

---

# **SfePy Documentation**

***Release 2013.2***

**Robert Cimrman and Contributors**

May 22, 2013



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Requirements . . . . .	3
2.2	Generic Installation Instructions . . . . .	4
2.3	Checking the SfePy installation . . . . .	4
2.4	Platform-specific notes . . . . .	5
<b>3</b>	<b>Tutorial</b>	<b>9</b>
3.1	Notes on solving PDEs by the Finite Element Method . . . . .	9
3.2	Basic notions . . . . .	12
3.3	Running a simulation . . . . .	13
3.4	Example problem description file . . . . .	14
3.5	Interactive Example: Linear Elasticity . . . . .	19
<b>4</b>	<b>Primer</b>	<b>27</b>
4.1	Introduction . . . . .	27
4.2	Meshing . . . . .	28
4.3	Problem description . . . . .	30
4.4	Running SfePy . . . . .	33
4.5	Post-processing . . . . .	34
4.6	Probing . . . . .	41
<b>5</b>	<b>User's Guide</b>	<b>45</b>
5.1	Running a Simulation . . . . .	46
5.2	Visualization of Results . . . . .	48
5.3	Problem Description File . . . . .	51
5.4	Building Equations in SfePy . . . . .	65
5.5	Term Evaluation . . . . .	66
5.6	Solution Postprocessing . . . . .	66
5.7	Probing . . . . .	68
5.8	Available Solvers . . . . .	68
<b>6</b>	<b>Examples</b>	<b>71</b>
6.1	SfePy autogenerated gallery examples . . . . .	71
<b>7</b>	<b>Developer Guide</b>	<b>269</b>
7.1	SfePy Directory Structure . . . . .	271
7.2	How to Contribute . . . . .	272
7.3	How to Regenerate Documentation . . . . .	291

7.4	How to Implement a New Term . . . . .	292
7.5	How To Make a Release . . . . .	297
7.6	Working with <i>SfePy</i> source code . . . . .	298
7.7	Module Index . . . . .	314
<b>8</b>	<b>Notes</b>	<b>587</b>
8.1	Linear Combination Boundary Conditions . . . . .	587
<b>9</b>	<b>Indices and tables</b>	<b>589</b>
	<b>Python Module Index</b>	<b>591</b>
	<b>Python Module Index</b>	<b>593</b>
	<b>Index</b>	<b>595</b>

# INTRODUCTION

*SfePy* is a finite element analysis software 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/SciPy](#) (an excellent collection of tools for scientific computations in Python).

*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 with many weak forms to build equations so there is a chance it might serve you as well.

New users should start by going through the [Tutorial](#) and then the more focused [Primer](#).

Features:

- solution of linear, nonlinear problems
- multi-platform (Linux, Mac OS X, Windows)
- **collection of modules (a library):**
  - FE engine, problem description facilities, interfaces to various solvers, postprocessing utilities
  - usable to build custom applications
- **“black box” PDE solver:**
  - no real programming involved
  - just prepare a problem description file (in Python!) and solve it
  - highly customizable behaviour (with a bit of coding)

To find more information regarding the code itself, go to [SfePy web site](#) where you can find:

- releases
- mailing lists
- issue tracking
- git repository (bleeding edge code)
- further documentation, examples, and some of the research projects the code has been developed for.

To discuss in real time, join our IRC channel [#sfepy](#) at freenode.



# INSTALLATION

## Table of Contents

- Requirements
- Generic Installation Instructions
  - In-place compilation of C extension modules
  - Installation
- Checking the SfePy installation
- Platform-specific notes
  - Gentoo
  - Archlinux
    - \* Instructions
  - Debian
  - Ubuntu
    - \* Prerequisites
      - Older Versions of Ubuntu
    - \* Installing SfePy
  - Fedora 8
  - Intel Mac
  - Windows using Python(x,y)
    - \* Steps to get a working *SfePy* on Windows using Python(x,y)

## 2.1 Requirements

Installation prerequisites:

- recent numpy, scipy (with umfpack wrapper, or umfpack scikit), cython

Dependencies:

- matplotlib, pyparsing, umfpack, pytables
- some tests and functions use sympy
- schroedinger.py requires pyparse, pexpect, gmsh (2D), tetgen (3D)
- log.py (live plotting) requires multiprocessing, matplotlib with GTKAgg
- isfepy requires ipython, matplotlib with WXAgg
- postproc.py requires mayavi2

- to be able to (re)generate the documentation: sphinx, numpydoc, LaTeX, see [How to Regenerate Documentation](#)

*SfePy* is known to work on various flavours of Linux, on Intel Macs and Windows.

On Linux, consult the package manager of your favourite distribution, see [Platform-specific notes](#).

On Windows, all the required packages are part of the [Enthought Python Distribution](#) (EPD), which is free for academic purposes. A completely free [Python\(x,y\)](#) can be used too, but `pyparsing` has to be installed manually. Instructions for installing [Python\(x,y\)](#) can be found in [Windows using Python\(x,y\)](#).

*SfePy* can be used without any installation by running the scripts from the top-level directory of the distribution (TOPDIR), or can be installed locally or system-wide.

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

## 2.2 Generic Installation Instructions

Download the latest source release or the development version from [SfePy git repository](#):

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

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

### 2.2.1 In-place compilation of C extension modules

1. Look at `site_cfg_template.py` and follow the instructions therein. Usually no changes are necessary.
2. Run:

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

### 2.2.2 Installation

(As mentioned above, this step is not required to use *SfePy*.)

- System-Wide (may require root privileges):

```
python setup.py install
```

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

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

See also `INSTALL` and `RELEASE_NOTES` files in the tarball.

If all went well, proceed with [Checking the SfePy installation](#).

## 2.3 Checking the SfePy installation

After installing *SfePy* you can check if all the functionalities are working by running the automated tests. From the source directory type:

```
./runTests.py
```



If a particular test fails, please run it in debug mode:

```
./runTests.py --debug tests/failing_test_name.py
```

and report the output to the sfepy-devel mailing list.

## 2.4 Platform-specific notes

### 2.4.1 Gentoo

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

### 2.4.2 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.

### 2.4.3 Debian

(old instructions, check also Ubuntu below)

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

### 2.4.4 Ubuntu

(tested on Ubuntu 11.10)

### Prerequisites

First, you have to install the dependencies packages:

```
sudo aptitude install python-scipy python-matplotlib python-tables python-pyparsing libsuitesparse-dev
```

The same packages work also in Kubuntu 11.10. If *aptitude* is not installed, install it, or try *apt-get* instead.

### Older Versions of Ubuntu

(tested on Jaunty Jackalope 9.04 and Lucid Lynx 10.04)

The following is required to get working umfpack. Download and install the umfpack scikits in some local dir. In the following example it will be installed in \$HOME/local:

```
svn checkout http://svn.scipy.org/svn/scikits/trunk/umfpack
cd umfpack
mkdir -p ${HOME}/local/lib/python2.6/site-packages
python setup.py install --prefix=${HOME}/local
```

Add to your `.bashrc` the line:

```
export PYTHONPATH="${HOME}/local"
```

then re-open a terminal and if the scikits was installed correctly importing `scikits.umfpack` in python should give no error:

```
python
>>> import scikits.umfpack
>>>
```

Next Download sympy 6.7 or later. Extract the contents.

```
cd sympy-0.6.7
```

```
python setup.py install --prefix=${HOME}/local
```

### Installing SfePy

Now download the latest *SfePy* tarball release (or the latest development version).

Uncompress the archive and enter the *SfePy* dir, then type:

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

after a few minutes the compilation finishes.

Finally you can test *SfePy* with:

```
./runTests.py
```

If some test fails see [Checking the SfePy installation](#) section for further details.

### 2.4.5 Fedora 8

Notes on using umfpack (contributed by David Huard).

entry in `numpy site.cfg`:

```
[umfpack]
library_dirs=/usr/lib64
include_dirs = /usr/include/suitesparse
```

Comment by david.huard, Mar 26, 2008:

Of course, `suitesparse` and `suitesparse-devel` must be installed.

### 2.4.6 Intel Mac

(thanks to Dominique Orban for his advice)

To build *SfePy* on an Intel Mac the following options need to be set in `site_cfg.py`:

```
opt_flags = '-g -O2 -fPIC -DPIC -fno-strict-aliasing -fno-common -dynamic'
link_flags = '-dynamiclib -undefined dynamic_lookup -fPIC -DPIC'
```

## 2.4.7 Windows using Python(x,y)

(tested on Windows 7)

Here we provide instructions for using *SfePy* on Windows through [Python\(x,y\)](#). We will also use [msysgit](#) to install the *umfpack* *scikit* to speed performance.

This procedure was tested on a Windows 7 machine. It should work in theory for any Windows machine supported by [Python\(x,y\)](#) and [msysgit](#), but your mileage may vary.

There several steps, but hopefully it is straightforward to follow this procedure. If you have any questions or difficulties please feel free to ask on the *sfepy-devel* mailing list (see [SfePy web site](#)). Also, if you have any suggestions for improving or streamlining this process, it would be very beneficial as well!

We assume the installation to be done in C:/ - substitute your path where appropriate.

### Steps to get a working *SfePy* on Windows using [Python\(x,y\)](#)

1. Minimum 4 Gigabytes of free disk space is required, Due to the installed size of [python\(x,y\)](#) and [msysgit](#).
2. Download the latest [Python\(x,y\)](#) windows installer (version 2.7.X.X), and make a *Full installation* in the default installation directory.
3. Download the latest [pyparsing](#) windows installer (Python version 2.7) and install it in the default installation directory.
4. Download the latest [msysgit](#) windows installer and install it in the default installation directory:
  - either get the file that begins with “Git-”, which gives you *gitbash* - a bash shell in Windows,
  - or get the file that begins with “msysGit-fullinstall”.

Below we refer to either *gitbash* or *msys* as “shell”.

5. Download the latest [umfpackpy](#) zip archive and follow the instructions below:
  - (a) Extract the *umfpackpy\_<version>.zip* to your convenient location in Hard disk, Lets assume it's extracted in C:/ . Now there will be two files on the extracted folder, *ez\_setup.py* and *scikits.umfpack-5.1.0-py2.7-win32.egg*.
  - (b) Start a shell (*gitbash* or *msys*, depending on the previous step) and write the following to go to the extracted folder:
 

```
cd /c/umfpackpy_<version>/
```
  - (c) Install the UMFPACK library for python:
 

```
ez_setup.py scikits.umfpack-5.1.0-py2.7-win32.egg
```
6. Either [download](#) the latest *sfepy* tarball and extract it to your convenient location in Hard disk, Lets assume it's extracted in C:/.

Or, If you want to use the latest features and contribute to the development of *SfePy*, clone the git development repository

- In shell, type:

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

Then follow the instructions below:

- (a) In shell, go to the extracted folder:

```
cd /c/sfepy_folder_name/
```

- (b) Compile SfePy C extensions:

```
python setup.py build_ext --inplace --compiler=mingw32
```

7. You should now have a working copy of SfePy on Windows, Please help aid SfePy development by running the built-in tests. Run the *runTests.py* in python IDLE or Write the following code in the shell:

```
./runTests.py --filter-less
```

- Report any failures to the sfepy-devel mailing list
- See [Checking the SfePy installation](#) for further details.

# TUTORIAL

## Table of Contents

- Notes on solving PDEs by the Finite Element Method
  - Strong form of Poisson’s equation and its integration
    - \* Dirichlet Boundary Conditions
    - \* Neumann Boundary Conditions
  - The weak form of the Poisson’s equation
  - Discussion of discretization and meshing
  - Numerical solution of the problem
- Basic notions
  - Sneak peek: what is going on under the hood
- Running a simulation
  - Invoking *SfePy* from the command line
  - Postprocessing the results
- Example problem description file
  - Long syntax of keywords
  - Short syntax of keywords
- Interactive Example: Linear Elasticity
  - Complete Example as a Script

*SfePy* can be used in two basic ways:

1. a black-box partial differential equation (PDE) solver,
2. a Python package to build custom applications involving solving PDEs by the finite element (FE) method.

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.

## 3.1 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.

### 3.1.1 Strong form of Poisson's equation and its integration

Let us start our discussion about FEM with the strong form of Poisson's equation

$$\Delta T = f(x), \quad x \in \Omega, \quad (3.1)$$

$$T = u(x), \quad x \in \Gamma_D, \quad (3.2)$$

$$\nabla T \cdot \mathbf{n} = g(x), \quad x \in \Gamma_N, \quad (3.3)$$

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 (3.1) is

$$\int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = 0,$$

where  $s$  is a **test** function. Integrating this equation by parts

$$\begin{aligned} 0 &= \int_{\Omega} (\Delta T - f) \cdot s \, d\Omega = \int_{\Omega} \nabla \cdot (\nabla T) \cdot s \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega = \\ &= - \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega + \int_{\Omega} \nabla \cdot (\nabla T \cdot s) \, d\Omega - \int_{\Omega} f \cdot s \, d\Omega \end{aligned}$$

and applying Gauss theorem we obtain:

$$0 = - \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega + \int_{\Gamma_D \cup \Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega$$

or

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_D \cup \Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega.$$

The surface integral term can be split into two integrals, one over the Dirichlet part of the surface and second over the Neumann part

$$\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_D} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma + \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega. \quad (3.4)$$

The equation (3.4) is the initial weak form of the Poisson's problem (3.1)–(3.3). 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 (3.4). 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

$$\begin{aligned} \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega &= \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \\ T(x) &= u(x), \quad x \in \Gamma_D. \end{aligned}$$

That is why Dirichlet conditions in FEM terminology are called **Essential Boundary Conditions**. These conditions are not a part of the weak form and they are used as they are.

## Neumann Boundary Conditions

The Neumann boundary conditions correspond to the known flux  $g(x) = \nabla T \cdot \mathbf{n}$ . The integral term over the Neumann surface in the equation (3.4) 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.

### 3.1.2 The weak form of the Poisson's equation

Now we can write the resulting weak form for the Poisson's problem (3.1)–(3.3). For any test function  $s \in V_0(\Omega)$  find  $T \in V(\Omega)$  such that

$$\boxed{\begin{aligned} \int_{\Omega} \nabla T \cdot \nabla s \, d\Omega &= \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \quad \text{and} \\ T(x) &= u(x), \quad x \in \Gamma_D. \end{aligned}} \quad (3.5)$$

### 3.1.3 Discussion of discretization and meshing

It is planned to have an example of the discretization based on the Poisson's equation weak form (3.5). For now, please refer to the wikipedia page [Finite Element Method](#) for a basic description of the discretization and meshing.

### 3.1.4 Numerical solution of the problem

To solve numerically given problem based on the weak form (3.5) 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  $\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega, \int_{\Gamma_N} g \cdot s \, d\Gamma, \int_{\Omega} f \cdot s \, d\Omega$ .

In the next section we will discuss how to define all these things in SfePy.

## 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, solution domain and subdomains etc., 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. Historically, the keywords exist in two flavors:

- **long syntax** is the original one - it is longer to type, but the individual fields are named, so it might be easier/understand to read for newcomers.
- **short syntax** was added later to offer brevity for “expert” use.

Below we show:

1. how to solve a problem given by a problem description file, and
2. 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 `ProblemDefinition` instance is created - this is the input file coming to life. Let us call the instance `problem`.
  - The `problem` 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. Prior to solution, `problem.time_update()` function has to be called to setup boundary conditions, material parameters and other potentially time-dependent data. This holds also for stationary problems with a single “time step”.
4. The solution is then obtained by calling `problem.solve()` function.
5. Finally, the solution can be stored using `problem.save_state()`

The above last three steps are essentially repeated for each time step. So that is it - using the code a black-box PDE solver shields the user from having to create the `ProblemDefinition` 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 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.

### 3.3.1 Invoking *SfePy* from the command line

This section introduces the basics of running *SfePy* on the command line. The `$` indicates the command prompt of your terminal.

- The script `simple.py` is the most basic starting point in *SfePy*. It is invoked as follows:

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

- `examples/diffusion/poisson.py` is the *SfePy* *problem description* file, which defines the problem to be solved in terms *SfePy* can understand
- Running the above command creates the output file `cylinder.vtk` in the *SfePy* top-level directory

- *SfePy* can also be invoked interactively with the `isfepy` script:

```
$ ./isfepy
```

- Follow the help information printed on startup to solve the Poisson’s equation example above

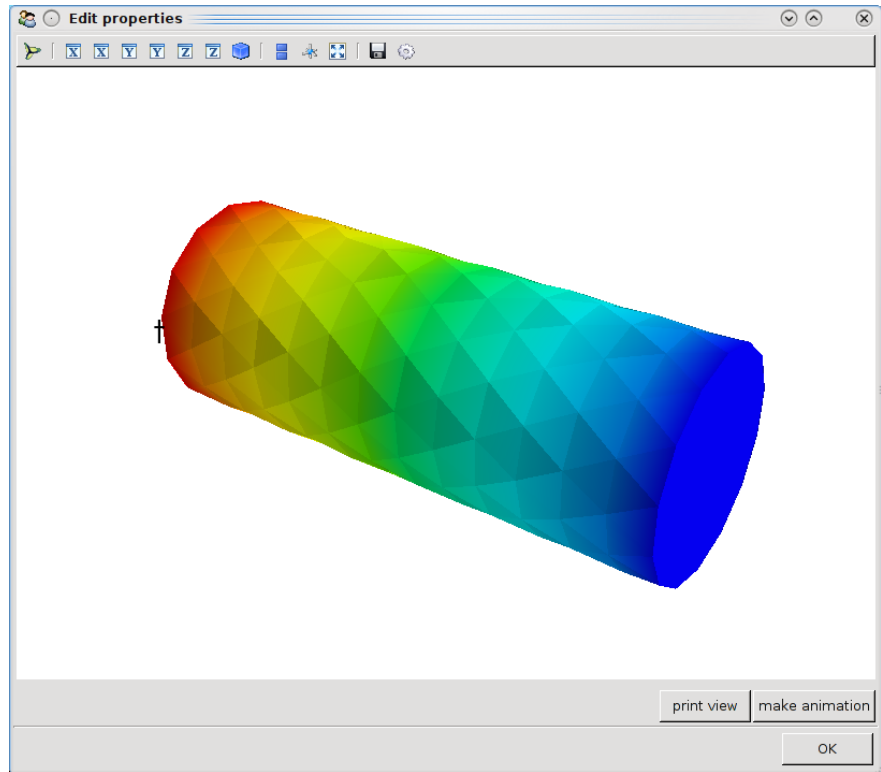
### 3.3.2 Postprocessing the results

- The `postproc.py` script can be used for quick postprocessing and visualization of the *SfePy* output files. It requires `mayavi2` installed on your system.

- As a simple example, try:

```
$ ./postproc.py cylinder.vtk
```

- The following interactive 3D window should display:



- 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.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:

$$c\Delta T = f \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma, \quad (3.6)$$

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 (3.6) in *weak formulation* (3.5). The  $f = 0$ ,  $g = \frac{\partial T}{\partial n} = 0$ . So only one term in weak form (3.5) remains:

$$\int_{\Omega} c \nabla T \cdot \nabla s = 0, \quad \forall s \in V_0. \quad (3.7)$$

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.

### 3.4.1 Long syntax of keywords

The example uses **long syntax** of the keywords. In next subsection, we show the same example written in **short syntax**.

Open the `examples/diffusion/poisson.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.

```
material_2 = {
    'name' : 'coef',
    'values' : {'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.

```
region_1000 = {
    'name' : 'Omega',
    'select' : 'elements of group 6',
}

region_03 = {
    'name' : 'Gamma_Left',
    'select' : 'nodes in (x < 0.00001)',
}

region_4 = {
    'name' : 'Gamma_Right',
    'select' : 'nodes in (x > 0.099999)',
}
```

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.

```
field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 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.

```
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',
}
```

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

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

```
ebc_1 = {
    'name' : 't1',
    'region' : 'Gamma_Left',
    'dofs' : {'t.0' : 2.0},
}

ebc_2 = {
    'name' : 't2',
    'region' : 'Gamma_Right',
    'dofs' : {'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.

```
integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'order' : 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.i1.Omega( coef.val, s, t ) = 0""
}
```

The equation above directly corresponds to the discrete version of (3.7), 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 definition 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 `il` 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.

```
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
    'method' : 'auto',
}
```

Here, we specify which kind of solver to use for linear equations.

```
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_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}
```

Here, we specify the nonlinear solver kind and options. The convergence parameters can be adjusted if necessary, otherwise leave the default.

Even linear problems are solved by a nonlinear solver (KISS rule) - only one iteration is needed and the final residual is obtained for free.

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

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

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

### 3.4.2 Short syntax of keywords

The same diffusion equation example as above in **short syntax** reads, see `examples/diffusion/poisson_short_syntax.py`, as follows:

```
1  r"""
2  Laplace equation.
3
4  The same example as poisson.py, but using the short syntax of keywords.
5
6  Find :math:`t` such that:
7
8  .. math::
9      \int_{\Omega} c \nabla s \cdot \nabla t
10     = 0
11     \;, \quad \forall s \;.
12 """
13 from sfepy import data_dir
14
15 filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'
16
17 materials = {
18     'coef' : ({'val' : 1.0},),
19 }
20
21 regions = {
22     'Omega' : ('all', {}), # or 'elements of group 6'
23     'Gamma_Left' : ('nodes in (x < 0.00001)', {}),
24     'Gamma_Right' : ('nodes in (x > 0.099999)', {}),
25 }
26
27 fields = {
28     'temperature' : ('real', 1, 'Omega', 1),
29 }
30
31 variables = {
32     't' : ('unknown field', 'temperature', 0),
33     's' : ('test field', 'temperature', 't'),
34 }
35
36 ebcs = {
37     't1' : ('Gamma_Left', {'t.0' : 2.0}),
38     't2' : ('Gamma_Right', {'t.0' : -2.0}),
39 }
40
41 integrals = {
42     'i1' : ('v', 2),
43 }
44
45 equations = {
46     'Temperature' : """dw_laplace.i1.Omega( coef.val, s, t ) = 0"""
47 }
48
49 solvers = {
50     'ls' : ('ls.scipy_direct', {}),
51     'newton' : ('nls.newton',
52                 {'i_max' : 1,
53                  'eps_a' : 1e-10,
54                 }),
```

```

55 }
56
57 options = {
58     'nls' : 'newton',
59     'ls'  : 'ls',
60 }

```

## 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 `isfepy` for the explanation, but regular Python shell, or IPython would do as well, provided the proper modules are imported (see the help information printed on startup of `isfepy`).

We wish to solve the following linear elasticity problem:

$$-\frac{\partial \sigma_{ij}(\underline{u})}{\partial x_j} + f_i = 0 \text{ in } \Omega, \quad \underline{u} = 0 \text{ on } \Gamma_1, \quad u_1 = \bar{u}_1 \text{ on } \Gamma_2, \quad (3.8)$$

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 (3.8) is

$$\int_{\Omega} D_{ijkl} e_{kl}(\underline{u}) e_{ij}(\underline{v}) + \int_{\Omega} f_i v_i = 0, \quad (3.9)$$

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 below in [Complete Example as a Script](#).

Run the `isfepy` script:

```
$ ./isfepy
```

The output should look like this:

```
Python 2.6.5 console for SfePy 2010.2-git-11cfd34 (8c4664610ed4b85851966326aaa7ce36e560ce7a)
```

These commands were executed:

```
>>> from sfepy.base.base import *
>>> from sfepy.fem import *
>>> from sfepy.applications import solve_pde
>>> from sfepy.postprocess import Viewer
```

When in SfePy source directory, try:

```
>>> pb, vec, data = solve_pde('examples/diffusion/poisson.py')
>>> view = Viewer(pb.get_output_name())
>>> view()
```

When in another directory (and SfePy is installed), try:

```
>>> from sfepy import data_dir
>>> pb, vec, data = solve_pde(data_dir + '/examples/diffusion/poisson.py')
>>> view = Viewer(pb.get_output_name())
>>> view()
```

Documentation can be found at <http://sfepy.org>

```
In [1]:
```

Read a finite element mesh, that defines the domain  $\Omega$ .

```
In [1]: mesh = Mesh.from_file('meshes/2d/rectangle_tri.mesh')
```

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

```
In [2]: domain = Domain('domain', mesh)
sfepy: setting up domain edges...
sfepy: ...done in 0.01 s
```

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 [3]: min_x, max_x = domain.get_mesh_bounding_box()[:,0]
In [4]: eps = 1e-8 * (max_x - min_x)
In [5]: omega = domain.create_region('Omega', 'all')
In [6]: gammal = domain.create_region('Gammal',
...:                                 'nodes in x < %.10f' % (min_x + eps))
In [7]: gamma2 = domain.create_region('Gamma2',
...:                                 'nodes in x > %.10f' % (max_x - eps))
```

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

```
In [8]: field = Field('fu', nm.float64, 'vector', omega,
...:                 space='H1', poly_space_base='lagrange', approx_order=2)
```

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

```
In [9]: u = FieldVariable('u', 'unknown', field, mesh.dim)
In [10]: v = FieldVariable('v', 'test', field, mesh.dim,
...:                      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 [11]: m = Material('m', lam=1.0, mu=1.0)
In [12]: 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 [14]: integral = Integral('i', order=3)
```

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

```
In [15]: from sfepy.terms import Term
In [16]: t1 = Term.new('dw_lin_elastic_iso(m.lam, m.mu, v, u)',
...:                  integral, omega, m=m, v=v, u=u)
In [17]: t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
In [18]: eq = Equation('balance', t1 + t2)
In [19]: eqs = Equations([eq])
sfepy: setting up dof connectivities...
sfepy: ...done in 0.00 s
sfepy: describing geometries...
sfepy: ...done in 0.00 s
```



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 [20]: from sfepy.fem.conditions import Conditions, EssentialBC
In [21]: fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})
In [22]: def shift_u_fun(ts, coors, bc=None, shift=0.0):
....:     val = shift * coors[:,1]**2
....:     return val
....:
In [23]: bc_fun = Function('shift_u_fun', shift_u_fun,
....:                     extra_args={'shift' : 0.01})
In [24]: 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 [25]: from sfepy.solvers.ls import ScipyDirect
In [26]: from sfepy.solvers.nls import Newton
In [27]: ls = ScipyDirect({})
In [28]: nls_status = IndexedStruct()
In [29]: nls = Newton({}, lin_solver=ls, status=nls_status)
```

Now we are ready to create a ProblemDefinition instance. Note that the step above is not really necessary - the above solvers are constructed by default. We did them to get the *nls\_status*.

```
In [30]: pb = ProblemDefinition('elasticity', equations=eqs, nls=nls, ls=ls)
```

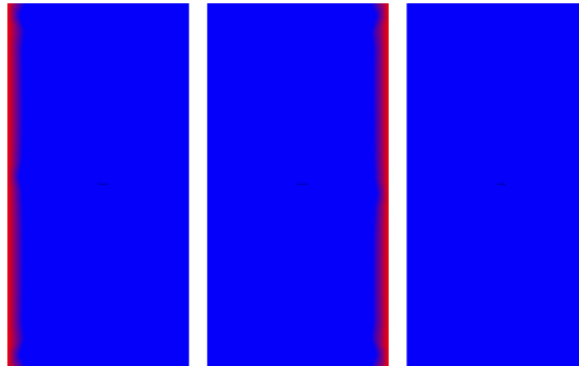
The ProblemDefinition has several handy methods for debugging. Let us try saving the regions into a VTK file.

```
In [31]: pb.save_regions_as_groups('regions')
sfepy: saving regions as groups...
sfepy:  Omega
sfepy:  Gamma1
sfepy:  Gamma2
sfepy:  Gamma1
sfepy:  ...done
```

And view them.

```
In [32]: view = Viewer('regions.vtk')
In [33]: view()
sfepy: point scalars Gamma1 [ 0.  0.  0.]
sfepy: point scalars Gamma2 [ 11.  0.  0.]
sfepy: point scalars Omega [ 22.  0.  0.]
Out[33]: <sfepy.postprocess.viewer.ViewerGUI object at 0x93ea5f0>
```

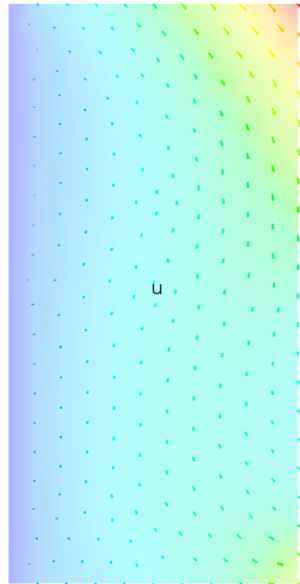
You should see this:



Finally, we apply the boundary conditions, solve the problem, save and view the results.

```
In [34]: pb.time_update(ebcs=Conditions([fix_u, shift_u]))
sfepy: updating materials...
sfepy:      m
sfepy:      f
sfepy: ...done in 0.01 s
sfepy: updating variables...
sfepy: ...done
sfepy: matrix shape: (1815, 1815)
sfepy: assembling matrix graph...
sfepy: ...done in 0.00 s
sfepy: matrix structural nonzeros: 39145 (1.19e-02% fill)
In [35]: vec = pb.solve()
sfepy: nls: iter: 0, residual: 1.343114e+01 (rel: 1.000000e+00)
sfepy:   rezidual:    0.00 [s]
sfepy:     solve:    0.01 [s]
sfepy:    matrix:    0.00 [s]
sfepy: nls: iter: 1, residual: 2.567997e-14 (rel: 1.911972e-15)
In [36]: print nls_status
-----> print(nls_status)
IndexedStruct
  condition:
    0
  err:
    2.56799662867e-14
  err0:
    13.4311385972
  time_stats:
    {'rezidual': 0.0, 'solve': 0.010000000000000001563, 'matrix': 0.0}
In [37]: pb.save_state('linear_elasticity.vtk', vec)
In [38]: view = Viewer('linear_elasticity.vtk')
In [39]: view()
sfepy: point vectors u [ 0.  0.  0.]
Out[39]: <sfepy.postprocess.viewer.ViewerGUI object at 0xad61bf0>
```

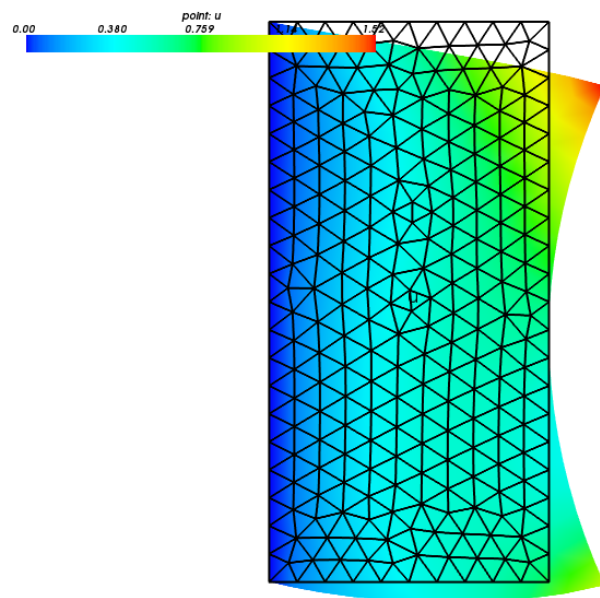
This is the resulting image:



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

```
In [56]: view(vector_mode='warp_norm', rel_scaling=2,
....:         is_scalar_bar=True, is_wireframe=True)
sfepy: point vectors u [ 0.  0.  0.]
Out[56]: <sfepy.postprocess.viewer.ViewerGUI object at 0xad61bf0>
```

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_elasticity.py`. It should be run from the *SfePy* source directory so that it finds the mesh file.

```
1  #!/usr/bin/env python
2  from optparse import OptionParser
3  import numpy as nm
4
5  import sys
6  sys.path.append('.')
7
8  from sfepy.base.base import IndexedStruct
9  from sfepy.fem import (Mesh, Domain, H1NodalVolumeField, FieldVariable,
10                        Material, Integral, Function, Equation, Equations,
11                        ProblemDefinition)
12  from sfepy.terms import Term
13  from sfepy.fem.conditions import Conditions, EssentialBC
14  from sfepy.solvers.ls import ScipyDirect
15  from sfepy.solvers.nls import Newton
16  from sfepy.postprocess import Viewer
17
18  def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):
19      """
20      Define a displacement depending on the y coordinate.
21      """
22      val = shift * coors[:,1]**2
23
24      return val
25
26  usage = """%prog [options]"""
27  help = {
28      'show' : 'show the results figure',
29  }
30
31  def main():
32      from sfepy import data_dir
33
34      parser = OptionParser(usage=usage, version='%prog')
35      parser.add_option('-s', '--show',
36                      action="store_true", dest='show',
37                      default=False, help=help['show'])
38      options, args = parser.parse_args()
39
40      mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
41      domain = Domain('domain', mesh)
42
43      min_x, max_x = domain.get_mesh_bounding_box()[:,0]
44      eps = 1e-8 * (max_x - min_x)
45      omega = domain.create_region('Omega', 'all')
46      gamma1 = domain.create_region('Gamma1',
47                                   'nodes in x < %.10f' % (min_x + eps))
48      gamma2 = domain.create_region('Gamma2',
49                                   'nodes in x > %.10f' % (max_x - eps))
50
51      field = H1NodalVolumeField('fu', nm.float64, 'vector', omega,
52                                approx_order=2)
53
54      u = FieldVariable('u', 'unknown', field, mesh.dim)
```

```

55     v = FieldVariable('v', 'test', field, mesh.dim, primary_var_name='u')
56
57     m = Material('m', lam=1.0, mu=1.0)
58     f = Material('f', val=[[0.02], [0.01]])
59
60     integral = Integral('i', order=3)
61
62     t1 = Term.new('dw_lin_elastic_iso(m.lam, m.mu, v, u)',
63                 integral, omega, m=m, v=v, u=u)
64     t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
65     eq = Equation('balance', t1 + t2)
66     eqs = Equations([eq])
67
68     fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})
69
70     bc_fun = Function('shift_u_fun', shift_u_fun, extra_args={'shift' : 0.01})
71     shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})
72
73     ls = ScipyDirect({})
74
75     nls_status = IndexedStruct()
76     nls = Newton({}, lin_solver=ls, status=nls_status)
77
78     pb = ProblemDefinition('elasticity', equations=eqs, nls=nls, ls=ls)
79     pb.save_regions_as_groups('regions')
80
81     pb.time_update(ebcs=Conditions([fix_u, shift_u]))
82
83     vec = pb.solve()
84     print nls_status
85
86     pb.save_state('linear_elasticity.vtk', vec)
87
88     if options.show:
89         view = Viewer('linear_elasticity.vtk')
90         view(vector_mode='warp_norm', rel_scaling=2,
91             is_scalar_bar=True, is_wireframe=True)
92
93 if __name__ == '__main__':
94     main()

```



# 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

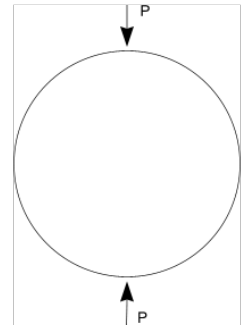
A beginner's tutorial highlighting the basics of *SfePy*.

## 4.1 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.

### 4.1.1 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:

$$\sigma_t = \frac{2P}{\pi t D} \quad (4.1)$$

$$\sigma_c = \frac{6P}{\pi t D} \quad (4.2)$$

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

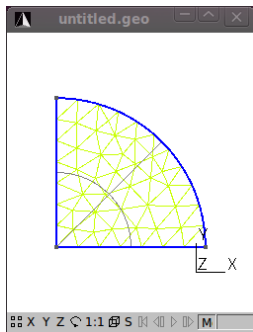
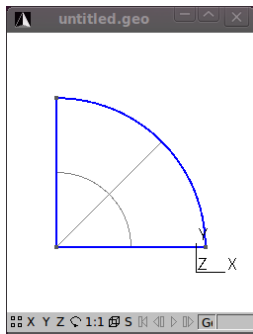
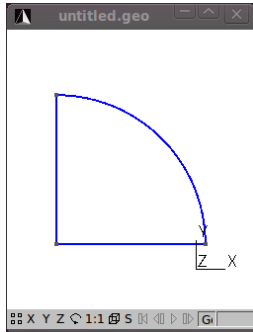
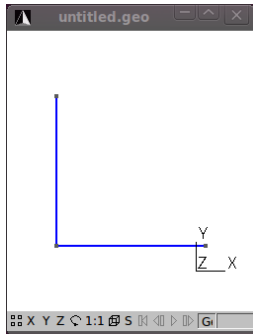
## 4.2 Meshing

Assuming plane strain conditions, the indirect tensile test may be modelled using a 2-D 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 2-D 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: (0.0 0.0), (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.

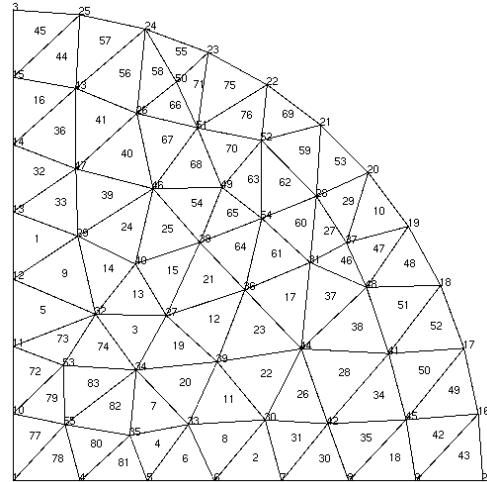




That's the meshing done. Save the mesh in a format that *SfePy* recognizes. For now use the **medit** .mesh format e.g. its2D.mesh.

**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 3-D format and includes some extra geometry items that should be deleted. Reformatted the mesh file to a 2-D 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.

## 4.3 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:

```

1  r"""
2  Diametrically point loaded 2-D disk. See :ref:'sec-primer'.
3
4  Find :math: \ul{u} ' such that:
5
6  .. math::
7      \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
8      = 0
9      \;, \quad \forall \ul{v} \;,
10
11  where
12
13  .. math::
14      D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
15      \lambda \delta_{ij} \delta_{kl}
16      \;.
17  """

```

```

18 from sfepy.mechanics.matcoefs import lame_from_youngpoisson
19 from sfepy.fem.utils import refine_mesh
20 from sfepy import data_dir
21
22 # Fix the mesh file name if you run this file outside the SfePy directory.
23 filename_mesh = data_dir + '/meshes/2d/its2D.mesh'
24
25 refinement_level = 0
26 filename_mesh = refine_mesh(filename_mesh, refinement_level)
27
28 output_dir = '.' # set this to a valid directory you have write access to
29
30 young = 2000.0 # Young's modulus [MPa]
31 poisson = 0.4 # Poisson's ratio
32
33 options = {
34     'output_dir' : output_dir,
35 }
36
37 regions = {
38     'Omega' : ('all', {}),
39     'Left' : ('nodes in (x < 0.001)', {}),
40     'Bottom' : ('nodes in (y < 0.001)', {}),
41     'Top' : ('node 2', {}),
42 }
43
44 materials = {
45     'Asphalt' : ({
46         'lam' : lame_from_youngpoisson(young, poisson)[0],
47         'mu' : lame_from_youngpoisson(young, poisson)[1],
48     },),
49     'Load' : ({'.val' : [0.0, -1000.0]},),
50 }
51
52 fields = {
53     'displacement': ('real', 'vector', 'Omega', 1),
54 }
55
56 equations = {
57     'balance_of_forces' :
58     """dw_lin_elastic_iso.2.Omega(Asphalt.lam, Asphalt.mu, v, u )
59     = dw_point_load.0.Top(Load.val, v)""",
60 }
61
62 variables = {
63     'u' : ('unknown field', 'displacement', 0),
64     'v' : ('test field', 'displacement', 'u'),
65 }
66
67 ebcs = {
68     'XSym' : ('Bottom', {'u.1' : 0.0}),
69     'YSym' : ('Left', {'u.0' : 0.0}),
70 }
71
72 solvers = {
73     'ls' : ('ls.scipy_direct', {}),
74     'newton' : ('nls.newton', {
75         'i_max' : 1,

```

```
76         'eps_a' : 1e-6,  
77         'problem' : 'nonlinear'  
78     }},  
79 }
```

Download and open the file 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:

```
1 regions = {  
2     'Omega' : ('all', {}),  
3     'Left' : ('nodes in (x < 0.001)', {}),  
4     'Bottom' : ('nodes in (y < 0.001)', {}),  
5     'Top' : ('node 2', {}),  
6 }
```

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:

```
1 ebc = {  
2     'XSym' : ('Bottom', {'u.1' : 0.0}),  
3     'YSym' : ('Left', {'u.0' : 0.0}),  
4 }
```

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:

```
1 materials = {  
2     'Asphalt' : ({  
3         'lam' : lame_from_youngpoisson(young, poisson)[0],  
4         'mu' : lame_from_youngpoisson(young, poisson)[1],  
5     })),  
6     'Load' : ({'.val' : [0.0, -1000.0]}),  
7 }
```

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

We provided the material constants in terms of Young's modulus and Poisson's ratio, but the linear elastic isotropic equation used requires as input Lamé's parameters. The `lame_from_youngpoisson()` function is thus used for 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.

## 4.4 Running SfePy

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

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

**Note:** For the purpose of this tutorial it is assumed that the problem definition 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`.

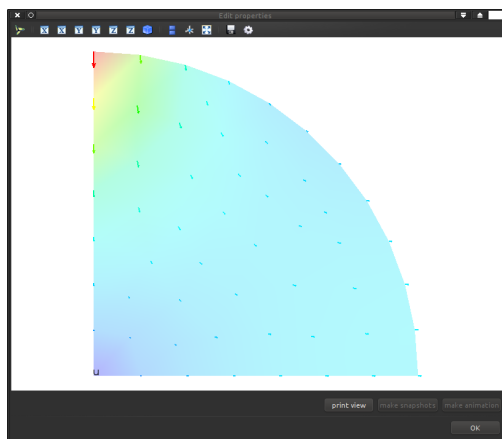
The `vtk` format is an `ascii` format. Open the file using a text editor. You'll notice that the output file includes separate sections:

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

SfePy includes 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 (with 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.



## 4.5 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`.

```
1  r"""
2  Diametrically point loaded 2-D disk with postprocessing. See
3  :ref:'sec-primer'.
4
5  Find :math:\ul{u}' such that:
6
7  .. math::
8      \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
9      = 0
10     \;, \quad \forall \ul{v} \;,
11
12  where
13
14  .. math::
15      D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
16      \lambda \delta_{ij} \delta_{kl}
17      \;.
18  """
19
20  from its2D_1 import *
21
22  from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
23
24  def stress_strain(out, pb, state, extend=False):
25      """
26      Calculate and output strain and stress for given displacements.
27      """
28      from sfepy.base.base import Struct
29
30      ev = pb.evaluate
31      strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
32      stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg')
33
34      out['cauchy_strain'] = Struct(name='output_data', mode='cell',
35                                   data=strain, dofs=None)
36      out['cauchy_stress'] = Struct(name='output_data', mode='cell',
37                                    data=stress, dofs=None)
38
39      return out
40
41  asphalt = materials['Asphalt'][0]
42  asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
43  options.update({'post_process_hook' : 'stress_strain', })
```

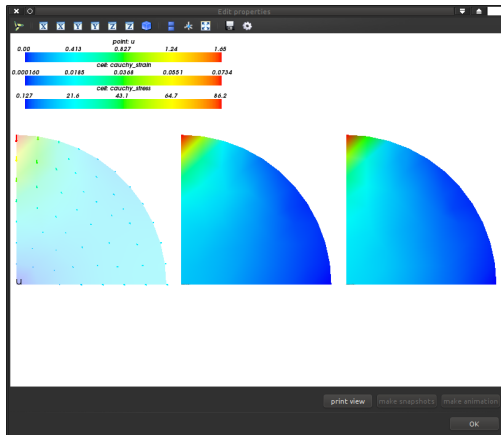
The updated file imports all of the previous definitions in `its2D_1.py`. The stress function (`de_cauchy_stress`) requires as input the stiffness tensor - thus it was necessary to update the materials accordingly. The problem options were also updated to call the `stress_strain` function as a `post_process_hook`.

Run *SfePy* to solve the updated problem and view the solution (assuring 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:  $\mathbf{f} = \mathbf{K}\mathbf{u}$ . Let's determine the force vector interactively.

### 4.5.1 Running SfePy in interactive mode

In addition to solving problems using the `simple.py` script you can also run SfePy interactively. This requires that [IPython](#) be installed. To run *SfePy* interactively, use *isfepy*:

```
$ ./isfepy
```

Once *isfepy* loads up, issue the following command:

```
In [1]: 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 [2]: u = state()
```

```
In [3]: u
Out[3]:
array([[ 0.          ,  0.          ,  0.37376671, ..., -0.19923848,
         0.08820237, -0.11201528])
```

```
In [4]: u.shape
Out[4]: (110,)
```

```
In [5]: u.shape = (55, 2)
```

```
In [6]: u
Out[6]:
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 [7]: u = state.get_parts()['u']

In [8]: u
Out[8]:
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 [9]: K = pb.mtx_a

In [10]: K
Out[10]:
<94x94 sparse matrix of type '<type 'numpy.float64''>'
      with 1070 stored elements in Compressed Sparse Row format>

In [11]: 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 [12]: K.shape
Out[12]: (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 [13]: pb.remove_bcs()
```

Now we can calculate the force vector ( $f$ ):

```
In [14]: f = pb.evaluator.eval_residual(u)
```

```
In [15]: f.shape
Out[15]: (110,)
```

```
In [16]: f
Out[16]:
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 [13]:

```
In [17]: 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:

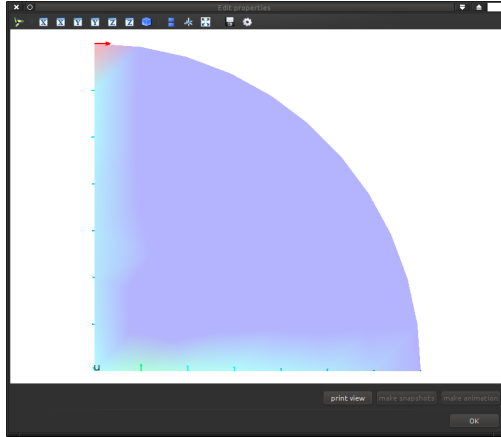
```
In [18]: state = pb.create_state()
```

```
In [19]: state.set_full(f)
```

```
In [20]: out = state.create_output_dict()
```

```
In [21]: 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 [22]: f.shape = (55, 2)
```

```
In [23]: f[2]
```

```
Out[23]: array([ 6.20373272e+02, -1.13686838e-13])
```

Great, we have an almost zero residual vertical load or force apparent at node 2 i.e.  $-1.13686838 \times 10^{-13}$  Newton. Let us now check the stress at node 0, the centre of the specimen.

## 4.5.2 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.

```
1  r"""
2  Diametrically point loaded 2-D disk with nodal stress calculation. See
3  :ref:'sec-primer'.
4
5  Find :math:\ul{u}' such that:
6
7  .. math::
8      \int_{\Omega} D_{ijkl} \ e_{ij}(\ul{v}) \ e_{kl}(\ul{u})
9      = 0
10     \;, \quad \forall \ul{v} \;,
11
12  where
13
14  .. math::
15      D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
16      \lambda \delta_{ij} \delta_{kl}
17      \;.
18  """
19  from its2D_1 import *
20
21  from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
22  from sfepy.fem.geometry_element import geometry_data
23  from sfepy.fem import H1NodalVolumeField, FieldVariable
24  import numpy as nm
25
```

```

26 gdata = geometry_data['2_3']
27 nc = len(gdata.coors)
28
29 def nodal_stress(out, pb, state, extend=False):
30     '''
31     Calculate stresses at nodal points.
32     '''
33
34     # Point load.
35     mat = pb.get_materials()['Load']
36     P = 2.0 * mat.get_data('special', None, 'val')[1]
37
38     # Calculate nodal stress.
39     pb.time_update()
40
41     stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D, u)', mode='qp')
42     sfield = H1NodalVolumeField('stress', nm.float64, (3,),
43                                pb.domain.regions['Omega'])
44     svar = FieldVariable('sigma', 'parameter', sfield, 3,
45                          primary_var_name='(set-to-None)')
46     svar.set_data_from_qp(stress, pb.integrals['ivn'])
47
48     print '\n=====
49     print 'Given load = %.2f N' % -P
50     print '\nAnalytical solution'
51     print '=====
52     print 'Horizontal tensile stress = %.5e MPa/mm' % (-2.*P/(nm.pi*150.))
53     print 'Vertical compressive stress = %.5e MPa/mm' % (-6.*P/(nm.pi*150.))
54     print '\nFEM solution'
55     print '=====
56     print 'Horizontal tensile stress = %.5e MPa/mm' % (svar()[0][0])
57     print 'Vertical compressive stress = %.5e MPa/mm' % (-svar()[0][1])
58     print '=====
59     return out
60
61 asphalt = materials['Asphalt'][0]
62 asphalt.update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
63 options.update({'post_process_hook' : 'nodal_stress',})
64
65 integrals = {
66     'ivn' : ('v', 'custom', gdata.coors, [gdata.volume / nc] * nc),
67 }

```

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 *isfepy*:

```
In [23]: from sfepy.fem.geometry_element import geometry_data

In [24]: gdata = geometry_data['2_3']
In [25]: nc = len(gdata.coors)
In [26]: ivn = Integral('ivn', order=-1,
.....:                  coors=gdata.coors, weights=[gdata.volume / nc] * nc)

In [27]: pb, state = solve_pde('examples/linear_elasticity/its2D_2.py')

In [28]: stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D,u)',
.....:                        mode='qp', integrals=Integrals([ivn]))
In [29]: sfield = Field('stress', nm.float64, (3,), pb.domain.regions['Omega'])
In [30]: svar = FieldVariable('sigma', 'parameter', sfield, 3,
.....:                        primary_var_name='(set-to-None)')
In [31]: svar.set_data_from_qp(stress, ivn)

In [32]: print 'Horizontal tensile stress = %.5e MPa/mm' % (svar()[0][0])
Horizontal tensile stress = 7.57220e+00 MPa/mm
In [33]: print 'Vertical compressive stress = %.5e MPa/mm' % (-svar()[0][1])
Vertical compressive stress = 2.58660e+01 MPa/mm

In [34]: mat = pb.get_materials()['Load']
In [35]: P = 2.0 * mat.get_data('special', None, 'val')[1]
In [36]: P
Out[36]: -2000.0

In [37]: print 'Horizontal tensile stress = %.5e MPa/mm' % (-2.*P/(nm.pi*150.))
Horizontal tensile stress = 8.48826e+00 MPa/mm
In [38]: 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.

## 4.6 Probing

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

```

1  r"""
2  Diametrically point loaded 2-D disk with postprocessing and probes. See
3  :ref: 'sec-primer'.
4
5  Find :math: \ul{u} such that:
6
7  .. math::
8      \int_{\Omega} D_{ijkl} \ul{e}_{ij} \ul{e}_{kl} \ul{u}
9      = 0
10     \;, \quad \forall \ul{v} \;,
11
12  where
13
14  .. math::
15      D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
16      \lambda \delta_{ij} \delta_{kl}
17      \;.
18  """
19  from its2D_1 import *
20
21  from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
22
23  def stress_strain(out, pb, state, extend=False):
24      """
25      Calculate and output strain and stress for given displacements.
26      """
27      from sfepy.base.base import Struct
28
29      ev = pb.evaluate
30      strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
31      stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg')
32
33      out['cauchy_strain'] = Struct(name='output_data', mode='cell',
34                                   data=strain, dofs=None)
35      out['cauchy_stress'] = Struct(name='output_data', mode='cell',
36                                   data=stress, dofs=None)
37
38      return out
39
40  def gen_lines(problem):
41      from sfepy.fem.probes import LineProbe
42      mesh = problem.domain.mesh
43      ps0 = [[0.0, 0.0], [0.0, 0.0]]
44      ps1 = [[75.0, 0.0], [0.0, 75.0]]
45
46      # Use adaptive probe with 10 initial points.
47      n_point = -10
48
49      labels = ['%s -> %s' % (p0, p1) for p0, p1 in zip(ps0, ps1)]
50      probes = []
51      for ip in xrange(len(ps0)):
52          p0, p1 = ps0[ip], ps1[ip]
53          probes.append(LineProbe(p0, p1, n_point, mesh))

```

```
54
55     return probes, labels
56
57
58 def probe_hook(data, probe, label, problem):
59     import matplotlib.pyplot as plt
60     import matplotlib.font_manager as fm
61
62     def get_it(name, var_name):
63         var = problem.create_variables([var_name])[var_name]
64         var.set_data(data[name].data)
65
66         pars, vals = probe(var)
67         vals = vals.squeeze()
68         return pars, vals
69
70     results = {}
71     results['u'] = get_it('u', 'u')
72     results['cauchy_strain'] = get_it('cauchy_strain', 's')
73     results['cauchy_stress'] = get_it('cauchy_stress', 's')
74
75     fig = plt.figure()
76     plt.clf()
77     fig.subplots_adjust(hspace=0.4)
78     plt.subplot(311)
79     pars, vals = results['u']
80     for ic in range(vals.shape[1]):
81         plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
82                 lw=1, ls='-', marker='+', ms=3)
83     plt.ylabel('displacements')
84     plt.xlabel('probe %s' % label, fontsize=8)
85     plt.legend(loc='best', prop=fm.FontProperties(size=10))
86
87     sym_indices = ['11', '22', '12']
88
89     plt.subplot(312)
90     pars, vals = results['cauchy_strain']
91     for ic in range(vals.shape[1]):
92         plt.plot(pars, vals[:,ic], label=r'$\epsilon_{%s}$' % sym_indices[ic],
93                 lw=1, ls='-', marker='+', ms=3)
94     plt.ylabel('Cauchy strain')
95     plt.xlabel('probe %s' % label, fontsize=8)
96     plt.legend(loc='best', prop=fm.FontProperties(size=8))
97
98     plt.subplot(313)
99     pars, vals = results['cauchy_stress']
100    for ic in range(vals.shape[1]):
101        plt.plot(pars, vals[:,ic], label=r'$\sigma_{%s}$' % sym_indices[ic],
102                lw=1, ls='-', marker='+', ms=3)
103    plt.ylabel('Cauchy stress')
104    plt.xlabel('probe %s' % label, fontsize=8)
105    plt.legend(loc='best', prop=fm.FontProperties(size=8))
106
107    return plt.gcf(), results
108
109 materials['Asphalt'][0].update({'D' : stiffness_from_youngpoisson(2, young, poisson)})
110
111 # Update fields and variables to be able to use probes for tensors.
```

```

112 fields.update({
113     'sym_tensor': ('real', 3, 'Omega', 0),
114 })
115
116 variables.update({
117     's' : ('parameter field', 'sym_tensor', None),
118 })
119
120 options.update({
121     'output_format' : 'h5', # VTK reader cannot read cell data yet for probing
122     'post_process_hook' : 'stress_strain',
123     'gen_probes' : 'gen_lines',
124     'probe_hook' : 'probe_hook',
125 })

```

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.

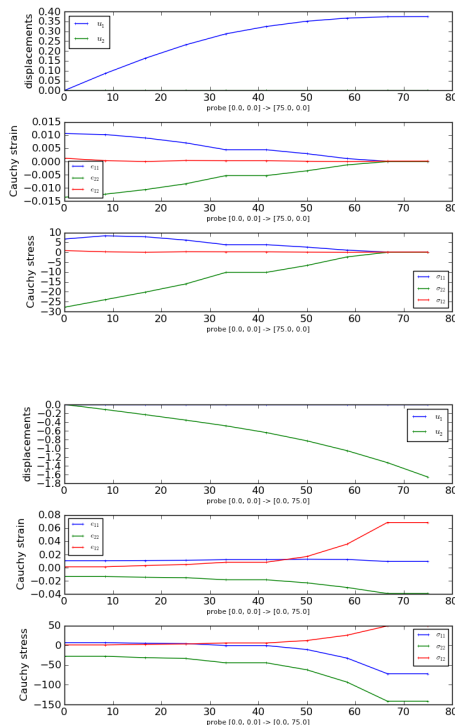
Notice that the `output_format` has been defined as `h5`. To apply probing first solve the problem as usual:

```
$ ./simple.py its2D_4.py
```

This will write the solution to the output directory indicated. Then run the *SfePy* `probe.py` script on the solution:

```
$ ./probe.py its2D_4.py <sfepy output path>/its2D.h5
```

The results of the probing will be written to text files and the following figures will be generated. These figures 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 script.



The end.





# USER'S GUIDE

## Table of Contents

- Running a Simulation
  - Basic Usage
  - Surface Extraction
  - Applications
  - Stand-Alone Examples
  - Running Tests
    - \* Common Tasks
  - Computations and Examples
    - \* Common Tasks
- Visualization of Results
- Problem Description File
  - FE Mesh
  - Regions
  - Fields
  - Variables
  - Integrals
  - 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
- Available Solvers
  - Nonlinear Solvers
  - Linear Solvers

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

## 5.1 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.

### 5.1.1 Basic Usage

- `$ ./simple.py examples/diffusion/poisson.py`
  - Creates `cylinder.vtk`
- `$ ./simple.py examples/navier_stokes/stokes.py`
  - Creates `channels_symm944t.vtk`
- `$ ./runTests.py`
  - See [Running Tests](#)
- `$ ./isfepy`
  - Follow the help information printed on startup

### 5.1.2 Surface Extraction

- `$ ./findSurf.py meshes/quantum/cube.node -`
  - Creates `surf_cube.mesh`

### 5.1.3 Applications

- Phononic Materials
  - `$ ./phonon.py -p examples/phononic/band_gaps.py`
    - \* see `examples/phononic/output/`
- `schroedinger.py`
  - (order is important below):
    1. `$ ./schroedinger.py --2d --create-mesh`
    2. `$ ./schroedinger.py --2d --hydrogen`
    3. `$ ./postproc.py mesh.vtk`

### 5.1.4 Stand-Alone Examples

- `$ python examples/rs_correctors.py`

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

### 5.1.5 Running Tests

The tests are run by the `runTests.py` script:

```
$ ./runTests.py -h
Usage: runTests.py [options] [test_filename[ test_filename ...]]
```

Options:

```
--version          show program's version number and exit
-h, --help         show this help message and exit
--print-doc        print the docstring of this file (howto write new
                  tests)
-d directory, --dir=directory
                  directory with tests [default: tests]
-o directory, --output=directory
                  directory for storing test results and temporary files
                  [default: output-tests]
--debug            raise silenced exceptions to see what was wrong
--filter-none      do not filter any messages
--filter-less      filter output (suppress all except test messages)
--filter-more      filter output (suppress all except test result
                  messages)
```

### Common Tasks

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

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

- Run a particular test file, filter output:

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

- Debug a failing test:

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

### 5.1.6 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
```

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
-o filename                     basename of output file(s) [default: <basename of
                                input file>]
--format=format                 output file format, one of: {vtk, h5, mesh} [default:
                                vtk]
--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 problem state showing EBC (Dirichlet conditions)
--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}
```

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.py
./simple.py examples/diffusion/poisson.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.py # -> produces an additional .vtk file with
```

## 5.2 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 runTests.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
```

- same as above, but slower frame rate

```
$ python postproc.py output-tests/test_hyperelastic.*.vtk --ranges=u,0,0
```

#### Options:

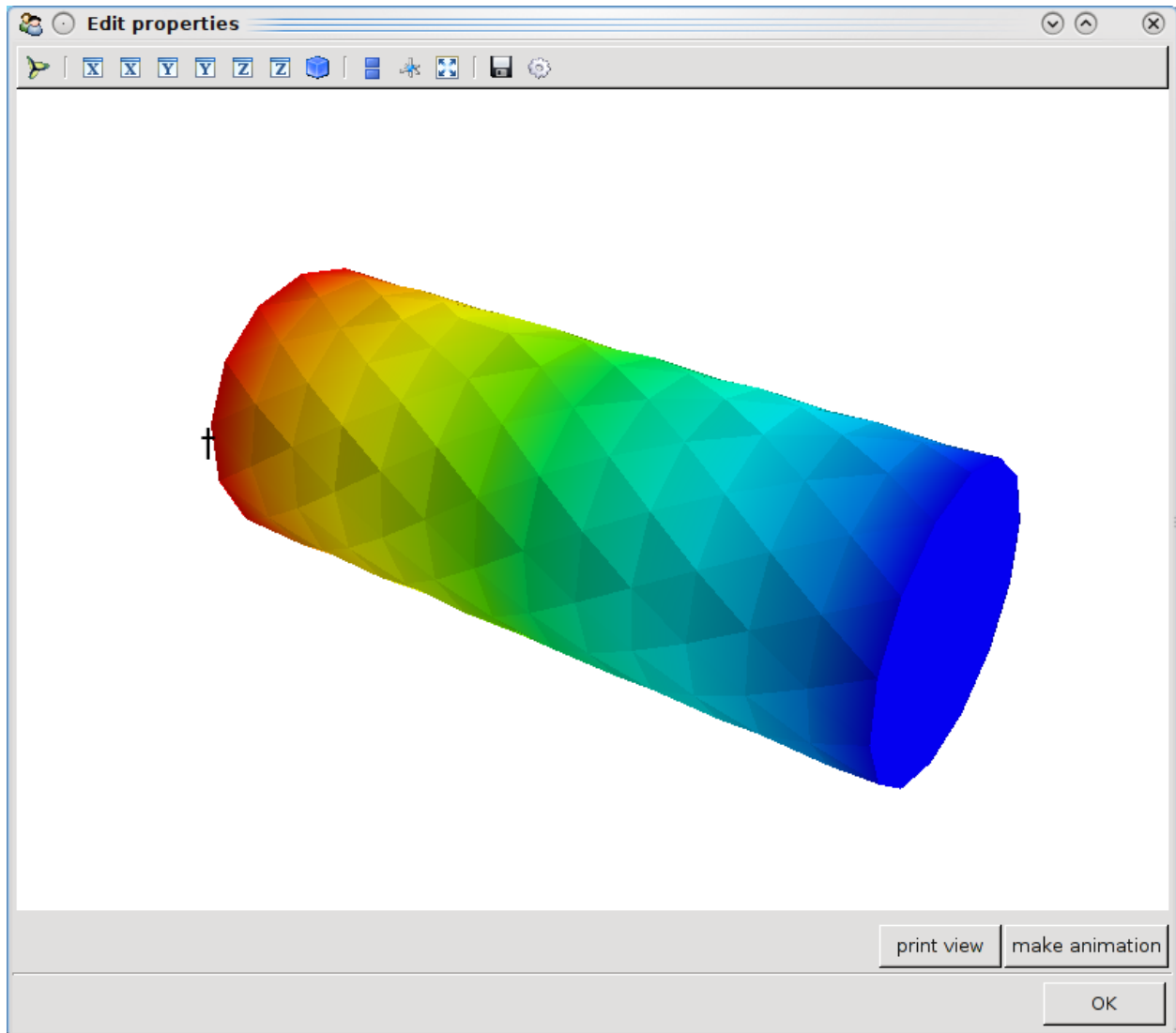
```
--version          show program's version number and exit
-h, --help         show this help message and exit
-l, --list-ranges  do not plot, only list names and ranges of all data
-n, --no-show      do not call mlab.show()
--no-offscreen     force no offscreen rendering for --no-show
--3d              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=vall,...:var_name1,...', --domain-specific='var_name0,...'
                        domain specific drawing functions and configurations
```

As a simple example, try:

```
$ ./simple.py examples/diffusion/poisson.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 l2_norm_range: 0.0 0.0
  "t" (354,) range: -2.0 2.0 l2_norm_range: 0.0106091 2.0
cell scalars
  "mat_id" (1348,) range: 6 6 l2_norm_range: 6.0 6.0
```

## 5.3 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.

### 5.3.1 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`)

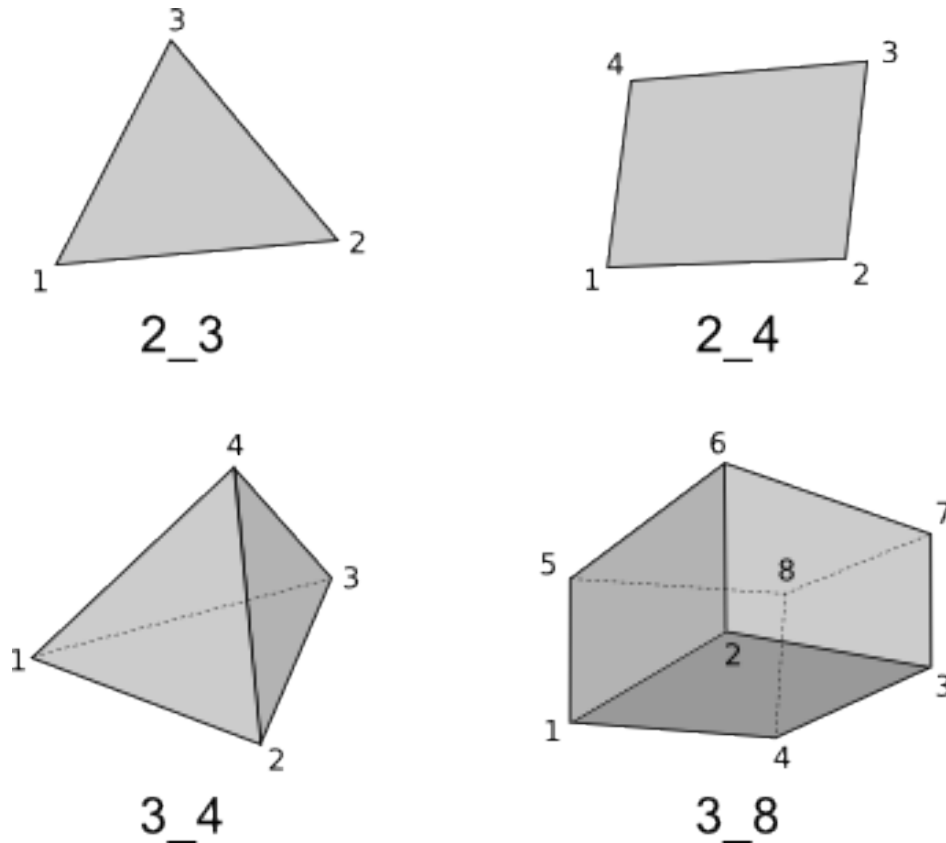
Example:

```
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:





### 5.3.2 Regions

Regions serve to select a certain part of the computational domain (= selection of nodes and elements of a FE mesh). They are used to define the boundary conditions, the domains of terms and materials etc.

- Region selection syntax
  - Entity selections
    - \* all
    - \* nodes of surface
    - \* nodes of group <integer>
    - \* nodes of group <str> (if mesh format supports reading boundary condition nodes)
    - \* nodes in <expr>
    - \* nodes by <function>
    - \* node <id>[, <id>, ...]
    - \* elements of group <integer>
    - \* elements by <efunction>
    - \* element <id>[, <id>, ...] assumes group 0 (ig = 0)
    - \* element (<ig>, <id>)[, (<ig>, <id>), ...]
    - \* r.<name of another region>

- Notation

- \* `<expr>` is a logical expression like `(y <= 0.00001) & (x < 0.11)`
- \* `<function>` is e.g., `afunction( x, y, z, otherArgs )`
- \* `<efunction>` is e.g., `efunction( domain )`

- Region operations

- \* Node-wise: `+n`, `-n`, `*n` (union, set difference, intersection)
- \* Element-wise: `+e`, `-e`, `*e` (union, set difference, intersection)

- Additional specification:

- \* `'forbid'` : `'group <integer>'` - forbid elements of listed groups
- \* `'can_cells'` : `<boolean>` - determines whether a region can have cells (volume in 3D)

- Region definition syntax

- Long syntax: a region is defined by the following Python dictionary (denote optional keys):

```
region_<number> = {
    'name' : <name>,
    'select' : <selection>,
    ['forbid'] : group <integer>[, <integer>],
    ['can_cells'] : <boolean>,
}
```

- \* Example definitions:

```
region_20 = {
    'name' : 'Left',
    'select' : 'nodes in (x < -0.499)'
}
region_21 = {
    'name' : 'Right',
    'select' : 'nodes in (x > 0.499)'
}
region_31 = {
    'name' : 'Gammal',
    'select' : """(elements of group 1 +n elements of group 4)
                  +n
                  (elements of group 2 +n elements of group 4)
                  +n
                  ((r.Left +n r.Right) *n elements of group 4)
                  """,
    'forbid' : 'group 1 2'
}
```

- Short syntax:

```
regions = {
    <name> : ( <selection>, {[<additional spec.>]} )
}
```

- \* Example definitions:

```
regions = {
    'Left' : ('nodes in (x < -0.499)', {}),
    'Right' : ('nodes in (x > 0.499)', {}),
}
```

```

    'Gamma1' : (""(elements of group 1 *n elements of group 4)
                +n
                (elements of group 2 *n elements of group 4)
                +n
                ((r.Left +n r.Right) *n elements of group 4)""",
                {'forbid' : 'group 1 2'}),
}

```

### 5.3.3 Fields

Fields correspond to FE spaces

- Long syntax:

```

field_<number> = {
    'name' : <name>,
    'dtype' : <data_type>,
    'shape' : <shape>,
    'region' : <region_name>,
    'approx_order' : <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 3 or (3,) or 'vector'; it can be other positive integer than just 1, 2, or 3
- <region\_name> is the name of region where the field is defined
- <approx\_order> is the FE approximation order, e.g. 0, 1, 2, '1B' (1 with bubble)
- Example: scalar P1 elements in 2D on a region Omega:

```

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : 'scalar',
    'region' : 'Omega',
    'approx_order' : 1
}

```

- Short syntax:

```

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

```

- 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'.

### 5.3.4 Variables

Variables use the FE approximation given by the specified field:

- Long syntax:

```
variables_<number> = {  
    'name' : <name>,  
    'kind' : <kind>,  
    'field' : <field_name>,  
    ['order' : <order>],  
    ['dual' : <variable_name>],  
    ['history' : <history_size>],  
}
```

where

- <kind> - 'unknown field', 'test field' or 'parameter field'
- <order> - primary variable - order in the global vector of unknowns
- <history\_size> - number of time steps to remember prior to current step
- Example, long syntax:

```
variable_1 = {  
    'name' : 't',  
    'kind' : 'unknown field',  
    'field' : 'temperature',  
    'order' : 0, # order in the global vector of unknowns  
    'history' : 1,  
}  
  
variable_2 = {  
    'name' : 's',  
    'kind' : 'test field',  
    'field' : 'temperature',  
    'dual' : 't',  
}
```

- Short syntax:

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

where

- <spec> - in case of: primary variable - order in the global vector of unknowns, dual variable - name of primary variable
- Example, short syntax:

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

### 5.3.5 Integrals

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

- Long syntax:

```
integral_<number> = {
    'name' : <name>,
    'kind' : <kind>,
    'order' : <order>,
}
```

where

- <name> - the integral name - it has to begin with 'i'!
- <kind> - volume 'v' or surface 's' integral
- <order> - the order of polynomials to integrate, or 'custom' for integrals with explicitly given values and weights
- Example, long syntax:

```
integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'order' : 2,
}

import numpy as nm
N = 2
integral_2 = {
    'name' : 'i2',
    'kind' : 'v',
    'order' : 'custom',
    'vals' : zip(nm.linspace( 1e-10, 0.5, N ),
                 nm.linspace( 1e-10, 0.5, N )),
    'weights' : [1./N] * N,
}
```

- Short syntax:

```
integrals = {
    <name> : (<kind>, <order>)
}
```

- Example, short syntax:

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

### 5.3.6 Boundary Conditions

The 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, long syntax:

```
ebc_<number> = {
    'name' : <name>,
    'region' : <region_name>,
    ['times' : <times_specification>,]
    'dofs' : {<dof_specification> : <value>[,
               <dof_specification> : <value>, ...]}
}
```

- Example:

```
ebc_1 = {
    'name' : 'ZeroSurface',
    'region' : 'Surface',
    'times' : [(0.5, 1.0), (2.3, 5)],
    'dofs' : {'u.all' : 0.0, 'phi.all' : 0.0},
}
```

- Dirichlet (essential) boundary conditions, short syntax:

```
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}),
}
```

### 5.3.7 Initial Conditions

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

- Long syntax:

```
ic_<number> = {
    'name' : <name>,
    'region' : <region_name>,
    'dofs' : {<dof_specification> : <value>[,
               <dof_specification> : <value>, ...]}
}
```

- Example:

```
ic_1 = {
    'name' : 'ic',
}
```

```

    'region' : 'Omega',
    'dofs' : {'T.0' : 5.0},
}

```

- Short syntax:

```

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

```

- Example:

```

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

```

### 5.3.8 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, long syntax:

```

material_10 = {
    'name' : 'm',
    'values' : {
        # This gets tiled to all physical QPs (constant function)
        'val' : [0.0, -1.0, 0.0],
        # This does not - '.' denotes a special value, e.g. a flag.
        '.val0' : [0.0, 0.1, 0.0],
    },
}

material_3 = {
    'name' : 'm2',
    'function' : 'some_function',
}

def some_function(ts, coor, region, ig, mode=None):
    out = {}
    if mode == 'qp':
        # <array of shape (coor.shape[0], n_row, n_col)>
        out['val'] = nm.ones((coor.shape[0], 1, 1), dtype=nm.float64)
    else: # special mode
        out['val0'] = True

```

- Example, short syntax:

```

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

```

- Example, short syntax, 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)}
    },),
}
```

### 5.3.9 Equations and Terms

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

#### Examples

- Laplace equation, named integral:

```
equations = {
    'Temperature' : """dw_laplace.i1.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""",
}
```

### 5.3.10 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 solver, long syntax:

```
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.umfpack',
}
```



- Nonlinear solver, long syntax:

```
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_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}
```

- Solvers, short syntax:

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

- Solver selection:

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

### 5.3.11 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):
    x, y, z = coors[:, 0], coors[:, 1], coors[:, 2]
    val = ts.step * nm.r_[x, y, z]
    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 a single one-dimensional vector with all values of the first component, then of the second one etc. concatenated together.

- function for defining usual material parameters:

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

        return {'x_coor' : val}

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

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

```
def get_pars_special(ts, coors, mode=None, region=None, ig=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, region=None, ig=None:
                  get_pars_special(ts, coors, mode, region, ig,
                                  extra_arg='hello!')),
}

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

- function combining both kinds of material parameters:

```
def get_pars_both(ts, coors, mode=None, region=None, ig=None):
    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,)
}
```

### 5.3.12 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 = {
    # 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',

    # int, number of time steps when results should be saved (spaced
    # regularly from 0 to n_step), or -1 for all time steps
    'save_steps' : -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>',
}

```

- `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

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

$$\frac{\partial T}{\partial t} + c\Delta T = 0 \text{ in } \Omega, \quad T(t) = \bar{T}(t) \text{ on } \Gamma. \quad (5.1)$$

The weak formulation of (5.1) is: Find  $T \in V$ , such that

$$\int_{\Omega} s \frac{\partial T}{\partial t} + \int_{\Omega} c \nabla T : \nabla s = 0, \quad \forall s \in V_0, \quad (5.2)$$

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

```
dw_mass_scalar.i1.Omega( s, dT/dt ) + dw_laplace.i1.Omega( coef, s, T ) = 0
```

which directly corresponds to the discrete version of (5.2): Find  $T \in V_h$ , such that

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

where  $u \approx \boldsymbol{\phi} u$ ,  $\nabla u \approx \mathbf{G} u$  for  $u \in \{s, T\}$ . The integrals over the discrete domain  $\Omega_h$  are approximated by a numerical quadrature, that is named `i1` in our case.

### 5.4.1 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 a unknown function and `<parameter>` to a known function arguments.

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

## 5.5 Term Evaluation

Terms can be evaluated in two ways:

1. implicitly by using them in equations;
2. explicitly using `ProblemDefinition.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\_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.
- *'el'* : The element integral value mode results in an array of a quantity integrated over each element of a region. This mode is supported only by some special terms.
- *'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'* mode
- *'ev\_\** terms support *'eval'*, *'el\_avg'* and *'qp'* modes

Note that the naming prefixes are due to history when the *mode* argument to `ProblemDefinition.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 `ProblemDefinition.evaluate()` function are shown below.

## 5.6 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 : ProblemDefinition instance
        The current ProblemDefinition 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.il.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.il.Omega(solid.D, u)',
                              mode='el_avg')
    out['cauchy_stress'] = Struct(name='output_data',
                                 mode='cell', data=stress,
                                 dofs=None)

    return out

```

The full example is *linear\_elasticity/linear\_elastic\_probes.py*.

- compute diffusion velocity given the pressure:

```

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

    dvel = pb.evaluate('ev_diffusion_velocity.il.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\_npb.py*.

- store values of a non-homogeneous material parameter:

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

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

    return out
```

The full example is [linear\\_elasticity/material\\_nonlinearity.py](#).

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

```
volume = problem.evaluate('d_volume.2.Omega(u)')
```

## 5.7 Probing

Probing applies interpolation to output the solution along specified paths. As mentioned in [Miscellaneous](#), it relies on defining two additional functions, namely the `'gen_probes'` function, that should create the required probes (see [sfepy.fem.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 [Primer](#) for additional explanation.

## 5.8 Available Solvers

This Section describes solvers available in SfePy from user's perspective. There internal/external solvers include linear, nonlinear, eigenvalue, optimization and time stepping solvers.

### 5.8.1 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'`: Newton solver with backtracking line-search - this is the default solver, that is used for almost all examples.
- `'nls.oseen'`: Oseen problem solver tailored for stabilized Navier-Stokes equations (see [navier\\_stokes/stabilized\\_navier\\_stokes.py](#)).
- `'nls.scipy_broyden_like'`: interface to Broyden and Anderson solvers from `scipy.optimize`.
- `'nls.semismooth_newton'`: Semismooth Newton method for contact/friction problems.

### 5.8.2 Linear Solvers

A good linear solver is key to solving efficiently stationary as well as transient PDEs with implicit time-stepping. The following solvers are available:



- 'ls.scipy\_direct': direct solver from SciPy - this is the default solver for all examples. It is strongly recommended to install umfpack and its SciPy wrappers to get good performance.
- 'ls.umfpack': alias to 'ls.scipy\_direct'.
- 'ls.scipy\_iterative': Interface to SciPy iterative solvers.
- 'ls.pyamg': Interface to PyAMG solvers.
- 'ls.petsc': Interface to Krylov subspace solvers of PETSc.
- 'ls.petsc\_parallel': Interface to Krylov subspace solvers of PETSc able to run in parallel by storing the system to disk and running a separate script via *mpiexec*.
- 'ls.schur\_complement': Schur complement problem solver.



# EXAMPLES

## 6.1 SfePy autogenerated gallery examples

### 6.1.1 acoustics examples

`acoustics/acoustics.py`

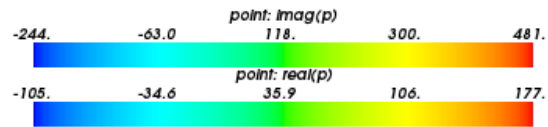
#### Description

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::
    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 \in V.
"""
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' : ('nodes in (x < 0.01)', {}),
    'Gamma_out' : ('nodes in (x > 0.99)', {}),
}

fields = {
    'acoustic_pressure' : ('complex', 1, 'Omega', 1),
}

variables = {
    'p' : ('unknown field', 'acoustic_pressure', 0),
    'q' : ('test field', 'acoustic_pressure', 'p'),
}

ebcs = {
}

integrals = {
    'ivol' : ('v', 2),
    'isurf' : ('s', 2),
}

equations = {
    'Acoustic pressure' :
    """%s * dw_laplace.ivol.Omega( one.one, q, p )
    - %s * dw_volume_dot.ivol.Omega( q, p )
    - %s * dw_surface_dot.isurf.Gamma_out( q, p )
    = %s * dw_surface_integrate.isurf.Gamma_in( q )"""
    % (c*c, w*w, 1j*w*c, 1j*w*c*c*rho*v_n)
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {'i_max' : 1,
                                'eps_a' : 1e-1,
                                'eps_r' : 1.0,
                                'macheps' : 1e-16,
                                # Linear system error < (eps_a * lin_red).
                                '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,
                                'is_plot' : False,
                                # 'nonlinear' or 'linear' (ignore i_max)
                                'problem' : 'nonlinear',
                                } )
}

```

```
}
```

### acoustics/acoustics3d.py

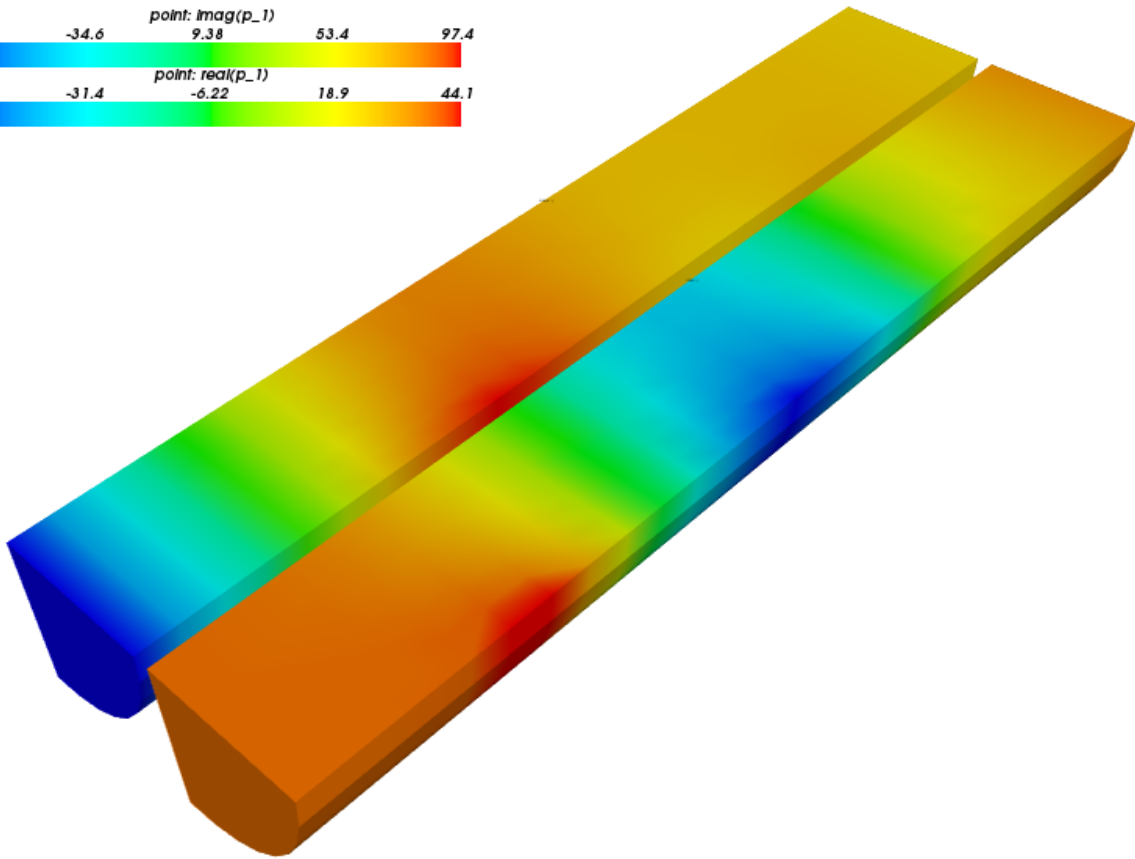
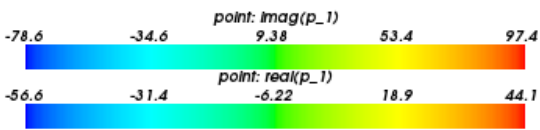
#### Description

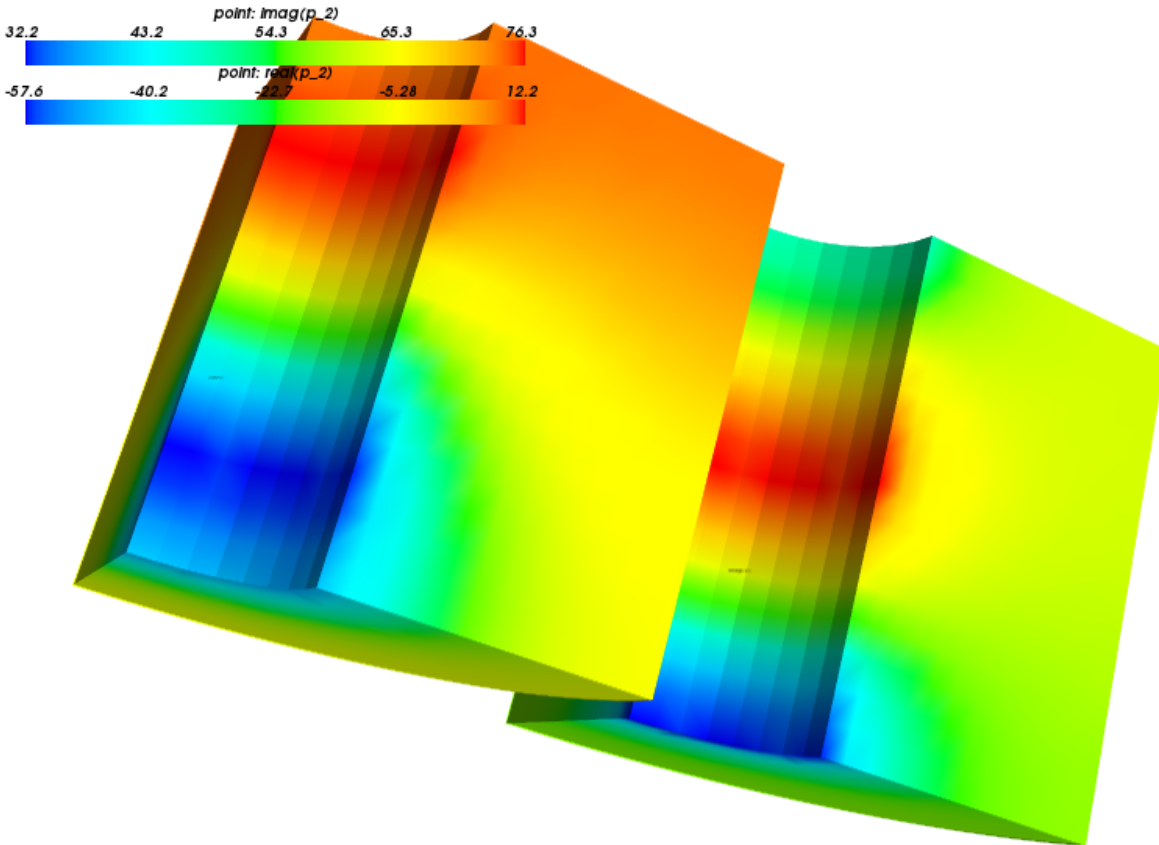
Acoustic pressure distribution in 3D.

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\prime, other in
:math:\Omega_2\prime, connected on the interface region :math:\Gamma_{12}\prime
using traces of variables.

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

.. math::
\int_{\Omega} k^2 q p - \int_{\Omega} \nabla q \cdot \nabla p \backslash
- i w/c \int_{\Gamma_{out}} q p
+ i w \rho/Z \int_{\Gamma_2} q (p_2 - p_1)
+ i w \rho/Z \int_{\Gamma_1} q (p_1 - p_2) \backslash
= i w \rho \int_{\Gamma_{in}} v_n q
\backslash, \quad \forall q \backslash;.
"""

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': ('elements of group 1', {}),
    'Omega_2': ('elements of group 2', {}),
    'Gamma_12': ('r.Omega_1 *n r.Omega_2', {}),
    'Gamma_12_1': ('copy r.Gamma_12', {'forbid' : 'group 2'}),
    'Gamma_12_2': ('copy r.Gamma_12', {'forbid' : 'group 1'}),
    'Gamma_in': ('nodes in (z < 0.001)', {}),
    'Gamma_out': ('nodes in (z > 0.157)', {}),
}

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 = {
    'ivol' : ('v', 2),
    'isurf' : ('s', 2),
}
```



```

equations = {
    'Acoustic pressure' :
        """%s * dw_volume_dot.ivol.Omega_1(q_1, p_1)
        + %s * dw_volume_dot.ivol.Omega_2(q_2, p_2)
        - dw_laplace.ivol.Omega_1(q_1, p_1)
        - dw_laplace.ivol.Omega_2(q_2, p_2)
        - %s * dw_surface_dot.isurf.Gamma_out(q_1, p_1)
        + %s * dw_jump.isurf.Gamma_l2_1(q_1, p_1, tr(p_2))
        + %s * dw_jump.isurf.Gamma_l2_2(q_2, p_2, tr(p_1))
        = %s * dw_surface_integrate.isurf.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-1,
        '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,
        'is_plot' : False,
        'problem' : 'nonlinear',
    } )
}

```

## 6.1.2 biot examples

### biot/biot.py

#### Description

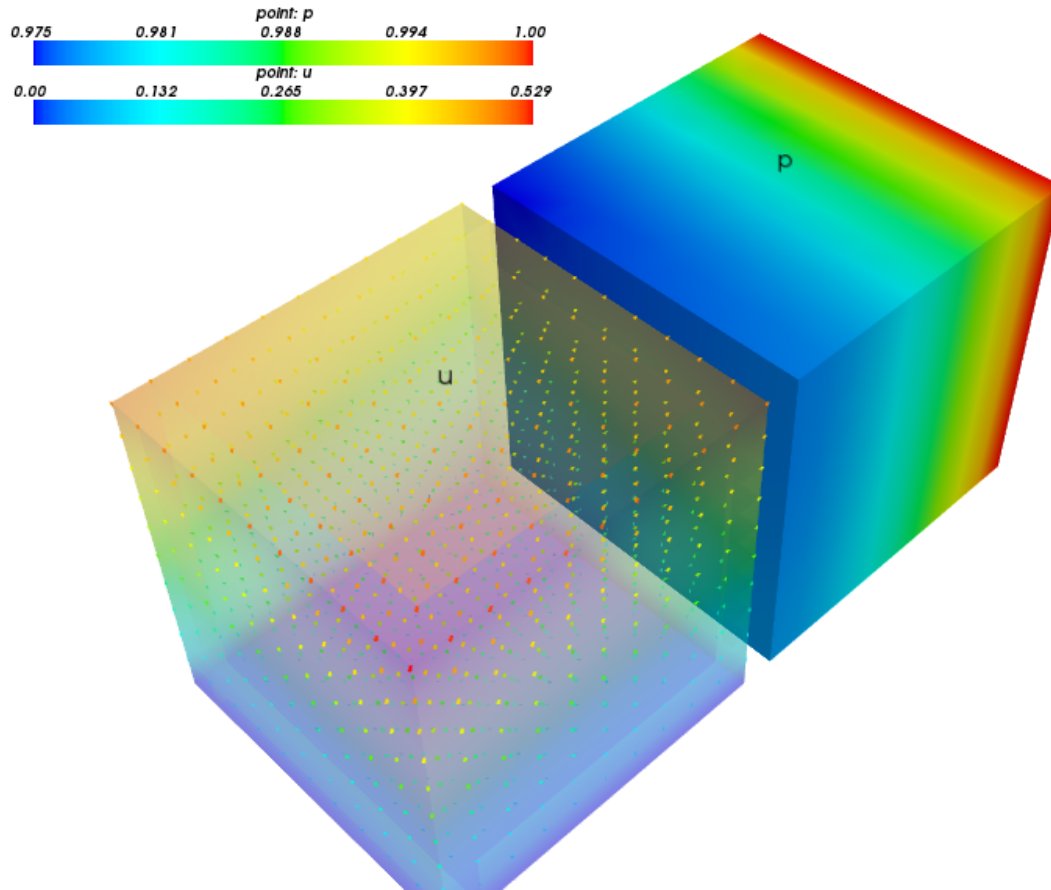
Biot problem - deformable porous medium.

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:\ul{u}', :math:p such that:

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

    \int_{\Omega} q \ \alpha_{ij} \ e_{ij}(\ul{u})
    + \int_{\Omega} K_{ij} \ \nabla_i q \ \nabla_j p
    = 0
    \;, \quad \forall q \;,

where

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

```

        \lambda \ \delta_{ij} \ \delta_{kl}
    \};
"""
import numpy as nm

from sfepy import data_dir

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

regions = {
    'Omega' : ('all', {}),
    'Bottom' : ('nodes in (z < -0.4999999)', {}),
    'Top' : ('nodes in (z > 0.4999999)', {}),
    'Left' : ('nodes in (x < -0.4999999)', {}),
}

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' : {
        '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',
    'kind' : 'v',
    'order' : 1,
}

integral_2 = {
    'name' : 'i2',
    'kind' : 'v',
    'order' : 2,
}

equations = {
    'eq_1' :
        """dw_lin_elastic_iso.i2.Omega( m.lam, m.mu, 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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}
```

## biot/biot\_npbc.py

### Description

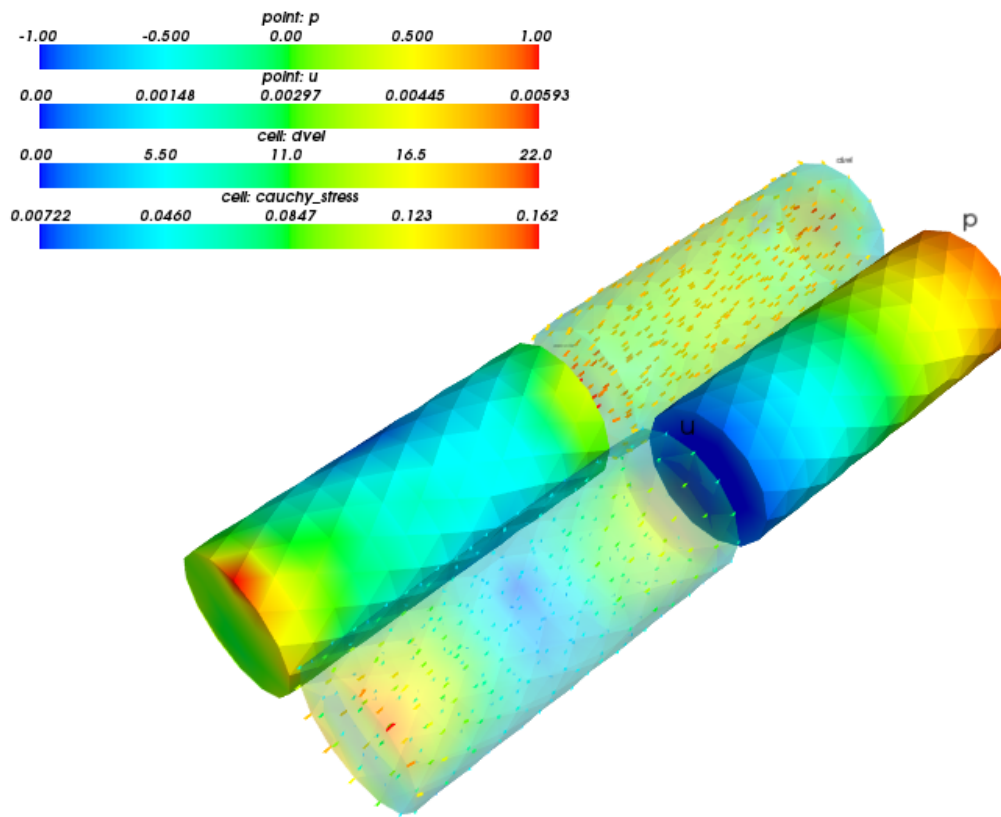
Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region.

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:`\underline{u}`, :math:`p` such that:

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

```

```
\int_{\Omega} q \alpha_{ij} e_{ij}(\ul{u})
+ \int_{\Omega} K_{ij} \nabla_i q \nabla_j p
= 0
\;, \quad \forall q \;,

\ul{u} \cdot \ul{n} = 0 \quad \text{on } \Gamma_{\text{walls}} \;,
```

where

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""
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' : ('nodes of surface -n (r.Outlet +n r.Inlet)',
                       {'can_cells' : True}),
            'Inlet' : ('nodes by cinc_simple0', {'can_cells' : False}),
            'Outlet' : ('nodes by cinc_simple1', {'can_cells' : False}),
            'Rigid' : ('nodes 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.i1.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.i1.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}),
    }

    lcbcs = {
        'rigid' : ('Outlet', {'u.all' : 'rigid'}),
        'no_penetration' : ('Walls', {'u.all' : 'no_penetration'}),
    }

    material_1 = {
        'name' : 'm',
        'function' : 'get_pars',
    }

    integral_1 = {
        'name' : 'il',
        'kind' : 'v',
        'order' : 2,
    }

    equations = {
        'eq_1' :
            """dw_lin_elastic.il.Omega( m.D, v, u )
            - dw_biot.il.Omega( m.alpha, v, p )
            = 0""",
        'eq_2' :
            """dw_biot.il.Omega( m.alpha, u, q )
            + dw_diffusion.il.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()

```

## biot/biot\_npbc\_lagrange.py

### Description

Biot problem - deformable porous medium with the no-penetration boundary condition on a boundary region enforced using Lagrange multipliers.

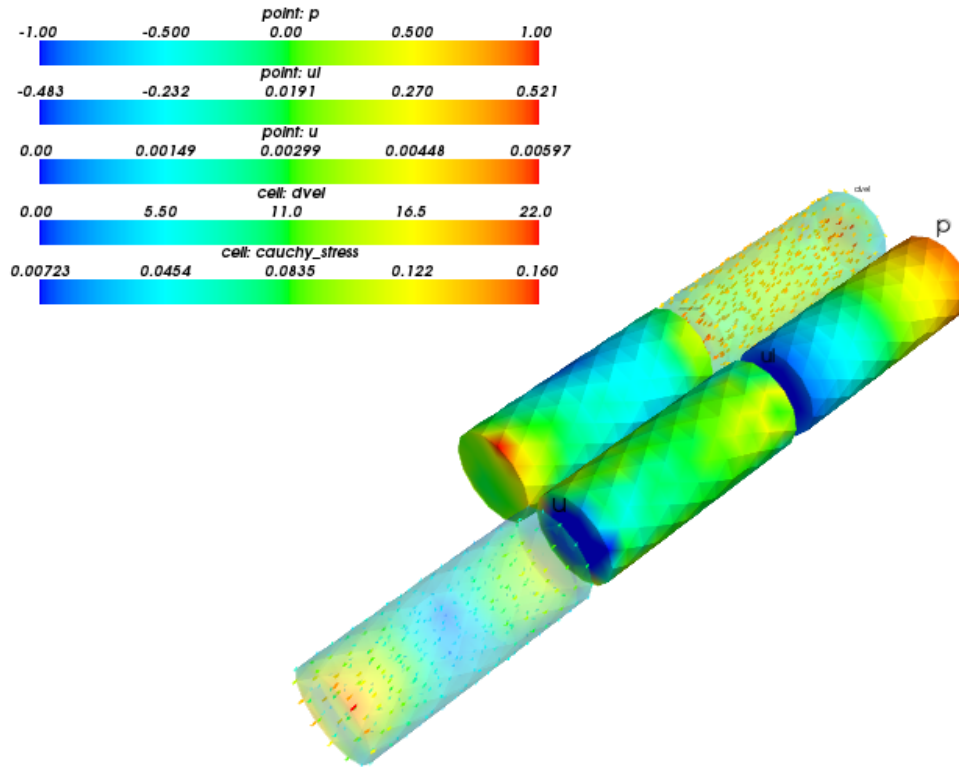
The non-penetration condition is enforced weakly using the Lagrange multiplier  $\lambda$ . There is also a rigid body movement constraint imposed on the  $\Gamma_{outlet}$  region using the linear combination boundary conditions.

