# CONTENTS

## 1 Documentation

1.1 Introduction ......................................................... 3
1.2 Installation ......................................................... 3
   1.2.1 Supported Platforms ......................................... 3
   1.2.2 Installing SfePy ............................................... 4
   1.2.3 Using SfePy Docker Images ................................... 4
   1.2.4 Installing SfePy from Sources .............................. 4
   1.2.5 Testing Installation ........................................... 7
   1.2.6 Debugging ..................................................... 8
   1.2.7 Using IPython ................................................. 8
   1.2.8 Notes on Multi-platform Python Distributions ............ 8
1.3 Tutorial ............................................................ 9
   1.3.1 Basic SfePy Usage ............................................ 9
   1.3.2 Basic Notions ................................................ 10
   1.3.3 Running a Simulation ........................................ 11
   1.3.4 Example Problem Description File .......................... 12
   1.3.5 Interactive Example: Linear Elasticity ................... 14
1.4 User’s Guide ....................................................... 21
   1.4.1 Running a Simulation ........................................ 21
   1.4.2 Visualization of Results .................................... 22
   1.4.3 Problem Description File ................................... 24
   1.4.4 Building Equations in SfePy ............................... 41
   1.4.5 Term Evaluation .............................................. 41
   1.4.6 Solution Postprocessing .................................... 42
   1.4.7 Probing ....................................................... 44
   1.4.8 Postprocessing filters ....................................... 44
   1.4.9 Solvers ....................................................... 44
   1.4.10 Solving Problems in Parallel .............................. 47
   1.4.11 Isogeometric Analysis ..................................... 48
1.5 Examples .......................................................... 49
   1.5.1 Primer ......................................................... 49
   1.5.2 Using Salome with SfePy .................................... 74
   1.5.3 Preprocessing: FreeCAD/OpenSCAD + Gmsh ................ 77
   1.5.4 Material Identification ..................................... 87
   1.5.5 Mesh parametrization ....................................... 90
   1.5.6 Examples ..................................................... 94
   1.5.7 Example Applications ...................................... 103
1.6 Useful Code Snippets and FAQ ................................... 103
   1.6.1 Miscellaneous ............................................... 103
   1.6.2 Mesh-Related Tasks ......................................... 105
• genindex
• modindex
• search
1.1 Introduction

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

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

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

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

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

1.2 Installation

1.2.1 Supported Platforms

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

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

It is only matter of taste to use either native OS Python installation, *pip*, or any other suitable distribution. On all supported platforms we could recommend multi-platform scientific Python distributions *Anaconda* as easy-to-use, stable and up-to-date Python distribution with all the required dependencies (including pre-built *sfepy* package). See also *Notes on Multi-platform Python Distributions* for further details.

On different platforms the following options can be recommended:

- **Linux**: *Anaconda*, OS native installation, if available, or *pip*.
- **macOS**: *Anaconda*.
- **Windows**: free versions of commercial scientific Python distributions *Anaconda* or *Enthought Deployment Manager*. In addition a completely free open-source portable distribution *WinPython* can be used.

1.2.2 Installing SfePy

The released versions of SfePy can be installed as follows.

- **Using pip**:

  ```
  pip install sfepy
  ```

- **Using Anaconda**: install *sfepy* from *conda-forge* channel:

  ```
  conda install -c conda-forge sfepy
  ```

  See *Notes on Multi-platform Python Distributions* for additional notes.

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

1.2.3 Using SfePy Docker Images

Besides the classical installation we also provide official Docker images with ready-to-run *Anaconda* and *SfePy* installation.

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

Currently available all-in-one image is:

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

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

1.2.4 Installing SfePy from Sources

The latest stable release can be obtained from the *download* page. Otherwise, download the development version of the code from *SfePy* git repository:

```
git clone git://github.com/sfepy/sfepy.git
```
git tag -l

to see the available releases - the release tags have form release_<year>..<int>.
See the download page for additional download options.

Requirements

Installation prerequisites, required to build SfePy:

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

Python packages required for using SfePy:

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

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

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

Other dependencies/suggestions:

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

```
python setup.py doxygendocs
```

• Mesh generation tools use pexpect and gmsh or tetgen.

• IPython is recommended over the regular Python shell to fluently follow some parts of primer/tutorial (see Using IPython).

• MUMPS library for using MUMPS linear direct solver (real and complex arithmetic, parallel factorization)

## Compilation of C Extension Modules

In the SfePy top-level directory:

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

2. For in-place use, compile the extension modules:

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

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

## Installation

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

• system-wide (may require root privileges):

```
pip install .
```

• local:

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

• development (editable install):

```
pip install -e .
```

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

If all went well, proceed with Testing Installation.
1.2.5 Testing Installation

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

Running Automated Test Suite

The test suite is based on pytest. Install it by:

```
pip install pytest
```

or:

```
conda install pytest
```

when working in Anaconda. If SfePy was installed, it can be tested using the command:

```
sfePy-test
```

that accepts all of the pytest options, for example:

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

The tests output directory can also be specified:

```
sfePy-test --output-dir=output-tests
```

In general, the tests can be run using:

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

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

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

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

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

See pytest usage instructions for other options and usage patterns.

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

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

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

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

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

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

Then re-run your code and report the output to the SfePy mailing list.

1.2.7 Using IPython

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

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

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

1. Create a new SfePy profile:
   ```bash
   ipython profile create sfepy
   ```

2. Open the ~/.ipython/profile_sfepy/ipython_config.py file in a text editor and add/edit after the `c = get_config()` line:
   ```python
   exec_lines = [
       'import numpy as nm',
       'import matplotlib as mpl',
       'mpl.use("WXAgg")',
       
       # Add your preferred SfePy customization here...
       
       ]
   
   c.InteractiveShellApp.exec_lines = exec_lines
   c.TerminalIPythonApp.gui = 'wx'
   c.TerminalInteractiveShell.colors = 'Linux' # NoColor, Linux, or LightBG
   ```

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

3. Run the customized IPython shell:
   ```bash
   ipython --profile=sfepy
   ```

1.2.8 Notes on Multi-platform Python Distributions

Anaconda

We highly recommend this scientific-oriented Python distribution.

(Currently regularly tested by developers on SfePy releases with Python 3.6 64-bit on Ubuntu 16.04 LTS, Windows 8.1+ and macOS 10.12+.)
Download appropriate Anaconda Python 3.x installer package and follow install instructions. We recommend to choose user-level install option (no admin privileges required).

Anaconda can be used for:

1. installing the latest release of SfePy directly from the conda-forge channel (see sfepy-feedstock). In this case, follow the instructions in Installing SfePy.
   
   Installing/upgrading SfePy from the conda-forge channel can also be achieved by adding conda-forge to your channels with:
   
   ```bash
   conda config --add channels conda-forge
   ```
   
   Once the conda-forge channel has been enabled, SfePy can be installed with:
   
   ```bash
   conda install sfepy
   ```
   
   It is possible to list all of the versions of SfePy available on your platform with:
   
   ```bash
   conda search sfepy --channel conda-forge
   ```
   
2. installing the SfePy dependencies only - then proceed with the Installing SfePy from Sources instructions.
   
   In this case, install the missing/required packages using built-in conda package manager:
   
   ```bash
   conda install wxpython
   ```
   
   See conda help for further information.

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

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

or try:

```bash
conda update --all
```

**Compilation of C Extension Modules on Windows**

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

**1.3 Tutorial**

**1.3.1 Basic SfePy Usage**

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

1. Black-box Partial Differential Equation (PDE) solver,
2. Python package to build custom applications involving solving PDEs by the Finite Element Method (FEM).
This tutorial focuses on the first way and introduces the basic concepts and nomenclature used in the following parts of the documentation. Check also the Primer which focuses on a particular problem in detail.

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

Invoking SfePy from the Command Line

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

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

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

Using SfePy Interactively

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

We recommend to use the IPython interactive shell for the best fluent user experience. You can customize your IPython startup profile as described in Using IPython.

1.3.2 Basic Notions

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

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

Below we show:

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

But let us begin with a slight detour...

Sneak Peek: What is Going on Under the Hood

1. A command (usually sfepy-run as in this tutorial) or a script reads in an input file.

2. Following the contents of the input file, a Problem instance is created – this is the input file coming to life. Let us call the instance problem.

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

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

Now let us continue with running a simulation.

### 1.3.3 Running a Simulation

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

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

### Postprocessing the Results

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

- The following interactive 3D window should display:

  ![3D visualization](image)

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

1.3.4 Example Problem Description File

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

The problem at hand is the following:

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

(1.1)

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

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

\[ \int_{\Omega} c \nabla T \cdot \nabla s = 0, \quad \forall s \in V_0. \]

(1.2)

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

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

```python
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.

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

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

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

```python
regions = {
    'Omega' : 'all', # or 'cells of group 6'
    'Gamma_Left' : ('vertices in (x < 0.00001)', 'facet'),
    'Gamma_Right' : ('vertices in (x > 0.099999)', 'facet'),
}
```
Regions assign names to various parts of the finite element mesh. The region names can later be referred to, for example when specifying portions of the mesh to apply boundary conditions to. Regions can be specified in a variety of ways, including by element or by node. Here, ‘Omega’ is the elemental domain over which the PDE is solved and ‘Gamma_Left’ and ‘Gamma_Right’ define surfaces upon which the boundary conditions will be applied.

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

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

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

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

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

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

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

```python
ebcs = {
    't1': ('Gamma_Left', {'t.0': 2.0}),
    't2', ('Gamma_Right', {'t.0': -2.0}),
}
```

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

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

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

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

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

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

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

where $\nabla u \approx G u$. 

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

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

```python
equations = {
    'Temperature' : """\text{dw}_\text{laplace}.2.\text{Omega}( \text{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.

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

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

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

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

That’s it! Now it is possible to proceed as described in Invoking SfePy from the Command Line.

### 1.3.5 Interactive Example: Linear Elasticity

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

We wish to solve the following linear elasticity problem:

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

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

In the weak form the equation (1.3) is

\[
\int_\Omega D_{ijkl}e_{kl}(u)e_{ij}(v) + \int_\Omega f_i v_i = 0, \quad (1.4)
\]
where \( v \) is the test function, and both \( u, v \) belong to a suitable function space.

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

The whole example summarized in a script is available below in *Complete Example as a Script*. In the *SfePy* top-level directory run

```
ipython
```

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

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

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

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

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

Define the regions – the whole domain \( \Omega \), where the solution is sought, and \( \Gamma_1, \Gamma_2 \), where the boundary conditions will be applied. As the domain is rectangular, we first get a bounding box to get correct bounds for selecting the boundary edges.

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

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

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

Using the field `fu`, we can define both the unknown variable \( u \) and the test variable \( v \).

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

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

```plaintext
In [14]: from sfepy.mechanics.matcoefs import stiffness_from_lame
```

(continues on next page)
In [15]: m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
In [16]: f = Material('f', val=[[0.02], [0.01]])

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

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

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

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

The equations have to be completed by boundary conditions. Let us clamp the left edge \( \Gamma_1 \), and shift the right edge \( \Gamma_2 \) in the \( x \) direction a bit, depending on the \( y \) coordinate.

In [23]: from sfepy.discrete.conditions import Conditions, EssentialBC
In [24]: fix_u = EssentialBC('fix_u', gamma1, {'u.all': 0.0})
In [25]: def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):
   ...:     val = shift * coors[:,1]**2
   ...:     return val
In [26]: bc_fun = Function('shift_u_fun', shift_u_fun, extra_args={'shift': 0.01})
In [27]: shift_u = EssentialBC('shift_u', gamma2, {'u.0': bc_fun})

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

In [28]: from sfepy.base.base import IndexedStruct
In [29]: from sfepy.solvers.ls import ScipyDirect
In [30]: from sfepy.solvers.nls import Newton
In [31]: ls = ScipyDirect({})
In [32]: nls_status = IndexedStruct()
In [33]: nls = Newton({}, lin_solver=ls, status=nls_status)

Now we are ready to create a Problem instance.

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

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

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

And view them
Finally, we set the boundary conditions and the top-level solver, solve the problem, save and view the results. For stationary problems, the top-level solver needs not to be a time-stepping solver - when a nonlinear solver is set instead, the default 'ts.stationary' time-stepping solver is created automatically.

```
In [39]: pb.set_bcs(ebcs=Conditions([fix_u, shift_u]))
In [40]: pb.set_solver(nls)
In [41]: status = IndexedStruct()
In [42]: variables = pb.solve(status=status)
In [43]: print('Nonlinear solver status:\n', nls_status)
In [44]: print('Stationary solver status:\n', status)
In [45]: pb.save_state('linear_elasticity.vtk', variables)
```

This

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

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

```
sfepy-view linear_elasticity.vtk -2 -f u:u:0 1:v:0
```

And the result is:
Complete Example as a Script

The source code: `linear_elastic_interactive.py`

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

```python
#!/usr/bin/env python

""
Linear elasticity example using the imperative API.
""
from argparse import ArgumentParser
import numpy as nm

import sys
sys.path.append('.

from sfepy.base.base import IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Function,
                           Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
from sfepy.conditions import Conditions, EssentialBC
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.mechanics.matcoefs import stiffness_from_lame

def shift_u_fun(ts, coors, bc=None, problem=None, shift=0.0):
    ""
    Define a displacement depending on the y coordinate.
    ""
    val = shift * coors[:,1]**2
    return val

def main():
    from sfepy import data_dir

    parser = ArgumentParser()
    parser.add_argument('--version', action='version', version='%(prog)s')
    options = parser.parse_args()

    mesh = Mesh.from_file(data_dir + '/meshes/2d/rectangle_tri.mesh')
    domain = FEDomain('domain', mesh)

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

(continues on next page)
field = Field.from_args('fu', nm.float64, 'vector', omega, approx_order=2)

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

m = Material('m', D=stiffness_from_lame(dim=2, lam=1.0, mu=1.0))
f = Material('f', val=[[0.02], [0.01]])

integral = Integral('i', order=3)
t1 = Term.new('dw_lin_elastic(m.D, v, u)', integral, omega, m=m, v=v, u=u)
t2 = Term.new('dw_volume_lvf(f.val, v)', integral, omega, f=f, v=v)
eq = Equation('balance', t1 + t2)
eqs = Equations([eq])

fix_u = EssentialBC('fix_u', gamma1, {u.all: 0.0})

bc_fun = Function('shift_u_fun', shift_u_fun,
extra_args={'shift': 0.01})
shift_u = EssentialBC('shift_u', gamma2, {'u.0': bc_fun})

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

pb = Problem('elasticity', equations=eqs)
pb.save_regions_as_groups('regions')
pb.set_bcs(ebcs=Conditions([fix_u, shift_u]))
pb.set_solver(nls)

status = IndexedStruct()
variables = pb.solve(status=status)

print('Nonlinear solver status:
', nls_status)
print('Stationary solver status:
', status)
pb.save_state('linear_elasticity.vtk', variables)

if __name__ == '__main__':
    main()
1.4 User’s Guide

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

1.4.1 Running a Simulation

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

Basic Usage

- \texttt{sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py}
  - Creates \texttt{cylinder.vtk}
- \texttt{sfepy-run sfepy/examples/navier_stokes/stokes.py}
  - Creates \texttt{channels_symm944t.vtk}

Running Tests

The tests are based on \texttt{pytest} and can be run using:

\texttt{python -c "import sfepy; sfepy.test()}"

See Testing Installation for additional information.

Computations and Examples

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

\texttt{python sfepy/examples/large_deformation/compare_elastic_materials.py}

Common Tasks

- Run a simulation:
  \texttt{sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py}
  \texttt{sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py -o some_results # -> produces some_results.vtk}

- Print available terms:
  \texttt{sfepy-run --list=terms}

- Run a simulation and also save Dirichlet boundary conditions:
  \texttt{sfepy-run --save-ebc sfepy/examples/diffusion/poisson_short_syntax.py # -> produces an additional .vtk file with BC visualization}
• Use a restart file to continue an interrupted simulation:
  
  – **Warning:** This feature is preliminary and does not support terms with internal state.
  
  – Run:
    
    ```bash
    sfepy-run sfepy/examples/large_deformation/balloon.py --save-restart=-1
    ```
    
    and break the computation after a while (hit Ctrl-C). The mode `--save-restart=-1` is currently the only supported mode. It saves a restart file for each time step, and only the last computed time step restart file is kept.

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

  – Restart the simulation by:
    
    ```bash
    sfepy-run sfepy/examples/large_deformation/balloon.py --load-restart=unit_ball.restart-04.h5
    ```
    
    The simulation should continue from the next time step. Verify that by running:
    
    ```bash
    sfepy-run sfepy/examples/large_deformation/balloon.py
    ```
    
    and compare the residuals printed in the corresponding time steps.

### 1.4.2 Visualization of Results

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

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

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

can be viewed using:

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

produces:
Using:

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

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

- `plot:0`: field $p$, 5 isosurfaces, mesh edges switched on, the surface opacity set to 0.2;
- `plot:1`: vector field $u$ streamlines, the surface opacity set to 0.2;
The actual camera position can be printed by pressing the ‘c’ key. The above figures use the following values:

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

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

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

### 1.4.3 Problem Description File

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

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

FE Mesh

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

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

Example:

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

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

The following geometry elements are supported:
Note the orientation of the vertices matters, the figure displays the correct orientation when interpreted in a right-handed coordinate system.

**Regions**

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

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

<table>
<thead>
<tr>
<th>topological entity</th>
<th>dimension</th>
<th>co-dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex</td>
<td>0</td>
<td>( D )</td>
</tr>
<tr>
<td>edge</td>
<td>1</td>
<td>( D - 1 )</td>
</tr>
<tr>
<td>face</td>
<td>2</td>
<td>( D - 2 )</td>
</tr>
<tr>
<td>facet</td>
<td>( D - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>cell</td>
<td>( D )</td>
<td>0</td>
</tr>
</tbody>
</table>

If \( D = 2 \), faces are not defined and facets are edges. If \( D = 3 \), facets are faces.

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

- **cell, facet, face, edge, vertex** - entities of higher dimension are not included.
- **cell_only, facet_only, face_only, edge_only, vertex_only** - only the specified entities are included, other entities are empty sets, so that set-like operators still work, see below.
• The cell kind is the most general and should be used with cell terms. It is also the default if the kind is not specified in region definition.
• The facet kind (same as edge in 2D and face in 3D) is to be used with boundary (facet integral) terms.
• The vertex (same as vertex_only) kind can be used with point-wise defined terms (e.g. point loads).

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

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

### Topological Entity Selection

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

<sup>2</sup> Only if mesh format supports reading boundary condition vertices as vertex sets.
<sup>3</sup> `<expr>` is a logical expression like `(y <= 0.1) & (x < 0.2)`. In 2D use `x, y`, in 3D use `x, y` and `z`. `&` stands for logical and, `|` stands for logical or.
<sup>4</sup> `<function>` is a function with signature `fun(coors, domain=None)`, where `coors` are coordinates of mesh vertices.
<sup>5</sup> `<efunction>` is a function with signature `fun(coors, domain=None)`, where `coors` are coordinates of mesh cell centroids.
topological entity selection footnotes

<table>
<thead>
<tr>
<th>set-like operator</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+v</td>
<td>vertex union</td>
</tr>
<tr>
<td>+e</td>
<td>edge union</td>
</tr>
<tr>
<td>+f</td>
<td>face union</td>
</tr>
<tr>
<td>+s</td>
<td>facet union</td>
</tr>
<tr>
<td>+c</td>
<td>cell union</td>
</tr>
<tr>
<td>-v</td>
<td>vertex difference</td>
</tr>
<tr>
<td>-e</td>
<td>edge difference</td>
</tr>
<tr>
<td>-f</td>
<td>face difference</td>
</tr>
<tr>
<td>-s</td>
<td>facet difference</td>
</tr>
<tr>
<td>-c</td>
<td>cell difference</td>
</tr>
<tr>
<td>*v</td>
<td>vertex intersection</td>
</tr>
<tr>
<td>*e</td>
<td>edge intersection</td>
</tr>
<tr>
<td>*f</td>
<td>face intersection</td>
</tr>
<tr>
<td>*s</td>
<td>facet intersection</td>
</tr>
<tr>
<td>*c</td>
<td>cell intersection</td>
</tr>
</tbody>
</table>

Region Definition Syntax

Regions are defined by the following Python dictionary:

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

or:

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

Example definitions:

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

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

```python
import numpy as np

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

(continues on next page)
\[ r = \sqrt{x^2 + y^2 + z^2} \]
\[ \text{flag} = \text{where}((r < 0.1))[0] \]

return flag

The function needs to be registered in \textit{Functions}:

\begin{verbatim}
functions = {
    'get_ball' : (get_ball,),
}
\end{verbatim}

The mirror region can be defined explicitly as:

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

\section*{Fields}

Fields correspond to FE spaces:

\begin{verbatim}
fields = {
    <name> : (<data_type>, <shape>, <region_name>, <approx_order>,
               [<space>, <poly_space_base>])
}
\end{verbatim}

where

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

\textbf{Example}: scalar P1 elements in 2D on a region Omega:

\begin{verbatim}
fields = {
    'temperature' : ('real', 1, 'Omega', 1),
}
\end{verbatim}

The following approximation orders can be used:

- simplex elements: \texttt{1, 2, 1B, 2B}
- tensor product elements: \texttt{0, 1, 1B}

Optional bubble function enrichment is marked by \texttt{B}.
The implemented combinations of spaces and bases are listed below, the space column corresponds to <space>, the basis column to <poly_space_base> and region type to the field region type.

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

**Variables**

Variables use the FE approximation given by the specified field:

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

where

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

**Example:**

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

**Integrals**

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

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

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

Example:

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

**Essential Boundary Conditions and Constraints**

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

**Dirichlet Boundary Conditions**

The Dirichlet, or essential, boundary conditions apply in a given region given by its name, and, optionally, in selected times. The times can be given either using a list of tuples $(t0, t1)$ making the condition active for $t0 <= t < t1$, 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:

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

Example:

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

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

```python
import numpy as nm
def rotate_yz(ts, coor, **kwargs):
    from sfepy.linalg import rotation_matrix2d
```
vec = coor[:,1:3] - centre
angle = 10.0 * ts.step
mtx = rotation_matrix2d(angle)
vec_rotated = nm.dot(vec, mtx)
displacement = vec_rotated - vec
return displacement

The function needs to be registered in *Functions*:

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

## Periodic Boundary Conditions

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

Periodic boundary conditions:

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

**Example:**

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

## Linear Combination Boundary Conditions

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

Available LCBC kinds:

- **'rigid'** - in linear elasticity problems, a region moves as a rigid body;
- **'no_penetration'** - in flow problems, the velocity vector is constrained to the plane tangent to the surface;
- **'normal_direction'** - the velocity vector is constrained to the normal direction;
- **'edge_direction'** - the velocity vector is constrained to the mesh edge direction;
• 'integral_mean_value' - all DOFs in a region are summed to a single new DOF;
• 'shifted_periodic' - generalized periodic BCs that work with two different variables and can have a non-zero mutual shift;
• 'match_dofs' - tie DOFs of two fields.

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

Linear combination boundary conditions:

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

**Initial Conditions**

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

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

**Example:**

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

**Materials**

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

**Example:**

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

**Example:** different material parameters in regions ‘Yc’, ‘Ym’:

```
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
dim = 3
materials = {
    'mat' : ({'D' : {
```
Ym': stiffness_from_youngpoisson(dim, 7.0e9, 0.4),
'Yc': stiffness_from_youngpoisson(dim, 70.0e9, 0.2)"
},
}

### Defining Material Parameters by Functions

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

The shape and type of data returned in the ‘special’ mode can be arbitrary (depending on the term used). On the other hand, in the ‘qp’ mode all the data have to be numpy float64 arrays with shape \((n_coor, n_row, n_col)\), 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 \((n_coor, 1, 1)\). The shape is determined by each term.

**Example**:

```python
def get_pars(ts, coors, mode='qp', **kwargs):
    if mode == 'qp':
        val = coors[:, 0]
        val.shape = (coors.shape[0], 1, 1)
        return {'x_coors': val}

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

The function needs to be registered in `Functions`:

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

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

### Equations and Terms

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

**Examples**

- Laplace equation, named integral:
  ```python
equations = {
    'Temperature': """\text{dw\_laplace\_i\_Omega(\ coef\_val, s, t ) = 0}"""
}
```

- Laplace equation, simplified integral given by order:
equations = {
    'Temperature': "dw_laplace.2.Omega( coef.val, s, t ) = 0"
}

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

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

• Navier-Stokes equations:

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

Configuring Solvers

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

Linear and nonlinear solver:

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

Solver selection:

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

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

solvers = {
    'ls': ('ls.mumps', {'fallback': 'ls2'}),
    'ls2': ('ls.scipy_umfpack', {}),
    'newton': ('nls.newton', {
                  'i_max': 1,
                  'eps_a': 1e-10,
              }),
}
Another possibility is to use a “virtual” solver that ensures an automatic selection of an available solver, see Virtual Linear Solvers with Automatic Selection.

Functions

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

Examples

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

- functions for defining regions:

```python
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:

```python
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:

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

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

(continues on next page)
def get_ebc_all(ts, coors):
    val = ts.step * coors
    return val

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

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

• function for defining usual material parameters:

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

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

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

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

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

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

functions = {
    'get_pars1': (lambda ts, coors, mode=None, **kwargs:
                  get_pars_special(ts, coors, mode,
                                   extra_arg='hello!'),),
}
# Just another way of adding a function, besides `functions` keyword.

```python
function_1 = {
    'name': 'get_pars2',
    'function': lambda ts, coors, mode=\texttt{None}, **kwargs:
        get_pars_special(ts, coors, mode, extra_arg=\texttt{hi!}),
}
```

- function combining both kinds of material parameters:

```python
def get_pars_both(ts, coors, mode=\texttt{None}, **kwargs):
    out = {}
    if mode == 'special':
        out['flag'] = coors.max() > 1.0
    elif mode == 'qp':
        val = coors[:,1]
        val.shape = (coors.shape[0], 1, 1)
        out['y_coor'] = val
    return out

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

- function for setting values of a parameter variable:

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

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

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

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

**Additional options (including solver selection):**

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

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)
• post_process_hook enables computing derived quantities, like stress or strain, from the primary unknown variables. See the examples in sfepy/examples/large_deformation/directory.

• parametric_hook makes it possible to run parametric studies by modifying the problem description programmatically. See sfepy/examples/diffusion/poisson_parametric_study.py for an example.

• output_dir redirects output files to specified directory
1.4.4 Building Equations in SfePy

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

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

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

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

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

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

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

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

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

### Syntax of Terms in Equations

The terms in equations are written in form:

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

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

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

1.4.5 Term Evaluation

Terms can be evaluated in two ways:

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

Each term supports one or more evaluation modes:

- 'weak': Assemble (in the finite element sense) either the vector or matrix depending on `diff_var` argument (the name of variable to differentiate with respect to) of `Term.evaluate()`. This mode is usually used implicitly when building the linear system corresponding to given equations.

- 'eval': The evaluation mode integrates the term (= integral) over a region. The result has the same dimension as the quantity being integrated. This mode can be used, for example, to compute some global quantities during postprocessing such as fluxes or total values of extensive quantities (mass, volume, energy, ...).

- 'el_eval': The element evaluation mode results in an array of a quantity integrated over each element of a region.
• 

  • ‘el_avg’ : The element average mode results in an array of a quantity averaged in each element of a region. This is the mode for postprocessing.

• ‘qp’ : The quadrature points mode results in an array of a quantity interpolated into quadrature points of each element in a region. This mode is used when further point-wise calculations with the result are needed. The same element type and number of quadrature points in each element are assumed.

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

• ‘dw_*’ terms support ‘weak’ mode

• ‘dq_*’ terms support ‘qp’ mode

• ‘d_*’ , ‘di_*’ terms support ‘eval’ and ‘el_eval’ modes

• ‘ev_*’ terms support ‘eval’, ‘el_eval’, ‘el_avg’ and ‘qp’ modes

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

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

1.4.6 Solution Postprocessing

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

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

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

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

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

    Returns
```

(continues on next page)
out : dict
    The updated output dictionary.
    
    from sfepy.base.base import Struct

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

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

return out

The full example is linear_elasticity-linear_elastic_probes.

• compute diffusion velocity given the pressure:

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

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

return out

The full example is biot-biot_npb.

• 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.

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

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

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

- **VTK probes**: It is the simple way of probing using the `post_process_hook`. It generates matplotlib figures with the probing results and previews of the mesh with the probe paths. See Primer or linear_elasticity-its2D_5 example.

- **SfePy probes**: As mentioned in Miscellaneous, it relies on defining two additional functions, namely the `gen_probes` function, that should create the required probes (see sfepy.discrete.probes), and the `probe_hook` function that performs the actual probing of the results for each of the probes. This function can return the probing results, as well as a handle to a corresponding matplotlib figure. See linear_elasticity-its2D_4 for additional explanation.

Using sfepy.discrete.probes allows correct probing of fields with the approximation order greater than one, see Interactive Example in Primer or linear_elasticity-its2D_interactive for an example of interactive use.

1.4.8 Postprocessing filters

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

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

The following code demonstrates the use of the postprocessing filters:

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

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

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

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

1.4.9 Solvers

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

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

The following solvers are available:

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

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

The following time step controllers are available:

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

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

Nonlinear Solvers

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

The following solvers are available:

- **nls.newton**: Solves a nonlinear system \( f(x) = 0 \) using the Newton method.
- **nls.oseen**: The Oseen solver for Navier-Stokes equations.
- **nls.petsc**: Interface to PETSc SNES (Scalable Nonlinear Equations Solvers).
- **nls.scipy_broyden_like**: Interface to Broyden and Anderson solvers from scipy.optimize.
- **nls.semismooth_newton**: The semi-smooth Newton method.
See `sfepy.solvers.nls`, `sfepy.solvers.oseen` and `sfepy.solvers.semismooth_newton` for all available nonlinear solvers and their options.

**Linear Solvers**

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

The following solvers are available:

- `ls.cm_pb`: Conjugate multiple problems.
- `ls.mumps`: Interface to MUMPS solver.
- `ls.mumps_par`: Interface to MUMPS parallel solver.
- `ls.petsc`: PETSc Krylov subspace solver.
- `ls.pyamg`: Interface to PyAMG solvers.
- `ls.pyamg_krylov`: Interface to PyAMG Krylov solvers.
- `ls.rmm`: Special solver for explicit transient elastodynamics.
- `ls.schur_mumps`: Mumps Schur complement solver.
- `ls.scipy_direct`: Direct sparse solver from SciPy.

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

**Virtual Linear Solvers with Automatic Selection**

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

The following solvers are available:

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

See `sfepy.solvers.auto_fallback` for all available virtual solvers.

**Eigenvalue Problem Solvers**

The following eigenvalue problem solvers are available:

- `eig.matlab`: Matlab eigenvalue problem solver.
- `eig.prime`: PRIMME eigenvalue problem solver.
- `eig.scipy`: SciPy-based solver for both dense and sparse problems.
- `eig.sgscipy`: SciPy-based solver for dense symmetric problems.
- `eig.slepc`: General SLEPe eigenvalue problem solver.
See `sfepy.solvers.eigen` for available \textit{eigenvalue problem} solvers and their options.

\textbf{Quadratic Eigenvalue Problem Solvers}

The following quadratic eigenvalue problem solvers are available:

- \texttt{eig.qevp}: Quadratic eigenvalue problem solver based on the problem linearization.

See `sfepy.solvers.qeigen` for available \textit{quadratic eigenvalue problem} solvers and their options.

\textbf{Optimization Solvers}

The following optimization solvers are available:

- \texttt{nls.scipy_fmin_like}: Interface to SciPy optimization solvers scipy.optimize.fmin_*.
- \texttt{opt.fmin_sd}: Steepest descent optimization solver.

See `sfepy.solvers.optimize` for available \textit{optimization} solvers and their options.

\subsection*{1.4.10 Solving Problems in Parallel}

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

\textbf{Current Implementation Drawbacks}

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

\textbf{Examples}

The examples demonstrating the use parallel problem solving in \texttt{SfePy} are:

- \texttt{diffusion-poisson_parallel_interactive}
- \texttt{multi_physics-biot_parallel_interactive}

See their help messages for further information.
1.4.11 Isogeometric Analysis

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

The implementation is *SfePy* is based on Bezier extraction of NURBS as developed in\(^1\). This approach allows reusing the existing finite element assembling routines, as still the evaluation of weak forms occurs locally in “elements” and the local contributions are then assembled to the global system.

Current Implementation

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

The following already works:

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

The following is not implemented yet:

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

Domain Description

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

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

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

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

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

Examples

The examples demonstrating the use of IGA in SfePy are:

- diffusion-poison_iga
- linear_elasticity-linear_elastic_iga
- navier_stokes-navier_stokes2d_iga

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

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

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

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

1.5 Examples

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

1.5.1 Primer

A beginner’s tutorial highlighting the basics of SfePy.

Introduction

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

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

![ITS schematic diagram](image)

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

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

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

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

Meshing

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

1. The ITS specimen has a diameter of 150 mm. Using Gmsh add three new points (geometry elementary entities) at the following coordinates: \((0.00.0)\), \((75.0, 0.0)\) and \((0.0, 75.0)\).

2. Next add two straight lines connecting the points.
3. Next add a Circle arc connecting two of the points to form the quarter circle segment.
4. Still under Geometry add a ruled surface.
5. With the geometry of the model defined, add a mesh by clicking on the 2D button under the Mesh functions.

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

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

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

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

To view .mesh files you can use a demo of medit. After loading your mesh file with medit you can see the node and element numbering by pressing P and F respectively. The numbering in medit starts at 1 as shown. Thus the node at the center of the model in SfePy numbering is 0, and elements 76 and 77 are connected to this node. Node and element numbers can also be viewed in Gmsh – under the mesh option under the Visibility tab enable the node and surface labels. Note that the surface labels as numbered in Gmsh follow on from the line numbering. So to get the corresponding element number in SfePy you’ll need to subtract the number of lines in the Gmsh file + 1. Confused yet? Luckily, SfePy provides some useful mesh functions to indicate which elements are connected to which nodes. Nodes and elements can also be identified by defining regions, which is addressed later.

Another open source python option to view .mesh files is the appropriately named Python Mesh Viewer.

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

**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:

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

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

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

(continues on next page)
where

.. math::
   \begin{align*}
   D_{ijkl} &= \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) + \\
   & \lambda \ \delta_{ij} \delta_{kl}.
   \end{align*}

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

# Fix the mesh file name if you run this file outside the SfePy directory.
filename_mesh = data_dir + '/meshes/2d/its2D.mesh'
refinement_level = 0
filename_mesh = refine_mesh(filename_mesh, refinement_level)
output_dir = '.

# set this to a valid directory you have write access to
young = 2000.0 # Young's modulus [MPa]
poisson = 0.4 # Poisson's ratio

options = {
    'output_dir' : output_dir,
}

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < 0.001)', 'facet'),
    'Bottom' : ('vertices in (y < 0.001)', 'facet'),
    'Top' : ('vertex 2', 'vertex'),
}

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

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

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

variables = {
    'u' : ('unknown field', 'displacement', 0),
    'v' : ('test field', 'displacement', 'u'),
}
ebcs = {
    'XSym': ('Bottom', {'u.1': 0.0}),
    'YSym': ('Left', {'u.0': 0.0}),
}

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

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

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

The following block of code defines regions on your mesh:

regions = {
    'Omega': 'all',
    'Left': ('vertices in (x < 0.001)', 'facet'),
    'Bottom': ('vertices in (y < 0.001)', 'facet'),
    'Top': ('vertex 2', 'vertex'),
}

Four regions are defined:

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

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

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

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

The load is specified in terms of the ‘Load’ material as follows:
materials = {
    'Asphalt': ({
        'lam': lame_from_youngpoisson(young, poisson)[0],
        'mu': lame_from_youngpoisson(young, poisson)[1],
    },),
    'Load': ({'val': [0.0, -1000.0]},),
}

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

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

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

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

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

### Running SfePy

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

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

**Note:** For the purpose of this tutorial it is assumed that the *problem description file* (`its2D_1.py`) is in the current directory. If you have the `its2D_1.py` file in another directory then make sure you include the path to this file as well. *SfePy* solves the problem and outputs the solution to the output path (`output_dir`) provided in the script. The output file will be in the VTK format by default if this is not explicitly specified and the name of the output file will be the same as that used for the mesh file except with the `.vtk` extension i.e. `its2D.vtk`.

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

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

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

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

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

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

```python
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson

def stress_strain(out, pb, state, extend=False):
    """
    Calculate and output strain and stress for given displacements.
    """
    ev = pb.evaluate
    strain = ev('ev_cauchy_strain.2.Omega(u)', mode='el_avg')
    stress = ev('ev_cauchy_stress.2.Omega(Asphalt.D, u)', mode='el_avg',
                copy_materials=False)
```

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

return out

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

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

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

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

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

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

### Running SfePy in interactive mode

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

In the SfePy top-level directory run

```
ipython
```

issue the following commands:
In [1]: from sfepy.applications import solve_pde

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

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

In [3]: u = variables()

In [4]: u

Out[4]:
array([[ 0. , 0. ],
       [ 0.37376671, 0. ],
       [ 0. , -1.65318152],
       ...
       [ 0.08716448, -0.23069047],
       [ 0.27741356, -0.19923848],
       [ 0.08820237, -0.11201528]])

In [5]: u.shape

Out[5]: (110,)

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

In [7]: u

Out[7]:
array([[ 0. , 0. ],
       [ 0.37376671, 0. ],
       [ 0. , -1.65318152],
       ...
       [ 0.08716448, -0.23069047],
       [ 0.27741356, -0.19923848],
       [ 0.08820237, -0.11201528]])

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

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

In [9]: u

Out[9]:
array([[ 0. , 0. ],
       [ 0.37376671, 0. ],
       [ 0. , -1.65318152],
       ...
       [ 0.08716448, -0.23069047],
       [ 0.27741356, -0.19923848],
       [ 0.08820237, -0.11201528]])

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

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

In [10]: K = pb.mtx_a

(continues on next page)
In [11]: K
Out[11]:
<94x94 sparse matrix of type 'numpy.float64'>
with 1070 stored elements in Compressed Sparse Row format>

In [12]: print(K)

(0, 0) 2443.95959851
(0, 7) -2110.99917491
(0, 14) -332.960423597
(0, 15) 1428.57142857
(1, 1) 2443.95959852
(1, 13) -2110.99917492
(1, 32) 1428.57142857
(1, 33) -332.960423596
(2, 2) 4048.78343529
(2, 3) -1354.87004384
(2, 52) -609.367453538
(2, 53) -1869.0018791
(2, 92) -357.41672785
(2, 93) 1510.24654193
(3, 2) -1354.87004384
(3, 3) 4121.03202907
(3, 4) -1696.54911732
(3, 48) 76.2400806561
(3, 49) -1669.59247304
(3, 52) -1145.85294856
(3, 53) 2062.13955556
(4, 3) -1696.54911732
(4, 4) 4410.17902905
(4, 5) -1872.87344838
(4, 42) -130.515009576
(91, 81) -1610.0550578
(91, 86) -199.343680224
(91, 87) -2330.41406097
(91, 90) -575.80373408
(91, 91) 7853.23899229
(92, 2) -357.41672785
(92, 8) 1735.59411191
(92, 50) -464.976034459
(92, 51) -1761.31189004
(92, 52) -3300.45367361
(92, 53) 1574.59387937
(92, 88) -250.325600254
(92, 89) 1334.11823335
(92, 92) 9219.18643706
(92, 93) -2607.52659081
(93, 2) 1510.24654193
(93, 8) -657.361661955
(93, 50) -1761.31189004
(93, 51) 54.1134516246
(93, 52) 1574.59387937

(continues on next page)
In [13]: K.shape
Out[13]: (94, 94)

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

In [14]: pb.remove_bcs()

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

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

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

This leads to:

```
ValueError: shape mismatch: value array of shape (55,2) could not be broadcast to...
˓→indexing result of shape (110,)
```

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

In [16]: u.shape = variables.vec.shape = (110,)

In [17]: f = pb.evaluator.eval_residual(u)

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

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

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

In [20]: pb.time_update()

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

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

Running the `sfepy-view` command on `file.vtk`, i.e.
In [25]: !sfepy-view file.vtk

displays the average nodal forces as shown below:

![average nodal forces](image)

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

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

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

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

**Generating output at element nodes**

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

```python
import print_function

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

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

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

where

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

from __future__ import print_function
```

(continues on next page)
from __future__ import absolute_import
from sfepy.examples.linear_elasticity.its2D_1 import *

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

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

def nodal_stress(out, pb, state, extend=False, integrals=None):
    
    # Point load.
    mat = pb.get_materials()['Load']
    P = 2.0 * mat.get_data('special', 'val')[1]

    # Calculate nodal stress.
    pb.time_update()

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

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

    print('
==================================================================
    
    Given load = %.2f N
    
    Analytical solution
    ====================
    
    Horizontal tensile stress = %.5e MPa/mm
    Vertical compressive stress = %.5e MPa/mm
    (2.*P/(nm.pi*150.))
    Vertical compressive stress = %.5e MPa/mm
    (6.*P/(nm.pi*150.))

    FEM solution
    =============

    Horizontal tensile stress = %.5e MPa/mm
    (svar()[0])
    Vertical compressive stress = %.5e MPa/mm
    (-svar()[1])

    =========================================================================

    return out

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

integrals = {
(continues on next page)
The output:

```
==================================================================
Given load = 2000.00 N
Analytical solution
=============
Horizontal tensile stress = 8.48826e+00 MPa/mm
Vertical compressive stress = 2.54648e+01 MPa/mm
FEM solution
=============
Horizontal tensile stress = 7.57220e+00 MPa/mm
Vertical compressive stress = 2.58660e+01 MPa/mm
==================================================================
```

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

```
==================================================================
Given load = 2000.00 N
Analytical solution
=============
Horizontal tensile stress = 8.48826e+00 MPa/mm
Vertical compressive stress = 2.54648e+01 MPa/mm
FEM solution
=============
Horizontal tensile stress = 8.50042e+00 MPa/mm
Vertical compressive stress = 2.54300e+01 MPa/mm
==================================================================
```

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

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

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

```
In [23]: from sfepy.applications import solve_pde
In [24]: from sfepy.discrete import (Field, FieldVariable, Material,
                            Integral, Integrals)
In [25]: from sfepy.discrete.fem.geometry_element import geometry_data
In [26]: gdata = geometry_data['2_3']
In [27]: nc = len(gdata.coors)
In [28]: ivn = Integral('ivn', order=-1,
                   coors=gdata.coors, weights=[gdata.volume / nc] * nc)
In [29]: pb, variables = solve_pde('sfepy/examples/linear_elasticity/its2D_2.py')
(continues on next page)
In [30]: stress = pb.evaluate('ev_cauchy_stress.ivn.Omega(Asphalt.D,u)',
    mode='qp', integrals=Integrals([ivn]))
In [31]: sfield = Field.from_args('stress', nm.float64, (3,), pb.domain.regions['Omega'])
In [32]: svar = FieldVariable('sigma', 'parameter', sfield,
    primary_var_name='(set-to-None)')
In [33]: svar.set_from_qp(stress, ivn)

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

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

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

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

Probing

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

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

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

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

where

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

""

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

(continues on next page)
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.postprocess.probes_vtk import Probe

import os
from six.moves import range

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

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

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

    probe = Probe(out, pb.domain.mesh, probe_view=True)
    ps0 = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
    ps1 = [[75.0, 0.0, 0.0], [0.0, 75.0, 0.0]]
    n_point = 10

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

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

        sym_labels = ['11', '22', '12']
        plt.subplot(312)
probing applies interpolation to output the solution along specified paths. for the tutorial, line probing is done along the x- and y-axes of the model.

run SfePy to solve the problem and apply the probes:

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

the probing function will generate the following figures that show the displacements, normal stresses and strains as well as shear stresses and strains along the probe paths. note that you need matplotlib installed to run this example.
The probing function also generates previews of the mesh with the probe paths.
Interactive Example

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

```python
#!/usr/bin/env python

""
Diametrically point loaded 2-D disk, using commands for interactive use. See
:ref:`sec-primer`.

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

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

The example shows also how to probe the results as in
correct probing of fields with the approximation order greater than one.

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

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

""

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

import numpy as nm
import matplotlib.pyplot as plt

from sfepy.base.base import assert_, output, ordered_iteritems, IndexedStruct
from sfepy.discrete import (FieldVariable, Material, Integral, Integrals,
                           Equation, Equations, Problem)
from sfepy.discrete.fem import Mesh, FEDomain, Field
from sfepy.terms import Term
```

(continues on next page)
from sfepy.discrete.conditions import Conditions, EssentialBC
from sfepy.mechanics.matcoefs import stiffness_from_youngpoisson
from sfepy.solvers.auto_fallback import AutoDirect
from sfepy.solvers.nls import Newton
from sfepy.discrete.fem.geometry_element import geometry_data
from sfepy.discrete.probes import LineProbe
from sfepy.discrete.projections import project_by_component

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

def gen_lines(problem):
    """
    Define two line probes.
    Additional probes can be added by appending to `ps0` (start points) and
    `ps1` (end points) lists.
    """
    ps0 = [[0.0, 0.0], [0.0, 0.0]]
    ps1 = [[75.0, 0.0], [0.0, 75.0]]

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

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

    return probes, labels

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

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

    fig = plt.figure()
    plt.clf()
    fig.subplots_adjust(hspace=0.4)
    plt.subplot(311)
    for ic in range(vals.shape[1]):
        plt.plot(pars, vals[:,ic], label=r'$u_{%d}$' % (ic + 1),
        "(continues on next page)"

1.5. Examples

(continues on next page)
lw=1, ls='-', marker='+', ms=3)
plt.ylabel('displacements')
plt.xlabel('probe %s' % label, fontsize=8)
plt.legend(loc='best', fontsize=10)

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

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

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

return fig, results

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

def main():
    from sfepy import data_dir

    parser = argparse.ArgumentParser(description=__doc__,
                                     formatter_class=RawDescriptionHelpFormatter)
    parser.add_argument('--version', action='version', version='%(prog)s')
    parser.add_argument('--young', metavar='float', type=float,
                        action='store', dest='young',
                        default=2000.0, help=helps['young'])
    parser.add_argument('--poisson', metavar='float', type=float,
                        action='store', dest='poisson',
                        default=0.4, help=helps['poisson'])
    parser.add_argument('--load', metavar='float', type=float,
                        action='store', dest='load',
                        default=-1000.0, help=helps['load'])

    (continues on next page)
parser.add_argument('--order', metavar='int', type=int, action='store', dest='order', default=1, help=helps['order'])
parser.add_argument('-r', '--refine', metavar='int', type=int, action='store', dest='refine', default=0, help=helps['refine'])
parser.add_argument('-p', '--probe', action='store_true', dest='probe', default=False, help=helps['probe'])

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

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

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

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

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

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

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

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

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

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

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

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

ls = AutoDirect({})

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

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

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

pb.set_solver(nls)

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

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

pb.save_state('its2D_interactive.vtk', out=out)

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

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

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

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

    sfield = Field.from_args('sym_tensor', nm.float64, 3, omega,
approx_order=options.order - 1)

    stress = FieldVariable('stress', 'parameter', sfield,
primary_var_name='(set-to-None)')

    strain = FieldVariable('strain', 'parameter', sfield,
primary_var_name='(set-to-None)')

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

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

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

all_results = []
for ii, probe in enumerate(probes):
    fig, results = probe_results(u, strain, stress, probe, labels[ii])
    fig.savefig('its2D_interactive_probe_%d.png' % ii)
    all_results.append(results)

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

if __name__ == '__main__':
    main()

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

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

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

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

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

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

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

The end.
1.5.2 Using Salome with SfePy

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

Introduction

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

Tutorial prerequisites

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

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

Note on notation used in this tutorial

We are using the following notations:

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

Step 1: Using *Salome*

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

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

This is the mesh you should end up with:
Step 2: Exporting mesh from Salome

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

Step 3: Copy SfePy project description files

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

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

Step 4: Modify linear_elastic.py

Mesh specification

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

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

to

```python
filename_mesh = 'Mesh_Partition_Hexa.med'
```
Region specification

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

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

to

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

Field specification

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

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

Boundary condition specifications

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

Change the lines

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

to

```
ebcs = {
    'TopFixed' : ('TopFixed', {'u.all' : 0.0}),
    'BotFixed' : ('BotFixed', {'u.all' : 0.0}),
    'Displaced' : ('Tail', {'u.2' : 10, 'u.[0,1]' : 0.0}),
}
```
Step 5: Run SfePy

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

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

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

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

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

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

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

1.5.3 Preprocessing: FreeCAD/OpenSCAD + Gmsh

Introduction

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

The collaboration of modeling, meshing and conversion tools and the workflow are illustrated in the following scheme.
Creating geometry using *FreeCAD*

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

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

```python
import sys
FREECADPATH = '/usr/lib/freecad/lib/'
sys.path.append(FREECADPATH)

from FreeCAD import Base, newDocument
import Part
import Draft
import ProfileLib.RegularPolygon as Poly
```

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

```python
doc = newDocument()
```

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

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

```python
radius = 0.01
height = 0.1

cyl = doc.addObject("Part::Cylinder", "cyl")
cyl.Radius = radius
cyl.Height = height

sph = doc.addObject("Part::Sphere", "sph")
sph.Radius = radius

uni = doc.addObject("Part::MultiFuse", "uni")
uni.Shapes = [cyl, sph]
```

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

```python
ske = doc.addObject('Sketcher::SketchObject', 'Sketch')
ske.Placement = Base.Placement(Base.Vector(0, 0, 0),
                               Base.Rotation(-0.707107, 0, 0, -0.707107))
Poly.makeRegularPolygon('Sketch', 5,
                        Base.Vector(-1.2 * radius, 0.9 * height, 0),
                        Base.Vector(-0.8 * radius, 0.9 * height, 0))

cut = doc.addObject("PartDesign::Revolution", "Revolution")
cut.Sketch = ske
cut_REFERENCE_AXIS = (ske, ['V_Axis'])
cut.Angle = 360.0

dif = doc.addObject("Part::Cut", "dif")
```

(continues on next page)
Create a cylinder, make a polar array of the cylinder objects and subtract it from the previous result:

```python
cyl1 = doc.addObject("Part::Cylinder", "cyl1")
cyl1.Radius = 0.2 * radius
cyl1.Height = 1.1 * height
cyl1.Placement = Base.Placement(Base.Vector(-1.1 * radius, 0, -0.2 * height),
                               Base.Rotation(0, 0, 0, 1))
arr = Draft.makeArray(cyl1, Base.Vector(1, 0, 0), Base.Vector(0, 1, 0), 2, 2)
arr.ArrayType = "polar"
arr.NumberPolar = 6
dif2 = doc.addObject("Part::Cut", "dif2")
dif2.Base = dif
dif2.Tool = arr
```

Create a middle hole for the screwdriver metal part:

```python
cyl2 = doc.addObject("Part::Cylinder", "cyl2")
cyl2.Radius = 0.3 * radius
cyl2.Height = height
dif3 = doc.addObject("Part::Cut", "dif3")
dif3.Base = dif3
dif3.Tool = cyl2
```

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

```python
doc.recompute()
Part.export([dif3], 'screwdriver_handle.step')
doc.saveAs('screwdriver_handle.FCStd')
```

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

```python
import MeshPart
mesh = doc.addObject("Mesh::Feature", "Mesh")
mesh.Mesh = MeshPart.meshFromShape(Shape=dif3.Shape, MaxLength=0.002)
mesh.Mesh.write("./screwdriver_handle.bdf", "NAS", "mesh")
```

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

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

**Creating geometry using OpenSCAD**

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

```scad
radius = 0.01;
height = 0.1;
$fn = 50;

difference() {
    difference() {
        difference() {
            union() {
                cylinder(center=false, h=height, r=radius);
                sphere(radius);
            }
        }
    }
}
```

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

```python
import sys
sys.path.append('/usr/lib/freecad/lib/)
sys.path.append('/usr/lib/freecad/Mod/OpenSCAD/)
import FreeCAD
import Part
import importCSG
importCSG.open('screwdriver_handle.csg')
Part.export([FreeCAD.ActiveDocument.Objects[-1]], 'screwdriver_handle.step')
```
Gmsh - generating finite element mesh

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

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

```plaintext
Merge "screwdriver_handle.step";
Field[1] = MathEval;
Field[1].F = "0.002";
Background Field = 1;
```

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

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

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

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

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

---

**Note: planar 2D meshes**

To create a planar 2D mesh, such as
that can be described by this *Gmsh* code, the mesh generator can be called as follows:

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

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

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

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

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

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

### Multipart models

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

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

```python
cyl3 = doc.addObject("Part::Cylinder", "cyl3")
cyl3.Radius = 0.3 * radius

cyl3.Height = height

cyl3.Placement = Base.Placement(Base.Vector(0, 0, height),
                                 Base.Rotation(0, 0, 0, 1))

tip1 = doc.addObject("Part::Box", "tip1")
tip1.Length = radius

tip1.Width = 2 * radius

tip1.Height = 3 * radius

tip1.Placement = Base.Placement(Base.Vector(0, -radius, 1.71 * height),
                                 Base.Rotation(Base.Vector(0, 1, 0), -10),
                                 Base.Vector(0, 0, 3 * radius))

tip2 = doc.addObject("Part::Mirroring", "tip2")
tip2.Source = tip1

tip2.Normal = (1, 0, 0)

tip3 = doc.addObject("Part::MultiFuse", "tip3")
tip3.Shapes = [tip1, tip2]

dif4 = doc.addObject("Part::Cut", "dif4")
dif4.Base = cyl3
dif4.Tool = tip3

uni2 = doc.addObject("Part::MultiFuse", "uni2")
uni2.Shapes = [cyl2, dif4]
```

---

**1.5. Examples**
The handle and shank are exported to the \textit{STEP} file as two separated parts:

\begin{verbatim}
1 doc.recompute()
2 Part.export([dif3, uni2], 'screwdriver_full.step')
3 doc.saveAs('screwdriver_full.FCStd')
\end{verbatim}

The full screwdriver example (handle + shank): \texttt{screwdriver_full.py}.

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

The input file for \textit{Gmsh} is than as follows (\texttt{screwdriver_full.geo}):

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

(continues on next page)
Field[1] = MathEval;
Field[1].F = "0.0015";
Background Field = 1;

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

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

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

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

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

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

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

```
radius = 0.01;
height = 0.1;
$fn = 50;

module tip() {
   rotate([0, -10, 0])
   translate([0, -radius, -3*radius])
   cube([radius, 2*radius, 3*radius], center=false);
}
difference() {
   difference() {
      difference() {
         union() {
```
(continues on next page)
and convert the CSG file to the STEP file by:

```python
importCSG.open('screwdriver_full.csg')
top_group = FreeCAD.ActiveDocument.Objects[-1]
Part.export(top_group.OutList, 'screwdriver_full.step')
```

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

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

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

Introduction

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

Composite structure

The unidirectional long fiber carbon-epoxy composite is considered. Its microstructure was analysed by the scanning electron microscopy and the data, volume fractions and fibers cross-sections, were used to generate a periodic finite element mesh (representative volume element - RVE) representing the random composite structure at the microscopic level (the random structure generation algorithm is described in\(^\text{1}\)):

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

Material testing

Several carbon-expoxy specimens with different fiber orientations (0, 30, 60 and 90 degrees) were subjected to the tensile test in order to obtain force-elongation dependencies, see\(^\text{2}\). The slopes of the linearized dependencies were used in an objective function of the identification process.


Numerical simulation

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

The numerical simulations of the tensile tests are based on the homogenization method applied to the linear elastic problem\(^3\). The homogenization procedure results in the microscopic problem solved within the RVE and the macroscopic problem that involves the homogenized elastic coefficients.

Homogenized coefficients

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

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

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

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

Macroscopic simulation

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

The SfePy description file for the macroscopic analysis: linear_elasticity_opt_src.

Identification procedure

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

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

(1.9)

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

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

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

Python script for material identification: material_opt_src.

Running identification script

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

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

The iteration process is monitored using graphs where the values of the objective function and material parameters are plotted.
The resulting values of $E_f$, $\nu_f$, $E_m$, $\nu_m$ can be found at the end of the script output:

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

So that:

- $E_f = 171.13$ GPa
- $\nu_f = 0.321$
- $E_m = 2.34$ GPa
- $\nu_m = 0.20$

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

### 1.5.5 Mesh parametrization

**Introduction**

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

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

The rectangular B-spline parametrization is created as follows:

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

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

\[
ncp_i = nsg_i + degree, \quad i = 1, 2, 3
\]

(1.10)

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

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

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

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

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

Example

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

```python
from sfepy.mesh.splinebox import SplineBox
from sfepy.discrete.fem import Mesh

mesh = Mesh.from_file('meshes/2d/square_tril.mesh')
spb = SplineBox([[-1, 1], [-1, 0.6]], mesh.coors, nsg=[2, 1])
```

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

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

- Evaluate the new coordinates:
  ```python
  mesh.cmesh.coors[:] = spb.evaluate()
  ```
- Write the deformed mesh and the spline control net (the net of control points) into vtk files:
  ```python
  spb.write_control_net('square_tril_spbox.vtk')
  mesh.write('square_tril_deform.vtk')
  ```

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

![Undeformed and deformed mesh with control net](image)

**SplineRegion2D**

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

```python
from sfepy.mesh.splinebox import SplineRegion2D
spb = SplineRegion2D([<bspl1>, <bspl2>, <bspl3>, <bspl4>], <coors>)
```

The curves must form a closed loop, must be oriented counterclockwise and the opposite curves (<bspl1>, <bspl3> and <bspl2>, <bspl4>) must have the same number of control points and the same knot vectors, see the figure below, on the left.
The position of the selected vertices, depicted in the figure on the right, are driven by the control points in the same way as explained above for SplineBox.

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

**Example**

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

```python
from sfepy.mesh.bspline import BSpline

# left / right boundary
line_l = nm.array([[-1, 1], [-1, .5], [-1, 0], [-1, -.5]])
line_r = nm.array([[0, -.2], [.1, .2], [.3, .6], [.4, 1]])

sp_l = BSpline()
sp_l.approximate(line_l, ncp=4)
kn_lr = sp_l.get_knot_vector()

sp_r = BSpline()
sp_r.approximate(line_r, knots=kn_lr)

# bottom / top boundary
line_b = nm.array([[-1, -.5], [-.8, -.6], [-.5, -.4], [-.2, -.2], [0, -.2]])
line_t = nm.array([[.4, 1], [0, 1], [-.2, 1], [-.6, 1], [-1, 1]])

sp_b = BSpline()
sp_b.approximate(line_b, ncp=5)
kn_bt = sp_b.get_knot_vector()

sp_t = BSpline()
sp_t.approximate(line_t, knots=kn_bt)
```

- Create a new 2D SplineRegion2D object:
```python
from sfepy.mesh.splinebox import SplineRegion2D
spb = SplineRegion2D([sp_b, sp_r, sp_t, sp_l], mesh.coors)
```

- Move the control points:
  ```python
  spb.move_control_point(5, [-.2, .1])
  spb.move_control_point(10, [-.3, .2])
  spb.move_control_point(15, [-.1, .2])
  ```

- Evaluate the new coordinates:
  ```python
  mesh.cmesh.coors[:] = spb.evaluate()
  ```

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

### 1.5.6 Examples

**multi_physics**

**multi_physics/piezoelectric_elastodynamic.py**

**Description**

The linear elastodynamics of a piezoelectric body loaded by a given base motion.

The generated voltage between the bottom and top surface electrodes is recorded and plotted. The scalar potential on the top surface electrode is modeled using a constant L^2 field. The Nitsche’s method is used to weakly apply the (unknown) potential Dirichlet boundary condition on the top surface.

Find the displacements \( u(t) \), the potential \( p(t) \) and the constant potential on the top electrode \( \bar{p}(t) \) such that:

\[
\begin{align*}
\int_{\Omega} \rho \cdot \ddot{u} + \int_{\Omega} C_{ijkl} \varepsilon_{ij}(u) \varepsilon_{kl}(u) - \int_{\Omega} e_{ijkl} \varepsilon_{ij}(u) \nabla_k p = 0, \quad \forall \nu, \\
\int_{\Omega} e_{ijkl} \varepsilon_{ij}(u) \nabla_k q + \int_{\Omega} \kappa_{ij} \nabla_i \psi \nabla_j p - \int_{\Gamma_{top}} \kappa_{ij} \nabla_j q n_i + \int_{\Gamma_{top}} \kappa_{ij} \nabla_j p n_i (p - \bar{p}) + \int_{\Gamma_{top}} kq(p - \bar{p}) = 0, \quad \forall q, \\
\int_{\Gamma_{top}} \kappa_{ij} \nabla_j \dot{m}_i + \bar{p} / R = 0, 
\end{align*}
\]
where $C_{ijkl}$ is the matrix of elastic properties under constant electric field intensity, $e_{kij}$ the piezoelectric modulus, $\kappa_{ij}$ the permittivity under constant deformation, $k$ a penalty parameter and $R$ the external circuit resistance (e.g. of an oscilloscope used to measure the voltage between the electrodes).

### Usage Examples

Run with the default settings, results stored in output/piezo-ed/:

```bash
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py
```

The `define()` arguments, see below, can be set using the `-d` option:

```bash
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py -d "order=2, ct1=2.5"
```

View the resulting potential $p$ on a deformed mesh (2000x magnified):

```bash
```

source code

```python
r''''
The linear elastodynamics of a piezoelectric body loaded by a given base
```

(continues on next page)
motion.

The generated voltage between the bottom and top surface electrodes is recorded and plotted. The scalar potential on the top surface electrode is modeled using a constant $L^2$ field. The Nitsche's method is used to weakly apply the (unknown) potential Dirichlet boundary condition on the top surface.

Find the displacements $\mathbf{u}(t)$, the potential $p(t)$ and the constant potential on the top electrode $\bar{p}(t)$ such that:

\[
\int_{\Omega} \rho \ \mathbf{v} \cdot \mathbf{\ddot{u}} + \int_{\Omega} C_{ijkl} \ \varepsilon_{ij}(\mathbf{v}) \varepsilon_{kl}(\mathbf{u}) - \int_{\Omega} e_{kij} \ \varepsilon_{ij}(\mathbf{v}) \nabla_k p = 0, \quad \forall \mathbf{v},
\]
\[
\int_{\Omega} e_{kij} \ \varepsilon_{ij}(\mathbf{u}) \nabla_k q + \int_{\Omega} \kappa_{ij} \nabla_i \psi \nabla_j p - \int_{\Gamma_{top}} \kappa_{ij} \nabla_j p n_i q + \int_{\Gamma_{top}} \kappa_{ij} \nabla_j q n_i (p - \bar{p}) + \int_{\Gamma_{top}} k q (p - \bar{p}) = 0, \quad \forall q,
\]
\[
\int_{\Gamma_{top}} \kappa_{ij} \nabla_j \dot{p} n_i + \bar{p} / R = 0,
\]

where $C_{ijkl}$ is the matrix of elastic properties under constant electric field intensity, $e_{kij}$ the piezoelectric modulus, $\kappa_{ij}$ the permittivity under constant deformation, $k$ a penalty parameter and $R$ the external circuit resistance (e.g. of an oscilloscope used to measure the voltage between the electrodes).

Usage Examples
-------------

Run with the default settings, results stored in `output/piezo-ed/`:

```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py
```

The `define()` arguments, see below, can be set using the `--d` option:

```
sfepy-run sfepy/examples/multi_physics/piezo_elastodynamic.py --d "order=2, ct1=2.5"
```

View the resulting potential $p$ on a deformed mesh (2000x magnified):

```
```

```python
from functools import partial

import numpy as nm
```

(continues on next page)
from sfepy.base.base import output
from sfepy.discrete.fem.meshio import UserMeshIO
from sfepy.mesh.mesh_generators import gen_block_mesh
from sfepy.homogenization.utils import define_box_regions

def post_process(out, problem, state, extend=False, pcs=None):
    """
    Calculate and output strain, stress and electric field vector for the given
    displacements and potential.
    """
    from sfepy.base.base import Struct
    ev = problem.evaluate
    strain = ev('ev_cauchy_strain.i.Omega(u)', mode='el_avg', verbose=False)
    stress = ev('ev_cauchy_stress.i.Omega(m.C, u)', mode='el_avg',
    copy_materials=False, verbose=False)
    E = ev('ev_grad.i.Omega(p)', mode='el_avg', verbose=False)

    out['cauchy_strain'] = Struct(name='output_data', mode='cell',
        data=strain)
    out['cauchy_stress'] = Struct(name='output_data', mode='cell',
        data=stress)
    out['E'] = Struct(name='output_data', mode='cell', data=E)

    top = problem.domain.regions['Top']
    p_top = state['p'].get_state_in_region(top)
    # = state['pc'](), but we want to test .get_state_in_region()
    pc_top = state['pc'].get_state_in_region(top)

    output('pc:', pc_top)
    output('|p - pc|_top:', nm.linalg.norm(p_top - pc_top))
    if pcs is not None:
        pcs.append(pc_top[0, 0])

    return out

def plot_voltage(problem, state, pcs=None):
    import os.path as op
    import matplotlib.pyplot as plt
    ts = problem.get_timestepper()

    fig, ax = plt.subplots()
    ax.plot(ts.times, pcs)
    ax.set_xlabel('$t$ [s]')
    ax.set_ylabel(r'$\bar p$ [V]')
    fig.tight_layout()
    fig.savefig(op.join(problem.output_dir, 'voltage.pdf'))

def define()
(continued from previous page)

dims=(1e-2, 1e-2, 5e-3),
shape=(5, 11, 21),
order=1,
amplitude=0.0000001,
c1=1.5,
dt=None,
tss_name='tsn',
tsc_name='tscedl',
adaptive=False,
ls_name='lsd',
active_only=False,
save_times='all',
output_dir='output/piezo-ed',
):
    ""
    Parameters
    ----------
dims: physical dimensions of the block mesh
shape: numbers of mesh vertices along each axis
order: the FE approximation order
amplitude: the seismic load amplitude
c1: final time in min(dims) / "longitudinal wave speed" units
dt: time step (None means automatic)
tss_name: time stepping solver name (see "solvers" section)
tsc_name: time step controller name (see "solvers" section)
adaptive: use adaptive time step control
ls_name: linear system solver name (see "solvers" section)
save_times: number of solutions to save
output_dir: output directory
    ""
    dim = len(dims)
    assert dim == 3

    # A PZT 5-H material, Voigt notation, strain - electric displacement form.
    epsT = nm.array([[1700., 0, 0],
                     [0, 1700., 0],
                     [0, 0, 1450.0]])
    dv = 1e-12 * nm.array([[0, 0, 0, 0, 741., 0],
                           [0, 0, 0, 741, 0, 0],
                           [-274., -274., 593., 0, 0, 0]])  # C / N = m / V

    # Convert to stress - electric displacement form.
    CEv = nm.array([[1.27e+011, 8.02e+010, 8.47e+010, 0, 0, 0],
                    [8.02e+010, 1.27e+011, 8.47e+010, 0, 0, 0],
                    [8.47e+010, 8.47e+010, 1.17e+011, 0, 0, 0],
                    [0, 0, 0, 2.34e+011, 0, 0],
                    [0, 0, 0, 2.30e+010, 0],
                    [0, 0, 0, 0, 2.35e+010]])

    ev = dv @ CEv
    epsS = epsT - dv @ ev.T

    # SfePy: 11 22 33 12 13 23

(continues on next page)
# Voigt: 11 22 33 23 13 12
ii = [0, 1, 2, 5, 4, 3]
ix, iy = nm.meshgrid(ii, ii, sparse=True)
CE = CEv[ix, iy]
e = ev[:, ii]

eps0 = 8.8541878128e-12
kappa = epsS * eps0

# Longitudinal and shear wave propagation speeds.
mu = CE[-1, -1]
lam = CE[0, 0] - 2 * mu
rho = 7800
cl = nm.sqrt((lam + 2.0 * mu) / rho)
cs = nm.sqrt(mu / rho)

# Element size.
L = nm.min(dims)
H = L / (nm.max(shape) - 1)

# Time-stepping parameters.
if dt is None:
    # For implicit schemes, dt based on the Courant number C0 = dt * cl / H
    # equal to 1.
    dt = H / cl # C0 = 1

t1 = ct1 * L / cl

# Time history record of pc.
pcs = []
_post_process = partial(post_process, pcs=pcs)
_plot_voltage = partial(plot_voltage, pcs=pcs)

def mesh_hook(mesh, mode):
    """
    Generate the block mesh.
    """
    if mode == 'read':
        mesh = gen_block_mesh(dims, shape, 0.5 * nm.array(dims),
                              name='user_block', verbose=False)
        return mesh
    elif mode == 'write':
        pass

filename_mesh = UserMeshIO(mesh_hook)

bbox = [[0] * dim, dims]
regions = define_box_regions(dim, bbox[0], bbox[1], 1e-5)
regions.update({
    'Omega': 'all',
})
materials = {
    'inclusion': (None, 'get_inclusion_pars')
}

fields = {
    'displacement': ('real', 'vector', 'Omega', order),
    'potential': ('real', 'scalar', 'Omega', order),
    'constant': ('real', 'scalar', 'Top', 0, 'L2', 'constant'),
}

variables = {
    'u': ('unknown field', 'displacement', 0),
    'v': ('test field', 'displacement', 'u'),
    'p': ('unknown field', 'potential', 1, 1),
    'q': ('test field', 'potential', 'p'),
    'pc': ('unknown field', 'constant', 2, 1),
    'qc': ('test field', 'constant', 'pc'),
}

materials = {
    'm': (i
        'C': CE,
        'e': e,
        'kappa': kappa,
        'rho': rho,
        'penalty': 1,
        'iR': 1.0 / (15e6 * dims[0] * dims[1]), # 1 / (R * top_area).
    ),),
}

integrals = {
    'i': 2 * order,
}

def get_ebcs(ts, coors, mode='u'):
    y = coors[:, 1]
    k = 2 * nm.pi / dims[1]
    shift = nm.pi / 3
    omega = cl * k
    time = ts.time
    if mode == 'u':
        val = (amplitude * nm.sin(time * omega) * nm.sin(k * y + shift))
    elif mode == 'du':
        val = (amplitude * omega * nm.cos(time * omega) * nm.sin(k * y + shift))
    elif mode == 'ddu':
        val = (-amplitude * omega**2 * nm.sin(time * omega) * nm.sin(k * y + shift))
return val

functions = {
    'get_u': (lambda ts, coor, **kwargs: get_ebcs(ts, coor),),
    'get_du': (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='du'),),
    'get_ddu': (lambda ts, coor, **kwargs: get_ebcs(ts, coor, mode='ddu'),),
}

ebcs = {
    'Seismic': ('Bottom', {'u.2': 'get_u', 'du.2': 'get_du',
                           'ddu.2': 'get_ddu'}),
    'Pot0': ('Bottom', {'p.all': 0.0}),
}

ics = {
    'ic': ('Omega', {'u.all': 0.0, 'du.all': 0.0, 'p.0': 0.0}),
}

equations = {
    '1': '"""dw_dot.i.Omega(m.rho, v, ddu) + dw_lin_elastic.i.Omega(m.C, v, u) -
         dw_piezo_coupling.i.Omega(m.e, v, p) = \$0\"""",
    '2': '"""dw_piezo_coupling.i.Omega(m.e, u, q) + dw_diffusion.i.Omega(m.kappa, q, p) -
         de_surface_flux.i.Top(m.kappa, q, p) + de_surface_flux.i.Top(m.kappa, p, q) -
         de_surface_flux.i.Top(m.kappa, pc, q) + dw_dot.i.Top(m.penalty, q, p) -
         dw_dot.i.Top(m.penalty, q, pc) = \$0\"""",
    '3': '"""de_surface_flux.i.Top(m.kappa, qc, dp/dt) + 0.5 * dw_dot.i.Top(m.iR, qc, pc)
         + 0.5 * dw_dot.i.Top(m.iR, qc, pc[-1]) = \$0\"""",
}

solvers = {
    'lsd': ('ls.auto_direct', {
        # Reuse the factorized linear system from the first time step.
        'use_presolve': True,
        # Speed up the above by omitting the matrix digest check used
        # normally for verification that the current matrix corresponds to
        # the factorized matrix stored in the solver instance. Use with
        # care!
        'use_mtx_digest': False,
        # Increase when getting MUMPS error -9.
        'memory_relaxation': 20,
    }),
    'newton': ('nls.newton', {
        'i_max': 1,
        'eps_a': 1e-6,
    }),
}
options = {
    'ts' : tss_name,
    'tsc' : tsc_name if adaptive else None,
    'nls' : 'newton',
    'ls' : ls_name,

    'save_times' : save_times,

    'active_only' : active_only,
    'auto_transform_equations' : True,

    'output_format' : 'h5',
    'output_dir' : output_dir,
    'post_process_hook' : _post_process,
    'post_process_hook_final' : _plot_voltage,
}

return locals()
1.5.7 Example Applications

- Homogenization of peristaltic flows in piezoelectric porous media (2023)
- Deformation of a foam-reinforced shell beam (2023)
- Two-scale numerical simulation of a large deforming fluid-saturated porous structure (2021)
- Homogenization of the vibro-acoustic transmission on perforated plates with embedded resonators (2021)
- Multiscale numerical modelling of perfusion in deformable double porous media described by the Biot-Darcy-Brinkman model (2020)
- Homogenization of piezoelectric porous media (2020)
- Numerical simulation of viscous flow in deformable double porous media (2020)
- Numerical simulations of large-deforming fluid-saturated porous media using an Eulerian incremental formulation (2017)
- Fish heart model (2010)
- Phononic materials (2010)

Note that older examples do not reflect the current state of SfePy.

