_fargs() (sfepy.terms.terms\_elastic.CauchyStressTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.DivGradTerm method), 777  
get\_fargs() (sfepy.terms.terms\_elastic.CauchyStressTHTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.DivOperatorTerm method), 776  
get\_fargs() (sfepy.terms.terms\_elastic.ElasticWaveCauchyTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.DivTerm method), 778  
get\_fargs() (sfepy.terms.terms\_elastic.ElasticWaveTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.GradDivStabilizationTerm method), 778  
get\_fargs() (sfepy.terms.terms\_elastic.LinearElasticETHTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.GradTerm method), 779  
get\_fargs() (sfepy.terms.terms\_elastic.LinearElasticIsotropicTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.LinearConvect2Term method), 779  
get\_fargs() (sfepy.terms.terms\_elastic.LinearElasticTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.LinearConvectTerm method), 781  
get\_fargs() (sfepy.terms.terms\_elastic.LinearElasticTHTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.PSPGCStabilizationTerm method), 780  
get\_fargs() (sfepy.terms.terms\_elastic.LinearPrestressTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.StokesTerm method), 781  
get\_fargs() (sfepy.terms.terms\_elastic.LinearStrainFiberTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm method), 782  
get\_fargs() (sfepy.terms.terms\_elastic.NonsymElasticTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.StokesWaveTerm method), 782  
get\_fargs() (sfepy.terms.terms\_elastic.SDLinearElasticTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.SUPGCStabilizationTerm method), 783  
get\_fargs() (sfepy.terms.terms\_electric.ElectricSourceTerm get\_fargs() (sfepy.terms.terms\_navier\_stokes.SUPGPStabilizationTerm method), 784

get\_fargs() (sfepy.terms.terms\_piezo.PiezoCouplingTerm method), 806  
 get\_fargs() (sfepy.terms.terms\_piezo.PiezoStrainTerm method), 806  
 get\_fargs() (sfepy.terms.terms\_piezo.PiezoStressTerm method), 807  
 get\_fargs() (sfepy.terms.terms\_point.ConcentratedPointLoadTerm method), 807  
 get\_fargs() (sfepy.terms.terms\_point.LinearPointSpringTerm method), 808  
 get\_fargs() (sfepy.terms.terms\_shells.Shell10XTerm method), 809  
 get\_fargs() (sfepy.terms.terms\_surface.ContactPlaneTerm method), 811  
 get\_fargs() (sfepy.terms.terms\_surface.ContactSphereTerm method), 812  
 get\_fargs() (sfepy.terms.terms\_surface.LinearTractionTerm method), 812  
 get\_fargs() (sfepy.terms.terms\_surface.SDSurfaceIntegrateTerm method), 813  
 get\_fargs() (sfepy.terms.terms\_surface.SurfaceNormalDotTerm method), 813  
 get\_fargs() (sfepy.terms.terms\_surface.SurfaceJumpTerm method), 814  
 get\_fargs() (sfepy.terms.terms\_volume.LinearVolumeForceTerm method), 815  
 get\_field() (sfepy.discrete.variables.FieldVariable method), 570  
 get\_filename\_trunk() (sfepy.discrete.fem.meshio.MeshIO method), 611  
 get\_filename\_trunk() (sfepy.discrete.fem.meshio.UserMeshIO method), 612  
 get\_full() (sfepy.discrete.variables.FieldVariable method), 570  
 get\_full() (sfepy.homogenization.convolutions.ConvolutionKernel method), 640  
 get\_full\_indices() (in module sfepy.mechanics.tensors), 662  
 get\_function() (sfepy.base.conf.ProblemConf method), 525  
 get\_gap\_ranges() (in module sfepy.homogenization.coefs\_phononic), 639  
 get\_geometry() (sfepy.discrete.fem.mappings.FEMapping method), 605  
 get\_geometry() (sfepy.discrete.iga.mappings.IGMapping method), 629  
 get\_geometry\_types() (sfepy.terms.terms.Term method), 739  
 get\_glyphs\_scale\_factor() (in module sfepy.postprocess.viewer), 688  
 get\_graph\_conns() (sfepy.discrete.equations.Equations method), 541  
 get\_green\_strain\_sym3d() (in module sfepy.mechanics.membranes), 659  
 get\_grid() (sfepy.discrete.fem.geometry\_element.GeometryElement method), 601  
 get\_grid\_plane() (in module sfepy.discrete.fem.periodic), 613  
 get\_homog\_coefs\_linear() (in module sfepy.homogenization.micmac), 643  
 get\_homog\_coefs\_nonlinear() (in module sfepy.homogenization.micmac), 643  
 get\_incident() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
 get\_idx() (sfepy.discrete.variables.Variables method), 574  
 get\_info() (sfepy.discrete.common.dof\_info.DofInfo method), 576  
 get\_initial\_condition() (sfepy.discrete.variables.Variable method), 572  
 get\_initial\_state() (sfepy.discrete.problem.Problem method), 558  
 get\_initial\_vec() (sfepy.solvers.ts\_solvers.ElastodynamicsBaseTS method), 715  
 get\_int\_value() (in module sfepy.base.multiproc\_proc), 533  
 get\_integrals() (sfepy.discrete.problem.Problem method), 558  
 get\_interp\_coors() (sfepy.discrete.variables.FieldVariable method), 571  
 get\_interpolation\_name() (sfepy.discrete.fem.geometry\_element.GeometryElement method), 601  
 get\_invariants() (in module sfepy.mechanics.membranes), 659  
 get\_item\_by\_name() (sfepy.base.conf.ProblemConf method), 525  
 get\_jacobian() (in module sfepy.discrete.common.mappings), 587  
 get\_keys() (sfepy.discrete.materials.Material method), 548  
 get\_knot\_vector() (sfepy.mesh.bspline.BSpline method), 667  
 get\_kw\_args() (sfepy.terms.terms.Term method), 740  
 get\_lattice\_volume() (in module sfepy.homogenization.utils), 646  
 get\_lcbc\_operator() (sfepy.discrete.equations.Equations method), 541  
 get\_lcbc\_operator() (sfepy.discrete.variables.Variables method), 574  
 get\_list() (in module sfepy.base.multiproc\_proc), 533  
 get\_local\_entities() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
 get\_local\_ids() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
 get\_lock() (in module sfepy.base.multiproc\_proc), 534  
 get\_log\_freqs() (in module sfepy.homogenization.coefs\_phononic), 639

get\_log\_name() (sfepy.base.log.Log method), 531  
get\_logging\_conf() (in module sfepy.base.log), 531  
get\_longest\_edge\_and\_gps() (in module sfepy.mechanics.extmods.ccontres), 665  
get\_ls() (sfepy.discrete.problem.Problem method), 558  
get\_manager() (in module sfepy.base.multiproc\_proc), 534  
get\_mapping() (sfepy.discrete.common.fields.Field method), 584  
get\_mapping() (sfepy.discrete.fem.mappings.SurfaceMapping method), 605  
get\_mapping() (sfepy.discrete.fem.mappings.VolumeMapping method), 606  
get\_mapping() (sfepy.discrete.iga.mappings.IGMapping method), 629  
get\_mapping() (sfepy.discrete.structural.mappings.Shell10X method), 631  
get\_mapping() (sfepy.discrete.variables.FieldVariable method), 571  
get\_mapping() (sfepy.terms.terms.Term method), 740  
get\_mapping\_data() (in module sfepy.discrete.common.mappings), 588  
get\_mapping\_data() (in module sfepy.mechanics.shell10x), 661  
get\_maps() (sfepy.solvers.oseen.StabilizationFunction method), 710  
get\_mat\_id() (sfepy.postprocess.sources.FileSource method), 681  
get\_mat\_id() (sfepy.postprocess.sources.GenericFileSource method), 682  
get\_material\_names() (sfepy.terms.terms.Term method), 740  
get\_material\_names() (sfepy.terms.terms.Terms method), 741  
get\_materials() (sfepy.discrete.problem.Problem method), 558  
get\_materials() (sfepy.terms.terms.Term method), 740  
get\_matrices() (sfepy.solvers.ts\_solvers.ElastodynamicsBaseTS method), 715  
get\_matrix\_shape() (sfepy.discrete.variables.Variables method), 574  
get\_mem\_usage() (in module sfepy.base.mem\_usage), 532  
get\_mesh\_bounding\_box() (sfepy.discrete.fem.domain.FEDomain method), 592  
get\_mesh\_coors() (sfepy.discrete.fem.domain.FEDomain method), 592  
get\_mesh\_coors() (sfepy.discrete.problem.Problem method), 558  
get\_micro\_cache\_key() (sfepy.homogenization.homogen\_appeHomogenizationApp method), 643  
get\_min\_dt() (in module sfepy.solvers.ts\_solvers), 718  
get\_min\_value() (in module sfepy.discrete.fem.utils), 617  
get\_min\_vertex\_distance() (in module sfepy.discrete.fem.mesh), 607  
get\_min\_vertex\_distance\_naive() (in module sfepy.discrete.fem.mesh), 607  
get\_mirror\_region() (sfepy.discrete.common.region.Region method), 591  
get\_mpdict\_value() (in module sfepy.base.multiproc\_proc), 534  
get\_mtx\_i() (sfepy.discrete.fem.poly\_spaces.LagrangeTensorProductPolySp method), 614  
get\_mtx\_i() (sfepy.discrete.fem.poly\_spaces.PolySpace method), 615  
get\_multiproc() (in module sfepy.base.multiproc), 533  
get\_n\_cells() (sfepy.discrete.common.region.Region method), 591  
get\_n\_dof\_total() (sfepy.discrete.common.dof\_info.DofInfo method), 576  
get\_names() (sfepy.base.base.Container method), 516  
get\_names() (sfepy.base.base.OneTypeList method), 517  
get\_nls() (sfepy.discrete.problem.Problem method), 558  
get\_nls\_functions() (sfepy.discrete.problem.Problem method), 558  
get\_non\_diagonal\_indices() (in module sfepy.mechanics.tensors), 662  
get\_normals() (in module sfepy.discrete.common.mappings), 588  
get\_num\_workers() (in module sfepy.base.multiproc), 533  
get\_number() (sfepy.base.testing.TestCommon method), 536  
get\_nums() (in module sfepy.base.resolve\_deps), 536  
get\_opacities() (in module sfepy.postprocess.viewer), 688  
get\_operator() (sfepy.discrete.common.dof\_info.EquationMap method), 577  
get\_or\_create\_hdf5\_group() (in module sfepy.base.ioutils), 528  
get\_orientations() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_output() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 634  
get\_output\_approx\_order() (sfepy.discrete.fem.fields\_base.FEField method), 597  
get\_output\_function() (sfepy.base.base.Output method), 517  
get\_output\_name() (sfepy.discrete.problem.Problem method), 558  
get\_output\_prefix() (sfepy.base.base.Output method), 517  
get\_output\_suffix() (in module sfepy.homogenization.recovery), 645  
get\_parameter\_variables() (sfepy.terms.terms.Term method), 740

get\_parents() (in module sfepy.discrete.common.region), 591  
 get\_parts() (sfepy.discrete.state.State method), 567  
 get\_patch\_box\_regions() (in module sfepy.discrete.iga.iga), 627  
 get\_perpendiculars() (in module sfepy.linalg.geometry), 649  
 get\_physical\_qps() (in module sfepy.discrete.common.mappings), 589  
 get\_physical\_qps() (sfepy.discrete.fem.mappings.FEMapping method), 605  
 get\_physical\_qps() (sfepy.discrete.iga.mappings.IGMapping method), 629  
 get\_physical\_qps() (sfepy.discrete.structural.mappings.Shell10XMapping method), 631  
 get\_physical\_qps() (sfepy.terms.terms.Term method), 740  
 get\_physical\_qps() (sfepy.terms.terms\_shells.Shell10XTermget\_shape() (sfepy.discrete.common.mappings.PhysicalQPs method), 810  
 get\_points() (sfepy.discrete.probes.CircleProbe method), 550  
 get\_points() (sfepy.discrete.probes.LineProbe method), 551  
 get\_points() (sfepy.discrete.probes.PointsProbe method), 551  
 get\_points() (sfepy.discrete.probes.RayProbe method), 553  
 get\_position\_counts() (in module sfepy.postprocess.viewer), 688  
 get\_potential\_cells() (in module sfepy.discrete.common.global\_interp), 585  
 get\_prefix() (sfepy.mechanics.units.Unit static method), 665  
 get\_primary() (sfepy.discrete.variables.Variable method), 572  
 get\_primary\_name() (sfepy.discrete.variables.Variable method), 572  
 get\_print\_info() (in module sfepy.base.ioutils), 528  
 get\_print\_info() (in module sfepy.solvers.ts), 714  
 get\_qp() (sfepy.discrete.fem.fields\_base.FEField method), 597  
 get\_qp() (sfepy.discrete.integrals.Integral method), 547  
 get\_qp\_key() (sfepy.terms.terms.Term method), 740  
 get\_queue() (in module sfepy.base.multiproc\_proc), 534  
 get\_range\_indices() (in module sfepy.terms.utils), 815  
 get\_ranges() (in module sfepy.homogenization.coefs\_phononic), 639  
 get\_raveled\_index() (in module sfepy.discrete.iga.iga), 627  
 get\_raw() (sfepy.base.conf.ProblemConf method), 525  
 get\_reduced() (sfepy.discrete.state.State method), 567  
 get\_reduced() (sfepy.discrete.variables.FieldVariable method), 571  
 get\_ref\_coors() (in module sfepy.discrete.common.global\_interp), 585  
 get\_ref\_coors\_convex() (in module sfepy.discrete.common.global\_interp), 586  
 get\_ref\_coors\_general() (in module sfepy.discrete.common.global\_interp), 586  
 get\_region() (sfepy.terms.terms.ConnInfo method), 738  
 get\_region() (sfepy.terms.terms.Term method), 740  
 get\_region\_name() (sfepy.terms.terms.ConnInfo method), 738  
 get\_restart\_filename() (sfepy.discrete.problem.Problem method), 558  
 get\_save\_name() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 634  
 get\_save\_name\_base() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 634  
 get\_schur() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
 get\_shape() (sfepy.discrete.common.mappings.PhysicalQPs method), 587  
 get\_shape\_kind() (in module sfepy.terms.terms), 741  
 get\_simplex\_circumcentres() (in module sfepy.linalg.geometry), 649  
 get\_simplex\_cubature() (in module sfepy.discrete.simplex\_cubature), 566  
 get\_simplex\_volumes() (in module sfepy.linalg.geometry), 649  
 get\_size\_hint() (sfepy.postprocess.viewer.Viewer method), 687  
 get\_solver() (sfepy.discrete.problem.Problem method), 558  
 get\_solver\_conf() (sfepy.discrete.problem.Problem method), 558  
 get\_sorted\_dependencies() (sfepy.homogenization.engine.HomogenizationWorker static method), 641  
 get\_sphinx\_make\_command() (in module build\_helpers), 506  
 get\_standard\_keywords() (in module sfepy.base.conf), 525  
 get\_standard\_type\_defs() (in module sfepy.base.parse\_conf), 534  
 get\_state() (sfepy.solvers.ts.TimeStepper method), 713  
 get\_state() (sfepy.solvers.ts.VariableTimeStepper method), 713  
 get\_state\_in\_region() (sfepy.discrete.variables.FieldVariable method), 571  
 get\_state\_names() (sfepy.terms.terms.Term method), 740  
 get\_state\_part\_view() (sfepy.discrete.variables.Variables method), 574  
 get\_state\_parts() (sfepy.discrete.equations.Equations method), 541  
 get\_state\_parts() (sfepy.discrete.variables.Variables method), 574  
 get\_state\_variables() (sfepy.terms.terms.Term method), 740