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:`u`, :math:`p` and :math:`\lambda` such that:
```

```
.. math::
\int_{\Omega} D_{ijkl} e_{ij}(v) e_{kl}(u)
- \int_{\Omega} p \alpha_{ij} e_{ij}(v)
+ \int_{\Gamma_{walls}} \lambda u_n \cdot v
= 0
\quad \forall v,

\int_{\Omega} q \alpha_{ij} e_{ij}(u)
+ \int_{\Omega} K_{ij} \nabla_i q \nabla_j p
= 0
\quad \forall q,

\int_{\Gamma_{walls}} \hat{\lambda} u_n \cdot u
= 0
```

```

\;, \quad \forall \hat{\lambda} \;,

\ul{u} \cdot \ul{n} = 0 \quad \text{on } \Gamma_{\text{walls}} \;,

where

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

### 6.1.3 diffusion examples

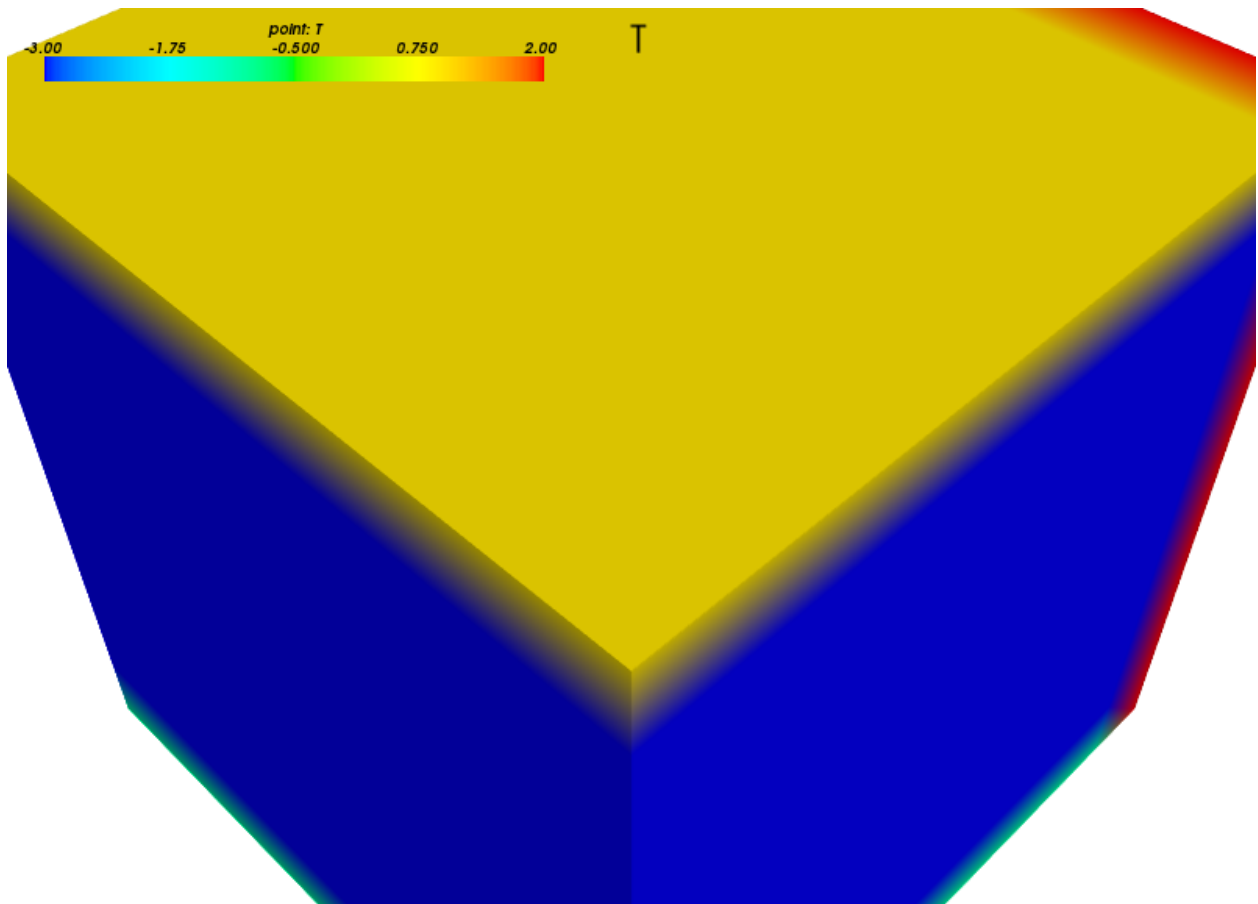
#### diffusion/cube.py

##### 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 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' : 'elements of group 0',
    }
    integral_1 = {
        'name' : 'i1',
        'kind' : 'v',
        'order' : 1,
    }
    solver_0 = {
        'name' : 'ls',
        'kind' : 'ls.scipy_direct',
    }

elif filename_mesh.find('cube_big_tetra.mesh') >= 0:
    region_1000 = {
        'name' : 'Omega',
        'select' : 'elements of group 6',
    }
    integral_1 = {
        'name' : 'i1',
        'kind' : 'v',
        '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' : 'nodes of surface',
}
region_1 = {
    'name' : 'Bottom',
    'select' : 'nodes in (z < -0.4999999)',
}
region_2 = {
    'name' : 'Top',
    'select' : 'nodes in (z > 0.4999999)',
}
region_03 = {
    'name' : 'Left',
    'select' : 'nodes in (x < -0.4999999)',
}

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.il.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,
'is_plot'    : False,
'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}
```

### diffusion/laplace\_time\_ebcs.py

#### Description

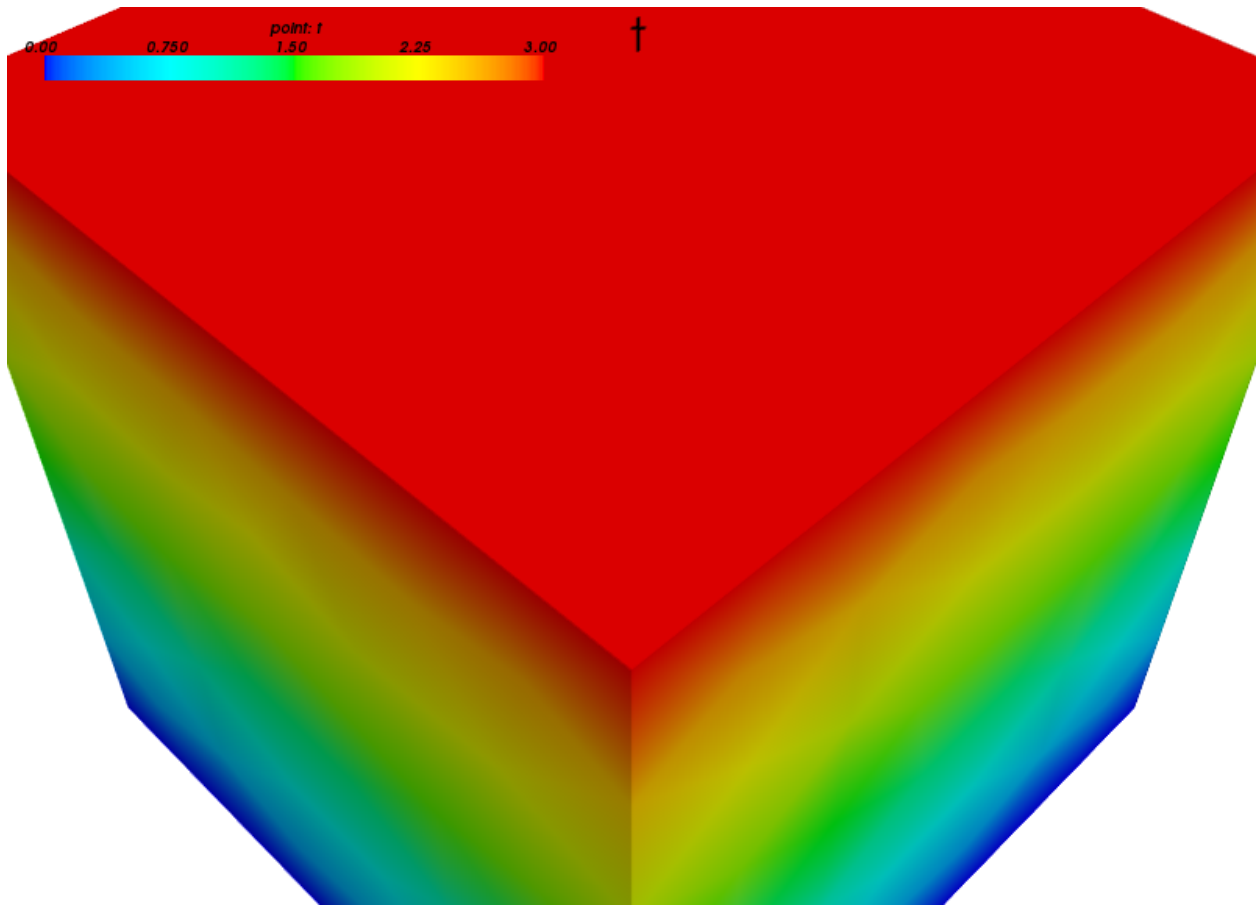
Example explaining how to change Dirichlet boundary conditions depending on time. It is shown on the stationary Laplace equation for temperature, so there is no dynamics, only the conditions change with time.

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 \text{for all } s \in V.
"""
from sfepy import data_dir

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

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

```

```
regions = {
    'Omega' : ('all', {}),
    'Left'  : ('nodes in (x < -0.499)', {}),
    'Right' : ('nodes in (x > 0.499)', {}),
    'Bottom' : ('nodes in (z < -0.499)', {}),
    'Top'    : ('nodes in (z > 0.499)', {}),
}

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,),
}

integrals = {
    'ivol' : ('v', 2),
    'isurf' : ('s', 2),
}

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,
        'problem' : 'nonlinear',
    }),
    'ts' : ('ts.simple', {
        't0' : 0.0,
        't1' : 4.0,
```

```

    'dt'      : None,
    'n_step'  : 5, # has precedence over dt!

    'quasistatic' : True,
  }),
}

```

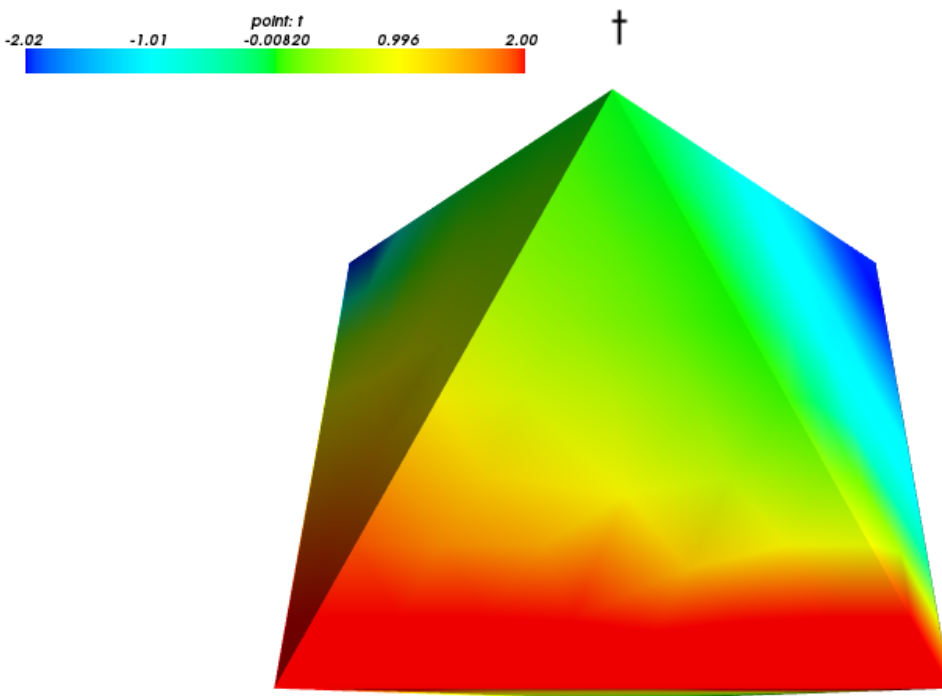
### diffusion/octahedron.py

#### Description

Diffusion (Laplace-like) equation with non-isotropic diffusion coefficient.

Find  $t$  such that:

$$\int_{\Omega} K_{ij} \nabla_i s \nabla_j t = 0, \quad \forall s.$$



source code

```

r"""
Diffusion (Laplace-like) equation with non-isotropic diffusion
coefficient.

Find :math:`t` such that:

```

```
.. math::
    \int_{\Omega} K_{ij} \nabla_i s \nabla_j t
    = 0
    \;, \quad \forall s \;.
"""
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/various_formats/octahedron.node'

material_2 = {
    'name' : 'coef',
    'values' : {'K' : [[1.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 1.0]],
                'val' : 1.0},
}

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

variable_1 = {
    'name' : 't',
    'kind' : 'unknown field',
    'field' : 'temperature',
    'order' : 0,
}

variable_2 = {
    'name' : 's',
    'kind' : 'test field',
    'field' : 'temperature',
    'dual' : 't',
}

region_1000 = {
    'name' : 'Omega',
    'select' : 'all',
}

def get_line(x, y, mode):
    import numpy as nm

    if mode == 0:
        val = nm.where((x + y) >= 3.5)[0]
    elif mode == 1:
        val = nm.where((x + y) <= -3.5)[0]
    print mode, val
    return val

functions = {
    'get_line0' : (lambda coors, domain=None:
                   get_line(coors[:,0], coors[:,1], 0)),
    'get_line1' : (lambda coors, domain=None:
                   get_line(coors[:,0], coors[:,1], 1)),
}
```

```

region_03 = {
    'name' : 'Gamma_Left',
    'select' : 'nodes by get_line0',
}
region_4 = {
    'name' : 'Gamma_Right',
    'select' : 'nodes by get_line1',
}

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' : 'i1',
    'kind' : 'v',
    'order' : 1,
}

equations = {
    'Temperature' : """dw_diffusion.i1.Omega( coef.K, s, t ) = 0"""
#    'Temperature' : """dw_laplace.i1.Omega( coef.val, s, t ) = 0"""
}

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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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

```

```
}
```

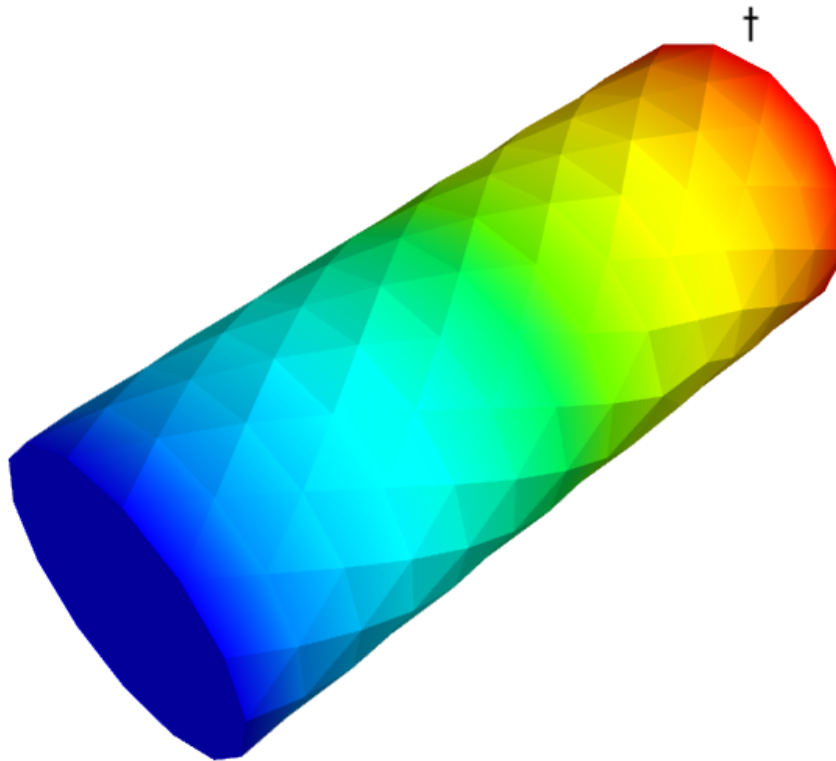
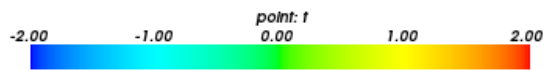
### diffusion/poisson.py

#### Description

Laplace equation with comments.

Find  $t$  such that:

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



source code

```
r"""
Laplace equation with comments.

Find :math:`t` such that:

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

---

```

#! Poisson Equation
#! =====
#$ \centerline{Example input file, \today}

#! Mesh
#! ----
from sfepy import data_dir

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

#! Materials
#! -----
## 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.

material_2 = {
    'name' : 'coef',
    'values' : {'val' : 1.0},
}

#! Regions
#! -----
region_1000 = {
    'name' : 'Omega',
    'select' : 'elements of group 6',
}

region_03 = {
    'name' : 'Gamma_Left',
    'select' : 'nodes in (x < 0.00001)',
}

region_4 = {
    'name' : 'Gamma_Right',
    'select' : 'nodes in (x > 0.099999)',
}

#! Fields
#! -----
## 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 P1 finite elements.

field_1 = {
    'name' : 'temperature',
    'dtype' : 'real',
    'shape' : (1,),
    'region' : 'Omega',
    'approx_order' : 1,
}

#! Variables
#! -----
## 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.

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',
}

#!/ Boundary Conditions
#!/ -----
#!/ Essential (Dirichlet) boundary conditions can be specified as follows:
ebc_1 = {
    'name' : 't1',
    'region' : 'Gamma_Left',
    'dofs' : {'t.0' : 2.0},
}

ebc_2 = {
    'name' : 't2',
    'region' : 'Gamma_Right',
    'dofs' : {'t.0' : -2.0},
}

#!/ Equations
#!/ -----
#!/ The weak formulation of the Poisson equation is:
#!/ \begin{center}
#!/ Find $t$ \in $V$, such that
#!/ $\int_{\Omega} c \nabla t : \nabla s = f, \quad \text{forall } s \in V_0$.
#!/ \end{center}
#!/ The equation below directly corresponds to the discrete version of the
#!/ above, namely:
#!/ \begin{center}
#!/ Find $\bm{t}$ \in $V_h$, such that
#!/ $\bm{s}^T (\int_{\Omega_h} c \bm{G}^T \bm{G}) \bm{t} = 0, \quad \text{forall } \bm{s}$
#!/ \in $V_{h0}$,
#!/ \end{center}
#!/ where $\nabla u \approx \bm{G} \bm{u}$. Below we use $f = 0$ (Laplace
#!/ equation).
#!/ We also define an integral here.
integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'order' : 2,
}
```



```

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

#! Linear solver parameters
#! -----
#! Use umfpack, if available, otherwise superlu.
solver_0 = {
    'name' : 'ls',
    'kind' : 'ls.scipy_direct',
    'method' : 'auto',
}

#! Nonlinear solver parameters
#! -----
#! Even linear problems are solved by a nonlinear solver (KISS rule) - only one
#! iteration is needed and the final rezidual is obtained for free.
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_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

#! Options
#! -----
#! Use them for anything you like... Here we show how to tell which solvers
#! should be used - reference solvers by their names.
options = {
    'nls' : 'newton',
    'ls'  : 'ls',
}

```

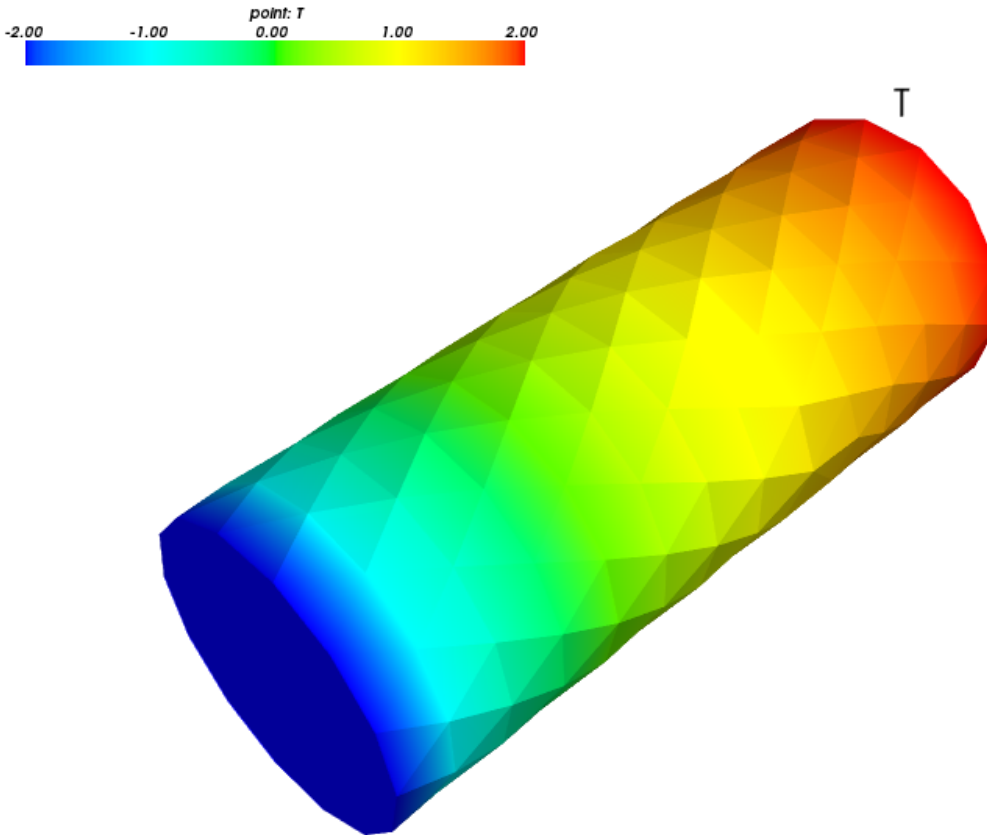
## diffusion/poisson\_field\_dependent\_material.py

### Description

Stationary Laplace equation. 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"""
Stationary Laplace equation.
Find  $T(t)$  for  $t \in [0, t_{\rm final}]$  such that:

.. math::
    \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  $c(T)$ .
"""
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 i1
    # - they are the same as in 'coors' argument.
    T_values = problem.evaluate('ev_volume_integrate.i1.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' : ('nodes in (x < 0.00001)', {}),
    'Gamma_Right' : ('nodes in (x > 0.099999)', {}),
}

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 = {
    'i1' : ('v', 1),
}

equations = {
    'Temperature' : """dw_laplace.i1.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,
        'problem' : 'nonlinear'
    )),
    'ts' : ('ts.simple', {
        't0' : t0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step, # has precedence over dt!
        'quasistatic' : True,
    })),
}

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'ts' : 'ts',
    'save_steps' : -1,
}
```

## diffusion/poisson\_functions.py

### Description

Poisson equation with source term.

Find  $u$  such that:

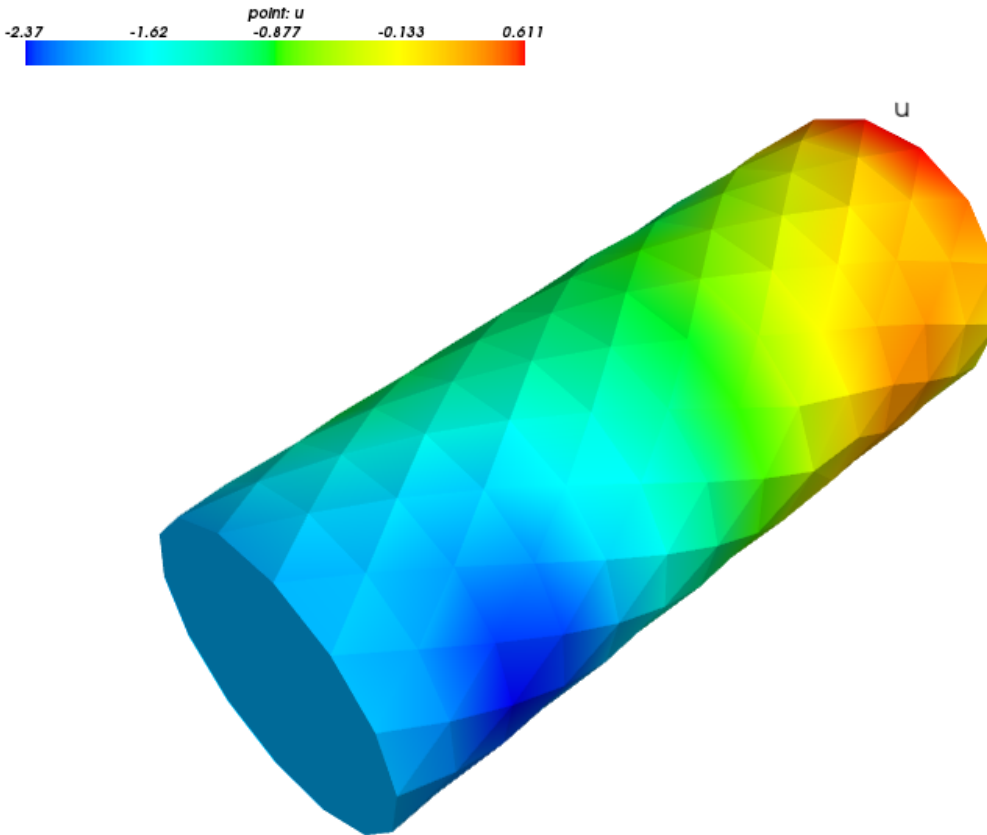
$$\int_{\Omega} c \nabla v \cdot \nabla u = - \int_{\Omega_L} b v = - \int_{\Omega_L} f v p, \quad \forall v,$$

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 :math:'f(x)' in quadrature
   points.
2. How to define a known function that belongs to a given FE space (field) -
   this function, :math:'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 :math:'b(x)' directly, the term ``dw_volume_dot``

```

```
should be replaced by ``dw_volume_integrate``.
"""
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' : ('nodes by get_middle_ball', {}),
    'Gamma_Left' : ('nodes in (x < 0.00001)', {}),
    'Gamma_Right' : ('nodes in (x > 0.099999)', {}),
}

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

variables = {
    'u' : ('unknown field', 'temperature', 0),
    'v' : ('test field', 'temperature', 'u'),
    'p' : ('parameter field', 'temperature',
          {'setter' : 'get_load_variable'}),
}

ebcs = {
    'u1' : ('Gamma_Left', {'u.0' : 'get_ebc'}),
    'u2' : ('Gamma_Right', {'u.0' : -2.0}),
}

integrals = {
    'i1' : ('v', 1),
}

equations = {
    'Laplace equation' :
        """dw_laplace.i1.Omega( m.c, v, u )
        = - dw_volume_dot.i1.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_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_middle_ball' : (get_middle_ball,),
    'get_ebc' : (lambda ts, coor, bc, problem, **kwargs: get_ebc(coor, 5.0),),
}

```

## diffusion/poisson\_parametric\_study.py

### Description

Poisson equation.

This example demonstrates parametric study capabilities of Application classes. In particular (written in the strong form):

$$\begin{aligned} c\Delta t &= f \text{ in } \Omega, \\ t &= 2 \text{ on } \Gamma_1, t = -2 \text{ on } \Gamma_2, f = 1 \text{ in } \Omega_1, f = 0 \text{ otherwise,} \end{aligned}$$

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:

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

source code

```
r"""
Poisson equation.

This example demonstrates parametric study capabilities of Application
classes. In particular (written in the strong form):

.. math::
    c \Delta t = f \text{ in } \Omega,

    t = 2 \text{ on } \Gamma_1,
    t = -2 \text{ on } \Gamma_2,
    f = 1 \text{ in } \Omega_1,
    f = 0 \text{ otherwise,}

where :math:`\Omega` is a square domain, :math:`\Omega_1` in :math:`\Omega` is
a circular domain.

Now let's see what happens if :math:`\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
:math:`\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 \;.
    """
import os
import numpy as nm

from sfepy import data_dir
from sfepy.base.base import output

# Mesh.
filename_mesh = data_dir + '/meshes/2d/special/circles_in_square.vtk'

# Options. The value of 'parametric_hook' is the function that does the
# parametric study.
options = {
    'nls' : 'newton', # Nonlinear solver
    'ls' : 'ls', # Linear solver

    'parametric_hook' : 'vary_omega1_size',
    'output_dir' : 'output/r_omega1',
}

# Domain and subdomains.
default_diameter = 0.25
regions = {
    'Omega' : ('all', {}),
    'Gamma_1' : ('nodes in (x < -0.999)', {}),
    'Gamma_2' : ('nodes in (x > 0.999)', {}),
    'Omega_1' : ('nodes 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' : 'il',
    'kind' : 'v',
    'order' : 2,
}

equations = {
    'Poisson' : """dw_laplace.il.Omega( coef.val, s, t )
                  = dw_volume_lvf.il.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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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 nodes selected! (%d)' % n )

return out

def vary_omegal_size( problem ):
    """Vary size of \Omega. Saves also the regions into options['output_dir'].

    Input:
        problem: ProblemDefinition 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.fem import ProblemDefinition
    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 = ProblemDefinition.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_if_can():
            print region
            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

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.

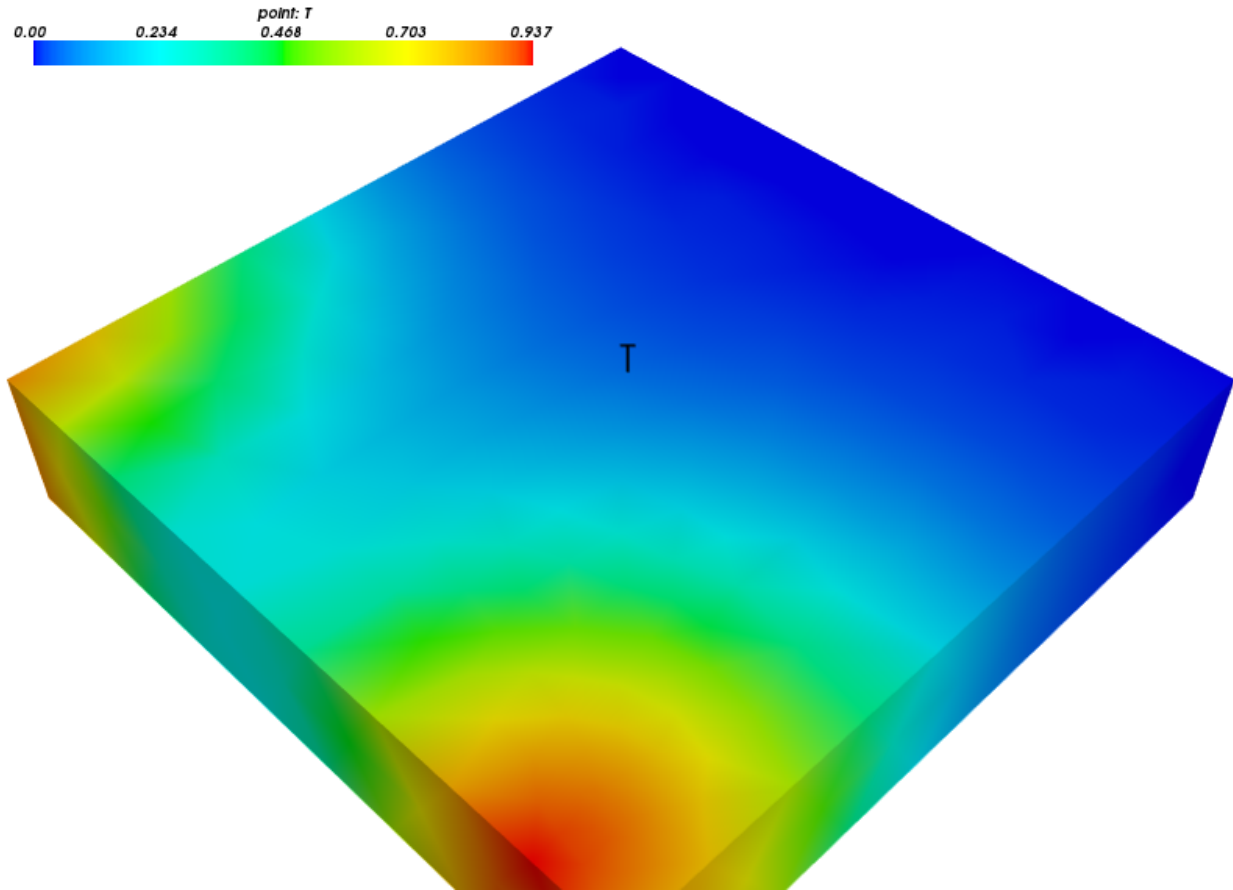
The mesh is suitable for periodic boundary conditions. It consists of a cylinder enclosed by a box in the x and y directions.

The cylinder will act as a power source.

Transient Laplace equation in time interval  $t \in [0, t_{\text{final}}]$  with a localized power source and periodic boundary conditions:

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"""
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.

Transient Laplace equation in time interval
:math:`t \in [0, t_{\rm final}]` with a localized power source and
periodic boundary conditions:

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

.. math::
    \int_{\Omega} c \, s \, \text{pdiff}\{T\}\{t\}
    + \int_{\Omega} \sigma_2 \, \nabla s \cdot \nabla T
    = \int_{\Omega_2} P_3 \, T
    \quad \forall s \in V;

"""

from sfepy import data_dir
import numpy as nm

```

```
import sfepy.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' : ('elements of group 444', {}),
    'fill' : ('elements of group 555', {}),
    'Gamma_Left' : ('nodes in (x < -2.4999)', {}),
    'y+' : ('nodes in (y > 2.4999)', {}),
    'y-' : ('nodes in (y < -2.4999)', {}),
    'z+' : ('nodes in (z > 0.4999)', {}),
}
```

```

    'z-' : ('nodes in (z <-0.4999)', {}),
}

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 = {
    'il' : ('v', 1),
}

equations = {
    'Temperature' :
        """dw_volume_dot.il.cylinder( cylinder.capacity, ds/dt, dT/dt )
        dw_volume_dot.il.fill( fill.capacity, ds/dt, dT/dt )
        dw_laplace.il.cylinder( cylinder.conductivity, s, T )
        dw_laplace.il.fill( fill.conductivity, s, T )
        = dw_volume_integrate.il.cylinder( cylinder.power, s )"""
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton', {
        'i_max' : 1,
        'eps_a' : 1e-10,
        'eps_r' : 1.0,
        'problem' : 'nonlinear'
    }),
    'ts' : ('ts.simple', {
        't0' : t0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step, # has precedence over dt!
        'quasistatic' : False,
    }),
}

options = {

```

```
'nls' : 'newton',  
'ls'  : 'ls',  
'ts'  : 'ts',  
'output_dir' : 'output',  
'save_steps' : -1,  
}
```

### diffusion/poisson\_short\_syntax.py

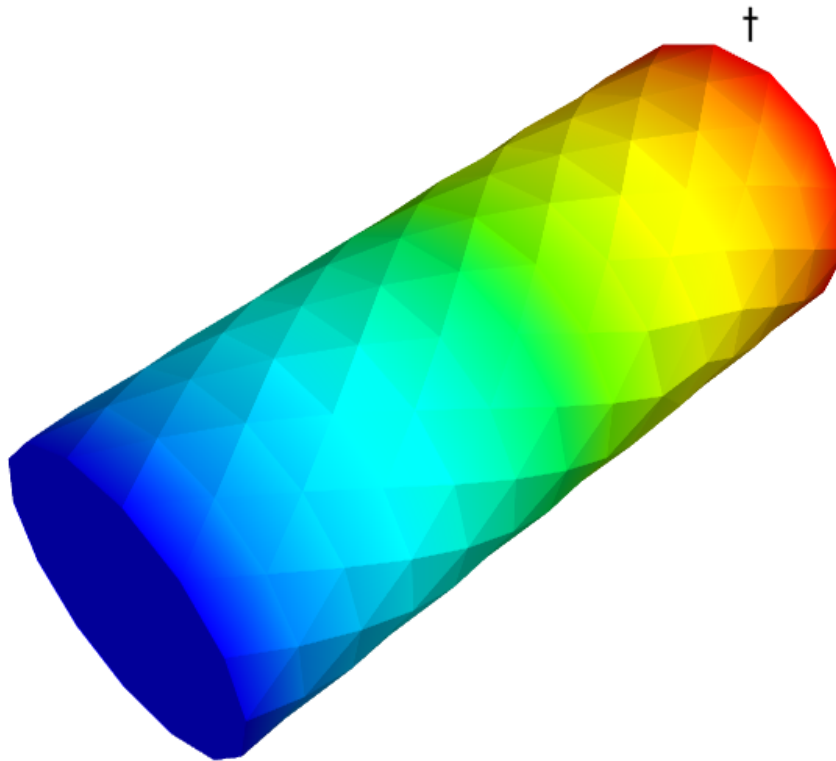
#### Description

Laplace equation.

The same example as poisson.py, but using the short syntax of keywords.

Find  $t$  such that:

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



source code

```
r"""  
Laplace equation.  
  
The same example as poisson.py, but using the short syntax of keywords.
```



Find :math:t such that:

```

.. math::
    \int_{\Omega} c \nabla s \cdot \nabla t
    = 0
    \;, \quad \forall s \;.
    """
from sfepy import data_dir

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

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

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

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 = {
    'i1' : ('v', 2),
}

equations = {
    'Temperature' : """dw_laplace.i1.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.

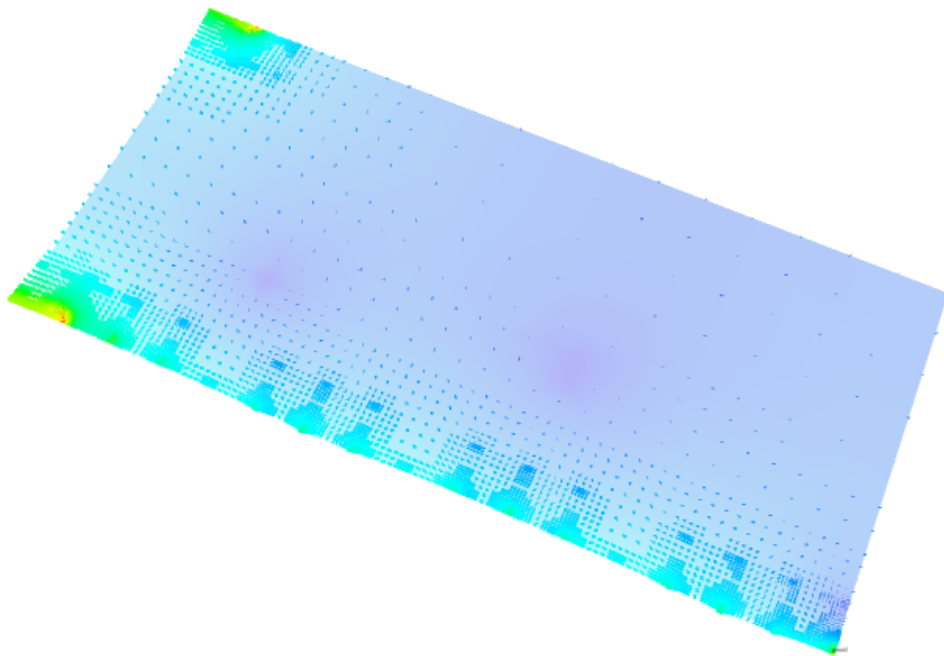
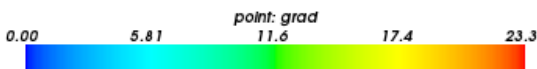
Find  $t$  such that:

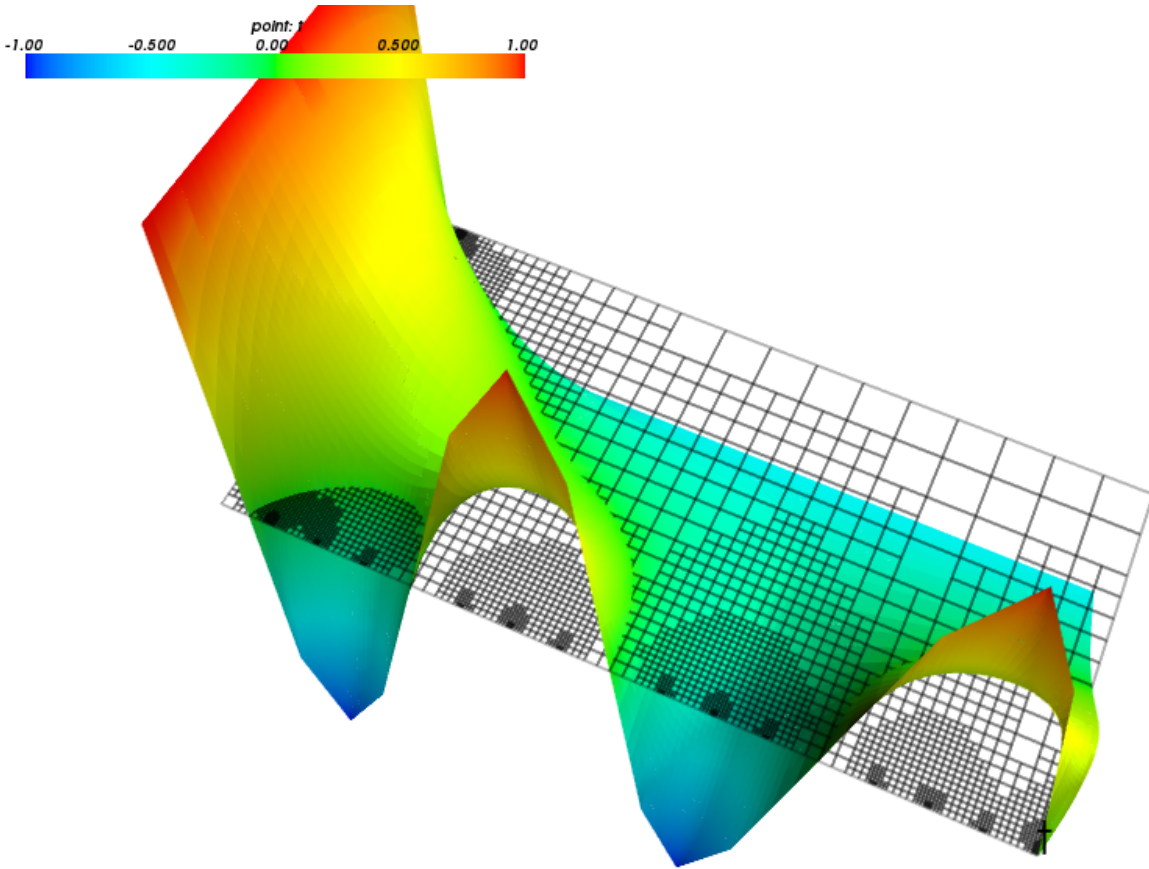
$$\int_{\Omega} c \nabla s \cdot \nabla t = 0, \quad \forall s.$$

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 `sfe.py.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.fem.meshio.UserMeshIO' class is used to refine the original
two-element mesh before the actual solution.
"""
import numpy as nm

```

```
from sfepy import data_dir

from sfepy.base.base import output
from sfepy.fem import Mesh, Domain
from sfepy.fem.meshio import UserMeshIO, MeshIO
from sfepy.homogenization.utils import define_box_regions

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 = Domain(mesh.name, mesh)
        for ii in range(3):
            output('refine %d...' % ii)
            domain = domain.refine()
            output('... %d nodes %d elements'
                  % (domain.shape.n_nod, domain.shape.n_el))

        domain.mesh.name = '2_4_2_refined'

        return domain.mesh

    elif mode == 'write':
        pass

def post_process(out, pb, state, extend=False):
    """
    Calculate gradient of the solution.
    """
    from sfepy.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,
    }),
}

```

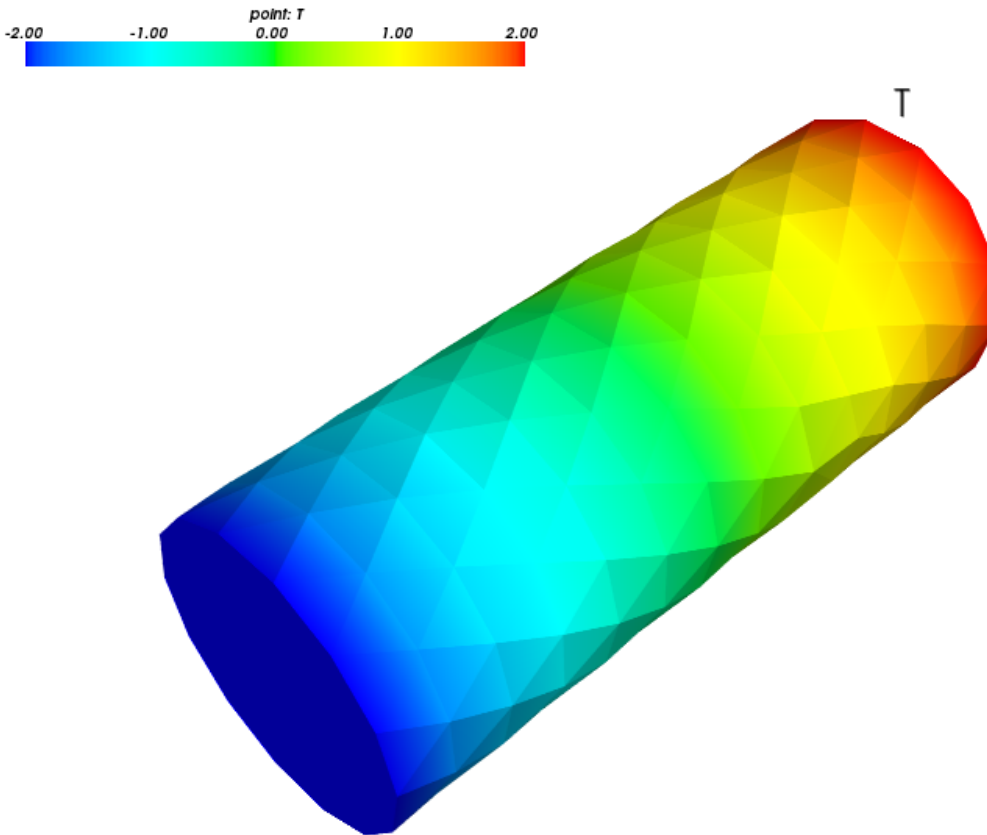
## diffusion/time\_poisson.py

### Description

Transient Laplace equation with a non-constant initial condition 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 \, \text{pdiff}\{T\}\{t\}
    + \int_{\Omega} c \, \nabla s \cdot \nabla T
    = 0
    \;, \quad \text{forall } s \;.
"""
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' : ('nodes in (x < 0.00001)', {}),
    'Gamma_Right' : ('nodes in (x > 0.099999)', {}),
}

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' : 'il',
        'kind' : 'v',
        'order' : 1,
    }

    equations = {
        'Temperature' :
        """dw_volume_dot.il.Omega( s, dT/dt )
           + dw_laplace.il.Omega( coef.val, s, T ) = 0"""
    }

    solver_0 = {
        'name' : 'ls',
        'kind' : 'ls.scipy_direct',

        '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_plot'     : False,
        'problem'     : 'linear', # 'nonlinear' or 'linear' (ignore i_max)
    }

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

        't0'      : t0,
        't1'      : t1,
        'dt'       : None,
        'n_step'  : n_step, # has precedence over dt!
    }

    options = {
        'nls' : 'newton',
        'ls'  : 'ls',
        'ts'  : 'ts',
        'save_steps' : -1,
    }
```



## 6.1.4 homogenization examples

### homogenization/linear\_homogenization.py

#### Description

missing description!

source code

```
# 04.08.2009
#!
#! Homogenization: Linear Elasticity
#! =====
#$ \centerline{Example input file, \today}

#! Homogenization of heterogeneous linear elastic material

import sfepy.fem.periodic as per
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.homogenization.utils import define_box_regions
import sfepy.homogenization.coefs_base as cb
from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.homogenization.recovery import compute_micro_u, compute_stress_strain_u, compute_mac_stress

def recovery_le( pb, corrs, macro ):

    out = {}

    dim = corrs['corrs_le']['u_00'].shape[1]
    mic_u = - compute_micro_u( corrs['corrs_le'], macro['strain'], 'u', dim )

    out['u_mic'] = Struct( name = 'output_data',
                          mode = 'vertex', data = mic_u,
                          var_name = 'u', dofs = None )

    stress_Y, strain_Y = compute_stress_strain_u( pb, 'il', 'Y', 'mat.D', 'u', mic_u )
    stress_Y += compute_mac_stress_part( pb, 'il', 'Y', 'mat.D', 'u', macro['strain'] )

    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

#! Mesh
#! ----
filename_mesh = data_dir + '/meshes/3d/matrix_fiber.mesh'
dim = 3
region_lbn = (0, 0, 0)
region_rtf = (1, 1, 1)
#! Regions
#! -----
#! Regions, edges, ...
```

```
regions = {
    'Y' : ('all', {}),
    'Ym' : ('elements of group 1', {}),
    'Yc' : ('elements of group 2', {}),
}
regions.update( define_box_regions( dim, region_lbn, region_rtf ) )
#! Materials
#! -----
materials = {
    'mat' : ({'D' : {'Ym': stiffness_from_youngpoisson(dim, 7.0e9, 0.4),
                    'Yc': stiffness_from_youngpoisson(dim, 70.0e9, 0.2)}}),
}
#! Fields
#! -----
#! Scalar field for corrector basis functions.
fields = {
    'corrector' : ('real', dim, 'Y', 1),
}
#! Variables
#! -----
#! Unknown and corresponding test variables. Parameter fields
#! used for evaluation of homogenized coefficients.
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
functions = {
    'match_x_plane' : (per.match_x_plane,),
    'match_y_plane' : (per.match_y_plane,),
    'match_z_plane' : (per.match_z_plane,),
}
#! Boundary Conditions
#! -----
#! Fixed nodes.
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
#! -----
#! Define the integral type Volume/Surface and quadrature rule.
integrals = {
    'il' : ('v', 2),
```

```

        'i2' : ('s', 2),
    }
    #! Options
    #! -----
    #! Various problem-specific options.
    options = {
        'coefs' : 'coefs',
        'requirements' : 'requirements',
        'ls' : 'ls', # linear solver to use
        'volume' : { 'variables' : ['u'],
                     'expression' : 'd_volume.il.Y( u )' },
        'output_dir' : 'output',
        'coefs_filename' : 'coefs_le',
        'recovery_hook' : 'recovery_le',
    }
    #! Equations
    #! -----
    #! Equations for corrector functions.
    equation_corrs = {
        'balance_of_forces' :
            """dw_lin_elastic.il.Y(mat.D, v, u ) =
              - dw_lin_elastic.il.Y(mat.D, v, Pi ) """
    }
    #! Expressions for homogenized linear elastic coefficients.
    expr_coefs = """dw_lin_elastic.il.Y(mat.D, Pi1, Pi2 )"""
    #! Coefficients
    #! -----
    #! Definition of homogenized acoustic coefficients.
    def set_elastic(variables, ir, ic, mode, pis, corrs_rs):
        mode2var = {'row' : 'Pi1', 'col' : 'Pi2'}

        val = pis.states[ir, ic]['u'] + corrs_rs.states[ir, ic]['u']

        variables[mode2var[mode]].set_data(val)

    coefs = {
        'D' : {
            'requires' : ['pis', 'corrs_rs'],
            'expression' : expr_coefs,
            'set_variables' : set_elastic,
            'class' : cb.CoeffSymSym,
        },
        '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',
        }
    }

```

```
        'dump_variables' : ['u'],
    },
}
#! Solvers
#! -----
#! Define linear and nonlinear solver.
solvers = {
    'ls' : ('ls.umfpack', {}),
    'newton' : ('nls.newton', {'i_max' : 1,
                              'eps_a' : 1e-4,
                              'problem' : 'nonlinear',
                              })
}
```

## homogenization/linear\_homogenization\_up.py

### Description

missing description!

source code

```
# mixed formulation

# 07.08.2009
#!
#! Homogenization: Linear Elasticity
#! =====
#$ \centerline{Example input file, \today}

#! Homogenization of heterogeneous linear elastic material - mixed formulation
import numpy as nm

import sfepy.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_stres

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, 'il', 'Y', 'mat.D', 'u', mic_u )
```

```

stress_Y += compute_mac_stress_part( pb, 'il', 'Y', 'mat.D', 'u', macro['strain'] )
add_stress_p( stress_Y, pb, 'il', '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

#! Mesh
#! ----
dim = 3
filename_mesh = data_dir + '/meshes/3d/matrix_fiber.mesh'
region_lbn = (0, 0, 0)
region_rtf = (1, 1, 1)
#! Regions
#! -----
#! Regions, edges, ...
regions = {
    'Y' : ('all', {}),
    'Ym' : ('elements of group 1', {}),
    'Yc' : ('elements of group 2', {}),
}
regions.update( define_box_regions( dim, region_lbn, region_rtf ) )
#! Materials
#! -----
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
#! -----
#! Scalar field for corrector basis functions.
fields = {
    'corrector_u' : ('real', dim, 'Y', 1),
    'corrector_p' : ('real', 1, 'Y', 0),
}
#! Variables
#! -----
#! Unknown and corresponding test variables. Parameter fields
#! used for evaluation of homogenized coefficients.
variables = {
    'u'      : ('unknown field', 'corrector_u'),
    'v'      : ('test field', 'corrector_u', 'u'),
    'p'      : ('unknown field', 'corrector_p'),
    'q'      : ('test field', 'corrector_p', 'p'),
    'Pi'     : ('parameter field', 'corrector_u', 'u'),
    'Pi1u'   : ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pi2u'   : ('parameter field', 'corrector_u', '(set-to-None)'),
    'Pi1p'   : ('parameter field', 'corrector_p', '(set-to-None)'),
    'Pi2p'   : ('parameter field', 'corrector_p', '(set-to-None)'),
}

```

```
#!/ Functions
functions = {
    'match_x_plane' : (per.match_x_plane,),
    'match_y_plane' : (per.match_y_plane,),
    'match_z_plane' : (per.match_z_plane,),
}
#!/ Boundary Conditions
#!/ -----
#!/ Fixed nodes.
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
#!/ -----
#!/ Define the integral type Volume/Surface and quadrature rule.
integrals = {
    'i1' : ('v', 2),
    'i2' : ('s', 2),
}
#!/ Options
#!/ -----
#!/ Various problem-specific options.
options = {
    'coefs' : 'coefs',
    'requirements' : 'requirements',
    'ls' : 'ls', # linear solver to use
    'volume' : { #'variables' : ['u'],
                  #'expression' : 'd_volume.il.Y( u )',
                  'value' : get_box_volume( dim, region_lbn, region_rtf ),
                },
    'output_dir' : 'output',
    'coefs_filename' : 'coefs_le_up',
    'recovery_hook' : 'recovery_le',
}
#!/ Equations
#!/ -----
#!/ Equations for corrector functions.
equation_corrs = {
    'balance_of_forces' :
        """ dw_lin_elastic.il.Y( mat.D, v, u )
            - dw_stokes.il.Y( v, p ) =
            - dw_lin_elastic.il.Y( mat.D, v, Pi ) """,
    'pressure constraint' :
        """- dw_stokes.il.Y( u, q )
            - dw_volume_dot.il.Y( mat.gamma, q, p ) =
            + dw_stokes.il.Y( Pi, q ) """,
}
```

```

}
#! Expressions for homogenized linear elastic coefficients.
expr_coefs = {
    'Q1' : ""dw_lin_elastic.il.Y( mat.D, Pilu, Pi2u )"",
    'Q2' : ""dw_volume_dot.il.Y( mat.gamma, Pilp, Pi2p )"",
}
#! Coefficients
#! -----
#! Definition of homogenized acoustic coefficients.
def set_elastic_u(variables, ir, ic, mode, pis, corrs_rs):
    mode2var = {'row' : 'Pilu', 'col' : 'Pi2u'}

    val = pis.states[ir, ic]['u'] + corrs_rs.states[ir, ic]['u']

    variables[mode2var[mode]].set_data(val)

coefs = {
    'elastic_u' : {
        'requires' : ['pis', 'corrs_rs'],
        'expression' : expr_coefs['Q1'],
        'set_variables' : set_elastic_u,
        'class' : cb.CoeffSymSym,
    },
    'elastic_p' : {
        'requires' : ['corrs_rs'],
        'expression' : expr_coefs['Q2'],
        'set_variables' : [('Pilp', 'corrs_rs', 'p'), ('Pi2p', 'corrs_rs', 'p')],
        'class' : cb.CoeffSymSym,
    },
    'D' : {
        'requires' : ['c.elastic_u', 'c.elastic_p'],
        'class' : cb.CoeffSum,
    },
    'filenames' : {},
}

requirements = {
    'pis' : {
        'variables' : ['u'],
        'class' : cb.ShapeDimDim,
    },
    'corrs_rs' : {
        'requires' : ['pis'],
        'ebcs' : ['fixed_u'],
        'epbcs' : all_periodic,
        'equations' : equation_corrs,
        'set_variables' : [('Pi', 'pis', 'u')],
        'class' : cb.CorrDimDim,
        'save_name' : 'corrs_le',
        'dump_variables' : ['u', 'p'],
        'is_linear' : True,
    },
}
#! Solvers
#! -----
#! Define linear and nonlinear solver.
solvers = {
    'ls' : ('ls.umfpack', {}),

```

```
    'newton' : ('nls.newton', {'i_max' : 1,
                              'eps_a' : 1e-4,
                              'problem' : 'nonlinear', })
}
```

## 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] [http://seth.asc.tuwien.ac.at/proc12/full\\_paper/Contribution183.pdf](http://seth.asc.tuwien.ac.at/proc12/full_paper/Contribution183.pdf)

source code

```
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] http://seth.asc.tuwien.ac.at/proc12/full\_paper/Contribution183.pdf
"""
```

```
from sfepy.fem.periodic import match_x_plane, match_y_plane
import sfepy.homogenization.coefs_base as cb
import numpy as nm
from sfepy import data_dir

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_coors = pb.domain.mesh.coors
    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 macro.iterkeys():
        if 'press' in k:
            channels[k[-1]] = 1

    channels = channels.keys()
```



```

varnames = ['pM']
for ch in channels:
    nodes_Y[ch] = pb.domain.regions['Y' + ch].all_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 \
        corrs['corrs_%s_pi%s' % (pb_def['name'], ch)].iteritems():
        kk = int( key[-1] )
        press_mic += val * press_mac_grad[kk,0]

    for key in corrs.iterkeys():
        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',
        dat =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' : ('v', 2),
        'iS' : ('s', 2),
    }

    functions = {
        'match_x_plane': (match_x_plane,),
        'match_y_plane': (match_y_plane,),
    }

    # forbid groups?
    forbid_grps = ''
    aux = []
    for ch, val in pb_def['channels'].iteritems():
        forbid_grps += ' %d' % val['mat_el_grp']
        aux.append( 'r.bYM' + ch )

    forbid_channels = {'forbid': 'group %s' % forbid_grps}
    forbid_matrix = {'forbid': 'group %d' % pb_def['matrix_mat_el_grp']}

    # basic regions
    regions = {
        'Y': ('all', {}),
        'YM': ('elements of group %d' % pb_def['matrix_mat_el_grp'], {}),
        # periodic boundaries
        'Pl': ('nodes in (x < 0.001)', {}),
        'Pr': ('nodes in (x > 0.999)', {}),
        'PLYM': ('r.Pl *n r.YM', {}),
        'PrYM': ('r.Pr *n r.YM', {}),
        'bYmp': ('r.bYp *n r.YM', forbid_channels),
        'bYmM': ('r.bYm *n r.YM', forbid_channels),
        'bYMpm': ('r.bYmp +n r.bYmM', forbid_channels),
    }

    # matrix/channel boundaries
    regions.update({
        'bYMchs': (' +n '.join( aux ), forbid_channels),
        'YmMchs': ('r.YM -n r.bYMchs', {}),
    })

    # boundary conditions Gamma+/-
    ebcs = {
        'gamma_pm_bYMchs': ('bYMchs', {'pM.0': 0.0}),
        'gamma_pm_YmMchs': ('YmMchs', {'pM.0': 1.0}),
    }

    # periodic boundary conditions - matrix, X-direction
    epbcs = {'periodic_xYM': ([ 'PLYM', 'PrYM' ], {'pM.0': 'pM.0'}, 'match_x_plane')}
    lcbcs = {}

    all_periodicYM = [ 'periodic_%sYM' % ii for ii in ['x', 'y'][:pb_def['dim']-1] ]
    all_periodicY = {}

    if pb_def['dim'] == 2:
        regions.update( {
            'bYm': ('nodes in (y < 0.001)', {}),
            'bYp': ('nodes in (y > 0.999)', {}),
        } )
    if pb_def['dim'] == 3:

```

```
regions.update( {
    'Pn': ('nodes in (y < 0.001)', {}),
    'Pf': ('nodes in (y > 0.999)', {}),
    'PnYM': ('r.Pn *n r.YM', {}),
    'PfYM': ('r.Pf *n r.YM', {}),
    'bYm': ('nodes in (z < 0.001)', {}),
    'bYp': ('nodes in (z > 0.999)', {}),
} )

# 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 pb_def['channels'].iteritems():

    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: ('elements of group %d' % val['mat_el_grp'], {}),
        'bYM' + ch: ('r.YM *n r.Y' + ch, forbid_channels),
        'PLY' + ch: ('r.Pl *n r.Y' + ch, {}),
        'PrY' + ch: ('r.Pr *n r.Y' + ch, {}),
    } )

    if 'fix_nd_grp' in val:
        regions.update({
            'fixedY' + ch: ('nodes of group %d' % val['fix_nd_grp'][0], {}),
        })

    ebcs_eta[ch] = []
    for ch2, val2 in pb_def['channels'].iteritems():
        aux = 'eta%s_bYM%s' % (ch, ch2)
        if ch2 == ch:
            ebcs.update( {aux: ('bYM' + ch2, {'pM.0': 1.0})} )
        else:
            ebcs.update( {aux: ('bYM' + ch2, {'pM.0': 0.0})} )

    ebcs_eta[ch].append(aux)

# boundary conditions
# periodic boundary conditions - channels, X-direction
epbcs.update({
    'periodic_xY' + ch: ([ 'PLY' + ch, 'PrY' + ch],
                        {'p%s.0' % ch: 'p%s.0' % ch},
                        'match_x_plane'),
})

if pb_def['dim'] == 3:
    regions.update({
        'PnY' + ch: ('r.Pn *n r.Y' + ch, {}),
        'PfY' + ch: ('r.Pf *n r.Y' + ch, {}),
    })
```

```

    })
    # 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 : ('nodes of group %d +n r.fixedY%s' % (aux, ch),
                               forbid_matrix),
            } )
        else:
            regions.update( {
                'bY%s' % io : ('nodes of group %d' % aux,
                               forbid_matrix),
            } )

        aux_bY.append('r.bY%s' % io)
        reg_io[ch].append('bY%s' % io)

    regions.update({
        'bY' + ch : ( ' +n '.join(aux_bY), forbid_matrix),
    })

    # 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 -n r.bY%s' % (ch, io), forbid_matrix),
        })

        # boundary conditions
        aux = 'fix_p%s_bY%s' % (ch, ion)
        ebcs.update({
            aux: ('bY%s' % ion, {'p%s.0' % ch: 0.0}),
        })

    lcbcs.update({
        'imv' + ch: ('Y' + ch, {'ls%s.all' % ch: 'integral_mean_value'})
    })

#####
# materials, fields, variables, integrals

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 pb_def['channels'].iteritems():

    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 reg_io.iterkeys():
    for ireg in reg_io[ch]:
        options['volumes'].update({
            ireg: {
                'variables': ['p' + ch],
                'expression': "d_surface.iS.%s(p%s)" % (ireg, ch),
            }
        })

#####
# equations, correctors, coefficients

coefs = {
    'vol_bYMpm': {
        'regions': ['bYmp', 'bYMm'],
        'expression': 'd_surface.iS.%s(pM)',
        'class': cb.VolumeFractions,
    },
    'filenames': {},
}

# requirements for perfusion homog. coefficients
requirements = {
    'corrs_one_YM': {
        'variable': ['pM'],
        'ebcs': ['gamma_pm_YMchs', 'gamma_pm_bYMchs'],
        'epbcs': [],
        'save_name': 'corrs_one_YM',
        'class': cb.CorrSetBCS,
        'dump_variables': ['pM'],
    },
}

```

```
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),
            'dump_variables': ['pM'],
        },
    })

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.CoeffOne,
        },
    })

def get_channel(keys, bn):
    for ii in keys:
        if bn in ii:
            return ii[(ii.rfind(bn) + len(bn)):]

    return None

def set_corrpis(variables, ir, ic, mode, **kwargs):
    ch = get_channel(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, **kwargs):
    ch = get_channel(kwargs.keys(), 'pis_')
    io = get_channel(kwargs.keys(), 'corrs_gamma_')

    pis = kwargs['pis_' + ch]
    corrs_gamma = kwargs['corrs_gamma_' + io]

    pi = pis.states[ir]['p' + ch]
    val = corrs_gamma.state['p' + ch]
    variables['corr1_' + ch].set_data(pi)
    variables['corr2_' + ch].set_data(val)
```



```

def set_corr_cc(variables, ir, **kwargs):
    ch = get_channel(kwargs.keys(), 'pis_')
    pis = kwargs['pis_' + ch]
    corrs_pi = kwargs['corrs_pi' + ch]

    pi = pis.states[ir]['p' + ch]
    pi = pi - nm.mean(pi)
    val = pi + corrs_pi.states[ir]['p' + ch]
    variables['corr1_' + ch].set_data(val)

for ch, val in pb_def['channels'].iteritems():

    coefs.update({
        'G' + ch: { # test+
            'requires': ['corrs_one' + ch, 'corrs_eta' + ch],
            'set_variables': [('corr1_M', 'corrs_one' + ch, 'pM'),
                             ('corr2_M', 'corrs_eta' + ch, 'pM')],
            'expression': 'dw_diffusion.iV.YM(mat2M.k, corr1_M, corr2_M)',
            'class': cb.CoeffOne,
        },
        'K' + ch: { # test+
            'requires': ['pis_' + ch, 'corrs_pi' + ch],
            'set_variables': set_corrpis,
            'expression': 'dw_diffusion.iV.Y%s(mat2%s.k, corr1_%s, corr2_%s)'\
                          % ((ch,) * 4),
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDimDim,
        },
    })

    requirements.update({
        'pis_' + ch: {
            'variables': ['p' + ch],
            'class': cb.ShapeDim,
        },
        'corrs_one' + ch: {
            'variable': ['pM'],
            'ebcs': ebcs_eta[ch],
            'epbcs': [],
            'save_name': 'corrs_%s_one%s' % (pb_def['name'], ch),
            'dump_variables': ['pM'],
            'class': cb.CorrSetBCS,
        },
        'corrs_eta' + ch: {
            'ebcs': ebcs_eta[ch],
            'epbcs': all_periodicYM,
            'equations': {
                'eq_eta': 'dw_diffusion.iV.YM(mat2M.k, qM, pM) = 0',
            },
            'class': cb.CorrOne,
            'save_name': 'corrs_%s_eta%s' % (pb_def['name'], ch),
            'dump_variables': ['pM'],
        },
        'corrs_pi' + ch: {
            'requires': ['pis_' + ch],
            'set_variables': [('Pi_' + ch, 'pis_' + ch, 'p' + ch)],
            'ebcs': [],
            'epbcs': all_periodicY[ch],
        },
    })

```

```
'lcbcs': ['imv' + ch],
'equations': {
    'eq_pi': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                + dw_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),
'dump_variables': ['p' + 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.CoeffOne,
        },
        'F' + ipm + ch: { # test+
            'requires': ['corrs_one' + ch, 'corrs_gamma_' + ipm],
            'set_variables': [('corr1_M', 'corrs_one' + ch, 'pM'),
                             ('corr2_M', 'corrs_gamma_' + ipm, 'pM')],
            'expression': """dw_diffusion.iV.YM(mat2M.k, corr1_M, corr2_M)
                            - %e * ev_surface_integrate.iS.bYM%s(corr1_M)"""\
                            % (1.0/param_h, ipm),
            'class': cb.CoeffOne,
        },
    })

for i_io in range(len(val['io_nd_grp'])):
    io = '%s_%d' % (ch, i_io + 1)

    coefs.update({
        'S' + io: { # [Rohan1] (4.28), test+
            'requires': ['corrs_gamma_' + io, 'pis_' + ch],
            'set_variables': set_corr_S,
            'expression': 'dw_diffusion.iV.Y%s(mat2%s.k, corr1_%s, corr2_%s)' \
                            % ((ch,) * 4),
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDim,
        },
        'P' + io: { # test+
            'requires': ['pis_' + ch, 'corrs_pi' + ch],
            'set_variables': set_corr_cc,
            'expression': 'ev_surface_integrate.iS.bY%s(corr1_%s)' \
                            % (io, ch),
            'set_volume': 'bYp',
            'dim': pb_def['dim'] - 1,
            'class': cb.CoeffDim,
        },
        'S_test' + io: {
            'requires': ['corrs_pi' + ch],
```

```

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

requirements.update({
    'corrs_gamma_' + io: {
        'requires': [],
        'variables': ['p' + ch, 'q' + ch],
        'ebcs': [],
        'epbcs': all_periodicY[ch],
        'lcbcs': ['imv' + ch],
        'equations': {
            'eq_gamma': """dw_diffusion.iV.Y%s(mat2%s.k, q%s, p%s)
                + dw_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),
        'dump_variables': ['p' + ch],
    },
})

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.CoeffOne,
        },
    })

#####
# solver, fe

solvers = {
    'ls': ('ls.umfpack', {}),
    'newton': ('nls.newton', {'i_max': 1,
        'problem': 'nonlinear', })
}

fe = {
    'chunk_size': 1000
}

```

# [Rohan1] Rohan E.: HOMOGENIZATION OF THE DARCY FLOW IN A DOUBLE-POROUS LAYER

## 6.1.5 large\_deformation examples

### large\_deformation/active\_fibres.py

#### Description

Nearly incompressible hyperelastic material model with active fibres.

Large deformation is described using the total Lagrangian formulation. Models of this kind can be used in biomechanics to model biological tissues, e.g. muscles.

Find  $\underline{u}$  such that:

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

where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$J$	$\det(F)$
$\underline{\underline{C}}$	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

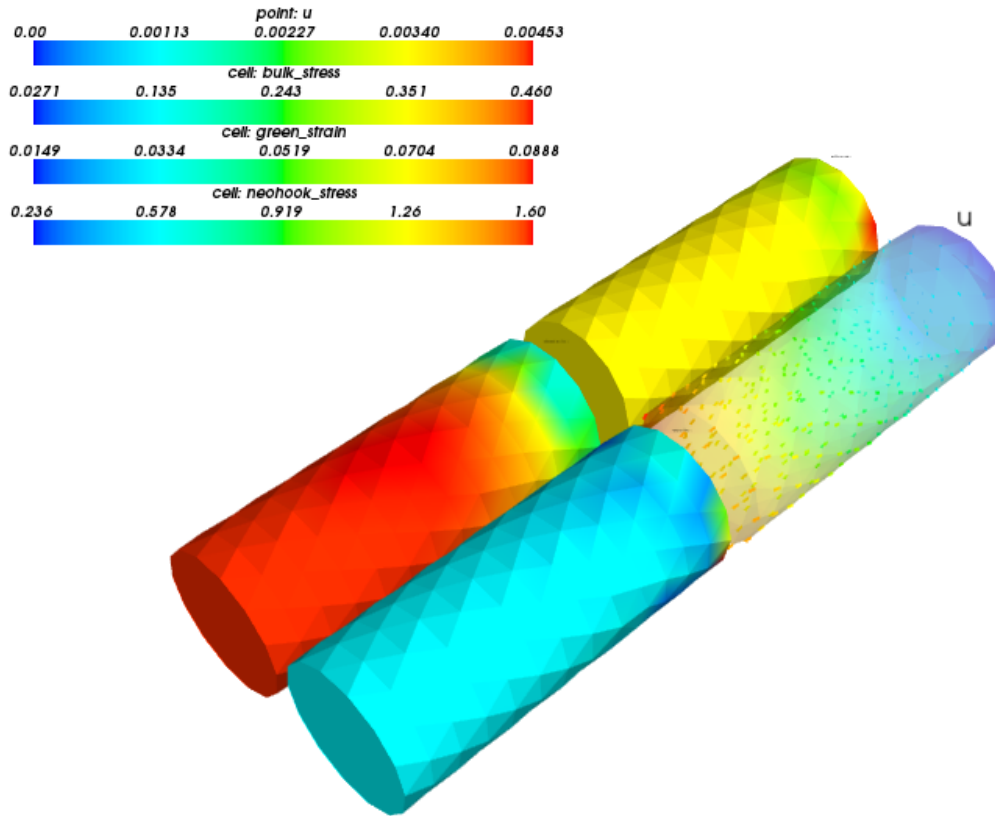
The effective stress  $\underline{\underline{S}}^{\text{eff}}(\underline{u})$  incorporates also the effects of the active fibres in two preferential directions:

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

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

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

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



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  $\mathbf{u}$  such that:

.. math::
    \int_{\Omega} \left( \frac{1}{J} \frac{\partial \mathcal{E}}{\partial \mathbf{v}} \right) \cdot \mathbf{v} \, dV = 0
    \quad \forall \mathbf{v} \in \mathbf{V},

```

where

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

   * -  $\mathbf{F}$ 
     - deformation gradient  $F_{ij} = \frac{\partial x_i}{\partial X_j}$ 
   * -  $J$ 
     -  $\det(F)$ 
```

```
* - :math:\ul{C}'
  - right Cauchy-Green deformation tensor :math:C = F^T F'
* - :math:\ul{E}(\ul{u})'
  - Green strain tensor :math:E_{ij} = \frac{1}{2}(\pdiff{u_i}{x_j} +
    \pdiff{u_j}{x_i} + \pdiff{u_m}{x_i}\pdiff{u_m}{x_j})'
* - :math:\ul{S}\eff(\ul{u})'
  - effective second Piola-Kirchhoff stress tensor
```

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

```
.. math::
    \ul{S}\eff(\ul{u}) = \mu J^{-\frac{2}{3}}(\ul{I}
    - \frac{1}{3}\tr(\ul{C}) \ul{C}^{-1})
    + \sum_{k=1}^2 \tau^k \ul{\omega}^k
    \;.
```

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

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

```
.. math::
    \tau = A f_{\rm max} \exp\{\left\{-\left(\frac{\epsilon - \varepsilon_{\rm opt}}{s}\right)^2\right\}\} \mbox{ , } \epsilon = \ul{E} : \ul{\omega}
    \;.
```

```
"""
```

```
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_steps' : -1,
    '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 should be defined.
    mode : 'qp' or 'special'
        Call mode.
    which : int
        Fibre system id.
    vf : float
        Fibre system volume fraction.
    """
    if mode != 'qp': return

    fmax = 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' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

##
# Dirichlet BC.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0}),
}

##
# Balance of forces.
integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'order' : 1,
}

equations = {
    'balance'
    : """dw_tl_he_neohook.i1.Omega( solid.mu, v, u )
      + dw_tl_bulk_penalty.i1.Omega( solid.K, v, u )
      + dw_tl_fib_a.i1.Omega( f1.fmax, f1.eps_opt, f1.s, f1.fdir, f1.act,
                             v, u )
      + dw_tl_fib_a.i1.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_neohook.i1.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=strain, dofs=None)

    stress = ev('dw_tl_he_neohook.i1.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_tl_bulk_penalty.i1.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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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

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

```

## 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:

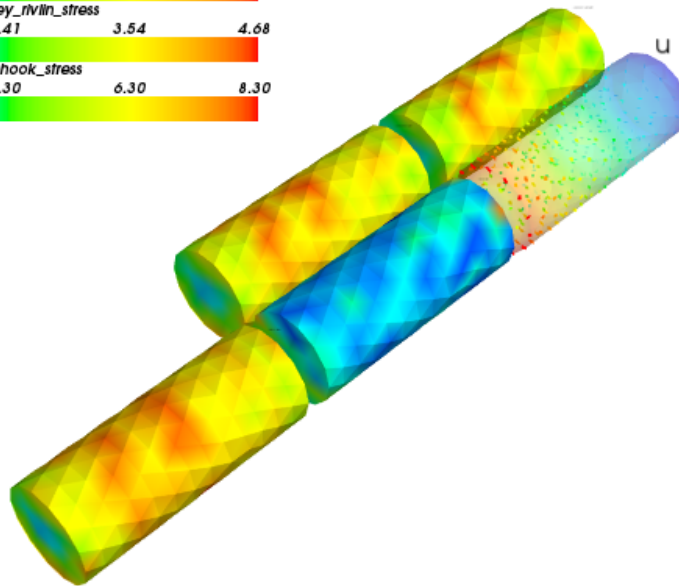
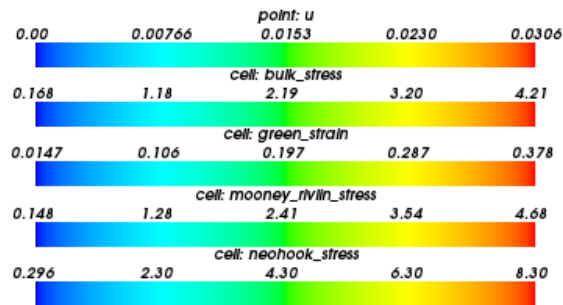
$$\int_{\Omega^{(0)}} \left( \underline{S}^{\text{eff}}(\underline{u}) + K(J - 1) J \underline{C}^{-1} \right) : \delta \underline{E}(\underline{v}) \, dV = 0, \quad \forall \underline{v},$$

where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$J$	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = F^T F$
$\underline{\underline{E}}(u)$	Green strain tensor $E_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} + \frac{\partial u_m}{\partial x_i} \frac{\partial u_m}{\partial x_j})$
$\underline{\underline{S}}^{\text{eff}}(u)$	effective second Piola-Kirchhoff stress tensor

The effective stress  $\underline{\underline{S}}^{\text{eff}}(u)$  is given by:

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



source code

```
# -*- coding: utf-8 -*-
r"""
Nearly incompressible Mooney-Rivlin hyperelastic material model.

Large deformation is described using the total Lagrangian formulation.
Models of this kind can be used to model e.g. rubber or some biological
materials.

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

.. math::
\intl{\Omega \ul{u}} \left( \ul{S} \eff(\ul{u})
```

```

+ K(J-1)\; J \ull{C}^{-1} \right) : \delta \ull{E}(\ul{v}) \difd{V}
= 0
\;, \quad \forall \ul{v} \;,

```

where

```

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

   * - :math:\'\ull{F}\'
     - deformation gradient :math:\'F_{ij} = \pdiff{x_i}{X_j}\'
   * - :math:\'J\'
     - :math:\'\det(F)\'
   * - :math:\'\ull{C}\'
     - right Cauchy-Green deformation tensor :math:\'C = F^T F\'
   * - :math:\'\ull{E}(\ul{u})\'
     - Green strain tensor :math:\'E_{ij} = \frac{1}{2}(\pdiff{u_i}{x_j} + \pdiff{u_j}{x_i} + \pdiff{u_m}{x_i}\pdiff{u_m}{x_j})\'
   * - :math:\'\ull{S}\eff(\ul{u})\'
     - effective second Piola-Kirchhoff stress tensor

```

The effective stress :math:\'\ull{S}\eff(\ul{u})\' is given by:

```

.. math::
   \ull{S}\eff(\ul{u}) = \mu J^{-\frac{2}{3}}(\ull{I}
   - \frac{1}{3}\tr(\ull{C}) \ull{C}^{-1})
   + \kappa J^{-\frac{4}{3}} (\tr(\ull{C})\ull{I} - \ull{C}
   - \frac{2}{6}((\tr(\ull{C}))^2 - \tr(\ull{C}^2))\ull{C}^{-1})
   \;.
"""

```

```
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_steps' : -1,
    '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' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

##
# 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.T.flat

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

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

    ev = problem.evaluate
    strain = ev('dw_tl_he_neohook.il.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                                mode='cell', data=strain, dofs=None)

    stress = ev('dw_tl_he_neohook.il.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_tl_he_mooney_rivlin.il.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.i1.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' : 'i1',
    'kind' : 'v',
    'order' : 1,
}
equations = {
    'balance' : """dw_tl_he_neohook.i1.Omega( solid.mu, v, u )
                  + dw_tl_he_mooney_rivlin.i1.Omega( solid.kappa, v, u )
                  + dw_tl_bulk_penalty.i1.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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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

    't0'    : 0,
    't1'    : 1,
    'dt'    : None,

```

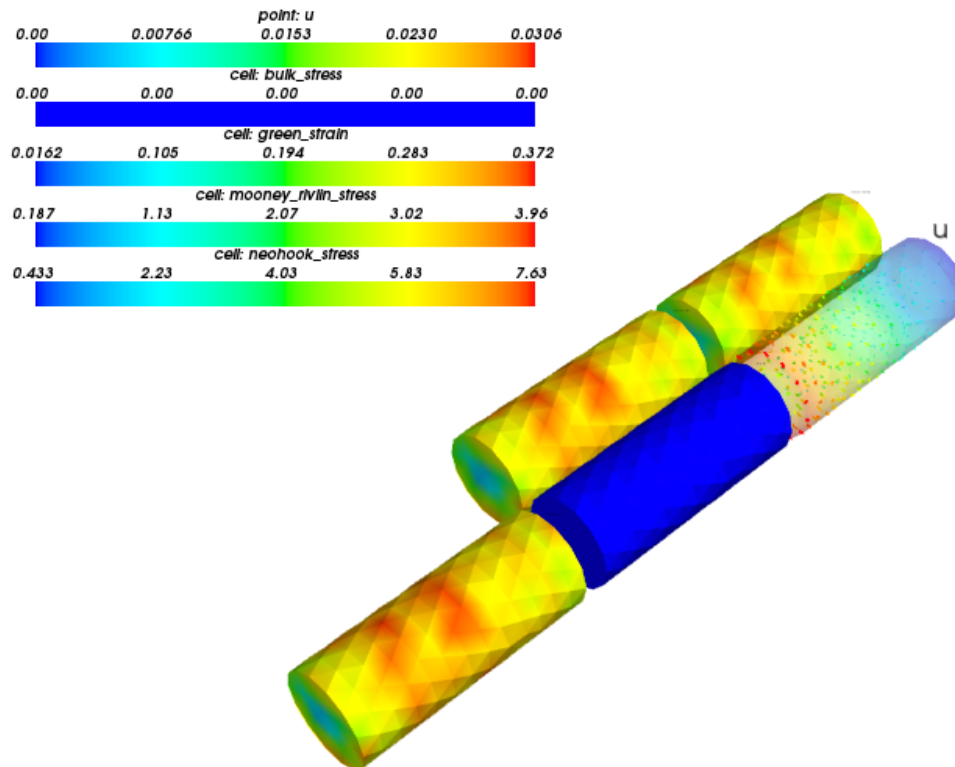
```
'n_step' : 11, # has precedence over dt!  
}
```

## 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.  
"""  
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'  : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

##
# 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.T.flat

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

```

```
}

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

    ev = problem.evaluate
    strain = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                               mode='cell', data=strain, dofs=None)

    stress = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='stress')
    out['neohook_stress'] = Struct(name='output_data',
                                  mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_he_mooney_rivlin.3.Omega( solid.kappa, v, u )',
               mode='el_avg', term_mode='stress')
    out['mooney_rivlin_stress'] = Struct(name='output_data',
                                         mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_bulk_penalty.3.Omega( solid.K, v, u )',
               mode='el_avg', term_mode='stress')
    out['bulk_stress'] = Struct(name='output_data',
                               mode='cell', data=stress, dofs=None)

    return out

equations = {
    'balance': """dw_ul_he_neohook.3.Omega( solid.mu, v, u )
                  + dw_ul_he_mooney_rivlin.3.Omega(solid.kappa, v, u)
                  + dw_ul_bulk_penalty.3.Omega( solid.K, v, u )
                  = 0""",
}

##
# Solvers etc.
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 25,
        'eps_a': 1e-8,
        'eps_r': 1.0,
        'macheps': 1e-16,
        'lin_red': 1e-2, # Linear system error < (eps_a * lin_red).
        'ls_red': 0.1,
        'ls_red_warp': 0.001,
        'ls_on': 1.1,
        'ls_min': 1e-5,
        'check': 0,
        'delta': 1e-6,
        'is_plot': False,
        'problem': 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max),
    }),
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'dt': None,
    })
}
```



```

    'n_step': 11, # has precedence over dt!
  }),
}

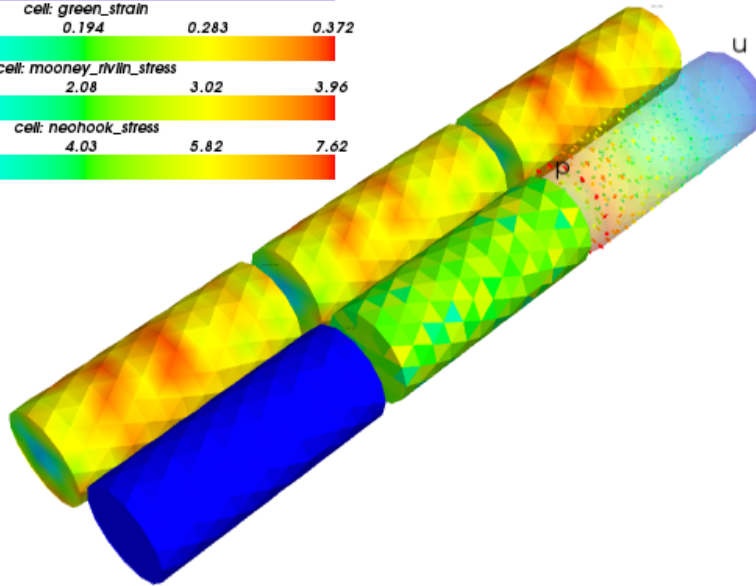
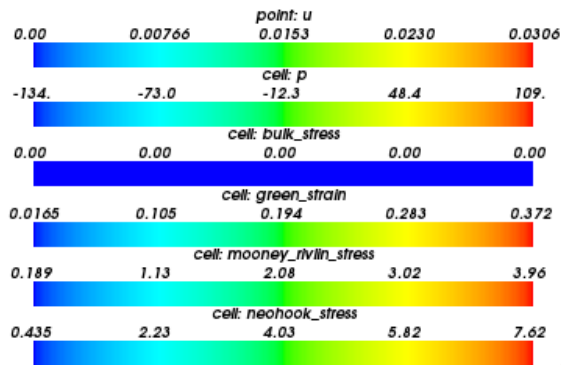
```

## 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.
"""
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' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

##
# 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.T.flat

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

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

    ev = problem.evaluate
    strain = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='strain')
    out['green_strain'] = Struct(name='output_data',
                               mode='cell', data=strain, dofs=None)

    stress = ev('dw_ul_he_neohook.3.Omega( solid.mu, v, u )',
               mode='el_avg', term_mode='stress')
    out['neohook_stress'] = Struct(name='output_data',
                                  mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_he_mooney_rivlin.3.Omega( solid.kappa, v, u )',
               mode='el_avg', term_mode='stress')
    out['mooney_rivlin_stress'] = Struct(name='output_data',
                                         mode='cell', data=stress, dofs=None)

    stress = ev('dw_ul_bulk_pressure.3.Omega( v, u, p )',
               mode='el_avg', term_mode='stress')
    out['bulk_stress'] = Struct(name='output_data',
                               mode='cell', data=stress, dofs=None)

    return out

equations = {
    'balance': """dw_ul_he_neohook.3.Omega( solid.mu, v, u )
                  + dw_ul_he_mooney_rivlin.3.Omega(solid.kappa, v, u)
                  + dw_ul_bulk_pressure.3.Omega( v, u, p ) = 0""",
    'volume': """dw_ul_volume.3.Omega( q, u )
                 + dw_ul_compressible.3.Omega( solid.iK, q, p, u ) = 0"""
}

##
# Solvers etc.
solvers = {
    'ls': ('ls.scipy_direct', {}),
    'newton': ('nls.newton', {
        'i_max': 25,
        'eps_a': 1e-8,
        'eps_r': 1.0,
        'macheps': 1e-16,
        'lin_red': 1e-2, # Linear system error < (eps_a * lin_red).
        'ls_red': 0.1,
        'ls_red_warp': 0.001,
        'ls_on': 1.1,
        'ls_min': 1e-5,
        'check': 0,
        'delta': 1e-6,
        'is_plot': False,
        'problem': 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max),
    })
}

```

```

    }),
    'ts': ('ts.simple', {
        't0': 0,
        't1': 1,
        'dt': None,
        'n_step': 11, # has precedence over dt!
    }),
}

```

## large\_deformation/perfusion\_tl.py

### Description

Porous nearly incompressible hyperelastic material with fluid perfusion.

Large deformation is described using the total Lagrangian formulation. Models of this kind can be used in biomechanics to model biological tissues, e.g. muscles.

Find  $\underline{u}$  such that:

(equilibrium equation with boundary tractions)

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

(mass balance equation (perfusion))

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

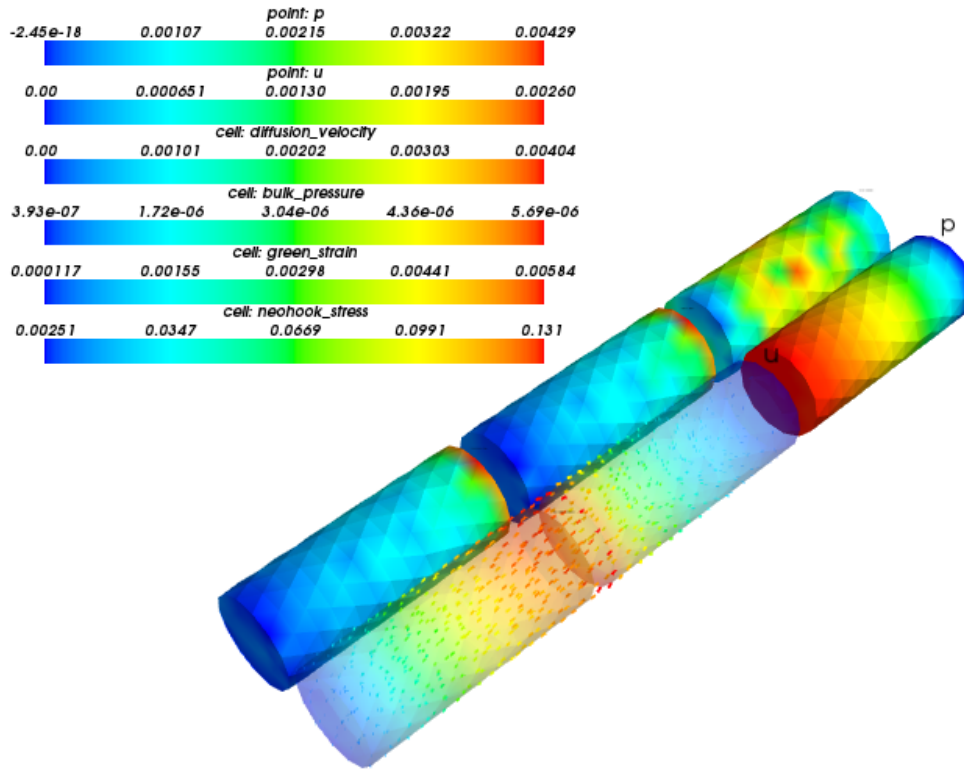
where

$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$J$	$\det(F)$
$\underline{\underline{C}}$	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 (neo-Hookean) stress  $\underline{\underline{S}}^{\text{eff}}(\underline{u})$  is given by:

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

The linearized deformation-dependent permeability is defined as  $\underline{\underline{K}}(\underline{u}) = J \underline{\underline{F}}^{-1} \underline{\underline{k}} f(J) \underline{\underline{F}}^{-T}$ , where  $\underline{u}$  relates to the previous time step  $(n-1)$  and  $f(J) = \max \left( 0, \left( 1 + \frac{(J-1)}{N_f} \right) \right)^2$  expresses the dependence on volume compression/expansion.



source code

```
# -*- coding: utf-8 -*-
r"""
Porous nearly incompressible hyperelastic material with fluid perfusion.

Large deformation is described using the total Lagrangian formulation.
Models of this kind can be used in biomechanics to model biological
tissues, e.g. muscles.

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

(equilibrium equation with boundary tractions)

.. math::
\intl{\Omega\ul{suz}}{} \left( \ul{S}\eff - p J \ul{C}^{-1} \right) : \delta \ul{E}(\ul{v}) \difd{V}
+ \intl{\Gamma_0\ul{suz}}{} \ul{\nu} \cdot \ul{F}^{-1} \cdot \ul{\sigma}
\cdot \ul{v} J \difd{S}
= 0
\;, \quad \forall \ul{v} \;,

(mass balance equation (perfusion))

.. math::
\intl{\Omega\ul{suz}}{} q J(\ul{u})
+ \intl{\Omega\ul{suz}}{} \ul{K}(\ul{u}\sum) : \pdiff{q}{X} \pdiff{p}{X}
```

```
= \intl{\Omega\suz}{} q J(\ul{u}\sunm)
\;, \quad \forall q \; ,
```

where

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

   * - :math:\ul{F} '
     - deformation gradient :math:F_{ij} = \pdiff{x_i}{X_j} '
   * - :math:J '
     - :math:\det(F) '
   * - :math:\ul{C} '
     - right Cauchy-Green deformation tensor :math:C = F^T F '
   * - :math:\ul{E}(\ul{u}) '
     - Green strain tensor :math:E_{ij} = \frac{1}{2}(\pdiff{u_i}{x_j} +
       \pdiff{u_j}{x_i} + \pdiff{u_m}{x_i}\pdiff{u_m}{x_j}) '
   * - :math:\ul{S}\eff(\ul{u}) '
     - effective second Piola-Kirchhoff stress tensor
```

The effective (neo-Hookean) stress :math:\ul{S}\eff(\ul{u}) ' is given by:

```
.. math::
   \ul{S}\eff(\ul{u}) = \mu J^{-\frac{2}{3}}(\ul{I}
   - \frac{1}{3}\tr(\ul{C}) \ul{C}^{-1})
   \; .
```

The linearized deformation-dependent permeability is defined as  
:math:\ul{K}(\ul{u}) = J \ul{F}^{-1} \ul{k} f(J) \ul{F}^{-T} ,  
where :math:\ul{u} ' relates to the previous time step :math:(n-1) ' and  
:math:f(J) = \max\left(0, \left(1 + \frac{(J - 1)}{N\_f}\right)\right)^2 ' expresses the dependence on volume  
compression/expansion.

"""

```
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_steps' : -1,
```

```
    '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'  : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

##
# Dirichlet BC.
ebcs = {
    'l' : ('Left', {'u.all' : 0.0, 'p.0' : 'get_pressure'}),
}

##
# Balance of forces.
integrals = {
    'i1' : ('v', 1),
    'i2' : ('s', 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.i1.Omega( ps.mu, v, u )',

```

```
        mode='el_avg', term_mode='strain')
out['green_strain'] = Struct(name='output_data',
                           mode='cell', data=val, dofs=None)

val = problem.evaluate('dw_tl_he_neohook.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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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

    't0'    : t0,
    't1'    : t1,
    'dt'    : None,
    'n_step': n_step, # has precedence over dt!
}

##
```



```

# Functions.
def get_traction(ts, coors, mode=None):
    """
    Pressure traction.

    Parameters
    -----
    ts : TimeStepper
        Time stepping info.
    coors : array_like
        The physical domain coordinates where the parameters should be defined.
    mode : 'qp' or 'special'
        Call mode.
    """
    if mode != 'qp': return

    tt = ts.nt * 2.0 * nm.pi

    dim = coors.shape[1]
    val = 0.05 * nm.sin(tt) * nm.eye(dim, dtype=nm.float64)
    val[1,0] = val[0,1] = 0.5 * val[0,0]

    shape = (coors.shape[0], 1, 1)
    out = {
        'pressure' : nm.tile(val, shape),
    }

    return out

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,),
}

```

## 6.1.6 linear\_elasticity examples

### 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:

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

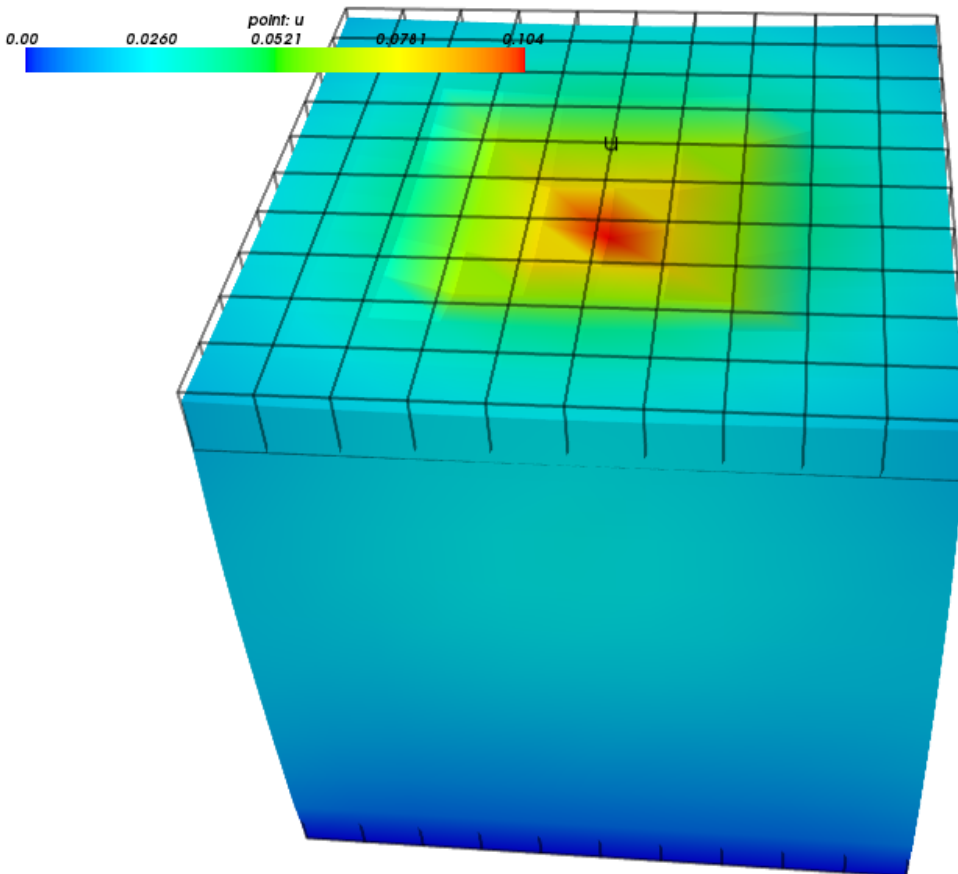
where

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

## Notes

Even though the material is linear elastic and small deformations are used, the problem is highly nonlinear due to contacts with the planes.

Checking the tangent matrix by finite differences by setting ‘check’ in ‘nls’ solver configuration to nonzero is rather tricky - the active contact points must not change during the test. This can be ensured by a sufficient initial penetration and large enough contact boundary polygons (hard!), or by tweaking the `dw_contact_plane` term to mask points only by undeformed coordinates.



source code

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

Four contact planes bounded by polygons (triangles in this case) form a very rigid pyramid shape simulating an indenter.

Find  $\mathbf{u}$  such that:

```
.. math::
    \int_{\Omega} D_{ijkl} \mathbf{e}_{ij}(\mathbf{v}) \mathbf{e}_{kl}(\mathbf{u})
    + \sum_{i=1}^4 \int_{\Gamma_i} \mathbf{v} \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 sfepy import data_dir
```

```
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' : ({
        'lam' : 5.769,
        'mu' : 3.846,
    },),
    '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' : ('nodes in (z < -0.499)', {}),
    'Top' : ('nodes in (z > 0.499)', {}),
}

ebcs = {
    'fixed' : ('Bottom', {'u.all' : 0.0}),
}

equations = {
    'elasticity' :
    """dw_lin_elastic_iso.2.Omega(solid.lam, solid.mu, 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,
        'problem' : 'nonlinear',
        'check' : 0,
        'delta' : 1e-6,
    }),
}

def main():
    import os

    import numpy as nm
    import matplotlib.pyplot as plt

    from sfepy.fem import MeshIO
    import sfepy.linalg as la
    from sfepy.mechanics.contact_planes 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/its2D\_1.py

### Description

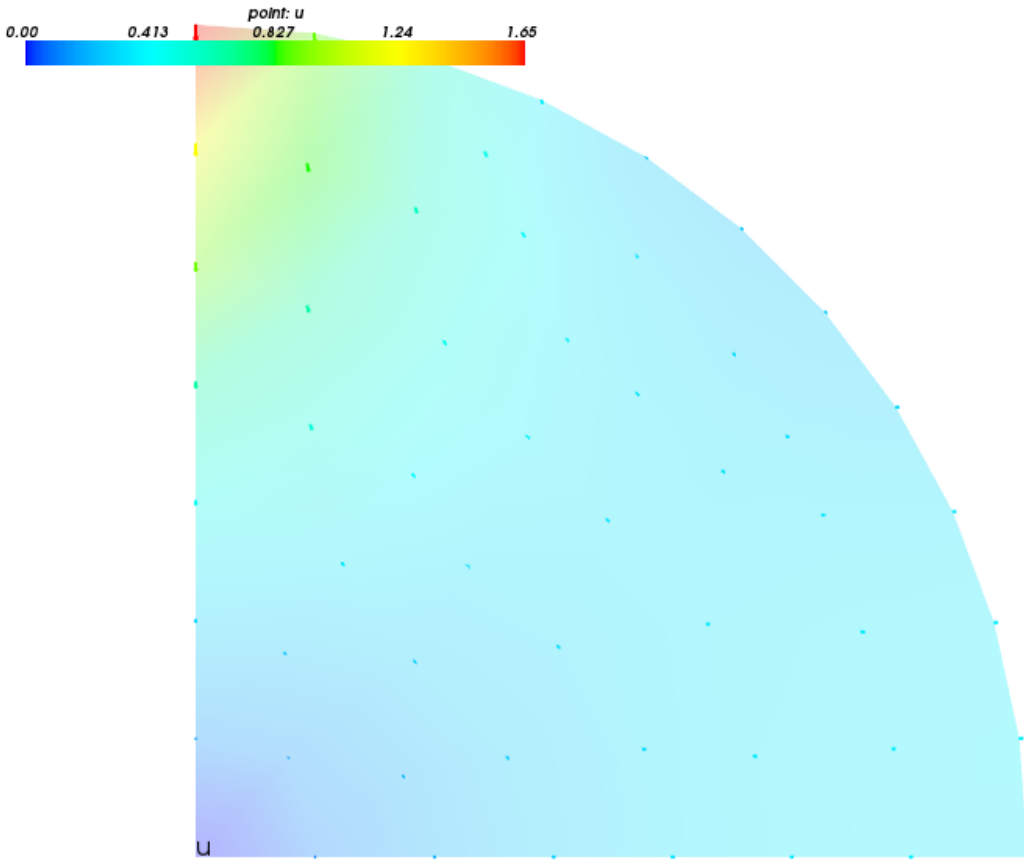
Diametrically point loaded 2-D disk. See *Primer*.