1.6 Useful Code Snippets and FAQ

Code examples below that use sfepy-* scripts assume the sfepy package to be installed, see also Installation.

1.6.1 Miscellaneous

1. No module named 'sfepy.discrete.common.extmods.mappings'.

When installing SfePy from sources or using the git version, its extension modules have to be compiled before using the package, see Compilation of C Extension Modules.

2. The extension modules are compiled in place, but ModuleNotFoundError: No module named 'sfepy' shows up when running some interactive examples/scripts/modules from the SfePy source directory.

On some platforms the current directory is not in the sys.path directory list. Add it using:

```python
export PYTHONPATH=.
```
or add the following code prior to sfepy imports into the module:

```python
import sys
sys.path.append('.')
```

3. Finite element approximation (field) order and numerical quadrature order.

SfePy supports reading only straight-facet (linear approximation) meshes, nevertheless field orders higher than one can be used, because internally, the mesh elements are enriched with the required additional nodes. The calculation then occurs on such an augmented mesh with appropriate higher order elements.

The quadrature order equal to two-times the field order (used in many examples) works well for bilinear forms with constant (on each element) material parameters. For example, a dot product involves integrating \( u \times v \), so if the approximation order of \( u \) and \( v \) is 1, their product’s order is 2. Of course, there are terms that could use
a lower quadrature order, or higher, depending on the data. Increased quadrature order is required e.g. in terms with highly oscillating material coefficients.

Example:

```python
approx_order = 2
# The finite element approximation order.
fields = {
    'displacement': ('real', 3, 'Omega', approx_order),
}
# The numerical quadrature order.
integrals = {
    'i': 2 * approx_order,
}
```

4. Higher order DOF visualization when using an approximation order greater than one.

By default, the additional, higher order DOFs, are not used in the VTK/HDF5 results files ('strip' linearization kind). To see the influence of those DOFs, 'adaptive' linearization has to be used, see diffusion-sinbc (declarative API) and diffusion-laplace_refine_interactive or multi_physics-biot_parallel_interactive (imperative API, search linearization).

5. Numbering of DOFs.

Locally (in a connectivity row), the DOFs are stored DOF-by-DOF ($u_0$ in all local nodes, $u_1$ in all local nodes, ...).

Globally (in a state vector), the DOFs are stored node-by-node ($u_0$, $u_1$, ..., $u_X$ in node 0, $u_0$, $u_1$, ..., $u_X$ in node 1, ...).

See also `create_adof_conn()`.

6. Visualization of various FEM-related information.

- Quadrature rules:
  ```
  python3 sfepy/scripts/plot_quadratures.py
  ```

- Facet orientations - run in the source code directory and make sure the current directory is in the Python’s path list (see Miscellaneous):
  ```
  python3 sfepy/postprocess/plot_facets.py
  ```

- Global and local numberings of mesh topological entities (cells, faces, edges, vertices):
  ```
  python3 sfepy/scripts/plot_mesh.py meshes/elements/2_4_2.mesh
  ```

  The global numbers serve as indices into connectivities. In the plot, the global numbers are on the entities, the cell-local ones are inside the cells next to each entity towards the cell centroids.

7. How to work with solvers/preconditioners?

See multi_physics-biot_short_syntax (user-defined preconditioners) or navier_stokes-stokes_slip_bc (petsc solver setup).

8. How to get the linear system components: the matrix and the right-hand side?

To get the residual vector $\mathbf{r}$ (see Implementation of Essential Boundary Conditions) and the tangent matrix $\mathbf{K}$, the imperative API can be used as follows:
9. Where is the code that calculates the element (e.g. stiffness) matrix?

The code that computes the per element residuals and matrices is organized in terms, see Term Overview - click on the term class name and then “source” link to see the code. The original terms are implemented in C, newer terms tend to be implemented directly in Python. The structure and attributes of a term class are described in How to Implement a New Term.

10. What structural elements (beams, shells, etc.) are available in SfePy?

The code is currently focused on solid elements. The only supported structural element is shell10x, see linear_elasticity-shell10x_cantilever.

1.6.2 Mesh-Related Tasks

1. Checking and fixing a mesh (double vertices, disconnected components, etc.).
   - Show the mesh Euler characteristic, number of components and other information:
     ```bash
sfepy-mesh info -d cylinder.mesh
     
     Fix double/disconnected vertices:
     ```
     ```bash
     sfepy-convert -m bad.mesh maybe-good.mesh
     ```

2. Convert a mesh to another format (as supported by meshio).
   - Simple conversion:
     ```bash
     sfepy-convert mesh.format1 mesh.format2
     ```
   - Scaling the mesh anisotropically:
     ```bash
     sfepy-convert -s 2,4,3 cylinder.mesh cylinder-scaled.mesh
     ```

3. Verify that regions are correctly defined.
   - Using the problem description files (declarative API):
     ```bash
     sfepy-run sfepy/examples/diffusion/poisson_short_syntax.py --save-regions-as-groups --solve-not
     
     sfepy-view -e cylinder_regions.vtk
     ```
   - In a script (imperative API):
4. Remove lower-dimensional entities from a mesh (e.g. edges).

   Use `sfepy-convert` with the `-d <dimension>` option, where `<dimension>` is the topological dimension of cells that should be in the mesh. For example, `-d 2` stores only the 2D cells.

5. It is suggested to use msh22 format instead of the default msh4 when generating a mesh with `gmsh`:

   ```
gmsh -2 cylinder.geo -o cylinder.msh -format msh22
   ```

   msh22 seems to be more reliable and foolproof when converting.

### 1.6.3 Regions

1. How to define a region using a function of coordinates in the interactive mode (imperative API)?

   Examples:

   • A facet region defined using a function of mesh vertex coordinates:

   ```python
   from sfepy.discrete import Function, Functions

   def _get_region(coors, domain=None):
       ii = np.nonzero(coors[:,0] < 0.5)[0]
       return ii

   get_region = Function('get_region', _get_region)
   region = domain.create_region('Region', 'vertices by get_region', 'facet',
                                  functions=Functions([get_region]),
                                  )
   ```

   • Analogously a cell region defined using the coordinates of cell centroids:

   ```python
   # ...
   region = domain.create_region('Region', 'cells by get_region', 'cell',
                                  functions=Functions([get_region]),
                                  )
   ```

### 1.6.4 Material Parameters

1. How to set material parameters per region in the interactive mode (imperative API)?

   Example: define `rho, D` to have different values in regions omega1, omega2:

   ```python
   m = Material('m', values={'rho': {'omega1': 2700, 'omega2': 6000},
                              'D': {'omega1': D1, 'omega2': D2}}
   ```

2. How to implement state dependent materials?

   Besides writing a custom solver, one can use pseudo-time-stepping for this purpose, as demonstrated in linear_elasticity-material_nonlinearity or diffusion-poisson_field_dependent_material. Note that the examples are contrived, and in practice care must be taken to ensure convergence.
3. Why are results of a 2D elasticity simulation not consistent with a properly constrained 3D elasticity simulation?

Possible reason: when using the Young’s modulus and Poisson’s ratio as input parameters, and then calling `stiffness_from_youngpoisson()`, note that the default value of the plane argument is ‘strain’, corresponding to the plane strain assumption, see also `lame_from_youngpoisson()`. Try setting `plane='stress'`.

4. How to set (time-dependent) material parameters by a function in the interactive mode (imperative API)?

Example (also showing the full material function signature):

```python
from sfepy.discrete import Material, Function

def get_pars(ts, coors, mode=None, equations=None, term=None, problem=None, **kwargs):
    value1 = a_function(ts.t, coors)
    value2 = another_function(ts.step, coors)
    if mode == 'qp':
        out = {
            'value1': value1.reshape(coors.shape[0], 1, 1),
            'value2': value2.reshape(coors.shape[0], 1, 1),
        }
    return out

m = Material('m', function=Function('get_pars', get_pars))
```

5. How to get cells corresponding to coordinates in a material function?

The full signature of the material function is:

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

Thus it has access to `term.region.cells`, hence access to the cells that correspond to the coordinates. The length of the coors is `n_cell * n_qp`, where `n_qp` is the number of quadrature points per cell, and `n_cell = len(term.region.cells)`, so that `coors.reshape((n_cell, n_qp, -1))` can be used.

### 1.7 Theoretical Background

This part introduces parts the theoretical mathematical background necessary to use SfePy effectively. It also discusses some implementation choices done in SfePy.

Contents:

#### 1.7.1 Notes on solving PDEs by the Finite Element Method

The Finite Element Method (FEM) is the numerical method for solving Partial Differential Equations (PDEs). FEM was developed in the middle of XX. century and now it is widely used in different areas of science and engineering, including mechanical and structural design, biomedicine, electrical and power design, fluid dynamics and other. FEM is based on a very elegant mathematical theory of weak solution of PDEs. In this section we will briefly discuss basic ideas underlying FEM.
Strong form of Poisson’s equation and its integration

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

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

(1.11)

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

(1.12)

$$\nabla T \cdot n = g(x), \quad x \in \Gamma_N,$$

(1.13)

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 (1.11) is

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

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

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

$$= -\int_\Omega \nabla T \cdot \nabla s \, d\Omega + \int_\Omega \nabla \cdot (\nabla T \cdot s) \, d\Omega - \int_\Omega f \cdot s \, d\Omega$$

and applying Gauss theorem we obtain:

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

or

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

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

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

(1.14)

The equation (1.14) is the initial weak form of the Poisson’s problem (1.11)–(1.13). But we can not work with it without applying the boundary conditions. So it is time to talk about the boundary conditions.

Dirichlet Boundary Conditions

On the Dirichlet part of the surface we have two restrictions. One is the Dirichlet boundary conditions $T(x) = u(x)$ as they are, and the second is the integral term over $\Gamma_D$ in equation (1.14). To be consistent we have to use only the Dirichlet conditions and avoid the integral term. To implement this we can take the function $T \in V(\Omega)$ and the test function $s \in V_0(\Omega)$, where

$$V(\Omega) = \{v(x) \in H^1(\Omega)\},$$

$$V_0(\Omega) = \{v(x) \in H^1(\Omega) : v(x) = 0 \text{ on } \Gamma_D\}.$$
\[ V_0(\Omega) = \{ v(x) \in H^1(\Omega); v(x) = 0, x \in \Gamma_D \}. \]

In other words the unknown function \( T \) must be continuous together with its gradient in the domain. In contrast the test function \( s \) must be also continuous together with its gradient in the domain but it should be zero on the surface \( \Gamma_D \).

With this requirement the integral term over Dirichlet part of the surface is vanishing and the weak form of the Poisson equation for \( T \in V(\Omega) \) and \( s \in V_0(\Omega) \) becomes

\[
\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_N} s \cdot (\nabla T \cdot \mathbf{n}) \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \\
T(x) = u(x), \quad x \in \Gamma_D.
\]

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

\**Neumann Boundary Conditions**

The Neumann boundary conditions correspond to the known flux \( g(x) = \nabla T \cdot \mathbf{n} \). The integral term over the Neumann surface in the equation (1.14) contains exactly the same flux. So we can use the known function \( g(x) \) in the integral term:

\[
\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega,
\]

where test function \( s \) also belongs to the space \( V_0 \).

That is why Neumann conditions in FEM terminology are called \textbf{Natural Boundary Conditions}. These conditions are a part of weak form terms.

\**The weak form of the Poisson’s equation**

Now we can write the resulting weak form for the Poisson’s problem (1.11)–(1.13). For any test function \( s \in V_0(\Omega) \) find \( T \in V(\Omega) \) such that

\[
\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega = \int_{\Gamma_N} g \cdot s \, d\Gamma - \int_{\Omega} f \cdot s \, d\Omega, \quad T(x) = u(x), \quad x \in \Gamma_D.
\]

\textbf{(1.15)}

\**Discussion of discretization and meshing**

It is planned to have an example of the discretization based on the Poisson’s equation weak form (1.15). For now, please refer to the wikipedia page \textbf{Finite Element Method} for a basic description of the discretization and meshing.

\**Numerical solution of the problem**

To solve numerically given problem based on the weak form (1.15) we have to go through 5 steps:

1. Define geometry of the domain \( \Omega \) 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 $\int_{\Omega} \nabla T \cdot \nabla s \, d\Omega, \int_{\Gamma_N} g \cdot s \, d\Gamma, \int_{\Omega} f \cdot s \, d\Omega.$

1.7.2 Implementation of Essential Boundary Conditions

The essential boundary conditions can be applied in several ways. Here we describe the implementation used in SfePy.

Motivation

Let us solve a linear system $Ax = b$ with $n \times n$ matrix $A$ with $n_f$ values in the $x$ vector known. The known values can be for example EBC values on a boundary, if $A$ comes from a PDE discretization. If we put the known fixed values into a vector $x_f$, that has the same size as $x$, and has zeros in positions that are not fixed, we can easily construct a $n \times n_r$ matrix $T$ that maps the reduced vector $x_r$ of size $n_r = n - n_f$, where the fixed values are removed, to the full vector $x$:

$$ x = T x_r + x_f. $$

With that the reduced linear system with a $n_r \times n_r$ can be formed:

$$ T^T A T x_r = T^T (b - A x_f) $$

that can be solved by a linear solver. We can see, that the (non-zero) known values are now on the right-hand side of the linear system. When the known values are all zero, we have simply

$$ T^T A T x_r = T^T b, $$

which is convenient, as it allows simply throwing away the $A$ and $b$ entries corresponding to the known values already during the finite element assembling.

Implementation

All PDEs in SfePy are solved in a uniform way as a system of non-linear equations

$$ f(u) = 0, $$

where $f$ is the nonlinear function and $u$ the vector of unknown DOFs. This system is solved iteratively by the Newton method

$$ u^{new} = u^{old} - (\frac{df}{du^{old}})^{-1} f(u^{old}) $$

until a convergence criterion is met. Each iteration involves solution of the system of linear equations

$$ K \Delta u = r, $$

where the tangent matrix $K$ and the residual $r$ are

$$ K \equiv \frac{df}{du^{old}}, $$

$$ r \equiv f(u^{old}). $$
Then

\[ u^{\text{new}} = u^{\text{old}} - \Delta u . \]

If the initial (old) vector \( u^{\text{old}} \) contains the values of EBCs at correct positions, the increment \( \Delta u \) is zero at those positions. This allows us to assemble directly the reduced matrix \( T^T K T \), the right-hand side \( T^T r \), and ignore the values of EBCs during assembling. The EBCs are satisfied automatically by applying them to the initial guess \( u^0 \), that is given to the Newton solver.

**Linear Problems**

For linear problems we have

\[ f(u) \equiv Au - b = 0 , \]
\[ \frac{df}{du} = A , \]

and so the Newton method converges in a single iteration:

\[ u^{\text{new}} = u^{\text{old}} - A^{-1}(Au^{\text{old}} - b) = A^{-1}b . \]

**Evaluation of Residual and Tangent Matrix**

The evaluation of the residual \( f \) as well as the tangent matrix \( K \) within the Newton solver proceeds in the following steps:

- The EBCs are applied to the full DOF vector \( u \).
- The reduced vector \( u_{r} \) is passed to the Newton solver.
- Newton iteration loop:
  - Evaluation of \( f_{r} \) or \( K_{r} \):
    1. \( u \) is reconstructed from \( u_{r} \);
    2. local element contributions are evaluated using \( u \);
    3. local element contributions are assembled into \( f_{r} \) or \( K_{r} \) - values corresponding to fixed DOF positions are thrown away.
  - The reduced system \( K_{r} \Delta u_{r} = r_{r} \) is solved.
  - Solution is updated: \( u_{r} \leftarrow u_{r} - \Delta u_{r} . \)
  - The loop is terminated if a stopping condition is satisfied, the solver returns the final \( u_{r} \).
- The final \( u \) is reconstructed from \( u_{r} \).

**1.8 Term Overview**

**1.8.1 Term Syntax**

In general, the syntax of a term call is:

\[ \langle \text{term name}\rangle.<i>.<r>(\langle \text{arg1}\rangle,\langle \text{arg2}\rangle,\ldots) , \]
where \(<i>\) denotes an integral name (i.e. a name of numerical quadrature to use) and \(<r>\) marks a region (domain of the integral).

The following notation is used:

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Omega)</td>
<td>cell (volume) (sub)domain</td>
</tr>
<tr>
<td>(\Gamma)</td>
<td>facet (surface) (sub)domain</td>
</tr>
<tr>
<td>(D)</td>
<td>cell or facet (sub)domain</td>
</tr>
<tr>
<td>(d)</td>
<td>dimension of space</td>
</tr>
<tr>
<td>(t)</td>
<td>time</td>
</tr>
<tr>
<td>(y)</td>
<td>any function</td>
</tr>
<tr>
<td>(\mathbf{y})</td>
<td>any vector function</td>
</tr>
<tr>
<td>(n)</td>
<td>unit outward normal</td>
</tr>
<tr>
<td>(q)</td>
<td>scalar test or parameter function</td>
</tr>
<tr>
<td>(p)</td>
<td>scalar unknown or parameter function</td>
</tr>
<tr>
<td>(\mathbf{z})</td>
<td>vector test or parameter function</td>
</tr>
<tr>
<td>(\mathbf{w}, \mathbf{u})</td>
<td>vector unknown or parameter function</td>
</tr>
<tr>
<td>(\varepsilon(\mathbf{u}))</td>
<td>Cauchy strain tensor (\frac{1}{2}((\nabla \mathbf{u}) + (\nabla \mathbf{u})^T))</td>
</tr>
<tr>
<td>(\mathbf{J})</td>
<td>deformation gradient (\mathbf{F}_{ij} = \frac{\partial \mathbf{u}_i}{\partial X_j})</td>
</tr>
<tr>
<td>(\mathbf{C})</td>
<td>right Cauchy-Green deformation tensor (\mathbf{C} = \mathbf{F}^T \mathbf{F})</td>
</tr>
<tr>
<td>(\varepsilon(\mathbf{u}))</td>
<td>Green strain tensor (\varepsilon_{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))</td>
</tr>
<tr>
<td>(\mathbf{S})</td>
<td>second Piola-Kirchhoff stress tensor</td>
</tr>
<tr>
<td>(\mathbf{F})</td>
<td>vector volume forces</td>
</tr>
<tr>
<td>(\mathbf{f})</td>
<td>scalar volume force (source)</td>
</tr>
<tr>
<td>(\rho)</td>
<td>density</td>
</tr>
<tr>
<td>(\nu)</td>
<td>kinematic viscosity</td>
</tr>
<tr>
<td>(c, \xi, \zeta)</td>
<td>any constant</td>
</tr>
<tr>
<td>(\delta_{ij}, \mathbf{I})</td>
<td>Kronecker delta, identity matrix</td>
</tr>
<tr>
<td>(\text{tr} \bullet)</td>
<td>trace of a second order tensor (\sum_{i=1}^d \bullet_{ii})</td>
</tr>
<tr>
<td>(\text{dev} \bullet)</td>
<td>deviator of a second order tensor (\bullet - \frac{1}{3} \text{tr} \bullet \mathbf{I})</td>
</tr>
<tr>
<td>(T_K \in \mathcal{T}_h)</td>
<td>(K)-th element of triangulation (= mesh) (\mathcal{T}_h) of domain (\Omega)</td>
</tr>
<tr>
<td>(K \leftarrow \mathcal{I}_h)</td>
<td>(K) is assigned values from ({0, 1, \ldots, N_h - 1} \equiv \mathcal{I}_h) in ascending order</td>
</tr>
</tbody>
</table>

The suffix “\(0\)” denotes a quantity related to a previous time step.

Term names are (usually) prefixed according to the following conventions:

<table>
<thead>
<tr>
<th>prefix</th>
<th>meaning</th>
<th>evaluation modes</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{dw})</td>
<td>discrete weak</td>
<td>‘weak’</td>
<td>terms having a virtual (test) argument and zero or more unknown arguments, used for FE assembling</td>
</tr>
<tr>
<td>(\text{ev})</td>
<td>evaluate</td>
<td>‘eval’, ‘el_eval’, ‘el_avg’, ‘qp’</td>
<td>terms having all arguments known, modes ‘el_avg’, ‘qp’ are not supported by all (\text{ev}) terms</td>
</tr>
<tr>
<td>(\text{de})</td>
<td>discrete einsum</td>
<td>any (work in progress)</td>
<td>multi-linear terms defined using an enriched einsum notation</td>
</tr>
</tbody>
</table>

Evaluation modes ‘eval’, ‘el_avg’ and ‘qp’ are defined as follows:
### Table 4: Evaluation modes.

<table>
<thead>
<tr>
<th>mode</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘eval’</td>
<td>( \int_{\mathcal{D}} (\cdot) )</td>
</tr>
<tr>
<td>‘el_avg’</td>
<td>vector for ( K \leftarrow \mathcal{I}<em>h : \int</em>{T_K} (\cdot) / \int_{T_K} 1 )</td>
</tr>
<tr>
<td>‘qp’</td>
<td>((\cdot)</td>
</tr>
</tbody>
</table>

#### 1.8.2 Term Table

Below we list all the terms available in automatically generated tables. The first column lists the name, the second column the argument lists and the third column the mathematical definition of each term. The terms are divided into the following tables:

- *Table of basic terms*
- *Table of large deformation terms* (total/updated Lagrangian formulation)
- *Table of sensitivity terms*
- *Table of special terms*
- *Table of multi-linear terms*

The notation `<virtual>` corresponds to a test function, `<state>` to a unknown function and `<parameter>` to a known function. By `<material>` we denote material (constitutive) parameters, or, in general, any given function of space and time that parameterizes a term, for example a given traction force vector.
Table of basic terms

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_advect_div_free</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$\int_{\Omega} \nabla \cdot (yp)q = \int_{\Omega} ((\nabla \cdot y) + y \cdot \nabla)p)q$</td>
<td>tim.adv.dif</td>
</tr>
<tr>
<td>AdvectDivFreeTerm</td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_bc_newton</td>
<td><code>&lt;material_1&gt;</code>, <code>&lt;material_2&gt;</code></td>
<td>$\int_{\Gamma} \alpha q(p - p_{outer})$</td>
<td>tim.hea.equ.mul.mat</td>
</tr>
<tr>
<td>BCNewtonTerm</td>
<td><code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_biot</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual/param_v&gt;</code>, <code>&lt;state/param_s&gt;</code></td>
<td>$\int_{\Omega} p \alpha_{ij} e_{ij}(u) \cdot \int_{\Omega} q \alpha_{ij} e_{ij}(u)$</td>
<td>bio, the.ela, bio.npb.lag, the.ela.ess, bio.sho.syn, bio.npb</td>
</tr>
<tr>
<td>BiotTerm</td>
<td><code>&lt;material&gt;</code>, <code>&lt;state&gt;</code>, <code>&lt;virtual&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_biot_stress</td>
<td><code>&lt;material&gt;</code>, <code>&lt;parameter&gt;</code></td>
<td>$-\int_{\Omega} \alpha_{ij} p$</td>
<td></td>
</tr>
<tr>
<td>BiotStressTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_cauchy_strain</td>
<td><code>&lt;parameter&gt;</code></td>
<td>$\int_{\mathcal{D}} e(w)$</td>
<td></td>
</tr>
<tr>
<td>CauchyStrainTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_cauchy_stress</td>
<td><code>&lt;material&gt;</code>, <code>&lt;parameter&gt;</code></td>
<td>$\int_{\mathcal{D}} D_{ijkl} e_{kl}(w)$</td>
<td></td>
</tr>
<tr>
<td>CauchyStressTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_contact</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td>$\int_{\Gamma} \varepsilon_{N}(g_N(u)) \n w$</td>
<td>two.bod.con</td>
</tr>
<tr>
<td>ContactTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_contact_plane</td>
<td><code>&lt;material_1&gt;</code>, <code>&lt;material_2&gt;</code>, <code>&lt;material_3&gt;</code></td>
<td>$\int_{\Gamma} v \cdot f(d(u)) \n$</td>
<td>ela.con.pla</td>
</tr>
<tr>
<td>ContactPlaneTerm</td>
<td><code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_contact_sphere</td>
<td>&lt;material_</td>
<td>$\int_{\Gamma} \mathbf{v} \cdot f(d(\mathbf{u})) \mathbf{n}(\mathbf{u})$</td>
<td>ela.con.sph</td>
</tr>
<tr>
<td>ContactSphereTerm</td>
<td>&lt;material_</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;material_</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_convect</td>
<td>&lt;virtual&gt;,</td>
<td>$\int_{\Omega} (\mathbf{u} \cdot \nabla \mathbf{u}) \cdot \mathbf{v}$</td>
<td>nav.sto,</td>
</tr>
<tr>
<td>ConvectTerm</td>
<td>&lt;state&gt;</td>
<td></td>
<td>nav.sto.iga,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nav.sto</td>
</tr>
<tr>
<td>dw_convect_v_grad</td>
<td>&lt;virtual&gt;,</td>
<td>$\int_{\Omega} q(\mathbf{u} \cdot \nabla p)$</td>
<td>poi.fun</td>
</tr>
<tr>
<td>ConvectVGradSTe</td>
<td>&lt;state_v&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state_s&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_def_grad</td>
<td>&lt;parameter&gt;</td>
<td>$\mathbf{F} = \frac{\partial x}{\partial \mathbf{X}}</td>
<td>_{qp} = \mathbf{I} + \frac{\partial \mathbf{u}}{\partial \mathbf{X}}</td>
</tr>
<tr>
<td>DeformationGrad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_dg_advect_lax</td>
<td>&lt;opt_material&gt;</td>
<td>$\int_{\partial T_K} \mathbf{n} \cdot f^*(p_{in}, p_{out}) q$</td>
<td>adv.2D, adv.1D,</td>
</tr>
<tr>
<td>AdvectionDGFlux</td>
<td>&lt;material_</td>
<td></td>
<td>adv.dif.2D</td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_dg_diffusion_f</td>
<td>&lt;material_</td>
<td>$\int_{\partial T_K} D(\nabla p)[q] , \int_{\partial T_K} D(\nabla q)[p]$</td>
<td>bur.2D,</td>
</tr>
<tr>
<td>DiffusionDGFlux</td>
<td>&lt;state&gt;,</td>
<td></td>
<td>adv.dif.2D,</td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;,</td>
<td></td>
<td>lap.2D</td>
</tr>
<tr>
<td></td>
<td>&lt;material_</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_dg_interior_penalty</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>( \int_{\partial T_K} D C_n \frac{\partial^2}{\partial (\nabla T_K)} [p][q] ) where ([\phi] = \phi_{in} - \phi_{out})</td>
<td>bur.2D, adv.dif.2D, lap.2D</td>
</tr>
<tr>
<td>NonlinearHyperbolic</td>
<td>&lt;opt_material&gt;, &lt;fun&gt;, &lt;fun_d&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>( \int_{\partial T_K} \mathbf{n} \cdot f^<em>(p_{in}, p_{out}) q ) where ( f^</em>(p_{in}, p_{out}) = \frac{f(p_{in}) + f(p_{out})}{2} + (1 - \alpha) n C (p_{in} - p_{out}) )</td>
<td>bur.2D</td>
</tr>
<tr>
<td>DiffusionTerm</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>( \int_{\Omega} K_{ij} \nabla_i q \nabla_j p )</td>
<td>bio, vib.aco, pie.ela, poi.neu, bio.npb.lag, pie.ela, bio.sho.syn, bio.npb, dar.flo.mul</td>
</tr>
<tr>
<td>DiffusionCoupling</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt; &lt;material&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>( \int_{\Omega} p K_{ij} \nabla_j q ), ( \int_{\Omega} q K_{ij} \nabla_j p )</td>
<td></td>
</tr>
<tr>
<td>DiffusionRTerm</td>
<td>&lt;material&gt;, &lt;virtual&gt;</td>
<td>( \int_{\Omega} K_{ij} \nabla_j q )</td>
<td></td>
</tr>
<tr>
<td>DiffusionVelocityTerm</td>
<td>&lt;material&gt;, &lt;parameter&gt;</td>
<td>( - \int_{\partial} K_{ij} \nabla_j p )</td>
<td></td>
</tr>
<tr>
<td>DivOperatorTerm</td>
<td>&lt;opt_material&gt;, &lt;virtual&gt;</td>
<td>( \int_{\Omega} \nabla \cdot \mathbf{u} ) or ( \int_{\Omega} c \nabla \cdot \mathbf{u} )</td>
<td></td>
</tr>
<tr>
<td>name/class</td>
<td>arguments</td>
<td>definition</td>
<td>examples</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ev_div</td>
<td>&lt;opt_material&gt;, &lt;parameter&gt;</td>
<td>$\int_D \nabla \cdot u$, $\int_D c \nabla \cdot u$</td>
<td>sto.sli.bc, nav.sto, sto, nav.sto.iga, nav.sto, sta.nav.sto</td>
</tr>
<tr>
<td>dw_div_grad</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>$\int_\Omega \nabla v : \nabla u$, $\int_\Omega \nabla v : \nabla u$</td>
<td>lin.ela.dam, hyd, wel, adv.2D, lin.ela.up, tim.adv.dif, tim.poi.exp,</td>
</tr>
<tr>
<td>dw_dot</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>$\int_D q_p$, $\int_D v \cdot u$</td>
<td>tim.hea.equ.mul.mat, adv.1D, mod.ana.dec, dar.flo.mul, tim.poi, hel.apa,</td>
</tr>
<tr>
<td>dw_elastic_wave</td>
<td>&lt;material_1&gt;, &lt;material_2&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega D_{ijkl} g_{ij}(v) g_{kl}(u)$</td>
<td>the.ele, sto.sli.bc, the.ele, aco, bur.2D</td>
</tr>
<tr>
<td>dw_elastic_wave_c</td>
<td>&lt;material_1&gt;, &lt;material_2&gt;, &lt;state&gt;, &lt;material_3&gt;, &lt;material_4&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>$\int_\Omega D_{ijkl} g_{ij}(v) c_{kl}(u)$</td>
<td></td>
</tr>
<tr>
<td>dw_electric_source</td>
<td>&lt;material_1&gt;, &lt;virtual&gt;, &lt;parameter&gt;</td>
<td>$\int_\Omega c s (\nabla \phi)^2$</td>
<td>the.ele</td>
</tr>
<tr>
<td>name/class</td>
<td>arguments</td>
<td>definition</td>
<td>examples</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ev_grad</td>
<td>&lt;opt_material&gt;, &lt;parameter&gt;</td>
<td>$\int_\mathcal{D} \nabla p$ or $\int_\mathcal{D} \nabla u$ $\int_\mathcal{D} c\nabla p$ or $\int_\mathcal{D} c\nabla u$</td>
<td></td>
</tr>
<tr>
<td>ev_integrate</td>
<td>&lt;opt_material&gt;, &lt;parameter&gt;</td>
<td>$\int_\mathcal{D} y$, $\int_\mathcal{D} y$, $\int_\Gamma y \cdot n$ $\int_\mathcal{D} cy$, $\int_\mathcal{D} cy$, $\int_\Gamma cy \cdot n$ flux</td>
<td></td>
</tr>
<tr>
<td>dw_integrate</td>
<td>&lt;opt_material&gt;, &lt;virtual&gt;</td>
<td>$\int_\mathcal{D} q$ or $\int_\mathcal{D} cq$</td>
<td>aco, vib.aco, hel.apa, tim.hea.equ.mul.mat, poi.neu, aco, dar.flo.mul, poi.per.bou.con</td>
</tr>
<tr>
<td>ev_integrate_mat</td>
<td>&lt;material&gt;, &lt;parameter&gt;</td>
<td>$\int_\mathcal{D} c$</td>
<td></td>
</tr>
<tr>
<td>dw_jump</td>
<td>&lt;opt_material&gt;, &lt;virtual&gt;, &lt;state_1&gt;, &lt;state_2&gt;</td>
<td>$\int_\Gamma cq(p_1 - p_2)$</td>
<td>aco</td>
</tr>
</tbody>
</table>

continues on next page
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_laplace</td>
<td>&lt;opt_mat&lt;virtual/, &lt;state/</td>
<td>[ \int_{\Omega} c\nabla q \cdot \nabla p ]</td>
<td>hyd, poi.par.stu, lap.2D, wel, tim.adv.dif, tim.poi.exp, the.ela.ess, lap.1d, tim.hea.equat.mul.mat, tim.poi, hel.apa, osc, vib.aco, adv.dif.2D, lap.flu.2d, lap.cou.1cb, lap.tim.abc, poi.fie.depm.mat, por, ref.evp, poi.fun, sin, poi.per.bou.con, poi, aco, sto.sli.bc, poi.sho.syn, the.ela, aco, bur.2D, cub, poi.iga sta.nav.sto</td>
</tr>
<tr>
<td>dw_lin_convect</td>
<td>&lt;virtual&gt;, &lt;parameter, &lt;state&gt;</td>
<td>[ \int_{\Omega} ((w \cdot \nabla u) \cdot v) ]</td>
<td>[ ((w \cdot \nabla u)_{qp} ]</td>
</tr>
<tr>
<td>dw_lin_convect2</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_{\Omega} ((c \cdot \nabla u) \cdot v) ]</td>
<td>[ ((c \cdot \nabla u)_{qp} ]</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_lin_elastic</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;,</td>
<td>( \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u}) )</td>
<td>lin.ela.dam, tru.bri, the.ela,</td>
</tr>
<tr>
<td>LinearElasticTerm</td>
<td>&lt;state/param_2&gt;</td>
<td></td>
<td>its.2, pie.ela.mac, lin.ela.up,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the.ela.ess, lin.ela, bio.sho.syn,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nod.lcb, mod.ana.dec, two.bod.con,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bio, mix.mes, lin.el.iga, pie.ela,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its.3, el.ela.con.pla, el.ela.con.sph,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vib.aco, ela.shi.per, el.tra, lin.ela.mM, pie.ela, its.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mat.non, bio.npb, lin.el.tra, pie.ela,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its.4, sei.loa</td>
</tr>
<tr>
<td>dw_lin_elastic_iso</td>
<td>&lt;material_1&gt;, &lt;material_2&gt;,</td>
<td>( \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u}) )</td>
<td></td>
</tr>
<tr>
<td>LinearElasticIsotropic</td>
<td>&lt;virtual/param_1&gt;,</td>
<td>with ( D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) + \lambda \delta_{ij} \delta_{kl} )</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>&lt;state/param_2&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_lin_elastic_l_ad</td>
<td>&lt;material_1&gt;, &lt;material_2&gt;,</td>
<td>( \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{v}) e_{kl}(\mathbf{u}) )</td>
<td></td>
</tr>
<tr>
<td>LinearElasticLADTerm</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>with ( D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) + \lambda \delta_{ij} \delta_{kl} )</td>
<td></td>
</tr>
</tbody>
</table>
Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_lin_elastic_ypl</td>
<td>&lt;material_\ LinearElasticYPAd\ Term&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega D_{ijkl} e_{ij}(\textbf{u})e_{kl}(\textbf{u}) ] [ \text{with} ] [ D_{ijkl} = \mu (\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl}, ] [ \text{where} ] [ \lambda = E\nu/((1+\nu)(1-2\nu)), ] [ \mu = E/2(1+\nu) ]</td>
<td>ela.ide</td>
</tr>
<tr>
<td>dw_lin_prestress</td>
<td>&lt;material&gt;, &lt;virtual/param&gt;</td>
<td>[ \int_\Omega \sigma_{ij}e_{ij}(\textbf{u}) ]</td>
<td>pie.ela.mac, pre.fib, non.hyp.mM</td>
</tr>
<tr>
<td>dw_lin_spring</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ f^{(i)} = -f^{(j)} = k(u^{(j)} - u^{(i)}) ] [ \forall \ elements \ T_{K}^{i,j} ] [ \text{in a region connecting nodes } i, j ]</td>
<td>pre.fib</td>
</tr>
<tr>
<td>dw_lin_strain_fib</td>
<td>&lt;material_\ LinearStrainFiber\ Term&gt;, &lt;material&gt;, &lt;virtual&gt;</td>
<td>[ \int_\Omega D_{ijkl} e_{ij}(\textbf{u})(d_{k}d_{l}) ]</td>
<td></td>
</tr>
<tr>
<td>dw_lin_truss</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ F^{(i)} = -F^{(j)} = EA/l(U^{(j)} - U^{(i)}) ] [ \forall \ elements \ T_{K}^{i,j} ] [ \text{in a region connecting nodes } i, j ]</td>
<td>tru.bri, tru.bri</td>
</tr>
<tr>
<td>ev_lin_truss_force</td>
<td>&lt;material_\ LinearTrussInternal\ Force\ Term&gt;, &lt;parameter&gt;</td>
<td>[ F = EA/l(U^{(j)} - U^{(i)}) ] [ \forall \ elements \ T_{K}^{i,j} ] [ \text{in a region connecting nodes } i, j ]</td>
<td></td>
</tr>
<tr>
<td>dw_mass_ad</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_{\mathcal{D}} \rho \textbf{v} \cdot \textbf{u} ]</td>
<td>ela.ide</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_nl_diffusion</td>
<td><code>&lt;fun&gt;</code>, <code>&lt;dfun&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td>$\int_\Omega \nabla q \cdot \nabla pf(p)$</td>
<td>poi.non.mat</td>
</tr>
<tr>
<td>dw_non_penetration</td>
<td><code>&lt;opt_material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code>, <code>&lt;opt_material&gt;</code>, <code>&lt;state&gt;</code>, <code>&lt;virtual&gt;</code></td>
<td>$\int_\Gamma c\lambda n \cdot v \cdot \int_\Gamma c\lambda n \cdot u \cdot \int_\Gamma \lambda n \cdot v \cdot \int_\Gamma \lambda n \cdot u$</td>
<td>bio.npb.lag</td>
</tr>
<tr>
<td>dw_non_penetration_p</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td>$\int_\Gamma c(n \cdot v)(n \cdot u)$</td>
<td>bio.sho.syn</td>
</tr>
<tr>
<td>dw_nonsym_elastic</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual/param_1&gt;</code>, <code>&lt;state/param_2&gt;</code></td>
<td>$\int_\Omega D \nabla u : \nabla v$</td>
<td>non.hyp.mM</td>
</tr>
<tr>
<td>dw_ns_dot_grad_s</td>
<td><code>&lt;fun&gt;</code>, <code>&lt;fun_d&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code>, <code>&lt;fun&gt;</code>, <code>&lt;fun_d&gt;</code>, <code>&lt;state&gt;</code>, <code>&lt;virtual&gt;</code></td>
<td>$\int_\Omega q \cdot \nabla f(p) = \int_\Omega q \cdot \text{div} f(p) \cdot \int_\Omega f(p) \cdot \nabla q$</td>
<td>bur.2D</td>
</tr>
<tr>
<td>dw_piezo_coupling</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual/param_v&gt;</code>, <code>&lt;state/param_s&gt;</code></td>
<td>$\int_\Omega g^{kij} e_{ij}(v) \nabla k p \cdot g^{kij} e_{ij}(u) \nabla k q$</td>
<td>pie.ela, pie.ela</td>
</tr>
<tr>
<td>ev_piezo_strain</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code></td>
<td>$\int_\Omega g^{kij} e_{ij}(u)$</td>
<td></td>
</tr>
<tr>
<td>ev_piezo_stress</td>
<td><code>&lt;material&gt;</code>, <code>&lt;parameter&gt;</code></td>
<td>$\int_\Omega g^{kij} \nabla k p$</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_point_load</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code></td>
<td>$f^i = \bar{f}^i \quad \forall$ FE node $i$ in a region</td>
<td>its.2, tru.bri,</td>
</tr>
<tr>
<td><strong>ConcentratedPointLoad</strong></td>
<td></td>
<td></td>
<td>its.1, she.can,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its.4, its.3</td>
</tr>
<tr>
<td>dw_point_lspring</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$f^i = -k\bar{u}^i \quad \forall$ FE node $i$ in a region</td>
<td></td>
</tr>
<tr>
<td><strong>LinearPointSpring</strong></td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_s_dot_grad_i</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$Z^i = \int_\Omega q \nabla p$</td>
<td></td>
</tr>
<tr>
<td><strong>ScalarDotGrad</strong></td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_s_dot_mgrad_i</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$\int_\Omega qy \cdot \nabla p$, $\int_\Omega py \cdot \nabla q$</td>
<td>adv.2D, adv.1D,</td>
</tr>
<tr>
<td><strong>ScalarDotMGrad</strong></td>
<td><code>&lt;state&gt;</code>, <code>&lt;material&gt;</code></td>
<td></td>
<td>adv.dif.2D</td>
</tr>
<tr>
<td>dw_shell10x</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$\int_\Omega D_{ijkl} e_{ij}(\bar{u}) e_{kl}(\bar{u})$</td>
<td>she.can</td>
</tr>
<tr>
<td><strong>Shell10XTerm</strong></td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_stokes</td>
<td><code>&lt;opt_material&gt;</code>, <code>&lt;virtual/</code></td>
<td>$\int_\Omega p \nabla \cdot \bar{u}$, $\int_\Omega q \nabla \cdot u$</td>
<td>sto.sli.bc, lin.ela.up, nav.sto, sto, nav.sto.iga, nav.sto, sta.nav.sto</td>
</tr>
<tr>
<td><strong>StokesTerm</strong></td>
<td><code>&lt;param_v&gt;</code>, <code>&lt;state/</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;param_s&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_stokes_wave</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$\int_\Omega (\kappa \cdot \bar{v})(\kappa \cdot u)$</td>
<td></td>
</tr>
<tr>
<td><strong>StokesWaveTerm</strong></td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_stokes_wave_d</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>,</td>
<td>$\int_\Omega (\kappa \cdot \bar{v})(\nabla \cdot u)$, $\int_\Omega (\kappa \cdot u)(\nabla \cdot \bar{v})$</td>
<td></td>
</tr>
<tr>
<td><strong>StokesWaveDivTerm</strong></td>
<td><code>&lt;state&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_sum_vals</td>
<td><code>&lt;parameter&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SumNodalValues</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_surface_flux</td>
<td>&lt;opt_material&gt;,</td>
<td>[ \int_{\Gamma} q u \cdot K \cdot \nabla p ]</td>
<td>mix.mes, tru.bri, el.a.ela.opt, el.a.shi.per, lin.vis, com.a.ela.mat, lin.a.ela.tra, nod.lcb</td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SurfaceFluxOpe</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_surface_flux</td>
<td>&lt;material&gt;</td>
<td>[ \int_{\Gamma} n \cdot K_{ij} \nabla_j p ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;parameter&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SurfaceFluxTerm</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_surface_ltr</td>
<td>&lt;opt_material&gt;,</td>
<td>[ \int_{\Gamma} v \cdot \sigma \cdot n, \int_{\Gamma} u \cdot n ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual/param&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>LinearTraction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_surface_moment</td>
<td>&lt;material&gt;</td>
<td>[ \int_{\Gamma} n(x-x_0) ]</td>
<td>lap.flu.2d</td>
</tr>
<tr>
<td></td>
<td>&lt;parameter&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SurfaceMoment</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_surface_ndot</td>
<td>&lt;material&gt;</td>
<td>[ \int_{\Gamma} q c \cdot n ]</td>
<td>vib.aco</td>
</tr>
<tr>
<td></td>
<td>&lt;virtual/param&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SurfaceNormalDo</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>VectorDotGradS</strong></td>
<td>[ \int_{\Omega} v \cdot \nabla p, \int_{\Omega} u \cdot \nabla q ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;opt_material&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual/param_v&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state/param_s&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>VectorDotGradS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;material&gt;</td>
<td>[ \int_{\Omega} c u \cdot \nabla q ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>VectorDotScalar</strong></td>
<td>[ \int_{\Omega} v \cdot c p, \int_{\Omega} u \cdot c q ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;material&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual/param_v&gt;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state/param_s&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>VectorDotScalar</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;material&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;state&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;virtual&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name/class</td>
<td>arguments</td>
<td>definition</td>
<td>examples</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ev_volume</td>
<td>&lt;parameter&gt;</td>
<td>( \int_{D}^{1} )</td>
<td></td>
</tr>
<tr>
<td>VolumeTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_volume_lvf</td>
<td>&lt;material&gt;, &lt;virtual&gt;</td>
<td>( \int_{\Omega}^{f \cdot u \text{ or } f q} )</td>
<td>bur.2D, adv.dif.2D, poi.par.stu, poi.iga</td>
</tr>
<tr>
<td>LinearVolumeFo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_volume_nvf</td>
<td>&lt;fun&gt;, &lt;dfun&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>( \int_{\Omega}^{q f(p)} )</td>
<td>poi.non.mat</td>
</tr>
<tr>
<td>NonlinearVolume</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_volume_surface</td>
<td>&lt;parameter&gt;</td>
<td>( 1/D \int_{\Gamma}^{x \cdot n} )</td>
<td>ela</td>
</tr>
<tr>
<td>VolumeSurfaceT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_zero</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ZeroTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Table of sensitivity terms

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_adj_convect1</td>
<td>&lt;virtual&gt;, &lt;state&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} (\langle \mathbf{u} \cdot \nabla \mathbf{u} \rangle \cdot \mathbf{w} ) ]</td>
<td></td>
</tr>
<tr>
<td>dw_adj_convect2</td>
<td>&lt;virtual&gt;, &lt;state&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} (\langle \mathbf{u} \cdot \nabla \mathbf{v} \rangle \cdot \mathbf{w} ) ]</td>
<td></td>
</tr>
<tr>
<td>dw_adj_div_grad</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;parameter&gt;</td>
<td>[ w \delta_u \Psi(u) \circ v ]</td>
<td></td>
</tr>
<tr>
<td>ev_sd_convect</td>
<td>&lt;parameter&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \left[ w_k \frac{\partial u_i}{\partial x_k} \left( \nabla \cdot \mathbf{v} \right) - w_k \frac{\partial v_i}{\partial x_k} \frac{\partial u_j}{\partial x_j} \right] ]</td>
<td></td>
</tr>
<tr>
<td>ev_sd_diffusion</td>
<td>&lt;material&gt;, &lt;parameter&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \hat{K}<em>{ij} \nabla_i q \nabla_j p ] [ \hat{K}</em>{ij} = K_{ij} \left( \delta_{ik} \delta_{jl} \nabla \cdot \mathbf{v} - \frac{\partial v_j}{\partial x_i} - \frac{\partial v_i}{\partial x_j} \right) ]</td>
<td></td>
</tr>
<tr>
<td>de_sd_diffusion</td>
<td>&lt;virtual&gt;, &lt;state&gt;, &lt;param_1&gt;, &lt;param_2&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \hat{K}<em>{ij} \nabla_i q \nabla_j p ] [ \hat{K}</em>{ij} = K_{ij} \left( \delta_{ik} \delta_{jl} \nabla \cdot \mathbf{v} - \frac{\partial v_j}{\partial x_i} - \frac{\partial v_i}{\partial x_j} \right) ]</td>
<td></td>
</tr>
<tr>
<td>ev_sd_div</td>
<td>&lt;parameter&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} p \left[ (\nabla \cdot \mathbf{u}) (\nabla \cdot \mathbf{v}) - \frac{\partial v_k}{\partial x_i} \frac{\partial w_i}{\partial x_k} \right] ]</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
Table 6 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ev_sd_div_grad</td>
<td>&lt;opt_material&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \hat{I} \nabla \cdot \nabla u, \int_{\Omega} \nu \hat{I} \nabla \cdot \nabla u ] [ \hat{I}<em>{ijkl} = \delta</em>{ik}\delta_{jl} \nabla \cdot \nabla \cdot - \delta_{ik}\delta_{k\ell} \frac{\partial \nu_i}{\partial x_\ell} - \delta_{i\ell}\delta_{jk} \frac{\partial \nu_j}{\partial x_\ell} ]</td>
<td></td>
</tr>
<tr>
<td>de_sd_div_grad</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \hat{I} \nabla \cdot \nabla u, \int_{\Omega} \nu \hat{I} \nabla \cdot \nabla u ] [ \hat{I}<em>{ijkl} = \delta</em>{ik}\delta_{jl} \nabla \cdot \nabla \cdot - \delta_{ik}\delta_{k\ell} \frac{\partial \nu_i}{\partial x_\ell} - \delta_{i\ell}\delta_{jk} \frac{\partial \nu_j}{\partial x_\ell} ]</td>
<td></td>
</tr>
<tr>
<td>ev_sd_dot</td>
<td>&lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} pq (\nabla \cdot \cdot) , \int_{\Omega} (u \cdot w)(\nabla \cdot \cdot) ]</td>
<td></td>
</tr>
<tr>
<td>de_sd_dot</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} qp (\nabla \cdot \cdot) , \int_{\Omega} (v \cdot u)(\nabla \cdot \cdot) ] [ \int_{\Omega} eqp (\nabla \cdot \cdot) , \int_{\Omega} e(v \cdot u)(\nabla \cdot \cdot) ] [ \int_{\Omega} v \cdot (M u)(\nabla \cdot \cdot) ]</td>
<td></td>
</tr>
<tr>
<td>de_sd_lin_elastic</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \dot{D}<em>{ijkl} e</em>{ij}(u) e_{kl}(u) ] [ \dot{D}<em>{ijkl} = D</em>{ijkl}(\nabla \cdot \cdot) - D_{ij\ell k} \frac{\partial \nu_i}{\partial x_\ell} - D_{ik\ell j} \frac{\partial \nu_i}{\partial x_\ell} ]</td>
<td></td>
</tr>
<tr>
<td>ev_sd_lin_elastic</td>
<td>&lt;material&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>[ \int_{\Omega} \dot{D}<em>{ijkl} e</em>{ij}(u) e_{kl}(u) ] [ \dot{D}<em>{ijkl} = D</em>{ijkl}(\nabla \cdot \cdot) - D_{ij\ell k} \frac{\partial \nu_i}{\partial x_\ell} - D_{ik\ell j} \frac{\partial \nu_i}{\partial x_\ell} ]</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
</tr>
</thead>
</table>
| ev_sd_piezo_coupl  | <material>, <parameter>, <parameter>, <parameter>                        | \( \int_{\Omega} \hat{g}_{ki} e_{ij}(u) \nabla k p \) \[
\hat{g}_{ki} = g_{ki}(\nabla \cdot \nabla) - g_{ki} \frac{\partial V_j}{\partial x_l} - g_{ij} \frac{\partial V_k}{\partial x_l} \]
| de_sd_piezo_coupl  | <material>, virtual/param_v>, state/param_s>, <parameter>, <material> <state>, <virtual>, <parameter> | \( \int_{\Omega} \hat{g}_{ki} e_{ij}(u) \nabla k p \), \( \int_{\Omega} \hat{g}_{ki} e_{ij}(u) \nabla k q \) \[
\hat{g}_{ki} = g_{ki}(\nabla \cdot \nabla) - g_{ki} \frac{\partial V_j}{\partial x_l} - g_{ij} \frac{\partial V_k}{\partial x_l} \]
| de_sd_stokes       | <opt_material>, virtual/param_v>, state/param_s>, <parameter>, <opt_material> <state>, <virtual>, <parameter> | \( \int_{\Omega} p \hat{I}_{ij} \frac{\partial V_i}{\partial x_j} \), \( \int_{\Omega} q \hat{I}_{ij} \frac{\partial u_i}{\partial x_j} \) \[
\hat{I}_{ij} = \delta_{ij} \nabla \cdot \nabla - \frac{\partial V_j}{\partial x_i} \]
| ev_sd_surface_inte | <parameter>, <parameter>                                                  | \( \int_{\Gamma} p \nabla \cdot \nabla \)                                  |
| de_sd_surface_ltr  | <opt_material>, virtual/param>, <parameter>                               | \( \int_{\Gamma} u \cdot \left[ (\hat{\sigma} \nabla \cdot \nabla - \hat{\sigma} \nabla \nabla) u \right] \) \[
\hat{\sigma} = I \cdot \hat{\sigma} = c I \text{ or } \hat{\sigma} = \sigma \]
<p>| ev_sd_surface_ltr  | &lt;opt_material&gt;, &lt;parameter&gt;, &lt;parameter&gt;                                  | ( \int_{\Gamma} u \cdot (\sigma \cdot n), \int_{\Gamma} \sigma \cdot n )  |</p>
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>de_sd_v_dot_grad</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_v&gt;, &lt;state/param_s&gt;, &lt;parameter&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>ESDVectorDotGrad</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Π_\text{ij} \frac{\partial p}{\partial x_j} v_i, Π_\text{ij} \frac{\partial q}{\partial x_j} u_i</strong></td>
</tr>
<tr>
<td></td>
<td><strong>\hat{\text{ij}} = \delta_{ij} \nabla \cdot \mathbf{V} - \frac{\partial V_j}{\partial x_i}</strong></td>
</tr>
</tbody>
</table>

Table 6 – continued from previous page
Table of large deformation terms

Table 7: Large deformation terms

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_tl_bulk_active</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td></td>
</tr>
<tr>
<td>BulkActiveTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_bulk_penalty</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td>act.fib, com.ela.mat, hyp</td>
</tr>
<tr>
<td>BulkPenaltyTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_bulk_pressure</td>
<td>&lt;state&gt;, &lt;state_p&gt;</td>
<td>[ \int_\Omega S_{ij}(p) \delta E_{ij}(u; v) ]</td>
<td>bal, per.tl</td>
</tr>
<tr>
<td>BulkPressureTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_diffusion</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;, &lt;parameter&gt;</td>
<td>[ \int_\Omega K(u^{(n-1)}) \left( \frac{\partial q}{\partial X} \frac{\partial p}{\partial X} \right) ]</td>
<td>per.tl</td>
</tr>
<tr>
<td>DiffusionTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_fib_a</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;material&gt;, &lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td>act.fib</td>
</tr>
<tr>
<td>FibresActiveTLi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_he_genyeh</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td></td>
</tr>
<tr>
<td>GenYeohTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_he_mooney</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td>bal, com.ela.mat, hyp</td>
</tr>
<tr>
<td>MooneyRivlinTLi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_he_neohook</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Omega S_{ij}(u) \delta E_{ij}(u; v) ]</td>
<td>hyp, per.tl, bal, com.ela.mat, act.fib</td>
</tr>
<tr>
<td>NeoHookeanTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_tl_he_neohook</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)$</td>
<td></td>
</tr>
<tr>
<td>NeoHookeanTLAD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_he_ogden</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)$</td>
<td></td>
</tr>
<tr>
<td>OgdenTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_he_ogden_a</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)$</td>
<td></td>
</tr>
<tr>
<td>OgdenTLADTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_membrane</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>bal</td>
<td></td>
</tr>
<tr>
<td>TLMembraneTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_tl_surface_flux</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;parameter&gt;, &lt;parameter&gt;</td>
<td>per tl</td>
<td></td>
</tr>
<tr>
<td>SurfaceFluxTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_surface_trac</td>
<td>&lt;opt_material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>bal, per tl</td>
<td></td>
</tr>
<tr>
<td>SurfaceTraction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_tl_volume</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_{\Omega} gJ(u)$ volume mode: vector for $K \leftarrow I_h : \int_{T_h} J(u)$ rel_volume mode: vector for $K \leftarrow I_h : \int_{T_h} J(u)/\int_{T_h} 1$</td>
<td></td>
</tr>
<tr>
<td>VolumeTLTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev_tl_volume_surf</td>
<td>&lt;parameter&gt;</td>
<td>$1/D \int_{\Gamma} \nu \cdot F^{-1} \cdot \delta J$</td>
<td>hyp.ul</td>
</tr>
<tr>
<td>VolumeSurfaceTL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_ul_bulk_penalti</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_{\Omega} \mathcal{L} \tau_{ij}(u) e_{ij}(\delta v)/J$</td>
<td></td>
</tr>
<tr>
<td>BulkPenaltyULT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_ul_bulk_pressure</td>
<td>&lt;virtual&gt;, &lt;state&gt;, &lt;state_p&gt;</td>
<td>$\int_\Omega \mathcal{L} \tau_{ij}(\mathbf{u}) e_{ij}(\delta \mathbf{v}) / J$</td>
<td>hyp.ul.up</td>
</tr>
<tr>
<td>BulkPressureULTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_ul_compressibility</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;, &lt;parameter&gt;</td>
<td>$\int_\Omega 1 / \gamma p q$</td>
<td>hyp.ul.up</td>
</tr>
<tr>
<td>CompressibilityULTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_ul_he_mooney_neohook</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega \mathcal{L} \tau_{ij}(\mathbf{u}) e_{ij}(\delta \mathbf{v}) / J$</td>
<td>hyp.ul.up</td>
</tr>
<tr>
<td>MooneyRivlinULTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_ul_he_neohook</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega \mathcal{L} \tau_{ij}(\mathbf{u}) e_{ij}(\delta \mathbf{v}) / J$</td>
<td>hyp.ul.up</td>
</tr>
<tr>
<td>NeoHookeanULTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dw_ul_volume</td>
<td>&lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega q J(\mathbf{u})$</td>
<td>hyp.ul.up</td>
</tr>
<tr>
<td>VolumeULTerm</td>
<td></td>
<td>volume mode: vector for $K \leftarrow \mathcal{I}<em>h : \int</em>{T_K} J(\mathbf{u})$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rel_volume mode: vector for $K \leftarrow \mathcal{I}<em>h : \int</em>{T_K} J(\mathbf{u}) / \int_{T_K} 1$</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8: Special terms

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_biot_eth</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;, &lt;ts&gt;, &lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega \left[ \int_0^t a_{ij} (t - \tau) p(\tau) , d\tau \right] e_{ij}(\nu)$, $\int_\Omega \left[ \int_0^t a_{ij} (t - \tau) e_{kl}(\nu(\tau)) , d\tau \right] q$</td>
<td></td>
</tr>
<tr>
<td>dw_biot_th</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;, &lt;ts&gt;, &lt;material&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>$\int_\Omega \left[ \int_0^t a_{ij} (t - \tau) p(\tau) , d\tau \right] e_{ij}(\nu)$, $\int_\Omega \left[ \int_0^t a_{ij} (t - \tau) e_{kl}(\nu(\tau)) , d\tau \right] q$</td>
<td></td>
</tr>
<tr>
<td>ev_cauchy_stress_eth</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;material&gt;, &lt;parameter&gt;</td>
<td>$\int_\Omega \int_0^t H_{ijkl} (t - \tau) e_{kl}(\nu(\tau)) , d\tau$</td>
<td></td>
</tr>
<tr>
<td>ev_cauchy_stress_th</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;parameter&gt;</td>
<td>$\int_\Omega \int_0^t H_{ijkl} (t - \tau) e_{kl}(\nu(\tau)) , d\tau$</td>
<td></td>
</tr>
<tr>
<td>dw_lin_elastic_eth</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega \left[ \int_0^t H_{ijkl} (t - \tau) e_{kl}(\nu(\tau)) , d\tau \right] e_{ij}(\nu)$</td>
<td>lin.vis</td>
</tr>
<tr>
<td>dw_lin_elastic_th</td>
<td>&lt;ts&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>$\int_\Omega \left[ \int_0^t H_{ijkl} (t - \tau) e_{kl}(\nu(\tau)) , d\tau \right] e_{ij}(\nu)$</td>
<td></td>
</tr>
<tr>
<td>ev_of_ns_surf_min_d_press</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;parameter&gt;</td>
<td>$\delta \Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} bpress \right)$</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_of_ns_surf_min_d_press_diff</td>
<td>&lt;material&gt; NSOFSurfMinDPressDiffTerm &lt;virtual&gt;</td>
<td>$u \delta_p \Psi(p) \circ q$</td>
<td></td>
</tr>
<tr>
<td>ev_sd_st_grad_div</td>
<td>&lt;material&gt; SDGradDivStabilizationTerm</td>
<td>$\gamma \int_{\Omega} [\nabla \cdot u)(\nabla \cdot w)(\nabla \cdot \mathcal{V}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} - (\nabla \cdot u)(\nabla \cdot w) \frac{\partial \mathcal{V}_k}{\partial x_k}</td>
<td></td>
</tr>
<tr>
<td>ev_sd_st_pspg_c</td>
<td>&lt;material&gt; SDPSPGCStabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \delta_K \left[ \frac{\partial r}{\partial x_i} (b \cdot \nabla u_i)(\nabla \cdot \mathcal{V}) - \frac{\partial r}{\partial x_k} (b \cdot \nabla \mathcal{V}_k) \frac{\partial u_k}{\partial x_i}\right]$</td>
<td></td>
</tr>
<tr>
<td>ev_sd_st_pspg_p</td>
<td>&lt;material&gt; SDPSPGPStabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \tau_K [(\nabla \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}$</td>
<td></td>
</tr>
<tr>
<td>ev_sd_st_supg_c</td>
<td>&lt;material&gt; SDSUPGCStabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \delta_K [((b \cdot \nabla u_k)(b \cdot \nabla w_k)(\nabla \cdot \mathcal{V}) - (b \cdot \nabla \mathcal{V}_k) \frac{\partial u_k}{\partial x_i}(b \cdot \nabla w_k) -$</td>
<td></td>
</tr>
<tr>
<td>dw_st_adj1_supg_p</td>
<td>&lt;material&gt; SUPGPAdj1StabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \delta_K \nabla p (\mathcal{V} \cdot \nabla \mathcal{W})$</td>
<td></td>
</tr>
<tr>
<td>dw_st_adj2_supg_p</td>
<td>&lt;material&gt; SUPGPAdj2StabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \tau_K \nabla r (\mathcal{V} \cdot \nabla \mathcal{U})$</td>
<td></td>
</tr>
<tr>
<td>dw_st_adj_supg_c</td>
<td>&lt;material&gt; SUPGCAdjStabilizationTerm</td>
<td>$\sum_{K \in \mathcal{I}} \int_{T_K} \delta_K [((\mathcal{V} \cdot \nabla) u)((\mathcal{V} \cdot \nabla) w) + ((\mathcal{V} \cdot \nabla) u)((\mathcal{V} \cdot \nabla) w)]$</td>
<td></td>
</tr>
<tr>
<td>dw_st_grad_div</td>
<td>&lt;material&gt; GradDivStabilizationTerm</td>
<td>$\gamma \int_{\Omega} (\nabla \cdot u)(\nabla \cdot v) \quad \text{sta.nav.sto}$</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
## Table 8 – continued from previous page

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_st_pspg_c</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;parameter&gt;</code>, <code>&lt;state&gt;</code></td>
<td>[ \sum_{K \in I_h} \int_{T_K} \tau_K \left( (\mathbf{h} \cdot \nabla) u \right) \cdot \nabla q ]</td>
<td>sta.nav.sto</td>
</tr>
<tr>
<td>dw_st_pspg_p</td>
<td><code>&lt;opt_material&gt;</code>, <code>&lt;virtual/param_1&gt;</code>, <code>&lt;state/param_2&gt;</code></td>
<td>[ \sum_{K \in I_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q ]</td>
<td>sta.nav.sto</td>
</tr>
<tr>
<td>dw_st_supg_c</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;parameter&gt;</code>, <code>&lt;state&gt;</code></td>
<td>[ \sum_{K \in I_h} \int_{T_K} \delta_K \left( (\mathbf{h} \cdot \nabla) u \right) \cdot \left( (\mathbf{h} \cdot \nabla) v \right) ]</td>
<td>sta.nav.sto</td>
</tr>
<tr>
<td>dw_st_supg_p</td>
<td><code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;parameter&gt;</code>, <code>&lt;state&gt;</code></td>
<td>[ \sum_{K \in I_h} \int_{T_K} \delta_K \nabla p \cdot \left( (\mathbf{h} \cdot \nabla) v \right) ]</td>
<td>sta.nav.sto</td>
</tr>
<tr>
<td>dw_volume_dot_w</td>
<td><code>&lt;ts&gt;</code>, <code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td>[ \int_{\Omega} \left[ \int_0^t G(t - \tau) p(\tau) , d\tau \right] , q ]</td>
<td></td>
</tr>
<tr>
<td>dw_volume_dot_w</td>
<td><code>&lt;ts&gt;</code>, <code>&lt;material&gt;</code>, <code>&lt;virtual&gt;</code>, <code>&lt;state&gt;</code></td>
<td>[ \int_{\Omega} \left[ \int_0^t G(t - \tau) p(\tau) , d\tau \right] , q ]</td>
<td></td>
</tr>
</tbody>
</table>
## Table of multi-linear terms

<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>de_cauchy_stress</td>
<td>ECauchyStressTerm</td>
<td>( \int_{\Omega} D_{ijkl} e_{kl}(u) )</td>
<td></td>
</tr>
<tr>
<td>de_convect</td>
<td>EConvectTerm</td>
<td>( \int_{\Omega} ((u \cdot \nabla)u) \cdot \mathbf{v} )</td>
<td></td>
</tr>
<tr>
<td>de_diffusion</td>
<td>EDiffusionTerm</td>
<td>( \int_{\Omega} K_{ij} \nabla q \nabla p )</td>
<td></td>
</tr>
<tr>
<td>de_div</td>
<td>EDivTerm</td>
<td>( \int_{\Omega} \nabla \cdot \mathbf{v}, \int_{\Omega} c \nabla \cdot \mathbf{v} )</td>
<td></td>
</tr>
<tr>
<td>de_grad</td>
<td>EGradTerm</td>
<td>( \int_{\Omega} \nabla \mathbf{v} : \nabla \mathbf{u}, \int_{\Omega} \nu \nabla \mathbf{v} : \nabla \mathbf{u} )</td>
<td></td>
</tr>
<tr>
<td>de_dot</td>
<td>EDotTerm</td>
<td>( \int_{\mathcal{D}} q p, \int_{\mathcal{D}} \mathbf{u} \cdot \mathbf{u}, \int_{\mathcal{D}} e_{qp} : \mathbf{u}, \int_{\mathcal{D}} \nu \nabla \cdot \mathbf{u} )</td>
<td></td>
</tr>
<tr>
<td>de_integrate</td>
<td>EIntegrateOperatorTerm</td>
<td>( \int_{\Omega} \nabla \mathbf{u}, \int_{\Omega} c \nabla \mathbf{v}, \int_{\Omega} \mathbf{c} \cdot \nabla \mathbf{u}, \int_{\Omega} \mathbf{c} \cdot \nabla \mathbf{v} )</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>name/class</th>
<th>arguments</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>de_laplace</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>[ \int_\Omega \nabla q \cdot \nabla p, \int_\Omega e \nabla q \cdot \nabla p ]</td>
<td></td>
</tr>
<tr>
<td>ELaplaceTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de_lin_convect</td>
<td>&lt;virtual/param_1&gt;, &lt;parameter&gt;, &lt;state/param_3&gt;</td>
<td>[ \int_\Omega ((w \cdot \nabla)u) \cdot v ]</td>
<td></td>
</tr>
<tr>
<td>ELinearConvectTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de_lin_elastic</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>[ \int_\Omega D_{ijkl} e_{ij}(u)e_{kl}(u) ]</td>
<td></td>
</tr>
<tr>
<td>ELinearElasticTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de_mass</td>
<td>&lt;material&gt;, &lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ M^C = \int_D \rho u \cdot u ]</td>
<td>sei.loa, ela</td>
</tr>
<tr>
<td>MassTerm</td>
<td></td>
<td>[ M^L = \text{humping}(M^C) ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ M^A = (1 - \beta)M^C + \beta M^L ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ A = \sum_e A_e ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ C = \sum_e A^T_e (M^A_e)^{-1} A_e ]</td>
<td></td>
</tr>
<tr>
<td>de_non_penetration</td>
<td>&lt;material&gt;, &lt;virtual&gt;, &lt;state&gt;</td>
<td>[ \int_\Gamma c(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot \mathbf{u}) ]</td>
<td></td>
</tr>
<tr>
<td>ENonPenetration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de_nonsym_elastic</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;</td>
<td>[ \int_\Omega D \nabla v : \nabla u ]</td>
<td></td>
</tr>
<tr>
<td>ENonSymElasticTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de_s_dot_mgrad_s</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;, &lt;material&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>[ \int_\Omega q^y \cdot \nabla p, \int_\Omega p^y \cdot \nabla q ]</td>
<td></td>
</tr>
<tr>
<td>EScalarDotMGradTerm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name/class</td>
<td>arguments</td>
<td>definition</td>
<td>examples</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>de_stokes</td>
<td>&lt;opt_material&gt;, &lt;virtual/param_v&gt;, &lt;state/param_s&gt;, &lt;opt_material&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>$\int_{\Omega} p \nabla \cdot v, \int_{\Omega} q \nabla \cdot u$ $\int_{\Omega} c p \nabla \cdot v, \int_{\Omega} c q \nabla \cdot u$</td>
<td></td>
</tr>
<tr>
<td>de_surface_flux</td>
<td>&lt;material&gt;, &lt;virtual/param_1&gt;, &lt;state/param_2&gt;, &lt;material&gt;, &lt;state&gt;, &lt;virtual&gt;</td>
<td>$\int_{\Gamma} q n \cdot K \cdot \nabla p, \int_{\Gamma} p n \cdot K \cdot \nabla q$</td>
<td>pie.ela</td>
</tr>
<tr>
<td>de_surface_ltr</td>
<td>&lt;opt_material&gt;, &lt;virtual/param&gt;</td>
<td>$\int_{\Gamma} v \cdot n, \int_{\Gamma} c v \cdot n$ $\int_{\Gamma} v \cdot (n n), \int_{\Gamma} v \cdot f$</td>
<td></td>
</tr>
</tbody>
</table>
The SfePy development takes place in the sfepy/sfepy repository on Github. The users and developers can also communicate using the mailing list.