get\_step\_time() (sfepy.postprocess.sources.FileSource method), 681  
get\_str() (sfepy.terms.terms.Term method), 740  
get\_subdict() (in module sfepy.base.base), 519  
get\_subset\_info() (sfepy.discrete.common.dof\_info.DofInfo method), 576  
get\_surface\_degrees() (in module sfepy.discrete.iga.iga), 627  
get\_surface\_entities() (sfepy.discrete.fem.geometry\_element method), 601  
get\_surface\_faces() (in module extract\_surface), 509  
get\_surface\_facets() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_sym\_indices() (in module sfepy.mechanics.tensors), 662  
get\_t4\_from\_t2s() (in module sfepy.mechanics.tensors), 662  
get\_tangent\_stress\_matrix() (in module sfepy.mechanics.membranes), 659  
get\_tensor\_product\_conn() (in module sfepy.mesh.mesh\_generators), 674  
get\_timestepper() (sfepy.discrete.problem.Problem method), 558  
get\_tolerance() (sfepy.solvers.solvers.LinearSolver method), 712  
get\_trace() (in module sfepy.mechanics.tensors), 662  
get\_true\_order() (sfepy.discrete.fem.fields\_base.FEField method), 597  
get\_true\_order() (sfepy.discrete.iga.fields.IGField method), 622  
get\_trunk() (in module sfepy.base.ioutils), 529  
get\_ts\_info() (sfepy.postprocess.sources.FileSource method), 681  
get\_ts\_val() (sfepy.homogenization.coefs\_base.CorrSolution method), 634  
get\_tss() (sfepy.discrete.problem.Problem method), 558  
get\_tss\_functions() (sfepy.discrete.problem.Problem method), 558  
get\_type() (sfepy.base.ioutils.DataSoftLink method), 527  
get\_unraveled\_indices() (in module sfepy.discrete.iga.iga), 628  
get\_user\_names() (sfepy.terms.terms.Term method), 740  
get\_user\_names() (sfepy.terms.terms.Terms method), 741  
get\_var\_names() (sfepy.discrete.conditions.LinearCombinati method), 538  
get\_variable() (sfepy.discrete.equations.Equations method), 542  
get\_variable\_dependencies() (sfepy.discrete.equations.Equations method), 542  
get\_variable\_names() (sfepy.discrete.equations.Equations method), 542  
get\_variable\_names() (sfepy.terms.terms.Term method), 740  
get\_variable\_names() (sfepy.terms.terms.Terms method), 741  
get\_variables() (sfepy.terms.terms.Terms method), 741  
get\_variables() (sfepy.discrete.problem.Problem method), 559  
get\_variables() (sfepy.homogenization.coefs\_perfusion.CoeffRegion method), 636  
get\_variables() (sfepy.homogenization.coefs\_perfusion.CorrRegion method), 636  
get\_variables() (sfepy.terms.terms.Term method), 740  
get\_vec() (sfepy.discrete.state.State method), 567  
get\_vector() (sfepy.terms.terms.Term method), 740  
getCMFormat() (sfepy.discrete.fem.meshio.MeshIO method), 611  
get\_vectors() (sfepy.discrete.fem.lcbc\_operators.EdgeDirectionOperator method), 602  
get\_vectors() (sfepy.discrete.fem.lcbc\_operators.NormalDirectionOperator method), 604  
get\_vertices() (sfepy.discrete.fem.fields\_base.FEField method), 597  
get\_virtual\_name() (sfepy.terms.terms.Term method), 740  
get\_virtual\_variable() (sfepy.terms.terms.Term method), 740  
get\_volume() (in module sfepy.homogenization.utils), 646  
get\_volumes() (sfepy.discrete.common.extmods.cmesh.CMesh method), 580  
get\_volumetric\_tensor() (in module sfepy.mechanics.tensors), 662  
get\_von\_mises\_stress() (in module sfepy.mechanics.tensors), 662  
get\_vtk\_by\_group() (in module sfepy.postprocess.utils\_vtk), 684  
get\_vtk\_edges() (in module sfepy.postprocess.utils\_vtk), 684  
get\_vtk\_from\_file() (in module sfepy.postprocess.utils\_vtk), 684  
get\_vtk\_from\_mesh() (in module sfepy.postprocess.utils\_vtk), 684  
get\_vtk\_surface() (in module sfepy.postprocess.utils\_vtk), 684  
get\_weighted\_norm() (sfepy.discrete.state.State method), 567  
getBCnum() (sfepy.mesh.geom\_tools.geometry method), 670  
getcenterpoint() (sfepy.mesh.geom\_tools.surface method), 671  
getele() (sfepy.discrete.fem.meshio.TetgenMeshIO static method), 612  
getholepoints() (sfepy.mesh.geom\_tools.surface method), 671  
getinsidepoint() (sfepy.mesh.geom\_tools.surface method), 671

|                          |                                                            |     |                                 |                                                                   |     |    |
|--------------------------|------------------------------------------------------------|-----|---------------------------------|-------------------------------------------------------------------|-----|----|
| getinsidepoint()         | (sfepy.mesh.geom_tools.volume method),                     | 671 | H1NodalVolumeField              | (class sfepy.discrete.fem.fields_nodal),                          | 601 | in |
| getlines()               | (sfepy.mesh.geom_tools.surface method),                    | 671 | has_attr()                      | (in module sfepy.config),                                         | 514 |    |
| getn()                   | (sfepy.mesh.geom_tools.geomobject method),                 | 671 | has_cells()                     | (sfepy.discrete.common.region.Region method),                     | 591 |    |
| getnodes()               | (sfepy.discrete.fem.meshio.TetgenMeshIO static method),    | 612 | has_ebc()                       | (sfepy.discrete.state.State method),                              | 568 |    |
| getpoints()              | (sfepy.mesh.geom_tools.line method),                       | 671 | has_ebc()                       | (sfepy.discrete.variables.Variables method),                      | 574 |    |
| getpoints()              | (sfepy.mesh.geom_tools.surface method),                    | 671 | has_extra_nodes()               | (sfepy.discrete.fem.poly_spaces.NodeDescription method),          | 614 |    |
| getstr()                 | (sfepy.mesh.geom_tools.point method),                      | 671 | has_faces()                     | (sfepy.discrete.common.domain.Domain method),                     | 578 |    |
| getsurfaces()            | (sfepy.mesh.geom_tools.physicalsurface method),            | 671 | has_key()                       | (sfepy.base.base.Container method),                               | 516 |    |
| getsurfaces()            | (sfepy.mesh.geom_tools.volume method),                     | 671 | has_same_mesh()                 | (sfepy.discrete.variables.FieldVariable method),                  | 571 |    |
| getvolumes()             | (sfepy.mesh.geom_tools.physicalvolume method),             | 671 | has_virtuals()                  | (sfepy.discrete.variables.Variables method),                      | 574 |    |
| getxyz()                 | (sfepy.mesh.geom_tools.point method),                      | 671 | have_good_cython()              | (in module build_helpers),                                        | 506 |    |
| grad()                   | (in module sfepy.linalg.sympy_operators),                  | 652 | HDF5BaseData                    | (class in sfepy.base.ioutils),                                    | 527 |    |
| grad_as_vector()         | (in module sfepy.terms.terms_adj_navier_stokes),           | 751 | HDF5ContextManager              | (class in sfepy.base.ioutils),                                    | 527 |    |
| grad_v()                 | (in module sfepy.linalg.sympy_operators),                  | 652 | HDF5Data                        | (class in sfepy.base.ioutils),                                    | 527 |    |
| grad_vector_to_matrix()  | (in module eval_ns_forms),                                 | 508 | HDF5MeshIO                      | (class in sfepy.discrete.fem.meshio),                             | 608 |    |
| GradDivStabilizationTerm | (class in sfepy.terms.terms_navier_stokes),                | 798 | he_eval_from_mtx()              | (in module sfepy.terms.extmods.terms),                            | 818 |    |
| GradTerm                 | (class in sfepy.terms.terms_navier_stokes),                | 799 | he_residuum_from_mtx()          | (in module sfepy.terms.extmods.terms),                            | 818 |    |
| graph_components()       | (in module sfepy.discrete.common.extmods.cmesh),           | 581 | Histories                       | (class in sfepy.discrete.fem.history),                            | 602 |    |
| group_by_variables()     | (sfepy.discrete.conditions.Conditions method),             | 537 | History                         | (class in sfepy.discrete.fem.history),                            | 602 |    |
| group_chains()           | (in module sfepy.discrete.common.dof_info),                | 577 | homogen                         | (module),                                                         | 502 |    |
| guess()                  | (sfepy.discrete.fem.meshio.AbaqusMeshIO static method),    | 608 | HomogenizationApp               | (class sfepy.homogenization.homogen_app),                         | 642 | in |
| guess()                  | (sfepy.discrete.fem.meshio.ANSYSCDBMMeshIO static method), | 607 | HomogenizationEngine            | (class sfepy.homogenization.engine),                              | 640 | in |
| guess()                  | (sfepy.discrete.fem.meshio.AVSUCDMeshIO static method),    | 608 | HomogenizationWorker            | (class sfepy.homogenization.engine),                              | 641 | in |
| guess_format()           | (in module sfepy.discrete.fem.meshio),                     | 613 | HomogenizationWorkerMulti       | (class sfepy.homogenization.engine),                              | 641 | in |
| guess_time_units()       | (in module sfepy.postprocess.time_history),                | 683 | HomogenizationWorkerMultiMPI    | (class sfepy.homogenization.engine),                              | 642 | in |
| <b>H</b>                 |                                                            |     |                                 |                                                                   |     |    |
| H1DiscontinuousField     | (class sfepy.discrete.fem.fields_nodal),                   | 600 | hyperelastic_mode               | (sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase attribute), | 789 |    |
| H1HierachicVolumeField   | (class sfepy.discrete.fem.fields_hierachic),               | 600 | hyperelastic_mode               | (sfepy.terms.terms_hyperelastic_ul.HyperElasticULBase attribute), | 793 |    |
| H1Mixin                  | (class in sfepy.discrete.fem.fields_base),                 | 598 | HyperElasticBase                | (class sfepy.terms.terms_hyperelastic_base),                      | 785 | in |
| H1NodalMixin             | (class in sfepy.discrete.fem.fields_nodal),                | 600 | HyperElasticFamilyData          | (class sfepy.terms.terms_hyperelastic_base),                      | 786 | in |
| H1NodalSurfaceField      | (class sfepy.discrete.fem.fields_nodal),                   | 601 | HyperElasticSurfaceTLBase       | (class sfepy.terms.terms_hyperelastic_tl),                        | 788 | in |
|                          |                                                            |     | HyperElasticSurfaceTLFamilyData | (class sfepy.terms.terms_hyperelastic_tl),                        | 788 | in |
|                          |                                                            |     | HyperElasticTLBase              | (class sfepy.terms.terms_hyperelastic_tl),                        | 788 | in |

HyperElasticTLFamilyData (class in sfepy.terms.terms\_hyperelastic\_tl), 789  
HyperElasticULBase (class in sfepy.terms.terms\_hyperelastic\_ul), 793  
HyperElasticULFamilyData (class in sfepy.terms.terms\_hyperelastic\_ul), 794  
HypermeshAsciiMeshIO (class in sfepy.discrete.fem.meshio), 609

|

icntl (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
icntl (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
icntl (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701  
icntl (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_x attribute), 704  
iel (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), 593  
iel (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 620  
IGDomain (class in sfepy.discrete.iga.domain), 618  
IGField (class in sfepy.discrete.iga.fields), 621  
IGMapping (class in sfepy.discrete.iga.mappings), 628  
import\_file() (in module sfepy.base.base), 519  
in1d() (in module sfepy.base.compat), 521  
IndexedStruct (class in sfepy.base.base), 516  
indices (sfepy.discrete.common.extmods.cmsh.CConnectivity attribute), 579  
InDir (class in sfepy.base.ioutils), 528  
infinity\_norm() (in module sfepy.linalg.sparse), 651  
info (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
info (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
info (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701  
infog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
infog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
infog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701  
init() (sfepy.mechanics.matcoefs.ElasticConstants method), 656  
init\_data() (sfepy.discrete.variables.Variable method), 572  
init\_data\_struct() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticFamilyData method), 786  
init\_global\_search() (in module sfepy.mechanics.extmods.ccontres), 665  
init\_history() (sfepy.discrete.state.State method), 568

in init\_history() (sfepy.discrete.variables.Variable method), 572  
in init\_history() (sfepy.discrete.variables.Variables method), 574  
in init\_seq\_selection() (sfepy.postprocess.viewer.SetStep method), 685  
in init\_solvers() (sfepy.discrete.problem.Problem method), 559  
init\_solvers() (sfepy.homogenization.coefs\_base.MiniAppBase method), 634  
init\_subproblems() (sfepy.solvers.ls.MultiProblem method), 690  
init\_time() (sfepy.discrete.equations.Equations method), 542  
init\_time() (sfepy.discrete.problem.Problem method), 559  
InitialCondition (class in sfepy.discrete.conditions), 537  
initialize\_options() (build\_helpers.NoOptionsDocs method), 506  
insert() (sfepy.base.base.Container method), 516  
insert() (sfepy.terms.terms.Terms method), 741  
insert\_as\_static\_method() (in module sfepy.base.base), 519  
insert\_knot() (sfepy.mesh.bspline.BSpline method), 667  
insert\_method() (in module sfepy.base.base), 519  
insert\_sparse\_to\_csr() (in module sfepy.linalg.sparse), 651  
insert\_static\_method() (in module sfepy.base.base), 519  
insert\_strided\_axis() (in module sfepy.linalg.utils), 653  
insert\_sub\_reqs() (in module sfepy.homogenization.engine), 642  
instance\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
instance\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
instance\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701  
int\_dt() (sfepy.homogenization.convolutions.ConvolutionKernel method), 640  
Integral (class in sfepy.discrete.integrals), 547  
integral (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
IntegralMeanValueOperator (class in sfepy.discrete.fem.lcbc\_operators), 602  
IntegralProbe (class in sfepy.discrete.probes), 551  
Integrals (class in sfepy.discrete.integrals), 547  
integrate() (sfepy.discrete.common.extmods.mappings.CMapping method), 582  
integrate() (sfepy.discrete.integrals.Integral method), 547  
integrate() (sfepy.terms.terms\_contact.ContactTerm static method), 762  
integrate() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase static method), 786  
integrate\_along\_line() (in module probe), 504

integrate\_in\_time() (in module sfepy.homogenization.utils), 646

IntegrateSurfaceMatTerm (class in sfepy.terms.terms\_basic), 751

IntegrateSurfaceOperatorTerm (class in sfepy.terms.terms\_basic), 752

IntegrateSurfaceTerm (class in sfepy.terms.terms\_basic), 752

IntegrateVolumeMatTerm (class in sfepy.terms.terms\_basic), 753

IntegrateVolumeOperatorTerm (class in sfepy.terms.terms\_basic), 753

IntegrateVolumeTerm (class in sfepy.terms.terms\_basic), 754

integration (sfepy.terms.terms.Term attribute), 740

integration (sfepy.terms.terms\_adj\_navier\_stokes.NSOFSurfMinDPressure attribute), 744

integration (sfepy.terms.terms\_basic.IntegrateSurfaceMatTerm attribute), 751

integration (sfepy.terms.terms\_basic.IntegrateSurfaceOperatorTerm attribute), 752

integration (sfepy.terms.terms\_basic.IntegrateSurfaceTerm attribute), 753

integration (sfepy.terms.terms\_basic.SurfaceMomentTerm attribute), 755

integration (sfepy.terms.terms\_basic.SurfaceTerm attribute), 756

integration (sfepy.terms.terms\_basic.VolumeSurfaceTerm attribute), 756

integration (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 760

integration (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 761

integration (sfepy.terms.terms\_contact.ContactTerm attribute), 762

integration (sfepy.terms.terms\_diffusion.SurfaceFluxOperatorTerm attribute), 767

integration (sfepy.terms.terms\_diffusion.SurfaceFluxTerm attribute), 768

integration (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 769

integration (sfepy.terms.terms\_elastic.CauchyStrainSTerm attribute), 774

integration (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm attribute), 790

integration (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 791

integration (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm attribute), 791

integration (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 796

integration (sfepy.terms.terms\_navier\_stokes.SurfaceDivTerm attribute), 805

module integration (sfepy.terms.terms\_navier\_stokes.SurfaceGradTerm attribute), 805

in integration (sfepy.terms.terms\_point.ConcentratedPointLoadTerm attribute), 807

in integration (sfepy.terms.terms\_point.LinearPointSpringTerm attribute), 808

integration (sfepy.terms.terms\_shells.Shell10XTerm attribute), 810

in integration (sfepy.terms.terms\_surface.ContactPlaneTerm attribute), 811

in integration (sfepy.terms.terms\_surface.ContactSphereTerm attribute), 812

integration (sfepy.terms.terms\_surface.LinearTractionTerm attribute), 812

integration (sfepy.terms.terms\_surface.SDSurfaceIntegrateTerm attribute), 813

integration (sfepy.terms.terms\_surface.SurfaceNormalDotTerm attribute), 813

integration (sfepy.terms.terms\_surface.SurfaceJumpTerm attribute), 814

interp\_conv\_mat() (in module sfepy.homogenization.utils), 647

interp\_to\_qp() (sfepy.discrete.fem.fields\_base.FEField method), 597

interp\_v\_vals\_to\_n\_vals() (sfepy.discrete.fem.fields\_nodal.H1NodalSurfaceField method), 601

interp\_v\_vals\_to\_n\_vals() (sfepy.discrete.fem.fields\_nodal.H1NodalVolumeField method), 601

invalidate\_evaluate\_cache() (sfepy.discrete.variables.FieldVariable method), 571