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{v}) \ 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}
    \;.
"""
from sfepy.mechanics.matcoefs import lame_from_youngpoisson
from sfepy.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' : ('nodes in (x < 0.001)', {}),
    'Bottom' : ('nodes in (y < 0.001)', {}),
    'Top' : ('node 2', {}),
}

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

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

equations = {
    'balance_of_forces' :
        """dw_lin_elastic_iso.2.Omega(Asphalt.lam, Asphalt.mu, 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,  
    'problem' : 'nonlinear'  
}),  
}
```

## linear\_elasticity/its2D\_2.py

### Description

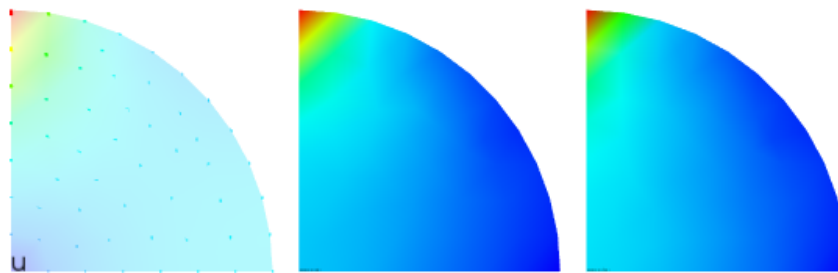
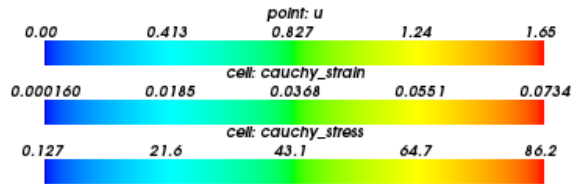
Diametrically point loaded 2-D disk with postprocessing. See *Primer*.

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}

```

```

"""

from 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')

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

    return out

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

```

## linear\_elasticity/its2D\_3.py

### Description

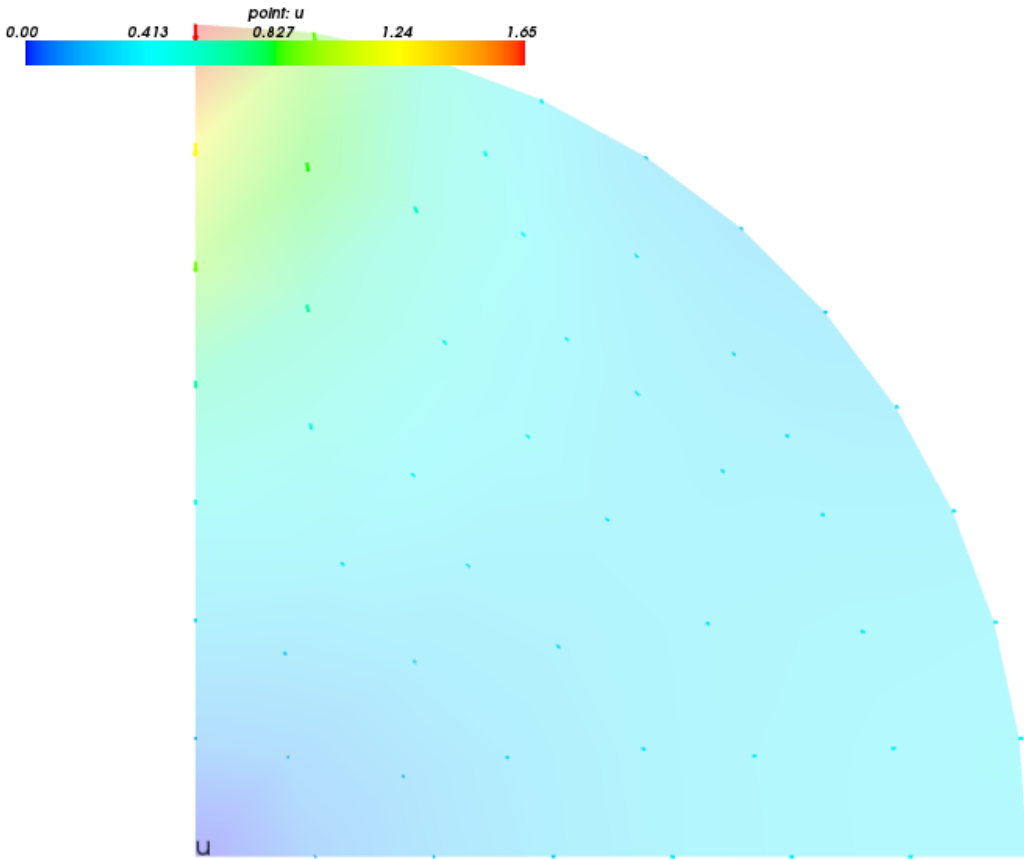
Diametrically point loaded 2-D disk with nodal stress calculation. See *Primer*.

Find  $\underline{u}$  such that:

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

where

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



source code

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

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

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

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""
from its2D_1 import *
```

```

from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.fem.geometry_element import geometry_data
from sfepy.fem import H1NodalVolumeField, FieldVariable
import numpy as nm

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

def nodal_stress(out, pb, state, extend=False):
    '''
    Calculate stresses at nodal points.
    '''

    # Point load.
    mat = pb.get_materials()['Load']
    P = 2.0 * mat.get_data('special', None, 'val')[1]

    # Calculate nodal stress.
    pb.time_update()

    stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D, u)', mode='qp')
    sfield = H1NodalVolumeField('stress', nm.float64, (3,),
                                pb.domain.regions['Omega'])
    svar = FieldVariable('sigma', 'parameter', sfield, 3,
                        primary_var_name='(set-to-None)')
    svar.set_data_from_qp(stress, pb.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][0])
    print 'Vertical compressive stress = %.5e MPa/mm' % (-svar()[0][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' : ('v', 'custom', gdata.coors, [gdata.volume / nc] * nc),
}

```

## linear\_elasticity/its2D\_4.py

### Description

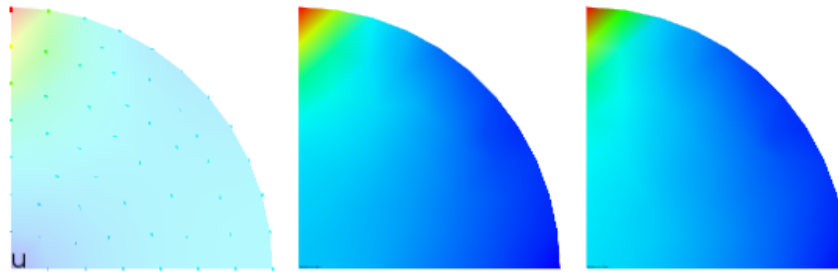
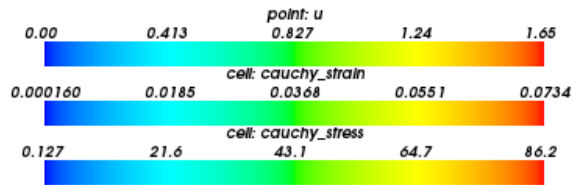
Diametrically point loaded 2-D disk with postprocessing and probes. See *Primer*.

Find  $\underline{u}$  such that:

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

where

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



source code

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

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

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

```
where
```

```
.. math::
```

```

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

    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.fem.probes import LineProbe
    mesh = problem.domain.mesh
    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 xrange(len(ps0)):
        p0, p1 = ps0[ip], ps1[ip]
        probes.append(LineProbe(p0, p1, n_point, mesh))

    return probes, labels

def probe_hook(data, probe, label, problem):
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm

    def get_it(name, var_name):
        var = problem.create_variables([var_name])[var_name]
        var.set_data(data[name].data)

        pars, vals = probe(var)
        vals = vals.squeeze()
        return pars, vals

    results = {}
    results['u'] = get_it('u', 'u')
    results['cauchy_strain'] = get_it('cauchy_strain', 's')

```

```
results['cauchy_stress'] = get_it('cauchy_stress', 's')

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

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

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

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

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/linear\_elastic.py

### Description

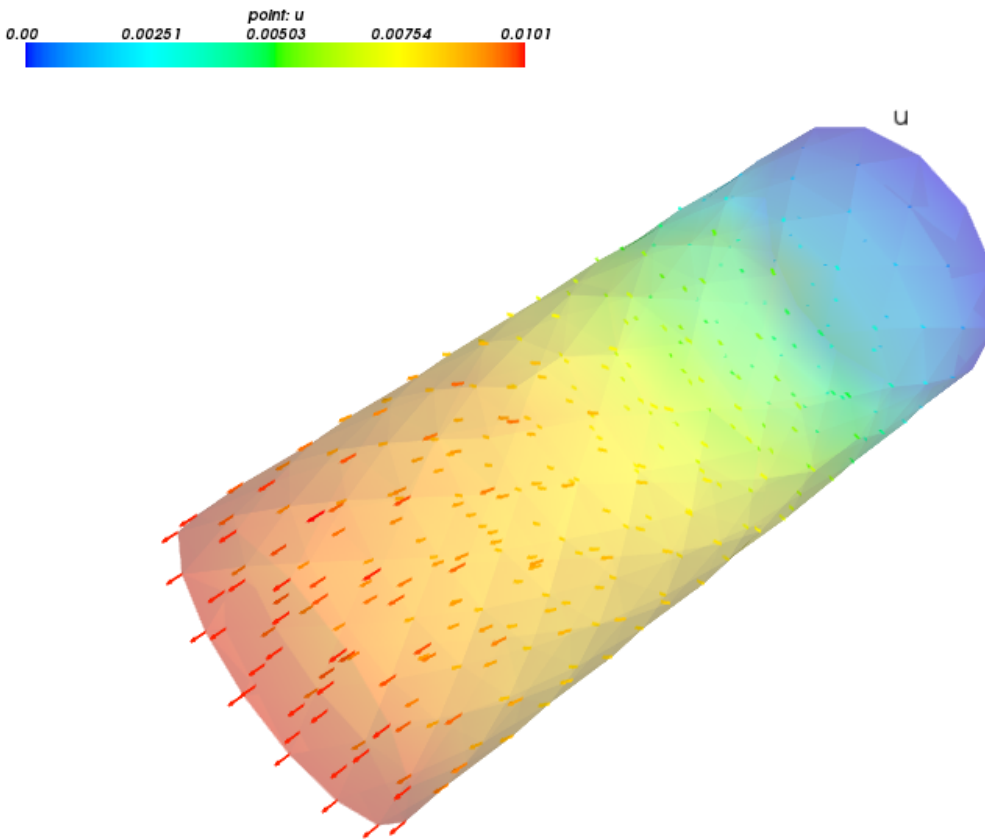
Linear elasticity with comments.

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"""
Linear elasticity with comments.

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

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

where
```

```
.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""
#!
#! 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 0.01 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 nodes for
#! which
#!          (z > 0.017) & (x > 0.03) & (x < 0.07).
#! The output is the displacement for each node, saved by default to
#! simple_out.vtk. The material is linear elastic and its properties are
#! specified as Lamé parameters (see
#! http://en.wikipedia.org/wiki/Lam%C3%A9\_parameters)
#!

#! Mesh
#! ----
from sfepy import data_dir

filename_mesh = data_dir + '/meshes/3d/cylinder.mesh'
#! Regions
#! -----
#! Whole domain 'Omega', left and right ends.
regions = {
    'Omega' : ('all', {}),
    'Left' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
    'SomewhereTop' : ('nodes in (z > 0.017) & (x > 0.03) & (x < 0.07)', {}),
}
#! Materials
#! -----
#! The linear elastic material model is used. Properties are
#! specified as Lamé parameters.
materials = {
    'solid' : ({'lam' : 1e1, 'mu' : 1e0},),
}
#! Fields
#! -----
#! A field is used to define the approximation on a (sub)domain
#! A displacement field (three DOFs/node) will be computed on a region
#! called 'Omega' using P1 (four-node tetrahedral) finite elements.
fields = {
    'displacement': ('real', 'vector', 'Omega', 1),
}
#! Integrals
#! -----
#! Define the integral type Volume/Surface and quadrature rule
#! (here: dim=3, order=1).
integrals = {
```



```

        'il' : ('v', 1),
    }
    #! Variables
    #! -----
    #! One field is used for unknown variable (generate discrete degrees
    #! of freedom) and the second field for the corresponding test variable of
    #! the weak formulation.
    variables = {
        'u' : ('unknown field', 'displacement', 0),
        'v' : ('test field', 'displacement', 'u'),
    }
    #! Boundary Conditions
    #! -----
    #! The left end of the cilinder 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}),
        'Displaced' : ('Right', {'u.0' : 0.01, 'u.[1,2]' : 0.0}),
        'PerturbedSurface' : ('SomewhereTop', {'u.2' : 0.005}),
    }
    #! Equations
    #! -----
    #! The weak formulation of the linear elastic problem.
    equations = {
        'balance_of_forces' :
            """dw_lin_elastic_iso.il.Omega( solid.lam, solid.mu, v, u ) = 0""",
    }
    #! Solvers
    #! -----
    #! Define linear and nonlinear solver.
    #! Even linear problems are solved by a nonlinear solver (KISS rule) - only one
    #! iteration is needed and the final rezidual is obtained for free.
    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,
              'is_plot' : False,
              # 'nonlinear' or 'linear' (ignore i_max)
              'problem' : 'nonlinear' } ),
    }

```

## linear\_elasticity/linear\_elastic\_damping.py

### Description

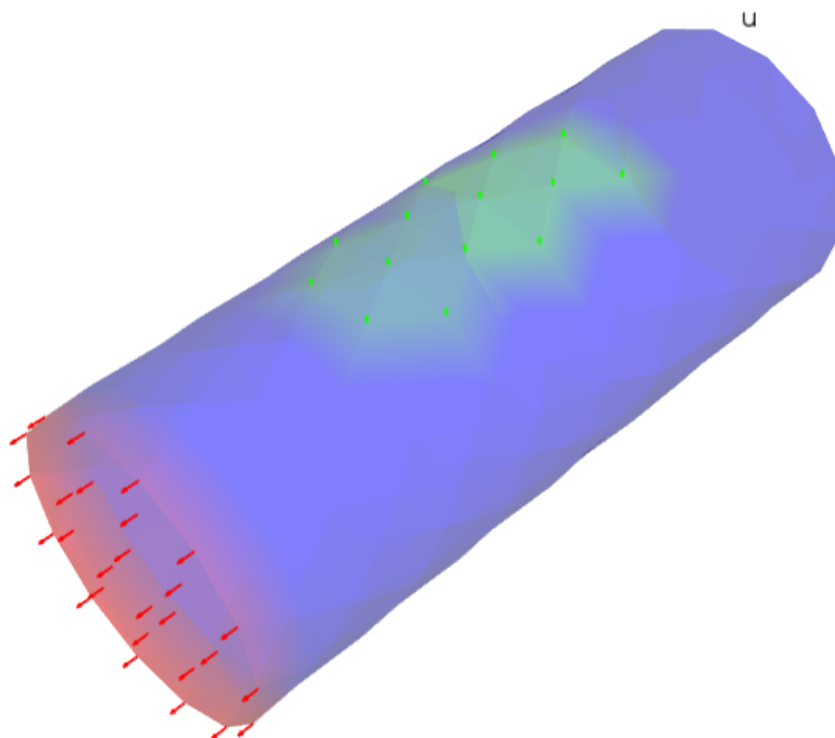
Time-dependent linear elasticity with a simple damping.

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:`\underline{u}` such that:

.. math::
    \int_{\Omega} c \underline{v} \cdot \text{pdiff}(\underline{u})\{t\}
    + \int_{\Omega} D_{ijkl} \underline{e}_{ij}(\underline{v}) \underline{e}_{kl}(\underline{u})
    = 0
    \quad \text{forall } \underline{v} \quad ;,

where

.. math::
```

---

```

    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;.
"""
import numpy as nm
from linear_elastic import \
    filename_mesh, materials, regions, fields, ebcs, \
    integrals, solvers

def print_times(problem, state):
    print nm.array(problem.ts.times)

options = {
    'ts' : 'ts',
    'save_steps' : -1,
    'post_process_hook_final' : print_times,
}

variables = {
    'u' : ('unknown field', 'displacement', 0, 1),
    'v' : ('test field', 'displacement', 'u'),
}

# Put density to 'solid'.
materials['solid'][0].update({'c' : 1000.0})

# Moving the PerturbedSurface region.
ebcs['PerturbedSurface'][1].update({'u.0' : 'ebc_sin'})

def ebc_sin(ts, coors, **kwargs):
    val = 0.01 * nm.sin(2.0*nm.pi*ts.nt)
    return nm.tile(val, (coors.shape[0],))

equations = {
    'balance_of_forces in time' :
    """dw_volume_dot.il.Omega( solid.c, v, du/dt )
    + dw_lin_elastic_iso.il.Omega( solid.lam, solid.mu, v, u ) = 0""",
}

solvers.update({
    'ts' : ('ts.adaptive', {
        't0' : 0.0,
        't1' : 1.0,
        'dt' : None,
        'n_step' : 101,

        'adapt_fun' : 'adapt_time_step',
    }),
})

def adapt_time_step(ts, status, adt, problem):
    if ts.time > 0.5:
        ts.set_time_step(0.1)

    return True

# Pre-assemble and factorize the matrix prior to time-stepping.
newton = solvers['newton']

```

```
newton[1].update({'problem' : 'nonlinear'}) # Change of time step changes
                                           # matrix!

ls = solvers['ls']
ls[1].update({'presolve' : True})

functions = {
    'ebc_sin' : (ebc_sin,),
    'adapt_time_step' : (adapt_time_step,),
}
```

## linear\_elasticity/linear\_elastic\_probes.py

### Description

This example shows how to use the `post_process_hook` and `probe_hook` options.

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

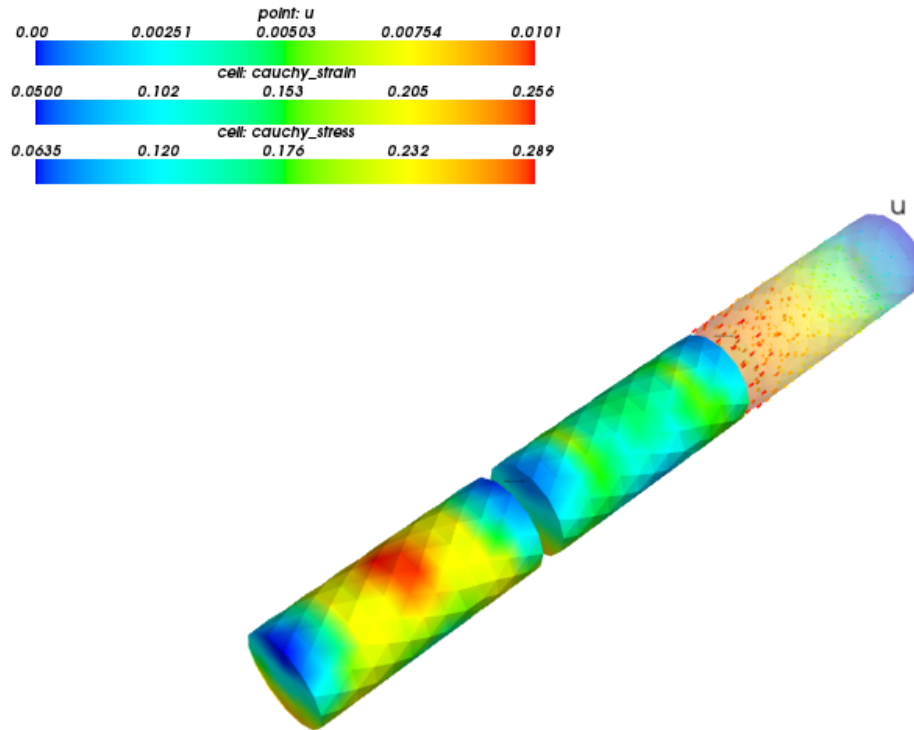
1. solve the problem:  
`./simple.py examples/linear_elasticity/linear_elastic_probes.py`
2. optionally, view the results:  
`./postproc.py cylinder.h5 -b`
3. optionally, convert results to VTK, and view again:  
`./extractor.py -d cylinder.h5 ./postproc.py cylinder.vtk -b`
4. probe the data:  
`./probe.py examples/linear_elasticity/linear_elastic_probes.py cylinder.h5`

Find  $\underline{u}$  such that:

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

where

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



source code

```
"""
This example shows how to use the post_process_hook and probe_hook options.

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/linear_elastic_probes.py

2. optionally, view the results:

    ./postproc.py cylinder.h5 -b

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

    ./extractor.py -d cylinder.h5
    ./postproc.py cylinder.vtk -b

4. probe the data:

    ./probe.py examples/linear_elasticity/linear_elastic_probes.py cylinder.h5

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

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

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad ;.
"""
# Just grab the problem definition of linear_elastic.py.
from linear_elastic import *

# Define options.
options = {
    'output_dir' : '.',
    'output_format' : 'h5', # VTK reader cannot read cell data yet...

    'post_process_hook' : 'post_process',
    'gen_probes' : 'gen_lines',
    'probe_hook' : 'probe_hook',
}

# Update materials, as ev_cauchy_stress below needs the elastic constants in
# the tensor form.
from sfepy.mechanics.matcoefs import stiffness_from_lame

solid = materials['solid'][0]
lam, mu = solid['lam'], solid['mu']
solid.update({
    'D' : stiffness_from_lame(3, lam=lam, mu=mu),
})

# Update fields and variables to be able to use probes for tensors.
fields.update({
    'sym_tensor' : ('real', 6, 'Omega', 0),
})

variables.update({
    's' : ('parameter field', 'sym_tensor', None),
})

# Define the function post_process, that will be called after the problem is
# solved.
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 : ProblemDefinition instance
        The current ProblemDefinition 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.il.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.il.Omega( solid.D, u )',
                              mode='el_avg')
    out['cauchy_stress'] = Struct(name='output_data',
                                  mode='cell', data=stress,
                                  dofs=None)

    return out

# This function will be called by probe.py.
def gen_lines(problem):
    """
    Define three line probes in axial directions.

    Parameters
    -----
    problem : ProblemDefinition instance
        The current ProblemDefinition instance.

    Returns
    -----
    probes : list
        The list of the probes.
    labels : list
        The list of probe labels.
    """
    from sfepy.fem.probes import LineProbe

    mesh = problem.domain.mesh
    bbox = mesh.get_bounding_box()
    cx, cy, cz = 0.5 * bbox.sum(axis=0)
    print bbox
    print cx, cy, cz

    # Probe end points.
    ps0 = [[bbox[0,0], cy, cz],
           [cx, bbox[0,1], cz],
           [cx, cy, bbox[0,2]]]
    ps1 = [[bbox[1,0], cy, cz],

```

```
[cx, bbox[1,1], cz],
[cx, cy, bbox[1,2]]

# 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 xrange(len(ps0)):
    p0, p1 = ps0[ip], ps1[ip]
    probes.append(LineProbe(p0, p1, n_point, mesh))

return probes, labels

# This function will be called by probe.py.
def probe_hook(data, probe, label, problem):
    """
    Parameters
    -----
    data : dict
        The output data.
    probe : Probe subclass instance
        The probe to be used on data.
    label : str
        The label describing the probe.
    problem : ProblemDefinition instance
        The current ProblemDefinition instance.

    Returns
    -----
    fig : figure
        The matplotlib figure with the probe plot.
    results : dict
        The dict of tuples (pars, vals) of the probe parametrization and the
        corresponding probed data.
    """
    import matplotlib.pyplot as plt
    import matplotlib.font_manager as fm
    from sfepy.fem import FieldVariable

    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', '33', '12', '13', '23']

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

plt.subplot(313)
pars, vals = results['cauchy_stress']
for ic in range(vals.shape[1]):
    plt.plot(pars, vals[:,ic], label=r'$\tau_{%s}$' % 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

```

## linear\_elasticity/linear\_elastic\_tractions.py

### Description

Linear elasticity with pressure traction load on a surface and constrained to one-dimensional motion.

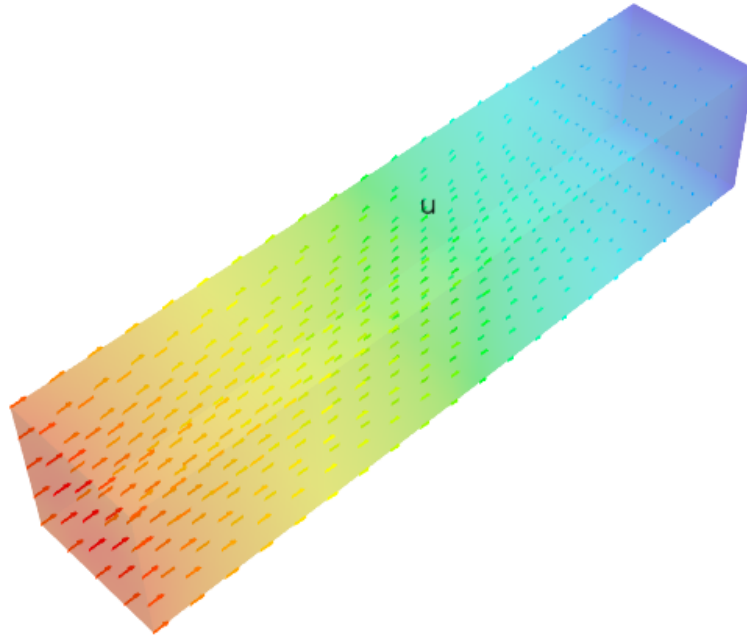
Find  $\underline{u}$  such that:

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

where

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

and  $\underline{\underline{\sigma}} \cdot \underline{n} = \bar{p} \underline{\underline{I}} \cdot \underline{n}$  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:\ul{u}' such that:

.. math::
    \int_{\Omega} D_{ijkl} \ul{e}_{ij} \ul{e}_{kl} \ul{u}
    = - \int_{\Gamma_{right}} \ul{v} \cdot \ul{\sigma} \cdot \ul{n}
    \;, \quad \forall \ul{v} \;,

where

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

and :math:\ul{\sigma} \cdot \ul{n} = \bar{p} \ul{I} \cdot \ul{n}'
with given traction pressure :math:\bar{p}'.
"""
import numpy as nm

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' : ({
            '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' : ('nodes in (x < -4.99)', {}),
        'Right' : ('nodes in (x > 4.99)', {}),
    }

    ebcs = {
        'fixb' : ('Left', {'u.all' : 0.0}),
        'fixt' : ('Right', {'u.[1,2]' : 0.0}),
    }

    ##
    # Balance of forces.
    equations = {
        'elasticity' :
        """dw_lin_elastic_iso.2.Omega( solid.lam, solid.mu, 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,
              'is_plot'    : False,
              # 'nonlinear' or 'linear' (ignore i_max)
              'problem'    : 'nonlinear' } ),
}

return locals()
```

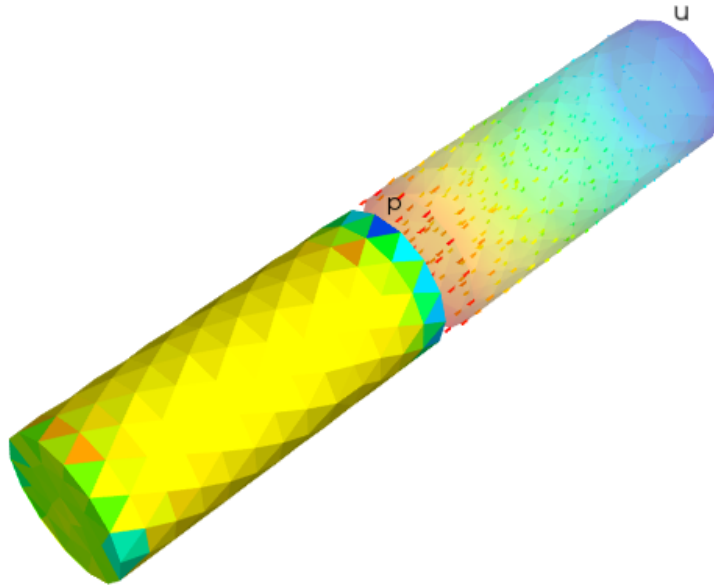
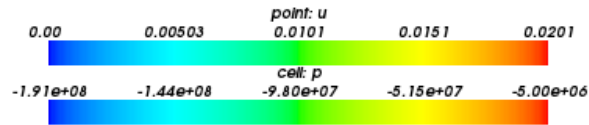
## linear\_elasticity/linear\_elastic\_up.py

### 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:\ul{u}', :math:'p' such that:

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

    - \int_{\Omega} q \ \nabla \cdot \ul{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 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' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}
#!/ 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 = {
    'il' : ('v', 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.il.Omega( solid.D, v, u )
            - dw_stokes.il.Omega( v, p )
            = 0 """ ,
    'pressure constraint' :
        """- dw_stokes.il.Omega( u, q )
            - dw_volume_dot.il.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 rezidual is obtained for free.
solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
        { 'i_max'      : 1,
          'eps_a'      : 1e-2,
          'eps_r'      : 1e-10,
          'problem'    : 'nonlinear' } ),
}
#! 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:

$$\begin{aligned} & \int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \\ & + \int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{\frac{\partial \underline{u}}{\partial \tau}}(\tau)) d\tau \right] e_{ij}(\underline{v}) \\ & = - \int_{\Gamma_{right}} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}, \quad \forall \underline{v}, \end{aligned}$$

where

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

$\mathcal{H}_{ijkl}(0)$  has the same structure as  $D_{ijkl}$  and  $\underline{\underline{\sigma}} \cdot \underline{n} = \bar{p} \underline{\underline{I}} \cdot \underline{n}$  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'
```

Use:

```
$ ./postproc.py -l block.h5
```

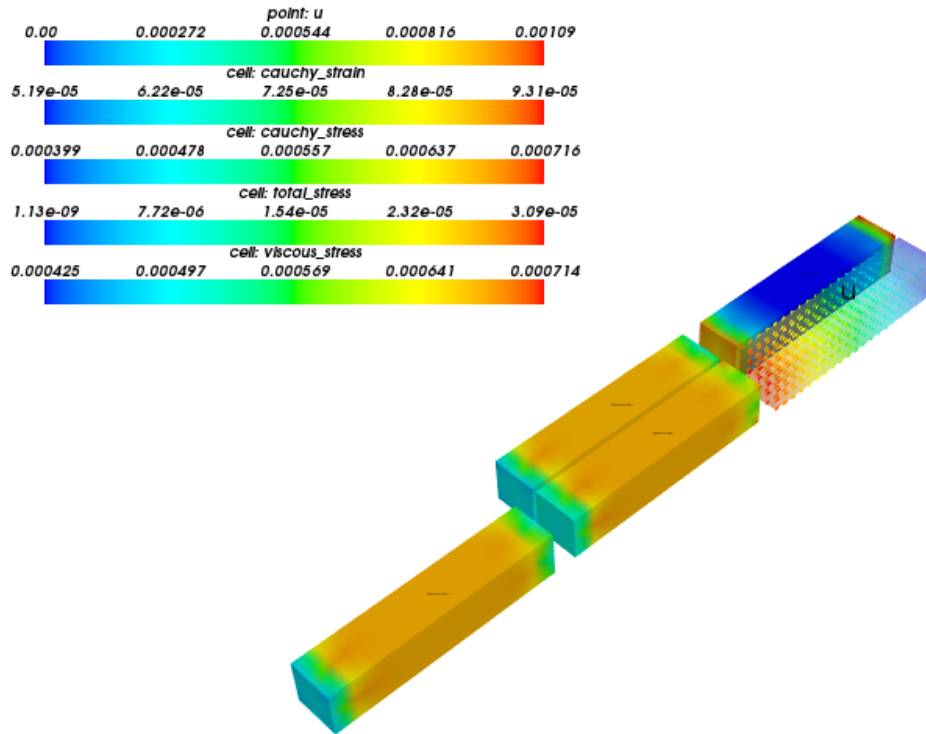
to see names and kinds of variables.

**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

```
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  

$$H_{cal\_ijkl}(t) = H_{cal\_ijkl}(0) e^{-d t}$$
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  $u$  such that:

```
.. math::
\int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u}) \, \,
+ \int_{\Omega} \left[ \int_0^t
H_{cal\_ijkl}(t-\tau) e_{kl}(\mathbf{pdiff}\{\mathbf{u}\}(\tau)) \, \text{d}\tau \right] e_{ij}(\mathbf{v}) \, \,
= - \int_{\Gamma} \gamma \, \mathbf{v} \cdot \boldsymbol{\sigma} \cdot \mathbf{n}
\, ;, \quad \text{forall } \mathbf{v} \, ;,
```

where

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

:math:`\Hcal_{ijkl}(0)` has the same structure as :math:`D_{ijkl}` and
:math:`\ul{\sigma} \cdot \ul{n} = \bar{p} \ul{I} \cdot \ul{n}` with
given traction pressure :math:`\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,
```

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.

"""

```
import numpy as nm
```

```
from sfepy.base.base import output
from sfepy.mechanics.matcoefs import stiffness_from_lame
from sfepy.homogenization.utils import interp_conv_mat
from sfepy import data_dir
```

```
def linear_tension(ts, coors, mode=None, verbose=True, **kwargs):
    if mode == 'qp':
        val = 1.0 * ((ts.step > 0) and (ts.nt <= 0.5))
```

```

        if verbose:
            output('load:', val)

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

        return {'val' : val}

def get_exp_fading_kernel(coef0, decay, times):
    val = coef0[None, ...] * nm.exp(-decay * times[:, None, None])
    return val

def get_th_pars(ts, coors, mode=None, times=None, kernel=None, **kwargs):
    out = {}

    if mode == 'special':
        out['H'] = interp_conv_mat(kernel, ts, times)

    elif mode == 'qp':
        out['H0'] = kernel[0]
        out['Hd'] = kernel[1, 0, 0] / kernel[0, 0, 0]

        for key, val in out.iteritems():
            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' : ('nodes in (x < -4.99)', {}),
    'Right' : ('nodes in (x > 4.99)', {}),
}

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,
          'problem' : 'nonlinear' }),
    'ts' : ('ts.simple', {
        't0' : t0,
        't1' : t1,
        'dt' : None,
        'n_step' : n_step,
        'quasistatic' : True,
    }),
}

def main():
    """
    Plot the load, displacement, strain and stresses w.r.t. time.
    """
    from optparse import OptionParser
    import matplotlib.pyplot as plt

    import sfepy.postprocess.time_history as th

    usage = """%prog <output file in HDF5 format>"""

```

```
msgs = {'node' : 'plot displacements in given node [default: %default]',
        'element' : 'plot tensors in given element [default: %default]',}

parser = OptionParser(usage=usage)
parser.add_option('-n', '--node', type=int, metavar='ii',
                  action='store', dest='node',
                  default=512, help=msgs['node'])
parser.add_option('-e', '--element', type=int, metavar='ii',
                  action='store', dest='element',
                  default=299, help=msgs['element'])
options, args = parser.parse_args()

if len(args) == 1:
    filename = args[0]
else:
    parser.print_help()
    return

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

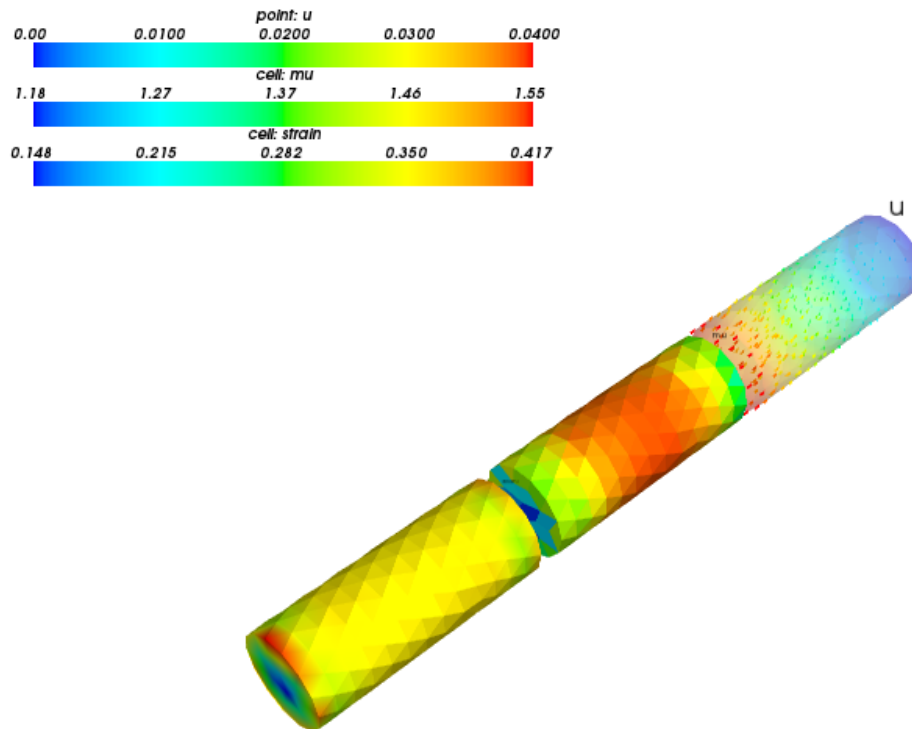
### Description

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 -*-
r"""
Example demonstrating how a linear elastic term can be used to solve an
elasticity problem with a material nonlinearity.

.. math::
    \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) 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 \forall i, j, k, l.
"""
import numpy as nm

from sfepy.linalg import norm_l2_along_axis
from sfepy import data_dir

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

def post_process(out, pb, state, extend=False):
```



```

from sfepy.base.base import Struct

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

strain = pb.evaluate('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
out['strain'] = Struct(name='strain', mode='cell', data=strain, dofs=None)

return out

strains = [None]

def get_pars(ts, coors, mode='qp',
            equations=None, term=None, problem=None, **kwargs):
    """
    The material nonlinearity function - the Lamé coefficient 'mu'
    depends on the strain.
    """
    if mode != 'qp': return

    val = nm.empty((coors.shape[0], 1, 1), 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[:, None, None]

    # Store history.
    strains[0] = strain

    return {'mu' : val}

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_steps' : -1,

    'post_process_hook' : 'post_process',
}

regions = {
    'Omega' : ('all', {}),
    'Left' : ('nodes in (x < 0.001)', {}),
    'Right' : ('nodes in (x > 0.099)', {}),
}

materials = {
    'linear' : ({'lam' : 1e1},),
    '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_iso.2.Omega(linear.lam, nonlinear.mu, v, u) = 0""",
}

solvers = {
    'ls' : ('ls.scipy_direct', {}),
    'newton' : ('nls.newton',
        { 'i_max' : 1,
          'eps_a' : 1e-10,
          'eps_r' : 1.0,
          'problem' : 'nonlinear' }),
    'ts' : ('ts.simple',
        { 't0' : 0.0,
          't1' : 1.0,
          'dt' : None,
          'n_step' : 5,
          'quasistatic' : True,
        }),
}
```

## 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

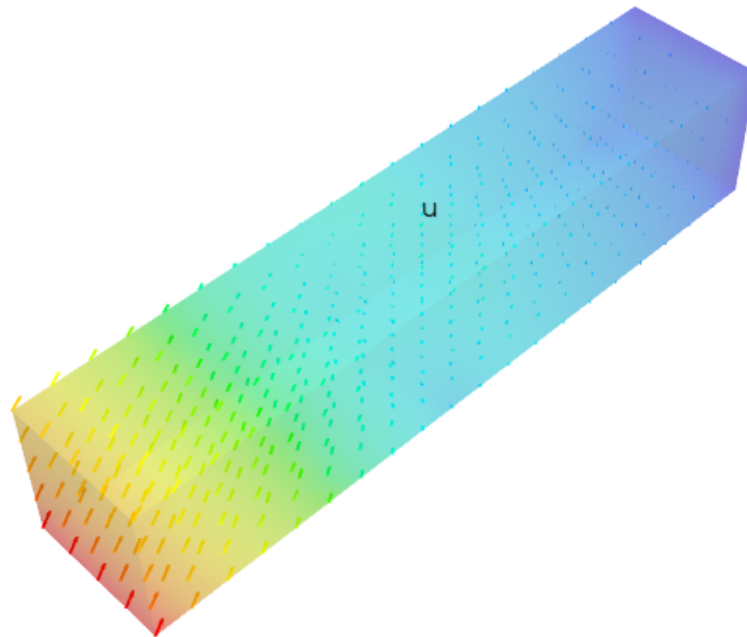
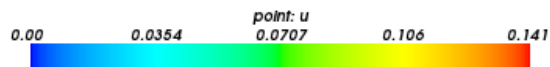
$$D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}.$$

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  $\mathbf{u}$  such that:

```
.. math::
    \int_{\Omega} D_{ijkl} \epsilon_{ij}(\mathbf{v}) \epsilon_{kl}(\mathbf{u})
    + \int_{\Omega_1} \sigma_{ij} \epsilon_{ij}(\mathbf{v})
    + \int_{\Omega_2} D^f_{ijkl} \epsilon_{ij}(\mathbf{v}) \left( d_k d_l \right)
    = 0
    \;, \quad \forall \mathbf{v} \; ;,
```

where

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

The stiffness of fibres  $D^f_{ijkl}$  is defined analogously,  
 $\mathbf{d}$  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
"""
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' : ('nodes in (x < -4.99)', {}),
    'Omega1' : ('nodes in (x < 0.001)', {}),
    'Omega2' : ('nodes 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.Omega1( solid.prestress, v )
    + dw_lin_strain_fib.2.Omega2( solid.DF, solid.nu, v )
    = 0""",
}

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

```

## 6.1.7 miscellaneous examples

### miscellaneous/compare\_scalar\_terms.py

#### Description

Example without a physical relevance comparing new and old-style terms with scalar variables.

Find  $p$  (new style terms),  $r$  (old\_style terms) such that:

$$\int_{\Omega} c \delta_{ij} \nabla_i q \nabla_j p + \int_{\Omega} q p = 0, \quad \forall q,$$

$$\int_{\Omega} c \delta_{ij} \nabla_i s \nabla_j r + \int_{\Omega} s r = 0, \quad \forall s.$$

The same values of  $p, r$  should be obtained.

source code

```

r"""
Example without a physical relevance comparing new and old-style terms
with scalar variables.

Find :math:`p` (new style terms), :math:`r` (old_style terms) such that:

.. math::
    \int_{\Omega} c \delta_{ij} \nabla_i q \nabla_j p
    + \int_{\Omega} q p
    = 0
    \;, \quad \text{forall } q \;,

    \int_{\Omega} c \delta_{ij} \nabla_i s \nabla_j r
    + \int_{\Omega} s r
    = 0
    \;, \quad \text{forall } s \;.

```

```
The same values of :math:'p', :math:'r' should be obtained.
"""
import os

import numpy as nm

from sfepy import data_dir
from sfepy.fem import MeshIO

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

xmin, ymin, zmin = bbox[0, :] + 1e-4 * dd
xmax, ymax, zmax = bbox[1, :] - 1e-4 * dd

def post_process(out, pb, state, extend=False):

    for vn in ['p', 'r']:
        try:
            dd = pb.evaluate('dw_new_diffusion.2.Omega(m.c, %s, %s)'
                             % (vn, vn), verbose=False)
            print 'dw_new_diffusion', vn, dd

            dd = pb.evaluate('dw_diffusion.2.Omega(m.c, %s, %s)'
                             % (vn, vn), verbose=False)
            print 'dw_diffusion', vn, dd

            mass = pb.evaluate('dw_new_mass.2.Omega(%s, %s)'
                               % (vn, vn), verbose=False)
            print 'dw_new_mass', vn, mass

            mass = pb.evaluate('dw_new_mass_scalar.2.Omega(%s, %s)'
                               % (vn, vn), verbose=False)
            print 'dw_new_mass_scalar', vn, mass

            mass = pb.evaluate('dw_volume_dot.2.Omega(%s, %s)'
                               % (vn, vn), verbose=False)
            print 'dw_volume_dot', vn, mass

        except:
            pass

    return out

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process',
}

materials = {
    'm' : ({'c' : 0.0001 * nm.eye(3)}),)
```

```

}

regions = {
    'Omega' : ('all', {}),
    'Gamma_Left' : ('nodes in (x < %f)' % xmin, {}),
    'Gamma_Right' : ('nodes in (x > %f)' % xmax, {}),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 2),
}

variables = {
    'p' : ('unknown field', 'temperature', 0),
    'q' : ('test field', 'temperature', 'p'),
    'r' : ('unknown field', 'temperature', 1),
    's' : ('test field', 'temperature', 'r'),
}

ebcs = {
    'p1' : ('Gamma_Left', {'p.0' : 2.0}),
    'p2' : ('Gamma_Right', {'p.0' : -2.0}),
    'r1' : ('Gamma_Left', {'r.0' : 2.0}),
    'r2' : ('Gamma_Right', {'r.0' : -2.0}),
}

equations = {
    'new equation' :
        """dw_new_diffusion.2.Omega(m.c, q, p)
        + dw_new_mass.2.Omega(q, p)
        = 0""",
    'equation' :
        """dw_diffusion.2.Omega(m.c, s, r)
        + dw_volume_dot.2.Omega(s, r)
        = 0""",
}

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

```

## miscellaneous/compare\_vector\_terms.py

### Description

Example without a physical relevance comparing new and old-style terms with vector variables.

Find  $\underline{u}$  (new style terms),  $\underline{r}$  (old\_style terms) such that:

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

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{s}) e_{kl}(\underline{r}) \int_{\Omega} \underline{s} \cdot \underline{r} = 0, \quad \forall \underline{s}.$$

The same values of  $u$ ,  $r$  should be obtained.

source code

```
r"""
Example without a physical relevance comparing new and old-style terms
with vector variables.

Find :math: \ul{u} (new style terms), :math: \ul{r} (old_style terms)
such that:
```

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

\int_{\Omega} D_{ijkl} \ul{s} e_{ij} \ul{s} e_{kl} \ul{r}
\int_{\Omega} \ul{s} \cdot \ul{r}
= 0
\;, \quad \forall \ul{s} \;.
```

```
The same values of :math: \ul{u}, :math: \ul{r} should be obtained.
"""
```

```
import os

from sfepy import data_dir
from sfepy.fem import MeshIO
from sfepy.mechanics.matcoefs import stiffness_from_lame

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

xmin, ymin, zmin = bbox[0, :] + 1e-4 * dd
xmax, ymax, zmax = bbox[1, :] - 1e-4 * dd

def post_process(out, pb, state, extend=False):

    for vn in ['u', 'r']:
        try:
            val = pb.evaluate('dw_new_mass.2.Omega(%s, %s)'
                              % (vn, vn), verbose=False)
            print 'dw_new_mass', vn, val

            val = pb.evaluate('dw_new_lin_elastic.2.Omega(m.D, %s, %s)'
                              % (vn, vn), verbose=False)
            print 'dw_new_lin_elastic', vn, val

            val = pb.evaluate('dw_lin_elastic.2.Omega(m.D, %s, %s)'
                              % (vn, vn), verbose=False)
            print 'dw_lin_elastic', vn, val

        except:
```



```

        pass

    return out

options = {
    'nls' : 'newton',
    'ls' : 'ls',
    'post_process_hook' : 'post_process',
}

materials = {
    'm' : ({'D' : stiffness_from_lame(3, lam=0.0007, mu=0.0003),
            'one' : 1.0}),
}

regions = {
    'Omega' : ('all', {}),
    'Gamma_Left' : ('nodes in (x < %f)' % xmin, {}),
    'Gamma_Right' : ('nodes in (x > %f)' % xmax, {}),
}

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

variables = {
    'u' : ('unknown field', 'displacements', 0),
    'v' : ('test field', 'displacements', 'u'),
    'r' : ('unknown field', 'displacements', 1),
    's' : ('test field', 'displacements', 'r'),
}

ebcs = {
    'u1' : ('Gamma_Left', {'u.all' : 0.0}),
    'u2' : ('Gamma_Right', {'u.0' : 0.1 * (xmax - xmin),
                             'u.1' : 0.1 * (ymax - ymin)}),
    'r1' : ('Gamma_Left', {'r.all' : 0.0}),
    'r2' : ('Gamma_Right', {'r.0' : 0.1 * (xmax - xmin),
                             'r.1' : 0.1 * (ymax - ymin)}),
}

equations = {
    'new equation' :
        """dw_new_lin_elastic.2.Omega(m.D, v, u)
        + dw_new_mass.2.Omega(v, u)
        = 0""",
    'equation' :
        """dw_lin_elastic.2.Omega(m.D, s, r)
        + dw_volume_dot.2.Omega(m.one, s, r)
        = 0""",
}

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

```

```
}
```

## 6.1.8 navier\_stokes examples

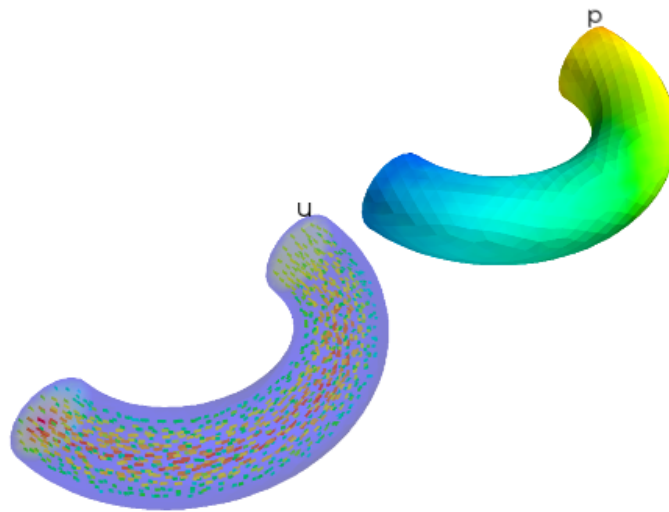
navier\_stokes/navier\_stokes.py

### Description

Navier-Stokes equations for incompressible fluid flow.

Find  $\underline{u}$ ,  $p$  such that:

$$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u} + \int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v} - \int_{\Omega} p \nabla \cdot \underline{v} = 0, \quad \forall \underline{v},$$
$$\int_{\Omega} q \nabla \cdot \underline{u} = 0, \quad \forall q.$$



source code

```
r"""
Navier-Stokes equations for incompressible fluid flow.

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

.. math::
```

```

\int_{\Omega} \nu \nabla \cdot \mathbf{u} : \nabla \mathbf{u}
+ \int_{\Omega} ((\mathbf{u} \cdot \nabla) \mathbf{u}) \cdot \mathbf{v}
- \int_{\Omega} p \nabla \cdot \mathbf{v}
= 0
\;, \quad \forall \mathbf{v} \;,

\int_{\Omega} q \nabla \cdot \mathbf{u}
= 0
\;, \quad \forall q \;.
"""
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' : 'elements of group 6',
}

region_0 = {
    'name' : 'Walls',
    'select' : 'nodes of surface -n (r.Outlet +n r.Inlet)',
    'can_cells' : False,
}

```

```
}
region_1 = {
    'name' : 'Inlet',
    'select' : 'nodes by cinc0', # In
    'can_cells' : False,
}
region_2 = {
    'name' : 'Outlet',
    'select' : 'nodes by cinc1', # Out
    'can_cells' : False,
}

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',
    'kind' : 'v',
    'order' : 2,
}

integral_2 = {
    'name' : 'i2',
    'kind' : 'v',
    'order' : 3,
}

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

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

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

    'i_max'      : 5,
    'eps_a'      : 1e-8,
    'eps_r'      : 1.0,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red'     : 0.1,
    'ls_red_warp' : 0.001,
    'ls_on'      : 0.99999,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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/stabilized\_navier\_stokes.py

### Description

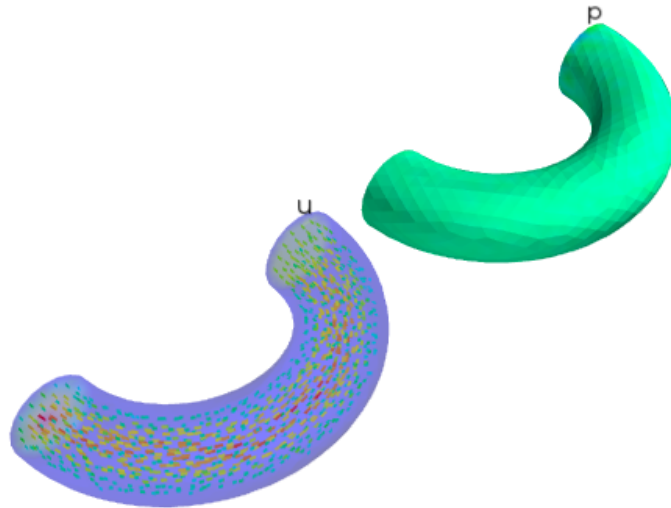
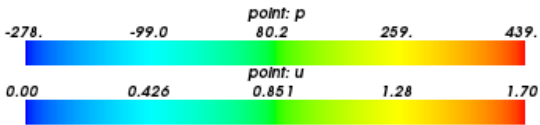
Stabilized Navier-Stokes problem with grad-div, SUPG and PSPG stabilization solved by a custom Oseen solver.

The stabilization used was introduced 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:\ul{u}', :math:'p' such that:

.. math::
\begin{array}{l}
\int_{\Omega} \nu \nabla \ul{v} : \nabla \ul{u} \\
\int_{\Omega} ((\ul{b} \cdot \nabla) \ul{u}) \cdot \ul{v} \\
- \int_{\Omega} p \nabla \cdot \ul{v} \\
+ \gamma \int_{\Omega} (\nabla \cdot \ul{u}) \cdot (\nabla \cdot \ul{v}) \\
+ \sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\ul{b} \cdot \nabla) \ul{u}) \cdot ((\ul{b} \cdot \nabla) \ul{v}) \\
+ \sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p \cdot ((\ul{b} \cdot \nabla) \ul{v}) \\
= 0 \\
\end{array}
\quad \forall \ul{v} \quad ;,
```

```
\end{array}

\begin{array}{l}
\int_{\Omega} q \nabla \cdot \mathbf{u} \, \, \\
+ \sum_{K \in \mathcal{I}_{cal\_h}} \int_{T_K} \tau_K \, ((\mathbf{b} \cdot \nabla) \mathbf{u}) \\
\quad \cdot \nabla q \, \, \\
+ \sum_{K \in \mathcal{I}_{cal\_h}} \int_{T_K} \tau_K \, \nabla p \cdot \nabla q \\
= 0 \\
\,; \, \quad \forall q \,; .
\end{array}
"""
from 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' : ('nodes of surface -n (r.Outlet +n r.Inlet)',
               {'can_cells' : False}),
    'Inlet' : ('nodes by cinc0', {'can_cells' : False}),
    'Outlet' : ('nodes by cinc1', {'can_cells' : False}),
}

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' : ('v', 2),
    'i2' : ('v', 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.il.Omega( v, p )
            + dw_st_grad_div.il.Omega( stabil.gamma, v, u )
            + dw_st_supg_c.il.Omega( stabil.delta, v, b, u )
            + dw_st_supg_p.il.Omega( stabil.delta, v, b, p )
            = 0""",
    'incompressibility' :
        """ dw_stokes.il.Omega( u, q )
            + dw_st_pspg_c.il.Omega( stabil.tau, q, b, u )
            + dw_st_pspg_p.il.Omega( stabil.tau, q, p )
            = 0""",
}

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

    'needs_problem_instance' : True,
    'stabil_mat' : 'stabil',

    'adimensionalize' : False,
    'check_navier_stokes_rezidual' : 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).
    'is_plot' : False,

    # 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', 'il' : 'il', '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

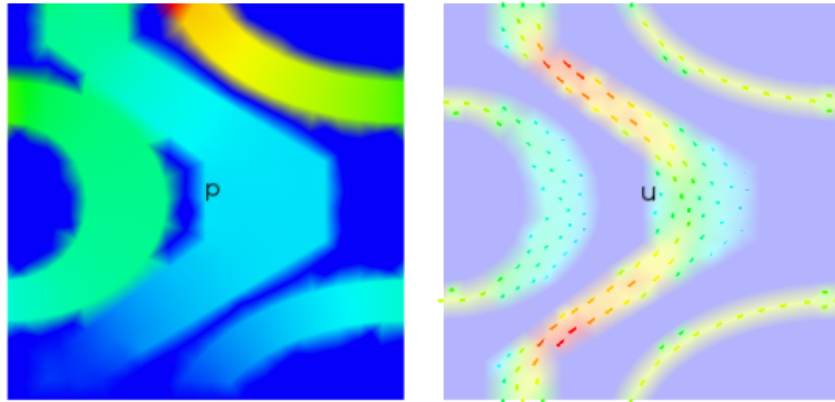
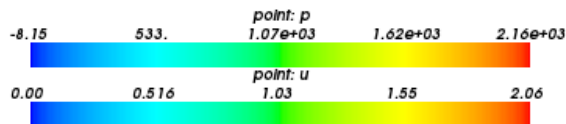
### Description

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  $\mathbf{u}$ ,  $p$  such that:

.. math::
\int_{Y_1 \cup Y_2} \nu \nabla \mathbf{v} : \nabla \mathbf{u}
- \int_{Y_1 \cup Y_2} p \nabla \cdot \mathbf{v}
= 0
\;, \quad \forall \mathbf{v} \;,

\int_{Y_1 \cup Y_2} q \nabla \cdot \mathbf{u}
= 0
\;, \quad \forall q \;.
"""
from sfepy import data_dir
from sfepy.fem.periodic import match_y_line

filename_mesh = data_dir + '/meshes/2d/special/channels_symm944t.mesh'

if filename_mesh.find( 'symm' ):
    region_1 = {
        'name' : 'Y1',
        'select' : ""elements of group 3"",
    }
    region_2 = {
        'name' : 'Y2',
        'select' : ""elements of group 4 +e elements of group 6
                  +e elements of group 8"",
    }
    region_4 = {
        'name' : 'Y1Y2',
        'select' : ""r.Y1 +e r.Y2"",
    }
    region_5 = {
        'name' : 'Walls',
        'select' : ""r.EBCGamma1 +n r.EBCGamma2"",
    }
    region_310 = {
        'name' : 'EBCGamma1',
        'select' : ""(elements of group 1 *n elements of group 3)
                  +n
                  (elements of group 2 *n elements of group 3)
                  "",
    }
    region_320 = {
        'name' : 'EBCGamma2',
        'select' : ""(elements of group 5 *n elements of group 4)
                  +n
                  (elements of group 1 *n elements of group 4)
                  +n
                  (elements of group 7 *n elements of group 6)
                  +n
                  (elements of group 2 *n elements of group 6)
                  +n

```

```
        (elements of group 9 *n elements of group 8)
        +n
        (elements of group 2 *n elements of group 8)
        """,
    }

w2 = 0.499
# Sides.
region_20 = {
    'name' : 'Left',
    'select' : 'nodes in (x < %.3f)' % -w2,
}
region_21 = {
    'name' : 'Right',
    'select' : 'nodes in (x > %.3f)' % w2,
}
region_22 = {
    'name' : 'Bottom',
    'select' : 'nodes in (y < %.3f)' % -w2,
}
region_23 = {
    'name' : 'Top',
    'select' : 'nodes in (y > %.3f)' % w2,
}

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' : 'i1',
        'kind' : 'v',
        'order' : 2,
    }

    equations = {
        'balance' :
            """dw_div_grad.i1.Y1Y2( fluid.viscosity, v, u )
            - dw_stokes.i1.Y1Y2( v, p ) = 0""",
        'incompressibility' :
            """dw_stokes.i1.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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

save_format = 'hdf5' # 'hdf5' or 'vtk'
```

## navier\_stokes/utils.py

### Description

missing description!

source code

```
##
# Functions.
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)
```

## 6.1.9 phononic examples

### 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 sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.base.ioutils import InDir
from sfepy.homogenization.coefficients import Coefficients

from band_gaps_conf import BandGapsConf, get_pars, clip_sqrt, normalize

clip_sqrt, normalize # Make pyflakes happy...

incwd = InDir(__file__)

filename = data_dir + '/meshes/2d/special/circle_in_square.mesh'

output_dir = incwd('output/band_gaps')

# aluminium, in 1e+10 Pa
D_m = get_pars(2, 5.898, 2.681)
density_m = 0.2799 # in 1e4 kg/m3

# epoxy, in 1e+10 Pa
D_c = get_pars(2, 0.1798, 0.148)
density_c = 0.1142 # in 1e4 kg/m3

mat_pars = Coefficients(D_m=D_m, density_m=density_m,
                        D_c=D_c, density_c=density_c)

region_selects = Struct(matrix=('elements of group 1', {}),
```

```
inclusion=('elements of group 2', {}))

corrs_save_names = {'evp' : 'evp', 'corrs_rs' : 'corrs_rs'}

options = {
    'plot_transform_angle' : None,
    'plot_transform_wave' : ('clip_sqrt', (0, 30)),
    'plot_transform' : ('normalize', (-2, 2)),

    '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',
                    'text.fontsize': 'large',
                    'legend.fontsize': 'large',
                    'legend.loc': 1,
                    'xtick.labelsize': 'large',
                    'ytick.labelsize': 'large',
                    'text.usetex': True},
    },
}

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]])
'freq_margins' : (10, 10), # % of freq_range
'freq_eps' : 1e-12, # frequency
'zezo_eps' : 1e-12, # zero finding
'freq_step' : 0.0001, # % of freq_range

'log_save_name' : 'band_gaps.log',
}

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.
"""
import numpy as nm

from sfepy.base.base import get_default, import_file, Struct
from sfepy.base.conf import ProblemConf
from sfepy.fem import MeshIO
import sfepy.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

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.umfpack', {}),
        'ls_i' : ('ls.scipy_iterative', {
            'method' : 'cg',
            'i_max'   : 1000,
            'eps_a'   : 1e-12,
        }),
        'newton' : ('nls.newton', {
            'i_max' : 1,
            'eps_a' : 1e-4,
            'problem' : 'nonlinear',
        }),
    }

    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 * n r.Y_c', {'can_cells' : False}),
    }

    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):
    ebcs = {
        'fixed_corners' : ('Corners', {'u_m.all' : 0.0}),
        'fixed_gamma_mc' : ('Gamma_mc', {'u_c.all' : 0.0}),
    }

    epbcs = {}
    all_periodic = []
    for vn in ['u_m']:
        val = {'%s.all' % vn : '%s.all' % vn}

        epbcs.update({
            'periodic_%s_x' % vn : ([ 'Left', 'Right'], val,
                                    'match_y_line'),
            'periodic_%s_y' % vn : ([ 'Top', 'Bottom'], val,
                                    'match_x_line'),
        })
        all_periodic.extend(['periodic_%s_x' % vn, 'periodic_%s_y' % vn])

    lcbcs = {}

    return ebcs, epbcs, lcbcs, all_periodic

def define_functions(self):
    functions = {
        'match_x_line' : (per.match_x_line,),
        'match_y_line' : (per.match_y_line,),
    }

    return functions

def define_integrals(self):
    integrals = {
        'il' : ('v', 2),
    }

    return integrals

def define_equations(self):
    equations = {}
    equations['corrs_rs'] = {
```

```

        'balance_of_forces' :
        """dw_lin_elastic.il.Y_m( m.D_m, v_m, u_m )
           = - dw_lin_elastic.il.Y_m( m.D_m, v_m, Pi )""",
    }
    equations['evp'] = {
        'lhs' : """dw_lin_elastic.il.Y_c( m.D_c, v_c, u_c )""",
        'rhs' : """dw_volume_dot.il.Y_c( m.density_c, v_c, u_c )""",
    }

    expr_coefs = {
        'D' : """dw_lin_elastic.il.Y_m( m.D_m, u1_m, u2_m )""",
        'VF' : """d_volume.il.%s(aux)""",
        'ema' : """ev_volume_integrate.il.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.CoeffSymSym,
    },
    'Gamma' : {
        'requires' : ['c.D'],
        'options' : {
            'mode' : 'simple',
            'incident_wave_dir' : None,
        },
        'class' : cp.ChristoffelAcousticTensor,
    },
    'dispersion' : {
        'requires' : ['evp', 'c.eigenmomenta', 'c.M', 'c.Gamma'],
        'options' : 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'],
            'dump_variables' : ['u_c'],
            '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'],
        'dump_variables' : ['u_m'],
        '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' : ('nodes by select_yr', {}),
            'Y_c' : ('r.Y_cr -e 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}),
        }
        lcbcs = {
            'rigid' : ('Y_r', {'u.all' : 'rigid'}),
        }

```

```
    return ebc, {}, lcbc, []

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.il.Y_r( m.D_r, v, u ) should have no effect!
    equations['evp'] = {
        'lhs' : """dw_lin_elastic.il.Y_c( m.D_c, v, u )
                  + dw_lin_elastic.il.Y_r( m.D_r, v, u )""",
        'rhs' : """dw_volume_dot.il.Y_c( m.density_c, v, u )
                  + dw_volume_dot.il.Y_r( m.density_r, v, u )""",
    }

    expr_coefs = {
        'VF' : """d_volume.il.%s(aux)""",
        'ema' : """ev_volume_integrate.il.Y_c( m.density_c, u )
                  + ev_volume_integrate.il.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,
            'lcbcs' : ['rigid'],
            'equations' : self.equations['evp'],
            'save_name' : self.corrs_save_names['evp'],
            'dump_variables' : ['u'],
            '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'.
"""
import numpy as nm

from sfepy import data_dir
from sfepy.base.base import Struct
from sfepy.base.ioutils import InDir
from sfepy.fem import extend_cell_data
from sfepy.linalg import norm_l2_along_axis
from sfepy.homogenization.coefficients import Coefficients

from 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, in 1e+10 Pa
D_m = get_pars(dim, 5.898, 2.681)
density_m = 0.2799 # in 1e4 kg/m3

# epoxy, in 1e+10 Pa
D_c = get_pars(dim, 0.1798, 0.148)
density_c = 0.1142 # in 1e4 kg/m3

# lead, in 1e+10 Pa, does not matter
D_r = get_pars(dim, 4.074, 0.5556)
density_r = 1.1340 # in 1e4 kg/m3

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=('elements of group 1', {}),
                        inclusion=('elements 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., 35.), # overrides eig_range!

    'freq_eps' : 1e-12, # frequency
    'zezo_eps' : 1e-12, # zero finding
    'freq_step' : 0.01, # % of freq_range

    'log_save_name' : 'band_gaps.log',
}

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.
    },
}

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.il.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()
```

## 6.1.10 piezo\_elasticity examples

### piezo\_elasticity/piezo.py

#### Description

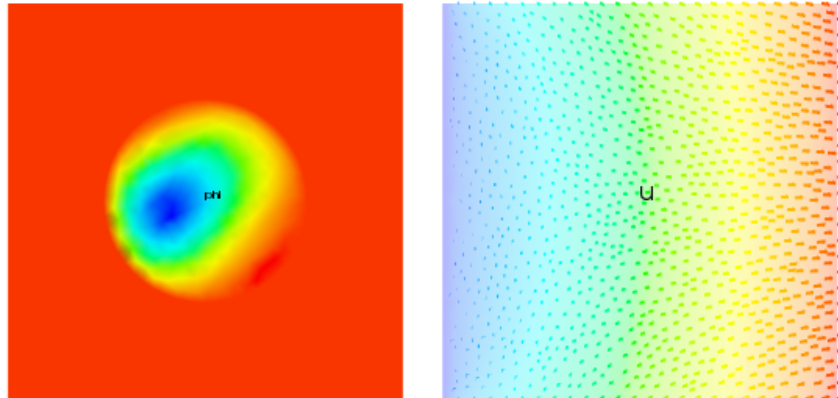
Piezo-elasticity problem - linear elastic material with piezoelectric effects.

Find  $\underline{u}, \phi$  such that:

$$\begin{aligned}
 -\omega^2 \int_Y \rho \underline{v} \cdot \underline{u} + \int_Y D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{Y_2} g_{kij} e_{ij}(\underline{v}) \nabla_k \phi &= 0, \quad \forall \underline{v}, \\
 \int_Y K_{ij} \nabla_i \psi \nabla_j \phi + \int_{Y_2} g_{kij} e_{ij}(\underline{u}) \nabla_k \psi &= 0, \quad \forall \psi,
 \end{aligned}$$

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:\ul{u}', :math:\ul{\phi}' such that:

.. math::
    - \omega^2 \int_{\Omega} \rho \ul{v} \cdot \ul{u}
    + \int_{\Omega} D_{ijkl} e_{ij}(\ul{v}) e_{kl}(\ul{u})
    - \int_{\Omega} g_{kij} e_{ij}(\ul{v}) \nabla_k \phi
    = 0
    \quad \forall \ul{v} \in V;

    \int_{\Omega} K_{ij} \nabla_i \psi \nabla_j \phi
    - \int_{\Omega} g_{kij} e_{ij}(\ul{u}) \nabla_k \psi
    = 0
    \quad \forall \psi \in W;

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \quad \forall i, j, k, l \in \{1, 2, 3\}
    """
```

```
import os
import numpy as nm

from sfepy import data_dir
from sfepy.fem import MeshIO

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]

regions = {
    'Y' : ('all', {}),
    'Y1' : ('elements of group 1', {}),
    'Y2' : ('elements of group 2', {}),
    'Y2_Surface': ('r.Y1 *n r.Y2', {'can_cells' : False}),
    'Left' : ('nodes in (x < %f)' % (x_left + 1e-3), {}),
    'Right' : ('nodes in (x > %f)' % (x_right - 1e-3), {}),
}

material_2 = {
    'name' : 'inclusion',

    # epoxy
    'function' : 'get_inclusion_pars',
}

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 = {
            # Lamé coefficients in 1e+10 Pa.
            '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 out.iteritems():
        out[key] = nm.tile(val, (coor.shape[0], 1, 1))
    return out

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

field_0 = {
    'name' : 'displacement',
    'dtype' : nm.float64,
    'shape' : dim,
    'region' : 'Y',
    'approx_order' : 1,
}

field_2 = {
    'name' : 'potential',
    'dtype' : nm.float64,
    'shape' : (1,),
    'region' : 'Y',
    'approx_order' : 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}),
}

integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'order' : 2,
}

equations = {
    '1' : """- %f * dw_volume_dot.i1.Y( inclusion.density, v, u )
              + dw_lin_elastic_iso.i1.Y( inclusion.lam, inclusion.mu, v, u )
              - dw_piezo_coupling.i1.Y2( inclusion.coupling, v, phi )
              = 0""" % omega_squared,
    '2' : """dw_diffusion.i1.Y( inclusion.dielectric, psi, phi )
              + dw_piezo_coupling.i1.Y2( inclusion.coupling, u, psi )
              = 0""",
}

##
# Solvers etc.

```

```
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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}
```

## 6.1.11 quantum examples

### quantum/boron.py

#### Description

missing description!

source code

```
from sfepy.linalg import norm_l2_along_axis

from quantum_common import common

def fun_v(ts, coor, mode=None, **kwargs):
    from numpy import sqrt

    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():
    l = common(fun_v, n_eigs=10, tau=-15)
    return l
```



## quantum/hydrogen.py

### Description

missing description!

source code

```

from sfepy.linalg import norm_l2_along_axis

from quantum_common import common

def fun_v(ts, coor, mode=None, **kwargs):
    from numpy import sqrt

    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():
    l = common(fun_v, n_eigs=5, tau=-1.0)
    return l

```

## quantum/oscillator.py

### Description

missing description!

source code

```

from sfepy.linalg import norm_l2_along_axis

from quantum_common import common

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():
    l = common(fun_v, n_eigs=20, tau=0.0)
    return l

```

**quantum/quantum\_common.py****Description**

missing description!

source code

```
def common(fun_v, mesh='../tmp/mesh.vtk', n_eigs=5, tau=0.0):
    filename_mesh = mesh

    options = {
        'save_eig_vectors' : None,
        'n_eigs' : n_eigs,
        'eigen_solver' : 'eigen1',
    }

    # Whole domain $Y$.
    region_1000 = {
        'name' : 'Omega',
        'select' : 'all',
    }

    # Domain $Y_2$.
    region_2 = {
        'name' : 'Surface',
        'select' : 'nodes of surface',
    }

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

    material_1 = {
        'name' : 'm',

        'values' : {
            'val' : 0.5,
        },
    }

    material_2 = {
        'name' : 'mat_v',

        'function' : 'fun_v',
    }

    field_0 = {
        'name' : 'field_Psi',
        'dtype' : 'real',
        'shape' : 'scalar',
        'region' : 'Omega',
        'approx_order' : 1,
    }

    integral_1 = {
        'name' : 'i1',
        'kind' : 'v',
        'order' : 2,
```

```

    }

    variable_1 = {
        'name' : 'Psi',
        'kind' : 'unknown field',
        'field' : 'field_Psi',
        'order' : 0,
    }
    variable_2 = {
        'name' : 'v',
        'kind' : 'test field',
        'field' : 'field_Psi',
        'dual' : 'Psi',
    }
    variable_3 = {
        'name' : 'V',
        'kind' : 'parameter field',
        'field' : 'field_Psi',
        'like' : 'Psi',
    }

    ebc_1 = {
        'name' : 'ZeroSurface',
        'region' : 'Surface',
        'dofs' : {'Psi.0' : 0.0},
    }

    equations = {
        'lhs' : """ dw_laplace.il.Omega( m.val, v, Psi )
                    + dw_volume_dot.il.Omega( mat_v.V, v, Psi )""",
        'rhs' : """dw_volume_dot.il.Omega( v, Psi )""",
    }

    solver_2 = {
        'name' : 'eigen1',
        'kind' : 'eig.pysparse',

        'tau' : tau,
        'eps_a' : 1e-10,
        'i_max' : 150,
        'method' : 'qmrs',
        'verbosity' : 0,
        'strategy' : 1,
    }

    return locals()

```

## quantum/well.py

### Description

missing description!

source code

```

from quantum_common import common

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():
    l = common(fun_v, n_eigs=10, tau=0.0)
    return l
```

## 6.1.12 standalone/elastic\_materials examples

### standalone/elastic\_materials/compare\_elastic\_materials.py

#### Description

Compare various elastic materials w.r.t. uniaxial tension/compression test.

Requires Matplotlib.

source code

```
"""
Compare various elastic materials w.r.t. uniaxial tension/compression test.

Requires Matplotlib.
"""
from optparse import OptionParser
import sys
sys.path.append( '.' )

import numpy as nm

def define():
    """Define the problem to solve."""

    filename_mesh = 'el3.mesh'

    options = {
        'nls' : 'newton',
        'ls' : 'ls',
        'ts' : 'ts',
        'save_steps' : -1,
    }

    functions = {
        'linear_tension' : (linear_tension,),
        'linear_compression' : (linear_compression,),
        'empty' : (lambda ts, coor, mode, region, ig: None,),
    }

    field_1 = {
        'name' : 'displacement',
```

```

        'dtype' : nm.float64,
        'shape' : (3,),
        'region' : 'Omega',
        'approx_order' : 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
    material_1 = {
        'name' : 'solid',

        'values' : {
            '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
            'lam' : 5.769, # Lamé coefficients for LE term
            'mu_le' : 3.846,
        }
    }

    material_2 = {
        'name' : 'load',
        'function' : 'empty'
    }

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

    regions = {
        'Omega' : ('all', {}),
        'Bottom' : ('nodes in (z < 0.1)', {}),
        'Top' : ('nodes in (z > 2.9)', {}),
    }

    ebcs = {
        'fixb' : ('Bottom', {'u.all' : 0.0}),
        'fixt' : ('Top', {'u.[0,1]' : 0.0}),
    }

    ##
    # Balance of forces.
    integral_1 = {
        'name' : 'il',
        'kind' : 'v',
        'order' : 1,
    }
    integral_3 = {
        'name' : 'isurf',
        'kind' : 's',
        'order' : 2,
    }
    equations = {
        'linear' : """dw_lin_elastic_iso.il.Omega( solid.lam, solid.mu_le, v, u )
                    = dw_surface_ltr.isurf.Top( load.val, v )""",
    }

```

```
    'neoHookean' : """dw_tl_he_neohook.il.Omega( solid.mu_nh, v, u )
                    + dw_tl_bulk_penalty.il.Omega( solid.K, v, u )
                    = dw_surface_ltr.isurf.Top( load.val, v )""",
    'Mooney-Rivlin' : """dw_tl_he_neohook.il.Omega( solid.mu_mr, v, u )
                        + dw_tl_he_mooney_rivlin.il.Omega( solid.kappa, v, u )
                        + dw_tl_bulk_penalty.il.Omega( solid.K, v, u )
                        = dw_surface_ltr.isurf.Top( load.val, v )""",
}

##
# 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,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

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

    't0'    : 0,
    't1'    : 1,
    'dt'     : None,
    'n_step' : 101, # has precedence over dt!
}

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 problem.conf.equations.iteritems():
        problem.set_equations( {key : eq} )

        load = problem.get_materials()['load']
        load.set_function(branch_function)

        time_solver = problem.get_time_solver()

        out = []
        time_solver(save_results=False, step_hook=store_top_u(out))
        displacements[key] = nm.array(out, dtype=nm.float64)

    return displacements

usage = """%prog [options]"""
helps = {
    'no_plot' : 'do not show plot window',
}

def main():
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.fem import ProblemDefinition
    from sfepy.base.plotutils import plt

    parser = OptionParser(usage=usage, version='%prog')
    parser.add_option('-n', '--no-plot',
                    action="store_true", dest='no_plot',
                    default=False, help=helps['no_plot'])
    options, args = 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 = ProblemDefinition.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:
    print 'matplotlib cannot be imported, printing raw data!'
    print displacements
    print load
else:
    legend = []
    for key, val in displacements.iteritems():
        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()
```

## 6.1.13 standalone/homogenized\_elasticity examples

### standalone/homogenized\_elasticity/rs\_correctors.py

#### Description

missing description!

source code

```
# c: 05.05.2008, r: 05.05.2008
from optparse import OptionParser
import sys
sys.path.append( '.' )

import numpy as nm

import sfepy.fem.periodic as per
from sfepy.homogenization.utils import define_box_regions
```



```

# c: 05.05.2008, r: 05.05.2008
def define_regions( filename ):
    """Define various subdomain for a given mesh file. This function is called
    below."""
    regions = {}
    dim = 2

    regions['Y'] = ('all', {})

    eog = 'elements 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'] = ( ' +e '.join( eog % im for im in mat_ids ), {} )
        wx = 0.865
        wy = 0.499

    regions['Yc'] = ('r.Y -e r.Ym', {})

    # Sides and corners.
    regions.update( define_box_regions( 2, (wx, wy) ) )
    return dim, regions, mat_ids

def get_pars(ts, coor, mode=None, term=None, group_indx=None,
            mat_ids = [], **kwargs):
    """Define material parameters:
        $D_{ijkl}$ (elasticity),
        in a given region."""
    if mode == 'qp':
        dim = coor.shape[1]
        sym = (dim + 1) * dim / 2

        m2i = term.region.domain.mat_ids_to_i_gs
        matrix_igs = [m2i[im] for im in mat_ids]

        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 out.iteritems():
            out[key] = nm.tile(val, (coor.shape[0], 1, 1))

        for ig, indx in group_indx.iteritems():
            if ig not in matrix_igs: # channels
                out['D'][indx] *= 1e-1

    return out

##
# Mesh file.
filename_mesh = 'meshes/osteonT1_11.mesh'

##
# Define regions (subdomains, boundaries) - $Y$, $Y_i$, ...

```

```
# depending on a mesh used.
dim, regions, mat_ids = define_regions( filename_mesh )

functions = {
    'get_pars' : (lambda ts, coors, **kwargs:
        get_pars(ts, coors, mat_ids=mat_ids, **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 $Y_m$.
field_1 = {
    'name' : 'displacement',
    'dtype' : nm.float64,
    'shape' : dim,
    'region' : 'Y',
    'approx_order' : 1,
}

##
# Define corrector variables: unknown displacements: uc, test: vc
# displacement-like variables: Pi, Pi1, Pi2
variables = {
    'uc' : ('unknown field', 'displacement', 0),
    'vc' : ('test field', 'displacement', 'uc'),
    'Pi' : ('parameter field', 'displacement', 'uc'),
    'Pi1' : ('parameter field', 'displacement', None),
    'Pi2' : ('parameter field', 'displacement', None),
}

##
# Periodic boundary conditions.
if dim == 3:
    epbc_10 = {
        'name' : 'periodic_x',
        'region' : ['Left', 'Right'],
        'dofs' : {'uc.all' : 'uc.all'},
        'match' : 'match_x_plane',
    }
    epbc_11 = {
        'name' : 'periodic_y',
        'region' : ['Near', 'Far'],
        'dofs' : {'uc.all' : 'uc.all'},
        'match' : 'match_y_plane',
    }
    epbc_12 = {
        'name' : 'periodic_z',
        'region' : ['Top', 'Bottom'],
        'dofs' : {'uc.all' : 'uc.all'},
        'match' : 'match_z_plane',
    }
else:
    epbc_10 = {
```

```

        'name' : 'periodic_x',
        'region' : ['Left', 'Right'],
        'dofs' : {'uc.all' : 'uc.all'},
        'match' : 'match_y_line',
    }
    epbc_l1 = {
        'name' : 'periodic_y',
        'region' : ['Top', 'Bottom'],
        'dofs' : {'uc.all' : 'uc.all'},
        'match' : 'match_x_line',
    }

##
# Dirichlet boundary conditions.
ebcs = {
    'fixed_u' : ('Corners', {'uc.all' : 0.0}),
}

##
# Material defining constitutive parameters of the microproblem.
material_1 = {
    'name' : 'm',
    'function' : 'get_pars',
}

##
# Numerical quadratures for volume (i3 - order 3) integral terms.
integral_1 = {
    'name' : 'i3',
    'kind' : 'v',
    'order' : 3,
}

##
# Homogenized coefficients to compute.
def set_elastic(variables, ir, ic, mode, pis, corrs_rs):
    mode2var = {'row' : 'Pi1', 'col' : 'Pi2'}

    val = pis.states[ir, ic]['uc'] + corrs_rs.states[ir, ic]['uc']

    variables[mode2var[mode]].set_data(val)

coefs = {
    'E' : {
        'requires' : ['pis', 'corrs_rs'],
        'expression' : 'dw_lin_elastic.i3.Y( m.D, Pi1, Pi2 )',
        'set_variables' : set_elastic,
    },
}

all_periodic = ['periodic_%s' % ii for ii in ['x', 'y', 'z'][:dim] ]
requirements = {
    'pis' : {
        'variables' : ['uc'],
    },
}
##
# Steady state correctors  $\bar{\omega}^{rs}$ .
'corrs_rs' : {

```

```
        'requires' : ['pis'],
        'save_variables' : ['uc'],
        'ebcs' : ['fixed_u'],
        'epbcs' : all_periodic,
        'equations' : {'eq' : """dw_lin_elastic.i3.Y( m.D, vc, uc )
                        = - dw_lin_elastic.i3.Y( m.D, vc, Pi )"""},
        'set_variables' : [('Pi', 'pis', 'uc')],
        'save_name' : 'corrs_elastic',
        'is_linear' : True,
    },
}

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

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'      : 0.99999,
    'ls_min'     : 1e-5,
    'check'      : 0,
    'delta'      : 1e-6,
    'is_plot'    : False,
    'problem'    : 'nonlinear', # 'nonlinear' or 'linear' (ignore i_max)
}

#####
# Mini-application below, computing the homogenized elastic coefficients.
usage = """%prog [options]"""
help = {
    'no_pauses' : 'do not make pauses',
}

##
# c: 05.05.2008, r: 28.11.2008
def main():
    import os
    from sfepy.base.base import spause, output
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.fem import ProblemDefinition
    import sfepy.homogenization.coefs_base as cb

    parser = OptionParser(usage=usage, version='%prog')
    parser.add_option('-n', '--no-pauses',
                    action="store_true", dest='no_pauses',
                    default=False, help=help['no_pauses'])
```

```

options, args = 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 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 ProblemDefinition instance.
['q'/other key to quit/continue...]""" )
problem = ProblemDefinition.from_conf(conf, init_equations=False)
# The homogenization mini-apps need the output_dir.
output_dir = os.path.join(os.path.split(__file__)[0], 'output')
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
problem.output_dir = output_dir
print problem
spause( r""">>>
...the ProblemDefinition instance.
['q'/other key to quit/continue...]""" )

spause( r""">>>
The homogenized elastic coefficient  $E_{ijkl}$  is expressed
using  $\Pi$  operators, computed now. In fact, those operators are permuted
coordinates of the mesh nodes.
['q'/other key to quit/continue...]""" )
req = conf.requirements['pis']
mini_app = cb.ShapeDimDim( 'pis', problem, req )
pis = mini_app()
print pis
spause( r""">>>
...the  $\Pi$  operators.
['q'/other key to quit/continue...]""" )

spause( r""">>>
Next,  $E_{ijkl}$  needs so called steady state correctors  $\bar{\omega}^r$ ,
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_format='vtk',
                      file_per_var=False)
corrs_rs = mini_app( data = {'pis': pis} )
print corrs_rs
spause( r""">>>
...the  $\bar{\omega}^{\text{rs}}$  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()
```

## 6.1.14 standalone/interactive examples

### standalone/interactive/linear\_elasticity.py

#### Description

missing description!

source code

```
#!/usr/bin/env python
from optparse import OptionParser
import numpy as nm

import sys
sys.path.append('.')

from sfepy.base.base import IndexedStruct
```

```

from sfepy.fem import (Mesh, Domain, H1NodalVolumeField, FieldVariable,
                        Material, Integral, Function, Equation, Equations,
                        ProblemDefinition)
from sfepy.terms import Term
from sfepy.fem.conditions import Conditions, EssentialBC
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.postprocess import Viewer

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

usage = """%prog [options]"""
help = {
    'show' : 'show the results figure',
}

def main():
    from sfepy import data_dir

    parser = OptionParser(usage=usage, version='%prog')
    parser.add_option('-s', '--show',
                      action="store_true", dest='show',
                      default=False, help=help['show'])
    options, args = parser.parse_args()

    mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
    domain = Domain('domain', mesh)

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

    field = H1NodalVolumeField('fu', nm.float64, 'vector', omega,
                                approx_order=2)

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

    m = Material('m', 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_iso(m.lam, m.mu, v, u)',
                  integral, omega, m=m, v=v, u=u)
    t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
    eq = Equation('balance', t1 + t2)
    eqs = Equations([eq])

```

```
fix_u = EssentialBC('fix_u', gamma1, {'u.all' : 0.0})

bc_fun = Function('shift_u_fun', shift_u_fun, extra_args={'shift' : 0.01})
shift_u = EssentialBC('shift_u', gamma2, {'u.0' : bc_fun})

ls = ScipyDirect({})

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

pb = ProblemDefinition('elasticity', equations=eqs, nls=nls, ls=ls)
pb.save_regions_as_groups('regions')

pb.time_update(ebcs=Conditions([fix_u, shift_u]))

vec = pb.solve()
print nls_status

pb.save_state('linear_elasticity.vtk', vec)

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()
```

## 6.1.15 standalone/live\_plot examples

### standalone/live\_plot/live\_plot.py

#### Description

missing description!

source code

```
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)', 'cos(x)', 'exp(x)'],
              yscales=['linear', 'log'],
              xlabel='angle', None, 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'],
```



```

        log_filename=os.path.join(cwd, 'live_plot2.log'))

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), nm.cos(x), nm.exp(x), x = [x, None])

    else:
        if added:
            log(nm.sin(x), 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'], 'linear', 'new x', '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()

```

## 6.1.16 standalone/thermal\_electric examples

### standalone/thermal\_electric/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
```

```
"""
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', {}),
    'Omega1' : ('elements of group 1', {}),
    'Omega2' : ('elements of group 2', {}),
    'Omega2_Surface' : ('r.Omega1 *n r.Omega2', {'can_cells' : False}),
    'Left' : ('nodes in (x < %f)' % -0.4999, {}),
    'Right' : ('nodes in (x > %f)' % 0.4999, {}),
}

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!
    }),
}

def main():
    from sfepy.base.base import output
    from sfepy.base.conf import ProblemConf, get_standard_keywords
    from sfepy.fem import ProblemDefinition

    output.prefix = 'therel:'

    required, other = get_standard_keywords()
    conf = ProblemConf.from_file(__file__, required, other)

    problem = ProblemDefinition.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']})
problem.time_update()
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())
time_solver = problem.get_time_solver()
time_solver()

output('results saved in %s' % problem.get_output_name(suffix = '*'))

if __name__ == '__main__':
    main()
```

## 6.1.17 thermo\_elasticity examples

### thermo\_elasticity/thermo\_elasticity.py

#### Description

Thermo-elasticity with a given temperature distribution.

Uses  $dw_{biot}$  term with an isotropic coefficient for thermo-elastic coupling.

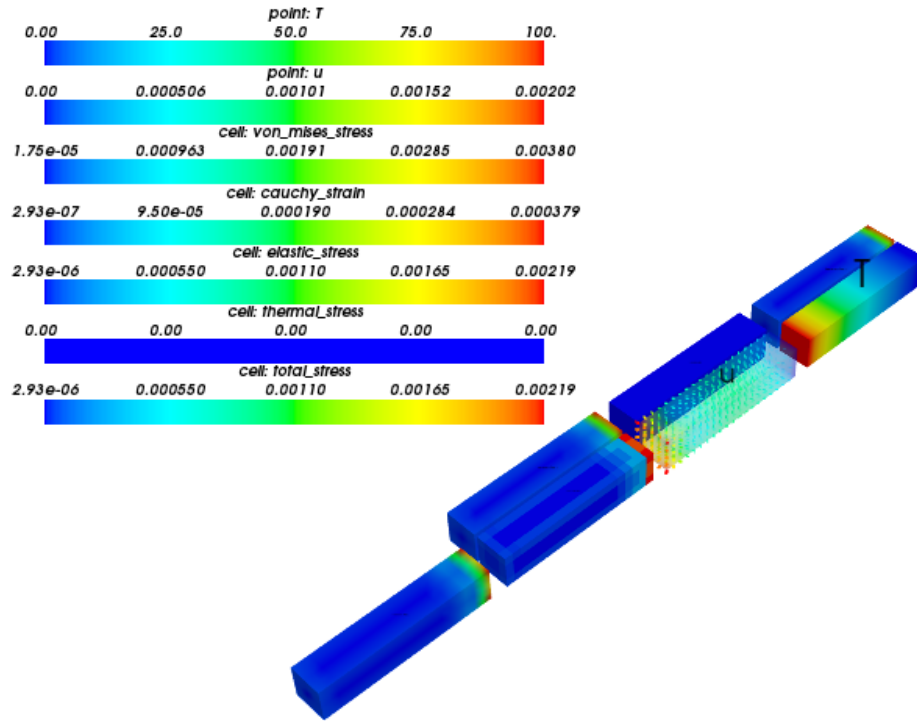
For given body temperature  $T$  and background temperature  $T_0$  find  $\underline{u}$  such that:

$$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) - \int_{\Omega} (T - T_0) \alpha_{ij} e_{ij}(\underline{v}) = 0, \quad \forall \underline{v},$$

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 \text{forall } \ul{v} \;,

where

.. math::
    D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) +
    \lambda \delta_{ij} \delta_{kl}
    \;, \quad
    \alpha_{ij} = (3 \lambda + 2 \mu) \alpha \delta_{ij}

and :math:'\alpha' is the thermal expansion coefficient.
"""

```

```
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' : ('nodes in (x < -4.99)', {}),
}

fields = {
    'displacement': ('real', 3, 'Omega', 1),
    'temperature': ('real', 1, 'Omega', 1),
}

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
    'T' : ('parameter field', 'temperature',
           {'setter' : 'get_temperature_load'}),
}

ebcs = {
    'fix_u' : ('Left', {'u.all' : 0.0}),
}

eye_sym = np.array([[1], [1], [1], [0], [0], [0]], dtype=np.float64)
materials = {
    'solid' : ({
        'D' : stiffness_from_lame(3, lam=lam, mu=mu),
        'alpha' : (3.0 * lam + 2.0 * mu) * thermal_expandability * eye_sym
    },),
}

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,
        'problem' : 'nonlinear',
    }),
}

```





# DEVELOPER GUIDE

**Table of Contents**

- SfePy Directory Structure
- How to Contribute
  - Reporting problems
  - Making changes
  - Coding style
  - Contributing changes
    - \* Without git
    - \* With git
    - \* Notes on commits and patches
    - \* 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
  - Example
  - Concluding remarks
- How To Make a Release
- Working with *SfePy* source code
- Module Index
  - Main scripts
  - Utility scripts
  - sfepy package
  - sfepy.applications package
  - sfepy.base package
  - sfepy.fem package
  - sfepy.homogenization package
  - sfepy.interactive package
  - sfepy.linalg package
  - sfepy.mechanics package
  - sfepy.mesh package
  - sfepy.optimize package
  - sfepy.physics 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.

## 7.1 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 convertors etc.)
<i>sfepy/</i>	the source code
<i>tests/</i>	the tests run by <i>runTests.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>applications/</i>	top level application classes (e.g. <code>PDESolverApp</code> 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>fem/</i>	the finite element core: modules taking care of boundary conditions, degrees of freedom, approximations, variables, equations, meshes, regions, quadratures, etc.	
<i>mesh/</i>	some utilities to interface with tetgen and triangle mesh generators	
<i>homogenization/</i>	the homogenization engine and supporting modules - highly specialized code, one of the reasons of <i>SfePy</i> existence	•
<i>interactive/</i>	setup of IPython-based shell <i>isfepy</i>	
<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>physics/</i>	small utilities for quantum physics ( <i>schroedinger.py</i> )	•
<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 <a href="#">Term Overview</a>	

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.

## 7.2 How to Contribute

Read this section if you wish to contribute some work to the *SfePy* project. 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 or find current problems, issues, and suggestions of particular topics for additional development
2. what to do to apply changes/fixes

3. what to do after you made your changes/fixes

## 7.2.1 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](#) 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](#) web page. 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 and post comments to issues. It is, however, not needed to create a new issue.

**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.

Our persisting all star top priority issues include:

- missing docstrings in many functions/classes/modules
- incomplete documentation
- lowering the barrier for new users
  - e.g., through generation of additional tutorial material

So 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.

## 7.2.2 Making changes

This step is simple, just keep in mind to use the latest development version of the code from the [SfePy git repository](#) page.

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 nice git patch than in another form.

Having said that, to download the latest snapshot, do either (with git):

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

or (without git):

- use the [SfePy tarball](#) link

Then make the changes as you wish, following our *Coding style*.

**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.

### 7.2.3 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.

### 7.2.4 Contributing changes

Even if you do not use git, try to follow the spirit of *Notes on commits and patches*

#### Without 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.

#### With git

Contents:

##### Introduction

These pages describe a [git](#) and [github](#) workflow for the [SfePy](#) project.

There are several different workflows here, for different ways of working with *SfePy*.

This is not a comprehensive git reference, it’s just a workflow for our own project. It’s tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see [git resources](#).

## Install git

Overview	Debian / Ubuntu	<code>sudo apt-get install git-core</code>
	Fedora	<code>sudo yum install git-core</code>
	Windows	Download and install <a href="#">msysGit</a>
	OS X	Use the <a href="#">git-osx-installer</a>

**In detail** See the git page for the most recent information.

Have a look at the github install help pages available from [github help](#)

There are good instructions here: [http://book.git-scm.com/2\\_installing\\_git.html](http://book.git-scm.com/2_installing_git.html)

## Following the latest source

These are the instructions if you just want to follow the latest *SfePy* source, but you don't need to do any development for now.

The steps are:

- [Install git](#)
- get local copy of the [SfePy github](#) git repository
- update local copy from time to time

**Get the local copy of the code** From the command line:

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

You now have a copy of the code tree in the new `sfepy` directory.

**Updating the code** From time to time you may want to pull down the latest code. Do this with:

```
cd sfepy
git pull
```

The tree in `sfepy` will now have the latest changes from the initial repository.

## Making a patch

You've discovered a bug or something else you want to change in [SfePy](#) .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the [Git for development](#) model instead.

## Making patches

### Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/sfepy/sfepy.git
# make a branch for your patching
cd sfepy
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the [SfePy mailing list](#) — where we will thank you warmly.

### In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the [SfePy](#) repository:

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

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#).