2.1 General Information

We are interested in any contribution. There are many ways how you can contribute:

- You can report bugs using our mailing list. You can also add a bug report (or a comment) into the issues.
- You can contribute interesting examples/tutorials.
- You can blog about how you use SfePy (let us know!).
- You can help with improving our documentation and these pages.
- ...

To get acquainted with SfePy, you can start by reading the Tutorial and Primer sections of the documentation and trying out the examples that come with the sources. Your first contribution could be pinpointing anything that is not clear in the docs.

We also recommend reading the How to Contribute section of our Developer Guide.

2.2 Possible Topics

Several specific topics that we wish to address in the future are listed below. If you would like to contribute code/advice to our project with respect to these topics, do not hesitate to contact us (either directly: cimrman3(at)ntc.zcu.cz, or on our mailing list)

- finish/improve IGA implementation (see Isogeometric Analysis):
  - support multiple patches
  - efficient quadrature formulas
  - local refinement?
- discretization methods:
  - implement vector elements (Nedelec, Raviart-Thomas, …)
  - implement the discontinuous Galerkin method
- material models: plasticity, viscoplasticity, damage, …
- improve parallelization (see Solving Problems in Parallel):
– cluster installation with fast BLAS
– parallel code speed-up
– remove (some of) the serial parts
– preconditioning for multi-physics problems

• solvers:
  – better defaults/recommendations for iterative solvers (PETSc) with respect to large problems
  – dynamics/time-stepping solvers, interface PETSc time-steppers
  – interface more sparse linear solvers (or enable via PETSc), for example BDDCML
  – interface more eigenvalue problem solvers

• visualization of large data
• automatic differentiation:
  – for tangent matrices
  – for identification of (material) parameters

• core data structures & programming:
  – using octree-based(?) mesh representation for local refinement
  – continue with/improve the current hanging nodes implementation
  – exploit lazy evaluation

See also the enhancement issues.

2.3 Developer Guide

This section purports to document the SfePy internals. It is mainly useful for those who wish to contribute to the development of SfePy and understand the inner workings of the code.

We use git to track source code, documentation, examples, and other files related to the project.

It is not necessary to learn git in order to contribute to SfePy but we strongly suggest you do so as soon as possible - it is an extremely useful tool not just for writing code, but also for tracking revisions of articles, Ph.D. theses, books, … it will also look well in your CV :-) It is also much easier for us to integrate changes that are in form of a github pull request than in another form.

2.3.1 Retrieving the Latest Code

The first step is to obtain the latest development version of the code from the SfePy git repository:

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

For development, it is preferable to build the extension modules in place (see Compilation of C Extension Modules):

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

On Unix-like systems, you can simply type make in the top-level folder to build in-place.

After the initial compilation, or after making changes, do not forget to run the tests, see Testing Installation.
2.3.2 SfePy Directory Structure

Here we list and describe the directories that are in the main sfepy directory.

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build/</td>
<td>directory created by the build process (generated)</td>
</tr>
<tr>
<td>doc/</td>
<td>source files of this documentation</td>
</tr>
<tr>
<td>meshes/</td>
<td>finite element mesh files in various formats shared by the examples</td>
</tr>
<tr>
<td>output/</td>
<td>default output directory for storing results of the examples</td>
</tr>
<tr>
<td>sfepy/</td>
<td>the source code including examples and tests</td>
</tr>
<tr>
<td>tools/</td>
<td>various helper scripts (build, documentation generation etc.)</td>
</tr>
</tbody>
</table>

New users/developers (after going through the Tutorial) should explore the sfepy/examples/ directory. For developers, the principal directory is sfepy/, which has the following contents:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applications/</td>
<td>top level application classes (e.g. PDESolverApp)</td>
</tr>
<tr>
<td>base/</td>
<td>common utilities and classes used by most of the other modules</td>
</tr>
<tr>
<td>discrete/</td>
<td>general classes and modules for describing a discrete problem, taking care of boundary conditions, degrees of freedom, approximations, variables, equations, meshes, regions, quadratures, etc. Discretization-specific classes are in subdirectories: • common/ - common parent classes for discretization-specific classes • fem/ - finite element specific classes • iga/ - isogeometric analysis specific classes</td>
</tr>
<tr>
<td>examples/</td>
<td>the examples using both the declarative and imperative problem description API</td>
</tr>
<tr>
<td>homogenization/</td>
<td>the homogenization engine and supporting modules - highly specialized code, one of the reasons of SfePy existence</td>
</tr>
<tr>
<td>linalg/</td>
<td>linear algebra functions not covered by NumPy and SciPy</td>
</tr>
<tr>
<td>mechanics/</td>
<td>modules for (continuum) mechanics: elastic constant conversions, tensor, units utilities, etc.</td>
</tr>
<tr>
<td>mesh/</td>
<td>some utilities to interface with tetgen and triangle mesh generators</td>
</tr>
<tr>
<td>parallel/</td>
<td>modules supporting parallel assembling and solution of problems</td>
</tr>
<tr>
<td>postprocess/</td>
<td>Matplotlib and VTK based post-processing modules</td>
</tr>
<tr>
<td>scripts/</td>
<td>the main as well as auxiliary scripts</td>
</tr>
<tr>
<td>solvers/</td>
<td>interface classes to various internal/external solvers (linear, nonlinear, eigenvalue, optimization, time stepping)</td>
</tr>
<tr>
<td>terms/</td>
<td>implementation of the terms (weak formulation integrals), see Term Overview</td>
</tr>
<tr>
<td>tests/</td>
<td>the tests</td>
</tr>
</tbody>
</table>

The directories in the “field-specific” column are mostly interesting for specialists working in the respective fields.
The discrete/ is the heart of the code, while the terms/ contains the particular integral forms usable to build equations - new term writers should look there.

### 2.3.3 Exploring the Code

It is convenient to install IPython (see also Using IPython) to have the tab completion available. Moreover, all SfePy classes can be easily examined by printing them:

```
In [1]: from sfepy.discrete.fem import Mesh

In [2]: mesh = Mesh.from_file('meshes/2d/rectangle_tri.mesh')
sfepy: reading mesh [line2, tri3, quad4, tetra4, hexa8] (meshes/2d/rectangle_tri.mesh)...sfepy: ...done in 0.00 s

In [3]: print(mesh)
```

```
Mesh:meshes/2d/rectangle_tri
cmesh:
   CMesh: n_coor: 258, dim 2, tdim: 2, n_el 454
descs:
   list: ['2_3']
dim:
   2
dims:
   list: [2]
io:
   None
n_el:
   454
n_nod:
   258
name:
   meshes/2d/rectangle_tri
nodal_bcs:
   dict with keys: []
```

We recommend going through the interactive example in the tutorial Interactive Example: Linear Elasticity in this way, printing all the variables.

Another useful tool is the debug() function, that can be used as follows:

```
from sfepy.base.base import debug; debug()
```

Try to use it in the examples with user defined functions to explore their parameters etc. It works best with IPython installed, as then the tab completion is available also when debugging.
2.3.4 How to Contribute

Read this section if you wish to contribute some work to the SfePy project - everyone is welcome to contribute. Contributions can be made in a variety of forms, not just code. Reporting bugs and contributing to the documentation, tutorials, and examples is in great need!

Below we describe

1. where to report problems or find existing issues and additional development suggestions
2. what to do to apply changes/fixes
3. what to do after you made your changes/fixes

Reporting problems

Reporting a bug is the first way in which to contribute to an open source project

Short version: go to the main SfePy site and follow the links given there.

When you encounter a problem, try searching that site first - an answer may already be posted in the SfePy mailing list (to which we suggest you subscribe...), or the problem might have been added to the SfePy issues. As is true in any open source project, doing your homework by searching for existing known problems greatly reduces the burden on the developers by eliminating duplicate issues. If you find your problem already exists in the issue tracker, feel free to gather more information and append it to the issue. In case the problem is not there, create a new issue with proper labels for the issue type and priority, and/or ask us using the mailing list.

Note: A google account (e.g., gmail account) is needed to join the mailing list. A github account is needed for working with the source code repository and issues.

Note: When reporting a problem, try to provide as much information as possible concerning the version of SfePy, the OS / Linux distribution, and the versions of Python, NumPy and SciPy, and other prerequisites. The versions found on your system can be printed by running:

```
python setup.py --help
```

If you are a new user, please let us know what difficulties you have with this documentation. We greatly welcome a variety of contributions not limited to code only.

Contributing changes

Note: To avoid duplicating work, it is highly advised that you contact the developers on the mailing list or create an enhancement issue before starting work on a non-trivial feature.

Before making any changes, read the Notes on commits and patches.

Using git and github

The preferred way to contribute to SfePy is to fork the main repository on github, then submit a “pull request” (PR):

1. Create a github account if you do not already have one.
2. Fork the project repository: click on the “Fork” button near the top of the sfe.py git repository page. This creates a copy of the repository under your account on the github server.
3. Clone your fork to your computer:
4. If you have never used git before, introduce yourself to git and make (optionally) some handy aliases either in `.gitconfig` in your home directory (global settings for all your git projects), or directly in `.git/config` in the repository:

```
[user]
    email = mail@mail.org
    name = Name Surname
[color]
    ui = auto
    interactive = true
[alias]
    ci = commit
    di = diff --color-words
    st = status
    co = checkout
```

5. Create a feature branch to hold your changes:

```
git checkout -b my-feature
```

Then you can start to make your changes. Do not work in the master branch!

6. Modify some files and use git to track your local changes. The changed added/modified files can be listed using:

```
git status
```

and the changes can be reviewed using:

```
git diff
```

A more convenient way of achieving the above is to run:

```
gitk --all
```

in order to visualize of project history (all branches). There are other GUIs for this purpose, e.g. qgit. You may need to install those tools, as they usually are not installed with git by default. Record a set of changes by:

```
# 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
# an editor will pop up where you should describe the commit
```

We recommend `git gui` command in case you want to add and commit only some changes in a modified file.

**Note:** Do not be afraid to experiment - git works with your local copy of the repository, so it is not possible to damage the master repository. It is always possible to re-clone a fresh copy, in case you do something that is really bad.

7. The commit(s) now reflect changes, but only in your local git repository. To update your github repository with your new commit(s), run:
Finally, when your feature is ready, and all tests pass, go to the github page of your sfepy repository fork, and click “Pull request” to send your changes to the maintainers for review. Continuous integration (CI) will run and check that the changes pass tests on Windows, Linux and Mac using Github Actions. The results will be displayed in the Pull Request discussion. The CI setup is located in the file `.github/workflows/build_and_test_matrix.yml`. It is recommended to check that your contribution complies with the Notes on commits and patches.

In the above setup, your origin remote repository points to `YourLogin/sfepy.git`. If you wish to fetch/merge from the main repository instead of your forked one, you will need to add another remote to use instead of origin. The main repository is usually called “upstream”. To add it, type:

```
git remote add upstream https://github.com/sfepy/sfepy.git
```

To synchronize your repository with the upstream, proceed as follows:

1. Fetch the upstream changes:

```
git fetch upstream
```

Never start with `git pull upstream`!

2. Check the changes of the upstream master branch. You can use `gitk --all` to visualize all your and remote branches. The upstream master is named `remotes/upstream/master`.

3. Make sure all your local changes are either committed in a feature branch or stashed (see `git stash`). Then reset your master to the upstream master:

```
git checkout master
```

```
git reset --hard upstream/master
```

**Warning** The above will remove all your local commits in the master branch that are not in `upstream/master`, and also reset all the changes in your non-committed modified files!

Optionally, the reset command can be run conveniently in `gitk` by right-clicking on a commit you want to reset the current branch onto.

4. Optionally, rebase your feature branch onto the upstream master:

```
git checkout my-feature
```

```
git rebase upstream/master
```

This is useful, for example, when the upstream master contains a change you need in your feature branch.

For additional information, see, for example, the `gitwash git tutorial`, or its incarnation NumPy `gitwash`.

**Notes on commits and patches**

- Follow our *Coding style*.
- Do not use lines longer than 79 characters (exception: tables of values, e.g., quadratures).
- Write descriptive docstrings in correct style, see *Docstring standard*.
- There should be one patch for one topic - do not mix unrelated things in one patch. For example, when you add a new function, then notice a typo in docstring in a nearby function and correct it, create two patches: one fixing the docstring, the other adding the new function.
The commit message and description should clearly state what the patch does. Try to follow the style of other commit messages. Some interesting notes can be found at tbaggery.com, namely that the commit message is better to be written in the present tense: “fix bug” and not “fixed bug”.

Without using git

Without using git, send the modified files to the SfePy mailing list or attach them using gist to the corresponding issue at the Issues web page. Do not forget to describe the changes properly, and to follow the spirit of Notes on commits and patches and the Coding style.

Coding style

All the code in SfePy should try to adhere to python style guidelines, see PEP-0008.

There are some additional recommendations:

• Prefer whole words to abbreviations in public APIs - there is completion after all. If some abbreviation is needed (really too long name), try to make it as comprehensible as possible. Also check the code for similar names - try to name things consistently with the existing code. Examples:
  - yes: equation, transform_variables(), filename
  - rather not: eq, transvar(), fname

• Functions have usually form <action>_<subject>() e.g.: save_data(), transform_variables(), do not use data_save(), variable_transform() etc.

• Variables like V, c, A, b, x should be tolerated only locally when expressing mathematical ideas.

Really minor recommendations:

• Avoid single letter names, if you can:
  - not even for loop variables - use e.g. ir, ic, .. instead of i, j for rows and columns
  - not even in generators, as they “leak” (this is fixed in Python 3.x)

These are recommendations only, we will not refuse code just on the ground that it uses slightly different formatting, as long as it follows the PEP.

Note: some old parts of the code might not follow the PEP, yet. We fix them progressively as we update the code.

Docstring standard

We use sphinx with the numpydoc extension to generate this documentation. Refer to the sphinx site for the possible markup constructs.

Basically (with a little tweak), we try to follow the NumPy/SciPy docstring standard as described in NumPy documentation guide. See also the complete docstring example. It is exaggerated a bit to show all the possibilities. Use your common sense here - the docstring should be sufficient for a new user to use the documented object. A good way to remember the format is to type:

```
In [1]: import numpy as nm
In [2]: nm.sin?
```

in ipython. The little tweak mentioned above is the starting newline:
```python
def function(arg1, arg2):
    
    This is a function.

    Parameters
    ----------
    arg1 : array
        The coordinates of ...
    arg2 : int
        The dimension ...

    Returns
    -------
    out : array
        The resulting array of shape ....
```

It seems visually better than:

```python
def function(arg1, arg2):
    
    This is a function.

    Parameters
    ----------
    arg1 : array
        The coordinates of ...
    arg2 : int
        The dimension ...

    Returns
    -------
    out : array
        The resulting array of shape ....
```

When using \LaTeX{} in a docstring, use a raw string:

```python
def function():
    r""
    This is a function with :math:`\frac{1}{\pi}`.
    ""
```

to prevent Python from interpreting and consuming the backslashes in common escape sequences like `\n`, `\f` etc.
2.3.5 How to Regenerate Documentation

The following steps summarize how to regenerate this documentation.

1. Install sphinx 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 LATEX 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):

   ```bash
   cd doc
   make html
   ```

4. View it (substitute your favorite browser):

   ```bash
   firefox _build/html/index.html
   ```

2.3.6 How to Implement a New Term

Warning Implementing a new term usually involves C. As Cython is now supported by our build system, it should not be that difficult. Python-only terms are possible as well.

Note There is an experimental way (newly from version 2021.1) of implementing multi-linear terms that is much easier than what is described here, see Multi-linear Terms.

Notes on terminology

Term integrals are over domains of the cell or facet kinds. For meshes with elements of the topological dimension \( t \leq d \), where \( d \) is the space dimension, cells have the topological \( t \), while facets \( t - 1 \). For example, in 3D meshes cell = volume, facet = surface, while in 2D cell = area, facet = curve.

Introduction

A term in SfePy usually corresponds to a single integral term in (weak) integral formulation of an equation. Both cell and facet integrals are supported. There are three types of arguments a term can have:

- **variables**, i.e. the unknown, test or parameter variables declared by the `variables` keyword, see sec-problem-definition-file,
- **materials**, corresponding to material and other parameters (functions) that are known, declared by the `materials` keyword,
- **user data** - anything, but user is responsible for passing them to the evaluation functions.

SfePy terms are subclasses of `sfepy.terms.terms.Term`. The purpose of a term is to implement a (vectorized) function that evaluates the term contribution to residual/matrix and/or evaluates the term integral in elements of the term region. Many such functions are currently implemented in C, but some terms are pure Python, vectorized using NumPy.
Evaluation modes

A term can support several evaluation modes, as described in Term Evaluation.

Basic attributes

A term class should inherit from sfepy.terms.terms.Term base class. The simplest possible term with cell integration and ‘weak’ evaluation mode needs to have the following attributes and methods:

- docstring (not really required per se, but we require it);
- name attribute - the name to be used in equations;
- arg_types attribute - the types of arguments the term accepts;
- integration attribute, optional - the kind of integral the term implements, one of ‘cell’ (the default, if not given), ‘facet’ or ‘facet_extra’;
- function() static method - the assembling function;
- get_fargs() method - the method that takes term arguments and converts them to arguments for function().

Argument types

The argument types can be (“[*]” denotes an optional suffix):

- ‘material[*]’ for a material parameter, i.e. any function that can be can evaluated in quadrature points and that is not a variable;
- ‘opt_material[*]’ for an optional material parameter, that can be left out - there can be only one in a term and it must be the first argument;
- ‘virtual’ for a virtual (test) variable (no value defined), ‘weak’ evaluation mode;
- ‘state[*]’ for state (unknown) variables (have value), ‘weak’ evaluation mode;
- ‘parameter[*]’ for parameter variables (have known value), any evaluation mode.

Only one ‘virtual’ variable is allowed in a term.

Integration kinds

The integration kinds have the following meaning:

- ‘cell’ for cell integral over a region that contains elements; uses cell connectivity for assembling;
- ‘facet’ for facet integral over a region that contains faces; uses facet connectivity for assembling;
- ‘facet_extra’ for facet integral over a region that contains faces; uses cell connectivity for assembling - this is needed if full gradients of a variable are required on the boundary.
function()

The function() static method has always the following arguments:

```python
out, *args
```

where `out` is the already preallocated output array (change it in place!) and `*args` are any other arguments the function requires. These function arguments have to be provided by the `get_fargs()` method. The function returns zero status on success, nonzero on failure.

The `out` array has shape `(n_el, 1, n_row, n_col)`, where `n_el` is the number of elements and `n_row`, `n_col` are matrix dimensions of the value on a single element.

get_fargs()

The `get_fargs()` method has always the same structure of arguments:

- positional arguments corresponding to `arg_types` attribute:
  - example for a typical weak term:
    - * for:
      ```python
      arg_types = ('material', 'virtual', 'state')
      ```
      the positional arguments are:
      ```
      material, virtual, state
      ```
  - keyword arguments common to all terms:
    ```
    mode=\texttt{None}, \texttt{term\_mode=\texttt{None}}, \texttt{diff\_var=\texttt{None}}, **\texttt{kwargs}
    ```
    here:
    - `mode` is the actual evaluation mode, default is `eval`;
    - `term_mode` is an optional term sub-mode influencing what the term should return (example: `dw_tl_he_neohook` term has `strain` and `stress` evaluation sub-modes);
    - `diff_var` is taken into account in the `weak` evaluation mode. It is either `None` (residual mode) or a name of variable with respect to differentiate to (matrix mode);
    - `**kwargs` are any other arguments that the term supports.

The `get_fargs()` method returns arguments for `function()`.

Additional attributes

These attributes are used mostly in connection with the `tests/test_term_call_modes.py` test for automatic testing of term calls.

- `arg_shapes` attribute - the possible shapes of term arguments;
- `geometries` attribute - the list of reference element geometries that the term supports;
- `mode` attribute - the default evaluation mode.
Argument shapes

The argument shapes are specified using a dict of the following form:

```python
global 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:

```python
global 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:

```python
global arg_types = ('opt_material', 'parameter')
arg_shapes = [{'opt_material': 1, 1}, 'parameter': 1},
               {'opt_material': None}]
```

Geometries

The default that most terms use is a list of all the geometries:

```python
global geometries = ['2_3', '2_4', '3_4', '3_8']
```

In that case, the attribute needs not to be define explicitly.

Examples

Let us now discuss the implementation of a simple weak term `dw_integrate` defined as \( \int_\mathcal{D} c q \), where \( c \) is a weight (material parameter) and \( q \) is a virtual variable. This term is implemented as follows:

```python
class IntegrateOperatorTerm(Term):
    r"""Integral of a test function weighted by a scalar function
    :math:`c`."
    
    :Definition:
    .. math::
        \int_{\mathcal{D}} q \text{ or } \int_{\mathcal{D}} c q
    
    :Arguments:
    - material : :math:`c` (optional)
```

(continues on next page)
```python
- virtual : :math:`q`

```"""

name = 'dw_integrate'
arg_types = ('opt_material', 'virtual')
arg_shapes = [{'opt_material': 1, 1}, {'opt_material': None}]
integration = ('cell', 'facet')

@staticmethod
def function(out, material, bf, geo):
    bf_t = nm.tile(bf.transpose((0, 1, 3, 2)), (out.shape[0], 1, 1, 1))
    bf_t = nm.ascontiguousarray(bf_t)
    if material is not None:
        status = geo.integrate(out, material * bf_t)
    else:
        status = geo.integrate(out, bf_t)
    return status

def get_fargs(self, material, virtual,
            mode=None, term_mode=None, diff_var=None, **kwargs):
    assert_(virtual.n_components == 1)
    geo, _ = self.get_mapping(virtual)
    return material, geo.bf, geo

• lines 2-14: the docstring - always write one!
• line 15: the name of the term, that can be referred to in equations;
• line 16: the argument types - here the term takes a single material parameter, and a virtual variable;
• lines 17-18: the possible argument shapes
• line 19: the integration mode is chosen according to a given domain
• lines 21-29: the term function
  – its arguments are:
    * the output array `out`, already having the required shape,
    * the material coefficient (array) `mat` evaluated in physical quadrature points of elements of the term region,
    * a base function (array) `bf` evaluated in the quadrature points of a reference element and
    * a reference element (geometry) mapping `geo`.
  – line 23: transpose the base function and tile it so that is has the correct shape - it is repeated for each element;
  – line 24: ensure C contiguous order;
  – lines 25-28: perform numerical integration in C - `geo.integrate()` requires the C contiguous order;
  – line 29: return the status.
• lines 31-36: prepare arguments for the function above:
  – line 33: verify that the variable is scalar, as our implementation does not support vectors;
A more complex term that involves an unknown variable and has two call modes, is \(dw_s\cdot m\cdot grad\cdot s\), defined as
\[
\int_{\Omega} q y \cdot \nabla p \quad \text{in the 'grad_state' mode or}
\int_{\Omega} p y \cdot \nabla q \quad \text{in the 'grad_virtual' mode,}
\]
where \(y\) is a vector material parameter, \(q\) is a virtual variable, and \(p\) is a state variable:

```python
class ScalarDotMGradScalarTerm(Term):
    r""
    Volume dot product of a scalar gradient dotted with a material vector
    with a scalar.
    :Definition:
    .. math::
        \int_{\Omega} q \ul{y} \cdot \nabla p \quad \text{, }
        \int_{\Omega} p \ul{y} \cdot \nabla q

    :Arguments 1:
    - material : :math:`\ul{y}`
    - virtual : :math:`q`
    - state : :math:`p`

    :Arguments 2:
    - material : :math:`\ul{y}`
    - state : :math:`p`
    - virtual : :math:`q`
    ""
    name = 'dw_s\cdot m\cdot grad\cdot s'
    arg_types = (('material', 'virtual', 'state'),
                 ('material', 'state', 'virtual'))
    arg_shapes = [{'material' : 'D, 1',
                   'virtual/grad_state' : (1, None),
                   'state/grad_state' : 1,
                   'virtual/grad_virtual' : (1, None),
                   'state/grad_virtual' : 1}]
    modes = ('grad_state', 'grad_virtual')

@staticmethod
def function(out, out_qp, geo, fmode):
    status = geo.integrate(out, out_qp)
    return status

def get_fargs(self, mat, var1, var2,
               mode=None, term_mode=None, diff_var=None, **kwargs):
    vg1, _ = self.get_mapping(var1)
    vg2, _ = self.get_mapping(var2)

    if diff_var is None:
        if self.mode == 'grad_state':
            geo = vg1
            bf_t = vg1.bf.transpose((0, 1, 3, 2))
            val_qp = self.get(var2, 'grad')
```
out_qp = bf_t * dot_sequences(mat, val_qp, 'ATB')

else:
    geo = vg2
    val_qp = self.get(var1, 'val')
    out_qp = dot_sequences(vg2.bfg, mat, 'ATB') * val_qp
    fmode = 0

else:
    if self.mode == 'grad_state':
        geo = vg1
        bf_t = vg1.bf.transpose((0, 1, 3, 2))
        out_qp = bf_t * dot_sequences(mat, vg2.bfg, 'ATB')
    else:
        geo = vg2
        out_qp = dot_sequences(vg2.bfg, mat, 'ATB') * vg1.bf

    fmode = 1

    return out_qp, geo, fmode

Only interesting differences with respect to the previous example will by discussed:

- the argument types and shapes (lines 23-29) have to be specified for all the call modes (line 30)
- the term function (lines 32-35) just integrates the element contributions, as all the other calculations are done by the get_fargs() function.
- the get_fargs() function (lines 37-68) contains:
  - residual computation (lines 43-54) for both modes
  - matrix computation (lines 57-66) for both modes

Concluding remarks

This is just a very basic introduction to the topic of new term implementation. Do not hesitate to ask the SfePy mailing list, and look at the source code of the already implemented terms.

2.3.7 Multi-linear Terms

tentative documentation, the enriched einsum notation is still in flux

Multi-linear terms can be implemented simply by using the following enriched einsum notation:
Table 3: The enriched einsum notation for defining multi-linear terms.

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>scalar</td>
<td>$p$</td>
</tr>
<tr>
<td>$i$</td>
<td>$i$-th vector component</td>
<td>$u_i$</td>
</tr>
<tr>
<td>$i.j$</td>
<td>gradient: derivative of $i$-th vector component w.r.t. $j$-th coordinate component</td>
<td>$\frac{\partial u_i}{\partial x_j}$</td>
</tr>
<tr>
<td>$i:j$</td>
<td>symmetric gradient</td>
<td>$\frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$</td>
</tr>
<tr>
<td>$s(i:j)\rightarrow$</td>
<td>vector storage of symmetric second order tensor, $I$ is the vector component</td>
<td>Cauchy strain tensor $e_{ij}(u)$</td>
</tr>
</tbody>
</table>

The examples below present the new way of implementing the terms shown in the original Examples, using `sfepy.terms.terms_multilinear.ETermBase`.

Examples

- `de_integrate` defined as $\int_{\Omega} c q$, where $c$ is a weight (material parameter) and $q$ is a virtual variable:

```python
class EIntegrateOperatorTerm(ETermBase):
    ""
    Volume and surface integral of a test function weighted by a scalar function :math:`c`.
    ""
    :Definition:
    .. math::
        \int_{\{\cal{D}\}} q \text{ or } \int_{\{\cal{D}\}} c \ q
    :Arguments:
    - material : :math:`c` (optional)
    - virtual : :math:`q`
    ""
    name = 'de_integrate'
    arg_types = ('opt_material', 'virtual')
    arg_shapes = [{"opt_material" : '1, 1', 'virtual' : (1, None),
                   'opt_material' : None}]
    def get_function(self, mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs):
        if mat is None:
            fun = self.make_function(
                '0', virtual, diff_var=diff_var,
            )
        else:
            fun = self.make_function(
                '00,0', mat, virtual, diff_var=diff_var,
            )
        return fun
```

- `de_s_dot_mgrad_s` defined as $\int_{\Omega} q y \cdot \nabla p$ in the `grad_state` mode or $\int_{\Omega} p y \cdot \nabla q$ in the `grad_virtual` mode,
where \( y \) is a vector material parameter, \( q \) is a virtual variable, and \( p \) is a state variable:

```python
class EScalarDotMGradScalarTerm(ETermBase):
    """
    Volume dot product of a scalar gradient dotted with a material vector with
    a scalar.
    :Definition:
    .. math::
        \int_{\Omega} q \ul{y} \cdot \nabla p \mbox{ , }
        \int_{\Omega} p \ul{y} \cdot \nabla q
    :Arguments 1:
    - material : :math:`\ul{y}`
    - virtual : :math:`q`
    - state : :math:`p`
    :Arguments 2:
    - material : :math:`\ul{y}`
    - state : :math:`p`
    - virtual : :math:`q`
    """

    name = 'de_s_dot_mgrad_s'
    arg_types = (('material', 'virtual', 'state'),
                 ('material', 'state', 'virtual'))
    arg_shapes = [
        {'material': 'D, 1',
         'virtual/grad_state' : (1, None),
         'state/grad_state' : 1,}
        {'virtual/grad_virtual' : (1, None),
         'state/grad_virtual' : 1}]
    modes = ('grad_state', 'grad_virtual')

    def get_function(self, mat, var1, var2, mode=None, term_mode=None,
                     diff_var=None, **kwargs):
        return self.make_function(
            'i0,0,i', mat, var1, var2, diff_var=diff_var,
        )
```

### 2.3.8 How To Make a Release

#### Release Tasks

A few notes on what to do during a release.
Things to check before a release

1. synchronize module documentation (dry run):

   ```
   $ python3 tools/sync_module_docs.py doc/src/ . -n
   ```

2. regenerate gallery page and examples:

   ```
   $ rm -rf doc/examples/
   $ python3 tools/gen_gallery.py
   ```

3. create temporary/testing tarball:

   ```
   $ python3 setup.py sdist
   ```

4. check in-place build:

   ```
   $ # unpack the tarball
   $ # cd into
   $ python3 setup.py build_ext --inplace
   $ python3 sfepy/scripts/test_install.py
   ```

5. check that documentation can be built:

   ```
   $ # copy site_cfg.py
   $ python3 setup.py htmldocs
   $ firefox doc/_build/html/index.html
   ```

   or use:

   ```
   $ cd doc/
   $ make html
   $ firefox _build/html/index.html
   ```

   try also:

   ```
   $ python3 setup.py pdfdocs
   ```

6. check installed build:

   ```
   $ python3 -m pip install . --user
   $ cd
   $ sfepy-test
   ```

   then remove the installed files so that they do not interfere with the local build

7. create final tarball

   - update doc/release_notes.rst, with the help of:
     ```
     $ python3 tools/gen_release_notes.py 2019.2
     ```
   - update doc/news.rst, doc/archived_news.rst
   - change version number (sfepy/version.py) so that previous release tarball is not overwritten!
   - set is_release = True in site_cfg.py

2.3. Developer Guide
• update pdfdocs:

```bash
$ python3 setup.py pdfdocs
```

• create tarball:

```bash
$ python3 setup.py sdist
```

8. tag the release using:

```bash
$ git tag release_XXXX.X
```

### Useful Git commands

- log

```bash
git log --pretty=format:"%s%n%b%n" --topo-order --reverse release_2016.4..HEAD
```

- who has contributed since <date>:

```bash
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
```

```bash
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:

```bash
git archive --format=tar HEAD | gzip > name.tar.gz
```

### Web update and file uploading

- upload the sources to TestPyPI and test:

  - bump version number as needed when testing
  - upload commands:

    ```bash
    python3 setup.py sdist # set `"is_release = True"` in site_cfg.py
    python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/
    --sfepy-2022.3[.-]`
    ```

  - testing:

    ```bash
    python3 -m venv venv
    source venv/bin/activate
    python3 -m pip install -U --index-url https://test.pypi.org/simple/ --extra-
    --index-url https://pypi.org/simple/ sfepy
    python3 -m pip install pytest
    python3 -c "import sfepy; sfepy.test('-v', '--durations=0')"
    ```

(continues on next page)
• upload the sources to PyPI and test:
  – upload commands:
    ```
    python3 setup.py sdist # set "is_release = True" in site_cfg.py
    python3 -m twine upload dist/sfepy-2022.3[.-]*
    ```
  – testing:
    ```
    python3 -m venv venv
    source venv/bin/activate
    python3 -m pip install -U sfepy
    python3 -m pip install pytest
    python3 -c "import sfepy; sfepy.test('-v', '--durations=0')"
    deactivate
    # rm -rf venv
    ```
• make a pull request with the updated version in sfepy-feedstock/recipe/meta.yaml from a fork (e.g. https://github.com/rc/sfepy-feedstock) of https://github.com/conda-forge/sfepy-feedstock.
• publish development docs also as new release docs
• send announcement to
  – sfepy@python.org
  – optionally to scipy-dev@python.org, scipy-user@python.org, numpy-discussion@python.org, python-announce-list@python.org

2.3.9 Module Index

sfepy package

sfepy.config module

class sfepy.config.Config
  compile_flags()
  debug_flags() → list
  is_release()
  numpydoc_path()
  python_include()
  python_version()
  refmap_memory_factor()
  system()
tetgen_path()

sfepy.config.compose_system_compile_flags(is_posix: bool) → list
Provides a list of compile flags that tries to be as similar as possible to what Python itself was built with. This has been done historically by numpy.distutils (now deprecated) and a squeezed version of it is brought over to here.
sfepy.config.has_attr(obj, attr)

sfepy.version module

sfepy.version.get_basic_info(version='2023.4')
Return SfePy installation directory information. Append current git commit hash to version.

sfepy.applications package

sfepy.applications.application module
class sfepy.applications.application.Application(conf, options, output_prefix, **kwargs)
Base class for applications.
Subclasses should implement: __init__(), call().
Automates parametric studies, see parametrize().
call_basic(**kwargs)
call_parametrized(**kwargs)
parametrize(parametric_hook)
    Add parametric_hook, set __call__() to call_parametrized().
restore()
    Remove parametric_hook, restore __call__() to call_basic().
setup_options()

sfepy.applications.evp_solver_app module

Eigenvalue problem solver application.
class sfepy.applications.evp_solver_app.EVPSolverApp(conf, options, output_prefix, **kwargs)
Solve an eigenvalue problem.
call(status=None)
make_full(svecs)
static process_options(options)
    Application options setup. Sets default values for missing non-compulsory options.
save_results(eigs, vecs, out=None, mesh_results_name=None, eig_results_name=None)
setup_options()
setup_output()

Setup various file names for the output directory given by self.problem.output_dir.

solve_eigen_problem()

sfepy.applications.pde_solver_app module

class sfepy.applications.pde_solver_app.PDESolverApp(conf, options, output_prefix, init_equations=True, **kwargs)

call(status=None)

load_dict(filename)

Utility function to load a dictionary data from a HDF5 file filename.

static process_options(options)

Application options setup. Sets default values for missing non-compulsory options.

save_dict(filename, data)

Utility function to save a dictionary data to a HDF5 file filename.

setup_options()

setup_output_info(problem, options)

Modifies both problem and options!

sfepy.applications.pde_solver_app.assign_standard_hooks(obj, get, conf)

Set standard hook function attributes from conf to obj using the get function.

sfepy.applications.pde_solver_app.save_only(conf, save_names, problem=None)

Save information available prior to setting equations and solving them.

sfepy.applications.pde_solver_app.solve_pde(conf, options=None, status=None, **app_options)

Solve a system of partial differential equations (PDEs).

This function is a convenience wrapper that creates and runs an instance of PDESolverApp.

Parameters

conf

[st or ProblemConf instance] Either the name of the problem description file defining the PDEs, or directly the ProblemConf instance.

options

[options] The command-line options.

status

[dict-like] The object for storing the solver return status.

app_options

[kwargs] The keyword arguments that can override application-specific options.
sfepy.base package

sfepy.base.base module

sfepy.base.base.debug(frame=None, frames_back=1)
Start debugger on line where it is called, roughly equivalent to:

```python
import pdb; pdb.set_trace()
```

First, this function tries to start an IPython-enabled debugger using the IPython API.
When this fails, the plain old `pdb` is used instead.
With IPython, one can say in what frame the debugger can stop.

class sfepy.base.base.Container(objs=None, **kwargs)

append(obj)

as_dict()
Return stored objects in a dictionary with object names as keys.

extend(objs)
Extend the container items by the sequence `objs`.

get(ii, default=None, msg_if_none=None)
Get an item from Container - a wrapper around Container.__getitem__() with defaults and custom error message.

Parameters
ii
[int or str] The index or name of the item.

default
[any, optional] The default value returned in case the item `ii` does not exist.

msg_if_none
[str, optional] If not None, and if `default` is None and the item `ii` does not exist, raise ValueError with this message.

get_names()

has_key(ii)

insert(ii, obj)

iteritems()

iterkeys()

itervalues()

print_names()

remove_name(name)

update(objs=None)
A dict-like update for Struct attributes.
class sfepy.base.base.IndexedStruct(**kwargs)

class sfepy.base.base.OneTypeList(item_class, seq=None)

    find(name, ret_indx=False)

    get_names()

    print_names()

class sfepy.base.base.Output(prefix, filename=None, quiet=False, combined=False, append=False, **kwargs)

Factory class providing output (print) functions. All SfePy printing should be accomplished by this class.

Examples

```python
>>> from sfepy.base.base import Output
>>> output = Output('sfepy:')
>>> output(1, 2, 3, 'hello')
sfepy: 1 2 3 hello
>>> output.prefix = 'my_cool_app:'
>>> output(1, 2, 3, 'hello')
my_cool_app: 1 2 3 hello
```

    get_output_function()

    get_output_prefix()

    property prefix

    set_output(filename=None, quiet=False, combined=False, append=False)

Set the output mode.

    If quiet is True, no messages are printed to screen. If simultaneously filename is not None, the messages are logged into the specified file.

    If quiet is False, more combinations are possible. If filename is None, output is to screen only, otherwise it is to the specified file. Moreover, if combined is True, both the ways are used.

Parameters

    filename

        [str or file object] Print messages into the specified file.

    quiet

        [bool] Do not print anything to screen.

    combined

        [bool] Print both on screen and into the specified file.

    append

        [bool] Append to an existing file instead of overwriting it. Use with filename.

set_output_prefix(prefix)

class sfepy.base.base.Struct(**kwargs)
**copy**(deep=False, name=None)

Make a (deep) copy of self.

**Parameters**

- **deep** [bool] Make a deep copy.
- **name** [str] Name of the copy, with default self.name + ‘_copy’.

**get**(key, default=None, msg_if_none=None)

A dict-like get() for Struct attributes.

**set_default**(key, default=None)

Behaves like dict.setdefault().

**str_all**()

**str_class**()

As __str__() , but for class attributes.

**to_dict**()

**update**(other, **kwargs)

A dict-like update for Struct attributes.

**sfepy.base.base.as_float_or_complex**(val)

Try to cast val to Python float, and if this fails, to Python complex type.

**sfepy.base.base.assert_**(condition, msg='assertion failed!')

**sfepy.base.base.check_names**(names1, names2, msg)

Check if all names in names1 are in names2, otherwise raise IndexError with the provided message msg.

**sfepy.base.base.configure_output**(options)

Configure the standard output() function using output_log_name and output_screen attributes of options.

**Parameters**

- **options** [Struct or dict] The options with output_screen and output_log_name items. Defaults are provided if missing.

**sfepy.base.base.debug**(frame=None, frames_back=1)

Start debugger on line where it is called, roughly equivalent to:

```python
import pdb; pdb.set_trace()
```

First, this function tries to start an IPython-enabled debugger using the IPython API.

When this fails, the plain old pdb is used instead.

With IPython, one can say in what frame the debugger can stop.

**sfepy.base.base.debug_on_error**()

Start debugger at the line where an exception was raised.

**sfepy.base.base.dict_extend**(d1, d2)
sfepy.base.base.dict_from_keys_init(keys, seq_class=None)

sfepy.base.base.dict_to_array(adict)
    Convert a dictionary of nD arrays of the same shapes with non-negative integer keys to a single (n+1)D array.

sfepy.base.base.dict_to_struct(*args, **kwargs)
    Convert a dict instance to a Struct instance.

sfepy.base.base.edit_dict_strings(str_dict, old, new, recur=False)
    Replace substrings old with new in string values of dictionary str_dict. Both old and new can be lists of the same length - items in old are replaced by items in new with the same index.

Parameters
    str_dict [dict] The dictionary with string values or tuples containing strings.
    old [str or list of str] The old substring or list of substrings.
    new [str or list of str] The new substring or list of substrings.
    recur [bool] If True, edit tuple values recursively.

Returns

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.
    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

```python
>>> solver_table = find_subclasses(vars().items(),
    [LinearSolver, NonlinearSolver,
    TimeSteppingSolver, EigenvalueSolver,
    OptimizationSolver])
```

`sfepy.base.base.get_arguments(omit=None)`
Get a calling function’s arguments.

Returns:
- **args** [dict] The calling function’s arguments.

`sfepy.base.base.get_debug()`
Utility function providing `debug()` function.

`sfepy.base.base.get_default(arg, default, msg_if_none=None)`

`sfepy.base.base.get_default_attr(obj, attr, default, msg_if_none=None)`

`sfepy.base.base.get_subdict(adict, keys)`
Get a sub-dictionary of `adict` with given `keys`.

`sfepy.base.base.import_file(filename, package_name=None, can_reload=True)`
Import a file as a module. The module is explicitly reloaded to prevent undesirable interactions.

`sfepy.base.base.insert_as_static_method(cls, name, function)`

`sfepy.base.base.insert_method(instance, function)`

`sfepy.base.base.insert_static_method(cls, function)`

`sfepy.base.base.invert_dict(d, is_val_tuple=False, unique=True)`
Invert a dictionary by making its values keys and vice versa.

**Parameters**
- **d** [dict] The input dictionary.
- **is_val_tuple** [bool] If True, the `d` values are tuples and new keys are the tuple items.
- **unique** [bool] If True, the `d` values are unique and so the mapping is one to one. If False, the `d` values (possibly) repeat, so the inverted dictionary will have as items lists of corresponding keys.

**Returns**
- **di** [dict] The inverted dictionary.

`sfepy.base.base.ipython_shell(frame=0)`

`sfepy.base.base.is_derived_class(cls, parent)`

`sfepy.base.base.is_integer(var)`
sfePy.base.base.is_sequence(var)

sfePy.base.base.is_string(var)

sfePy.base.base.iter_dict_of_lists(dol, return_keys=False)

sfePy.base.base.load_classes(filenames, classes, package_name=None, ignore_errors=False, name_attr='name')

For each filename in filenames, load all subclasses of classes listed.

sfePy.base.base.ordered_iteritems(adict)

sfePy.base.base.pause(msg=None)

Prints the line number and waits for a keypress.

If you press: “q” ............. it will call sys.exit() any other key ... it will continue execution of the program

This is useful for debugging.

sfePy.base.base.print_structs(objs)

Print Struct instances in a container, works recursively. Debugging utility function.

sfePy.base.base.python_shell(frame=0)

sfePy.base.base.remap_dict(d, map)

Utility function to remap state dict keys according to var_map.

sfePy.base.base.select_by_names(objs_all, names, replace=None, simple=True)

sfePy.base.base.set_defaults(dict_, defaults)

sfePy.base.base.shell(frame=0)

Embed an IPython (if available) or regular Python shell in the given frame.

sfePy.base.base.spause(msg=None)

Waits for a keypress.

If you press: “q” ............. it will call sys.exit() any other key ... it will continue execution of the program

This is useful for debugging. This function is called from pause().

sfePy.base.base.structify(obj)

Convert a (nested) dict obj into a (nested) Struct.

sfePy.base.base.try_imports(imports, fail_msg=None)

Try import statements until one succeeds.

Parameters

imports

[list] The list of import statements.

fail_msg

[str] If not None and no statement succeeds, a ValueError is raised with the given message, appended to all failed messages.

Returns

locals

[dict] The dictionary of imported modules.
sfepy.base.base.update_dict_recursively : 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 : Use method with specified 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 : Test whether each element of a 1-D array is also present in a second array.

**Parameters**

- **ar1**
  - [(M,) array_like] Input array.
- **ar2**
  - [array_like] The values against which to test each value of ar1.
- **assume_unique**
  - [bool, optional] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.
- **invert**
  - [bool, optional] If True, the values in the returned array are inverted (that is, False where an element of ar1 is in ar2 and True otherwise). Default is False. np.in1d(a, b, invert=True) is equivalent to (but is faster than) np.invert(in1d(a, b)).
- **kind**
  - [None, ‘sort’, ‘table’], optional] The algorithm to use. This will not affect the final result, but will affect the speed and memory use. The default, None, will select automatically based on memory considerations.
  - If ‘sort’, will use a mergesort-based approach. This will have a memory usage of roughly 6 times the sum of the sizes of ar1 and ar2, not accounting for size of dtypes.
If ‘table’, will use a lookup table approach similar to a counting sort. This is only available for boolean and integer arrays. This will have a memory usage of the size of \( ar1 \) plus the max-min value of \( ar2 \). \texttt{assume_unique} has no effect when the ‘table’ option is used.

- If None, will automatically choose ‘table’ if the required memory allocation is less than or equal to 6 times the sum of the sizes of \( ar1 \) and \( ar2 \), otherwise will use ‘sort’. This is done to not use a large amount of memory by default, even though ‘table’ may be faster in most cases. If ‘table’ is chosen, \texttt{assume_unique} will have no effect.

New in version 1.8.0.

Returns

\texttt{in1d}

\ldbrack (M,) ndarray, bool\rdbrack The values \( ar1[in1d] \) are in \( ar2 \).

See also:

\texttt{isin}

Version of this function that preserves the shape of \( ar1 \).

\texttt{numpy.lib.arraysetops}

Module with a number of other functions for performing set operations on arrays.

Notes

\texttt{in1d} can be considered as an element-wise function version of the python keyword \texttt{in}, for 1-D sequences. \texttt{in1d(a, b)} is roughly equivalent to \texttt{np.array([[item in b for item in a]])}. However, this idea fails if \( ar2 \) is a set, or similar (non-sequence) container: As \( ar2 \) is converted to an array, in those cases \texttt{asarray(ar2)} is an object array rather than the expected array of contained values.

Using \texttt{kind=’table’} tends to be faster than \texttt{kind=’sort’} if the following relationship is true: \( \log_{10}(\text{len}(ar2)) > (\log_{10}(\text{max}(ar2) - \text{min}(ar2)) - 2.27) / 0.927 \), but may use greater memory. The default value for \texttt{kind} will be automatically selected based only on memory usage, so one may manually set \texttt{kind=’table’} if memory constraints can be relaxed.

New in version 1.4.0.

Examples

```python
>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
>>> mask
array([ True, False,  True, False,  True])
>>> test[mask]
array([0, 2, 0])
>>> mask = np.in1d(test, states, invert=True)
>>> mask
array([False, True, False, True, False])
>>> test[mask]
array([1, 5])
```

\texttt{sfepy.base.compat.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None, *, equal_nan=True)}
Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

Parameters

- **ar** [array_like] Input array. Unless *axis* is specified, this will be flattened if it is not already 1-D.

- **return_index** [bool, optional] If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

- **return_inverse** [bool, optional] If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.

- **return_counts** [bool, optional] If True, also return the number of times each unique item appears in *ar*.

- **axis** [int or None, optional] The axis to operate on. If None, *ar* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.

  New in version 1.13.0.

- **equal_nan** [bool, optional] If True, collapses multiple NaN values in the return array into one.

  New in version 1.24.

Returns

- **unique** [ndarray] The sorted unique values.

- **unique_indices** [ndarray, optional] The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

- **unique_inverse** [ndarray, optional] The indices to reconstruct the original array from the unique array. Only provided if *return_inverse* is True.

- **unique_counts** [ndarray, optional] The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

  New in version 1.9.0.

See also:
numpy.lib.arraysetops

Module with a number of other functions for performing set operations on arrays.

repeat

Repeat elements of an array.

Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order. The flattened subarrays are then viewed as a structured type with each element given a label, with the effect that we end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

Examples

```python
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])

>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])

Return the unique rows of a 2D array

```python
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Return the indices of the original array that give the unique values:

```python
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'], dtype='<U1')
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> a[indices]
array(['a', 'b', 'c'], dtype='<U1')
```

Reconstruct the input array from the unique values and inverse:

```python
>>> 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])
```

Reconstruct the input values from the unique values and counts:
```python
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> values, counts = np.unique(a, return_counts=True)
>>> values
array([1, 2, 3, 4, 6])
>>> counts
array([1, 3, 1, 1, 1])
>>> np.repeat(values, counts)
array([1, 2, 2, 2, 3, 4, 6])  # original order not preserved
```

**sfepy.base.conf module**

Problem description file handling.

**Notes**

Short syntax: key is suffixed with ‘__<number>’ to prevent collisions with long syntax keys -> both cases can be used in a single input.

```python
class sfepy.base.conf.ProblemConf(define_dict=None, funmod=None, filename=None, required=None, other=None, verbose=True, override=None, setup=True)
```

Problem configuration, corresponding to an input (problem description file). It validates the input using lists of required and other keywords that have to/can appear in the input. Default keyword lists can be obtained by sfepy.base.conf.get_standard_keywords().

ProblemConf instance is used to construct a Problem instance via Problem.from_conf(conf).

```python
add_missing(conf)
```

Add missing values from another problem configuration.

Missing keys/values are added also to values that are dictionaries.

Parameters:

- **conf** [ProblemConf instance] The other configuration.

```python
edit(key, newval)
```

```python
static from_dict(dict_, funmod=None, required=None, other=None, verbose=True, override=None, setup=True)
```

```python
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:

```python
def define():
    options = {
        'save_eig_vectors': None,
    }
```

(continues on next page)
'eigen_solver' : 'eigen1',
}
region_2 = {
    'name' : 'Surface',
    'select' : 'nodes of surface',
}
return locals()

Optionally, the define() function can accept additional arguments that should be defined using the define_args tuple or dictionary.

static from_file_and_options(filename, options, required=None, other=None, verbose=True, define_args=None, setup=True)

Utility function, a wrapper around ProblemConf.from_file() with possible override taken from options.

static from_module(module, required=None, other=None, verbose=True, override=None, setup=True)

get_function(name)

Get a function object given its name.

It can be either in ProblemConf.funmod, or a ProblemConf attribute directly.

Parameters

name [str or function or None] The function name or directly the function.

Returns

fun [function or None] The required function, or None if name was None.

get_item_by_name(key, item_name)

Return item with name item_name in configuration group given by key.

get_raw(key=None)

setup(define_dict=None, funmod=None, filename=None, required=None, other=None)

transform_input()

transform_input_trivial()

Trivial input transformations.

update_conf(conf)

Update configuration by values in another problem configuration.

Values that are dictionaries are updated in-place by dict.update().

Parameters

conf [ProblemConf instance] The other configuration.

validate(required=None, other=None)

sfepy.base.conf.dict_from_options(options)

Return a dictionary that can be used to construct/override a ProblemConf instance based on options.

See --conf and --options options of the simple.py script.
sfepy.base.conf.dict_from_string(string, allow_tuple=False, free_word=False)

Parse string and return a dictionary that can be used to construct/override a ProblemConf instance.
sfepy.base.conf.get_standard_keywords()
sfepy.base.conf.transform_conditions(adict, prefix)
sfepy.base.conf.transform_dgebc(adict)
sfepy.base.conf.transform_dgepbcs(adict)
sfepy.base.conf.transform_ebc(adict)
sfepy.base.conf.transform_epbcs(adict, prefix='epbc')
sfepy.base.conf.transform_fields(adict)
sfepy.base.conf.transform_functions(adict)
sfepy.base.conf.transform_ics(adict)
sfepy.base.conf.transform_integrals(adict)
sfepy.base.conf.transform_lcbcs(adict)
sfepy.base.conf.transform_materials(adict)
sfepy.base.conf.transform_regions(adict)
sfepy.base.conf.transform_solvers(adict)
sfepy.base.conf.transform_to_i_struct_1(adict)
sfepy.base.conf.transform_to_struct_01(adict)
sfepy.base.conf.transform_to_struct_1(adict)
sfepy.base.conf.transform_to_struct_10(adict)
sfepy.base.conf.transform_variables(adict)
sfepy.base.conf.tuple_to_conf(name, vals, order)

Convert a configuration tuple vals into a Struct named name, with attribute names given in and ordered by order.

Items in order at indices outside the length of vals are ignored.

sfepy.base.getch module

getch()-like unbuffered character reading from stdin on both Windows and Unix

Getch classes inspired by Danny Yoo, iskeydown() based on code by Zachary Pincus.
sfepy.base.goptions module

Various global options/parameters.

Notes

Inspired by rcParams of matplotlib.

```python
class sfepy.base.goptions.ValidatedDict
    A dictionary object including validation.
    keys()
        Return sorted list of keys.
    validate = {'check_term_finiteness': <function validate_bool>, 'verbose': <function validate_bool>}
    values()
        Return values in order of sorted keys.
```

```python
sfepy.base.goptions.validate_bool(val)
    Convert b to a boolean or raise a ValueError.
```

sfepy.base.ioutils module

```python
class sfepy.base.ioutils.Cached(data)
    The wrapper class that marks data, that should be checked during saving, whether it has been stored to the hdf5
    file already and if so, a softlink to the already created instance is created instead of saving.

class sfepy.base.ioutils.DataMarker(data)
    The Base class for classes for marking data to be handled in a special way during saving to a HDF5 file by
    write_to_hdf5(). The usage is simple: just “decorate” the desired data element, e.g.:
    ```python
data = [data1, Cached(data2)]
write_to_hdf5(... , ... , data)
```
```
```python
unpack_data()
    One can request unpacking of the wrappers during saving.
    Returns
    object
        The original object, if possible, or self.
```

```python
class sfepy.base.ioutils.DataSoftLink(type, destination, cache=None)
    This object is written to the HDF5 file as a softlink to the given path. The destination of the softlink should
    contain only data, so the structure {type: type, data: softlink_to(destination)} is created in the place where the
    softlink is written.
    get_type()
    unpack_data()
    One can request unpacking of the wrappers during saving.
    Returns
```
object
The original object, if possible, or self.

write_data(fd, group, cache=None)
Create the softlink to the destination and handle the caching.

class sfepy.base.ioutils.HDF5BaseData
When storing values to HDF5, special classes can be used that wrap the stored data and modify the way the storing is done. This class is the base of those.

unpack_data()
One can request unpacking of the wrappers during saving.

Returns

object
The original object, if possible, or self.

class sfepy.base.ioutils.HDF5ContextManager(filename, *args, **kwargs)

class sfepy.base.ioutils.HDF5Data
SomedatawrittentotheHDF5filecanhaveacustomformat. Descendants of this class should have the method .write_data() or redefine the .write() method.

write(fd, group, name, cache=None)
Write a data structure to the HDF5 file.

CreatethefollowingstructureintheHDF5file: {type: self.get_type(), anything writed by self.write_data()}

Parameters

fd: tables.File
The hdf5 file handle the data should be writed in.

group: tables.group.Group
The group the data will be stored to

name: str
Name of node that will be appended to group and will contain the data

write_data(fd, group)
Write data to the HDF5 file. Redefine this function in sub-classes.

Parameters

fd: tables.File
The hdf5 file handle the data should be writed to.

group: tables.group.Group
The group the data should be stored to.

class sfepy.base.ioutils.InDir(filename)
Store the directory name a file is in, and prepend this name to other files.
Examples

```python
>>> indir = InDir('output/file1')
>>> print indir('file2')
```

class `sfepy.base.ioutils.SoftLink` *(destination)*

This object is written to the HDF5 file as a softlink to the given path.

```python
write(fd, group, name, cache=None)
```

Create the softlink to the destination.

class `sfepy.base.ioutils.Uncached`(data)

The wrapper class that marks data, that should be always stored to the hdf5 file, even if the object has been already stored at a different path in the file and so it would have been stored by a softlink otherwise (IGDomain, Mesh and sparse matrices behave so).

`sfepy.base.ioutils.dec`(val, encoding='utf-8')

Decode given bytes using the specified encoding.

`sfepy.base.ioutils.edit_filename`(filename, prefix='', suffix='', new_ext=None)

Edit a file name by add a prefix, inserting a suffix in front of a file name extension or replacing the extension.

Parameters

- **filename**
  [str] The file name.
- **prefix**
  [str] The prefix to be added.
- **suffix**
  [str] The suffix to be inserted.
- **new_ext**
  [str, optional] If not None, it replaces the original file name extension.

Returns

- **new_filename**
  [str] The new file name.

`sfepy.base.ioutils.enc`(string, encoding='utf-8')

Encode given string or bytes using the specified encoding.

`sfepy.base.ioutils.ensure_path`(filename)

Check if path to `filename` exists and if not, create the necessary intermediate directories.

`sfepy.base.ioutils.get_or_create_hdf5_group`(fd, path, from_group=None)

Returns the max. number of digits in range(n_step) and the corresponding format string.

Examples:

```python
>>> get_print_info(11)
(2, '%2d')
>>> get_print_info(8)
(1, '%1d')
>>> get_print_info(100)
(continues on next page)
```
sfepy.base.ioutils.get_trunk(filename)

sfepy.base.ioutils.locate_files(pattern, root_dir='.', **kwargs)

Locate all files matching given filename pattern in and below supplied root directory.

The **kwargs arguments are passed to os.walk().

sfepy.base.ioutils.look_ahead_line(fd)

Read and return a line from the given file object. Saves the current position in the file before the reading occurs and, after the reading, restores the saved (original) position.

sfepy.base.ioutils.path_of_hdf5_group(group)

sfepy.base.ioutils.read_array(fd, n_row, n_col, dtype)

Read a NumPy array of shape (n_row, n_col) from the given file object and cast it to type dtype. If n_col is None, determine the number of columns automatically.

sfepy.base.ioutils.read_dict_hdf5(filename, level=0, group=None, fd=None)

sfepy.base.ioutils.read_from_hdf5(fd, group, cache=None)

Read custom data from a HDF5 file group saved by write_to_hdf5().

The data are stored in a general (possibly nested) structure: {

    'type': string type identifier
    'data': stored data
    'cache': string, optional - another possible location of object

}

Parameters

fd: tables.File

The hdf5 file handle the data should be restored from.

group: tables.group.Group

The group in the hdf5 file the data will be restored from.

cache: dict or None

Some objects (e.g. Mesh instances) can be stored on more places in the HDF5 file tree using softlinks, so when the data are restored, the restored objects are stored and searched in cache so that they are created only once. The keys to cache are the (real) paths of the created objects. Moreover, if some stored object has a ‘cache’ key (see e.g. DataSoftLink class), and the object with a given ‘path’ has been already created, it is returned instead of creating a new object. Otherwise, the newly created object is associated both with its real path and with the cache key path.

The caching is not active for scalar data types.

Returns

data

[object] The restored custom data.
sfepy.base.ioutils.read_list(fd, n_item, dtype)

sfepy.base.ioutils.read_sparse_matrix_from_hdf5(fd, group, output_format=None)
Read sparse matrix from given data group of hdf5 file

Parameters

fd: tables.File
   The hdf5 file handle the matrix will be read from.

group: tables.group.group
   The hdf5 file group of the file the matrix will be read from.

output_format: {'csr', 'csc', None}, optional
   The resulting matrix will be in CSR or CSC format if this parameter is not None (which is default), otherwise it will be in the format the matrix was stored.

Returns

scipy.sparse.base.spmatrix
   Readed matrix

sfepy.base.ioutils.read_sparse_matrix_hdf5(filename, output_format=None)

sfepy.base.ioutils.read_token(fd)
Read a single token (sequence of non-whitespace characters) from the given file object.

Notes

Consumes the first whitespace character after the token.

sfepy.base.ioutils.remove_files(root_dir, **kwargs)
Remove all files and directories in supplied root directory.
The **kwargs arguments are passed to os.walk().

sfepy.base.ioutils.remove_files_patterns(root_dir, patterns, ignores=None, verbose=False)
Remove files with names satisfying the given glob patterns in a supplied root directory. Files with patterns in ignores are omitted.

sfepy.base.ioutils.save_options(filename, options_groups, save_command_line=True, quote_command_line=False)
Save groups of options/parameters into a file.
Each option group has to be a sequence with two items: the group name and the options in {key : value} form.

sfepy.base.ioutils.skip_read_line(fd, no_eof=False)
Read the first non-empty line (if any) from the given file object. Return an empty string at EOF, if no_eof is False. If it is True, raise the EOFError instead.

sfepy.base.ioutils.write_dict_hdf5(filename, adict, level=0, group=None, fd=None)

sfepy.base.ioutils.write_sparse_matrix_hdf5(filename, mtx, name='a sparse matrix')
Assume CSR/CSC.

sfepy.base.ioutils.write_sparse_matrix_to_hdf5(fd, group, mtx)
Write sparse matrix to given data group of hdf5 file

Parameters
**group**: `tables.group.group`
   The hdf5 file group the matrix will be read from.

**mtx**: `scipy.sparse.base.spmatrix`
   The written matrix

`sfePy.base.iouUtils.write_to_hdf5(fd, group, name, data, cache=None, unpack_markers=False)`

Save custom data to a HDF5 file group to be restored by `read_from_hdf5()`.

Allows saving lists, dicts, numpy arrays, scalars, sparse matrices, meshes and iga domains and all pickleable objects.

**Parameters**

**fd**: `tables.File`
   The hdf5 file handle the data should be written in.

**group**: `tables.group.Group`
   The group the data will be stored to.

**name**: `str`
   The name of the node that will be appended to the group and will contain the data.

**data**: `object`
   Data to be stored in the HDF5 file.

**cache**: `dict` or `None`
   The cache where the paths to stored objects (currently meshes and iga domains) are stored, so subsequent attempts to store such objects create only softlinks to the initially stored object. The id() of objects serve as the keys into the cache. Mark the object with `Cached()` or `Uncached()` for (no) softlinking.

**unpack_markers**:
   If True, the input data is modified so that Cached and Uncached markers are removed from all sub-elements of the data.

**Returns**

`tables.group.Group`
   The HDF5 group the data was stored to.

**sfePy.base.log module**

**class** `sfePy.base.log.Log(data_names=None, plot_kwargs=None, xlabels=None, ylabels=None, yscales=None, show_legends=True, is_plot=True, aggregate=100, sleep=1.0, log_filename=None, formats=None)`

Log data and (optionally) plot them in the second process via LogPlotter.

**add_group**(names, plot_kwargs=None,yscale=None,xlabel=None,ylabel=None, formats=None)
   Add a new data group. Notify the plotting process if it is already running.

**count** = -1

**static from_conf**(conf, data_names)

**Parameters**

**data_names**
   [list of lists of str] The data names grouped by subplots: [[[name1, name2, ...]], [[name3, name4, ...]]], where name<n> are strings to display in (sub)plot legends.
get_log_name()

plot_data(igs)

plot_vlines(igs=None, **kwargs)
    Plot vertical lines in axes given by igs at current x locations to mark some events.

terminate()

dsfepy.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.

dsfepy.base.log.iter_names(data_names, igs=None)

dsfepy.base.log.plot_log(axs, log, info, xticks=None, yticks=None, xnbins=None, ynbins=None, groups=None, show_legends=True, swap_axes=False)
    Plot log data returned by read_log() into a specified figure.

    Parameters
    ----------
    axs
        [sequence of matplotlib.axes.Axes] The list of axes for the log data plots.
    log
        [dict] The log with data names as keys and (xs, ys, vlines) as values.
    info
        [dict] The log plot configuration with subplot numbers as keys.
    xticks
        [list of arrays, optional] The list of x-axis ticks (array or None) for each subplot.
    yticks
        [list of arrays, optional] The list of y-axis ticks (array or None) for each subplot.
    xnbins
        [list, optional] The list of x-axis number of bins (int or None) for each subplot.
    ynbins
        [list, optional] The list of y-axis number of bins (int or None) for each subplot.
    groups
        [list, optional] The list of data groups subplots. If not given, all groups are plotted.
    show_legends
        [bool] If True, show legends in plots.
    swap_axes
        [bool] If True, swap the axes of the plots.
sfepy.base.log.read_log(filename)

Read data saved by Log into a text file.

Parameters

filename

[str] The name of a text log file.

Returns

log

[dict] The log with data names as keys and (xs, ys, vlines) as values.

info

[dict] The log plot configuration with subplot numbers as keys.
	sfepy.base.log.write_log(output, log, info)

sfepy.base.log_plotter module

Plotting class to be used by Log.

class sfepy.base.log_plotter.LogPlotter(aggregate=100, sleep=1.0)

LogPlotter to be used by sfepy.base.log.Log.

apply_commands()

make_axes()

output = Output

poll_draw()

process_command(command)

terminate()
	sfepy.base.log_plotter.draw_data(ax, xdata, ydata, label, plot_kwargs, swap_axes=False)

Draw log data to a given axes, obeying swap_axes.

sfepy.base.mem_usage module

Memory usage functions.
	sfepy.base.mem_usage.get_mem_usage(obj, usage=None, name=None, traversal_order=None, level=0)

Get lower bound of memory usage of an object.

Takes into account strings, numpy arrays and scipy CSR sparse matrices, descends into sequences, mappings and objects.

Parameters

obj

[any object] The object to be measured.

usage

[dict] The dict with memory usage records, serving also as a cache of already traversed objects.
name
[st] The name to be given to the object in its record.

traversal_order
[list, internal] The traversal order of the object.

level
[int, internal] The recurrence level.

Returns

usage
[int] The object’s lower bound of memory usage.

sfepy.base.mem_usage.print_mem_usage(usage, order_by='usage', direction='up', print_key=False)
Print memory usage dictionary.

Parameters

usage
[dict] The dict with memory usage records.

order_by
[‘usage’, ‘name’, ‘kind’, ‘nrefs’, ‘traversal_order’, or ‘level’] The sorting field name.

direction
[‘up’ or ‘down’] The sorting direction.

print_key
[bool] If True, print also the record key (object’s id).

sfepy.base.mem_usage.raise_if_too_large(size, factor=1.0)
Raise MemoryError if the total system memory is lower than size times safety factor. Use factor=None for skipping the memory check.

sfepy.base.multiproc module

Multiprocessing functions.

sfepy.base.multiproc.get_multiproc(mpi=False)

sfepy.base.multiproc.get_num_workers()
Get the number of slave nodes.

sfepy.base.multiproc.is_remote_dict(d)

sfepy.base.multiproc_mpi module

Multiprocessing functions.

class sfepy.base.multiproc_mpi.MPIFileHandler(filename, mode=4, comm=<mpi4py.MPI.Intracomm object>)

MPI file class for logging process communication.

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

2.3. Developer Guide 183
emit(record)
    Emit a record.
    If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an ‘encoding’ attribute, it is used to determine how to do the output to the stream.

class sfepy.base.multiproc_mpi.MPILogFile(comm, filename, mode)
    close()
    sync()
    write(msg)

class sfepy.base.multiproc_mpi.RemoteDict(name, mutable=False)
    Remote dictionary class - slave side.
    get(key, default=None)
    keys()
    update(other)

class sfepy.base.multiproc_mpi.RemoteDictMaster(name, mutable=False, soft_set=False, *args)
    Remote dictionary class - master side.
    remote_get(key, slave)
    remote_get_in(key, slave)
    remote_get_keys(slave)
    remote_get_len(slave)
    remote_set(data, slave, mutable=False)

class sfepy.base.multiproc_mpi.RemoteInt(remote_dict, value=None)
    Remote integer class, data saved in RemoteDict.

class IntDesc
    value

class sfepy.base.multiproc_mpi.RemoteLock
    Remote lock class - lock and unlock restricted access to the master.
    acquire()
    release()

class sfepy.base.multiproc_mpi.RemoteQueue(name)
    Remote queue class - slave side.
    get()
    put(value)
class sfepy.base.multiproc_mpi.RemoteQueueMaster(name, mode='fifo', *args)
    Remote queue class - master side.
    clean()
    get()
    static get_gdict_key(name)
    put(value)
    remote_get(slave)
    remote_put(value, slave)

sfepy.base.multiproc_mpi.cpu_count()
    Get the number of MPI nodes.

sfepy.base.multiproc_mpi.enum(*sequential)

sfepy.base.multiproc_mpi.get_dict(name, mutable=False, clear=False, soft_set=False)
    Get the remote dictionary.

sfepy.base.multiproc_mpi.get_int_value(name, init_value=0)
    Get the remote integer value.

sfepy.base.multiproc_mpi.get_logger(log_filename='multiproc_mpi.log')
    Get the MPI logger which log information into a shared file.

sfepy.base.multiproc_mpi.get_queue(name)
    Get the queue.

sfepy.base.multiproc_mpi.get_slaves()
    Get the list of slave nodes

sfepy.base.multiproc_mpi.is_remote_dict(d)
    Return True if ‘d’ is RemoteDict or RemoteDictMaster instance.

sfepy.base.multiproc_mpi.master_loop()
    Run the master loop - wait for requests from slaves.

sfepy.base.multiproc_mpi.master_send_continue()
    Send ‘continue’ to all slaves.

sfepy.base.multiproc_mpi.master_send_task(task, data)
    Send task to all slaves.

sfepy.base.multiproc_mpi.set_logging_level(log_level='info')

sfepy.base.multiproc_mpi.slave_get_task(name='')
    Start the slave nodes.

sfepy.base.multiproc_mpi.slave_task_done(task='')
    Stop the slave nodes.

sfepy.base.multiproc_mpi.wait_for_tag(wtag, num=1)
sfepy.base.multiproc_proc module

Multiprocessing functions - using multiprocessing (process based) module.

class sfepy.base.multiproc_proc.MyQueue

    get()

    put(value)

sfepy.base.multiproc_proc.get_dict(name, clear=False, **kwargs)

    Get the remote dictionary.

sfepy.base.multiproc_proc.get_int_value(name, val0=0)

    Get the remote integer value.

sfepy.base.multiproc_proc.get_list(name, clear=False)

    Get the remote list.

sfepy.base.multiproc_proc.get_lock(name)

    Get the global lock.

sfepy.base.multiproc_proc.get_manager()

    Get the multiprocessing manager. If not in the global cache, create a new instance.

    Returns

        manager

            [manager] The multiprocessing manager.

sfepy.base.multiproc_proc.get_mpdict_value(mode, key, clear=False)

    Get the item from the global multiprocessing cache.

    Parameters

        mode

            [str] The type of the required object.

        key

            [immutable type] The key of the required object.

        clear

            [bool] If True, clear the dictionary or list (for modes ‘dict’ and ‘list’).

    Returns

        value

            [remote object] The remote object.

sfepy.base.multiproc_proc.get_queue(name)

    Get the global queue.

sfepy.base.multiproc_proc.is_remote_dict(d)

    Return True if ‘d’ is instance.
sfepy.base.parse_conf module

Create pyparsing grammar for problem configuration and options.

*sfepy.base.parse_conf.create_bnf*(allow_tuple=False, free_word=False)

*sfepy.base.parse_conf.cvt_array_index*(toks)

*sfepy.base.parse_conf.cvt_cmplx*(toks)

*sfepy.base.parse_conf.cvt_int*(toks)

*sfepy.base.parse_conf.cvt_none*(toks)

*sfepy.base.parse_conf.cvt_real*(toks)

*sfepy.base.parse_conf.get_standard_type_defs*(word={W:(ABCD...) [{{Suppress:("{") Forward: None} Suppress:("}")} Forward: None]})

Return dict of the pyparsing base lexical elements.

The compound types (tuple, list, dict) can contain compound types or simple types such as integers, floats and words.

**Parameters**
- **word**
  - [lexical element] A custom lexical element for word.

**Returns**
- **defs**
  - [dict] The dictionary with the following items:
    - tuple: (..., 
    - list: [...,
    - dict: {...:..., ...:..., ...}, or {...=..., ...=..., ...}
    - list_item: any of preceding compound types or simple types

*sfepy.base.parse_conf.list_dict*(word={W:(ABCD...) [[[Suppress:("{") Forward: None} Suppress:("}")} Forward: None]})

Return the pyparsing lexical element, that parses a string either as a list or as a dictionary.

**Parameters**
- **word**
  - [lexical element] A custom lexical element for word.

**Returns**
- **ld**
  - [lexical element] The returned lexical element parses a string in the form ..., .. , ..., or key1:..., key2=..., key3: ... where ... is a list_item from *get_standard_type_defs()* and interprets it as a list or a dictionary.

*sfepy.base.parse_conf.list_of*(element, *elements*)

Return lexical element that parses a list of items. The items can be a one or several lexical elements. For example, result of *list_of(real, integer)* parses list of real or integer numbers.
sfepy.base.plotutils module

sfepy.base.plotutils.font_size(size)

sfepy.base.plotutils.iplot(*args, **kwargs)

sfepy.base.plotutils.plot_matrix_diff(mtx1, mtx2, delta, legend, mode)

sfepy.base.plotutils.print_matrix_diff(title, legend, mtx1, mtx2, mtx_da, mtx_dr, ii)

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)

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:

```python
>>> class A:
...     pass

>>> read = Reader( '.' )

>>> instance_of_a = read( A, 'file.py' )
```

It is equivalent to:

```python
>>> mod = __import__( 'file' )
>>> instance_of_a = A()
>>> instance_of_a.__dict__.update( mod.__dict__ )
```

The first way does not create the `file.pyc`.

class sfepy.base.resolve_deps module

Functions for resolving dependencies.

sfepy.base.resolve_deps.get_nums(deps)

Get number of prerequisite names for each name in dependencies.

sfepy.base.resolve_deps.remove_known(deps, known)

Remove known names from dependencies.

sfepy.base.resolve_deps.resolve(deps)

Resolve dependencies among equations so that smaller blocks are solved first.

The dependencies are given in terms of variable names.

Parameters
deps
[dict] The dependencies as a dictionary with names as keys and sets of prerequisite names as values.

Returns
order
[list] The list of blocks in the order of solving. Each block is a list of names.

sfepy.base.resolve_deps.solvable(deps, names)
Return True if names form a solvable block, i.e. the set of names equals to the set of their prerequisites.

sfepy.base.resolve_deps.try_block(deps, num)
Return generator of lists of solvable blocks of the length num.

sfepy.base.testing module

class sfepy.base.testing.NLSStatus(**kwargs)
Custom nonlinear solver status storing stopping condition of all time steps.

sfepy.base.testing.assert_equal(a, b, msg='assertion of equality failed!')

sfepy.base.testing.check_conditions(conditions)

sfepy.base.testing.compare_vectors(vec1, vec2, allowed_error=1e-08, label1='vec1', label2='vec2', norm=None)

sfepy.base.testing.eval_coor_expression(expression, coor)

sfepy.base.testing.report(*argc)
All tests should print via this function.

sfepy.base.testing.run_declarative_example(ex_filename, output_dir, ext='.vtk', remove_prefix='')
Run a declarative example in ex_filename given relatively to sfepy.base_dir.

sfepy.base.timing module

Elapsed time measurement utilities.

class sfepy.base.timing.Timer(name='timer', start=False)

reset()

start(reset=False)

stop()
sfepy.discrete package

This package implements various PDE discretization schemes (FEM or IGA).

sfepy.discrete.conditions module

The Dirichlet, periodic and linear combination boundary condition classes, as well as the initial condition class.

class sfepy.discrete.conditions.Condition(name, **kwargs)

Common boundary condition methods.

    canonize_dof_names(dofs)
    
    Canonize the DOF names using the full list of DOFs of a variable.
    
    Assumes single condition instance.

    iter_single()
    
    Create a single condition instance for each item in self.dofs and yield it.

class sfepy.discrete.conditions.Conditions(objs=None, **kwargs)

Container for various conditions.

    canonize_dof_names(dofs)
    
    Canonize the DOF names using the full list of DOFs of a variable.

    static from_conf(conf, regions)

    group_by_variables(groups=None)
    
    Group boundary conditions of each variable. Each condition is a group is a single condition.

    Parameters

    groups
    
    [dict, optional] If present, update the groups dictionary.

    Returns

    out
    
    [dict] The dictionary with variable names as keys and lists of single condition instances as values.

    sort()
    
    Sort boundary conditions by their key.

    zero_dofs()
    
    Set all boundary condition values to zero, if applicable.

class sfepy.discrete.conditions.DGEssentialBC(*args, diff=0, **kwargs)

This class is empty, it serves the same purpose as EssentialBC, and is created only for branching in dof_info.py

class sfepy.discrete.conditions.DGPeriodicBC(name, regions, dofs, match, key='', times=None)

This class is empty, it serves the same purpose as PeriodicBC, and is created only for branching in dof_info.py

class sfepy.discrete.conditions.EssentialBC(name, region, dofs, key='', times=None)

Essential boundary condition.

    Parameters

    name
    
    [str] The boundary condition name.
region
   [Region instance] The region where the boundary condition is applied.

dofs
   [dict] The boundary condition specification defining the constrained DOFs and their values.

key
   [str, optional] The sorting key.

times
   [list or str, optional] The list of time intervals or a function returning True at time steps, when
   the condition applies.

zero_dofs()
   Set all essential boundary condition values to zero.

class sfepy.discrete.conditions.InitialCondition(name, region, dofs, key='')
   Initial 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.discrete.conditions.LinearCombinationBC(name, regions, dofs, dof_map_fun, kind, key='',
   times=None, arguments=None)
   Linear combination boundary condition.

Parameters

name
   [str] The boundary condition name.

regions
   [list of two Region instances] The constrained (master) DOFs region and the new (slave)
   DOFs region. The latter can be None if new DOFs are not field variable DOFs.

dofs
   [dict] The boundary condition specification defining the constrained DOFs and the new
   DOFs (can be None).

dof_map_fun
   [str] The name of function for mapping the constrained DOFs to new DOFs (can be None).

kind
   [str] The linear combination condition kind.

key
   [str, optional] The sorting key.

times
   [list or str, optional] The list of time intervals or a function returning True at time steps, when
   the condition applies.
arguments: tuple, optional
Additional arguments, depending on the condition kind.

canonize_dof_names(dofs0, dofs1=None)
Canonize the DOF names using the full list of DOFs of a variable.
Assumes single condition instance.

get_var_names()
Get names of variables corresponding to the constrained and new DOFs.

class sfepy.discrete.conditions.PeriodicBC(name, regions, dofs, match, key=",", times=None)
Periodic boundary condition.

Parameters

name
  [str] The boundary condition name.

regions
  [list of two Region instances] The master region and the slave region where the DOFs should match.

dofs
  [dict] The boundary condition specification defining the DOFs in the master region and the corresponding DOFs in the slave region.

match
  [str] The name of function for matching corresponding nodes in the two regions.

key
  [str, optional] The sorting key.

times
  [list or str, optional] The list of time intervals or a function returning True at time steps, when the condition applies.

canonize_dof_names(dofs)
Canonize the DOF names using the full list of DOFs of a variable.
Assumes single condition instance.
	sfepy.discrete.conditions.get_condition_value(val, functions, kind, name)
Check a boundary/initial condition value type and return the value or corresponding function.

sfepy.discrete.equations module

Classes of equations composed of terms.

class sfepy.discrete.equations.Equation(name, terms, setup=True)

collect_conn_info(conn_info)

collect_materials()
Collect materials present in the terms of the equation.

collect_variables()
Collect variables present in the terms of the equation.

Ensures that corresponding primary variables of test/parameter variables are always in the list, even if they are not directly used in the terms.
evaluate (mode='eval', dw_mode='vector', term_mode=None, diff_vars=None, asm_obj=None)
Evaluate the equation.

Parameters

mode
[one of 'eval', 'el_eval', 'el_avg', 'qp', 'weak'] The evaluation mode.

dw_mode
[one of 'vector', 'matrix', 'sensitivity'] The particular evaluation mode if mode is 'weak'.

term_mode
[str] The term evaluation mode, used mostly if mode is 'eval' in some terms.

diff_vars
[list of str] The names of parameters with respect to the equation is differentiated if
dw_mode is 'sensitivity'.

asm_obj
[ndarray or spmatrix] The object for storing the evaluation result in the 'weak' mode.

Returns

out
[result] The evaluation result. In 'weak' mode it is the asm_obj.

static from_desc(name, desc, variables, regions, materials, integrals, user=None, eterm_options=None, allow_derivatives=False)

class sfepy.discrete.equations.Equations(equations)

add_equation(equation)
Add a new equation.

Parameters

equation

advance(ts)

apply_ebc(vec=None, force_values=None)
Apply essential (Dirichlet) boundary conditions to equations' variables, or a given vector.

apply_ic(vec=None, force_values=None)
Apply initial conditions to equations' variables, or a given vector.

collect_conn_info()
Collect connectivity information as defined by the equations.

collect_materials()
Collect materials present in the terms of all equations.

collect_variables()
Collect variables present in the terms of all equations.

create_matrix_graph(any_dof_conn=False, rdcs=None, cdcs=None, shape=None, active_only=True, verbose=True)
Create tangent matrix graph, i.e. preallocate and initialize the sparse storage needed for the tangent matrix.
Order of DOF connectivities is not important.

Parameters
any_dof_conn
[bool] By default, only cell region DOF connectivities are used, with the exception of trace facet DOF connectivities. If True, any DOF connectivities are used.

rdc, cdc
[arrays, optional] Additional row and column DOF connectivities, corresponding to the variables used in the equations.

shape
[tuple, optional] The required shape, if it is different from the shape determined by the equations variables. This may be needed if additional row and column DOF connectivities are passed in.

active_only
[bool] If True, the matrix graph has reduced size and is created with the reduced (active DOFs only) numbering.

verbose
[bool] If False, reduce verbosity.

Returns

matrix
[csr_matrix] The matrix graph in the form of a CSR matrix with preallocated structure and zero data.

create_reduced_vec()

create_subequations(var_names, known_var_names=None)
Create sub-equations containing only terms with the given virtual variables.

Parameters

var_names
[list] The list of names of virtual variables.

known_var_names
[list] The list of names of (already) known state variables.

Returns

subequations
[Equations instance] The sub-equations.

create_vec()

eval_residuals(state, by_blocks=False, names=None)
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 equa-
tion should then correspond to one required block and should be named as "block_name,
test_variable_name, unknown_variable_name".
names
[list of str, optional] Optionally, select only blocks with the given names, if by_blocks is
True.

Returns

out
[csr_matrix or dict of csr_matrix] The assembled matrix. If by_blocks is True, a dictionary
is returned instead, with keys given by block_name part of the individual equation names.

evaluate(names=None, mode=’eval’, dw_mode=’vector’, term_mode=None, diff_vars=None,
asm_obj=None)
Evaluate the equations.

Parameters

names
[str or sequence of str, optional] Evaluate only equations of the given name(s).
mode
[one of ‘eval’, ‘el_avg’, ‘qp’, ‘weak’] The evaluation mode.
dw_mode
[one of ‘vector’, ‘matrix’, ‘sensitivity’] The particular evaluation mode if mode is ‘weak’.
term_mode
[str] The term evaluation mode, used mostly if mode is ‘eval’ in some terms.
diff_vars
[list of str] The names of parameters with respect to the equations are differentiated if
dw_mode is ‘sensitivity’.
asm_obj
[ndarray or spmatrix] The object for storing the evaluation result in the ’weak’ mode.

Returns

out
[dict or result] The evaluation result. In ‘weak’ mode it is the asm_obj. Otherwise, it is a
dict of results with equation names as keys or a single result for a single equation.
static from_conf

\texttt{get\_domain()}

\texttt{get\_graph\_conns\(\text{any\_dof\_conn=False, rdc=}None, cdc=}None, active\_only=True\)}

Get DOF connectivities needed for creating tangent matrix graph.

**Parameters**

\begin{itemize}
  \item \texttt{any\_dof\_conn} \\
    [bool] By default, only cell DOF connectivities are used, with the exception of trace facet DOF connectivities. If True, any kind of DOF connectivities is allowed.
  \item \texttt{rdcs, cdc}s \\
    [arrays, optional] Additional row and column DOF connectivities, corresponding to the variables used in the equations.
  \item \texttt{active\_only} \\
    [bool] If True, the active DOF connectivities have reduced size and are created with the reduced (active DOFs only) numbering.
\end{itemize}

**Returns**

\begin{itemize}
  \item \texttt{rdcs, cdc}s \\
    [arrays] The row and column DOF connectivities defining the matrix graph blocks.
\end{itemize}

\texttt{get\_lcbc\_operator()}

\texttt{get\_variable(name)}

\texttt{get\_variable\_dependencies()}

For each virtual variable get names of state/parameter variables that are present in terms with that virtual variable.

The virtual variables define the actual equations and their dependencies define the variables needed to evaluate the equations.

**Returns**

\begin{itemize}
  \item \texttt{deps} \\
    [dict] The dependencies as a dictionary with virtual variable names as keys and sets of state/parameter variables as values.
\end{itemize}

\texttt{get\_variable\_names()}

Return the list of names of all variables used in equations.

\texttt{init\_state(vec=None)}

\texttt{init\_time(ts)}

\texttt{invalidate\_term\_caches()}

Invalidate evaluate caches of variables present in equations.

\texttt{make\_full\_vec(svec, force\_value=None)}

Make a full DOF vector satisfying E(P)BCs from a reduced DOF vector.

\texttt{print\_terms()}

Print names of equations and their terms.

\texttt{reduce\_vec(vec, follow\_epbc=False)}

Get the reduced DOF vector, with EBC and PBC DOFs removed.
Notes

If ‘follow_epbc’ is True, values of EPBC master dofs are not simply thrown away, but added to the corresponding slave dofs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

reset_materials()
Clear material data so that next materials.time_update() is performed even for stationary materials.

set_data(data, step=0, ignore_unknown=False)
Set data (vectors of DOF values) of variables.

Parameters

data [dict] The dictionary of {variable_name : data vector}.
step [int, optional] The time history step, 0 (default) = current.
ignore_unknown [bool, optional] Ignore unknown variable names if data is a dict.

set_state(vec, reduced=False, force=False, preserve_caches=False)

setup_initial_conditions(ics, functions=None)

time_update(ts, ebcs=None, epbcs=None, lcbcs=None, functions=None, problem=None, active_only=True, verbose=True)
Update the equations for current time step.

The update involves creating the mapping of active DOFs from/to all DOFs for all state variables, the setup of linear combination boundary conditions operators and the setup of active DOF connectivities.

Parameters

ebcs [Conditions instance, optional] The essential (Dirichlet) boundary conditions.
epbcs [Conditions instance, optional] The periodic boundary conditions.
lcbcs [Conditions instance, optional] The linear combination boundary conditions.
functions [Functions instance, optional] The user functions for boundary conditions, materials, etc.
problem [Problem instance, optional] The problem that can be passed to user functions as a context.
active_only [bool] If True, the active DOF connectivities and matrix graph have reduced size and are created with the reduced (active DOFs only) numbering.
verbose [bool] If False, reduce verbosity.

Returns
graph_changed
    [bool] The flag set to True if the current time step set of active boundary conditions differs from the set of the previous time step.

time_update_materials(ts, mode='normal', problem=None, verbose=True)
    Update data materials for current time and possibly also state.

Parameters

ts
    [TimeStepper instance] The time stepper.

mode
    ['normal', 'update' or 'force'] The update mode, see sfepy.discrete.materials. Material.time_update().

problem
    [Problem instance, optional] The problem that can be passed to user functions as a context.

verbose
    [bool] If False, reduce verbosity.

sfepy.discrete.equations.get_expression_arg_names(expression, strip_dot=True)
    Parse expression and return set of all argument names. For arguments with attribute-like syntax (e.g. materials), if strip_dot is True, only base argument names are returned.
	sfepy.discrete.equations.parse_definition(equation_def)
    Parse equation definition string to create term description list.

sfepy.discrete.evaluate module

class sfepy.discrete.evaluate.Evaluator(problem, matrix_hook=None)
    This class provides the functions required by a nonlinear solver for a given problem.

    eval_residual(vec, is_full=False)

    eval_tangent_matrix(vec, mtx=None, is_full=False)

    make_full_vec(vec)

    static new_ulf_iteration(problem, nls, vec, it, err, err0)
	sfepy.discrete.evaluate.apply_ebc_to_matrix(mtx, ebc_rows, epbc_rows=None)
    Apply E(P)BC to matrix rows: put 1 to the diagonal for EBC DOFs, 1 to the diagonal for master EPBC DOFs, -1 to the [master, slave] entries. It is assumed, that the matrix contains zeros in EBC and master EPBC DOFs rows and columns.

    When used within a nonlinear solver, the actual values on the EBC DOFs diagonal positions do not matter, as the residual is zero at those positions.
	sfepy.discrete.evaluate.assemble_by_blocks(conf_equations, problem, ebcs=None, epbcs=None, dw_mode='matrix', active_only=True)
    Instead of a global matrix, return its building blocks as defined in conf_equations. The name and row/column variables of each block have to be encoded in the equation’s name, as in:

```
conf_equations = {
}
```
Notes

*ebcs, epbcs* must be either lists of BC names, or BC configuration dictionaries.

```python
def sfepy.discrete.evaluate.create_evaluable(expression, fields, materials, variables, integrals, 
    regions=None, ebcs=None, epbcs=None, lcbcs=None, 
    ts=None, functions=None, auto_init=False, mode='eval', 
    extra_args=None, active_only=True, eterm_options=None, 
    verbose=True, kwargs=None)
```

Create evaluable object (equations and corresponding variables) from the *expression* string.

Parameters

- **expression**
  - [str] The expression to evaluate.

- **fields**
  - [dict] The dictionary of fields used in *variables*.

- **materials**
  - [Materials instance] The materials used in the expression.

- **variables**
  - [Variables instance] The variables used in the expression.

- **integrals**
  - [Integrals instance] The integrals to be used.

- **regions**
  - [Region instance or list of Region instances] The region(s) to be used. If not given, the regions defined within the fields domain are used.

- **ebcs**
  - [Conditions instance, optional] The essential (Dirichlet) boundary conditions for ‘weak’ mode.

- **epbcs**
  - [Conditions instance, optional] The periodic boundary conditions for ‘weak’ mode.

- **lcbcs**
  - [Conditions instance, optional] The linear combination boundary conditions for ‘weak’ mode.

- **ts**
  - [TimeStepper instance, optional] The time stepper.

- **functions**
  - [Functions instance, optional] The user functions for boundary conditions, materials etc.

- **auto_init**
  - [bool] Set values of all variables to all zeros.

- **mode**
  - [one of ‘eval’, ‘el_avg’, ‘qp’, ‘weak’] The evaluation mode - ‘weak’ means the finite element assembling, ‘qp’ requests the values in quadrature points, ‘el_avg’ element averages and ‘eval’ means integration over each term region.

- **extra_args**
  - [dict, optional] Extra arguments to be passed to terms in the expression.
active_only
[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

eterm_options

verbose
[bool] If False, reduce verbosity.

kwargs
[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression.

Returns

equation
[Equation instance] The equation that is ready to be evaluated.

variables
[Variables instance] The variables used in the equation.

sfepy.discrete.evaluate.eval_equations(equations, variables, names=None, preserve_caches=False, mode='eval', dw_mode='vector', term_mode=None, active_only=True, any_dof_conn=False, verbose=True)

Evaluate the equations.

Parameters

equations
[Equations instance] The equations returned by create_evaluable().

variables
[Variables instance] The variables returned by create_evaluable().

names
[str or sequence of str, optional] Evaluate only equations of the given name(s).

preserve_caches
[bool] If True, do not invalidate evaluate caches of variables.

mode
[one of ‘eval’, ‘el_avg’, ‘qp’, ‘weak’] The evaluation mode - ‘weak’ means the finite element assembling, ‘qp’ requests the values in quadrature points, ‘el_avg’ element averages and ‘eval’ means integration over each term region.

dw_mode
[‘vector’ or ‘matrix’] The assembling mode for ‘weak’ evaluation mode.

term_mode
[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

active_only
[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

any_dof_conn
[bool] If True, in ‘weak’ mode and ‘matrix’ dw_mode, all DOF connectivities are used to pre-allocate the matrix graph. If False, only cell region connectivities are used.

verbose
[bool] If False, reduce verbosity.


Returns

out

[dict or result] The evaluation result. In ‘weak’ mode it is the vector or sparse matrix, depending on \texttt{dw\_mode}. Otherwise, it is a dict of results with equation names as keys or a single result for a single equation.

\texttt{sfepy.discrete.evaluate.eval\_in\_els\_and\_qp}(expression, iels, coors, fields, materials, variables, functions=None, mode='eval', term\_mode=None, extra\_args=None, active\_only=True, verbose=True, kwargs=None)

Evaluate an expression in given elements and points.

Parameters

expression

[str] The expression to evaluate.

fields

[dict] The dictionary of fields used in \texttt{variables}.

materials

[Materials instance] The materials used in the expression.

variables

[Variables instance] The variables used in the expression.

functions

[Functions instance, optional] The user functions for materials etc.

mode

[one of ‘eval’, ‘el\_avg’, ‘qp’] The evaluation mode - ‘qp’ requests the values in quadrature points, ‘el\_avg’ element averages and ‘eval’ means integration over each term region.

term\_mode

[str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

extra\_args

[dict, optional] Extra arguments to be passed to terms in the expression.

active\_only

[bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

verbose

[bool] If False, reduce verbosity.

tkwargs

[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression.

Returns

out

[array] The result of the evaluation.
**sfepy.discrete.evaluate_variable module**

`sfepy.discrete.evaluate_variable.eval_complex(vec, conn, geo, mode, shape, bf=None)`

Evaluate basic derived quantities of a complex variable given its DOF vector, connectivity and reference mapping.

`sfepy.discrete.evaluate_variable.eval_real(vec, conn, geo, mode, shape, bf=None)`

Evaluate basic derived quantities of a real variable given its DOF vector, connectivity and reference mapping.

**sfepy.discrete.functions module**

`class sfepy.discrete.functions.ConstantFunction(values, no_tile=False)`

Function with constant values.

`class sfepy.discrete.functions.ConstantFunctionByRegion(values)`

Function with constant values in regions.

`class sfepy.discrete.functions.Function(name, function, is_constant=False, extra_args=None)`

Base class for user-defined functions.

`set_extra_args(**extra_args)`

`set_function(function, is_constant=False)`

`class sfepy.discrete.functions.Functions(objs=None, **kwargs)`

Container to hold all user-defined functions.

`static from_conf(conf)`

`sfepy.discrete.functions.make_sfepy_function(fun_or_name=None)`

Convenience decorator to quickly create `sfepy.discrete.functions.Function` objects.

Has two modes of use either without parameter:

```python
@make_sfepy_function
def my_function(...):
...
```

or with name:

```python
@make_sfepy_function("new_name_for_my_function")
def my_function(...):
...
```

**Parameters**

**fun_or_name**

[string, optional] Name to be saved within `Function` instance, if None name of decorated function is used.

**Returns**

**new_fun**

[sfepy.discrete.functions.Function] With attribute name set to provided name or original function name.
**sfepy.discrete.integrals module**

Classes for accessing quadrature points and weights for various reference element geometries.

```python
class sfepy.discrete.integrals.Integral(name, order=1, coors=None, weights=None, bounds=None, tp_fix=1.0, weight_fix=1.0, symmetric=False)
```

Wrapper class around quadratures.

```python
def get_qp(geometry)
    Get quadrature point coordinates and corresponding weights for given geometry. For built-in quadratures, the integration order is given by `self.order`.
```

**Parameters**

- **geometry**
  - [str] The geometry key describing the integration domain, see the keys of `sfepy.discrete.quadratures.quadrature_tables`.

**Returns**

- **coors**
  - [array] The coordinates of quadrature points.

- **weights**: array
  - The quadrature weights.

```python
def integrate(function, order=1, geometry='1_2')
    Integrate numerically a given scalar function.
```

**Parameters**

- **function**
  - [callable(coors)] The function of space coordinates to integrate.

- **order**
  - [int, optional] The integration order. For tensor product geometries, this is the 1D (line) order.

- **geometry**
  - [str] The geometry key describing the integration domain. Default is ‘1_2’, i.e. a line integral in [0, 1]. For other values see the keys of `sfepy.discrete.quadratures.quadrature_tables`.

**Returns**

- **val**
  - [float] The value of the integral.

```python
class sfepy.discrete.integrals.Integrals(objs=None, **kwargs)
    Container for instances of `Integral`.
```

```python
static from_conf(conf)
    Return existing or new integral.
```

**Parameters**

- **name**
  - [str] The name can either be a non-negative integer, a string representation of a non-negative integer (the integral order) or ‘a’ (automatic order) or a string beginning with ‘i’ (existing custom integral name).
sfepy.discrete.materials module

class sfepy.discrete.materials.Material(name, kind='time-dependent', function=None, values=None, flags=None, **kwargs)

A class holding constitutive and other material parameters.

Example input:

```python
material_2 = {
    'name' : 'm',
    'values' : {'E' : 1.0},
}
```

Material parameters are passed to terms using the dot notation, i.e. `m.E` in our example case.

__init__(name, kind='time-dependent', function=None, values=None, flags=None, **kwargs)

A material is defined either by a function, or by a set of constant values, potentially distinct per region. Therefore, either function must be specified, or a combination of values and **kwargs.

For constant materials, **kwargs are simply combined with values into a dictionary mapping material parameter names to parameter values. The parameter values may either be specified as a constant value, or as another dictionary mapping region names to constant values (see sfepy.discrete.functions.ConstantFunctionByRegion).

Special material parameters, that are not evaluated in quadrature points - for example flags or geometry independent data - are denoted by parameter names starting with `.` in this case the values argument need to be used, or a function that returns the parameters when mode == 'special'.

Parameters

name [str] The name of the material.

kind ['time-dependent' or 'stationary'] The kind of the material.

function [function] The function for setting up the material values.

values [dict] Constant material values.

flags [dict, optional] Special flags.

**kwargs [keyword arguments, optional] Constant material values passed by their names.

static from_conf(conf, functions)

Construct Material instance from configuration.

get_constant_data(name)

Get constant data by name.

get_data(key, name)

name can be a dict - then a Struct instance with data as attributes named as the dict keys is returned.

get_keys(region_name=None)

Get all data keys.
Parameters

region_name
[ str ] If not None, only keys with this region are returned.

iter_terms (equations, only_new=True)
Iterate terms for which the material data should be evaluated.

reduce_on_datas (reduce_fun, init=0.0)
For non-special values only!

reset()
Clear all data created by a call to time_update(), set self.mode to None.

set_all_data (datas)
Use the provided data, set mode to ‘user’.

set_data (key, qps, data)
Set the material data in quadrature points.

Parameters

key
[tuple] The (region_name, integral_name) data key.

qps
[Struct] Information about the quadrature points.

data
[dict] The material data.

set_extra_args (**extra_args)
Extra arguments passed to the material function.

set_function (function)

time_update (ts, equations, mode='normal', problem=None)
Evaluate material parameters in physical quadrature points.

Parameters

ts
[TimeStepper instance] The time stepper.

equations
[Equations instance] The equations using the materials.

mode
['normal', 'update' or 'force'] The update mode. In ‘force’ mode, self.datas is cleared and all updates are redone. In ‘update’ mode, existing data are preserved and new can be added. The ‘normal’ mode depends on other attributes: for stationary (self.kind == 'stationary') materials and materials in ‘user’ mode, nothing is done if self.datas is not empty. For time-dependent materials (self.kind == 'time-dependent', the default) that are not constant, i.e., are given by a user function, ‘normal’ mode behaves like ‘force’ mode. For constant materials it behaves like ‘update’ mode - existing data are reused.

problem
[Problem instance, optional] The problem that can be passed to user functions as a context.
update_data(key, ts, equations, term, problem=None)
Update the material parameters in quadrature points.

   Parameters
   key
       [tuple] The (region_name, integral_name) data key.
   ts
   equations
       [Equations] The equations for which the update occurs.
   term
       [Term] The term for which the update occurs.
   problem
       [Problem, optional] The problem definition for which the update occurs.

update_special_constant_data(equations=None, problem=None)
Update the special constant material parameters.

   Parameters
   equations
       [Equations] The equations for which the update occurs.
   problem
       [Problem, optional] The problem definition for which the update occurs.

update_special_data(ts, equations, problem=None)
Update the special material parameters.

   Parameters
   ts
   equations
       [Equations] The equations for which the update occurs.
   problem
       [Problem, optional] The problem definition for which the update occurs.

class sfepy.discrete.materials.Materials(objs=None, **kwargs)

   static from_conf(conf, functions, wanted=None)
Construct Materials instance from configuration.

   reset()
Clear material data so that next materials.time_update() is performed even for stationary materials.

   time_update(ts, equations, mode='normal', problem=None, verbose=True)
Update material parameters for given time, problem, and equations.

   Parameters
   ts
       [TimeStepper instance] The time stepper.
   equations
       [Equations instance] The equations using the materials.
mode
    ['normal', 'update' or 'force'] The update mode, see Material.time_update().

problem
    [Problem instance, optional] The problem that can be passed to user functions as a context.

verbose
    [bool] If False, reduce verbosity.

sfepy.discrete.parse_equations module

class sfepy.discrete.parse_equations.TermParse

sfepy.discrete.parse_equations.collect_term(term_descs, lc)
sfepy.discrete.parse_equations.create_bnf(term_descs)
    term_descs .. list of TermParse objects (sign, term_name, term_arg_names), where sign can be real or complex multiplier

sfepy.discrete.parse_equations.rhs(lc)

sfepy.discrete.parse_regions module

Grammar for selecting regions of a domain.
Regions serve for selection of certain parts of the computational domain represented as a finite element mesh. They are used to define the boundary conditions, the domains of terms and materials etc.

Notes

History: pre-git versions already from 13.06.2006.
sfepy.discrete.parse_regions.create_bnf(stack)
sfepy.discrete.parse_regions.join_tokens(str, loc, toks)
sfepy.discrete.parse_regions.print_leaf(level, op)
sfepy.discrete.parse_regions.print_op(level, op, item1, item2)
sfepy.discrete.parse_regions.print_stack(stack)
sfepy.discrete.parse_regions.replace(what, keep=False)
sfepy.discrete.parse_regions.replace_with_region(what, r_index)
sfepy.discrete.parse_regions.to_stack(stack)
sfepy.discrete.parse_regions.visit_stack(stack, op_visitor, leaf_visitor)
sfepy.discrete.probes module

Classes for probing values of Variables, for example, along a line.

**class sfepy.discrete.probes.CircleProbe(centre, normal, radius, n_point, share_geometry=True)**

Probes variables along a circle.

If `n_point` is positive, that number of evenly spaced points is used. If `n_point` is None or non-positive, an adaptive refinement based on element diameters is used and the number of points and their spacing are determined automatically. If it is negative, `-n_point` is used as an initial guess.

**get_points(refine_flag=None)**

Get the probe points.

- **Returns**
  - **pars**
    - [array_like] The independent coordinate of the probe.
  - **points**
    - [array_like] The probe points, parametrized by pars.

  **is_cyclic** = True

**report()**

Report the probe parameters.

**class sfepy.discrete.probes.IntegralProbe(name, problem, expressions, labels)**

Evaluate integral expressions.

**class sfepy.discrete.probes.LineProbe(p0, p1, n_point, share_geometry=True)**

Probes variables along a line.

If `n_point` is positive, that number of evenly spaced points is used. If `n_point` is None or non-positive, an adaptive refinement based on element diameters is used and the number of points and their spacing are determined automatically. If it is negative, `-n_point` is used as an initial guess.

**get_points(refine_flag=None)**

Get the probe points.

- **Returns**
  - **pars**
    - [array_like] The independent coordinate of the probe.
  - **points**
    - [array_like] The probe points, parametrized by pars.

**report()**

Report the probe parameters.

**class sfepy.discrete.probes.PointsProbe(points, share_geometry=True)**

Probes variables in given points.

**get_points(refine_flag=None)**

Get the probe points.

- **Returns**
  - **pars**
    - [array_like] The independent coordinate of the probe.
points
[array_like] The probe points, parametrized by pars.

refine_points(variable, points, cache)
No refinement for this probe.

report()
Report the probe parameters.

class sfepy.discrete.probes.Probe(name, share_geometry=True, n_point=None, **kwargs)
Base class for all point probes. Enforces two points minimum.

cache = Struct:probe_shared_evaluate_cache

giveget actual_cache(pars, cache, hash_chunk_size=100000)
Return the actual evaluate cache, which is a combination of the (mesh-based) evaluate cache and probe-specific data, like the reference element coordinates. The reference element coordinates are reused, if the sha1 hash of the probe parameter vector does not change.

giveget_evaluate_cache()
Return the evaluate cache for domain-related data given by self.share geometry.

is cyclic = False

probe(variable, mode='val', ret_points=False)
Probe the given variable.

Parameters

variable
[Variable instance] The variable to be sampled along the probe.

mode
[['val', 'grad'], optional] The evaluation mode: the variable value (default) or the variable value gradient.

ret_points
[bool] If True, return also the probe points.

Returns

pars
[array] The parametrization of the probe points.

points
[array, optional] If ret points is True, the coordinates of points corresponding to pars, where the variable is evaluated.

vals
[array] The probed values.

static refinepars(pars, refine_flag, cyclic_val=None)
Refine the probe parametrization based on the refine_flag.

refine_points(variable, points, cells)
Mark intervals between points for a refinement, based on element sizes at those points. Assumes the points to be ordered.

Returns
refine_flag
[bool array] True at places corresponding to intervals between subsequent points that need to be refined.

report()
Report the probe parameters.

reset_refinement()
Reset the probe refinement state.

set_n_point(n_point)
Set the number of probe points.

Parameters
n_point
[int] The (fixed) number of probe points, when positive. When non-positive, the number of points is adaptively increased starting from -n_point, until the neighboring point distance is less than the diameter of the elements enclosing the points. When None, it is set to -10.

set_options(close_limit=None, size_hint=None)
Set the probe options.

Parameters
close_limit
[float] The maximum limit distance of a point from the closest element allowed for extrapolation.

size_hint
[float] Element size hint for the refinement of probe parametrization.

class sfepy.discrete.probes.RayProbe(p0, dirvec, p_fun, n_point, both_dirs, share_geometry=True)
Probe variables along a ray. The points are parametrized by a function of radial coordinates from a given point in a given direction.

gen_points(sign)
Generate the probe points and their parametrization.

get_points(refine_flag=None)
Get the probe points.

Returns
pars
[array_like] The independent coordinate of the probe.

points
[array_like] The probe points, parametrized by pars.

refine_points(variable, points, cache)
No refinement for this probe.

report()
Report the probe parameters.

sfepy.discrete.probes.get_data_name(fd)
Try to read next data name in file fd.

Returns
name
[str] The data name.

c
[int] The number of data columns.
sfepy.discrete.probes.read_header(fd)
Read the probe data header from file descriptor fd.

Returns

header
[Struct instance] The probe data header.
sfepy.discrete.probes.read_results(filename, only_names=None)
Read probing results from a file.

Parameters

filename
[Opc] The probe results file name.

Returns

header
[Struct instance] The probe data header.
results
[dict] The dictionary of probing results. Keys are data names, values are the probed values.
sfepy.discrete.probes.write_results(filename, probe, results)
Write probing results into a file.

Parameters

filename
[Opc] The output file name.
probe
[Probe subclass instance] The probe used to obtain the results.
results
[dict] The dictionary of probing results. Keys are data names, values are the probed values.

sfepy.discrete.problem module

class sfepy.discrete.problem.Problem(name, conf=None, functions=None, domain=None, fields=None, equations=None, auto_conf=True, active_only=True)

Problem definition, the top-level class holding all data necessary to solve a problem.

It can be constructed from a ProblemConf instance using Problem.from_conf() or directly from a problem description file using Problem.from_conf_file()

For interactive use, the constructor requires only the equations, nls and ls keyword arguments, see below.

Parameters

name
[Opc] The problem name.

conf
functions
[Functions instance, optional] The user functions for boundary conditions, materials, etc.

domain
[Domain instance, optional] The solution Domain.

fields
[dict, optional] The dictionary of Field instances.

equations
[Equations instance, optional] The Equations to solve. This argument is required when auto_conf is True.

auto_conf
[bool] If True, fields and domain are determined by equations.

active_only
[bool] If True, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs, see below.

Notes
The Problem is by default created with active_only set to True. Then the (tangent) matrices and residual vectors (right-hand sides) have reduced sizes and contain only the active DOFs, i.e., DOFs not constrained by EBCs or EPBCs.

Setting active_only to False results in full-size vectors and matrices. Then the matrix size non-zeros structure does not depend on the actual E(P)BCs applied. It must be False when using parallel PETSc solvers.

The active DOF connectivities contain all DOFs, with the E(P)BC-constrained ones stored as \(-1 - \langle DOF number \rangle\), so that the full connectivities can be reconstructed for the matrix graph creation. However, the negative entries mean that the assembled matrices/residuals have zero values at positions corresponding to constrained DOFs.

The resulting linear system then provides a solution increment, that has to be added to the initial guess used to compute the residual, just like in the Newton iterations. The increment of the constrained DOFs is automatically zero.

When solving with a direct solver, the diagonal entries of a matrix at positions corresponding to constrained DOFs has to be set to ones, so that the matrix is not singular, see sfepy.discrete.evaluate.apply_ebc_to_matrix(), which is called automatically in sfepy.discrete.evaluate.Evaluator.eval_tangent_matrix(). It is not called automatically in Problem.evaluate(). Note that setting the diagonal entries to one might not be necessary with iterative solvers, as the zero matrix rows match the zero residual rows, i.e. if the reduced matrix would be regular, then the right-hand side (the residual) is orthogonal to the kernel of the matrix.

advance(ts=None)

block_solve(state0=None, status=None, save_results=True, step_hook=None, post_process_hook=None, verbose=True)
Call Problem.solve() sequentially for the individual matrix blocks of a block-triangular matrix. It is called by Problem.solve() if the 'block_solve' option is set to True.

clear_equations()

copy(name=None)
Make a copy of Problem.
create_evaluable(expression, try_equations=True, auto_init=False, preserve_caches=False,
copy_materials=True, integrals=None, ebcs=None, epbcs=None, lcbcs=None, ts=None,
functions=None, mode='eval', var_dict=None, strip_variables=True, extra_args=None,
active_only=True, eterm_options=None, verbose=True, **kwargs)

Create evaluable object (equations and corresponding variables) from the expression string. Convenience function calling create_evaluable() with defaults provided by the Problem instance self.

The evaluable can be repeatedly evaluated by calling eval_equations(), e.g. for different values of variables.

Parameters

expression
[str] The expression to evaluate.

try_equations
[bool] Try to get variables from self.equations. If this fails, variables can either be provided in var_dict, as keyword arguments, or are created automatically according to the expression.

auto_init
[bool] Set values of all variables to all zeros.

preserve_caches
[bool] If True, do not invalidate evaluate caches of variables.

copy_materials
[bool] Work with a copy of self.equations.materials instead of reusing them. Safe but can be slow.

integrals
[Integrals instance, optional] The integrals to be used. Automatically created as needed if not given.

ebcs
[Conditions instance, optional] The essential (Dirichlet) boundary conditions for ‘weak’ mode. If not given, self.ebcs are used.

epbcs
[Conditions instance, optional] The periodic boundary conditions for ‘weak’ mode. If not given, self.epbcs are used.

lcbcs
[Conditions instance, optional] The linear combination boundary conditions for ‘weak’ mode. If not given, self.lcbcs are used.

ts
[TimeStepper instance, optional] The time stepper. If not given, self.ts is used.

functions
[Functions instance, optional] The user functions for boundary conditions, materials etc. If not given, self.functions are used.

mode
[one of ‘eval’, ‘el_avg’, ‘qp’, ‘weak’] The evaluation mode - ‘weak’ means the finite element assembling, ‘qp’ requests the values in quadrature points, ‘el_avg’ element averages and ‘eval’ means integration over each term region.

var_dict
[dict, optional] The variables (dictionary of (variable name) : (Variable instance)) to be used in the expression. Use this if the name of a variable conflicts with one of the parameters of this method.
strip_variables
  [bool] If False, the variables in var_dict or kwargs not present in the expression are added to the actual variables as a context.

eextra_args
  [dict, optional] Extra arguments to be passed to terms in the expression.

active_only
  [bool] If True, in ‘weak’ mode, the (tangent) matrices and residual vectors (right-hand sides) contain only active DOFs.

eterm_options

verbose
  [bool] If False, reduce verbosity.

**kwargs
  [keyword arguments] Additional variables can be passed as keyword arguments, see var_dict.

Returns

equations
  [Equations instance] The equations that can be evaluated.

variables
  [Variables instance] The corresponding variables. Set their values and use eval_equations().

Examples

problem is Problem instance.

```python
>>> out = problem.create_evaluable('ev_integrate.ii.Omega(u)')
>>> equations, variables = out
```

tvec is a vector of coefficients compatible with the field of ‘u’ - let's use all ones.

```python
>>> 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:

```python
>>> 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.
create_state()

create_subproblem(var_names, known_var_names)

Create a sub-problem with equations containing only terms with the given virtual variables.

Parameters

var_names
  [list] The list of names of virtual variables.

known_var_names
  [list] The list of names of (already) known state variables.

Returns

subpb

create_variables(var_names=None)

Create variables with names in var_names. Their definitions have to be present in self.conf.variables.

Notes

This method does not change self.equations, so it should not have any side effects.

eval_equations(names=None, preserve_caches=False, mode='eval', dw_mode='vector', term_mode=None, active_only=True, any_dof_conn=False, verbose=True)

Evaluate (some of) the problem's equations, convenience wrapper of eval_equations().

Parameters

names
  [str or sequence of str, optional] Evaluate only equations of the given name(s).

preserve_caches
  [bool] If True, do not invalidate evaluate caches of variables.

mode
  [one of 'eval', 'el_avg', 'qp', 'weak'] The evaluation mode - 'weak' means the finite element assembling, 'qp' requests the values in quadrature points, 'el_avg' element averages and 'eval' means integration over each term region.

dw_mode
  ['vector' or 'matrix'] The assembling mode for 'weak' evaluation mode.

term_mode
  [str] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

verbose
  [bool] If False, reduce verbosity.

Returns

out
  [dict or result] The evaluation result. In ‘weak’ mode it is the vector or sparse matrix,
depending on \texttt{dw\_mode}. Otherwise, it is a dict of results with equation names as keys or a single result for a single equation.

\textbf{evaluate}(\texttt{expression}, \texttt{try\_equations=True}, \texttt{auto\_init=False}, \texttt{preserve\_caches=False}, \texttt{copy\_materials=True}, \texttt{integrals=None}, \texttt{ebcs=None}, \texttt{epbcs=None}, \texttt{lcbcs=None}, \texttt{ts=None}, \texttt{functions=None}, \texttt{mode=\textquoteleft eval\textquoteleft}, \texttt{dw\_mode=\textquoteleft vector\textquoteleft}, \texttt{term\_mode=None}, \texttt{var\_dict=None}, \texttt{strip\_variables=True}, \texttt{ret\_variables=False}, \texttt{any\_dof\_conn=False}, \texttt{active\_only=True}, \texttt{eterm\_options=None}, \texttt{verbose=True}, \texttt{extra\_args=None}, \texttt{**kwargs})

Evaluate an expression, convenience wrapper of \texttt{Problem.create\_evaluable()} and \texttt{eval\_equations()}.

\textbf{Parameters}

\texttt{dw\_mode}

[\textquoteleft vector\textquoteright{} or \textquoteleft matrix\textquoteright{}] The assembling mode for \textquoteleft weak\textquoteright{} evaluation mode.

\texttt{term\_mode}

[\texttt{str}] The term call mode - some terms support different call modes and depending on the call mode different values are returned.

\texttt{ret\_variables}

[\texttt{bool}] If True, return the variables that were created to evaluate the expression.

\texttt{other}

[arguments] See docstrings of \texttt{Problem.create\_evaluable()}.

\textbf{Returns}

\texttt{out}

[\texttt{array}] The result of the evaluation.

\texttt{variables}

[\texttt{Variables instance}] The variables that were created to evaluate the expression. Only provided if \texttt{ret\_variables} is True.

\textbf{static from\_conf}(\texttt{conf}, \texttt{init\_fields=True}, \texttt{init\_equations=True}, \texttt{init\_solvers=True})

\textbf{static from\_conf\_file}(\texttt{conf\_filename}, \texttt{required=None}, \texttt{other=None}, \texttt{init\_fields=True}, \texttt{init\_equations=True}, \texttt{init\_solvers=True})

\textbf{get\_default\_ts}(\texttt{t0=None}, \texttt{t1=None}, \texttt{dt=None}, \texttt{n\_step=None}, \texttt{step=None})

\textbf{get\_dim}(\texttt{get\_sym=False})

Returns mesh dimension, symmetric tensor dimension (if \texttt{get\_sym} is True).

\textbf{get\_evaluator} (\texttt{reuse=False})

Either create a new Evaluator instance (\texttt{reuse == False}), or return an existing instance, created in a preceding call to \texttt{Problem.init\_solvers()}.

\textbf{get\_initial\_state}(\texttt{vec=None})

Create a zero state and apply initial conditions.

\textbf{get\_integrals}(\texttt{names=None})

Get integrals, initialized from problem configuration if available.

\textbf{Parameters}
names
[ list, optional ] If given, only the named integrals are returned.

Returns
integrals
[Integrals instance] The requested integrals.

get_ls()
get_materials()
get_mesh_coors( actual=False )
get_meshes_from_region( reg_names )
get_nls()
get_nls_functions()
Returns functions to be used by a nonlinear solver to evaluate the nonlinear function value (the residual) and its gradient (the tangent matrix) corresponding to the problem equations.

Returns
 fun
 [ function ] The function \( f(x) \) for computing the residual.

 fun_grad
 [ function ] The function \( \nabla f(x) \) for computing the tangent matrix.

 iter_hook
 [ function ] The optional (user-defined) function to be called before each nonlinear solver iteration.

get_output_name( suffix=None, extra=None, mode=None )
Return default output file name, based on the output directory, output format, step suffix and mode. If present, the extra string is put just before the output format suffix.

get_restart_filename( ts=None )
If restarts are allowed in problem definition options, return the restart file name, based on the output directory and time step.

get_solver()
get_solver_conf( name )
get_timestepper()
get_tss()
get_tss_functions( update_bcs=True, update_materials=True, save_results=True, step_hook=None, post_process_hook=None )
Get the problem-dependent functions required by the time-stepping solver during the solution process.

Parameters
 update_bcs
 [ bool, optional ] If True, update the boundary conditions in each \( \text{prestep}_ \text{fun} \) call.

 update_materials
 [ bool, optional ] If True, update the values of material parameters in each \( \text{prestep}_ \text{fun} \) call.
save_results
[bool, optional] If True, save the results in each poststep_fun call.

step_hook
[callable, optional] The optional user-defined function that is called in each poststep_fun call before saving the results.

post_process_hook
[callable, optional] The optional user-defined function that is passed in each poststep_fun to Problem.save_state().

Returns

init_fun
[callable] The initialization function called before the actual time-stepping.

prestep_fun
[callable] The function called in each time (sub-)step prior to the nonlinear solver call.

poststep_fun
[callable] The function called at the end of each time step.

get_variables\((auto_create=False)\)

init_solvers\((status=None, ls_conf=None, nls_conf=None, tsc_conf=None, ts_conf=None, force=False)\)
Create and initialize solver instances.

Parameters

status
[dict-like, IndexedStruct, optional] The user-supplied object to hold the time-stepping/nonlinear solver convergence statistics.

ls_conf
[Struct, optional] The linear solver options.

nls_conf
[Struct, optional] The nonlinear solver options.

tsc_conf
[Struct, optional] The time step controller options.

ts_conf
[Struct, optional] The time stepping solver options.

force
[bool] If True, re-create the solver instances even if they already exist in self.nls attribute.

init_time\(ts\)

is_linear()

load_restart\(filename, ts=None\)
Load the current state and time step from a restart file.
Alternatively, a regular output file in the HDF5 format can be used in place of the restart file. In that case the restart is only approximate, because higher order field DOFs (if any) were stripped out. Files with the adaptive linearization are not supported. Use with caution!

Parameters

filename
[str] The restart file name.
ts
[TimeStepper instance, optional] The time stepper. If not given, a default one is created. Otherwise, it is modified in place.

Returns

variables
[Variables instance] The loaded variables.

refine_uniformly(level)
Refine the mesh uniformly level-times.

Notes
This operation resets almost everything (fields, equations, …) - it is roughly equivalent to creating a new Problem instance with the refined mesh.

remove_bcs()
Convenience function to remove boundary conditions.

reset()

save_ebc(filename, epcs=None, epbcs=None, force=True, default=0.0)
Save essential boundary conditions as state variables.

Parameters

filename
[ str] The output file name.

ebcs
[Conditions instance, optional] The essential (Dirichlet) boundary conditions. If not given, self.conf.ebcs are used.

epbcs
[Conditions instance, optional] The periodic boundary conditions. If not given, self.conf.epbcs are used.

force
[ bool] If True, sequential nonzero values are forced to individual ebcs so that the conditions are visible even when zero.

default
[ float] The default constant value of state vector.

save_regions(filename_trunk, region_names=None)
Save regions as meshes.

Parameters

filename_trunk
[ str] The output filename without suffix.

region_names
[ list, optional] If given, only the listed regions are saved.

save_regions_as_groups(filename_trunk, region_names=None)
Save regions in a single mesh but mark them by using different element/node group numbers.

See Domain.save_regions_as_groups() for more details.
Parameters

filename_trunk
[str] The output filename without suffix.

region_names
[list, optional] If given, only the listed regions are saved.

save_restart(filename, ts=None)
Save the current state and time step to a restart file.

Parameters

filename
[str] The restart file name.

ts
[TimeStepper instance, optional] The time stepper. If not given, a default one is created.

Notes

Does not support terms with internal state.

save_state(filename, state=None, out=None, fill_value=None, post_process_hook=None, linearization=None, split_results_by=None, **kwargs)

Parameters

split_results_by
[None, ‘region’, ‘variable’] If ‘region’ or ‘variable’, data of each region/variable are stored in a separate file. If None, it is set to the application option value.

linearization
[Struct or None] The linearization configuration for higher order approximations. If its kind is ‘adaptive’, split_results_by is assumed ‘variable’.

select_bcs(ebc_names=None, epbc_names=None, lcbc_names=None, create_matrix=False, any_dof_conn=None)

select_materials(material_names, only_conf=False)

select_variables(variable_names, only_conf=False)

set_bcs(ebcs=None, epbcs=None, lcbcs=None)
Update boundary conditions.

set_conf_solvers(conf_solvers=None, options=None)
Choose which solvers should be used. If solvers are not set in options, use the ones named ls, nls, ts and optionally tsc. If such solver names do not exist, use the first of each required solver kind listed in conf_solvers.

set_default_state(vec=None)
Return variables with an initialized state.

A convenience function that obtains the problem equations’ variables, initializes the state ones with zeros (default) or using vec and then returns the variables.
**set_equations** *(conf_equations=None, user=None, keep_solvers=False, make_virtual=False)*

Set equations of the problem using the *equations* problem description entry.

Fields and Regions have to be already set.

**set_equations_instance** *(equations, keep_solvers=False)*

Set equations of the problem to *equations*.

**set_fields** *(conf_fields=None)*

**set_ics** *(ics=None)*

Set the initial conditions to use.

**set_linear** *(is_linear)*

**set_materials** *(conf_materials=None)*

Set definition of materials.

**set_mesh_coors** *(coors, update_fields=False, actual=False, clear_all=True, extra_dofs=False)*

Set mesh coordinates.

**Parameters**

- **coors** *(array)* The new coordinates.
- **update_fields** *(bool)* If True, update also coordinates of fields.
- **actual** *(bool)* If True, update the actual configuration coordinates, otherwise the undeformed configuration ones.

**set_output_dir** *(output_dir=None)*

Set the directory for output files.

The directory is created if it does not exist.

**set_regions** *(conf_regions=None, conf_materials=None, functions=None, allow_empty=False)*

**set_solver** *(solver, status=None)*

Set a time-stepping or nonlinear solver to be used in *Problem.solve()* call.

**Parameters**

- **solver** *(NonlinearSolver or TimeSteppingSolver instance)* The nonlinear or time-stepping solver.
- **status** *(dict-like, optional)* The user-supplied object to hold the solver convergence statistics.
Notes

A copy of the solver is used, and the nonlinear solver functions are set to those returned by `Problem.get_nls_functions()`, if not set already. If a nonlinear solver is set, a default StationarySolver instance is created automatically as the time-stepping solver. Also sets `self.ts` attribute.

If `self.conf.options.auto_transform_equations` is True (the default is False), the problem equations are automatically transformed to a form suitable for the given solver. Implemented for `ElastodynamicsBaseTS`-based solvers. If it is False, `solver.var_names` have to be defined.

`set_variables(conf_variables=None)`
Set definition of variables.

`setup_default_output(conf=None, options=None)`
Provide default values to `Problem.setup_output()` from `conf.options` and `options`.

`setup_hooks(options=None)`
Setup various hooks (user-defined functions), as given in `options`.

Supported hooks:

- `matrix_hook`
  - check/modify tangent matrix in each nonlinear solver iteration
- `nls_iter_hook`
  - called prior to every iteration of nonlinear solver, if the solver supports that
  - takes the Problem instance (`self`) as the first argument

`setup_output(output_filename_trunk=None, output_dir=None, output_format=None, file_format=None, float_format=None, split_results_by=None, linearization=None)`
Sets output options to given values, or uses the defaults for each argument that is None.

`solve(state0=None, status=None, force_values=None, var_data=None, update_bcs=True, update_materials=True, save_results=True, step_hook=None, post_process_hook=None, post_process_hook_final=None, verbose=True)`
Solve the problem equations by calling the top-level solver.

Before calling this function the top-level solver has to be set, see `Problem.set_solver()`. Also, the boundary conditions and the initial conditions (for time-dependent problems) has to be set, see `Problem.set_bcs()`, `Problem.set_ics()`.

Parameters

- `state0` [array, optional] If given, the initial state - then the initial conditions stored in the Problem instance are ignored. By default, the initial state is created and the initial conditions are applied automatically.

- `status` [dict-like, optional] The user-supplied object to hold the solver convergence statistics.

- `force_values` [dict of floats or float, optional] If given, the supplied values override the values of the essential boundary conditions.

- `var_data` [dict, optional] A dictionary of `{variable_name : data vector}` used to initialize parameter variables.
update_bcs
- [bool, optional] If True, update the boundary conditions in each `prestep_fun` call. See `Problem.get_tss_functions()`.

update_materials
- [bool, optional] If True, update the values of material parameters in each `prestep_fun` call. See `Problem.get_tss_functions()`.

save_results
- [bool, optional] If True, save the results in each `poststep_fun` call. See `Problem.get_tss_functions()`.

step_hook
- [callable, optional] The optional user-defined function that is called in each `poststep_fun` call before saving the results. See `Problem.get_tss_functions()`.

post_process_hook
- [callable, optional] The optional user-defined function that is passed in each `poststep_fun` to `Problem.save_state()`. See `Problem.get_tss_functions()`.

post_process_hook_final
- [callable, optional] The optional user-defined function that is called after the top-level solver returns.

Returns

variables
- [Variables] The variables with the final time step state.

time_update
- `(ts=None, ebcs=None, epbcs=None, lcbcs=None, functions=None, create_matrix=False, is_matrix=True, any_dof_conn=None)`

try_presolve
- `(mtx)`

update_equations
- `(ts=None, ebcs=None, epbcs=None, lcbcs=None, functions=None, create_matrix=False, is_matrix=True, any_dof_conn=None)`

Update equations for current time step.

The tangent matrix graph is automatically recomputed if the set of active essential or periodic boundary conditions changed w.r.t. the previous time step.

Parameters

ts
- [TimeStepper instance, optional] The time stepper. If not given, `self.ts` is used.

ebcs
- [Conditions instance, optional] The essential (Dirichlet) boundary conditions. If not given, `self.ebcs` are used.

epbcs
- [Conditions instance, optional] The periodic boundary conditions. If not given, `self.epbcs` are used.

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.
create_matrix
[bool] If True, force the matrix graph computation.

is_matrix
[bool] If False, the matrix is not created. Has precedence over create_matrix.

any_dof_conn
[bool or None, default False] If True, all DOF connectivities are used to pre-allocate the matrix graph. If False, only cell region connectivities are used. If None, the value is, if available, taken from conf.options.

update_materials(ts=None, mode='normal', verbose=True)
Update materials used in equations.

Parameters

- ts
  [TimeStepper instance] The time stepper.

- mode
  ['normal', 'update' or 'force'] The update mode, see Material.time_update().

- verbose
  [bool] If False, reduce verbosity.

update_time_stepper(ts)

sfepy.discrete.problem.make_is_save(options)
Given problem options, return a callable that determines whether to save results of a time step.

sfepy.discrete.problem.prepare_matrix(problem, state)
Pre-assemble tangent system matrix.

sfepy.discrete.projections module

Construct projections between FE spaces.

sfepy.discrete.projections.create_mass_matrix(field)
Create scalar mass matrix corresponding to the given field.

Returns

- mtx
  [csr_matrix] The mass matrix in CSR format.

sfepy.discrete.projections.make_h1_projection_data(target, eval_data)
Project scalar data given by a material-like eval_data() function to a scalar target field variable using the $H^1$ dot product.

sfepy.discrete.projections.make_l2_projection(target, source, ls=None, nls_options=None)
Project a scalar source field variable to a scalar target field variable using the $L^2$ dot product.

sfepy.discrete.projections.make_l2_projection_data(target, eval_data, order=None, ls=None, nls_options=None)
Project scalar data to a scalar target field variable using the $L^2$ dot product.

Parameters

- target
  [FieldVariable instance] The target variable.
eval_data
[callable or array] Either a material-like function eval_data(), or an array of values in quadrature points that has to be reshapable to the shape required by order.

order
[int, optional] The quadrature order. If not given, it is set to 2 * target.field.approx_order.

sfepy.discrete.projections.project_by_component(tensor, tensor_qp, component, order, ls=None, nls_options=None)
Wrapper around make_l2_projection_data() for non-scalar fields.

sfepy.discrete.projections.project_to_facets(region, fun, dpn, field)
Project a function fun to the field in facets of the given region.

sfepy.discrete.quadratures module

quadrature_tables are organized as follows:

```
quadrature_tables = {
    '<geometry1>' : {
        order1 : QuadraturePoints(args1),
        order2 : QuadraturePoints(args2),
        ...
    },
    '<geometry2>' : {
        order1 : QuadraturePoints(args1),
        order2 : QuadraturePoints(args2),
        ...
    },
    ...
}
```

Note The order for quadratures on tensor product domains ('2_4', '3_8' geometries) in case of composite Gauss quadratures (products of 1D quadratures) holds for each component separately, so the actual polynomial order may be much higher (up to order * dimension).

Naming conventions in problem description files:
```
`<family>_<order>_<dimension>`
```

Integral ‘family’ is just an arbitrary name given by user.


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
The generating function was adapted from pytools/hedge codes (http://mathema.tician.de/software/hedge) by Andreas Kloeckner.

```python
class sfepy.discrete.quadratures.QuadraturePoints(data, coors=None, weights=None, bounds=None, tp_fix=1.0, weight_fix=1.0, symmetric=False)
```

Representation of a set of quadrature points.

**Parameters**

- **data**
  - [array_like] The array of shape \((n\_point, \text{dim} + 1)\) of quadrature point coordinates (first \(\text{dim}\) columns) and weights (the last column).

- **coors**
  - [array_like, optional] Optionally, instead of using \(data\), the coordinates and weights can be provided separately - \(data\) are then ignored.

- **weights**
  - [array_like, optional] Optionally, instead of using \(data\), the coordinates and weights can be provided separately - \(data\) are then ignored.

- **bounds**
  - [(float, float), optional] The coordinates and weights should correspond to a reference element in \([0, 1] \times \text{dim}\). Provide the correct bounds if this is not the case.

- **tp_fix**
  - [float, optional] The value that is used to multiply the tensor product element volume (= 1.0) to get the correct volume.

- **weight_fix**
  - [float, optional] The value that is used to multiply the weights to get the correct values.

- **symmetric**
  - [bool] If True, the integral is 1D and the given coordinates and weights are symmetric w.r.t. the centre of bounds; only the non-negative coordinates are given.

```python
static from_table(geometry, order)
```

Create a new \(QuadraturePoints\) instance, given reference element geometry name and polynomial order.

For tensor product geometries, the polynomial order is the 1D (line) order.

```python
sfepy.discrete.quadratures.get_actual_order(geometry, order)
```

Return the actual integration order for given geometry.

**Parameters**

- **geometry**
  - [str] The geometry key describing the integration domain, see the keys of \(quadrature\_tables\).

**Returns**

- **order**
  - [int] If \(order\) is in quadrature tables it is this value. Otherwise it is the closest higher order. If no higher order is available, a warning is printed and the highest available order is used.
**sfepy.discrete.simplex_cubature module**

Generate simplex quadrature points. Code taken and adapted from pytools/hedge by Andreas Kloeckner.

```python
sfepy.discrete.simplex_cubature.factorial(n)
```

```python
sfepy.discrete.simplex_cubature.generate_decreasing_nonnegative_tuples_summing_to(n, length, min=0, max=None)
```

```python
sfepy.discrete.simplex_cubature.generate_permutations(original)
```

Generate all permutations of the list `original`.

Nicked from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/252178

```python
sfepy.discrete.simplex_cubature.generate_unique_permutations(original)
```

Generate all unique permutations of the list `original`.

```python
sfepy.discrete.simplex_cubature.get_simplex_cubature(order, dimension)
```

Cubature on an $M{n}$-simplex.


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

\[ T_n := \{(x_1, \ldots, x_n) : x_i \geq -1, \sum x_i \leq -1\} \]

```python
sfepy.discrete.simplex_cubature.wandering_element(length, wanderer=1, landscape=0)
```

**sfepy.discrete.variables module**

Classes of variables for equations/terms.

```python
class sfepy.discrete.variables.DGFieldVariable(name, kind, field, order=None, primary_var_name=None, special=None, flags=None, history=None, **kwargs)
```

Field variable specifically intended for use with DGFields, bypasses application of EBC and EPBC as this is done in DGField.

Is instance checked in create_adof_conns.

```python
apply_ebc(vec, offset=0, force_values=None)
```

Apply essential (Dirichlet) and periodic boundary conditions to vector `vec`, starting at `offset`.