invalidate\_term\_caches() (sfepy.discrete.equations.Equations method), 542

inverse\_element\_mapping() (in module sfepy.linalg.geometry), 649

invert\_dict() (in module sfepy.base.base), 519

invert\_remap() (in module sfepy.discrete.fem.utils), 617

iplot() (in module sfepy.base.plotutils), 535

ipython\_shell() (in module sfepy.base.base), 520

irhs\_ptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695

irhs\_ptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698

irhs\_ptr (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 701

irhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695

irhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698

irhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702

irn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
irn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
irn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
irn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 695  
irn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 698  
irn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
is\_active\_bc() (in module sfepy.discrete.common.dof\_info), 578  
is\_adjust (sfepy.postprocess.viewer.SetStep attribute), 685  
is\_bubble (sfepy.discrete.fem.extmods.bases.CLagrangeConText attribute), 593  
is\_complex() (sfepy.discrete.variables.Variable method), 572  
is\_cyclic (sfepy.discrete.probes.CircleProbe attribute), 551  
is\_cyclic (sfepy.discrete.probes.Probe attribute), 551  
is\_derived\_class() (in module sfepy.base.base), 520  
is\_finite() (sfepy.discrete.variables.Variable method), 572  
is\_higher\_order() (sfepy.discrete.fem.fields\_base.FEField method), 597  
is\_higher\_order() (sfepy.discrete.iga.fields.IGField method), 622  
is\_integer() (in module sfepy.base.base), 520  
is\_kind() (sfepy.discrete.variables.Variable method), 573  
is\_linear() (sfepy.discrete.problem.Problem method), 559  
is\_nurbs() (in module sfepy.discrete.iga.extmods.igac), 621  
is\_parameter() (sfepy.discrete.variables.Variable method), 573  
is\_real() (sfepy.discrete.variables.Variable method), 573  
is\_release() (sfepy.config.Config method), 514  
is\_remote\_dict() (in module sfepy.base.multiproc), 533  
is\_remote\_dict() (in module sfepy.base.multiproc\_proc), 534  
is\_sequence() (in module sfepy.base.base), 520  
is\_state() (sfepy.discrete.variables.Variable method), 573  
is\_state\_or\_parameter() (sfepy.discrete.variables.Variable method), 573  
is\_string() (in module sfepy.base.base), 520  
is\_sym (sfepy.homogenization.coefs\_base.CoefNonSym attribute), 633  
is\_sym (sfepy.homogenization.coefs\_base.CoefNonSymNonSym attribute), 633  
is\_sym (sfepy.homogenization.coefs\_base.CoefSym attribute), 633  
is\_sym (sfepy.homogenization.coefs\_base.CoefSymSym attribute), 633  
is\_virtual() (sfepy.discrete.variables.Variable method), 573  
isol\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
isol\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
isol\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
iter0() (in module sfepy.discrete.fem.facets), 594  
iter01() (in module sfepy.discrete.fem.facets), 594  
iter01x01y() (in module sfepy.discrete.fem.facets), 594  
iter01x10y() (in module sfepy.discrete.fem.facets), 594  
iter01y01x() (in module sfepy.discrete.fem.facets), 594  
iter01y10x() (in module sfepy.discrete.fem.facets), 594  
iter02() (in module sfepy.discrete.fem.facets), 594  
iter1() (in module sfepy.discrete.fem.facets), 594  
iter10() (in module sfepy.discrete.fem.facets), 594  
iter10x01y() (in module sfepy.discrete.fem.facets), 594  
iter10x10y() (in module sfepy.discrete.fem.facets), 594  
iter10y01x() (in module sfepy.discrete.fem.facets), 594  
iter10y10x() (in module sfepy.discrete.fem.facets), 594  
iter12() (in module sfepy.discrete.fem.facets), 594  
iter20() (in module sfepy.discrete.fem.facets), 594  
iter21() (in module sfepy.discrete.fem.facets), 594  
iter\_dict\_of\_lists() (in module sfepy.base.base), 520  
iter\_from() (sfepy.solvers.ts.TimeStepper method), 713  
iter\_from() (sfepy.solvers.ts.VariableTimeStepper method), 713  
iter\_from\_current() (sfepy.solvers.ts.VariableTimeStepper method), 713  
iter\_names() (sfepy.base.log.Log method), 531  
iter\_nonsym() (in module sfepy.homogenization.utils), 647  
iter\_single() (sfepy.discrete.conditions.Condition method), 537  
iter\_solutions() (sfepy.homogenization.coefs\_base.CorrSolution method), 634  
iter\_state() (sfepy.discrete.variables.Variables method), 574  
iter\_sym() (in module sfepy.homogenization.utils), 647  
iter\_sym() (sfepy.homogenization.coefs\_base.CoefNonSym static method), 633  
iter\_sym() (sfepy.homogenization.coefs\_base.CoefNonSymNonSym static method), 633  
iter\_sym() (sfepy.homogenization.coefs\_base.CoefSym static method), 633  
iter\_sym() (sfepy.homogenization.coefs\_base.CoefSymSym static method), 633  
iter\_symterms() (sfepy.discrete.materials.Material method), 548  
iter\_time\_steps() (sfepy.homogenization.coefs\_base.CorrSolution method), 634  
iteritems() (sfepy.base.base.Container method), 516  
iterkeys() (sfepy.base.base.Container method), 516

itervalues() (sfepy.base.base.Container method), 516

## J

jcn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
jcn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
jcn (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
jcn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
jcn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
jcn\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
job (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
job (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
job (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
job (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_x attribute), 704  
join\_tokens() (in module sfepy.discrete.parse\_regions), 550

## K

keep (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
keep (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
keep8 (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
keep8 (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
key (sfepy.base.gooptions.ValidatedDict attribute), 526  
key\_to\_index (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 580  
keys (sfepy.discrete.fem.poly\_spaces.PolySpace attribute), 615  
keys() (sfepy.base.gooptions.ValidatedDict method), 526  
kind (sfepy.discrete.fem.lcbc\_operators.EdgeDirectionOperator attribute), 602  
kind (sfepy.discrete.fem.lcbc\_operators.IntegralMeanValueOperator attribute), 603  
kind (sfepy.discrete.fem.lcbc\_operators.NodalLCOperator attribute), 604  
kind (sfepy.discrete.fem.lcbc\_operators.NoPenetrationOperator attribute), 604  
kind (sfepy.discrete.fem.lcbc\_operators.NormalDirectionOperator attribute), 604  
kind (sfepy.discrete.fem.lcbc\_operators.RigidOperator attribute), 604

kind (sfepy.discrete.fem.lcbc\_operators.ShiftedPeriodicOperator attribute), 605

## L

label\_dofs() (in module sfepy.parallel.plot\_parallel\_dofs), 676  
label\_global\_entities() (in module sfepy.postprocess.plot\_cmesh), 678  
label\_local\_entities() (in module sfepy.postprocess.plot\_cmesh), 678  
label\_points() (in module sfepy.postprocess.plot\_quadrature), 680  
LagrangeNodes (class in sfepy.discrete.fem.poly\_spaces), 614  
LagrangePolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
LagrangeSimplexBPolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
LagrangeSimplexPolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
LagrangeTensorProductPolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
lame\_from\_stiffness() (in module sfepy.mechanics.matcoefs), 656  
lame\_from\_youngpoisson() (in module sfepy.mechanics.matcoefs), 657  
laplace() (in module sfepy.linalg.sympy\_operators), 652  
LaplaceTerm (class in sfepy.terms.terms\_diffusion), 765  
LCBCOperator (class in sfepy.discrete.fem.lcbc\_operators), 603  
LCBCOperators (class in sfepy.discrete.fem.lcbc\_operators), 603  
leaveonlyphysicalsurfaces() (sfepy.mesh.geom\_tools.geometry method), 670  
leaveonlyphysicalvolumes() (sfepy.mesh.geom\_tools.geometry method), 670  
light\_copy() (sfepy.discrete.common.region.Region method), 591  
line (class in sfepy.mesh.geom\_tools), 671  
LinearCombinationBC (class in sfepy.discrete.conditions), 538  
LinearConvect2Term (class in sfepy.terms.terms\_navier\_stokes), 799  
LinearConvectTerm (class in sfepy.terms.terms\_navier\_stokes), 800  
LinearElasticETHTerm (class in sfepy.terms.terms\_elastic), 778  
LinearElasticIsotropicTerm (class in sfepy.terms.terms\_elastic), 779  
LinearElasticTerm (class in sfepy.terms.terms\_elastic), 780

LinearElasticTHTerm (class in sfepy.terms.terms\_elastic), 780  
linearize() (sfepy.discrete.fem.fields\_base.FEField method), 597  
LinearPointSpringTerm (class in sfepy.terms.terms\_point), 808  
LinearPrestressTerm (class in sfepy.terms.terms\_elastic), 781  
LinearSolver (class in sfepy.solvers.solvers), 712  
LinearStrainFiberTerm (class in sfepy.terms.terms\_elastic), 781  
LinearTractionTerm (class in sfepy.terms.terms\_surface), 812  
LinearVolumeForceTerm (class in sfepy.terms.terms\_volume), 814  
LineProbe (class in sfepy.discrete.probes), 551  
link\_duals() (sfepy.discrete.variables.Variables method), 574  
link\_flags() (sfepy.config.Config method), 514  
list\_dict() (in module sfepy.base.parse\_conf), 535  
list\_of() (in module sfepy.base.parse\_conf), 535  
listvar\_schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
listvar\_schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
listvar\_schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
load\_classes() (in module sfepy.base.base), 520  
load\_dict() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 515  
load\_library() (in module sfepy.solvers.ls\_mumps), 695  
load\_mumps\_libraries() (in module sfepy.solvers.ls\_mumps), 695  
load\_restart() (sfepy.discrete.problem.Problem method), 559  
LobattoTensorProductPolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
LOBPCGEigenvalueSolver (class in sfepy.solvers.eigen), 688  
locate\_files() (in module sfepy.base.ioutils), 529  
lock\_drilling\_rotations() (in module sfepy.mechanics.shell10x), 661  
Log (class in sfepy.base.log), 531  
LogPlotter (class in sfepy.base.log\_plotter), 532  
look\_ahead\_line() (in module sfepy.base.ioutils), 529  
LQuadraticEVPSolver (class in sfepy.solvers.qeigen), 710  
lredrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
lredrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
lredrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
in lrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
lrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
in lrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
lwk\_user (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
lwk\_user (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
lwk\_user (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702

## M

main() (in module blockgen), 507  
main() (in module convert\_mesh), 507  
main() (in module cylindergen), 507  
main() (in module edit\_identifiers), 508  
main() (in module eval\_ns\_forms), 508  
main() (in module eval\_tl\_forms), 508  
main() (in module extract\_edges), 509  
main() (in module extract\_surface), 509  
main() (in module extractor), 502  
main() (in module gen\_gallery), 510  
main() (in module gen\_iga\_patch), 510  
main() (in module gen\_lobatto1d\_c), 510  
main() (in module gen\_mesh\_prev), 511  
main() (in module gen\_release\_notes), 511  
main() (in module gen\_solver\_table), 511  
main() (in module gen\_term\_table), 511  
main() (in module homogen), 502  
main() (in module phonon), 502  
main() (in module plot\_condition\_numbers), 512  
main() (in module plot\_logs), 512  
main() (in module plot\_mesh), 512  
main() (in module plot\_quadratures), 512  
main() (in module plot\_times), 512  
main() (in module postproc), 503  
main() (in module probe), 504  
main() (in module run\_tests), 505  
main() (in module save\_basis), 513  
main() (in module sfepy.mesh.bspline), 669  
main() (in module sfepy.mesh.mesh\_generators), 674  
main() (in module show\_authors), 513  
main() (in module show\_mesh\_info), 513  
main() (in module show\_terms\_use), 513  
main() (in module simple), 505  
main() (in module sync\_module\_docs), 513  
main() (in module test\_install), 507  
main() (in module tile\_periodic\_mesh), 513  
make\_animation() (in module sfepy.postprocess.viewer), 688  
make\_axes() (sfepy.base.log\_plotter.LogPlotter method), 532

|                                                                       |                                                   |     |                                                                           |                                          |     |
|-----------------------------------------------------------------------|---------------------------------------------------|-----|---------------------------------------------------------------------------|------------------------------------------|-----|
| make_axis_rotation_matrix()                                           | (in module sfepy.linalg.geometry),                | 649 | match_coors()                                                             | (in module sfepy.discrete.fem.periodic), | 613 |
| make_format()                                                         | (sfepy.discrete.fem.meshio.ANSYSCDBMeshIO),       | 607 | match_grid_line()                                                         | (in module sfepy.discrete.fem.periodic), | 613 |
| make_full()                                                           | (sfepy.applications.evp_solver_app.EVPSolver),    | 515 | match_grid_plane()                                                        | (in module sfepy.discrete.fem.periodic), | 613 |
| make_full_vec()                                                       | (sfepy.discrete.equations.Equations),             | 542 | match_plane_by_dir()                                                      | (in module sfepy.discrete.fem.periodic), | 613 |
| make_full_vec()                                                       | (sfepy.discrete.evaluate.Evaluator),              | 544 | match_x_line()                                                            | (in module sfepy.discrete.fem.periodic), | 613 |
| make_full_vec()                                                       | (sfepy.discrete.variables.Variables),             | 574 | match_x_plane()                                                           | (in module sfepy.discrete.fem.periodic), | 613 |
| make_get_conf()                                                       | (in module sfepy.solvers.solvers),                | 713 | match_y_line()                                                            | (in module sfepy.discrete.fem.periodic), | 613 |
| make_global_operator()                                                | (sfepy.discrete.fem.lcbc_operators),              | 603 | LCBCOperators                                                             |                                          |     |
| make_h1_projection_data()                                             | (in module sfepy.discrete.projections),           | 564 | match_y_plane()                                                           | (in module sfepy.discrete.fem.periodic), | 613 |
| make_is_save()                                                        | (in module sfepy.discrete.problem),               | 564 | match_z_line()                                                            | (in module sfepy.discrete.fem.periodic), | 613 |
| make_knot_vector()                                                    | (sfepy.mesh.bspline.BSpline),                     | 667 | match_z_plane()                                                           | (in module sfepy.discrete.fem.periodic), | 613 |
| make_knot_vector()                                                    | (sfepy.mesh.bspline.BSplineSurface),              | 668 | Material (class in sfepy.discrete.materials),                             | 548                                      |     |
| make_l2_projection()                                                  | (in module sfepy.discrete.projections),           | 564 | Materials (class in sfepy.discrete.materials),                            | 549                                      |     |
| make_l2_projection_data()                                             | (in module sfepy.discrete.projections),           | 564 | MatrixAction (class in sfepy.linalg.utils),                               | 652                                      |     |
| make_line_matrix()                                                    | (in module sfepy.discrete.fem.facets),            | 594 | max_diff_csr()                                                            | (in module sfepy.linalg.utils),          | 654 |
| make_mesh()                                                           | (in module sfepy.discrete.fem.mesh),              | 607 | mbfg (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute),       | 593                                      |     |
| make_option_docstring()                                               | (in module sfepy.solvers.solvers),                | 713 | mblock (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute),               | 696                                      |     |
| make_square_matrix()                                                  | (in module sfepy.discrete.fem.facets),            | 594 | mblock (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),             | 699                                      |     |
| make_triangle_matrix()                                                | (in module sfepy.discrete.fem.facets),            | 594 | mblock (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),             | 702                                      |     |
| map_equations()                                                       | (sfepy.discrete.common.dof_info.Equation method), | 577 | mc2us() (in module edit_identifiers),                                     | 508                                      |     |
| map_permutations()                                                    | (in module sfepy.linalg.utils),                   | 654 | MeditMeshIO (class in sfepy.discrete.fem.meshio),                         | 609                                      |     |
| Mapping (class in sfepy.discrete.common.mappings),                    |                                                   | 587 | MEDMeshIO (class in sfepy.discrete.fem.meshio),                           | 609                                      |     |
| mapping (sfepy.solvers.ls_mumps.mumps_struct_c_4 attribute),          |                                                   | 696 | Merge_lines() (in module extract_edges),                                  | 509                                      |     |
| mapping (sfepy.solvers.ls_mumps.mumps_struct_c_5_0 attribute),        |                                                   | 699 | merge_mesh() (in module sfepy.discrete.fem.mesh),                         | 607                                      |     |
| mapping (sfepy.solvers.ls_mumps.mumps_struct_c_5_1 attribute),        |                                                   | 702 | Mesh (class in sfepy.discrete.fem.mesh),                                  | 606                                      |     |
| mark_subdomains()                                                     | (in module sfepy.parallel.plot_parallel_dofs),    | 676 | Mesh3DMeshIO (class in sfepy.discrete.fem.meshio),                        | 610                                      |     |
| mark_time()                                                           | (in module sfepy.base.base),                      | 520 | mesh_conn (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute),  | 593                                      |     |
| mask_points()                                                         | (sfepy.mechanics.contact_bodies.ContactPlane),    | 655 | mesh_coors (sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), | 593                                      |     |
| mask_points()                                                         | (sfepy.mechanics.contact_bodies.ContactSphere),   | 655 | mesh_from_groups()                                                        | (in module sfepy.discrete.fem.meshio),   | 613 |
| match_candidate()                                                     | (in module edit_identifiers),                     | 508 | MeshIO (class in sfepy.discrete.fem.meshio),                              | 610                                      |     |
| mode (sfepy.terms.terms_diffusion.AdvectDivFreeTerm attribute),       |                                                   | 763 | mini_newton() (in module sfepy.linalg.utils),                             | 654                                      |     |
| MiniAppBase (class in sfepy.homogenization.coefs_base),               |                                                   | 634 | MiniAppBase (class in sfepy.mechanics.contact_bodies),                    |                                          |     |
| CMapping (sfepy.discrete.common.extmods.mappings.CMapping attribute), |                                                   | 582 | Node (class in sfepy.mechanics.contact_bodies),                           |                                          |     |
| mode (sfepy.terms.terms_diffusion.AdvectDivFreeTerm attribute),       |                                                   | 763 | Point (class in sfepy.mechanics.contact_bodies),                          |                                          |     |

mode (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 768  
modes (sfepy.terms.terms\_biot.BiotETHTerm attribute), 758  
modes (sfepy.terms.terms\_biot.BiotTerm attribute), 760  
modes (sfepy.terms.terms\_biot.BiotTHTerm attribute), 759  
modes (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 761  
modes (sfepy.terms.terms\_diffusion.DiffusionCoupling attribute), 764  
modes (sfepy.terms.terms\_diffusion.DiffusionTerm attribute), 765  
modes (sfepy.terms.terms\_diffusion.LaplaceTerm attribute), 766  
modes (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 769  
modes (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 770  
modes (sfepy.terms.terms\_dot.ScalarDotMGradScalarTerm attribute), 772  
modes (sfepy.terms.terms\_dot.VectorDotGradScalarTerm attribute), 773  
modes (sfepy.terms.terms\_dot.VectorDotScalarTerm attribute), 774  
modes (sfepy.terms.terms\_elastic.ElasticWaveCauchyTerm attribute), 778  
modes (sfepy.terms.terms\_elastic.LinearElasticTerm attribute), 781  
modes (sfepy.terms.terms\_elastic.LinearPrestressTerm attribute), 781  
modes (sfepy.terms.terms\_elastic.NonsymElasticTerm attribute), 782  
modes (sfepy.terms.terms\_navier\_stokes.DivGradTerm attribute), 797  
modes (sfepy.terms.terms\_navier\_stokes.StokesTerm attribute), 803  
modes (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm attribute), 804  
modes (sfepy.terms.terms\_piezo.PiezoCouplingTerm attribute), 806  
modes (sfepy.terms.terms\_surface.LinearTractionTerm attribute), 812  
modes (sfepy.terms.terms\_surface.SurfaceNormalDotTerm attribute), 813  
MooneyRivlinTLTerm (class in sfepy.terms.terms\_hypereelastic\_tl), 789  
MooneyRivlinULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 794  
move\_control\_point() (sfepy.mesh.splinebox.SplineBox method), 676  
MRLCBCOperator (class in sfepy.discrete.fem.lcbc\_operators), 603  
Msh2MeshIO (class in sfepy.discrete.fem.meshio), 611  
msh\_cells (sfepy.discrete.fem.meshio.Msh2MeshIO attribute), 611  
mtx\_t (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
mulAB\_integrate() (in module sfepy.terms.extmods.terms), 818  
MultiProblem (class in sfepy.solvers.ls), 690  
mumps\_pcomplex (in module sfepy.solvers.ls\_mumps), 695  
mumps\_preal (in module sfepy.solvers.ls\_mumps), 695  
mumps\_struc\_c\_4 (class in sfepy.solvers.ls\_mumps), 695  
mumps\_struc\_c\_5\_0 (class in sfepy.solvers.ls\_mumps), 698  
mumps\_struc\_c\_5\_1 (class in sfepy.solvers.ls\_mumps), 701  
mumps\_struc\_c\_x (class in sfepy.solvers.ls\_mumps), 704  
MUMPSParallelSolver (class in sfepy.solvers.ls), 689  
MUMPSSolver (class in sfepy.solvers.ls), 689  
MumpsSolver (class in sfepy.solvers.ls\_mumps), 694  
MyQueue (class in sfepy.base.multiproc\_proc), 533