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:



```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the [SfePy mailing list](#).

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

**Moving from patching to development** If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the [SfePy](#) repository on github — *Making your own copy (fork) of SfePy*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/sfepy.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the [Development workflow](#).

## Git for development

Contents:

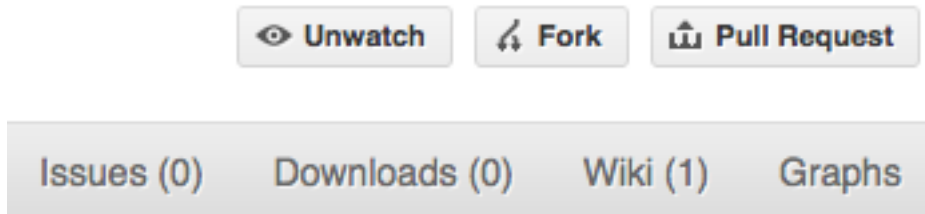
**Making your own copy (fork) of SfePy** You need to do this only once. The instructions here are very similar to the instructions at <http://help.github.com/forking/> — please see that page for more detail. We're repeating some of it here just to give the specifics for the [SfePy](#) project, and to suggest some default names.

**Set up and configure a github account** If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the [Generating SSH keys help on github help](#).

### Create your own forked copy of SfePy

1. Log into your github account.
2. Go to the [SfePy](#) github home at [SfePy github](#).
3. Click on the *fork* button:



Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of [SfePy](#).

**Set up your fork** First you follow the instructions for *Making your own copy (fork) of SfePy*.

### Overview

```
git clone git@github.com:your-user-name/sfepy.git
cd sfepy
git remote add upstream git://github.com/sfepy/sfepy.git
```

### In detail

#### Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/sfepy.git`
2. Investigate. Change directory to your new repo: `cd sfepy`. Then `git branch -a` to show you all branches. You’ll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the `master` branch, and that you also have a remote connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream [SfePy github](#) repository, so you can merge in changes from trunk.

#### Linking your repository to the upstream repo

```
cd sfepy
git remote add upstream git://github.com/sfepy/sfepy.git
```

`upstream` here is just the arbitrary name we’re using to refer to the main [SfePy](#) repository at [SfePy github](#).

Note that we’ve used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can’t accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new ‘remote’, with `git remote -v` show, giving you something like:

```
upstream    git://github.com/sfepy/sfepy.git (fetch)
upstream    git://github.com/sfepy/sfepy.git (push)
origin      git@github.com:your-user-name/sfepy.git (fetch)
origin      git@github.com:your-user-name/sfepy.git (push)
```

## Configure git

**Overview** Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com

[alias]
    ci = commit -a
    co = checkout
    st = status
    stat = status
    br = branch
    wdiff = diff --color-words

[core]
    editor = vim

[merge]
    summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

## In detail

**user.name and user.email** It is good practice to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

**Aliases** You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an alias section in your `.gitconfig` file with contents like this:

```
[alias]
    ci = commit -a
    co = checkout
    st = status -a
    stat = status -a
    br = branch
    wdiff = diff --color-words
```

**Editor** You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

**Merging** To enforce summaries when doing merges (~/.gitconfig file again):

```
[merge]
    log = true
```

Or from the command line:

```
git config --global merge.log true
```

**Fancy log output** This is a very nice alias to get a fancy log output; it should go in the alias section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)[%an
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Matthias]
* d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks ago) [Hugo]
| \
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
| /
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran Wootton]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a class method
* 376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
| \
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
```

```
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| * 956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| | \
| | /
```

Thanks to Yury V. Zaytsev for posting it.

**Development workflow** You already have your own forked copy of the [SfePy](#) repository, by following [Making your own copy \(fork\) of SfePy](#). You have [Set up your fork](#). You have configured git by following [Configure git](#). Now you are ready for some real work.

**Workflow summary** In what follows we'll refer to the upstream SfePy master branch, as “trunk”.

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — “one task, one branch” ([ipython git workflow](#)).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider [Rebasing on trunk](#)
- Ask on the [SfePy mailing list](#) if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See [linux git workflow](#) and [ipython git workflow](#) for some explanation.

**Consider deleting your master branch** It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See [deleting master on github](#) for details.

**Update the mirror of trunk** First make sure you have done [Linking your repository to the upstream repo](#).

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, ‘trunk’ is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

**Make a new feature branch** When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called ‘feature branches’.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public [github](#) fork of [SfePy](#). To do this, you `git push` this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git  $\geq$  1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

## The editing workflow

### Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

### In more detail

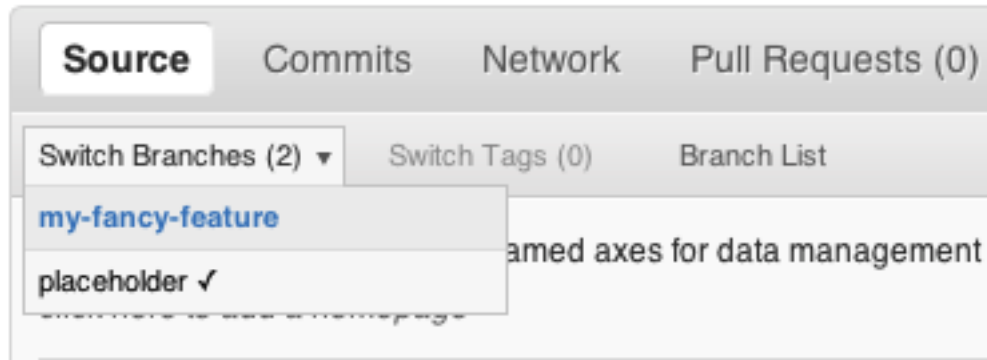
1. Make some changes
2. See which files have changed with `git status` (see [git status](#)). You'll see a listing like this one:

```
# On branch my-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

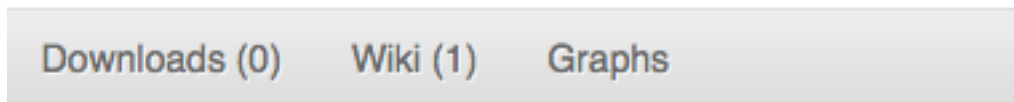
3. Check what the actual changes are with `git diff` ([git diff](#)).
4. Add any new files to version control `git add new_file_name` (see [git add](#)).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#) — and the helpful use-case description in the [tangled working copy problem](#). The [git commit](#) manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see [git push](#)).

**Ask for your changes to be reviewed or merged** When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/sfepy`.
2. Use the ‘Switch Branches’ dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the ‘Pull request’ button:



Enter a title for the set of changes, and some explanation of what you’ve done. Say if there is anything you’d like particular attention for - like a complicated change or some code you are not happy with.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

## Some other things you might want to do

### Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

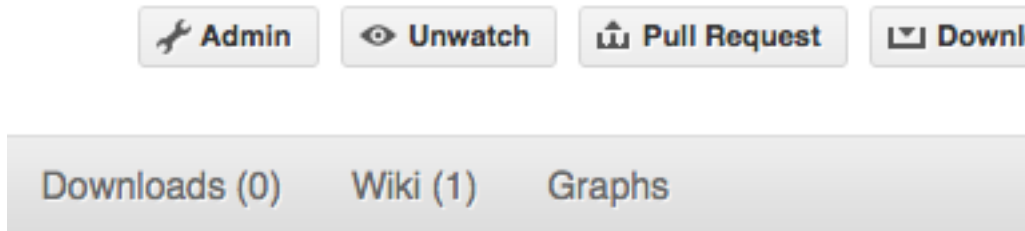
(Note the colon `:` before `test-branch`. See also: <http://github.com/guides/remove-a-remote-branch>)

**Several people sharing a single repository** If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork SfePy into your account, as from *Making your own copy (fork) of SfePy*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/sfepy`

Click on the ‘Admin’ button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@github.com:your-user-name/sfepy.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

**Explore your repository** To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the [network graph visualizer](#) for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

**Rebasing on trunk** Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
      /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
      A'--B'--C' cool-feature
      /
D---E---F---G trunk
```

See [rebase without tears](#) for more detail.

To do a rebase on trunk:



```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at [Recovering from mess-ups](#).

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the [git rebase](#) man page for some instructions at the end of the “Description” section. There is some related help on merging in the git user manual - see [resolving a merge](#).

**Recovering from mess-ups** Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

---

## Rewriting commit history

**Note:** Do this only for your own feature branches.

---

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2declac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and 6ad92e5 is the last commit in the cool-feature branch. Suppose we want to make the following changes:

- Rewrite the commit message for 13d7934 to something more sensible.
- Combine the commits 2declac, a815645, eadc391 into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2declac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2declac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for 13d7934, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained [above](#).

**Maintainer workflow** This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in [Development workflow](#).

The instructions in [Linking your repository to the upstream repo](#) add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:sfepy/sfepy.git
git fetch upstream-rw
```

**Integrating changes** Let's say you have some changes that need to go into trunk (upstream-rw/master).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/sfepy.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

**A few commits** If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

**A long series of commits** If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

**Check the history** Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

### Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

### git resources

#### Tutorials and summaries

- [github help](#) has an excellent series of how-to guides.
- [learn.github](#) has an excellent series of tutorials
- The [pro git book](#) is a good in-depth book on git.
- A [git cheat sheet](#) is a page giving summaries of common commands.
- The [git user manual](#)
- The [git tutorial](#)
- The [git community book](#)
- [git ready](#) — a nice series of tutorials
- [git casts](#) — video snippets giving git how-tos.
- [git magic](#) — extended introduction with intermediate detail
- The [git parable](#) is an easy read explaining the concepts behind git.
- [git foundation](#) expands on the [git parable](#).
- Fernando Perez' [git page](#) — [Fernando's git page](#) — many links and tips
- A good but technical page on [git concepts](#)
- [git svn crash course](#): git for those of us used to [subversion](#)

**Advanced git workflow** There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on [git management](#)
- Linus Torvalds on [linux git workflow](#) . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

**Manual pages online** You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- [git add](#)
- [git branch](#)
- [git checkout](#)
- [git clone](#)
- [git commit](#)
- [git config](#)
- [git diff](#)
- [git log](#)
- [git pull](#)
- [git push](#)
- [git remote](#)
- [git status](#)

**Note:** This section will get quickly get you started using git and github. For more in-depth reading about how these tools work with the *SfePy* source code and the general git development, read [Working with SfePy source code](#), which was adapted from Matthew Brett's excellent [gitwash](#) git tutorial.

With git there are some additional options for how to send changes to *SfePy*. Before listing them, let us describe a typical development session and the related git commands:

1. Either clone a fresh copy by:

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

or update your local repository:

```
# look for changes at origin
git fetch origin

# difference between local and origin master branch
git diff master origin/master

# apply the changes to local master branch
git pull origin master
```

2. Introduce yourself to git and make (optionally) some handy aliases either in `.gitconfig` in your home directory (global setting for all your git projects), or directly in `.git/config` in the repository:

```
[user]
  email = mail@mail.org
  name = Name Surname

[color]
  ui = auto
  interactive = true

[alias]
  ci = commit
  di = diff --color-words
  st = status
  co = checkout
```

3. Change some file(s), and review the changes:

```
# text diff
git diff

# use GUI to visualize of project history (all branches)
gitk --all
```

4. Create one or more commits:

```
# schedule some of the changed files for the next commit
git add file1 file2 ...
# an editor will pop up where you should describe the commit
git commit
```

5. The commit(s) now reflect changes, but only in your *local* git repository. Then you must somehow allow others to see them. This can be done, for example, by sending a patch (or through the other option below). So create the patch(es):

```
# create patches for, e.g., the last two commits
git format-patch HEAD~2
```

6. Send the patch(es) to the [SfePy mailing list](#) or attach them to the corresponding issue at the [Issues](#) web page.
7. If the patches are fine, they will appear in the master repository. Then synchronize your repository with the master:
  - either clone a fresh copy
  - or use the fetch, pull, merge or rebase commands. This may require a deeper git-fu in case of conflicts. For beginners, it is advisable to clone always a fresh copy if they see a conflict.

There is another option than submitting patches, however, useful when you wish to get feedback on a larger set of changes. This option is to publish your repository using [Github](#) and let the other developers know about it - follow the instructions in *Git for development* of [Working with SfePy source code](#).

## 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 [tbagery.com](#), namely that the commit message is better to be written in the present tense: “fix bug” and not “fixed bug”.

## Docstring standard

We use [sphinx](#) with the [numpydoc](#) extension to generate this documentation. Refer to the sphinx site for the possible markup constructs.

Basically (with a little tweak), we try to follow the NumPy/SciPy docstring standard as described in [NumPy documentation guide](#). See also the complete [docstring example](#). It is exaggerated a bit to show all the possibilities. Use your

common sense here - the docstring should be sufficient for a new user to use the documented object. A good way to remember the format is to type:

```
In [1]: import numpy as nm
In [2]: nm.sin?
```

in *ipython*. The little tweak mentioned above is the starting newline:

```
1 def function(arg1, arg2):
2     """
3     This is a function.
4
5     Parameters
6     -----
7     arg1 : array
8         The coordinates of ...
9     arg2 : int
10        The dimension ...
11
12    Returns
13    -----
14    out : array
15        The resulting array of shape ....
16    """
```

It seems visually better than:

```
1 def function(arg1, arg2):
2     """This is a function.
3
4     Parameters
5     -----
6     arg1 : array
7         The coordinates of ...
8     arg2 : int
9         The dimension ...
10
11    Returns
12    -----
13    out : array
14        The resulting array of shape ....
15    """
```

When using  $\LaTeX$  in a docstring, use a raw string:

```
1 def function():
2     r"""
3     This is a function with :math: '\mbox{\LaTeX}' math:
4     :math: '\frac{1}{\pi}'.
5     """
```

to prevent Python from interpreting and consuming the backslashes in common escape sequences like `'\n'`, `'\f'` etc.

## 7.3 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
```

## 7.4 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.

### 7.4.1 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.

### 7.4.2 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.

Terms come in two flavors:

- standard terms are subclasses of `sfepy.terms.terms.Term`
- *new* terms are subclasses of `sfepy.terms.terms_new.NewTerm`

As new terms are now not much more than a highly experimental proof of concept, we will focus on the standard terms here.



The purpose of a standard term class is to implement a (vectorized) function that assembles the term contribution to residual/matrix and/or evaluates the term integral in a group of elements simultaneously. Most such functions are currently implemented in C, but some terms are pure Python, vectorized using NumPy. A term with a C function needs to be able to extract the real data from its arguments and then pass those data to the C function.

### 7.4.3 Evaluation modes

A term can support several evaluation modes, as described in *Term Evaluation*.

### 7.4.4 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  $(n_{el}, 1, n_{row}, n_{col})$ , where *n\_el* is the number of elements in a group and *n\_row*, *n\_col* are matrix dimensions of the value on a single element.

### ***get\_fargs()***

The *get\_fargs()* method has always the same structure of arguments:

- positional arguments corresponding to *arg\_types* attribute:

- example for a typical weak term:

```
* for:
```

```
arg_types = ('material', 'virtual', 'state')
```

the positional arguments are:

```
material, virtual, state
```

- keyword arguments common to all terms:

```
mode=None, term_mode=None, diff_var=None, **kwargs
```

here:

- *mode* is the actual evaluation mode, default is 'eval';
- *term\_mode* is an optional term sub-mode influencing what the term should return (example: *dw\_tl\_he\_neohook* term has 'strain' and 'stress' evaluation sub-modes);
- *diff\_var* is taken into account in the 'weak' evaluation mode. It is either *None* (residual mode) or a name of variable with respect to differentiate to (matrix mode);
- *\*\*kwargs* are any other arguments that the term supports.

The *get\_fargs()* method returns arguments for *function()*.

## **7.4.5 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.

### 7.4.6 Example

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 \, \text{or} \, \int_{\Omega} c \, q
10
11     :Arguments:
12         - material : :math: 'c' (optional)
13         - virtual  : :math: 'q'
14     """
15     name = 'dw_volume_integrate'
16     arg_types = ('opt_material', 'virtual')
17     arg_shapes = [{'opt_material' : '1, 1', 'virtual' : (1, None)},
```

```
18         {'opt_material' : None}]
19
20     @staticmethod
21     def function(out, material, bf, geo):
22         bf_t = nm.tile(bf.transpose((0, 1, 3, 2)), (out.shape[0], 1, 1, 1))
23         bf_t = nm.ascontiguousarray(bf_t)
24         if material is not None:
25             status = geo.integrate(out, material * bf_t)
26         else:
27             status = geo.integrate(out, bf_t)
28         return status
29
30     def get_fargs(self, material, virtual,
31                  mode=None, term_mode=None, diff_var=None, **kwargs):
32         assert_(virtual.n_components == 1)
33         geo, _ = self.get_mapping(virtual)
34
35         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 all elements of an element group,
    - \* 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.

### 7.4.7 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.

## 7.5 How To Make a Release

### 7.5.1 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:

```
$ script/gen_gallery.py -l ../doc-devel
```

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:

```
$ python setup.py install --root=<some path>
$ cd
$ runTests.py
$ 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
- 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
```

## Useful Git commands

- log

```
git log --pretty=format:"%s%n%b%n" release_2009.1..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 <http://code.google.com/p/sfepy/downloads/list>
  - make it featured, un-feature the previous release
  - update download link at <http://code.google.com/p/sfepy/wiki/Downloads>
- publish development docs also as new release docs
- send announcement to
  - [sfepy-devel@googlegroups.com](mailto:sfepy-devel@googlegroups.com), [scipy-dev@scipy.org](mailto:scipy-dev@scipy.org), [scipy-user@scipy.org](mailto:scipy-user@scipy.org), [python-announce-list@python.org](mailto:python-announce-list@python.org)

## 7.6 Working with *SfePy* source code

This section was adapted from Matthew Brett's excellent [gitwash](#) git tutorial. It complements the above sections and details several aspects of working with Git and Github.

It can be updated by running:

```
$ curl -O https://raw.github.com/matthew-brett/gitwash/master/gitwash_dumper.py
$ python gitwash_dumper.py doc/dev SfePy --repo-name=sfepy --github-user=sfepy --project-url=http://
```

in the SfePy source directory. Do not forget to delete the section title in *doc/dev/gitwash/index.rst*, as it is already here. Contents:

## 7.6.1 Introduction

These pages describe a [git](#) and [github](#) workflow for the SfePy project.

There are several different workflows here, for different ways of working with *SfePy*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see [git resources](#).

## 7.6.2 Install git

### Overview

Debian / Ubuntu	<code>sudo apt-get install git-core</code>
Fedora	<code>sudo yum install git-core</code>
Windows	Download and install <a href="#">msysGit</a>
OS X	Use the <a href="#">git-osx-installer</a>

### In detail

See the git page for the most recent information.

Have a look at the github install help pages available from [github help](#)

There are good instructions here: [http://book.git-scm.com/2\\_installing\\_git.html](http://book.git-scm.com/2_installing_git.html)

## 7.6.3 Following the latest source

These are the instructions if you just want to follow the latest *SfePy* source, but you don't need to do any development for now.

The steps are:

- [Install git](#)
- get local copy of the [SfePy github](#) git repository
- update local copy from time to time

### Get the local copy of the code

From the command line:

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

You now have a copy of the code tree in the new `sfepy` directory.

## Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd sfepy
git pull
```

The tree in `sfepy` will now have the latest changes from the initial repository.

## 7.6.4 Making a patch

You've discovered a bug or something else you want to change in [SfePy](#) .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the [Git for development](#) model instead.

## Making patches

### Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/sfepy/sfepy.git
# make a branch for your patching
cd sfepy
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the [SfePy mailing list](#) — where we will thank you warmly.

### In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the [SfePy](#) repository:



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

3. Make a ‘feature branch’. This will be where you work on your bug fix. It’s nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you’ve made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you’re going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#).

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the [SfePy mailing list](#).

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

## Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the [SfePy](#) repository on github — *Making your own copy (fork) of SfePy*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/sfepy.git
# push up any branches you’ve made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the [Development workflow](#).

## 7.6.5 Git for development

Contents:

### Making your own copy (fork) of SfePy

You need to do this only once. The instructions here are very similar to the instructions at <http://help.github.com/forking/> — please see that page for more detail. We’re repeating some of it here just to give the specifics for the SfePy project, and to suggest some default names.

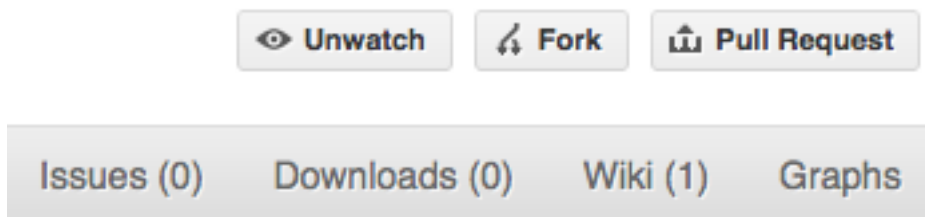
#### Set up and configure a github account

If you don’t have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the [Generating SSH keys help](#) on github help.

#### Create your own forked copy of SfePy

1. Log into your github account.
2. Go to the SfePy github home at [SfePy github](#).
3. Click on the *fork* button:



Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of SfePy.

#### Set up your fork

First you follow the instructions for *Making your own copy (fork) of SfePy*.

#### Overview

```
git clone git@github.com:your-user-name/sfepy.git
cd sfepy
git remote add upstream git://github.com/sfepy/sfepy.git
```

#### In detail

##### Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/sfepy.git`

- Investigate. Change directory to your new repo: `cd sfepy`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the master branch, and that you also have a remote connection to origin/master. What remote repository is remote/origin? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream [SfePy github](#) repository, so you can merge in changes from trunk.

### Linking your repository to the upstream repo

```
cd sfepy
git remote add upstream git://github.com/sfepy/sfepy.git
```

upstream here is just the arbitrary name we're using to refer to the main [SfePy](#) repository at [SfePy github](#).

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream      git://github.com/sfepy/sfepy.git (fetch)
upstream      git://github.com/sfepy/sfepy.git (push)
origin        git@github.com:your-user-name/sfepy.git (fetch)
origin        git@github.com:your-user-name/sfepy.git (push)
```

## Configure git

### Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com

[alias]
    ci = commit -a
    co = checkout
    st = status
    stat = status
    br = branch
    wdiff = diff --color-words

[core]
    editor = vim

[merge]
    summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

### In detail

**user.name and user.email** It is good practice to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
  name = Your Name
  email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

**Aliases** You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an alias section in your `.gitconfig` file with contents like this:

```
[alias]
  ci = commit -a
  co = checkout
  st = status -a
  stat = status -a
  br = branch
  wdiff = diff --color-words
```

**Editor** You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

**Merging** To enforce summaries when doing merges (~/.gitconfig file again):

```
[merge]
  log = true
```

Or from the command line:

```
git config --global merge.log true
```

**Fancy log output** This is a very nice alias to get a fancy log output; it should go in the alias section of your .gitconfig file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)%l'
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Matthias]
* d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks ago) [Hugo]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran Wootton]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a class method
* 376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| * 956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

## Development workflow

You already have your own forked copy of the [SfePy](#) repository, by following *Making your own copy (fork) of SfePy*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

### Workflow summary

In what follows we'll refer to the upstream SfePy master branch, as "trunk".

- Don't use your master branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.

- Make a new branch for each separable set of changes — “one task, one branch” ([ipython git workflow](#)).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the [SfePy mailing list](#) if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you’ve done, and why you did it.

See [linux git workflow](#) and [ipython git workflow](#) for some explanation.

### Consider deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See [deleting master on github](#) for details.

### Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don’t have, and set the remote branches to point to the right commit. For example, ‘trunk’ is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

### Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called ‘feature branches’.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public [github](#) fork of [SfePy](#). To do this, you `git push` this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

## The editing workflow

### Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

### In more detail

1. Make some changes
2. See which files have changed with `git status` (see [git status](#)). You'll see a listing like this one:

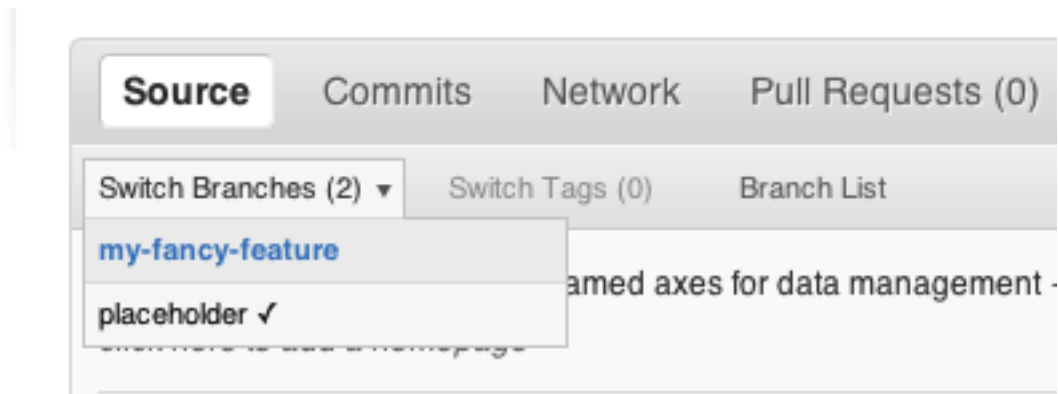
```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` ([git diff](#)).
4. Add any new files to version control `git add new_file_name` (see [git add](#)).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to commit. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see [why the -a flag?](#) — and the helpful use-case description in the [tangled working copy problem](#). The [git commit](#) manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see [git push](#)).

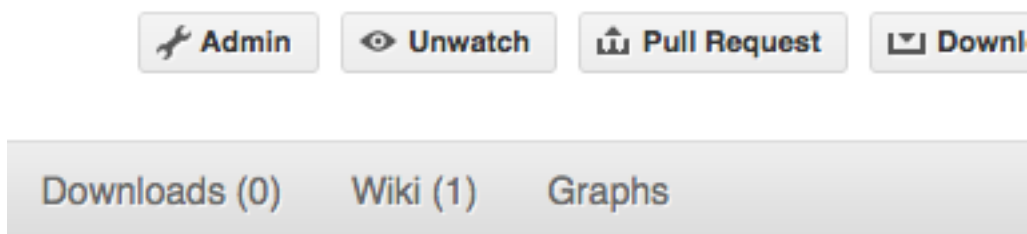
### Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say <http://github.com/your-user-name/sfepy>.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:



Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

### Some other things you might want to do

#### Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon : before test-branch. See also: <http://github.com/guides/remove-a-remote-branch>)

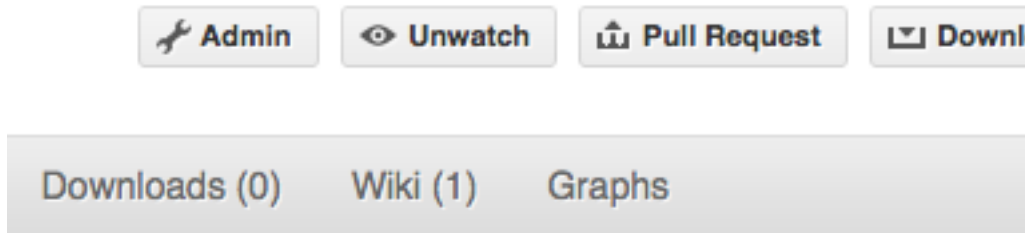
**Several people sharing a single repository** If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork SfePy into your account, as from *Making your own copy (fork) of SfePy*.

Then, go to your forked repository github page, say <http://github.com/your-user-name/sfepy>

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:





Now all those people can do:

```
git clone git@github.com:your-user-name/sfepy.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

**Explore your repository** To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the [network graph visualizer](#) for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

**Rebasing on trunk** Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```

      A---B---C cool-feature
      /
D---E---F---G trunk

```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```

      A'--B'--C' cool-feature
      /
D---E---F---G trunk

```

See [rebase without tears](#) for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at [Recovering from mess-ups](#).

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the [git rebase](#) man page for some instructions at the end of the “Description” section. There is some related help on merging in the git user manual - see [resolving a merge](#).

**Recovering from mess-ups** Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature
```

```
8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...
```

```
# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

---

### Rewriting commit history

**Note:** Do this only for your own feature branches.

---

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2declac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and 6ad92e5 is the last commit in the cool-feature branch. Suppose we want to make the following changes:

- Rewrite the commit message for 13d7934 to something more sensible.
- Combine the commits 2declac, a815645, eadc391 into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2declac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2declac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for 13d7934, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained [above](#).

### Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in [Development workflow](#).

The instructions in [Linking your repository to the upstream repo](#) add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:sfepy/sfepy.git
git fetch upstream-rw
```

### Integrating changes

Let's say you have some changes that need to go into trunk (upstream-rw/master).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/sfepy.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

**A few commits** If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

**A long series of commits** If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

**Check the history** Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

### Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

## 7.6.6 git resources

### Tutorials and summaries

- [github help](#) has an excellent series of how-to guides.
- [learn.github](#) has an excellent series of tutorials
- The [pro git book](#) is a good in-depth book on git.
- A [git cheat sheet](#) is a page giving summaries of common commands.
- The [git user manual](#)
- The [git tutorial](#)
- The [git community book](#)
- [git ready](#) — a nice series of tutorials
- [git casts](#) — video snippets giving git how-tos.
- [git magic](#) — extended introduction with intermediate detail
- The [git parable](#) is an easy read explaining the concepts behind git.
- [git foundation](#) expands on the [git parable](#).
- Fernando Perez' git page — [Fernando's git page](#) — many links and tips
- A good but technical page on [git concepts](#)
- [git svn crash course](#): git for those of us used to [subversion](#)

### Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on [git management](#)
- Linus Torvalds on [linux git workflow](#) . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

## Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- `git add`
- `git branch`
- `git checkout`
- `git clone`
- `git commit`
- `git config`
- `git diff`
- `git log`
- `git pull`
- `git push`
- `git remote`
- `git status`

## 7.7 Module Index

### 7.7.1 Main scripts

#### extractor.py script

##### 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 `runTests.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"
```

```
postproc.main()
postproc.parse_domain_specific(option, opt, value, parser)
postproc.parse_group_names(option, opt, value, parser)
postproc.parse_opacity(option, opt, value, parser)
postproc.parse_ranges(option, opt, value, parser)
postproc.parse_resolution(option, opt, value, parser)
postproc.parse_subdomains(option, opt, value, parser)
postproc.parse_view(option, opt, value, parser)
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

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

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.
```

## runTests.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 runTests.OutputFilter(allowed_lines)
```

```
    start()
    stop()
    write(msg)

runTests.get_dir(default)
runTests.main()
runTests.run_test(conf_name, options)
```



```
runTests.wrap_run_tests(options)
```

### **schroedinger.py script**

Electronic structure solver.

Type:

```
$ ./schroedinger.py
```

for usage and help.

```
schroedinger.fix_path(filename)
```

```
schroedinger.main()
```

### **shaper.py script**

```
shaper.main()
```

```
shaper.solve_adjoint(conf, options, dpb, state_dp, data)
```

Solve the adjoint (linear) problem.

```
shaper.solve_direct(conf, options)
```

Solve the direct (nonlinear) problem.

```
shaper.solve_generic_direct(conf, options)
```

```
shaper.solve_navier_stokes(conf, options)
```

```
shaper.solve_optimize(conf, options)
```

```
shaper.solve_stokes(dpb, equations_stokes, nls_conf)
```

### **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_terms()
```

## **7.7.2 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 -
    inplace.

    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 distu-
    tils.version.LooseVersion at 0x5afb808>, version_getter=None, mes-
    sages=None)
    Check if package pkg_name is present, and correct version

    Parameters  pkg_name : str or sequence of str
        name of package as imported into python. Alternative names (e.g. for different versions)
        may be given in a list.

    version : {None, str}, optional
        minimum version of the package that we require. If None, we don't check the version.
        Default is None

    optional : {False, True}, optional
```

If False, raise error for absent package or wrong version; otherwise warn

**checker** : callable, optional

callable with which to return comparable thing from version string. Default is `distutils.version.LooseVersion`

**version\_getter** : {None, callable}:

Callable that takes *pkg\_name* as argument, and returns the package version string - as in:

```
``version = version_getter(pkg_name)``
```

If None, equivalent to:

```
mod = __import__(pkg_name); version = mod.__version__``
```

**messages** : None or dict, optional

dictionary giving output messages

`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.

## findSurf.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 [group, 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 'surf\_<original mesh file name>.mesh'.

Try `./find_surf.py -help` to see more options.

```
findSurf.main()
```

```
findSurf.surface_components(gr_s, surf_faces)
```

Determine surface components given surface mesh connectivity graph.

```
findSurf.surface_graph(surf_faces, n_nod)
```

## genPerMesh.py script

```
genPerMesh.main()
```

```
genPerMesh.parse_repeat(option, opt, value, parser)
```

### plotPerfusionCoefs.py script

```
plotPerfusionCoefs.load_coefs(filename)
plotPerfusionCoefs.main()
plotPerfusionCoefs.plot_and_save(filename_in, dir_name, fname=None, keys=None, interactive=True)
    If keys == None take all.
plotPerfusionCoefs.plot_history(arr, name)
plotPerfusionCoefs.plot_volume_fractions(vfs, name)
plotPerfusionCoefs.show_array(arr, name)
```

### 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.
```

### script/blockgen.py script

Block mesh generator.

```
blockgen.main()
```

### script/config.py script

SfePy configuration script. It is used in the main Makefile to detect the correct python paths and other options that can be specified in site\_cfg.py. Prints results to stdout.

options:

python\_version

The default Python version that is installed on the system.

system

The operating system (posix or windows).

compile\_flags

Extra compile flags added to the flags supplied by distutils.

link\_flags

Extra linker flags added to the flags supplied by distutils.

`debug_flags`

Debugging flags.

`numpydoc_path`

The path to numpydoc (required for the sphinx documentation).

`is_release`

True for a release, False otherwise. If False, current git commit hash is appended to version string, if the sources are in a repository.

`tetgen_path`

Tetgen executable path.

New options should be added both to `site_cfg_template.py` and `Config` class below.

Examples:

```
$ ./config.py python_version
2.7
```

```
$ ./script/config.py system
posix
```

```
config.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
```

```
convert_mesh.main()
```

### **script/cylindergen.py script**

```
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/evalForms.py script

```
evalForms.create_scalar(name, n_ep)
evalForms.create_scalar_base(name, n_ep)
evalForms.create_scalar_base_grad(name, phic, dim)
evalForms.create_scalar_var_data(name, phi, g, u)
evalForms.create_u_operator(u, transpose=False)
evalForms.create_vector(name, n_ep, dim)
    ordering is DOF-by-DOF
evalForms.create_vector_base(name, phic, dim)
evalForms.create_vector_base_grad(name, gc, transpose=False)
evalForms.create_vector_var_data(name, phi, vidx, g, gt, vgidx, u)
evalForms.grad_vector_to_matrix(name, gv)
evalForms.main()
evalForms.substitute_continuous(expr, names, u, phi)
```

### script/eval\_tl\_forms.py script

### 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/gallery.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\_lobatto1d\_c.py script

Generate lobatto1d.c and lobatto1h.c files.

`gen_lobattold_c.append_declarations` (*out*, *cpolys*, *comment*, *cvar\_name*, *shift=0*)

`gen_lobattold_c.append_lists` (*out*, *names*, *length*)

`gen_lobattold_c.append_polys` (*out*, *cpolys*, *comment*, *cvar\_name*, *var\_name='x'*, *shift=0*)

`gen_lobattold_c.gen_lobatto` (*max\_order*)

`gen_lobattold_c.main()`

`gen_lobattold_c.plot_polys` (*fig*, *polys*, *var\_name='x'*)

### script/gen\_term\_table.py script

```
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, table)
    Terms are sorted by name without the d*_ prefix.
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/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()
```

### script/show\_authors.py script

```
show_authors.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()
```



### 7.7.3 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='2013.2')

Return SfePy installation directory information. Append current git commit hash to *version*.

### 7.7.4 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.pde\_solver\_app module

**class** sfepy.applications.pde\_solver\_app.**PDESolverApp**(*conf*, *options*, *output\_prefix*,  
*init\_equations=True*, *\*\*kwargs*)

**call** (*nls\_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*, *nls\_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.

**nls\_status** : dict-like

The object for storing the nonlinear solver return status.

**app\_options** : kwargs

The keyword arguments that can override application-specific options.

## 7.7.5 sfepy.base package

### sfepy.base.base module

sfepy.base.base.**debug** (*frame=None*)

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.

```

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 **:function:'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*)

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.

`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 1D arrays of the same lengths with non-negative integer keys to a single 2D 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)`  
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.is_derived_class (cls, parent)`

`sfepy.base.base.is_sequence (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
[0.10000000000000001, 0.05000000000000003]
```

`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 ()`

`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.spause (msg=None)`

Waits for a keypress.

If you press: “q” ..... it will call `sys.exit()` any other key ... it will continue execution of the program

This is useful for debugging. This function is called from `pause()`.

`sfepy.base.base.try_imports(imports, fail_msg=None)`

Try import statements until one succeeds.

**Parameters** `imports` : list

The list of import statements.

`fail_msg` : str

If not None and no statement succeeds, a *ValueError* is raised with the given message, appended to all failed messages.

**Returns** `locals` : dict

The dictionary of imported modules.

`sfepy.base.base.update_dict_recursively(dst, src, tuples_too=False, overwrite_by_none=True)`

Update *dst* dictionary recursively using items in *src* dictionary.

**Parameters** `dst` : dict

The destination dictionary.

`src` : dict

The source dictionary.

`tuples_too` : bool

If True, recurse also into dictionaries that are members of tuples.

`overwrite_by_none` : bool

If False, do not overwrite destination dictionary values by None.

**Returns** `dst` : dict

The destination dictionary.

`sfepy.base.base.use_method_with_name(instance, method, new_name)`

## sfepy.base.compat module

This module contains functions that have different names or behavior depending on NumPy and Scipy versions.

`sfepy.base.compat.in1d(ar1, ar2, assume_unique=False)`

Test whether each element of a 1D 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.

**Parameters** `ar1` : array\_like, shape (M,)

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.

**Returns** `mask` : ndarray of bools, shape(M,)

The values *ar1[mask]* are in *ar2*.



**See Also:**

**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 1D sequences. *in1d(a, b)* is roughly equivalent to `np.array([item in b for item in a])`. 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])
```

`sfePy.base.compat.unique` (*ar*, *return\_index=False*, *return\_inverse=False*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are two optional outputs in addition to the unique elements: the indices of the input array that give the unique values, and the indices of the unique array that reconstruct the input array.

**Parameters** *ar* : array\_like

Input array. This will be flattened if it is not already 1-D.

**return\_index** : bool, optional

If True, also return the indices of *ar* that result in the unique array.

**return\_inverse** : bool, optional

If True, also return the indices of the unique array that can be used to reconstruct *ar*.

**Returns** *unique* : ndarray

The sorted unique values.

**unique\_indices** : ndarray, optional

The indices of the unique values in the (flattened) original array. Only provided if *return\_index* is True.

**unique\_inverse** : ndarray, optional

The indices to reconstruct the (flattened) original array from the unique array. Only provided if *return\_inverse* is True.

**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 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

**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 ProblemDefinition instance via `ProblemDefinition.from_conf( conf )`.

**edit** (*key*, *newval*)

**static from\_dict** (*dict\_*, *funmod*, *required=None*, *other=None*, *verbose=True*, *override=None*, *setup=True*)

**static from\_file** (*filename*, *required=None*, *other=None*, *verbose=True*, *define\_args=None*, *override=None*, *setup=True*)

Loads the problem definition from a file.

The filename can either contain plain definitions, or it can contain the `define()` function, in which case it will be called to return the input definitions.

The job of the `define()` function is to return a dictionary of parameters. How the dictionary is constructed is not our business, but the usual way is to simply have a function `define()` along these lines in the input file:

```
def define():
    options = {
        'save_eig_vectors' : None,
        'eigen_solver' : 'eigen1',
    }
    region_2 = {
        'name' : 'Surface',
        'select' : 'nodes of surface',
    }
    return locals()
```

Optionally, the `define()` function can accept additional arguments that should be defined using the *define\_args* tuple or dictionary.

**static from\_file\_and\_options** (*filename, options, required=None, other=None, verbose=True, define\_args=None, setup=True*)

Utility function, a wrapper around `ProblemConf.from_file()` with possible override taken from *options*.

**static from\_module** (*module, required=None, other=None, verbose=True, override=None, setup=True*)

**get\_function** (*name*)

Get a function object given its name.

It can be either in *ProblemConf.funmod*, or a *ProblemConf* attribute directly.

**Parameters** *name* : str or function or None

The function name or directly the function.

**Returns** *fun* : function or None

The required function, or None if *name* was None.

**get\_item\_by\_name** (*key, item\_name*)

Return item with name *item\_name* in configuration group given by *key*.

**get\_raw** (*key=None*)

**setup** (*define\_dict=None, funmod=None, filename=None, required=None, other=None*)

**transform\_input** ()

**transform\_input\_trivial** ()

Trivial input transformations.

**validate** (*required=None, other=None*)

`sfepy.base.conf.dict_from_options` (*options*)

Return a dictionary that can be used to construct/override a *ProblemConf* instance based on *options*.

See `--conf` and `--options` options of the `simple.py` script.

`sfepy.base.conf.dict_from_string` (*string*)

Parse *string* and return a dictionary that can be used to construct/override a *ProblemConf* instance.

`sfepy.base.conf.get_standard_keywords` ()

`sfepy.base.conf.transform_conditions` (*adict, prefix*)

`sfepy.base.conf.transform_ebcs` (*adict*)

`sfepy.base.conf.transform_epbcs` (*adict*)

`sfepy.base.conf.transform_fields` (*adict*)

`sfepy.base.conf.transform_functions` (*adict*)  
`sfepy.base.conf.transform_ics` (*adict*)  
`sfepy.base.conf.transform_integrals` (*adict*)  
`sfepy.base.conf.transform_lcbcs` (*adict*)  
`sfepy.base.conf.transform_materials` (*adict*)  
`sfepy.base.conf.transform_regions` (*adict*)  
`sfepy.base.conf.transform_solvers` (*adict*)  
`sfepy.base.conf.transform_to_i_struct_1` (*adict*)  
`sfepy.base.conf.transform_to_struct_01` (*adict*)  
`sfepy.base.conf.transform_to_struct_1` (*adict*)  
`sfepy.base.conf.transform_to_struct_10` (*adict*)  
`sfepy.base.conf.transform_variables` (*adict*)  
`sfepy.base.conf.tuple_to_conf` (*name, vals, order*)

Convert a configuration tuple *vals* into a Struct named *name*, with attribute names given in and ordered by *order*.

Items in *order* at indices outside the length of *vals* are ignored.

## sfepy.base.getch module

getch()-like unbuffered character reading from stdin on both Windows and Unix

\_Getch classes inspired by Danny Yoo, iskeydown() based on code by Zachary Pincus.

## sfepy.base.goptions module

Various global options/parameters.

### Notes

Inspired by rcParams of matplotlib.

**class** `sfepy.base.goptions.ValidatedDict`

A dictionary object including validation.

**default** = `False`

**key** = `'check_term_finiteness'`

**keys** ()

Return sorted list of keys.

**validate** = `{'verbose': <function validate_bool at 0x3bd1578>, 'check_term_finiteness': <function validate_bool at 0x3bd1578>}`

**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.**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')
```

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.**ensure\_path**(*filename*)

Check if path to *filename* exists and if not, create the necessary intermediate directories.

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*='.')

Locate all files matching given filename pattern in and below supplied root directory.

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_list (fd, n_item, dtype)
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)
    Remove all files and directories in supplied root directory.

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.log module

```
class sfePy.base.log.Log (data_names=None, xlabels=None, ylabels=None, yscales=None,
                        is_plot=True, aggregate=200, log_filename=None, formats=None)
    Log data and (optionally) plot them in the second process via LogPlotter.

add_group (names, 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 ()

sfePy.base.log.get_logging_conf (conf, log_name='log')
    Check for a log configuration ('log' attribute by default) in conf. Supply default values if necessary.

    Parameters conf : Struct
        The configuration object.

    log_name : str, optional
        The name of the log configuration attribute in conf.
```

**Returns** **log** : dict

The dictionary {'plot' : <figure\_file>, 'text' : <text\_log\_file>}. One or both values can be None.

`sfepy.base.log.name_to_key(name, ii)`

`sfepy.base.log.plot_log(fig_num, log, info, xticks=None, yticks=None)`

Plot log data returned by `read_log()` into a specified figure.

**Parameters** **fig\_num** : int

The figure number.

**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.

`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\_plotter module

Plotting class to be used by `Log`.

**class** `sfepy.base.log_plotter.LogPlotter(aggregate=100)`

LogPlotter to be used by `sfepy.base.log.Log`.

**make\_axes** ()

**output** = `Output`

**poll\_draw** ()

**process\_command** (*command*)

**terminate** ()

### **sfePy.base.parse\_conf module**

Create parsing 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_bool (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)
sfePy.base.parse_conf.list_of (element, *elements)
```

### **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.progressbar module**

Text progressbar library for python.

This library provides a text mode progressbar. This is typically used to display the progress of a long running operation, providing a visual clue that processing is underway.

The `ProgressBar` class manages the progress, and the format of the line is given by a number of widgets. A widget is an object that may display differently depending on the state of the progress. There are three types of widget: - a string, which always shows itself; - a `ProgressBarWidget`, which may return a different value every time its update method is called; and - a `ProgressBarWidgetHFill`, which is like `ProgressBarWidget`, except it expands to fill the remaining width of the line.

The progressbar module is very easy to use, yet very powerful. And automatically supports features like auto-resizing when available.

```
class sfePy.base.progressbar.Bar (marker='#', left='|', right='|')
    The bar of progress. It will stretch to fill the line.
    update (pbar, width)

class sfePy.base.progressbar.ETA
    Widget for the Estimated Time of Arrival
    format_time (seconds)
```



```

    update (pbar)

class sfepy.base.progressbar.FileTransferSpeed
    Widget for showing the transfer speed (useful for file transfer).

    update (pbar)

class sfepy.base.progressbar.MyBar (text, verbose=True)
    Encapsulation of a nice progress bar

    init (max)

    update (i)

class sfepy.base.progressbar.Percentage
    Just the percentage done.

    update (pbar)

class sfepy.base.progressbar.ProgressBar (maxval=100, widgets=[<sfepy.base.progressbar.Percentage
                                                    object at 0x42c5190>, ' ',
                                                    <sfepy.base.progressbar.Bar object at 0x42c51d0>],
                                                    term_width=None, fd=<open file '<stderr>', mode
                                                    'w' at 0x2b1629fb4270>)

```

This is the ProgressBar class, it updates and prints the bar.

The `term_width` parameter may be an integer. Or None, in which case it will try to guess it, if it fails it will default to 80 columns.

The simple use is like this: `>>> pbar = ProgressBar().start() >>> for i in xrange(100): ... # do something ... pbar.update(i+1) ... >>> pbar.finish()`

But anything you want to do is possible (well, almost anything). You can supply different widgets of any type in any order. And you can even write your own widgets! There are many widgets already shipped and you should experiment with them.

When implementing a widget update method you may access any attribute or function of the ProgressBar object calling the widget's update method. The most important attributes you would like to access are: - `currval`: current value of the progress, `0 <= currval <= maxval` - `maxval`: maximum (and final) value of the progress - `finished`: True if the bar is have finished (reached 100%), False o/w - `start_time`: first time `update()` method of ProgressBar was called - `seconds_elapsed`: seconds elapsed since `start_time` - `percentage()`: percentage of the progress (this is a method)

```

finish ()
    Used to tell the progress is finished.

```

```

handle_resize (signum, frame)

```

```

percentage ()
    Returns the percentage of the progress.

```

```

start ()
    Start measuring time, and prints the bar at 0%.

```

It returns self so you can use it like this: `>>> pbar = ProgressBar().start() >>> for i in xrange(100): ... # do something ... pbar.update(i+1) ... >>> pbar.finish()`

```

update (value)
    Updates the progress bar to a new value.

```

```

class sfepy.base.progressbar.ProgressBarWidget
    This is an element of ProgressBar formatting.

```

The `ProgressBar` object will call its update value when an update is needed. Its size may change between call, but the results will not be good if the size changes drastically and repeatedly.

**update** (*pbar*)

Returns the string representing the widget.

The parameter *pbar* is a reference to the calling `ProgressBar`, where one can access attributes of the class for knowing how the update must be made.

At least this function must be overridden.

**class** `sfePy.base.progressBar.ProgressBarWidgetHFill`

This is a variable width element of `ProgressBar` formatting.

The `ProgressBar` object will call its update value, informing the width this object must be made. This is like TeX `hfill`, it will expand to fill the line. You can use more than one in the same line, and they will all have the same width, and together will fill the line.

**update** (*pbar*, *width*)

Returns the string representing the widget.

The parameter *pbar* is a reference to the calling `ProgressBar`, where one can access attributes of the class for knowing how the update must be made. The parameter *width* is the total horizontal width the widget must have.

At least this function must be overridden.

**class** `sfePy.base.progressBar.ReverseBar` (*marker*='#', *left*='|', *right*='|')

The reverse bar of progress, or bar of regress. :)

**update** (*pbar*, *width*)

**class** `sfePy.base.progressBar.RotatingMarker` (*markers*='|/\\')

A rotating marker for filling the bar of progress.

**update** (*pbar*)

`sfePy.base.progressBar.progressBar` (*text*='calculating', *maxval*=100)

Returns a useful default progressbar.

Usage:

```
pbar=progressbar(maxval=10000) for i in range(10000):
    pbar.update(i) #do some heavy calculation in each step
pbar.finish()
```

## **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.testing module

```
class sfepy.base.testing.TestCommon(**kwargs)

    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)

    static xfail(test_method)
        Decorator that allows a test to fail.
```

## 7.7.6 sfepy.fem package

WARNING: The code in the fem package is undergoing rapid change. It is best to refer directly to the code base until the code stabilizes.

## sfepy.fem.conditions module

The Dirichlet, periodic and linear combination boundary condition classes, as well as the initial condition class.

```
class sfepy.fem.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.fem.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.fem.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.fem.conditions.**InitialCondition** (*name, region, dofs, key=''*)  
Initial condition.

**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.fem.conditions.**LinearCombinationBC** (*name, region, dofs, key='', times=None, filename=None*)

Linear combination 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 the constraint type.

**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.

**filename** : str, optional

Some conditions can store data (e.g. normal vectors) into a file.

**class** `sfepy.fem.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.

**canonicalize\_dof\_names** (*dofs*)

Canonize the DOF names using the full list of DOFs of a variable.

Assumes single condition instance.

## sfepy.fem.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.fem.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.fem.dof\_info.**EdgeDirectionOperator** (*name*, *nodes*, *region*, *field*, *dof\_names*, *filename=None*)

Transformation matrix operator for edges direction LCBCs.

The substitution (in 3D) is:

$$[u_1, u_2, u_3]^T = [d_1, d_2, d_3]^T w,$$

where  $\underline{d}$  is an edge direction vector averaged into a node. The new DOF is  $w$ .

**get\_vectors** (*nodes*, *region*, *field*, *filename=None*)

**class** sfepy.fem.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** : ProblemDefinition 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).

**class** `sfepy.fem.dof_info.IntegralMeanValueOperator` (*name, nodes, region, field, dof\_names, filename=None*)

Transformation matrix operator for integral mean value LCBCs. All node DOFs are summed to the new one.

**class** `sfepy.fem.dof_info.LCBCOperator` (*\*\*kwargs*)

Base class for LCBC operators.

**treat\_pbcs** (*dofs, master*)

Treat dofs with periodic BC.

**class** `sfepy.fem.dof_info.LCBCOperators` (*name, eq\_map, offset*)

Container holding instances of LCBCOperator subclasses for a single variable.

**Parameters** **name** : str

The object name.

**eq\_map** : EquationMap instance

The equation mapping of the variable.

**offset** : int

The offset added to markers distinguishing the individual LCBCs.

**add\_from\_bc** (*bc, field*)

Create a new LCBC operator described by *bc*, and add it to the container.

**Parameters** **bc** : LinearCombinationBC instance

The LCBC condition description.

**field** : Field instance

The field of the variable.

**append** (*op*)

**finalize** ()

Call this after all LCBCs of the variable have been added.

Initializes the global column indices.

**class** sfepy.fem.dof\_info.**NoPenetrationOperator** (*name, nodes, region, field, dof\_names, filename=None*)

Transformation matrix operator for no-penetration LCBCs.

**class** sfepy.fem.dof\_info.**NormalDirectionOperator** (*name, nodes, region, field, dof\_names, filename=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*)

**class** sfepy.fem.dof\_info.**RigidOperator** (*name, nodes, field, dof\_names, all\_dof\_names*)

Transformation matrix operator for rigid LCBCs.

sfepy.fem.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.fem.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.

sfepy.fem.dof\_info.**group\_chains** (*chain\_list*)

Group EPBC chains.

sfepy.fem.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.fem.dof\_info.**make\_global\_lcbc\_operator** (*lcbc\_ops, adi, new\_only=False*)

Assemble all LCBC operators into a single matrix.

**Returns** **mtx\_lc** : csr\_matrix

The global LCBC operator in the form of a CSR matrix.

**lcdi** : DofInfo

The global active LCBC-constrained DOF information.

**new\_only** : bool

If True, the operator columns will contain only new DOFs.



`sfepy.fem.dof_info.resolve_chains` (*master\_slave, chains*)  
 Resolve EPBC chains - e.g. in corner nodes.

## sfepy.fem.domain module

Computational domain, consisting of the mesh and regions.

**class** `sfepy.fem.domain.Domain` (*name, mesh, verbose=False*)  
 Domain is divided into groups, whose purpose is to have homogeneous data shapes.

**clear\_surface\_groups** ()  
 Remove surface group data.

**create\_region** (*name, select, flags=None, check\_parents=True, functions=None, add\_to\_regions=True*)  
 Region factory constructor. Append the new region to self.regions list.

**create\_regions** (*region\_defs, functions=None*)

**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.fem.mappings.SurfaceMapping`.

**fix\_element\_orientation** ()  
 Ensure element nodes ordering giving positive element volume.

The groups with elements of lower dimension than the space dimension are skipped.

**get\_cell\_offsets** ()

**get\_conns** ()  
 Return the element connectivity groups of the underlying mesh.

**get\_diameter** ()  
 Return the diameter of the domain.

### Notes

The diameter corresponds to the Friedrichs constant.

**get\_element\_diameters** (*ig, cells, vg, mode, square=True*)

**get\_facets** (*force\_faces=False*)  
 Return edge and face descriptions.

**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.

**has\_faces** ()

**iter\_groups** (*igs=None*)

**refine** ()

Uniformly refine the domain mesh.

**Returns domain** : Domain instance

The new domain with the refined mesh.

#### Notes

Works only for meshes with single element type! Does not preserve node groups!

**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.

**setup\_facets** (*create\_edges=True, create\_faces=True, verbose=False*)

Setup the edges and faces (in 3D) of domain elements.

**setup\_groups** ()

**surface\_faces** ()

`sfepy.fem.domain.region_leaf` (*domain, regions, rdef, functions*)

Create/setup a region instance according to rdef.

`sfepy.fem.domain.region_op` (*level, op, item1, item2*)

#### sfepy.fem.equations module

**class** `sfepy.fem.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\_avg', 'qp', 'weak'  
 The evaluation mode.

**static from\_desc** (*name*, *desc*, *variables*, *regions*, *materials*, *integrals*, *user*=None)

**class** sfepy.fem.equations.**Equations** (*equations*, *setup*=True, *make\_virtual*=False, *verbose*=True)

**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)  
 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.

**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** ()

**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 by *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** (*mode*='eval', *dw\_mode*='vector', *term\_mode*=None, *asm\_obj*=None)

**Parameters mode** : one of 'eval', 'el\_avg', 'qp', 'weak'

The evaluation mode.

**static from\_conf** (*conf*, *variables*, *regions*, *materials*, *integrals*, *setup*=True, *user*=None, *make\_virtual*=False, *verbose*=True)

**get\_domain** ()

**get\_graph\_conns** (*any\_dof\_conn*=False, *rdcs*=None, *cdcs*=None)

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.

**rdcs, cdcs** : arrays, optional

Additional row and column DOF connectivities, corresponding to the variables used in the equations.

**Returns rdcs, 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\_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** (*make\_virtual=False, verbose=True*)

**setup\_initial\_conditions** (*ics, functions*)

**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, *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** : ProblemDefinition instance, optional

The problem that can be passed to user functions as a context.

**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.fem.materials.Material.time_update()`.

**problem** : ProblemDefinition instance, optional

The problem that can be passed to user functions as a context.

**verbose** : bool

If False, reduce verbosity.

`sfePy.fem.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.fem.equations.parse_definition(equation_def)`  
 Parse equation definition string to create term description list.

## sfepy.fem.evaluate module

`class sfepy.fem.evaluate.BasicEvaluator(problem, matrix_hook=None)`

```

    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)

```

`class sfepy.fem.evaluate.Evaluator(**kwargs)`

`class sfepy.fem.evaluate.LCBCEvaluator(problem, matrix_hook=None)`

```

    eval_residual(vec, is_full=False)
    eval_tangent_matrix(vec, mtx=None, is_full=False)

```

`sfepy.fem.evaluate.assemble_by_blocks(conf_equations, problem, ebcs=None, epbcs=None, dw_mode='matrix')`

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_iso.i1.Y2( inclusion.lame, v, u )",
}

```

## Notes

`ebcs`, `epbcs` must be either lists of BC names, or BC configuration dictionaries.

`sfepy.fem.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, verbose=True, kwargs=None)`

Create evaluable object (equations and corresponding variables) from the `expression` string.

**Parameters** `expression` : str

The expression to evaluate.

`fields` : dict

The dictionary of fields used in `variables`.

`materials` : Materials instance

The materials used in the expression.

`variables` : Variables instance

The variables used in the expression.

`integrals` : Integrals instance

The integrals to be used.

**regions** : Region instance or list of Region instances

The region(s) to be used. If not given, the regions defined within the fields domain are used.

**ebcs** : Conditions instance, optional

The essential (Dirichlet) boundary conditions for 'weak' mode.

**epbcs** : Conditions instance, optional

The periodic boundary conditions for 'weak' mode.

**lbcs** : Conditions instance, optional

The linear combination boundary conditions for 'weak' mode.

**ts** : TimeStepper instance, optional

The time stepper.

**functions** : Functions instance, optional

The user functions for boundary conditions, materials etc.

**auto\_init** : bool

Set values of all variables to all zeros.

**mode** : one of 'eval', 'el\_avg', 'qp', 'weak'

The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el\_avg' element averages and 'eval' means integration over each term region.

**extra\_args** : dict, optional

Extra arguments to be passed to terms in the expression.

**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.fem.evaluate.eval_equations` (*equations, variables, preserve\_caches=False, mode='eval', dw\_mode='vector', term\_mode=None*)

Evaluate the equations.

**Parameters** **equations** : Equations instance

The equations returned by `create_evaluable()`.

**variables** : Variables instance

The variables returned by `create_evaluable()`.



**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.

**Returns out** : array

The result of the evaluation.

`sfepy.fem.evaluate.eval_in_els_and_qp` (*expression, ig, iels, coors, fields, materials, variables, functions=None, mode='eval', term\_mode=None, extra\_args=None, 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.

**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.

### **sfe.py.fem.evaluate\_variable module**

`sfe.py.fem.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.

`sfe.py.fem.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.

### **sfe.py.fem.facets module**

**class** `sfe.py.fem.facets.Facets` (*name, kind, domain, single\_facets, n\_obj, indices, facets*)

**find\_group\_interfaces** (*return\_surface=True*)

Find facets that create boundary between different element groups, i.e. facets that each belongs to two elements in different groups.

**Parameters** **return\_surface** : bool

If True, the surface facets are also returned.

**Returns** **inter\_facets** : array

The array with indices to *self.facets* of shape (*n\_i*, 2), where *n\_i* is the number of the interface facets. Each row corresponds to a single unique facet, each column to the corresponding two facets from each side of the interface.

**surface\_facets** : array, optional

The array with indices to *self.facets* of shape (*n\_s*,), where *n\_s* is the number of the surface facets.

**static from\_domain** (*domain, kind*)

**get\_complete\_facets** (*vertices, ig=0, mask=None*)

Get complete facets in group *ig* that are defined by the given vertices, or mask, if given.

**Parameters** **vertices** : array

The list of vertices.

**ig** : int

The group index.

**mask** : array, optional

Alternatively to *vertices*, a mask can be given with 1 at indices equal to the vertices and 0 elsewhere.

**Returns** **ifacets** : array

The indices into *self.facets*.

**get\_coors** (*ig=None*)

Get the coordinates of vertices of unique facets in group *ig*.

**Parameters** *ig* : int, optional

The element group. If None, the coordinates for all groups are returned, filled with zeros at places of missing vertices, i.e. where facets having less then the full number of vertices (*n\_v*) are.

**Returns** *coors* : array

The coordinates in an array of shape (*n\_f*, *n\_v*, *dim*).

**uid** : array

The unique ids of facets in the order of *coors*.

**get\_dof\_orientation\_maps** (*nodes*)

Given description of facet DOF nodes, return the corresponding integer coordinates and orientation maps.

### Notes

Assumes single facet type in all groups.

**get\_facet\_dof\_permutations** (*nodes*)

Given description of facet DOF nodes, return the DOF permutations for all possible facet orientations.

**get\_orientation** (*ig*, *tp\_edge\_ori=None*)

Get the orientation flag in group *ig*.

**Parameters** *ig* : int

The group index.

**tp\_edge\_ori** : array, optional

If given, use the tensor edge orientation to fix the flag - the flag is flipped for edges, where *tp\_edge\_ori* is False.

**Returns** *ori* : array

The orientation flag.

**get\_uid\_per\_elements** (*ig=0*)

Get unique ids of facets for all elements in group *ig*.

**mark\_surface\_facets** ()

flag: 0 .. inner, 2 .. edge, 3 .. triangle, 4 .. quadrangle

**setup\_group\_interfaces** ()

Setup facets that create boundary between different element groups.

**setup\_neighbours** ()

For each unique facet: - indices of facets - sparse matrix (*n\_unique* x *n\_all\_obj*)

*mtx*[*i*, *j*] == 1 if *facet*[*j*] has *uid*[*i*]

•number of elements it is in

**setup\_unique** ()

*sorted\_facets* == *permuted\_facets*[*perm*] *permuted\_facets* == *sorted\_facets*[*perm\_i*] *uid* : unique id in order of *sorted\_facets* *uid\_i* : unique id in order of *permuted\_facets* or *facets*

**sort\_and\_orient** ()

`sfe.py.fem.facets.get_facet_dof_permutations` (*int\_coors*, *ori\_maps*)  
Prepare DOF permutation vector for each possible facet orientation.

### **sfe.py.fem.fe\_surface module**

**class** `sfe.py.fem.fe_surface.FESurface` (*name*, *region*, *efaces*, *volume\_econn*, *ig*)  
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*)  
Setup mirror surface connectivity required to integrate over a mirror region.

### **sfe.py.fem.fea module**

**class** `sfe.py.fem.fea.Approximation` (*name*, *interp*, *region*, *ig*, *is\_surface=False*)

**clear\_qp\_base** ()  
Remove cached quadrature points and base functions.

**create\_bqp** (*region\_name*, *integral*)

**describe\_geometry** (*field*, *gtype*, *region*, *integral=None*, *return\_mapping=False*)  
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**

- volume mappings can be defined on a part of an element group, although the field has to be defined always on the whole group.
- surface mappings are defined on the surface region
- surface mappings require field order to be > 0

**eval\_extra\_coor** (*coors*, *mesh\_coors*)  
Compute coordinates of extra nodes.

**get\_base** (*key*, *derivative*, *integral*, *iels=None*, *from\_geometry=False*, *base\_only=True*)

**get\_connectivity** (*region*, *integration*, *is\_trace=False*)  
Return the DOF connectivity for the given geometry type.

**Parameters** *region* : Region instance

The region, used to index surface and volume connectivities.

**integration** : one of ('volume', 'surface', 'surface\_extra')

The term integration type.

**get\_poly\_space** (*key, from\_geometry=False*)

Get the polynomial space.

**Parameters** **key** : 'v' or 's?'

The key denoting volume or surface.

**from\_geometry** : bool

If True, return the polynomial space for affine geometrical interpolation.

**Returns** **ps** : PolySpace instance

The polynomial space.

**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\_s\_data\_shape** (*integral, key*)

returns (n\_fa, n\_qp, dim, n\_fp)

**get\_v\_data\_shape** (*integral=None*)

returns (n\_el, n\_qp, dim, n\_ep)

**setup\_point\_data** (*field, region*)

**setup\_surface\_data** (*region*)

nodes[leconn] == econn

**class** sfepy.fem.fea.**DiscontinuousApproximation** (*name, interp, region, ig, is\_surface=False*)

**eval\_extra\_coors** (*coors, mesh\_coors*)

Compute coordinates of extra nodes. For discontinuous approximations, all nodes are treated as extra.

**class** sfepy.fem.fea.**Interpolant** (*name, gel, space='H1', base='lagrange', approx\_order=1, force\_bubble=False*)

A simple wrapper around PolySpace.

**describe\_nodes** ()

**get\_geom\_poly\_space** (*key*)

**get\_n\_nodes** ()

**class** sfepy.fem.fea.**SurfaceApproximation** (*name, interp, region, ig*)

**get\_qp** (*key, integral*)

Get quadrature points and weights corresponding to the given key and integral. The key is 's#', where # is the number of face vertices.

**class** sfepy.fem.fea.**SurfaceInterpolant** (*name, gel, space='H1', base='lagrange', approx\_order=1, force\_bubble=False*)

Like Interpolant, but for use with SurfaceField and SurfaceApproximation.

**get\_geom\_poly\_space** (*key*)

**sfepy.fem.fea.eval\_nodal\_coors** (*coors, mesh\_coors, region, poly\_space, geom\_poly\_space, econn, ig, only\_extra=True*)

Compute coordinates of nodes corresponding to *poly\_space*, given mesh coordinates and *geom\_poly\_space*.

```
sfepy.fem.fea.set_mesh_coors(domain, fields, coors, update_fields=False, actual=False,
                             clear_all=True)
```

## sfepy.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.fem.fields_base.Field(name, dtype, shape, region, approx_order=1)`  
Base class for finite element fields.

### Notes

- Region can span over several groups -> different Aproximation instances
- interps and hence node\_descs are per region (must have single geometry!)
- no two interps can be in a same group -> no two aps (with different regions) can be in a same group -> aps can be uniquely indexed with ig

**clear\_dof\_conns** ()

**clear\_mappings** (clear\_all=False)  
Clear current reference mappings.

**create\_mapping** (ig, region, integral, integration)  
Create a new reference mapping.

**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.

**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'.

The field shape determines the shape of the FE base functions and can be different from a FieldVariable instance shape. (TODO)

**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\_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\_dofs\_in\_region** (*region*, *merge=False*, *clean=False*, *warn=False*, *igs=None*)

Return indices of DOFs that belong to the given region.

**get\_dofs\_in\_region\_group** (*region, ig, merge=True*)

Return indices of DOFs that belong to the given region and group.

**get\_mapping** (*ig, 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** : VolumeGeometry or SurfaceGeometry instance

The geometry object that describes the mapping.

**mapping** : VolumeMapping or SurfaceMapping instance

The mapping.

**key** : tuple

The key of the mapping in *mappings* or *mappings0*.

**get\_output\_approx\_order** ()

Get the approximation order used in the output file.

**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).

**save\_mappings** ()

Save current reference mappings to *mappings0* attribute.

**setup\_coors** (*coors=None*)

Setup coordinates of field nodes.

**setup\_dof\_conns** (*dof\_conns, dpn, dc\_type, region, is\_trace=False*)

Setup dof connectivities of various kinds as needed by terms.

**class** sfepy.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.

$$u_n = \sum_e (u_{e,avg} * area_e) / \sum_e area_e = \sum_e \int_{area_e} u / \sum_e area_e$$

**setup\_dof\_conns** (*dof\_conns, dpn, dc\_type, region, is\_trace=False*)

Setup dof connectivities of various kinds as needed by terms.

**setup\_extra\_data** (*geometry, info, is\_trace*)

**class** sfepy.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.

$$u_n = \sum_e (u_{e,avg} * volume_e) / \sum_e volume_e = \sum_e \int_{volume_e} u / \sum_e volume_e$$

**setup\_dof\_conns** (*dof\_conns, dpn, dc\_type, region, is\_trace=False*)

Setup dof connectivities of various kinds as needed by terms.

**setup\_extra\_data** (*geometry, info, is\_trace*)

```
sfepy.fem.fields_base.create_dof_conn(conn, dpn)
```

Given element a node connectivity, create the dof connectivity.

```
sfepy.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.fem.fields_base.fields_from_conf(conf, regions)`

`sfepy.fem.fields_base.get_eval_expression(expression, ig, 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.fem.fields_base.parse_approx_order(approx_order)`

Parse the uniform approximation order value (str or int).

`sfepy.fem.fields_base.setup_dof_conns(conn_info, dof_conns=None, make_virtual=False, verbose=True)`

Dof connectivity key: (field.name, var.n\_components, region.name, type, ig)

`sfepy.fem.fields_base.setup_extra_data(conn_info)`

Setup extra data required for non-volume integration.

## sfepy.fem.fields\_hierarchic module

`class sfepy.fem.fields_hierarchic.H1HierarchicVolumeField(name, dtype, shape, region, approx_order=1)`

`evaluate_at(coors, source_vals, strategy='kdtree', close_limit=0.1, cache=None, ret_cells=False, ret_status=False, ret_ref_coors=False)`

Evaluate source DOF values corresponding to the field in the given coordinates using the field interpolation.

**Parameters coors** : array

The coordinates the source values should be interpolated into.

**source\_vals** : array

The source DOF values corresponding to the field.

**strategy** : str, optional

The strategy for finding the elements that contain the coordinates. Only 'kdtree' is supported for the moment.

**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, the inverse connectivity of the field mesh and the KDTree instance can be cached as `cache.mesh`, `cache.offsets`, `cache.iconn` and `cache.kdtree`. 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 KDTree related data are ignored.

**ret\_cells** : bool, optional

If True, return also the cell indices the coordinates are in.

**ret\_status** : bool, optional

If True, return also the status for each point: 0 is success, 1 is extrapolation within *close\_limit*, 2 is extrapolation outside *close\_limit*, 3 is failure.

**ret\_ref\_coors** : bool, optional

If True, return also the found reference element coordinates.

**Returns** **vals** : array

The interpolated values.

**cells** : array

The cell indices, if *ret\_cells* or *ret\_status* are True.

**status** : array

The status, if *ret\_status* is True.

**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.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.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.fem.fields_nodal.H1NodalMixin` (\*\**kwargs*)

**evaluate\_at** (*coors*, *source\_vals*, *strategy*='kdtree', *close\_limit*=0.1, *cache*=None, *ret\_cells*=False, *ret\_status*=False, *ret\_ref\_coors*=False)

Evaluate source DOF values corresponding to the field in the given coordinates using the field interpolation.

**Parameters** **coors** : array

The coordinates the source values should be interpolated into.

**source\_vals** : array

The source DOF values corresponding to the field.

**strategy** : str, optional

The strategy for finding the elements that contain the coordinates. Only 'kdtree' is supported for the moment.

**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, the inverse connectivity of the field mesh and the KDTree instance can be cached as *cache.mesh*, *cache.offsets*, *cache.iconn* and *cache.kdtree*. 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 KDTree related data are ignored.

**ret\_cells** : bool, optional

If True, return also the cell indices the coordinates are in.

**ret\_status** : bool, optional

If True, return also the status for each point: 0 is success, 1 is extrapolation within *close\_limit*, 2 is extrapolation outside *close\_limit*, 3 is failure.

**ret\_ref\_coors** : bool, optional

If True, return also the found reference element coordinates.

**Returns vals** : array

The interpolated values.

**cells** : array

The cell indices, if *ret\_cells* or *ret\_status* are True.

**status** : array

The status, if *ret\_status* is True.

**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.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.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.fem.functions module**

**class** `sfepy.fem.functions.ConstantFunction` (*values*)

Function with constant values.

**class** `sfepy.fem.functions.ConstantFunctionByRegion` (*values*)

Function with constant values in regions.

**class** `sfepy.fem.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.fem.functions.Functions` (*objs=None, \*\*kwargs*)

Container to hold all user-defined functions.

**static from\_conf** (*conf*)

### **sfepy.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.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.fem.geometry_element.setup_orientation` (*vecs\_tuple*)

## sfepy.fem.global\_interp module

Global interpolation functions.

`sfepy.fem.global_interp.get_ref_coors` (*field*, *coors*, *strategy*='kdtree', *close\_limit*=0.1, *cache*=None)

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* : str, optional

The strategy for finding the elements that contain the coordinates. Only 'kdtree' is supported for the moment.

*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, the inverse connectivity of the field mesh and the KDTree instance can be cached as *cache.mesh*, *cache.offsets*, *cache.iconn* and *cache.kdtree*. 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 KDTree related data are ignored.

**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.

## sfepy.fem.history module

`class sfepy.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.fem.history.History` (*name*, *th*=None, *steps*=None, *times*=None)

**append** (*item*, *step*, *time*)

**static from\_sequence** (*seq*, *name*)

### sfepy.fem.linearizer module

Linearization of higher order solutions for the purposes of visualization.

`sfepy.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.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.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.fem.integrals module

**class** `sfepy.fem.integrals.Integral` (*name*, *kind='v'*, *order=1*, *coors=None*, *weights=None*)

Wrapper class around quadratures.

**get\_key** ()

Get the key string corresponding to the integral kind and order, that can be used to distinguish various cached data evaluated using the integral.

**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.fem.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.fem.quadratures.quadrature\_tables*.

**Returns** *val* : float

The value of the integral.



```
class sfepy.fem.integrals.Integrals (objs=None, **kwargs)
    Container for instances of Integral.

    static from_conf (conf)

    get (name, kind='v')
        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' (exist-
            ing custom integral name).
```

## sfepy.fem.mappings module

Finite element reference mappings.

```
class sfepy.fem.mappings.Mapping (coors, conn, poly_space=None, gel=None, order=1)
    Base class for 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.fem.mappings.PhysicalQPs (**kwargs)
    Physical quadrature points in a region.

    get_merged_values ()

    get_shape (rshape, ig=None)
        Get shape from raveled shape.

class sfepy.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_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.

class sfepy.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)
        Get the mapping for given quadrature points, weights, and polynomial space.

        Returns
        cmap : CMapping instance

            The volume mapping.
```

`sfepy.fem.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.fem.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.fem.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.fem.mappings.get_physical_qps` (*region*, *integral*)

Get physical quadrature points corresponding to the given region and integral.

## sfepy.fem.mass\_operator module

**class** `sfepy.fem.mass_operator.MassOperator` (*problem*, *options*)

Encapsulation of action and inverse action of a mass matrix operator  $M$ .

**action** (*vec*)

Action of mass matrix operator on a vector:  $Mx$ .

**inverse\_action** (*vec*)

Inverse action of mass matrix operator on a vector:  $M^{-1}x$ .

## sfepy.fem.materials module

**class** `sfepy.fem.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, ig, 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, ig, qps, data, indx*)

Set the material data in quadrature points.

**Parameters** **key** : tuple

The (region\_name, integral\_name) data key.

**ig** : int

The element group id.

**qps** : Struct

Information about the quadrature points.

**data** : dict

The material data. Changes the shape of data!

**indx** : array

The indices of quadrature points in the group *ig*.

**set\_data\_from\_variable** (*var, name, equations*)

**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** : ProblemDefinition 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** : ProblemDefinition, 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** : ProblemDefinition, 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** : ProblemDefinition, optional

The problem definition for which the update occurs.

**class** `sfepy.fem.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.

**semideep\_copy** (*reset=True*)

Copy materials, while external data (e.g. region) remain shared.

**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** : ProblemDefinition instance, optional

The problem that can be passed to user functions as a context.

**verbose** : bool

If False, reduce verbosity.

## sfepy.fem.mesh module

**class** sfepy.fem.mesh.**Mesh** (*name='mesh', filename=None, prefix\_dir=None, \*\*kwargs*)

Contains the FEM mesh together with all utilities related to it.

Input and output is handled by the MeshIO class and subclasses. The Mesh class only contains the real mesh - nodes, connectivity, regions, plus methods for doing operations on this mesh.

Example of creating and working with a mesh:

```
In [1]: from sfepy.fem import Mesh
In [2]: m = Mesh.from_file("meshes/3d/cylinder.vtk")
sfepy: reading mesh (meshes/3d/cylinder.vtk)...
sfepy: ...done in 0.04 s

In [3]: m.coors
Out[3]:
array([[ 1.00000000e-01,  2.00000000e-02, -1.22460635e-18],
       [ 1.00000000e-01,  1.80193774e-02,  8.67767478e-03],
       [ 1.00000000e-01,  1.24697960e-02,  1.56366296e-02],
       ...,
       [ 8.00298527e-02,  5.21598617e-03, -9.77772215e-05],
       [ 7.02544004e-02,  3.61610291e-04, -1.16903153e-04],
       [ 3.19633596e-02, -1.00335972e-02,  9.60460305e-03]])

In [4]: m.ngroups
Out[4]: array([0, 0, 0, ..., 0, 0, 0])

In [5]: m.conns
Out[5]:
[array([[ 28,  60,  45,  29],
```

```

    [ 28, 60, 57, 45],
    [ 28, 57, 27, 45],
    ...,
    [353, 343, 260, 296],
    [353, 139, 181, 140],
    [353, 295, 139, 140]])]

In [6]: m.mat_ids
Out[6]: [array([6, 6, 6, ..., 6, 6, 6])]

In [7]: m.descs
Out[7]: ['3_4']

In [8]: m
Out[8]: Mesh:meshes/3d/cylinder

In [9]: print m
Mesh:meshes/3d/cylinder
  conns:
    [array([[ 28, 60, 45, 29],
             [ 28, 60, 57, 45],
             [ 28, 57, 27, 45],
             ...,
             [353, 343, 260, 296],
             [353, 139, 181, 140],
             [353, 295, 139, 140]])]
  coors:
    [[ 1.00000000e-01  2.00000000e-02 -1.22460635e-18]
     [ 1.00000000e-01  1.80193774e-02  8.67767478e-03]
     [ 1.00000000e-01  1.24697960e-02  1.56366296e-02]
     ...,
     [ 8.00298527e-02  5.21598617e-03 -9.77772215e-05]
     [ 7.02544004e-02  3.61610291e-04 -1.16903153e-04]
     [ 3.19633596e-02 -1.00335972e-02  9.60460305e-03]]
  descs:
    ['3_4']
  dim:
    3
  el_offsets:
    [ 0 1348]
  io:
    None
  mat_ids:
    [array([6, 6, 6, ..., 6, 6, 6])]
  n_e_ps:
    [4]
  n_el:
    1348
  n_els:
    [1348]
  n_nod:
    354
  name:
    meshes/3d/cylinder
  ngroups:
    [0 0 0 ..., 0 0 0]
  setup_done:
    0

```

The `Mesh().coors` is an array of node coordinates and `Mesh().conns` is the list of elements of each type (see `Mesh().desc`), so for example if you want to know the coordinates of the nodes of the fifth finite element of the type `3_4` do:

```
In [10]: m.descs
Out[10]: ['3_4']
```

So now you know that the finite elements of the type `3_4` are in `a.conns[0]`:

```
In [11]: m.coors[m.conns[0][4]]
Out[11]:
array([[ 1.00000000e-01,  1.80193774e-02, -8.67767478e-03],
       [ 1.00000000e-01,  1.32888539e-02, -4.35893200e-04],
       [ 1.00000000e-01,  2.00000000e-02, -1.22460635e-18],
       [ 9.22857574e-02,  1.95180454e-02, -4.36416134e-03]])
```

The element ids are of the form “<dimension>\_<number of nodes>”, i.e.:

- `2_2` ... line
- `2_3` ... triangle
- `2_4` ... quadrangle
- `3_2` ... line
- `3_4` ... tetrahedron
- `3_8` ... hexahedron

**copy** (*name=None*)

Make a deep copy of self.

**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.

**explode\_groups** (*eps, return\_emap=False*)

Explode the mesh element groups by *eps*, i.e. split group interface nodes and shrink each group towards its centre by *eps*.

**Parameters** **eps** : float in *[0.0, 1.0]*

The group shrinking factor.

**return\_emap** : bool, optional

If True, also return the mapping against original mesh coordinates that result in the exploded mesh coordinates. The mapping can be used to map mesh vertex data to the exploded mesh vertices.

**Returns** **mesh** : Mesh

The new mesh with exploded groups.

**emap** : spmatrix, optional

The mapping for exploding vertex values. Only provided if *return\_emap* is True.



**static from\_data** (*name, coors, ngroups, conns, mat\_ids, desc, igs=None, nodal\_bcs=None*)

Create a mesh from mesh 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, save\_edges=False, save\_faces=False, localize=False, is\_surface=False*)

Create a mesh corresponding to a given region.

**static from\_surface** (*surf\_faces, mesh\_in*)

Create a mesh given a set of surface faces and the original mesh.

**get\_bounding\_box** ()

**get\_element\_coors** (*ig=None*)

Get the coordinates of vertices elements in group *ig*.

**Parameters** **ig** : int, optional

The element group. If None, the coordinates for all groups are returned, filled with zeros at places of missing vertices, i.e. where elements having less then the full number of vertices (*n\_ep\_max*) are.

**Returns** **coors** : array

The coordinates in an array of shape (*n\_el, n\_ep\_max, dim*).

**localize** (*inod*)

Strips nodes not in inod and remaps connectivities. Omits elements where remap[conn] contains -1...

**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, coors=None, igs=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*.

**coors** : array, optional

The coordinates that can be used instead of the mesh coordinates.

**igs** : array\_like, optional

Passing a list of group ids selects only those groups for writing.

**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.fem.mesh.find_map(x1, x2, eps=1e-08, allow_double=False, join=True)`

Find a mapping between common coordinates in x1 and x2, such that `x1[cmap[:,0]] == x2[cmap[:,1]]`

`sfepy.fem.mesh.fix_double_nodes(coor, ngroups, conns, eps)`

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.fem.mesh.get_min_edge_size(coor, conns)`

Get the smallest edge length.

`sfepy.fem.mesh.get_min_vertex_distance(coor, guess)`

Can miss the minimum, but is enough for our purposes.

`sfepy.fem.mesh.get_min_vertex_distance_naive(coor)`

`sfepy.fem.mesh.make_inverse_connectivity(conns, n_nod, ret_offsets=True)`

For each mesh node referenced in the connectivity conns, make a list of elements it belongs to.

`sfepy.fem.mesh.make_mesh(coor, ngroups, conns, mesh_in)`

Create a mesh reusing *mat\_ids* and *descs* of *mesh\_in*.

`sfepy.fem.mesh.make_point_cells(indx, dim)`

`sfepy.fem.mesh.merge_mesh(x1, ngroups1, conns1, mat_ids1, x2, ngroups2, conns2, mat_ids2, cmap, eps=1e-08)`

Merge two meshes in common coordinates found in x1, x2.

## Notes

Assumes the same number and kind of element groups in both meshes!

**sfepy.fem.meshio module**

```
class sfepy.fem.meshio.ANSYSCDBMeshIO (filename, **kwargs)
```

```

    format = 'ansys_cdb'
    static guess (filename)
    static make_format (format)
    read (mesh, **kwargs)
    read_bounding_box ()
    read_dimension (ret_fd=False)
    write (filename, mesh, out=None, **kwargs)
```

```
class sfepy.fem.meshio.AVSUCDMeshIO (filename, **kwargs)
```

```

    format = 'avs_uct'
    static guess (filename)
    read (mesh, **kwargs)
    read_dimension ()
    write (filename, mesh, out=None, **kwargs)
```

```
class sfepy.fem.meshio.AbaqusMeshIO (filename, **kwargs)
```

```

    format = 'abaqus'
    static guess (filename)
    read (mesh, **kwargs)
    read_dimension ()
    write (filename, mesh, out=None, **kwargs)
```

```
class sfepy.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.fem.meshio.ComsolMeshIO (filename, **kwargs)
```

```

    format = 'comsol'
    read (mesh, **kwargs)
    write (filename, mesh, out=None, **kwargs)
```

```
class sfepy.fem.meshio.HDF5MeshIO (filename, **kwargs)
```

```
ch = '\xfe'
format = 'hdf5'
read(mesh, **kwargs)
read_bounding_box(ret_fd=False, ret_dim=False)
read_data(step, filename=None)
read_data_header(dname, step=0, filename=None)
read_last_step(filename=None)
read_time_history(node_name, indx, filename=None)
read_time_stepper(filename=None)
read_times(filename=None)
    Read true time step data from individual time steps.

    Returns steps : array
        The time steps.
    times : array
        The times of the time steps.
    nts : array
        The normalized times of the time steps, in [0, 1].
read_variables_time_history(var_names, ts, filename=None)
string = <module 'string' from '/usr/lib/python2.7/string.pyc'>
write(filename, mesh, out=None, ts=None, **kwargs)
class sfepy.fem.meshio.HypermeshAsciiMeshIO(filename, **kwargs)

    format = 'hmaskii'
    read(mesh, **kwargs)
    read_dimension()
    write(filename, mesh, out=None, **kwargs)
class sfepy.fem.meshio.MEDMeshIO(filename, **kwargs)

    format = 'med'
    read(mesh, **kwargs)
class sfepy.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.fem.meshio.Mesh3DMeshIO(filename, **kwargs)
```

```
    format = 'mesh3d'
    read(mesh, **kwargs)
    read_dimension()
```

```
class sfepy.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 = ...
    conns = ...
    mat_ids = ...
    descs = ...
    mesh._set_data(nodes, conns, mat_ids, descs)
    return mesh
```

See the Mesh class' docstring how the nodes, conns, mat\_ids and descs should look like. You just need to read them from your specific format from disk.

To write a mesh to disk, just implement the write() method and use the information from the mesh instance (e.g. nodes, conns, mat\_ids and descs) to construct your specific format.

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 of read\_last\_step() just returns 0. It should be reimplemented in subclasses capable of storing several steps.

```
static any_from_filename(filename, prefix_dir=None)
```

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*)

**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.fem.meshio.NEUMeshIO` (*filename*, *\*\*kwargs*)

**format** = 'gambit'

**read** (*mesh*, *\*\*kwargs*)

**read\_dimension** (*ret\_fd=False*)

**write** (*filename*, *mesh*, *out=None*, *\*\*kwargs*)

```
class sfepy.fem.meshio.TetgenMeshIO(filename, **kwargs)
```

```
format = 'tetgen'
```

```
static getele (fele, up, verbose=False)
```

Reads t.l.ele, returns a list of elements.

Example:

```
>>> elements, regions = self.getele("t.l.ele", MyBar("elements:"))
>>> 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, up, verbose=False)
```

Reads t.l.nodes, returns a list of nodes.

Example:

```
>>> self.getnodes("t.l.node", MyBar("nodes:"))
[(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.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.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)  
    Point data only!  
read_dimension (ret_fd=False)  
write (filename, mesh, out=None, ts=None, **kwargs)  
sfepy.fem.meshio.convert_complex_output (out_in)  
    Convert complex values in the output dictionary out_in to pairs of real and imaginary parts.  
sfepy.fem.meshio.guess_format (filename, ext, formats, io_table)  
    Guess the format of filename, candidates are in formats.  
sfepy.fem.meshio.join_conn_groups (conns, desc, mat_ids, concat=False)  
    Join groups of the same element type.  
sfepy.fem.meshio.mesh_from_groups (mesh, ids, coors, ngroups, tris, mat_tris, quads, mat_quads,  
                                   tetras, mat_tetras, hexas, mat_hexas, remap=None)  
sfepy.fem.meshio.output_writable_meshes ()  
sfepy.fem.meshio.sort_by_mat_id (conns_in)  
    Sort by mat_id within a group, preserve order.  
sfepy.fem.meshio.sort_by_mat_id2 (conns_in, mat_ids_in)  
    Sort by mat_id within a group, preserve order.  
sfepy.fem.meshio.split_by_mat_id (conns_in, mat_ids_in, desc_in)
```

### Notes

*conns\_in* must be sorted by *mat\_id* within a group!

```
sfepy.fem.meshio.write_bb (fd, array, dtype)
```

### sfepy.fem.parseEq module

```
class sfepy.fem.parseEq.TermParse  
sfepy.fem.parseEq.collect_term (term_descs, lc)  
sfepy.fem.parseEq.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.fem.parseEq.rhs (lc)
```

### sfepy.fem.parseReg module

Grammar for selecting regions of a domain.

Regions serve for selection of certain parts of the computational domain (= selection of nodes and elements of a FE 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.fem.parseReg.create_bnf (stack)
```



```

sfepy.fem.parseReg.join_tokens (str, loc, toks)
sfepy.fem.parseReg.print_leaf (level, op)
sfepy.fem.parseReg.print_op (level, op, item1, item2)
sfepy.fem.parseReg.print_stack (stack)
sfepy.fem.parseReg.replace (what, keep=False)
sfepy.fem.parseReg.replace_with_region (what, r_index)
sfepy.fem.parseReg.to_stack (stack)
sfepy.fem.parseReg.visit_stack (stack, op_visitor, leaf_visitor)

```

### sfepy.fem.periodic module

```

sfepy.fem.periodic.match_coors (coors1, coors2)
    Match coordinates coors1 with coors2.
sfepy.fem.periodic.match_grid_line (coor1, coor2, which)
    Match coordinates coor1 with coor2 along the axis which.
sfepy.fem.periodic.match_grid_plane (coor1, coor2, which)
    Match coordinates coor1 with coor2 along the plane with normal axis which.
sfepy.fem.periodic.match_x_line (coor1, coor2)
sfepy.fem.periodic.match_x_plane (coor1, coor2)
sfepy.fem.periodic.match_y_line (coor1, coor2)
sfepy.fem.periodic.match_y_plane (coor1, coor2)
sfepy.fem.periodic.match_z_line (coor1, coor2)
sfepy.fem.periodic.match_z_plane (coor1, coor2)
sfepy.fem.periodic.set_accuracy (eps)

```

### sfepy.fem.poly\_spaces module

```

class sfepy.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.fem.poly_spaces.LagrangeSimplexBPolySpace (name, geometry, order)
    Lagrange polynomial space with forced bubble function on a simplex domain.
    name = 'lagrange_simplex_bubble'
class sfepy.fem.poly_spaces.LagrangeSimplexPolySpace (name, geometry, order)
    Lagrange polynomial space on a simplex domain.

```

```
name = 'lagrange_simplex'
```

```
class sfepy.fem.poly_spaces.LagrangeTensorProductPolySpace (name, geometry, order)
```

Lagrange polynomial space on a tensor product domain.

```
get_mtx_i ()
```

```
name = 'lagrange_tensor_product'
```

```
class sfepy.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.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.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=False, ori=None, force_axis=False, suppress_errors=False, eps=1e-15)
```

Evaluate the basis in points given by coordinates. The real work is done in `_eval_base()` implemented in subclasses.

**Parameters** `coors` : array\_like

The coordinates of points where the basis is evaluated. See Notes.

`diff` : bool

If True, return the first 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.

`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 derivative (shape (n\_coor, 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.fem.probes module

Classes for probing values of Variables, for example, along a line.

**class** `sfepy.fem.probes.CircleProbe` (*centre, normal, radius, n\_point, mesh, share\_mesh=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.fem.probes.IntegralProbe` (*name, problem, expressions, labels*)

Evaluate integral expressions.

**class** `sfepy.fem.probes.LineProbe` (*p0, p1, n\_point, mesh, share\_mesh=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.fem.probes.PointsProbe` (*points, mesh, share\_mesh=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.fem.probes.Probe` (*name, mesh, share\_mesh=True, n\_point=None, \*\*kwargs*)

Base class for all point probes. Enforces two points minimum.

**cache** = **Struct:probe\_shared\_cache**

**is\_cyclic** = **False**

**probe** (*variable*)

Probe the given variable.

**Parameters** **variable** : Variable instance

The variable to be sampled along the probe.

**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.fem.probes.RayProbe` (*p0, dirvec, p\_fun, n\_point, both\_dirs, mesh, share\_mesh=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.

`sfepy.fem.probes.get_data_name` (*fd*)

Try to read next data name in file fd.

**Returns** **name** : str

The data name.

**nc** : int

The number of data columns.

`sfepy.fem.probes.read_header` (*fd*)

Read the probe data header from file descriptor fd.

**Returns** **header** : Struct instance

The probe data header.

`sfepy.fem.probes.read_results` (*filename, only\_names=None*)

Read probing results from a file.

**Parameters** **filename** : str or file object

The probe results file name.

**Returns** **header** : Struct instance

The probe data header.

**results** : dict

The dictionary of probing results. Keys are data names, values are the probed values.

`sfepy.fem.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.fem.problemDef module**

```
class sfePy.fem.problemDef.ProblemDefinition(name, conf=None, functions=None, do-
                                             main=None, fields=None, equations=None,
                                             auto_conf=True, nls=None, ls=None,
                                             ts=None, auto_solvers=True)
```

Problem definition, the top-level class holding all data necessary to solve a problem.

It can be constructed from a `ProblemConf` instance using `ProblemDefinition.from_conf()` or directly from a problem description file using `ProblemDefinition.from_conf_file()`

For interactive use, the constructor requires only the *equations*, *nls* and *ls* keyword arguments.

**advance** (*ts=None*)

**clear\_equations** ()

**copy** (*name=None*)

Make a copy of `ProblemDefinition`.

```
create_evaluable(expression, try_equations=True, auto_init=False, preserve_caches=False,
                  copy_materials=True, integrals=None, ebcs=None, epbcs=None, lcbcs=None,
                  ts=None, functions=None, mode='eval', var_dict=None, strip_variables=True,
                  extra_args=None, verbose=True, **kwargs)
```

Create evaluable object (equations and corresponding variables) from the *expression* string. Convenience function calling `create_evaluable()` with defaults provided by the `ProblemDefinition` instance *self*.

The evaluable can be repeatedly evaluated by calling `eval_equations()`, e.g. for different values of variables.

**Parameters** **expression** : str

The expression to evaluate.

**try\_equations** : bool

Try to get variables from *self.equations*. If this fails, variables can either be provided in *var\_dict*, as keyword arguments, or are created automatically according to the expression.

**auto\_init** : bool

Set values of all variables to all zeros.

**preserve\_caches** : bool

If True, do not invalidate evaluate caches of variables.

**copy\_materials** : bool

Work with a copy of *self.equations.materials* instead of reusing them. Safe but can be slow.

**integrals** : Integrals instance, optional

The integrals to be used. Automatically created as needed if not given.

**ebcs** : Conditions instance, optional

The essential (Dirichlet) boundary conditions for 'weak' mode. If not given, *self.ebcs* are used.

**epbcs** : Conditions instance, optional

The periodic boundary conditions for 'weak' mode. If not given, *self.epbcs* are used.

**lcbcs** : Conditions instance, optional

The linear combination boundary conditions for 'weak' mode. If not given, *self.lcbcs* are used.

**ts** : TimeStepper instance, optional

The time stepper. If not given, *self.ts* is used.

**functions** : Functions instance, optional

The user functions for boundary conditions, materials etc. If not given, *self.functions* are used.

**mode** : one of 'eval', 'el\_avg', 'qp', 'weak'

The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el\_avg' element averages and 'eval' means integration over each term region.

**var\_dict** : dict, optional

The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression. Use this if the name of a variable conflicts with one of the parameters of this method.

**strip\_variables** : bool

If False, the variables in *var\_dict* or *kwargs* not present in the expression are added to the actual variables as a context.

**extra\_args** : dict, optional

Extra arguments to be passed to terms in the expression.

**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 ProblemDefinition instance.

```
>>> out = problem.create_evaluable('dq_state_in_volume_qp.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\_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.

**evaluate** (*expression*, *try\_equations=True*, *auto\_init=False*, *preserve\_caches=False*, *copy\_materials=True*, *integrals=None*, *ebcs=None*, *epbcs=None*, *lcbcs=None*, *ts=None*, *functions=None*, *mode='eval'*, *dw\_mode='vector'*, *term\_mode=None*, *var\_dict=None*, *strip\_variables=True*, *ret\_variables=False*, *verbose=True*, *extra\_args=None*, *\*\*kwargs*)

Evaluate an expression, convenience wrapper of `ProblemDefinition.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 `ProblemDefinition.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\_evaluator** (*reuse=False*)

Either create a new Evaluator instance (*reuse == False*), or return an existing instance, created in a preceding call to `ProblemDefinition.init_solvers()`.

**get\_integrals** (*names=None*, *kind=None*)

Get integrals, initialized from problem configuration if available.

**Parameters** *names* : list, optional

If given, only the named integrals are returned.

**kind** : 'v' or 's', optional

If given, only integrals of the given kind are returned.

**Returns** *integrals* : Integrals instance

The requested integrals.

**get\_materials** ()

**get\_mesh\_coors** ()

**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\_solver\_conf** (*name*)

**get\_solvers** ()

**get\_time\_solver** (*ts\_conf=None*, *\*\*kwargs*)

Create and return a TimeSteppingSolver instance.

## Notes

Also sets *self.ts* attribute.

**get\_timestepper** ()

**get\_variables** (*auto\_create=False*)

**init\_solvers** (*nls\_status=None*, *ls\_conf=None*, *nls\_conf=None*, *mtx=None*, *presolve=False*)

Create and initialize solvers.

**init\_time** (*ts*)

**init\_variables** (*state*)

Initialize variables with history.

**is\_linear** ()

**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 ProblemDefinition instance with the refined mesh.

**remove\_bcs** ()

Convenience function to remove boundary conditions.

**reset** ()

**save\_ebc** (*filename*, *force=True*, *default=0.0*)

Save essential boundary conditions as state variables.

**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\_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\_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\_linear** (*is\_linear*)

**set\_materials** (*conf\_materials=None*)

Set definition of materials.

**set\_mesh\_coors** (*coors, update\_fields=False, actual=False, clear\_all=True*)

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*)

**set\_solvers** (*conf\_solvers=None, options=None*)

Choose which solvers should be used. If solvers are not set in *options*, use first suitable in *conf\_solvers*.

**set\_solvers\_instances** (*ls=None, nls=None*)

Set the instances of linear and nonlinear solvers that will be used in *ProblemDefinition.solve()* call.

**set\_variables** (*conf\_variables=None*)

Set definition of variables.

**setup\_default\_output** (*conf=None, options=None*)

Provide default values to *ProblemDefinition.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 *ProblemDefinition* instance (*self*) as the first argument

**setup\_ic** (*conf\_ics=None, functions=None*)

**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, nls\_status=None, ls\_conf=None, nls\_conf=None, force\_values=None, var\_data=None*)

Solve self.equations in current time step.

**Parameters** *var\_data* : dict

A dictionary of {variable\_name : data vector} used to initialize parameter variables.

**time\_update** (*ts=None, ebcs=None, epbcs=None, lcbcs=None, functions=None, create\_matrix=False*)

**update\_equations** (*ts=None, ebcs=None, epbcs=None, lcbcs=None, functions=None, create\_matrix=False*)

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.

**lcbcs** : Conditions instance, optional

The linear combination boundary conditions. If not given, *self.lcbcs* are used.

**functions** : Functions instance, optional

The user functions for boundary conditions, materials, etc. If not given, *self.functions* are used.

**update\_materials** (*ts=None, mode='normal'*)

Update materials used in equations.

**update\_time\_stepper** (*ts*)

## **sfePy.fem.projections module**

Construct projections between FE spaces.

**sfePy.fem.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.fem.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.fem.projections.make\_l2\_projection** (*target, source*)

Project a scalar *source* field variable to a scalar *target* field variable using the  $L^2$  dot product.

`sfe.py.fem.projections.make_12_projection_data(target, eval_data)`

Project scalar data given by a material-like `eval_data()` function to a scalar *target* field variable using the  $L^2$  dot product.

## sfe.py.fem.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/hegde>) by Andreas Kloeckner.