```python
get_full(r_vec, r_offset=0, force_value=None, vec=None, offset=0)
```

Get the full DOF vector satisfying E(P)BCs from a reduced DOF vector.
Notes

The reduced vector starts in $r_{vec}$ at $r_{offset}$. Passing a $force_{value}$ overrides the EBC values. Optionally, $vec$ argument can be provided to store the full vector (in place) starting at $offset$.

class sfepy.discrete.variables.FieldVariable

A finite element field variable.

field .. field description of variable (borrowed)

apply_ebc(vec, offset=0, force_values=None)

Apply essential (Dirichlet) and periodic boundary conditions to vector $vec$, starting at $offset$.

apply_ic(vec, offset=0, force_values=None)

Apply initial conditions conditions to vector $vec$, starting at $offset$.

clear_evaluate_cache()

Clear current evaluate cache.

create_output(vec=None, key=None, extend=True, fill_value=None, linearization=None)

Convert the DOF vector to a dictionary of output data usable by Mesh.write().

Parameters

vec
[array, optional] An alternative DOF vector to be used instead of the variable DOF vector.

key
[str, optional] The key to be used in the output dictionary instead of the variable name.

extend
[bool] Extend the DOF values to cover the whole domain.

fill_value
[float or complex] The value used to fill the missing DOF values if $extend$ is True.

linearization
[Struct or None] The linearization configuration for higher order approximations.

equation_mapping(bcs, var_di, ts, functions, problem=None, warn=False)

Create the mapping of active DOFs from/to all DOFs.

Sets $n_{adof}$.

Returns

active_bcs
[set] The set of boundary conditions active in the current time.

evaluate(mode='val', region=None, integral=None, integration=None, step=0, time_derivative=None, trace_region=None, dt=None, bf=None)

Evaluate various quantities related to the variable according to $mode$ in quadrature points defined by $integral$.

The evaluated data are cached in the variable instance in $evaluate_cache$ attribute.

Parameters

mode
region
[Region instance, optional] The region where the evaluation occurs. If None, the underlying field region is used.

integral
[Integral instance, optional] The integral defining quadrature points in which the evaluation occurs. If None, the first order volume integral is created. Must not be None for surface integrations.

integration
[‘cell’, ‘facet’, ‘facet_extra’, or ‘point’] The term integration type. If None, it is derived from the region kind.

step
[int, default 0] The time step (0 means current, -1 previous, ...).

time_derivative
[None or ‘dt’] If not None, return time derivative of the data, approximated by the backward finite difference.

trace_region
[None or str] If not None, evaluate of trace of the variable on a boundary region.

dt
[float, optional] The time step to be used if derivative is ‘dt’. If None, the dt attribute of the variable is used.

bf
[Base function, optional] The base function to be used in ‘val’ mode.

Returns

out
[array] The 4-dimensional array of shape \((n_{el}, n_{qp}, n_{row}, n_{col})\) with the requested data, where \(n_{row}, n_{col}\) depend on mode.

evaluate_at(coors, mode=’val’, strategy=’general’, close_limit=0.1, get_cells_fun=None, cache=None, ret_cells=False, ret_status=False, ret_ref_coors=False, verbose=False)

Evaluate the variable in the given physical coordinates. Convenience wrapper around Field. evaluate_at(), see its docstring for more details.

get_data_shape(integral, integration=’cell’, region_name=None)
Get element data dimensions for given approximation.

Parameters

integral
[Integral instance] The integral describing used numerical quadrature.

integration
[‘cell’, ‘facet’, ‘facet_extra’, ‘point’ or ‘custom’] The term integration mode.

region_name
[str] The name of the region of the integral.

Returns

data_shape
[5 ints] The \((n_{el}, n_{qp}, dim, n_{en}, n_{comp})\) for volume shape kind, \((n_{fa}, n_{qp}, dim, n_{fn}, n_{comp})\) for surface shape kind and \((n_{nod}, 0, 0, 1, n_{comp})\) for point shape kind.
Notes

• \( n_{el}, n_{fa} \) = number of elements/facets
• \( n_{qp} \) = number of quadrature points per element/facet
• \( \text{dim} \) = 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

\texttt{get\_dof\_conn(region\_name, dct, trace\_region=None)}

Get active dof connectivity of a variable.

Notes

The primary and dual variables must have the same Region.

\texttt{get\_dof\_info(active=False)}

\texttt{get\_element\_diameters(cells, mode, square=False)}

Get diameters of selected elements.

\texttt{get\_field()}

\texttt{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 \( offset \).

\texttt{get\_interp\_coors(strategy='interpolation', interp\_term=None)}

Get the physical coordinates to interpolate into, based on the strategy used.

\texttt{get\_mapping(region, integral, integration, get\_saved=False, return\_key=False)}

Get the reference element mapping of the underlying field.

See also:

\texttt{sfepy.discrete.common.fields.Field.get\_mapping}

\texttt{get\_reduced(vec, offset=0, follow\_epbc=False)}

Get the reduced DOF vector, with EBC and PBC DOFs removed.
Notes

The full vector starts in `vec` at `offset`. If ‘follow_epbc’ is True, values of EPBC master DOFs are not simply thrown away, but added to the corresponding slave DOFs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

`get_state_in_region(region, reshape=True, step=0)`

Get DOFs of the variable in the given region.

Parameters

- **region**
  - [Region] The selected region.
- **reshape**
  - [bool] If True, reshape the DOF vector to a 2D array with the individual components as columns. Otherwise a 1D DOF array of the form [all DOFs in region node 0, all DOFs in region node 1, ...] is returned.
- **step**
  - [int, default 0] The time step (0 means current, -1 previous, ...).

Returns

- **out**
  - [array] The selected DOFs.

`has_ebc(vec=None, force_values=None)`

`has_same_mesh(other)`

Returns

- **flag**
  - [int] The flag can be either ‘different’ (different meshes), ‘deformed’ (slightly deformed same mesh), or ‘same’ (same).

`invalidate_evaluate_cache(step=0)`

Invalidate variable data in evaluate cache for time step given by `step` (0 is current, -1 previous, ...).

This should be done, for example, prior to every nonlinear solver iteration.

`save_as_mesh(filename)`

Save the field mesh and the variable values into a file for visualization. Only the vertex values are stored.

`set_from_function(fun, step=0)`

Set the variable data (the vector of DOF values) using a function of space coordinates.

Parameters

- **fun**
  - [callable] The function of coordinates returning DOF values of shape `(n_coor, n_components)`.
- **step**
  - [int, optional] The time history step, 0 (default) = current.

`set_from_mesh_vertices(data)`

Set the variable using values at the mesh vertices.
set_from_other(other, strategy='projection', close_limit=0.1)

Set the variable using another variable. Undefined values (e.g. outside the other mesh) are set to numpy.nan, or extrapolated.

Parameters

strategy

[‘projection’ or ‘interpolation’] The strategy to set the values: the L^2 orthogonal projection (not implemented!), or a direct interpolation to the nodes (nodal elements only!)

Notes

If the other variable uses the same field mesh, the coefficients are set directly.

set_from_qp(data_qp, integral, step=0)

Set DOFs of variable using values in quadrature points corresponding to the given integral.

setup_initial_conditions(ics, di, functions, warn=False)

Setup of initial conditions.

time_update(ts, functions)

Store time step, set variable data for variables with the setter function.

class sfepy.discrete.variables.Variable(name, kind, order=None, primary_var_name=None, special=None, flags=None, **kwargs)

advance(ts)

Advance in time the DOF state history. A copy of the DOF vector is made to prevent history modification.

static from_conf(key, conf, fields)

get_dual()

Get the dual variable.

Returns

var

[Variable instance] The primary variable for non-state variables, or the dual variable for state variables.

get_initial_condition()

get_primary()

Get the corresponding primary variable.

Returns

var

[Variable instance] The primary variable, or self for state variables or if primary_var_name is None, or None if no other variables are defined.

get_primary_name()

init_data(step=0)

Initialize the dof vector data of time step step to zeros.

init_history()

Initialize data of variables with history.
is_complex()
is_finite(step=0, derivative=None, dt=None)
is_kind(kind)
is_parameter()
is_real()
is_state()
is_state_or_parameter()
is_virtual()

set_constant(val=0.0, step=0)
    Set the variable dof vector data of time step step to a scalar val.

set_data(data=None, indx=None, step=0, preserve_caches=False)
    Set data (vector of DOF values) of the variable.

    Parameters
    ----------
    data : array
        The vector of DOF values.
    indx : int, optional
        If given, data[indx] is used.
    step : int, optional
        The time history step, 0 (default) = current.
    preserve_caches : bool
        If True, do not invalidate evaluate caches of the variable.

time_update(ts, functions)
    Implemented in subclasses.

class sfepy.discrete.variables.Variables(variables=None)
    Container holding instances of Variable.

    advance(ts)

apply_ebc(vec=None, force_values=None)
    Apply essential (Dirichlet) and periodic boundary conditions to state all variables or the given vector vec.

apply_ic(vec=None, force_values=None)
    Apply initial conditions to all state variables or the given vector vec.

check_vec_size(vec, reduced=False)
    Check whether the shape of the DOF vector corresponds to the total number of DOFs of the state variables.

    Parameters
    ----------
    vec : array
        The vector of DOF values.
    reduced : bool
        If True, the size of the DOF vector should be reduced, i.e. without DOFs fixed by boundary conditions.
create_output(vec=None, fill_value=None, var_info=None, extend=True, linearization=None)

Creates an output dictionary with state variables data, that can be passed as ‘out’ kwarg to Mesh.write(). Then the dictionary entries are formed by components of the state vector corresponding to unknown variables according to kind of linearization given by linearization.

create_reduced_vec()

create_vec()

equation_mapping(ebcs, epbcs, ts, functions, problem=None, active_only=True)

Create the mapping of active DOFs from/to all DOFs for all state variables.

Parameters

- **ebcs**
  [Conditions instance] The essential (Dirichlet) boundary conditions.

- **epbcs**
  [Conditions instance] The periodic boundary conditions.

- **ts**
  [TimeStepper instance] The time stepper.

- **functions**
  [Functions instance] The user functions for boundary conditions.

- **problem**
  [Problem instance, optional] The problem that can be passed to user functions as a context.

- **active_only**
  [bool] If True, the active DOF info self.adi uses the reduced (active DOFs only) numbering. Otherwise it is the same as self.di.

Returns

- **active_bcs**
  [set] The set of boundary conditions active in the current time.

fill_state(value)

Fill the DOF vector with given value.

static from_conf(conf, fields)

get_dual_names()

Get names of pairs of dual variables.

Returns

- **duals**
  [dict] The dual names as virtual name : state name pairs.

get_index(var_name, reduced=False, allow_dual=False)

get_lcbc_operator()

get_matrix_shape()

get_reduced_state(follow_epbc=False, force=False)

Get the reduced DOF vector, with EBC and PBC DOFs removed.

get_state(reduced=False, follow_epbc=False, force=False)
get_state_parts(vec=None)
Return parts of a state vector corresponding to individual state variables.

Parameters
vec
[array, optional] The state vector. If not given, then the data stored in the variables are returned instead.

Returns
out
[dict] The dictionary of the state parts.

get_vec_part(vec, var_name, reduced=False)

has_ebc(vec=None, force_values=None, verbose=False)

has_virtuals()

init_history()

init_state(vec=None)

invalidate_evaluate_caches(step=0)

iter_state(ordered=True)

link_duals()
Link state variables with corresponding virtual variables, and assign link to self to each variable instance. Usually, when solving a PDE in the weak form, each state variable has a corresponding virtual variable.

make_full_vec(svec, force_value=None, vec=None)
Make a full DOF vector satisfying E(P)BCs from a reduced DOF vector.

Parameters
svec
[array] The reduced DOF vector.

force_value
[float, optional] Passing a force_value overrides the EBC values.

vec
[array, optional] If given, the buffer for storing the result (zeroed).

Returns
vec
[array] The full DOF vector.

reduce_vec(vec, follow_epbc=False, svec=None)
Get the reduced DOF vector, with EBC and PBC DOFs removed.
Notes

If ‘follow_epbc’ is True, values of EPBC master dofs are not simply thrown away, but added to the corresponding slave dofs, just like when assembling. For vectors with state (unknown) variables it should be set to False, for assembled vectors it should be set to True.

set_adof_conns(adof_conns)
Set all active DOF connectivities to self as well as relevant sub-dicts to the individual variables.

set_data(data, step=0, ignore_unknown=False, preserve_caches=False)
Set data (vectors of DOF values) of variables.

Parameters

data
[array] The state vector or dictionary of {variable_name : data vector}.

step
[int, optional] The time history step, 0 (default) = current.

ignore_unknown
[bool, optional] Ignore unknown variable names if data is a dict.

preserve_caches
[bool] If True, do not invalidate evaluate caches of variables.

set_full_state(vec, force=False, preserve_caches=False)
Set the full DOF vector (including EBC and PBC DOFs). If var_name is given, set only the DOF sub-vector corresponding to the given variable. If force is True, setting variables with LCBC DOFs is allowed.

set_reduced_state(r_vec, preserve_caches=False)
Set the reduced DOF vector, with EBC and PBC DOFs removed.

Parameters

r_vec
[array] The reduced DOF vector corresponding to the variables.

preserve_caches
[bool] If True, do not invalidate evaluate caches of variables.

set_state(vec, reduced=False, force=False, preserve_caches=False, apply_ebc=False)

set_state_parts(parts, vec=None, force=False)
Set parts of the DOF vector corresponding to individual state variables.

Parameters

parts
[dict] The dictionary of the DOF vector parts.

force
[bool] If True, proceed even with LCBCs present.

set_vec_part(vec, var_name, part, reduced=False)

setup_dof_info(make_virtual=False)
Setup global DOF information.
setup_dtype()
    Setup data types of state variables - all have to be of the same data type, one of `nm.float64` or `nm.complex128`.

setup_initial_conditions(ics, functions)

setup_lcbc_operators(lcbcs, ts=None, functions=None)
    Prepare linear combination BC operator matrix and right-hand side vector.

setup_ordering()
    Setup ordering of variables.

time_update(ts, functions, verbose=True)

sfepy.discrete.variables.create_adof_conn(eq, conn, dpn, offset)
    Given a node connectivity, number of DOFs per node and equation mapping, create the active dof connectivity. Locally (in a connectivity row), the DOFs are stored DOF-by-DOF (\(u_0\) in all local nodes, \(u_1\) in all local nodes, ...). Globally (in a state vector), the DOFs are stored node-by-node (\(u_0, u_1, \ldots, u_X\) in node 0, \(u_0, u_1, \ldots, u_X\) in node 1, ...).

sfepy.discrete.variables.create_adof_conns(conn_info, var_indx=None, active_only=True, verbose=True)
    Create active DOF connectivities for all variables referenced in `conn_info`.
    If a variable has not the equation mapping, a trivial mapping is assumed and connectivity with all DOFs active is created.
    DOF connectivity key is a tuple (primary variable name, region name, type, trace_region).

**Notes**

If `active_only` is False, the DOF connectivities contain all DOFs, with the E(P)BC-constrained ones stored as -1 - <DOF number>, so that the full connectivities can be reconstructed for the matrix graph creation.

sfepy.discrete.variables.expand_basis(basis, dpn)
    Expand basis for variables with several components (DOFs per node), in a way compatible with `create_adof_conn()`, according to `dpn` (DOF-per-node count).

sfepy.discrete.common sub-package

Common lower-level code and parent classes for FEM and IGA.

sfepy.discrete.common.dof_info module

Classes holding information on global DOFs and mapping of all DOFs - equations (active DOFs).
Helper functions for the equation mapping.

class sfepy.discrete.common.dof_info.DofInfo(name)
    Global DOF information, i.e. ordering of DOFs of the state (unknown) variables in the global state vector.
append_raw(name, n_dof)
Append raw DOFs.

Parameters

name
    [str] The name of variable the DOFs correspond to.

n_dof
    [int] The number of DOFs.

append_variable(var, active=False, shared=None)
Append DOFs of the given variable.

Parameters

var
    [Variable instance] The variable to append.

active
    [bool, optional] When True, only active (non-constrained) DOFs are considered.

get_info(var_name)
Return information on DOFs of the given variable.

Parameters

var_name
    [str] The name of the variable.

get_n_dof_total()
Return the total number of DOFs of all state variables.

get_subset_info(var_names)
Return global DOF information for selected variables only. Silently ignores non-existing variable names.

Parameters

var_names
    [list] The names of the selected variables.

class sfepy.discrete.common.dof_info.EquationMap(name, dof_names, var_di)
Map all DOFs to equations for active DOFs.

get_operator()
Get the matrix operator $R$ corresponding to the equation mapping, such that the restricted matrix $A_r$ can be obtained from the full matrix $A$ by $A_r = R^T A R$. All the matrices are w.r.t. a single variables that uses this mapping.

Returns

mtx

map_equations(bcs, field, ts, functions, problem=None, warn=False)
Create the mapping of active DOFs from/to all DOFs.

Parameters

bcs
    [Conditions instance] The Dirichlet or periodic boundary conditions (single condition instances). The dof names in the conditions must already be canonized.
field
[Field instance] The field of the variable holding the DOFs.

ts
[TimeStepper instance] The time stepper.

functions
[Functions instance] The registered functions.

problem
[Problem instance, optional] The problem that can be passed to user functions as a context.

warn
[bool, optional] If True, warn about BC on non-existent nodes.

Returns
active_bcs
[set] The set of boundary conditions active in the current time.

Notes
• Periodic bc: master and slave DOFs must belong to the same field (variables can differ, though).

sfepy.discrete.common.dof_info.expand_nodes_to_dofs(nods, n_dof_per_node)
Expand DOF node indices into DOFs given a constant number of DOFs per node.

sfepy.discrete.common.dof_info.expand_nodes_to_equations(nods, dof_names, all_dof_names)
Expand vector of node indices to equations (DOF indices) based on the DOF-per-node count.
DOF names must be already canonized.

Returns
eq
[array] The equations/DOF indices in the node-by-node order.

sfepy.discrete.common.dof_info.group_chains(chain_list)
Group EPBC chains.

sfepy.discrete.common.dof_info.is_active_bc(bc, ts=None, functions=None)
Check whether the given boundary condition is active in the current time.

Returns
active
[bool] True if the condition bc is active.

sfepy.discrete.common.dof_info.resolve_chains(master_slave, chains)
Resolve EPBC chains - e.g. in corner nodes.
sfepy.discrete.common.domain module

class sfepy.discrete.common.domain.Domain(name, mesh=None, nurbs=None, bmesh=None, regions=None, verbose=False)

create_extra_tdim_region(region, functions, tdim)

create_region(name, select, kind='cell', parent=None, check_parents=True, extra_options=None, functions=None, add_to_regions=True, allow_empty=False)

Region factory constructor. Append the new region to self.regions list.

create_regions(region_defs, functions=None, allow_empty=False)

get_centroids(dim)

Return the coordinates of centroids of mesh entities with dimension dim.

has_faces()

reset_regions()

Reset the list of regions associated with the domain.

save_regions(filename, region_names=None)

Save regions as individual meshes.

Parameters

filename
[ str ] The output filename.

region_names
[ list, optional ] If given, only the listed regions are saved.

save_regions_as_groups(filename, region_names=None)

Save regions in a single mesh but mark them by using different element/node group numbers.

If regions overlap, the result is undetermined, with exception of the whole domain region, which is marked by group id 0.

Region masks are also saved as scalar point data for output formats that support this.

Parameters

filename
[ str ] The output filename.

region_names
[ list, optional ] If given, only the listed regions are saved.

sfepy.discrete.common.domain.region_leaf(domain, regions, rdef, functions, tdim)

Create/setup a region instance according to rdef.

sfepy.discrete.common.domain.region_op(level, op_code, item1, item2)
sfepy.discrete.common.extmods._fmfield module

sfepy.discrete.common.extmods._geommech module

Low level functions.
sfepy.discrete.common.extmods._geommech.geme_mulAVSB3py()

sfepy.discrete.common.extmods.assemble module

Low level finite element assembling functions.
sfepy.discrete.common.extmods.assemble.assemble_matrix()
sfepy.discrete.common.extmods.assemble.assemble_matrix_complex()
sfepy.discrete.common.extmods.assemble.assemble_vector()
sfepy.discrete.common.extmods.assemble.assemble_vector_complex()

sfepy.discrete.common.extmods.cmapping module

Low level reference mapping functionality.
class sfepy.discrete.common.extmods.cmapping.CMapping

sfepy.discrete.common.extmods.cmesh module

C Mesh data structures and functions.
class sfepy.discrete.common.extmods.cmesh.CConnectivity

Notes

The memory is allocated/freed in C - this class just wraps NumPy arrays around that data without copying.
cprint()
indices
n_incident
num
offset
offsets
class sfepy.discrete.common.extmods.cmesh.CMesh

cell_groups
cell_types
```

conns
coors
cprint()
create_new()

Create a new CMesh instance, with cells corresponding to the given entities of dimension dent.

**Parameters**

entities
[array, optional] The selected topological entities of the mesh to be in the new mesh. If not given, a copy of the mesh based on the cell-vertex connectivity is returned.

dent
[int, optional] The topological dimension of the entities.

localize
[bool] If True, strip the vertices not used in the the resulting sub-mesh cells and renumber the connectivity.

**Returns**

cmesh
[CMesh] The new mesh with the cell-vertex connectivity. Other connectivities have to be created and local entities need to be set manually.

dim
edge_oris
entities
face_oris
facet_oris
free_connectivity()
from_data()

Fill CMesh data using Python data.

get_cell_conn()

get_centroids()

Return the coordinates of centroids of mesh entities with dimension dim.

get_complete()

Get entities of dimension dim that are completely given by entities of dimension dent listed in entities.

get_conn()

get_conn_as_graph()

Get d1 -> d2 connectivity as a sparse matrix graph (values = ones).

For safety, creates a copy of the connectivity arrays. The connectivity is created if necessary.
```
get_facet_normals()
Return the normals of facets for each mesh cell. The normals can be accessed using the cell-facet connectivity.

If which is -1, two normals of each quadrilateral face are averaged. If it is 0 or 1, the corresponding normal is used.

get_incident()
Get non-unique entities indices of dimension dim that are contained in entities of dimension dent listed in entities. As each of entities can be in several entities of dimension dent, offsets array is returned optionally.

get_local_entities()

get_local_ids()
Get local ids of entities of dimension dent in non-unique entities incident of dimension dim (with given offsets per entities) incident to entities, see mesh_get_incident().

The function searches entities in incident -> entities connectivity for each non-unique entity in incident.

get_orientations()
Get orientations of entities of dimension dim. Alternatively, co-dimension can be specified using codim argument.

get_surface_facets()
Get facets (edges in 2D, faces in 3D) on the mesh surface.

get_volumes()
Return the volumes of mesh entities with dimension dim > 0.

key_to_index

n_coor
n_el
num

set_local_entities()

setup_connectivity()

setup_entities()
Set up mesh edge (2D and 3D) and face connectivities (3D only) as well as their orientations.

tdim

vertex_groups

sfepy.discrete.common.extmods.cmesh.cmem_statistics()

sfepy.discrete.common.extmods.cmesh.create_mesh_graph()
Create sparse (CSR) graph corresponding to given row and column connectivities.

Parameters

n_row
[int] The number of row connectivity nodes.

n_col
[int] The number of column connectivity nodes.
\[ n_{\text{gr}} \]
[\text{int}] The number of element groups.

\[ \text{rconns} \]
[\text{list of arrays}] The list of length \( n_{\text{gr}} \) of row connectivities.

\[ \text{cconns} \]
[\text{list of arrays}] The list of length \( n_{\text{gr}} \) of column connectivities.

\textbf{Returns}

\[ \text{nnz} \]
[\text{int}] The number of graph nonzeros.

\[ \text{prow} \]
[\text{array}] The array of CSR row pointers.

\[ \text{icol} \]
[\text{array}] The array of CSR column indices.

\texttt{sfepy.discrete.common.extmods.cmesh.get_cmem_usage()}  

\texttt{sfepy.discrete.common.extmods.cmesh.graph_components()}  

Determine connected components of a compressed sparse graph.

\textbf{Returns}

\[ n_{\text{comp}} \]
[\text{int}] The number of components.

\[ \text{flag} \]
[\text{array}] The flag marking for each node its component.

\texttt{sfepy.discrete.common.extmods.cmesh.orient_elements()}  

Swap element nodes so that its volume is positive.

\texttt{sfepy.discrete.common.extmods.crefcoors module}

class \texttt{sfepy.discrete.common.extmods.crefcoors.CBasisContext}

\texttt{sfepy.discrete.common.extmods.crefcoors.evaluate_in_rc()}  

Evaluate source field DOF values or gradients in the given reference element coordinates using the given interpolation.

1. Evaluate basis functions or gradients of basis functions in the reference coordinates. For gradients, transform the values to the material coordinates. 2. Interpolate source values using the basis functions/gradients.

Interpolation uses field approximation connectivity.

\texttt{sfepy.discrete.common.extmods.crefcoors.find_ref_coors()}

\texttt{sfepy.discrete.common.extmods.crefcoors.find_ref_coors_convex()}
**sfepy.discrete.common.fields module**

**class sfepy.discrete.common.fields.Field(**kwargs**)**

Base class for fields.

**clear_mappings(clear_all=False)**

Clear current reference mappings.

**create_eval_mesh()**

Create a mesh for evaluating the field. The default implementation returns None, because this mesh is for most fields the same as the one created by `Field.create_mesh()`.

**evaluate_at(coors, source_vals, mode='val', strategy='general', close_limit=0.1, get_cells_fun=None, cache=None, ret_cells=False, ret_status=False, ret_ref_coors=False, verbose=False)**

Evaluate source DOF values corresponding to the field in the given coordinates using the field interpolation.

**Parameters**

- **coors**
  
  [array, shape (n_coor, dim)] The coordinates the source values should be interpolated into.

- **source_vals**
  
  [array, shape (n_nod, n_components)] The source DOF values corresponding to the field.

- **mode**
  
  [\{'val', 'grad', 'div', 'cauchy_strain',\}] The evaluation mode: the field value (default), the field value gradient, divergence, or cauchy strain.

- **strategy**
  
  [\{'general', 'convex'\}] The strategy for finding the elements that contain the coordinates. For convex meshes, the 'convex' strategy might be faster than the 'general' one.

- **close_limit**
  
  [float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

- **get_cells_fun**
  
  [callable, optional] If given, a function with signature get_cells_fun(coors, cmesh, **kwargs) returning cells and offsets that potentially contain points with the coordinates coors. Applicable only when strategy is 'general'. When not given, `get_potential_cells()` is used.

- **cache**
  
  [Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as cache.ref_coors, cache.cells and cache.status, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored. See `Field.get_evaluate_cache()`.

- **ret_ref_coors**
  
  [bool, optional] If True, return also the found reference element coordinates.

- **ret_status**
  
  [bool, optional] If True, return also the enclosing cell status for each point.

- **ret_cells**
  
  [bool, optional] If True, return also the cell indices the coordinates are in.
verbose
[bool] If False, reduce verbosity.

Returns
vals
[array] The interpolated values with shape \((n_{\text{coor}}, n_{\text{components}}, 1)\) or gradients with shape \((n_{\text{coor}}, n_{\text{components}}, \text{dim})\) according to the mode. If ret_status is False, the values where the status is greater than one are set to \textbf{numpy.nan}.

ref_coors
[array] The found reference element coordinates, if ret_ref_coors is True.

cells
[array] The cell indices, if ret_ref_coors or ret_cells or ret_status are True.

status
[array] The status, if ret_ref_coors or ret_status are True, with the following meaning: 0 is success, 1 is extrapolation within close_limit, 2 is extrapolation outside close_limit, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If close_limit is 0, then for the 'general' strategy the status 5 indicates points outside of the field domain that had no potential cells.

\textbf{static from_args}(\textit{name}, \textit{dtype}, \textit{shape}, \textit{region}, \textit{approx_order=1}, \textit{space='H1', poly_space_base='lagrange'})
Create a Field subclass instance corresponding to a given space.

Parameters

\textit{name}
[str] The field name.

\textit{dtype}
[numpy.dtype] The field data type: float64 or complex128.

\textit{shape}
[int/tuple/str] The field shape: 1 or (1,) or ‘scalar’, space dimension (2, or (2,) or 3 or (3,)) or ‘vector’, or a tuple. The field shape determines the shape of the FE base functions and is related to the number of components of variables and to the DOF per node count, depending on the field kind.

\textit{region}
[Region] The region where the field is defined.

\textit{approx_order}
[int/str] The FE approximation order, e.g. 0, 1, 2, ‘1B’ (1 with bubble).

\textit{space}
[str] The function space name.

\textit{poly_space_base}
[str] The name of polynomial space base.
Notes

Assumes one cell type for the whole region!

```python
static from_conf(conf, regions)
```
Create a Field subclass instance based on the configuration.

```python
def get_mapping(region, integral, integration, get_saved=False, return_key=False)
```
For given region, integral and integration type, get a reference mapping, i.e. jacobians, element volumes and base function derivatives for Volume-type geometries, and jacobians, normals and base function derivatives for Surface-type geometries corresponding to the field approximation.

The mappings are cached in the field instance in `mappings` attribute. The mappings can be saved to `mappings0` using `Field.save_mappings`. The saved mapping can be retrieved by passing `get_saved=True`. If the required (saved) mapping is not in cache, a new one is created.

Returns
- mapping [FEMapping or IGMapping instance] The mapping.
- key [tuple] The key of the mapping in `mappings` or `mappings0`.

```python
def save_mappings()
```
Save current reference mappings to `mappings0` attribute.

```python
def set_dofs(fun=0.0, region=None, dpn=None, warn=None)
```
Set the values of DOFs in a given region using a function of space coordinates or value `fun`.

If `fun` is a function, the l2 projection that is global for all region facets is used to set the DOFs.

If `dpn > 1`, and `fun` is a function, it has to return the values point-by-point, i.e. all components in the first point, in the second point etc., concatenated to an array that is reshappable to the shape `(n_point, dpn)`.

Parameters
- fun [float or array of length dpn or callable] The DOF values.
- region [Region] The region containing the DOFs.
- dpn [int, optional] The DOF-per-node count. If not given, the number of field components is used.
- warn [str, optional] The warning message printed when the region selects no DOFs.

Returns
- nods [array, shape (n_dof,)] The field DOFs (or node indices) given by the region.
- vals [array, shape (n_dof, dpn)] The values of the DOFs, node-by-node when raveled in C (row-major) order.
Notes

The nodal basis fields (lagrange) reimplement this function to set DOFs directly.
The hierarchical basis field (lobatto) do not support surface mappings, so also reimplement this function.

```
sfepy.discrete.common.fields.fields_from_conf(conf, regions)
sfepy.discrete.common.fields.parse_approx_order(approx_order)
    Parse the uniform approximation order value (str or int).
sfepy.discrete.common.fields.parse_shape(shape, dim)
sfepy.discrete.common.fields.setup_extra_data(conn_info)
    Setup extra data required for non-volume integration.
```

**sfepy.discrete.common.global_interp module**

Global interpolation functions.

```
sfepy.discrete.common.global_interp.get_potential_cells(coors, cmesh, centroids=None, extrapolate=True)
    Get cells that potentially contain points with the given physical coordinates.
```

**Parameters**

- **coors**
  - [array] The physical coordinates.
- **cmesh**
  - [CMesh instance] The cmesh defining the cells.
- **centroids**
  - [array, optional] The centroids of the cells.
- **extrapolate**
  - [bool] If True, even the points that are surely outside of the cmesh are considered and assigned potential cells.

**Returns**

- **potential_cells**
  - [array] The indices of the cells that potentially contain the points.
- **offsets**
  - [array] The offsets into `potential_cells` for each point: a point `ip` is potentially in cells `potential_cells[offsets[ip]:offsets[ip+1]]`.

```
sfepy.discrete.common.global_interp.get_ref_coors(field, coors, strategy='general', close_limit=0.1, get_cells_fun=None, cache=None, verbose=False)
```

Get reference element coordinates and elements corresponding to given physical coordinates.

**Parameters**

- **field**
  - [Field instance] The field defining the approximation.
- **coors**
  - [array] The physical coordinates.
strategy

[['general', 'convex'], optional] The strategy for finding the elements that contain the coordinates. For convex meshes, the 'convex' strategy might be faster than the 'general' one.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

generate_cells_fun

[callable, optional] If given, a function with signature `generate_cells_fun(coors, cmesh, **kwargs)` returning cells and offsets that potentially contain points with the coordinates `coors`. Applicable only when `strategy` is 'general'. When not given, `generate_potential_cells()` is used.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as `cache.ref_coors`, `cache.cells` and `cache.status`, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

verbose

[bool] If False, reduce verbosity.

Returns

ref_coors

[array] The reference coordinates.

cells

[array] The cell indices corresponding to the reference coordinates.

status

[array] The status: 0 is success, 1 is extrapolation within `close_limit`, 2 is extrapolation outside `close_limit`, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If close_limit is 0, then for the 'general' strategy the status 5 indicates points outside of the field domain that had no potential cells.

sfepy.discrete.common.global_interp.get_ref_coors_convex(field, coors, close_limit=0.1, cache=None, verbose=False)

Get reference element coordinates and elements corresponding to given physical coordinates.

Parameters

field

[Field instance] The field defining the approximation.

coors

[array] The physical coordinates.

close_limit

[float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

cache

[Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as `cache.ref_coors`, `cache.cells` and `cache.status`, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

verbose

[bool] If False, reduce verbosity.
Returns

- **ref_coors**
  [array] The reference coordinates.

- **cells**
  [array] The cell indices corresponding to the reference coordinates.

- **status**
  [array] The status: 0 is success, 1 is extrapolation within `close_limit`, 2 is extrapolation outside `close_limit`, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells.

Notes

Outline of the algorithm for finding $x_i$ such that $X(x_i) = P$:

1. make inverse connectivity - for each vertex have cells it is in.
2. find the closest vertex $V$.
3. choose initial cell: $i_0 = \text{first from cells incident to } V$.
4. while not $P$ in $C_i$, change $C_i$ towards $P$, check if $P$ in new $C_i$.

`sfePy.discrete.common.global_interp.get_ref_coors_general(field, coors, close_limit=0.1, get_cells_fun=None, cache=None, verbose=False)`

Get reference element coordinates and elements corresponding to given physical coordinates.

Parameters

- **field**
  [Field instance] The field defining the approximation.

- **coors**
  [array] The physical coordinates.

- **close_limit**
  [float, optional] The maximum limit distance of a point from the closest element allowed for extrapolation.

- **get_cells_fun**
  [callable, optional] If given, a function with signature `get_cells_fun(coors, cmesh, **kwargs)` returning cells and offsets that potentially contain points with the coordinates `coors`. When not given, `get_potential_cells()` is used.

- **cache**
  [Struct, optional] To speed up a sequence of evaluations, the field mesh and other data can be cached. Optionally, the cache can also contain the reference element coordinates as `cache.ref_coors`, `cache.cells` and `cache.status`, if the evaluation occurs in the same coordinates repeatedly. In that case the mesh related data are ignored.

- **verbose**
  [bool] If False, reduce verbosity.

Returns

- **ref_coors**
  [array] The reference coordinates.
cells
[array] The cell indices corresponding to the reference coordinates.

status
[array] The status: 0 is success, 1 is extrapolation within close_limit, 2 is extrapolation outside close_limit, 3 is failure, 4 is failure due to non-convergence of the Newton iteration in tensor product cells. If close_limit is 0, then status 5 indicates points outside of the field domain that had no potential cells.

sfepy.discrete.common.mappings module

Reference-physical domain mappings.

class sfepy.discrete.common.mappings.Mapping(**kwargs)
Base class for mappings.

    static from_args(region, kind='v')
Create mapping from reference to physical entities in a given region, given the integration kind ('v' or 's').
This mapping can be used to compute the physical quadrature points.

Parameters

    region
[Region instance] The region defining the entities.

    kind
['v' or 's'] The kind of the entities: 'v' - cells, 's' - facets.

Returns

    mapping
[FEMapping or IGMapping instance] The requested mapping.

class sfepy.discrete.common.mappings.PhysicalQPs(num=0)
Physical quadrature points in a region.

get_shape(rshape)
Get shape from raveled shape.

class sfepy.discrete.common.mappings.PyCMapping(bf, det, volume, bfg, normal, dim)
Class for storing mapping data. Primary data in numpy arrays. Data for C functions translated to FMFields and embedded in CMapping.

    integrate(out, field, mode=0)

sfepy.discrete.common.mappings.get_jacobian(field, integral, region=None, integration='volume')
Get the jacobian of reference mapping corresponding to field.

Parameters

    field
[Field instance] The field defining the reference mapping.

    integral
[Integral instance] The integral defining quadrature points.

    region
[Region instance, optional] If given, use the given region instead of field region.
integration
    [one of ('volume', 'surface', 'surface_extra')] The integration type.

Returns

jac
    [array] The jacobian merged for all element groups.

See also:

*get_mapping_data*

Notes

Assumes the same element geometry in all element groups of the field!

```python
def get_mapping_data(name, field, integral, region=None, integration=None):
    pass
```

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!

```python
def get_normals(field, integral, region):
    pass
```

Get the normals of element faces in `region`.

Parameters

field
    [Field instance] The field defining the reference mapping.
integral
    [Integral instance] The integral defining quadrature points.
region
[Region instance] The given of the element faces.

Returns

normals
[array] The normals merged for all element groups.

See also:

get_mapping_data

Notes

Assumes the same element geometry in all element groups of the field!

sfepy.discrete.common.mappings.get_physical_qps(region, integral, map_kind=None)
Get physical quadrature points corresponding to the given region and integral.

sfepy.discrete.common.poly_spaces module

class sfepy.discrete.common.poly_spaces.PolySpace(name, geometry, order)
Abstract polynomial space class.

static any_from_args(name, geometry, order, base='lagrange', force_bubble=False)
Construct a particular polynomial space classes according to the arguments passed in.

eval_base(coors, diff=0, ori=None, force_axis=False, transform=None, suppress_errors=False, eps=1e-15)
Evaluate the basis or its first or second derivatives in points given by coordinates. The real work is done in _eval_base() implemented in subclasses.

Note that the second derivative code is a work-in-progress and only coors and transform arguments are used.

Parameters

coors
[array_like] The coordinates of points where the basis is evaluated. See Notes.

diff
[0, 1 or 2] If nonzero, return the given derivative.

ori
[array_like, optional] Optional orientation of element facets for per element basis.

force_axis
[bool] If True, force the resulting array shape to have one more axis even when ori is None.

transform
[array_like, optional] The basis transform array.

suppress_errors
[bool] If True, do not report points outside the reference domain.

transform
[float] Accuracy for comparing coordinates.

Returns
The basis (shape (n_coors, 1, n_base)) or its first derivative (shape (n_coors, dim, n_base)) or its second derivative (shape (n_coors, dim, dim, n_base)) evaluated in the given points. An additional axis is pre-pended of length n_cell, if ori is given, or of length 1, if force_axis is True.

Notes

If coors.ndim == 3, several point sets are assumed, with equal number of points in each of them. This is the case, for example, of the values of the volume base functions on the element facets. The indexing (of bf_b(g)) is then (ifa,iqp,:,n_ep), so that the facet can be set in C using FMF_SetCell.

keys = {(0, 1): 'simplex', (1, 2): 'simplex', (2, 3): 'simplex', (2, 4): 'tensor_product', (3, 4): 'simplex', (3, 8): 'tensor_product'}

static suggest_name(geometry, order, base='lagrange', force_bubble=False)

Suggest the polynomial space name given its constructor parameters.

sfepy.discrete.common.poly_spaces.register_poly_space(cls)

sfepy.discrete.common.poly_spaces.transform_basis(transform, bf)

Transform a basis bf using transform array of matrices.

sfepy.discrete.common.region module

class sfepy.discrete.common.region.Region(name, definition, domain, parse_def, kind='cell', parent=None, tdim=None)

Region defines a subset of a FE domain.

Region kinds:

- cell_only, facet_only, face_only, edge_only, vertex_only - only the specified entities are included, others are empty sets (so that the operators are still defined)
- cell, facet, face, edge, vertex - entities of higher dimension are not included

The ‘cell’ kind is the most general and it is the default.

Region set-like operators: + (union), - (difference), * (intersection), followed by one of ('v', 'e', 'f', 'c', and 's') for vertices, edges, faces, cells, and facets.

Created: 31.10.2005

property cells

contains(other)

Return True in the region contains the other region.

The check is performed using entities corresponding to the other region kind.

copy()

Make a copy based on the region kind.

delete_zero_faces(eps=1e-14)

property edges
eval_op_cells(other, op)
eval_op_edges(other, op)
eval_op_faces(other, op)
eval_op_facets(other, op)
eval_op_vertices(other, op)

property faces

property facets

finalize(allow_empty=False)

Initialize the entities corresponding to the region kind and regenerate all already existing (accessed) entities of lower topological dimension from the kind entities.

static from_cells(cells, domain, name='region', kind='cell', parent=None)

Create a new region containing given cells.

Parameters

cells
  [array] The array of cells.

domain
  [Domain instance] The domain containing the facets.

name
  [str, optional] The name of the region.

kind
  [str, optional] The kind of the region.

parent
  [str, optional] The name of the parent region.

Returns

obj
  [Region instance] The new region.

static from_facets(facets, domain, name='region', kind='facet', parent=None)

Create a new region containing given facets.

Parameters

facets
  [array] The array with indices to unique facets.

domain
  [Domain instance] The domain containing the facets.

name
  [str, optional] The name of the region.

kind
  [str, optional] The kind of the region.

parent
  [str, optional] The name of the parent region.

Returns
obj
[Region instance] The new region.

**static from_vertices** *(vertices, domain, name='region', kind='cell')*
Create a new region containing given vertices.

**Parameters**

- **vertices**
  [array] The array of vertices.

- **domain**
  [Domain instance] The domain containing the vertices.

- **name**
  [str, optional] The name of the region.

- **kind**
  [str, optional] The kind of the region.

**Returns**

obj
[Region instance] The new region.

**get_cell_indices** *(cells, true_cells_only=True)*
Return indices of cells in the region cells.

Raises ValueError if *true_cells_only* is True and the region kind does not allow cells. For *true_cells_only* equal to False, cells incident to facets are returned if the region itself contains no cells.

Raises ValueError if all cells are not in the region cells.

**get_cells** *(true_cells_only=True)*
Get cells of the region.

Raises ValueError if *true_cells_only* is True and the region kind does not allow cells. For *true_cells_only* equal to False, cells incident to facets are returned if the region itself contains no cells. Obey parent region, if given.

**get_charfun** *(by_cell=False, val_by_id=False)*
Return the characteristic function of the region as a vector of values defined either in the mesh vertices (by_cell == False) or cells. The values are either 1 (val_by_id == False) or sequential id + 1.

**get_edge_graph** *
Return the graph of region edges as a sparse matrix having uid(k) + 1 at (i, j) if vertex[i] is connected with vertex[j] by the edge k.
Degenerate edges are ignored.

**get_entities** *(dim)*
Return mesh entities of dimension *dim*.

**get_facet_indices** *
Return an array (per group) of (iel, ifa) for each facet. A facet can be in 1 (surface) or 2 (inner) cells.

**get_mirror_region** *(name)*

**get_n_cells** *(is_surface=False)*
Get number of region cells.

**Parameters**
is_surface
[bool] If True, number of edges or faces according to domain dimension is returned instead.

Returns

n_cells
[int] The number of cells.

has_cells()

light_copy(name, parse_def, tdim=None)

set_kind(kind)

set_kind_tdim()

setup_from_highest(dim, allow_lower=True, allow_empty=False)
Setup entities of topological dimension \textit{dim} using the available entities of the highest topological dimension.

setup_from_vertices(dim)
Setup entities of topological dimension \textit{dim} using the region vertices.

setup_mirror_region(mirror_name=None, ret_name=False)
Find the corresponding mirror region, set up element mapping.

update_shape()
Update shape of each group according to region vertices, edges, faces and cells.

property vertices

sfepy.discrete.common.region.are_disjoint(r1, r2)
Check if the regions \textit{r1} and \textit{r2} are disjoint.

Uses vertices for the check - *\_only regions not allowed.

sfepy.discrete.common.region.get_dependency_graph(region_defs)
Return a dependency graph and a name-sort name mapping for given region definitions.

sfepy.discrete.common.region.get_parents(selector)
Given a region selector, return names of regions it is based on.

sfepy.discrete.common.region.sort_by_dependency(graph)

sfepy.discrete.fem sub-package

sfepy.discrete.fem.domain module

Computational domain, consisting of the mesh and regions.

class sfepy.discrete.fem.domain.FEDomain(name, mesh, verbose=False, **kwargs)
Domain is divided into groups, whose purpose is to have homogeneous data shapes.

clear_surface_groups()
Remove surface group data.

create_surface_group(region)
Create a new surface group corresponding to \textit{region} if it does not exist yet.
Notes

Surface groups define surface facet connectivity that is needed for \texttt{sfepy.discrete.fem.mappings.FEMapping}.

\texttt{fix_element_orientation}(geom_els=None, force_check=False)
Ensure element vertices ordering giving positive cell volumes.

\texttt{get_conn}(ret_gel=False, tdim=None, cells=None)
Get the cell-vertex connectivity and, if \texttt{ret_gel} is True, also the corresponding reference geometry element. If \texttt{tdim} is not None get the connectivity of the cells with topological dimension \texttt{tdim}.

\texttt{get_diameter}()
Return the diameter of the domain.

Notes

The diameter corresponds to the Friedrichs constant.

\texttt{get_element_diameters}(cells, volume, mode, square=True)

\texttt{get_mesh_bounding_box}()
Return the bounding box of the underlying mesh.

Returns

\texttt{bbox}
\texttt{[ndarray (2, dim)]} The bounding box with min. values in the first row and max. values in the second row.

\texttt{get_mesh_coors}(actual=False)
Return the coordinates of the underlying mesh vertices.

\texttt{refine}()
Uniformly refine the domain mesh.

Returns

\texttt{domain}
\texttt{[FEDomain instance]} The new domain with the refined mesh.

Notes

Works only for meshes with single element type! Does not preserve node groups!

\texttt{sfepy.discrete.fem.extmods.bases module}

Polynomial base functions and related utilities.

\texttt{class sfepy.discrete.fem.extmods.bases.CLagrangeContext}

\texttt{baseld}
\texttt{cprint()}
\texttt{e_coors_max}
sfepy.discrete.fem.extmods.lobatto_bases module

Interface to Lobatto bases.

sfepy.discrete.fem.extmods.lobatto_bases.eval_lobatto1d()
Evaluate 1D Lobatto functions of the given order in given points.

sfepy.discrete.fem.extmods.lobatto_bases.eval_lobatto_tensor_product()
Evaluate tensor product Lobatto functions of the given order in given points.
Base functions are addressed using the nodes array with rows corresponding to individual functions and columns to 1D indices (= orders when >= 1) into lobatto[] and d_lobatto[] lists for each axis.

sfepy.discrete.fem.facets module

Helper functions related to mesh facets and Lagrange FE approximation.

Line: ori - iter:
0 - iter0 1 - iter1

Triangle: ori - iter:
0 - iter21 1 - iter12 3 - iter02 4 - iter20 6 - iter10 7 - iter01
Possible couples:
1, 4, 7 <-> 0, 3, 6

Square: ori - iter:
0 - iter10x01y 7 - iter10y01x
11 - iter01y01x 30 - iter01x10y 33 - iter10x10y 52 - iter01y10x 56 - iter10y10x 63 - iter01x01y
Possible couples:
7, 33, 52, 63 <-> 0, 11, 30, 56

_quad_ori_groups:
i < j < k < l
all faces are permuted to
l — k || | | i — j
ijkl

2.3. Developer Guide 259
which is the same as
l — |||j — k
ikjl
k — |||i — j
ijlk

- start at one vertex and go around clock-wise or anticlock-wise
-> 8 groups of 3 -> same face nodes order in ijkl (63), ikjl (59), ikjl (31) ilkj (11), iklj (15), iljk (43) jkli ( 7), jkli ( 3), kjli ( 6) kjil (56), jkli (57), jkli (48) lijk (52), ilkj (20), klij (60) lkji (33), lkji (32), iljk (41)

```
sfepy.discrete.fem.facets.build_orientation_map(n_fp)
```

The keys are binary masks of the lexicographical ordering of facet vertices. A bit set to one means \( v[i] < v[i+1] \).

The values are \([\text{original\_order, permutation}]\), where \text{permutation} can be used to sort facet vertices lexicographically. Hence \( \text{permuted\_facet} = \text{facet}[\text{permutation}] \).

```
sfepy.discrete.fem.facets.get_facet_dof_permutations(n_fp, order)
```

Prepare DOF permutation vector for each possible facet orientation.

```
sfepy.discrete.fem.facets.iter0(num)
sfepy.discrete.fem.facets.iter01(num)
sfepy.discrete.fem.facets.iter01x01y(num)
sfepy.discrete.fem.facets.iter01x10y(num)
sfepy.discrete.fem.facets.iter01y01x(num)
sfepy.discrete.fem.facets.iter01y10x(num)
sfepy.discrete.fem.facets.iter02(num)
sfepy.discrete.fem.facets.iter1(num)
sfepy.discrete.fem.facets.iter10(num)
sfepy.discrete.fem.facets.iter10x01y(num)
sfepy.discrete.fem.facets.iter10x10y(num)
sfepy.discrete.fem.facets.iter10y01x(num)
sfepy.discrete.fem.facets.iter10y10x(num)
sfepy.discrete.fem.facets.iter12(num)
sfepy.discrete.fem.facets.iter20(num)
sfepy.discrete.fem.facets.iter21(num)
sfepy.discrete.fem.facets.make_line_matrix(order)
sfepy.discrete.fem.facets.make_square_matrix(order)
sfepy.discrete.fem.facets.make_triangle_matrix(order)
```
sfepy.discrete.fem.fe_surface module

class sfepy.discrete.fem.fe_surface.FEPhantomSurface(name, region, volume_econn)
A phantom surface of the region with tdim=2.

class sfepy.discrete.fem.fe_surface.FESurface(name, region, efaces, volume_econn, volume_region=None)
Description of a surface of a finite element domain.

static from_region(name, region, ret_gel=False)

get_connectivity(local=False, trace_region=None)
Return the surface element connectivity.

Parameters

local
[bool] If True, return local connectivity w.r.t. surface nodes, otherwise return global connectivity w.r.t. all mesh nodes.

trace_trace
[None or str] If not None, return mirror connectivity according to local.

set_orientation_map()

setup_mirror_connectivity(region, mirror_name)
Setup mirror surface connectivity required to integrate over a mirror region.
1. Get orientation of the faces: a) for own elements -> ooris b) for mirror elements -> moris
2. orientation -> permutation.

sfepy.discrete.fem.fields_base module

Notes

Important attributes of continuous (order > 0) Field and SurfaceField instances:

- vertex_remap : econn[:, :n_vertex] = vertex_remap[conn]
- vertex_remap_i : conn = vertex_remap_i[econn[:, :n_vertex]]

where conn is the mesh vertex connectivity, econn is the region-local field connectivity.

class sfepy.discrete.fem.fields_base.FEField(name, dtype, shape, region, approx_order=1)
Base class for finite element fields.

Notes

- inters and hence node_descs are per region (must have single geometry!)

Field shape information:

- shape - the shape of the base functions in a point
- n_components - the number of DOFs per FE node
- val_shape - the shape of field value (the product of DOFs and base functions) in a point
**average_qp_to_vertices**(*data_qp*, *integral*)

Average data given in quadrature points in region elements into region vertices.

\[ u_n = \frac{\sum_e (u_{e,avg} \cdot area_e)}{\sum_e area_e} = \frac{\int area_e \ u}{\sum_e area_e} \]

**clear_qp_base**()

Remove cached quadrature points and base functions.

**create_bqp**(region_name, integral)

**create_mapping**(region, integral, integration, return_mapping=True)

Create a new reference mapping.

Compute jacobians, element volumes and base function derivatives for Volume-type geometries (volume mappings), and jacobians, normals and base function derivatives for Surface-type geometries (surface mappings).

**Notes**

- surface mappings are defined on the surface region
- surface mappings require field order to be > 0

**create_mesh**(extra_nodes=True)

Create a mesh from the field region, optionally including the field extra nodes.

**create_output**(dofs, var_name, dof_names=None, key=None, extend=True, fill_value=None, linearization=None)

Convert the DOFs corresponding to the field to a dictionary of output data usable by Mesh.write().

**Parameters**

- **dofs**
  - [array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

- **var_name**
  - [str] The variable name corresponding to dofs.

- **dof_names**
  - [tuple of str] The names of DOF components.

- **key**
  - [str, optional] The key to be used in the output dictionary instead of the variable name.

- **extend**
  - [bool] Extend the DOF values to cover the whole domain.

- **fill_value**
  - [float or complex] The value used to fill the missing DOF values if extend is True.

- **linearization**
  - [Struct or None] The linearization configuration for higher order approximations.

**Returns**

- **out**
  - [dict] The output dictionary.
\textbf{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.

\textbf{get_base}(key, derivative, integral, iels=None, from_geometry=False, base_only=True)

\textbf{get_coor}(nods=None)

Get coordinates of the field nodes.

**Parameters**

\textbf{nods}

[array, optional] The indices of the required nodes. If not given, the coordinates of all the nodes are returned.

\textbf{get_data_shape}(integral, integration='cell', region_name=None)

Get element data dimensions.

**Parameters**

\textbf{integral}

[Integral instance] The integral describing used numerical quadrature.

\textbf{integration}

['cell', 'facet', 'facet_extra', 'point' or 'custom'] The term integration mode.

\textbf{region_name}

[str] The name of the region of the integral.

**Returns**

\textbf{data_shape}

[4 ints] The \((n_{el}, n_{qp}, dim, n_{en})\) for volume shape kind, \((n_{fa}, n_{qp}, dim, n_{fn})\) for surface shape kind and \((n_{nod}, 0, 0, 1)\) for point shape kind.

**Notes**

Integration modes: - 'cell': integration over cells/elements - 'facet': integration over cell facets (faces, edges) - 'facet_extra': same as 'facet' but also the normal derivatives are evaluated

- 'point': point integration
- 'custom': user defined integration

Dimensions: - \(n_{el}\), \(n_{fa}\) = number of elements/facets - \(n_{qp}\) = number of quadrature points per element/facet - \(dim\) = spatial dimension - \(n_{en}\), \(n_{fn}\) = number of element/facet nodes - \(n_{nod}\) = number of element nodes

\textbf{get_dofs_in_region}(region, merge=True)

Return indices of DOFs that belong to the given region.

\textbf{get_econn}(conn_type, region, trace_region=None, local=False)

Get extended connectivity of the given type in the given region.

**Parameters**

\textbf{conn_type}: tuple or string

DOF connectivity type, eg. ('cell', 3) or 'cell'. If the topological dimension not specified, it is taken from region.tdim.
region: sfepy.discrete.common.region.Region
The region for which the connectivity is required.

trace_region: None or string
If not None, return mirror connectivity according to local.

local: bool
If True, return local connectivity w.r.t. facet nodes, otherwise return global connectivity w.r.t. all mesh nodes.

Returns

econn: numpy.ndarray
The extended connectivity array.

get_evaluate_cache(cache=None, share_geometry=False, verbose=False)
Get the evaluate cache for Variable.evaluate_at().

Parameters

cache
[Struct instance, optional] Optionally, use the provided instance to store the cache data.

share_geometry
[bool] Set to True to indicate that all the evaluations will work on the same region. Certain data are then computed only for the first probe and cached.

verbose
[bool] If False, reduce verbosity.

Returns

cache
[Struct instance] The evaluate cache.

get_output_approx_order()
Get the approximation order used in the output file.

get_qp(key, integral)
Get quadrature points and weights corresponding to the given key and integral. The key is ‘v’, ‘s#' or ‘b#’, where # is the number of face vertices. For ‘b#’, the quadrature must already be created by calling FEField.create_bqp(), usually through FEField.create_mapping().

get_true_order()
Get the true approximation order depending on the reference element geometry.
For example, for P1 (linear) approximation the true order is 1, while for Q1 (bilinear) approximation in 2D the true order is 2.

get_vertices()
Return indices of vertices belonging to the field region.

interp_to_qp(dofs)
Interpolate DOFs into quadrature points.
The quadrature order is given by the field approximation order.

Parameters
dofs
[array] The array of DOF values of shape (n_nod, n_component).

Returns
**data_qp**
[array] The values interpolated into the quadrature points.

**integral**
[Integral] The corresponding integral defining the quadrature points.

**is_higher_order()**
Return True, if the field’s approximation order is greater than one.

**linearize(dofs, min_level=0, max_level=1, eps=0.0001)**
Linearize the solution for post-processing.

**Parameters**

dofs
[array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

min_level
[int] The minimum required level of mesh refinement.

max_level
[int] The maximum level of mesh refinement.

eps
[float] The relative tolerance parameter of mesh adaptivity.

**Returns**

mesh
[Mesh instance] The adapted, nonconforming, mesh.

vdofs
[array] The DOFs defined in vertices of mesh.

levels
[array of ints] The refinement level used for each element group.

**remove_extra_dofs(dofs)**
Remove DOFs defined in higher order nodes (order > 1).

**restore_dofs(store=False)**
Undoes the effect of FEFIELD. substitute_dofs().

**restore_substituted(vec)**
Restore values of the unused DOFs using the transpose of the applied basis transformation.

**set_basis_transform(transform)**
Set local element basis transformation.

The basis transformation is applied in FEFIELD. get_base() and FEFIELD. create_mapping().

**Parameters**

transform
[array, shape (n_cell, n_ep, n_ep)] The array with (n_ep, n_ep) transformation matrices for each cell in the field’s region, where n_ep is the number of element DOFs.

**set_coors(coors, extra_dofs=False)**
Set coordinates of field nodes.

**setup_bar_data(field, region)**
setup_coors()
    Setup coordinates of field nodes.

setup_extra_data(info)

setup_point_data(field, region)

setup_surface_data(region, trace_region=None)
    nodes[econn] == econn

substitute_dofs(subs, restore=False)
    Perform facet DOF substitutions according to subs.
    Modifies self.econn in-place and sets self.econn0, self.unused_dofs and self.basis_transform.

class sfepy.discrete.fem.fields_base.H1Mixin(**kwargs)
    Methods of fields specific to H1 space.

sfepy.discrete.fem.fields_base.create_expression_output(expression, name, primary_field_name, fields, materials, variables, functions=None, mode='eval', term_mode=None, extra_args=None, verbose=True, kwargs=None, min_level=0, max_level=1, eps=0.0001)

Create output mesh and data for the expression using the adaptive linearizer.

Parameters

expression
    [str] The expression to evaluate.

name
    [str] The name of the data.

primary_field_name
    [str] The name of field that defines the element groups and polynomial spaces.

fields
    [dict] The dictionary of fields used in variables.

materials
    [Materials instance] The materials used in the expression.

variables
    [Variables instance] The variables used in the expression.

functions
    [Functions instance, optional] The user functions for materials etc.

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**


```python
sfepy.discrete.fem.fields_base.eval_nodal_coors(coors, mesh_coors, region, poly_space,
  geom_poly_space, econn, only_extra=True)
```

Compute coordinates of nodes corresponding to `poly_space`, given mesh coordinates and `geom_poly_space`.

```python
sfepy.discrete.fem.fields_base.get_eval_expression(expression, fields, materials, variables,
  functions=None, mode='eval', term_mode=None,
  extra_args=None, verbose=True, kwargs=None)
```

Get the function for evaluating an expression given a list of elements, and reference element coordinates.

```python
sfepy.discrete.fem.fields_base.set_mesh_coors(domain, fields, coors, update_fields=False,
  actual=False, clear_all=True, extra_dofs=False)
```

**sfepy.discrete.fem.fields_hierarchic module**

**class** `sfepy.discrete.fem.fields_hierarchic.H1HierarchicVolumeField`

Hierarchical basis approximation with Lobatto polynomials.

```python
create_basis_context()
```

Create the context required for evaluating the field basis.

```python
family_name = 'volume_H1_lobatto'
```

```python
set_dofs(fun=0.0, region=None, dpn=None, warn=None)
```

Set the values of DOFs in a given `region` using a function of space coordinates or value `fun`.

**sfepy.discrete.fem.fields_l2 module**

**class** `sfepy.discrete.fem.fields_l2.L2ConstantSurfaceField`

The L2 constant-in-a-region approximation.

```python
family_name = 'surface_L2_constant'
```

**class** `sfepy.discrete.fem.fields_l2.L2ConstantVolumeField`

The L2 constant-in-a-region approximation.
create_mapping(region, integral, integration, return_mapping=True)

create_output(dofs, var_name, dof_names=None, key=None, extend=True, fill_value=None, linearization=None)

family_name = 'volume_L2_constant'

get_base(key, derivative, integral, iels=None, from_geometry=False, base_only=True)

get_coor(nods=None)
    Returns the barycenter of the field region.

    Parameters

    nods
        [array, optional] Ignored.

get_data_shape(integral, integration='cell', region_name=None)
    Get element data dimensions.

    Parameters

    integral
        [Integral instance] The integral describing used numerical quadrature.

    integration
        ['cell'] The term integration type. Ignored.

    region_name
        [str] The name of the region of the integral.

    Returns

    data_shape
        [4 ints] The (n_el, n_qp, dim, n_en) for volume shape kind, (n_fa, n_qp, dim, n_fn) for surface shape kind.

Notes

- n_el, n_fa = number of elements/facets
- n_qp = number of quadrature points per element/facet
- dim = spatial dimension
- n_en, n_fn = number of element/facet nodes
- n_nod = number of element nodes

get_dofs_in_region(region, merge=True)
    Return indices of DOFs that belong to the given region.

get_econn(conn_type, region, trace_region=None, local=False)
    Get extended connectivity of the given type in the given region.

    Parameters

    conn_type: tuple or string
        DOF connectivity type, ignored.

    region: sfepy.discrete.common.region.Region
        The region for which the connectivity is required.
trace_region: None or string
  Ignored.
local: bool
  Ignored.

Returns

econn: numpy.ndarray
  The extended connectivity array.

setup_extra_data(info)

sfepy.discrete.fem.fields_nodal module

Notes

Important attributes of continuous (order > 0) Field and SurfaceField instances:

- vertex_remap : econn[:, :n_vertex] = vertex_remap[conn]
- vertex_remap_i : conn = vertex_remap_i[econn[:, :n_vertex]]

where conn is the mesh vertex connectivity, econn is the region-local field connectivity.

class sfepy.discrete.fem.fields_nodal.GlobalNodalLikeBasis(**kwargs)

get_surface_basis(region)
  Get basis for projections to region’s facets.

  Notes

  Cannot be uses for all fields because IGA does not support surface mappings.

class sfepy.discrete.fem.fields_nodal.H1DiscontinuousField(name, dtype, shape, region, approx_order=1)

  The C0 constant-per-cell approximation.

  average_to_vertices(dofs)
    Average DOFs of the discontinuous field into the field region vertices.

  extend_dofs(dofs, fill_value=None)
    Extend DOFs to the whole domain using the fill_value, or the smallest value in dofs if fill_value is None.

  family_name = 'volume_H1_lagrange_discontinuous'

  remove_extra_dofs(dofs)
    Remove DOFs defined in higher order nodes (order > 1).

class sfepy.discrete.fem.fields_nodal.H1NodalMixin(**kwargs)

  create_basis_context()
    Create the context required for evaluating the field basis.

  set_dofs(fun=0.0, region=None, dpn=None, warn=None)
    Set the values of DOFs in a given region using a function of space coordinates or value fun.
class sfepy.discrete.fem.fields_nodal.H1NodalSurfaceField(name, dtype, shape, region, approx_order=1)

A field defined on a surface region.

family_name = 'surface_H1_lagrange'

interp_v_vals_to_n_vals(vec)

Interpolate a function defined by vertex DOF values using the FE surface geometry base (P1 or Q1) into the extra nodes, i.e. define the extra DOF values.

class sfepy.discrete.fem.fields_nodal.H1NodalVolumeField(name, dtype, shape, region, approx_order=1)

Lagrange basis nodal approximation.

family_name = 'volume_H1_lagrange'

interp_v_vals_to_n_vals(vec)

Interpolate a function defined by vertex DOF values using the FE geometry base (P1 or Q1) into the extra nodes, i.e. define the extra DOF values.

class sfepy.discrete.fem.fields_nodal.H1SEMSurfaceField(name, dtype, shape, region, approx_order=1)

family_name = 'surface_H1_sem'

class sfepy.discrete.fem.fields_nodal.H1SEMVolumeField(name, dtype, shape, region, approx_order=1)

Spectral element method approximation.

Uses the Lagrange basis with Legendre-Gauss-Lobatto nodes and quadrature.

create_basis_context()

Create the context required for evaluating the field basis.

family_name = 'volume_H1_sem'

class sfepy.discrete.fem.fields_nodal.H1SNodalSurfaceField(name, dtype, shape, region, approx_order=1)

family_name = 'surface_H1_serendipity'

class sfepy.discrete.fem.fields_nodal.H1SNodalVolumeField(name, dtype, shape, region, approx_order=1)

Lagrange basis nodal serendipity approximation with order <= 3.

create_basis_context()

Create the context required for evaluating the field basis.

family_name = 'volume_H1_serendipity'
sfepy.discrete.fem.fields_positive module

class sfepy.discrete.fem.fields_positive.H1BernsteinSurfaceField(name, dtype, shape, region, approx_order=1)

    family_name = 'surface_H1_bernstein'

class sfepy.discrete.fem.fields_positive.H1BernsteinVolumeField(name, dtype, shape, region, approx_order=1)

    Bernstein basis approximation with positive-only basis function values.

    create_basis_context()
        Create the context required for evaluating the field basis.

    family_name = 'volume_H1_bernstein'

sfepy.discrete.fem.geometry_element module

GeometryElement describes the geometric entities of a finite element mesh.

Notes

• geometry_data: surface facets are assumed to be of the same kind for each geometry element - wedges or pyramids are not supported.
• the orientation is a tuple: (root1, vertices of direction vectors, swap from, swap to, root2, ...)

class sfepy.discrete.fem.geometry_element.GeometryElement(name)

    The geometric entities of a finite element mesh.

    create_surface_facet()
        Create a GeometryElement instance corresponding to this instance surface facet.

    get_conn_permutations()
        Get all possible connectivity permutations corresponding to different spatial orientations of the geometry element.

    get_edges_per_face()
        Return the indices into self.edges per face.

    get_grid(n_nod)
        Get a grid of n_nod interpolation points, including the geometry element vertices. The number of points must correspond to a valid number of FE nodes for each geometry.

    get_interpolation_name()
        Get the name of corresponding linear interpolant.

    get_surface_entities()
        Return self.vertices in 1D, self.edges in 2D and self.faces in 3D.

sfepy.discrete.fem.geometry_element.create_geometry_elements(names=None)

    Utility function to create GeometryElement instances.

Parameters
names
[<str>, optional] The names of the entity, one of the keys in geometry_data dictionary. If None, all keys of geometry_data are used.

Returns
gels
[<dict>] The dictionary of geometry elements with names as keys.
sfepy.discrete.fem.geometry_element.setup_orientation(vecs_tuple)

sfepy.discrete.fem.history module

class sfepy.discrete.fem.history.Histories(objs=None, **kwargs)
    static from_file_hdf5(filename, var_names)
        TODO: do not read entire file, provide data on demand.
class sfepy.discrete.fem.history.History(name, th=None, steps=None, times=None)
    append(item, step, time)
    static from_sequence(seq, name)

sfepy.discrete.fem.lcbc_operators module

Operators for enforcing linear combination boundary conditions in nodal FEM setting.
class sfepy.discrete.fem.lcbc_operators.EdgeDirectionOperator(name, regions, dof_names,
    dof_map_fun, filename, variables,
    ts=None, functions=None)
    Transformation matrix operator for edges direction LCBCs.
The substitution (in 3D) is:
\[
[u_1, u_2, u_3]^T = [d_1, d_2, d_3]^T w,
\]
where \(d\) is an edge direction vector averaged into a node. The new DOF is \(w\).
get_vectors(nodes, region, field, filename=None)
kind = 'edge_direction'
class sfepy.discrete.fem.lcbc_operators.IntegralMeanValueOperator(name, regions, dof_names,
    dof_map_fun, variables,
    ts=None, functions=None)
    Transformation matrix operator for integral mean value LCBCs. All DOFs in a region are summed to form a single new DOF.
    kind = 'integral_mean_value'
class sfepy.discrete.fem.lcbc_operators.LCBCOperator(name, regions, dof_names, dof_map_fun,
    variables, functions=None)
    Base class for LCBC operators.
    setup()
class sfepy.discrete.fem.lcbc_operators.LCBCOperators(name, variables, functions=None)
Container holding instances of LCBCOperator subclasses for a single variable.

add_from_bc(bc, ts)
Create a new LCBC operator described by bc, and add it to the container.

Parameters
bc
[LinearCombinationBC instance] The LCBC condition description.
ts
[TimeStepper instance] The time stepper.

append(op)

finalize()
Call this after all LCBCs of the variable have been added.
Initializes the global column indices and DOF counts.

make_global_operator(adi, new_only=False)
Assemble all LCBC operators into a single matrix.

Parameters
adi
[DofInfo] The active DOF information.
new_only
[bool] If True, the operator columns will contain only new DOFs.

Returns
mtx_lc
[csr_matrix] The global LCBC operator in the form of a CSR matrix.
rhs_lc
[array] The right-hand side for non-homogeneous LCBCs.
lcdi
[DofInfo] The global active LCBC-constrained DOF information.

class sfepy.discrete.fem.lcbc_operators.MRLCBCOperator(name, regions, dof_names, dof_map_fun, variables, functions=None)
Base class for model-reduction type LCBC operators.
These operators are applied to a single field, and replace its DOFs in a given region by new DOFs. In case some field DOFs are to be preserved, those have to be “copied” explicitly, by setting the corresponding row of the operator matrix to a single value one (see, for example, NoPenetrationOperator).

setup()

treat_pbcs(dofs, master)
Treat dofs with periodic BC.

class sfepy.discrete.fem.lcbc_operators.MatchDOFsOperator(name, regions, dof_names, dof_map_fun, variables, ts, functions)
Transformation matrix operator for match DOFs boundary conditions.
This operator ties DOFs of two fields in two disjoint regions together. It does not create any new DOFs.
class sfepy.discrete.fem.lcbc_operators.NoPenetrationOperator(name, regions, dof_names, dof_map_fun, filename, variables, ts=None, functions=None)

Transformation matrix operator for no-penetration LCBCs.

class sfepy.discrete.fem.lcbc_operators.NodalLCOperator(name, regions, dof_names, dof_map_fun, constraints, variables, ts=None, functions=None)

Transformation matrix operator for the general linear combination of DOFs in each node of a field in the given region.

The DOFs can be fully constrained - then the operator corresponds to enforcing Dirichlet boundary conditions. The linear combination is given by:

\[ \sum_{j=1}^{n} A_{ij} u_j = b_i, \forall i, \]

where \( u_j \), \( j = 1, \ldots, n \) are the DOFs in the node and \( i = 1, \ldots, m, m < n \), are the linear constraint indices. SymPy is used to solve the constraint linear system in each node for the dependent DOF(s).

class sfepy.discrete.fem.lcbc_operators.NormalDirectionOperator(name, regions, dof_names, dof_map_fun, filename, variables, ts=None, functions=None)

Transformation matrix operator for normal direction LCBCs.

The substitution (in 3D) is:

\[ \left[u_1, u_2, u_3\right]^T = \left[n_1, n_2, n_3\right]^T w \]

The new DOF is \( w \).

get_vectors(nodes, region, field, filename=None)

class sfepy.discrete.fem.lcbc_operators.RigidOperator(name, regions, dof_names, dof_map_fun, variables, ts=None, functions=None)

Transformation matrix operator for rigid LCBCs.

class sfepy.discrete.fem.lcbc_operators.ShiftedPeriodicOperator(name, regions, dof_names, dof_map_fun, shift_fun, variables, ts, functions)

Transformation matrix operator for shifted periodic boundary conditions.

This operator ties existing DOFs of two fields in two disjoint regions together. Unlike MRLCBCOperator subclasses, it does not create any new DOFs.
sfepy.discrete.fem.linearizer module

Linearization of higher order solutions for the purposes of visualization.

sfepy.discrete.fem.linearizer.create_output (eval_dofs, eval_coors, n_el, ps, min_level=0, max_level=2, eps=0.0001)

Create mesh with linear elements that approximates DOFs returned by eval_dofs() corresponding to a higher order approximation with a relative precision given by eps. The DOFs are evaluated in physical coordinates returned by eval_coors().

sfepy.discrete.fem.linearizer.get_eval_coors (coors, conn, ps)

Get default function for evaluating physical coordinates given a list of elements and reference element coordinates.

sfepy.discrete.fem.linearizer.get_eval_dofs (dofs, dof_conn, ps, ori=None)

Get default function for evaluating field DOFs given a list of elements and reference element coordinates.

sfepy.discrete.fem.mappings module

Finite element reference mappings.

class sfepy.discrete.fem.mappings.FEMapping (coors, conn, poly_space=None, gel=None, order=1)

Base class for finite element mappings.

get_base (coors, diff=False)

Get basis functions or their gradient evaluated in given coordinates.

get_geometry ()

Return reference element geometry as a GeometryElement instance.

get_mapping (qp_coors, weights, bf=None, poly_space=None, ori=None, transform=None, is_face=False, extra=(None, None, None))

Get the mapping for given quadrature points, weights, and polynomial space.

Parameters

qp_coors: numpy.ndarray
The coordinates of the integration points.

weights:
The integration weights.

bf: numpy.ndarray
The basis functions.

poly_space: PolySpace instance
The PolySpace instance.

ori: numpy.ndarray
Element orientation, used by hierarchical basis.

transform: numpy.ndarray
The transformation matrix applied to the basis functions.

is_face: bool
Is it the boundary of a region?
extra: tuple
The extra data for surface derivatives: - the derivatives of the field boundary basis functions with respect to the reference coordinates
- the boundary connectivity
- the derivatives of the domain boundary basis functions with respect to the reference coordinates

Returns
pycmap: PyCMapping instance
The domain mapping data.

get_physical_qps(qp_coors)
Get physical quadrature points corresponding to given reference element quadrature points.

Returns
qps
[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

set_basis_indices(indices)
Set indices to cell-based basis that give the facet-based basis.
sfepy.discrete.fem.mappings.eval_mapping_data_in_qp(coors, conn, bf_g, weights, ebf_g=None, is_face=False, eps=1e-15, se_conn=None, se_bf_bg=None, ecoors=None)
Evaluate mapping data.

Parameters
coors: numpy.ndarray
The nodal coordinates.
conn: numpy.ndarray
The element connectivity.
bf_g: numpy.ndarray
The derivatives of the domain basis functions with respect to the reference coordinates.
weights: numpy.ndarray
The weights of the quadrature points.
ebf_g: numpy.ndarray
The derivatives of the field basis functions with respect to the reference coordinates.
is_face: bool
Is it the boundary of a region?
eps: float
The tolerance for the normal vectors calculation.
se_conn: numpy.ndarray
The connectivity for the calculation of surface derivatives.
se_bf_bg: numpy.ndarray
The surface basis function derivatives with respect to the reference coordinates.
ecoors: numpy.ndarray
The element nodal coordinates.

Returns

det: numpy.ndarray
The determinant of the mapping evaluated in integration points.

volume: numpy.ndarray
The element (volume or surface) volumes in integration points.

bfg: numpy.ndarray
The derivatives of the basis functions with respect to the spatial coordinates. Can be evaluated either for surface elements if $bf_g$, $se_conn$, and $se_bf_bg$ are given.

normal: numpy.ndarray
The normal vectors for the surface elements in integration points.

sfepy.discrete.fem.mappings.transform_coors_to_lower_dim(coors, to_dim)
Transform element coordinates into XY plane.

See:
https://math.stackexchange.com/questions/1167717/transform-a-plane-to-the-xy-plane
https://en.wikipedia.org/wiki/Rotation_matrix#Rotation_matrix_from_axis_and_angle

sfepy.discrete.fem.mesh module

class sfepy.discrete.fem.mesh.Mesh(name='mesh', cmesh=None)
The Mesh class is a light proxy to CMesh.
Input and output is handled by the MeshIO class and subclasses.

property coors

copy(name=None)
Make a deep copy of the mesh.

Parameters

name
[str] Name of the copied mesh.

create_conn_graph(verbose=True)
Create a graph of mesh connectivity.

Returns

graph

static from_data(name, coors, ngroups, conns, mat_ids, desc, nodal_bcs=None)
Create a mesh from mesh IO data.

static from_file(filename=None, io='auto', prefix_dir=None, omit_facets=False, file_format=None)
Read a mesh from a file.

Parameters

filename
[string or function or MeshIO instance or Mesh instance] The name of file to read the mesh
from. For convenience, a mesh creation function or a MeshIO instance or directly a Mesh instance can be passed in place of the file name.

io
[*MeshIO instance] Passing *MeshIO instance has precedence over filename.

prefix_dir
[sh] If not None, the filename is relative to that directory.

omit_facets
[bool] If True, do not read cells of lower dimension than the space dimension (faces and/or edges). Only some MeshIO subclasses support this!

static from_region(region, mesh_in, localize=False, is_surface=False, tdim=None)
Create a mesh corresponding to cells, or lower dimensional entities according tdim parameter, of a given region. If is_surface is True, then tdim = dim - 1.

get_bounding_box()

get_cmesh(desc)

get_conn(desc, ret_cells=False, tdim=None)
Get the rectangular cell-vertex connectivity corresponding to desc. If ret_cells is True, the corresponding cells are returned as well.

transform_coors(mtx_t, ref_coors=None)
Transform coordinates of the mesh by the given transformation matrix.

Parameters

mtx_t
[array] The transformation matrix $T$ (2D array). It is applied depending on its shape:
- $(dim, dim)$: $x = T \times x$
- $(dim, dim + 1)$: $x = T[:, :, -1] \times x + T[:, :, -1]$

ref_coors
[array, optional] Alternative coordinates to use for the transformation instead of the mesh coordinates, with the same shape as self.coors.

write(filename=None, io=None, out=None, float_format=None, file_format=None, **kwargs)
Write mesh + optional results in out to a file.

Parameters

filename
[str, optional] The file name. If None, the mesh name is used instead.

io
[MeshIO instance or ‘auto’, optional] Passing ‘auto’ respects the extension of filename.

out
[dict, optional] The output data attached to the mesh vertices and/or cells.

float_format
[str, optional] The format string used to print floats in case of a text file format.

**kwargs
[dict, optional] Additional arguments that can be passed to the MeshIO instance.
**sfepy.discrete.fem.mesh**

- **find_map**(*x1, x2, allow_double=False, join=True*)
  Find a mapping between common coordinates in x1 and x2, such that `x1[cmap[:,0]] == x2[cmap[:,1]]`

- **fix_double_nodes**(*coor, ngroups, conns*)
  Detect and attempt fixing double nodes in a mesh.

- **get_min_vertex_distance**(*coor, guess*)
  Can miss the minimum, but is enough for our purposes.

- **get_min_vertex_distance_naive**(*coor*)

- **make_mesh**(*coor, ngroups, conns, mesh_in*)
  Create a mesh reusing mat_ids and descs of mesh_in.

- **merge_mesh**(*x1, ngroups1, conn1, mat_ids1, x2, ngroups2, conn2, mat_ids2, cmap*)
  Merge two meshes in common coordinates found in x1, x2.

**Notes**

Assumes the same number and kind of element groups in both meshes!

- **set_accuracy**(*eps*)

**sfepy.discrete.fem.meshio module**

- **class sfepy.discrete.fem.meshio.ANSYSCDBMeshIO**(*filename, **kwargs*)
  - format = 'ansys_cdb'
  - static guess(*filename*)
  - static make_format(*format, nchar=1000*)
  - read(*mesh, **kwargs*)
  - read_bounding_box()
  - read_dimension(*ret_fd=False*)
  - write(*filename, mesh, out=None, **kwargs*)

- **class sfepy.discrete.fem.meshio.ComsolMeshIO**(*filename, **kwargs*)
  - format = 'comsol'
  - read(*mesh, **kwargs*)
  - write(*filename, mesh, out=None, **kwargs*)
class sfepy.discrete.fem.meshio.GmshIO(filename, file_format=None, **kwargs)

Used to read and write data in .msh format when file_format gmsh-dg is specified. Tailored for use with Discontinuous galerking methods, mesh and ElementNodeData with InterpolationScheme can be written and read. It however omits mat_ids and node_groups.

For details on format see [1].

For details on representing and visualization of DG FEM data using gmsh see [2].


format = 'gmshio'

load_slices = {'all': slice(0, None, None), 'first': slice(0, 1, None), 'last': slice(-1, None, None)}

read_data(step=None, filename=None, cache=None)

Reads file or files with basename filename or self.filename. Considers all files to contain data from time steps of solution of single transient problem i.e. all data have the same shape, mesh and same interpolation scheme in case of ElementNodeData. Does not read mulitple NodeData or ElementData. For stationary problems just reads one file with time 0.0 and time step 0.

Providing filename allows reading multiple files of format basename.*[0-9].msh

Parameters

step
[String, int, optional] “all”, “last”, “first” or number of step to read: if “all” read all files with the basename and varying step, if “last” read only last step of all files with the filename, if “first” reads step=0, if None reads file with filename provided or specified in object.

filename
[string, optional] Filename of the files to use, if None filename from object is used. Base-name is extracted as basename.*[0-9].msh

cache
[has no effect]

Returns

out
[dictionary] Keys represent name of data, values are Structs with attributes:

data
[list, array] For ElementNodeData with shape (n_cell, n_cell_dof) contains for each time step. For other contains array of data from last time step.

time
[list] Contains times.

time_n
[list] Contains time step numbers.

scheme
[Struct] Interpolation scheme used in data, only one interpolation scheme is allowed.

scheme_name
[str] Name of the interpolation scheme, repeated for convenience.
mode

[**str**] Represents type of data. cell_nodes: for ElementNodeData; vertex or cell: Note that for vertex and cell data reading multiple time steps does not work yet.

Notes

The interpolation scheme *Struct* contains the following items:

**name**

[**string**] Name of the scheme.

**F**

[array] Coefficients matrix as defined in [1] and [2].

**P**

[array] Exponents matrix as defined in [1] and [2].

write(filename, mesh, out=None, ts=None, **kwargs)

Writes mesh and data, handles cell DOFs data from DGField as ElementNodeData.

Omits gmsh:ref for cells and vertices i.e. mat_ids and node_groups to prevent cluttering the GMSH post-processing.

Parameters

**filename**

[**string**] Path to file.

**mesh**

[**sfepy.discrete.fem.mesh.Mesh**] Computational mesh to write.

**out**

[**dictionary**] Keys represent name of the data, values are Structs with attributes:

**data**

[array] For ElementNodeData shape is (n_cell, n_cell_dof)

**mode**

[**str**] Represents type of data, cell_nodes for ElementNodeData.

For ElementNodeData:

**scheme**

[**Struct**] Interpolation scheme used in data, only one interpolation scheme is allowed.

**scheme_name**

[**str**] Name of the interpolation scheme, associated with data, repeated fo convenience.

**ts**

[**sfepy.solvers.ts.TimeStepper instance, optional**] Provides data to write time step.
Notes

The interpolation scheme Struct contains the following items:

- **name**
  [string] Name of the scheme.

- **F**
  [array] Coefficients matrix as defined in [1] and [2].

- **P**
  [array] Exponents matrix as defined in [1] and [2].

class sfepy.discrete.fem.meshio.HDF5MeshIO(filename, **kwargs)

  - **format** = 'hdf5'
  - **read**(mesh=None, **kwargs)
  - **read_bounding_box**(ret_fd=False, ret_dim=False)
  - **read_data**(step, filename=None, cache=None)
  - **read_data_header**(dname, step=None, filename=None)
  - **read_dimension**(ret_fd=False)
  - **read_last_step**(filename=None)
    The default implementation: just return 0 as the last step.
  - **static read_mesh_from_hdf5**(filename, group=None, mesh=None)
    Read the mesh from a HDF5 file.
    - **filename**: str or tables.File
      The HDF5 file to read the mesh from.
    - **group**: tables.group.Group or str, optional
      The HDF5 file group to read the mesh from. If None, the root group is used.
    - **mesh**: sfepy.discrete.fem.Mesh or None
      If None, the new mesh is created and returned, otherwise content of this argument is replaced by the read mesh.

    **Returns**
    - sfepy.discrete.fem.Mesh
      readed mesh

    **read_time_history**(node_name, indx, filename=None)

    **read_time_stepper**(filename=None)

    **read_times**(filename=None)
      Read true time step data from individual time steps.

    **Returns**
    - **steps**
      [array] The time steps.
times
[array] The times of the time steps.

nts
[array] The normalized times of the time steps, in [0, 1].

\texttt{read_variables_time_history}(var_names, ts, filename=None)

\texttt{string = <module 'string' from '/usr/lib/python3.8/string.py'>}

\texttt{write(filename, mesh, out=None, ts=None, cache=None, xdmf=False, **kwargs)}

\texttt{static write_mesh_to_hdf5(filename, group, mesh, force_3d=False)}
Write mesh to a hdf5 file.

\texttt{filename: str or tables.File}
The HDF5 file to write the mesh to.

\texttt{group: tables.group.Group or None or str}
The HDF5 file group to write the mesh to. If None, the root group is used. The group can be given as a path from root, e.g. /path/to/mesh

\texttt{mesh: sfepy.discrete.fem.Mesh}
The mesh to write.

\texttt{static write_xdmf_file(filename, **kwargs)}

class \texttt{sfepy.discrete.fem.meshio.HDF5XdmfMeshIO(filename, **kwargs)}

\texttt{format = 'hdf5-xdmf'}

\texttt{write(filename, mesh, out=None, ts=None, cache=None, xdmf=False, **kwargs)}

class \texttt{sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO(filename, **kwargs)}

\texttt{format = 'hmascii'}

\texttt{read(mesh, **kwargs)}

\texttt{read_dimension()}

\texttt{write(filename, mesh, out=None, **kwargs)}

class \texttt{sfepy.discrete.fem.meshio.Mesh3DMeshIO(filename, **kwargs)}

\texttt{format = 'mesh3d'}

\texttt{read(mesh, **kwargs)}

\texttt{read_dimension()}

class \texttt{sfepy.discrete.fem.meshio.MeshIO(filename, **kwargs)}
The abstract class for importing and exporting meshes.

Read the docstring of the Mesh() class. Basically all you need to do is to implement the read() method:

\begin{verbatim}
def read(self, mesh, **kwargs):
    nodes = ...
    ngroups = ...
    conns = ...
    mat_ids = ...
\end{verbatim}

(continues on next page)
See the Mesh class’ docstring how the nodes, ngroups, conns, mat_ids and descs should look like. You just need to read them from your specific format from disk.

To write a mesh to disk, just implement the write() method and use the information from the mesh instance (e.g. nodes, conns, mat_ids and descs) to construct your specific format.

Optionally, subclasses can implement read_data() to read also computation results. This concerns mainly the subclasses with implemented write() supporting the ‘out’ kwarg.

The default implementation od read_last_step() just returns 0. It should be reimplemented in subclasses capable of storing several steps.

**static any_from_filename** (filename, prefix_dir=None, file_format=None, mode='r')

Create a MeshIO instance according to the kind of filename.

**Parameters**

- **filename**
  [str, function or MeshIO subclass instance] The name of the mesh file. It can be also a user-supplied function accepting two arguments: mesh, mode, where mesh is a Mesh instance and mode is one of ‘read’, ‘write’, or a MeshIO subclass instance.

- **prefix_dir**
  [str] The directory name to prepend to filename.

**Returns**

- **io**
  [MeshIO subclass instance] The MeshIO subclass instance corresponding to the kind of filename.

**call_msg**  = 'called an abstract MeshIO instance!'

**format**  = None

**get_filename_trunk**()

**get_vector_format**(dim)

**read**(mesh, omit_facets=False, **kwargs)

**read_data**(step, filename=None, cache=None)

**read_last_step**()

The default implementation: just return 0 as the last step.

**read_times**(filename=None)

Read true time step data from individual time steps.

**Returns**

- **steps**
  [array] The time steps.

- **times**
  [array] The times of the time steps.
nts

[array] The normalized times of the time steps, in [0, 1].

Notes

The default implementation returns empty arrays.

set_float_format(format=None)

write(filename, mesh, **kwargs)

class sfepy.discrete.fem.meshio.MeshtoolIO(filename, file_format=None, **kwargs)


format = 'meshio'

read(mesh, omit_facets=False, **kwargs)

read_dimension(ret_fd=False)

write(filename, mesh, out=None, **kwargs)

class sfepy.discrete.fem.meshio.NEUMeshIO(filename, **kwargs)

format = 'gambit'

read(mesh, **kwargs)

read_dimension(ret_fd=False)

write(filename, mesh, out=None, **kwargs)

class sfepy.discrete.fem.meshio.UserMeshIO(filename, **kwargs)

Special MeshIO subclass that enables reading and writing a mesh using a user-supplied function.

format = 'function'

generate_filename_trunk()
class sfepy.discrete.fem.meshio.XYZMeshIO(filename, **kwargs)

Trivial XYZ format working only with coordinates (in a .XYZ file) and the connectivity stored in another file with the same base name and .IEN suffix.

format = 'xyz'

read(mesh, omit_facets=False, **kwargs)
read_bounding_box(ret_fd=False, ret_dim=False)
read_dimension(ret_fd=False)
write(filename, mesh, out=None, **kwargs)

sfepy.discrete.fem.meshio.check_format_suffix(file_format, suffix)

Check compatibility of a mesh file format and a mesh file suffix.

sfepy.discrete.fem.meshio.convert_complex_output(out_in)

Convert complex values in the output dictionary out_in to pairs of real and imaginary parts.

sfepy.discrete.fem.meshio.mesh_from_groups(mesh, ids, coors, ngroups, tris, mat_tris, quads, mat_quads, tetras, mat_tetras, hexas, mat_hexas, remap=None)

sfepy.discrete.fem.meshio.output_mesh_formats(mode='r')

sfepy.discrete.fem.meshio.split_conns_mat_ids(conns_in)

Split connectivities (columns except the last ones in conns_in) from cell groups (the last columns of conns_in).

sfepy.discrete.fem.meshio.update_supported_formats(formats)

sfepy.discrete.fem.meshio.var

alias of XYZMeshIO

sfepy.discrete.fem.periodic module

sfepy.discrete.fem.periodic.get_grid_plane(idim)

sfepy.discrete.fem.periodic.match_coors(coors1, coors2, get_saved=True)

sfepy.discrete.fem.periodic.match_grid_line(coors1, coors2, which, get_saved=True)

Match coordinates coors1 with coors2 along the axis which.

sfepy.discrete.fem.periodic.match_grid_plane(coors1, coors2, idim, get_saved=True)

sfepy.discrete.fem.periodic.match_plane_by_dir(coors1, coors2, direction, get_saved=True)

Match coordinates coors1 with coors2 in a given direction.

sfepy.discrete.fem.periodic.match_x_line(coors1, coors2, get_saved=True)

sfepy.discrete.fem.periodic.match_x_plane(coors1, coors2, get_saved=True)

sfepy.discrete.fem.periodic.match_y_line(coors1, coors2, get_saved=True)

sfepy.discrete.fem.periodic.match_y_plane(coors1, coors2, get_saved=True)
**SfePy Documentation, Release version: 2023.4+git.4f968b9d**