**N**

n (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 696  
n (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 699  
n (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 702  
n\_coor (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 580  
n\_el (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 581  
n\_el (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
n\_ep (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
n\_incident (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 579  
n\_qp (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
name (sfepy.discrete.fem.poly\_spaces.LagrangeSimplexBPolySpace attribute), 614  
name (sfepy.discrete.fem.poly\_spaces.LagrangeSimplexPolySpace attribute), 614  
name (sfepy.discrete.fem.poly\_spaces.LagrangeTensorProductPolySpace attribute), 614  
name (sfepy.discrete.fem.poly\_spaces.LobattoTensorProductPolySpace attribute), 614  
name (sfepy.solvers.eigen.LOBPCGEigenvalueSolver attribute), 688  
name (sfepy.solvers.eigen.PysparseEigenvalueSolver attribute), 689  
name (sfepy.solvers.eigen.ScipyEigenvalueSolver attribute), 689

name (sfepy.solvers.eigen.ScipySGEigenvalueSolver attribute), 689  
 name (sfepy.solvers.ls.MultiProblem attribute), 690  
 name (sfepy.solvers.ls.MUMPSParallelSolver attribute), 689  
 name (sfepy.solvers.ls.MUMPSSolver attribute), 690  
 name (sfepy.solvers.ls.PETScKrylovSolver attribute), 691  
 name (sfepy.solvers.ls.PyAMGKrylovSolver attribute), 691  
 name (sfepy.solvers.ls.PyAMGSolver attribute), 692  
 name (sfepy.solvers.ls.SchurMumps attribute), 692  
 name (sfepy.solvers.ls.ScipyDirect attribute), 692  
 name (sfepy.solvers.ls.ScipyIterative attribute), 693  
 name (sfepy.solvers.ls.ScipySuperLU attribute), 693  
 name (sfepy.solvers.ls.ScipyUmfpack attribute), 693  
 name (sfepy.solvers.nls.Newton attribute), 706  
 name (sfepy.solvers.nls.PETScNonlinearSolver attribute), 707  
 name (sfepy.solvers.nls.ScipyBroyden attribute), 707  
 name (sfepy.solvers.optimize.FMinSteepestDescent attribute), 708  
 name (sfepy.solvers.optimize.ScipyFMinSolver attribute), 709  
 name (sfepy.solvers.oseen.Oseen attribute), 710  
 name (sfepy.solvers.qeigen.LQuadraticEVPSolver attribute), 710  
 name (sfepy.solvers.semismooth\_newton.SemismoothNewton attribute), 711  
 name (sfepy.solvers.ts\_solvers.AdaptiveTimeSteppingSolver attribute), 714  
 name (sfepy.solvers.ts\_solvers.BatheTS attribute), 715  
 name (sfepy.solvers.ts\_solvers.GeneralizedAlphaTS attribute), 716  
 name (sfepy.solvers.ts\_solvers.NewmarkTS attribute), 717  
 name (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver attribute), 717  
 name (sfepy.solvers.ts\_solvers.StationarySolver attribute), 717  
 name (sfepy.solvers.ts\_solvers.VelocityVerletTS attribute), 718  
 name (sfepy.terms.terms.Term attribute), 740  
 name (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvect1Term attribute), 742  
 name (sfepy.terms.terms\_adj\_navier\_stokes.AdjConvect2Term attribute), 742  
 name (sfepy.terms.terms\_adj\_navier\_stokes.AdjDivGradTerm attribute), 743  
 name (sfepy.terms.terms\_adj\_navier\_stokes.NSOFMinGradTerm attribute), 743  
 name (sfepy.terms.terms\_adj\_navier\_stokes.NSOSurfMinDPressTerm attribute), 744  
 name (sfepy.terms.terms\_adj\_navier\_stokes.NSOSurfMinDPressOffTerm attribute), 744  
 name (sfepy.terms.terms\_adj\_navier\_stokes.NSOSurfMinDPressTHTerm attribute), 745  
 name (sfepy.terms.terms\_adj\_navier\_stokes.NSOSurfMinDPressTHTerm attribute), 745  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDConvectTerm attribute), 745  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDDivGradTerm attribute), 746  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDDivTerm attribute), 746  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDDotVolumeTerm attribute), 746  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDGradDivStabilizationTerm attribute), 747  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGCStabilizationTerm attribute), 748  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDPSPGPStabilizationTerm attribute), 748  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SDSUPGCStabilizationTerm attribute), 749  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SUPGCAdjStabilizationTerm attribute), 750  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj1StabilizationTerm attribute), 750  
 name (sfepy.terms.terms\_adj\_navier\_stokes.SUPGPAdj2StabilizationTerm attribute), 751  
 name (sfepy.terms.terms\_basic.IntegrateSurfaceMatTerm attribute), 752  
 name (sfepy.terms.terms\_basic.IntegrateSurfaceOperatorTerm attribute), 752  
 name (sfepy.terms.terms\_basic.IntegrateSurfaceTerm attribute), 753  
 name (sfepy.terms.terms\_basic.IntegrateVolumeMatTerm attribute), 753  
 name (sfepy.terms.terms\_basic.IntegrateVolumeOperatorTerm attribute), 754  
 name (sfepy.terms.terms\_basic.IntegrateVolumeTerm attribute), 755  
 name (sfepy.terms.terms\_basic.SumNodalValuesTerm attribute), 755  
 name (sfepy.terms.terms\_basic.SurfaceMomentTerm attribute), 755  
 name (sfepy.terms.terms\_basic.SurfaceTerm attribute), 756  
 name (sfepy.terms.terms\_basic.VolumeSurfaceTerm attribute), 756  
 name (sfepy.terms.terms\_basic.VolumeTerm attribute), 757  
 name (sfepy.terms.terms\_basic.ZeroTerm attribute), 757  
 name (sfepy.terms.terms\_biot.BiotETHTerm attribute), 758  
 name (sfepy.terms.terms\_biot.BiotStressTerm attribute), 758  
 name (sfepy.terms.terms\_biot.BiotTerm attribute), 760  
 name (sfepy.terms.terms\_biot.BiotTHTerm attribute), 759

name (sfepy.terms.terms\_constraints.NonPenetrationPenaltyTerm attribute), 761  
name (sfepy.terms.terms\_constraints.NonPenetrationTerm name (sfepy.terms.terms\_elastic.ElasticWaveCauchyTerm attribute), 778  
attribute), 761  
name (sfepy.terms.terms\_contact.ContactTerm attribute), 762  
name (sfepy.terms.terms\_diffusion.AdvectDivFreeTerm name (sfepy.terms.terms\_elastic.ElasticWaveTerm attribute), 778  
attribute), 763  
name (sfepy.terms.terms\_diffusion.ConvectVGradSTerm name (sfepy.terms.terms\_elastic.LinearElasticETHTerm attribute), 779  
attribute), 763  
name (sfepy.terms.terms\_diffusion.DiffusionCoupling name (sfepy.terms.terms\_elastic.LinearElasticIsotropicTerm attribute), 780  
attribute), 764  
name (sfepy.terms.terms\_diffusion.DiffusionRTerm name (sfepy.terms.terms\_elastic.LinearElasticTerm attribute), 781  
attribute), 764  
name (sfepy.terms.terms\_diffusion.DiffusionTerm name (sfepy.terms.terms\_elastic.LinearElasticTHTerm attribute), 780  
attribute), 765  
name (sfepy.terms.terms\_diffusion.DiffusionVelocityTerm name (sfepy.terms.terms\_elastic.LinearPrestressTerm attribute), 781  
attribute), 765  
name (sfepy.terms.terms\_diffusion.LaplaceTerm name (sfepy.terms.terms\_elastic.LinearStrainFiberTerm attribute), 782  
attribute), 766  
name (sfepy.terms.terms\_diffusion.SDDiffusionTerm name (sfepy.terms.terms\_elastic.NonsymElasticTerm attribute), 783  
attribute), 767  
name (sfepy.terms.terms\_diffusion.SurfaceFluxOperatorTerm name (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 784  
attribute), 767  
name (sfepy.terms.terms\_diffusion.SurfaceFluxTerm name (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientTerm attribute), 785  
attribute), 768  
name (sfepy.terms.terms\_dot.BCNewtonTerm name (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm attribute), 786  
attribute), 768  
name (sfepy.terms.terms\_dot.DotProductSurfaceTerm name (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm attribute), 787  
attribute), 769  
name (sfepy.terms.terms\_dot.DotProductVolumeTerm name (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 787  
attribute), 770  
name (sfepy.terms.terms\_dot.DotSProductVolumeOperator name (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 788  
attribute), 770  
name (sfepy.terms.terms\_dot.DotSProductVolumeOperator name (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm attribute), 789  
attribute), 771  
name (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm name (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm attribute), 789  
attribute), 771  
name (sfepy.terms.terms\_dot.ScalarDotMGradScalarTerm name (sfepy.terms.terms\_hyperelastic\_tl.SurfaceFluxTLTerm attribute), 790  
attribute), 772  
name (sfepy.terms.terms\_dot.VectorDotGradScalarTerm name (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 791  
attribute), 773  
name (sfepy.terms.terms\_dot.VectorDotScalarTerm name (sfepy.terms.terms\_hyperelastic\_tl.VolumeSurfaceTLTerm attribute), 791  
attribute), 774  
name (sfepy.terms.terms\_elastic.CauchyStrainSTerm name (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 792  
attribute), 774  
name (sfepy.terms.terms\_elastic.CauchyStrainTerm name (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm attribute), 792  
attribute), 775  
name (sfepy.terms.terms\_elastic.CauchyStressETHTerm name (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 793  
attribute), 776  
name (sfepy.terms.terms\_elastic.CauchyStressTerm name (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 793  
attribute), 777  
name (sfepy.terms.terms\_elastic.CauchyStressTHTerm name (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm attribute), 794  
attribute), 776

name (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm attribute), 794  
 name (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 795  
 name (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 796  
 name (sfepy.terms.terms\_navier\_stokes.ConvectTerm attribute), 796  
 name (sfepy.terms.terms\_navier\_stokes.DivGradTerm attribute), 797  
 name (sfepy.terms.terms\_navier\_stokes.DivOperatorTerm attribute), 798  
 name (sfepy.terms.terms\_navier\_stokes.DivTerm attribute), 798  
 name (sfepy.terms.terms\_navier\_stokes.GradDivStabilizationTerm attribute), 799  
 name (sfepy.terms.terms\_navier\_stokes.GradTerm attribute), 799  
 name (sfepy.terms.terms\_navier\_stokes.LinearConvect2Term attribute), 800  
 name (sfepy.terms.terms\_navier\_stokes.LinearConvectTerm attribute), 800  
 name (sfepy.terms.terms\_navier\_stokes.PSPGCStabilizationTerm attribute), 801  
 name (sfepy.terms.terms\_navier\_stokes.PSPGPStabilizationTerm attribute), 801  
 name (sfepy.terms.terms\_navier\_stokes.StokesTerm attribute), 803  
 name (sfepy.terms.terms\_navier\_stokes.StokesWaveDivTerm attribute), 804  
 name (sfepy.terms.terms\_navier\_stokes.StokesWaveTerm attribute), 804  
 name (sfepy.terms.terms\_navier\_stokes.SUPGCStabilizationTerm attribute), 802  
 name (sfepy.terms.terms\_navier\_stokes.SUPGPStabilizationTerm attribute), 802  
 name (sfepy.terms.terms\_navier\_stokes.SurfaceDivTerm attribute), 805  
 name (sfepy.terms.terms\_navier\_stokes.SurfaceGradTerm attribute), 805  
 name (sfepy.terms.terms\_piezo.PiezoCouplingTerm attribute), 806  
 name (sfepy.terms.terms\_piezo.PiezoStrainTerm attribute), 806  
 name (sfepy.terms.terms\_piezo.PiezoStressTerm attribute), 807  
 name (sfepy.terms.terms\_point.ConcentratedPointLoadTerm attribute), 807  
 name (sfepy.terms.terms\_point.LinearPointSpringTerm attribute), 808  
 name (sfepy.terms.terms\_shells.Shell10XTerm attribute), 810  
 name (sfepy.terms.terms\_surface.ContactPlaneTerm attribute), 811  
 name (sfepy.terms.terms\_surface.ContactSphereTerm attribute), 812  
 name (sfepy.terms.terms\_surface.LinearTractionTerm attribute), 812  
 name (sfepy.terms.terms\_surface.SDSurfaceIntegrateTerm attribute), 813  
 name (sfepy.terms.terms\_surface.SurfaceNormalDotTerm attribute), 813  
 name (sfepy.terms.terms\_surface.SurfaceJumpTerm attribute), 814  
 name (sfepy.terms.terms\_volume.LinearVolumeForceTerm attribute), 815  
 name\_to\_key() (in module sfepy.base.log), 531  
 nblock (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
 nblock (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
 nblock (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
 nel (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
 nel (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
 nel (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
 NeoHookeanTLTerm (class in sfepy.terms.hyperelastic\_tl), 789  
 NeoHookeanULTerm (class in sfepy.terms.hyperelastic\_ul), 794  
 NEUMeshIO (class in sfepy.discrete.fem.meshio), 612  
 new() (sfepy.terms.Term static method), 740  
 new\_ulf\_iteration() (sfepy.discrete.evaluate.Evaluator method), 544  
 new\_vtk\_polyline() (sfepy.postprocess.probes\_vtk.Probe method), 681  
 NewmarkTS (class in sfepy.solvers.ts\_solvers), 716  
 Newton (class in sfepy.solvers.nls), 704  
 nnz (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
 nnz\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
 NodalLCOperator (class in sfepy.discrete.fem.lcbc\_operators), 604  
 NodeDescription (class in sfepy.discrete.fem.poly\_spaces), 614  
 NonlinearSolver (class in sfepy.solvers.solvers), 712  
 NonPenetrationPenaltyTerm (class in sfepy.terms.constraints), 760  
 NonPenetrationTerm (class in sfepy.terms.constraints), 761  
 NonsymElasticTerm (class in sfepy.terms.terms\_elastic), 782  
 NoOptionsDocs (class in build\_helpers), 505

NoPenetrationOperator (class in sfepy.discrete.fem.lcbc\_operators), 603  
norm\_l2\_along\_axis() (in module sfepy.linalg.utils), 654  
normal (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
NormalDirectionOperator (class in sfepy.discrete.fem.lcbc\_operators), 604  
normalize\_time() (sfepy.solvers.ts.TimeStepper method), 713  
normalize\_vectors() (in module sfepy.linalg.utils), 654  
ncol (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
ncol (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
ncol (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 702  
nrow (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nrow (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
nrow (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
nrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
nrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
NSOFGMinGradTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 743  
NSOFSurfMinDPressDiffTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 743  
NSOFSurfMinDPressTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 744  
num (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 579  
num (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 581  
numpydoc\_path() (sfepy.config.Config method), 514  
NurbsPatch (class in sfepy.discrete.iga.domain), 618  
nz (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nz (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
nz (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
nz\_alloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nz\_alloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
nz\_alloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
nz\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nz\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 699  
nz\_loc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
nz\_rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
nz\_rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
nz\_rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703

**O**

object\_button\_quit\_changed() (sfepy.postprocess.viewer.ClosingHandler method), 685  
offset (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 579  
offsets (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 579  
OnesDim (class in sfepy.homogenization.coefs\_base), 635  
OneTypeList (class in sfepy.base.base), 516  
ooc\_prefix (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 696  
ooc\_prefix (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
ooc\_prefix (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
ooc\_tmpdir (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
ooc\_tmpdir (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
ooc\_tmpdir (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703