```
class sfe.py.fem.quadratures.QuadraturePoints(data,          coors=None,      weights=None,
                                              bounds=None,    tp_fix=1.0,   weight_fix=1.0,
                                              symmetric=False)
```

Representation of a set of quadrature points.

**Parameters** `data` : array\_like

The array of shape  $(n\_point, dim + 1)$  of quadrature point coordinates (first *dim* columns) and weights (the last column).

**coors** : array\_like, optional

Optionally, instead of using *data*, the coordinates and weights can be provided separately - *data* are then ignored.

**weights** : array\_like, optional

Optionally, instead of using *data*, the coordinates and weights can be provided separately - *data* are then ignored.

**bounds** : (float, float), optional

The coordinates and weights should correspond to a reference element in  $[0, 1] \times \dim$ . Provide the correct bounds if this is not the case.

**tp\_fix** : float, optional

The value that is used to multiply the tensor product element volume (= 1.0) to get the correct volume.

**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.

`sfepy.fem.quadratures.get_actual_order` (*geometry*, *order*)

Return the actual integration order for given geometry.

**Parameters** *geometry* : str

The geometry key describing the integration domain, see the keys of *quadrature\_tables*.

**Returns** *order* : int

If *order* is in quadrature tables it is this value. Otherwise it is the closest higher order. If no higher order is available, a warning is printed and the highest available order is used.

## sfepy.fem.refine module

Basic uniform mesh refinement functions.

`sfepy.fem.refine.refine_2_3` (*mesh\_in*, *ed*)

Refines mesh out of triangles by cutting each edge in half and making 4 new finer triangles out of one coarser one.

`sfepy.fem.refine.refine_2_4` (*mesh\_in*, *ed*)

Refines mesh out of quadrilaterals by cutting each edge in half and making 4 new finer quadrilaterals out of one coarser one.

`sfepy.fem.refine.refine_3_4` (*mesh\_in*, *ed*)

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.fem.refine.refine_3_8 (mesh_in, ed, fa)`

Refines hexahedral mesh by cutting each edge in half and making 8 new finer hexahedrons out of one coarser one.

`sfepy.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.fem.region module

**class** `sfepy.fem.region.Region (name, definition, domain, parse_def)`

Region defines a subset of a FE domain.

Created: 31.10.2005

**add\_e** (*other*)

**add\_n** (*other*)

**complete\_description** (*ed, fa, surface\_integral=False*)

Complete the region description by listing edges and faces for each element group.

**Parameters** **ed** : Facets instance

The edge facets.

**fa** : Facets instance

The face facets.

**surface\_integral** : bool

If True, the each region surface facet (edge in 2D, face in 3D) can be listed only in one group. Sub-entities are updated accordingly (vertices in 2D, vertices and edges in 3D).

### Notes

If *surface\_integral* is False, *self.edges*, *self.faces* simply list edge/face indices per group (pointers to *ed.facets*, *fa.facets*) - repetitions among groups are possible.

**contains** (*other*)

Tests only igs for now!!!

**copy** ()

Vertices-based copy.

**create\_mapping** (*kind, ig*)

Create mapping from reference elements to physical elements, given the integration kind ('v' or 's').

This mapping can be used to compute the physical quadrature points.

**Returns** **mapping** : VolumeMapping or SurfaceMapping instance

The requested mapping.

**delete\_groups** (*digs*)

Delete given element groups from the region.

**delete\_zero\_faces** (*eps=1e-14*)

Delete faces with zero area.

**static from\_faces** (*faces, domain, name='region', igs=None, can\_cells=False*)

Create a new region containing given faces.

**Parameters** **faces** : array

The array with indices to *domain.fa*.

**domain** : Domain instance

The domain containing the faces.

**name** : str, optional

The name of the region.

**igs** : list, optional

The allowed element groups. Other groups will be ignored, even though the region might have faces in them - the same effect the 'forbid' flag has.

**can\_cells** : bool, optional

If True, the region can have cells.

**Returns** **obj** : Region instance

The new region.

**static from\_vertices** (*vertices, domain, name='region', igs=None, can\_cells=False, surface\_integral=False*)

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.

**igs** : list, optional

The allowed element groups. Other groups will be ignored, even though the region might have vertices in them - the same effect the 'forbid' flag has.

**can\_cells** : bool, optional

If True, the region can have cells.

**surface\_integral** : bool, optional

If True, then each region surface facet (edge in 2D, face in 3D) can be listed only in one group.

**Returns** **obj** : Region instance



The new region.

**get\_cell\_offsets** ()

**get\_cells** (*ig*, *true\_cells\_only=True*)

Get cells of the region.

Raises ValueError if *true\_cells\_only* is True and the cells are not true cells (e.g. surface integration region).

**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 nodes (*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\_edges** (*ig*)

**get\_faces** (*ig*)

**get\_mirror\_region** ()

**get\_n\_cells** (*ig=None*, *is\_surface=False*)

Get number of region cells.

**Parameters** *ig* : int, optional

The group index. If None, counts from all groups are added together.

**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.

**get\_surface\_entities** (*ig*)

Return either region edges (in 2D) or faces (in 3D) .

**get\_vertices** (*ig*)

**get\_vertices\_of\_cells** (*return\_per\_group=False*)

Return all vertices, that are in some cell of the region.

**Parameters** *return\_per\_group* : bool

If True, return also a dict of vertices per element group.

**has\_cells** ()

**has\_cells\_if\_can** ()

**intersect\_e** (*other*)

**intersect\_n** (*other*)

**iter\_cells** ()

**light\_copy** (*name*, *parse\_def*)

**select\_cells** (*n\_verts*)

Select cells containing at least *n\_verts[ii]* vertices per group *ii*.

**select\_cells\_of\_surface** (*reset=True*)

Select cells corresponding to faces (or edges in 2D).

**set\_cells** (*cells*)

**set\_faces** (*faces, igs=None, can\_cells=False*)

Set region data using given faces. The region description is complete afterwards.

**Parameters** **faces** : array

The array with indices to *domain.fa*.

**igs** : list, optional

The allowed element groups. Other groups will be ignored, even though the region might have faces in them.

**can\_cells** : bool, optional

If True, the region can have cells.

**set\_from\_group** (*ig, vertices, n\_cell*)

Set region to contain the given element group.

**set\_vertices** (*vertices*)

**setup\_face\_indices** (*reset=True*)

Initialize an array (per group) of (iel, ifa) for each face.

**setup\_mirror\_region** ()

Find the corresponding mirror region, set up element mapping.

**sub\_e** (*other*)

**sub\_n** (*other*)

**switch\_cells** (*can\_cells*)

**update\_groups** (*force=False*)

Vertices common to several groups are listed only in all of them - *fa*, *ed.unique\_idx* contain no edge/face duplicates already.

**update\_shape** ()

Update shape of each group according to region vertices, edges, faces and cells.

**update\_vertices** ()

`sfepy.fem.region.get_dependency_graph` (*region\_defs*)

Return a dependency graph and a name-sort name mapping for given region definitions.

`sfepy.fem.region.get_parents` (*selector*)

Given a region selector, return names of regions it is based on.

`sfepy.fem.region.sort_by_dependency` (*graph*)

## sfepy.fem.simplex\_cubature module

Generate simplex quadrature points. Code taken and adapted from pytools/hedge by Andreas Kloeckner.

`sfepy.fem.simplex_cubature.factorial` (*n*)

`sfepy.fem.simplex_cubature.generate_decreasing_nonnegative_tuples_summing_to` (*n*,  
*length*,  
*min=0*,  
*max=None*)

`sfepy.fem.simplex_cubature.generate_permutations` (*original*)

Generate all permutations of the list ‘original’.

Nicked from <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/252178>

`sfepy.fem.simplex_cubature.generate_unique_permutations` (*original*)

Generate all unique permutations of the list ‘original’.

`sfepy.fem.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.fem.simplex_cubature.wandering_element` (*length, wanderer=1, landscape=0*)

## sfepy.fem.state module

Module for handling state variables.

**class** `sfepy.fem.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\_etc** (*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\_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.

## sfepy.fem.utils module

**sfepy.fem.utils.compute\_nodal\_edge\_dirs** (*nodes, region, field, return\_imap=False*)

Nodal edge directions are computed by simple averaging of direction vectors of edges a node is contained in. Edges are assumed to be straight and a node must be on a single edge (a border node) or shared by exactly two edges.

**sfepy.fem.utils.compute\_nodal\_normals** (*nodes, region, field, return\_imap=False*)

Nodal normals are computed by simple averaging of element normals of elements every node is contained in.

**sfepy.fem.utils.extend\_cell\_data** (*data, domain, rname, val=None, is\_surface=False*)

Extend cell data defined in a region to the whole domain.

**Parameters** *data* : array

The data defined in the region.

**domain** : Domain 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 into the cells containing the region surface faces.

**Returns** *edata* : array

The data extended to all domain elements.

**sfepy.fem.utils.get\_edge\_paths** (*graph, mask*)

Get all edge paths in a graph with non-masked vertices. The mask is updated.

`sfepy.fem.utils.get_min_value(dofs)`

Get a reasonable minimal value of DOFs suitable for extending over a whole domain.

`sfepy.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.fem.utils.prepare_remap(indices, n_full)`

Prepare vector for remapping range  $[0, n\_full]$  to its subset given by *indices*.

`sfepy.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.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.fem.variables module

`class sfepy.fem.variables.CloseNodesIterator(field, create_mesh=True, create_graph=True, strategy=None)`

`get_permutation(strategy=None)`

`next()`

`test_permutations(strategy='rcm')`

`class sfepy.fem.variables.FieldVariable(name, kind, field, n_components, order=None, primary_var_name=None, special=None, flags=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_bases()`

Clear base functions, base function gradients and element data dimensions.

`clear_current_group()`

Clear current group data.

`clear_evaluate_cache()`

Clear current evaluate cache.

`create_lcbc_operators(bcs, offset, ts=None, functions=None)`

**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** (*ig, mode='val', region=None, integral=None, integration=None, step=0, time\_derivative=None, is\_trace=False, 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** **ig** : int

The element group index.

**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** : one of 'volume', 'surface', 'surface\_extra'

The term integration type. If None, it is derived from *integral*.

**step** : int, default 0

The time step (0 means current, -1 previous, ...).

**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*, *strategy*='kdtree', *close\_limit*=0.1, *cache*=None, *ret\_cells*=False, *ret\_status*=False, *ret\_ref\_coors*=False)

Evaluate self in the given physical coordinates. Convenience wrapper around `Field.evaluate_at()`, see its docstring for more details.

**get\_approximation** (*ig*)

**get\_component\_indices** ()

Return indices of variable components according to current term evaluation mode.

**Returns indx** : list of tuples

The list of  $(ii, slice(ii, ii + 1))$  of the variable components. The first item is the index itself, the second item is a convenience slice to index components of material parameters.

**get\_data\_shape** (*ig*, *integral*, *integration*='volume', *region\_name*=None)

Get element data dimensions for given approximation.

**Parameters ig** : int

The element group index.

**integral** : Integral instance

The integral describing used numerical quadrature.

**integration** : 'volume', 'surface', 'surface\_extra' or 'point'

The term integration type.

**region\_name** : str

The name of surface region, required when *shape\_kind* is 'surface'.

**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, ig, active=False, is\_trace=False*)

Get active dof connectivity of a variable.

Note that primary and dual variables must have same Region!

**get\_dof\_info** (*active=False*)

**get\_element\_diameters** (*cells, mode, square=False*)

Get diameters of selected elements.

**get\_element\_zeros** ()

Return array of zeros with correct shape and type for term evaluation.

**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** (*ig, region, integral, integration, get\_saved=False, return\_key=False*)

Get the reference element mapping of the underlying field.

**See Also:**

`sfepy.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, igs=None, reshape=True, step=0*)

**grad** (*ic=None, ider=None*)

Return base function gradient (space elements) values in quadrature points.

**Parameters** *ic* : int, optional

The index of variable component.

*ider* : int, optional

The spatial derivative index. If not given, the whole gradient is returned.

**grad\_qp** (*ic=None, ider=None*)

Return variable gradient evaluated in quadrature points.

**Parameters** *ic* : int, optional

The index of variable component.

**ider** : int, optional

The spatial derivative index. If not given, the whole gradient is returned.

**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.

**iter\_dofs** ()

Iterate over element DOFs (DOF by DOF).

**save\_as\_mesh** (*filename*)

Save the field mesh and the variable values into a file for visualization. Only the vertex values are stored.

**set\_current\_group** (*geo\_key, ig*)

Set current group data, initialize current DOF counter to *None*.

The current group data are the approximation, element data dimensions, base functions and base function gradients.

**set\_data\_from\_qp** (*data\_qp, integral, step=0*)

Set DOFs of variable using values in quadrature points corresponding to the given integral.

**set\_from\_mesh\_vertices** (*data*)

Set the variable using values at the mesh vertices.

**set\_from\_other** (*other, strategy='projection', search\_strategy='kdtree', ordering\_strategy='rcm', 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  $L^2$  orthogonal projection, 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.

If the other variable uses the same field mesh, only deformed slightly, it is advisable to provide directly the node ids as a hint where to start searching for a containing element; the order of nodes does not matter then.

Otherwise (large deformation, unrelated meshes, ...) there are basically two ways: a) query each node (its coordinates) using a KDTree of the other nodes - this completely disregards the connectivity information; b) iterate the mesh nodes so that the subsequent ones are close to each other - then also the elements of the other mesh should be close to each other: the previous one can be used as a start for the directional neighbour element crawling to the target point.

Not sure which way is faster, depends on implementation efficiency and the particular meshes.

**setup\_adof\_conns** (*adof\_conns, adi*)  
 Translate dof connectivity of the variable to active dofs.

**Active dof connectivity key:** (variable.name, region.name, type, ig)

**setup\_bases** (*geo\_key, ig, geo, integral, shape\_kind='volume'*)  
 Setup and cache base functions and base function gradients for given geometry. Also cache element data dimensions.

**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.

**val** (*ic=None*)  
 Return base function values in quadrature points.

**Parameters** *ic* : int, optional  
 The index of variable component.

**val\_qp** (*ic=None*)  
 Return variable evaluated in quadrature points.

**Parameters** *ic* : int, optional  
 The index of variable component.

**class** sfepy.fem.variables.**Variable** (*name, kind, n\_components, 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\_full\_state** (*step=0*)

**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, indx=None, step=0, preserve\_caches=False*)

Set data (vector of DOF values) of the variable.

**Parameters** **data** : array

The vector of DOF values.

**indx** : int, optional

If given, *data[indx]* is used.

**step** : int, optional

The time history step, 0 (default) = current.

**preserve\_caches** : bool

If True, do not invalidate evaluate caches of the variable.

**time\_update** (*ts, functions*)

Implemented in subclasses.

**class** `sfePy.fem.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** (*ebscs, epbcs, ts, functions, problem=None*)

Create the mapping of active DOFs from/to all DOFs for all state variables.

**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\_idx** (*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*)

Make a full DOF vector satisfying E(P)BCs from a reduced DOF vector.

Passing a *force\_value* overrides the EBC values.

**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\_data\_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\_adof\_conns** ()  
Translate dofs to active dofs. Active dof connectivity key = (variable.name, region.name, type, ig)

**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** (*lcbs, ts=None, functions=None*)  
Prepare linear combination BC operator matrix.

**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*)  
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.fem.variables.create_adof_conn` (*eq, dc, indx*)  
Given a dof connectivity and equation mapping, create the active dof connectivity.

## sfepy.fem.extmods.\_fmfield module

## sfepy.fem.extmods.assemble module

Low level finite element assembling functions.

`sfepy.fem.extmods.assemble.assemble_matrix` ()  
`sfepy.fem.extmods.assemble.assemble_matrix_complex` ()  
`sfepy.fem.extmods.assemble.assemble_vector` ()  
`sfepy.fem.extmods.assemble.assemble_vector_complex` ()

## sfepy.fem.extmods.bases module

Polynomial base functions and related utilities.

`sfepy.fem.extmods.bases.eval_lagrange_simplex` ()  
Evaluate Lagrange base polynomials in given points on simplex domain.

**Parameters** `coors` : array

The coordinates of the points, shape  $(n\_coord, dim)$ .

**mtx\_i** : array

The inverse of simplex coordinates matrix, shape  $(dim + 1, dim + 1)$ .

**nodes** : array

The description of finite element nodes, shape  $(n\_nod, dim + 1)$ .

**order** : int

The polynomial order.

**diff** : bool

If True, return base function derivatives.

**eps** : float

The tolerance for snapping out-of-simplex point back to the simplex.

**check\_errors** : bool

If True, raise ValueError if a barycentric coordinate is outside the snap interval  $[-eps, 1 + eps]$ .

**Returns** **out** : array

The evaluated base functions, shape  $(n\_coord, 1 \text{ or } dim, n\_nod)$ .

`sfepy.fem.extmods.bases.eval_lagrange_tensor_product()`  
Evaluate Lagrange base polynomials in given points on tensor product domain.

**Parameters** **coors** : array

The coordinates of the points, shape  $(n\_coord, dim)$ .

**mtx\_i** : array

The inverse of 1D simplex coordinates matrix, shape  $(2, 2)$ .

**nodes** : array

The description of finite element nodes, shape  $(n\_nod, 2 * dim)$ .

**order** : int

The polynomial order.

**diff** : bool

If True, return base function derivatives.

**eps** : float

The tolerance for snapping out-of-simplex point back to the simplex.

**check\_errors** : bool

If True, raise ValueError if a barycentric coordinate is outside the snap interval  $[-eps, 1 + eps]$ .

**Returns** **out** : array

The evaluated base functions, shape  $(n\_coord, 1 \text{ or } dim, n\_nod)$ .

`sfepy.fem.extmods.bases.evaluate_in_rc()`  
Evaluate source field DOF values in the given reference element coordinates using the given interpolation.

1. Evaluate base functions in the reference coordinates.

2. Interpolate source values using the base functions.

Interpolation uses field approximation connectivity.

`sfepy.fem.extmods.bases.find_ref_coors()`

Find reference element coordinates corresponding to physical coordinates *coors*.

This function works only with a geometry mesh (order 1 connectivity), not a field mesh! The polynomial space arguments have to correspond to that.

**Returns** `ref_coors` : array

The reference coordinates.

`cells` : array

The array of (ig, iel) corresponding to the reference coordinates.

`status` : array

The status array: 0 is success, 1 means the point is extrapolated within *close\_limit*, 2 extrapolated outside *close\_limit* and 3 extrapolated with no extrapolation allowed.

`sfepy.fem.extmods.bases.get_barycentric_coors()`

Get barycentric (area in 2D, volume in 3D) coordinates of points.

**Parameters** `coors` : array

The coordinates of the points, shape  $(n\_coord, dim)$ .

`mtx_i` : array

The inverse of simplex coordinates matrix, shape  $(dim + 1, dim + 1)$ .

`eps` : float

The tolerance for snapping out-of-simplex point back to the simplex.

`check_errors` : bool

If True, raise `ValueError` if a barycentric coordinate is outside the snap interval  $[-eps, 1 + eps]$ .

**Returns** `bc` : array

The barycentric coordinates, shape  $(n\_coord, dim + 1)$ . Then reference element coordinates  $xi = dot(bc, ref\_coors)$ .

## sfepy.fem.extmods.lobatto\_bases module

Interface to Lobatto bases.

`sfepy.fem.extmods.lobatto_bases.eval_lobattoid()`

Evaluate 1D Lobatto functions of the given order in given points.

`sfepy.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.fem.extmods.mappings module**

Low level reference mapping functionality.

**class** `sfepy.fem.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
n_el
n_ep
n_qp
normal
ps
qp
shape
volume

```

**sfepy.fem.extmods.mesh module**

Low level mesh functions employing element connectivity.

`sfepy.fem.extmods.mesh.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.fem.extmods.mesh.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.fem.extmods.mesh.orient_elements()`

Swap element nodes so that its volume is positive.

## 7.7.7 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 according to options.

**plot\_band\_gaps** (*coefs*)

**plot\_dispersion** (*coefs*)

**static process\_options** (*options*)

Application options setup. Sets default values for missing non-compulsory options.

**static process\_options\_pv** (*options*)

Application options setup for phase velocity computation. Sets default values for missing non-compulsory options.

**setup\_options** ()

`sfepy.homogenization.band_gaps_app.plot_eigs`(*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, ii, f0, f1, kind, kind_desc, gmin, gmax,
                                             plot_range, plot_rsc)
```

Plot a single band gap as a rectangle.

```
sfepy.homogenization.band_gaps_app.plot_gaps(fig_num, plot_rsc, gaps, kinds, 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.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)
```

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*.

```
to_file_txt(filename, names, format)
```

```
to_latex(attr_name, dim, style='table', format='%f', step=None)
```

**sfepy.homogenization.coefs\_base module**

```
class sfepy.homogenization.coefs_base.CoeffDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffDimDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffDimSym(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffDummy(name, problem, kwargs)
    Dummy class serving for computing and returning its requirements.
class sfepy.homogenization.coefs_base.CoeffEval(name, problem, kwargs)
    Evaluate expression.
class sfepy.homogenization.coefs_base.CoeffFMSym(name, problem, kwargs)
    Fading memory sym coefficients.
class sfepy.homogenization.coefs_base.CoeffFMSymSym(name, problem, kwargs)
    Fading memory sym x sym coefficients.
class sfepy.homogenization.coefs_base.CoeffN(name, problem, kwargs)

    static set_variables_default(variables, ir, set_var, data)
class sfepy.homogenization.coefs_base.CoeffNN(name, problem, kwargs)

    static set_variables_default(variables, ir, ic, mode, set_var, data)
class sfepy.homogenization.coefs_base.CoeffNone(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffOne(name, problem, kwargs)

    static set_variables_default(variables, set_var, data)
class sfepy.homogenization.coefs_base.CoeffSum(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffSym(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CoeffSymSym(name, problem, kwargs)

    static set_variables_default(variables, ir, ic, mode, set_var, data)
class sfepy.homogenization.coefs_base.CopyData(name, problem, kwargs)
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.CorrMiniApp(name, problem, kwargs)
```

```

    get_dump_name ()
    get_dump_name_base ()
    get_output (corr_sol, is_dump=False, extend=True, variables=None)
    get_save_name ()
    get_save_name_base ()
    save (state, problem, variables=None)
    setup_output (save_format=None,          dump_format=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.

    iter_solutions ()
class sfepy.homogenization.coefs_base.MinAppBase (name, problem, kwargs)

    static any_from_conf (name, problem, kwargs)
    init_solvers (problem)
        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)
     $G = B \cdot A_I \cdot B^T$  or  $B \cdot A_I \cdot B^T + D$ 
```

```
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, dump_name, save_name, problem=None, vec_g=None)
```

```
save (dump_name, save_name, vec_u, vec_p, vec_dp, ts, problem)
```

1.saves raw correctors into hdf5 files (filename)

2.saves correctors transformed to output for visualization

```
setup_equations (equations, problem=None)
```

Set equations, update boundary conditions and materials.

```
verify_correctors (sign, state0, filename, problem=None)
```

```
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\_elastic module

```
class sfepy.homogenization.coefs_elastic.CorrectorsPermeability (name, problem,
                                                                kwargs)
```

```
class sfepy.homogenization.coefs_elastic.GBarCoef (name, problem, kwargs)
```

Asymptotic Barenblatt coefficient.

```
data = [p{infy}]
```

Note:

solving “dw\_diffusion.il.Y3( m.K, qc, pc ) = 0” solve, in fact “ $C p^{\infty} = \hat{C} \hat{\pi}$ ” with the result “ $\hat{p}^{\infty}$ ”, where the rhs comes from E(P)BC. - it is preferable to computing directly by “ $\hat{p}^{\infty} = \hat{C}^{-1} \text{strip}(\hat{C} \hat{\pi})$ ”, as it checks explicitly the rezidual.

```
class sfepy.homogenization.coefs_elastic.GPlusCoef (name, problem, kwargs)
```

```
get_filename (data)
```

```
class sfepy.homogenization.coefs_elastic.PressureRHSVector (name, problem, kwargs)
```

```
class sfepy.homogenization.coefs_elastic.RBiotCoef (name, problem, kwargs)
```

Homogenized fading memory Biot-like coefficient.

```
get_filename (data, ir, ic)
```

```
get_variables (problem, io, step, data, mode)
```

```
class sfepy.homogenization.coefs_elastic.TCorrectorsPressureViaPressureEVP (name,
                                                                                   prob-
                                                                                   lem,
                                                                                   kwargs)
```

```

    get_dump_name_base()
    get_save_name_base()
class sfepy.homogenization.coefs_elastic.TCorrectorsRSViaPressureEVP (name,
                                                                    problem,
                                                                    kwargs)

    get_dump_name_base()
    get_save_name_base()
sfepy.homogenization.coefs_elastic.eval_boundary_diff_vel_grad(problem, uc,
                                                                pc, equation,
                                                                region_name,
                                                                pi=None)

```

### 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 re-
    quired 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** **eigsolver** : 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.

**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 eigenmo-
            menta.

        progress_bar : bool
            If True, use a progress bar to show computation progress.

    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,  
progress_bar=None)`

Compute the eigenmomenta corresponding to given eigenvectors.

`sfepy.homogenization.coefs_phononic.cut_freq_range(freq_range, eigs, valid,  
freq_margins, eig_range,  
fixed_freq_range, freq_eps)`

Cut off masked resonance frequencies. Margins are preserved, like no resonances were cut.

**Returns** `freq_range` : array

The new range of frequencies.

**freq\_range\_margins** : array

The range of frequencies with prepended/appended margins equal to *fixed\_freq\_range* if it is not None.

`sfepy.homogenization.coefs_phononic.describe_gaps(gaps)`

`sfepy.homogenization.coefs_phononic.detect_band_gaps(mass, freq_info, opts,  
gap_kind='normal',  
mtx_b=None)`

Detect band gaps given solution to eigenproblem (eigs, eig\_vectors). Only valid resonance frequencies (e.i. those for which corresponding eigenmomenta are above a given threshold) are taken into account.

## Notes

- make `freq_eps` relative to `]f0, f1[` size?

`sfepy.homogenization.coefs_phononic.find_zero(f0, f1, callback, freq_eps, zero_eps, mode)`

For `f` in `]f0, f1[` find frequency `f` for which either the smallest (`mode = 0`) or the largest (`mode = 1`) eigenvalue of problem `P` given by `callback` is zero.

**Returns** `flag` : 0, 1, or 2

The flag, see Notes below.

**frequency** : float

The found frequency.

**eigenvalue** : float

The eigenvalue corresponding to the found frequency.

## Notes

Meaning of the return value combinations:

mode	flag	meaning
0, 1	0	eigenvalue $\rightarrow 0$ for $f$ in $]f_0, f_1[$
0	1	$f \rightarrow f_1$ , smallest eigenvalue $< 0$
0	2	$f \rightarrow f_0$ , smallest eigenvalue $> 0$ and $\rightarrow -\infty$
1	1	$f \rightarrow f_1$ , largest eigenvalue $< 0$ and $\rightarrow +\infty$
1	2	$f \rightarrow f_0$ , largest eigenvalue $> 0$

`sfepy.homogenization.coefs_phononic.get_callback` (*mass*, *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  $\omega^2 M w = \eta B w$

`sfepy.homogenization.coefs_phononic.get_log_freqs` (*f0*, *f1*, *df*, *freq\_eps*, *n\_point\_min*,  
*n\_point\_max*)

Get logging frequencies.

The frequencies get denser towards the interval boundaries.

`sfepy.homogenization.coefs_phononic.get_ranges` (*freq\_range*, *eigs*)

Get an eigenvalue range slice and a corresponding initial frequency range within a given frequency range.

`sfepy.homogenization.coefs_phononic.split_chunks` (*indx*)

Split index vector to chunks of consecutive numbers.

## sfepy.homogenization.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  $c[i]$ ,  $i = 0, 1, \dots$

$$c_0 \equiv c[0], c_{e0} \equiv c_0 c_0^e,$$

$$c(t) \approx c_0 d(t) \approx c_0 e(t) = c_{e0} e_n(t),$$

where  $d(0) = e_n(0) = 1$ ,  $d$  is the synchronous decay and  $e$  its exponential approximation,  $e = c_0^e \exp(-c_1^e t)$ .

`diff_dt` (*use\_exp=False*)

The derivative of the kernel w.r.t. time.

`get_exp` ()

Get the exponential synchronous decay kernel approximation.

`get_full` ()

Get the original (full) kernel.

**int\_dt** (*use\_exp=False*)

The integral of the kernel in time.

`sfepy.homogenization.convolution.approximate_exponential` (*x*, *y*)

Approximate  $y = f(x)$  by  $y_a = c_1 \exp(-c_2 x)$ .

Initial guess is given by assuming *y* has already the required exponential form.

`sfepy.homogenization.convolution.compute_mean_decay` (*coef*)

Compute mean decay approximation of a non-scalar fading memory coefficient.

`sfepy.homogenization.convolution.eval_exponential` (*coefs*, *x*)

`sfepy.homogenization.convolution.fit_exponential` (*x*, *y*, *return\_coefs=False*)

Evaluate  $y = f(x)$  after approximating *f* by an exponential.

### `sfepy.homogenization.engine` module

**class** `sfepy.homogenization.engine.HomogenizationEngine` (*problem*, *options*,  
*app\_options=None*,  
*volume=None*, *output\_prefix='he:'*, *\*\*kwargs*)

**call** (*ret\_all=False*)

**compute\_requirements** (*requirements*, *dependencies*, *store*)

**static process\_options** (*options*)

**setup\_options** (*app\_options=None*)

`sfepy.homogenization.engine.insert_sub_reqs` (*reqs*, *levels*, *req\_info*)

Recursively build all requirements in correct order.

### `sfepy.homogenization.homogen_app` module

**class** `sfepy.homogenization.homogen_app.HomogenizationApp` (*conf*, *options*, *output\_prefix*,  
*\*\*kwargs*)

**call** (*verbose=False*, *ret\_all=None*)

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.

**Returns** **coefs** : Coefficients instance

The homogenized coefficients.

**dependencies** : dict

The dependencies, if *ret\_all* is True.

**static process\_options** (*options*)

Application options setup. Sets default values for missing non-compulsory options.

```
setup_options()
```

```
class sfepy.homogenization.homogen_app.Volume(name, problem, kwargs)
```

```
sfepy.homogenization.homogen_app.get_volume_from_options(options, problem)
```

### sfepy.homogenization.micmac module

```
sfepy.homogenization.micmac.get_correctors_from_file(coefs_filename='coefs.h5',
                                                    dump_names=None)
```

```
sfepy.homogenization.micmac.get_homog_coefs_linear(ts, coor, mode, micro_filename=None, regenerate=False, coefs_filename=None)
```

### sfepy.homogenization.recovery module

```
sfepy.homogenization.recovery.add_strain_rs(corrs_rs, strain, vu, dim, iel, out=None)
```

```
sfepy.homogenization.recovery.add_stress_p(out, pb, integral, region, vp, data)
```

```
sfepy.homogenization.recovery.combine_scalar_grad(corrs, grad, vn, ii, shift_coors=None)
```

$$\eta_k \partial_k^x p$$

or

$$(y_k + \eta_k) \partial_k^x p$$

```
sfepy.homogenization.recovery.compute_mac_stress_part(pb, integral, region, material,
                                                    vu, mac_strain)
```

```
sfepy.homogenization.recovery.compute_micro_u(corrs, strain, vu, dim, out=None)
```

Micro displacements.

$$u^1 = \chi^{ij} e_{ij}^x(u^0)$$

```
sfepy.homogenization.recovery.compute_p_corr_steady(corrs_pressure, pressure, vp, iel)
```

$\widetilde{\text{depi}}^{\text{P,p}}$

```
sfepy.homogenization.recovery.compute_p_corr_time(corrs_rs, dstrains, corrs_pressure,
                                                    pressures, vdp, dim, iel, ts)
```

$$\sum_{\{ij\}} \int_0^t \int_{\mathcal{D}} \widetilde{\text{depi}}^{\{ij\}}(t-s), \int_{\mathcal{D}} s) e_{\{ij\}}(\text{bm}\{u\}(s)), ds + \int_0^t \int_{\mathcal{D}} \widetilde{\text{depi}}^{\text{P}}(t-s), p(s), ds$$

```
sfe.py.homogenization.recovery.compute_p_from_macro(p_grad, coor, iel, centre=None,
                                                    extdim=0)
```

Macro-induced pressure.

$$\partial_j^x p(y_j - y_j^c)$$

```
sfe.py.homogenization.recovery.compute_stress_strain_u(pb, integral, region, material,
                                                       vu, data)
```

```
sfe.py.homogenization.recovery.compute_u_corr_steady(corrs_rs, strain, vu, dim, iel)
```

$$\sum_{ij} \omega^{ij} e_{ij}(\mathbf{u})$$

### Notes

- iel = element number

```
sfe.py.homogenization.recovery.compute_u_corr_time(corrs_rs, dstrains, corrs_pressure,
                                                    pressures, vu, dim, iel, ts)
```

$\text{sum}_{\{ij\}} \left[ \int_0^t \omega^{\{ij\}}(t-s) \left\{ \frac{d}{ds} e_{\{ij\}}(\mathbf{u}(s)) \right\} ds \right] + \int_0^t \omega^{\{ij\}}(t-s) P(t-s) ds$

```
sfe.py.homogenization.recovery.compute_u_from_macro(strain, coor, iel, centre=None)
```

Macro-induced displacements.

$$e_{ij}^x(\mathbf{u})(y_j - y_j^c)$$

```
sfe.py.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]

```
sfe.py.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` (*ig, 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=''*)

`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*)

$P_i^{rs} = y_s \delta_{ir}$  for  $r = ir, s = ic$ .

`sfepy.homogenization.utils.coor_to_sym` (*ir, ic, dim*)

`sfepy.homogenization.utils.create_pis` (*problem, var\_name*)

$P_i^{rs} = y_s \delta_{ir}$ ,  $ul\{y\}$  in Y coordinates.

`sfepy.homogenization.utils.create_scalar_pis` (*problem, var\_name*)

$P_i^k = y_k$ ,  $ul\{y\}$  in Y coordinates.

`sfepy.homogenization.utils.define_box_regions` (*dim, lbn, rtf=None, eps=0.001, can\_cells=None*)

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.

**can\_cells** : bool, optional

If given, use its value for the 'can\_cells' region flag.

**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_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

## 7.7.8 sfepy.interactive package

### sfepy.interactive.session module

Functions for setting up interactive sessions.

`sfepy.interactive.session.init_session` (*ipython=None, message=None, quiet=False, silent=False, is\_viewer=True, is\_wx=True, argv=[*  
*]*)

Initialize embedded IPython or Python session.

## 7.7.9 sfepy.linalg package

### 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, precondition=None, i\_max=None, eps\_r=1e-10, shift=None, select\_indices=None, verbose=False, report\_step=10*)

Make several iterations of the conjugate gradients and estimate so the eigenvalues of a (sparse SPD) matrix (Lanczos algorithm).

**Parameters** **mtx** : spmatrix or array

The sparse matrix  $A$ .

**precond** : spmatrix or array, optional

The preconditioner matrix. Any object that can be multiplied by vector can be passed.

**i\_max** : int

The maximum number of the Lanczos algorithm iterations.

**eps\_r** : float

The relative stopping tolerance.

**shift** : float, optional

Eigenvalue shift for non-SPD matrices. If negative, the shift is computed as  $|shift| ||A||_{\infty}$ .

**select\_indices** : (min, max), optional

If given, computed only the eigenvalues with indices  $min \leq i \leq max$ .

**verbose** : bool

Verbosity control.

**report\_step** : int

If *verbose* is True, report in every *report\_step*-th step.

**Returns** **vec** : array

The approximate solution to the linear system.

**n\_it** : int

The number of CG iterations used.

**norm\_rs** : array

Convergence history of residual norms.

**eigs** : array

The approximate eigenvalues sorted in ascending order.

`sfepy.linalg.eigen.sym_tri_eigen` (*diags*, *select\_indices=None*)

Compute eigenvalues of a symmetric tridiagonal matrix using `scipy.linalg.eigvals_banded()`.

## sfepy.linalg.geometry module

`sfepy.linalg.geometry.barycentric_coors` (*coors*, *s\_coors*)

Get barycentric (area in 2D, volume in 3D) coordinates of points with coordinates *coors* w.r.t. the simplex given by *s\_coors*.

**Returns** **bc** : array

The barycentric coordinates. Then reference element coordinates  $xi = dot(bc.T, ref\_coors)$ .

`sfepy.linalg.geometry.flag_points_in_polygon2d` (*polygon*, *coors*)

Test if points are in a 2D polygon.

**Parameters** **polygon** : array, (:, 2)

The polygon coordinates.

**coors**: array, (:, 2) :

The coordinates of points.

**Returns** **flag** : bool array

The flag that is True for points that are in the polygon.

## Notes

This is a semi-vectorized version of [1].

[1] PNPOLY - Point Inclusion in Polygon Test, W. Randolph Franklin (WRF)

`sfepy.linalg.geometry.get_coors_in_ball` (*coors*, *centre*, *radius*, *inside=True*)

Return indices of coordinates inside or outside a ball given by centre and radius.

## Notes

All float comparisons are done using `<=` or `>=` 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 then *force\_inside\_eps* so that they are inside the simplices at the distance given by *force\_inside\_eps*. It is ignored for edges.

**Returns** *centres* : array

The circumcentre coordinates as an array of shape (*n\_s*, *dim*).

`sfepy.linalg.geometry.inverse_element_mapping` (*coors*, *e\_coors*, *eval\_base*, *ref\_coors*, *suppress\_errors=False*)

Given spatial element coordinates, find the inverse mapping for points with coordinats  $X = X(xi)$ , 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.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  $(n_1, \dots, n_r) + out\_item\_shape$ . The *out\_item\_shape* must be compatible with the *fun*.

sfepy.linalg.utils.**argsort\_rows** (*seq*)

Returns an index array that sorts the sequence *seq*. Works along rows if *seq* is two-dimensional.

sfepy.linalg.utils.**assemble1d** (*ar\_out, indx, ar\_in*)

Perform  $ar\_out[indx] += ar\_in$ , where items of *ar\_in* corresponding to duplicate indices in *indx* are summed together.

sfepy.linalg.utils.**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  $[0 \leq i < a, 0 \leq j < b, 0 \leq k < c, \dots]$  for all *i, j, k, ...*

Examples: In [9]: `list(cycle([3, 2]))` Out[9]:  `[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]`

In [14]: `list(cycle([3, 4]))`  `[[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1], [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]`

sfepy.linalg.utils.**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  $(n\_l, \dots, n\_r, ?, ?)$  the arrays are first reshaped to  $(n\_l * \dots * n\_r, ?, ?)$ , then the dot is performed, and finally the shape is restored to  $(n\_l, \dots, n\_r, ?, ?)$ .

`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.mini_newton` (*fun*, *x0*, *dfun*, *i\_max=100*, *eps=1e-08*)

`sfepy.linalg.utils.norm_l2_along_axis` (*ar*, *axis=1*, *n\_item=None*, *squared=False*)

Compute l2 norm of rows (*axis=1*) or columns (*axis=0*) of a 2D array.

*n\_item* ... use only the first *n\_item* columns/rows squared ... if True, return the norm squared

`sfepy.linalg.utils.normalize_vectors` (*vecs*, *eps=1e-08*)

Normalize an array of vectors in place.

**Parameters** *vecs* : array

The 2D array of vectors in rows.

*eps* : float

The tolerance for considering a vector to have zero norm. Such vectors are left unchanged.

`sfepy.linalg.utils.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()*.

## sfepy.linalg.extmods.crcm module

`sfepy.linalg.extmods.crcm.permute_in_place` ()

Permute a graph (= CSR sparse matrix with boolean values) in place, given a permutation vector.



```
sfepy.linalg.extmods.crcm.rcm()
```

Generate the reversed Cuthil-McKee permutation for a CSR matrix.

## 7.7.10 sfepy.mechanics package

### sfepy.mechanics.contact\_planes module

```
class sfepy.mechanics.contact_planes.ContactPlane(anchor, normal, bounds)
```

```
    get_distance(points)
```

```
    mask_points(points)
```

```
sfepy.mechanics.contact_planes.plot_points(ax, points, marker, **kwargs)
```

```
sfepy.mechanics.contact_planes.plot_polygon(ax, polygon)
```

### sfepy.mechanics.elastic\_constants module

### sfepy.mechanics.friction module

Friction-slip model formulated as the implicit complementarity problem.

To integrate over a (dual) mesh, one needs:

- coordinates of element vertices
- element connectivity
- local base for each element \* constant in each sub-triangle of the dual mesh

Data for each dual element:

- connectivity of its sub-triangles
- base directions  $t_1, t_2$

Normal stresses:

- Assemble the rezidual and apply the LCBC operator described below.

Solution in  $\hat{V}_h$ :

- construct a restriction operator via LCBC just like in the no-penetration case
- use the substitution:  $u_1 = n_1 * w$   $u_2 = n_2 * w$   $u_3 = n_3 * w$  The new DOF is  $w$ .
- for the record, no-penetration does:  $w_1 = -(1 / n_1) * (u_2 * n_2 + u_3 * n_3)$   $w_2 = u_2$   $w_3 = u_3$

```
class sfepy.mechanics.friction.DualMesh(region)
```

Dual mesh corresponding to a (surface) region.

```
    create_friction_bcs(dof_name)
```

Fix friction DOFs on surface boundary edges, i.e. that are not shared by two friction surface faces.

```
    describe_dual_surface(surface)
```

```
    iter_groups(igs=None)
```

Domain-like functionality.

```
    save(filename)
```

```
    save_axes(filename)
```

`sfepy.mechanics.friction.edge_data_to_output` (*coors, conn, e\_sort, data*)

## **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_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.

$$\mathbf{D}_{(2D)} = \begin{bmatrix} \tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 \\ \tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

$$D_{(3D)} = \begin{bmatrix} \tilde{\lambda} + 2\mu & \tilde{\lambda} & \tilde{\lambda} & 0 & 0 & 0 \\ \tilde{\lambda} & \tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 & 0 & 0 \\ \tilde{\lambda} & \tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}$$

where

$$\tilde{\lambda} = -\frac{2}{3}\mu$$

[illegible]

Compute stiffness tensor corresponding to Young's modulus and Poisson's ratio.

[illegible]

Compute stiffness tensor corresponding to Young's modulus and Poisson's ratio for mixed formulation.

## 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(ig, field, region, integral)`

Describe membrane geometry in a given region.

**Parameters** `ig` : int

The element group index.

`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.tensors module

Functions to compute some tensor-related quantities usual in continuum mechanics.

**class** `sfepy.mechanics.tensors.StressTransform` (*def\_grad*, *jacobian=None*)

Encapsulates functions to convert various stress tensors in the symmetric storage given the deformation state.

**get\_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_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 its name.

## Examples

Construct some units:

```
>>> from sfePy.mechanics.units import Unit
>>> unit = Unit('mm')
>>> print unit
Unit:mm
coef:
  0.001
name:
  mm
prefix:
  m
prefix_length:
  1
unit:
  m
>>> unit = Unit('kg')
>>> print unit
Unit:kg
coef:
  1000.0
name:
  kg
prefix:
  k
```

```
prefix_length:
    1
unit:
    g
```

Get prefixes for a coefficient:

```
>>> Unit.get_prefix(100.0)
('d', 10.0)
>>> Unit.get_prefix(100.0, omit=('d',))
('k', 0.10000000000000001)
```

**static get\_prefix** (*coef*, *bias*=0.1, *omit*=())

Get the prefix and numerical multiplier corresponding to a numerical coefficient, omitting prefixes in omit tuple.

`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.

## 7.7.11 sfepy.mesh package

### sfepy.mesh.femlab module

```
sfepy.mesh.femlab.axissym (a, b, c)
    a, b ... line. c ... center

sfepy.mesh.femlab.conv (x)

sfepy.mesh.femlab.conv2 (x)

sfepy.mesh.femlab.conv3 (x)

sfepy.mesh.femlab.conv4 (a, b, c)

sfepy.mesh.femlab.crossproduct (a, b)

sfepy.mesh.femlab.curve2 (f, p1, p2)

sfepy.mesh.femlab.femlabline (f, n, p)

sfepy.mesh.femlab.femlabsurface (f, n, points)

sfepy.mesh.femlab.femlabsurface3 (f, n, p)

sfepy.mesh.femlab.femlabsurface3_old (f, n, p)

sfepy.mesh.femlab.getnormal (a, b, c)

sfepy.mesh.femlab.getp3 (p1, p2, p3)

sfepy.mesh.femlab.norm (x)

sfepy.mesh.femlab.norm2 (x)

sfepy.mesh.femlab.tri3d (points)

sfepy.mesh.femlab.triangulate (points)

sfepy.mesh.femlab.triangulate2 (points)

sfepy.mesh.femlab.triangulate_old (p)
```

```
sfe.py.mesh.femlab.vec(a, b)
```

```
sfe.py.mesh.femlab.write_femlab(g, filename, export0D=False, export1D=False, export2D=False,
                                export3D=False)
```

## sfe.py.mesh.geom\_tools module

**class** `sfe.py.mesh.geom_tools.geometry` (*dim=3*)

The geometry is given by a sets of points (d0), lines (d1), surfaces (d2) and volumes (d3). A lines are constructed from 2 points, a surface from any number of lines, a volume from any number of surfaces.

Physical volumes are contruted from any number of volumes.

The self.d0, self.d1, self.d2 and self.d3 are dictionaries holding a map  
geometry element number -> instance of point,line,surface of volume

### Examples

To get all the points which define a surface 5, use:

```
self.d2[5].getpoints()
```

This would give you a list [...] of point() instances.

```
addline (n, l)
```

```
l=[p1,p2]
```

```
addlines (ls, off=1)
```

```
ls=[l1, l2, ...]
```

```
addphysicalsurface (n, surfacelist)
```

```
surfacelist=[s1,s2,s3,...]
```

```
addphysicalvolume (n, volumelist)
```

```
volumelist=[v1,v2,v3,...]
```

```
addpoint (n, p)
```

```
p=[x,y,z]
```

```
addpoints (ps, off=1)
```

```
ps=[p1, p2, ...]
```

```
addsurface (n, s, is_hole=False)
```

```
s=[l1,l2,l3,...]
```

```
addsurfaces (ss, off=1)
```

```
s=[s1,s2,s3,...]
```

```
addvolume (n, v)
```

```
v=[s1,s2,s3,...]
```

```
addvolumes (vs, off=1)
```

```
v=[v1,v2,v3,...]
```

```
static from_gmsh_file (filename)
```

Import geometry - Gmsh geometry format.

**Parameters** `filename` : string

file name

**Returns** **geo** : geometry  
geometry description

**getBCnum** (*snum*)

**leaveonlyphysicalsurfaces** ()

**leaveonlyphysicalvolumes** ()

**printinfo** (*verbose=False*)

**splitlines** (*ls, n*)

**to\_poly\_file** (*filename*)

Export geometry to poly format (tetgen and triangle geometry format).

**Parameters** **geo** : geometry

geometry description

**filename** : string

file name

**class** sfepy.mesh.geom\_tools.**geomobject**

**getn** ()

**class** sfepy.mesh.geom\_tools.**line** (*g, n, l*)

**getpoints** ()

**class** sfepy.mesh.geom\_tools.**physicalsurface** (*g, n, s*)

**getsurfaces** ()

**class** sfepy.mesh.geom\_tools.**physicalvolume** (*g, n, v*)

**getvolumes** ()

**class** sfepy.mesh.geom\_tools.**point** (*g, n, p*)

**getstr** ()

**getxyz** ()

**class** sfepy.mesh.geom\_tools.**surface** (*g, n, s, is\_hole=False*)

**getcenterpoint** ()

**getholepoints** ()

**getinsidepoint** ()

**getlines** ()

**getpoints** ()

**separate** (*s*)

**class** sfepy.mesh.geom\_tools.**volume** (*g, n, v*)

`getinsidepoint()`

`getsurfaces()`

## **sfepy.mesh.mesh\_generators module**

`sfepy.mesh.mesh_generators.gen_block_mesh(dims, shape, centre, mat_id=0, name='block', verbose=True)`

Generate a 2D or 3D block mesh. The dimension is determined by the length 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_goem(geo, a=None, quadratic=False, verbose=True,
                                                refine=False, polyfile_name='./meshgen.poly', out='mesh',
                                                **kwargs)
```

Runs mesh generator - tetgen for 3D or triangle for 2D meshes.

**Parameters geo** : geometry

geometry description

**a** : int, optional  
 a maximum area/volume constraint

**quadratic** : bool, optional  
 set True for quadratic elements

**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_poly(filename, verbose=True)`  
 Import mesh generated by tetgen or triangle.

**Parameters** **filename** : string  
 file name

**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.main()
```

```
sfepy.mesh.mesh_generators.tilted_mesh1d(conns, coors, ngrps, idim, n_rep, bb, eps=1e-06,  
                                             mybar=None, ndmap=False)
```

## sfepy.mesh.mesh\_tools module

```
sfepy.mesh.mesh_tools.elems_q2t(el)
```

```
sfepy.mesh.mesh_tools.smooth_mesh(mesh, n_iter=4, lam=0.6307, mu=-0.6347, weights=None,  
                                     bconstr=True, volume_corr=False)
```

FE mesh smoothing.

Based on:

[1] Steven K. Boyd, Ralph Muller, Smooth surface meshing for automated finite element model generation from 3D image data, Journal of Biomechanics, Volume 39, Issue 7, 2006, Pages 1287-1295, ISSN 0021-9290, 10.1016/j.jbiomech.2005.03.006. (<http://www.sciencedirect.com/science/article/pii/S0021929005001442>)

**Parameters** **mesh** : mesh

FE mesh.

**n\_iter** : integer, optional

Number of iteration steps.

**lam** : float, optional

Smoothing factor, see [1].

**mu** : float, optional

Unshrinking factor, see [1].

**weights** : array, optional

Edge weights, see [1].

**bconstr**: logical, optional :

Boundary constraints, if True only surface smoothing performed.

**volume\_corr**: logical, optional :

Correct volume after smoothing process.



**Returns** `coors` : array

Coordinates of mesh nodes.

## **sfePy.mesh.meshutils module**

Finite element mesh utilities.

**exception** `sfePy.mesh.meshutils.MeshUtilsCheckError`  
Function check failed.

**exception** `sfePy.mesh.meshutils.MeshUtilsError`  
Base class for exceptions in meshutils.

**exception** `sfePy.mesh.meshutils.MeshUtilsParseError`  
Parse error in input file.

**exception** `sfePy.mesh.meshutils.MeshUtilsWarning`  
Not necessarily error, but some strange thing happened.

**class** `sfePy.mesh.meshutils.bound`  
Handles all physical entities  $\leq$  `mshmodelnum`.

This class is optionally filled in `mesh.readmsh()` by calling the method `bound.handle2()`. It extracts **only** the node numbers of the entity. All entities are stored in `self.f` dictionary (key is the entity number).

Has methods to get the node numbers in PMD notation: (1 2 4 1 2 3 4 8 9 10 -> 1:4 8:10) ie. it removes any repeating numbers and shortens the list using ":"

Has method to find elements from a given list, which lie on the boundary formed by nodes (also returns appropriate side of elements).

So - all boundary conditions should be done using this class.

**associateelements** (*elements*)

**associateelements\_key** (*key*, *elements*)  
Finds the elements under the key *key* in *elements*.

This is a regular (very slow, but bullet proof) version.

**associateelements\_key\_fast** (*key*, *elements*)  
Finds the elements under the key *key* in *elements*.

This is a fast version, which assumes an undocumented feature, that all the elements which gmsh exports are in *exactly* the same order both in the entity *key* and *mshmodelnum*.

**finde** (*p*, *elements*)

**findelements** (*key*, *elements*)  
Returns element numbers (in a list), which lies at the boundary determined by `self.nodes`.

Also returns the side of element. Currently only support triangles and quadrangles.

**fromstr** (*str*)

**getf** (*key*)

**getstr** (*key*)

**handle** (*n*, *list*)  
Appends number in 'list' to internal dictionary (key=*n*)

**handle2** (*p*)

**handleelement** (*p*)

Handles element - not for boundary conditions

**readpmd** (*filename*)

Loads internal dictionary from 'filename'.

Format must be the same as from write()

**simplify** (*l*)

Removes repeating numbers and sorts the internal list l.

Note: it's very slow for large meshes. This is the thing which slows down everything.

**str** (*l*)

Converts l (must be a sorted list) to string (using ':').

**writeSV** (*f, els, key*)

**writepmd** (*filename*)

`sfepy.mesh.meshutils.check` (*s, what*)

`sfepy.mesh.meshutils.error` (*s, type=0*)

`sfepy.mesh.meshutils.flat` (*a*)

`sfepy.mesh.meshutils.formatpos` (*n, T*)

`sfepy.mesh.meshutils.formatpos2` (*n, T*)

**class** `sfepy.mesh.meshutils.mesh`

**average** (*list*)

**average\_vectors** (*list*)

**clean** ()

Deletes the whole mesh.

**computegrad** (*scalars*)

Returns the gradient of *scalars* (both are given in nodes).

**computenorm** (*vectors*)

Returns  $\sqrt{x^2+y^2+z^2}$  for all vectors (x,y,z) in *vectors*.

**convert\_el\_to\_nodes** (*els*)

**convert\_stress\_to\_nodes** ()

**det** (*x, y*)

**dist** (*a, b*)

**dist2** (*p, q*)

**getnumhexahedra** ()

**getnumprisms** ()

**getnumquadrangles** ()

**getnumtetrahedra** ()

**getnumtriangles** ()

**getscalar** (*n*)

Returns a scalar of a node whose number is n.

**getscalar2** (*n, scalars*)

Returns a scalar of a node whose number is *n*.

**getvector** (*n*)

Returns a vector associated to a node whose number is *n*.

**getxyz** (*n*)

Returns a tuple (x,y,z) of a node whose number is *n*.

**printinfo** ()

**readELE** (*filename, symmetric=False*)

Read elements from filename (\*.ELE).

**readELE2** (*filename*)

Read elements from filename (\*.ELE).

**readGMV** (*filename, what=2*)

Reads GMV file.

**what ... 0 read only mesh ... 1 read only data ... 2 read both**

**readNOD** (*filename, scale=1.0*)

Read nodes from filename (\*.NOD).

**readmsh** (*filename, b=None, symmetric=False, associateelements=True*)

Reads mesh from filename (\*.msh).

it will read the physical entity “mshmodelnum” (default 100), which must contain every node (which will be used) and every element.

Optional parameter *b* of type bound will be filled with all other physical (!=mshmodelnum) nodes.

**example:** gmsh exports these physical entities: 100,1,2,101,200

then readmsh will read 100 into self.elements and self.nodes, and optionally fills “b” with entities 1,2,101 and 200. The assumption is, that entity 100 will contain every node and element used in entities 1,2,101 and 200. Also the nodes and elements in 100 must be consecutively sorted (use mshsort for this purpose)

it will convert msh types to PMD types, so that self.elements only contains PMD types

symmetric.... is the problem rot symmetric? if yes, readmsh() will automatically convert triangles to rottriangles and quadrangles to rotquadrangles.

**readmsh2** (*filename, b=None, symmetric=False, associateelements=True*)

Reads mesh from filename (\*.msh). Version 2.0

it will read the physical entity “mshmodelnum” (default 100), which must contain every node (which will be used) and every element.

Optional parameter *b* of type bound will be filled with all other physical (!=mshmodelnum) nodes.

**example:** gmsh exports these physical entities: 100,1,2,101,200

then readmsh will read 100 into self.elements and self.nodes, and optionally fills “b” with entities 1,2,101 and 200. The assumption is, that entity 100 will contain every node and element used in entities 1,2,101 and 200. Also the nodes and elements in 100 must be consecutively sorted (use mshsort for this purpose)

it will convert msh types to PMD types, so that self.elements only contains PMD types

symmetric.... is the problem rot symmetric? if yes, readmsh() will automatically convert triangles to rottriangles and quadrangles to rotquadrangles.

**readpmd** (*filename*)

Read the mesh from filename (\*.I1).

Can read I1, which was produced by writepmd().

Should be able to read some hand written I1s from PMD Example Manual. Sometimes you will have to tweak the parser to read your syntax.

I suggest to only use the writepmd() I1 syntax. This is easily readable/writable.

**readstr2STR** (*filename*)

Reads temperature data from filename (\*.STR) to scalars.

**readstr3STR** (*filename*)

Reads temperature data from filename (\*.STR) to scalars.

**readxt2sSTR** (*filename*)

Reads temperature data from filename (\*.STR) to scalars.

**removecentralnodes** (*nods*)

**renumber\_elements** ()

**scalars\_elements2nodes** (*scalarsel*)

**sortnodes** ()

**vectors\_elements2nodes** (*scalarsel*)

**writeBC** (*filename, verbose=True*)

self.faces contain triplets (p1,p2,p3) which are triangles of tetrahedrons on the boundary. We need to find the number of each corresponding tetrahedron and it's side.

**writeELE** (*filename*)

Write nodes to filename (\*.ELE).

**writeNOD** (*filename*)

Write nodes to filename (\*.NOD).

**writemsh** (*filename, verbose=True*)

Writes mesh to filename (\*.msh).

**writemsh2** (*filename*)

Writes mesh to filename (\*.msh). Version 2.0

**writepmd** (*filename*)

Writes the mesh to filename (\*.I1).

**writeregions** (*filename*)

**writescalars** (*filename, scalars, C=0.0*)

**writescalarspos** (*filename, infotext='PMD\_scalars'*)

Writes self.scalars to \*.pos.

Optional parameter infotext specifies the name of the view in the pos file.

1.Associates a scalar with every node, so gmsh shows points.

2.Associates a scalar with every node of all elements, so gmsh fills the whole element (triangle, quadrangle, tetrahedra, ... etc.) with an extrapolated color.

You can set visibility in gmsh (only points, only triangles...).

**writescalarspos2** (*filename, scaltime, infotext='PMD\_scalars', dt=1.0*)

Writes self.scalars to \*.pos.

Optional parameter infotext specifies the name of the view in the pos file.

1. Associates a scalar with every node, so gmsh shows points.
2. Associates a scalar with every node of all elements, so gmsh fills the whole element (triangle, quadrangle, tetrahedra, ... etc.) with an extrapolated color.

You can set visibility in gmsh (only points, only triangles...).

**writescalarspos3** (*filename, scalars, infotext='PMD\_scalars'*)

**writestresspos** (*filename, infotext='PMD\_stress'*)

Writes scalars together with nodes and elements to filename.

Optional parameter infotext specifies the name of the view in the pos file.

- 1) Associates a scalar with every node, so gmsh shows points.
- 2) Associates a scalar with every node of all elements, so gmsh fills the whole element (triangle, quadrangle etc.) with an extrapolated color.

You can set visibility in gmsh (only points, only triangles...).

**writevectorspos** (*filename, infotext='PMD\_vectors'*)

Writes scalars together with nodes and elements to filename.

Optional parameter infotext specifies the name of the view in the pos file.

- 1) Associates a scalar with every node, so gmsh shows points.
- 2) Associates a scalar with every node of all elements, so gmsh fills the whole element (triangle, quadrangle etc.) with an extrapolated color.

You can set visibility in gmsh (only points, only triangles...).

**writevectorspos3** (*filename, vectorfield, infotext='PMD\_vectors'*)

**writexda** (*filename, verbose=True, b=None*)

Writes mesh to filename (\*.xda).

We try to be byte to byte compatible with the xda output from libmesh (so I use the same tabs and spaces as libmesh does).

`sfepy.mesh.meshutils.myfloat(s)`

Converts s to float, including PMD float format (without E).

`sfepy.mesh.meshutils.numlist2str(x)`

## sfepy.mesh.splinebox module

**class** `sfepy.mesh.splinebox.SplineBox` (*bbox, coors, name='spbox', \*\*kwargs*)

B-spline geometry parametrization. Geometry can be modified by moving spline control points.

**static augknt** (*knots, k, mults=1*)

**change\_shape** (*cpoint, val*)

Change shape of spline parametrization.

**Parameters** *cpoint* : list

The indices of the spline control point.

**val** : array

Displacement.

**static create\_spb** (*bbox, coors, nsg=None*)

**dvelocity** (*cpoint, dir*)

Evaluate derivative of spline in a given control point and direction.

**Parameters** **cpoint** : list

The indices of the spline control point.

**dir** : array

The directional vector.

**Returns** **dvel** : array

The design velocity field.

**evaluate** (*cp\_coors=None*)

Evaluate SplineBox.

**Returns** **coors** : array

The coordinates corresponding to the actual spline control points position.

**cp\_coors** : array

If is not None, use as control points coordinates.

**get\_control\_points** (*init=False*)

Get spline control points coordinates.

**static mmax** (*x, y*)

**set\_control\_points** (*cpt\_coors, add=False*)

Set spline control points position.

**Parameters** **cpt\_coors** : array

The coordinates of the spline control points.

**add** : bool

If True, coors += cpt\_coors

**static spcol** (*knots, k, tau*)

**static spsorted** (*meshsites, sites*)

**write\_vtk** (*filename*)

## 7.7.12 sfepy.optimize package

### sfepy.optimize.freeFormDef module

**class** sfepy.optimize.freeFormDef.**DesignVariables** (*\*\*kwargs*)

**normalize\_null\_space\_base** (*magnitude=1.0*)

**renumber\_by\_boxes** (*sp\_boxes*)

**class** sfepy.optimize.freeFormDef.**SplineBox** (*\*\*kwargs*)

**class** sfepy.optimize.freeFormDef.**SplineBoxes** (*\*\*kwargs*)

**create\_mesh\_from\_control\_points** ()

```

    interp_coordinates ()
    interp_mesh_velocity (shape, dsg_vars, idsg)
    set_control_points (dsg_vars=None)
sfepy.optimize.freeFormDef.interp_box_coordinates (spb, cxyz=None)
sfepy.optimize.freeFormDef.read_dsg_vars_hdf5 (filename)
sfepy.optimize.freeFormDef.read_spline_box_hdf5 (filename)

```

## sfepy.optimize.shapeOptim module

```

class sfepy.optimize.shapeOptim.ShapeOptimFlowCase (**kwargs)

    check_custom_sensitivity (term_desc, idsg, delta, dp_var_data, state_ap)
    check_sensitivity (idsgs, delta, dp_var_data, state_ap)
    create_evaluables ()
    static from_conf (conf, dpb, apb)
    generate_mesh_velocity (shape, idsgs=None)
    obj_fun (state_dp)
        Objective function evaluation for given direct problem state.
    sensitivity (dp_var_data, state_ap, select=None)
        Sensitivity of objective function evaluation for given direct and adjoint problem states.
sfepy.optimize.shapeOptim.obj_fun (design, shape_opt, opts)
    The objective function evaluation.
sfepy.optimize.shapeOptim.obj_fun_grad (design, shape_opt, opts)
    The objective function gradient evaluation.
sfepy.optimize.shapeOptim.solve_problem_for_design (problem, design, shape_opt, opts,
                                                    var_data=None, use_cache=True,
                                                    is_mesh_update=True)
    use_cache == True means direct problem...
sfepy.optimize.shapeOptim.test_terms (idsgs, delta, shape_opt, dp_var_data, state_ap)
    Test individual shape derivative terms.
sfepy.optimize.shapeOptim.update_mesh (shape_opt, pb, design)

```

## 7.7.13 sfepy.physics package

### sfepy.physics.energy module

```

sfepy.physics.energy.eval_ion_ion_energy (centres, charges)
    Compute the ion-ion nergy.
sfepy.physics.energy.eval_non_local_interaction (problem, region_name, var_name, in-
                                                    tegral_name, f1, f2, kernel_function,
                                                    pbar=None)
    Single element group only!

```

### **sfepy.physics.potentials module**

Classes for constructing potentials of atoms and molecules.

```
class sfepy.physics.potentials.CompoundPotential (objs=None)
    Sum of several potentials.

    append (obj)
    insert (ii, obj)
    update_expression ()

class sfepy.physics.potentials.Potential (name, function, centre=None, dim=3, args=None)
    Single spherically symmetric potential.

    get_charge (coors, eps=1e-06)
        Get charge corresponding to the potential by numerically applying Laplacian in spherical coordinates.

    get_distance (coors)
        Get the distance of points with coordinates coors of the potential centre.

class sfepy.physics.potentials.PotentialBase (**kwargs)
    Base class for potentials.
```

### **sfepy.physics.radial\_mesh module**

```
class sfepy.physics.radial_mesh.ExplicitRadialMesh (coors)

    get_coors ()
    get_index (r)
    get_midpoint_mesh (to=None)
    get_mixing (r)
    get_parent_mesh ()
    get_r (index)
    interpolate (potential, r)
    last_point ()
    shape
    size
    slice (x, y)
    sparse_vector (vector)

class sfepy.physics.radial_mesh.RadialHyperbolicMesh (jm, ap=None, size=None, from_zero=False)

    size = None

class sfepy.physics.radial_mesh.RadialMesh
    Radial mesh.

    dot (vector_a, vector_b, norm='spherical')
```



int f(r) g(r) r^2 dr  
**integrate** (*vector*, *norm*)

int f(r) r^2 dr  
**interpolate\_3d** (*potential*, *coors*, *centre=None*)  
**intervals** ()  
**linear\_integral** (*vector*, *from\_zero=False*)

$$a_n = \int_{r_0}^{r_n} f(r) dr$$

*from\_zero* starts to integrate from zero, instead of starting between the first two points  
**linear\_integrate** (*vector*)

int f(r) dr  
**static merge** (*meshes*)  
**norm** (*vector*, *norm='spherical'*)  
**output\_vector** (*vector*, *filename=None*)  
**plot** (*vector*, *cmd='plot'*)  
**class** sfepy.physics.radial\_mesh.**RadialVector** (*mesh*, *values=None*)  
  
**derivatives** (*radial=True*)  
**extrapolated\_derivatives** (*at=None*, *precision=0.0001*, *attempts=10*)  
**extrapolated\_values** (*at=None*, *precision=0.0001*, *grade=10*, *attempts=10*)  
**static from\_file** (*file*)  
**get\_coors** ()  
**get\_extrapolated** (*precision=0.0001*, *grade=10*, *attempts=10*)  
**integrate** (*precision=0.0001*)  
**interpolate** (*x*)  
**interpolate\_3d** (*coors*, *centre=(0, 0, 0)*)  
**linear\_derivatives** ()  
**linear\_integral** (*from\_zero=False*)  
**linear\_integrate** ()  
**output\_vector** (*filename=None*)  
**plot** ()  
**pretty** (*values*)  
**running\_mean** ()

```
slice (x, y)  
static sparse_merge (vectors)  
to_file (filename=None)
```

### **sfepy.physics.schroedinger\_app** module

```
class sfepy.physics.schroedinger_app.SchroedingerApp (conf, options, output_prefix,  
                                                    **kwargs)
```

Base application for electronic structure calculations.

Subclasses should typically override *solve\_eigen\_problem()* method.

This class allows solving only simple single electron problems, e.g. well, oscillator, hydrogen atom and boron atom with 1 electron.

```
call ()  
make_full (mtx_s_phi)  
static process_options (options)  
    Application options setup. Sets default values for missing non-compulsory options.  
    Options:  
        save_eig_vectors [(from_largest, from_smallest) or None] If None, save all.  
save_results (eigs, mtx_phi, 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.physics.schroedinger_app.guess_n_eigs (n_electron, n_eigs=None)
```

Guess the number of eigenvalues (energies) to compute so that the smearing iteration converges. Passing *n\_eigs* overrides the guess.

## 7.7.14 **sfepy.postprocess** package

### **sfepy.postprocess.dataset\_manager** module

Code to help with managing a TVTK data set in Pythonic ways.

```
class sfepy.postprocess.dataset_manager.DatasetManager
```

```
    activate (name, category='point')  
        Make the specified array the active one.  
    add_array (array, name, category='point')  
        Add an array to the dataset to specified category ('point' or 'cell').  
    remove_array (name, category='point')  
        Remove an array by its name and optional category (point and cell). Returns the removed array.  
    rename_array (name1, name2, category='point')  
        Rename a particular array from name1 to name2.
```

**update()**

Update the dataset when the arrays are changed.

`sfepy.postprocess.dataset_manager.get_all_attributes(obj)`

Gets the scalar, vector and tensor attributes that are available in the given VTK data object.

`sfepy.postprocess.dataset_manager.get_array_type(arr)`

Returns if the array is a scalar ('scalars'), vector ('vectors') or tensor ('tensors'). It looks at the number of components to decide. If it has a wierd number of components it returns the empty string.

`sfepy.postprocess.dataset_manager.get_attribute_list(data)`

Gets scalar, vector and tensor information from the given data (either cell or point data).

## 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\_dofs module**

Functions to visualize the mesh connectivity with global and local DOF numberings.

`sfepy.postprocess.plot_dofs.plot_global_dofs(ax, coors, econn, show=False)`

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, show=False)`

Plot local DOF numbers corresponding to an extended connectivity.

`sfepy.postprocess.plot_dofs.plot_mesh(ax, coors, conn, edges, show=False)`

Plot a finite element mesh as a wireframe.

`sfepy.postprocess.plot_dofs.plot_nodes(ax, coors, econn, ref_nodes, dofs, show=False)`

Plot Lagrange reference element nodes corresponding to global DOF numbers given in an extended connectivity.

### **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, show=False)`

Plot edges of a geometry element as numbered arrows.

`sfepy.postprocess.plot_facets.plot_faces(ax, gel, radius, n_point, show=False)`

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, show=False)`

Plot a geometry element as a wireframe.

### **sfepy.postprocess.sources module**

**class** `sfepy.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\_range()**

**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.

**set\_step** (*step=0*)

Set step of a data sequence.

**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** `sfeipy.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.

**get\_step\_range** ()

**read\_common** (*filename*)

**set\_filename** (*filename, vis\_source*)

**class** `sfeipy.postprocess.sources.GenericSequenceFileSource` (*\*args, \*\*kwargs*)

File source usable with any format supported by MeshIO classes, with exception of HDF5 (.h5), for file sequences.

**create\_source** ()

Create a VTK source from data in a SfePy-supported file.

**get\_step\_range** ()

**set\_filename** (*filename, vis\_source*)

**class** `sfeipy.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** ()

**get\_step\_range** ()

**set\_filename** (*filename, vis\_source*)

**class** `sfepy.postprocess.sources.VTKSequenceFileSource` (*filename*, *watch=False*, *offscreen=True*)

A thin wrapper around `mlab.pipeline.open()` for VTK file sequences.

**create\_source** ()

Create a VTK file source

**get\_step\_range** ()

**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.

## sfepy.postprocess.time\_history module

`sfepy.postprocess.time_history.average_vertex_var_in_cells` (*ths\_in*)

Average histories in the element nodes for each nodal variable originally requested in elements.

`sfepy.postprocess.time_history.dump_to_vtk` (*filename*, *output\_filename\_trunk=None*, *step0=0*, *steps=None*, *fields=None*, *linearization=None*)

Dump a multi-time-step results file into a sequence of VTK files.

`sfepy.postprocess.time_history.extract_time_history` (*filename*, *extract*, *verbose=True*)

Extract time history of a variable from a multi-time-step results file.

**Parameters** *filename* : str

The name of file to extract from.

**extract** : str

The description of what to extract in a string of comma-separated description items. A description item consists of: name of the variable to extract, mode ('e' for elements, 'n' for nodes), ids of the nodes or elements (given by the mode). Example: 'u n 10 15, p e 0' means variable 'u' in nodes 10, 15 and variable 'p' in element 0.

**verbose** : bool

Verbosity control.

**Returns** *ths* : dict

The time histories in a dict with variable names as keys. If a nodal variable is requested in elements, its value is a dict of histories in the element nodes.

**ts** : TimeStepper instance

The time stepping information.

`sfepy.postprocess.time_history.extract_times` (*filename*)

Read true time step data from individual time steps.

**Returns** *steps* : array

The time steps.

**times** : array

The times of the time steps.

**nts** : array

The normalized times of the time steps, in [0, 1].

**dts** : array

The true time deltas.

`sfepy.postprocess.time_history.guess_time_units` (*times*)

Given a vector of times in seconds, return suitable time units and new vector of times suitable for plotting.

**Parameters** **times** : array

The vector of times in seconds.

**Returns** **new\_times** : array

The vector of times in *units*.

**units** : str

The time units.

`sfepy.postprocess.time_history.save_time_history` (*ths, ts, filename\_out*)

Save time history and time-stepping information in a HDF5 file.

### `sfepy.postprocess.utils` module

`sfepy.postprocess.utils.get_data_ranges` (*obj, return\_only=False, use\_names=None, filter\_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.viewer` module

**class** `sfepy.postprocess.viewer.ReloadSource`

**class** `sfepy.postprocess.viewer.SetStep`

**step** = None

**class** `sfepy.postprocess.viewer.Viewer` (*filename, watch=False, animate=False, anim\_format=None, ffmpeg\_options=None, output\_dir='.', offscreen=False, auto\_screenshot=True*)

Class to automate visualization of various data using Mayavi. It can be used via `postproc.py` or `isfepy` the most easily.



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.

**animate** [bool] If True, save a view snapshost for each time step and exit.

**anim\_format** [str] 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** [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, fgcolor=(0.0, 0.0,
0.0), bgcolor=(1.0, 1.0, 1.0), layout='rowcol', scalar_mode='iso_surface', vec-
tor_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, fig_filename='view.png', resolution=None, filter_names=None,
only_names=None, group_names=None, step=0, anti_aliasing=None, do-
main_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()`.

**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.

**layout** : str

Grid layout for placing the datasets. Possible values are: 'row', 'col', 'rowcol', 'col-row'.

**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, if self.auto\_screenshot is True.

**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

The time step to display.

**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, rng=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*)

Animate the current scene view for all the time steps and save a snapshot of each step view.

**save\_image** (*filename*)

Save a snapshot of the current scene.

**set\_source\_filename** (*filename*)

**show\_scalarBars** (*scalarBars*)

**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, reuse\_viewer=None*)

## 7.7.15 sfepy.solvers package

### sfepy.solvers.eigen module

**class** sfepy.solvers.eigen.**LOBPCGEigenvalueSolver** (*conf*, *\*\*kwargs*)  
SciPy-based LOBPCG solver for sparse symmetric problems.

**name** = 'eig.scipy\_lobpcg'

**static process\_conf** (*conf*, *kwargs*)  
Missing items are set to default values.

Example configuration, all items:

```
solver_2 = {
    'name' : 'lobpcg',
    'kind' : 'eig.scipy_lobpcg',

    'i_max' : 20,
    'n_eigs' : 5,
    'eps_a' : None,
    'largest' : True,
    'precond' : None,
    'verbosity' : 0,
}
```

**class** sfepy.solvers.eigen.**PysparseEigenvalueSolver** (*conf*, *\*\*kwargs*)  
Pysparse-based eigenvalue solver for sparse symmetric problems.

**name** = 'eig.pysparse'

**static process\_conf** (*conf*, *kwargs*)  
Missing items are set to default values.

Example configuration, all items:

```
solver_2 = {
    'name' : 'eigen1',
    'kind' : 'eig.pysparse',

    'i_max' : 150,
    'eps_a' : 1e-5,
    'tau' : -10.0,
    'method' : 'qmrs',
    'verbosity' : 0,
    'strategy' : 1,
}
```

**class** sfepy.solvers.eigen.**ScipyEigenvalueSolver** (*conf*, *\*\*kwargs*)  
SciPy-based solver for both dense and sparse problems (if *n\_eigs* is given).

**name** = 'eig.scipy'

**class** sfepy.solvers.eigen.**ScipySGEEigenvalueSolver** (*conf*, *\*\*kwargs*)  
SciPy-based solver for dense symmetric problems.

**name** = 'eig.sgscipy'

**static process\_conf** (*conf*, *kwargs*)  
Missing items are set to default values.

Example configuration, all items:

```

solver_20 = {
    'name' : 'eigen2',
    'kind' : 'eig.sgscipy',

    'force_n_eigs' : True,
}

```

`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.PETScKrylovSolver`(*conf*, \*\**kwargs*)

PETSc Krylov subspace solver.

The solver and preconditioner types are set upon the solver object creation. Tolerances can be overridden when called by passing a *conf* object.

### Notes

Convergence is reached when  $rnorm < \max(eps_r * rnorm_0, eps_a)$ , where, in PETSc, *rnorm* is by default the norm of *preconditioned* residual.

**name** = 'ls.petsc'

**static process\_conf**(*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```

solver_120 = {
    'name' : 'ls120',
    'kind' : 'ls.petsc',

    'method' : 'cg', # ksp_type
    'precond' : 'icc', # pc_type
    'precond_side' : 'left', # ksp_pc_side
    'eps_a' : 1e-12, # abstol
    'eps_r' : 1e-12, # rtol
    'eps_d' : 1e5, # divtol
    'i_max' : 1000, # maxits
}

```

**set\_matrix**(*mtx*)

**class** `sfepy.solvers.ls.PETScParallelKrylovSolver`(*conf*, \*\**kwargs*)

PETSc Krylov subspace solver able to run in parallel by storing the system to disk and running a separate script via *mpiexec*.

The solver and preconditioner types are set upon the solver object creation. Tolerances can be overridden when called by passing a *conf* object.

### Notes

Convergence is reached when  $rnorm < \max(eps\_r * rnorm\_0, eps\_a)$ , where, in PETSc,  $rnorm$  is by default the norm of *preconditioned* residual.

**name = 'ls.petsc\_parallel'**

**static process\_conf** (*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```
solver_1 = {
    'name' : 'ls',
    'kind' : 'ls.petsc_parallel',

    'n_proc' : 5, # Number of processes to run.

    'method' : 'cg', # ksp_type
    'precond' : 'bjacobi', # pc_type
    'sub_precond' : 'icc', # sub_pc_type
    'eps_a' : 1e-12, # abstol
    'eps_r' : 1e-12, # rtol
    'eps_d' : 1e5, # divtol
    'i_max' : 1000, # maxits
}
```

**class** sfepy.solvers.ls.**PyAMGSolver** (*conf*, **\*\*kwargs**)

Interface to PyAMG solvers.

### Notes

Uses relative convergence tolerance, i.e.  $eps\_r$  is scaled by  $\|b\|$ .

**name = 'ls.pyamg'**

**static process\_conf** (*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```
solver_102 = {
    'name' : 'ls102',
    'kind' : 'ls.pyamg',

    'method' : 'smoothed_aggregation_solver',
    'accel' : 'cg',
    'eps_r' : 1e-12,
}
```

**class** sfepy.solvers.ls.**SchurComplement** (*conf*, **\*\*kwargs**)

Schur complement.

Solution of the linear system

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

is obtained by solving the following equation:

$$(D - CA^{-1}B) \cdot v = g - CA^{-1}f$$

variable(s)  $u$  are specified in “eliminate” list, variable(s)  $v$  are specified in “keep” list,

See: [http://en.wikipedia.org/wiki/Schur\\_complement](http://en.wikipedia.org/wiki/Schur_complement)

**name** = 'ls.schur\_complement'

**static process\_conf** (*conf*, *kwargs*)  
Setup solver configuration options.

Example configuration:

```
solvers = {
    'ls': ('ls.schur_complement',
          {'eliminate': ['displacement'],
           'keep': ['pressure'],
           'needs_problem_instance': True,
          })
}
```

**static schur\_fun** (*res*, *mtx*, *rhs*, *nn*)

**class** sfepy.solvers.ls.**SchurGeneralized** (*conf*, **\*\*kwargs**)  
Generalized Schur complement.

Defines the matrix blocks and calls user defined function.

**name** = 'ls.schur\_generalized'

**static process\_conf** (*conf*, *kwargs*)  
Setup solver configuration options.

Example configuration:

```
solvers = {
    'ls': ('ls.schur_generalized',
          {'blocks':
           {'u': ['displacement1', 'displacement2'],
            'v': ['velocity1', 'velocity2'],
            'w': ['pressure1', 'pressure2'],
           },
           'function': my_schur,
           'needs_problem_instance': True,
          })
}
```

**class** sfepy.solvers.ls.**ScipyDirect** (*conf*, **\*\*kwargs**)

**name** = 'ls.scipy\_direct'

**static process\_conf** (*conf*, *kwargs*)  
Missing items are set to default values.

Example configuration, all items:

```
solver_1100 = {
    'name' : 'dls1100',
    'kind' : 'ls.scipy_direct',
}
```

```
        'method' : 'superlu',
        'presolve' : False,
        'warn' : True,
    }
```

**class** sfepy.solvers.ls.**ScipyIterative** (*conf*, **\*\*kwargs**)  
Interface to SciPy iterative solvers.

### Notes

The *eps\_r* tolerance is both absolute and relative - the solvers stop when either the relative or the absolute residual is below it.

A preconditioner can be anything that the SciPy solvers accept (sparse matrix, dense matrix, LinearOperator).

**name** = 'ls.scipy\_iterative'

**static process\_conf** (*conf*, *kwargs*)  
Missing items are set to default values.

Example configuration, all items:

```
solver_l10 = {
    'name' : 'ls10',
    'kind' : 'ls.scipy_iterative',

    'method' : 'cg',
    'precond' : None,
    'callback' : None,
    'i_max' : 1000,
    'eps_r' : 1e-12,
}
```

**class** sfepy.solvers.ls.**Umfpack** (*conf*, **\*\*kwargs**)  
This class stays for compatability with old input files. Use ScipyDirect instead.

**name** = 'ls.umfpack'

**sfepy.solvers.ls.standard\_call** (*call*)  
Decorator handling argument preparation and timing for linear solvers.

### sfepy.solvers.nls module

**class** sfepy.solvers.nls.**Newton** (*conf*, **\*\*kwargs**)  
Solves a nonlinear system  $f(x) = 0$  using the Newton method with backtracking line-search, starting with an initial guess  $x^0$ .

For common configuration parameters, see [Solver](#).

**Parameters** **i\_max** : int

The maximum number of iterations.

**eps\_a** : float

The absolute tolerance for the residual, i.e.  $\|f(x^i)\|$ .

**eps\_r** : float

The relative tolerance for the residual, i.e.  $\|f(x^i)\|/\|f(x^0)\|$ .



**macheps** : float

The float considered to be machine “zero”.

**lin\_red** : float

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

Start the backtracking line-search by reducing the step, if  $\|f(x^i)\|/\|f(x^{i-1})\|$  is larger than *ls\_on*.

**ls\_red** :  $0.0 < \text{float} < 1.0$

The step reduction factor in case of correct residual assembling.

**ls\_red\_warp** :  $0.0 < \text{float} < 1.0$

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** :  $0.0 < \text{float} < 1.0$

The minimum step reduction factor.

**give\_up\_warp** : bool

If True, abort on the “warp violation” error.

**check** : 0, 1 or 2

If  $\geq 1$ , check the tangent matrix using finite differences. If 2, plot the resulting sparsity patterns.

**delta** : float

If *check*  $\geq 1$ , the finite difference matrix is taken as  $A_{ij} = \frac{f_i(x_j+\delta) - f_i(x_j-\delta)}{2\delta}$ .

**is\_plot** : False

If True, plot the solution and residual in each step.

**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.

**problem** : ‘nonlinear’ or ‘linear’

Specifies if the problem is linear or non-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.problem* == *'linear'* means a pre-assembled and possibly pre-solved matrix. This is mostly useful for linear time-dependent problems.

```
__init__(conf, **kwargs)
```

```
__module__ = 'sfepy.solvers.nls'
```

```
name = 'nls.newton'
```

```
static process_conf(conf, kwargs)
```

Missing items are set to default values for a linear problem.

Example configuration, all items:

```
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).
    'lin_precision' : None,
    'ls_on' : 0.99999,
    'ls_red' : 0.1,
    'ls_red_warp' : 0.001,
    'ls_min' : 1e-5,
    'give_up_warp' : False,
    'check' : 0,
    'delta' : 1e-6,
```

```

        'is_plot' : False,
        'log' : None, # 'nonlinear' or 'linear' (ignore i_max)
        'problem' : 'nonlinear',
    }

```

**class** sfepy.solvers.nls.**ScipyBroyden**(*conf*, **\*\*kwargs**)

Interface to Broyden and Anderson solvers from scipy.optimize.

**\_\_call\_\_**(*vec\_x0*, *conf=None*, *fun=None*, *fun\_grad=None*, *lin\_solver=None*, *iter\_hook=None*, *status=None*)

**\_\_init\_\_**(*conf*, **\*\*kwargs**)

**\_\_module\_\_** = 'sfepy.solvers.nls'

**name** = 'nls.scipy\_broyden\_like'

**static process\_conf**(*conf*, *kwargs*)

Missing items are left to scipy defaults. Unused options are ignored.

Example configuration, all items:

```

solver_1 = {
    'name' : 'broyden',
    'kind' : 'nls.scipy_broyden_like',

    'method' : 'broyden3',
    'i_max' : 10,
    'alpha' : 0.9,
    'M' : 5,
    'w0' : 0.1,
    'f_tol' : 6e-6,
    'verbose' : True,
}

```

**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\_x* items perturbed by a small delta.

**sfepy.solvers.nls.conv\_test**(*conf*, *it*, *err*, *err0*)

## sfepy.solvers.optimize module

**class** sfepy.solvers.optimize.**FMinSteepestDescent**(*conf*, **\*\*kwargs**)

**name** = 'opt.fmin\_sd'

**static process\_conf**(*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```

solver_0 = {
    'name' : 'fmin_sd',
    'kind' : 'opt.fmin_sd',

    'i_max' : 10,
    'eps_rd' : 1e-5, # Relative delta of objective function
}

```

```
'eps_of'      : 1e-4,
'eps_ofg'     : 1e-8,
'norm'        : nm.Inf,
'ls'          : True, # Linesearch.
'ls_method'   : 'backtracking', # 'backtracking' or 'full'
'ls0'         : 0.25,
'ls_red'      : 0.5,
'ls_red_warp' : 0.1,
'ls_on'       : 0.99999,
'ls_min'      : 1e-5,
'check'       : 0,
'delta'       : 1e-6,
'output'      : None, # 'itc'
'log'         : {'text' : 'output/log.txt',
                 'plot' : 'output/log.png'},
'yscales'     : ['linear', 'log', 'log', 'linear'],
}
```

**class** sfepy.solvers.optimize.**ScipyFMinSolver** (*conf*, **\*\*kwargs**)

Interface to SciPy optimization solvers `scipy.optimize.fmin_*`.

**name** = 'nls.scipy\_fmin\_like'

**static process\_conf** (*conf*, *kwargs*)

Missing items are left to SciPy defaults. Unused options are ignored.

Besides 'i\_max', use option names according to `scipy.optimize` function arguments. The 'i\_max' translates either to 'maxiter' or 'maxfun' as available.

Example configuration:

```
solver_1 = {
    'name' : 'fmin',
    'kind' : 'nls.scipy_fmin_like',

    'method' : 'bfgs',
    'i_max' : 10,
    'verbose' : True,

    'gtol' : 1e-7
}
```

**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*, *\*\*kwargs*)

**name** = 'nls.oseen'

**static process\_conf**(*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```

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

    'needs_problem_instance' : True,
    'stabil_mat' : 'stabil',

    'adimensionalize' : False,
    'check_navier_stokes_rezidual' : 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).
    'is_plot'    : False,
    'log'        : {'text' : 'oseen_log.txt',
                    'plot' : 'oseen_log.png'},
}

```

**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.petsc\_worker module

PETSc solver worker process.

sfepy.solvers.petsc\_worker.**solve**()

### **sfePy.solvers.semismooth\_newton module**

**class** `sfePy.solvers.semismooth_newton.SemismoothNewton` (*conf*, *\*\*kwargs*)

The semi-smooth Newton method 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:  $y \leftarrow y - \beta J(y)\Phi(y)$

#### **Notes**

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.

**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'

**static process\_conf** (*conf*, *kwargs*)

Missing items are set to default values.

Example configuration, all items:

```
solver_1 = {
    'name' : 'semismooth_newton',
    'kind' : 'nls.semismooth_newton',

    'semismooth' : True,

    'i_max'      : 10,
    'eps_a'      : 1e-8,
    'eps_r'      : 1e-2,
    'macheps'    : 1e-16,
    'lin_red'    : 1e-2, # Linear system error < (eps_a * lin_red).
    'ls_red_reg' : 0.1,
    'ls_red_alt' : 0.01,
    'ls_red_warp' : 0.001,
    'ls_on'      : 0.9,
    'ls_min'     : 1e-10,
    'log'        : {'plot' : 'convergence.png'},
}
```

### **sfePy.solvers.solvers module**

Base (abstract) solver classes.

**class** `sfePy.solvers.solvers.EigenvalueSolver` (*conf*, *mtx\_a=None*, *mtx\_b=None*, *n\_eigs=None*, *eigenvectors=None*, *status=None*)

Abstract eigenvalue solver class.

```
class sfepy.solvers.solvers.LinearSolver (conf, mtx=None, status=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.

class sfepy.solvers.solvers.NonlinearSolver (conf,          fun=None,          fun_grad=None,
                                              lin_solver=None,  iter_hook=None,  sta-
                                              tus=None, **kwargs)

    Abstract nonlinear solver class.

class sfepy.solvers.solvers.OptimizationSolver (conf, obj_fun=None, obj_fun_grad=None,
                                                  status=None, obj_args=None, **kwargs)

    Abstract optimization solver class.

class sfepy.solvers.solvers.Solver (conf=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__(). The subclasses that implement process_conf()
    have to call Solver.process_conf().

    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
            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.

    static process_conf (conf, kwargs=None)
        Ensures conf contains 'name' and 'kind'.

class sfepy.solvers.solvers.TimeSteppingSolver (conf, **kwargs)
    Abstract time stepping solver class.

sfepy.solvers.solvers.make_get_conf (conf, kwargs)
```

## sfepy.solvers.ts module

```
class sfepy.solvers.ts.TimeStepper (t0,    t1,    dt=None,    n_step=None,    step=None,
                                     is_quasistatic=False)
    Time stepper class.

    static from_conf (conf)

    iter_from (step)

    normalize_time ()

    set_from_data (t0, t1, dt=None, n_step=None, step=None)

    set_from_ts (ts, step=None)
```

```
    set_step (step=0, nt=0.0)
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.
    static from_conf (conf)
    get_default_time_step ()
    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_step (step=0, nt=0.0)
    set_time_step (dt)
sfepy.solvers.ts.get_print_info (n_step)
```

### sfepy.solvers.ts\_solvers module

Time stepping solvers.