```python
sfepy.discrete.fem.periodic.match_z_line(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.match_z_plane(coors1, coors2, get_saved=True)
sfepy.discrete.fem.periodic.set_accuracy(eps)
```

**sfepy.discrete.fem.poly_spaces module**

```python
class sfepy.discrete.fem.poly_spaces.BernsteinSimplexPolySpace(name, geometry, order)
   Bernstein polynomial space on simplex domains.

   Notes
   Naive proof-of-concept implementation, does not use recurrent formulas or Duffy transformation to obtain tensor product structure.
   name = 'bernstein_simplex'

class sfepy.discrete.fem.poly_spaces.BernsteinTensorProductPolySpace(name, geometry, order)
   Bernstein polynomial space.
   Each row of the nodes attribute defines indices of 1D Bernstein basis functions that need to be multiplied together to evaluate the corresponding shape function. This defines the ordering of basis functions on the reference element.
   name = 'bernstein_tensor_product'

class sfepy.discrete.fem.poly_spaces.FEPolySpace(name, geometry, order)
   Base for FE polynomial space classes.
   describe_nodes()
   get_mtx_i()

class sfepy.discrete.fem.poly_spaces.LagrangeNodes(**kwargs)
   Helper class for defining nodes of Lagrange elements.
   static append_bubbles(nodes, nts, iseq, nt, order)
   static append_edges(nodes, nts, iseq, nt, edges, order)
   static append_faces(nodes, nts, iseq, nt, faces, order)
   static append_tp_bubbles(nodes, nts, iseq, nt, ao)
   static append_tp_edges(nodes, nts, iseq, nt, edges, ao)
   static append_tp_faces(nodes, nts, iseq, nt, faces, ao)

class sfepy.discrete.fem.poly_spaces.LagrangePolySpace(name, geometry, order)
   create_context(cmesh, eps, check_errors, i_max, newton_eps, tdim=None)

class sfepy.discrete.fem.poly_spaces.LagrangeSimplexBPolySpace(name, geometry, order, init_context=True)
   Lagrange polynomial space with forced bubble function on a simplex domain.
```
create_context(*args, **kwargs)

name = 'lagrange_simplex_bubble'

class sfepy.discrete.fem.poly_spaces.LagrangeSimplexPolySpace(name, geometry, order, init_context=True)

Lagrange polynomial space on a simplex domain.

name = 'lagrange_simplex'

class sfepy.discrete.fem.poly_spaces.LagrangeTensorProductPolySpace(name, geometry, order, init_context=True)

Lagrange polynomial space on a tensor product domain.

get_mtx_i()

name = 'lagrange_tensor_product'

class sfepy.discrete.fem.poly_spaces.LobattoTensorProductPolySpace(name, geometry, order)

Hierarchical polynomial space using Lobatto functions.

Each row of the nodes attribute defines indices of Lobatto functions that need to be multiplied together to evaluate the corresponding shape function. This defines the ordering of basis functions on the reference element.

name = 'lobatto_tensor_product'

class sfepy.discrete.fem.poly_spaces.NodeDescription(node_types, nodes)

Describe FE nodes defined on different parts of a reference element.

has_extra_nodes()

Return True if the element has some edge, face or bubble nodes.

class sfepy.discrete.fem.poly_spaces.SEMTensorProductPolySpace(name, geometry, order, init_context=True)

Spectral element method polynomial space = Lagrange polynomial space with Legendre-Gauss-Lobatto nodes. The same nodes and corresponding weights should be used for numerical quadrature to obtain a diagonal mass matrix.

name = 'sem_tensor_product'

class sfepy.discrete.fem.poly_spaces.SerendipityTensorProductPolySpace(name, geometry, order)

Serendipity polynomial space using Lagrange functions.

Notes

- Orders >= 4 (with bubble functions) are not supported.
- Does not use CLagrangeContext, basis functions are hardcoded.
- self.nodes, self.node_coors are not used for basis evaluation and assembling.
2.3. Developer Guide

SfePy Documentation, Release version: 2023.4+git.4f968b9d
create_context(cmesh, eps, check_errors, i_max, newton_eps, tdim=None)

name = 'serendipity_tensor_product'
supported_orders = {1, 2, 3}

sfepy.discrete.fem.poly_spaces.eval_lagrange1d_basis(coors, ncoors)
sfepy.discrete.fem.poly_spaces.get_lgl_nodes(p)
    Compute the Legendre-Gauss-Lobatto nodes and weights.

sfepy.discrete.fem.refine module

Basic uniform mesh refinement functions.
sfepy.discrete.fem.refine.refine_1_2(mesh_in)
    Refines 1D mesh by cutting each element in half
sfepy.discrete.fem.refine.refine_2_3(mesh_in)
    Refines mesh out of triangles by cutting each edge in half and making 4 new finer triangles out of one coarser one.
sfepy.discrete.fem.refine.refine_2_4(mesh_in)
    Refines mesh out of quadrilaterals by cutting each edge in half and making 4 new finer quadrilaterals out of one coarser one.
sfepy.discrete.fem.refine.refine_3_4(mesh_in)
    Refines tetrahedra by cutting each edge in half and making 8 new finer tetrahedra out of one coarser one. Old nodal coordinates come first in coors, then the new ones. The new tetrahedra are similar to the old one, no degeneration is supposed to occur as at most 3 congruence classes of tetrahedra appear, even when re-applied iteratively (provided that conns are not modified between two applications - ordering of vertices in tetrahedra matters not only for positivity of volumes).

References:


sfepy.discrete.fem.refine.refine_3_8(mesh_in)
    Refines hexahedral mesh by cutting each edge in half and making 8 new finer hexahedrons out of one coarser one.
sfepy.discrete.fem.refine.refine_reference(geometry, level)
    Refine reference element given by geometry.
Notes

The error edges must be generated in the order of the connectivity of the previous (lower) level.

**sfepy.discrete.fem.refine_hanging module**

Functions for a mesh refinement with hanging nodes.

Notes

Using LCBCs with hanging nodes is not supported.

**sfepy.discrete.fem.refine_hanging.find_facet_substitutions**(*facets, cells, sub_cells, refine_facets*)

Find facet substitutions in connectivity.

\[
\text{sub} = [\text{coarse cell}, \text{coarse facet}, \text{fine1 cell}, \text{fine1 facet}, \text{fine2 cell}, \\
\text{fine2 facet}]
\]

**sfepy.discrete.fem.refine_hanging.find_level_interface**(*domain, refine_flag*)

Find facets of the coarse mesh that are on the coarse-refined cell boundary.

ids w.r.t. current mesh: - facets: global, local w.r.t. cells[:, 0], local w.r.t. cells[:, 1]

- interface cells: - cells[:, 0] - cells to refine - cells[:, 1] - their facet sharing neighbors (w.r.t. both meshes) -
  cells[:, 2] - facet kind: 0 = face, 1 = edge

**sfepy.discrete.fem.refine_hanging.refine**(*domain0, refine, subs=None, ret_sub_cells=False*)

**sfepy.discrete.fem.refine_hanging.refine_region**(*domain0, region0, region1*)

Coarse cell sub_cells[ii, 0] in mesh0 is split into sub_cells[ii, :] in mesh1.

The new fine cells are interleaved among the original coarse cells so that the indices of the coarse cells do not change.

The cell groups are preserved. The vertex groups are preserved only in the coarse (non-refined) cells.

**sfepy.discrete.fem._serendipity module**

**sfepy.discrete.fem.utils module**

**sfepy.discrete.fem.utils.compute_nodal_edge_dirs**(*nodes, region, field, return imap=False*)

Nodal edge directions are computed by simple averaging of direction vectors of edges a node is contained in. Edges are assumed to be straight and a node must be on a single edge (a border node) or shared by exactly two edges.

**sfepy.discrete.fem.utils.compute_nodal_normals**(*nodes, region, field, return imap=False*)

Nodal normals are computed by simple averaging of element normals of elements every node is contained in.

**sfepy.discrete.fem.utils.extend_cell_data**(*data, domain, rname, val=None, is_surface=False, average_surface=True*)

Extend cell data defined in a region to the whole domain.

Parameters

- **data**

  [array] The data defined in the region.
domain
[FEDomain instance] The FE domain.

rname
[str] The region name.

val
[float, optional] The value for filling cells not covered by the region. If not given, the smallest value in data is used.

is_surface
[bool] If True, the data are defined on a surface region. In that case the values are averaged or summed into the cells containing the region surface faces (a cell can have several faces of the surface), see \texttt{average\_surface}.

average_surface
[bool] If True, the data defined on a surface region are averaged, otherwise the data are summed.

Returns
edata
[array] The data extended to all domain elements.

sfepy.discrete.fem.utils.\texttt{get\_edge\_paths}(graph, mask)
Get all edge paths in a graph with non-masked vertices. The mask is updated.

sfepy.discrete.fem.utils.\texttt{get\_min\_value}(dofs)
Get a reasonable minimal value of DOFs suitable for extending over a whole domain.

sfepy.discrete.fem.utils.\texttt{invert\_remap}(remap)
Return the inverse of \texttt{remap}, i.e. a mapping from a sub-range indices to a full range, see \texttt{prepare\_remap()}.

sfepy.discrete.fem.utils.\texttt{prepare\_remap}(indices, n_full)
Prepare vector for remapping range $[0, n\_full]$ to its subset given by \texttt{indices}.

sfepy.discrete.fem.utils.\texttt{prepare\_translate}(old_indices, new_indices)
Prepare vector for translating \texttt{old\_indices} to \texttt{new\_indices}.

Returns
translate

sfepy.discrete.fem.utils.\texttt{refine\_mesh}(filename, level)
Uniformly refine \texttt{level}-times a mesh given by \texttt{filename}.

The refined mesh is saved to a file with name constructed from base name of \texttt{filename} and \texttt{level}-times appended \texttt{"_r"} suffix.

Parameters
filename
[str] The mesh file name.

level
[int] The refinement level.
sfepy.discrete.dg sub-package

sfepy.discrete.dg.dg_1D_vizualizer module

Module for animating solutions in 1D. Can also save them but requires ffmpeg package see save_animation method.

Animates solution of 1D problem into current figure. Keep reference to returned animation object otherwise it is discarded.

Parameters

- **Y**: solution, array |T| x |X| x n, where n is dimension of the solution
- **X**: space interval discretization
- **T**: time interval discretization
- **ax**: specify axes to plot to (Default value = None)
- **fig**: specify figure to plot to (Default value = None)
- **ylims**: limits for y axis, default are 10% offsets of Y extremes
- **labs**: labels to use for parts of the solution (Default value = None)
- **plott**: plot type - how to plot data: tested plot, step (Default value = None)
- **delay**: (Default value = None)

Returns

**anim**: the animation object, keep it to see the animation, used for savig too

Animates solution to 1D problem produced by DG:

1. animates DOF values in elements as steps
2. animates reconstructed solution with discontinuities

Parameters

- **coors**: coordinates of the mesh
- **t0**: [float] starting time

2.3. Developer Guide
\[ t1 \]
[\text{float}] final time

\[ u : \]
vectors of DOFs, for each order one, shape\((u) = (\text{order}, \text{nspace\_steps}, \text{ntime\_steps}, 1)\)

\[ \text{ic} : \]
analytical initial condition, optional (Default value = lambda x: 0.0)

\[ \text{tn} : \]
number of time steps to plot, starting at 0, if None and dt is not None run animation through all time steps, spaced dt within \([t0, tn]\) (Default value = None)

\[ \text{dt} : \]
time step size, if None and tn is not None computed as \((t1 - t0) / \text{tn}\) otherwise set to 1 if dt and tn are both None, t0 and t1 are ignored and solution is animated as if in time 0 \ldots \text{ntime\_steps} (Default value = None)

\[ \text{exact} : \]
(Default value = lambda x)

\[ t: 0 : \]
delay : (Default value = None)

\[ \text{polar} : \]
(Default value = False)

**Returns**

\[ \text{anim\_dofs} \]
[animation object of DOFs,]

\[ \text{anim\_recon} \]
[animation object of reconstructed solution]

\texttt{sfepy.discrete.dg.dg\_1D\_vizualizer.\texttt{head}(l)}

Maybe get head of the list.

**Parameters**

\[ l \]
[indexable]

**Returns**

\[ \text{head} \]
[first element in l or None is l is empty]

\texttt{sfepy.discrete.dg.dg\_1D\_vizualizer.\texttt{load\_1D\_vtks}(fold, name)}

Reads series of .vtk files and crunches them into form suitable for plot10\_DG\_sol.

Attempts to read modal cell data for variable mod\_data. i.e. ?\_modal\{i\}, where i is number of modal DOF

Resulting solution data have shape: \((\text{order}, \text{nspace\_steps}, \text{ntime\_steps}, 1)\)

**Parameters**

\[ \text{fold} : \]
folder where to look for files
name:
used in \{name\}.i.vtk, i = 0,1, ... tns - 1

Returns
coors
[ndarray]
mod_data
[ndarray] solution data

\texttt{sfepy.discrete.dg.dg_1D_vizualizer.load_state_1D_vtk(name)}
Load one VTK file containing state in time

Parameters
name
[\texttt{str}]

Returns
coops
[ndarray]
u
[ndarray]

\texttt{sfepy.discrete.dg.dg_1D_vizualizer.plot1D_legendre_dofs(coors, dofss, fun=None)}
Plots values of DOFs as steps

Parameters
coops:
coordinates of nodes of the mesh
dofss:
iterable of different projections’ DOFs into legendre space
fun:
analytical function to plot (Default value = None)

\texttt{sfepy.discrete.dg.dg_1D_vizualizer.plotsXT(Y1, Y2, YE, extent, lab1=None, lab2=None, lab3=None)}
Plots Y1 and Y2 to one axes and YE to the second axes, Y1 and Y2 are presumed to be two solutions and YE their error

Parameters
Y1:
solution 1, shape = (space nodes, time nodes)
Y2:
solution 2, shape = (space nodes, time nodes)
YE:
solution 1 - solution 2
extent:
imshow extent
lab1:
(\texttt{Default value = None})
lab2:
(\texttt{Default value = None})
lab3:
    (Default value = None)

sfepy.discrete.dg.dg_1D_vizualizer.reconstruct_legendre_dofs(coors, tn, u)

Creates solution and coordinates vector which when plotted as
plot(xx, ww)

represent solution reconstructed from DOFs in Legendre poly space at cell borders.

Works only as linear interpolation between cell boundary points

Parameters

coors:
    coors of nodes of the mesh

u:
    vectors of DOFs, for each order one, shape(u) = (order, nspace_steps, ntime_steps, 1)

tn:
    number of time steps to reconstruct, if None all steps are reconstructed

Returns

ww
    [ndarray] solution values vector, shape is (3 * nspace_steps - 1, ntime_steps, 1),

xx
    [ndarray] corresponding coordinates vector, shape is (3 * nspace_steps - 1, 1)

sfepy.discrete.dg.dg_1D_vizualizer.save_animation(anim, filename)

Saves animation as .mp4, requires ffmpeg package

Parameters

anim:
    animation object

filename:
    name of the file, without the .mp4 ending

sfepy.discrete.dg.dg_1D_vizualizer.save_sol_snap(Y, X, T, t0=0.5, filename=None, name=None, ylims=None, labs=None, plott=None)

Wrapper for sol_frame, saves the frame to file specified.

Parameters

name:
    name of the solution e.g. name of the solver used (Default value = None)

filename:
    name of the file, overrides automatic generation (Default value = None)

Y:
    solution, array [T] x [X] x n, where n is dimension of the solution

X:
    space interval discretization

T:
    time interval discretization

t0:
    time to take snap at (Default value = .5)
ylims :
limits for y axis, default are 10% offsets of Y extremes

labs :
labels to use for parts of the solution (Default value = None)

plott :
plot type - how to plot data: tested plot, step (Default value = None)

Returns
fig

sfepy.discrete.dg.dg_1D_vizualizer.setup_axis(X, Y, ax=None, fig=None, ylims=None)
Setup axis, including timer for animation or snaps

Parameters
X :
space discretization to get limits

Y :
solution to get limits

ax :
ax where to put everything, if None current axes are used (Default value = None)

fig :
fig where to put everything, if None current figure is used (Default value = None)

ylims :
custom ylims, if None y axis limits are calculated from Y (Default value = None)

Returns
ax
fig
time_text
object to fill in text

sfepy.discrete.dg.dg_1D_vizualizer.setup_lines(ax, Yshape, labs, plott)
Sets up artist for animation or solution snaps

Parameters
ax :
axes to use for artist

Yshape
[tuple] shape of the solution array

labs
[list] labels for the solution

plott
[str (“steps” or “plot”)] type of plot to use

Returns
lines

sfepy.discrete.dg.dg_1D_vizualizer.sol_frame(Y, X, T, t0=0.5, ax=None, fig=None, ylims=None, labs=None, plott=None)
Creates snap of solution at specified time frame $t_0$, basically gets one frame from animate1D_dgsol, but colors won't be the same :-(

**Parameters**

- **Y**: solution, array $|T| \times |X| \times n$, where $n$ is dimension of the solution
- **X**: space interval discretization
- **T**: time interval discretization
- **$t_0$**: time to take snap at (Default value = .5)
- **ax**: specify axes to plot to (Default value = None)
- **fig**: specify figure to plot to (Default value = None)
- **ylims**: limits for y axis, default are 10% offsets of Y extremes
- **labs**: labels to use for parts of the solution (Default value = None)
- **plott**: plot type - how to plot data: tested plot, step (Default value = None)

**Returns**

- **fig**

### sfepy.discrete.dg.fields module

Fields for Discontinuous Galerkin method

**class** sfepy.discrete.dg.fields.DGField**

$(name, dtype, shape, region, space='H1',
 poly_space_base='legendre', approx_order=1, integral=None)$

Discontinuous Galerkin method approximation with Legendre basis.

Class for usage with DG terms, provides functionality for Discontinuous Galerkin method like neighbour look up, projection to discontinuous basis and correct DOF treatment.

**clear_facet_neighbour_idx_cache**(region=None)

If region is None clear all!

**Parameters**

- **region**:
  [sfepy.discrete.common.region.Region] If None clear all.

**clear_facet_qp_base**

Clears facet_qp_base cache
clear_facet_vols_cache(region=None)

Clears facet volume cache for given region or all regions.

Parameters

region
[sfepy.discrete.common.region.Region] region to clear cache or None to clear all

clear_normals_cache(region=None)

Clears normals cache for given region or all regions.

Parameters

region
[sfepy.discrete.common.region.Region] region to clear cache or None to clear all

create_mapping(region, integral, integration, return_mapping=True)

Creates and returns mapping

Parameters

region
[sfepy.discrete.common.region.Region]

integral
[Integral]

integration
[str] ‘volume’ is only accepted option

return_mapping
[default True] (Default value = True)

Returns

mapping
[FEMapping]

create_output(dofs, var_name, dof_names=None, key=None, extend=True, fill_value=None, linearization=None)

Converts the DOFs corresponding to the field to a dictionary of output data usable by Mesh.write().

For 1D puts DOFs into vairables u_modal[0] ... u_modal[n], where n = approx_order and marks them for writing as cell data.

For 2+D puts dofs into name_cell_nodes and creates sturct with: mode = “cell_nodes”, data and interpolation scheme.

Also get node values and adds them to dictionary as cell_nodes

Parameters

dofs
[ndarray, shape (n_nod, n_component)] The array of DOFs reshaped so that each column corresponds to one component.

var_name
[str] The variable name corresponding to dofs.

dof_names
[tuple of str] The names of DOF components. (Default value = None)
key
  [str, optional] The key to be used in the output dictionary instead of the variable name. (Default value = None)

extend
  [bool, not used] Extend the DOF values to cover the whole domain. (Default value = True)

fill_value
  [float or complex, not used] The value used to fill the missing DOF values if extend is True. (Default value = None)

linearization
  [Struct or None, not used] The linearization configuration for higher order approximations. (Default value = None)

Returns

out
  [dict]

family_name = 'volume_DG_legendre_discontinuous'

get_bc_facet_idx(region)
  Caches results in self.boundary_facet_local_idx

Parameters

region
  [sfepy.discrete.common.region.Region] surface region defining BCs

Returns

bc2bfi
  [ndarray] index of cells on boundary along with corresponding facets

get_bc_facet_values(fun, region, ret_coors=False, diff=0)
  Returns values of fun in facet QPs of the region

Parameters

diff: derivative 0 or 1 supported
fun: Function value or values to set qps values to
region
  [sfepy.discrete.common.region.Region] boundary region

ret_coors: default False,
  Return physical coors of qps in shape (n_cell, n_qp, dim).

Returns

vals
  [ndarray] In shape (n_cell,) + (self.dim,) * diff + (n_qp,)

get_both_facet_base_vals(state, region, derivative=None)
  Returns values of the basis function in quadrature points on facets broadcasted to all cells inner to the
element as well as outer ones along with weights for the qps broadcasted and transformed to elements.
  Contains quick fix to flip facet QPs for right integration order.

Parameters

state
  [used to get EPBC info]
region
    [sfepy.discrete.common.region.Region for connectivity]

derivative
    [if u need derivative] (Default value = None)

Returns

outer_facet_base_vals:
inner_facet_base_vals:

shape (n_cell, n_el_nod, n_el_facet, n_qp) or
    (n_cell, n_el_nod, n_el_facet, dim, n_qp)

when derivative is True or 1

whs: shape (n_cell, n_el_facet, n_qp)

get_both_facet_state_vals(state, region, derivative=None, reduce_nod=True)

Computes values of the variable represented by dofs in quadrature points located at facets, returns both values - inner and outer, along with weights.

Parameters

state
    [state variable containing BC info]

region
    [sfepy.discrete.common.region.Region]

derivative
    [compute derivative if truthy,] compute n-th derivative if a number (Default value = None)

reduce_nod
    [if False DOES NOT sum nodes into values at QPs] (Default value = True)

Returns

inner_facet_values (n_cell, n_el_facets, n_qp),
outer_facet_values (n_cell, n_el_facets, n_qp), weights, if derivative is True:
    inner_facet_values (n_cell, n_el_facets, dim, n_qp), outer_facet_values (n_cell, n_el_facets, dim, n_qp)

get_cell_normals_per_facet(region)

Caches results, use clear_normals_cache to clear the cache.

Parameters

region: sfepy.discrete.common.region.Region
    Main region, must contain cells.

Returns

normals: ndarray
    normals of facets in array of shape (n_cell, n_el_facets, dim)

get_coors(nods=None)

Returns coors for matching nodes # TODO revise DG_EPBC and EPBC matching?

Parameters

nods :
    if None use all nodes (Default value = None)
Returns

coors
[ndarray] coors on surface

get_data_shape(integral, integration='cell', region_name=None)
Returns data shape (n_nod, n_qp, self.gel.dim, self.n_el_nod)

Parameters

integral
[integral used]

integration:
‘volume’ is only supported value (Default value = ‘volume’)

region_name
[not used] (Default value = None)

Returns

data_shape
[tuple]

get_dofs_in_region(region, merge=True)
Return indices of DOFs that belong to the given region.
Not Used in BC treatment

Parameters

region
[sfepy.discrete.common.region.Region]

merge
[bool] merge dof tuple into one numpy array, default True

Returns

dofs
[ndarray]

get_econn(conn_type, region, trace_region=None)
Getter for econn

Parameters

conn_type
[tuple or string] (‘cell’, dim) or ‘cell’ is only supported

region
[sfepy.discrete.common.region.Region]

trace_region
[ignored] (Default value = None)

Returns

econn
[ndarray] connectivity information

get_facet_base(derivative=False, base_only=False)
Returns values of base in facets quadrature points, data shape is a bit crazy right now:

(number of qps, 1, n_el_facets, 1, n_el_nod)
end for derivative:
(1, number of qps, (dim,) * derivative, n_el_facets, 1, n_el_nod)

Parameters

derivative: truthy or integer
base_only: do not return weights

Returns

facet_bf
[ndarray] values of basis functions in facet qps

weights
[ndarray, optionally] weights of qps

get_facet_neighbor_idx(region=None, eq_map=None)

Returns index of cell neighbours sharing facet, along with local index of the facet within neighbour, also treats periodic boundary conditions i.e. plugs correct neighbours for cell on periodic boundary. Where there are no neighbours specified puts -1 instead of neighbour and facet id

Cashes neighbour index in self.facet_neighbours

Parameters

region
[sfepy.discrete.common.region.Region] Main region, must contain cells.

eq_map :
eq_map from state variable containing information on EPBC and DG EPBC. (Default value = None)

Returns

facet_neighbours
[ndarray]

Shape is
(n_cell, n_el_facet, 2),
first value is index of the neighbouring cell, the second is index of the facet in said nb. cell.

get_facet_qp()

Returns quadrature points on all facets of the reference element in array of shape (n_qp, 1, n_el_facets, dim)

Returns

qps
[ndarray] quadrature points

weights
[ndarray] Still needs to be transformed to actual facets!

get_facet_vols(region)

Caches results, use clear_facet_vols_cache to clear the cache

Parameters

region
[sfepy.discrete.common.region.Region]

Returns
vols_out: ndarray
volumes of the facets by cells shape (n_cell, n_el_facets, 1)

get_nodal_values(dofs, region, ref_nodes=None)
Computes nodal representation of the DOFs

Parameters

dofs
[array_like] dofs to transform to nodes

region : ignored

ref_nodes:
reference node to use instead of default qps

Parameters

dofs
[array_like]

region
[Region]

ref_nodes
[array_like] (Default value = None)

Returns

nodes
[ndarray]

nodal_vals
[ndarray]

static get_region_info(region)
Extracts information about region needed in various methods of DGField

Parameters

region
[sfepy.discrete.common.region.Region]

Returns

dim, n_cell, n_el_facets

is_surface = False

set_cell_dofs(fun=0.0, region=None, dpn=None, warn=None)
Compute projection of fun onto the basis, in main region, alternatively set DOFs directly to provided value or values

Parameters

fun
[callable, scalar or array corresponding to dofs] (Default value = 0.0)

region
[sfepy.discrete.common.region.Region] region to set DOFs on (Default value = None)
set_dofs(fun=0.0, region=None, dpn=None, warn=None)
Compute projection of fun into the basis, alternatively set DOFs directly to provided value or values either
in main volume region or in boundary region.

Parameters
fun
[callable, scalar or array corresponding to dofs] (Default value = 0.0)
region
[sfePy.discrete.common.region.Region] region to set DOFs on (Default value = None)
dpn
[number of dofs per element] (Default value = None)
warn : (Default value = None)

Returns
nods
[ndarray]
vals
[ndarray]

set_facet_dofs(fun, region, dpn, warn)
Compute projection of fun onto the basis on facets, alternatively set DOFs directly to provided value or values

Parameters
fun
[callable, scalar or array corresponding to dofs]
region
[sfePy.discrete.common.region.Region] region to set DOFs on
dpn
[int] number of dofs per element
warn : not used

Returns
nods
[ndarray]
vals
[ndarray]

setup_extra_data(info)

This is called in create_adof_conns(conn_info, var_indx=None, active_only=True, verbose=True)
for each variable but has no effect.

Parameters

info :
set to self.info

sfepy.discrete.dg.fields.get_gel(region)

Parameters

region
[sfepy.discrete.common.region.Region]

Returns

gel :
base geometry element of the region

sfepy.discrete.dg.fields.get_raveler(n_el_nod, n_cell)

Returns function for raveling i.e. packing dof data from two dimensional array of shape (n_cell, n_el_nod, 1) to (n_el_nod*n_cell, 1)
The raveler returns view into the input array.

Parameters

n_el_nod :
param n_el_nod, n_cell: expected dimensions of dofs array

n_cell
[int]

Returns

ravel
[callable]

sfepy.discrete.dg.fields.get_unraveler(n_el_nod, n_cell)

Returns function for unraveling i.e. unpacking dof data from serialized array from shape (n_el_nod*n_cell, 1) to (n_cell, n_el_nod, 1).
The unraveler returns non-writeable view into the input array.

Parameters

n_el_nod
[int] expected dimensions of dofs array

n_cell
[int]

Returns

unravel
[callable]
sfepy.discrete.dg.poly_spaces module

class sfepy.discrete.dg.poly_spaces.LegendrePolySpace(name, geometry, order, extended)
  Legendre hierarchical polynomials basis, over [0, 1] domain.
  
  get_interpol_scheme()
  For dim > 1 returns F and P matrices according to gmsh basis specification [1]: Let us assume that
  the approximation of the view’s value over an element is written as a linear combination of d basis
  functions \( f_i, i = 0, \ldots, n - 1 \) (the coefficients being stored in list-of-values).
  
  Defining
  \[
  f_i = \sum_{j=0}^{d-1} F_{ij} \cdot p_j,
  \]
  
  with
  \[
  p_j(u, v, w) = u^{P_j(0)} \cdot v^{P_j(1)} \cdot w^{P_j(2)}
  \]
  (u, v and w being the coordinates in the element’s parameter space),
  then val-coef-matrix denotes the n x n matrix F and val-exp-matrix denotes the n x 3 matrix P
  where n is number of basis functions as calculated by get_n_el_nod.
  
  Expects matrices to be saved in attributes coefM and expoM!
  
  Returns

  interp_scheme_struct
  [Struct] Struct with name of the scheme, geometry desc and P and F

  get_nth_fun(n)
  Uses shifted Legendre polynomials formula on interval [0, 1].
  
  Convenience function for testing
  
  Parameters

  n
  [int]
  
  Returns

  fun
  [callable] n-th function of the legendre basis

  get_nth_fun_der(n, diff=1)
  Returns diff derivative of nth function. Uses shifted legendre polynomials formula on interval [0, 1].
  
  Useful for testing.
  
  Parameters

  n
  [int]
  
  diff
  [int] (Default value = 1)
  
  Returns

  fun
  [callable] derivative of n-th function of the 1D legendre basis
\texttt{gradjacobiP(coors, alpha, beta, diff=1)}

diff derivative of the jacobi polynomials on interval [-1, 1] up to self.order + 1 at coors

\textbf{Parameters}

coors :  
alpha
[\texttt{float}]

\textbf{beta}
[\texttt{float}]

diff  
[int] (Default value = 1)

\textbf{Returns}

values  
[\texttt{ndarray}] output shape is shape(coor) + (self.order + 1,)

\texttt{gradlegendreP(coors, diff=1)}

Parameters

diff  
[int] default 1

coops  
[array_like] coordinates, preferably in interval [-1, 1] for which this basis is intended

\textbf{Returns}

values  
[\texttt{ndarray}] values at coors of all the legendre polynomials up to self.order

\texttt{jacobiP(coors, alpha, beta)}

Values of the jacobi polynomials on interval [-1, 1] up to self.order + 1 at coors

\textbf{Parameters}

coors  
[array_like]

\textbf{beta}
[\texttt{float}]

\textbf{alpha}
[\texttt{float}]

\textbf{Returns}

values  
[\texttt{ndarray}] output shape is shape(coor) + (self.order + 1,)

\texttt{legendreP(coors)}

\textbf{Parameters}

coops  
[array_like] coordinates, preferably in interval [-1, 1] for which this basis is intended

\textbf{Returns}

values  
[\texttt{ndarray}] values at coors of all the legendre polynomials up to self.order
legendre_funs = [\texttt{<function LegendrePolySpace.<lambda>>}, \texttt{<function LegendrePolySpace.<lambda>>}, \texttt{<function LegendrePolySpace.<lambda>>}, \texttt{<function LegendrePolySpace.<lambda>>}, \texttt{<function LegendrePolySpace.<lambda>>}]

\texttt{class sfepy.discrete.dg.poly_spaces.LegendreSimplexPolySpace(name, geometry, order, extended=False)}

\texttt{name = 'legendre_simplex'}

\texttt{class sfepy.discrete.dg.poly_spaces.LegendreTensorProductPolySpace(name, geometry, order)}

\texttt{build_interpol_scheme()}

Builds F and P matrices returned by self.get_interpol_scheme.

Note that this function returns coefficients according to gmsh parametrization of Quadrangle i.e. \([-1, 1] \times [-1, 1]\) and hence the form of basis function is not the same as exhibited by the LegendreTensorProductPolySpace object which acts on parametrization \([0, 1] \times [0, 1]\).

\textbf{Returns}

- \texttt{F} [ndarray] coefficient matrix
- \texttt{P} [ndarray] exponent matrix

\texttt{name = 'legendre_tensor_product'}

\texttt{sfepy.discrete.dg.poly_spaces.get_n_el_nod(order, dim, extended=False)}

Number of nodes per element for discontinuous legendre basis, i.e. number of iterations yielded by \texttt{iter_by_order}.

When extended is False

\[ N_p = \frac{(n + 1) \cdot (n + 2) \cdot \ldots \cdot (n + d)}{d!} \]

where \( n \) is the order and \( d \) the dimension. When extended is True

\[ N_p = (n + 1)^d \]

where \( n \) is the order and \( d \) the dimension.

\textbf{Parameters}

- \texttt{order} [int] desired order of multidimensional basis
- \texttt{dim} [int] dimension of the basis
- \texttt{extended} [bool] iterate over extended tensor product basis (Default value = False)

\textbf{Returns}

- \texttt{n_el_nod} [int] number of basis functions in basis
Iterates over all combinations of basis functions indexes needed to create multidimensional basis in a way that creates hierarchical basis

**Parameters**
- **order**
  - [int] desired order of multidimensional basis
- **dim**
  - [int] dimension of the basis
- **extended**
  - [bool] iterate over extended tensor product basis (Default value = False)

**Yields**
- **idx**
  - [tuple] containing basis function indexes, used in `_combine_polyvals` and `_combine_polyvals_der`

### sfepy.discrete.dg.limiters module

Limiters for high order DG methods

**class** `sfepy.discrete.dg.limiters.ComposedLimiter(fields, limiters, verbose=False)`

**class** `sfepy.discrete.dg.limiters.DGLimiter(field, verbose=False)`

- **name = 'abstract DG limiter'**

**class** `sfepy.discrete.dg.limiters.IdentityLimiter(field, verbose=False)`

Neutral limiter returning unchanged solution.

- **name = 'identity'**

**class** `sfepy.discrete.dg.limiters.MomentLimiter1D(field, verbose=False)`

Moment limiter for 1D based on [1]

- **name = 'moment_1D_limiter'**

**class** `sfepy.discrete.dg.limiters.MomentLimiter2D(field, verbose=False)`


- **name = 'moment_limiter_2D'**

**sfepy.discrete.dg.limiters.minmod(a, b, c)**

Minmod function of three variables, returns:

- 0, where `sign(a) != sign(b) != sign(c)`
- `min(a,b,c)`, elsewhere

**Parameters**
- **a**
  - [array_like]
- **c**
  - [array_like]
b
  [array_like]

Returns
  out
  [ndarray]

```
sfepy.discrete.dg.limiters.minmod_seq(abc)
```

Minmod function of n variables, returns:
0, where \(\text{sign}(a_1) \neq \text{sign}(a_2) \neq ... \neq \text{sign}(a_n)\)
min\((a_1, a_2, a_3, \ldots, a_n)\), elsewhere

Parameters
  abc
  [sequence of array_like]

Returns
  out
  [ndarray]

```
sfepy.solvers.ts_dg_solvers module
```

Explicit time stepping solvers for use with DG FEM

```
class sfepy.solvers.ts_dg_solvers.DGMultiStageTSS(conf, nls=None, context=None, **kwargs)
```

Explicit time stepping solver with multistage solve_step method

Kind: ‘ts.multistaged’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters
  t0
  [float (default: 0.0)] The initial time.

  t1
  [float (default: 1.0)] The final time.

  dt
  [float] The time step. Used if n_step is not given.

  n_step
  [int (default: 10)] The number of time steps. Has precedence over dt.

  quasistatic
  [bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver is invoked also for the initial time.

  limiters
  [dictionary] Limiters for DGFields, keys: field name, values: limiter class

  name = 'ts.multistaged'

  output_step_info(ts)
solve_step(ts, nls, vec, prestep_fun=None, poststep_fun=None, status=None)
solve_step0(nls, vec0)

class sfepy.solvers.ts_dg_solvers.EulerStepSolver(conf, nls=None, context=None, **kwargs)
Simple forward euler method
Kind: ‘ts.euler’
For common configuration parameters, see Solver.
Specific configuration parameters:
name = 'ts.euler'
solve_step(ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None)

class sfepy.solvers.ts_dg_solvers.RK4StepSolver(conf, nls=None, context=None, **kwargs)
Classical 4th order Runge-Kutta method, implementations is based on [1]
Kind: ‘ts.runge_kutta_4’
For common configuration parameters, see Solver.
Specific configuration parameters:
name = 'ts.runge_kutta_4'
solve_step(ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None)

stage_updates = (<function RK4StepSolver.<lambda>>, <function RK4StepSolver.<lambda>>, <function RK4StepSolver.<lambda>>, <function RK4StepSolver.<lambda>>)

class sfepy.solvers.ts_dg_solvers.TVDRK3StepSolver(conf, nls=None, context=None, **kwargs)
3rd order Total Variation Diminishing Runge-Kutta method based on [1]

\[
\begin{align*}
  p^{(1)} &= p^n - \Delta t \bar{L}(p^n), \\
  p^{(2)} &= \frac{3}{4} p^n + \frac{1}{4} p^{(1)} - \frac{1}{4} \Delta t \bar{L}(p^{(1)}), \\
  p^{(n+1)} &= \frac{1}{3} p^n + \frac{2}{3} p^{(2)} - \frac{2}{3} \Delta t \bar{L}(p^{(2)}).
\end{align*}
\]
Kind: ‘ts.tvd_runge_kutta_3’
For common configuration parameters, see Solver.
Specific configuration parameters:
name = 'ts.tvd_runge_kutta_3'
solve_step(ts, nls, vec_x0, status=None, prestep_fun=None, poststep_fun=None)
sfepy.discrete.iga sub-package

sfepy.discrete.iga.domain module

Computational domain for isogeometric analysis.

class sfepy.discrete.iga.domain.IGDomain(name, nurbs, bmesh, regions=None, **kwargs)
    Bézier extraction based NURBS domain for isogeometric analysis.

    static from_data(knots, degrees, cps, weights, cs, conn, bcps, bweights, bconn, regions,
                      name='iga_domain_from_data')
    Create the IGA domain from the given data.

    static from_file(filename)
        filename [str] The name of the IGA domain file.

    static read_domain_from_hdf5(fd, group)
        Create a domain from the given hdf5 data group.

        fd: tables.File
            HDF5 file handle to read the mesh from.

        group: tables.group.Group
            HDF5 data group (of file fd) to read the mesh from.

    write_domain_to_hdf5(fd, group)
        Save the domain to a hdf5 file.

        fd: tables.File
            HDF5 file handle to write the mesh to.

        group: tables.group.Group
            HDF5 data group (of file fd) to write the mesh to.

class sfepy.discrete.iga.domain.NurbsPatch(knots, degrees, cps, weights, cs, conn)
    Single NURBS patch data.

elevate(times=0)
    Elevate the patch degrees several times by one.

    Returns

    nurbs [NurbsPatch instance] Either self if times is zero, or a new instance.

    evaluate(field, u=None, v=None, w=None)
        Igakit-like interface for NURBS evaluation.
sfepy.discrete.iga.domain_generators module

IGA domain generators.

sfepy.discrete.iga.domain_generators.create_from_igakit(inurbs, verbose=False)

Create IGDomain data from a given igakit NURBS object.

Parameters

- inurbs
  [igakit.nurbs.NURBS instance] The igakit NURBS object.

Returns

- nurbs
  [NurbsPatch instance] The NURBS data. The igakit NURBS object is stored as nurbs attribute.

- bmesh
  [Struct instance] The Bezier mesh data.

- regions
  [dict] The patch surface regions.

sfepy.discrete.iga.domain_generators.gen_patch_block_domain(dims, shape, centre, degrees, continuity=None, cp_mode='greville', name='block', verbose=True)

Generate a single IGA patch block in 2D or 3D of given degrees and continuity using igakit.

Parameters

- dims

- shape
  [array of D ints] Numbers of unique knot values along each axis.

- centre
  [array of D floats] Centre of the block.

- degrees
  [array of D floats] NURBS degrees along each axis.

- continuity
  [array of D ints, optional] NURBS continuity along each axis. If None, degrees-1 is used.

- cp_mode
  ['greville' or 'uniform'] The control points mode. The default ‘greville’ results in a uniform Bezier mesh, while the ‘uniform’ mode results in a uniform grid of control points a finer Bezier mesh inside the block and a coarser Bezier mesh near the block boundary.

- name
  [string] Domain name.

- verbose
  [bool] If True, report progress of the domain generation.

Returns

- nurbs
  [NurbsPatch instance] The NURBS data. The igakit NURBS object is stored as nurbs attribute.
bmesh
    [Struct instance] The Bezier mesh data.

regions
    [dict] The patch surface regions.

sfepy.discrete.iga.extmods.igac module

class sfepy.discrete.iga.extmods.igac.CNURBSContext
    R
    bf
    bfg
    bufBN
    cprint()
    dR_dx
    dR_dxi
    e_coors_max
    evaluate()
    iel

sfepy.discrete.iga.extmods.igac.eval_bernstein_basis()

sfepy.discrete.iga.extmods.igac.eval_in_tp_coors()
    Evaluate a field variable (if given) or the NURBS geometry in the given tensor-product reference coordinates.
The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters

variable
    [array] The DOF values of the variable with n_c components, shape (, n_c).

indices
    [list of arrays] The indices of knot spans for each axis, defining the Bezier element numbers.

ref_coors
    [list of arrays] The reference coordinates in [0, 1] for each knot span for each axis, defining the reference coordinates in the Bezier elements given by indices.

control_points
    [array] The NURBS control points.

weights
    [array] The NURBS weights.

degrees
    [sequence of ints or int] The basis degrees in each parametric dimension.

cs
    [list of lists of 2D arrays] The element extraction operators in each parametric dimension.
The connectivity of the global NURBS basis.

Returns

out
array] The field variable values or NURBS geometry coordinates for the given reference coordinates.

sfepy.discrete.iga.extmods.igac.eval_mapping_data_in_qp()

Evaluate data required for the isogeometric domain reference mapping in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree.

Parameters

qps
array] The quadrature points coordinates with components in [0, 1] reference element domain.

cells
array, optional] If given, use only the given Bezier elements.

Returns

bfs
array] The NURBS shape functions in the physical quadrature points of all elements.

bfgs
array] The NURBS shape functions derivatives w.r.t. the physical coordinates in the physical quadrature points of all elements.

dets
array] The Jacobians of the mapping to the unit reference element in the physical quadrature points of all elements.

sfepy.discrete.iga.extmods.igac.eval_variable_in_qp()

Evaluate a field variable in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree. The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters

variable
array] The DOF values of the variable with n_c components, shape (; n_c).

qps
array] The quadrature points coordinates with components in [0, 1] reference element domain.
control_points
[array] The NURBS control points.

weights
[array] The NURBS weights.

degrees
[sequence of ints or int] The basis degrees in each parametric dimension.

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
[array] The connectivity of the global NURBS basis.

cells
[array, optional] If given, use only the given Bezier elements.

Returns

coors
[array] The physical coordinates of the quadrature points of all elements.

vals
[array] The field variable values in the physical quadrature points.

dets
[array] The Jacobians of the mapping to the unit reference element in the physical quadrature points.

sfepy.discrete.iga.extmods.igac.is_nurbs()
Return True if some weights are not one.

sfepy.discrete.iga.fields module

Fields for isogeometric analysis.

class sfepy.discrete.iga.fields.IGField(name, dtype, shape, region, approx_order=None, **kwargs)
Beziers extraction based NURBS approximation for isogeometric analysis.

Notes

The field has to cover the whole IGA domain. The field’s NURBS basis can have higher degree than the domain NURBS basis.

create_basis_context()
Create the context required for evaluating the field basis.

create_eval_mesh()
Create a mesh with the original NURBS connectivity for evaluating the field. The mesh coordinates are the NURBS control points.

create_mapping(region, integral, integration)
Create a new reference mapping.

create_mesh(extra_nodes=True)
Create a mesh corresponding to the field region. For IGA fields, this is directly the topological mesh. The extra_nodes argument is ignored.
create_output(dofs, var_name, dof_names=None, key=None, **kwargs)

Convert the DOFs corresponding to the field to a dictionary of output data usable by Mesh.write().

Parameters

dofs
[array, shape (n_nod, n_component)] The array of DOFs reshaped so that each column
  corresponds to one component.

var_name
[str] The variable name corresponding to dofs.

dof_names
[tuple of str] The names of DOF components.

key
[str, optional] The key to be used in the output dictionary instead of the variable name.

Returns

out

family_name = 'volume_H1_iga'

get_data_shape(integral, integration='cell', region_name=None)

Get element data dimensions.

Parameters

integral
[Integral instance] The integral describing used numerical quadrature.

integration
['cell'] The term integration type. Only 'cell' type is implemented.

region_name
[str] The name of the region of the integral.

Returns

data_shape
[4 ints] The (n_el, n_qp, dim, n_en) for volume shape kind.

Notes

* n_el = number of elements
* n_qp = number of quadrature points per element/facet
* dim = spatial dimension
* n_en = number of element nodes

get_dofs_in_region(region, merge=True)

Return indices of DOFs that belong to the given region and group.
Notes

merge is not used.

get_econn(conn_type, region, trace_region=None, local=False)
Get DOF connectivity of the given type in the given region.

get_surface_basis(region)

get_true_order()

is_higher_order()
Return True, if the field’s approximation order is greater than one.

setup_extra_data(info)

sfepy.discrete.iga.fields.parse_approx_order(approx_order)

sfepy.discrete.iga.iga module

Isogeometric analysis utilities.

Notes

The functions compute_bezier_extraction_1d() and eval_nurbs_basis_tp() implement the algorithms described in [1].

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

sfepy.discrete.iga.iga.combine_bezier_extraction(cs)
For a nD B-spline parametric domain, combine the 1D element extraction operators in each parametric dimension into a single operator for each nD element.

Parameters

 cs [list of lists of 2D arrays] The element extraction operators in each parametric dimension.

Returns

 ccs [list of 2D arrays] The combined element extraction operators.

sfepy.discrete.iga.iga.compute_bezier_control(control_points, weights, ccs, conn, bconn)
Compute the control points and weights of the Bezier mesh.

Parameters

 control_points [array] The NURBS control points.

 weights [array] The NURBS weights.

 ccs [list of 2D arrays] The combined element extraction operators.
conn
[array] The connectivity of the global NURBS basis.

bconn
[array] The connectivity of the Bezier basis.

Returns

bezier_control_points
[array] The control points of the Bezier mesh.

bezier_weights
[array] The weights of the Bezier mesh.

sfepy.discrete.iga.iga.compute_bezier_extraction(knots, degrees)
Compute local (element) Bezier extraction operators for a nD B-spline parametric domain.

Parameters

knots
[sequence of array or array] The knot vectors.

degrees
[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

cs
[list of lists of 2D arrays] The element extraction operators in each parametric dimension.

sfepy.discrete.iga.iga.compute_bezier_extraction_1d(knots, degree)
Compute local (element) Bezier extraction operators for a 1D B-spline parametric domain.

Parameters

knots
[array] The knot vector.

degree
[int] The curve degree.

Returns

cs
[array of 2D arrays (3D array)] The element extraction operators.

sfepy.discrete.iga.iga.create_boundary_qp(coors, dim)
Create boundary quadrature points from the surface quadrature points.
Uses the Bezier element tensor product structure.

Parameters

coors
[array, shape (n_qp, d)] The coordinates of the surface quadrature points.

dim
[int] The topological dimension.

Returns

bcoors
[array, shape (n_qp, d + 1)] The coordinates of the boundary quadrature points.
Create connectivity arrays of nD Bezier elements.

**Parameters**

- `n_els`:
  - [sequence of ints] The number of elements in each parametric dimension.

- `knots`:
  - [sequence of array or array] The knot vectors.

- `degrees`:
  - [sequence of ints or int] The basis degrees in each parametric dimension.

**Returns**

- `conn`:
  - [array] The connectivity of the global NURBS basis.

- `bconn`:
  - [array] The connectivity of the Bezier basis.

Create connectivity arrays of 1D Bezier elements.

**Parameters**

- `n_el`:
  - [int] The number of elements.

- `knots`:
  - [array] The knot vector.

- `degree`:
  - [int] The basis degree.

**Returns**

- `conn`:
  - [array] The connectivity of the global NURBS basis.

- `bconn`:
  - [array] The connectivity of the Bezier basis.

Evaluate the Bernstein polynomial basis of the given `degree`, and its derivatives, in a point `x` in [0, 1].

**Parameters**

- `x`:
  - [float] The point in [0, 1].

- `degree`:
  - [int] The basis degree.

**Returns**

- `funs`:
  - [array] The `degree + 1` values of the Bernstein polynomial basis.

- `ders`:
  - [array] The `degree + 1` values of the Bernstein polynomial basis derivatives.
sfepy.discrete.iga.iga.eval_mapping_data_in_qp(qps, control_points, weights, degrees, cs, conn, cells=None)

Evaluate data required for the isogeometric domain reference mapping in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree.

**Parameters**

- **qps**
  - [array] The quadrature points coordinates with components in [0, 1] reference element domain.

- **control_points**
  - [array] The NURBS control points.

- **weights**
  - [array] The NURBS weights.

- **degrees**
  - [sequence of ints or int] The basis degrees in each parametric dimension.

- **cs**
  - [list of lists of 2D arrays] The element extraction operators in each parametric dimension.

- **conn**
  - [array] The connectivity of the global NURBS basis.

- **cells**
  - [array, optional] If given, use only the given Bezier elements.

**Returns**

- **bfs**
  - [array] The NURBS shape functions in the physical quadrature points of all elements.

- **bfgs**
  - [array] The NURBS shape functions derivatives w.r.t. the physical coordinates in the physical quadrature points of all elements.

- **dets**
  - [array] The Jacobians of the mapping to the unit reference element in the physical quadrature points of all elements.

sfepy.discrete.iga.iga.eval_nurbs_basis_tp(qp, ie, control_points, weights, degrees, cs, conn)

Evaluate the tensor-product NURBS shape functions in a quadrature point for a given Bezier element.

**Parameters**

- **qp**
  - [array] The quadrature point coordinates with components in [0, 1] reference element domain.

- **ie**
  - [int] The Bezier element index.

- **control_points**
  - [array] The NURBS control points.

- **weights**
  - [array] The NURBS weights.

- **degrees**
  - [sequence of ints or int] The basis degrees in each parametric dimension.
cs
  [list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
  [array] The connectivity of the global NURBS basis.

Returns

R
  [array] The NURBS shape functions.

dR_dx
  [array] The NURBS shape functions derivatives w.r.t. the physical coordinates.

det
  [array] The Jacobian of the mapping to the unit reference element.

sfepy.discrete.iga.iga.eval_variable_in_qp(variable, qps, control_points, weights, degrees, cs, conn, cells=None)

Evaluate a field variable in the given quadrature points. The quadrature points are the same for all Bezier elements and should correspond to the Bernstein basis degree. The field variable is defined by its DOFs - the coefficients of the NURBS basis.

Parameters

variable
  [array] The DOF values of the variable with n_c components, shape (, n_c).

qps
  [array] The quadrature points coordinates with components in [0, 1] reference element domain.

control_points
  [array] The NURBS control points.

weights
  [array] The NURBS weights.

degrees
  [sequence of ints or int] The basis degrees in each parametric dimension.

cs
  [list of lists of 2D arrays] The element extraction operators in each parametric dimension.

conn
  [array] The connectivity of the global NURBS basis.

cells
  [array, optional] If given, use only the given Bezier elements.

Returns

coors
  [array] The physical coordinates of the quadrature points of all elements.

vals
  [array] The field variable values in the physical quadrature points.

dets
  [array] The Jacobians of the mapping to the unit reference element in the physical quadrature points.
**sfepy.discrete.iga.iga.get_bezier_element_entities**(*degrees*)

Get faces and edges of a Bezier mesh element in terms of indices into the element’s connectivity (reference Bezier element entities).

**Parameters**

- **degrees**
  
  [sequence of ints or int] Polynomial degrees in each parametric dimension.

**Returns**

- **faces**
  
  [list of arrays] The indices for each face or None if not 3D.

- **edges**
  
  [list of arrays] The indices for each edge or None if not at least 2D.

- **vertices**
  
  [list of arrays] The indices for each vertex.

**Notes**

The ordering of faces and edges has to be the same as in `sfepy.discrete.fem.geometry_element.geometry_data`.

**sfepy.discrete.iga.iga.get_bezier_topology**(*bconn, degrees*)

Get a topology connectivity corresponding to the Bezier mesh connectivity.

In the referenced Bezier control points the Bezier mesh is interpolatory.

**Parameters**

- **bconn**
  
  [array] The connectivity of the Bezier basis.

- **degrees**
  
  [sequence of ints or int] The basis degrees in each parametric dimension.

**Returns**

- **tconn**
  
  [array] The topology connectivity (corner nodes, or vertices, of Bezier elements) with vertex ordering suitable for a FE mesh.

**sfepy.discrete.iga.iga.get_facet_axes**(*dim*)

For each reference Bezier element facet return the facet axes followed by the remaining (perpendicular) axis, as well as the remaining axis coordinate of the facet.

**Parameters**

- **dim**
  
  [int] The topological dimension.

**Returns**

- **axes**
  

- **coors**
  
  [array] The remaining coordinate of the reference element facets.
sfepy.discrete.iga.iga.get_patch_box_regions(n_els, degrees)
Get box regions of Bezier topological mesh in terms of element corner vertices of Bezier mesh.

Parameters

n_els
[sequence of ints] The number of elements in each parametric dimension.

degrees
[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

regions
[dict] The Bezier mesh vertices of box regions.

sfepy.discrete.iga.iga.get_raveled_index(indices, shape)
Get a global raveled index corresponding to nD indices into an array of the given shape.

sfepy.discrete.iga.iga.get_surface_degrees(degrees)
Get degrees of the NURBS patch surfaces.

Parameters

degrees
[sequence of ints or int] Polynomial degrees in each parametric dimension.

Returns

sdegrees
[list of arrays] The degrees of the patch surfaces, in the order of the reference Bezier element facets.

sfepy.discrete.iga.iga.get_unraveled_indices(index, shape)
Get nD indices into an array of the given shape corresponding to a global raveled index.

sfepy.discrete.iga.iga.tensor_product(a, b)
Compute tensor product of two 2D arrays with possibly different shapes. The result has the form:

\[
\begin{align*}
c & = \begin{bmatrix}
a_{00} b, & a_{01} b, & \ldots, \\
a_{10} b, & a_{11} b, & \ldots, \\
\vdots, & \vdots, & \ldots \\
\end{bmatrix}, \\
\end{align*}
\]

sfepy.discrete.iga.io module

IO for NURBS and Bezier extraction data.

sfepy.discrete.iga.io.read_iga_data(filename, group=None)
Read IGA-related data from a HDF5 file using pytables.

filename: str or tables.File
File to read the hdf5 mesh to.

group: tables.group.Group or None
HDF5 file group to read the mesh from. If it's None, the root of file is used.

Returns
sfepy.discrete.iga.io.write_iga_data:

```
write_iga_data(filename, group, knots, degrees, control_points, weights, cs, conn,
               bezier_control_points, bezier_weights, bezier_conn, regions,
               name=None)
```

Write IGA-related data into a HDF5 file using pytables.

- **filename**: str or tables.File
  - File to read the hdf5 mesh to.
- **group**: tables.group.Group, optional
  - HDF5 file group to read the data from. If None, the root of file is used.

**Returns**

tuple
  - Data for restoring IGA domain.

**sfepy.discrete.iga.mappings module**

Reference mappings for isogeometric analysis.

**class sfepy.discrete.iga.mappings.IGMapping**: (domain, cells, nurbs=None)

Reference mapping for isogeometric analysis based on Bezier extraction.

**Parameters**

- **domain**
  - [IGDomain instance] The mapping domain.
- **cells**
  - [array] The mapping region cells. (All domain cells required.)
- **nurbs**
  - [NurbsPatch instance, optional] If given, the nurbs is used instead of domain.nurbs. The nurbs has to be obtained by degree elevation of domain.nurbs.

**get_geometry()**

Return reference element geometry as a GeometryElement instance.

**get_mapping(qp_coors, weights)**

Get the mapping for given quadrature points and weights.

**Returns**

cmap
  - [CMapping instance] The reference mapping.
Notes

Does not set total volume of the C mapping structure!

get_physical_qps(qp_coors)
Get physical quadrature points corresponding to given reference Bezier element quadrature points.

Returns

qps
[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

sfepy.discrete.iga.plot_nurbs module

sfepy.discrete.iga.plot_nurbs.plot_bezier_mesh(ax, control_points, conn, degrees, label=False)
Plot the Bezier mesh of a NURBS given by its control points and connectivity.

sfepy.discrete.iga.plot_nurbs.plot_bezier_nurbs_basis_1d(ax, control_points, weights, degrees, cs, conn, n_points=20)
Plot a 1D NURBS basis using the Bezier extraction and local Bernstein basis.

sfepy.discrete.iga.plot_nurbs.plot_control_mesh(ax, control_points, label=False)
Plot the control mesh of a NURBS given by its control points.

sfepy.discrete.iga.plot_nurbs.plot_iso_lines(ax, nurbs, color='b', n_points=100)
Plot the NURBS object using iso-lines in Greville abscissae coordinates.

sfepy.discrete.iga.plot_nurbs.plot_nurbs_basis_1d(ax, nurbs, n_points=100, x_axis='parametric', legend=False)
Plot a 1D NURBS basis.

sfepy.discrete.iga.plot_nurbs.plot_parametric_mesh(ax, knots)
Plot the parametric mesh of a NURBS given by its knots.

sfepy.discrete.iga.utils module

Utility functions based on igakit.

sfepy.discrete.iga.utils.create_linear_fe_mesh(nurbs, pars=None)
Convert a NURBS object into a nD-linear tensor product FE mesh.

Parameters

nurbs
[igakit.nurbs.NURBS instance] The NURBS object.

pars
[sequence of array, optional] The values of parameters in each parametric dimension. If not given, the values are set so that the resulting mesh has the same number of vertices as the number of control points/basis functions of the NURBS object.

Returns

coors
[array] The coordinates of mesh vertices.
conn
[array] The vertex connectivity array.

desc
[str] The cell kind.

`sfePy.discrete.iga.utils.create_mesh_and_output(nurbs, pars=None, **kwargs)`
Create a nD-linear tensor product FE mesh using `create_linear_fe_mesh()`, evaluate field variables given as keyword arguments in the mesh vertices and create a dictionary of output data usable by `Mesh.write()`.

Parameters

- **nurbs**
  [igakit.nurbs.NURBS instance] The NURBS object.

- **pars**
  [sequence of array, optional] The values of parameters in each parametric dimension. If not given, the values are set so that the resulting mesh has the same number of vertices as the number of control points/basis functions of the NURBS object.

- **kwargs**
  [kwargs] The field variables as keyword arguments. Their names serve as keys in the output dictionary.

Returns

- **mesh**
  [Mesh instance] The finite element mesh.

- **out**

`sfePy.discrete.iga.utils.save_basis(nurbs, pars)`
Save a NURBS object basis on a FE mesh corresponding to the given parametrization in VTK files.

Parameters

- **nurbs**
  [igakit.nurbs.NURBS instance] The NURBS object.

- **pars**
  [sequence of array, optional] The values of parameters in each parametric dimension.

`sfePy.discrete.structural sub-package`

`sfePy.discrete.structural.fields module`

Fields corresponding to structural elements.

**class** `sfepy.discrete.structural.fields.Shell10XField(name, dtype, shape, region, approx_order=1)`
The approximation for the shell10x element.

- **create_mapping(region, integral, integration, return_mapping=True)**
  Create a new reference mapping.

- **create_output(dofs, var_name, dof_names=..., key=..., thickness=..., **kwargs)**
  Convert the DOFs corresponding to the field to a dictionary of output data usable by `Mesh.write()`.

Parameters
**sfepy.discrete.structural.mappings module**

Finite element reference mappings for structural elements.