**P**

ordered\_iteritems() (in module sfepy.base.base), 520  
orient\_elements() (in module sfepy.discrete.common.extmods.cmesh), 581  
Oseen (class in sfepy.solvers.oseen), 709  
Output (class in sfepy.base.base), 517  
output (sfepy.base.log\_plotter.LogPlotter attribute), 532  
output\_array\_stats() (in module sfepy.linalg.utils), 654  
output\_mesh\_formats() (in module sfepy.discrete.fem.meshio), 613  
output\_step\_info() (sfepy.solvers.ts\_solvers.AdaptiveTimeSteppingSolver method), 714  
output\_step\_info() (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver method), 717  
OutputFilter (class in run\_tests), 504

package\_check() (in module build\_helpers), 506

par (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 par (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 par (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 par (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_x attribute), 704  
 parametrize() (sfepy.applications.application.Application method), 514  
 parse\_approx\_order() (in module sfepy.discrete.common.fields), 584  
 parse\_approx\_order() (in module sfepy.discrete.iga.fields), 623  
 parse\_definition() (in module sfepy.discrete.equations), 543  
 parse\_linearization() (in module extractor), 502  
 parse\_shape() (in module sfepy.discrete.common.fields), 584  
 ParseDomainSpecific (class in postproc), 503  
 ParseGroupNames (class in postproc), 503  
 ParseOpacity (class in postproc), 503  
 ParseRanges (class in postproc), 503  
 ParseRc (class in plot\_logs), 512  
 ParseRepeat (class in tile\_periodic\_mesh), 513  
 ParseResolution (class in postproc), 503  
 ParseSubdomains (class in postproc), 503  
 ParseView (class in postproc), 503  
 path\_of\_hdf5\_group() (in module sfepy.base.ioutils), 529  
 pause() (in module sfepy.base.base), 520  
 PDESolverApp (class in sfepy.applications.pde\_solver\_app), 515  
 PeriodicBC (class in sfepy.discrete.conditions), 538  
 perm\_in (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 perm\_in (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 perm\_in (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 permutations() (in module sfepy.linalg.utils), 654  
 petsc\_call() (in module sfepy.solvers.ls), 693  
 PETScKrylovSolver (class in sfepy.solvers.ls), 690  
 PETScNonlinearSolver (class in sfepy.solvers.nls), 706  
 PhaseVelocity (class in sfepy.homogenization.coefs\_phononic), 637  
 phonon (module), 502  
 PhysicalQPs (class in sfepy.discrete.common.mappings), 587  
 physicalsurface (class in sfepy.mesh.geom\_tools), 671  
 physicalvolume (class in sfepy.mesh.geom\_tools), 671  
 PiezoCouplingTerm (class in sfepy.terms.terms\_piezo), 805  
 PiezoStrainTerm (class in sfepy.terms.terms\_piezo), 806  
 PiezoStressTerm (class in sfepy.terms.terms\_piezo), 806  
 pivnul\_list (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 pivnul\_list (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 pivnul\_list (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 plot\_band\_gaps() (sfepy.homogenization.band\_gaps\_app.AcousticBandGap method), 631  
 plot\_bezier\_mesh() (in module sfepy.discrete.iga.plot\_nurbs), 629  
 plot\_bezier\_nurbs\_basis\_1d() (in module sfepy.discrete.iga.plot\_nurbs), 629  
 plot\_cmsh() (in module sfepy.postprocess.plot\_cmsh), 678  
 plot\_condition\_numbers (module), 512  
 plot\_control\_mesh() (in module sfepy.discrete.iga.plot\_nurbs), 629  
 plot\_data() (sfepy.base.log.Log method), 531  
 plot\_dispersion() (sfepy.homogenization.band\_gaps\_app.AcousticBandGap method), 631  
 plot\_displacements() (in module sfepy.postprocess.domain\_specific), 677  
 plot\_edges() (in module sfepy.postprocess.plot\_facets), 679  
 plot\_eigs() (in module sfepy.homogenization.band\_gaps\_app), 631  
 plot\_entities() (in module sfepy.postprocess.plot\_cmsh), 679  
 plot\_faces() (in module sfepy.postprocess.plot\_facets), 679  
 plot\_gap() (in module sfepy.homogenization.band\_gaps\_app), 631  
 plot\_gaps() (in module sfepy.homogenization.band\_gaps\_app), 632  
 plot\_geometry() (in module sfepy.postprocess.plot\_facets), 679  
 plot\_global\_dofs() (in module sfepy.postprocess.plot\_dofs), 679  
 plot\_iso\_lines() (in module sfepy.discrete.iga.plot\_nurbs), 629  
 plot\_local\_dofs() (in module sfepy.parallel.plot\_parallel\_dofs), 676  
 plot\_local\_dofs() (in module sfepy.postprocess.plot\_dofs), 679  
 plot\_log() (in module sfepy.base.log), 531  
 plot\_logs (module), 512  
 plot\_logs() (in module sfepy.homogenization.band\_gaps\_app), 632  
 plot\_matrix\_diff() (in module sfepy.base.plotutils), 535  
 plot\_mesh (module), 512  
 plot\_mesh() (in module sfepy.postprocess.plot\_dofs), 679  
 plot\_nodes() (in module sfepy.postprocess.plot\_dofs), 679

plot\_nurbs\_basis\_1d() (in module sfepy.discrete.iga.plot\_nurbs), 629  
plot\_parametric\_mesh() (in module sfepy.discrete.iga.plot\_nurbs), 629  
plot\_partitioning() (in module sfepy.parallel.plot\_parallel\_dofs), 677  
plot\_points() (in module sfepy.mechanics.contact\_bodies), 655  
plot\_points() (in module sfepy.postprocess.plot\_dofs), 679  
plot\_polygon() (in module sfepy.mechanics.contact\_bodies), 655  
plot\_polys() (in module gen\_lobatto1d\_c), 511  
plot\_quadrature() (in module sfepy.postprocess.plot\_quadrature), 680  
plot\_quadratures (module), 512  
plot\_times (module), 512  
plot\_velocity() (in module sfepy.postprocess.domain\_specific), 677  
plot\_vlines() (sfepy.base.log.Log method), 531  
plot\_warp\_scalar() (in module sfepy.postprocess.domain\_specific), 678  
plot\_weighted\_points() (in module sfepy.postprocess.plot\_quadrature), 680  
plot\_wireframe() (in module sfepy.postprocess.plot\_cmsh), 679  
point (class in sfepy.mesh.geom\_tools), 671  
points\_in\_poly() (sfepy.mesh.splinebox.SplineRegion2D static method), 676  
points\_in\_simplex() (in module sfepy.linalg.geometry), 650  
PointsProbe (class in sfepy.discrete.probes), 551  
PolarizationAngles (class in sfepy.homogenization.coefs\_phononic), 638  
poll\_draw() (sfepy.base.log\_plotter.LogPlotter method), 532  
poll\_file() (sfepy.postprocess.sources.FileSource method), 681  
poly\_space\_base (sfepy.terms.terms\_shells.Shell10XTerm attribute), 810  
PolySpace (class in sfepy.discrete.fem.poly\_spaces), 614  
post\_process() (sfepy.homogenization.coefs\_phononic.SchuhFEMConf method), 638  
post\_process() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 638  
postproc (module), 502  
postprocess() (in module probe), 504  
prefix (sfepy.base.base.Output attribute), 517  
prepare\_cylindrical\_transform() (in module sfepy.mechanics.tensors), 662  
prepare\_matrices() (sfepy.homogenization.coefs\_phononic.SchuhFEMConf method), 638  
prepare\_matrices() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 638

module prepare\_matrix() (in module sfepy.discrete.problem), 564  
module prepare\_remap() (in module sfepy.discrete.fem.utils), 617  
module prepare\_translate() (in module sfepy.discrete.fem.utils), 617  
module presolve() (sfepy.homogenization.coefs\_base.PressureEigenvalueProblem method), 635  
module presolve() (sfepy.solvers.ls.MUMPSSolver method), 690  
presolve() (sfepy.solvers.ls.ScipyDirect method), 693  
presolve() (sfepy.solvers.solvers.LinearSolver method), 712  
PressureEigenvalueProblem (class in sfepy.homogenization.coefs\_base), 635  
PressureRHSVector (class in sfepy.homogenization.coefs\_elastic), 635  
print\_array\_info() (in module sfepy.linalg.utils), 655  
print\_leaf() (in module sfepy.discrete.parse\_regions), 550  
print\_matrix\_diff() (in module sfepy.base.plotutils), 535  
print\_mem\_usage() (in module sfepy.base.mem\_usage), 533  
print\_names() (sfepy.base.base.Container method), 516  
print\_names() (sfepy.base.base.OneTypeList method), 517  
print\_op() (in module sfepy.discrete.parse\_regions), 550  
print\_solvers() (in module simple), 505  
print\_stack() (in module sfepy.discrete.parse\_regions), 550  
print\_structs() (in module sfepy.base.base), 520  
print\_terms() (in module simple), 505  
print\_terms() (sfepy.discrete.equations.Equations method), 542  
printinfo() (sfepy.mesh.geom\_tools.geometry method), 670  
prism2hexa (sfepy.discrete.fem.meshio.Msh2MeshIO attribute), 611  
Probe (class in sfepy.discrete.probes), 551  
Probe (class in sfepy.postprocess.probes\_vtk), 680  
probe (module), 503  
probe() (sfepy.discrete.probes.Probe method), 552  
ProbeFromFile (class in sfepy.postprocess.probes\_vtk), 681  
Problem (class in sfepy.discrete.problem), 554  
FEMConf (class in sfepy.base.conf), 524  
process\_command() (sfepy.base.log\_plotter.LogPlotter SimpleEVP method), 532  
process\_conf() (sfepy.solvers.solvers.Solver class method), 712  
process\_options() (sfepy.applications.evp\_solver\_app.EVPSolverApp static method), 515  
process\_options() (sfepy.applications.pde\_solver\_app.PDESolverApp static method), 515  
process\_options() (sfepy.applications.evp\_solver\_app.EVPSolverApp static method), 515  
process\_options() (sfepy.applications.pde\_solver\_app.PDESolverApp static method), 515  
SimpleEVPOptions() (sfepy.homogenization.band\_gaps\_app.AcousticBandGap static method), 631  
SimpleEVPOptions() (sfepy.homogenization.coefs\_base.MiniAppBase method), 635

**P**  
 process\_options() (sfepy.homogenization.coefs\_phononic.BandGaps method), 637  
 process\_options() (sfepy.homogenization.coefs\_phononic.CholeskyDenseTension method), 637  
 process\_options() (sfepy.homogenization.coefs\_phononic.Eigenvalues method), 637  
 process\_options() (sfepy.homogenization.coefs\_phononic.PressVelocity method), 638  
 process\_options() (sfepy.homogenization.coefs\_phononic.PotentialAngle method), 638  
 process\_options() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 638  
 process\_options() (sfepy.homogenization.engine.Homogenization) (sfepy.discrete.fem.meshio.MEDMeshIO method), static method), 640  
 process\_options() (sfepy.homogenization.homogen\_app.Homogenization) (sfepy.discrete.fem.meshio.Mesh3DMeshIO method), static method), 643  
 process\_options\_pv() (sfepy.homogenization.band\_gaps\_appeared) (sfepy.discrete.fem.meshio.MeshIO method), static method), 631  
 process\_reqs\_coefs() (sfepy.homogenization.engine.HomogenizationWorkerMulti static method), 642  
 project\_by\_component() (in module sfepy.discrete.projections), 564  
 ps (sfepy.common.extmods.mappings.CMapping attribute), 582  
 PSPGCStabilizationTerm (class in sfepy.terms.terms\_navier\_stokes), 800  
 PSPGPStabilizationTerm (class in sfepy.terms.terms\_navier\_stokes), 801  
 put() (sfepy.base.multiproc\_proc.MyQueue method), 533  
 PyAMGKrylovSolver (class in sfepy.solvers.ls), 691  
 PyAMGSolver (class in sfepy.solvers.ls), 691  
 PysparseEigenvalueSolver (class in sfepy.solvers.eigen), 688  
 python\_include() (sfepy.config.Config method), 514  
 python\_shell() (in module sfepy.base.base), 520  
 python\_version() (sfepy.config.Config method), 514

**Q**

qp (sfepy.common.extmods.mappings.CMapping attribute), 582  
 QuadraticEVPSolver (class in sfepy.solvers.solvers), 712  
 QuadraturePoints (class in sfepy.discrete.quadratures), 565  
 Quantity (class in sfepy.mechanics.units), 663

**R**

R (sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 619  
 RayProbe (class in sfepy.discrete.probes), 552  
 read() (sfepy.discrete.fem.meshio.AbaqusMeshIO method), 608  
 read() (sfepy.discrete.fem.meshio.ANSYSCDBMeshIO method), 607

read() (sfepy.discrete.fem.meshio.AVSUCDMeshIO method), 608  
 read() (sfepy.discrete.fem.meshio.ComsolMeshIO method), 608  
 read() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
 read() (sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO method), 609  
 read() (sfepy.discrete.fem.meshio.MeditMeshIO method), 610  
 read() (sfepy.discrete.fem.meshio.MEDMeshIO method), 609  
 read() (sfepy.discrete.fem.meshio.Mesh3DMeshIO method), 610  
 read() (sfepy.discrete.fem.meshio.MeshIO method), 611  
 read() (sfepy.discrete.fem.meshio.Msh2MeshIO method),  
 read() (sfepy.discrete.fem.meshio.NEUMeshIO method), 612  
 read() (sfepy.discrete.fem.meshio.TetgenMeshIO method), 612  
 read() (sfepy.discrete.fem.meshio.UserMeshIO method), 612  
 read() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613  
 read\_array() (in module sfepy.base.ioutils), 529  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.ANSYSCDBMeshIO method), 608  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.MeditMeshIO method), 610  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.MeshIO method), 611  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.Msh2MeshIO method), 612  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.TetgenMeshIO method), 612  
 read\_bounding\_box() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613  
 read\_common() (sfepy.postprocess.sources.GenericFileSource method), 682  
 read\_common() (sfepy.postprocess.sources.GenericSequenceFileSource method), 682  
 read\_coors() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613  
 read\_data() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
 read\_data() (sfepy.discrete.fem.meshio.MeshIO method), 611  
 read\_data() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613

read\_data\_header() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
read\_dict\_hdf5() (in module sfepy.base.ioutils), 529  
read\_dimension() (sfepy.discrete.fem.meshio.AbaqusMeshIO method), 608  
read\_dimension() (sfepy.discrete.fem.meshio.ANSYSDBMeshIO method), 608  
read\_dimension() (sfepy.discrete.fem.meshio.AVSUCDMeshIO method), 608  
read\_dimension() (sfepy.discrete.fem.meshio.BDFMeshIO method), 608  
read\_dimension() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
read\_dimension() (sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO method), 609  
read\_dimension() (sfepy.discrete.fem.meshio.MeditMeshIO method), 610  
read\_dimension() (sfepy.discrete.fem.meshio.Mesh3DMeshIO method), 610  
read\_dimension() (sfepy.discrete.fem.meshio.MeshIO method), 611  
read\_dimension() (sfepy.discrete.fem.meshio.Msh2MeshIO method), 612  
read\_dimension() (sfepy.discrete.fem.meshio.NEUMeshIO method), 612  
read\_dimension() (sfepy.discrete.fem.meshio.TetgenMeshIO method), 612  
read\_dimension() (sfepy.discrete.fem.meshio.VTKMeshIO method), 613  
read\_domain\_from\_hdf5() (sfepy.discrete.iga.domain.IGDomain static method), 618  
read\_from\_hdf5() (in module sfepy.base.ioutils), 529  
read\_header() (in module sfepy.discrete.probes), 553  
read\_iga\_data() (in module sfepy.discrete.iga.io), 628  
read\_last\_step() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 608  
read\_last\_step() (sfepy.discrete.fem.meshio.MeshIO method), 611  
read\_list() (in module sfepy.base.ioutils), 529  
read\_log() (in module sfepy.base.log), 532  
read\_mesh\_from\_hdf5() (sfepy.discrete.fem.meshio.HDF5MeshIO static method), 609  
read\_results() (in module sfepy.discrete.probes), 553  
read\_sparse\_matrix\_from\_hdf5() (in module sfepy.base.ioutils), 529  
read\_sparse\_matrix\_hdf5() (in module sfepy.base.ioutils), 530  
read\_time\_history() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 609  
read\_time stepper() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 609  
read\_times() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 609  
read\_times() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 611  
read\_token() (in module sfepy.base.ioutils), 530  
read\_variables\_time\_history() (sfepy.discrete.fem.meshio.HDF5MeshIO Reader (class in sfepy.base.reader), 535  
read\_over\_bones() (in sfepy.homogenization.recovery), 645  
recover\_micro\_hook() (in sfepy.homogenization.recovery), 645  
recover\_micro\_hook\_eps() (in sfepy.homogenization.recovery), 645  
recover\_glob() (in module build\_helpers), 506  
redrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
redrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
redrhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
reduce\_on\_datas() (sfepy.discrete.materials.Material method), 548  
refine() (in module sfepy.discrete.fem.refine\_hanging), 616  
refine() (sfepy.discrete.fem.domain.FEDomain method), 592  
refine\_2\_3() (in module sfepy.discrete.fem.refine), 615  
refine\_2\_4() (in module sfepy.discrete.fem.refine), 616  
refine\_3\_4() (in module sfepy.discrete.fem.refine), 616  
refine\_3\_8() (in module sfepy.discrete.fem.refine), 616  
refine\_mesh() (in module sfepy.discrete.fem.utils), 618  
refine\_pars() (sfepy.discrete.probes.Probe static method), 552  
refine\_points() (sfepy.discrete.probes.PointsProbe method), 551  
refine\_points() (sfepy.discrete.probes.Probe method), 552  
refine\_points() (sfepy.discrete.probes.RayProbe method), 553  
refine\_reference() (in module sfepy.discrete.fem.refine), 616  
refine\_region() (in sfepy.discrete.fem.refine\_hanging), 617  
refine\_uniformly() (sfepy.discrete.problem.Problem method), 559  
Region (class in sfepy.discrete.common.region), 589  
region\_leaf() (in module sfepy.discrete.common.domain), 578  
region\_op() (in module sfepy.discrete.common.domain), 578  
ReloadSource (class in sfepy.postprocess.viewer), 685  
remap\_dict() (in module sfepy.base.base), 520