```
class sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver (conf, **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.
    name = 'ts.adaptive'
    static process_conf (conf, kwargs)
        Process configuration options.
    solve_step (ts, state0, nls_status=None)
        Solve a single time step.
class sfepy.solvers.ts_solvers.ExplicitTimeSteppingSolver (conf, **kwargs)
    Explicit time stepping solver with a fixed time step.
    name = 'ts.explicit'
    static process_conf (conf, kwargs)
        Process configuration options.
    solve_step (ts, state0, nls_status=None)
        Solve a single time step.
class sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver (conf, **kwargs)
    Implicit time stepping solver with a fixed time step.
    name = 'ts.simple'
    static process_conf (conf, kwargs)
        Process configuration options.
    solve_step (ts, state0, nls_status=None)
        Solve a single time step.
class sfepy.solvers.ts_solvers.StationarySolver (conf, **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.
```



**name** = 'ts.stationary'

`sfepy.solvers.ts_solvers.adapt_time_step(ts, status, adt, problem=None)`

Adapt the time step of *ts* according to the exit status of the nonlinear solver.

The time step *dt* is reduced, if the nonlinear solver did not converge. If it converged in less then a specified number of iterations for several time steps, the time step is increased. This is governed by the following parameters:

- **red\_factor** : time step reduction factor
- **red\_max** : maximum time step reduction factor
- **inc\_factor** : time step increase factor
- **inc\_on\_iter** : increase time step if the nonlinear solver converged in less than this amount of iterations...
- **inc\_wait** : ...for this number of consecutive time steps

**Parameters** **ts** : VariableTimeStepper instance

The time stepper.

**status** : IndexedStruct instance

The nonlinear solver exit status.

**adt** : Struct instance

The adaptivity parameters of the time solver:

**problem** : ProblemDefinition instance, optional

This 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.get_initial_state(problem)`

Create a zero state vector and apply initial conditions.

`sfepy.solvers.ts_solvers.get_min_dt(adt)`

`sfepy.solvers.ts_solvers.make_explicit_step(ts, state0, problem, mass, nls_status=None)`

Make a step of an explicit time stepping solver.

`sfepy.solvers.ts_solvers.make_implicit_step(ts, state0, problem, nls_status=None)`

Make a step of an implicit time stepping solver.

`sfepy.solvers.ts_solvers.prepare_matrix(problem, state)`

Pre-assemble tangent system matrix.

`sfepy.solvers.ts_solvers.prepare_save_data(ts, conf)`

Given a time stepper configuration, return a list of time steps when the state should be saved.

## 7.7.16 sfepy.terms package

### Term Overview

### Term Syntax

In general, the syntax of a term call is:

$\langle \text{term name} \rangle . \langle i \rangle . \langle r \rangle ( \langle \text{arg1} \rangle , \langle \text{arg2} \rangle , \dots ) ,$

where  $\langle i \rangle$  denotes an integral name (i.e. a name of numerical quadrature to use) and  $\langle r \rangle$  marks a region (domain of the integral).

The following notation is used:

Table 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 $(\frac{1}{2}((\nabla \underline{u}) + (\nabla \underline{u})^T))$
$\underline{\underline{F}}$	deformation gradient $F_{ij} = \frac{\partial x_i}{\partial X_j}$
$J$	$\det(F)$
$\underline{\underline{C}}$	right Cauchy-Green deformation tensor $C = F^T F$
$\underline{\underline{E}}(\underline{u})$	Green strain tensor $E_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} + \frac{\partial u_m}{\partial x_i} \frac{\partial u_m}{\partial x_j})$
$\underline{\underline{S}}$	second Piola-Kirchhoff stress tensor
$\underline{f}$	vector volume forces
$f$	scalar volume force (source)
$\rho$	density
$\nu$	kinematic viscosity
$c$	any constant
$\delta_{ij}, \underline{\underline{I}}$	Kronecker delta, identity matrix
$\text{tr } \underline{\bullet}$	trace of a second order tensor $(\sum_{i=1}^d \bullet_{ii})$
$\text{dev } \underline{\bullet}$	deviator of a second order tensor $(\underline{\bullet} - \frac{1}{d} \text{tr } \underline{\bullet})$
$T_K \in \mathcal{T}_h$	$K$ -th element of triangulation (= mesh) $\mathcal{T}_h$ of domain $\Omega$
$K \leftarrow \mathcal{I}_h$	$K$ is assigned values from $\{0, 1, \dots, N_h - 1\} \equiv \mathcal{I}_h$ in ascending order

The suffix “<sub>0</sub>” 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.

pre-fix	meaning	evaluation modes	meaning
dw	discrete weak	'weak'	terms having a virtual (test) argument and zero or more unknown arguments, used for FE assembling
d	discrete	'eval'	terms having all arguments known, the result is the scalar value of the integral
di	discrete integrated	'eval'	like 'd' but the result is not a scalar (e.g. a vector)
dq	discrete quadrature	'qp'	terms having all arguments known, the result are the values in quadrature points of elements
ev	evaluate	'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 an automatically generated table. The first column lists the name, the second column the argument lists and the third column the mathematical definition of each term.

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 7.5: Table of all terms.

name/class/link	arguments	definition
dw_adj_convect1 AdjConvect1Term termsAdjointNavierStokes	<virtual>, <state>, <parameter>	$\int_{\Omega} ((\underline{v} \cdot \nabla) \underline{u}) \cdot \underline{w}$
dw_adj_convect2 AdjConvect2Term termsAdjointNavierStokes	<virtual>, <state>, <parameter>	$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{v}) \cdot \underline{w}$
dw_adj_div_grad AdjDivGradTerm termsAdjointNavierStokes	<material_1>, <material_2>, <virtual>, <parameter>	$w \delta_u \Psi(\underline{u}) \circ \underline{v}$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_bc_newton BCNewtonTerm terms_dot	<material_1>, <material_2>, <virtual>, <state>	$\int_{\Gamma} \alpha q (p - p_{\text{outer}})$
dw_biot BiotTerm termsBiot	<material>, <virtual>, <state> <material>, <state>, <virtual> <material>, <parameter_v>, <parameter_s>	$\int_{\Omega} p \alpha_{ij} e_{ij}(\underline{v}), \int_{\Omega} q \alpha_{ij} e_{ij}(\underline{u})$
dw_biot_eth BiotETHTerm termsBiot	<ts>, <material_0>, <material_1>, <virtual>, <state> <ts>, <material_0>, <material_1>, <state>, <virtual>	$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) p(\tau) d\tau \right] e_{ij}(\underline{v}),$ $\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$
ev_biot_stress BiotStressTerm termsBiot	<material>, <parameter>	$- \int_{\Omega} \alpha_{ij} \bar{p}$  vector for $K \leftarrow \mathcal{I}_h : - \int_{T_K} \alpha_{ij} \bar{p} / \int_{T_K} 1$  $- \alpha_{ij} \bar{p} _{qp}$
dw_biot_th BiotTHTerm termsBiot	<ts>, <material>, <virtual>, <state> <ts>, <material>, <state>, <virtual>	$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) p(\tau) d\tau \right] e_{ij}(\underline{v}),$ $\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] q$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
ev_cauchy_strain  CauchyStrainTerm termsLinElasticity	<parameter>	$\int_{\Omega} \underline{\underline{e}}(\underline{w})$ vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$ $\underline{\underline{e}}(\underline{w}) _{qp}$
ev_cauchy_strain_s  CauchyStrainSTerm termsLinElasticity	<parameter>	$\int_{\Gamma} \underline{\underline{e}}(\underline{w})$ vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$ $\underline{\underline{e}}(\underline{w}) _{qp}$
ev_cauchy_stress  CauchyStressTerm termsLinElasticity	<material>, <parameter>	$\int_{\Omega} D_{ijkl} e_{kl}(\underline{w})$ vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} D_{ijkl} e_{kl}(\underline{w}) / \int_{T_K} 1$ $D_{ijkl} e_{kl}(\underline{w}) _{qp}$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
ev_cauchy_stress_eth CauchyStressETHTerm termsLinElasticity	<ts>, <material_0>, <material_1>, <parameter>	$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$ <p>vector for <math>K \leftarrow \mathcal{I}_h</math> : <math>\int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau / \int_{T_K} 1</math></p> $\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau _{qp}$
ev_cauchy_stress_th CauchyStressTHTerm termsLinElasticity	<ts>, <material>, <parameter>	$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$ <p>vector for <math>K \leftarrow \mathcal{I}_h</math> : <math>\int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau / \int_{T_K} 1</math></p> $\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau _{qp}$
dw_contact_plane ContactPlaneTerm termsSurface	<material_f>, <material_n>, <material_a>, <material_b>, <virtual>, <state>	$\int_{\Gamma} \underline{v} \cdot f(d(\underline{u})) \underline{n}$
dw_convect ConvectTerm termsNavierStokes	<virtual>, <state>	$\int_{\Omega} ((\underline{u} \cdot \nabla) \underline{u}) \cdot \underline{v}$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
ev_def_grad  DeformationGradientTerm terms_hyperelastic_base	<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  DiffusionTerm termsLaplace	<material>, <virtual>, <state> <material>, <parameter_1>, <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  DiffusionCoupling termsLaplace	<material>, <virtual>, <state> <material>, <state>, <virtual> <material>, <parameter_1>, <parameter_2>	$\int_{\Omega} p K_j \nabla_j q$
dw_diffusion_r  DiffusionRTerm termsLaplace	<material>, <virtual>	$\int_{\Omega} K_j \nabla_j q$
d_diffusion_sa  DiffusionSATerm termsAcoustic	<material>, <parameter_q>, <parameter_p>, <parameter_v>	$\int_{\Omega} [(\operatorname{div} \underline{\underline{v}}) K_{ij} \nabla_i q \nabla_j p - K_{ij} (\nabla_j \underline{\underline{v}} \nabla q) \nabla_i p - K_{ij} \nabla_j q (\nabla_i \underline{\underline{v}} \nabla p)]$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
ev_diffusion_velocity DiffusionVelocityTerm termsLaplace	<material>, <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> $-K_{ij} \nabla_j \bar{p}$
ev_div DivTerm termsNavierStokes	<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> $(\nabla \cdot \underline{u}) _{qp}$
dw_div DivOperatorTerm termsNavierStokes	<opt_material>, <virtual>	$\int_{\Omega} \nabla \cdot \underline{v} \text{ or } \int_{\Omega} c \nabla \cdot \underline{v}$
dw_div_grad DivGradTerm termsNavierStokes	<opt_material>, <virtual>, <state> <opt_material>, <parameter_1>, <parameter_2>	$\int_{\Omega} \nu \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nu \nabla \underline{u} : \nabla \underline{w}$ $\int_{\Omega} \nabla \underline{v} : \nabla \underline{u}, \int_{\Omega} \nabla \underline{u} : \nabla \underline{w}$
Continued on next page		



Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_electric_source ElectricSourceTerm termsElectric	<material>, <virtual>, <parameter>	$\int_{\Omega} cs(\nabla \phi)^2$
ev_grad GradTerm termsNavierStokes	<parameter>	$\int_{\Omega} \nabla p \text{ or } \int_{\Omega} \nabla \underline{w}$ <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> $(\nabla p) _{qp} \text{ or } \nabla \underline{w} _{qp}$
ev_integrate_mat IntegrateMatTerm termsBasic	<material>, <parameter>	$\int_{\Omega} m$ <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} m / \int_{T_K} 1</math></p> $m _{qp}$
dw_jump SurfaceJumpTerm termsSurface	<opt_material>, <virtual>, <state_1>, <state_2>	$\int_{\Gamma} c q(p_1 - p_2)$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_laplace LaplaceTerm sfepy.terms.termsLaplace	<opt_material>, <virtual>, <state> <opt_material>, <parameter_1>, <parameter_2>	$\int_{\Omega} c \nabla q \cdot \nabla p, \int_{\Omega} c \nabla \bar{p} \cdot \nabla r$
dw_lin_convect  LinearConvectTerm termsNavierStokes	<virtual>, <parameter>, <state>	$\int_{\Omega} ((\underline{b} \cdot \nabla) \underline{u}) \cdot \underline{v}$  $((\underline{b} \cdot \nabla) \underline{u}) _{qp}$
dw_lin_elastic  LinearElasticTerm termsLinElasticity	<material>, <virtual>, <state> <material>, <parameter_1>, <parameter_2>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$
dw_lin_elastic_eth  LinearElasticETHTerm termsLinElasticity	<ts>, <material_0>, <material_1>, <virtual>, <state>	$\int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$
dw_lin_elastic_iso  LinearElasticIsotropicTerm termsLinElasticity	<material_1>, <material_2>, <virtual>, <state>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u}) \text{ with } D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}$
dw_lin_elastic_th  LinearElasticTHTerm termsLinElasticity	<ts>, <material>, <virtual>, <state>	$\int_{\Omega} \left[ \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{u}(\tau)) d\tau \right] e_{ij}(\underline{v})$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_lin_prestress  LinearPrestressTerm termsLinElasticity	<material>, <virtual> <material>, <parameter>	$\int_{\Omega} \sigma_{ij} e_{ij}(\underline{v})$
dw_lin_strain_fib  LinearStrainFiberTerm termsLinElasticity	<material_1>, <material_2>, <virtual>	$\int_{\Omega} D_{ijkl} e_{ij}(\underline{v}) (d_k d_l)$
dw_new_diffusion  NewDiffusionTerm terms_new	<material>, <virtual>, <state>	
dw_new_lin_elastic  NewLinearElasticTerm terms_new	<material>, <virtual>, <state>	
dw_new_mass NewMassTerm terms_new	<virtual>, <state>	
dw_new_mass_scalar  NewMassScalarTerm terms_new	<virtual>, <state>	
dw_non_penetration  NonPenetrationTerm terms_constraints	<opt_material>, <virtual>, <state> <opt_material>, <state>, <virtual>	$\int_{\Gamma} c \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} c \hat{\lambda} \underline{n} \cdot \underline{u}$ $\int_{\Gamma} \lambda \underline{n} \cdot \underline{v}, \int_{\Gamma} \hat{\lambda} \underline{n} \cdot \underline{u}$
d_of_ns_surf_min_d_press  NSOFSurfMinDPressTerm termsAdjointNavierStokes	<material_1>, <material_2>, <parameter>	$\delta \Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} b_{press} \right)$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_of_ns_surf_min_d_press_diff  NSOFSurfMinDPressDiffTerm termsAdjointNavierStokes	<material>, <virtual>	$w\delta_p\Psi(p)\circ q$
dw_permeability_r  PermeabilityRTerm termsLaplace	<material>, <virtual>, <index>	$\int_{\Omega} K_{ij}\nabla_j q$
dw_piezo_coupling  PiezoCouplingTerm termsPiezo	<material>, <virtual>, <state> <material>, <state>, <virtual> <material>, <parameter_v>, <parameter_s>	$\int_{\Omega} g_{kij} e_{ij}(\underline{v})\nabla_k p, \int_{\Omega} g_{kij} e_{ij}(\underline{u})\nabla_k q$
dw_point_load  ConcentratedPointLoadTerm termsPoint	<material>, <virtual>	$\underline{f}^i = \underline{\bar{f}}^i \quad \forall \text{ FE node } i \text{ in a region}$
dw_point_lspring  LinearPointSpringTerm termsPoint	<material>, <virtual>, <state>	$\underline{f}^i = -k\underline{u}^i \quad \forall \text{ FE node } i \text{ in a region}$
dw_s_dot_grad_i_s  ScalarDotGradIScalarTerm terms_dot	<material>, <virtual>, <state>	$Z^i = \int_{\Omega} q\nabla_i p$
d_sd_convect SDConvectTerm termsAdjointNavierStokes	<parameter_u>, <parameter_w>, <parameter_mesh_velocity>	$\int_{\Omega_D} [u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \mathcal{V}) - u_k \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i]$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
d_sd_div SDDivTerm termsAdjointNavierStokes	<parameter_u>, <parameter_p>, <parameter_mesh_velocity>	$\int_{\Omega_D} p[(\nabla \cdot \underline{w})(\nabla \cdot \underline{v}) - \frac{\partial \mathcal{V}_k}{\partial x_i} \frac{\partial w_i}{\partial x_k}]$
d_sd_div_grad SDDivGradTerm termsAdjointNavierStokes	<material_1>, <material_2>, <parameter_u>, <parameter_w>, <parameter_mesh_velocity>	$wv \int_{\Omega_D} [\frac{\partial u_i}{\partial x_k} \frac{\partial w_i}{\partial x_k} (\nabla \cdot \underline{v}) - \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} \frac{\partial w_i}{\partial x_k} - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_l}{\partial x_k} \frac{\partial w_i}{\partial x_k}]$
d_sd_lin_elastic SDLinearElasticTerm termsLinElasticity	<material>, <parameter_w>, <parameter_u>, <parameter_mesh_velocity>	$\int_{\Omega} \hat{D}_{ijkl} e_{ij}(\underline{v}) e_{kl}(\underline{u})$ $\hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \underline{v}) - D_{ijkq} \frac{\partial \mathcal{V}_l}{\partial x_q} - D_{iqkl} \frac{\partial \mathcal{V}_j}{\partial x_q}$
d_sd_st_grad_div SDGradDivStabilizationTerm termsAdjointNavierStokes	<material>, <parameter_u>, <parameter_w>, <parameter_mesh_velocity>	$\gamma \int_{\Omega} [(\nabla \cdot \underline{u})(\nabla \cdot \underline{w})(\nabla \cdot \underline{v}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot \underline{w}) - (\nabla \cdot \underline{u}) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i}]$
d_sd_st_pspg_c SDPSPGCStabilizationTerm termsAdjointNavierStokes	<material>, <parameter_b>, <parameter_u>, <parameter_r>, <parameter_mesh_velocity>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K [\frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i)(\nabla \cdot \underline{v}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k)]$
d_sd_st_pspg_p SDPSPGPStabilizationTerm termsAdjointNavierStokes	<material>, <parameter_r>, <parameter_p>, <parameter_mesh_velocity>	$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K [(\nabla r \cdot \nabla p)(\nabla \cdot \underline{v}) - \frac{\partial r}{\partial x_k} (\nabla \mathcal{V}_k \cdot \nabla p) - (\nabla r \cdot \nabla \mathcal{V}_k) \frac{\partial p}{\partial x_k}]$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
d_sd_st_supg_c  SDSUPGCStabilizationTerm termsAdjointNavierStokes	<material>, <parameter_b>, <parameter_u>, <parameter_w>, <parameter_mesh_velocity>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [(\underline{b} \cdot \nabla u_k)(\underline{b} \cdot \nabla w_k)(\nabla \cdot \underline{v}) - (\underline{b} \cdot \nabla v_i) \frac{\partial u_k}{\partial x_i} (\underline{b} \cdot \nabla w_k)]$
d_sd_surface_ndot  SDSurfaceNormalDotTerm termsSurface	<material>, <parameter>, <parameter_mesh_velocity>	$\int_{\Gamma} p \underline{c} \cdot \underline{n} \nabla \cdot \underline{v}$
d_sd_volume_dot SDDotVolumeTerm termsAdjointNavierStokes	<parameter_1>, <parameter_2>, <parameter_mesh_velocity>	$\int_{\Omega_D} p q (\nabla \cdot \underline{v}), \int_{\Omega_D} (\underline{u} \cdot \underline{w})(\nabla \cdot \underline{v})$
dw_st_adj1_supg_p  SUPGPAdj1StabilizationTerm termsAdjointNavierStokes	<material>, <virtual>, <state>, <parameter>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p(\underline{v} \cdot \nabla \underline{w})$
dw_st_adj2_supg_p  SUPGPAdj2StabilizationTerm termsAdjointNavierStokes	<material>, <virtual>, <parameter>, <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla r(\underline{v} \cdot \nabla \underline{u})$
dw_st_adj_supg_c  SUPGCAdjStabilizationTerm termsAdjointNavierStokes	<material>, <virtual>, <parameter>, <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [((\underline{v} \cdot \nabla) \underline{u})((\underline{u} \cdot \nabla) \underline{w}) + ((\underline{u} \cdot \nabla) \underline{u})((\underline{v} \cdot \nabla) \underline{w})]$
dw_st_grad_div  GradDivStabilizationTerm termsNavierStokes	<material>, <virtual>, <state>	$\gamma \int_{\Omega} (\nabla \cdot \underline{u}) \cdot (\nabla \cdot \underline{v})$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_st_pspg_c PSPGCStabilizationTerm termsNavierStokes	<material>, <virtual>, <parameter>, <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$
dw_st_pspg_p PSPGPStabilizationTerm termsNavierStokes	<opt_material>, <virtual>, <state> <opt_material>, <parameter_1>, <parameter_2>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q$
dw_st_supg_c SUPGCStabilizationTerm termsNavierStokes	<material>, <virtual>, <parameter>, <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v})$
dw_st_supg_p SUPGPStabilizationTerm termsNavierStokes	<material>, <virtual>, <parameter>, <state>	$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K \nabla p \cdot ((\underline{b} \cdot \nabla) \underline{v})$
dw_stokes StokesTerm termsNavierStokes	<opt_material>, <virtual>, <state> <opt_material>, <state>, <virtual> <opt_material>, <parameter_v>, <parameter_s>	$\int_{\Omega} p \nabla \cdot \underline{v}, \int_{\Omega} q \nabla \cdot \underline{u} \text{ or } \int_{\Omega} c p \nabla \cdot \underline{v}, \int_{\Omega} c q \nabla \cdot \underline{u}$
d_sum_vals SumNodalValuesTerm termsBasic	<parameter>	
d_surface SurfaceTerm termsBasic	<parameter>	$\int_{\Gamma} 1$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_surface_dot  DotProductSurfaceTerm terms_dot	<opt_material>, <virtual>, <state> <opt_material>, <parameter_1>, <parameter_2>	$\int_{\Gamma} qp, \int_{\Gamma} \underline{v} \cdot \underline{u}, \int_{\Gamma} \underline{v} \cdot \underline{np}, \int_{\Gamma} \underline{qn} \cdot \underline{u}, \int_{\Gamma} pr, \int_{\Gamma} \underline{u} \cdot \underline{w}, \int_{\Gamma} \underline{w} \cdot \underline{np}$ $\int_{\Gamma} cqp, \int_{\Gamma} \underline{cv} \cdot \underline{u}, \int_{\Gamma} cpr, \int_{\Gamma} \underline{cu} \cdot \underline{w}$ $\int_{\Gamma} \underline{v} \cdot \underline{M} \cdot \underline{u}, \int_{\Gamma} \underline{u} \cdot \underline{M} \cdot \underline{w}$
d_surface_flux SurfaceFluxTerm termsLaplace	<material>, <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_integrate  IntegrateSurfaceTerm termsBasic	<opt_material>, <parameter>	$\int_{\Gamma} y, \int_{\Gamma} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n}$ $\int_{\Gamma} cy, \int_{\Gamma} \underline{cy}, \int_{\Gamma} \underline{cy} \cdot \underline{n} \text{ flux}$ <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}) /</math></p> <p>vector for <math>K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} \underline{cy} / \int_{T_K} 1, \int_{T_K} (\underline{cy} \cdot \underline{n}) /</math></p> <p><math>y _{qp}, \underline{y} _{qp}, (\underline{y} \cdot \underline{n}) _{qp} \text{ flux}</math></p> <p><math>cy _{qp}, \underline{cy} _{qp}, (\underline{cy} \cdot \underline{n}) _{qp} \text{ flux}</math></p>

Continued on next page



Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_surface_integrate  IntegrateSurfaceOperatorTerm <a href="#">termsBasic</a>	<opt_material>, <virtual>	$\int_{\Gamma} q \text{ or } \int_{\Gamma} cq$
dw_surface_laplace  SurfaceLaplaceLayerTerm <a href="#">termsAcoustic</a>	<material>, <virtual>, <state> <material>, <parameter_2>, <parameter_1>	$\int_{\Gamma} c \partial_{\alpha} q \partial_{\alpha} p, \alpha = 1, \dots, N - 1$
dw_surface_lcouple  SurfaceCoupleLayerTerm <a href="#">termsAcoustic</a>	<material>, <virtual>, <state> <material>, <state>, <virtual> <material>, <parameter_1>, <parameter_2>	$\int_{\Gamma} cq \partial_{\alpha} p, \int_{\Gamma} c \partial_{\alpha} p q, \int_{\Gamma} c \partial_{\alpha} r s, \alpha = 1, \dots, N - 1$
dw_surface_ltr  LinearTractionTerm <a href="#">termsSurface</a>	<material>, <virtual>	$\int_{\Gamma} \underline{v} \cdot \underline{\sigma} \cdot \underline{n}$
di_surface_moment  SurfaceMomentTerm <a href="#">termsBasic</a>	<parameter>, <shift>	$\int_{\Gamma} \underline{n} (\underline{x} - \underline{x}_0)$
dw_surface_ndot  SurfaceNormalDotTerm <a href="#">termsSurface</a>	<material>, <virtual> <material>, <parameter>	$\int_{\Gamma} qc \cdot \underline{n}$
dw_tl_bulk_active  BulkActiveTLTerm <a href="#">terms_hyperelastic_tl</a>	<material>, <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_tl_bulk_penalty  BulkPenaltyTLTerm terms_hyperelastic_tl	<material>, <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$
dw_tl_bulk_pressure  BulkPressureTLTerm terms_hyperelastic_tl	<virtual>, <state>, <state_p>	$\int_{\Omega} S_{ij}(p) \delta E_{ij}(\underline{u}; \underline{v})$
dw_tl_diffusion DiffusionTLTerm terms_hyperelastic_tl	<material_1>, <material_2>, <virtual>, <state>, <parameter>	$\int_{\Omega} \underline{K}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial X} \frac{\partial p}{\partial X}$
dw_tl_fib_a  FibresActiveTLTerm terms_fibres	<material_1>, <material_2>, <material_3>, <material_4>, <material_5>, <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$
dw_tl_he_mooney_rivlin  MooneyRivlinTLTerm terms_hyperelastic_tl	<material>, <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$
dw_tl_he_neohook  NeoHookeanTLTerm terms_hyperelastic_tl	<material>, <virtual>, <state>	$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$
dw_tl_membrane TLMembraneTerm terms_membrane	<material_a1>, <material_a2>, <material_h0>, <virtual>, <state>	

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_tl_surface_traction SurfaceTractionTLTerm terms_hyperelastic_tl	<material>, <virtual>, <state>	$\int_{\Gamma} \underline{\nu} \cdot \underline{F}^{-1} \cdot \underline{\sigma} \cdot \underline{\nu} J$
dw_tl_volume VolumeTLTerm terms_hyperelastic_tl	<virtual>, <state>	$\int_{\Omega} q J(\underline{u})$ volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$ rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$
dw_ul_bulk_penalty BulkPenaltyULTerm terms_hyperelastic_ul	<material>, <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$
dw_ul_bulk_pressure BulkPressureULTerm terms_hyperelastic_ul	<virtual>, <state>, <state_p>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$
dw_ul_compressible CompressibilityULTerm terms_hyperelastic_ul	<material>, <virtual>, <state>, <parameter_u>	$\frac{\int_{\Omega} 1}{\gamma p q}$
dw_ul_he_mooney_rivlin MooneyRivlinULTerm terms_hyperelastic_ul	<material>, <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$
dw_ul_he_neohook NeoHookeanULTerm terms_hyperelastic_ul	<material>, <virtual>, <state>	$\int_{\Omega} \mathcal{L}\tau_{ij}(\underline{u}) e_{ij}(\delta \underline{v}) / J$
Continued on next page		

Table 7.5 – continued from previous page

name/class/link	arguments	definition
dw_ul_volume VolumeULTerm terms_hyperelastic_ul	<virtual>, <state>	$\int_{\Omega} q J(\underline{u})$ volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u})$ rel_volume mode: vector for $K \leftarrow \mathcal{I}_h : \int_{T_K} J(\underline{u}) / \int_{T_K} 1$
dw_v_dot_grad_s VectorDotGradScalarTerm terms_dot	<opt_material>, <virtual>, <state> <opt_material>, <state>, <virtual> <opt_material>, <parameter_v>, <parameter_s>	$\int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q$ $\int_{\Omega} c \underline{v} \cdot \nabla p, \int_{\Omega} c \underline{u} \cdot \nabla q$ $\int_{\Omega} \underline{v} \cdot \underline{\underline{M}} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \underline{\underline{M}} \cdot \nabla q$
d_volume VolumeTerm termsBasic	<parameter>	$\int_{\Omega} 1$
dw_volume_dot DotProductVolumeTerm terms_dot	<opt_material>, <virtual>, <state> <opt_material>, <parameter_1>, <parameter_2>	$\int_{\Omega} qp, \int_{\Omega} \underline{v} \cdot \underline{u}, \int_{\Omega} pr, \int_{\Omega} \underline{u} \cdot \underline{w}$ $\int_{\Omega} cqp, \int_{\Omega} c \underline{v} \cdot \underline{u}, \int_{\Omega} cpr, \int_{\Omega} c \underline{u} \cdot \underline{w}$ $\int_{\Omega} \underline{v} \cdot \underline{\underline{M}} \cdot \underline{u}, \int_{\Omega} \underline{u} \cdot \underline{\underline{M}} \cdot \underline{w}$
dw_volume_dot_w_scalar_eth DotSPProductVolumeOperatorVolumeTerm terms_dot	<ts>, <material_0>, <material_1>, <virtual>, <state>	$\int_{\Omega} \left[ \int_0^t \mathcal{G}(t - \tau) p(\tau) d\tau \right] q$
dw_volume_dot_w_scalar_th DotSPProductVolumeOperatorVolumeTerm terms_dot	<ts>, <material>, <virtual>, <state>	$\int_{\Omega} \left[ \int_0^t \mathcal{G}(t - \tau) p(\tau) d\tau \right] q$

Continued on next page

Table 7.5 – continued from previous page

name/class/link	arguments	definition
ev_volume_integrate  IntegrateVolumeTerm termsBasic	<opt_material>, <parameter>	$\int_{\Omega} y, \int_{\Omega} \underline{y}$ $\int_{\Omega} cy, \int_{\Omega} \underline{cy}$ vector for $K \leftarrow \mathcal{I}_h$ : $\int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1$ vector for $K \leftarrow \mathcal{I}_h$ : $\int_{T_K} cy / \int_{T_K} 1, \int_{T_K} \underline{cy} / \int_{T_K} 1$ $y _{qp}, \underline{y} _{qp}$ $cy _{qp}, \underline{cy} _{qp}$
dw_volume_integrate  IntegrateVolumeOperatorTerm termsBasic	<opt_material>, <virtual>	$\int_{\Omega} q \text{ or } \int_{\Omega} cq$
dw_volume_lvf  LinearVolumeForceTerm termsVolume	<material>, <virtual>	$\int_{\Omega} \underline{f} \cdot \underline{v} \text{ or } \int_{\Omega} fq$
d_volume_surface  VolumeSurfaceTerm termsBasic	<parameter>	$\int_{\Gamma} \underline{x} \cdot \underline{n}$

### sfePy.terms.terms module

**class** `sfePy.terms.terms.CharacteristicFunction` (*region*)

**get\_local\_chunk** ()

**set\_current\_group** (*ig*)

```
class sfepy.terms.terms.ConnInfo (**kwargs)

    get_region (can_trace=True)
    get_region_name (can_trace=True)
    iter_igs ()

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 = ()

    assemble_to (asm_obj, val, iels, mode='vector', diff_var=None)

    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 subdomain lists (igs).

    call_function (out, fargs)
    call_get_fargs (args, kwargs)

    check_args ()
        Common checking to all terms.

        Check compatibility of field and term subdomain lists (igs).

    check_shapes (*args, **kwargs)
        Default implementation of function to 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 = ['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\_approximation** (*variable*, *get\_saved=False*)  
Return approximation corresponding to *variable*. Also return the corresponding geometry (actual or saved, according to *get\_saved*).

**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*)  
According to the term integration type, return either the term region cell indices or local index sequence.

**get\_conn\_info** ()

**get\_conn\_key** ()  
The key to be used in DOF connectivity information.

**get\_current\_group** ()

**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\_integral\_info** ()

Get information on the term integral.

**Returns** **kind** : 'v' or 's'

The integral kind.

**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\_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** ()

**igs** ()

**integration** = 'volume'

**iter\_groups** ()

**name** = ''

**static new** (*name*, *integral*, *region*, *\*\*kwargs*)



```

set_arg_types ()
set_current_group (ig)
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`.**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)
    set_current_group (ig)
    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', '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.vector_chunk_generator(total_size, chunk_size, shape_in, zero=False,
                                          set_shape=True, dtype=<type 'numpy.float64'>)
```

## sfepy.terms.termsAcoustic module

**class** `sfepy.terms.termsAcoustic.DiffusionSATerm` (*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_diffusion_sa</b>	( <i>material, parameter_q, parameter_p, parameter_v</i> )
-----------------------	--

### Arguments

- *material*:  $K_{ij}$
- *parameter\_q*:  $q$
- *parameter\_p*:  $p$
- *parameter\_v*:  $\underline{\mathcal{V}}$

```
arg_shapes = {'parameter_q': 1, 'material': 'D, D', 'parameter_v': 'D', 'parameter_p': 1}
```

```
arg_types = ('material', 'parameter_q', 'parameter_p', 'parameter_v')
```

```
function()
```

```
get_eval_shape(mat, parameter_q, parameter_p, parameter_v, mode=None, term_mode=None,
               diff_var=None, **kwargs)
```

```
get_fargs(mat, parameter_q, parameter_p, parameter_v, mode=None, term_mode=None,
          diff_var=None, **kwargs)
```

```
name = 'd_diffusion_sa'
```

**class** `sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm` (*name, arg\_str, integral, region, \*\*kwargs*)  
Acoustic ‘layer’ term - derivatives in surface directions.

**Definition**

$$\int_{\Gamma} c q \partial_{\alpha} p, \int_{\Gamma} c \partial_{\alpha} p q, \int_{\Gamma} c \partial_{\alpha} r s, \alpha = 1, \dots, N - 1$$

**Call signature**

<b>dw_surface_lcouple</b>	(material, virtual, state)
	(material, state, virtual)
	(material, parameter_1, parameter_2)

**Arguments 1**

- material:  $c$
- virtual:  $q$
- state:  $p$

**Arguments 2**

- material:  $c$
- virtual:  $q$
- state:  $p$

**Arguments 3**

- material:  $c$
- parameter\_1:  $s$
- parameter\_2:  $r$

**arg\_shapes** = {'parameter\_2': 1, 'state': 1, 'material': '1, 1', 'parameter\_1': 1, 'virtual': (1, 'state')}

**arg\_types** = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter\_1', 'parameter\_2'))

**geometries** = ['2\_3', '2\_4']

**get\_eval\_shape** (mat, virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**get\_fargs** (mat, virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**integration** = 'surface'

**modes** = ('bv\_ns', 'nv\_bs', 'eval')

**name** = 'dw\_surface\_lcouple'

**set\_arg\_types** ()

**class** sfepy.terms.termsAcoustic.**SurfaceLaplaceLayerTerm** (name, arg\_str, integral, region, \*\*kwargs)

Acoustic 'layer' term - derivatives in surface directions.

**Definition**

$$\int_{\Gamma} c \partial_{\alpha} q \partial_{\alpha} p, \alpha = 1, \dots, N - 1$$

**Call signature**

<b>dw_surface_laplace</b>	(material, virtual, state)
	(material, parameter_2, parameter_1)

**Arguments 1**

- material:  $c$
- virtual:  $q$
- state:  $p$

**Arguments 2**

- material:  $c$
- parameter\_1:  $q$
- parameter\_2:  $p$

```
arg_shapes = {'parameter_2': 1, 'state': 1, 'material': '1, 1', 'parameter_1': 1, 'virtual': (1, 'state')}
arg_types = [('material', 'virtual', 'state'), ('material', 'parameter_2', 'parameter_1')]
geometries = ['2_3', '2_4']
get_eval_shape (mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs (mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'surface'
modes = ('weak', 'eval')
name = 'dw_surface_laplace'
set_arg_types ()
```

**sfepy.terms.termsAdjointNavierStokes module**

```
class sfepy.terms.termsAdjointNavierStokes.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')
```

```

function()
get_fargs (virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_convect1'

```

```

class sfepy.terms.termsAdjointNavierStokes.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')
```

```
function()
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs (virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_adj_convect2'
```

```

class sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm(name, arg_str, integral, re-
                                                           gion, **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

$$w \delta_{\underline{u}} \Psi(\underline{u}) \circ \underline{v}$$

#### 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')
function()
get_fargs(mat1, mat2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_div_grad'
class sfepy.terms.termsAdjointNavierStokes.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, 'material_2': 1, 'parameter': 1}
arg_types = ('material_1', 'material_2', 'parameter')
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.termsAdjointNavierStokes.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.termsAdjointNavierStokes.NSOFSurfMinDPressTerm(name, arg_str,
                                                                    integral, region,
                                                                    **kwargs)
```

Sensitivity of  $\Psi(p)$ .

#### Definition

$$\delta\Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} b_{press} \right)$$

### Call signature

<b>d_of_ns_surf_min_d_press</b>	(material_1, material_2, parameter)
---------------------------------	-------------------------------------

### Arguments

- material\_1 :  $w$  (weight)
- material\_2 :  $b_{press}$  (given pressure)
- parameter :  $p$

**arg\_shapes** = {'material\_1': 1, 'material\_2': 1, 'parameter': 1}

**arg\_types** = ('material\_1', 'material\_2', 'parameter')

**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.termsAdjointNavierStokes.**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

$$\int_{\Omega_D} \left[ u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \mathcal{V}) - u_k \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i \right]$$

### Call signature

<b>d_sd_convect</b>	(parameter_u, parameter_w, parameter_mesh_velocity)
---------------------	---

### Arguments

- parameter\_u :  $\underline{u}$
- parameter\_w :  $\underline{w}$
- 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')

**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.termsAdjointNavierStokes.**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

$$w\nu \int_{\Omega_D} \left[ \frac{\partial u_i}{\partial x_k} \frac{\partial w_i}{\partial x_k} (\nabla \cdot \underline{v}) - \frac{\partial \mathcal{V}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} \frac{\partial w_i}{\partial x_k} - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_l}{\partial x_k} \frac{\partial w_i}{\partial x_k} \right]$$

#### 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 :  $\underline{u}$
- parameter\_w :  $\underline{w}$
- parameter\_mesh\_velocity :  $\underline{v}$

**arg\_shapes** = {'parameter\_mesh\_velocity': 'D', 'material\_1': '1, 1', 'material\_2': '1, 1', 'parameter\_w': 'D', 'parameter\_u': '1, 1'}

**arg\_types** = ('material\_1', 'material\_2', 'parameter\_u', 'parameter\_w', 'parameter\_mesh\_velocity')

**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.termsAdjointNavierStokes.**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

$$\int_{\Omega_D} p[(\nabla \cdot \underline{w})(\nabla \cdot \underline{v}) - \frac{\partial \mathcal{V}_k}{\partial x_i} \frac{\partial w_i}{\partial x_k}]$$

#### 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')
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.termsAdjointNavierStokes.SDDotVolumeTerm(name, arg_str, integral,
                                                           region, **kwargs)

```

Sensitivity (shape derivative) of dot product of scalars or vectors.

#### Definition

$$\int_{\Omega_D} pq(\nabla \cdot \underline{\mathcal{V}}), \int_{\Omega_D} (\underline{u} \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}})$$

#### 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'}
arg_types = ('parameter_1', 'parameter_2', 'parameter_mesh_velocity')
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.termsAdjointNavierStokes.SDGradDivStabilizationTerm(name,
                                                                        arg_str,
                                                                        integral,
                                                                        region,
                                                                        **kwargs)

```

Sensitivity (shape derivative) of stabilization term  $dw_{st\_grad\_div}$ .

#### Definition

$$\gamma \int_{\Omega_D} [(\nabla \cdot \underline{u})(\nabla \cdot \underline{w})(\nabla \cdot \underline{\mathcal{V}}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot \underline{w}) - (\nabla \cdot \underline{u}) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i}]$$

#### 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')

**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.termsAdjointNavierStokes.SDPSPGCStabilizationTerm(name,
                                                                    arg_str,
                                                                    integral,
                                                                    region,
                                                                    **kwargs)
```

Sensitivity (shape derivative) of stabilization terms  $dw\_st\_supg\_p$  or  $dw\_st\_pspg\_c$ .

**Definition**

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \left[ \frac{\partial r}{\partial x_i} (\underline{b} \cdot \nabla u_i) (\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\underline{b} \cdot \nabla u_i) - \frac{\partial r}{\partial x_k} (\underline{b} \cdot \nabla \mathcal{V}_k) \frac{\partial u_i}{\partial x_k} \right]$$

**Call signature**

<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 :  $\underline{u}$
- parameter\_r :  $r$
- parameter\_mesh\_velocity :  $\underline{\mathcal{V}}$
- 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')

**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.termsAdjointNavierStokes.SDPSPGPStabilizationTerm (name,
                                                                    arg_str,
                                                                    integral,
                                                                    region,
                                                                    **kwargs)
```

Sensitivity (shape derivative) of stabilization term  $dw_{st\_pspg\_p}$ .

#### 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

<b>d_sd_st_pspg_p</b>	(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')
```

```
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.termsAdjointNavierStokes.SDSUPGCStabilizationTerm (name,
                                                                    arg_str,
                                                                    integral,
                                                                    region,
                                                                    **kwargs)
```

Sensitivity (shape derivative) of stabilization term  $dw_{st\_supg\_c}$ .

#### 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, parameter_mesh_velocity)
-----------------------	--

**Arguments**

- material :  $\delta_K$
- parameter\_b :  $\underline{b}$
- parameter\_u :  $\underline{u}$
- parameter\_w :  $\underline{w}$
- parameter\_mesh\_velocity :  $\underline{v}$
- 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')

**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.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm(name,
                                                                    arg_str,
                                                                    integral,
                                                                    region,
                                                                    **kwargs)
```

Adjoint term to SUPG stabilization term  $dw\_st\_supg\_c$ .

**Definition**

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K [((\underline{v} \cdot \nabla) \underline{u})((\underline{u} \cdot \nabla) \underline{w}) + ((\underline{u} \cdot \nabla) \underline{u})((\underline{v} \cdot \nabla) \underline{w})]$$

**Call signature**

<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')

**function**()

```
geometries = ['3_4', '3_8']
```

```
get_fargs (mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj_supg_c'
```

```
class sfepy.terms.termsAdjointNavierStokes.SUPGPAdj1StabilizationTerm (name,
                                                                    arg_str,
                                                                    integral,
                                                                    region,
                                                                    **kwargs)
```

The first adjoint term to SUPG stabilization term  $dw\_st\_supg\_p$ .

#### Definition

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \delta_K \nabla p(\underline{v} \cdot \nabla \underline{w})$$

#### Call signature

<b>dw_st_adj1_supg_p</b>	(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')
```

```
function ()
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs (mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj1_supg_p'
```

```
class sfepy.terms.termsAdjointNavierStokes.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

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \nabla r(\underline{v} \cdot \nabla \underline{u})$$

#### 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')
```

```
function()
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs(mat, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_adj2_supg_p'
```

```
sfePy.terms.termsAdjointNavierStokes.grad_as_vector(grad)
```

**sfePy.terms.termsBasic module**

```
class sfePy.terms.termsBasic.IntegrateMatTerm(name, arg_str, integral, region, **kwargs)
```

Evaluate material parameter  $m$  in a volume/surface region.

Depending on evaluation mode, integrate a material parameter over a volume/surface region ('eval'), average it in elements/faces ('el\_avg') or interpolate it into volume/surface quadrature points ('qp').

Uses reference mapping of  $y$  variable.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

**Definition**

$$\int_{\Omega} m$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} m / \int_{T_K} 1$$

$$m|_{qp}$$

**Call signature**

<b>ev_integrate_mat</b>	(material, parameter)
-------------------------	-----------------------

**Arguments**

- material :  $m$  (can have up to two dimensions)
- parameter :  $y$

```
arg_shapes = [{'material': '1, 1', 'parameter': 1}, {'material': 'D, D'}, {'material': 'S, S'}, {'material': 'D, S'}]
```

```
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_integrate_mat'
```

```
class sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm (name, arg_str, integral, re-
                                                                gion, **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.termsBasic.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

$$\int_{\Gamma} y, \int_{\Gamma} \underline{y}, \int_{\Gamma} \underline{y} \cdot \underline{n}$$

$$\int_{\Gamma} cy, \int_{\Gamma} c\underline{y}, \int_{\Gamma} c\underline{y} \cdot \underline{n} \text{ flux}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1, \int_{T_K} (\underline{y} \cdot \underline{n}) / \int_{T_K} 1$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} c\underline{y} / \int_{T_K} 1, \int_{T_K} (c\underline{y} \cdot \underline{n}) / \int_{T_K} 1$$

$$y|_{qp}, \underline{y}|_{qp}, (\underline{y} \cdot \underline{n})|_{qp} \text{ flux}$$
$$cy|_{qp}, c\underline{y}|_{qp}, (c\underline{y} \cdot \underline{n})|_{qp} \text{ flux}$$

#### 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’: 1}, {‘opt\_material’: None}, {‘opt\_material’: ‘1, 1’, ‘parameter’: ‘D’}]

**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.termsBasic.**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.termsBasic.**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

$$\int_{\Omega} y, \int_{\Omega} \underline{y}$$

$$\int_{\Omega} cy, \int_{\Omega} \underline{cy}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} y / \int_{T_K} 1, \int_{T_K} \underline{y} / \int_{T_K} 1$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} cy / \int_{T_K} 1, \int_{T_K} \underline{cy} / \int_{T_K} 1$$

$$y|_{qp}, \underline{y}|_{qp}$$

$$cy|_{qp}, \underline{cy}|_{qp}$$

### 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’: 1}, {‘opt\_material’: None}, {‘opt\_material’: ‘1, 1’, ‘parameter’: ‘D’}]

**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.termsBasic.**SumNodalValuesTerm** (name, arg\_str, integral, region, \*\*kwargs)  
Sum nodal values.

### Call signature

<b>d_sum_vals</b>	(parameter)
-------------------	-------------

### Arguments

- parameter :  $p$  or  $\underline{u}$

**arg\_shapes** = [{‘parameter’: 1}, {‘parameter’: ‘D’}]

**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.termsBasic.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

di_surface_moment	(parameter, shift)
-------------------	--------------------

#### Arguments

- parameter : any variable
- shift :  $\underline{x}_0$

```
arg_types = ('parameter', 'shift')
```

```
function ()
```

```
get_eval_shape (parameter, shift, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs (parameter, shift, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'surface'
```

```
name = 'di_surface_moment'
```

```
class sfepy.terms.termsBasic.SurfaceTerm (name, arg_str, integral, region, **kwargs)
```

Surface of a domain. Uses approximation of the parameter variable.

#### Definition

$$\int_{\Gamma} 1$$

#### Call signature

d_surface	(parameter)
-----------	-------------

#### Arguments

- parameter : any variable

```
arg_shapes = {'parameter': 1}
```

```
arg_types = ('parameter',)
```

```
integration = 'surface'
```

```
name = 'd_surface'
```

```
class sfepy.terms.termsBasic.VolumeSurfaceTerm (name, arg_str, integral, region, **kwargs)
```

Volume of a domain, using a surface integral. Uses approximation of the parameter variable.

**Definition**

$$\int_{\Gamma} \underline{x} \cdot \underline{n}$$

**Call signature**

<b>d_volume_surface</b>	(parameter)
-------------------------	-------------

**Arguments**

- parameter : any variable

```
arg_shapes = {'parameter': 1}
```

```
arg_types = ('parameter',)
```

```
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.termsBasic.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': 1}
```

```
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'
```

**sfepy.terms.termsBiot module**

**class** `sfepy.terms.termsBiot.BiotETHTerm` (*name, arg\_str, integral, region, \*\*kwargs*)

This term has the same definition as `dw_biot_th`, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

**Definition**

$$\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) p(\tau) \, d\tau \right] e_{ij}(\underline{v}) , \\ \int_{\Omega} \left[ \int_0^t \alpha_{ij}(t - \tau) e_{kl}(\underline{u}(\tau)) \, d\tau \right] q$$

**Call signature**

<b>dw_biot_eth</b>	( <i>ts, material_0, material_1, virtual, state</i> )
	( <i>ts, material_0, material_1, state, virtual</i> )

**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** = {}

**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.termsBiot.BiotStressTerm` (*name, arg\_str, integral, region, \*\*kwargs*)

Evaluate Biot stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

**Definition**

$$-\int_{\Omega} \alpha_{ij} \bar{p}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : -\int_{T_K} \alpha_{ij} \bar{p} / \int_{T_K} 1$$

$$-\alpha_{ij} \bar{p}|_{qp}$$

### Call signature

<b>ev_biot_stress</b>	(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, fmode)

**get\_fargs** (mat, parameter, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'ev\_biot\_stress'

**class** sfepy.terms.termsBiot.**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** = {}

**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.termsBiot.**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]. 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 :  $\underline{v}$
- state :  $p$

#### Arguments 2

- material :  $\alpha_{ij}$
- state :  $\underline{u}$
- virtual :  $q$

#### Arguments 3

- material :  $\alpha_{ij}$
- parameter\_v :  $\underline{u}$
- parameter\_s :  $p$

**arg\_shapes** = {'state/grad': 1, 'state/div': 'D', 'material': 'S, 1', '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\_biot'

```
set_arg_types()
```

### sfepy.terms.termsElectric module

```
class sfepy.terms.termsElectric.ElectricSourceTerm(name, arg_str, integral, region,
                                                    **kwargs)
```

Electric source term.

#### Definition

$$\int_{\Omega} cs(\nabla\phi)^2$$

#### Call signature

<b>dw_electric_source</b>	(material, virtual, parameter)
---------------------------	--------------------------------

#### 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')
```

```
function()
```

```
get_fargs(mat, virtual, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_electric_source'
```

### sfepy.terms.termsLaplace module

```
class sfepy.terms.termsLaplace.DiffusionCoupling(name, arg_str, integral, region,
                                                    **kwargs)
```

Diffusion coupling term with material parameter  $K_j$ .

#### Definition

$$\int_{\Omega} pK_j\nabla_j q$$

#### 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.termsLaplace.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')
function ()
get_fargs (mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_diffusion_r'
```

```
class sfepy.terms.termsLaplace.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.termsLaplace.**DiffusionVelocityTerm** (*name, arg\_str, integral, region, \*\*kwargs*)

Evaluate diffusion velocity.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

#### Definition

$$- \int_{\Omega} K_{ij} \nabla_j \bar{p}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : - \int_{T_K} K_{ij} \nabla_j \bar{p} / \int_{T_K} 1$$

$$-K_{ij} \nabla_j \bar{p}$$

#### 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.termsLaplace.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': 'D, D', 'state': 1, 'parameter_1': 1, 'virtual': (1, 'state'), 'parameter_2': 1 }, { 'opt_mate
```

```
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.termsLaplace.PermeabilityRTerm(name, arg_str, integral, region, **kwargs)
```

Special-purpose diffusion-like term with permeability  $K_{ij}$  (to use on the right-hand side).

#### Definition

$$\int_{\Omega} K_{ij} \nabla_j q$$

#### Call signature

<b>dw_permeability_r</b>	(material, virtual, index)
--------------------------	----------------------------

#### Arguments

- material :  $K_{ij}$
- virtual :  $q$
- index :  $i$

**arg\_types** = ('material', 'virtual', 'index')

**function** ()

**get\_fargs** (mat, virtual, index, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_permeability\_r'

**class** sfepy.terms.termsLaplace.**SurfaceFluxTerm** (name, arg\_str, integral, region, \*\*kwargs)  
Surface flux term.

Supports 'eval', 'el\_avg' and 'el' evaluation modes.

#### Definition

$$\int_{\Gamma} \underline{n} \cdot K_{ij} \nabla_j \bar{p}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p} / \int_{T_K} 1$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{n} \cdot K_{ij} \nabla_j \bar{p}$$

#### Call signature

<b>d_surface_flux</b>	(material, parameter)
-----------------------	-----------------------

#### Arguments

- material:  $K$
- parameter:  $\bar{p}$ ,

**arg\_shapes** = {'material': 'D, D', 'parameter': 1}

**arg\_types** = ('material', 'parameter')

**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.termsLinElasticity module**

**class** `sfePy.terms.termsLinElasticity.CauchyStrainTerm`(*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**

$$\int_{\Gamma} \underline{\underline{e}}(\underline{w})$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$$

$$\underline{\underline{e}}(\underline{w})|_{qp}$$

**Call signature**

<code>ev_cauchy_strain_s</code>	(parameter)
---------------------------------	-------------

**Arguments**

- parameter :  $\underline{w}$

**arg\_types** = (‘parameter’,)

**integration** = ‘surface\_extra’

**name** = ‘ev\_cauchy\_strain\_s’

**class** `sfePy.terms.termsLinElasticity.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**

$$\int_{\Omega} \underline{\underline{e}}(\underline{w})$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \underline{\underline{e}}(\underline{w}) / \int_{T_K} 1$$

$$\underline{\underline{e}}(\underline{w})|_{qp}$$

**Call signature**

<b>ev_cauchy_strain</b>	(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.termsLinElasticity.**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**

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau / \int_{T_K} 1$$

$$\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau|_{qp}$$

**Call signature**

<b>ev_cauchy_stress_eth</b>	(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** = {}

**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.termsLinElasticity.**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

$$\int_{\Omega} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau / \int_{T_K} 1$$

$$\int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(\underline{w}(\tau)) \, d\tau|_{qp}$$

### Call signature

<b>ev_cauchy_stress_th</b>	(ts, material, parameter)
----------------------------	---------------------------

### Arguments

- ts : TimeStepper instance
- material :  $\mathcal{H}_{ijkl}(\tau)$
- parameter :  $\underline{w}$

**arg\_shapes** = {}

**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.termsLinElasticity.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

$$\int_{\Omega} D_{ijkl} e_{kl}(\underline{w})$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} D_{ijkl} e_{kl}(\underline{w}) / \int_{T_K} 1$$

$$D_{ijkl} e_{kl}(\underline{w})|_{qp}$$

#### 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.termsLinElasticity.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\_types** = ('ts', 'material\_0', 'material\_1', 'virtual', 'state')

**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.termsLinElasticity.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)
---------------------------	--

**Arguments**

- `material_1` :  $\lambda$
- `material_2` :  $\mu$
- `virtual` :  $\underline{v}$
- `state` :  $\underline{u}$

**arg\_shapes** = {'material\_1': '1, 1', 'material\_2': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}

**arg\_types** = ('material\_1', 'material\_2', 'virtual', 'state')

**check\_shapes** (*lam, mu, virtual, state*)

**function** ()

**get\_fargs** (*lam, mu, virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs*)

**name** = 'dw\_lin\_elastic\_iso'

```
class sfepy.terms.termsLinElasticity.LinearElasticTHTerm(name, arg_str, integral, re-
                                                         gion, **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\_types** = ('ts', 'material', 'virtual', 'state')

**function** ()

**get\_fargs** (ts, mats, virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_lin\_elastic\_th'

**class** sfepy.terms.termsLinElasticity.**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'))

**check\_shapes** (mat, virtual, state)

```
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.termsLinElasticity.LinearPrestressTerm (name, arg_str, integral, re-  
gion, **kwargs)
```

Linear prestress term, with the prestress  $\sigma_{ij}$  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]. 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)}
```

```
arg_types = (('material', 'virtual'), ('material', 'parameter'))
```

```
check_shapes (mat, virtual)
```

```
d_lin_prestress (out, strain, mat, vg, fmode)
```

```
get_eval_shape (mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs (mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('weak', 'eval')
```

```
name = 'dw_lin_prestress'
```

```
set_arg_types ()
```

```
class sfepy.terms.termsLinElasticity.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')

**check\_shapes** (mat1, mat2, virtual)

**function** ()

**get\_fargs** (mat1, mat2, virtual, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_lin\_strain\_fib'

**class** sfepy.terms.termsLinElasticity.**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>	(material, parameter_w, parameter_u, parameter_mesh_velocity)
-------------------------	---

**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** ()

```
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'
static op_dv (vgrad, sym)
```

## sfepy.terms.termsNavierStokes module

class sfepy.terms.termsNavierStokes.**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')

**function** ()

**geometries** = ['3\_4', '3\_8']

**get\_fargs** (virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_convect'

class sfepy.terms.termsNavierStokes.**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'}

**arg\_types** = (('opt\_material', 'virtual', 'state'), ('opt\_material', 'parameter\_1', 'parameter\_2'))

**d\_div\_grad** (out, grad1, grad2, mat, vg, fmode)

**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.termsNavierStokes.**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.termsNavierStokes.**DivTerm** (name, arg\_str, integral, region, \*\*kwargs)

Evaluate divergence of a vector field.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

#### Definition

$$\int_{\Omega} \nabla \cdot \underline{u}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla \cdot \underline{u} / \int_{T_K} 1$$

$$(\nabla \cdot \underline{u})|_{qp}$$

#### Call signature

<b>ev_div</b>	(parameter)
---------------	-------------

#### Arguments

- parameter :  $\underline{u}$

**arg\_shapes** = {'parameter': 'D'}

**arg\_types** = ('parameter',)

**static function** (out, div, 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\_div'

```
class sfepy.terms.termsNavierStokes.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

<b>dw_st_grad_div</b>	(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')

**function** ()

**get\_fargs** (*gamma*, *virtual*, *state*, *mode=None*, *term\_mode=None*, *diff\_var=None*, *\*\*kwargs*)

**name** = 'dw\_st\_grad\_div'

**class** sfepy.terms.termsNavierStokes.**GradTerm** (*name*, *arg\_str*, *integral*, *region*, *\*\*kwargs*)

Evaluate gradient of a scalar or vector field.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

#### Definition

$$\int_{\Omega} \nabla p \text{ or } \int_{\Omega} \nabla \underline{w}$$

$$\text{vector for } K \leftarrow \mathcal{I}_h : \int_{T_K} \nabla p / \int_{T_K} 1 \text{ or } \int_{T_K} \nabla \underline{w} / \int_{T_K} 1$$

$$(\nabla p)|_{qp} \text{ or } \nabla \underline{w}|_{qp}$$

#### Call signature

<b>ev_grad</b>	(parameter)
----------------	-------------

#### Arguments

- parameter : *p* or *w*

**arg\_shapes** = [{'parameter': 1}, {'parameter': 'D'}]

**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.termsNavierStokes.**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')
```

```
function()
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs (virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_lin_convect'
```

```
class sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm (name, arg_str, integral,  
                                                            region, **kwargs)
```

PSPG stabilization term, convective part (  $\tau$  is a local stabilization parameter).

**Definition**

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot \nabla q$$

**Call signature**

<b>dw_st_pspg_c</b>	(material, virtual, parameter, state)
---------------------	---------------------------------------

**Arguments**

- material :  $\tau_K$
- virtual :  $q$
- parameter :  $\underline{b}$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'material': '1, 1', 'parameter': 'D', 'virtual': (1, None)}
```

```
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```
function()
```

```
geometries = ['3_4', '3_8']
```

```
get_fargs (tau, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'dw_st_pspg_c'
```

```
class sfepy.terms.termsNavierStokes.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**

$$\sum_{K \in \mathcal{T}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q$$



**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.termsNavierStokes.**SUPGCStabilizationTerm**(name, arg\_str, integral, region, \*\*kwargs)  
 SUPG stabilization term, convective part (  $\delta$  is a local stabilization parameter).

**Definition**

$$\sum_{K \in \mathcal{I}_h} \int_{T_K} \delta_K ((\underline{b} \cdot \nabla) \underline{u}) \cdot ((\underline{b} \cdot \nabla) \underline{v})$$

**Call signature**

<b>dw_st_supg_c</b>	(material, virtual, parameter, state)
---------------------	---------------------------------------

**Arguments**

- material :  $\delta_K$
- virtual :  $\underline{v}$
- parameter :  $\underline{b}$
- state :  $\underline{u}$

**arg\_shapes** = {'state': 'D', 'material': '1, 1', 'parameter': 'D', 'virtual': ('D', 'state')}

**arg\_types** = ('material', 'virtual', 'parameter', 'state')

**function**()

**geometries** = ['3\_4', '3\_8']

**get\_fargs**(delta, virtual, parameter, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_st\_supg\_c'

**class** sfepy.terms.termsNavierStokes.**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')

**function** ()

**geometries** = ['3\_4', '3\_8']

**get\_fargs** (delta, virtual, parameter, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**name** = 'dw\_st\_supg\_p'

**class** sfepy.terms.termsNavierStokes.**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, 'pa

```

arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state', 'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))
static d_eval (out, coef, vec_qp, div, vvg)
get_eval_shape (coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs (coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('grad', 'div', 'eval')
name = 'dw_stokes'
set_arg_types ()

```

## sfepy.terms.termsPiezo module

**class sfepy.terms.termsPiezo.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()
```

### **sfePy.terms.termsPoint module**

```
class sfePy.terms.termsPoint.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**

<b>dw_point_load</b>	(material, virtual)
----------------------	---------------------

#### **Arguments**

- material :  $\bar{f}^i$
- virtual :  $\underline{v}$ ,

```
arg_types = ('material', 'virtual')
```

```
check_shapes(mat, 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.termsPoint.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**

<b>dw_point_lspring</b>	(material, virtual, state)
-------------------------	----------------------------

#### **Arguments**

- material :  $k$

- virtual :  $\underline{v}$
- state :  $\underline{u}$

**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'

**class** `sfePy.terms.termsPoint.PointTermBase` (*name, arg\_str, integral, region, \*\*kwargs*)

Common methods of point terms.

**get\_integral\_info** ()

Get information on the term integral.

**Returns kind** : 'v' or 's'

The integral kind.

## sfePy.terms.termsSurface module

**class** `sfePy.terms.termsSurface.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.termsSurface.**LinearTractionTerm** (name, arg\_str, integral, region, \*\*kwargs)

Linear traction forces (weak form), where, depending on dimension of 'material' argument,  $\underline{\underline{\sigma}} \cdot \underline{n}$  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.

#### Definition

$$\int_{\Gamma} \underline{v} \cdot \underline{\underline{\sigma}} \cdot \underline{n}$$

#### Call signature

<b>dw_surface_ltr</b>	(material, virtual)
-----------------------	---------------------

#### Arguments

- material :  $\underline{\underline{\sigma}}$
- virtual :  $\underline{v}$

**arg\_shapes** = [{ 'material': 'S, 1', 'virtual': ('D', None)}, { 'material': 'D, 1' }, { 'material': '1, 1' }]

**arg\_types** = ('material', 'virtual')

**function** ()

**get\_fargs** (traction, virtual, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)

**integration** = 'surface'

**name** = 'dw\_surface\_ltr'

```
class sfepy.terms.termsSurface.SDSurfaceNormalDotTerm(name, arg_str, integral, region,
                                                         **kwargs)
```

Sensitivity of scalar traction.

#### Definition

$$\int_{\Gamma} p \underline{c} \cdot \underline{n} \nabla \cdot \underline{v}$$

#### Call signature

<b>d_sd_surface_ndot</b>	(material, parameter, parameter_mesh_velocity)
--------------------------	--

#### Arguments

- material :  $\underline{c}$
- parameter :  $p$
- parameter\_mesh\_velocity :  $\underline{v}$

```
arg_shapes = {'parameter_mesh_velocity': 'D', 'material': 'D, 1', 'parameter': 1}
```

```
arg_types = ('material', 'parameter', 'parameter_mesh_velocity')
```

```
static function (out, material, val_p, div_mv, sg)
```

```
get_eval_shape (mat, par, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs (mat, par, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'surface'
```

```
name = 'd_sd_surface_ndot'
```

```
class sfepy.terms.termsSurface.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.termsSurface.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.termsVolume module

```
class sfepy.terms.termsVolume.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 :  $\underline{f}$  or  $f$
- virtual :  $\underline{v}$  or  $q$

```
arg_shapes = [{‘material’: ‘D’, 1}, {‘virtual’: (‘D’, None)}, {‘material’: ‘1’, 1}, {‘virtual’: (1, None)}]
```

```
arg_types = (‘material’, ‘virtual’)
```

```
function()
```

```
get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = ‘dw_volume_lvf’
```

**sfepy.terms.terms\_constraints module**

```
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_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')

**check\_shapes** (*alpha, p\_outer, 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

$$\begin{aligned} \int_{\Gamma} qp, \int_{\Gamma} \underline{v} \cdot \underline{u}, \int_{\Gamma} \underline{v} \cdot \underline{np}, \int_{\Gamma} q\underline{n} \cdot \underline{u}, \int_{\Gamma} pr, \int_{\Gamma} \underline{u} \cdot \underline{w}, \int_{\Gamma} \underline{w} \cdot \underline{np} \\ \int_{\Gamma} cqp, \int_{\Gamma} \underline{cv} \cdot \underline{u}, \int_{\Gamma} cpr, \int_{\Gamma} \underline{cu} \cdot \underline{w} \\ \int_{\Gamma} \underline{v} \cdot \underline{\underline{M}} \cdot \underline{u}, \int_{\Gamma} \underline{u} \cdot \underline{\underline{M}} \cdot \underline{w} \end{aligned}$$

#### Call signature

<b>dw_surface_dot</b>	(opt_material, virtual, state)
	(opt_material, parameter_1, parameter_2)

#### Arguments 1

- material :  $c$  or  $\underline{\underline{M}}$  (optional)
- virtual :  $q$  or  $\underline{v}$
- 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\_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**

$$\begin{aligned} & \int_{\Omega} qp, \int_{\Omega} \underline{v} \cdot \underline{u}, \int_{\Omega} pr, \int_{\Omega} \underline{u} \cdot \underline{w} \\ & \int_{\Omega} cqp, \int_{\Omega} c\underline{v} \cdot \underline{u}, \int_{\Omega} cpr, \int_{\Omega} c\underline{u} \cdot \underline{w} \\ & \int_{\Omega} \underline{v} \cdot \underline{\underline{M}} \cdot \underline{u}, \int_{\Omega} \underline{u} \cdot \underline{\underline{M}} \cdot \underline{w} \end{aligned}$$

**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  $\underline{v}$
- 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\_mater

**arg\_types** = (('opt\_material', 'virtual', 'state'), ('opt\_material', 'parameter\_1', 'parameter\_2'))

**check\_shapes** (*mat*, *virtual*, *state*)

**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, material_0, material_1, virtual, 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_types = ('ts', 'material_0', 'material_1', 'virtual', 'state')

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>	<code>(ts, material, virtual, state)</code>
--	---

**Arguments**

- `ts` : `TimeStepper` instance
- `material` :  $\mathcal{G}(\tau)$
- `virtual` :  $q$
- `state` :  $p$

```
arg_types = ('ts', 'material', 'virtual', 'state')
```

```
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.VectorDotGradScalarTerm (name, arg_str, integral, region,
                                                         **kwargs)
```

Volume dot product of a vector and a gradient of scalar. Can be evaluated.

**Definition**

$$\begin{aligned} & \int_{\Omega} \underline{v} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \nabla q \\ & \int_{\Omega} c \underline{v} \cdot \nabla p, \int_{\Omega} c \underline{u} \cdot \nabla q \\ & \int_{\Omega} \underline{v} \cdot \underline{\underline{M}} \cdot \nabla p, \int_{\Omega} \underline{u} \cdot \underline{\underline{M}} \cdot \nabla q \end{aligned}$$

**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  $\underline{\underline{M}}$  (optional)
- virtual :  $\underline{v}$
- state :  $p$

**Arguments 2**

- material :  $c$  or  $\underline{\underline{M}}$  (optional)
- state :  $\underline{u}$
- virtual :  $q$

**Arguments 3**

- material :  $c$  or  $\underline{\underline{M}}$  (optional)
- parameter\_v :  $\underline{u}$
- 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

**check\_shapes** (coef, vvar, svar)

**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** ()

**sfepy.terms.terms\_fibres module**

**class** sfepy.terms.terms\_fibres.**FibresActiveTLTerm** (\*args, \*\*kwargs)

Hyperelastic active fibres term. Effective stress  $S_{ij} = Af_{\max} \exp \left\{ -\left( \frac{\epsilon - \epsilon_{\text{opt}}}{s} \right)^2 \right\} d_i d_j$ , where  $\epsilon = E_{ij} d_i d_j$  is the Green strain  $\underline{\underline{E}}$  projected to the fibre direction  $\underline{d}$ .

**Definition**

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

### Call signature

<b>dw_tl_fib_a</b>	(material_1, material_2, material_3, material_4, material_5, virtual, state)
--------------------	--

### Arguments

- material\_1 :  $f_{\max}$
- material\_2 :  $\varepsilon_{\text{opt}}$
- material\_3 :  $s$
- material\_4 :  $\underline{d}$
- 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)

**static tan\_mod\_function** (out, pars, green\_strain)

**sfepy.terms.terms\_fibres.fibre\_function** (out, pars, green\_strain, fmode)

Depending on *fmode*, compute fibre stress (0) or tangent modulus (!= 0).

### sfepy.terms.terms\_hyperelastic\_base module

**class sfepy.terms.terms\_hyperelastic\_base.DeformationGradientTerm** (name, arg\_str, integral, region, \*\*kwargs)

Deformation gradient  $\underline{\underline{F}}$  in quadrature points for *term\_mode*='def\_grad' (default) or the jacobian  $J$  if *term\_mode*='jacobian'.

Supports 'eval', 'el\_avg' and 'qp' evaluation modes.

### Definition

$$\underline{\underline{F}} = \frac{\partial \underline{x}}{\partial \underline{X}}|_{qp} = \underline{\underline{I}} + \frac{\partial \underline{u}}{\partial \underline{X}}|_{qp},$$

$$\underline{x} = \underline{X} + \underline{u}, J = \det(\underline{\underline{F}})$$

**Call signature**

<b>ev_def_grad</b>	(parameter)
--------------------	-------------

**Arguments**

- parameter : 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 (-> rezidual) 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\_family\_data** (state, cache\_name, data\_names)**Notes***data\_names* argument is ignored for now.**get\_fargs** (mat, virtual, state, mode=None, term\_mode=None, diff\_var=None, \*\*kwargs)**static integrate** (out, val\_qp, vg, fmode)



**sfepy.terms.terms\_hyperelastic\_tl module**

**class** sfepy.terms.terms\_hyperelastic\_tl.**BulkActiveTLTerm**(\*args, \*\*kwargs)  
 Hyperelastic bulk active term. Stress  $S_{ij} = AJC_{ij}^{-1}$ , where  $A$  is the activation in  $[0, F_{\max}]$ .

**Definition**

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

**Call signature**

<b>dw_tl_bulk_active</b>	(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'

**stress\_function**()

**tan\_mod\_function**()

**class** sfepy.terms.terms\_hyperelastic\_tl.**BulkPenaltyTLTerm**(\*args, \*\*kwargs)  
 Hyperelastic bulk penalty term. Stress  $S_{ij} = K(J - 1) JC_{ij}^{-1}$ .

**Definition**

$$\int_{\Omega} S_{ij}(\underline{u}) \delta E_{ij}(\underline{u}; \underline{v})$$

**Call signature**

<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'

**stress\_function**()

**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']

**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\_tl\_bulk\_pressure'

**stress\_function** ()

**tan\_mod\_u\_function** ()

**weak\_dp\_function** ()

**weak\_function** ()

**class** sfepy.terms.terms\_hyperelastic\_tl.**DiffusionTLTerm** (\*args, \*\*kwargs)

Diffusion term in the total Lagrangian formulation with linearized deformation-dependent permeability  $\underline{\underline{K}}(\underline{u}) = J \underline{\underline{F}}^{-1} \underline{\underline{k}} f(J) \underline{\underline{F}}^{-T}$ , where  $\underline{u}$  relates to the previous time step ( $n - 1$ ) and  $f(J) = \max\left(0, \left(1 + \frac{(J-1)}{N_f}\right)\right)^2$  expresses the dependence on volume compression/expansion.

**Definition**

$$\int_{\Omega} \underline{\underline{K}}(\underline{u}^{(n-1)}) : \frac{\partial q}{\partial X} \frac{\partial p}{\partial X}$$

**Call signature**

<b>dw_tl_diffusion</b>	(material_1, material_2, virtual, state, parameter)
------------------------	---

**Arguments**

- material\_1 :  $\underline{\underline{k}}$
- material\_2 :  $N_f$
- virtual :  $q$

- state :  $p$
- parameter :  $\underline{u}^{(n-1)}$

```
arg_shapes = {'material_1': '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']
```

```
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.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.

```
compute_family_data (state)
```

```
family_function()
```

```
fd_cache_name = 'tl_common'
```

```
hyperelastic_mode = 0
```

```
weak_function()
```

```
class sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm (*args, **kwargs)
```

Hyperelastic Mooney-Rivlin term. Effective stress  $S_{ij} = \kappa J^{-\frac{4}{3}} (C_{kk} \delta_{ij} - C_{ij} - \frac{2}{3} I_2 C_{ij}^{-1})$ .

#### 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'
```

```
stress_function()
```

```
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'

**stress\_function**()

**tan\_mod\_function**()

**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>	(material, virtual, state)
-------------------------------	----------------------------

#### Arguments

- material :  $\underline{\underline{\sigma}}$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

**arg\_shapes** = {'state': 'D', 'material': 'D, D', 'virtual': ('D', 'state')}

**arg\_types** = ('material', 'virtual', 'state')

**check\_shapes** (mat, virtual, state)

**compute\_family\_data** (state)

**family\_data\_names** = ['det\_f', 'inv\_f']

**family\_function**()

```

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.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>	( <i>virtual</i> , <i>state</i> )
---------------------	-----------------------------------

**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']
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>	( <i>material</i> , <i>virtual</i> , <i>state</i> )
---------------------------	---

**Arguments**

- *material* :  $K$
- *virtual* :  $\underline{v}$

- state :  $\underline{u}$

**family\_data\_names** = ['det\_f']

**name** = 'dw\_ul\_bulk\_penalty'

**stress\_function**()

**tan\_mod\_function**()

**class** sfepy.terms.terms\_hyperelastic\_ul.**BulkPressureULTerm**(\*args, \*\*kwargs)  
Hyperelastic bulk pressure term. Stress  $S_{ij} = -pJ\delta_{ij}$ .

#### 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']

**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'

**stress\_function**()

**tan\_mod\_u\_function**()

**weak\_dp\_function**()

**weak\_function**()

**class** sfepy.terms.terms\_hyperelastic\_ul.**CompressibilityULTerm**(\*args, \*\*kwargs)  
Compressibility term for the updated Lagrangian formulation

#### Definition

$$\frac{\int_{\Omega} 1}{\gamma p q}$$

#### Call signature

<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']

**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\_hyperelastic\_ul.**HyperElasticULBase** (\*args, \*\*kwargs)

Base class for all hyperelastic terms in UL formulation family.

The subclasses should have the following static method attributes: - *stress\_function()* (the stress) - *tan\_mod\_function()* (the tangent modulus)

The common (family) data are cached in the evaluate cache of state variable.

**compute\_family\_data** (state)

**family\_function** ()

**fd\_cache\_name** = 'ul\_common'

**hyperelastic\_mode** = 1

**weak\_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**

<b>dw_ul_he_mooney_rivlin</b>	(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'
```

```
stress_function()
```

```
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>	<code>(material, virtual, state)</code>
-------------------------------	---

#### Arguments

- material :  $\mu$
- virtual :  $\underline{v}$
- state :  $\underline{u}$

```
family_data_names = ['det_f', 'tr_b', 'sym_b']
```

```
name = 'dw_ul_he_neohook'
```

```
stress_function()
```

```
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>	<code>(virtual, state)</code>
---------------------------	-------------------------------

#### Arguments

- virtual :  $q$
- state :  $\underline{u}$

```
arg_shapes = {'state': 'D', 'virtual': (1, None)}
```

```
arg_types = ('virtual', 'state')
```

```
family_data_names = ['mtx_f', 'det_f']
```

```
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

<b>dw_tl_membrane</b>	(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\_new module

todo:

- get row variable, col variable (if diff\_var)
- determine out shape
- set current group to all variable arguments
- **loop over row/col dofs:**
  - call term

? how to deal with components of (vector) variables?

(\*) for given group, Variable has to be able to:

- evaluate value in quadrature points
- evaluate gradient in quadrature points
- ? evaluate divergence in quadrature points
- ? lazy evaluation, cache!

?? base function gradients in space elements stored now in terms - in geometry, shared dict of geometries belongs to Equations -> where to cache stuff? - in variables!

```
class sfepy.terms.terms_new.NewDiffusionTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

dw_new_diffusion	(material, virtual, state)
------------------	----------------------------

arg\_types = ('material', 'virtual', 'state')

name = 'dw\_new\_diffusion'

```
class sfepy.terms.terms_new.NewLinearElasticTerm(name, arg_str, integral, region,
                                                  **kwargs)
```

Call signature

dw_new_lin_elastic	(material, virtual, state)
--------------------	----------------------------

arg\_types = ('material', 'virtual', 'state')

name = 'dw\_new\_lin\_elastic'

```
class sfepy.terms.terms_new.NewMassScalarTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

dw_new_mass_scalar	(virtual, state)
--------------------	------------------

arg\_types = ('virtual', 'state')

name = 'dw\_new\_mass\_scalar'

```
class sfepy.terms.terms_new.NewMassTerm(name, arg_str, integral, region, **kwargs)
```

Call signature

dw_new_mass	(virtual, state)
-------------	------------------

Works for both scalar and vector variables.

arg\_types = ('virtual', 'state')

name = 'dw\_new\_mass'

---

```
class sfepy.terms.terms_new.NewTerm(name, arg_str, integral, region, **kwargs)
```

```
    evaluate (mode='eval', diff_var=None, **kwargs)
```

```
    get_geometry (variable)
```

```
    get_geometry_key (variable)
```

```
    integrate (val_qp, variable)
```

```
    set_current_group (ig)
```

```
        Set current group for the term and all variables in its arguments.
```

### 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.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.d_biot_div()
```

```
sfepy.terms.extmods.terms.d_diffusion()
```

```
sfepy.terms.extmods.terms.d_diffusion_sa()
```

```
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_div()
```

```
sfepy.terms.extmods.terms.d_sd_div_grad()
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_surf_laplace()
sfepy.terms.extmods.terms.d_surf_lcouple()
sfepy.terms.extmods.terms.d_surface_flux()
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_tl()
sfepy.terms.extmods.terms.dq_finite_strain_ul()
sfepy.terms.extmods.terms.dq_grad()
sfepy.terms.extmods.terms.dq_state_in_qp()
sfepy.terms.extmods.terms.dq_tl_finite_strain_surface()
sfepy.terms.extmods.terms.dq_tl_he_stress_bulk()
sfepy.terms.extmods.terms.dq_tl_he_stress_bulk_active()
sfepy.terms.extmods.terms.dq_tl_he_stress_mooney_rivlin()
sfepy.terms.extmods.terms.dq_tl_he_stress_neohook()
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_bulk()
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_bulk_active()
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_mooney_rivlin()
sfepy.terms.extmods.terms.dq_tl_he_tan_mod_neohook()
sfepy.terms.extmods.terms.dq_tl_stress_bulk_pressure()
sfepy.terms.extmods.terms.dq_tl_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_diffusion()
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_elastic_iso()
sfepy.terms.extmods.terms.dw_lin_prestress()
sfepy.terms.extmods.terms.dw_lin_strain_fib()
sfepy.terms.extmods.terms.dw_permeability_r()
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_surf_laplace()
sfepy.terms.extmods.terms.dw_surf_lcouple()
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_tl_diffusion()
sfepy.terms.extmods.terms.dw_tl_surface_traction()
sfepy.terms.extmods.terms.dw_tl_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.mulATB_integrate()  
sfepy.terms.extmods.terms.term_ns_asm_convect()  
sfepy.terms.extmods.terms.term_ns_asm_div_grad()
```

# NOTES

Contents:

## 8.1 Linear Combination Boundary Conditions

By linear combination boundary conditions (LCBCs) we mean conditions of the following type:

$$\sum_{i=1}^L a_i u_i(\underline{x}) = 0, \quad \forall \underline{x} \in \omega, \quad (8.1)$$

where  $a_i$  are given coefficients,  $u_i(\underline{x})$  are some components of unknown fields evaluated point-wise in points  $\underline{x} \in \omega$ , and  $\omega$  is a subset of the entire domain  $\Omega$  (e.g. a part of its boundary). Note that the coefficients  $a_i$  can also depend on  $\underline{x}$ .

A typical example is the *no penetration* condition  $\underline{u}(\underline{x}) \cdot \underline{n}(\underline{x}) = 0$ , where  $\underline{u}$  are the velocity (or displacement) components, and  $\underline{n}$  is the unit normal outward to the domain boundary.

### 8.1.1 Enforcing LCBCs

There are several methods to enforce the conditions:

- penalty method
- substitution method

We use the substitution method, e.i. we choose  $j$  such that  $a_j \neq 0$  and substitute

$$u_j(\underline{x}) = -\frac{1}{a_j} \sum_{i=1, i \neq j}^L a_i u_i(\underline{x}), \quad \forall \underline{x} \in \omega, \quad (8.2)$$

into the equations. This is done, however, after the discretization by the finite element method, as explained below.

Let us denote  $c_i = \frac{a_i}{a_j}$  ( $j$  is fixed). Then

$$u_j(\underline{x}) = - \sum_{i=1, i \neq j}^L c_i u_i(\underline{x}), \quad \forall \underline{x} \in \omega. \quad (8.3)$$

### 8.1.2 Weak Formulation

We multiply (8.3) by a test function  $v_j$  and integrate the equation over  $\omega$  to obtain

$$\int_{\omega} v_j u_j(\underline{x}) = - \int_{\omega} v_j \sum_{i=1, i \neq j}^L c_i u_i(\underline{x}), \quad \forall v_j \in H(\omega), \quad (8.4)$$

where  $H(\omega)$  is some suitable function space (e.g. the same space which  $u_j$  belongs to).

### 8.1.3 Finite Element Approximation

On a finite element  $T_K$  (surface or volume) we have  $u_i(\underline{x}) = \sum_{k=1}^N u_i^k \phi^k(\underline{x})$ , where  $\phi^k$  are the local (element) base functions. Using the more compact matrix notation  $\mathbf{u}_i = [u_i^1, \dots, u_i^N]$ ,  $\boldsymbol{\varphi} = [\varphi^1, \dots, \varphi^N]^T$  we have  $u_i(\underline{x}) = \boldsymbol{\varphi}(\underline{x}) \mathbf{u}_i$  and similarly  $v_i(\underline{x}) = \boldsymbol{\varphi}(\underline{x}) \mathbf{v}_i$ .

The relation (8.3), restricted to  $T_K$ , can be then written (we omit the  $\underline{x}$  arguments) as

$$\int_{T_K} \mathbf{v}_j^T \boldsymbol{\varphi}^T \boldsymbol{\varphi} \mathbf{u}_j = - \int_{T_K} \mathbf{v}_j^T \boldsymbol{\varphi}^T \sum_{i=1, i \neq j}^L c_i \boldsymbol{\varphi} \mathbf{u}_i, \quad \forall \mathbf{v}_j \quad (8.5)$$

As (8.5) holds for any  $\mathbf{v}_j$ , we have a linear system to solve. After denoting the “mass” matrices  $\mathbf{M} = \int_{T_K} \boldsymbol{\varphi}^T \boldsymbol{\varphi}$ ,  $\mathbf{M}_i = \int_{T_K} c_i \boldsymbol{\varphi}^T \boldsymbol{\varphi}$  the linear system is

$$\mathbf{M} \mathbf{u}_j = - \sum_{i=1, i \neq j}^L \mathbf{M}_i \mathbf{u}_i. \quad (8.6)$$

Then the individual coefficients  $\mathbf{u}_j$  can be expressed as

$$\mathbf{u}_j = -\mathbf{M}^{-1} \sum_{i=1, i \neq j}^L \mathbf{M}_i \mathbf{u}_i. \quad (8.7)$$

### 8.1.4 Implementation

Above is the general treatment. The code uses its somewhat simplified version described here. If the coefficients  $c_i$  are constant in the element  $T_K$ , i.e.  $c_i(\underline{x}) = \bar{c}_i$  for  $x \in T_K$ , we can readily see that  $\mathbf{M}_i = \bar{c}_i \mathbf{M}$ . The relation (8.7) then reduces to

$$\mathbf{u}_j = -\mathbf{M}^{-1} \sum_{i=1, i \neq j}^L \bar{c}_i \mathbf{M} \mathbf{u}_i = \sum_{i=1, i \neq j}^L \bar{c}_i \mathbf{u}_i, \quad (8.8)$$

hence we can work with the individual components of the coefficient vectors (= degrees of freedom) only, as the above relation means, that  $u_j^k = \bar{c}_i u_i^k$  for  $k = 1, \dots, N$ .

PDF version of the documentation: `sfePy_manual.pdf`



# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*



# PYTHON MODULE INDEX

## b

blockgen, 320  
build\_helpers, 317

## c

config, 320  
convert\_mesh, 321  
cylindergen, 321

## e

edit\_identifiers, 321  
evalForms, 322  
extractor, 314

## f

findSurf, 319

## g

gen\_gallery, 322  
gen\_lobattold\_c, 323  
gen\_term\_table, 324  
genPerMesh, 319

## h

homogen, 314

## p

phonon, 314  
plot\_condition\_numbers, 324  
plotPerfusionCoefs, 320  
postproc, 314  
probe, 315

## r

runTests, 316

## s

save\_basis, 324  
schroedinger, 317  
sfepy.applications.application, 325  
sfepy.applications.pde\_solver\_app, 326

sfepy.base.base, 326  
sfepy.base.compat, 332  
sfepy.base.conf, 334  
sfepy.base.getch, 336  
sfepy.base.goptions, 336  
sfepy.base.ioutils, 337  
sfepy.base.log, 338  
sfepy.base.log\_plotter, 339  
sfepy.base.parse\_conf, 340  
sfepy.base.plotutils, 340  
sfepy.base.progressbar, 340  
sfepy.base.reader, 342  
sfepy.base.testing, 343  
sfepy.config, 325  
sfepy.fem.conditions, 343  
sfepy.fem.dof\_info, 345  
sfepy.fem.domain, 349  
sfepy.fem.equations, 350  
sfepy.fem.evaluate, 355  
sfepy.fem.evaluate\_variable, 358  
sfepy.fem.extmods.\_fmfield, 418  
sfepy.fem.extmods.assemble, 418  
sfepy.fem.extmods.bases, 418  
sfepy.fem.extmods.lobatto\_bases, 420  
sfepy.fem.extmods.mappings, 421  
sfepy.fem.extmods.mesh, 421  
sfepy.fem.facets, 358  
sfepy.fem.fe\_surface, 360  
sfepy.fem.fea, 360  
sfepy.fem.fields\_base, 362  
sfepy.fem.fields\_hierarchic, 367  
sfepy.fem.fields\_nodal, 368  
sfepy.fem.functions, 370  
sfepy.fem.geometry\_element, 370  
sfepy.fem.global\_interp, 371  
sfepy.fem.history, 371  
sfepy.fem.integrals, 372  
sfepy.fem.linearizer, 372  
sfepy.fem.mappings, 373  
sfepy.fem.mass\_operator, 375  
sfepy.fem.materials, 375

`sfepy.fem.mesh`, 378  
`sfepy.fem.meshio`, 383  
`sfepy.fem.parseEq`, 388  
`sfepy.fem.parseReg`, 388  
`sfepy.fem.periodic`, 389  
`sfepy.fem.poly_spaces`, 389  
`sfepy.fem.probes`, 391  
`sfepy.fem.problemDef`, 394  
`sfepy.fem.projections`, 400  
`sfepy.fem.quadratures`, 401  
`sfepy.fem.refine`, 402  
`sfepy.fem.region`, 403  
`sfepy.fem.simplex_cubature`, 406  
`sfepy.fem.state`, 407  
`sfepy.fem.utils`, 409  
`sfepy.fem.variables`, 410  
`sfepy.homogenization.band_gaps_app`, 422  
`sfepy.homogenization.coefficients`, 423  
`sfepy.homogenization.coefs_base`, 424  
`sfepy.homogenization.coefs_elastic`, 426  
`sfepy.homogenization.coefs_perfusion`, 427  
`sfepy.homogenization.coefs_phononic`, 427  
`sfepy.homogenization.convolution`, 431  
`sfepy.homogenization.engine`, 432  
`sfepy.homogenization.homogen_app`, 432  
`sfepy.homogenization.micmac`, 433  
`sfepy.homogenization.recovery`, 433  
`sfepy.homogenization.utils`, 435  
`sfepy.interactive.session`, 437  
`sfepy.linalg.eigen`, 437  
`sfepy.linalg.extmods.crcm`, 444  
`sfepy.linalg.geometry`, 438  
`sfepy.linalg.sparse`, 440  
`sfepy.linalg.utils`, 442  
`sfepy.mechanics.contact_planes`, 445  
`sfepy.mechanics.elastic_constants`, 445  
`sfepy.mechanics.friction`, 445  
`sfepy.mechanics.matcoefs`, 446  
`sfepy.mechanics.membranes`, 448  
`sfepy.mechanics.tensors`, 450  
`sfepy.mechanics.units`, 452  
`sfepy.mesh.femlab`, 454  
`sfepy.mesh.geom_tools`, 455  
`sfepy.mesh.mesh_generators`, 457  
`sfepy.mesh.mesh_tools`, 460  
`sfepy.mesh.meshutils`, 461  
`sfepy.mesh.splinebox`, 465  
`sfepy.optimize.freeFormDef`, 466  
`sfepy.optimize.shapeOptim`, 467  
`sfepy.physics.energy`, 467  
`sfepy.physics.potentials`, 468  
`sfepy.physics.radial_mesh`, 468  
`sfepy.physics.schroedinger_app`, 470  
`sfepy.postprocess.dataset_manager`, 470  
`sfepy.postprocess.domain_specific`, 471  
`sfepy.postprocess.plot_dofs`, 473  
`sfepy.postprocess.plot_facets`, 473  
`sfepy.postprocess.sources`, 473  
`sfepy.postprocess.time_history`, 475  
`sfepy.postprocess.utils`, 476  
`sfepy.postprocess.viewer`, 476  
`sfepy.solvers.eigen`, 480  
`sfepy.solvers.ls`, 481  
`sfepy.solvers.nls`, 484  
`sfepy.solvers.optimize`, 487  
`sfepy.solvers.oseen`, 489  
`sfepy.solvers.petsc_worker`, 489  
`sfepy.solvers.semismooth_newton`, 490  
`sfepy.solvers.solvers`, 490  
`sfepy.solvers.ts`, 491  
`sfepy.solvers.ts_solvers`, 492  
`sfepy.terms.extmods.terms`, 583  
`sfepy.terms.terms`, 513  
`sfepy.terms.terms_constraints`, 565  
`sfepy.terms.terms_dot`, 566  
`sfepy.terms.terms_fibres`, 570  
`sfepy.terms.terms_hyperelastic_base`, 571  
`sfepy.terms.terms_hyperelastic_tl`, 573  
`sfepy.terms.terms_hyperelastic_ul`, 577  
`sfepy.terms.terms_membrane`, 581  
`sfepy.terms.terms_new`, 582  
`sfepy.terms.terms_th`, 583  
`sfepy.terms.termsAcoustic`, 518  
`sfepy.terms.termsAdjointNavierStokes`, 520  
`sfepy.terms.termsBasic`, 530  
`sfepy.terms.termsBiot`, 536  
`sfepy.terms.termsElectric`, 539  
`sfepy.terms.termsLaplace`, 539  
`sfepy.terms.termsLinElasticity`, 544  
`sfepy.terms.termsNavierStokes`, 552  
`sfepy.terms.termsPiezo`, 559  
`sfepy.terms.termsPoint`, 560  
`sfepy.terms.termsSurface`, 561  
`sfepy.terms.termsVolume`, 564  
`sfepy.terms.utils`, 583  
`sfepy.version`, 325  
`shaper`, 317  
`show_authors`, 324  
`simple`, 317  
`sync_module_docs`, 324

**t**  
`test_install`, 320

# PYTHON MODULE INDEX

## b

blockgen, 320  
build\_helpers, 317

## c

config, 320  
convert\_mesh, 321  
cylindergen, 321

## e

edit\_identifiers, 321  
evalForms, 322  
extractor, 314

## f

findSurf, 319

## g

gen\_gallery, 322  
gen\_lobattold\_c, 323  
gen\_term\_table, 324  
genPerMesh, 319

## h

homogen, 314

## p

phonon, 314  
plot\_condition\_numbers, 324  
plotPerfusionCoefs, 320  
postproc, 314  
probe, 315

## r

runTests, 316

## s

save\_basis, 324  
schroedinger, 317  
sfepy.applications.application, 325  
sfepy.applications.pde\_solver\_app, 326

sfepy.base.base, 326  
sfepy.base.compat, 332  
sfepy.base.conf, 334  
sfepy.base.getch, 336  
sfepy.base.goptions, 336  
sfepy.base.ioutils, 337  
sfepy.base.log, 338  
sfepy.base.log\_plotter, 339  
sfepy.base.parse\_conf, 340  
sfepy.base.plotutils, 340  
sfepy.base.progressbar, 340  
sfepy.base.reader, 342  
sfepy.base.testing, 343  
sfepy.config, 325  
sfepy.fem.conditions, 343  
sfepy.fem.dof\_info, 345  
sfepy.fem.domain, 349  
sfepy.fem.equations, 350  
sfepy.fem.evaluate, 355  
sfepy.fem.evaluate\_variable, 358  
sfepy.fem.extmods.\_fmfield, 418  
sfepy.fem.extmods.assemble, 418  
sfepy.fem.extmods.bases, 418  
sfepy.fem.extmods.lobatto\_bases, 420  
sfepy.fem.extmods.mappings, 421  
sfepy.fem.extmods.mesh, 421  
sfepy.fem.facets, 358  
sfepy.fem.fe\_surface, 360  
sfepy.fem.fea, 360  
sfepy.fem.fields\_base, 362  
sfepy.fem.fields\_hierarchic, 367  
sfepy.fem.fields\_nodal, 368  
sfepy.fem.functions, 370  
sfepy.fem.geometry\_element, 370  
sfepy.fem.global\_interp, 371  
sfepy.fem.history, 371  
sfepy.fem.integrals, 372  
sfepy.fem.linearizer, 372  
sfepy.fem.mappings, 373  
sfepy.fem.mass\_operator, 375  
sfepy.fem.materials, 375

`sfepy.fem.mesh`, 378  
`sfepy.fem.meshio`, 383  
`sfepy.fem.parseEq`, 388  
`sfepy.fem.parseReg`, 388  
`sfepy.fem.periodic`, 389  
`sfepy.fem.poly_spaces`, 389  
`sfepy.fem.probes`, 391  
`sfepy.fem.problemDef`, 394  
`sfepy.fem.projections`, 400  
`sfepy.fem.quadratures`, 401  
`sfepy.fem.refine`, 402  
`sfepy.fem.region`, 403  
`sfepy.fem.simplex_cubature`, 406  
`sfepy.fem.state`, 407  
`sfepy.fem.utils`, 409  
`sfepy.fem.variables`, 410  
`sfepy.homogenization.band_gaps_app`, 422  
`sfepy.homogenization.coefficients`, 423  
`sfepy.homogenization.coefs_base`, 424  
`sfepy.homogenization.coefs_elastic`, 426  
`sfepy.homogenization.coefs_perfusion`, 427  
`sfepy.homogenization.coefs_phononic`, 427  
`sfepy.homogenization.convolution`, 431  
`sfepy.homogenization.engine`, 432  
`sfepy.homogenization.homogen_app`, 432  
`sfepy.homogenization.micmac`, 433  
`sfepy.homogenization.recovery`, 433  
`sfepy.homogenization.utils`, 435  
`sfepy.interactive.session`, 437  
`sfepy.linalg.eigen`, 437  
`sfepy.linalg.extmods.crcm`, 444  
`sfepy.linalg.geometry`, 438  
`sfepy.linalg.sparse`, 440  
`sfepy.linalg.utils`, 442  
`sfepy.mechanics.contact_planes`, 445  
`sfepy.mechanics.elastic_constants`, 445  
`sfepy.mechanics.friction`, 445  
`sfepy.mechanics.matcoefs`, 446  
`sfepy.mechanics.membranes`, 448  
`sfepy.mechanics.tensors`, 450  
`sfepy.mechanics.units`, 452  
`sfepy.mesh.femlab`, 454  
`sfepy.mesh.geom_tools`, 455  
`sfepy.mesh.mesh_generators`, 457  
`sfepy.mesh.mesh_tools`, 460  
`sfepy.mesh.meshutils`, 461  
`sfepy.mesh.splinebox`, 465  
`sfepy.optimize.freeFormDef`, 466  
`sfepy.optimize.shapeOptim`, 467  
`sfepy.physics.energy`, 467  
`sfepy.physics.potentials`, 468  
`sfepy.physics.radial_mesh`, 468  
`sfepy.physics.schroedinger_app`, 470  
`sfepy.postprocess.dataset_manager`, 470  
`sfepy.postprocess.domain_specific`, 471  
`sfepy.postprocess.plot_dofs`, 473  
`sfepy.postprocess.plot_facets`, 473  
`sfepy.postprocess.sources`, 473  
`sfepy.postprocess.time_history`, 475  
`sfepy.postprocess.utils`, 476  
`sfepy.postprocess.viewer`, 476  
`sfepy.solvers.eigen`, 480  
`sfepy.solvers.ls`, 481  
`sfepy.solvers.nls`, 484  
`sfepy.solvers.optimize`, 487  
`sfepy.solvers.oseen`, 489  
`sfepy.solvers.petsc_worker`, 489  
`sfepy.solvers.semismooth_newton`, 490  
`sfepy.solvers.solvers`, 490  
`sfepy.solvers.ts`, 491  
`sfepy.solvers.ts_solvers`, 492  
`sfepy.terms.extmods.terms`, 583  
`sfepy.terms.terms`, 513  
`sfepy.terms.terms_constraints`, 565  
`sfepy.terms.terms_dot`, 566  
`sfepy.terms.terms_fibres`, 570  
`sfepy.terms.terms_hyperelastic_base`, 571  
`sfepy.terms.terms_hyperelastic_tl`, 573  
`sfepy.terms.terms_hyperelastic_ul`, 577  
`sfepy.terms.terms_membrane`, 581  
`sfepy.terms.terms_new`, 582  
`sfepy.terms.terms_th`, 583  
`sfepy.terms.termsAcoustic`, 518  
`sfepy.terms.termsAdjointNavierStokes`, 520  
`sfepy.terms.termsBasic`, 530  
`sfepy.terms.termsBiot`, 536  
`sfepy.terms.termsElectric`, 539  
`sfepy.terms.termsLaplace`, 539  
`sfepy.terms.termsLinElasticity`, 544  
`sfepy.terms.termsNavierStokes`, 552  
`sfepy.terms.termsPiezo`, 559  
`sfepy.terms.termsPoint`, 560  
`sfepy.terms.termsSurface`, 561  
`sfepy.terms.termsVolume`, 564  
`sfepy.terms.utils`, 583  
`sfepy.version`, 325  
`shaper`, 317  
`show_authors`, 324  
`simple`, 317  
`sync_module_docs`, 324

**t**  
`test_install`, 320

# INDEX

## Symbols

`__call__()` (sfepy.solvers.nls.Newton method), 485  
`__call__()` (sfepy.solvers.nls.ScipyBroyden method), 487  
`__init__()` (sfepy.solvers.nls.Newton method), 486  
`__init__()` (sfepy.solvers.nls.ScipyBroyden method), 487  
`__module__` (sfepy.solvers.nls.Newton attribute), 486  
`__module__` (sfepy.solvers.nls.ScipyBroyden attribute), 487

## A

AbaqusMeshIO (class in sfepy.fem.meshio), 383  
AcousticBandGapsApp (class in sfepy.homogenization.band\_gaps\_app), 422  
AcousticMassLiquidTensor (class in sfepy.homogenization.coefs\_phononic), 427  
AcousticMassTensor (class in sfepy.homogenization.coefs\_phononic), 427  
action() (sfepy.fem.mass\_operator.MassOperator method), 375  
activate() (sfepy.postprocess.dataset\_manager.DatasetManager method), 470  
adapt\_time\_step() (in module sfepy.solvers.ts\_solvers), 493  
AdaptiveTimeSteppingSolver (class in sfepy.solvers.ts\_solvers), 492  
add\_array() (sfepy.postprocess.dataset\_manager.DatasetManager method), 470  
add\_data\_to\_dataset() (sfepy.postprocess.sources.GenericFileSource method), 474  
add\_e() (sfepy.fem.region.Region method), 403  
add\_from\_bc() (sfepy.fem.dof\_info.LCBCOperators method), 347  
add\_glyphs() (in module sfepy.postprocess.viewer), 479  
add\_group() (sfepy.base.log.Log method), 338  
add\_iso\_surface() (in module sfepy.postprocess.viewer), 479  
add\_n() (sfepy.fem.region.Region method), 403  
add\_scalar\_cut\_plane() (in module sfepy.postprocess.viewer), 479  
add\_strain\_rs() (in module sfepy.homogenization.recovery), 433  
add\_stress\_p() (in module sfepy.homogenization.recovery), 433  
add\_subdomains\_surface() (in module sfepy.postprocess.viewer), 479  
add\_surf() (in module sfepy.postprocess.viewer), 479  
add\_text() (in module sfepy.postprocess.viewer), 479  
add\_vector\_cut\_plane() (in module sfepy.postprocess.viewer), 479  
addline() (sfepy.mesh.geom\_tools.geometry method), 455  
addlines() (sfepy.mesh.geom\_tools.geometry method), 455  
addphysicalsurface() (sfepy.mesh.geom\_tools.geometry method), 455  
addphysicalvolume() (sfepy.mesh.geom\_tools.geometry method), 455  
addpoint() (sfepy.mesh.geom\_tools.geometry method), 455  
addpoints() (sfepy.mesh.geom\_tools.geometry method), 455  
addsurface() (sfepy.mesh.geom\_tools.geometry method), 455  
addsurfaces() (sfepy.mesh.geom\_tools.geometry method), 455  
addvolume() (sfepy.mesh.geom\_tools.geometry method), 455  
addvolumes() (sfepy.mesh.geom\_tools.geometry method), 455  
AdjConvect1Term (class in sfepy.terms.termsAdjointNavierStokes), 520  
AdjConvect2Term (class in sfepy.terms.termsAdjointNavierStokes), 521  
AdjDivGradTerm (class in sfepy.terms.termsAdjointNavierStokes), 521  
advance() (sfepy.fem.equations.Equations method), 351  
advance() (sfepy.fem.problemDef.ProblemDefinition method), 394  
advance() (sfepy.fem.variables.Variable method), 415  
advance() (sfepy.fem.variables.Variables method), 416  
advance() (sfepy.terms.terms.Term method), 514  
advance\_eth\_data() (sfepy.terms.terms\_th.ETHTerm method), 583

`alloc_extra_data()` (sfepy.fem.extmods.mappings.CMappingare\_close() (in module sfepy.solvers.oseen), 489  
method), 421

`ANSYSCDBMeshIO` (class in sfepy.fem.meshio), 383

`any_from_args()` (sfepy.fem.poly\_spaces.PolySpace  
static method), 390

`any_from_conf()` (sfepy.homogenization.coefs\_base.MinAppBase  
static method), 425

`any_from_conf()` (sfepy.solvers.solvers.Solver static  
method), 491

`any_from_filename()` (sfepy.fem.meshio.MeshIO static  
method), 385

`append()` (sfepy.base.base.Container method), 327

`append()` (sfepy.fem.dof\_info.LCBCOperators method),  
348

`append()` (sfepy.fem.history.History method), 371

`append()` (sfepy.physics.potentials.CompoundPotential  
method), 468

`append()` (sfepy.terms.terms.Terms method), 517

`append_bubbles()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_declarations()` (in module gen\_lobatto1d\_c), 323

`append_edges()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_faces()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_lists()` (in module gen\_lobatto1d\_c), 323

`append_polys()` (in module gen\_lobatto1d\_c), 323

`append_raw()` (sfepy.fem.dof\_info.DofInfo method), 345

`append_tp_bubbles()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_tp_edges()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_tp_faces()` (sfepy.fem.poly\_spaces.LagrangeNodes  
static method), 389

`append_variable()` (sfepy.fem.dof\_info.DofInfo method),  
346

`Application` (class in sfepy.applications.application), 325

`AppliedLoadTensor` (class in  
sfepy.homogenization.coefs\_phononic), 427

`apply_ebc()` (sfepy.fem.equations.Equations method),  
351

`apply_ebc()` (sfepy.fem.state.State method), 407

`apply_ebc()` (sfepy.fem.variables.FieldVariable method),  
410

`apply_ebc()` (sfepy.fem.variables.Variables method), 416

`apply_ic()` (sfepy.fem.equations.Equations method), 351

`apply_ic()` (sfepy.fem.state.State method), 407

`apply_ic()` (sfepy.fem.variables.FieldVariable method),  
410

`apply_ic()` (sfepy.fem.variables.Variables method), 416

`apply_to_sequence()` (in module sfepy.linalg.utils), 442

`approximate_exponential()` (in module  
sfepy.homogenization.convolution), 432

`Approximation` (class in sfepy.fem.fea), 360

`arg_shapes` (sfepy.terms.terms.Term attribute), 514

`arg_shapes` (sfepy.terms.terms\_constraints.NonPenetrationTerm  
attribute), 565

`arg_shapes` (sfepy.terms.terms\_dot.BCNewtonTerm at-  
tribute), 566

`arg_shapes` (sfepy.terms.terms\_dot.DotProductVolumeTerm  
attribute), 567

`arg_shapes` (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm  
attribute), 569

`arg_shapes` (sfepy.terms.terms\_dot.VectorDotGradScalarTerm  
attribute), 570

`arg_shapes` (sfepy.terms.terms\_fibres.FibresActiveTLTerm  
attribute), 571

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientTerm  
attribute), 572

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase  
attribute), 572

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm  
attribute), 574

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm  
attribute), 575

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm  
attribute), 576

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm  
attribute), 577

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm  
attribute), 578

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm  
attribute), 579

`arg_shapes` (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm  
attribute), 580

`arg_shapes` (sfepy.terms.terms\_membrane.TLMembraneTerm  
attribute), 581

`arg_shapes` (sfepy.terms.termsAcoustic.DiffusionSATerm  
attribute), 518

`arg_shapes` (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm  
attribute), 519

`arg_shapes` (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm  
attribute), 520

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.AdjConvect1Term  
attribute), 520

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term  
attribute), 521

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm  
attribute), 521

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm  
attribute), 522

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPressD  
attribute), 522

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPressT  
attribute), 523

`arg_shapes` (sfepy.terms.termsAdjointNavierStokes.SDConvectTerm  
attribute), 523



arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm attribute), 524

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDDivTerm attribute), 525

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm attribute), 525

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDGradDivStabilizationTerm attribute), 526

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabilizationTerm attribute), 526

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDPSPGStabilizationTerm attribute), 527

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SDSUPGCStabilizationTerm attribute), 528

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SUPGCStabilizationTerm attribute), 528

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SUPGPAd1StabilizationTerm attribute), 529

arg\_shapes (sfepy.terms.termsAdjointNavierStokes.SUPGPAd2StabilizationTerm attribute), 530

arg\_shapes (sfepy.terms.termsBasic.IntegrateMatTerm attribute), 530

arg\_shapes (sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm attribute), 531

arg\_shapes (sfepy.terms.termsBasic.IntegrateSurfaceTerm attribute), 532

arg\_shapes (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm attribute), 532

arg\_shapes (sfepy.terms.termsBasic.IntegrateVolumeTerm attribute), 533

arg\_shapes (sfepy.terms.termsBasic.SumNodalValuesTerm attribute), 533

arg\_shapes (sfepy.terms.termsBasic.SurfaceTerm attribute), 534

arg\_shapes (sfepy.terms.termsBasic.VolumeSurfaceTerm attribute), 535

arg\_shapes (sfepy.terms.termsBasic.VolumeTerm attribute), 535

arg\_shapes (sfepy.terms.termsBiot.BiotETHTerm attribute), 536

arg\_shapes (sfepy.terms.termsBiot.BiotStressTerm attribute), 537

arg\_shapes (sfepy.terms.termsBiot.BiotTerm attribute), 538

arg\_shapes (sfepy.terms.termsBiot.BiotTHTerm attribute), 538

arg\_shapes (sfepy.terms.termsElectric.ElectricSourceTerm attribute), 539

arg\_shapes (sfepy.terms.termsLaplace.DiffusionCoupling attribute), 540

arg\_shapes (sfepy.terms.termsLaplace.DiffusionRTerm attribute), 540

arg\_shapes (sfepy.terms.termsLaplace.DiffusionTerm attribute), 541

arg\_shapes (sfepy.terms.termsLaplace.DiffusionVelocityTerm attribute), 541

arg\_shapes (sfepy.terms.termsLaplace.LaplaceTerm attribute), 542

arg\_shapes (sfepy.terms.termsLaplace.SurfaceFluxTerm attribute), 543

arg\_shapes (sfepy.terms.termsLinElasticity.CauchyStrainTerm attribute), 545

arg\_shapes (sfepy.terms.termsLinElasticity.CauchyStressETHTerm attribute), 546

arg\_shapes (sfepy.terms.termsLinElasticity.CauchyStressTerm attribute), 547

arg\_shapes (sfepy.terms.termsLinElasticity.CauchyStressTHTerm attribute), 546

arg\_shapes (sfepy.terms.termsLinElasticity.LinearElasticIsotropicTerm attribute), 548

arg\_shapes (sfepy.terms.termsLinElasticity.LinearElasticTerm attribute), 549

arg\_shapes (sfepy.terms.termsLinElasticity.LinearPrestressTerm attribute), 550

arg\_shapes (sfepy.terms.termsLinElasticity.LinearStrainFiberTerm attribute), 551

arg\_shapes (sfepy.terms.termsLinElasticity.SDLinearElasticTerm attribute), 551

arg\_shapes (sfepy.terms.termsNavierStokes.ConvectTerm attribute), 552

arg\_shapes (sfepy.terms.termsNavierStokes.DivGradTerm attribute), 553

arg\_shapes (sfepy.terms.termsNavierStokes.DivOperatorTerm attribute), 553

arg\_shapes (sfepy.terms.termsNavierStokes.DivTerm attribute), 554

arg\_shapes (sfepy.terms.termsNavierStokes.GradDivStabilizationTerm attribute), 554

arg\_shapes (sfepy.terms.termsNavierStokes.GradTerm attribute), 555

arg\_shapes (sfepy.terms.termsNavierStokes.LinearConvectTerm attribute), 556

arg\_shapes (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm attribute), 556

arg\_shapes (sfepy.terms.termsNavierStokes.StokesTerm attribute), 558

arg\_shapes (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm attribute), 557

arg\_shapes (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm attribute), 558

arg\_shapes (sfepy.terms.termsPiezo.PiezoCouplingTerm attribute), 559

arg\_shapes (sfepy.terms.termsSurface.ContactPlaneTerm attribute), 562

arg\_shapes (sfepy.terms.termsSurface.LinearTractionTerm attribute), 562

arg\_shapes (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm attribute), 563