```python
class sfepy.discrete.structural.mappings.Shell10XMapping(region, field)
```

The reference mapping for the shell10x element.

```python
get_mapping(qp_coors, weights)
```

Get the mapping for given quadrature points and weights.

```python
get_physical_qps(qp_coors)
```

Get physical quadrature points corresponding the given reference element quadrature points.

Returns

```python
qps
```

[array] The physical quadrature points ordered element by element, i.e. with shape (n_el, n_qp, dim).

**sfepy.homogenization package**

**sfepy.homogenization.band_gaps_app module**

```python
class sfepy.homogenization.band_gaps_app.AcousticBandGapsApp(conf, options, output_prefix, **kwargs)
```

Application for computing acoustic band gaps.

```python
call()
```

Construct and call the homogenization engine according to options.

```python
plot_band_gaps(coefs)
```

```python
plot_dispersion(coefs)
```

**static process_options(options)**

Application options setup. Sets default values for missing non-compulsory options.
static process_options_pv(options)
    Application options setup for phase velocity computation. Sets default values for missing non-compulsory options.

setup_options()

sfepy.homogenization.band_gaps_app.plot_eigs(ax, plot_rsc, plot_labels, valid, freq_range, plot_range)
    Plot resonance/eigen-frequencies.
    valid must correspond to freq_range
    resonances : red masked resonances: dotted red

sfepy.homogenization.band_gaps_app.plot_gap(ax, ranges, kind, kind_desc, plot_range, plot_rsc)
    Plot single band gap frequency ranges as rectangles.

sfepy.homogenization.band_gaps_app.plot_gaps(ax, plot_rsc, gaps, kinds, gap_ranges, freq_range, plot_range)
    Plot band gaps as rectangles.

sfepy.homogenization.band_gaps_app.plot_logs(ax, plot_rsc, plot_labels, freqs, logs, valid, freq_range, plot_range, draw_eigs=True, show_legend=True)
    Plot logs of min/middle/max eigs of a mass matrix.

sfepy.homogenization.band_gaps_app.save_raw_bg_logs(filename, logs)
    Save raw band gaps logs into the filename file.

sfepy.homogenization.band_gaps_app.transform_plot_data(datas, plot_transform, conf)

sfepy.homogenization.band_gaps_app.try_set_defaults(obj, attr, defaults, recur=False)

sfepy.homogenization.coefficients module

class sfepy.homogenization.coefficients.Coefficients(**kwargs)
    Class for storing (homogenized) material coefficients.

    static from_file_hdf5(filename)

    to_file_hdf5(filename)

    to_file_latex(filename, names, format='%.2e', cdot=False, filter=None, idx=None)
        Save the coefficients to a file in LaTeX format.

        Parameters

        filename
            [str] The name of the output file.

        names
            [dict] Mapping of attribute names to LaTeX names.

        format
            [str] Format string for numbers.

        cdot
            [bool] For ‘%.e’ formats only. If True, replace ‘e’ by LaTeX ‘cdot 10^{exponent}’ format.

        filter
            [int] For ‘%.e’ formats only. Typeset as 0, if exponent is less than filter.
idx
    [int] For multi-coefficients, set the coefficient index.

to_file_txt(filename, names, format)
to_latex(attr_name, dim, style='table', format='%.f', step=None)
sfepy.homogenization.coefficients.coef_arrays_to_dicts(idict, format='%.s/%d')

sfepy.homogenization.coefs_base module

class sfepy.homogenization.coefs_base.CofDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CofDimDim(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CofDimSym(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CofDummy(name, problem, kwargs)
    Dummy class serving for computing and returning its requirements.
class sfepy.homogenization.coefs_base.CofEval(name, problem, kwargs)
    Evaluate expression.
class sfepy.homogenization.coefs_base.CofExprPar(name, problem, kwargs)
    The coefficient which expression can be parametrized via 'expr_pars', the dimension is given by the number of
    parameters.
    Example:
    
    'expression': 'dw_surface_ndot.5.Ys(mat_norm.k%d, corr1)', 'expr_pars': [ii for ii in range(dim)],
    'class': cb.CofExprPar,
    static set_variables_default(variables, ir, set_var, data)
class sfepy.homogenization.coefs_base.CofMN(name, problem, kwargs)
    get_coef(row, col, volume, problem, data)
    static set_variables_default(variables, ir, ic, mode, set_var, data, dtype)
class sfepy.homogenization.coefs_base.CofN(name, problem, kwargs)
    get_coef(row, volume, problem, data)
    static set_variables_default(variables, ir, ic, mode, set_var, data, dtype)
class sfepy.homogenization.coefs_base.CofNonSym(name, problem, kwargs)
    is_sym = False
    static iter_sym(dim)
class sfepy.homogenization.coefs_base.CofNonSymNonSym(name, problem, kwargs)
    is_sym = False
    static iter_sym(dim)
class sfepy.homogenization.coefs_base.CofNone(name, problem, kwargs)
class sfepy.homogenization.coefs_base.CofOne(name, problem, kwargs)
    static set_variables_default(variables, set_var, data, dtype)

class sfepy.homogenization.coefs_base.CofSum(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CofSym(name, problem, kwargs)
    is_sym = True
    static iter_sym(dim)

class sfepy.homogenization.coefs_base.CofSymSym(name, problem, kwargs)
    is_sym = True
    static iter_sym(dim)

class sfepy.homogenization.coefs_base.CorrDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CorrDimDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CorrEqPar(name, problem, kwargs)
    The corrector which equation can be parametrized via 'eq_pars', the dimension is given by the number of parameters.
    Example:
    'equations': 'dw_diffusion.5.Y(mat.k, q, p) = dw_integrate.5.%s(q)',
    'eq_pars': ('bYMp', 'bYMm'), 'class': cb.CorrEqPar,

class sfepy.homogenization.coefs_base.CorrEval(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CorrMiniApp(name, problem, kwargs)
    get_output(corr_sol, is_dump=False, extend=True, variables=None, var_map=None)
    get_save_name(save_format='h5', stamp='')
    get_save_name_base()
    save(state, problem, variables=None, ts=None, var_map=None)
    setup_output(save_formats=None, post_process_hook=None, split_results_by=None)
    Instance attributes have precedence!

class sfepy.homogenization.coefs_base.CorrN(name, problem, kwargs)
    static set_variables_default(variables, ir, set_var, data)

class sfepy.homogenization.coefs_base.CorrNN(name, problem, kwargs)
    _init__(kwsargs: {}
        '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)
```python
static set_variables_default(variables, set_var, data)

class sfepy.homogenization.coefs_base.CorrelationSetBCSCorrelationSetBCS(name, problem, kwargs)

class sfepy.homogenization.coefs_base.CorrelationSolutionCorrelationSolution(**kwargs)
    Class for holding solutions of corrector problems.
    get_output(is_dump=False, var_map=None)
    get_ts_val(step)
    iter_solutions()
    iter_time_steps()

class sfepy.homogenization.coefs_base.MiniAppBaseMiniAppBase(name, problem, kwargs)
    static any_from_conf(name, problem, kwargs)
    init_solvers(problem)
        Setup solvers. Use local options if these are defined, otherwise use the global ones.
        For linear problems, assemble the matrix and try to presolve the linear system.
    process_options()
        Setup application-specific options.
        Subclasses should implement this method as needed.
        Returns
        app_options [Struct instance] The application options.

class sfepy.homogenization.coefs_base.OnesDimOnesDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.PressureEigenvalueProblemPressureEigenvalueProblem(name, problem, kwargs)
    Pressure eigenvalue problem solver for time-dependent correctors.
    presolve(mtx)
        Prepare $A^{-1}B^T$ for the Schur complement.
    solve_pressure_eigenproblem(mtx, eig_problem=None, n_eigs=0, check=False)
        $G = B^*A^*B^T$ or $B^*A^*B^T + D$

class sfepy.homogenization.coefs_base.ShapeDimShapeDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.ShapeDimDimShapeDimDim(name, problem, kwargs)

class sfepy.homogenization.coefs_base.TCorrectorsViaPressureEVPTCorrectorsViaPressureEVP(name, problem, kwargs)
    Time correctors via the pressure eigenvalue problem.
    compute_correctors(evp, sign, state0, ts, problem=None, vec_g=None)
    save(corrs, problem, ts)
    setup_equations(equations, problem=None)
        Set equations, update boundary conditions and materials.

class sfepy.homogenization.coefs_base.TSTimesTSTimes(name, problem, kwargs)
    Coefficient-like class, returns times of the time stepper.
```

2.3. Developer Guide
class sfepy.homogenization.coefs_base.VolumeFractions(name, problem, kwars)
    Coefficient-like class, returns volume fractions of given regions within the whole domain.
sfepy.homogenization.coefs_base.create_ts_coef(cls)
    Define a new class with modified call method which accepts time dependent data (correctors).

sfepy.homogenization.coefs_elastic module

class sfepy.homogenization.coefs_elastic.PressureRHSVector(name, problem, kwars)
class sfepy.homogenization.coefs_elastic.TCorrectorsPressureViaPressureEVP(name, problem, kwars)
class sfepy.homogenization.coefs_elastic.TCorrectorsRSViaPressureEVP(name, problem, kwars)

sfepy.homogenization.coefs_perfusion module

class sfepy.homogenization.coefs_perfusion.CoeffRegion(name, problem, kwars)
    get_variables(problem, ir, data)
class sfepy.homogenization.coefs_perfusion.CorrRegion(name, problem, kwars)
    get_variables(ir, data)

sfepy.homogenization.coefs_phononic module

class sfepy.homogenization.coefs_phononic.AcousticMassLiquidTensor(name, problem, kwars)
    get_coefs(freq)
    Get frequency-dependent coefficients.
class sfepy.homogenization.coefs_phononic.AcousticMassTensor(name, problem, kwars)
    The acoustic mass tensor for a given frequency.
    Returns
    self
        [AcousticMassTensor instance] This class instance whose evaluate() method computes for a
given frequency the required tensor.

Notes

eigenmomenta, eigs should contain only valid resonances.

evaluate(freq)

get_coefs(freq)
    Get frequency-dependent coefficients.
to_file_txt = None
class sfepy.homogenization.coefs_phononic.AppliedLoadTensor(name, problem, kwargs)

The applied load tensor for a given frequency.

Returns

self

[AppliedLoadTensor instance] This class instance whose evaluate() method computes for a given frequency the required tensor.

Notes

eigenmomenta, ueigenmomenta, eigs should contain only valid resonances.

evaluate(freq)

to_file_txt = None

class sfepy.homogenization.coefs_phononic.BandGaps(name, problem, kwargs)

Band gaps detection.

Parameters

eigensolver

[str] The name of the eigensolver for mass matrix eigenvalues.

eig_range

[(int, int)] The eigenvalues range (squared frequency) to consider.

freq_margins

[[float, float]] Margins in percents of initial frequency range given by eig_range by which the range is increased.

fixed_freq_range

[(float, float)] The frequency range to consider. Has precedence over eig_range and freq_margins.

freq_step

[float] The frequency step for tracing, in percent of the frequency range.

freq_eps

[float] The frequency difference smaller than freq_eps is considered zero.

zero_eps

[float] The tolerance for finding zeros of mass matrix eigenvalues.

detect_fun

[callable] The function for detecting the band gaps. Default is detect_band_gaps().

log_save_name

[str] If not None, the band gaps log is to be saved under the given name.

raw_log_save_name

[str] If not None, the raw band gaps log is to be saved under the given name.

fix_eig_range(n_eigs)

process_options()

Setup application-specific options.

Subclasses should implement this method as needed.

Returns
**app_options**

[Struct instance] The application options.

**static save_log** *(filename, float_format, bg)*

Save band gaps, valid flags and eigenfrequencies.

**static to_file_txt** *(fd, float_format, bg)*

**class** `sfepy.homogenization.coefs_phononic.ChristoffelAcousticTensor` *(name, problem, kwargs)*

**process_options** ()

Setup application-specific options.

Subclasses should implement this method as needed.

**Returns**

**app_options**

[Struct instance] The application options.

**class** `sfepy.homogenization.coefs_phononic.DensityVolumeInfo` *(name, problem, kwargs)*

Determine densities of regions specified in `region_to_material`, and compute average density based on region volumes.

**static to_file_txt** *(fd, float_format, dv_info)*

**class** `sfepy.homogenization.coefs_phononic.Eigenmomenta` *(name, problem, kwargs)*

Eigenmomenta corresponding to eigenvectors.

**Parameters**

var_name

[str] The name of the variable used in the integral.

threshold

[float] The threshold under which an eigenmomentum is considered zero.

threshold_is_relative

[bool] If True, the threshold is relative w.r.t. max. norm of eigenmomenta.

transform

[callable, optional] Optional function for transforming the eigenvectors before computing the eigenmomenta.

**Returns**

eigenmomenta

[Struct] The resulting eigenmomenta. An eigenmomentum above threshold is marked by the attribute ‘valid’ set to True.

**process_options** ()

Setup application-specific options.

Subclasses should implement this method as needed.

**Returns**

**app_options**

[Struct instance] The application options.

**class** `sfepy.homogenization.coefs_phononic.PhaseVelocity` *(name, problem, kwargs)*

Compute phase velocity.
process_options()
Setup application-specific options.
Subclasses should implement this method as needed.

**Returns**

/app_options
[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.PolarizationAngles(name, problem, kwargs)
Compute polarization angles, i.e., angles between incident wave direction and wave vectors. Vector length does not matter - eigenvectors are used directly.

process_options()
Setup application-specific options.
Subclasses should implement this method as needed.

**Returns**

/app_options
[Struct instance] The application options.

class sfepy.homogenization.coefs_phononic.SchurEVP(name, problem, kwargs)
Schur complement eigenvalue problem.

post_process(eigs, mtx_s_phi, mtx_dib, problem)

prepare_matrices(problem)
A = K + B^T D^{-1} B

class sfepy.homogenization.coefs_phononic.SimpleEVP(name, problem, kwargs)
Simple eigenvalue problem.

post_process(eigs, mtx_s_phi, data, problem)

prepare_matrices(problem)

process_options()
Setup application-specific options.
Subclasses should implement this method as needed.

**Returns**

/app_options
[Struct instance] The application options.

save(eigs, mtx_phi, problem)

sfepy.homogenization.coefs_phononic.compute_cat_dim_dim(coef, iw_dir)
Christoffel acoustic tensor part of dielectric tensor dimension.

sfepy.homogenization.coefs_phononic.compute_cat_dim_sym(coef, iw_dir)
Christoffel acoustic tensor part of piezo-coupling tensor dimension.

sfepy.homogenization.coefs_phononic.compute_cat_sym_sym(coef, iw_dir)
Christoffel acoustic tensor (part) of elasticity tensor dimension.
sfepy.homogenization.coefs_phononic.compute_eigenmomenta(em_equation, var_name, problem, eig_vectors, transform=None)

Compute the eigenmomenta corresponding to given eigenvectors.

sfepy.homogenization.coefs_phononic.cut_freq_range(freq_range, eigs, valid, freq_margins, eig_range, fixed_freq_range, freq_eps)

Cut off masked resonance frequencies. Margins are preserved, like no resonances were cut.

Returns

freq_range
[array] The new range of frequencies.

freq_range_margins
[array] The range of frequencies with prepended/appended margins equal to fixed_freq_range if it is not None.

sfepy.homogenization.coefs_phononic.describe_gaps(gaps)

sfepy.homogenization.coefs_phononic.detect_band_gaps(mass, freq_info, opts, gap_kind='normal', mtx_b=None)

Detect band gaps given solution to eigenproblem (eigs, eig_vectors). Only valid resonance frequencies (i.e. those for which corresponding eigenmomenta are above a given threshold) are taken into account.

Notes

- make freq_eps relative to \([f_0, f_1]\) size?

sfepy.homogenization.coefs_phononic.find_zero(f0, f1, callback, freq_eps, zero_eps, mode)

For f in \([f_0, f_1]\) 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:

<table>
<thead>
<tr>
<th>mode</th>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1</td>
<td>0</td>
<td>eigenvalue (-\infty) for f in ([f_0, f_1])</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>f \rightarrow f_1, smallest eigenvalue &lt; 0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>f \rightarrow f_0, smallest eigenvalue &gt; 0 and (-\infty)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>f \rightarrow f_1, largest eigenvalue &lt; 0 and (+\infty)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>f \rightarrow f_0, largest eigenvalue &gt; 0</td>
</tr>
</tbody>
</table>
sfepy.homogenization.coefs Phononic.get_callback(mass, solver_kind, mtx_b=None, mode='trace')

Return callback to solve band gaps or dispersion eigenproblem P.

Notes

Find zero callbacks return:

eigenvalues

Trace callbacks return:

(eigenvalues,)

or

(eigenvalues, eigenvectors) (in full (dispoersion) mode)

If mtx_b is None, the problem P is

\[ M w = \lambda w, \]

otherwise it is

\[ \omega^2 M w = \eta B w \]

sfepy.homogenization.coefs Phononic.get_gap_ranges(freq_range, gaps, kinds)

For each (potential) band gap in gaps, return the frequency ranges of its parts according to kinds.

sfepy.homogenization.coefs Phononic.get_log_freqs(f0, f1, df, freq_eps, n_point_min, n_point_max)

Get logging frequencies.

The frequencies get denser towards the interval boundaries.

sfepy.homogenization.coefs Phononic.get_ranges(freq_range, eigs)

Get an eigenvalue range slice and a corresponding initial frequency range within a given frequency range.

sfepy.homogenization.coefs Phononic.split_chunks(indx)

Split index vector to chunks of consecutive numbers.

sfepy.homogenization.convolution module

class sfepy.homogenization.convolution.ConvolutionKernel(name, times, kernel, decay=None, exp_coefs=None, exp_decay=None)

The convolution kernel with exponential synchronous decay approximation approximating the original kernel represented by the array \( c[i], i = 0, 1, \ldots \)

\[
c_0 \equiv c[0], c_{e0} \equiv c_0 e^0, \\
c(t) \approx c_0 d(t) \approx c_0 e(t) = c_{e0} e(t),
\]

where \( d(0) = e_n(0) = 1 \), \( d \) is the synchronous decay and \( e \) its exponential approximation, \( e = c_{e0} \exp(-c_{e1} t). \)

**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.
The integral of the kernel in time.

```python
int_dt(use_exp=False)
```

The integral of the kernel in time.

```python
sfepy.homogenization.convolutions.approximate_exponential(x, y)
```

Approximate \( y = f(x) \) by \( y_a = c_1 \exp(-c_2 x) \).

Initial guess is given by assuming \( y \) has already the required exponential form.

```python
sfepy.homogenization.convolutions.compute_mean_decay(coef)
```

Compute mean decay approximation of a non-scalar fading memory coefficient.

```python
sfepy.homogenization.convolutions.eval_exponential(coefs, x)
```

Evaluate \( y = f(x) \) after approximating \( f \) by an exponential.

```python
sfepy.homogenization.convolutions.fit_exponential(x, y, return_coefs=False)
```

Evaluate \( y = f(x) \) after approximating \( f \) by an exponential.

### sfepy.homogenization.engine module

```python
class sfepy.homogenization.engine.CoefVolume(name, problem, kwargs)

class sfepy.homogenization.engine.HomogenizationEngine(problem, options, app_options=None, volumes=None, output_prefix='he:', **kwargs)
```

### Call

```python
call(ret_all=False, time_tag='')
```

### static define_volume_coef(coef_info, volumes)

Define volume coefficients and make all other dependent on them.

**Parameters**

- `coef_info` [dict] The coefficient definitions.
- `volumes` [dict] The definitions of volumes.

**Returns**

- `coef_info` [dict] The coefficient definitions extended by the volume coefficients.

### static process_options(options)

Application options setup. Sets default values for missing non-compulsory options.

### set_micro_states(states)

### setup_options(app_options=None)

```python
class sfepy.homogenization.engine.HomogenizationWorker
```

### static calculate(mini_app, problem, dependencies, dep_requires, save_names, micro_states, chunk_tab, mode, proc_id)

Calculate a requirement, i.e. correctors or coefficients.

**Parameters**
problem
[problem] The problem definition related to the microstructure.

opts
[struct] The options of the homogenization application.

post_process_hook
[function] The postprocessing hook.

name
[det] The name of the requirement.

req_info
[dict] The definition of correctors.

coef_info
[dict] The definition of homogenized coefficients.

save_names
[dict] The dictionary containing names of saved correctors.

dependencies
[dict] The dependencies required by the correctors/coefficients.

micro_states
[array] The configurations of multiple microstructures.

time_tag
[det] The label corresponding to the actual time step and iteration, used in the corrector file names.

chunk_tab
[list] In the case of multiprocessing the requirements are divided into several chunks that are solved in parallel.

proc_id
[int] The id number of the processor (core) which is solving the actual chunk.

Returns
val
[coefficient/corrector or list of coefficients/correctors] The resulting homogenized coefficients or correctors.

static get_sorted_dependencies(req_info, coef_info, compute_only)
Make corrs and coefs list sorted according to the dependencies.

class sfepy.homogenization.engine.HomogenizationWorkerMulti(num_workers)

static calculate_req_multi(tasks, lock, remaining, numdeps, inverse_deps, problem, opts, post_process_hook, req_info, coef_info, save_names, dependencies, micro_states, time_tag, chunk_tab, proc_id)
Calculate a requirement in parallel.

Parameters

tasks
[queue] The queue of requirements to be solved.

lock
[lock] The multiprocessing lock used to ensure save access to the global variables.
remaining
  [int] The number of remaining requirements.

numdeps
  [dict] The number of dependencies for the each requirement.

inverse_deps
  [dict] The inverse dependencies - which requirements depend on a given one.

For the definition of other parameters see ‘calculate_req’.

static chunk_micro_tasks(num_workers, num_micro, reqs, coefs, chunks_per_worker=1, 
store_micro_idxs=[])

Split multiple microproblems into several chunks that can be processed in parallel.

Parameters

num_workers
  [int] The number of available CPUs.

num_micro
  [int] The number of microstructures.

reqs
  [dict] The requirement definitions.

coefs
  [dict] The coefficient definitions.

chunks_per_worker
  [int] The number of chunks per one worker.

store_micro_idxs
  [list of int] The indices of microstructures whose results are to be stored.

Returns

micro_tab
  [list of slices] The indices of microproblems contained in each chunk.

new_reqs

new_coefs

static dechunk_reqs_coefs(deps, num_chunks)

Merge the results related to the multiple microproblems.

Parameters

deps
  [dict] The calculated dependencies.

num_chunks
  [int] The number of chunks.

Returns

new_deps
  [dict] The merged dependencies.

static process_reqs_coefs(old, num_workers, store_idxs=[])
class sfepy.homogenization.engine.HomogenizationWorkerMultiMPI(num_workers)

sfepy.homogenization.engine.get_dict_idxval(dict_array, idx)

sfepy.homogenization.engine.insert_sub_reqs(reqs, levels, req_info)
   Recursively build all requirements in correct order.

sfepy.homogenization.homogen_app module

class sfepy.homogenization.homogen_app.HomogenizationApp(conf, options, output_prefix, **kwargs)
   call(verbose=False, ret_all=None, itime=None, iiter=None)
      Call the homogenization engine and compute the homogenized coefficients.

      Parameters
      verbose [bool] If True, print the computed coefficients.
      ret_all [bool or None] If not None, it can be used to override the ‘return_all’ option. If True, also
      the dependencies are returned.
      time_tag: str
         The time tag used in file names.

      Returns
      coefs [Coefficients instance] The homogenized coefficients.
      dependencies [dict] The dependencies, if ret_all is True.

get_micro_cache_key(key, icoor, itime)

static process_options(options)
   Application options setup. Sets default values for missing non-compulsory options.

setup_macro_data(data)
   Setup macroscopic deformation gradient.

setup_options()

update_micro_states()
   Update microstructures state according to the macroscopic data and corrector functions.

sfepy.homogenization.micmac module

sfepy.homogenization.micmac.get_correctors_from_file_hdf5(coefs_filename='coefs.h5',
   dump_names=None)

sfepy.homogenization.micmac.get_homog_coefs_linear(ts, coor, mode, micro_filename=None,
   regenerate=False, coefs_filename=None, define_args=None, output_dir=None)
sfepy.homogenization.micmac.get_homog_coefs_nonlinear(ts, coor, mode, macro_data=None, 
term=None, problem=None, iteration=None, 
define_args=None, output_dir=None, 
**kwargs)

sfepy.homogenization.recovery module

sfepy.homogenization.recovery.add_strain_rs(corrs_rs, strain, vu, dim, iel, out=None)
sfepy.homogenization.recovery.add_stress_p(out, pb, integral, region, vp, data)
sfepy.homogenization.recovery.combine_scalar_grad(corrs, grad, vn, ii, shift_coors=None)

\[ \eta_k \partial_k p \]

or

\[ (y_k + \eta_k) \partial_k p \]
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.

\[ u^1 = \chi^{ij} e^{ij}(u^0) \]
sfepy.homogenization.recovery.compute_p_corr_steady(corrs_pressure, pressure, vp, iel)

\[ \pi^P p \]
sfepy.homogenization.recovery.compute_p_corr_time(corrs_rs, dstrains, corrs_pressure, pressures, vdp, 
dim, iel, ts)

\[ \sum_{ij} \int_0^t \frac{d}{dt} \pi^{ij}(t-s) \frac{d}{ds} e_{ij}(u(s)) \, ds + \int_0^t \frac{d}{dt} \pi^P(t-s) p(s) \, ds \]
sfepy.homogenization.recovery.compute_p_from_macro(p_grad, coor, iel, centre=None, extdim=0)

Macro-induced pressure.

\[ \partial_j^y (y_j - y^c_j) \]
sfepy.homogenization.recovery.compute_stress_strain_u(pb, integral, region, material, vu, data)
sfepy.homogenization.recovery.compute_u_corr_steady(corrs_rs, strain, vu, dim, iel)

\[ \sum_j \omega^{ij} e_{ij}(u) \]
Notes

• iel = element number

\[
\sum_{ij} \left[ \int_0^t \omega^{ij}(t-s) \frac{d}{ds} e_{ij}(u(s)) \, ds \right] + \int_0^t \bar{\omega}^P(t-s) p(s) \, ds
\]

sfepy.homogenization.recovery.compute_u_corr_time(corrRs, dstrains, corrs_pressure, pressures, vu, dim, iel, ts)

Macro-induced displacements.

\[
e^x_{ij}(u)(y_j - y^c_j)
\]

sfepy.homogenization.recovery.compute_u_from_macro(strain, coor, iel, centre=None)

Notes

• t is given by step

• f: fvars scalar field variables, defined in a micro domain, have shape [step][fmf dims]

• p: pvars scalar point variables, a scalar in a point of macro-domain, FMField style have shape [n_step][var_dims]

sfepy.homogenization.recovery.convolve_field_scalar(fvars, pvars, iel, ts)

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.convolve_field_sym_tensor(fvars, pvars, var_name, dim, iel, ts)

sfepy.homogenization.recovery.destroy_pool()

sfepy.homogenization.recovery.get_output_suffix(iel, ts, naming_scheme, format, output_format)

sfepy.homogenization.recovery.get_recovery_points(region, eps0)

sfepy.homogenization.recovery.recover_bones(problem, micro_problem, region, eps0, ts, strain, dstrains, p_grad, pressures, corrs_permeability, corrs_rs, corrs_time_rs, corrs_pressure, corrs_time_pressure, var_names, naming_scheme='step_iel')
Notes

- note that

\[ \tilde{\pi} P \]

is in corrs_pressure -> from time correctors only ‘u’, ‘dp’ are needed.

\[ \text{sfepy.homogenization.recovery.recover_micro_hook}( \text{micro_filename}, \text{region}, \text{macro}, \text{eps0}, \text{region_mode}='el_centers', \text{eval_mode}='constant', \text{eval_vars}=None, \text{corrs}=None, \text{recovery_file_tag}=\”, \text{define_args}=None, \text{output_dir}=None, \text{verbose}=False) \]

Parameters

**micro_filename**
- [str] The definition file of the microproblem.

**region**
- [Region or array] The macroscopic region to be recovered. If array, the centers of microscopic RVEs (Representative Volume Element) are assumed to be stored in it. If Region, the RVE centers are computed according to \text{region_mode}, see below.

**macro**
- [dict of arrays or tuples] Either macroscopic values (if array) or the tuple (mode, eval_var, nodal_values) is expected. The tuple is used to evaluate the macroscopic values in given points of RVEs (see ‘eval_mode’). \text{mode} can be ‘val’, ‘grad’, ‘div’, or ‘cauchy_strain’.

**eps0**
- [float] The size of the microstructures (RVE).

**region_mode**
- [\{‘el_centers’, ‘tiled’\}] If ‘el_centers’, the RVE centers are identical to the element centers of the macroscopic FE mesh. If ‘tiled’, the recovered region is tiled by rescaled RVEs.

**eval_mode**
- [\{‘constant’, ‘continuous’\}] If ‘constant’, the macroscopic fields are evaluated only at the RVE centers. If ‘continuous’, the fields are evaluated at all points of the RVE mesh.

**eval_vars**
- [list of variables] The list of variables use to evaluate the macroscopic fields.

**corrs**
- [dict of CorrSolution] The correctors for recovery.

**recovery_file_tag**
- [str] The tag which is appended to the output file.

**define_args**
- [dict] The define arguments for the microscopic problem.

**output_dir**
- [str] The output directory.

**verbose**
- [bool] The verbose terminal output.
sfepy.homogenization.recovery.recover_paraflow(problem, micro_problem, region, ts, strain, dstrains, pressures1, pressures2, corrs_rs, corrs_time_rs, corrs_alpha1, corrs_time_alpha1, corrs_alpha2, corrs_time_alpha2, var_names, naming_scheme='step_iel')

sfepy.homogenization.recovery.save_recovery_region(mac_pb, rname, filename=None)

sfepy.homogenization.utils module

sfepy.homogenization.utils.build_op_pi(var, ir, ic)
   \Pi_{ir}^{rs} = y_s \delta_{ir} \text{ for } r = ir, s = ic.

sfepy.homogenization.utils.coor_to_sym(ir, ic, dim)

sfepy.homogenization.utils.create_pis(problem, var_name)
   \Pi_{ir}^{rs} = y_s \delta_{ir}, \ul{y} \text{ in Y coordinates.}

sfepy.homogenization.utils.create_scalar_pis(problem, var_name)
   \Pi_{ik}^{k} = y_k, \ul{y} \text{ in Y coordinates.}

sfepy.homogenization.utils.define_box_regions(dim, lbn=None, rtf=None, eps=0.001, kind='facet')
   Define sides and corner regions for a box aligned with coordinate axes.

   Parameters
   
   dim
      [int] Space dimension

   lbn
      [tuple] Left bottom near point coordinates if rtf is not None. If rtf is None, lbn are the (positive) distances from the origin.

   rtf
      [tuple] Right top far point coordinates.

   eps
      [float] A parameter, that should be smaller than the smallest mesh node distance.

   kind
      [bool, optional] The region kind.

   Returns
   
   regions
      [dict] The box regions.

sfepy.homogenization.utils.get_box_volume(dim, lbn, rtf=None)
   Volume of a box aligned with coordinate axes.

   Parameters:

   dim
      [int] Space dimension

   lbn
      [tuple] Left bottom near point coordinates if rtf is not None. If rtf is None, lbn are the (positive) distances from the origin.
rtf
[tuple] Right top far point coordinates.
Returns:
volume
[float] The box volume.
sfepy.homogenization.utils.get_lattice_volume(axes)
Volume of a periodic cell in a rectangular 3D (or 2D) lattice.
Parameters
axes
[array] The array with the periodic cell axes $a_1, \ldots, a_3$ as rows.
Returns
volume
[float] The periodic cell volume $V = (a_1 \times a_2) \cdot a_3$. In 2D $V = |(a_1 \times a_2)|$ with zeros as the third components of vectors $a_1, a_2$.
sfepy.homogenization.utils.get_volume(problem, field_name, region_name, quad_order=1)
Get volume of a given region using integration defined by a given field. Both the region and the field have to be defined in problem.
sfepy.homogenization.utils.integrate_in_time(coef, ts, scheme='forward')
Forward difference or trapezoidal rule. ‘ts’ can be anything with ‘times’ attribute.
sfepy.homogenization.utils.interp_conv_mat(mat, ts, tdiff)
sfepy.homogenization.utils.iter_nonsym(dim)
sfepy.homogenization.utils.iter_sym(dim)
sfepy.homogenization.utils.rm_multi(s)
sfepy.homogenization.utils.set_nonlin_states(variables, nl_state, problem)
Setup reference state for nonlinear homogenization
Parameters
variables
[dict] All problem variables
nl_state
[reference state]
problem
[problem description]
sfepy.linalg package

sfepy.linalg.check_derivatives module

Utilities for checking derivatives of functions.

sfepy.linalg.check_derivatives.check_fx(x0, fx, dx_args=None, dfx=None, dfx_args=None, delta=1e-05)

Check derivatives of a (vectorized) scalar function of a scalar variable.

sfepy.linalg.check_derivatives.check_vfvx(x0, fx, dfx, dx_args=None, dfx_args=None, delta=1e-05)

Check derivatives of a (vectorized) vector or scalar function of a vector variable.

sfepy.linalg.eigen module

sfepy.linalg.eigen.cg_eigs(mtx, rhs=None, precond=None, i_max=None, eps_r=1e-10, shift=None, select_indices=None, verbose=False, report_step=10)

Make several iterations of the conjugate gradients and estimate so the eigenvalues of a (sparse SPD) matrix (Lanczos algorithm).

Parameters

mtx
[spmatrix or array] The sparse matrix \( A \).

precond
[spmatrix or array, optional] The preconditioner matrix. Any object that can be multiplied by vector can be passed.

i_max
[int] The maximum number of the Lanczos algorithm iterations.

eps_r
[float] The relative stopping tolerance.

shift
[float, optional] Eigenvalue shift for non-SPD matrices. If negative, the shift is computed as \(|\text{shift}||A||_\infty|\).

select_indices
[(min, max), optional] If given, computed only the eigenvalues with indices \( \text{min} \leq i \leq \text{max} \).

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.

```python
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**

```python
sfepy.linalg.geometry.barycentric_coors(coors, s_coors)
```

Get barycentric (area in 2D, volume in 3D) coordinates of points with coordinates `coors` w.r.t. the simplex given by `s_coors`.

**Returns**

- `bc` [array] The barycentric coordinates. Then reference element coordinates \( xi = \text{dot}(bc.T, ref_coors) \).

```python
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)

```python
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.

```python
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 <= n_v <= (dim + 1).
- `force_inside_eps` (float, optional) If not None, move the circumcentres that are outside of their simplices or closer to their boundary then `force_inside_eps` so that they are inside the simplices at the distance given by `force_inside_eps`. It is ignored for edges.

**Returns**

- `centres` (array) The circumcentre coordinates as an array of shape (n_s, dim).

```
sfepy.linalg.geometry.get_simplex_volumes(cells, coors)
```

Get volumes of simplices in nD.

**Parameters**

- `cells` (array, shape (n, d]) The indices of n simplices with d vertices into `coors`.
- `coors` (array) The coordinates of simplex vertices.

**Returns**

- `volumes` (array) The volumes of the simplices.

```
sfepy.linalg.geometry.inverse_element_mapping(coors, e_coors, eval_base, ref_coors, suppress_errors=False)
```

Given spatial element coordinates, find the inverse mapping for points with coordinates X = X(xi), i.e. xi = xi(X).
Returns

\( \mathbf{x} \)


\texttt{sfepy.linalg.geometry.make_axis_rotation_matrix(direction, angle)}

Create a rotation matrix \( \mathbf{R} \) corresponding to the rotation around a general axis \( \mathbf{d} \) by a specified angle \( \alpha \).

\[ \mathbf{R} = \mathbf{d} \mathbf{d}^T + \cos(\alpha)(\mathbf{I} - \mathbf{d} \mathbf{d}^T) + \sin(\alpha)\text{skew}(\mathbf{d}) \]

Parameters

\textbf{direction}

[array] The rotation axis direction vector \( \mathbf{d} \).

\textbf{angle}

[float] The rotation angle \( \alpha \).

Returns

\( \mathbf{mtx} \)

[array] The rotation matrix \( \mathbf{R} \).

Notes

The matrix follows the right hand rule: if the right hand thumb points along the axis vector \( \mathbf{d} \) the fingers show the positive angle rotation direction.

Examples

Make transformation matrix for rotation of coordinate system by 90 degrees around ‘z’ axis.

```python
>>> mtx = make_axis_rotation_matrix([0., 0., 1.], np.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.)

```python
>>> np.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.)

```python
>>> np.dot(mtx.T, [1., 0., 0.])
>>> array([ 0., 1., 0.])
```

\texttt{sfepy.linalg.geometry.points_in_simplex(coors, s_coors, eps=1e-08)}

Test if points with coordinates \texttt{coors} are in the simplex given by \texttt{s_coors}.

\texttt{sfepy.linalg.geometry.rotation_matrix2d(angle)}

Construct a 2D (plane) rotation matrix corresponding to \texttt{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.

```python
>>> 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:

```python
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

```python
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`.

```python
sfepy.linalg.sparse.save_sparse_txt(filename, mtx, fmt='%d %d %f
')
```
Save a CSR/CSC sparse matrix into a text file

**sfepy.linalg.sympy_operators module**

```python
sfepy.linalg.sympy_operators.boundary(f, variables)
```
```python
sfepy.linalg.sympy_operators.default_space_variables(variables)
```
```python
sfepy.linalg.sympy_operators.div(field, variables=None)
```
```python
sfepy.linalg.sympy_operators.grad(f, variables=None)
```
```python
sfepy.linalg.sympy_operators.grad_v(f, variables=None)
```
```python
sfepy.linalg.sympy_operators.laplace(f, variables=None)
```
```python
sfepy.linalg.sympy_operators.set_dim(dim)
```

**sfepy.linalg.utils module**

```python
class sfepy.linalg.utils.MatrixAction(**kwargs)
```
```python
    static from_array(arr)
```
```python
    static from_function(fun, expected_shape, dtype)
```
```python
    to_array()
```

```python
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, ..., n_r, m_1, ..., 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 \((n_f, \ldots, n_r) + \text{out\_item\_shape}\). The \text{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 \([0<=i<a, 0<=j<b, 0<=k<c, \ldots]\) 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])) Out[14]: [[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1], [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]

sfepy.linalg.utils.dets_fast(a)

Fast determinant calculation of 3-dimensional array.

Parameters

a

[array] The input array with shape \((m, n, n)\).

Returns

out

[array] The output array with shape \((m,)\): out[i] = det(a[i, :, :]).

sfepy.linalg.utils.dot_sequences(mtx, vec, mode='AB')

Computes dot product for each pair of items in the two sequences.

Equivalent to

```python
>>> 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])
```
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 :py:func:`numpy.matmul()` via the @ operator.

`sfepy.linalg.utils.get_blocks_stats(blocks, *args)`

Return statistics of array/matrix blocks defined by indices in args.

Returns

stats: structured array

The array with ‘shape’, ‘min’, ‘mean’ and ‘max’ fields at positions of each matrix block.

Examples

```python
>>> import numpy as nm
>>> from sfepy.linalg.utils import get_blocks_stats

>>> A = nm.eye(3)
>>> B = nm.full((3,2), 2)
>>> C = nm.full((1,3), 3)
>>> D = nm.full((1,2), 4)
>>> M = nm.block([[A, B], [C, D]])

>>> sr = [slice(0, 3), slice(3, 5)]
>>> sc = [slice(0, 3), slice(3, 4)]
>>> stats = get_blocks_stats(M, sr, sc)

>>> print(stats['shape'])
[[[3, 3] [3, 1]]
 [[1, 3] [1, 1]]

>>> print(stats['min'])
[[0. 2.]
 [3. 4.]]
```

`sfepy.linalg.utils.insert_strided_axis(ar, axis, length)`

Insert a new axis of given length into an array using numpy stride tricks, i.e. no copy is made.
Parameters

ar
[array] The input array.

axis
[int] The axis before which the new axis will be inserted.

length
[int] The length of the inserted axis.

Returns

out
[array] The output array sharing data with ar.

Examples

>>> import numpy as nm
>>> from sfepy.linalg import insert_strided_axis
>>> ar = nm.random.rand(2, 1, 2)
>>> ar
array([[ 0.18905119, 0.44552425],
       [ 0.78593989, 0.71852473]],
       [[ 0.78593989, 0.71852473]])

>>> ar.shape
(2, 1, 2)
>>> ar2 = insert_strided_axis(ar, 1, 3)
>>> ar2
array([[[[ 0.18905119, 0.44552425]],
         [[ 0.18905119, 0.44552425]],
         [[ 0.18905119, 0.44552425]]],
       [[[ 0.78593989, 0.71852473]],
         [[ 0.78593989, 0.71852473]],
         [[ 0.78593989, 0.71852473]]],
       [[[ 0.78593989, 0.71852473]],
         [[ 0.78593989, 0.71852473]],
         [[ 0.78593989, 0.71852473]]])

>>> ar2.shape
(2, 3, 1, 2)

sfepy.linalg.utils.invs_fast(a, det=None)
Fast inversion calculation of 4-dimensional array.

Parameters

a
[array] The input array with shape (c, q, n, n).

det: array
To speed up the calculation, enter the already calculated determinant.

Returns
out
            [array] The output array with shape (c, q, n, n): out[c, q] = inv(a[c, q, :, :]).

`sfepy.linalg.utils.map_permutations(seq1, seq2, check_same_items=False)`

Returns an index array `imap` such that `seq1[imap] == seq2`, if both sequences have the same items - this is not checked by default!

In other words, finds the indices of items of `seq2` in `seq1`.

`sfepy.linalg.utils.max_diff_csr(mtx1, mtx2)`

`sfepy.linalg.utils.mini_newton(fun, x0, dfun, i_max=100, eps=1e-08)`

`sfepy.linalg.utils.norm_l2_along_axis(ar, axis=1, n_item=None, squared=False)`

Compute l2 norm of rows (axis=1) or columns (axis=0) of a 2D array.

`n_item ... use only the first n_item columns/rows squared ... if True, return the norm squared`

`sfepy.linalg.utils.normalize_vectors(vecs, eps=1e-08)`

Normalize an array of vectors in place.

**Parameters**

- **vecs**
  - [array] The 2D array of vectors in rows.

- **eps**
  - [float] The tolerance for considering a vector to have zero norm. Such vectors are left unchanged.

`sfepy.linalg.utils.output_array_stats(ar, name, verbose=True)`

`sfepy.linalg.utils.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.mechanics package**

**sfepy.mechanics.contact_bodies module**

**class** `sfepy.mechanics.contact_bodies.ContactPlane(anchor, normal, bounds)`

- **get_distance(points)**

- **mask_points(points)**

**class** `sfepy.mechanics.contact_bodies.ContactSphere(centre, radius)`

- **get_distance(points)**

  Get the penetration distance and normals of points w.r.t. the sphere surface.

  **Returns**
The penetration distance.

The normals from the points to the sphere centre.

`mask_points(points, eps)`

`sfePy.mechanics.contact_bodies.plot_points(ax, points, marker, **kwargs)`

`sfePy.mechanics.contact_bodies.plot_polygon(ax, polygon)`

`sfePy.mechanics.elastic_constants module`

`sfePy.mechanics.matcoefs module`

Conversion of material parameters and other utilities.

```python
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.

\[
\gamma = \lambda + \frac{2\mu}{3}
\]

**sfepy.mechanics.matcoefs.bulk_from_youngpoisson(young, poisson, plane='strain')**

Compute bulk modulus corresponding to Young’s modulus and Poisson’s ratio.

**sfepy.mechanics.matcoefs.lame_from_stiffness(stiffness, plane='strain')**

Compute Lamé parameters from an isotropic stiffness tensor.

**sfepy.mechanics.matcoefs.lame_from_youngpoisson(young, poisson, plane='strain')**

Compute Lamé parameters from Young’s modulus and Poisson’s ratio.

The relationship between Lamé parameters and Young’s modulus, Poisson’s ratio (see [1],[2]):

\[
\lambda = \frac{\nu E}{(1 + \nu)(1 - 2\nu)}, \quad \mu = \frac{E}{2(1 + \nu)}
\]

The plain stress hypothesis:

\[
\bar{\lambda} = \frac{2\lambda\mu}{\lambda + 2\mu}
\]


sfepy.mechanics.matcoefs.stiffness_from_lame \((dim, lam, mu)\)

Compute stiffness tensor corresponding to Lamé parameters.

\[
D_{(2D)} = \begin{bmatrix}
\lambda + 2\mu & \lambda & 0 \\
\lambda & \lambda + 2\mu & 0 \\
0 & 0 & \mu \\
\end{bmatrix}
\]

\[
D_{(3D)} = \begin{bmatrix}
\lambda + 2\mu & \lambda & \lambda & 0 & 0 & 0 \\
\lambda & \lambda + 2\mu & \lambda & 0 & 0 & 0 \\
\lambda & \lambda & \lambda + 2\mu & 0 & 0 & 0 \\
0 & 0 & 0 & \mu & 0 & 0 \\
0 & 0 & 0 & 0 & \mu & 0 \\
0 & 0 & 0 & 0 & 0 & \mu \\
\end{bmatrix}
\]

sfepy.mechanics.matcoefs.stiffness_from_lame_mixed \((dim, lam, mu)\)

Compute stiffness tensor corresponding to Lamé parameters for mixed formulation.

\[
D_{(2D)} = \begin{bmatrix}
\tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 \\
\tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 \\
0 & 0 & \mu \\
\end{bmatrix}
\]

\[
D_{(3D)} = \begin{bmatrix}
\tilde{\lambda} + 2\mu & \tilde{\lambda} & \tilde{\lambda} & 0 & 0 & 0 \\
\tilde{\lambda} & \tilde{\lambda} + 2\mu & \tilde{\lambda} & 0 & 0 & 0 \\
\tilde{\lambda} & \tilde{\lambda} & \tilde{\lambda} + 2\mu & 0 & 0 & 0 \\
0 & 0 & 0 & \mu & 0 & 0 \\
0 & 0 & 0 & 0 & \mu & 0 \\
0 & 0 & 0 & 0 & 0 & \mu \\
\end{bmatrix}
\]

where

\[
\tilde{\lambda} = -\frac{2}{3}\mu
\]

sfepy.mechanics.matcoefs.stiffness_from_youngpoisson \((dim, young, poisson, plane='strain')\)

Compute stiffness tensor corresponding to Young’s modulus and Poisson’s ratio.

sfepy.mechanics.matcoefs.stiffness_from_youngpoisson_mixed \((dim, young, poisson, plane='strain')\)

Compute stiffness tensor corresponding to Young’s modulus and Poisson’s ratio for mixed formulation.

sfepy.mechanics.matcoefs.wave_speeds_from_youngpoisson \((young, poisson, rho)\)

Compute the P- and S-wave speeds from the Young’s modulus \(E\) and Poisson’s ratio \(\nu\) in a homogeneous isotropic material.

\[
v_p^2 = \frac{E(1-\nu)}{\rho(1+\nu)(1-2\nu)} = \frac{(\lambda + 2\mu)}{\rho}
\]

\[
v_s^2 = \frac{E}{2\rho(1+\nu)} = \frac{\mu}{\rho}
\]

Parameters

- **young**
  - [float or array] The Young’s modulus.

- **poisson**
  - [float or array] The Poisson’s ratio.
rho
[float or array] The density.

Returns

vp
[float or array] The P-wave speed.

vs
[float or array] The S-wave speed.

sfepy.mechanics.matcoefs.youngpoisson_from_stiffness(stiffness, plane='strain')
Compute Young’s modulus and Poisson’s ratio from an isotropic stiffness tensor.

sfepy.mechanics.matcoefs.youngpoisson_from_wave_speeds(vp, vs, rho)
Compute the Young’s modulus $E$ and Poisson’s ratio $\nu$ from the P- and S-wave speeds in a homogeneous isotropic material.

$E = \frac{\rho v_p^2 (3v_p^2 - 4v_s^2)}{(v_p^2 - v_s^2)}$

$\nu = \frac{(v_p^2/2 - v_s^2)}{(v_p^2 - v_s^2)}$

Parameters

vp
[float or array] The P-wave speed.

vs
[float or array] The S-wave speed.

rho
[float or array] The density.

Returns

young
[float or array] The Young’s modulus.

poisson
[float or array] The Poisson’s ratio.

sfepy.mechanics.membranes module

sfepy.mechanics.membranes.create_mapping(coors, gel, order)
Create mapping from transformed (in x-y plane) element faces to reference element faces.

Parameters

coors
[array] The transformed coordinates of element nodes, shape (n_el, n_ep, dim). The function verifies that all z components are zero.

gel
[GeometryElement instance] The geometry element corresponding to the faces.

order
[int] The polynomial order of the mapping.

Returns
mapping
[FEMapping instance] The reference element face mapping.

`sfePy.mechanics.membranes.create_transformation_matrix(coors)`
Create a transposed coordinate transformation matrix, that transforms 3D coordinates of element face nodes so that the transformed nodes are in the x-y plane. The rotation is performed w.r.t. the first node of each face.

**Parameters**

- **coors**
  - [array] The coordinates of element nodes, shape \((n_{el}, n_{ep}, \text{dim})\).

**Returns**

- **mtx_t**
  - [array] The transposed transformation matrix \(T\), i.e. \(X_{\text{inplane}} = T^T X_{3D}\).

**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 \((n_{el}, n_{ep}, \text{dim})\).

- **bfg**
  - [array] The in-plane base function gradients, shape \((n_{el}, n_{qp}, \text{dim}-1, n_{ep})\).

**Returns**

- **mtx_c ; array**
  - The in-plane right Cauchy-Green deformation tensor \(C_{ij}, i, j = 1, 2\).

- **c33**
  - [array] The component \(C_{33}\) computed from the incompressibility condition.

- **mtx_b**
  - [array] The discrete Green strain variation operator.

`sfePy.mechanics.membranes.describe_geometry(field, region, integral)`
Describe membrane geometry in a given region.

**Parameters**

- **field**
  - [Field instance] The field defining the FE approximation.

- **region**
  - [Region instance] The surface region to describe.

- **integral**
  - [Integral instance] The integral defining the quadrature points.

**Returns**
mtx_t
[array] The transposed transformation matrix $T$, see `create_transformation_matrix()`.

membrane_geo
[CMapping instance] The mapping from transformed elements to a reference elements.

`sfe.py.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 $(n_{el}, n_{qp}, \text{dim}-1, \text{dim}-1)$.

- **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 $(n_{el}, n_{qp}, \text{sym}, 1)$.

`sfe.py.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 $(n_{el}, n_{qp}, \text{dim}-1, \text{dim}-1)$.

- **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 $C_{ij}$.

`sfe.py.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 $(n_{el}, n_{qp}, \text{dim}-1, n_{ep})$.

**Returns**

- **mtx**
  [array] The tangent stress matrix, shape $(n_{el}, n_{qp}, \text{dim}^*n_{ep}, \text{dim}^*n_{ep})$. 

364 Chapter 2. Development
sfepy.mechanics.membranes.transform_asm_matrices(out, mtx_t)
Transform matrix assembling contributions to global coordinate system, one node at a time.

Parameters
- out
  [array] The array of matrices, transformed in-place.
- mtx_t
  [array] The transposed transformation matrix $T$, see create_transformation_matrix().

sfepy.mechanics.membranes.transform_asm_vectors(out, mtx_t)
Transform vector assembling contributions to global coordinate system, one node at a time.

Parameters
- out
  [array] The array of vectors, transformed in-place.
- mtx_t
  [array] The transposed transformation matrix $T$, see create_transformation_matrix().

sfepy.mechanics.shell10x module
Functions implementing the shell10x element.
sfepy.mechanics.shell10x.add_eas_dofs(mtx_b, qp_coors, det, det0, dxidx0)
Add additional strain components [Andelfinger and Ramm] (7 parameters to be condensed out).
sfepy.mechanics.shell10x.create_drl_transform(ebs)
Create the transformation matrix for locking of the drilling rotations.
sfepy.mechanics.shell10x.create_elastic_tensor(young, poisson, shear_correction=True)
Create the elastic tensor with the applied shear correction (the default) for the shell10x element.
sfepy.mechanics.shell10x.create_local_bases(coors)
Create local orthonormal bases in each vertex of quadrilateral cells.

Parameters
- coors
  [array] The coordinates of cell vertices, shape $(n_{el}, 4, 3)$.

Returns
ebs
  [array] The local bases, shape $(n_{el}, 4, 3, 3)$. The basis vectors are rows of the $(..., 3, 3)$ blocks.
sfepy.mechanics.shell10x.create_rotation_ops(ebs)
Create operators associated to rotation DOFs.

Parameters
ebs
  [array] The local bases, shape $(n_{el}, 4, 3, 3)$.

Returns
- rops
  [array] The rotation operators, shape $(n_{el}, 4, 3, 3)$.
sfepy.mechanics.shell10x.create_strain_matrix(bfgm, dxidx, dsg)
Create the strain operator matrix.
sfepy.mechanics.shell10x.create_strain_transform(mtx_ts)
Create strain tensor transformation matrices, given coordinate transformation matrices.

Notes
Expresses $TET^T$ in terms of symmetrix storage as $Qe$, with the ordering of components: $e = [e_{11}, e_{22}, e_{33}, 2e_{12}, 2e_{13}, 2e_{23}]$.
sfepy.mechanics.shell10x.create_transformation_matrix(coors)
Create a transposed coordinate transformation matrix, that transforms 3D coordinates of quadrilateral cell vertices so that the transformed vertices of a plane cell are in the $x - y$ plane. The rotation is performed w.r.t. the centres of quadrilaterals.

Parameters
coors
[array] The coordinates of cell vertices, shape $(n_{el}, 4, 3)$.

Returns
mtx_t
[array] The transposed transformation matrix $T$, i.e. $X_{inplane} = T^T X_{3D}$.

Notes
$T = [t_1, t_2, n]$, where $t_1, t_2$, are unit in-plane (column) vectors and $n$ is the unit normal vector, all mutually orthonormal.
sfepy.mechanics.shell10x.get_dsg_strain(coors_loc, qp_coors)
Compute DSG strain components.

Returns
dsg
[array] The strain matrix components corresponding to $e_{13}, e_{23}$, shape $(n_{el}, n_{qp}, 2, 24)$.

Notes
Involves $w, \alpha, \beta$ DOFs.
sfepy.mechanics.shell10x.get_mapping_data(ebs, rops, ps, coors_loc, qp_coors, qp_weights, special_dx3=False)
Compute reference element mapping data for shell10x elements.
Notes

The code assumes that the quadrature points are w.r.t. \((t = \text{thickness of the shell}) [0, 1] \times [0, 1] \times [-t/2, t/2]\) reference cell and the quadrature weights are multiplied by \(t\).

```python
sfepy.mechanics.shell10x.lock_drilling_rotations(mtx, ebs, coefs)
```

Lock the drilling rotations in the stiffness matrix.

```python
sfepy.mechanics.shell10x.rotate_elastic_tensor(mtx_d, bfu, ebs)
```

Rotate the elastic tensor into the local coordinate system of each cell. The local coordinate system results from interpolation of \(ebs\) with the bilinear basis.

```python
sfepy.mechanics.shell10x.transform_asm_matrices(out, mtx_t, blocks)
```

Transform matrix assembling contributions to global coordinate system, one node at a time.

Parameters

- `mtx_t` [array] The array of transposed transformation matrices \(T\), see `create_transformation_matrix()`.
- `blocks` [array] The DOF blocks that are

**sfepy.mechanics.tensors module**

Functions to compute some tensor-related quantities usual in continuum mechanics.

```python
class sfepy.mechanics.tensors.StressTransform(def_grad, jacobian=None)
```

Encapsulates functions to convert various stress tensors in the symmetric storage given the deformation state.

```python
get_cauchy_from_2pk(stress_in)
```

Get the Cauchy stress given the second Piola-Kirchhoff stress.

\[
\sigma_{ij} = J^{-1} F_{ik} S_{kl} F_{jl}
\]

```python
sfepy.mechanics.tensors.dim2sym(dim)
```

Given the space dimension, return the symmetric storage size.

```python
sfepy.mechanics.tensors.get_cauchy_strain(grad)
```

Given a gradient, return the corresponding Cauchy strain (symmetric gradient).

```python
sfepy.mechanics.tensors.get_deviator(tensor, sym_storage=True)
```

The deviatoric part (deviator) of a tensor.

```python
sfepy.mechanics.tensors.get_full_indices(dim)
```

The indices for converting the symmetric storage to the full storage.

```python
sfepy.mechanics.tensors.get_non_diagonal_indices(dim)
```

The non_diagonal indices for the full vector storage.

```python
sfepy.mechanics.tensors.get_sym_indices(dim)
```

The indices for converting the full storage to the symmetric storage.
sfepy.mechanics.tensors.get_t4_from_t2s(t2s)

Get the full 4D tensor with major/minor symmetries from its 2D matrix representation.

**Parameters**

- **t2s**
  - [array] The symmetrically-stored tensor of shape (S, S), where S is the symmetric storage size.

**Returns**

- **t4**
  - [array] The full 4D tensor of shape (D, D, D, D), where D is the space dimension.

sfepy.mechanics.tensors.get_trace(tensor, sym_storage=True)

The trace of a tensor.

sfepy.mechanics.tensors.get_volumetric_tensor(tensor, sym_storage=True)

The volumetric part of a tensor.

sfepy.mechanics.tensors.get_von_mises_stress(stress, sym_storage=True)

Given a symmetric stress tensor, compute the von Mises stress (also known as Equivalent tensile stress).

**Notes**

\[
\sigma_V = \sqrt{\frac{(\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{33})^2 + (\sigma_{11} - \sigma_{33})^2 + 6(\sigma_{12}^2 + \sigma_{13}^2 + \sigma_{23}^2)}}{2}}
\]

sfepy.mechanics.tensors.prepare_cylindrical_transform(coors, origin, mode='axes')

Prepare matrices for transforming tensors into cylindrical coordinates with the axis ‘z’ in a given origin.

**Parameters**

- **coors**
  - [array] The Cartesian coordinates.

- **origin**
  - [array of length 3] The origin.

- **mode**
  - ['axes' or 'data'] In ‘axes’ (default) mode the matrix transforms data to different coordinate system, while in ‘data’ mode the matrix transforms the data in the same coordinate system and is transpose of the matrix in the ‘axes’ mode.

**Returns**

- **mtx**
  - [array] The array of transformation matrices for each coordinate in coors.

sfepy.mechanics.tensors.sym2dim(sym)

Given the symmetric storage size, return the space dimension.
Notes

This function works for any space dimension.

```
sfepy.mechanics.tensors.transform_data(data, coors=None, mode='cylindrical', mtx=None)
```

Transform vector or tensor data components between orthogonal coordinate systems in 3D using transformation matrix $M$, that should express rotation of the original coordinate system to the new system denoted by \( \bullet' \) below.

For vectors:

$$ v' = M \cdot v $$

For second order tensors:

$$ t'_{ij} = M_{ip} M_{jq} t_{pq} $$

or

$$ t' = M \cdot t \cdot M^T $$

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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> q2('m')
'1000000.0 mPa'
>>> q2('')
'1000.0 Pa'
>>> q2('k')
'1.0 kPa'
>>> q2('M')
'0.001 MPa'
```

class sfepy.mechanics.units.Unit(name)
A unit of a physical quantity. The prefix and coefficient of the unit are determined from to its name.

Examples

Construct some units:

```python
>>> from sfepy.mechanics.units import Unit
>>> unit = Unit('mm')
>>> print unit
Unit:mm
coeff:
```

0.001
name:
   mm
prefix:
   m
prefix_length:
   1
unit:
   m
>>> unit = Unit('kg')
>>> print unit
Unit:kg
c coef:
   1000.0
name:
   kg
prefix:
   k
prefix_length:
   1
unit:
   g

Get prefixes for a coefficient:

```python
>>> Unit.get_prefix(100.0)
('d', 10.0)
>>> Unit.get_prefix(100.0, omit=('d',))
('k', 0.10000000000000001)
```

**static get_prefix**(coef, bias=0.1, omit=None)

Get the prefix and numerical multiplier corresponding to a numerical coefficient, omitting prefixes in omit tuple.

**sfepy.mechanics.units.apply_unit_multipliers**(values, unit_kinds, unit_multipliers)

Apply time, length and mass unit multipliers to given values with units corresponding to unit kinds.

**Returns**

```
new_values [list] The new values with applied unit multipliers
```

**sfepy.mechanics.units.apply_units_to_pars**(pars, pars_kinds, unit_multipliers)

Apply units in unit_multipliers to pars according to their kinds.

**Parameters**

```
pars [dict] The input parameters given as name : value items.

pars_kinds [dict] The kinds of the parameters given as name : kind items, with kinds defined in apply_unit_multipliers().

unit_multipliers [tuple] The time, length and mass unit multipliers.
```
Returns

```
new_pars
```

[dict] The output parameters.