remove\_bcs() (sfepy.discrete.problem.Problem method), 560  
 remove\_extra\_dofs() (sfepy.discrete.fem.fields\_base.FEFieldhs method), 598  
 remove\_extra\_dofs() (sfepy.discrete.fem.fields\_nodal.H1DiscontinuousFEField method), 600  
 remove\_files() (in module sfepy.base.ioutils), 530  
 remove\_files\_patterns() (in module sfepy.base.ioutils), 530  
 remove\_known() (in module sfepy.base.resolve\_deps), 536  
 remove\_name() (sfepy.base.base.Container method), 516  
 render\_scene() (sfepy.postprocess.viewer.Viewer method), 687  
 replace() (in module sfepy.discrete.parse\_regions), 550  
 replace\_with\_region() (in module sfepy.discrete.parse\_regions), 550  
 report() (in module test\_install), 507  
 report() (sfepy.base.testing.TestCommon static method), 536  
 report() (sfepy.discrete.probes.CircleProbe method), 551  
 report() (sfepy.discrete.probes.LineProbe method), 551  
 report() (sfepy.discrete.probes.PointsProbe method), 551  
 report() (sfepy.discrete.probes.Probe method), 552  
 report() (sfepy.discrete.probes.RayProbe method), 553  
 report2() (in module test\_install), 507  
 report\_tests() (in module test\_install), 507  
 reset() (sfepy.discrete.materials.Material method), 548  
 reset() (sfepy.discrete.materials.Materials method), 549  
 reset() (sfepy.discrete.problem.Problem method), 560  
 reset() (sfepy.discrete.variables.Variable static method), 573  
 reset() (sfepy.postprocess.sources.FileSource method), 681  
 reset\_materials() (sfepy.discrete.equations.Equations method), 542  
 reset\_refinement() (sfepy.discrete.probes.Probe method), 552  
 reset\_regions() (sfepy.discrete.common.domain.Domain method), 578  
 reset\_view() (sfepy.postprocess.viewer.Viewer method), 687  
 resolve() (in module sfepy.base.resolve\_deps), 536  
 resolve\_chains() (in module sfepy.discrete.common.dof\_info), 578  
 restore() (sfepy.applications.application.Application method), 514  
 restore\_dofs() (sfepy.discrete.fem.fields\_base.FEField method), 598  
 restore\_step\_time() (sfepy.solvers.ts.TimeStepper method), 713  
 restore\_substituted() (sfepy.discrete.fem.fields\_base.FEField method), 598

rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rhs (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 rhs() (in module sfepy.discrete.parse\_equations), 550  
 rhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 rhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rhs\_sparse (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 RigidOperator (class in sfepy.discrete.fem.lcbc\_operators), 604  
 rinfo (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 rinfo (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rinfo (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 rinfofog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 rinfog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rinfog (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 rm\_multi() (in module sfepy.homogenization.engine), 642  
 rotate\_elastic\_tensor() (in module sfepy.mechanics.shell10x), 661  
 rotation\_matrix2d() (in module sfepy.linalg.geometry), 650  
 rowsca (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 rowsca (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rowsca (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 rowsca\_from\_mumps (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700  
 rowsca\_from\_mumps (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703  
 run() (build\_helpers.Clean method), 505  
 run() (build\_helpers.DoxygenDocs method), 505  
 run() (build\_helpers.SphinxHTMLDocs method), 506  
 run() (build\_helpers.SphinxPDFDocs method), 506  
 run() (sfepy.base.testing.TestCommon method), 536  
 run\_test() (in module run\_tests), 505  
 run\_tests (module), 504

**S**

save() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 634

save() (sfepy.homogenization.coefs\_base.TCorrectorsViaPressureScalarTerm method), 635

save() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 638

save\_animation() (sfepy.postprocess.viewer.Viewer method), 687

save\_as\_mesh() (sfepy.discrete.variables.FieldVariable method), 571

save\_basis (module), 513

save\_basis() (in module sfepy.discrete.iga.utils), 630

save\_basis\_on\_mesh() (in module save\_basis), 513

save\_dict() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 515

save\_dir (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703

save\_ebc() (sfepy.discrete.problem.Problem method), 560

save\_field\_meshes() (sfepy.discrete.problem.Problem method), 560

save\_image() (sfepy.postprocess.viewer.Viewer method), 687

save\_log() (sfepy.homogenization.coefs\_phononic.BandGaps static method), 637

save\_mappings() (sfepy.discrete.common.fields.Field method), 584

save\_only() (in module sfepy.applications.pde\_solver\_app), 515

save\_options() (in module sfepy.base.ioutils), 530

save\_prefix (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703

save\_raw\_bg\_logs() (in module sfepy.homogenization.band\_gaps\_app), 632

save\_recovery\_region() (in module sfepy.homogenization.recovery), 645

save\_regions() (sfepy.discrete.common.domain.Domain method), 578

save\_regions() (sfepy.discrete.problem.Problem method), 560

save\_regions\_as\_groups() (sfepy.discrete.common.domain.Domain method), 578

save\_regions\_as\_groups() (sfepy.discrete.problem.Problem method), 560

save\_restart() (sfepy.discrete.problem.Problem method), 560

save\_results() (sfepy.applications.evp\_solver\_app.EVPSolver method), 515

save\_sparse\_txt() (in module sfepy.linalg.sparse), 651

save\_state() (sfepy.discrete.problem.Problem method), 561

save\_time\_history() (in module sfepy.postprocess.time\_history), 683

ScalarDotGradIScalarTerm (class in sfepy.terms.terms\_dot), 771

SDPMGradScalarTerm (class in sfepy.terms.terms\_dot), 772

scale\_matrix() (in module sfepy.solvers.oseen), 710

schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697

schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700

schur (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 703

schur\_lld (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697

schur\_lld (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700

schur\_lld (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 704

schur\_mloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697

schur\_mloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700

schur\_mloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 704

schur\_nloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697

schur\_nloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 700

schur\_nloc (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 704

SchurEVP (class in sfepy.homogenization.coefs\_phononic), 638

SchurMumps (class in sfepy.solvers.ls), 692

ScipyBroyden (class in sfepy.solvers.nls), 707

ScipyDirect (class in sfepy.solvers.ls), 692

ScipyEigenvalueSolver (class in sfepy.solvers.eigen), 689

ScipyFMinSolver (class in sfepy.solvers.optimize), 708

ScipyIterative (class in sfepy.solvers.ls), 693

ScipySGEEigenvalueSolver (class in sfepy.solvers.eigen), 689

ScipySuperLU (class in sfepy.solvers.ls), 693

ScipyUmfpack (class in sfepy.solvers.ls), 693

SDConvectTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 744

SDDiffusionTerm (class in sfepy.terms.terms\_diffusion), 766

SDDivGradTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 745

SDDivTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 745

SDDotVolumeTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 746

SDGradDivStabilizationTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 747

SDLinearElasticTerm (class in sfepy.terms.terms\_elastic), 783

|                          |                                                             |                                                                                      |     |                                                                             |     |
|--------------------------|-------------------------------------------------------------|--------------------------------------------------------------------------------------|-----|-----------------------------------------------------------------------------|-----|
| SDPSPGCStabilizationTerm | (class<br>sfepy.terms.terms_adj_navier_stokes),             | 747                                                                                  | in  | set_arg_types() (sfepy.terms.terms_navier_stokes.StokesTerm<br>method),     | 803 |
| SDPSPGPStabilizationTerm | (class<br>sfepy.terms.terms_adj_navier_stokes),             | 748                                                                                  | in  | set_arg_types() (sfepy.terms.terms_piezo.PiezoCouplingTerm<br>method),      | 806 |
| SDSurfaceIntegrateTerm   | (class<br>sfepy.terms.terms_surface),                       | 812                                                                                  | in  | set_arg_types() (sfepy.terms.terms_surface.LinearTractionTerm<br>method),   | 812 |
| SDSUPGStabilizationTerm  | (class<br>sfepy.terms.terms_adj_navier_stokes),             | 748                                                                                  | in  | set_arg_types() (sfepy.terms.terms_surface.SurfaceNormalDotTerm<br>method), | 813 |
| select_bcs()             | (sfepy.discrete.problem.Problem<br>method),                 | 561                                                                                  |     | set_axes_font_size() (in module sfepy.base.plotutils),                      | 535 |
| select_by_names()        | (in module sfepy.base.base),                                | 520                                                                                  |     | set_basis_indices() (sfepy.discrete.fem.mappings.SurfaceMapping<br>method), | 605 |
| select_materials()       | (sfepy.discrete.problem.Problem<br>method),                 | 561                                                                                  |     | set_basis_transform() (sfepy.discrete.fem.fields_base.FEField<br>method),   | 598 |
| select_variables()       | (sfepy.discrete.problem.Problem<br>method),                 | 561                                                                                  |     | set_bcs() (sfepy.discrete.problem.Problem method),                          | 561 |
| SemismoothNewton         | (class<br>sfepy.solvers.semismooth_newton),                 | 711                                                                                  | in  | set_colormap() (in module sfepy.postprocess.viewer),                        | 688 |
| separate()               | (sfepy.mesh.geom_tools.surface method),                     | 671                                                                                  |     | set_conf_solvers() (sfepy.discrete.problem.Problem<br>method),              | 561 |
| set_accuracy()           | (in module sfepy.discrete.fem.mesh),                        | 607                                                                                  |     | set_constant() (sfepy.discrete.variables.Variable method),                  | 573 |
| set_accuracy()           | (in module sfepy.discrete.fem.periodic),                    | 613                                                                                  |     | set_control_points() (sfepy.mesh.bspline.BSpline<br>method),                | 667 |
| set_adof_conn()          | (sfepy.discrete.variables.Variables<br>method),             | 574                                                                                  |     | set_control_points() (sfepy.mesh.bspline.BSplineSurf<br>method),            | 669 |
| set_all_data()           | (sfepy.discrete.materials.Material method),                 | 548                                                                                  |     | set_control_points() (sfepy.mesh.splinebox.SplineBox<br>method),            | 676 |
| set_approx_points()      | (sfepy.mesh.bspline.BSpline<br>method),                     | 667                                                                                  |     | set_coors() (sfepy.discrete.fem.fields_base.FEField<br>method),             | 598 |
| set_approx_points()      | (sfepy.mesh.bspline.BSplineSurf<br>method),                 | 668                                                                                  |     | set_data() (sfepy.discrete.equations.Equations method),                     | 542 |
| set_arg_types()          | (sfepy.terms.terms.Term method),                            | 740                                                                                  |     | set_data() (sfepy.discrete.materials.Material method),                      | 548 |
| set_arg_types()          | (sfepy.terms.terms_biot.BiotTerm<br>method),                | 760                                                                                  |     | set_data() (sfepy.discrete.variables.Variable method),                      | 573 |
| set_arg_types()          | (sfepy.terms.terms_diffusion.DiffusionCoupling<br>method),  | 764                                                                                  |     | set_data() (sfepy.discrete.variables.Variables method),                     | 575 |
| set_arg_types()          | (sfepy.terms.terms_diffusion.DiffusionTerm<br>method),      | 765                                                                                  |     | set_default() (sfepy.base.base.Struct method),                              | 517 |
| set_arg_types()          | (sfepy.terms.terms_diffusion.LaplaceTerm<br>method),        | 766                                                                                  |     | set_defaults() (in module sfepy.base.base),                                 | 520 |
| set_arg_types()          | (sfepy.terms.terms_dot.DotProductVolumeT                    | set_dofs() (sfepy.discrete.fem.fields_hierarchic.H1HierarchicVolumeField<br>method), | 600 | set_dim() (in module sfepy.linalg.sympy_operators),                         | 652 |
| set_arg_types()          | (sfepy.terms.terms_dot.DotProductVolumeT<br>method),        | 770                                                                                  |     | set_dofs() (sfepy.discrete.fem.fields_nodal.H1NodalMixin<br>method),        | 601 |
| set_arg_types()          | (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm<br>method), | 772                                                                                  |     | set_igf_field() (sfepy.discrete.iga.fields.IGField method),                 | 623 |
| set_arg_types()          | (sfepy.terms.terms_dot.VectorDotGradScalarTerm<br>method),  | 773                                                                                  |     | set_equations() (sfepy.discrete.problem.Problem<br>method),                 | 561 |
| set_arg_types()          | (sfepy.terms.terms_dot.VectorDotScalarTerm<br>method),      | 774                                                                                  |     | set_equations_instance() (sfepy.discrete.problem.Problem<br>method),        | 561 |
| set_arg_types()          | (sfepy.terms.terms_elastic.LinearElasticTerm<br>method),    | 781                                                                                  |     | set_extra_args() (sfepy.discrete.functions.Function<br>method),             | 561 |
| set_arg_types()          | (sfepy.terms.terms_elastic.LinearPrestressTerm<br>method),  | 781                                                                                  |     | set_extra_args() (sfepy.discrete.materials.Material<br>method),             | 546 |
| set_arg_types()          | (sfepy.terms.terms_elastic.NonsymElasticTerm<br>method),    | 783                                                                                  |     | set_field_split() (sfepy.solvers.ls.PETScKrylovSolver<br>method),           | 548 |
| set_arg_types()          | (sfepy.terms.terms_navier_stokes.DivGradTerm<br>method),    | 797                                                                                  |     | set_field_split() (sfepy.solvers.solvers.Solver<br>method),                 | 691 |

set\_fields() (sfepy.discrete.problem.Problem method), set\_mesh\_coors() (sfepy.discrete.problem.Problem method), 561  
set\_filename() (sfepy.postprocess.sources.GenericFileSource method), 682  
set\_filename() (sfepy.postprocess.sources.GenericSequenceFile method), 682  
set\_filename() (sfepy.postprocess.sources.VTKFileSource method), 682  
set\_filename() (sfepy.postprocess.sources.VTKSequenceFile method), 682  
set\_float\_format() (sfepy.discrete.fem.meshio.MeshIO method), 611  
set\_from\_data() (sfepy.solvers.ts.TimeStepper method), 713  
set\_from\_data() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_from\_function() (sfepy.discrete.variables.FieldVariable method), 571  
set\_from\_mesh\_vertices() (sfepy.discrete.variables.FieldVariable method), 572  
set\_from\_other() (sfepy.discrete.variables.FieldVariable method), 572  
set\_from\_qp() (sfepy.discrete.variables.FieldVariable method), 572  
set\_from\_state() (sfepy.discrete.variables.Variables method), 575  
set\_from\_ts() (sfepy.solvers.ts.TimeStepper method), 713  
set\_from\_ts() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_full() (sfepy.discrete.state.State method), 568  
set\_function() (sfepy.discrete.functions.Function method), 546  
set\_function() (sfepy.discrete.materials.Material method), 548  
set\_ics() (sfepy.discrete.problem.Problem method), 561  
set\_integral() (sfepy.terms.terms.Term method), 740  
set\_integral() (sfepy.terms.terms\_shells.Shell10XTerm method), 810  
set\_kind() (sfepy.discrete.common.region.Region method), 591  
set\_kind\_tdim() (sfepy.discrete.common.region.Region method), 591  
set\_knot\_vector() (sfepy.mesh.bspline.BSpline method), 668  
set\_linear() (sfepy.discrete.problem.Problem method), 561  
set\_local\_entities() (sfepy.discrete.common.extmods.cmesh.CMesh method), 581  
set\_materials() (sfepy.discrete.problem.Problem method), 561  
set\_mesh\_coors() (in module sfepy.discrete.fem.fields\_base), 600  
set\_mesh\_coors() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
set\_micro\_coors() (sfepy.homogenization.engine.HomogenizationEngine method), 641  
set\_n\_digit\_from\_min\_dt() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_n\_point() (sfepy.discrete.probes.Probe method), 552  
set\_nonlin\_states() (in module sfepy.homogenization.utils), 647  
set\_options() (sfepy.discrete.probes.Probe method), 552  
set\_output() (sfepy.base.base.Output method), 517  
set\_output\_dir() (sfepy.discrete.problem.Problem method), 561  
set\_output\_prefix() (sfepy.base.base.Output method), 517  
set\_param() (sfepy.mesh.bspline.BSpline method), 668  
set\_param\_n() (sfepy.mesh.bspline.BSpline method), 668  
set\_param\_n() (sfepy.mesh.bspline.BSplineSurf method), 669  
set\_parts() (sfepy.discrete.state.State method), 568  
set\_rcd\_centralized() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
set\_reduced() (sfepy.discrete.state.State method), 568  
set\_regions() (sfepy.discrete.problem.Problem method), 561  
set\_rhs() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
set\_section() (in module gen\_term\_table), 512  
set\_silent() (sfepy.solvers.ls\_mumps.MumpsSolver method), 694  
set\_solver() (sfepy.discrete.problem.Problem method), 561  
set\_source\_filename() (sfepy.postprocess.viewer.Viewer method), 687  
set\_state() (sfepy.solvers.ts.TimeStepper method), 713  
set\_state() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_state\_part() (sfepy.discrete.variables.Variables method), 575  
set\_step() (sfepy.solvers.ts.TimeStepper method), 713  
set\_step() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_substep\_time() (sfepy.solvers.ts.TimeStepper method), 713  
set\_time\_step() (sfepy.solvers.ts.VariableTimeStepper method), 714  
set\_variables() (sfepy.discrete.problem.Problem method), 562