`arg_shapes` (sfepy.terms.termsSurface.SurfaceNormalDotTerm attribute), 518  
`arg_shapes` (sfepy.terms.termsSurface.SurfaceJumpTerm attribute), 563  
`arg_shapes` (sfepy.terms.termsSurface.SurfaceJumpTerm attribute), 564  
`arg_shapes` (sfepy.terms.termsVolume.LinearVolumeForceTerm attribute), 519  
`arg_shapes` (sfepy.terms.termsVolume.LinearVolumeForceTerm attribute), 565  
`arg_types` (sfepy.terms.terms.Term attribute), 514  
`arg_types` (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 520  
`arg_types` (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 565  
`arg_types` (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 521  
`arg_types` (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 566  
`arg_types` (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 521  
`arg_types` (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 567  
`arg_types` (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 522  
`arg_types` (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 567  
`arg_types` (sfepy.terms.terms\_dot.DotSPProductVolumeOperatorTerm attribute), 522  
`arg_types` (sfepy.terms.terms\_dot.DotSPProductVolumeOperatorTerm attribute), 568  
`arg_types` (sfepy.terms.terms\_dot.DotSPProductVolumeOperatorTerm attribute), 569  
`arg_types` (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm attribute), 523  
`arg_types` (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm attribute), 569  
`arg_types` (sfepy.terms.terms\_dot.VectorDotGradScalarTerm attribute), 523  
`arg_types` (sfepy.terms.terms\_dot.VectorDotGradScalarTerm attribute), 570  
`arg_types` (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 524  
`arg_types` (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 571  
`arg_types` (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientsTerm attribute), 525  
`arg_types` (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientsTerm attribute), 572  
`arg_types` (sfepy.terms.terms\_hyperelastic\_base.HyperElasticityTerm attribute), 526  
`arg_types` (sfepy.terms.terms\_hyperelastic\_base.HyperElasticityTerm attribute), 572  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 527  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 574  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 528  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 575  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 528  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 576  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 529  
`arg_types` (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 577  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 530  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 578  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 530  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 579  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 531  
`arg_types` (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 580  
`arg_types` (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 532  
`arg_types` (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 581  
`arg_types` (sfepy.terms.terms\_new.NewDiffusionTerm attribute), 533  
`arg_types` (sfepy.terms.terms\_new.NewDiffusionTerm attribute), 582  
`arg_types` (sfepy.terms.terms\_new.NewLinearElasticTerm attribute), 533  
`arg_types` (sfepy.terms.terms\_new.NewLinearElasticTerm attribute), 582  
`arg_types` (sfepy.terms.terms\_new.NewMassScalarTerm attribute), 534  
`arg_types` (sfepy.terms.terms\_new.NewMassScalarTerm attribute), 582  
`arg_types` (sfepy.terms.terms\_new.NewMassTerm attribute), 534  
`arg_types` (sfepy.terms.terms\_new.NewMassTerm attribute), 582  
`arg_types` (sfepy.terms.termsAcoustic.DiffusionSATerm attribute), 535  
`arg_types` (sfepy.terms.termsAcoustic.DiffusionSATerm attribute), 582  
`arg_types` (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 536  
`arg_types` (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 582  
`arg_types` (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 537  
`arg_types` (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 582  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjConvect1Term attribute), 538  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjConvect1Term attribute), 520  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term attribute), 539  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term attribute), 521  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm attribute), 540  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm attribute), 521  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm attribute), 541  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm attribute), 522  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresDiffTerm attribute), 542  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresDiffTerm attribute), 522  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresTerm attribute), 543  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresTerm attribute), 523  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDConvectTerm attribute), 544  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDConvectTerm attribute), 523  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm attribute), 545  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm attribute), 524  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDivTerm attribute), 546  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDivTerm attribute), 525  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm attribute), 547  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm attribute), 525  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDGradDivStabilizationTerm attribute), 548  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDGradDivStabilizationTerm attribute), 526  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabilizationTerm attribute), 549  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabilizationTerm attribute), 526  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDPSPGPStabilizationTerm attribute), 550  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDPSPGPStabilizationTerm attribute), 527  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDSUPGCStabilizationTerm attribute), 551  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SDSUPGCStabilizationTerm attribute), 528  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm attribute), 552  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm attribute), 528  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj1StabilizationTerm attribute), 553  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj1StabilizationTerm attribute), 529  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj2StabilizationTerm attribute), 554  
`arg_types` (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj2StabilizationTerm attribute), 530  
`arg_types` (sfepy.terms.termsBasic.IntegrateMatTerm attribute), 555  
`arg_types` (sfepy.terms.termsBasic.IntegrateMatTerm attribute), 530  
`arg_types` (sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm attribute), 556  
`arg_types` (sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm attribute), 531  
`arg_types` (sfepy.terms.termsBasic.IntegrateSurfaceTerm attribute), 557  
`arg_types` (sfepy.terms.termsBasic.IntegrateSurfaceTerm attribute), 532  
`arg_types` (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm attribute), 558  
`arg_types` (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm attribute), 532  
`arg_types` (sfepy.terms.termsBasic.IntegrateVolumeTerm attribute), 559  
`arg_types` (sfepy.terms.termsBasic.IntegrateVolumeTerm attribute), 533  
`arg_types` (sfepy.terms.termsBasic.SumNodalValuesTerm attribute), 560  
`arg_types` (sfepy.terms.termsBasic.SumNodalValuesTerm attribute), 533  
`arg_types` (sfepy.terms.termsBasic.SurfaceMomentTerm attribute), 561  
`arg_types` (sfepy.terms.termsBasic.SurfaceMomentTerm attribute), 534  
`arg_types` (sfepy.terms.termsBasic.SurfaceTerm attribute), 562  
`arg_types` (sfepy.terms.termsBasic.SurfaceTerm attribute), 535

- tribute), 534
- arg\_types (sfepy.terms.termsBasic.VolumeSurfaceTerm attribute), 535
- arg\_types (sfepy.terms.termsBasic.VolumeTerm attribute), 535
- arg\_types (sfepy.terms.termsBiot.BiotETHTerm attribute), 536
- arg\_types (sfepy.terms.termsBiot.BiotStressTerm attribute), 537
- arg\_types (sfepy.terms.termsBiot.BiotTerm attribute), 538
- arg\_types (sfepy.terms.termsBiot.BiotTHTerm attribute), 538
- arg\_types (sfepy.terms.termsElectric.ElectricSourceTerm attribute), 539
- arg\_types (sfepy.terms.termsLaplace.DiffusionCoupling attribute), 540
- arg\_types (sfepy.terms.termsLaplace.DiffusionRTerm attribute), 540
- arg\_types (sfepy.terms.termsLaplace.DiffusionTerm attribute), 541
- arg\_types (sfepy.terms.termsLaplace.DiffusionVelocityTerm attribute), 541
- arg\_types (sfepy.terms.termsLaplace.LaplaceTerm attribute), 542
- arg\_types (sfepy.terms.termsLaplace.PermeabilityRTerm attribute), 543
- arg\_types (sfepy.terms.termsLaplace.SurfaceFluxTerm attribute), 543
- arg\_types (sfepy.terms.termsLinElasticity.CauchyStrainSTerm attribute), 544
- arg\_types (sfepy.terms.termsLinElasticity.CauchyStrainTerm attribute), 545
- arg\_types (sfepy.terms.termsLinElasticity.CauchyStressETHTerm attribute), 546
- arg\_types (sfepy.terms.termsLinElasticity.CauchyStressTerm attribute), 547
- arg\_types (sfepy.terms.termsLinElasticity.CauchyStressTHTerm attribute), 546
- arg\_types (sfepy.terms.termsLinElasticity.LinearElasticETHTerm attribute), 548
- arg\_types (sfepy.terms.termsLinElasticity.LinearElasticIsotropicTerm attribute), 548
- arg\_types (sfepy.terms.termsLinElasticity.LinearElasticTerm attribute), 549
- arg\_types (sfepy.terms.termsLinElasticity.LinearElasticTHTerm attribute), 549
- arg\_types (sfepy.terms.termsLinElasticity.LinearPrestressTerm attribute), 550
- arg\_types (sfepy.terms.termsLinElasticity.LinearStrainFiberTerm attribute), 551
- arg\_types (sfepy.terms.termsLinElasticity.SDLinearElasticTerm attribute), 551
- arg\_types (sfepy.terms.termsNavierStokes.ConvectTerm attribute), 552
- arg\_types (sfepy.terms.termsNavierStokes.DivGradTerm attribute), 553
- arg\_types (sfepy.terms.termsNavierStokes.DivOperatorTerm attribute), 553
- arg\_types (sfepy.terms.termsNavierStokes.DivTerm attribute), 554
- arg\_types (sfepy.terms.termsNavierStokes.GradDivStabilizationTerm attribute), 554
- arg\_types (sfepy.terms.termsNavierStokes.GradTerm attribute), 555
- arg\_types (sfepy.terms.termsNavierStokes.LinearConvectTerm attribute), 556
- arg\_types (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm attribute), 556
- arg\_types (sfepy.terms.termsNavierStokes.StokesTerm attribute), 558
- arg\_types (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm attribute), 557
- arg\_types (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm attribute), 558
- arg\_types (sfepy.terms.termsPiezo.PiezoCouplingTerm attribute), 559
- arg\_types (sfepy.terms.termsPoint.ConcentratedPointLoadTerm attribute), 560
- arg\_types (sfepy.terms.termsPoint.LinearPointSpringTerm attribute), 561
- arg\_types (sfepy.terms.termsSurface.ContactPlaneTerm attribute), 562
- arg\_types (sfepy.terms.termsSurface.LinearTractionTerm attribute), 562
- arg\_types (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm attribute), 563
- arg\_types (sfepy.terms.termsSurface.SurfaceNormalDotTerm attribute), 563
- arg\_types (sfepy.terms.termsSurface.SurfaceJumpTerm attribute), 564
- arg\_types (sfepy.terms.termsVolume.LinearVolumeForceTerm attribute), 565
- argsort\_rows() (in module sfepy.linalg.utils), 442
- arpack\_eigs() (in module sfepy.linalg.eigen), 437
- as\_float\_or\_complex() (in module sfepy.base.base), 329
- as\_float\_or\_complex() (in module sfepy.base.base), 329
- assemble1d() (in module sfepy.linalg.utils), 442
- assemble\_by\_blocks() (in module sfepy.fem.evaluate), 355
- assemble\_matrix() (in module sfepy.fem.extmods.assemble), 418
- assemble\_matrix\_complex() (in module sfepy.fem.extmods.assemble), 418
- assemble\_to() (sfepy.terms.terms.Term method), 514
- assemble\_vector() (in module sfepy.fem.extmods.assemble), 418
- assemble\_vector\_complex() (in module sfepy.fem.extmods.assemble), 418

- sfepy.fem.extmods.assemble), 418  
 assert\_() (in module sfepy.base.base), 329  
 assign\_args() (sfepy.terms.terms.Term method), 514  
 assign\_args() (sfepy.terms.terms.Terms method), 517  
 assign\_standard\_hooks() (in module sfepy.applications.pde\_solver\_app), 326  
 associateelements() (sfepy.mesh.meshutils.bound method), 461  
 associateelements\_key() (sfepy.mesh.meshutils.bound method), 461  
 associateelements\_key\_fast() (sfepy.mesh.meshutils.bound method), 461  
 augknt() (sfepy.mesh.splinebox.SplineBox static method), 465  
 average() (sfepy.mesh.meshutils.mesh method), 462  
 average\_qp\_to\_vertices() (sfepy.fem.fields\_base.SurfaceField method), 365  
 average\_qp\_to\_vertices() (sfepy.fem.fields\_base.VolumeField method), 365  
 average\_to\_vertices() (sfepy.fem.fields\_nodal.H1DiscontinuousField method), 368  
 average\_vectors() (sfepy.mesh.meshutils.mesh method), 462  
 average\_vertex\_var\_in\_cells() (in module sfepy.postprocess.time\_history), 475  
 AVSUCDMeshIO (class in sfepy.fem.meshio), 383  
 axissym() (in module sfepy.mesh.femlab), 454
- ## B
- BandGaps (class in sfepy.homogenization.coefs\_phononic), 428  
 Bar (class in sfepy.base.progressbar), 340  
 barycentric\_coors() (in module sfepy.linalg.geometry), 438  
 BasicEvaluator (class in sfepy.fem.evaluate), 355  
 BCNewtonTerm (class in sfepy.terms.terms\_dot), 566  
 BDFMeshIO (class in sfepy.fem.meshio), 383  
 bf (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 bfg (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 BiotETHTerm (class in sfepy.terms.termsBiot), 536  
 BiotStressTerm (class in sfepy.terms.termsBiot), 536  
 BiotTerm (class in sfepy.terms.termsBiot), 538  
 BiotTHTerm (class in sfepy.terms.termsBiot), 537  
 blockgen (module), 320  
 bound (class in sfepy.mesh.meshutils), 461  
 build\_helpers (module), 317  
 build\_mlab\_pipeline() (sfepy.postprocess.viewer.Viewer method), 477  
 build\_op\_pi() (in module sfepy.homogenization.utils), 435  
 bulk\_from\_lame() (in module sfepy.mechanics.matcoefs), 447  
 bulk\_from\_youngpoisson() (in module sfepy.mechanics.matcoefs), 447  
 BulkActiveTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 573  
 BulkPenaltyTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 573  
 BulkPenaltyULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 577  
 BulkPressureTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 573  
 BulkPressureULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 578
- ## C
- cache (sfepy.fem.probes.Probe attribute), 392  
 call() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 326  
 call() (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp method), 422  
 call() (sfepy.homogenization.engine.HomogenizationEngine method), 432  
 call() (sfepy.homogenization.homogen\_app.HomogenizationApp method), 432  
 call() (sfepy.physics.schroedinger\_app.SchroedingerApp method), 470  
 call\_basic() (sfepy.applications.application.Application method), 325  
 call\_empty() (sfepy.postprocess.viewer.Viewer method), 477  
 call\_function() (sfepy.terms.terms.Term method), 514  
 call\_get\_fargs() (sfepy.terms.terms.Term method), 514  
 call\_mlab() (sfepy.postprocess.viewer.Viewer method), 477  
 call\_msg (sfepy.fem.meshio.MeshIO attribute), 385  
 call\_parametrized() (sfepy.applications.application.Application method), 325  
 canonize\_dof\_names() (sfepy.fem.conditions.Condition method), 343  
 canonize\_dof\_names() (sfepy.fem.conditions.Conditions method), 343  
 canonize\_dof\_names() (sfepy.fem.conditions.PeriodicBC method), 345  
 CauchyStrainSTerm (class in sfepy.terms.termsLinElasticity), 544  
 CauchyStrainTerm (class in sfepy.terms.termsLinElasticity), 544  
 CauchyStressETHTerm (class in sfepy.terms.termsLinElasticity), 545  
 CauchyStressTerm (class in sfepy.terms.termsLinElasticity), 546  
 CauchyStressTHTerm (class in sfepy.terms.termsLinElasticity), 546

- `cg_eigs()` (in module `sfepy.linalg.eigen`), 437
- `ch` (`sfepy.fem.meshio.HDF5MeshIO` attribute), 383
- `change_shape()` (`sfepy.mesh.splinbox.SplineBox` method), 465
- `CharacteristicFunction` (class in `sfepy.terms.terms`), 513
- `check()` (in module `sfepy.mesh.meshutils`), 462
- `check_args()` (`sfepy.terms.terms.Term` method), 514
- `check_custom_sensitivity()` (`sfepy.optimize.shapeOptim.ShapeOptimFlowCase` method), 467
- `check_finiteness()` (in module `sfepy.terms.utils`), 583
- `check_gradient()` (in module `sfepy.solvers.optimize`), 488
- `check_names()` (in module `sfepy.base.base`), 329
- `check_output()` (in module `test_install`), 320
- `check_sensitivity()` (`sfepy.optimize.shapeOptim.ShapeOptimFlowCase` method), 467
- `check_shapes()` (`sfepy.terms.terms.Term` method), 514
- `check_shapes()` (`sfepy.terms.terms_dot.BCNewtonTerm` method), 566
- `check_shapes()` (`sfepy.terms.terms_dot.DotProductVolumeTerm` method), 567
- `check_shapes()` (`sfepy.terms.terms_dot.VectorDotGradScalarTerm` method), 570
- `check_shapes()` (`sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTerm` method), 576
- `check_shapes()` (`sfepy.terms.termsLinElasticity.LinearElasticityTerm` method), 548
- `check_shapes()` (`sfepy.terms.termsLinElasticity.LinearElasticityTerm` method), 549
- `check_shapes()` (`sfepy.terms.termsLinElasticity.LinearPrestressTerm` method), 550
- `check_shapes()` (`sfepy.terms.termsLinElasticity.LinearStrainEnergyTerm` method), 551
- `check_shapes()` (`sfepy.terms.termsPoint.ConcentratedPointLoadTerm` method), 560
- `check_tangent_matrix()` (in module `sfepy.solvers.nls`), 487
- `check_vector_size()` (`sfepy.fem.variables.Variables` method), 416
- `ChristoffelAcousticTensor` (class in `sfepy.homogenization.coefs_phononic`), 428
- `CircleProbe` (class in `sfepy.fem.probes`), 391
- `classify_args()` (`sfepy.terms.terms.Term` method), 514
- `Clean` (class in `build_helpers`), 318
- `clean()` (`sfepy.mesh.meshutils.mesh` method), 462
- `clear_bases()` (`sfepy.fem.variables.FieldVariable` method), 410
- `clear_current_group()` (`sfepy.fem.variables.FieldVariable` method), 410
- `clear_dof_conns()` (`sfepy.fem.fields_base.Field` method), 362
- `clear_equations()` (`sfepy.fem.problemDef.ProblemDefinition` method), 394
- `clear_evaluate_cache()` (`sfepy.fem.variables.FieldVariable` method), 410
- `clear_mappings()` (`sfepy.fem.fields_base.Field` method), 362
- `clear_qp_base()` (`sfepy.fem.fea.Approximation` method), 360
- `clear_surface_groups()` (`sfepy.fem.domain.Domain` method), 349
- `CloseNodesIterator` (class in `sfepy.fem.variables`), 410
- `CMapping` (class in `sfepy.fem.extmods.mappings`), 421
- `CoefDim` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefDimDim` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefDimSym` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefDummy` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefEval` (class in `sfepy.homogenization.coefs_base`), 424
- `Coefficients` (class in `sfepy.homogenization.coefficients`), 423
- `CoefMOne` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefMISym` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefMISymSym` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefNN` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefNone` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefOne` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefRegion` (class in `sfepy.homogenization.coefs_perfusion`), 427
- `CoefSum` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefSym` (class in `sfepy.homogenization.coefs_base`), 424
- `CoefSymSym` (class in `sfepy.homogenization.coefs_base`), 424
- `collect_conn_info()` (`sfepy.fem.equations.Equation` method), 350
- `collect_conn_info()` (`sfepy.fem.equations.Equations` method), 351
- `collect_materials()` (`sfepy.fem.equations.Equation` method), 350
- `collect_materials()` (`sfepy.fem.equations.Equations` method), 351
- `collect_term()` (in module `sfepy.fem.parseEq`), 388
- `collect_variables()` (`sfepy.fem.equations.Equation` method), 350
- `collect_variables()` (`sfepy.fem.equations.Equations` method), 351
- `combine()` (in module `sfepy.linalg.utils`), 442



combine\_scalar\_grad() (in module sfepy.homogenization.recovery), 433  
 compare\_vectors() (sfepy.base.testing.TestCommon static method), 343  
 compile\_flags() (sfepy.config.Config method), 325  
 complete\_description() (sfepy.fem.region.Region method), 403  
 compose\_sparse() (in module sfepy.linalg.sparse), 440  
 CompoundPotential (class in sfepy.physics.potentials), 468  
 CompressibilityULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 578  
 compute\_cat\_dim\_dim() (in module sfepy.homogenization.coefs\_phononic), 429  
 compute\_cat\_dim\_sym() (in module sfepy.homogenization.coefs\_phononic), 430  
 compute\_cat\_sym\_sym() (in module sfepy.homogenization.coefs\_phononic), 430  
 compute\_correctors() (sfepy.homogenization.coefs\_base.TCConfig static method), 426  
 compute\_data() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTerm method), 574  
 compute\_data() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureTerm method), 578  
 compute\_eigenmomenta() (in module sfepy.homogenization.coefs\_phononic), 430  
 compute\_family\_data() (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLBase method), 575  
 compute\_family\_data() (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm method), 576  
 compute\_family\_data() (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULBase method), 579  
 compute\_jacobian() (sfepy.solvers.semismooth\_newton.SemismoothNewton method), 490  
 compute\_mac\_stress\_part() (in module sfepy.homogenization.recovery), 433  
 compute\_mean\_decay() (in module sfepy.homogenization.convolution), 432  
 compute\_micro\_u() (in module sfepy.homogenization.recovery), 433  
 compute\_nodal\_edge\_dirs() (in module sfepy.fem.utils), 409  
 compute\_nodal\_normals() (in module sfepy.fem.utils), 409  
 compute\_p\_corr\_steady() (in module sfepy.homogenization.recovery), 433  
 compute\_p\_corr\_time() (in module sfepy.homogenization.recovery), 433  
 compute\_p\_from\_macro() (in module sfepy.homogenization.recovery), 433  
 compute\_requirements() (sfepy.homogenization.engine.HomogenizationEngine method), 432  
 compute\_stress() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase method), 572  
 compute\_stress\_strain\_u() (in module sfepy.homogenization.recovery), 434  
 compute\_tan\_mod() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase method), 572  
 compute\_u\_corr\_steady() (in module sfepy.homogenization.recovery), 434  
 compute\_u\_corr\_time() (in module sfepy.homogenization.recovery), 434  
 compute\_u\_from\_macro() (in module sfepy.homogenization.recovery), 434  
 compute\_grad() (sfepy.mesh.meshutils.mesh method), 462  
 computenorm() (sfepy.mesh.meshutils.mesh method), 462  
 ComsolMeshIO (class in sfepy.fem.meshio), 383  
 ConcentratedPointLoadTerm (class in sfepy.terms.termsPoint), 560  
 Condition (class in sfepy.fem.conditions), 343  
 Conditions (class in sfepy.fem.conditions), 343  
 Config (sfepy.config.Config), 325  
 config (module), 320  
 console\_output() (in module sfepy.base.base), 329  
 ConnInfo (class in sfepy.terms.terms), 513  
 ConstantFunction (class in sfepy.fem.functions), 370  
 ConstantFunctionByRegion (class in sfepy.fem.functions), 370  
 ContactPlane (class in sfepy.mechanics.contact\_planes), 451  
 ContactPlaneTerm (class in sfepy.terms.termsSurface), 560  
 Container (class in sfepy.base.base), 326  
 create\_mesh() (sfepy.fem.region.Region method), 403  
 conv() (in module sfepy.mesh.femlab), 454  
 conv2() (in module sfepy.mesh.femlab), 454  
 conv3() (in module sfepy.mesh.femlab), 454  
 conv4() (in module sfepy.mesh.femlab), 454  
 conv\_test() (in module sfepy.solvers.nls), 487  
 conv\_test() (in module sfepy.solvers.optimize), 488  
 ConvectTerm (class in sfepy.terms.termsNavierStokes), 552  
 convert\_complex\_output() (in module sfepy.fem.meshio), 388  
 convert\_el\_to\_nodes() (sfepy.mesh.meshutils.mesh method), 462  
 convert\_mesh (module), 321  
 convert\_stress\_to\_nodes() (sfepy.mesh.meshutils.mesh method), 462  
 ConvolutionKernel (class in sfepy.homogenization.convolution), 431  
 convolve\_field\_scalar() (in module sfepy.homogenization.recovery), 434  
 convolve\_field\_sym\_tensor() (in module sfepy.homogenization.recovery), 434  
 create\_mesh() (in module sfepy.homogenization.utils), 435

copy() (sfepy.base.base.Struct method), 328  
 copy() (sfepy.fem.mesh.Mesh method), 380  
 copy() (sfepy.fem.problemDef.ProblemDefinition method), 394  
 copy() (sfepy.fem.region.Region method), 403  
 copy() (sfepy.fem.state.State method), 407  
 CopyData (class in sfepy.homogenization.coefs\_base), 424  
 CorrDim (class in sfepy.homogenization.coefs\_base), 424  
 CorrDimDim (class in sfepy.homogenization.coefs\_base), 424  
 CorrectorsPermeability (class in sfepy.homogenization.coefs\_elastic), 426  
 CorrEqPar (class in sfepy.homogenization.coefs\_base), 424  
 CorrMiniApp (class in sfepy.homogenization.coefs\_base), 424  
 CorrN (class in sfepy.homogenization.coefs\_base), 425  
 CorrNN (class in sfepy.homogenization.coefs\_base), 425  
 CorrOne (class in sfepy.homogenization.coefs\_base), 425  
 CorrRegion (class in sfepy.homogenization.coefs\_perfusion), 427  
 CorrSetBCS (class in sfepy.homogenization.coefs\_base), 425  
 CorrSolution (class in sfepy.homogenization.coefs\_base), 425  
 count (sfepy.base.log.Log attribute), 338  
 cprint() (sfepy.fem.extmods.mappings.CMapping method), 421  
 create\_adof\_conn() (in module sfepy.fem.variables), 418  
 create\_arg\_parser() (in module sfepy.terms.terms), 517  
 create\_bnf() (in module sfepy.base.parse\_conf), 340  
 create\_bnf() (in module sfepy.fem.parseEq), 388  
 create\_bnf() (in module sfepy.fem.parseReg), 388  
 create\_bqp() (sfepy.fem.fea.Approximation method), 360  
 create\_conn\_graph() (sfepy.fem.mesh.Mesh method), 380  
 create\_dataset() (sfepy.postprocess.sources.GenericFileSource method), 474  
 create\_dof\_conn() (in module sfepy.fem.fields\_base), 365  
 create\_evaluable() (in module sfepy.fem.evaluate), 355  
 create\_evaluable() (sfepy.fem.problemDef.ProblemDefinition method), 394  
 create\_evaluables() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase method), 465  
 create\_evaluables() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase method), 467  
 create\_expression\_output() (in module sfepy.fem.fields\_base), 366  
 create\_file\_source() (in module sfepy.postprocess.sources), 475  
 create\_friction\_bcs() (sfepy.mechanics.friction.DualMesh method), 445  
 create\_lcbc\_operators() (sfepy.fem.variables.FieldVariable method), 410  
 create\_mapping() (in module sfepy.mechanics.membranes), 448  
 create\_mapping() (sfepy.fem.fields\_base.Field method), 362  
 create\_mapping() (sfepy.fem.region.Region method), 403  
 create\_mass\_matrix() (in module sfepy.fem.projections), 400  
 create\_materials() (sfepy.fem.problemDef.ProblemDefinition method), 396  
 create\_matrix\_graph() (sfepy.fem.equations.Equations method), 351  
 create\_mesh() (sfepy.fem.fields\_base.Field method), 362  
 create\_mesh\_from\_control\_points() (sfepy.optimize.freeFormDef.SplineBoxes method), 466  
 create\_mesh\_graph() (in module sfepy.fem.extmods.mesh), 421  
 create\_output() (in module sfepy.fem.linearizer), 372  
 create\_output() (sfepy.fem.fields\_base.Field method), 362  
 create\_output() (sfepy.fem.variables.FieldVariable method), 410  
 create\_output\_dict() (sfepy.fem.state.State method), 407  
 create\_parser() (in module gen\_term\_table), 324  
 create\_pis() (in module sfepy.homogenization.utils), 435  
 create\_problem() (in module extractor), 314  
 create\_region() (sfepy.fem.domain.Domain method), 349  
 create\_regions() (sfepy.fem.domain.Domain method), 349  
 create\_scalar() (in module evalForms), 322  
 create\_scalar\_base() (in module evalForms), 322  
 create\_scalar\_base\_grad() (in module evalForms), 322  
 create\_scalar\_pis() (in module sfepy.homogenization.utils), 435  
 create\_scalar\_var\_data() (in module evalForms), 322  
 create\_source() (sfepy.postprocess.sources.GenericFileSource method), 474  
 create\_source() (sfepy.postprocess.sources.GenericSequenceFileSource method), 474  
 create\_source() (sfepy.postprocess.sources.VTKFileSource method), 474  
 create\_source() (sfepy.postprocess.sources.VTKSequenceFileSource method), 475  
 create\_spb() (sfepy.mesh.splinebox.SplineBox static method), 465  
 create\_state() (sfepy.fem.problemDef.ProblemDefinition method), 396  
 create\_state\_vector() (sfepy.fem.equations.Equations method), 351  
 create\_state\_vector() (sfepy.fem.variables.Variables method), 416  
 create\_stripped\_state\_vector() (sfepy.fem.equations.Equations method), 351  
 create\_stripped\_state\_vector()

- (sfepy.fem.variables.Variables method), 416
- create\_surface\_facet() (sfepy.fem.geometry\_element.GeometryElement method), 370
- create\_surface\_group() (sfepy.fem.domain.Domain method), 349
- create\_transformation\_matrix() (in module sfepy.mechanics.membranes), 448
- create\_u\_operator() (in module evalForms), 322
- create\_variables() (sfepy.fem.problemDef.ProblemDefinition method), 396
- create\_vector() (in module evalForms), 322
- create\_vector\_base() (in module evalForms), 322
- create\_vector\_base\_grad() (in module evalForms), 322
- create\_vector\_var\_data() (in module evalForms), 322
- crossproduct() (in module sfepy.mesh.femlab), 454
- curve2() (in module sfepy.mesh.femlab), 454
- cut\_freq\_range() (in module sfepy.homogenization.coefs\_phononic), 430
- cvt\_array\_index() (in module sfepy.base.parse\_conf), 340
- cvt\_bool() (in module sfepy.base.parse\_conf), 340
- cvt\_int() (in module sfepy.base.parse\_conf), 340
- cvt\_none() (in module sfepy.base.parse\_conf), 340
- cvt\_real() (in module sfepy.base.parse\_conf), 340
- cw2us() (in module edit\_identifiers), 321
- cycle() (in module sfepy.linalg.utils), 442
- cylindergen (module), 321
- ## D
- d\_biot\_div() (in module sfepy.terms.extmods.terms), 583
- d\_diffusion() (in module sfepy.terms.extmods.terms), 583
- d\_diffusion\_sa() (in module sfepy.terms.extmods.terms), 583
- d\_div\_grad() (sfepy.terms.termsNavierStokes.DivGradTerm method), 553
- d\_dot() (sfepy.terms.terms\_dot.DotProductVolumeTerm static method), 567
- d\_eval() (sfepy.terms.termsNavierStokes.StokesTerm static method), 559
- d\_fun() (sfepy.terms.termsLaplace.DiffusionCoupling static method), 540
- d\_fun() (sfepy.terms.termsSurface.SurfaceNormalDotTerm static method), 563
- d\_laplace() (in module sfepy.terms.extmods.terms), 583
- d\_lin\_elastic() (in module sfepy.terms.extmods.terms), 583
- d\_lin\_prestress() (sfepy.terms.termsLinElasticity.LinearPrestressTerm method), 550
- d\_of\_nsMinGrad() (in module sfepy.terms.extmods.terms), 583
- d\_of\_nsSurfMinDPress() (in module sfepy.terms.extmods.terms), 583
- d\_piezo\_coupling() (in module sfepy.terms.extmods.terms), 583
- d\_sd\_convect() (in module sfepy.terms.extmods.terms), 583
- d\_sd\_div() (in module sfepy.terms.extmods.terms), 583
- d\_sd\_div\_grad() (in module sfepy.terms.extmods.terms), 583
- d\_sd\_st\_grad\_div() (in module sfepy.terms.extmods.terms), 584
- d\_sd\_st\_pspg\_c() (in module sfepy.terms.extmods.terms), 584
- d\_sd\_st\_pspg\_p() (in module sfepy.terms.extmods.terms), 584
- d\_sd\_st\_supg\_c() (in module sfepy.terms.extmods.terms), 584
- d\_sd\_volume\_dot() (in module sfepy.terms.extmods.terms), 584
- d\_surf\_laplace() (in module sfepy.terms.extmods.terms), 584
- d\_surf\_lcouple() (in module sfepy.terms.extmods.terms), 584
- d\_surface\_flux() (in module sfepy.terms.extmods.terms), 584
- d\_volume\_surface() (in module sfepy.terms.extmods.terms), 584
- DatasetManager (class in sfepy.postprocess.dataset\_manager), 470
- de\_cauchy\_strain() (in module sfepy.terms.extmods.terms), 584
- de\_cauchy\_stress() (in module sfepy.terms.extmods.terms), 584
- de\_he\_rtm() (in module sfepy.terms.extmods.terms), 584
- debug() (in module sfepy.base.base), 326, 329
- debug\_flags() (sfepy.config.Config method), 325
- default (sfepy.base.goptions.ValidatedDict attribute), 336
- define\_box\_regions() (in module sfepy.homogenization.utils), 435
- DeformationGradientTerm (class in sfepy.terms.terms\_hyperelastic\_base), 571
- delete\_groups() (sfepy.fem.region.Region method), 404
- delete\_zero\_faces() (sfepy.fem.region.Region method), 404
- DensityVolumeInfo (class in sfepy.homogenization.coefs\_phononic), 428
- derivatives() (sfepy.physics.radial\_mesh.RadialVector method), 469
- describe() (sfepy.fem.extmods.mappings.CMapping method), 421
- describe\_deformation() (in module sfepy.mechanics.membranes), 449
- describe\_dual\_surface() (sfepy.mechanics.friction.DualMesh method), 445
- describe\_gaps() (in module sfepy.homogenization.coefs\_phononic), 430
- describe\_geometry() (in module sfepy.mechanics.membranes), 449



- describe\_geometry() (sfepy.fem.fea.Approximation method), 360
- describe\_nodes() (sfepy.fem.fea.Interpolant method), 361
- describe\_nodes() (sfepy.fem.poly\_spaces.PolySpace method), 390
- description (build\_helpers.DoxygenDocs attribute), 318
- description (build\_helpers.SphinxHTMLDocs attribute), 318
- description (build\_helpers.SphinxPDFDocs attribute), 318
- DesignVariables (class in sfepy.optimize.freeFormDef), 466
- det (sfepy.fem.extmods.mappings.CMapping attribute), 421
- det() (sfepy.mesh.meshutils.mesh method), 462
- detect\_band\_gaps() (in module sfepy.homogenization.coefs\_phononic), 430
- di\_surface\_moment() (in module sfepy.terms.extmods.terms), 584
- dict\_extend() (in module sfepy.base.base), 329
- dict\_from\_keys\_init() (in module sfepy.base.base), 329
- dict\_from\_options() (in module sfepy.base.conf), 335
- dict\_from\_string() (in module sfepy.base.conf), 335
- dict\_to\_array() (in module sfepy.base.base), 329
- dict\_to\_struct() (in module sfepy.base.base), 329
- diff\_dt() (sfepy.homogenization.convolution.ConvolutionKernel method), 431
- DiffusionCoupling (class in sfepy.terms.termsLaplace), 539
- DiffusionRTerm (class in sfepy.terms.termsLaplace), 540
- DiffusionSATerm (class in sfepy.terms.termsAcoustic), 518
- DiffusionTerm (class in sfepy.terms.termsLaplace), 540
- DiffusionTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 574
- DiffusionVelocityTerm (class in sfepy.terms.termsLaplace), 541
- dim (sfepy.fem.extmods.mappings.CMapping attribute), 421
- dim2sym() (in module sfepy.mechanics.tensors), 450
- DiscontinuousApproximation (class in sfepy.fem.fea), 361
- dist() (sfepy.mesh.meshutils.mesh method), 462
- dist2() (sfepy.mesh.meshutils.mesh method), 462
- DivGradTerm (class in sfepy.terms.termsNavierStokes), 552
- DivOperatorTerm (class in sfepy.terms.termsNavierStokes), 553
- DivTerm (class in sfepy.terms.termsNavierStokes), 553
- DofInfo (class in sfepy.fem.dof\_info), 345
- Domain (class in sfepy.fem.domain), 349
- DomainSpecificPlot (class in sfepy.postprocess.domain\_specific), 471
- dot() (sfepy.physics.radial\_mesh.RadialMesh method), 468
- dot\_sequences() (in module sfepy.linalg.utils), 442
- DotProductSurfaceTerm (class in sfepy.terms.terms\_dot), 566
- DotProductVolumeTerm (class in sfepy.terms.terms\_dot), 567
- DotSPProductVolumeOperatorWETHTerm (class in sfepy.terms.terms\_dot), 568
- DotSPProductVolumeOperatorWTHTerm (class in sfepy.terms.terms\_dot), 568
- DoxygenDocs (class in build\_helpers), 318
- dq\_cauchy\_strain() (in module sfepy.terms.extmods.terms), 584
- dq\_def\_grad() (in module sfepy.terms.extmods.terms), 584
- dq\_div\_vector() (in module sfepy.terms.extmods.terms), 584
- dq\_finite\_strain\_tl() (in module sfepy.terms.extmods.terms), 584
- dq\_finite\_strain\_ul() (in module sfepy.terms.extmods.terms), 584
- dq\_grad() (in module sfepy.terms.extmods.terms), 584
- dq\_state\_in\_qp() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_finite\_strain\_surface() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_stress\_bulk() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_stress\_bulk\_active() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_stress\_mooney\_rivlin() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_stress\_neohook() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_tan\_mod\_bulk() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_tan\_mod\_bulk\_active() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_tan\_mod\_mooney\_rivlin() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_he\_tan\_mod\_neohook() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_stress\_bulk\_pressure() (in module sfepy.terms.extmods.terms), 584
- dq\_tl\_tan\_mod\_bulk\_pressure\_u() (in module sfepy.terms.extmods.terms), 584
- dq\_ul\_he\_stress\_bulk() (in module sfepy.terms.extmods.terms), 584
- dq\_ul\_he\_stress\_mooney\_rivlin() (in module sfepy.terms.extmods.terms), 584
- dq\_ul\_he\_stress\_neohook() (in module sfepy.terms.extmods.terms), 584
- dq\_ul\_he\_tan\_mod\_bulk() (in module sfepy.terms.extmods.terms), 584

`dq_ul_he_tan_mod_mooney_rivlin()` (in module `sfepy.terms.extmods.terms`), 584

`dq_ul_he_tan_mod_neohook()` (in module `sfepy.terms.extmods.terms`), 585

`dq_ul_stress_bulk_pressure()` (in module `sfepy.terms.extmods.terms`), 585

`dq_ul_tan_mod_bulk_pressure_u()` (in module `sfepy.terms.extmods.terms`), 585

`draw_arrow()` (in module `sfepy.postprocess.plot_facets`), 473

`DualMesh` (class in `sfepy.mechanics.friction`), 445

`dump_to_vtk()` (in module `sfepy.postprocess.time_history`), 475

`dvelocity()` (`sfepy.mesh.splinbox.SplineBox` method), 465

`dw_adj_convect1()` (in module `sfepy.terms.extmods.terms`), 585

`dw_adj_convect2()` (in module `sfepy.terms.extmods.terms`), 585

`dw_biot_div()` (in module `sfepy.terms.extmods.terms`), 585

`dw_biot_grad()` (in module `sfepy.terms.extmods.terms`), 585

`dw_diffusion()` (in module `sfepy.terms.extmods.terms`), 585

`dw_div()` (in module `sfepy.terms.extmods.terms`), 585

`dw_dot()` (`sfepy.terms.terms_dot.DotProductVolumeTerm` static method), 567

`dw_electric_source()` (in module `sfepy.terms.extmods.terms`), 585

`dw_fun()` (`sfepy.terms.terms_dot.ScalarDotGradIScalarTerm` static method), 569

`dw_fun()` (`sfepy.terms.termsLaplace.DiffusionCoupling` static method), 540

`dw_fun()` (`sfepy.terms.termsSurface.SufaceNormalDotTerm` static method), 563

`dw_grad()` (in module `sfepy.terms.extmods.terms`), 585

`dw_he_rtm()` (in module `sfepy.terms.extmods.terms`), 585

`dw_laplace()` (in module `sfepy.terms.extmods.terms`), 585

`dw_lin_convect()` (in module `sfepy.terms.extmods.terms`), 585

`dw_lin_elastic()` (in module `sfepy.terms.extmods.terms`), 585

`dw_lin_elastic_iso()` (in module `sfepy.terms.extmods.terms`), 585

`dw_lin_prestress()` (in module `sfepy.terms.extmods.terms`), 585

`dw_lin_strain_fib()` (in module `sfepy.terms.extmods.terms`), 585

`dw_permeability_r()` (in module `sfepy.terms.extmods.terms`), 585

`dw_piezo_coupling()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_adj1_supg_p()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_adj2_supg_p()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_adj_supg_c()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_grad_div()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_pspg_c()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_supg_c()` (in module `sfepy.terms.extmods.terms`), 585

`dw_st_supg_p()` (in module `sfepy.terms.extmods.terms`), 585

`dw_surf_laplace()` (in module `sfepy.terms.extmods.terms`), 585

`dw_surf_lcouple()` (in module `sfepy.terms.extmods.terms`), 585

`dw_surface_ltr()` (in module `sfepy.terms.extmods.terms`), 585

`dw_surface_s_v_dot_n()` (in module `sfepy.terms.extmods.terms`), 585

`dw_surface_v_dot_n_s()` (in module `sfepy.terms.extmods.terms`), 585

`dw_tl_diffusion()` (in module `sfepy.terms.extmods.terms`), 585

`dw_tl_surface_traction()` (in module `sfepy.terms.extmods.terms`), 585

`dw_tl_volume()` (in module `sfepy.terms.extmods.terms`), 585

`dw_ul_volume()` (in module `sfepy.terms.extmods.terms`), 585

`dw_v_dot_grad_s_sw()` (in module `sfepy.terms.extmods.terms`), 586

`dw_v_dot_grad_s_vw()` (in module `sfepy.terms.extmods.terms`), 586

`dw_volume_dot_scalar()` (in module `sfepy.terms.extmods.terms`), 586

`dw_volume_dot_vector()` (in module `sfepy.terms.extmods.terms`), 586

`dw_volume_lvf()` (in module `sfepy.terms.extmods.terms`), 586

## E

`edge_data_to_output()` (in module `sfepy.mechanics.friction`), 446

`EdgeDirectionOperator` (class in `sfepy.fem.dof_info`), 346

`edit()` (in module `edit_identifiers`), 321

`edit()` (`sfepy.base.conf.ProblemConf` method), 334

`edit_dict_strings()` (in module `sfepy.base.base`), 329

`edit_filename()` (in module `sfepy.base.ioutils`), 337

`edit_identifiers` (module), 321

`edit_tuple_strings()` (in module `sfepy.base.base`), 329

`eig()` (in module `sfepy.solvers.eigen`), 481

- Eigenmomenta (class in sfepy.homogenization.coefs\_phononic), 429
- EigenvalueSolver (class in sfepy.solvers.solvers), 490
- ElasticConstants (class in sfepy.mechanics.matcoefs), 446
- ElectricSourceTerm (class in sfepy.terms.termsElectric), 539
- elems\_q2t() (in module sfepy.mesh.mesh\_tools), 460
- encode\_animation() (sfepy.postprocess.viewer.Viewer method), 479
- ensure\_path() (in module sfepy.base.ioutils), 337
- Equation (class in sfepy.fem.equations), 350
- equation\_mapping() (sfepy.fem.variables.FieldVariable method), 411
- equation\_mapping() (sfepy.fem.variables.Variables method), 416
- EquationMap (class in sfepy.fem.dof\_info), 346
- Equations (class in sfepy.fem.equations), 351
- errclear() (in module sfepy.terms.extmods.terms), 586
- error() (in module sfepy.mesh.meshutils), 462
- EssentialBC (class in sfepy.fem.conditions), 344
- ETA (class in sfepy.base.progressbar), 340
- ETHTerm (class in sfepy.terms.terms\_th), 583
- eval\_base() (sfepy.fem.poly\_spaces.PolySpace method), 390
- eval\_boundary\_diff\_vel\_grad() (in module sfepy.homogenization.coefs\_elastic), 427
- eval\_complex() (in module sfepy.fem.evaluate\_variable), 358
- eval\_complex() (sfepy.terms.terms.Term method), 514
- eval\_coor\_expression() (sfepy.base.testing.TestCommon static method), 343
- eval\_equations() (in module sfepy.fem.evaluate), 356
- eval\_exponential() (in module sfepy.homogenization.convolution), 432
- eval\_extra\_coor() (sfepy.fem.fea.Approximation method), 360
- eval\_extra\_coor() (sfepy.fem.fea.DiscontinuousApproximation method), 361
- eval\_function() (sfepy.terms.terms\_membrane.TLMembraneTerm static method), 581
- eval\_in\_els\_and\_qp() (in module sfepy.fem.evaluate), 357
- eval\_ion\_ion\_energy() (in module sfepy.physics.energy), 467
- eval\_lagrange\_simplex() (in module sfepy.fem.extmods.bases), 418
- eval\_lagrange\_tensor\_product() (in module sfepy.fem.extmods.bases), 419
- eval\_lobatto1d() (in module sfepy.fem.extmods.lobatto\_bases), 420
- eval\_lobatto\_tensor\_product() (in module sfepy.fem.extmods.lobatto\_bases), 420
- eval\_membrane\_mooney\_rivlin() (in module sfepy.terms.terms\_membrane), 581
- eval\_nodal\_coors() (in module sfepy.fem.fea), 361
- eval\_non\_local\_interaction() (in module sfepy.physics.energy), 467
- eval\_real() (in module sfepy.fem.evaluate\_variable), 358
- eval\_real() (sfepy.terms.terms.Term method), 514
- eval\_real() (sfepy.terms.terms\_th.THTerm method), 583
- eval\_residual() (sfepy.fem.evaluate.BasicEvaluator method), 355
- eval\_residual() (sfepy.fem.evaluate.LCBCEvaluator method), 355
- eval\_residuals() (sfepy.fem.equations.Equations method), 351
- eval\_tangent\_matrices() (sfepy.fem.equations.Equations method), 352
- eval\_tangent\_matrix() (sfepy.fem.evaluate.BasicEvaluator method), 355
- eval\_tangent\_matrix() (sfepy.fem.evaluate.LCBCEvaluator method), 355
- evalForms (module), 322
- evaluate() (sfepy.fem.equations.Equation method), 351
- evaluate() (sfepy.fem.equations.Equations method), 352
- evaluate() (sfepy.fem.problemDef.ProblemDefinition method), 396
- evaluate() (sfepy.fem.variables.FieldVariable method), 411
- evaluate() (sfepy.homogenization.coefs\_phononic.AcousticMassTensor method), 427
- evaluate() (sfepy.homogenization.coefs\_phononic.AppliedLoadTensor method), 428
- evaluate() (sfepy.mesh.splinebox.SplineBox method), 466
- evaluate() (sfepy.terms.terms.Term method), 514
- evaluate() (sfepy.terms.terms\_new.NewTerm method), 583
- evaluate\_at() (sfepy.fem.fields\_hierarchic.H1HierarchicVolumeField method), 367
- evaluate\_at() (sfepy.fem.fields\_nodal.H1NodalMixin method), 368
- evaluate\_at() (sfepy.fem.variables.FieldVariable method), 412
- evaluate\_bfbgm() (sfepy.fem.extmods.mappings.CMapping method), 421
- evaluate\_in\_rc() (in module sfepy.fem.extmods.bases), 419
- Evaluator (class in sfepy.fem.evaluate), 355
- expand\_nodes\_to\_dofs() (in module sfepy.fem.dof\_info), 348
- expand\_nodes\_to\_equations() (in module sfepy.fem.dof\_info), 348
- ExplicitRadialMesh (class in sfepy.physics.radial\_mesh), 468
- ExplicitTimeSteppingSolver (class in sfepy.solvers.ts\_solvers), 492

- explode\_groups() (sfepy.fem.mesh.Mesh method), 380  
 extend() (sfepy.base.base.Container method), 327  
 extend\_cell\_data() (in module sfepy.fem.utils), 409  
 extend\_dofs() (sfepy.fem.fields\_base.Field method), 363  
 extend\_dofs() (sfepy.fem.fields\_nodal.H1DiscontinuousField method), 368  
 extract\_time\_history() (in module sfepy.postprocess.time\_history), 475  
 extract\_times() (in module sfepy.postprocess.time\_history), 475  
 extractor (module), 314  
 extrapolated\_derivatives() (sfepy.physics.radial\_mesh.RadialVector method), 469  
 extrapolated\_values() (sfepy.physics.radial\_mesh.RadialVector method), 469
- ## F
- Facets (class in sfepy.fem.facets), 358  
 factorial() (in module sfepy.fem.simplex\_cubature), 406  
 family\_data\_names (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 571  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm attribute), 573  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm attribute), 573  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 574  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 575  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm attribute), 575  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm attribute), 576  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 576  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 577  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm attribute), 578  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 578  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 579  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm attribute), 579  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm attribute), 580  
 family\_data\_names (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 580  
 family\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 574  
 family\_function() (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLBase method), 575  
 family\_function() (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm method), 576  
 family\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 578  
 family\_function() (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULBase method), 579  
 family\_name (sfepy.fem.fields\_hierarchic.H1HierarchicVolumeField attribute), 368  
 family\_name (sfepy.fem.fields\_nodal.H1DiscontinuousField attribute), 368  
 family\_name (sfepy.fem.fields\_nodal.H1NodalSurfaceField attribute), 369  
 family\_name (sfepy.fem.fields\_nodal.H1NodalVolumeField attribute), 369  
 fd\_cache\_name (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLBase attribute), 575  
 fd\_cache\_name (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULBase attribute), 579  
 femlabline() (in module sfepy.mesh.femlab), 454  
 femlabsurface() (in module sfepy.mesh.femlab), 454  
 femlabsurface3() (in module sfepy.mesh.femlab), 454  
 femlabsurface3\_old() (in module sfepy.mesh.femlab), 454  
 FESurface (class in sfepy.fem.fe\_surface), 360  
 fibre\_function() (in module sfepy.terms.terms\_fibres), 571  
 FibresActiveTLTerm (class in sfepy.terms.terms\_fibres), 570  
 Field (class in sfepy.fem.fields\_base), 362  
 fields\_from\_conf() (in module sfepy.fem.fields\_base), 367  
 FieldVariable (class in sfepy.fem.variables), 410  
 file\_changed() (sfepy.postprocess.sources.FileSource method), 473  
 file\_changed() (sfepy.postprocess.sources.GenericFileSource method), 474  
 FileSource (class in sfepy.postprocess.sources), 473  
 FileTransferSpeed (class in sfepy.base.progressbar), 341  
 fill() (sfepy.fem.state.State method), 408  
 finalize() (sfepy.fem.dof\_info.LCBCOperators method), 348  
 finalize\_options() (build\_helpers.NoOptionsDocs method), 318  
 find() (sfepy.base.base.OneTypeList method), 327  
 find\_group\_interfaces() (sfepy.fem.facets.Facets method), 358  
 find\_man() (in module sfepy.fem.mesh), 382  
 find\_ref\_coors() (in module sfepy.fem.extmods.bases), 420  
 find\_subclasses() (in module sfepy.base.base), 330  
 find\_zero() (in module sfepy.homogenization.coefs\_phononic), 430  
 find\_zero() (sfepy.mesh.meshutils.bound method), 461  
 findelements() (sfepy.mesh.meshutils.bound method), 461

- 461
- findSurf (module), 319
- finish() (sfepy.base.progressbar.ProgressBar method), 341
- fit\_exponential() (in module sfepy.homogenization.convolution), 432
- fix\_double\_nodes() (in module sfepy.fem.mesh), 382
- fix\_eig\_range() (sfepy.homogenization.coefs\_phononic.BandGaps method), 428
- fix\_element\_orientation() (sfepy.fem.domain.Domain method), 349
- fix\_path() (in module schroedinger), 317
- flag\_points\_in\_polygon2d() (in module sfepy.linalg.geometry), 438
- flat() (in module sfepy.mesh.meshutils), 462
- FMinSteepestDescent (class in sfepy.solvers.optimize), 487
- font\_size() (in module sfepy.base.plotutils), 340
- for\_format() (sfepy.fem.meshio.MeshIO static method), 385
- format (sfepy.fem.meshio.AbaqusMeshIO attribute), 383
- format (sfepy.fem.meshio.ANSYSCDBMeshIO attribute), 383
- format (sfepy.fem.meshio.AVSUCDMeshIO attribute), 383
- format (sfepy.fem.meshio.BDFMeshIO attribute), 383
- format (sfepy.fem.meshio.ComsolMeshIO attribute), 383
- format (sfepy.fem.meshio.HDF5MeshIO attribute), 384
- format (sfepy.fem.meshio.HypermeshAsciiMeshIO attribute), 384
- format (sfepy.fem.meshio.MeditMeshIO attribute), 384
- format (sfepy.fem.meshio.MEDMeshIO attribute), 384
- format (sfepy.fem.meshio.Mesh3DMeshIO attribute), 385
- format (sfepy.fem.meshio.MeshIO attribute), 386
- format (sfepy.fem.meshio.NEUMeshIO attribute), 386
- format (sfepy.fem.meshio.TetgenMeshIO attribute), 387
- format (sfepy.fem.meshio.UserMeshIO attribute), 387
- format (sfepy.fem.meshio.VTKMeshIO attribute), 387
- format\_next() (in module gen\_term\_table), 324
- format\_str() (sfepy.fem.meshio.BDFMeshIO static method), 383
- format\_time() (sfepy.base.progressbar.ETA method), 340
- formatpos() (in module sfepy.mesh.meshutils), 462
- formatpos2() (in module sfepy.mesh.meshutils), 462
- from\_args() (sfepy.fem.fields\_base.Field static method), 363
- from\_array() (sfepy.linalg.utils.MatrixAction static method), 442
- from\_conf() (sfepy.base.log.Log static method), 338
- from\_conf() (sfepy.fem.conditions.Conditions static method), 343
- from\_conf() (sfepy.fem.equations.Equations static method), 352
- from\_conf() (sfepy.fem.fields\_base.Field static method), 363
- from\_conf() (sfepy.fem.functions.Functions static method), 370
- from\_conf() (sfepy.fem.integrals.Integrals static method), 373
- from\_conf() (sfepy.fem.materials.Material static method), 375
- from\_conf() (sfepy.fem.materials.Materials static method), 377
- from\_conf() (sfepy.fem.problemDef.ProblemDefinition static method), 397
- from\_conf() (sfepy.fem.variables.Variable static method), 415
- from\_conf() (sfepy.fem.variables.Variables static method), 417
- from\_conf() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase static method), 467
- from\_conf() (sfepy.solvers.ts.TimeStepper static method), 491
- from\_conf() (sfepy.solvers.ts.VariableTimeStepper static method), 492
- from\_conf\_file() (sfepy.fem.problemDef.ProblemDefinition static method), 397
- from\_data() (sfepy.fem.mesh.Mesh static method), 380
- from\_desc() (sfepy.fem.equations.Equation static method), 351
- from\_desc() (sfepy.terms.terms.Term static method), 514
- from\_desc() (sfepy.terms.terms.Terms static method), 517
- from\_dict() (sfepy.base.conf.ProblemConf static method), 334
- from\_domain() (sfepy.fem.facets.Facets static method), 358
- from\_faces() (sfepy.fem.region.Region static method), 404
- from\_file() (sfepy.base.conf.ProblemConf static method), 334
- from\_file() (sfepy.fem.mesh.Mesh static method), 381
- from\_file() (sfepy.physics.radial\_mesh.RadialVector static method), 469
- from\_file\_and\_options() (sfepy.base.conf.ProblemConf static method), 335
- from\_file\_hdf5() (sfepy.fem.history.Histories static method), 371
- from\_file\_hdf5() (sfepy.homogenization.coefficients.Coefficients static method), 423
- from\_function() (sfepy.linalg.utils.MatrixAction static method), 442
- from\_gmsh\_file() (sfepy.mesh.geom\_tools.geometry static method), 455
- from\_module() (sfepy.base.conf.ProblemConf static method), 335
- from\_region() (sfepy.fem.mesh.Mesh static method), 381
- from\_sequence() (sfepy.fem.history.History static method), 371



from_surface() (sfepy.fem.mesh.Mesh static method), 381	function() (sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabilizationTerm static method), 526
from_table() (sfepy.fem.quadratures.QuadraturePoints static method), 402	function() (sfepy.terms.termsAdjointNavierStokes.SDPSPGPStabilizationTerm static method), 527
from_variables() (sfepy.fem.state.State static method), 408	function() (sfepy.terms.termsAdjointNavierStokes.SDSUPGCStabilizationTerm static method), 528
from_vertices() (sfepy.fem.region.Region static method), 404	function() (sfepy.terms.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm static method), 528
fromstr() (sfepy.mesh.meshutils.bound method), 461	function() (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj1StabilizationTerm static method), 529
Function (class in sfepy.fem.functions), 370	function() (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj2StabilizationTerm static method), 530
function() (sfepy.terms.terms_constraints.NonPenetrationTerm static method), 565	function() (sfepy.terms.termsBasic.IntegrateMatTerm static method), 531
function() (sfepy.terms.terms_dot.DotSProductVolumeOperatorTerm static method), 568	function() (sfepy.terms.termsBasic.IntegrateSurfaceTerm static method), 532
function() (sfepy.terms.terms_dot.DotSProductVolumeOperatorTerm static method), 569	function() (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm static method), 532
function() (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm static method), 572	function() (sfepy.terms.termsBasic.IntegrateVolumeTerm static method), 533
function() (sfepy.terms.terms_hyperelastic_base.HyperElasticTerm static method), 572	function() (sfepy.terms.termsBasic.SumNodalValuesTerm static method), 533
function() (sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm static method), 575	function() (sfepy.terms.termsBasic.SurfaceMomentTerm static method), 534
function() (sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm static method), 576	function() (sfepy.terms.termsBasic.VolumeSurfaceTerm static method), 535
function() (sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm static method), 577	function() (sfepy.terms.termsBasic.VolumeTerm static method), 535
function() (sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm static method), 579	function() (sfepy.terms.termsBiot.BiotStressTerm static method), 537
function() (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm static method), 580	function() (sfepy.terms.termsElectric.ElectricSourceTerm static method), 539
function() (sfepy.terms.terms_membrane.TLMembraneTerm static method), 581	function() (sfepy.terms.termsLaplace.DiffusionRTerm static method), 540
function() (sfepy.terms.termsAcoustic.DiffusionSATerm static method), 518	function() (sfepy.terms.termsLaplace.DiffusionVelocityTerm static method), 541
function() (sfepy.terms.termsAdjointNavierStokes.AdjConvectionTerm static method), 520	function() (sfepy.terms.termsLaplace.PermeabilityRTerm static method), 543
function() (sfepy.terms.termsAdjointNavierStokes.AdjConvectionTerm static method), 521	function() (sfepy.terms.termsLaplace.SurfaceFluxTerm static method), 543
function() (sfepy.terms.termsAdjointNavierStokes.AdjDivGradientTerm static method), 522	function() (sfepy.terms.termsLinElasticity.CauchyStrainTerm static method), 545
function() (sfepy.terms.termsAdjointNavierStokes.NSOFMifGradientTerm static method), 522	function() (sfepy.terms.termsLinElasticity.CauchyStressTerm static method), 547
function() (sfepy.terms.termsAdjointNavierStokes.NSOFSummationTerm static method), 523	function() (sfepy.terms.termsLinElasticity.LinearElasticETHTerm static method), 548
function() (sfepy.terms.termsAdjointNavierStokes.SDConvectionTerm static method), 523	function() (sfepy.terms.termsLinElasticity.LinearElasticIsotropicTerm static method), 548
function() (sfepy.terms.termsAdjointNavierStokes.SDDivGradientTerm static method), 524	function() (sfepy.terms.termsLinElasticity.LinearElasticTHTerm static method), 549
function() (sfepy.terms.termsAdjointNavierStokes.SDDivTractionTerm static method), 525	function() (sfepy.terms.termsLinElasticity.LinearStrainFiberTerm static method), 551
function() (sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm static method), 525	function() (sfepy.terms.termsLinElasticity.SDLinearElasticTerm static method), 551
function() (sfepy.terms.termsAdjointNavierStokes.SDGradientTerm static method), 526	

- function() (sfepy.terms.termsNavierStokes.ConvectTerm method), 552
- function() (sfepy.terms.termsNavierStokes.DivGradTerm method), 553
- function() (sfepy.terms.termsNavierStokes.DivOperatorTerm static method), 553
- function() (sfepy.terms.termsNavierStokes.DivTerm static method), 554
- function() (sfepy.terms.termsNavierStokes.GradDivStabilizationTerm method), 554
- function() (sfepy.terms.termsNavierStokes.GradTerm static method), 555
- function() (sfepy.terms.termsNavierStokes.LinearConvectTerm method), 556
- function() (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm method), 556
- function() (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm method), 557
- function() (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm method), 558
- function() (sfepy.terms.termsPoint.ConcentratedPointLoadTerm static method), 560
- function() (sfepy.terms.termsPoint.LinearPointSpringTerm static method), 561
- function() (sfepy.terms.termsSurface.ContactPlaneTerm static method), 562
- function() (sfepy.terms.termsSurface.LinearTractionTerm method), 562
- function() (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm static method), 563
- function() (sfepy.terms.termsSurface.SurfaceJumpTerm static method), 564
- function() (sfepy.terms.termsVolume.LinearVolumeForceTerm method), 565
- Functions (class in sfepy.fem.functions), 370
- G**
- GBarCoef (class in sfepy.homogenization.coefs\_elastic), 426
- gen\_block\_mesh() (in module sfepy.mesh.mesh\_generators), 457
- gen\_cylinder\_mesh() (in module sfepy.mesh.mesh\_generators), 457
- gen\_extended\_block\_mesh() (in module sfepy.mesh.mesh\_generators), 458
- gen\_gallery (module), 322
- gen\_lobatto() (in module gen\_lobatto1d\_c), 323
- gen\_lobatto1d\_c (module), 323
- gen\_mesh\_from\_goem() (in module sfepy.mesh.mesh\_generators), 458
- gen\_mesh\_from\_poly() (in module sfepy.mesh.mesh\_generators), 459
- gen\_mesh\_from\_string() (in module sfepy.mesh.mesh\_generators), 459
- gen\_mesh\_from\_voxels() (in module sfepy.mesh.mesh\_generators), 459
- gen\_misc\_mesh() (in module sfepy.mesh.mesh\_generators), 459
- gen\_points() (sfepy.fem.probes.RayProbe method), 393
- gen\_term\_table (module), 324
- gen\_term\_table() (in module gen\_term\_table), 324
- gen\_tiled\_mesh() (in module sfepy.mesh.mesh\_generators), 459
- generate\_a\_pyrex\_source() (in module build\_helpers), 318
- generate\_decreasing\_nonnegative\_tuples\_summing\_to() (in module sfepy.fem.simplex\_cubature), 406
- generate\_gallery\_html() (in module gen\_gallery), 323
- gen\_images() (in module gen\_gallery), 323
- generate\_mesh\_velocity() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase method), 467
- gen\_permutations() (in module sfepy.fem.simplex\_cubature), 406
- generate\_probes() (in module probe), 316
- generate\_rst\_files() (in module gen\_gallery), 323
- generate\_thumbnails() (in module gen\_gallery), 323
- generate\_unique\_permutations() (in module sfepy.fem.simplex\_cubature), 407
- GenericFileSource (class in sfepy.postprocess.sources), 474
- GenericSequenceFileSource (class in sfepy.postprocess.sources), 474
- genPerMesh (module), 319
- geometries (sfepy.terms.terms.Term attribute), 514
- geometries (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 581
- geometries (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 519
- geometries (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 520
- geometries (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term attribute), 521
- geometries (sfepy.terms.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm attribute), 529
- geometries (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj1StabilizationTerm attribute), 529
- geometries (sfepy.terms.termsAdjointNavierStokes.SUPGPAdj2StabilizationTerm attribute), 530
- geometries (sfepy.terms.termsNavierStokes.ConvectTerm attribute), 552
- geometries (sfepy.terms.termsNavierStokes.LinearConvectTerm attribute), 556
- geometries (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm attribute), 556
- geometries (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm attribute), 557
- geometries (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm attribute), 558

attribute), 558  
geometries (sfepy.terms.termsSurface.ContactPlaneTerm attribute), 562  
geometry (class in sfepy.mesh.geom\_tools), 455  
GeometryElement (class in sfepy.fem.geometry\_element), 370  
geomobject (class in sfepy.mesh.geom\_tools), 456  
get() (sfepy.base.base.Container method), 327  
get() (sfepy.base.base.Struct method), 328  
get() (sfepy.fem.integrals.Integrals method), 373  
get() (sfepy.mechanics.matcoefs.ElasticConstants method), 446  
get() (sfepy.terms.terms.Term method), 515  
get\_actual\_order() (in module sfepy.fem.quadratures), 402  
get\_all\_attributes() (in module sfepy.postprocess.dataset\_manager), 471  
get\_animation\_info() (sfepy.postprocess.viewer.Viewer method), 479  
get\_approximation() (sfepy.fem.variables.FieldVariable method), 412  
get\_approximation() (sfepy.terms.terms.Term method), 515  
get\_arg\_kinds() (in module sfepy.terms.terms), 517  
get\_arg\_name() (sfepy.terms.terms.Term method), 515  
get\_args() (sfepy.terms.terms.Term method), 515  
get\_args\_by\_name() (sfepy.terms.terms.Term method), 515  
get\_arguments() (in module sfepy.base.base), 330  
get\_array\_type() (in module sfepy.postprocess.dataset\_manager), 471  
get\_assembling\_cells() (sfepy.terms.terms.Term method), 515  
get\_attribute\_list() (in module sfepy.postprocess.dataset\_manager), 471  
get\_barycentric\_coors() (in module sfepy.fem.extmods.bases), 420  
get\_base() (sfepy.fem.fea.Approximation method), 360  
get\_base() (sfepy.fem.mappings.Mapping method), 373  
get\_basic\_info() (in module sfepy.version), 325  
get\_bounding\_box() (sfepy.fem.mesh.Mesh method), 381  
get\_bounding\_box() (sfepy.postprocess.sources.GenericFileSource method), 474  
get\_bounding\_box() (sfepy.postprocess.sources.VTKFileSource method), 474  
get\_box\_volume() (in module sfepy.homogenization.utils), 436  
get\_callback() (in module sfepy.homogenization.coefs\_phononic), 431  
get\_cauchy\_from\_2pk() (sfepy.mechanics.tensors.StressTransformation method), 450  
get\_cell\_offsets() (sfepy.fem.domain.Domain method), 349  
get\_cell\_offsets() (sfepy.fem.region.Region method), 405  
get\_cells() (sfepy.fem.region.Region method), 405  
get\_charfun() (sfepy.fem.region.Region method), 405  
get\_charge() (sfepy.physics.potentials.Potential method), 468  
get\_coefs() (sfepy.homogenization.coefs\_phononic.AcousticMassLiquidTensor method), 427  
get\_coefs() (sfepy.homogenization.coefs\_phononic.AcousticMassTensor method), 427  
get\_complete\_facets() (sfepy.fem.facets.Facets method), 358  
get\_component\_indices() (sfepy.fem.variables.FieldVariable method), 412  
get\_conn\_info() (sfepy.terms.terms.Term method), 515  
get\_conn\_key() (sfepy.terms.terms.Term method), 515  
get\_conn\_permutations() (sfepy.fem.geometry\_element.GeometryElement method), 370  
get\_connectivity() (sfepy.fem.fe\_surface.FESurface method), 360  
get\_connectivity() (sfepy.fem.fea.Approximation method), 360  
get\_conns() (sfepy.fem.domain.Domain method), 349  
get\_consistent\_unit\_set() (in module sfepy.mechanics.units), 454  
get\_constant\_data() (sfepy.fem.materials.Material method), 376  
get\_control\_points() (sfepy.mesh.splinebox.SplineBox method), 466  
get\_coor() (sfepy.fem.fields\_base.Field method), 363  
get\_coors() (sfepy.fem.facets.Facets method), 358  
get\_coors() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468  
get\_coors() (sfepy.physics.radial\_mesh.RadialVector method), 469  
get\_coors\_in\_ball() (in module sfepy.linalg.geometry), 438  
get\_coors\_in\_tube() (in module sfepy.linalg.geometry), 438  
get\_correctors\_from\_file() (in module sfepy.homogenization.micmac), 433  
get\_current\_group() (sfepy.terms.terms.Term method), 515  
get\_data() (sfepy.fem.materials.Material method), 376  
get\_data\_name() (in module sfepy.fem.probes), 393  
get\_data\_names() (sfepy.postprocess.viewer.Viewer method), 479  
get\_data\_ranges() (in module sfepy.postprocess.utils), 476  
get\_data\_shape() (sfepy.fem.variables.FieldVariable method), 412  
get\_data\_shape() (sfepy.terms.terms.Term method), 515  
get\_debug() (in module sfepy.base.base), 330  
get\_default() (in module sfepy.base.base), 330



- [get\\_default\\_attr\(\)](#) (in module `sfepy.base.base`), 330  
[get\\_default\\_time\\_step\(\)](#) (`sfepy.solvers.ts.VariableTimeStepper` method), 492  
[get\\_default\\_ts\(\)](#) (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
[get\\_dependency\\_graph\(\)](#) (in module `sfepy.fem.region`), 406  
[get\\_deviator\(\)](#) (in module `sfepy.mechanics.tensors`), 450  
[get\\_diameter\(\)](#) (`sfepy.fem.domain.Domain` method), 349  
[get\\_dim\(\)](#) (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
[get\\_dimension\(\)](#) (`sfepy.fem.meshio.VTKMeshIO` method), 387  
[get\\_dir\(\)](#) (in module `runTests`), 316  
[get\\_distance\(\)](#) (`sfepy.mechanics.contact_planes.ContactPlane` method), 445  
[get\\_distance\(\)](#) (`sfepy.physics.potentials.Potential` method), 468  
[get\\_dof\\_conn\(\)](#) (`sfepy.fem.variables.FieldVariable` method), 413  
[get\\_dof\\_conn\\_type\(\)](#) (`sfepy.terms.terms.Term` method), 515  
[get\\_dof\\_info\(\)](#) (`sfepy.fem.variables.FieldVariable` method), 413  
[get\\_dof\\_orientation\\_maps\(\)](#) (`sfepy.fem.facets.Facets` method), 359  
[get\\_dofs\(\)](#) (in module `save_basis`), 324  
[get\\_dofs\\_in\\_region\(\)](#) (`sfepy.fem.fields_base.Field` method), 363  
[get\\_dofs\\_in\\_region\\_group\(\)](#) (`sfepy.fem.fields_base.Field` method), 363  
[get\\_domain\(\)](#) (`sfepy.fem.equations.Equations` method), 352  
[get\\_dual\(\)](#) (`sfepy.fem.variables.Variable` method), 415  
[get\\_dump\\_name\(\)](#) (`sfepy.homogenization.coefs_base.CoefficientsBase` method), 424  
[get\\_dump\\_name\\_base\(\)](#) (`sfepy.homogenization.coefs_base.CoefficientsBase` method), 425  
[get\\_dump\\_name\\_base\(\)](#) (`sfepy.homogenization.coefs_elastic.CoefficientsElastic` method), 426  
[get\\_dump\\_name\\_base\(\)](#) (`sfepy.homogenization.coefs_elastic.CoefficientsElastic` method), 427  
[get\\_edge\\_graph\(\)](#) (`sfepy.fem.region.Region` method), 405  
[get\\_edge\\_paths\(\)](#) (in module `sfepy.fem.utils`), 409  
[get\\_edges\(\)](#) (`sfepy.fem.region.Region` method), 405  
[get\\_edges\\_per\\_face\(\)](#) (`sfepy.fem.geometry_element.GeometryElement` method), 370  
[get\\_element\\_coors\(\)](#) (`sfepy.fem.mesh.Mesh` method), 381  
[get\\_element\\_diameters\(\)](#) (`sfepy.fem.domain.Domain` method), 349  
[get\\_element\\_diameters\(\)](#) (`sfepy.fem.extmods.mappings.CMMapping` method), 421  
[get\\_element\\_diameters\(\)](#) (`sfepy.fem.variables.FieldVariable` method), 413  
[get\\_element\\_zeros\(\)](#) (`sfepy.fem.variables.FieldVariable` method), 413  
[get\\_eth\\_data\(\)](#) (`sfepy.terms.terms_th.ETHTerm` method), 583  
[get\\_eval\\_coors\(\)](#) (in module `sfepy.fem.linearizer`), 372  
[get\\_eval\\_dofs\(\)](#) (in module `sfepy.fem.linearizer`), 372  
[get\\_eval\\_expression\(\)](#) (in module `sfepy.fem.fields_base`), 367  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_dot.DotProductVolumeTerm` method), 568  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_dot.VectorDotGradScalarTerm` method), 570  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_fibres.FibresActiveTLTerm` method), 571  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm` method), 572  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_base.HyperElasticBase` method), 572  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm` method), 574  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm` method), 575  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm` method), 577  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm` method), 578  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_hyperelastic_ul.VolumeULTerm` method), 580  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.terms_membrane.TLMembraneTerm` method), 581  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAcoustic.DiffusionSATerm` method), 518  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm` method), 519  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm` method), 520  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm` method), 522  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.NSOFMinDP` method), 523  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.NSOFMinDP` method), 523  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDConvectTerm` method), 523  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm` method), 524  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDDivTerm` method), 525  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm` method), 525  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDGradDivStabiliz` method), 526  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabiliz` method), 526  
[get\\_eval\\_shape\(\)](#) (`sfepy.terms.termsAdjointNavierStokes.SDPSPGPStabiliz` method), 527

[getUPGDisplacement\(\)](#) (sfepy.problem.Problem method), 397  
[get\\_exp\(\)](#) (sfepy.homogenization.convolution.Convolution method), 431  
[get\\_expression\\_arg\\_names\(\)](#) (in module sfepy.fem.equations), 354  
[get\\_extrapolated\(\)](#) (sfepy.physics.radial\_mesh.RadialVector method), 469  
[get\\_face\\_areas\(\)](#) (in module sfepy.linalg.geometry), 439  
[get\\_faces\(\)](#) (sfepy.fem.region.Region method), 405  
[get\\_facet\\_dof\\_permutations\(\)](#) (in module sfepy.fem.facets), 360  
[get\\_facet\\_dof\\_permutations\(\)](#) (sfepy.fem.facets.Facets method), 359  
[get\\_facets\(\)](#) (sfepy.fem.domain.Domain method), 349  
[get\\_family\\_data\(\)](#) (sfepy.terms.terms\_hyperelastic\_base.HyperElasticTerm method), 572  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_constraints.NonPenetrationTerm method), 565  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.BCNewtonTerm method), 566  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.DotProductVolumeTerm method), 568  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.DotSPProductVolumeTerm method), 568  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.DotSPProductVolumeTerm method), 569  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm method), 569  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_dot.VectorDotGradScalarTerm method), 570  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_fibres.FibresActiveTLTerm method), 571  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_base.DeformationElasticTerm method), 572  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_base.HyperElasticTerm method), 572  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureElasticTerm method), 574  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm method), 575  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTerm method), 577  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm method), 577  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureTerm method), 578  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityTerm method), 579  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm method), 581  
[get\\_fargs\(\)](#) (sfepy.terms.terms\_membrane.TLMembraneTerm method), 581  
[get\\_fargs\(\)](#) (sfepy.terms.termsAcoustic.DiffusionSATerm method), 582

method), 518

get\_fargs() (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm method), 519

get\_fargs() (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm method), 520

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.AdjConvectTerm method), 521

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term method), 521

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm method), 522

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm method), 522

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.NSOFSuperfPressureTerm method), 522

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.NSOFSuperfPressureTerm method), 523

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDConvectTerm method), 523

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm method), 524

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDDivTerm method), 525

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDDotVelocityTerm method), 525

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDGradDivStabilizationTerm method), 526

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDPSPGCSFibersTerm method), 527

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDPSPGCSFibersTerm method), 527

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SDSUPGCSFibersTerm method), 528

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SUPGCA1StabilizationTerm method), 529

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SUPGPA1StabilizationTerm method), 529

get\_fargs() (sfepy.terms.termsAdjointNavierStokes.SUPGPA2StabilizationTerm method), 530

get\_fargs() (sfepy.terms.termsBasic.IntegrateMatTerm method), 531

get\_fargs() (sfepy.terms.termsBasic.IntegrateSurfaceTerm method), 532

get\_fargs() (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm method), 532

get\_fargs() (sfepy.terms.termsBasic.IntegrateVolumeTerm method), 533

get\_fargs() (sfepy.terms.termsBasic.SumNodalValuesTerm method), 534

get\_fargs() (sfepy.terms.termsBasic.SurfaceMomentTerm method), 534

get\_fargs() (sfepy.terms.termsBasic.VolumeSurfaceTerm method), 535

get\_fargs() (sfepy.terms.termsBasic.VolumeTerm method), 535

get\_fargs() (sfepy.terms.termsBiot.BiotETHTerm method), 536

get\_fargs() (sfepy.terms.termsBiot.BiotStressTerm method), 537

get\_fargs() (sfepy.terms.termsBiot.BiotTerm method), 538

get\_fargs() (sfepy.terms.termsBiot.BiotTHTerm method), 538

get\_fargs() (sfepy.terms.termsElectric.ElectricSourceTerm method), 539

get\_fargs() (sfepy.terms.termsLaplace.DiffusionCouplingTerm method), 540

get\_fargs() (sfepy.terms.termsLaplace.DiffusionRTerm method), 540

get\_fargs() (sfepy.terms.termsLaplace.DiffusionTerm method), 541

get\_fargs() (sfepy.terms.termsLaplace.DiffusionVelocityTerm method), 542

get\_fargs() (sfepy.terms.termsLaplace.PermeabilityRTerm method), 543

get\_fargs() (sfepy.terms.termsLaplace.SurfaceFluxTerm method), 543

get\_fargs() (sfepy.terms.termsLinElasticity.CauchyStrainTerm method), 545

get\_fargs() (sfepy.terms.termsLinElasticity.CauchyStressETHTerm method), 546

get\_fargs() (sfepy.terms.termsLinElasticity.CauchyStressTerm method), 547

get\_fargs() (sfepy.terms.termsLinElasticity.CauchyStressTHTerm method), 546

get\_fargs() (sfepy.terms.termsLinElasticity.LinearElasticETHTerm method), 548

get\_fargs() (sfepy.terms.termsLinElasticity.LinearElasticIsotropicTerm method), 548

get\_fargs() (sfepy.terms.termsLinElasticity.LinearElasticTerm method), 550

get\_fargs() (sfepy.terms.termsLinElasticity.LinearElasticTHTerm method), 549

get\_fargs() (sfepy.terms.termsLinElasticity.LinearPrestressTerm method), 550

get\_fargs() (sfepy.terms.termsLinElasticity.LinearStrainFiberTerm method), 551

get\_fargs() (sfepy.terms.termsLinElasticity.SDLinearElasticTerm method), 552

get\_fargs() (sfepy.terms.termsNavierStokes.ConvectTerm method), 552

get\_fargs() (sfepy.terms.termsNavierStokes.DivGradTerm method), 553

get\_fargs() (sfepy.terms.termsNavierStokes.DivOperatorTerm method), 553

get\_fargs() (sfepy.terms.termsNavierStokes.DivTerm method), 554

get\_fargs() (sfepy.terms.termsNavierStokes.GradDivStabilizationTerm method), 554

method), 554

get\_fargs() (sfepy.terms.termsNavierStokes.GradTerm method), 555

get\_fargs() (sfepy.terms.termsNavierStokes.LinearConvectiveTerm method), 556

get\_fargs() (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm method), 556

get\_fargs() (sfepy.terms.termsNavierStokes.StokesTerm method), 559

get\_fargs() (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm method), 557

get\_fargs() (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm method), 558

get\_fargs() (sfepy.terms.termsPiezo.PiezoCouplingTerm method), 559

get\_fargs() (sfepy.terms.termsPoint.ConcentratedPointLoadTerm method), 560

get\_fargs() (sfepy.terms.termsPoint.LinearPointSpringTerm method), 561

get\_fargs() (sfepy.terms.termsSurface.ContactPlaneTerm method), 562

get\_fargs() (sfepy.terms.termsSurface.LinearTractionTerm method), 562

get\_fargs() (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm method), 563

get\_fargs() (sfepy.terms.termsSurface.SurfaceNormalDotTerm method), 564

get\_fargs() (sfepy.terms.termsSurface.SurfaceJumpTerm method), 564

get\_fargs() (sfepy.terms.termsVolume.LinearVolumeForceTerm method), 565

get\_field() (sfepy.fem.variables.FieldVariable method), 413

get\_filename() (sfepy.homogenization.coefs\_elastic.GPlusCoef method), 426

get\_filename() (sfepy.homogenization.coefs\_elastic.RBiotCoef method), 426

get\_filename\_trunk() (sfepy.fem.meshio.MeshIO method), 386

get\_filename\_trunk() (sfepy.fem.meshio.UserMeshIO method), 387

get\_full() (sfepy.fem.variables.FieldVariable method), 413

get\_full() (sfepy.homogenization.convolution.ConvolutionKernel method), 431

get\_full\_indices() (in module sfepy.mechanics.tensors), 451

get\_full\_state() (sfepy.fem.variables.Variable method), 415

get\_function() (sfepy.base.conf.ProblemConf method), 335

get\_geom\_poly\_space() (sfepy.fem.fea.Interpolant method), 361

get\_geom\_poly\_space() (sfepy.fem.fea.SurfaceInterpolant method), 361

get\_geometry() (sfepy.fem.mappings.Mapping method), 373

get\_geometry() (sfepy.terms.terms\_new.NewTerm method), 583

get\_geometry\_key() (sfepy.terms.terms\_new.NewTerm method), 583

get\_geometry\_types() (sfepy.terms.terms.Term method), 515

get\_homo\_scale\_factor() (in module sfepy.postprocess.viewer), 479

get\_homo\_conns() (sfepy.fem.equations.Equations method), 352

get\_green\_strain\_sym3d() (in module sfepy.mechanics.membranes), 449

get\_grid() (sfepy.fem.geometry\_element.GeometryElement method), 370

get\_homog\_coefs\_linear() (in module sfepy.homogenization.micmac), 433

get\_index() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468

get\_indx() (sfepy.fem.variables.Variables method), 417

get\_info() (sfepy.fem.dof\_info.DofInfo method), 346

get\_initial\_condition() (sfepy.fem.variables.Variable method), 415

get\_initial\_state() (in module sfepy.solvers.ts\_solvers), 493

get\_integral\_info() (sfepy.terms.terms.Term method), 516

get\_integral\_info() (sfepy.terms.termsPoint.PointTermBase method), 561

get\_integrals() (sfepy.fem.problemDef.ProblemDefinition method), 397

get\_interp\_coors() (sfepy.fem.variables.FieldVariable method), 413

get\_interpolation\_name() (sfepy.fem.geometry\_element.GeometryElement method), 370

get\_invariants() (in module sfepy.mechanics.membranes), 450

get\_item\_by\_name() (sfepy.base.conf.ProblemConf method), 335

get\_jacobian() (in module sfepy.fem.mappings), 373

get\_key() (sfepy.fem.integrals.Integral method), 372

get\_keys() (sfepy.fem.materials.Material method), 376

get\_kwargs() (sfepy.terms.terms.Term method), 516

get\_lattice\_volume() (in module sfepy.homogenization.utils), 436

get\_lcbc\_operator() (sfepy.fem.equations.Equations method), 353

get\_lcbc\_operator() (sfepy.fem.variables.Variables method), 417

get\_local\_chunk() (sfepy.terms.terms.CharacteristicFunction method), 513

get\_log\_freqs() (in module

sfepy.homogenization.coefs\_phononic), 431  
 get\_log\_name() (sfepy.base.log.Log method), 338  
 get\_logging\_conf() (in module sfepy.base.log), 338  
 get\_mapping() (sfepy.fem.fields\_base.Field method), 364  
 get\_mapping() (sfepy.fem.mappings.SurfaceMapping method), 373  
 get\_mapping() (sfepy.fem.mappings.VolumeMapping method), 373  
 get\_mapping() (sfepy.fem.variables.FieldVariable method), 413  
 get\_mapping() (sfepy.terms.terms.Term method), 516  
 get\_mapping\_data() (in module sfepy.fem.mappings), 374  
 get\_maps() (sfepy.solvers.oseen.StabilizationFunction method), 489  
 get\_mat\_id() (sfepy.postprocess.sources.FileSource method), 473  
 get\_mat\_id() (sfepy.postprocess.sources.GenericFileSource method), 474  
 get\_material\_names() (sfepy.terms.terms.Term method), 516  
 get\_material\_names() (sfepy.terms.terms.Terms method), 517  
 get\_materials() (sfepy.fem.problemDef.ProblemDefinition method), 397  
 get\_materials() (sfepy.terms.terms.Term method), 516  
 get\_matrix\_shape() (sfepy.fem.variables.Variables method), 417  
 get\_merged\_values() (sfepy.fem.mappings.PhysicalQPs method), 373  
 get\_mesh\_bounding\_box() (sfepy.fem.domain.Domain method), 349  
 get\_mesh\_coors() (sfepy.fem.domain.Domain method), 349  
 get\_mesh\_coors() (sfepy.fem.problemDef.ProblemDefinition method), 397  
 get\_midpoint\_mesh() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468  
 get\_min\_dt() (in module sfepy.solvers.ts\_solvers), 493  
 get\_min\_edge\_size() (in module sfepy.fem.mesh), 382  
 get\_min\_value() (in module sfepy.fem.utils), 409  
 get\_min\_vertex\_distance() (in module sfepy.fem.mesh), 382  
 get\_min\_vertex\_distance\_naive() (in module sfepy.fem.mesh), 382  
 get\_mirror\_region() (sfepy.fem.region.Region method), 405  
 get\_mixing() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468  
 get\_mtx\_i() (sfepy.fem.poly\_spaces.LagrangeTensorProductPolySpace method), 390  
 get\_mtx\_i() (sfepy.fem.poly\_spaces.PolySpace method), 391  
 get\_n\_cells() (sfepy.fem.region.Region method), 405  
 get\_n\_dof\_total() (sfepy.fem.dof\_info.DofInfo method), 346  
 get\_n\_nodes() (sfepy.fem.fea.Interpolant method), 361  
 get\_names() (sfepy.base.base.Container method), 327  
 get\_names() (sfepy.base.base.OneTypeList method), 327  
 get\_non\_diagonal\_indices() (in module sfepy.mechanics.tensors), 451  
 get\_normals() (in module sfepy.fem.mappings), 375  
 get\_number() (sfepy.base.testing.TestCommon method), 343  
 get\_opacities() (in module sfepy.postprocess.viewer), 479  
 get\_operator() (sfepy.fem.dof\_info.EquationMap method), 346  
 get\_orientation() (sfepy.fem.facets.Facets method), 359  
 get\_output() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 425  
 get\_output\_approx\_order() (sfepy.fem.fields\_base.Field method), 364  
 get\_output\_function() (sfepy.base.base.Output method), 328  
 get\_output\_name() (sfepy.fem.problemDef.ProblemDefinition method), 397  
 get\_output\_prefix() (sfepy.base.base.Output method), 328  
 get\_output\_suffix() (in module sfepy.homogenization.recovery), 435  
 get\_parameter\_names() (sfepy.terms.terms.Term method), 516  
 get\_parameter\_variables() (sfepy.terms.terms.Term method), 516  
 get\_parent\_mesh() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468  
 get\_parents() (in module sfepy.fem.region), 406  
 get\_parts() (sfepy.fem.state.State method), 408  
 get\_permutation() (sfepy.fem.variables.CloseNodesIterator method), 410  
 get\_perpendiculars() (in module sfepy.linalg.geometry), 454  
 get\_physical\_qps() (in module sfepy.fem.mappings), 375  
 get\_physical\_qps() (sfepy.fem.mappings.Mapping method), 373  
 get\_physical\_qps() (sfepy.terms.terms.Term method), 516  
 get\_points() (sfepy.fem.probes.CircleProbe method), 391  
 get\_points() (sfepy.fem.probes.LineProbe method), 391  
 get\_points() (sfepy.fem.probes.PointsProbe method), 392  
 get\_points() (sfepy.fem.probes.RayProbe method), 393  
 get\_poly\_space() (sfepy.fem.fea.Approximation method), 361  
 get\_position\_counts() (in module sfepy.postprocess.viewer), 479  
 get\_poly\_space() (sfepy.mechanics.units.Unit static method), 454  
 get\_primary() (sfepy.fem.variables.Variable method), 415  
 get\_primary\_name() (sfepy.fem.variables.Variable method), 415



`get_print_info()` (in module `sfepy.base.ioutils`), 337  
`get_print_info()` (in module `sfepy.solvers.ts`), 492  
`get_qp()` (`sfepy.fem.fea.Approximation` method), 361  
`get_qp()` (`sfepy.fem.fea.SurfaceApproximation` method), 361  
`get_qp()` (`sfepy.fem.integrals.Integral` method), 372  
`get_qp_key()` (`sfepy.terms.terms.Term` method), 516  
`get_r()` (`sfepy.physics.radial_mesh.ExplicitRadialMesh` method), 468  
`get_range_indices()` (in module `sfepy.terms.utils`), 583  
`get_ranges()` (in module `sfepy.homogenization.coefs_phononic`), 431  
`get_raw()` (`sfepy.base.conf.ProblemConf` method), 335  
`get_reduced()` (`sfepy.fem.state.State` method), 408  
`get_reduced()` (`sfepy.fem.variables.FieldVariable` method), 413  
`get_ref_coors()` (in module `sfepy.fem.global_interp`), 371  
`get_region()` (`sfepy.terms.terms.ConnInfo` method), 514  
`get_region()` (`sfepy.terms.terms.Term` method), 516  
`get_region_name()` (`sfepy.terms.terms.ConnInfo` method), 514  
`get_s_data_shape()` (`sfepy.fem.fea.Approximation` method), 361  
`get_save_name()` (`sfepy.homogenization.coefs_base.CorrMiniApp` method), 425  
`get_save_name_base()` (`sfepy.homogenization.coefs_base.CorrMiniApp` method), 425  
`get_save_name_base()` (`sfepy.homogenization.coefs_elastic.TCorrector` method), 427  
`get_save_name_base()` (`sfepy.homogenization.coefs_elastic.TCorrector` method), 427  
`get_shape()` (`sfepy.fem.mappings.PhysicalQPs` method), 373  
`get_shape_kind()` (in module `sfepy.terms.terms`), 517  
`get_simplex_circumcentres()` (in module `sfepy.linalg.geometry`), 439  
`get_simplex_cubature()` (in module `sfepy.fem.simplex_cubature`), 407  
`get_size_hint()` (`sfepy.postprocess.viewer.Viewer` method), 479  
`get_solver_conf()` (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
`get_solvers()` (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
`get_sphinx_make_command()` (in module `build_helpers`), 318  
`get_standard_keywords()` (in module `sfepy.base.conf`), 335  
`get_standard_type_defs()` (in module `sfepy.base.parse_conf`), 340  
`get_state_in_region()` (`sfepy.fem.variables.FieldVariable` method), 413  
`get_state_names()` (`sfepy.terms.terms.Term` method), 516  
`get_state_part_view()` (`sfepy.fem.variables.Variables` method), 417  
`get_state_parts()` (`sfepy.fem.equations.Equations` method), 353  
`get_state_parts()` (`sfepy.fem.variables.Variables` method), 417  
`get_state_variables()` (`sfepy.terms.terms.Term` method), 516  
`get_step_range()` (`sfepy.postprocess.sources.FileSource` method), 473  
`get_step_range()` (`sfepy.postprocess.sources.GenericFileSource` method), 474  
`get_step_range()` (`sfepy.postprocess.sources.GenericSequenceFileSource` method), 474  
`get_step_range()` (`sfepy.postprocess.sources.VTKFileSource` method), 474  
`get_step_range()` (`sfepy.postprocess.sources.VTKSequenceFileSource` method), 475  
`get_subdict()` (in module `sfepy.base.base`), 330  
`get_subset_info()` (`sfepy.fem.dof_info.DofInfo` method), 346  
`get_surface_entities()` (`sfepy.fem.geometry_element.GeometryElement` method), 370  
`get_surface_entities()` (`sfepy.fem.region.Region` method), 451  
`get_sym_indices()` (in module `sfepy.mechanics.tensors`), 451  
`get_tangent_stress_matrix()` (in module `sfepy.mechanics.tensors`), 451  
`get_time_solver()` (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
`get_timestepper()` (`sfepy.fem.problemDef.ProblemDefinition` method), 397  
`get_tolerance()` (`sfepy.solvers.solvers.LinearSolver` method), 491  
`get_trace()` (in module `sfepy.mechanics.tensors`), 451  
`get_true_order()` (`sfepy.fem.fields_base.Field` method), 364  
`get_trunk()` (in module `sfepy.base.ioutils`), 337  
`get_uid_per_elements()` (`sfepy.fem.facets.Facets` method), 359  
`get_user_names()` (`sfepy.terms.terms.Term` method), 516  
`get_user_names()` (`sfepy.terms.terms.Terms` method), 517  
`get_v_data_shape()` (`sfepy.fem.fea.Approximation` method), 361  
`get_variable()` (`sfepy.fem.equations.Equations` method), 353  
`get_variable_names()` (`sfepy.fem.equations.Equations` method), 353  
`get_variable_names()` (`sfepy.terms.terms.Term` method), 516  
`get_variable_names()` (`sfepy.terms.terms.Terms` method), 517  
`get_variables()` (`sfepy.fem.problemDef.ProblemDefinition` method), 397

get\_variables() (sfepy.homogenization.coefs\_elastic.RBiotCoef method), 462  
 get\_variables() (sfepy.homogenization.coefs\_perfusion.CoeffRegion method), 427  
 get\_variables() (sfepy.homogenization.coefs\_perfusion.CorrRegion method), 427  
 get\_variables() (sfepy.terms.terms.Term method), 516  
 get\_vector() (sfepy.terms.terms.Term method), 516  
 get\_vector\_format() (sfepy.fem.meshio.MeshIO method), 386  
 get\_vectors() (sfepy.fem.dof\_info.EdgeDirectionOperator method), 346  
 get\_vectors() (sfepy.fem.dof\_info.NormalDirectionOperator method), 348  
 get\_vertices() (sfepy.fem.fields\_base.Field method), 364  
 get\_vertices() (sfepy.fem.region.Region method), 405  
 get\_vertices\_of\_cells() (sfepy.fem.region.Region method), 405  
 get\_virtual\_name() (sfepy.terms.terms.Term method), 516  
 get\_virtual\_variable() (sfepy.terms.terms.Term method), 516  
 get\_volume() (in module sfepy.homogenization.utils), 436  
 get\_volume\_from\_options() (in module sfepy.homogenization.homogen\_app), 433  
 get\_volumetric\_tensor() (in module sfepy.mechanics.tensors), 451  
 get\_von\_mises\_stress() (in module sfepy.mechanics.tensors), 451  
 get\_weighted\_norm() (sfepy.fem.state.State method), 408  
 getBCnum() (sfepy.mesh.geom\_tools.geometry method), 456  
 getcenterpoint() (sfepy.mesh.geom\_tools.surface method), 456  
 getele() (sfepy.fem.meshio.TetgenMeshIO static method), 387  
 getf() (sfepy.mesh.meshutils.bound method), 461  
 getholepoints() (sfepy.mesh.geom\_tools.surface method), 456  
 getinsidepoint() (sfepy.mesh.geom\_tools.surface method), 456  
 getinsidepoint() (sfepy.mesh.geom\_tools.volume method), 456  
 getlines() (sfepy.mesh.geom\_tools.surface method), 456  
 getn() (sfepy.mesh.geom\_tools.geomobject method), 456  
 getnodes() (sfepy.fem.meshio.TetgenMeshIO static method), 387  
 getnormal() (in module sfepy.mesh.femlab), 454  
 getnumhexahedra() (sfepy.mesh.meshutils.mesh method), 462  
 getnumprisms() (sfepy.mesh.meshutils.mesh method), 462  
 getnumquadrangles() (sfepy.mesh.meshutils.mesh method), 462  
 getnumtetrahedra() (sfepy.mesh.meshutils.mesh method), 462  
 getnumtriangles() (sfepy.mesh.meshutils.mesh method), 462  
 getp3() (in module sfepy.mesh.femlab), 454  
 getpoints() (sfepy.mesh.geom\_tools.line method), 456  
 getpoints() (sfepy.mesh.geom\_tools.surface method), 456  
 getscalar() (sfepy.mesh.meshutils.mesh method), 462  
 getscalar2() (sfepy.mesh.meshutils.mesh method), 462  
 getstr() (sfepy.mesh.geom\_tools.point method), 456  
 getstr() (sfepy.mesh.meshutils.bound method), 461  
 getsurfaces() (sfepy.mesh.geom\_tools.physicalsurface method), 456  
 getsurfaces() (sfepy.mesh.geom\_tools.volume method), 457  
 getvector() (sfepy.mesh.meshutils.mesh method), 463  
 getvolumes() (sfepy.mesh.geom\_tools.physicalvolume method), 456  
 getxyz() (sfepy.mesh.geom\_tools.point method), 456  
 getxyz() (sfepy.mesh.meshutils.mesh method), 463  
 GPlusCoef (class in sfepy.homogenization.coefs\_elastic), 426  
 grad() (sfepy.fem.variables.FieldVariable method), 413  
 grad\_as\_vector() (in module sfepy.terms.termsAdjointNavierStokes), 530  
 grad\_qp() (sfepy.fem.variables.FieldVariable method), 413  
 grad\_vector\_to\_matrix() (in module evalForms), 322  
 GradDivStabilizationTerm (class in sfepy.terms.termsNavierStokes), 554  
 GradTerm (class in sfepy.terms.termsNavierStokes), 555  
 graph\_components() (in module sfepy.fem.extmods.mesh), 422  
 group\_by\_variables() (sfepy.fem.conditions.Conditions method), 343  
 group\_chains() (in module sfepy.fem.dof\_info), 348  
 guess() (sfepy.fem.meshio.AbaqusMeshIO static method), 383  
 guess() (sfepy.fem.meshio.ANSYSCDBMeshIO static method), 383  
 guess() (sfepy.fem.meshio.AVSUCDMeshIO static method), 383  
 guess\_format() (in module sfepy.fem.meshio), 388  
 guess\_n\_eigs() (in module sfepy.physics.schroedinger\_app), 470  
 guess\_time\_units() (in module sfepy.postprocess.time\_history), 476

## H

H1DiscontinuousField (class in sfepy.fem.fields\_nodal), 368  
 H1HierarchicVolumeField (class in sfepy.fem.fields\_hierarchic), 367

- H1NodalMixin (class in sfepy.fem.fields\_nodal), 368
- H1NodalSurfaceField (class in sfepy.fem.fields\_nodal), 369
- H1NodalVolumeField (class in sfepy.fem.fields\_nodal), 369
- handle() (sfepy.mesh.meshutils.bound method), 461
- handle2() (sfepy.mesh.meshutils.bound method), 461
- handle\_resize() (sfepy.base.progressbar.ProgressBar method), 341
- handleelement() (sfepy.mesh.meshutils.bound method), 461
- has\_attr() (in module sfepy.config), 325
- has\_cells() (sfepy.fem.region.Region method), 405
- has\_cells\_if\_can() (sfepy.fem.region.Region method), 405
- has\_ebc() (sfepy.fem.state.State method), 408
- has\_ebc() (sfepy.fem.variables.Variables method), 417
- has\_extra\_nodes() (sfepy.fem.poly\_spaces.NodeDescription method), 390
- has\_faces() (sfepy.fem.domain.Domain method), 349
- has\_key() (sfepy.base.base.Container method), 327
- has\_same\_mesh() (sfepy.fem.variables.FieldVariable method), 414
- has\_virtuals() (sfepy.fem.variables.Variables method), 417
- have\_good\_cython() (in module build\_helpers), 318
- HDF5MeshIO (class in sfepy.fem.meshio), 383
- he\_eval\_from\_mtx() (in module sfepy.terms.extmods.terms), 586
- he\_residuum\_from\_mtx() (in module sfepy.terms.extmods.terms), 586
- Histories (class in sfepy.fem.history), 371
- History (class in sfepy.fem.history), 371
- homogen (module), 314
- HomogenizationApp (class in sfepy.homogenization.homogen\_app), 432
- HomogenizationEngine (class in sfepy.homogenization.engine), 432
- hyperelastic\_mode (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLBase attribute), 575
- hyperelastic\_mode (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULBase attribute), 579
- HyperElasticBase (class in sfepy.terms.terms\_hyperelastic\_base), 572
- HyperElasticTLBase (class in sfepy.terms.terms\_hyperelastic\_tl), 575
- HyperElasticULBase (class in sfepy.terms.terms\_hyperelastic\_ul), 579
- HypermeshAsciiMeshIO (class in sfepy.fem.meshio), 384
- IndexedStruct (class in sfepy.base.base), 327
- InDir (class in sfepy.base.ioutils), 337
- infinity\_norm() (in module sfepy.linalg.sparse), 441
- init() (sfepy.base.progressbar.MyBar method), 341
- init() (sfepy.mechanics.matcoefs.ElasticConstants method), 446
- init\_data() (sfepy.fem.variables.Variable method), 415
- init\_history() (sfepy.fem.state.State method), 408
- init\_history() (sfepy.fem.variables.Variable method), 415
- init\_history() (sfepy.fem.variables.Variables method), 417
- init\_session() (in module sfepy.interactive.session), 437
- init\_solvers() (sfepy.fem.problemDef.ProblemDefinition method), 397
- init\_solvers() (sfepy.homogenization.coefs\_base.MiniAppBase method), 425
- init\_time() (sfepy.fem.equations.Equations method), 353
- init\_time() (sfepy.fem.problemDef.ProblemDefinition method), 397
- init\_variables() (sfepy.fem.problemDef.ProblemDefinition method), 397
- InitialCondition (class in sfepy.fem.conditions), 344
- initialize\_options() (build\_helpers.NoOptionsDocs method), 318
- insert() (sfepy.base.base.Container method), 327
- insert() (sfepy.physics.potentials.CompoundPotential method), 468
- insert() (sfepy.terms.terms.Terms method), 517
- insert\_as\_static\_method() (in module sfepy.base.base), 330
- insert\_method() (in module sfepy.base.base), 330
- insert\_sparse\_to\_csr() (in module sfepy.linalg.sparse), 441
- insert\_static\_method() (in module sfepy.base.base), 330
- insert\_strided\_axis() (in module sfepy.linalg.utils), 443
- insert\_sub\_reqs() (in module sfepy.homogenization.engine), 432
- int\_dt() (sfepy.homogenization.convolution.ConvolutionKernel method), 431
- IntegralBase (class in sfepy.fem.integrals), 372
- integral (sfepy.fem.extmods.mappings.CMapping attribute), 421
- IntegralMeanValueOperator (class in sfepy.fem.dof\_info), 347
- IntegralProbe (class in sfepy.fem.probes), 391
- Integrals (class in sfepy.fem.integrals), 372
- integrate() (sfepy.fem.extmods.mappings.CMapping method), 421
- integrate() (sfepy.fem.integrals.Integral method), 372
- integrate() (sfepy.physics.radial\_mesh.RadialMesh method), 469
- integrate() (sfepy.physics.radial\_mesh.RadialVector method), 469
- integrate() (sfepy.terms.terms\_hyperelastic\_base.HyperElasticBase static method), 572
- igs() (sfepy.terms.terms.Term method), 516
- import\_file() (in module sfepy.base.base), 330
- in1d() (in module sfepy.base.compat), 332



- `integrate()` (sfepy.terms.terms\_new.NewTerm method), 583
- `integrate_along_line()` (in module probe), 316
- `integrate_in_time()` (in module sfepy.homogenization.utils), 436
- `IntegrateMatTerm` (class in sfepy.terms.termsBasic), 530
- `IntegrateSurfaceOperatorTerm` (class in sfepy.terms.termsBasic), 531
- `IntegrateSurfaceTerm` (class in sfepy.terms.termsBasic), 531
- `IntegrateVolumeOperatorTerm` (class in sfepy.terms.termsBasic), 532
- `IntegrateVolumeTerm` (class in sfepy.terms.termsBasic), 532
- `integration` (sfepy.terms.terms.Term attribute), 516
- `integration` (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 565
- `integration` (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 567
- `integration` (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTerm attribute), 577
- `integration` (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 581
- `integration` (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 519
- `integration` (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 520
- `integration` (sfepy.terms.termsAdjointNavierStokes.NSOFSurfaceTerm attribute), 523
- `integration` (sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm attribute), 531
- `integration` (sfepy.terms.termsBasic.IntegrateSurfaceTerm attribute), 532
- `integration` (sfepy.terms.termsBasic.SurfaceMomentTerm attribute), 534
- `integration` (sfepy.terms.termsBasic.SurfaceTerm attribute), 534
- `integration` (sfepy.terms.termsBasic.VolumeSurfaceTerm attribute), 535
- `integration` (sfepy.terms.termsLaplace.SurfaceFluxTerm attribute), 543
- `integration` (sfepy.terms.termsLinElasticity.CauchyStrainStressTerm attribute), 544
- `integration` (sfepy.terms.termsPoint.ConcentratedPointLoadTerm attribute), 560
- `integration` (sfepy.terms.termsPoint.LinearPointSpringTerm attribute), 561
- `integration` (sfepy.terms.termsSurface.ContactPlaneTerm attribute), 562
- `integration` (sfepy.terms.termsSurface.LinearTractionTerm attribute), 562
- `integration` (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm attribute), 563
- `integration` (sfepy.terms.termsSurface.SurfaceNormalDotTerm attribute), 564
- `integration` (sfepy.terms.termsSurface.SurfaceJumpTerm attribute), 564
- `interp_box_coordinates()` (in module sfepy.optimize.freeFormDef), 467
- `interp_conv_mat()` (in module sfepy.homogenization.utils), 436
- `interp_coordinates()` (sfepy.optimize.freeFormDef.SplineBoxes method), 466
- `interp_mesh_velocity()` (sfepy.optimize.freeFormDef.SplineBoxes method), 467
- `interp_to_qp()` (sfepy.fem.fields\_base.Field method), 364
- `interp_v_vals_to_n_vals()` (sfepy.fem.fields\_nodal.H1NodalSurfaceField method), 369
- `interp_v_vals_to_n_vals()` (sfepy.fem.fields\_nodal.H1NodalVolumeField method), 369
- `Interpolant` (class in sfepy.fem.fea), 361
- `interpolate()` (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468
- `interpolate()` (sfepy.physics.radial\_mesh.RadialVector method), 469
- `interpolate_3d()` (sfepy.physics.radial\_mesh.RadialMesh method), 469
- `interpolate_3d()` (sfepy.physics.radial\_mesh.RadialVector method), 469
- `interpolate_3d()` (sfepy.fem.region.Region method), 405
- `intersect_n()` (sfepy.fem.region.Region method), 405
- `intervals()` (sfepy.physics.radial\_mesh.RadialMesh method), 469
- `invalidate_evaluate_cache()` (sfepy.fem.variables.FieldVariable method), 414
- `invalidate_term_caches()` (sfepy.fem.equations.Equations method), 353
- `inverse_action()` (sfepy.fem.mass\_operator.MassOperator method), 375
- `inverse_element_mapping()` (in module sfepy.linalg.geometry), 439
- `invert_dict()` (in module sfepy.base.base), 330
- `invert_remap()` (in module sfepy.fem.utils), 410
- `iplot()` (in module sfepy.base.plotutils), 340
- `is_active_bc()` (in module sfepy.fem.dof\_info), 348
- `is_complex()` (sfepy.fem.variables.Variable method), 415
- `is_cyclic` (sfepy.fem.probes.CircleProbe attribute), 391
- `is_cyclic` (sfepy.fem.probes.Probe attribute), 392
- `is_derived_class()` (in module sfepy.base.base), 331
- `is_finite()` (sfepy.fem.variables.Variable method), 415
- `is_higher_order()` (sfepy.fem.fields\_base.Field method), 364
- `is_kind()` (sfepy.fem.variables.Variable method), 416
- `is_linear()` (sfepy.fem.problemDef.ProblemDefinition method), 397

- is\_parameter() (sfepy.fem.variables.Variable method), 416
- is\_real() (sfepy.fem.variables.Variable method), 416
- is\_release() (sfepy.config.Config method), 325
- is\_sequence() (in module sfepy.base.base), 331
- is\_state() (sfepy.fem.variables.Variable method), 416
- is\_state\_or\_parameter() (sfepy.fem.variables.Variable method), 416
- is\_virtual() (sfepy.fem.variables.Variable method), 416
- iter\_cells() (sfepy.fem.region.Region method), 405
- iter\_dict\_of\_lists() (in module sfepy.base.base), 331
- iter\_dofs() (sfepy.fem.variables.FieldVariable method), 414
- iter\_from() (sfepy.solvers.ts.TimeStepper method), 491
- iter\_groups() (sfepy.fem.domain.Domain method), 349
- iter\_groups() (sfepy.mechanics.friction.DualMesh method), 445
- iter\_groups() (sfepy.terms.terms.Term method), 516
- iter\_igs() (sfepy.terms.terms.ConnInfo method), 514
- iter\_names() (sfepy.base.log.Log method), 338
- iter\_single() (sfepy.fem.conditions.Condition method), 343
- iter\_solutions() (sfepy.homogenization.coefs\_base.CorrSolution method), 425
- iter\_state() (sfepy.fem.variables.Variables method), 417
- iter\_sym() (in module sfepy.homogenization.utils), 436
- iter\_terms() (sfepy.fem.materials.Material method), 376
- iteritems() (sfepy.base.base.Container method), 327
- iterkeys() (sfepy.base.base.Container method), 327
- itervalues() (sfepy.base.base.Container method), 327
- ## J
- join\_conn\_groups() (in module sfepy.fem.meshio), 388
- join\_tokens() (in module sfepy.fem.parseReg), 388
- ## K
- key (sfepy.base.goptions.ValidatedDict attribute), 336
- keys (sfepy.fem.poly\_spaces.PolySpace attribute), 391
- keys() (sfepy.base.goptions.ValidatedDict method), 336
- ## L
- LagrangeNodes (class in sfepy.fem.poly\_spaces), 389
- LagrangeSimplexBPolySpace (class in sfepy.fem.poly\_spaces), 389
- LagrangeSimplexPolySpace (class in sfepy.fem.poly\_spaces), 389
- LagrangeTensorProductPolySpace (class in sfepy.fem.poly\_spaces), 390
- lamé\_from\_youngpoisson() (in module sfepy.mechanics.matcoefs), 447
- LaplaceTerm (class in sfepy.terms.termsLaplace), 542
- last\_point() (sfepy.physics.radial\_mesh.ExplicitRadialMesh method), 468
- LCBCEvaluator (class in sfepy.fem.evaluate), 355
- LCBCOperator (class in sfepy.fem.dof\_info), 347
- LCBCOperators (class in sfepy.fem.dof\_info), 347
- leaveonlyphysicalsurfaces() (sfepy.mesh.geom\_tools.geometry method), 456
- leaveonlyphysicalvolumes() (sfepy.mesh.geom\_tools.geometry method), 456
- light\_copy() (sfepy.fem.region.Region method), 405
- line (class in sfepy.mesh.geom\_tools), 456
- linear\_derivatives() (sfepy.physics.radial\_mesh.RadialVector method), 469
- linear\_integral() (sfepy.physics.radial\_mesh.RadialMesh method), 469
- linear\_integral() (sfepy.physics.radial\_mesh.RadialVector method), 469
- linear\_integrate() (sfepy.physics.radial\_mesh.RadialMesh method), 469
- linear\_integrate() (sfepy.physics.radial\_mesh.RadialVector method), 469
- LinearCombinationBC (class in sfepy.fem.conditions), 344
- LinearConvectTerm (class in sfepy.terms.termsNavierStokes), 555
- LinearElasticETHTerm (class in sfepy.terms.termsLinElasticity), 547
- LinearElasticIsotropicTerm (class in sfepy.terms.termsLinElasticity), 548
- LinearElasticTerm (class in sfepy.terms.termsLinElasticity), 549
- LinearElasticTHTerm (class in sfepy.terms.termsLinElasticity), 548
- linearize() (sfepy.fem.fields\_base.Field method), 364
- LinearPointSpringTerm (class in sfepy.terms.termsPoint), 560
- LinearPrestressTerm (class in sfepy.terms.termsLinElasticity), 550
- LinearSolver (class in sfepy.solvers.solvers), 490
- LinearStrainFiberTerm (class in sfepy.terms.termsLinElasticity), 550
- LinearTractionTerm (class in sfepy.terms.termsSurface), 562
- LinearVolumeForceTerm (class in sfepy.terms.termsVolume), 564
- LineProbe (class in sfepy.fem.probes), 391
- link\_duals() (sfepy.fem.variables.Variables method), 417
- link\_flags() (sfepy.config.Config method), 325
- list\_of() (in module sfepy.base.parse\_conf), 340
- load\_classes() (in module sfepy.base.base), 331
- load\_coefs() (in module plotPerfusionCoefs), 320
- load\_dict() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 326
- LobattoTensorProductPolySpace (class in sfepy.fem.poly\_spaces), 390

- LOBPCGEigenvalueSolver (class in sfepy.solvers.eigen), 480
- localize() (sfepy.fem.mesh.Mesh method), 381
- locate\_files() (in module sfepy.base.ioutils), 337
- Log (class in sfepy.base.log), 338
- LogPlotter (class in sfepy.base.log\_plotter), 339
- ## M
- main() (in module blockgen), 320
- main() (in module config), 321
- main() (in module convert\_mesh), 321
- main() (in module cylindergen), 321
- main() (in module edit\_identifiers), 321
- main() (in module evalForms), 322
- main() (in module extractor), 314
- main() (in module findSurf), 319
- main() (in module gen\_gallery), 323
- main() (in module gen\_lobatto1d\_c), 323
- main() (in module gen\_term\_table), 324
- main() (in module genPerMesh), 319
- main() (in module homogen), 314
- main() (in module phonon), 314
- main() (in module plot\_condition\_numbers), 324
- main() (in module plotPerfusionCoefs), 320
- main() (in module postproc), 315
- main() (in module probe), 316
- main() (in module runTests), 316
- main() (in module save\_basis), 324
- main() (in module schroedinger), 317
- main() (in module sfepy.mesh.mesh\_generators), 460
- main() (in module shaper), 317
- main() (in module show\_authors), 324
- main() (in module simple), 317
- main() (in module sync\_module\_docs), 324
- main() (in module test\_install), 320
- make\_animation() (in module sfepy.postprocess.viewer), 479
- make\_axes() (sfepy.base.log\_plotter.LogPlotter method), 339
- make\_axis\_rotation\_matrix() (in module sfepy.linalg.geometry), 439
- make\_explicit\_step() (in module sfepy.solvers.ts\_solvers), 493
- make\_format() (sfepy.fem.meshio.ANSYSCDBMeshIO static method), 383
- make\_full() (sfepy.physics.schroedinger\_app.SchroedingerApp method), 470
- make\_full\_vec() (sfepy.fem.equations.Equations method), 353
- make\_full\_vec() (sfepy.fem.evaluate.BasicEvaluator method), 355
- make\_full\_vec() (sfepy.fem.variables.Variables method), 417
- make\_get\_conf() (in module sfepy.solvers.solvers), 491
- make\_global\_lcbc\_operator() (in module sfepy.fem.dof\_info), 348
- make\_h1\_projection\_data() (in module sfepy.fem.projections), 400
- make\_implicit\_step() (in module sfepy.solvers.ts\_solvers), 493
- make\_inverse\_connectivity() (in module sfepy.fem.mesh), 382
- make\_l2\_projection() (in module sfepy.fem.projections), 400
- make\_l2\_projection\_data() (in module sfepy.fem.projections), 400
- make\_mesh() (in module sfepy.fem.mesh), 382
- make\_point\_cells() (in module sfepy.fem.mesh), 382
- map\_equations() (sfepy.fem.dof\_info.EquationMap method), 346
- map\_permutations() (in module sfepy.linalg.utils), 444
- Mapping (class in sfepy.fem.mappings), 373
- mark\_surface\_facets() (sfepy.fem.facets.Facets method), 359
- mark\_time() (in module sfepy.base.base), 331
- mask\_points() (sfepy.mechanics.contact\_planes.ContactPlane method), 445
- MassOperator (class in sfepy.fem.mass\_operator), 375
- match\_candidate() (in module edit\_identifiers), 321
- match\_coors() (in module sfepy.fem.periodic), 389
- match\_grid\_line() (in module sfepy.fem.periodic), 389
- match\_grid\_plane() (in module sfepy.fem.periodic), 389
- match\_x\_line() (in module sfepy.fem.periodic), 389
- match\_x\_plane() (in module sfepy.fem.periodic), 389
- match\_y\_line() (in module sfepy.fem.periodic), 389
- match\_y\_plane() (in module sfepy.fem.periodic), 389
- match\_z\_line() (in module sfepy.fem.periodic), 389
- match\_z\_plane() (in module sfepy.fem.periodic), 389
- Material (class in sfepy.fem.materials), 375
- Materials (class in sfepy.fem.materials), 377
- MatrixAction (class in sfepy.linalg.utils), 442
- mc2us() (in module edit\_identifiers), 321
- MeditMeshIO (class in sfepy.fem.meshio), 384
- MEDMeshIO (class in sfepy.fem.meshio), 384
- merge() (sfepy.physics.radial\_mesh.RadialMesh static method), 469
- merge\_mesh() (in module sfepy.fem.mesh), 382
- Mesh (class in sfepy.fem.mesh), 378
- mesh (class in sfepy.mesh.meshutils), 462
- Mesh3DMeshIO (class in sfepy.fem.meshio), 384
- mesh\_from\_groups() (in module sfepy.fem.meshio), 388
- MeshIO (class in sfepy.fem.meshio), 385
- MeshUtilsCheckError, 461
- MeshUtilsError, 461
- MeshUtilsParseError, 461
- MeshUtilsWarning, 461
- mini\_newton() (in module sfepy.linalg.utils), 444

MiniAppBase (class in sfepy.homogenization.coefs\_base), 425  
 mmax() (sfepy.mesh.splinebox.SplineBox static method), 466  
 mode (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 mode (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 566  
 modes (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 565  
 modes (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 567  
 modes (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 568  
 modes (sfepy.terms.terms\_dot.VectorDotGradScalarTerm attribute), 570  
 modes (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 519  
 modes (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 520  
 modes (sfepy.terms.termsBiot.BiotETHTerm attribute), 536  
 modes (sfepy.terms.termsBiot.BiotTerm attribute), 538  
 modes (sfepy.terms.termsBiot.BiotTHTerm attribute), 538  
 modes (sfepy.terms.termsLaplace.DiffusionCoupling attribute), 540  
 modes (sfepy.terms.termsLaplace.DiffusionTerm attribute), 541  
 modes (sfepy.terms.termsLaplace.LaplaceTerm attribute), 542  
 modes (sfepy.terms.termsLinElasticity.LinearElasticTerm attribute), 550  
 modes (sfepy.terms.termsLinElasticity.LinearPrestressTerm attribute), 550  
 modes (sfepy.terms.termsNavierStokes.DivGradTerm attribute), 553  
 modes (sfepy.terms.termsNavierStokes.StokesTerm attribute), 559  
 modes (sfepy.terms.termsPiezo.PiezoCouplingTerm attribute), 559  
 modes (sfepy.terms.termsSurface.SurfaceNormalDotTerm attribute), 564  
 MooneyRivlinTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 575  
 MooneyRivlinULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 579  
 mulATB\_integrate() (in module sfepy.terms.extmods.terms), 586  
 MyBar (class in sfepy.base.progressbar), 341  
 myfloat() (in module sfepy.mesh.meshutils), 465

**N**

n\_el (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 n\_ep (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 n\_qp (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 name (sfepy.fem.poly\_spaces.LagrangeSimplexBPolySpace attribute), 389  
 name (sfepy.fem.poly\_spaces.LagrangeSimplexPolySpace attribute), 389  
 name (sfepy.fem.poly\_spaces.LagrangeTensorProductPolySpace attribute), 390  
 name (sfepy.fem.poly\_spaces.LobattoTensorProductPolySpace attribute), 390  
 name (sfepy.solvers.eigen.LOBPCGEigenvalueSolver attribute), 480  
 name (sfepy.solvers.eigen.PysparseEigenvalueSolver attribute), 480  
 name (sfepy.solvers.eigen.ScipyEigenvalueSolver attribute), 480  
 name (sfepy.solvers.eigen.ScipySGEigenvalueSolver attribute), 480  
 name (sfepy.solvers.ls.PETScKrylovSolver attribute), 481  
 name (sfepy.solvers.ls.PETScParallelKrylovSolver attribute), 482  
 name (sfepy.solvers.ls.PyAMGSolver attribute), 482  
 name (sfepy.solvers.ls.SchurComplement attribute), 483  
 name (sfepy.solvers.ls.SchurGeneralized attribute), 483  
 name (sfepy.solvers.ls.ScipyDirect attribute), 483  
 name (sfepy.solvers.ls.ScipyIterative attribute), 484  
 name (sfepy.solvers.ls.Umfpack attribute), 484  
 name (sfepy.solvers.nls.Newton attribute), 486  
 name (sfepy.solvers.nls.ScipyBroyden attribute), 487  
 name (sfepy.solvers.optimize.FMinSteepestDescent attribute), 487  
 name (sfepy.solvers.optimize.ScipyFMinSolver attribute), 488  
 name (sfepy.solvers.oseen.Oseen attribute), 489  
 name (sfepy.solvers.semismooth\_newton.SemismoothNewton attribute), 490  
 name (sfepy.solvers.ts\_solvers.AdaptiveTimeSteppingSolver attribute), 492  
 name (sfepy.solvers.ts\_solvers.ExplicitTimeSteppingSolver attribute), 492  
 name (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver attribute), 492  
 name (sfepy.solvers.ts\_solvers.StationarySolver attribute), 492  
 name (sfepy.terms.terms.Term attribute), 516  
 name (sfepy.terms.terms\_constraints.NonPenetrationTerm attribute), 565  
 name (sfepy.terms.terms\_dot.BCNewtonTerm attribute), 566  
 name (sfepy.terms.terms\_dot.DotProductSurfaceTerm attribute), 567

tribute), 567

name (sfepy.terms.terms\_dot.DotProductVolumeTerm attribute), 568

name (sfepy.terms.terms\_dot.DotSPProductVolumeOperator attribute), 568

name (sfepy.terms.terms\_dot.DotSPProductVolumeOperator attribute), 569

name (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm attribute), 569

name (sfepy.terms.terms\_dot.VectorDotGradScalarTerm attribute), 570

name (sfepy.terms.terms\_fibres.FibresActiveTLTerm attribute), 571

name (sfepy.terms.terms\_hyperelastic\_base.DeformationGradientsTLTerm attribute), 572

name (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm attribute), 573

name (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm attribute), 573

name (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm attribute), 574

name (sfepy.terms.terms\_hyperelastic\_tl.DiffusionTLTerm attribute), 575

name (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm attribute), 575

name (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm attribute), 576

name (sfepy.terms.terms\_hyperelastic\_tl.SurfaceTractionTLTerm attribute), 577

name (sfepy.terms.terms\_hyperelastic\_tl.VolumeTLTerm attribute), 577

name (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm attribute), 578

name (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm attribute), 578

name (sfepy.terms.terms\_hyperelastic\_ul.CompressibilityULTerm attribute), 579

name (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm attribute), 579

name (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm attribute), 580

name (sfepy.terms.terms\_hyperelastic\_ul.VolumeULTerm attribute), 581

name (sfepy.terms.terms\_membrane.TLMembraneTerm attribute), 581

name (sfepy.terms.terms\_new.NewDiffusionTerm attribute), 582

name (sfepy.terms.terms\_new.NewLinearElasticTerm attribute), 582

name (sfepy.terms.terms\_new.NewMassScalarTerm attribute), 582

name (sfepy.terms.terms\_new.NewMassTerm attribute), 582

name (sfepy.terms.termsAcoustic.DiffusionSATerm attribute), 582

tribute), 518

name (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm attribute), 519

name (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm attribute), 520

name (sfepy.terms.termsAdjointNavierStokes.AdjConvect1Term attribute), 521

name (sfepy.terms.termsAdjointNavierStokes.AdjConvect2Term attribute), 521

name (sfepy.terms.termsAdjointNavierStokes.AdjDivGradTerm attribute), 522

name (sfepy.terms.termsAdjointNavierStokes.NSOFMinGradTerm attribute), 522

name (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresDiffTerm attribute), 522

name (sfepy.terms.termsAdjointNavierStokes.NSOFSurfMinDPresTerm attribute), 523

name (sfepy.terms.termsAdjointNavierStokes.SDConvectTerm attribute), 523

name (sfepy.terms.termsAdjointNavierStokes.SDDivGradTerm attribute), 524

name (sfepy.terms.termsAdjointNavierStokes.SDDivTerm attribute), 525

name (sfepy.terms.termsAdjointNavierStokes.SDDotVolumeTerm attribute), 525

name (sfepy.terms.termsAdjointNavierStokes.SDGradDivStabilizationTerm attribute), 526

name (sfepy.terms.termsAdjointNavierStokes.SDPSPGCStabilizationTerm attribute), 527

name (sfepy.terms.termsAdjointNavierStokes.SDPSPGPStabilizationTerm attribute), 527

name (sfepy.terms.termsAdjointNavierStokes.SDSUPGCStabilizationTerm attribute), 528

name (sfepy.terms.termsAdjointNavierStokes.SUPGCAdjStabilizationTerm attribute), 529

name (sfepy.terms.termsAdjointNavierStokes.SUPGPAAdj1StabilizationTerm attribute), 529

name (sfepy.terms.termsAdjointNavierStokes.SUPGPAAdj2StabilizationTerm attribute), 530

name (sfepy.terms.termsBasic.IntegrateMatTerm attribute), 531

name (sfepy.terms.termsBasic.IntegrateSurfaceOperatorTerm attribute), 531

name (sfepy.terms.termsBasic.IntegrateSurfaceTerm attribute), 532

name (sfepy.terms.termsBasic.IntegrateVolumeOperatorTerm attribute), 532

name (sfepy.terms.termsBasic.IntegrateVolumeTerm attribute), 533

name (sfepy.terms.termsBasic.SumNodalValuesTerm attribute), 534

name (sfepy.terms.termsBasic.SurfaceMomentTerm attribute), 534

name (sfepy.terms.termsBasic.SurfaceTerm attribute), 534



534

name (sfepy.terms.termsBasic.VolumeSurfaceTerm attribute), 535

name (sfepy.terms.termsBasic.VolumeTerm attribute), 535

name (sfepy.terms.termsBiot.BiotETHTerm attribute), 536

name (sfepy.terms.termsBiot.BiotStressTerm attribute), 537

name (sfepy.terms.termsBiot.BiotTerm attribute), 538

name (sfepy.terms.termsBiot.BiotTHTerm attribute), 538

name (sfepy.terms.termsElectric.ElectricSourceTerm attribute), 539

name (sfepy.terms.termsLaplace.DiffusionCoupling attribute), 540

name (sfepy.terms.termsLaplace.DiffusionRTerm attribute), 540

name (sfepy.terms.termsLaplace.DiffusionTerm attribute), 541

name (sfepy.terms.termsLaplace.DiffusionVelocityTerm attribute), 542

name (sfepy.terms.termsLaplace.LaplaceTerm attribute), 542

name (sfepy.terms.termsLaplace.PermeabilityRTerm attribute), 543

name (sfepy.terms.termsLaplace.SurfaceFluxTerm attribute), 543

name (sfepy.terms.termsLinElasticity.CauchyStrainSTerm attribute), 544

name (sfepy.terms.termsLinElasticity.CauchyStrainTerm attribute), 545

name (sfepy.terms.termsLinElasticity.CauchyStressETHTerm attribute), 546

name (sfepy.terms.termsLinElasticity.CauchyStressTerm attribute), 547

name (sfepy.terms.termsLinElasticity.CauchyStressTHTerm attribute), 546

name (sfepy.terms.termsLinElasticity.LinearElasticETHTerm attribute), 548

name (sfepy.terms.termsLinElasticity.LinearElasticIsotropicTerm attribute), 548

name (sfepy.terms.termsLinElasticity.LinearElasticTerm attribute), 550

name (sfepy.terms.termsLinElasticity.LinearElasticTHTerm attribute), 549

name (sfepy.terms.termsLinElasticity.LinearPrestressTerm attribute), 550

name (sfepy.terms.termsLinElasticity.LinearStrainFiberTerm attribute), 551

name (sfepy.terms.termsLinElasticity.SDLinElasticTerm attribute), 552

name (sfepy.terms.termsNavierStokes.ConvectTerm attribute), 552

name (sfepy.terms.termsNavierStokes.DivGradTerm attribute), 553

name (sfepy.terms.termsNavierStokes.DivOperatorTerm attribute), 553

name (sfepy.terms.termsNavierStokes.DivTerm attribute), 554

name (sfepy.terms.termsNavierStokes.GradDivStabilizationTerm attribute), 555

name (sfepy.terms.termsNavierStokes.GradTerm attribute), 555

name (sfepy.terms.termsNavierStokes.LinearConvectTerm attribute), 556

name (sfepy.terms.termsNavierStokes.PSPGCStabilizationTerm attribute), 556

name (sfepy.terms.termsNavierStokes.PSPGPStabilizationTerm attribute), 557

name (sfepy.terms.termsNavierStokes.StokesTerm attribute), 559

name (sfepy.terms.termsNavierStokes.SUPGCStabilizationTerm attribute), 557

name (sfepy.terms.termsNavierStokes.SUPGPStabilizationTerm attribute), 558

name (sfepy.terms.termsPiezo.PiezoCouplingTerm attribute), 559

name (sfepy.terms.termsPoint.ConcentratedPointLoadTerm attribute), 560

name (sfepy.terms.termsPoint.LinearPointSpringTerm attribute), 561

name (sfepy.terms.termsSurface.ContactPlaneTerm attribute), 562

name (sfepy.terms.termsSurface.LinearTractionTerm attribute), 562

name (sfepy.terms.termsSurface.SDSurfaceNormalDotTerm attribute), 563

name (sfepy.terms.termsSurface.SurfaceNormalDotTerm attribute), 564

name (sfepy.terms.termsSurface.SurfaceJumpTerm attribute), 564

name (sfepy.terms.termsVolume.LinearVolumeForceTerm attribute), 565

Term\_to\_key() (in module sfepy.base.log), 339

NeoHookeanTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 575

NeoHookeanULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 580

NEUMeshIO (class in sfepy.fem.meshio), 386

new() (sfepy.terms.terms.Term static method), 516

new\_ulf\_iteration() (sfepy.fem.evaluate.BasicEvaluator method), 355

NewDiffusionTerm (class in sfepy.terms.terms\_new), 582

NewLinearElasticTerm (class in sfepy.terms.terms\_new), 582

NewMassScalarTerm (class in sfepy.terms.terms\_new), 582

NewMassTerm (class in sfepy.terms.terms\_new), 582

NewTerm (class in sfepy.terms.terms\_new), 583  
 Newton (class in sfepy.solvers.nls), 484  
 next() (sfepy.fem.variables.CloseNodesIterator method), 410  
 NodeDescription (class in sfepy.fem.poly\_spaces), 390  
 NonlinearSolver (class in sfepy.solvers.solvers), 491  
 NonPenetrationTerm (class in sfepy.terms.terms\_constraints), 565  
 NoOptionsDocs (class in build\_helpers), 318  
 NoPenetrationOperator (class in sfepy.fem.dof\_info), 348  
 norm() (in module sfepy.mesh.femlab), 454  
 norm() (sfepy.physics.radial\_mesh.RadialMesh method), 469  
 norm2() (in module sfepy.mesh.femlab), 454  
 norm\_l2\_along\_axis() (in module sfepy.linalg.utils), 444  
 normal (sfepy.fem.extmods.mappings.CMapping attribute), 421  
 NormalDirectionOperator (class in sfepy.fem.dof\_info), 348  
 normalize\_null\_space\_base() (sfepy.optimize.freeFormDef.DesignVariables method), 466  
 normalize\_time() (sfepy.solvers.ts.TimeStepper method), 491  
 normalize\_vectors() (in module sfepy.linalg.utils), 444  
 NSOFMinGradTerm (class in sfepy.terms.termsAdjointNavierStokes), 522  
 NSOFSurfMinDPressDiffTerm (class in sfepy.terms.termsAdjointNavierStokes), 522  
 NSOFSurfMinDPressTerm (class in sfepy.terms.termsAdjointNavierStokes), 522  
 numlist2str() (in module sfepy.mesh.meshutils), 465  
 numpydoc\_path() (sfepy.config.Config method), 325

## O

obj\_fun() (in module sfepy.optimize.shapeOptim), 467  
 obj\_fun() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase method), 467  
 obj\_fun\_grad() (in module sfepy.optimize.shapeOptim), 467  
 OnesDim (class in sfepy.homogenization.coefs\_base), 425  
 OneTypeList (class in sfepy.base.base), 327  
 op\_dv() (sfepy.terms.termsLinElasticity.SDLinearElasticTerm static method), 552  
 OptimizationSolver (class in sfepy.solvers.solvers), 491  
 ordered\_iteritems() (in module sfepy.base.base), 331  
 orient\_elements() (in module sfepy.fem.extmods.mesh), 422  
 Oseen (class in sfepy.solvers.oseen), 489  
 Output (class in sfepy.base.base), 327  
 output (sfepy.base.log\_plotter.LogPlotter attribute), 339  
 output\_vector() (sfepy.physics.radial\_mesh.RadialMesh method), 469  
 output\_vector() (sfepy.physics.radial\_mesh.RadialVector method), 469  
 output\_writable\_meshes() (in module sfepy.fem.meshio), 388  
 OutputFilter (class in runTests), 316

## P

package\_check() (in module build\_helpers), 318  
 parametrize() (sfepy.applications.application.Application method), 325  
 parse\_approx\_order() (in module sfepy.fem.fields\_base), 367  
 parse\_definition() (in module sfepy.fem.equations), 354  
 parse\_domain\_specific() (in module postproc), 315  
 parse\_group\_names() (in module postproc), 315  
 parse\_linearization() (in module extractor), 314  
 parse\_opacity() (in module postproc), 315  
 parse\_ranges() (in module postproc), 315  
 parse\_repeat() (in module genPerMesh), 319  
 parse\_resolution() (in module postproc), 315  
 parse\_subdomains() (in module postproc), 315  
 parse\_view() (in module postproc), 315  
 pause() (in module sfepy.base.base), 331  
 PDESolverApp (class in sfepy.applications.pde\_solver\_app), 326  
 Percentage (class in sfepy.base.progressbar), 341  
 percentage() (sfepy.base.progressbar.ProgressBar method), 341  
 PeriodicBC (class in sfepy.fem.conditions), 345  
 PermeabilityRTerm (class in sfepy.terms.termsLaplace), 542  
 permutations() (in module sfepy.linalg.utils), 444  
 permute\_in\_place() (in module sfepy.linalg.extmods.crcm), 444  
 PETScKrylovSolver (class in sfepy.solvers.ls), 481  
 PETScParallelKrylovSolver (class in sfepy.solvers.ls), 481  
 PhaseVelocity (class in sfepy.homogenization.coefs\_phononic), 429  
 phonon (module), 314  
 PhysicalQPs (class in sfepy.fem.mappings), 373  
 physicalsurface (class in sfepy.mesh.geom\_tools), 456  
 physicalvolume (class in sfepy.mesh.geom\_tools), 456  
 PiezoCouplingTerm (class in sfepy.terms.termsPiezo), 559  
 plot() (sfepy.physics.radial\_mesh.RadialMesh method), 469  
 plot() (sfepy.physics.radial\_mesh.RadialVector method), 469  
 plot\_and\_save() (in module plotPerfusionCoefs), 320  
 plot\_band\_gaps() (sfepy.homogenization.band\_gaps\_app.AcousticBandGap method), 422  
 plot\_condition\_numbers (module), 324  
 plot\_data() (sfepy.base.log.Log method), 338

- plot\_dispersion() (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp method), 429
- plot\_displacements() (in module sfepy.postprocess.domain\_specific), 471
- plot\_edges() (in module sfepy.postprocess.plot\_facets), 473
- plot\_eigs() (in module sfepy.homogenization.band\_gaps\_app), 422
- plot\_faces() (in module sfepy.postprocess.plot\_facets), 473
- plot\_gap() (in module sfepy.homogenization.band\_gaps\_app), 423
- plot\_gaps() (in module sfepy.homogenization.band\_gaps\_app), 423
- plot\_geometry() (in module sfepy.postprocess.plot\_facets), 473
- plot\_global\_dofs() (in module sfepy.postprocess.plot\_dofs), 473
- plot\_history() (in module plotPerfusionCoefs), 320
- plot\_local\_dofs() (in module sfepy.postprocess.plot\_dofs), 473
- plot\_log() (in module sfepy.base.log), 339
- plot\_logs() (in module sfepy.homogenization.band\_gaps\_app), 423
- plot\_matrix\_diff() (in module sfepy.base.plotutils), 340
- plot\_mesh() (in module sfepy.postprocess.plot\_dofs), 473
- plot\_nodes() (in module sfepy.postprocess.plot\_dofs), 473
- plot\_points() (in module sfepy.mechanics.contact\_planes), 445
- plot\_polygon() (in module sfepy.mechanics.contact\_planes), 445
- plot\_polys() (in module gen\_lobatto1d\_c), 323
- plot\_velocity() (in module sfepy.postprocess.domain\_specific), 471
- plot\_vlines() (sfepy.base.log.Log method), 338
- plot\_volume\_fractions() (in module plotPerfusionCoefs), 320
- plot\_warp\_scalar() (in module sfepy.postprocess.domain\_specific), 472
- plotPerfusionCoefs (module), 320
- point (class in sfepy.mesh.geom\_tools), 456
- points\_in\_simplex() (in module sfepy.linalg.geometry), 440
- PointsProbe (class in sfepy.fem.probes), 391
- PointTermBase (class in sfepy.terms.termsPoint), 561
- PolarizationAngles (class in sfepy.homogenization.coefs\_phononic), 429
- poll\_draw() (sfepy.base.log\_plotter.LogPlotter method), 339
- poll\_file() (sfepy.postprocess.sources.FileSource method), 473
- PolySpace (class in sfepy.fem.poly\_spaces), 390
- post\_process() (sfepy.homogenization.coefs\_phononic.SchurEVP method), 429
- post\_process() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 429
- postproc (module), 314
- postprocess() (in module probe), 316
- Potential (class in sfepy.physics.potentials), 468
- PotentialBase (class in sfepy.physics.potentials), 468
- prefix (sfepy.base.base.Output attribute), 328
- prepare\_cylindrical\_transform() (in module sfepy.mechanics.tensors), 451
- prepare\_matrices() (sfepy.homogenization.coefs\_phononic.SchurEVP method), 429
- prepare\_matrices() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 429
- prepare\_matrix() (in module sfepy.solvers.ts\_solvers), 493
- prepare\_remap() (in module sfepy.fem.utils), 410
- prepare\_save\_data() (in module sfepy.solvers.ts\_solvers), 493
- prepare\_translate() (in module sfepy.fem.utils), 410
- presolve() (sfepy.homogenization.coefs\_base.PressureEigenvalueProblem method), 425
- PressureEigenvalueProblem (class in sfepy.homogenization.coefs\_base), 425
- PressureRHSVector (class in sfepy.homogenization.coefs\_elastic), 426
- pretty() (sfepy.physics.radial\_mesh.RadialVector method), 469
- print\_array\_info() (in module sfepy.linalg.utils), 444
- print\_leaf() (in module sfepy.fem.parseReg), 389
- print\_matrix\_diff() (in module sfepy.base.plotutils), 340
- print\_names() (sfepy.base.base.Container method), 327
- print\_names() (sfepy.base.base.OneTypeList method), 327
- print\_op() (in module sfepy.fem.parseReg), 389
- print\_stack() (in module sfepy.fem.parseReg), 389
- print\_structs() (in module sfepy.base.base), 331
- print\_terms() (in module simple), 317
- print\_terms() (sfepy.fem.equations.Equations method), 353
- printinfo() (sfepy.mesh.geom\_tools.geometry method), 456
- printinfo() (sfepy.mesh.meshutils.mesh method), 463
- Probe (class in sfepy.fem.probes), 392
- probe (module), 315
- probe() (sfepy.fem.probes.Probe method), 392
- ProblemConf (class in sfepy.base.conf), 334
- ProblemDefinition (class in sfepy.fem.problemDef), 394
- process\_command() (sfepy.base.log\_plotter.LogPlotter method), 339
- process\_conf() (sfepy.solvers.eigen.LOBPCGEigenvalueSolver static method), 480
- process\_conf() (sfepy.solvers.eigen.PysparseEigenvalueSolver static method), 480



`process_conf()` (sfepy.solvers.eigen.ScipySGEigenvalueSolver static method), 480  
`process_conf()` (sfepy.solvers.ls.PETScKrylovSolver static method), 481  
`process_conf()` (sfepy.solvers.ls.PETScParallelKrylovSolver static method), 482  
`process_conf()` (sfepy.solvers.ls.PyAMGSolver static method), 482  
`process_conf()` (sfepy.solvers.ls.SchurComplement static method), 483  
`process_conf()` (sfepy.solvers.ls.SchurGeneralized static method), 483  
`process_conf()` (sfepy.solvers.ls.ScipyDirect static method), 483  
`process_conf()` (sfepy.solvers.ls.ScipyIterative static method), 484  
`process_conf()` (sfepy.solvers.nls.Newton static method), 486  
`process_conf()` (sfepy.solvers.nls.ScipyBroyden static method), 487  
`process_conf()` (sfepy.solvers.optimize.FMinSteepestDescent static method), 487  
`process_conf()` (sfepy.solvers.optimize.ScipyFMinSolver static method), 488  
`process_conf()` (sfepy.solvers.oseen.Oseen static method), 489  
`process_conf()` (sfepy.solvers.semismooth\_newton.SemismoothNewton static method), 490  
`process_conf()` (sfepy.solvers.solvers.Solver static method), 491  
`process_conf()` (sfepy.solvers.ts\_solvers.AdaptiveTimeSteppingSolver static method), 492  
`process_conf()` (sfepy.solvers.ts\_solvers.ExplicitTimeSteppingSolver static method), 492  
`process_conf()` (sfepy.solvers.ts\_solvers.SimpleTimeSteppingSolver static method), 492  
`process_options()` (sfepy.applications.pde\_solver\_app.PDESolverApp static method), 326  
`process_options()` (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp static method), 422  
`process_options()` (sfepy.homogenization.coefs\_base.MiniAppBase method), 425  
`process_options()` (sfepy.homogenization.coefs\_phononic.BandGaps method), 428  
`process_options()` (sfepy.homogenization.coefs\_phononic.ChristoffelAcousticTensor method), 428  
`process_options()` (sfepy.homogenization.coefs\_phononic.Eigenmomenta method), 429  
`process_options()` (sfepy.homogenization.coefs\_phononic.PhaseVelocity method), 429  
`process_options()` (sfepy.homogenization.coefs\_phononic.PolarizationAngles method), 429  
`process_options()` (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 429  
`process_options()` (sfepy.homogenization.engine.HomogenizationEngine static method), 432  
`process_options()` (sfepy.homogenization.homogen\_app.HomogenizationApp static method), 432  
`process_options()` (sfepy.physics.schroedinger\_app.SchroedingerApp static method), 470  
`process_options_pv()` (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp static method), 422  
`ProgressBar` (class in sfepy.base.progressbar), 341  
`progressbar()` (in module sfepy.base.progressbar), 342  
`ProgressBarWidget` (class in sfepy.base.progressbar), 341  
`ProgressBarWidgetHFill` (class in sfepy.base.progressbar), 342  
`ps` (sfepy.fem.extmods.mappings.CMapping attribute), 421  
`PSPGCStabilizationTerm` (class in sfepy.terms.termsNavierStokes), 556  
`PSPGPStabilizationTerm` (class in sfepy.terms.termsNavierStokes), 556  
`PyAMGSolver` (class in sfepy.solvers.ls), 482  
`PysparseEigenvalueSolver` (class in sfepy.solvers.eigen), 480  
`python_include()` (sfepy.config.Config method), 325  
`python_shell()` (in module sfepy.base.base), 331  
`python_version()` (sfepy.config.Config method), 325

## Q

`qp` (sfepy.fem.extmods.mappings.CMapping attribute), 421  
`QuadraturePoints` (class in sfepy.fem.quadratures), 401  
`Quantity` (class in sfepy.mechanics.units), 452

## R

`RadialHyperbolicMesh` (class in sfepy.physics.radial\_mesh), 468  
`RadialMesh` (class in sfepy.physics.radial\_mesh), 468  
`RadialVector` (class in sfepy.physics.radial\_mesh), 469  
`RayProbe` (class in sfepy.fem.probes), 393  
`RBIotCoef` (class in sfepy.homogenization.coefs\_elastic), 426  
`read()` (in module sfepy.linalg.extmods.crcm), 444  
`read()` (sfepy.fem.meshio.AbaqusMeshIO method), 383  
`read()` (sfepy.fem.meshio.ANSYSCDBMeshIO method), 383  
`read()` (sfepy.fem.meshio.AVSUCDMeshIO method), 383  
`read()` (sfepy.fem.meshio.BDFMeshIO method), 383  
`read()` (sfepy.fem.meshio.ComsolMeshIO method), 383  
`read()` (sfepy.fem.meshio.HDF5MeshIO method), 384  
`read()` (sfepy.fem.meshio.HypermeshAsciiMeshIO method), 384  
`read()` (sfepy.fem.meshio.MeditMeshIO method), 384  
`read()` (sfepy.fem.meshio.MEDMeshIO method), 384  
`read()` (sfepy.fem.meshio.Mesh3DMeshIO method), 385  
`read()` (sfepy.fem.meshio.MeshIO method), 386

- read() (sfepy.fem.meshio.NEUMeshIO method), 386
- read() (sfepy.fem.meshio.TetgenMeshIO method), 387
- read() (sfepy.fem.meshio.UserMeshIO method), 387
- read() (sfepy.fem.meshio.VTKMeshIO method), 387
- read\_array() (in module sfepy.base.ioutils), 337
- read\_bounding\_box() (sfepy.fem.meshio.ANSYSCDBMeshIO method), 383
- read\_bounding\_box() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_bounding\_box() (sfepy.fem.meshio.MeditMeshIO method), 384
- read\_bounding\_box() (sfepy.fem.meshio.MeshIO method), 386
- read\_bounding\_box() (sfepy.fem.meshio.TetgenMeshIO method), 387
- read\_bounding\_box() (sfepy.fem.meshio.VTKMeshIO method), 387
- read\_common() (sfepy.postprocess.sources.GenericFileSource method), 474
- read\_coors() (sfepy.fem.meshio.VTKMeshIO method), 387
- read\_data() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_data() (sfepy.fem.meshio.MeshIO method), 386
- read\_data() (sfepy.fem.meshio.VTKMeshIO method), 387
- read\_data\_header() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_dict\_hdf5() (in module sfepy.base.ioutils), 337
- read\_dimension() (sfepy.fem.meshio.AbaqusMeshIO method), 383
- read\_dimension() (sfepy.fem.meshio.ANSYSCDBMeshIO method), 383
- read\_dimension() (sfepy.fem.meshio.AVSUCDMeshIO method), 383
- read\_dimension() (sfepy.fem.meshio.BDFMeshIO method), 383
- read\_dimension() (sfepy.fem.meshio.HypermeshAsciiMeshIO method), 384
- read\_dimension() (sfepy.fem.meshio.MeditMeshIO method), 384
- read\_dimension() (sfepy.fem.meshio.Mesh3DMeshIO method), 385
- read\_dimension() (sfepy.fem.meshio.MeshIO method), 386
- read\_dimension() (sfepy.fem.meshio.NEUMeshIO method), 386
- read\_dimension() (sfepy.fem.meshio.TetgenMeshIO method), 387
- read\_dimension() (sfepy.fem.meshio.VTKMeshIO method), 388
- read\_dsg\_vars\_hdf5() (in module sfepy.optimize.freeFormDef), 467
- read\_header() (in module sfepy.fem.probes), 393
- read\_last\_step() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_last\_step() (sfepy.fem.meshio.MeshIO method), 386
- read\_list() (in module sfepy.base.ioutils), 337
- read\_log() (in module sfepy.base.log), 339
- read\_results() (in module sfepy.fem.probes), 393
- read\_sparse\_matrix\_hdf5() (in module sfepy.base.ioutils), 338
- read\_spline\_box\_hdf5() (in module sfepy.optimize.freeFormDef), 467
- read\_time\_history() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_time\_stepper() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_times() (sfepy.fem.meshio.HDF5MeshIO method), 384
- read\_times() (sfepy.fem.meshio.MeshIO method), 386
- read\_token() (in module sfepy.base.ioutils), 338
- read\_variables\_time\_history() (sfepy.fem.meshio.HDF5MeshIO method), 384
- readELE() (sfepy.mesh.meshutils.mesh method), 463
- readELE2() (sfepy.mesh.meshutils.mesh method), 463
- Reader (class in sfepy.base.reader), 342
- readGMV() (sfepy.mesh.meshutils.mesh method), 463
- readmsh() (sfepy.mesh.meshutils.mesh method), 463
- readmsh2() (sfepy.mesh.meshutils.mesh method), 463
- readNOD() (sfepy.mesh.meshutils.mesh method), 463
- readpmd() (sfepy.mesh.meshutils.bound method), 462
- readpmd() (sfepy.mesh.meshutils.mesh method), 463
- readstr2STR() (sfepy.mesh.meshutils.mesh method), 464
- readstr3STR() (sfepy.mesh.meshutils.mesh method), 464
- readxt2sSTR() (sfepy.mesh.meshutils.mesh method), 464
- recover\_bones() (in module sfepy.homogenization.recovery), 435
- recover\_micro\_hook() (in module sfepy.homogenization.recovery), 435
- recover\_parafflow() (in module sfepy.homogenization.recovery), 435
- recursive\_glob() (in module build\_helpers), 319
- reduce\_on\_datas() (sfepy.fem.materials.Material method), 376
- refine() (sfepy.fem.domain.Domain method), 350
- refine\_2\_3() (in module sfepy.fem.refine), 402
- refine\_2\_4() (in module sfepy.fem.refine), 402
- refine\_3\_4() (in module sfepy.fem.refine), 402
- refine\_3\_8() (in module sfepy.fem.refine), 403
- refine\_mesh() (in module sfepy.fem.utils), 410
- refine\_pars() (sfepy.fem.probes.Probe static method), 392
- refine\_points() (sfepy.fem.probes.PointsProbe method), 392
- refine\_points() (sfepy.fem.probes.Probe method), 392
- refine\_points() (sfepy.fem.probes.RayProbe method), 393
- refine\_reference() (in module sfepy.fem.refine), 403

- refine\_uniformly() (sfepy.fem.problemDef.ProblemDefinition method), 397  
 Region (class in sfepy.fem.region), 403  
 region\_leaf() (in module sfepy.fem.domain), 350  
 region\_op() (in module sfepy.fem.domain), 350  
 ReloadSource (class in sfepy.postprocess.viewer), 476  
 remap\_dict() (in module sfepy.base.base), 331  
 remove\_array() (sfepy.postprocess.dataset\_manager.DatasetManager method), 470  
 remove\_bcs() (sfepy.fem.problemDef.ProblemDefinition method), 398  
 remove\_extra\_dofs() (sfepy.fem.fields\_base.Field method), 365  
 remove\_extra\_dofs() (sfepy.fem.fields\_nodal.H1DiscontinuousField method), 368  
 remove\_files() (in module sfepy.base.ioutils), 338  
 remove\_name() (sfepy.base.base.Container method), 327  
 removecentralnodes() (sfepy.mesh.meshutils.mesh method), 464  
 rename\_array() (sfepy.postprocess.dataset\_manager.DatasetManager method), 470  
 render\_scene() (sfepy.postprocess.viewer.Viewer method), 479  
 renumber\_by\_boxes() (sfepy.optimize.freeFormDef.DesignVariables method), 466  
 renumber\_elements() (sfepy.mesh.meshutils.mesh method), 464  
 replace() (in module sfepy.fem.parseReg), 389  
 replace\_with\_region() (in module sfepy.fem.parseReg), 389  
 report() (in module test\_install), 320  
 report() (sfepy.base.testing.TestCommon static method), 343  
 report() (sfepy.fem.probes.CircleProbe method), 391  
 report() (sfepy.fem.probes.LineProbe method), 391  
 report() (sfepy.fem.probes.PointsProbe method), 392  
 report() (sfepy.fem.probes.Probe method), 392  
 report() (sfepy.fem.probes.RayProbe method), 393  
 reset() (sfepy.fem.materials.Material method), 376  
 reset() (sfepy.fem.materials.Materials method), 378  
 reset() (sfepy.fem.problemDef.ProblemDefinition method), 398  
 reset() (sfepy.fem.variables.Variable static method), 416  
 reset() (sfepy.postprocess.sources.FileSource method), 473  
 reset\_materials() (sfepy.fem.equations.Equations method), 353  
 reset\_refinement() (sfepy.fem.probes.Probe method), 392  
 reset\_regions() (sfepy.fem.domain.Domain method), 350  
 reset\_view() (sfepy.postprocess.viewer.Viewer method), 479  
 resolve\_chains() (in module sfepy.fem.dof\_info), 348  
 restore() (sfepy.applications.application.Application method), 325  
 ReverseBar (class in sfepy.base.progressbar), 342  
 rhs() (in module sfepy.fem.parseEq), 388  
 RigidOperator (class in sfepy.fem.dof\_info), 348  
 RotatingMarker (class in sfepy.base.progressbar), 342  
 rotation\_matrix2d() (in module sfepy.linalg.geometry), 440  
 run() (build\_helpers.Clean method), 318  
 run() (build\_helpers.DoxygenDocs method), 318  
 run() (build\_helpers.SphinxHTMLDocs method), 318  
 run() (build\_helpers.SphinxPDFDocs method), 318  
 run() (sfepy.base.testing.TestCommon method), 343  
 run\_test() (in module runTests), 316  
 running\_mean() (sfepy.physics.radial\_mesh.RadialVector method), 469  
 runTests (module), 316
- ## S
- save() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 425  
 save() (sfepy.homogenization.coefs\_base.TCorrectorsViaPressureEVP method), 426  
 save() (sfepy.homogenization.coefs\_phononic.SimpleEVP method), 429  
 save() (sfepy.mechanics.friction.DualMesh method), 445  
 save\_animation() (sfepy.postprocess.viewer.Viewer method), 479  
 save\_as\_mesh() (sfepy.fem.variables.FieldVariable method), 414  
 save\_axes() (sfepy.mechanics.friction.DualMesh method), 445  
 save\_basis (module), 324  
 save\_dict() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 326  
 save\_ebc() (sfepy.fem.problemDef.ProblemDefinition method), 398  
 save\_field\_meshes() (sfepy.fem.problemDef.ProblemDefinition method), 398  
 save\_image() (sfepy.postprocess.viewer.Viewer method), 479  
 save\_log() (sfepy.homogenization.coefs\_phononic.BandGaps static method), 428  
 save\_mappings() (sfepy.fem.fields\_base.Field method), 365  
 save\_only() (in module sfepy.applications.pde\_solver\_app), 326  
 save\_recovery\_region() (in module sfepy.homogenization.recovery), 435  
 save\_regions() (sfepy.fem.domain.Domain method), 350  
 save\_regions() (sfepy.fem.problemDef.ProblemDefinition method), 398  
 save\_regions\_as\_groups() (sfepy.fem.domain.Domain method), 350  
 save\_regions\_as\_groups() (sfepy.fem.problemDef.ProblemDefinition method), 398

- method), 398
- save\_results() (sfepy.physics.schroedinger\_app.SchroedingerApp method), 470
- save\_sparse\_txt() (in module sfepy.linalg.sparse), 441
- save\_state() (sfepy.fem.problemDef.ProblemDefinition method), 398
- save\_time\_history() (in module sfepy.postprocess.time\_history), 476
- ScalarDotGradIScalarTerm (class in sfepy.terms.terms\_dot), 569
- scalars\_elements2nodes() (sfepy.mesh.meshutils.mesh method), 464
- scale\_matrix() (in module sfepy.solvers.oseen), 489
- schroedinger (module), 317
- SchroedingerApp (class in sfepy.physics.schroedinger\_app), 470
- schur\_fun() (sfepy.solvers.ls.SchurComplement static method), 483
- SchurComplement (class in sfepy.solvers.ls), 482
- SchurEVP (class in sfepy.homogenization.coefs\_phononic), 429
- SchurGeneralized (class in sfepy.solvers.ls), 483
- ScipyBroyden (class in sfepy.solvers.nls), 487
- ScipyDirect (class in sfepy.solvers.ls), 483
- ScipyEigenvalueSolver (class in sfepy.solvers.eigen), 480
- ScipyFMinSolver (class in sfepy.solvers.optimize), 488
- ScipyIterative (class in sfepy.solvers.ls), 484
- ScipySGEigenvalueSolver (class in sfepy.solvers.eigen), 480
- SDConvectTerm (class in sfepy.terms.termsAdjointNavierStokes), 523
- SDDivGradTerm (class in sfepy.terms.termsAdjointNavierStokes), 524
- SDDivTerm (class in sfepy.terms.termsAdjointNavierStokes), 524
- SDDotVolumeTerm (class in sfepy.terms.termsAdjointNavierStokes), 525
- SDGradDivStabilizationTerm (class in sfepy.terms.termsAdjointNavierStokes), 525
- SDLinearElasticTerm (class in sfepy.terms.termsLinElasticity), 551
- SDPSPGCStabilizationTerm (class in sfepy.terms.termsAdjointNavierStokes), 526
- SDPSPGPStabilizationTerm (class in sfepy.terms.termsAdjointNavierStokes), 527
- SDSurfaceNormalDotTerm (class in sfepy.terms.termsSurface), 562
- SDSUPGCStabilizationTerm (class in sfepy.terms.termsAdjointNavierStokes), 527
- select\_bcs() (sfepy.fem.problemDef.ProblemDefinition method), 398
- select\_by\_names() (in module sfepy.base.base), 331
- select\_cells() (sfepy.fem.region.Region method), 405
- select\_cells\_of\_surface() (sfepy.fem.region.Region method), 405
- select\_materials() (sfepy.fem.problemDef.ProblemDefinition method), 398
- select\_variables() (sfepy.fem.problemDef.ProblemDefinition method), 398
- semideep\_copy() (sfepy.fem.materials.Materials method), 378
- SemismoothNewton (class in sfepy.solvers.semismooth\_newton), 490
- sensitivity() (sfepy.optimize.shapeOptim.ShapeOptimFlowCase method), 467
- separate() (sfepy.mesh.geom\_tools.surface method), 456
- set\_accuracy() (in module sfepy.fem.periodic), 389
- set\_all\_data() (sfepy.fem.materials.Material method), 376
- set\_arg\_types() (sfepy.terms.terms.Term method), 516
- set\_arg\_types() (sfepy.terms.terms\_dot.DotProductVolumeTerm method), 568
- set\_arg\_types() (sfepy.terms.terms\_dot.ScalarDotGradIScalarTerm method), 569
- set\_arg\_types() (sfepy.terms.terms\_dot.VectorDotGradScalarTerm method), 570
- set\_arg\_types() (sfepy.terms.termsAcoustic.SurfaceCoupleLayerTerm method), 519
- set\_arg\_types() (sfepy.terms.termsAcoustic.SurfaceLaplaceLayerTerm method), 520
- set\_arg\_types() (sfepy.terms.termsBiot.BiotTerm method), 538
- set\_arg\_types() (sfepy.terms.termsLaplace.DiffusionCoupling method), 540
- set\_arg\_types() (sfepy.terms.termsLaplace.DiffusionTerm method), 541
- set\_arg\_types() (sfepy.terms.termsLaplace.LaplaceTerm method), 542
- set\_arg\_types() (sfepy.terms.termsLinElasticity.LinearElasticTerm method), 550
- set\_arg\_types() (sfepy.terms.termsLinElasticity.LinearPrestressTerm method), 550
- set\_arg\_types() (sfepy.terms.termsNavierStokes.DivGradTerm method), 553
- set\_arg\_types() (sfepy.terms.termsNavierStokes.StokesTerm method), 559
- set\_arg\_types() (sfepy.terms.termsPiezo.PiezoCouplingTerm method), 560
- set\_arg\_types() (sfepy.terms.termsSurface.SurfaceNormalDotTerm method), 564
- set\_axes\_font\_size() (in module sfepy.base.plotutils), 340
- set\_bcs() (sfepy.fem.problemDef.ProblemDefinition method), 398
- set\_cells() (sfepy.fem.region.Region method), 406
- set\_constant() (sfepy.fem.variables.Variable method), 416
- set\_control\_points() (sfepy.mesh.splinebox.SplineBox method), 466
- set\_control\_points() (sfepy.optimize.freeFormDef.SplineBoxes method), 467

- `set_current_group()` (sfepy.fem.variables.FieldVariable method), 414
- `set_current_group()` (sfepy.terms.terms.CharacteristicFunction method), 513
- `set_current_group()` (sfepy.terms.terms.Term method), 517
- `set_current_group()` (sfepy.terms.terms.Terms method), 517
- `set_current_group()` (sfepy.terms.terms\_new.NewTerm method), 583
- `set_data()` (sfepy.fem.equations.Equations method), 353
- `set_data()` (sfepy.fem.materials.Material method), 376
- `set_data()` (sfepy.fem.variables.Variable method), 416
- `set_data()` (sfepy.fem.variables.Variables method), 417
- `set_data_from_qp()` (sfepy.fem.variables.FieldVariable method), 414
- `set_data_from_state()` (sfepy.fem.variables.Variables method), 417
- `set_data_from_variable()` (sfepy.fem.materials.Material method), 376
- `set_default()` (sfepy.base.base.Struct method), 328
- `set_defaults()` (in module sfepy.base.base), 331
- `set_dofs()` (sfepy.fem.fields\_hierarchic.H1HierarchicVolumeField method), 368
- `set_dofs()` (sfepy.fem.fields\_nodal.H1NodalMixin method), 369
- `set_equations()` (sfepy.fem.problemDef.ProblemDefinition method), 398
- `set_equations_instance()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_extra_args()` (sfepy.fem.functions.Function method), 370
- `set_extra_args()` (sfepy.fem.materials.Material method), 376
- `set_faces()` (sfepy.fem.region.Region method), 406
- `set_fields()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_filename()` (sfepy.postprocess.sources.GenericFileSource method), 474
- `set_filename()` (sfepy.postprocess.sources.GenericSequenceFileSource method), 474
- `set_filename()` (sfepy.postprocess.sources.VTKFileSource method), 474
- `set_filename()` (sfepy.postprocess.sources.VTKSequenceFileSource method), 475
- `set_float_format()` (sfepy.fem.meshio.MeshIO method), 386
- `set_from_data()` (sfepy.solvers.ts.TimeStepper method), 491
- `set_from_data()` (sfepy.solvers.ts.VariableTimeStepper method), 492
- `set_from_group()` (sfepy.fem.region.Region method), 406
- `set_from_mesh_vertices()` (sfepy.fem.variables.FieldVariable method), 414
- `set_from_other()` (sfepy.fem.variables.FieldVariable method), 414
- `set_from_ts()` (sfepy.solvers.ts.TimeStepper method), 491
- `set_from_ts()` (sfepy.solvers.ts.VariableTimeStepper method), 492
- `set_full()` (sfepy.fem.state.State method), 408
- `set_function()` (sfepy.fem.functions.Function method), 370
- `set_function()` (sfepy.fem.materials.Material method), 376
- `set_integral()` (sfepy.terms.terms.Term method), 517
- `set_linear()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_materials()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_matrix()` (sfepy.solvers.ls.PETScKrylovSolver method), 481
- `set_mesh_coors()` (in module sfepy.fem.fea), 361
- `set_mesh_coors()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_method()` (sfepy.solvers.nls.ScipyBroyden method), 487
- `set_method()` (sfepy.solvers.optimize.ScipyFMinSolver method), 488
- `set_n_digit_from_min_dt()` (sfepy.solvers.ts.VariableTimeStepper method), 492
- `set_npoint()` (sfepy.fem.probes.Probe method), 392
- `set_nonlin_states()` (in module sfepy.homogenization.utils), 436
- `set_options()` (sfepy.fem.probes.Probe method), 392
- `set_output()` (sfepy.base.base.Output method), 328
- `set_output_dir()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_output_prefix()` (sfepy.base.base.Output method), 328
- `set_parts()` (sfepy.fem.state.State method), 409
- `set_reduced()` (sfepy.fem.state.State method), 409
- `set_regions()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_section()` (in module gen\_term\_table), 324
- `set_solvers()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_solvers_instances()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_source_filename()` (sfepy.postprocess.viewer.Viewer method), 479
- `set_state_part()` (sfepy.fem.variables.Variables method), 417
- `set_step()` (sfepy.postprocess.sources.FileSource method), 474
- `set_step()` (sfepy.solvers.ts.TimeStepper method), 491
- `set_step()` (sfepy.solvers.ts.VariableTimeStepper method), 492



- `set_time_step()` (sfepy.solvers.ts.VariableTimeStepper method), 492
- `set_variables()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CoeffN static method), 424
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CoeffNN static method), 424
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CoeffOne static method), 424
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CoeffSym static method), 424
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CorrN static method), 425
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CorrNN static method), 425
- `set_variables_default()` (sfepy.homogenization.coefs\_base.CorrOne static method), 425
- `set_variables_from_state()` (sfepy.fem.equations.Equations method), 353
- `set_vertices()` (sfepy.fem.region.Region method), 406
- `SetStep` (class in sfepy.postprocess.viewer), 476
- `setup()` (in module gen\_term\_table), 324
- `setup()` (sfepy.base.conf.ProblemConf method), 335
- `setup()` (sfepy.fem.equations.Equations method), 353
- `setup()` (sfepy.solvers.oseen.StabilizationFunction method), 489
- `setup()` (sfepy.terms.terms.Term method), 517
- `setup()` (sfepy.terms.terms.Terms method), 517
- `setup_adof_conns()` (sfepy.fem.variables.FieldVariable method), 414
- `setup_adof_conns()` (sfepy.fem.variables.Variables method), 418
- `setup_args()` (sfepy.terms.terms.Term method), 517
- `setup_bases()` (sfepy.fem.variables.FieldVariable method), 415
- `setup_coors()` (sfepy.fem.fields\_base.Field method), 365
- `setup_default_output()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `setup_dof_conns()` (in module sfepy.fem.fields\_base), 367
- `setup_dof_conns()` (sfepy.fem.fields\_base.Field method), 365
- `setup_dof_conns()` (sfepy.fem.fields\_base.SurfaceField method), 365
- `setup_dof_conns()` (sfepy.fem.fields\_base.VolumeField method), 365
- `setup_dof_info()` (sfepy.fem.variables.Variables method), 418
- `setup_dtype()` (sfepy.fem.variables.Variables method), 418
- `setup_equations()` (sfepy.homogenization.coefs\_base.TCorrections static method), 426
- `setup_extra_data()` (in module sfepy.fem.fields\_base), 367
- `setup_extra_data()` (sfepy.fem.fields\_base.SurfaceField method), 365
- `setup_extra_data()` (sfepy.fem.fields\_base.VolumeField method), 365
- `setup_face_indices()` (sfepy.fem.region.Region method), 406
- `setup_facets()` (sfepy.fem.domain.Domain method), 350
- `setup_formal_args()` (sfepy.terms.terms.Term method), 517
- `setup_interfaces()` (sfepy.fem.facets.Facets method), 359
- `setup_groups()` (sfepy.fem.domain.Domain method), 350
- `setup_hooks()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `setup_ic()` (sfepy.fem.problemDef.ProblemDefinition method), 399
- `setup_initial_conditions()` (sfepy.fem.equations.Equations method), 353
- `setup_initial_conditions()` (sfepy.fem.variables.FieldVariable method), 415
- `setup_initial_conditions()` (sfepy.fem.variables.Variables method), 418
- `setup_integration()` (sfepy.terms.terms.Term method), 517
- `setup_lcbc_operators()` (sfepy.fem.variables.Variables method), 418
- `setup_mat_id()` (sfepy.postprocess.sources.FileSource method), 474
- `setup_mirror_connectivity()` (sfepy.fem.fe\_surface.FESurface method), 360
- `setup_mirror_region()` (sfepy.fem.region.Region method), 406
- `setup_neighbours()` (sfepy.fem.facets.Facets method), 359
- `setup_notification()` (sfepy.postprocess.sources.FileSource method), 474
- `setup_options()` (sfepy.applications.application.Application method), 325
- `setup_options()` (sfepy.applications.pde\_solver\_app.PDESolverApp method), 326
- `setup_options()` (sfepy.homogenization.band\_gaps\_app.AcousticBandGapsApp method), 422
- `setup_options()` (sfepy.homogenization.engine.HomogenizationEngine method), 432
- `setup_options()` (sfepy.homogenization.homogen\_app.HomogenizationApp method), 432
- `setup_options()` (sfepy.physics.schroedinger\_app.SchroedingerApp method), 470
- `setup_ordering()` (sfepy.fem.variables.Variables method), 418
- `setup_von_neumann_EVP` (in module sfepy.fem.geometry\_element), 370

setup\_output() (sfepy.fem.problemDef.ProblemDefinition method), 399  
 setup\_output() (sfepy.homogenization.coefs\_base.CorrMiniApp method), 425  
 setup\_output() (sfepy.physics.schroedinger\_app.SchroedingerApp method), 470  
 setup\_output\_info() (sfepy.applications.pde\_solver\_app.PDESolverApp method), 326  
 setup\_point\_data() (sfepy.fem.fea.Approximation method), 361  
 setup\_surface\_data() (sfepy.fem.fea.Approximation method), 361  
 setup\_unique() (sfepy.fem.facets.Facets method), 359  
 sfepy.applications.application (module), 325  
 sfepy.applications.pde\_solver\_app (module), 326  
 sfepy.base.base (module), 326  
 sfepy.base.compat (module), 332  
 sfepy.base.conf (module), 334  
 sfepy.base.getch (module), 336  
 sfepy.base.goptions (module), 336  
 sfepy.base.ioutils (module), 337  
 sfepy.base.log (module), 338  
 sfepy.base.log\_plotter (module), 339  
 sfepy.base.parse\_conf (module), 340  
 sfepy.base.plotutils (module), 340  
 sfepy.base.progressbar (module), 340  
 sfepy.base.reader (module), 342  
 sfepy.base.testing (module), 343  
 sfepy.config (module), 325  
 sfepy.fem.conditions (module), 343  
 sfepy.fem.dof\_info (module), 345  
 sfepy.fem.domain (module), 349  
 sfepy.fem.equations (module), 350  
 sfepy.fem.evaluate (module), 355  
 sfepy.fem.evaluate\_variable (module), 358  
 sfepy.fem.extmods.\_fmfield (module), 418  
 sfepy.fem.extmods.assemble (module), 418  
 sfepy.fem.extmods.bases (module), 418  
 sfepy.fem.extmods.lobatto\_bases (module), 420  
 sfepy.fem.extmods.mappings (module), 421  
 sfepy.fem.extmods.mesh (module), 421  
 sfepy.fem.facets (module), 358  
 sfepy.fem.fe\_surface (module), 360  
 sfepy.fem.fea (module), 360  
 sfepy.fem.fields\_base (module), 362  
 sfepy.fem.fields\_hierarchic (module), 367  
 sfepy.fem.fields\_nodal (module), 368  
 sfepy.fem.functions (module), 370  
 sfepy.fem.geometry\_element (module), 370  
 sfepy.fem.global\_interp (module), 371  
 sfepy.fem.history (module), 371  
 sfepy.fem.integrals (module), 372  
 sfepy.fem.linearizer (module), 372  
 sfepy.fem.mappings (module), 373  
 sfepy.fem.mass\_operator (module), 375  
 sfepy.fem.materials (module), 375  
 sfepy.fem.mesh (module), 378  
 sfepy.fem.meshio (module), 383  
 sfepy.fem.parseEq (module), 388  
 sfepy.fem.parseReg (module), 388  
 sfepy.fem.PDESolverApp (module), 389  
 sfepy.fem.poly\_spaces (module), 389  
 sfepy.fem.probes (module), 391  
 sfepy.fem.problemDef (module), 394  
 sfepy.fem.projections (module), 400  
 sfepy.fem.quadratures (module), 401  
 sfepy.fem.refine (module), 402  
 sfepy.fem.region (module), 403  
 sfepy.fem.simplex\_cubature (module), 406  
 sfepy.fem.state (module), 407  
 sfepy.fem.utils (module), 409  
 sfepy.fem.variables (module), 410  
 sfepy.homogenization.band\_gaps\_app (module), 422  
 sfepy.homogenization.coefficients (module), 423  
 sfepy.homogenization.coefs\_base (module), 424  
 sfepy.homogenization.coefs\_elastic (module), 426  
 sfepy.homogenization.coefs\_perfusion (module), 427  
 sfepy.homogenization.coefs\_phononic (module), 427  
 sfepy.homogenization.convolution (module), 431  
 sfepy.homogenization.engine (module), 432  
 sfepy.homogenization.homogen\_app (module), 432  
 sfepy.homogenization.micmac (module), 433  
 sfepy.homogenization.recovery (module), 433  
 sfepy.homogenization.utils (module), 435  
 sfepy.interactive.session (module), 437  
 sfepy.linalg.eigen (module), 437  
 sfepy.linalg.extmods.crcm (module), 444  
 sfepy.linalg.geometry (module), 438  
 sfepy.linalg.sparse (module), 440  
 sfepy.linalg.utils (module), 442  
 sfepy.mechanics.contact\_planes (module), 445  
 sfepy.mechanics.elastic\_constants (module), 445  
 sfepy.mechanics.friction (module), 445  
 sfepy.mechanics.matcoefs (module), 446  
 sfepy.mechanics.membranes (module), 448  
 sfepy.mechanics.tensors (module), 450  
 sfepy.mechanics.units (module), 452  
 sfepy.mesh.femlab (module), 454  
 sfepy.mesh.geom\_tools (module), 455  
 sfepy.mesh.mesh\_generators (module), 457  
 sfepy.mesh.mesh\_tools (module), 460  
 sfepy.mesh.meshutils (module), 461  
 sfepy.mesh.splinebox (module), 465  
 sfepy.optimize.freeFormDef (module), 466  
 sfepy.optimize.shapeOptim (module), 467  
 sfepy.physics.energy (module), 467  
 sfepy.physics.potentials (module), 468  
 sfepy.physics.radial\_mesh (module), 468

`sfepy.physics.schroedinger_app` (module), 470  
`sfepy.postprocess.dataset_manager` (module), 470  
`sfepy.postprocess.domain_specific` (module), 471  
`sfepy.postprocess.plot_dofs` (module), 473  
`sfepy.postprocess.plot_facets` (module), 473  
`sfepy.postprocess.sources` (module), 473  
`sfepy.postprocess.time_history` (module), 475  
`sfepy.postprocess.utils` (module), 476  
`sfepy.postprocess.viewer` (module), 476  
`sfepy.solvers.eigen` (module), 480  
`sfepy.solvers.ls` (module), 481  
`sfepy.solvers.nls` (module), 484  
`sfepy.solvers.optimize` (module), 487  
`sfepy.solvers.oseen` (module), 489  
`sfepy.solvers.petsc_worker` (module), 489  
`sfepy.solvers.semismooth_newton` (module), 490  
`sfepy.solvers.solvers` (module), 490  
`sfepy.solvers.ts` (module), 491  
`sfepy.solvers.ts_solvers` (module), 492  
`sfepy.terms.extmods.terms` (module), 583  
`sfepy.terms.terms` (module), 513  
`sfepy.terms.terms_constraints` (module), 565  
`sfepy.terms.terms_dot` (module), 566  
`sfepy.terms.terms_fibres` (module), 570  
`sfepy.terms.terms_hyperelastic_base` (module), 571  
`sfepy.terms.terms_hyperelastic_tl` (module), 573  
`sfepy.terms.terms_hyperelastic_ul` (module), 577  
`sfepy.terms.terms_membrane` (module), 581  
`sfepy.terms.terms_new` (module), 582  
`sfepy.terms.terms_th` (module), 583  
`sfepy.terms.termsAcoustic` (module), 518  
`sfepy.terms.termsAdjointNavierStokes` (module), 520  
`sfepy.terms.termsBasic` (module), 530  
`sfepy.terms.termsBiot` (module), 536  
`sfepy.terms.termsElectric` (module), 539  
`sfepy.terms.termsLaplace` (module), 539  
`sfepy.terms.termsLinElasticity` (module), 544  
`sfepy.terms.termsNavierStokes` (module), 552  
`sfepy.terms.termsPiezo` (module), 559  
`sfepy.terms.termsPoint` (module), 560  
`sfepy.terms.termsSurface` (module), 561  
`sfepy.terms.termsVolume` (module), 564  
`sfepy.terms.utils` (module), 583  
`sfepy.version` (module), 325  
`shape` (`sfepy.fem.extmods.mappings.CMapping` attribute), 421  
`shape` (`sfepy.physics.radial_mesh.ExplicitRadialMesh` attribute), 468  
`ShapeDim` (class in `sfepy.homogenization.coefs_base`), 426  
`ShapeDimDim` (class in `sfepy.homogenization.coefs_base`), 426  
`ShapeOptimFlowCase` (class in `sfepy.optimize.shapeOptim`), 467  
`shaper` (module), 317  
`show_array()` (in module `plotPerfusionCoefs`), 320  
`show_authors` (module), 324  
`show_scalarBars()` (`sfepy.postprocess.viewer.Viewer` method), 479  
`simple` (module), 317  
`SimpleEVP` (class in `sfepy.homogenization.coefs_phononic`), 429  
`SimpleTimeSteppingSolver` (class in `sfepy.solvers.ts_solvers`), 492  
`simplify()` (`sfepy.mesh.meshutils.bound` method), 462  
`size` (`sfepy.physics.radial_mesh.ExplicitRadialMesh` attribute), 468  
`size` (`sfepy.physics.radial_mesh.RadialHyperbolicMesh` attribute), 468  
`skip_read_line()` (in module `sfepy.base.ioutils`), 338  
`slice()` (`sfepy.physics.radial_mesh.ExplicitRadialMesh` method), 468  
`slice()` (`sfepy.physics.radial_mesh.RadialVector` method), 469  
`smooth_f()` (`sfepy.terms.termsSurface.ContactPlaneTerm` static method), 562  
`smooth_mesh()` (in module `sfepy.mesh.mesh_tools`), 460  
`solve()` (in module `sfepy.solvers.petsc_worker`), 489  
`solve()` (`sfepy.fem.problemDef.ProblemDefinition` method), 400  
`solve_adjoint()` (in module `shaper`), 317  
`solve_direct()` (in module `shaper`), 317  
`solve_eigen_problem()` (`sfepy.physics.schroedinger_app.SchroedingerApp` method), 470  
`solve_generic_direct()` (in module `shaper`), 317  
`solve_navier_stokes()` (in module `shaper`), 317  
`solve_optimize()` (in module `shaper`), 317  
`solve_pde()` (in module `sfepy.applications.pde_solver_app`), 326  
`solve_pressure_eigenproblem()` (`sfepy.homogenization.coefs_base.PressureEigenvalueProblem` method), 425  
`solve_problem_for_design()` (in module `sfepy.optimize.shapeOptim`), 467  
`solve_step()` (`sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver` method), 492  
`solve_step()` (`sfepy.solvers.ts_solvers.ExplicitTimeSteppingSolver` method), 492  
`solve_step()` (`sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver` method), 492  
`solve_stokes()` (in module `shaper`), 317  
`Solver` (class in `sfepy.solvers.solvers`), 491  
`sort()` (`sfepy.fem.conditions.Conditions` method), 344  
`sort_and_orient()` (`sfepy.fem.facets.Facets` method), 359  
`sort_by_dependency()` (in module `sfepy.fem.region`), 406  
`sort_by_mat_id()` (in module `sfepy.fem.meshio`), 388  
`sort_by_mat_id2()` (in module `sfepy.fem.meshio`), 388  
`sortnodes()` (`sfepy.mesh.meshutils.mesh` method), 464



- `sparse_merge()` (sfepy.physics.radial\_mesh.RadialVector static method), 470
- `sparse_vector()` (sfepy.physics.radial\_mesh.ExplicitRadialMesh static method), 468
- `spause()` (in module sfepy.base.base), 331
- `spcol()` (sfepy.mesh.splinebox.SplineBox static method), 466
- `SphinxHTMLDocs` (class in build\_helpers), 318
- `SphinxPDFDocs` (class in build\_helpers), 318
- `SplineBox` (class in sfepy.mesh.splinebox), 465
- `SplineBox` (class in sfepy.optimize.freeFormDef), 466
- `SplineBoxes` (class in sfepy.optimize.freeFormDef), 466
- `split_by_mat_id()` (in module sfepy.fem.meshio), 388
- `split_chunks()` (in module sfepy.homogenization.coefs\_phononic), 431
- `split_complex_args()` (in module sfepy.terms.terms), 517
- `split_on()` (in module edit\_identifiers), 321
- `split_range()` (in module sfepy.linalg.utils), 444
- `splitlines()` (sfepy.mesh.geom\_tools.geometry method), 456
- `spsorted()` (sfepy.mesh.splinebox.SplineBox static method), 466
- `spy()` (in module sfepy.base.plotutils), 340
- `spy_and_show()` (in module sfepy.base.plotutils), 340
- `StabilizationFunction` (class in sfepy.solvers.oseen), 489
- `standalone_setup()` (sfepy.terms.terms.Term method), 517
- `standard_call()` (in module sfepy.solvers.eigen), 481
- `standard_call()` (in module sfepy.solvers.ls), 484
- `start()` (runTests.OutputFilter method), 316
- `start()` (sfepy.base.progressbar.ProgressBar method), 341
- `State` (class in sfepy.fem.state), 407
- `state_to_output()` (sfepy.fem.equations.Equations method), 353
- `state_to_output()` (sfepy.fem.variables.Variables method), 418
- `StationarySolver` (class in sfepy.solvers.ts\_solvers), 492
- `step` (sfepy.postprocess.viewer.SetStep attribute), 476
- `stiffness_from_lame()` (in module sfepy.mechanics.matcoefs), 447
- `stiffness_from_lame_mixed()` (in module sfepy.mechanics.matcoefs), 448
- `stiffness_from_youngpoisson()` (in module sfepy.mechanics.matcoefs), 448
- `stiffness_from_youngpoisson_mixed()` (in module sfepy.mechanics.matcoefs), 448
- `StokesTerm` (class in sfepy.terms.termsNavierStokes), 558
- `stop()` (runTests.OutputFilter method), 316
- `str()` (sfepy.mesh.meshutils.bound method), 462
- `str_all()` (sfepy.base.base.Struct method), 328
- `str_class()` (sfepy.base.base.Struct method), 328
- `stress_function()` (sfepy.terms.terms\_fibres.FibresActiveTLTerm static method), 571
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm method), 573
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm method), 573
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 574
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm method), 575
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm method), 576
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm method), 578
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 578
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm method), 580
- `stress_function()` (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm method), 580
- `StressTransform` (class in sfepy.mechanics.tensors), 450
- `string` (sfepy.fem.meshio.HDF5MeshIO attribute), 384
- `strip_state_vector()` (sfepy.fem.equations.Equations method), 353
- `strip_state_vector()` (sfepy.fem.variables.Variables method), 418
- `Struct` (class in sfepy.base.base), 328
- `sub_e()` (sfepy.fem.region.Region method), 406
- `sub_n()` (sfepy.fem.region.Region method), 406
- `substitute_continuous()` (in module evalForms), 322
- `SurfaceNormalDotTerm` (class in sfepy.terms.termsSurface), 563
- `suggest_name()` (sfepy.fem.poly\_spaces.PolySpace static method), 391
- `SumNodalValuesTerm` (class in sfepy.terms.termsBasic), 533
- `SUPGCAdjStabilizationTerm` (class in sfepy.terms.termsAdjointNavierStokes), 528
- `SUPGCStabilizationTerm` (class in sfepy.terms.termsNavierStokes), 557
- `SUPGPAdj1StabilizationTerm` (class in sfepy.terms.termsAdjointNavierStokes), 529
- `SUPGPAdj2StabilizationTerm` (class in sfepy.terms.termsAdjointNavierStokes), 529
- `SUPGPStabilizationTerm` (class in sfepy.terms.termsNavierStokes), 557
- `surface` (class in sfepy.mesh.geom\_tools), 456
- `surface_components()` (in module findSurf), 319
- `surface_faces()` (sfepy.fem.domain.Domain method), 350
- `surface_graph()` (in module findSurf), 319
- `SurfaceApproximation` (class in sfepy.fem.fea), 361
- `SurfaceCoupleLayerTerm` (class in sfepy.terms.termsAcoustic), 518
- `SurfaceField` (class in sfepy.fem.fields\_base), 365
- `SurfaceFluxTerm` (class in sfepy.terms.termsLaplace), 543

- SurfaceInterpolant (class in sfepy.fem.fea), 361
- SurfaceJumpTerm (class in sfepy.terms.termsSurface), 564
- SurfaceLaplaceLayerTerm (class in sfepy.terms.termsAcoustic), 519
- SurfaceMapping (class in sfepy.fem.mappings), 373
- SurfaceMomentTerm (class in sfepy.terms.termsBasic), 534
- SurfaceTerm (class in sfepy.terms.termsBasic), 534
- SurfaceTractionTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 576
- switch\_cells() (sfepy.fem.region.Region method), 406
- sym2dim() (in module sfepy.mechanics.tensors), 451
- sym\_tri\_eigen() (in module sfepy.linalg.eigen), 438
- symbolic (sfepy.terms.termsLaplace.DiffusionTerm attribute), 541
- symbolic (sfepy.terms.termsLaplace.LaplaceTerm attribute), 542
- sync\_module\_docs (module), 324
- system() (sfepy.config.Config method), 325
- ## T
- tan\_mod\_function() (sfepy.terms.terms\_fibres.FibresActiveTLTerm static method), 571
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkActiveTLTerm method), 573
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPenaltyTLTerm method), 573
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_tl.MooneyRivlinTLTerm method), 575
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_tl.NeoHookeanTLTerm method), 576
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPenaltyULTerm method), 578
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_ul.MooneyRivlinULTerm method), 580
- tan\_mod\_function() (sfepy.terms.terms\_hyperelastic\_ul.NeoHookeanULTerm method), 580
- tan\_mod\_u\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 574
- tan\_mod\_u\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 578
- TCorrectorsPressureViaPressureEVP (class in sfepy.homogenization.coefs\_elastic), 426
- TCorrectorsRSViaPressureEVP (class in sfepy.homogenization.coefs\_elastic), 427
- TCorrectorsViaPressureEVP (class in sfepy.homogenization.coefs\_base), 426
- tensor\_plane\_stress() (sfepy.mechanics.matcoefs.TransformToPlane method), 446
- Term (class in sfepy.terms.terms), 514
- term\_ns\_asm\_convect() (in module sfepy.terms.extmods.terms), 586
- term\_ns\_asm\_div\_grad() (in module sfepy.terms.extmods.terms), 586
- terminate() (sfepy.base.log.Log method), 338
- terminate() (sfepy.base.log\_plotter.LogPlotter method), 339
- TermParse (class in sfepy.fem.parseEq), 388
- Terms (class in sfepy.terms.terms), 517
- test\_install (module), 320
- test\_permutations() (sfepy.fem.variables.CloseNodesIterator method), 410
- test\_terms() (in module sfepy.optimize.shapeOptim), 467
- TestCommon (class in sfepy.base.testing), 343
- tetgen\_path() (sfepy.config.Config method), 325
- TetgenMeshIO (class in sfepy.fem.meshio), 386
- THTerm (class in sfepy.terms.terms\_th), 583
- tilled\_mesh1d() (in module sfepy.mesh.mesh\_generators), 460
- time\_update() (sfepy.fem.equations.Equations method), 354
- time\_update() (sfepy.fem.materials.Material method), 376
- time\_update() (sfepy.fem.materials.Materials method), 378
- time\_update() (sfepy.fem.problemDef.ProblemDefinition method), 400
- time\_update() (sfepy.fem.variables.FieldVariable method), 415
- time\_update() (sfepy.fem.variables.Variable method), 416
- time\_update() (sfepy.fem.variables.Variables method), 418
- time\_update() (sfepy.terms.terms.Term method), 517
- time\_update\_materials() (sfepy.fem.equations.Equations method), 354
- TimeStepper (class in sfepy.solvers.ts), 491
- TimeSteppingSolver (class in sfepy.solvers.solvers), 491
- TLMembraneTerm (class in sfepy.terms.terms\_membrane), 581
- to\_array() (sfepy.linalg.utils.MatrixAction method), 442
- to\_dict() (sfepy.base.base.Struct method), 328
- to\_file() (sfepy.physics.radial\_mesh.RadialVector method), 470
- to\_file\_hdf5() (sfepy.homogenization.coefficients.Coefficients method), 423
- to\_file\_latex() (sfepy.homogenization.coefficients.Coefficients method), 423
- to\_file\_txt (sfepy.homogenization.coefs\_phononic.AcousticMassTensor attribute), 427
- to\_file\_txt (sfepy.homogenization.coefs\_phononic.AppliedLoadTensor attribute), 428
- to\_file\_txt() (sfepy.homogenization.coefficients.Coefficients method), 423
- to\_file\_txt() (sfepy.homogenization.coefs\_phononic.BandGaps static method), 428
- to\_file\_txt() (sfepy.homogenization.coefs\_phononic.DensityVolumeInfo

- static method), 429
  - to\_latex() (sfepy.homogenization.coefficients.Coefficients method), 423
  - to\_list() (in module gen\_term\_table), 324
  - to\_poly\_file() (sfepy.mesh.geom\_tools.geometry method), 456
  - to\_stack() (in module sfepy.fem.parseReg), 389
  - transform\_bar\_to\_space\_coors() (in module sfepy.linalg.geometry), 440
  - transform\_conditions() (in module sfepy.base.conf), 335
  - transform\_coors() (sfepy.fem.mesh.Mesh method), 381
  - transform\_data() (in module sfepy.mechanics.tensors), 451
  - transform\_ebcs() (in module sfepy.base.conf), 335
  - transform\_epbcs() (in module sfepy.base.conf), 335
  - transform\_fields() (in module sfepy.base.conf), 335
  - transform\_functions() (in module sfepy.base.conf), 335
  - transform\_ics() (in module sfepy.base.conf), 336
  - transform\_input() (sfepy.base.conf.ProblemConf method), 335
  - transform\_input\_trivial() (sfepy.base.conf.ProblemConf method), 335
  - transform\_integrals() (in module sfepy.base.conf), 336
  - transform\_lcbcs() (in module sfepy.base.conf), 336
  - transform\_materials() (in module sfepy.base.conf), 336
  - transform\_plot\_data() (in module sfepy.homogenization.band\_gaps\_app), 423
  - transform\_regions() (in module sfepy.base.conf), 336
  - transform\_solvers() (in module sfepy.base.conf), 336
  - transform\_to\_i\_struct\_1() (in module sfepy.base.conf), 336
  - transform\_to\_struct\_01() (in module sfepy.base.conf), 336
  - transform\_to\_struct\_1() (in module sfepy.base.conf), 336
  - transform\_to\_struct\_10() (in module sfepy.base.conf), 336
  - transform\_variables() (in module sfepy.base.conf), 336
  - TransformToPlane (class in sfepy.mechanics.matcoefs), 446
  - treat\_pbc() (sfepy.fem.dof\_info.LCBCOperator method), 347
  - tri3d() (in module sfepy.mesh.femlab), 454
  - triangulate() (in module sfepy.mesh.femlab), 454
  - triangulate2() (in module sfepy.mesh.femlab), 454
  - triangulate\_old() (in module sfepy.mesh.femlab), 454
  - try\_imports() (in module sfepy.base.base), 332
  - try\_set\_defaults() (in module sfepy.homogenization.band\_gaps\_app), 423
  - TSTimes (class in sfepy.homogenization.coefs\_base), 426
  - tuple\_to\_conf() (in module sfepy.base.conf), 336
  - typeset() (in module gen\_term\_table), 324
  - typeset\_term\_syntax() (in module gen\_term\_table), 324
  - typeset\_term\_table() (in module gen\_term\_table), 324
  - typeset\_to\_indent() (in module gen\_term\_table), 324
- ## U
- Umfpack (class in sfepy.solvers.ls), 484
  - unique() (in module sfepy.base.compat), 333
  - unique\_rows() (in module sfepy.linalg.utils), 444
  - Unit (class in sfepy.mechanics.units), 453
  - update() (sfepy.base.base.Container method), 327
  - update() (sfepy.base.base.Struct method), 328
  - update() (sfepy.base.progressbar.Bar method), 340
  - update() (sfepy.base.progressbar.ETA method), 340
  - update() (sfepy.base.progressbar.FileTransferSpeed method), 341
  - update() (sfepy.base.progressbar.MyBar method), 341
  - update() (sfepy.base.progressbar.Percentage method), 341
  - update() (sfepy.base.progressbar.ProgressBar method), 341
  - update() (sfepy.base.progressbar.ProgressBarWidget method), 342
  - update() (sfepy.base.progressbar.ProgressBarWidgetHFill method), 342
  - update() (sfepy.base.progressbar.ReverseBar method), 342
  - update() (sfepy.base.progressbar.RotatingMarker method), 342
  - update() (sfepy.fem.dof\_info.DofInfo method), 346
  - update() (sfepy.postprocess.dataset\_manager.DatasetManager method), 470
  - update\_data() (sfepy.fem.materials.Material method), 377
  - update\_dict\_recursively() (in module sfepy.base.base), 332
  - update\_equations() (sfepy.fem.problemDef.ProblemDefinition method), 400
  - update\_expression() (sfepy.physics.potentials.CompoundPotential method), 468
  - update\_expression() (sfepy.terms.terms.Terms method), 517
  - update\_groups() (sfepy.fem.region.Region method), 406
  - update\_materials() (sfepy.fem.problemDef.ProblemDefinition method), 400
  - update\_mesh() (in module sfepy.optimize.shapeOptim), 467
  - update\_shape() (sfepy.fem.region.Region method), 406
  - update\_special\_constant\_data() (sfepy.fem.materials.Material method), 377
  - update\_special\_data() (sfepy.fem.materials.Material method), 377
  - update\_time\_stepper() (sfepy.fem.problemDef.ProblemDefinition method), 400
  - update\_vertices() (sfepy.fem.region.Region method), 406
  - us2cw() (in module edit\_identifiers), 322
  - us2mc() (in module edit\_identifiers), 322
  - use\_method\_with\_name() (in module sfepy.base.base), 332
  - user\_options (build\_helpers.NoOptionsDocs attribute), 318

UserMeshIO (class in sfepy.fem.meshio), 387

## V

val() (sfepy.fem.variables.FieldVariable method), 415

val\_qp() (sfepy.fem.variables.FieldVariable method), 415

validate (sfepy.base.goptions.ValidatedDict attribute), 336

validate() (sfepy.base.conf.ProblemConf method), 335

validate\_bool() (in module sfepy.base.goptions), 336

ValidatedDict (class in sfepy.base.goptions), 336

validator() (sfepy.base.goptions.ValidatedDict method), 336

values() (sfepy.base.goptions.ValidatedDict method), 336

Variable (class in sfepy.fem.variables), 415

Variables (class in sfepy.fem.variables), 416

VariableTimeStepper (class in sfepy.solvers.ts), 492

vec() (in module sfepy.mesh.femlab), 454

vector\_chunk\_generator() (in module sfepy.terms.terms), 518

VectorDotGradScalarTerm (class in sfepy.terms.terms\_dot), 569

vectors\_elements2nodes() (sfepy.mesh.meshutils.mesh method), 464

verify\_correctors() (sfepy.homogenization.coefs\_base.TCorrectorVMPressure method), 426

view\_file() (in module postproc), 315

Viewer (class in sfepy.postprocess.viewer), 476

ViewerGUI (class in sfepy.postprocess.viewer), 479

visit\_stack() (in module sfepy.fem.parseReg), 389

Volume (class in sfepy.homogenization.homogen\_app), 433

volume (class in sfepy.mesh.geom\_tools), 456

volume (sfepy.fem.extmods.mappings.CMapping attribute), 421

VolumeField (class in sfepy.fem.fields\_base), 365

VolumeFractions (class in sfepy.homogenization.coefs\_base), 426

VolumeMapping (class in sfepy.fem.mappings), 373

VolumeSurfaceTerm (class in sfepy.terms.termsBasic), 534

VolumeTerm (class in sfepy.terms.termsBasic), 535

VolumeTLTerm (class in sfepy.terms.terms\_hyperelastic\_tl), 577

VolumeULTerm (class in sfepy.terms.terms\_hyperelastic\_ul), 580

VTKFileSource (class in sfepy.postprocess.sources), 474

VTKMeshIO (class in sfepy.fem.meshio), 387

VTKSequenceFileSource (class in sfepy.postprocess.sources), 474

## W

wandering\_element() (in module sfepy.fem.simplex\_cubature), 407

weak\_dp\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 574

weak\_dp\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 578

weak\_function() (sfepy.terms.terms\_hyperelastic\_tl.BulkPressureTLTerm method), 574

weak\_function() (sfepy.terms.terms\_hyperelastic\_tl.HyperElasticTLBase method), 575

weak\_function() (sfepy.terms.terms\_hyperelastic\_ul.BulkPressureULTerm method), 578

weak\_function() (sfepy.terms.terms\_hyperelastic\_ul.HyperElasticULBase method), 579

weak\_function() (sfepy.terms.terms\_membrane.TLMembraneTerm static method), 581

wrap\_function() (in module sfepy.solvers.optimize), 488

wrap\_run\_tests() (in module runTests), 316

write() (runTests.OutputFilter method), 316

write() (sfepy.fem.mesh.Mesh method), 381

write() (sfepy.fem.meshio.AbaqusMeshIO method), 383

write() (sfepy.fem.meshio.ANSYSCDBMeshIO method), 383

write() (sfepy.fem.meshio.AVSUCDMeshIO method), 383

write() (sfepy.fem.meshio.BDFMeshIO method), 383

write() (sfepy.fem.meshio.ComsolMeshIO method), 383

write() (sfepy.fem.meshio.HDF5MeshIO method), 384

write() (sfepy.fem.meshio.HypermeshAsciiMeshIO method), 384

write() (sfepy.fem.meshio.MeditMeshIO method), 384

write() (sfepy.fem.meshio.MeshIO method), 386

write() (sfepy.fem.meshio.NEUMeshIO method), 386

write() (sfepy.fem.meshio.TetgenMeshIO method), 387

write() (sfepy.fem.meshio.UserMeshIO method), 387

write() (sfepy.fem.meshio.VTKMeshIO method), 388

write\_bb() (in module sfepy.fem.meshio), 388

write\_dict\_hdf5() (in module sfepy.base.ioutils), 338

write\_femlab() (in module sfepy.mesh.femlab), 455

write\_results() (in module sfepy.fem.probes), 393

write\_sparse\_matrix\_hdf5() (in module sfepy.base.ioutils), 338

write\_vtk() (sfepy.mesh.splinebox.SplineBox method), 466

writeBC() (sfepy.mesh.meshutils.mesh method), 464

writeELE() (sfepy.mesh.meshutils.mesh method), 464

writemsh() (sfepy.mesh.meshutils.mesh method), 464

writemsh2() (sfepy.mesh.meshutils.mesh method), 464

writeNOD() (sfepy.mesh.meshutils.mesh method), 464

writepmd() (sfepy.mesh.meshutils.bound method), 462

writepmd() (sfepy.mesh.meshutils.mesh method), 464

writeregions() (sfepy.mesh.meshutils.mesh method), 464

writescalars() (sfepy.mesh.meshutils.mesh method), 464

writescalarspos() (sfepy.mesh.meshutils.mesh method), 464

writescalarspos2() (sfepy.mesh.meshutils.mesh method),  
[464](#)

writescalarspos3() (sfepy.mesh.meshutils.mesh method),  
[465](#)

writestresspos() (sfepy.mesh.meshutils.mesh method),  
[465](#)

writeSV() (sfepy.mesh.meshutils.bound method), [462](#)

writevectorspos() (sfepy.mesh.meshutils.mesh method),  
[465](#)

writevectorspos3() (sfepy.mesh.meshutils.mesh method),  
[465](#)

writexda() (sfepy.mesh.meshutils.mesh method), [465](#)

## X

xfail() (sfepy.base.testing.TestCommon static method),  
[343](#)

## Z

zero\_dofs() (sfepy.fem.conditions.Conditions method),  
[344](#)

zero\_dofs() (sfepy.fem.conditions.EssentialBC method),  
[344](#)