setup\_output() (sfepy.discrete.problem.Problem method), 562  
setup\_output() (sfepy.homogenization.coeffs\_base.CorrMinimap method), 634  
setup\_output\_info() (sfepy.applications.pde\_solver\_app.PDESetupApp method), 515  
setup\_point\_data() (sfepy.discrete.fem.fields\_base.VolumeField method), 599  
setup\_surface\_data() (sfepy.discrete.fem.fields\_base.VolumeField method), 599  
sfepy.applications.application (module), 514  
sfepy.applications.evp\_solver\_app (module), 515  
sfepy.applications.pde\_solver\_app (module), 515  
sfepy.base.base (module), 516  
sfepy.base.compat (module), 521  
sfepy.base.conf (module), 524  
sfepy.base.getch (module), 526  
sfepy.base.goptions (module), 526  
sfepy.base.ioutils (module), 527  
sfepy.base.log (module), 531  
sfepy.base.log\_plotter (module), 532  
sfepy.base.mem\_usage (module), 532  
sfepy.base.multiproc (module), 533  
sfepy.base.multiproc\_proc (module), 533  
sfepy.base.parse\_conf (module), 534  
sfepy.base.plotutils (module), 535  
sfepy.base.reader (module), 535  
sfepy.base.resolve\_deps (module), 536  
sfepy.base.testing (module), 536  
sfepy.config (module), 514  
sfepy.discrete.common.dof\_info (module), 576  
sfepy.discrete.common.domain (module), 578  
sfepy.discrete.common.extmods.\_fmfield (module), 578  
sfepy.discrete.common.extmods.\_geommech (module), 578  
sfepy.discrete.common.extmods.assemble (module), 579  
sfepy.discrete.common.extmods.cmesh (module), 579  
sfepy.discrete.common.extmods.crefcoors (module), 581  
sfepy.discrete.common.extmods.mappings (module), 582  
sfepy.discrete.common.fields (module), 582  
sfepy.discrete.common.global\_interp (module), 585  
sfepy.discrete.common.mappings (module), 587  
sfepy.discrete.common.region (module), 589  
sfepy.discrete.conditions (module), 537  
sfepy.discrete.equations (module), 539  
sfepy.discrete.evaluate (module), 544  
sfepy.discrete.evaluate\_variable (module), 546  
sfepy.discrete.fem.domain (module), 592  
sfepy.discrete.fem.extmods.bases (module), 593  
sfepy.discrete.fem.extmods.lobatto\_bases (module), 593  
sfepy.discrete.fem.facets (module), 593  
sfepy.discrete.fem.fe\_surface (module), 594  
sfepy.discrete.fem.fields\_base (module), 595  
sfepy.discrete.fem.fields\_hierarchic (module), 600  
sfepy.discrete.fem.fields\_nodal (module), 600  
sfepy.discrete.fem.geometry\_element (module), 601  
sfepy.discrete.fem.history (module), 602  
sfepy.discrete.fem.lcbc\_operators (module), 602  
sfepy.discrete.fem.linearizer (module), 605  
sfepy.discrete.fem.mappings (module), 605  
sfepy.discrete.fem.mesh (module), 606  
sfepy.discrete.fem.meshio (module), 607  
sfepy.discrete.fem.periodic (module), 613  
sfepy.discrete.fem.poly\_spaces (module), 614  
sfepy.discrete.fem.refine (module), 615  
sfepy.discrete.fem.refine\_hanging (module), 616  
sfepy.discrete.fem.utils (module), 617  
sfepy.discrete.functions (module), 546  
sfepy.discrete.iga.domain (module), 618  
sfepy.discrete.iga.domain\_generators (module), 618  
sfepy.discrete.iga.extmods.igac (module), 619  
sfepy.discrete.iga.fields (module), 621  
sfepy.discrete.iga.iga (module), 623  
sfepy.discrete.iga.io (module), 628  
sfepy.discrete.iga.mappings (module), 628  
sfepy.discrete.iga.plot\_nurbs (module), 629  
sfepy.discrete.iga.utils (module), 629  
sfepy.discrete.integrals (module), 547  
sfepy.discrete.materials (module), 548  
sfepy.discrete.parse\_equations (module), 550  
sfepy.discrete.parse\_regions (module), 550  
sfepy.discrete.probes (module), 550  
sfepy.discrete.problem (module), 554  
sfepy.discrete.projections (module), 564  
sfepy.discrete.quadratures (module), 564  
sfepy.discrete.simplex\_cubature (module), 566  
sfepy.discrete.state (module), 566  
sfepy.discrete.structural.fields (module), 630  
sfepy.discrete.structural.mappings (module), 631  
sfepy.discrete.variables (module), 568  
sfepy.homogenization.band\_gaps\_app (module), 631  
sfepy.homogenization.coefficients (module), 632  
sfepy.homogenization.coeffs\_base (module), 632  
sfepy.homogenization.coeffs\_elastic (module), 635  
sfepy.homogenization.coeffs\_perfusion (module), 636  
sfepy.homogenization.coeffs\_phononic (module), 636  
sfepy.homogenization.convolutions (module), 640  
sfepy.homogenization.engine (module), 640  
sfepy.homogenization.homogen\_app (module), 642  
sfepy.homogenization.micmac (module), 643  
sfepy.homogenization.recovery (module), 643  
sfepy.homogenization.utils (module), 646  
sfepy.linalg.check\_derivatives (module), 647  
sfepy.linalg.eigen (module), 647  
sfepy.linalg.geometry (module), 648  
sfepy.linalg.sparse (module), 650  
sfepy.linalg.sympy\_operators (module), 652  
sfepy.linalg.utils (module), 652

sfepy.mechanics.contact\_bodies (module), 655  
 sfepy.mechanics.elastic\_constants (module), 655  
 sfepy.mechanics.extmods.ccontres (module), 665  
 sfepy.mechanics.matcoefs (module), 655  
 sfepy.mechanics.membranes (module), 658  
 sfepy.mechanics.shell10x (module), 660  
 sfepy.mechanics.tensors (module), 661  
 sfepy.mechanics.units (module), 663  
 sfepy.mesh.bspline (module), 666  
 sfepy.mesh.geom\_tools (module), 669  
 sfepy.mesh.mesh\_generators (module), 671  
 sfepy.mesh.mesh\_tools (module), 674  
 sfepy.mesh.splinebox (module), 675  
 sfepy.parallel.plot\_parallel\_dofs (module), 676  
 sfepy.postprocess.domain\_specific (module), 677  
 sfepy.postprocess.plot\_cmsh (module), 678  
 sfepy.postprocess.plot\_dofs (module), 679  
 sfepy.postprocess.plot\_facets (module), 679  
 sfepy.postprocess.plot\_quadrature (module), 680  
 sfepy.postprocess.probes\_vtk (module), 680  
 sfepy.postprocess.sources (module), 681  
 sfepy.postprocess.time\_history (module), 683  
 sfepy.postprocess.utils (module), 683  
 sfepy.postprocess.utils\_vtk (module), 684  
 sfepy.postprocess.viewer (module), 685  
 sfepy.solvers.eigen (module), 688  
 sfepy.solvers.ls (module), 689  
 sfepy.solvers.ls\_mumps (module), 694  
 sfepy.solvers.nls (module), 704  
 sfepy.solvers.optimize (module), 708  
 sfepy.solvers.oseen (module), 709  
 sfepy.solvers.qeigen (module), 710  
 sfepy.solvers.semismooth\_newton (module), 711  
 sfepy.solvers.solvers (module), 711  
 sfepy.solvers.ts (module), 713  
 sfepy.solvers.ts\_solvers (module), 714  
 sfepy.terms.extmods.terms (module), 815  
 sfepy.terms.terms (module), 738  
 sfepy.terms.terms\_adj\_navier\_stokes (module), 742  
 sfepy.terms.terms\_basic (module), 751  
 sfepy.terms.terms\_biot (module), 757  
 sfepy.terms.terms\_constraints (module), 760  
 sfepy.terms.terms\_contact (module), 761  
 sfepy.terms.terms\_diffusion (module), 762  
 sfepy.terms.terms\_dot (module), 768  
 sfepy.terms.terms\_elastic (module), 774  
 sfepy.terms.terms\_electric (module), 783  
 sfepy.terms.terms\_fibres (module), 784  
 sfepy.terms.terms\_hyperelastic\_base (module), 785  
 sfepy.terms.terms\_hyperelastic\_tl (module), 786  
 sfepy.terms.terms\_hyperelastic\_ul (module), 792  
 sfepy.terms.terms\_membrane (module), 795  
 sfepy.terms.terms\_navier\_stokes (module), 796  
 sfepy.terms.terms\_piezo (module), 805  
 sfepy.terms.terms\_point (module), 807  
 sfepy.terms.terms\_shells (module), 808  
 sfepy.terms.terms\_surface (module), 810  
 sfepy.terms.terms\_th (module), 814  
 sfepy.terms.terms\_volume (module), 814  
 sfepy.terms.utils (module), 815  
 sfepy.version (module), 514  
 shape (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
 ShapeDim (class in sfepy.homogenization.coefs\_base), 635  
 ShapeDimDim (class in sfepy.homogenization.coefs\_base), 635  
 shell() (in module sfepy.base.base), 520  
 Shell10XField (class in sfepy.discrete.structural.fields), 630  
 Shell10XMapping (class in sfepy.discrete.structural.mappings), 631  
 Shell10XTerm (class in sfepy.terms.terms\_shells), 808  
 ShiftedPeriodicOperator (class in sfepy.discrete.fem.lcbc\_operators), 604  
 show\_authors (module), 513  
 show\_mesh\_info (module), 513  
 show\_scalar\_bars() (sfepy.postprocess.viewer.Viewer method), 687  
 show\_terms\_use (module), 513  
 simple (module), 505  
 simple\_example() (in module sfepy.mesh.bspline), 669  
 SimpleEVP (class in sfepy.homogenization.coefs\_phononic), 638  
 SimpleTimeSteppingSolver (class in sfepy.solvers.ts\_solvers), 717  
 size\_schur (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 697  
 size\_schur (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 700  
 size\_schur (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 704  
 skip\_read\_line() (in module sfepy.base.ioutils), 530  
 smooth\_f() (sfepy.terms.terms\_surface.ContactPlaneTerm static method), 811  
 smooth\_mesh() (in module sfepy.mesh.mesh\_tools), 674  
 SoftLink (class in sfepy.base.ioutils), 528  
 sol\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_4 attribute), 697  
 sol\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_0 attribute), 700  
 sol\_loc (sfepy.solvers.ls\_mumps.mumps\_struc\_c\_5\_1 attribute), 704  
 solvable() (in module sfepy.base.resolve\_deps), 536  
 solve() (in module sfepy.solvers.ls), 694  
 solve() (sfepy.discrete.problem.Problem method), 562  
 solve\_eigen\_problem() (sfepy.applications.evp\_solver\_app.EVPSolverApp method), 515

solve\_pde() (in module sfepy.applications.pde\_solver\_app), 515  
solve\_pressure\_eigenproblem() (sfepy.homogenization.coefs\_base.PressureEigenvalueProblem), 635  
solve\_step() (sfepy.solvers.ts\_solvers.AdaptiveTimeSteppingSolver method), 714  
solve\_step() (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver method), 717  
solve\_step0() (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver method), 717  
Solver (class in sfepy.solvers.solvers), 712  
SolverMeta (class in sfepy.solvers.solvers), 712  
sort() (sfepy.discrete.conditions.Conditions method), 537  
sort\_by\_dependency() (in module sfepy.discrete.common.region), 592  
sparse\_submat() (sfepy.solvers.ls.MultiProblem method), 690  
spause() (in module sfepy.base.base), 520  
SphinxHTMLDocs (class in build\_helpers), 506  
SphinxPDFDocs (class in build\_helpers), 506  
SplineBox (class in sfepy.mesh.splinebox), 675  
SplineRegion2D (class in sfepy.mesh.splinebox), 676  
split\_chunks() (in module sfepy.homogenization.coefs\_phononic), 639  
split\_complex\_args() (in module sfepy.terms.terms), 741  
split\_conns\_mat\_ids() (in module sfepy.discrete.fem.meshio), 613  
split\_on() (in module edit\_identifiers), 508  
split\_range() (in module sfepy.linalg.utils), 655  
splittlines() (sfepy.mesh.geom\_tools.geometry method), 670  
spy() (in module sfepy.base.plotutils), 535  
spy\_and\_show() (in module sfepy.base.plotutils), 535  
StabilizationFunction (class in sfepy.solvers.oseen), 710  
standalone\_setup() (sfepy.terms.terms.Term method), 741  
standard\_call() (in module sfepy.solvers.eigen), 689  
standard\_call() (in module sfepy.solvers.ls), 694  
standard\_call() (in module sfepy.solvers.qeigen), 710  
standard\_ts\_call() (in module sfepy.solvers.ts\_solvers), 718  
start() (run\_tests.OutputFilter method), 504  
State (class in sfepy.discrete.state), 566  
state\_to\_output() (sfepy.discrete.equations.Equations method), 542  
state\_to\_output() (sfepy.discrete.variables.Variables method), 575  
StationarySolver (class in sfepy.solvers.ts\_solvers), 717  
step (sfepy.postprocess.viewer.SetStep attribute), 685  
stiffness\_from\_lame() (in module sfepy.mechanics.matcoefs), 657  
stiffness\_from\_lame\_mixed() (in module sfepy.mechanics.matcoefs), 657  
stiffness\_from\_youngpoisson() (in module sfepy.mechanics.matcoefs), 657  
stiffness\_from\_youngpoisson\_mixed() (in module sfepy.mechanics.matcoefs), 657  
StokesTerm (class in sfepy.terms.terms\_navier\_stokes), 802  
StokesWaveDivTerm (class in sfepy.terms.terms\_navier\_stokes), 803  
StokesWaveTerm (class in sfepy.terms.terms\_navier\_stokes), 804  
stop() (run\_tests.OutputFilter method), 505  
str\_all() (sfepy.base.base.Struct method), 517  
str\_class() (sfepy.base.base.Struct method), 518  
stress\_function() (sfepy.terms.terms\_fibres.FibresActiveTLTerm static method), 784  
stress\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm static method), 786  
stress\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm static method), 787  
stress\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm static method), 787  
stress\_function() (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm static method), 789  
stress\_function() (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm static method), 789  
stress\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm static method), 792  
stress\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm static method), 793  
stress\_function() (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm static method), 794  
stress\_function() (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm static method), 794  
StressTransform (class in sfepy.mechanics.tensors), 661  
string (sfepy.discrete.fem.meshio.HDF5MeshIO attribute), 609  
strip\_state\_vector() (sfepy.discrete.equations.Equations method), 542  
strip\_state\_vector() (sfepy.discrete.variables.Variables method), 575  
Struct (class in sfepy.base.base), 517  
structify() (in module sfepy.base.base), 521  
substitute\_continuous() (in module eval\_ns\_forms), 508  
substitute\_dofs() (sfepy.discrete.fem.fields\_base.FEField method), 598  
SurfaceNormalDotTerm (class in sfepy.terms.terms\_surface), 813  
suggest\_name() (sfepy.discrete.fem.poly\_spaces.PolySpace static method), 615  
SumNodalValuesTerm (class in sfepy.terms.terms\_basic), 755  
SUPGCAjStabilizationTerm (class in sfepy.terms.terms\_adj\_navier\_stokes), 749

|                                                                    |                                                 |     |    |                                                                                               |                                                |
|--------------------------------------------------------------------|-------------------------------------------------|-----|----|-----------------------------------------------------------------------------------------------|------------------------------------------------|
| SUPGCSStabilizationTerm                                            | (class<br>sfepy.terms.terms_navier_stokes),     | 801 | in | system() (sfepy.config.Config method),                                                        | 514                                            |
| SUPGPAdj1StabilizationTerm                                         | (class<br>sfepy.terms.terms_adj_navier_stokes), | 750 | in | T                                                                                             |                                                |
| SUPGPAdj2StabilizationTerm                                         | (class<br>sfepy.terms.terms_adj_navier_stokes), | 750 | in | tan_mod_function() (sfepy.terms.terms_fibres.FibresActiveTLTerm<br>static method),            | 784                                            |
| SUPGPStabilizationTerm                                             | (class<br>sfepy.terms.terms_navier_stokes),     | 802 | in | tan_mod_function() (sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm<br>static method),     | 786                                            |
| surface (class in sfepy.mesh.geom_tools),                          | 671                                             |     |    | tan_mod_function() (sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm<br>static method),    | 787                                            |
| surface_components() (in module extract_surface),                  | 509                                             |     |    | tan_mod_function() (sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm<br>static method),   | 789                                            |
| surface_graph() (in module extract_surface),                       | 509                                             |     |    | tan_mod_function() (sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm<br>static method),     | 790                                            |
| SurfaceDivTerm                                                     | (class<br>sfepy.terms.terms_navier_stokes),     | 804 | in | tan_mod_function() (sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm<br>static method),    | 792                                            |
| SurfaceField (class in sfepy.discrete.fem.fields_base),            | 598                                             |     |    | tan_mod_function() (sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm<br>static method),   | 794                                            |
| SurfaceFluxOperatorTerm                                            | (class<br>sfepy.terms.terms_diffusion),         | 767 | in | tan_mod_function() (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm<br>static method),     | 794                                            |
| SurfaceFluxTerm (class in sfepy.terms.terms_diffusion),            |                                                 |     |    | tan_mod_u_function() (sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm<br>static method), | 787                                            |
| SurfaceFluxTLTerm                                                  | (class<br>sfepy.terms.terms_hyperelastic_tl),   | 790 | in | tan_mod_u_function() (sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm<br>static method), | 793                                            |
| SurfaceGradTerm                                                    | (class<br>sfepy.terms.terms_navier_stokes),     | 805 | in | TCorrectorsPressureViaPressureEVP                                                             | (class<br>sfepy.homogenization.coefs_elastic), |
| SurfaceJumpTerm (class in sfepy.terms.terms_surface),              |                                                 |     |    | 635                                                                                           | in                                             |
| 814                                                                |                                                 |     |    | TCorrectorsRSViaPressureEVP                                                                   | (class<br>sfepy.homogenization.coefs_elastic), |
| SurfaceMapping (class in sfepy.discrete.fem.mappings),             |                                                 |     |    | 635                                                                                           | in                                             |
| 605                                                                |                                                 |     |    | TCorrectorsViaPressureEVP                                                                     | (class<br>sfepy.homogenization.coefs_base),    |
| SurfaceMomentTerm (class in sfepy.terms.terms_basic),              |                                                 |     |    | 635                                                                                           | in                                             |
| 755                                                                |                                                 |     |    | tdim (sfepy.discrete.common.extmods.cmesh.CMesh<br>attribute),                                | 581                                            |
| SurfaceTerm (class in sfepy.terms.terms_basic),                    |                                                 |     |    | tensor_plane_stress() (sfepy.mechanics.matcoefs.TransformToPlane<br>method),                  | 656                                            |
| 755                                                                |                                                 |     |    | tensor_product() (in module sfepy.discrete.iga.iga),                                          | 628                                            |
| SurfaceTractionTLTerm                                              | (class<br>sfepy.terms.terms_hyperelastic_tl),   | 790 | in | Term (class in sfepy.terms.terms),                                                            | 738                                            |
| sym (sfepy.solvers.ls_mumps.mumps_struct_c_4<br>attribute),        | 697                                             | at- |    | term_ns_asm_convect() (in<br>sfepy.terms.extmods.terms),                                      | 818                                            |
| sym (sfepy.solvers.ls_mumps.mumps_struct_c_5_0<br>attribute),      | 700                                             | at- |    | term_ns_asm_div_grad() (in<br>sfepy.terms.extmods.terms),                                     | 818                                            |
| sym (sfepy.solvers.ls_mumps.mumps_struct_c_5_1<br>attribute),      | 704                                             | at- |    | terminate() (sfepy.base.log.Log method),                                                      | 531                                            |
| sym (sfepy.solvers.ls_mumps.mumps_struct_c_x<br>attribute),        | 704                                             | at- |    | terminate() (sfepy.base.log_plotter.LogPlotter<br>method),                                    | 532                                            |
| sym2dim() (in module sfepy.mechanics.tensors),                     | 663                                             |     |    | TermParse (class in sfepy.discrete.parse_equations),                                          | 550                                            |
| sym2nonsym() (in module sfepy.terms.extmods.terms),                |                                                 |     |    | Terms (class in sfepy.terms.terms),                                                           | 741                                            |
| 818                                                                |                                                 |     |    | test_install (module),                                                                        | 507                                            |
| sym_perm (sfepy.solvers.ls_mumps.mumps_struct_c_4<br>attribute),   | 697                                             |     |    | TestCommon (class in sfepy.base.testing),                                                     | 536                                            |
| sym_perm (sfepy.solvers.ls_mumps.mumps_struct_c_5_0<br>attribute), | 700                                             |     |    | tetgen_path() (sfepy.config.Config method),                                                   | 514                                            |
| sym_perm (sfepy.solvers.ls_mumps.mumps_struct_c_5_1<br>attribute), | 704                                             |     |    | TetgenMeshIO (class in sfepy.discrete.fem.meshio),                                            | 612                                            |
| sym_tri_eigen() (in module sfepy.linalg.eigen),                    | 648                                             |     |    | tetrahedralize_vtk_mesh() (in<br>sfepy.postprocess.utils_vtk),                                | 684                                            |
| symbolic (sfepy.terms.terms_diffusion.DiffusionTerm<br>attribute), | 765                                             |     |    | THTerm (class in sfepy.terms.terms_th),                                                       | 814                                            |
| symbolic (sfepy.terms.terms_diffusion.LaplaceTerm<br>attribute),   | 766                                             |     |    | tile_periodic_mesh (module),                                                                  | 513                                            |
| sync_module_docs (module),                                         | 513                                             |     |    | tiled_mesh1d() (in module sfepy.mesh.mesh_generators),                                        |                                                |
|                                                                    |                                                 |     |    | 674                                                                                           |                                                |

time (sfepy.postprocess.viewer.SetStep attribute), 685  
time\_update() (sfepy.discrete.equations.Equations method), 543  
time\_update() (sfepy.discrete.materials.Material method), 548  
time\_update() (sfepy.discrete.materials.Materials method), 549  
time\_update() (sfepy.discrete.problem.Problem method), 563  
time\_update() (sfepy.discrete.variables.FieldVariable method), 572  
time\_update() (sfepy.discrete.variables.Variable method), 573  
time\_update() (sfepy.discrete.variables.Variables method), 575  
time\_update() (sfepy.terms.terms.Term method), 741  
time\_update\_materials() (sfepy.discrete.equations.Equationstransform\_integrals() (in module sfepy.base.conf), 525  
method), 543  
TimeStepper (class in sfepy.solvers.ts), 713  
TimeSteppingSolver (class in sfepy.solvers.solvers), 712  
TLMembraneTerm (class in sfepy.terms.terms\_membrane), 795  
to\_array() (sfepy.linalg.utils.MatrixAction method), 652  
to\_dict() (sfepy.base.base.Struct method), 518  
to\_file\_hdf5() (sfepy.homogenization.coefficients.Coefficients method), 632  
to\_file\_latex() (sfepy.homogenization.coefficients.Coefficients method), 632  
to\_file\_txt() (sfepy.homogenization.coefs\_phononic.AcousticMassTensor variables() (in module sfepy.base.conf), 526  
attribute), 636  
to\_file\_txt() (sfepy.homogenization.coefs\_phononic.AppliedLoadTensor variables() (in module sfepy.base.conf), 526  
attribute), 636  
to\_file\_txt() (sfepy.homogenization.coefficients.Coefficients method), 632  
to\_file\_txt() (sfepy.homogenization.coefficients.BandGaps static method), 637  
to\_file\_txt() (sfepy.homogenization.coeffs\_phononic.Density static method), 637  
to\_latex() (sfepy.homogenization.coefficients.Coefficients method), 632  
to\_list() (in module gen\_term\_table), 512  
to\_ndarray() (in module sfepy.mesh.bspline), 669  
to\_poly\_file() (sfepy.mesh.geom\_tools.geometry method), 670  
to\_stack() (in module sfepy.discrete.parse\_regions), 550  
transform\_asm\_matrices() (in module sfepy.mechanics.membranes), 659  
transform\_asm\_matrices() (in module sfepy.mechanics.shell10x), 661  
transform\_asm\_vectors() (in module sfepy.mechanics.membranes), 659  
transform\_bar\_to\_space\_coors() (in module sfepy.linalg.geometry), 650  
transform\_basis() (in module sfepy.discrete.fem.poly\_spaces), 615  
transform\_conditions() (in module sfepy.base.conf), 525  
transform\_coors() (in module sfepy.discrete.fem.mesh.Mesh method), 606  
transform\_data() (in module sfepy.mechanics.tensors), 663  
transform\_ebc() (in module sfepy.base.conf), 525  
transform\_epbc() (in module sfepy.base.conf), 525  
transform\_fields() (in module sfepy.base.conf), 525  
transform\_functions() (in module sfepy.base.conf), 525  
transform\_ics() (in module sfepy.base.conf), 525  
transform\_input() (sfepy.base.conf ProblemConf method), 525  
transform\_input\_trivial() (sfepy.base.conf ProblemConf method), 525  
transform\_lcbc() (in module sfepy.base.conf), 525  
transform\_materials() (in module sfepy.base.conf), 526  
transform\_plot\_data() (in module sfepy.homogenization.band\_gaps\_app), 632  
transform\_regions() (in module sfepy.base.conf), 526  
transform\_solvers() (in module sfepy.base.conf), 526  
transform\_to\_i\_struct\_1() (in module sfepy.base.conf), 526  
transform\_to\_struct\_01() (in module sfepy.base.conf), 526  
transform\_to\_struct\_1() (in module sfepy.base.conf), 526  
MassTensor variables() (in module sfepy.base.conf), 526  
TransformToPlane (class in sfepy.mechanics.matcoefs), 656  
treat\_pbcs() (sfepy.discrete.fem.lcbc\_operators.MRLCBCOperator method), 603  
triangulate() (in module sfepy.mesh.mesh\_tools), 675  
VohrInfile module (in module gen\_solver\_table), 511  
try\_block() (in module sfepy.base.resolve\_deps), 536  
try\_imports() (in module sfepy.base.base), 521  
try\_presolve() (sfepy.discrete.problem.Problem method), 563  
try\_set\_defaults() (in module sfepy.homogenization.band\_gaps\_app), 632  
TSTimes (class in sfepy.homogenization.coefs\_base), 635  
tuple\_to\_conf() (in module sfepy.base.conf), 526  
typeset() (in module gen\_solver\_table), 511  
typeset() (in module gen\_term\_table), 512  
typeset\_solvers\_table() (in module gen\_solver\_table), 511  
typeset\_term\_syntax() (in module gen\_term\_table), 512  
typeset\_term\_table() (in module gen\_term\_table), 512  
typeset\_term\_tables() (in module gen\_term\_table), 512  
typeset\_to\_indent() (in module gen\_term\_table), 512  
typeset\_to\_indent() (in module sfepy.solvers.solvers), 713

**U**

Uncached (class in sfepy.base.ioutils), 528  
 unique() (in module sfepy.base.compat), 522  
 unique\_rows() (in module sfepy.linalg.utils), 655  
 Unit (class in sfepy.mechanics.units), 664  
 unpack\_data() (sfepy.base.ioutils.DataMarker method), 527  
 unpack\_data() (sfepy.base.ioutils.DataSoftLink method), 527  
 unpack\_data() (sfepy.base.ioutils.HDF5BaseData method), 527  
 uns\_perm (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 uns\_perm (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 701  
 uns\_perm (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 704  
 update() (sfepy.base.base.Container method), 516  
 update() (sfepy.base.base.Struct method), 518  
 update() (sfepy.discrete.common.dof\_info.DofInfo method), 577  
 update() (sfepy.terms.terms\_contact.ContactInfo method), 761  
 update\_conf() (sfepy.base.conf.ProblemConf method), 525  
 update\_data() (sfepy.discrete.materials.Material method), 549  
 update\_dict\_recursively() (in module sfepy.base.base), 521  
 update\_equations() (sfepy.discrete.problem.Problem method), 563  
 update\_expression() (sfepy.terms.terms.Terms method), 741  
 update\_materials() (sfepy.discrete.problem.Problem method), 563  
 update\_micro\_coors() (sfepy.homogenization.homogen\_app method), 643  
 update\_shape() (sfepy.discrete.common.region.Region method), 591  
 update\_special\_constant\_data() (sfepy.discrete.materials.Material method), 549  
 update\_special\_data() (sfepy.discrete.materials.Material method), 549  
 update\_time stepper() (sfepy.discrete.problem.Problem method), 563  
 us2cw() (in module edit\_identifiers), 508  
 us2mc() (in module edit\_identifiers), 508  
 use\_first\_available() (in module sfepy.solvers.solvers), 713  
 use\_method\_with\_name() (in module sfepy.base.base), 521  
 user\_options (build\_helpers.NoOptionsDocs attribute), 506

UserMeshIO (class in sfepy.discrete.fem.meshio), 612

**V**

validate (sfepy.base.goptions.ValidatedDict attribute), 526  
 validate() (sfepy.base.conf.ProblemConf method), 525  
 validate\_bool() (in module sfepy.base.goptions), 526  
 ValidatedDict (class in sfepy.base.goptions), 526  
 validator() (sfepy.base.goptions.ValidatedDict method), 526  
 values() (sfepy.base.goptions.ValidatedDict method), 526  
 Variable (class in sfepy.discrete.variables), 572  
 Variables (class in sfepy.discrete.variables), 573  
 VariableTimeStepper (class in sfepy.solvers.ts), 713  
 VectorDotGradScalarTerm (class in sfepy.terms.terms\_dot), 772  
 VectorDotScalarTerm (class in sfepy.terms.terms\_dot), 773  
 VelocityVerletTS (class in sfepy.solvers.ts\_solvers), 717  
 version\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_4 attribute), 697  
 version\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_0 attribute), 701  
 version\_number (sfepy.solvers.ls\_mumps.mumps\_struct\_c\_5\_1 attribute), 704  
 vertex\_groups (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 581  
 vertices (sfepy.discrete.common.region.Region attribute), 591  
 view\_file() (in module postproc), 503  
 Viewer (class in sfepy.postprocess.viewer), 685  
 ViewerGUI (class in sfepy.postprocess.viewer), 687  
 visit\_stack() (in module sfepy.discrete.parse\_regions), 550  
 volume (class in sfepy.mesh.geom\_tools), 671  
 Volume (sfepy.discrete.common.extmods.mappings.CMapping attribute), 582  
 VolumeField (class in sfepy.discrete.fem.fields\_base), 598  
 VolumeFractions (class in sfepy.homogenization.coefs\_base), 635  
 VolumeMapping (class in sfepy.discrete.fem.mappings), 605  
 VolumeSurfaceTerm (class in sfepy.terms.terms\_basic), 756  
 VolumeSurfaceTLTerm (class in sfepy.terms.terms\_hypereastic\_tl), 791  
 VolumeTerm (class in sfepy.terms.terms\_basic), 756  
 VolumeTLTerm (class in sfepy.terms.terms\_hypereastic\_tl), 791  
 VolumeULTerm (class in sfepy.terms.terms\_hypereastic\_ul), 795  
 VTKFileSource (class in sfepy.postprocess.sources), 682  
 VTKMeshIO (class in sfepy.discrete.fem.meshio), 613

VTKSequenceFileSource (class in `sfePy.postprocess.sources`), 682

**W**

wandering\_element() (in module `sfePy.discrete.simplex_cubature`), 566

weak\_dp\_function() (`sfePy.terms.terms_hyperelastic_tl.BulkPressureTLTerm` static method), 787

weak\_dp\_function() (`sfePy.terms.terms_hyperelastic_ul.BulkPressureULTerm` static method), 793

weak\_function() (`sfePy.terms.terms_hyperelastic_tl.BulkPressureTLTerm` static method), 787

weak\_function() (`sfePy.terms.terms_hyperelastic_tl.HyperELasticTerm` static method), 789

weak\_function() (`sfePy.terms.terms_hyperelastic_ul.HyperELasticTerm` static method), 793

weak\_function() (`sfePy.terms.terms_membrane.TLMembraneTerm` static method), 796

wk\_user (`sfePy.solvers.ls_mumps.mumps_struct_c_4` attribute), 697

wk\_user (`sfePy.solvers.ls_mumps.mumps_struct_c_5_0` attribute), 701

wk\_user (`sfePy.solvers.ls_mumps.mumps_struct_c_5_1` attribute), 704

wrap\_function() (in module `sfePy.solvers.optimize`), 709

wrap\_run\_tests() (in module `run_tests`), 505

write() (`run_tests.OutputFilter` method), 505

write() (`sfePy.base.ioutils.HDF5Data` method), 527

write() (`sfePy.base.ioutils.SoftLink` method), 528

write() (`sfePy.discrete.fem.mesh.Mesh` method), 607

write() (`sfePy.discrete.fem.meshio.AbaqusMeshIO` method), 608

write() (`sfePy.discrete.fem.meshio.ANSYSCDBMMeshIO` method), 608

write() (`sfePy.discrete.fem.meshio.AVSUCDMeshIO` method), 608

write() (`sfePy.discrete.fem.meshio.BDFMeshIO` method), 608

write() (`sfePy.discrete.fem.meshio.ComsolMeshIO` method), 608

write() (`sfePy.discrete.fem.meshio.HDF5MeshIO` method), 609

write() (`sfePy.discrete.fem.meshio.HypermeshAsciiMeshIO` method), 609

write() (`sfePy.discrete.fem.meshio.MeditMeshIO` method), 610

write() (`sfePy.discrete.fem.meshio.MeshIO` method), 611

write() (`sfePy.discrete.fem.meshio.NEUMeshIO` method), 612

write() (`sfePy.discrete.fem.meshio.TetgenMeshIO` method), 612

in `write()` (`sfePy.discrete.fem.meshio.UserMeshIO` method), 613

`write()` (`sfePy.discrete.fem.meshio.VTKMeshIO` method), 613

`write_control_net()` (`sfePy.mesh.splinebox.SplineBox` method), 676

`write_control_polygon_vtk()` (`sfePy.mesh.bspline.BSplineSurf` method), 669

`write_data()` (`sfePy.base.ioutils.DataSoftLink` method), 597

`write_data()` (`sfePy.base.ioutils.HDF5Data` method), 527

`write_domain_to_hdf5()` (`sfePy.discrete.iga.domain.IGDomain` method), 618

`write_iga_data()` (in module `sfePy.discrete.iga.io`), 628

`write_log()` (in module `sfePy.base.log`), 532

`write_mesh_to_hdf5()` (`sfePy.discrete.fem.meshio.HDF5MeshIO` method), 609

`write_problem()` (`sfePy.solvers.ls_mumps.mumps_struct_c_4` attribute), 698

`write_problem()` (`sfePy.solvers.ls_mumps.mumps_struct_c_5_0` attribute), 701

`write_problem()` (`sfePy.solvers.ls_mumps.mumps_struct_c_5_1` attribute), 704

`write_results()` (in module `sfePy.discrete.probes`), 553

`write_sparse_matrix_hdf5()` (in module `sfePy.base.ioutils`), 530

`write_sparse_matrix_to_hdf5()` (in module `sfePy.base.ioutils`), 530

`write_surface_vtk()` (`sfePy.mesh.bspline.BSplineSurf` method), 669

`write_to_hdf5()` (in module `sfePy.base.ioutils`), 530

`write_vtk_to_file()` (in module `sfePy.postprocess.utils_vtk`), 685

**X**

`xfail()` (`sfePy.base.testing.TestCommon` static method), 536

**Y**

`youngpoisson_from_stiffness()` (in module `sfePy.mechanics.matcoefs`), 657

**Z**

`zero_dofs()` (`sfePy.discrete.conditions.Conditions` method), 537

`zero_dofs()` (`sfePy.discrete.conditions.EssentialBC` method), 537

`ZeroTerm` (class in `sfePy.terms.terms_basic`), 757