```python
def get_consistent_unit_set(length=None, time=None, mass=None, temperature=None)
```

Given a set of basic units, return a consistent set of derived units for quantities listed in the units_of_quantities dictionary.

**sfepy.mechanics.extmods.ccontres module**

```python
def assemble_contact_residual_and_stiffness()
```

```python
def evaluate_contact_constraints()
```

```python
def get_AABB()
```

```python
def get_longest_edge_and_gps()
```

```python
def init_global_search()
```

The linked list initialization. The head array contains, at the position Ic, the index of the first point that belongs to the cell Ic, the second point index is then next[head[Ic]], the third point index is next[next[head[Ic]]] etc. - the next array points from the i-th point in each cell to the (i+1)-th point, until -1 is reached.

**sfepy.mesh package**

**sfepy.mesh.bspline module**

```python
class BSpline(degree=3, is_cyclic=False, ncp=0)
```

B-spline curve representation

```
def approximate(coors, ncp=None, knot_type='clamped', knots=None, alpha=0.5, do_eval=False, do_param_correction=False)
```

Approximate set of points by the B-spline curve.

**Parameters**

- coors
  - [array] The coordinates of the approximated points.
- ncp
  - [int] The number of control points.
- knot_type
  - [str] The knot vector type.
- knots
  - [array] The knot vector.
- alpha
  - [float]

**The parameter vector distribution:**

- 1.0 = chordal
- 0.5 = centripetal
- do_eval
  - [bool] Evaluate the curve coordinates?
do_param_correction
[bool] Perform parametric corrections to improve the approximation?

static basis_function_dg(degree, t, knots, n)
B-spline basis functions.

Parameters

degree
[int] The degree of the spline function.

t
[array] The parametric vector.

knots
[array] The knot vector.

n
[int] The number of intervals.

Returns

bfun
[array] The spline basis function evaluated for given values.

static basis_function_dg0(t, knots, n)
Basis function: degree = 0

Parameters

t
[array] The parametric vector.

knots
[array] The knot vector.

n
[int] The number of intervals.

Returns

bfun
[array] The spline basis function evaluated for given values.

draw(ret_ax=False, ax=None, color='r', cp_id=True)
Draw B-spline curve.

Parameters

ret_ax
[bool] Return an axes object?

ax
[axes object] The axes to which will be drawn.

color
[str] Line color.

cp_id
[bool] If True, label control points.

draw_basis()
Draw B-spline curve.
eval\( (t=None, cp\_coors=None) \)
Evaluate the coordinates of the bpsline curve.

Parameters
- \( t \) [array] The parameter vector of the B-spline.
- \( cp\_coors \) [array] The coordinates of the control points.

eval\_basis\( (t=None, return\_val=False) \)
Evaluate the basis of the bpsline.

Parameters
- \( t \) [array] The parameter vector of the B-spline.

get\_control\_points()
Get the B-spline control points.

Returns
- \( coors \) [array] The coordinates of control points.

get\_knot\_vector()
Return the knot vector.

Returns
- \( knots \) [array] The knot vector.

insert\_knot\( (new) \)
Insert a new knot into the knot vector.

Parameters
- \( new \) [float] The new knot value.

make\_knot\_vector\( (knot\_type='clamped', knot\_data=None, knot\_range=(0.0, 1.0)) \)
Create a knot vector of the requested type.

Parameters
- \( knot\_type \) [str] The knot vector type: clamped/cyclic/userdef.
- \( knot\_data \) : The extra knot data.

set\_approx\_points\( (coors) \)
Set the coordinates of approximated points.

Parameters
- \( coors \) [array] The coordinates of approximated points.
set_control_points(coors, cyclic_form=False)

Set the B-spline control points.

Parameters

coors
[array] The coordinates of unique control points.

cyclic_form
[bool] Are the control points in the cyclic form?

set_knot_vector(knots)

Set the knot vector.

Parameters

knots
[array] The knot vector.

set_param(t)

Set the B-spline parametric vector.

Parameters

t
[array] The parameter vector of the B-spline.

set_param_n(n=100, knot_range=(0.0, 1.0))

Generate the B-spline parametric vector using the number of steps.

Parameters

n
[array] The number of steps in the B-spline parametric vector.

class sfepy.mesh.bspline.BSplineSurf(degree=(3, 3), is_cyclic=(False, False))

B-spline surface representation

approximate(coors, ncp, do_eval=False)

Approximate set of points by the B-spline surface.

Parameters

coors
[array] The coordinates of the approximated points.

ncp
[tuple of int] The number of control points.

draw(ret_ax=False, ax=None)

Draw B-spline surface.

Parameters

ret_ax
[bool] Return an axes object?

ax
[axes object] The axes to which will be drawn.

eval(t=(None, None), cp_coors=None)

Evaluate the coordinates of the bpsline curve.

Parameters
t
[tuple of array] The parametric vector of the B-splines.

cp_coors
[array] The coordinates of the control points.

get_control_points()
Get the B-spline surface control points.

Returns

coops
[array] The coordinates of control points.

make_knot_vector(knot_type=('clamped', 'clamped'), knot_data=(None, None))
Create a knot vector of the requested type.

Parameters

c knot_type
[tuple of str] The knot vector types.

c knot_data
[tuple of ANY] The extra knot data.

set_approx_points(coors)
Set the coordinates of approximated points.

Parameters

c coors
[array] The coordinates of approximated points.

set_control_points(coors, cyclic_form=False)
Set the B-spline control points.

Parameters

c coors
[array] The coordinates of unique control points.

c cyclic_form
[bool] Are the control points in the cyclic form?

set_param_n(n=(100, 100))
Generate the B-spline parametric vector using the number of steps.

Parameters

c n
[tuple of array] The number of steps in the B-spline parametric vectors.

write_control_polygon_vtk(filename, float_format="%.6f")
Write the control polygon to VTK file.

Parameters

c filename: str
    Name of the VTK file.

c float_format: str
    Float formatting.
write_surface_vtk(filename, float_format="%.6f")

Write the spline surface to VTK file.

Parameters

filename: str
   Name of the VTK file.

float_format: str
   Float formatting.

sfepy.mesh.bspline.approximation_example()

The example of using BSplineSurf for approximation of the surface given by the set of points.

sfepy.mesh.bspline.get_2d_points(is3d=False)

Returns the set of points.

Parameters

is3d
   [bool] 3D coordinates?

sfepy.mesh.bspline.main(argv)

sfepy.mesh.bspline.simple_example()

The example of using B-spline class.

sfepy.mesh.bspline.to_ndarray(a)

sfepy.mesh.geom_tools module

class sfepy.mesh.geom_tools.geometry(dim=3)

The geometry is given by a sets of points (d0), lines (d1), surfaces (d2) and volumes (d3). A lines are constructed from 2 points, a surface from any number of lines, a volume from any number of surfaces.

Physical volumes are contruted from any number of volumes.

The self.d0, self.d1, self.d2 and self.d3 are dictionaries holding a map

geometry element number -> instance of point, line, surface of volume

Examples

To get all the points which define a surface 5, use:

self.d2[5].getpoints()

This would give you a list [...] of point() instances.

addline(n, l)
   l=[p1,p2]

addlines(ls, off=1)
   ls=[[l1, l2, ...]]

addphysicalsurface(n, surfacelist)
   surfacelist=[s1,s2,s3,...]
addphysicalvolume(n, volumelist)
    volumelist=[v1,v2,v3,...]
addpoint(n, p)
    p=[x,y,z]
addpoints(ps, off=1)
    ps=[p1, p2, ...]
addsurface(n, s, is_hole=False)
    s=[l1,l2,l3,...]
addsurfaces(ss, off=1)
    s=[s1,s2,s3,...]
addvolume(n, v)
    v=[s1,s2,s3,...]
addvolumes(vs, off=1)
    v=[v1,v2,v3,...]

static from_gmsh_file(filename)
    Import geometry - Gmsh geometry format.
        Parameters
        filename
            [string] file name
        Returns
        geo
            [geometry] geometry description
getBCnum(snum)
leaveonlyphysicalsurfaces()
leaveonlyphysicalvolumes()
printinfo(verboset=on)
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.meshgeom_tools.geomobject
getn()

class sfepy.meshgeom_tools.line(g, n, l)
.. py:class:: sfepy.mesh.geom_tools.physicalsurface(g, n, s)

    :func:`getpoints`

.. py:class:: sfepy.mesh.geom_tools.physicalvolume(g, n, v)

    :func:`getvolumes`

.. py:class:: sfepy.mesh.geom_tools.point(g, n, p)

    :func:`getstr`
    :func:`getxyz`

.. py:class:: sfepy.mesh.geom_tools.surface(g, n, s, is_hole=False)

    :func:`getcenterpoint`
    :func:`getholepoints`
    :func:`getinsidepoint`
    :func:`getlines`
    :func:`getpoints`
    :func:`separate(s)`

.. py:class:: sfepy.mesh.geom_tools.volume(g, n, v)

    :func:`getinsidepoint`
    :func:`getsurfaces`

---

.. py:module:: sfepy.mesh.mesh_generators

**sfepy.mesh.mesh_generators.gen_block_mesh**

.. py:func:: sfepy.mesh.mesh_generators.gen_block_mesh(dims, shape, centre, mat_id=0, name='block', coors=None, verbose=True)

    Generate a 2D or 3D block mesh. The dimension is determined by the length of the shape argument.

    Parameters

    .. py:attr:: dims
        .. py:obj:: array of 2 or 3 floats
            Dimensions of the block.

    .. py:attr:: shape
        .. py:obj:: array of 2 or 3 ints
            Shape (counts of nodes in x, y, z) of the block mesh.

    .. py:attr:: centre
        .. py:obj:: array of 2 or 3 floats
            Centre of the block.

    .. py:attr:: mat_id
        .. py:obj:: int, optional
            The material id of all elements.

    .. py:attr:: name
        .. py:obj:: string
            Mesh name.

    .. py:attr:: verbose
        .. py:obj:: bool
            If True, show progress of the mesh generation.
Returns

mesh
[Mesh instance]

`sfepy.mesh.mesh_generators.gen_cylinder_mesh(dims, shape, centre, axis='x', force_hollow=False,
       is_open=False, open_angle=0.0, non_uniform=False,
       name='cylinder', verbose=True)`

Generate a cylindrical mesh along an axis. Its cross-section can be ellipsoidal.

Parameters

- **dims**
  [array of 5 floats] Dimensions of the cylinder: inner surface semi-axes a1, b1, outer surface
  semi-axes a2, b2, length.

- **shape**
  [array of 3 ints] Shape (counts of nodes in radial, circumferential and longitudinal directions)
  of the cylinder mesh.

- **centre**
  [array of 3 floats] Centre of the cylinder.

- **axis**: one of ‘x’, ‘y’, ‘z’
  The axis of the cylinder.

- **force_hollow**
  [boolean] Force hollow mesh even if inner radii a1 = b1 = 0.

- **is_open**
  [boolean] Generate an open cylinder segment.

- **open_angle**
  [float] Opening angle in radians.

- **non_uniform**
  [boolean] If True, space the mesh nodes in radial direction so that the element volumes are
  (approximately) the same, making thus the elements towards the outer surface thinner.

- **name**
  [string] Mesh name.

- **verbose**
  [bool] If True, show progress of the mesh generation.

Returns

mesh
[Mesh instance]

`sfepy.mesh.mesh_generators.gen_extended_block_mesh(b_dims, b_shape, e_dims, e_shape, centre,
       grading_fun=None, name=None)`

Generate a 3D mesh with a central block and (coarse) extending side meshes.

The resulting mesh is again a block. Each of the components has a different material id.

Parameters

- **b_dims**
  [array of 3 floats] The dimensions of the central block.

- **b_shape**
  [array of 3 ints] The shape (counts of nodes in x, y, z) of the central block mesh.
e_dims
[array of 3 floats] The dimensions of the complete block (central block + extensions).

e_shape
[int] The count of nodes of extending blocks in the direction from the central block.

centre
[array of 3 floats] The centre of the mesh.

grading_fun
[callable, optional] A function of \( x \in [0, 1] \) that can be used to shift nodes in the extension axis directions to allow smooth grading of element sizes from the centre. The default function is \( x ^ {**p} \) with \( p \) determined so that the element sizes next to the central block have the size of the shortest edge of the central block.

name
[string, optional] The mesh name.

Returns
mesh
[Mesh instance]

```
sfepy.mesh.mesh_generators.gen_mesh_from_geom(geo, a=None, verbose=False, refine=False)
```
Runs mesh generator - tetgen for 3D or triangle for 2D meshes.

Parameters
geo
[geometry] geometry description

a
[int, optional] a maximum area/volume constraint

verbose
[bool, optional] detailed information

refine
[bool, optional] refines mesh

Returns
mesh
[Mesh instance] triangular or tetrahedral mesh

```
sfepy.mesh.mesh_generators.gen_mesh_from_string(mesh_name, mesh_dir)
sfepy.mesh.mesh_generators.gen_mesh_from_voxels(voxels, dims, etype='q')
```
Generate FE mesh from voxels (volumetric data).

Parameters
voxels
[array] Voxel matrix, 1=material.

dims
[array] Size of one voxel.

etype
[integer, optional] ‘q’ - quadrilateral or hexahedral elements ‘t’ - triangular or tetrahedral elements
Returns

---

mesh

<Mesh instance> Finite element mesh.

`sfepy.mesh.mesh_generators.gen_misc_mesh(mesh_dir, force_create, kind, args, suffix='mesh', verbose=False)`

Create sphere or cube mesh according to `kind` in the given directory if it does not exist and return path to it.

`sfepy.mesh.mesh_generators.gen_tiled_mesh(mesh, grid=None, scale=1.0, eps=1e-06, ret_ndmap=False)`

Generate a new mesh by repeating a given periodic element along each axis.

Parameters

- **mesh**
  
  <Mesh instance> The input periodic FE mesh.

- **grid**
  
  [array] Number of repetition along each axis.

- **scale**
  
  [float, optional] Scaling factor.

- **eps**
  
  [float, optional] Tolerance for boundary detection.

- **ret_ndmap**
  
  [bool, optional] If True, return global node map.

Returns

- **mesh_out**
  
  <Mesh instance> FE mesh.

- **ndmap**
  

`sfepy.mesh.mesh_generators.get_tensor_product_conn(shape)`

Generate vertex connectivity for cells of a tensor-product mesh of the given shape.

Parameters

- **shape**
  
  [array of 2 or 3 ints] Shape (counts of nodes in x, y, z) of the mesh.

Returns

- **conn**
  
  [array] The vertex connectivity array.

- **desc**
  
  [str] The cell kind.

`sfepy.mesh.mesh_generators.main()`

`sfepy.mesh.mesh_generators.tiled_mesh1d(conn, coors, ngrps, idim, n_rep, bb, eps=1e-06, ndmap=False)`
**sfepy.mesh.mesh_tools module**

**sfepy.mesh.mesh_tools.elems_q2t(el)**

**sfepy.mesh.mesh_tools.expand2d(mesh2d, dist, rep)**

Expand 2D planar mesh into 3D volume, convert triangular/quad mesh to tetrahedrons/hexahedrons.

**Parameters**

- **mesh2d**  
  [Mesh] The 2D mesh.

- **dist**  
  [float] The elements size in the 3rd direction.

- **rep**  
  [int] The number of elements in the 3rd direction.

**Returns**

- **mesh3d**  
  [Mesh] The 3D mesh.

**sfepy.mesh.mesh_tools.extract_edges(mesh, eps=1e-16)**

Extract outline edges of a given mesh. The outline edge is an edge for which \(\text{norm}(nvec_1 - nvec_2) < \varepsilon\), where \(nvec_1\) and \(nvec_2\) are the normal vectors of the incident facets.

**Parameters**

- **mesh**  
  [Mesh] The 3D or 2D mesh.

- **eps**  
  [float] The tolerance parameter of the outline edge searching algorithm.

**Returns**

- **mesh_out**  
  [tuple] The data of the outline mesh, Mesh.from_data() format, i.e. (coors, ngroups, ed_conns, mat_ids, descs).

**sfepy.mesh.mesh_tools.get_surface_faces(domain)**

**sfepy.mesh.mesh_tools.merge_lines(mesh, eps=1e-18)**

Merge edges of an edge-only mesh that are in the same direction w.r.t. the tolerance \(\varepsilon\).

**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.

**Parameters**

- **mesh**  
  [mesh] FE mesh.

- **n_iter**  
  [integer, optional] Number of iteration steps.

**Based on:**

lam
[float, optional] Smoothing factor, see [1].

mu
[float, optional] Unshrinking factor, see [1].

weights
[array, optional] Edge weights, see [1].

bconstr: logical, optional
Boundary constraints, if True only surface smoothing performed.

volume_corr: logical, optional
Correct volume after smoothing process.

Returns

coors
[array] Coordinates of mesh nodes.

sfepy.mesh.mesh_tools.surface_components(gr_s, surf_faces)
Determine surface components given surface mesh connectivity graph.

sfepy.mesh.mesh_tools.surface_graph(surf_faces, n_nod)

sfepy.mesh.mesh_tools.triangulate(mesh, verbose=False)
Triangulate a 2D or 3D tensor product mesh: quadrilaterals->triangles, hexahedrons->tetrahedrons.

Parameters

mesh
[Mesh] The input mesh.

Returns

mesh
[Mesh] The triangulated mesh.

sfepy.mesh.splinebox module

class sfepy.mesh.splinebox.SplineBox(bbox, coors, nsg=None, field=None)
B-spline geometry parametrization. The geometry can be modified by moving spline control points.

static create_spb(bbox, coors, degree=3, nsg=None)

evaluate(cp_values=None, outside=True)
Evaluate the new position of the mesh coordinates.

Parameters

cp_values
[array] The actual control point values. If None, use self.control_values.

outside
[bool] If True, return also the coordinates outside the spline box.

Returns

new_coors
[array] The new position of the mesh coordinates.
**evaluate_derivative**(*cpoint, dirvec*)

Evaluate derivative of the spline in a given control point and direction.

**Parameters**

- **cpoint**
  [int, list] The position (index or grid indicies) of the spline control point.

- **dirvec**
  [array] The directional vector.

**Returns**

- **diff**
  [array] The derivative field.

**static gen_cp_ids**(*ncp*)

**get_box_matrix**()

Returns:

- **mtx**
  [2D array] The matrix containing the coefficients of b-spline basis functions.

**get_control_points**(init=False)

Get the spline control points coordinates.

**Returns**

- **cpt_coors**
  [array] The coordinates of the spline control points.

- **init**
  [bool] If True, return the initial state.

**get_coors_shape**()

Get the shape of the coordinates.

**move_control_point**(*cpoint, val*)

Change shape of spline parametrization.

**Parameters**

- **cpoint**
  [int, list] The position (index or grid indicies) of the spline control point.

- **val**
  [array] Displacement.

**set_control_points**(*cpt_coors, add=False*)

Set the spline control points position.

**Parameters**

- **cpt_coors**
  [array] The coordinates of the spline control points.

- **add**
  [bool] If True, coors += cpt_coors
write_control_net(filename, deform_by_values=True)
Write the SplineBox shape to the VTK file.

Parameters
filename
[str] The VTK file name.

class sfepy.mesh.splinebox.SplineRegion2D(spl_bnd, coors, rho=1000.0)
B-spline geometry parametrization. The boundary of the SplineRegion2D is defined by BSpline curves.

static create_spb(spl_bnd, coors, rho=10)
Initialize SplineBox knots, control points, base functions, ...

static define_control_points(cp_bnd_coors, ncp)
Find positions of “inner” control points depending on boundary splines.

find_ts(coors)
Function finds parameters (t, s) corresponding to given points (coors).

static points_in_poly(points, poly, tol=1e-06)
Find which points are located inside the polygon.

sfepy.parallel package

sfepy.parallel.evaluate module

PETSc-related parallel evaluation of problem equations.

class sfepy.parallel.evaluate.PETScParallelEvaluator(problem, pdofs, drange, is_overlap, psol, comm, matrix_hook=None, verbose=False)
The parallel evaluator of the problem equations for PETScNonlinearSolver.
Its methods can be used as the function and Jacobian callbacks of the PETSc SNES (Scalable Nonlinear Equations Solvers).

Notes
Assumes problem.active_only == False.

eval_residual(snes, psol, prhs)
eval_tangent_matrix(snes, psol, pmtx, ppmtx)

sfepy.parallel.parallel module

Functions for a high-level PETSc-based parallelization.

sfepy.parallel.parallel.assemble_mtx_to_petsc(pmtx, mtx, pdofs, drange, is_overlap=True, comm=None, verbose=False)
Assemble a local CSR matrix to a global PETSc matrix.

sfepy.parallel.parallel.assemble_rhs_to_petsc(prhs, rhs, pdofs, drange, is_overlap=True, comm=None, verbose=False)
Assemble a local right-hand side vector to a global PETSc vector.
sfepy.parallel.parallel.call_in_rank_order(fun, comm=None)
Call a function fun task by task in the task rank order.

sfepy.parallel.parallel.create_gather_scatter(pdofs, pvec_i, pvec, comm=None)
Create the gather() function for updating a global PETSc vector from local ones and the scatter() function for updating local PETSc vectors from the global one.

sfepy.parallel.parallel.create_gather_to_zero(pvec)
Create the gather_to_zero() function for collecting the global PETSc vector on the task of rank zero.

sfepy.parallel.parallel.create_local_petsc_vector(pdofs)
Create a local PETSc vector with the size corresponding to pdofs.

sfepy.parallel.parallel.create_petsc_matrix(sizes, mtx_prealloc=None, comm=None)
Create and allocate a PETSc matrix.

sfepy.parallel.parallel.create_petsc_system(mtx, sizes, pdofs, drange, is_overlap=True, comm=None, verbose=False)
Create and pre-allocate (if is_overlap is True) a PETSc matrix and related solution and right-hand side vectors.

sfepy.parallel.parallel.create_prealloc_data(mtx, pdofs, drange, verbose=False)
Create CSR preallocation data for a PETSc matrix based on the owned PETSc DOFs and a local matrix with EBCs not applied.

sfepy.parallel.parallel.create_task_dof_maps(field, cell_tasks, inter_facets, is_overlap=True, use_expand_dofs=False, save_inter_regions=False, output_dir=None)
Foreach task list its inner and interface DOFs of the given field and create PETSc numbering that is consecutive in each subdomain.

For each task, the DOF map has the following structure:

```
[inner,
 [own_inter1, own_inter2, ...],
 [overlap_cells1, overlap_cells2, ...],
 n_task_total, task_offset]
```

The overlapping cells are defined so that the system matrix corresponding to each task can be assembled independently, see [1]. TODO: Some “corner” cells may be added even if not needed - filter them out by using the PETSc DOFs range.

When debugging domain partitioning problems, it is advisable to set save_inter_regions to True to save the task interfaces as meshes as well as vertex-based markers - to be used only with moderate problems and small numbers of tasks.


sfepy.parallel.parallel.distribute_field_dofs(field, gfd, use_expand_dofs=False, comm=None, verbose=False)
Distribute the owned cells and DOFs of the given field to all tasks.

The DOFs use the PETSc ordering and are in form of a connectivity, so that each task can easily identify them with the DOFs of the original global ordering or local ordering.

sfepy.parallel.parallel.distribute_fields_dofs(fields, cell_tasks, is_overlap=True, use_expand_dofs=False, save_inter_regions=False, output_dir=None, comm=None, verbose=False)
Distribute the owned cells and DOFs of the given field to all tasks.

Uses interleaved PETSc numbering in each task, i.e., the PETSc DOFs of each tasks are consecutive and correspond to the first field DOFs block followed by the second etc.

Expand DOFs to equations if `use_expand_dofs` is True.

```
sfepy.parallel.parallel.expand_dofs(dofs, n_components)
```
Expand DOFs to equation numbers.

```
sfepy.parallel.parallel.get_composite_sizes(lfds)
```
Get (local, total) sizes of a vector and local equation range for a composite matrix built from field blocks described by `lfds` local field distributions information.

```
sfepy.parallel.parallel.get_inter_facets(domain, cell_tasks)
```
For each couple of neighboring task subdomains get the common boundary (interface) facets.

```
sfepy.parallel.parallel.get_local_ordering(field_i, petsc_dofs_conn, use_expand_dofs=False)
```
Get PETSc DOFs in the order of local DOFs of the localized field `field_i`.

Expand DOFs to equations if `use_expand_dofs` is True.

```
sfepy.parallel.parallel.get_sizes(petsc_dofs_range, n_dof, n_components)
```
Get (local, total) sizes of a vector and local equation range.

```
sfepy.parallel.parallel.init_petsc_args()
```

```
sfepy.parallel.parallel.partition_mesh(mesh, n_parts, usemetis=True, verbose=False)
```
Partition the mesh cells into `n_parts` subdomains, using metis, if available.

```
sfepy.parallel.parallel.setup_composite_dofs(lfds, fields, local_variables, verbose=False)
```
Setup composite DOFs built from field blocks described by `lfds` local field distributions information.

Returns (local, total) sizes of a vector, local equation range for a composite matrix, and the local ordering of composite PETSc DOFs, corresponding to `local_variables` (must be in the order of `fields`).

```
sfepy.parallel.parallel.verify_task_dof_maps(dof_maps, id_map, field, use_expand_dofs=False, verbose=False)
```
Verify the counts and values of DOFs in `dof_maps` and `id_map` corresponding to `field`.

Returns the vector with a task number for each DOF.

```
sfepy.parallel.parallel.view_petsc_local(data, name=’data’, viewer=None, comm=None)
```
View local PETSc data called `name`. The data object has to have `.view()` method.

**sfepy.parallel.plot_parallel_dofs module**

Functions to visualize the partitioning of a domain and a field DOFs.

```
sfepy.parallel.plot_parallel_dofs.label_dofs(ax, coors, dofs, colors)
```
Label DOFs using the given colors.

```
sfepy.parallel.plot_parallel_dofs.mark_subdomains(ax, cmesh, cell_tasks, size=None, icolor=0, alpha=1.0, mask=False)
```
Mark cells of subdomains corresponding to each task by a different color. Plots nothing in 3D.
sfepy.parallel.plot_parallel_dofs.plot_local_dofs(axs, field, field_i, omega_gi, output_dir, rank)
Plot the local and global field DOFs local to the subdomain on the task with the given rank.
sfepy.parallel.plot_parallel_dofs.plot_partitioning(axs, field, cell_tasks, gfd, output_dir, size)
Plot the partitioning of the domain and field DOFs.

**sfepy.postprocess package**

**sfepy.postprocess.plot_cmesh module**

Functions to visualize the CMesh geometry and topology.
sfepy.postprocess.plot_cmesh.label_global_entities(ax, cmesh, edim, color='b', fontsize=10, **kwargs)
Label mesh topology entities using global ids.
sfepy.postprocess.plot_cmesh.label_local_entities(ax, cmesh, edim, color='b', fontsize=10, **kwargs)
Label mesh topology entities using cell-local ids.
sfepy.postprocess.plot_cmesh.plot_cmesh(ax, cmesh, wireframe_opts=None, entities_opts=None)
Convenience function for plotting all entities of a finite element mesh.
  Pass plot() arguments to wireframe_opts dict.
  Pass 'color', 'label_global', 'label_local' for text() color and font sizes arguments and 'size' for scatter() to each dict for topological entities in entities_opts list.

**Examples**

```python
>>> # 2D mesh.
>>> plot_cmesh(None, cmesh,

>>>  wireframe_opts = {'color': 'k', 'linewidth': 2},
   entities_opts=[
   {'color': 'k', 'label_local': 8, 'size': 20},
   {'color': 'b', 'label_global': 12, 'label_local': 8, 'size': 10},
   {'color': 'r', 'label_global': 12, 'size': 20},
])
```
sfepy.postprocess.plot_cmesh.plot_entities(ax, cmesh, edim, color='b', size=10, **kwargs)
Plot mesh topology entities using scatter plot.
sfepy.postprocess.plot_cmesh.plot_wireframe(ax, cmesh, color='k', **kwargs)
Plot a finite element mesh as a wireframe using edges connectivity.
**sfepy.postprocess.plot_dofs module**

Functions to visualize the mesh connectivity with global and local DOF numberings.

**sfepy.postprocess.plot_dofs.plot_global_dofs** *(ax, coors, econn)*  
Plot global DOF numbers given in an extended connectivity.  
The DOF numbers are plotted for each element, so on common facets they are plotted several times - this can be used to check the consistency of the global DOF connectivity.

**sfepy.postprocess.plot_dofs.plot_local_dofs** *(ax, coors, econn)*  
Plot local DOF numbers corresponding to an extended connectivity.

**sfepy.postprocess.plot_dofs.plot_mesh** *(ax, coors, conn, edges, color='k', **plot_kwargs)*  
Plot a finite element mesh as a wireframe.

**sfepy.postprocess.plot_dofs.plot_nodes** *(ax, coors, econn, ref_nodes, dofs)*  
Plot Lagrange reference element nodes corresponding to global DOF numbers given in an extended connectivity.

**sfepy.postprocess.plot_dofs.plot_points** *(ax, coors, vals=None, point_size=20, show_colorbar=False)*  
Plot points with given coordinates, optionally colored using vals values.

**sfepy.postprocess.plot_facets module**

Functions to visualize the geometry elements and numbering and orientation of their facets (edges and faces).

The standard geometry elements can be plotted by running:

```
$ python sfepy/postprocess/plot_facets.py
```

**sfepy.postprocess.plot_facets.draw_arrow** *(ax, coors, angle=20.0, length=0.3, **kwargs)*  
Draw a line ended with an arrow head, in 2D or 3D.

**sfepy.postprocess.plot_facets.plot_edges** *(ax, gel, length)*  
Plot edges of a geometry element as numbered arrows.

**sfepy.postprocess.plot_facets.plot_faces** *(ax, gel, radius, n_point)*  
Plot faces of a 3D geometry element as numbered oriented arcs. An arc centre corresponds to the first node of a face. It points from the first edge towards the last edge of the face.

**sfepy.postprocess.plot_facets.plot_geometry** *(ax, gel)*  
Plot a geometry element as a wireframe.

**sfepy.postprocess.plot_quadrature module**

Functions to visualize quadrature points in reference elements.

**sfepy.postprocess.plot_quadrature.label_points** *(ax, coors)*  
Label points with their indices.

**sfepy.postprocess.plot_quadrature.plot_quadrature** *(ax, geometry, order, boundary=False, min_radius=10, max_radius=50, show_colorbar=False, show_labels=False)*
Plot quadrature points for the given geometry and integration order.
The points are plotted as circles/spheres with radii given by quadrature weights - the weights are mapped to $[\min_radius, \max_radius]$ interval.

```python
sfepy.postprocess.plot_quadrature.plot_weighted_points(ax, coors, weights, min_radius=10, max_radius=50, show_colorbar=False)
```

Plot points with given coordinates as circles/spheres with radii given by weights.

### sfepy.postprocess.probes_vtk module

Classes for probing values of Variables, for example, along a line, using PyVTK library

```python
class sfepy.postprocess.probes_vtk.Probe(data, mesh, **kwargs)
```

Probe class.

```python
add_circle_probe(name, centre, normal, radius, n_point)
```
Create the ray (line) probe - VTK object.

**Parameters**

- **name**
  - [str] The probe name.
- **centre**
  - [array] The coordinates of the circle center point.
- **normal**
  - [array] The normal vector perpendicular to the circle plane.
- **radius**
  - [float] The radius of the circle.
- **n_point**
  - [int] The number of probe points.

```python
add_line_probe(name, p0, p1, n_point)
```
Create the line probe - VTK object.

**Parameters**

- **name**
  - [str] The probe name.
- **p0**
  - [array_like] The coordinates of the start point.
- **p1**
  - [array_like] The coordinates of the end point.
- **n_point**
  - [int] The number of probe points.

```python
add_points_probe(name, coors)
```
Create the point probe - VTK object.

**Parameters**

- **name**
  - [str] The probe name.
coors
[array] The coordinates of the probe points.

add_ray_probe(name, p0, dirvec, p_fun, n_point)
Create the ray (line) probe - VTK object.

Parameters
    name
        [str] The probe name.
    p0
        [array] The coordinates of the start point.
    dirvec
        [array] The probe direction vector.
    p_fun
        [function] The function returning the probe parametrization along the dirvec direction.
    n_point
        [int] The number of probe points.

gen_mesh_probe_png(probe, png_filename)
Generate PNG image of the FE mesh.

Parameters
    probe
        [VTK object]str] The probe, VTKPolyData or VTKSource.
    png_filename
        [str] The name of the output PNG file.

new_vtk_polyline(points, closed=False)
Create the VTKPolyData object and store the line data.

Parameters
    points
        [array] The line points.

Returns
    vtkpd
        [VTK object] VTKPolyData with the polyline.

class sfepy.postprocess.probes_vtk.ProbeFromFile(filename, **kwargs)
Probe class - read a given VTK file.

sfepy.postprocess.time_history module

sfepy.postprocess.time_history.average_vertex_var_in_cells(ths_in)
Average histories in the element nodes for each nodal variable originally requested in elements.

sfepy.postprocess.time_history.dump_to_vtk(filename, output_filename_trunk=None, step0=0, steps=None, fields=None, linearization=None)
Dump a multi-time-step results file into a sequence of VTK files.
sfepy.postprocess.time_history.extract_time_history(filename, extract, verbose=True)

Extract time history of a variable from a multi-time-step results file.

**Parameters**

- **filename**
  - [str] The name of file to extract from.

- **extract**
  - [str] The description of what to extract in a string of comma-separated description items. A description item consists of: name of the variable to extract, mode ('e' for elements, 'n' for nodes), ids of the nodes or elements (given by the mode). Example: 'u n 10 15, p e 0' means variable 'u' in nodes 10, 15 and variable 'p' in element 0.

- **verbose**
  - [bool] Verbosity control.

**Returns**

- **ths**
  - [dict] The time histories in a dict with variable names as keys. If a nodal variable is requested in elements, its value is a dict of histories in the element nodes.

- **ts**
  - [TimeStepper instance] The time stepping information.

sfepy.postprocess.time_history.extract_times(filename)

Read true time step data from individual time steps.

**Returns**

- **steps**
  - [array] The time steps.

- **times**
  - [array] The times of the time steps.

- **nts**
  - [array] The normalized times of the time steps, in [0, 1].

- **dts**
  - [array] The true time deltas.

sfepy.postprocess.time_history.guess_time_units(times)

Given a vector of times in seconds, return suitable time units and new vector of times suitable for plotting.

**Parameters**

- **times**
  - [array] The vector of times in seconds.

**Returns**

- **new_times**
  - [array] The vector of times in units.

- **units**
  - [str] The time units.

sfepy.postprocess.time_history.save_time_history(ths, ts, filename_out)

Save time history and time-stepping information in a HDF5 file.
sfepy.postprocess.utils_vtk module

Postprocessing utils based on VTK library

sfepy.postprocess.utils_vtk.get_vtk_by_group(vtkdata, group_lower, group_upper=None)

Get submesh by material group id.

Parameters

- **vtkdata**  
  [VTK object] Mesh, scalar, vector and tensor data.

- **group_lower**  
  [int] The lower material id.

- **group_upper**  
  [int] The upper material id.

Returns

- **selection**  
  [VTK object] Mesh, scalar, vector and tensor data.

sfepy.postprocess.utils_vtk.get_vtk_edges(vtkdata)

Get mesh edges.

Parameters

- **vtkdata**  
  [VTK object] Mesh, scalar, vector and tensor data.

Returns

- **edges**  
  [VTK object] Mesh, scalar, vector and tensor data.

sfepy.postprocess.utils_vtk.get_vtk_from_file(filename)

Read VTK file.

Parameters

- **filename**  
  [str] Name of the VTK file.

Returns

- **vtkdata**  
  [VTK object] Mesh, scalar, vector and tensor data.

sfepy.postprocess.utils_vtk.get_vtk_from_mesh(mesh, data, prefix="")

sfepy.postprocess.utils_vtk.get_vtk_surface(vtkdata)

Get mesh surface.

Parameters

- **vtkdata**  
  [VTK object] Mesh, scalar, vector and tensor data.

Returns

- **surface**  
  [VTK object] Mesh, scalar, vector and tensor data.
sfepy.postprocess.utils_vtk.tetrahedralize_vtk_mesh(vtkdata)

3D cells are converted to tetrahedral meshes, 2D cells to triangles.

Parameters

vtkdata

[VTK object] Mesh, scalar, vector and tensor data.

Returns

tetra

[VTK object] Mesh, scalar, vector and tensor data.

sfepy.postprocess.utils_vtk.write_vtk_to_file(filename, vtkdata)

Write VTK file.

Parameters

filename

[str] Name of the VTK file.

vtkdata

[VTK object] Mesh, scalar, vector and tensor data.

sfepy.solvers package

sfepy.solvers.auto_fallback module

class sfepy.solvers.auto_fallback.AutoDirect(conf, **kwargs)

The automatically selected linear direct solver.

The first available solver from the following list is used: `ls.mumps <sfepy.solvers.ls.MUMPSSolver>`, `ls.scipy.umfpack <sfepy.solvers.ls.ScipyUmfpack>` and `ls.scipy.superlu <sfepy.solvers.ls.ScipySuperLU>`.

Kind: 'ls.auto_direct'

For common configuration parameters, see Solver.

Specific configuration parameters:

name = 'ls.auto_direct'

class sfepy.solvers.auto_fallback.AutoFallbackSolver(conf, **kwargs)

Base class for virtual solvers with the automatic fallback.

class sfepy.solvers.auto_fallback.AutoIterative(conf, **kwargs)

The automatically selected linear iterative solver.

The first available solver from the following list is used: `ls.petsc <sfepy.solvers.ls.PETScKrylovSolver>` and `ls.scipy_iterative <sfepy.solvers.ls.ScipyIterative>`

Kind: 'ls.auto_iterative'

For common configuration parameters, see Solver.

Specific configuration parameters:

name = 'ls.auto_iterative'
**sfepy.solvers.eigen module**

```python
class sfepy.solvers.eigen.LOBPCGEigenvalueSolver(conf, **kwargs)
SciPy-based LOBPCG solver for sparse symmetric problems.
Kind: ‘eig.scipy_lobpcg’
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

i_max
[int (default: 20)] The maximum number of iterations.

eps_a
[float] The absolute tolerance for the convergence.

largest
[bool (default: True)] If True, solve for the largest eigenvalues, otherwise the smallest.

precond
[(dense matrix, sparse matrix, LinearOperator)] The preconditioner.
```

```python
class sfepy.solvers.eigen.MatlabEigenvalueSolver(conf, comm=None, context=None, **kwargs)
Matlab eigenvalue problem solver.
Kind: ‘eig.matlab’
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

method
[{'eig', 'eigs', None} (default: ‘eigs’)] The solution method. Note that eig() function cannot
be used for all inputs. If n_eigs is not None, eigs() is used regardless of this parameter.

balance
[{'balance', 'nobalance'} (default: ‘balance’)] The balance option for eig().

algorithm
[{'chol', 'qz'} (default: ‘chol’)] The algorithm option for eig().

which
[{'lm', 'sm', 'la', 'sa', 'be', ‘lr’, ‘sr’, ‘li’, ‘si’, sigma} (default: ‘lm’)] Which eigenvectors and
eigenvalues to find with eigs().

* Additional parameters supported by eigs().
```

```python
class sfepy.solvers.eigen.PrimmeEigenvalueSolver(conf, comm=None, context=None, **kwargs)
PRIMME eigenvalue problem solver.
https://github.com/primme/primme
Installation: pip install primme
```
Kind: ‘eig.primme’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters

which


sigma

[float] Find eigenvalues near sigma.

maxiter

[int] Maximum number of iterations.

tol

[float (default: 0)] Tolerance for eigenpairs (stopping criterion).

* [*] Additional parameters supported by eigsh().

name = 'eig.primme'

class sfepy.solvers.eigen.SLEPCEigenvalueSolver(conf, comm=None, context=None, **kwargs)

General SLEPc eigenvalue problem solver.

Kind: ‘eig.slepc’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters

method

[str (default: ‘krylovshur’)] The actual solver to use.

problem

[str (default: ‘gnhep’)] The problem type: Hermitian (hep), non-Hermitian (nhep), generalized Hermitian (ghep), generalized non-Hermitian (gnhep), generalized non-Hermitian with positive semi-definite B (pgnhep), and generalized Hermitian indefinite (ghiep).

i_max

[int (default: 20)] The maximum number of iterations.

eps

[float] The convergence tolerance.

conv_test

[{{“abs”, “rel”, “norm”, “user”}}, (default: ‘abs’)] The type of convergence test.

which

[{{‘largest_magnitude’, ‘smallest_magnitude’}, ‘largest_real’, ‘smallest_real’,

* [*] Additional parameters supported by the method.
create_eps(options=None, comm=None)
create_petsc_matrix(mtx, comm=None)

name = 'eig.slepc'

class sfepy.solvers.eigen.ScipyEigenvalueSolver(conf, **kwargs)
SciPy-based solver for both dense and sparse problems.
The problem is considered sparse if n_eigs argument is not None.
Kind: 'eig.scipy'
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

method
[{'eig', 'eigh', 'eigs', 'eigsh'}] (default: 'eigs') The method for solving general or symmetric
eigenvalue problems: for dense problems eig() or eigh() can be used, for sparse problems
eigs() or eigsh() should be used.

which
['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'] (default: 'SM') Which eigenvectors and eigenvalues
to find, see scipy.sparse.linalg.eigs() or scipy.sparse.linalg.eigsh(). For
dense problems, only 'LM' and 'SM' can be used

* Additional parameters supported by the method.

name = 'eig.scipy'

class sfepy.solvers.eigen.ScipySGEigenvalueSolver(conf, **kwargs)
SciPy-based solver for dense symmetric problems.
Kind: 'eig.sgscipy'
For common configuration parameters, see Solver.
Specific configuration parameters:

name = 'eig.sgscipy'

sfepy.solvers.eigen.eig(mtx_a, mtx_b=None, n_eigs=None, eigenvectors=True, return_time=None,
solver_kind='eig.scipy', **ckwargs)

Utility function that constructs an eigenvalue solver given by solver_kind, calls it and returns its output.

sfepy.solvers.eigen.init_slepc_args()

sfepy.solvers.eigen.standard_call(call)
Decorator handling argument preparation and timing for eigensolvers.
**sfepy.solvers.ls module**

```python
class sfepy.solvers.ls.MUMPSParallelSolver(conf, **kwargs)
    Interface to MUMPS parallel solver.
    Kind: ‘ls.mumps_par’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    Parameters
        memory_relaxation
            [int (default: 20)] The percentage increase in the estimated working space.
        name = 'ls.mumps_par'

class sfepy.solvers.ls.MUMPSSolver(conf, **kwargs)
    Interface to MUMPS solver.
    Kind: ‘ls.mumps’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    Parameters
        use_presolve
            [bool (default: False)] If True, pre-factorize the matrix.
        use_mtx_digest
            [bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest.
            If False, .clear() has to be called manually whenever the matrix changes - expert use only!
        memory_relaxation
            [int (default: 20)] The percentage increase in the estimated working space.
        clear()
        name = 'ls.mumps'
        presolve(mtx, use_mtx_digest=True)

class sfepy.solvers.ls.MultiProblem(conf, context=None, **kwargs)
    Conjugate multiple problems.
    Allows to define conjugate multiple problems.
    Kind: ‘ls.cm_pb’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    Parameters
        method
            [{'auto', 'umfpack', 'superlu'} (default: ‘auto’)] The actual solver to use.
        use_presolve
            [bool (default: False)] If True, pre-factorize the matrix.
```

2.3. Developer Guide
use_mtx_digest
    [bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest.
    If False, .clear() has to be called manually whenever the matrix changes - expert use only!

others
    [list] The list of auxiliary problem definition files.

coupling_variables
    [list] The list of coupling variables.

init_subproblems(conf, **kwargs)

name = 'ls.cm_pb'

sparse_submat(Ad, Ar, Ac, gr, gc, S)
    A[gr,gc] = S

class sfepy.solvers.ls.PETScKrylovSolver(conf, comm=None, context=None, **kwargs)

    PETSc Krylov subspace solver.
    The solver supports parallel use with a given MPI communicator (see comm argument of PETScKrylovSolver.
    __init__() ) and allows passing in PETSc matrices and vectors. Returns a (global) PETSc solution vector
    instead of a (local) numpy array, when given a PETSc right-hand side vector.
    The solver and preconditioner types are set upon the solver object creation. Tolerances can be overridden when
    called by passing a conf object.
    Convergence is reached when \( rnorm < \max(\text{eps}_r \times rnorm_0, \text{eps}_a) \), where, in PETSc, \( rnorm \) is by default the
    norm of preconditioned residual.
    Kind: ‘ls.petsc’
    For common configuration parameters, see Solver.

    Specific configuration parameters:

    Parameters

method
    [str (default: ‘cg’)] The actual solver to use.

setup_precond
    [callable] User-supplied function for the preconditioner initialization/setup. It is called as
    setup_precond(mtx, context), where mtx is the matrix, context is a user-supplied context, and
    should return an object with setUp(self, pc) and apply(self, pc, x, y) methods. Has precedence
    over the precond/sub_precond parameters.

precond
    [str (default: ‘icc’)] The preconditioner.

sub_precond
    [str (default: ‘none’)] The preconditioner for matrix blocks (in parallel runs).

precond_side
    [{‘left’, ‘right’, ‘symmetric’, None}] The preconditioner side.

i_max
    [int (default: 100)] The maximum number of iterations.

eps_a
    [float (default: 1e-08)] The absolute tolerance for the residual.

eps_r
    [float (default: 1e-08)] The relative tolerance for the residual.
eps_d
[float (default: 100000.0)] The divergence tolerance for the residual.

force_reuse
[bool (default: False)] If True, skip the check whether the KSP solver object corresponds to
the mtx argument: it is always reused.

[*] Additional parameters supported by the method. Can be used to pass all PETSc options
supported by petsc.Options().

create_ksp(options=None, comm=None)
create_petsc_matrix(mtx, comm=None)

name = 'ls.petsc'

set_field_split(field_ranges, comm=None)
Setup local PETSc ranges for fields to be used with ‘fieldsplit’ preconditioner.
This function must be called before solving the linear system.

class sfepy.solvers.ls.PyAMGKrylovSolver(conf, context=None, **kwargs)
Interface to PyAMG Krylov solvers.
Kind: ‘ls.pyamg_krylov’
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

method
[str (default: ‘cg’)] The actual solver to use.

setup_precond
[callback (default: <function PyAMGKrylovSolver.<lambda> at 0x7f41b7179940>)]
User-supplied function for the preconditioner initialization/setup. It is called as
setup_precond(mtx, context), where mtx is the matrix, context is a user-supplied context,
and should return one of {sparse matrix, dense matrix, LinearOperator}.

callback
[callback] User-supplied function to call after each iteration. It is called as callback(xk),
where xk is the current solution vector, except the gmres method, where the argument is the
residual norm.

i_max
[int (default: 100)] The maximum number of iterations.

eps_r
[float (default: 1e-08)] The relative tolerance for the residual.

[*] Additional parameters supported by the method.

name = 'ls.pyamg_krylov'

class sfepy.solvers.ls.PyAMGSolver(conf, **kwargs)
Interface to PyAMG solvers.
The method parameter can be one of: ‘smoothed_aggregation_solver’, ‘ruge_stuben_solver’. The accel param-
eter specifies the Krylov solver name, that is used as an accelerator for the multigrid solver.
Kind: ‘ls.pyamg’

For common configuration parameters, see Solver.

Specific configuration parameters:

**Parameters**

- **method**
  - [str (default: ‘smoothed_aggregation_solver’)] The actual solver to use.

- **accel**
  - [str] The accelerator.

- **callback**
  - [callable] User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector, except the gmres accelerator, where the argument is the residual norm.

- **i_max**
  - [int (default: 100)] The maximum number of iterations.

- **eps_r**
  - [float (default: 1e-08)] The relative tolerance for the residual.

- **force_reuse**
  - [bool (default: False)] If True, skip the check whether the MG solver object corresponds to the mtx argument: it is always reused.

- **rmm_term**
  - [str] The RMM term definition, see MassTerm.

- **debug**
  - [bool (default: False)] If True, run in debug mode.

- **init_rmm** (mtx)

---

class sfepy.solvers.ls.RMMSolver(conf, context=None, **kwargs)

Special solver for explicit transient elastodynamics.

The solver uses the reciprocal mass matrix algorithm [1], [2] to directly construct a sparse inverse mass matrix. Instead of solving a linear system, calling the solver simply performs a sparse matrix multiplication.

Limitations:

- Assumes that the density is constant in time.
- Uses the direct EBC application, i.e., no EBC projection matrix.

Kind: ‘ls.rmm’

For common configuration parameters, see Solver.

Specific configuration parameters:

**Parameters**

- **rmm_term**
  - [str] The RMM term definition, see MassTerm.

- **debug**
  - [bool (default: False)] If True, run in debug mode.

- **init_rmm** (mtx)
**name** = 'ls.rmm'

**class** sfepy.solvers.ls.SchurMumps(conf, **kwargs)**
Mumps Schur complement solver.
Kind: ‘ls.schur_mumps’
For common configuration parameters, see *Solver*. Specific configuration parameters:

**Parameters**

- `use_presolve` (bool, default: False) If True, pre-factorize the matrix.
- `use_mtx_digest` (bool, default: True) If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!
- `memory_relaxation` (int, default: 20) The percentage increase in the estimated working space.
- `schur_variables` (list) The list of Schur variables.

**name** = 'ls.schur_mumps'

**class** sfepy.solvers.ls.ScipyDirect(conf, method=None, **kwargs)**
Direct sparse solver from SciPy.
Kind: ‘ls.scipy_direct’
For common configuration parameters, see *Solver*. Specific configuration parameters:

**Parameters**

- `use_presolve` (bool, default: False) If True, pre-factorize the matrix.
- `use_mtx_digest` (bool, default: True) If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

**clear()**

**name** = 'ls.scipy_direct'

**presolve(mtx, use_mtx_digest=True)**

**class** sfepy.solvers.ls.ScipyIterative(conf, context=None, **kwargs)**
Interface to SciPy iterative solvers.
The `eps_r` tolerance is both absolute and relative - the solvers stop when either the relative or the absolute residual is below it.
Kind: ‘ls.scipy_iterative’
For common configuration parameters, see *Solver*.  

---

**2.3. Developer Guide**
Specific configuration parameters:

**Parameters**

- **method**
  - [str (default: ‘cg’)] The actual solver to use.

- **setup_precond**
  - [callable (default: <function ScipyIterative.<lambda> at 0x7f41b7179550>)] User-supplied function for the preconditioner initialization/setup. It is called as setup_precond(mtx, context), where mtx is the matrix, context is a user-supplied context, and should return one of {sparse matrix, dense matrix, LinearOperator}.

- **callback**
  - [callable] User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector, except the gmres method, where the argument is the residual.

- **i_max**
  - [int (default: 100)] The maximum number of iterations.

- **eps_a**
  - [float (default: 1e-08)] The absolute tolerance for the residual.

- **eps_r**
  - [float (default: 1e-08)] The relative tolerance for the residual.

- **name** = ‘ls.scipy_iterative’

**class** sfepy.solvers.ls.ScipySuperLU(conf, **kwargs)

SuperLU - direct sparse solver from SciPy.

Kind: ‘ls.scipy_superlu’

For common configuration parameters, see *Solver*.

Specific configuration parameters:

- **use_presolve**
  - [bool (default: False)] If True, pre-factorize the matrix.

- **use_mtx_digest**
  - [bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

**name** = ‘ls.scipy_superlu’

**class** sfepy.solvers.ls.ScipyUmfpack(conf, **kwargs)

UMFPACK - direct sparse solver from SciPy.

Kind: ‘ls.scipy_umfpack’

For common configuration parameters, see *Solver*.

Specific configuration parameters:

- **use_presolve**
  - [bool (default: False)] If True, pre-factorize the matrix.
use_mtx_digest

[bool (default: True)] If True, determine automatically a reused matrix using its SHA1 digest. If False, .clear() has to be called manually whenever the matrix changes - expert use only!

name = 'ls.scipy_umfpack'

sfepy.solvers.ls.petsc_call(call)
Decorator handling argument preparation and timing for PETSc-based linear solvers.

sfepy.solvers.ls.solve(mtx, rhs, solver_class=None, solver_conf=None)
Solve the linear system with the matrix mtx and the right-hand side rhs.
Convenience wrapper around the linear solver classes below.

sfepy.solvers.ls.standard_call(call)
Decorator handling argument preparation and timing for linear solvers.

sfepy.solvers.ls_mumps module

class sfepy.solvers.ls_mumps.MumpsSolver(is_sym=False, mpi_comm=None, system='real', silent=True, mem_relax=20)
MUMPS object.

expand_schur(x2)
Expand the Schur local solution on the complete solution.

Parameters
x2
[array] The local Schur solution.

Returns
x
[array] The global solution.

get_schur(schur_list)
Get the Schur matrix and the condensed right-hand side vector.

Parameters
schur_list
[array] The list of the Schur DOFs (indexing starts with 1).

Returns
schur_arr
[array] The Schur matrix of order ‘schur_size’.
schur_rhs
[array] The reduced right-hand side vector.

set_mtx_centralized(mtx)
Set the sparse matrix.

Parameters
mtx
[scipy sparse matrix] The sparse matrix in COO format.
**set_rcd_centralized**(*ir, ic, data, n*)

Set the matrix by row and column indicies and data vector. The matrix shape is determined by the maximal values of row and column indicies. The indices start with 1.

**Parameters**

- **ir** ([array]) The row indicies.
- **ic** ([array]) The column indicies.
- **data** ([array]) The matrix entries.
- **n** ([int]) The matrix dimension.

**set_rhs**(*rhs*)

Set the right hand side of the linear system.

**set_silent()**

**set_verbose()**

---

**sfepy.solvers.ls_mumps.coo_is_symmetric**(*mtx, tol=1e-06*)

**sfepy.solvers.ls_mumps.dec**(*val, encoding='utf-8*)

Decode given bytes using the specified encoding.

**sfepy.solvers.ls_mumps.load_library**(*libname*)

Load shared library in a system dependent way.

**sfepy.solvers.ls_mumps.load_mumps_libraries()**

**sfepy.solvers.ls_mumps.mumps_pcomplex**

alias of LP_c_double

**sfepy.solvers.ls_mumps.mumps_preal**

alias of LP_c_double

**class sfepy.solvers.ls_mumps.mumps_struc_c_4**

- **a**
  Structure/Union member
- **a_elt**
  Structure/Union member
- **a_loc**
  Structure/Union member
- **cntl**
  Structure/Union member
- **colsca**
  Structure/Union member
- **comm_fortran**
  Structure/Union member
deficiency  
    Structure/Union member
eltptr  
    Structure/Union member
eltvar  
    Structure/Union member
icntl  
    Structure/Union member
info  
    Structure/Union member
infog  
    Structure/Union member
instance_number  
    Structure/Union member
irhs_ptr  
    Structure/Union member
irhs_sparse  
    Structure/Union member
irn  
    Structure/Union member
irn_loc  
    Structure/Union member
isol_loc  
    Structure/Union member
jcn  
    Structure/Union member
jcn_loc  
    Structure/Union member
job  
    Structure/Union member
listvar_schur  
    Structure/Union member
lredrhs  
    Structure/Union member
lrhs  
    Structure/Union member
lsol_loc  
    Structure/Union member
lwk_user  
    Structure/Union member
mapping
    Structure/Union member
mblock
    Structure/Union member
n
    Structure/Union member
nblock
    Structure/Union member
neit
    Structure/Union member
npcol
    Structure/Union member
nprow
    Structure/Union member
nrhs
    Structure/Union member
nz
    Structure/Union member
nz_alloc
    Structure/Union member
nz_loc
    Structure/Union member
nz_rhs
    Structure/Union member
ooc_prefix
    Structure/Union member
ooc_tmpdir
    Structure/Union member
par
    Structure/Union member
perm_in
    Structure/Union member
pivmul_list
    Structure/Union member
redrhs
    Structure/Union member
rhs
    Structure/Union member
rhs_sparse
    Structure/Union member
rinfo
  Structure/Union member
rinfog
  Structure/Union member
rowsca
  Structure/Union member
schur
  Structure/Union member
schur_1ld
  Structure/Union member
schur_nloc
  Structure/Union member
schur_nloc
  Structure/Union member
size_schur
  Structure/Union member
sol_loc
  Structure/Union member
sym
  Structure/Union member
sym_perm
  Structure/Union member
uns_perm
  Structure/Union member
version_number
  Structure/Union member
wk_user
  Structure/Union member
write_problem
  Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struc_c_5_0
  a
    Structure/Union member
  a_elt
    Structure/Union member
  a_loc
    Structure/Union member
  cnt1
    Structure/Union member
colsca
    Structure/Union member

colsca_from_mumps
    Structure/Union member

comm_fortran
    Structure/Union member

deficiency
    Structure/Union member

dkeep
    Structure/Union member

eltptr
    Structure/Union member

eltvar
    Structure/Union member

icntl
    Structure/Union member

info
    Structure/Union member

infog
    Structure/Union member

instance_number
    Structure/Union member

irhs_ptr
    Structure/Union member

irhs_sparse
    Structure/Union member

irn
    Structure/Union member

irn_loc
    Structure/Union member

isol_loc
    Structure/Union member

jcn
    Structure/Union member

jcn_loc
    Structure/Union member

job
    Structure/Union member

keep
    Structure/Union member
keep8
   Structure/Union member
listvar_schur
   Structure/Union member
lredrhs
   Structure/Union member
lrhs
   Structure/Union member
lsol_loc
   Structure/Union member
lwk_user
   Structure/Union member
mapping
   Structure/Union member
mblock
   Structure/Union member
n
   Structure/Union member
nblocK
   Structure/Union member
nelt
   Structure/Union member
npcol
   Structure/Union member
nprow
   Structure/Union member
nrhs
   Structure/Union member
nz
   Structure/Union member
nz_alloc
   Structure/Union member
nz_loc
   Structure/Union member
nz_rhs
   Structure/Union member
ooc_prefix
   Structure/Union member
ooc_tmpdir
   Structure/Union member
par
  Structure/Union member

perm_in
  Structure/Union member

pivmul_list
  Structure/Union member

redrhs
  Structure/Union member

rhs
  Structure/Union member

rhs_sparse
  Structure/Union member

rinfo
  Structure/Union member

rinfog
  Structure/Union member

rowsca
  Structure/Union member

rowsca_from_mumps
  Structure/Union member

schur
  Structure/Union member

schur_lld
  Structure/Union member

schur_mloc
  Structure/Union member

schur_nloc
  Structure/Union member

size_schur
  Structure/Union member

sol_loc
  Structure/Union member

sym
  Structure/Union member

sym_perm
  Structure/Union member

uns_perm
  Structure/Union member

version_number
  Structure/Union member
class sfepy.solvers.ls_mumps.mumps_struc_c_5_1

    a     Structure/Union member
    a_elt Structure/Union member
    a_loc Structure/Union member
    cntl  Structure/Union member
    colsca Structure/Union member
    colsca_from_mumps Structure/Union member
    comm_fortran Structure/Union member
    deficiency Structure/Union member
    dkeep Structure/Union member
    eltptr Structure/Union member
    eltvar Structure/Union member
    icntl Structure/Union member
    info  Structure/Union member
    infog Structure/Union member
    instance_number Structure/Union member
    irhs_ptr Structure/Union member
    irhs_sparse Structure/Union member
irn  
    Structure/Union member
irn_loc  
    Structure/Union member
isol_loc  
    Structure/Union member
jcn  
    Structure/Union member
jcn_loc  
    Structure/Union member
job  
    Structure/Union member
keep  
    Structure/Union member
keep8  
    Structure/Union member
listvar_schur  
    Structure/Union member
lredrhs  
    Structure/Union member
lrhs  
    Structure/Union member
lsol_loc  
    Structure/Union member
lwk_user  
    Structure/Union member
mapping  
    Structure/Union member
mblock  
    Structure/Union member
n  
    Structure/Union member
nblock  
    Structure/Union member
nelt  
    Structure/Union member
nnz  
    Structure/Union member
nnz_loc  
    Structure/Union member
npcol
   Structure/Union member

nprow
   Structure/Union member

nrhs
   Structure/Union member

nz
   Structure/Union member

nz_alloc
   Structure/Union member

nz_loc
   Structure/Union member

nz_rhs
   Structure/Union member

ooc_prefix
   Structure/Union member

ooc_tmpdir
   Structure/Union member

par
   Structure/Union member

perm_in
   Structure/Union member

pivnul_list
   Structure/Union member

redrhs
   Structure/Union member

rhs
   Structure/Union member

rhs_sparse
   Structure/Union member

rinfo
   Structure/Union member

rinfog
   Structure/Union member

rowsca
   Structure/Union member

rowsca_from_mumps
   Structure/Union member

save_dir
   Structure/Union member
save_prefix
    Structure/Union member

schur
    Structure/Union member

schur_lld
    Structure/Union member

schur_nloc
    Structure/Union member

size_schur
    Structure/Union member

sol_loc
    Structure/Union member

sym
    Structure/Union member

sym_perm
    Structure/Union member

uns_perm
    Structure/Union member

version_number
    Structure/Union member

wk_user
    Structure/Union member

write_problem
    Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struct_c_5_2

    a
        Structure/Union member

    a_elt
        Structure/Union member

    a_loc
        Structure/Union member

    cnt1
        Structure/Union member

    colsca
        Structure/Union member

    colsca_from_mumps
        Structure/Union member
comm_fortran
  Structure/Union member
deficiency
  Structure/Union member
dkeep
  Structure/Union member
eltptr
  Structure/Union member
eltvar
  Structure/Union member
icntl
  Structure/Union member
info
  Structure/Union member
infog
  Structure/Union member
instance_number
  Structure/Union member
irhs_loc
  Structure/Union member
irhs_ptr
  Structure/Union member
irhs_sparse
  Structure/Union member
irn
  Structure/Union member
irn_loc
  Structure/Union member
isol_loc
  Structure/Union member
jcn
  Structure/Union member
jcn_loc
  Structure/Union member
job
  Structure/Union member
keep
  Structure/Union member
keep8
  Structure/Union member
listvar_schur
  Structure/Union member
lreddrhs
  Structure/Union member
lrhs
  Structure/Union member
lrhs_loc
  Structure/Union member
lsol_loc
  Structure/Union member
lwk_user
  Structure/Union member
mapping
  Structure/Union member
mblock
  Structure/Union member
metis_options
  Structure/Union member
n
  Structure/Union member
nblock
  Structure/Union member
nelt
  Structure/Union member
nloc_rhs
  Structure/Union member
nnz
  Structure/Union member
nnz_loc
  Structure/Union member
npcol
  Structure/Union member
nprow
  Structure/Union member
nrhs
  Structure/Union member
nz
  Structure/Union member
nz_alloc
  Structure/Union member
nz_loc
    Structure/Union member

nz_rhs
    Structure/Union member

ooc_prefix
    Structure/Union member

ooc_tmpdir
    Structure/Union member

par
    Structure/Union member

perm_in
    Structure/Union member

pivnul_list
    Structure/Union member

redrhs
    Structure/Union member

rhs
    Structure/Union member

rhs_loc
    Structure/Union member

rhs_sparse
    Structure/Union member

rinfo
    Structure/Union member

rinfog
    Structure/Union member

rowsca
    Structure/Union member

rowsca_from_mumps
    Structure/Union member

save_dir
    Structure/Union member

save_prefix
    Structure/Union member

schur
    Structure/Union member

schur_lld
    Structure/Union member

schur_mloc
    Structure/Union member
```python
class sfepy.solvers.ls_mumps.mumps_struc_c_5_3
    a
    Structure/Union member
    a_elt
    Structure/Union member
    a_loc
    Structure/Union member
    blkptr
    Structure/Union member
    blkvar
    Structure/Union member
    cnt1
    Structure/Union member
    colsca
    Structure/Union member
    colsca_from_mumps
    Structure/Union member
    comm_fortran
    Structure/Union member
    deficiency
    Structure/Union member
```
dkeep
   Structure/Union member
eltptr
   Structure/Union member
eltvar
   Structure/Union member
icntl
   Structure/Union member
info
   Structure/Union member
infog
   Structure/Union member
instance_number
   Structure/Union member
irhs_loc
   Structure/Union member
irhs_ptr
   Structure/Union member
irhs_sparse
   Structure/Union member
irn
   Structure/Union member
irn_loc
   Structure/Union member
isol_loc
   Structure/Union member
jcn
   Structure/Union member
jcn_loc
   Structure/Union member
job
   Structure/Union member
keep
   Structure/Union member
keep8
   Structure/Union member
listvar_schur
   Structure/Union member
lredrhs
   Structure/Union member
lrhs
  Structure/Union member

lrhs_loc
  Structure/Union member

lsol_loc
  Structure/Union member

lwk_user
  Structure/Union member

mapping
  Structure/Union member

mblock
  Structure/Union member

metis_options
  Structure/Union member

n
  Structure/Union member

nbl
  Structure/Union member

nbloc
  Structure/Union member

nelt
  Structure/Union member

nloc_rhs
  Structure/Union member

nnz
  Structure/Union member

nnz_loc
  Structure/Union member

npcol
  Structure/Union member

nprow
  Structure/Union member

nrhs
  Structure/Union member

nz
  Structure/Union member

nz_alloc
  Structure/Union member

nz_loc
  Structure/Union member
nz_rhs
    Structure/Union member

ooc_prefix
    Structure/Union member

ooc_tmpdir
    Structure/Union member

par
    Structure/Union member

perm_in
    Structure/Union member

pivnul_list
    Structure/Union member

redrhs
    Structure/Union member

rhs
    Structure/Union member

rhs_loc
    Structure/Union member

rhs_sparse
    Structure/Union member

rinfo
    Structure/Union member

rinfog
    Structure/Union member

rowsca
    Structure/Union member

rowsca_from_mumps
    Structure/Union member

save_dir
    Structure/Union member

save_prefix
    Structure/Union member

schur
    Structure/Union member

schur_lld
    Structure/Union member

schur_mloc
    Structure/Union member

schur_nloc
    Structure/Union member
size_schur  
Structure/Union member

sol_loc  
Structure/Union member

sym  
Structure/Union member

sym_perm  
Structure/Union member

uns_perm  
Structure/Union member

version_number  
Structure/Union member

wk_user  
Structure/Union member

write_problem  
Structure/Union member

class sfepy.solvers.ls_mumps.mumps_struc_c_x

aux  
Structure/Union member

comm_fortran  
Structure/Union member

icntl  
Structure/Union member

job  
Structure/Union member

par  
Structure/Union member

sym  
Structure/Union member

sfepy.solvers.ls_mumps_parallel module

sfepy.solvers.ls_mumps_parallel.mumps_parallel_solve()

sfepy.solvers.ls_mumps_parallel.tmpfile(fname)
sfepy.solvers.nls module

Nonlinear solvers.

class sfepy.solvers.nls.Newton(conf, **kwargs)

Solves a nonlinear system \( f(x) = 0 \) using the Newton method.

The solver uses a backtracking line-search on divergence.

Kind: ‘nls.newton’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters

  i_max
  [int (default: 1)] The maximum number of iterations.

  eps_a
  [float (default: 1e-10)] The absolute tolerance for the residual, i.e. \( ||f(x^i)|| \).

  eps_r
  [float (default: 1.0)] The relative tolerance for the residual, i.e. \( ||f(x^i)||/||f(x^0)|| \).

  eps_mode
  ['and' or 'or' (default: ‘and’)] The logical operator to use for combining the absolute and relative tolerances.

  macheps
  [float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

  lin_red
  [float (default: 1.0)] The linear system solution error should be smaller than (eps_a * lin_red), otherwise a warning is printed.

  lin_precision
  [float or None] If not None, the linear system solution tolerances are set in each nonlinear iteration relative to the current residual norm by the lin_precision factor. Ignored for direct linear solvers.

  step_red
  [0.0 < float <= 1.0 (default: 1.0)] Step reduction factor. Equivalent to the mixing parameter \( (1-a)x + a(x + dx) = x + adx \)

  ls_on
  [float (default: 0.999999)] Start the backtracking line-search by reducing the step, if \( ||f(x^i)||/||f(x^{i-1})|| \) is larger than ls_on.

  ls_red
  [0.0 < float < 1.0 (default: 0.1)] The step reduction factor in case of correct residual assembling.

  ls_red_warp
  [0.0 < float < 1.0 (default: 0.001)] The step reduction factor in case of failed residual assembling (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

  ls_min
  [0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.
give_up_warp
   [bool (default: False)] If True, abort on the “warp violation” error.

check
   [0, 1 or 2 (default: 0)] If >= 1, check the tangent matrix using finite differences. If 2, plot
   the resulting sparsity patterns.

delta
   [float (default: 1e-06)] If check >= 1, the finite difference matrix is taken as
   \[ A_{ij} = \frac{f_i(x_j + \delta) - f_i(x_j - \delta)}{2\delta} \].

log
   [dict or None] If not None, log the convergence according to the configuration in the following form:
   \{ 'text' : 'log.txt', 'plot' : 'log.pdf' \}. Each of the dict items can be None.

is_linear
   [bool (default: False)] If True, the problem is considered to be linear.

__call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None)
   Nonlinear system solver call.
   Solves a nonlinear system \( f(x) = 0 \) using the Newton method with backtracking line-search, starting with
   an initial guess \( x^0 \).

Parameters

   vec_x0
      [array] The initial guess vector \( x_0 \).

   conf
      [Struct instance, optional] The solver configuration parameters,

   fun
      [function, optional] The function \( f(x) \) whose zero is sought - the residual.

   fun_grad
      [function, optional] The gradient of \( f(x) \) - the tangent matrix.

   lin_solver
      [LinearSolver instance, optional] The linear solver for each nonlinear iteration.

   iter_hook
      [function, optional] User-supplied function to call before each iteration.

   status
      [dict-like, optional] The user-supplied object to hold convergence statistics.

Notes

   • The optional parameters except iter_hook and status need to be given either here or upon Newton
     construction.

   • Setting conf.is_linear == True means a pre-assembled and possibly pre-solved matrix. This is mostly
     useful for linear time-dependent problems.

__init__(conf, **kwargs)

__module__ = 'sfepy.solvers.nls'
name = 'nls.newton'

class sfepy.solvers.nls.PETScNonlinearSolver(conf, pmtx=None, prhs=None, comm=None, **kwargs)

   Interface to PETSc SNES (Scalable Nonlinear Equations Solvers).

   The solver supports parallel use with a given MPI communicator (see comm argument of PETScNonlinearSolver.__init__()). Returns a (global) PETSc solution vector instead of a (local) numpy array, when given a PETSc initial guess vector.

   For parallel use, the fun and fun_grad callbacks should be provided by PETScParallelEvaluator.

   Kind: 'nls.petsc'

   For common configuration parameters, see Solver.

   Specific configuration parameters:

      Parameters

         method
            [str (default: 'newtonls')] The SNES type.

         i_max
            [int (default: 10)] The maximum number of iterations.

         if_max
            [int (default: 100)] The maximum number of function evaluations.

         eps_a
            [float (default: 1e-10)] The absolute tolerance for the residual, i.e. $||f(x^i)||$.

         eps_r
            [float (default: 1.0)] The relative tolerance for the residual, i.e. $||f(x^i)||/||f(x^0)||$.

         eps_s
            [float (default: 0.0)] The convergence tolerance in terms of the norm of the change in the solution between steps, i.e. $||\delta x|| < \text{epsilon}_s ||x||$

   __call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None, pmtx=None, prhs=None, comm=None)

      Call self as a function.

   __init__(conf, pmtx=None, prhs=None, comm=None, **kwargs)

   __module__ = 'sfepy.solvers.nls'

   name = 'nls.petsc'

class sfepy.solvers.nls.ScipyBroyden(conf, **kwargs)

   Interface to Broyden and Anderson solvers from scipy.optimize.

   Kind: 'nls.scipy_broyden_like'

   For common configuration parameters, see Solver.

   Specific configuration parameters:

      Parameters

         method
            [str (default: 'anderson')] The name of the solver in scipy.optimize.

         i_max
            [int (default: 10)] The maximum number of iterations.
alpha
[float (default: 0.9)] See scipy.optimize.

M
[float (default: 5)] See scipy.optimize.

f_tol
[float (default: 1e-06)] See scipy.optimize.

w0
[float (default: 0.1)] See scipy.optimize.

__call__(vec_x0, conf=None, fun=None, fun_grad=None, lin_solver=None, iter_hook=None, status=None)
Call self as a function.

__init__(conf, **kwargs)

__module__ = 'sfepy.solvers.nls'
nname = 'nls.scipy_broyden_like'

set_method(conf)

sfepy.solvers.nls."check_tangent_matrix"(conf, vec_x0, fun, fun_grad)
Verify the correctness of the tangent matrix as computed by fun_grad() by comparing it with its finite difference approximation evaluated by repeatedly calling fun() with vec_x0 items perturbed by a small delta.

sfepy.solvers.nls."conv_test"(conf, it, err, err0)
Nonlinear solver convergence test.

Parameters
conf
[Struct instance] The nonlinear solver configuration.

it
[int] The current iteration.

err
[float] The current iteration error.

err0
[float] The initial error.

Returns

status
[int] The convergence status: -1 = no convergence (yet), 0 = solver converged - tolerances were met, 1 = max. number of iterations reached.

sfepy.solvers.optimize module

class sfepy.solvers.optimize.FMinSteepestDescent(conf, **kwargs)
Steepest descent optimization solver.

Kind: 'opt.fmin_sd'

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters
i_max
  [int (default: 10)] The maximum number of iterations.

eps_rd
  [float (default: 1e-05)] The relative delta of the objective function.

eps_of
  [float (default: 0.0001)] The tolerance for the objective function.

eps_ofg
  [float (default: 1e-08)] The tolerance for the objective function gradient.

norm
  [numpy norm (default: inf)] The norm to be used.

ls
  [bool (default: True)] If True, use a line-search.

ls_method
  [{‘backtracking’, ‘full’} (default: ‘backtracking’)] The line-search method.

ls_on
  [float (default: 0.99999)] Start the backtracking line-search by reducing the step, if
  \(|f(x^i)|/|f(x^{i-1})|\) is larger than \(ls_on\).

ls0
  [0.0 < float < 1.0 (default: 1.0)] The initial step.

ls_red
  [0.0 < float < 1.0 (default: 0.5)] The step reduction factor in case of correct residual assembling.

ls_red_warp
  [0.0 < float < 1.0 (default: 0.1)] The step reduction factor in case of failed residual assembling
  (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

ls_min
  [0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.

check
  [0, 1 or 2 (default: 0)] If >= 1, check the tangent matrix using finite differences. If 2, plot
  the resulting sparsity patterns.

delta
  [float (default: 1e-06)] If check >= 1, the finite difference matrix is taken as
  \(A_{ij} = \frac{f_i(x_j + \delta) - f_i(x_j - \delta)}{2 \delta}\).

output
  [function] If given, use it instead of output() function.

yscales
  [list of str (default: [‘linear’, ‘log’, ‘log’, ‘linear’])] The list of four convergence log subplot scales.

log
  [dict or None] If not None, log the convergence according to the configuration in the following form:
  {‘text’ : ’log.txt’, ‘plot’ : ’log.pdf’}. Each of the dict items can be None.

name = ‘opt.fmin_sd’
class sfepy.solvers.optimize.ScipyFMinSolver(conf, **kwargs)

Interface to SciPy optimization solvers scipy.optimize.fmin_*.
Kind: ‘nls.scipy_fmin_like’

For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

**method**

‘fmin_powell’, ‘fmin_slsqp’, ‘fmin_tnc’] (default: ‘fmin’)] The actual optimization
method to use.

**i_max**

[int (default: 10)] The maximum number of iterations.

* Additional parameters supported by the method.

name = 'nls.scipy_fmin_like'

set_method(conf)

sfepy.solvers.optimize.check_gradient(xit, aofg, fn_of, delta, check)

sfepy.solvers.optimize.conv_test(conf, it, of, of0, ofg_norm=None)

Returns

**flag**

[int]

• -1 ... continue

• 0 ... small OF -> stop

• 1 ... i_max reached -> stop

• 2 ... small OFG -> stop

• 3 ... small relative decrase of OF

sfepy.solvers.optimize.wrap_function(function, args)

sfepy.solvers.oseen module

class sfepy.solvers.oseen.Oseen(conf, context=None, **kwargs)

The Oseen solver for Navier-Stokes equations.
Kind: ‘nls.oseen’

For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

**stabil_mat**

[str] The name of stabilization material.
adimensionalize
[bool (default: False)] If True, adimensionalize the problem (not implemented!).

check_navier_stokes_residual
[bool (default: False)] If True, check the Navier-Stokes residual after the nonlinear loop.

i_max
[int (default: 1)] The maximum number of iterations.

eps_a
[float (default: 1e-10)] The absolute tolerance for the residual, i.e. $||f(x^i)||$.

eps_r
[float (default: 1.0)] The relative tolerance for the residual, i.e. $||f(x^i)||/||f(x^0)||$.

macheps
[float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

lin_red
[float (default: 1.0)] The linear system solution error should be smaller than $(\text{eps}_a \times \text{lin}_red)$, otherwise a warning is printed.

lin_precision
[float or None] If not None, the linear system solution tolerances are set in each nonlinear iteration relative to the current residual norm by the lin_precision factor. Ignored for direct linear solvers.

name = 'nls.oseen'

class sfepy.solvers.oseen.StabilizationFunction(name_map, gamma=None, delta=None, tau=None, tau_red=1.0, tau_mul=1.0, delta_mul=1.0, gamma_mul=1.0, diameter_mode='max')

Definition of stabilization material function for the Oseen solver.

Notes

• tau_red <= 1.0; if tau is None: tau = tau_red * delta

• diameter mode: ‘edge’: longest edge ‘volume’: volume-based, ‘max’: max. of previous

get_maps()
Get the maps of names and indices of variables in state vector.

setup(problem)
Setup common problem-dependent data.

sfepy.solvers.oseen.are_close(a, b, rtol=0.2, atol=1e-08)
sfepy.solvers.oseen.scale_matrix(mtx, indx, factor)
**sfepy.solvers.qeigen module**

Quadratic eigenvalue problem solvers.

```python
class sfepy.solvers.qeigen.LQuadraticEVPSolver:
    Quadratic eigenvalue problem solver based on the problem linearization.
    \((w^2 M + w D + K) x = 0\).
    Kind: 'eig.qevp'
    For common configuration parameters, see `Solver`.
    Specific configuration parameters:
    Parameters
    method
        [['companion', 'cholesky'] (default: 'companion')] The linearization method.
    solver
        [dict (default: {'kind': 'eig.scipy', 'method': 'eig'})] The configuration of an eigenvalue
        solver for the linearized problem \((A - w B) x = 0\).
    mode
        [['normal', 'inverted'] (default: 'normal')] Solve either \(A - w B\) (normal), or \(B - 1/w A\)
        (inverted).
    debug
        [bool (default: False)] If True, print debugging information.

name = 'eig.qevp'
```

**sfepy.solvers.semismooth_newton module**

```python
class sfepy.solvers.semismooth_newton.SemismoothNewton:
    The semi-smooth Newton method.
    This method is suitable for solving problems of the following structure:
    \[ F(y) = 0 \]
    \[ A(y) \geq 0, \quad B(y) \geq 0, \quad \langle A(y), B(y) \rangle = 0 \]
    The function \(F(y)\) represents the smooth part of the problem.
    Regular step: \(y \leftarrow y - J(y)^{-1} \Phi(y)\)
    Steepest descent step: \(y \leftarrow y - \beta J(y) \Phi(y)\)
    Although `fun_smooth_grad()` computes the gradient of the smooth part only, it should return the global matrix,
    where the non-smooth part is uninitialized, but pre-allocated.
    Kind: 'nls.semismooth_newton'
    For common configuration parameters, see `Solver`.
    Specific configuration parameters:
```

432 Chapter 2. Development
Parameters

**semismooth**
- [bool (default: True)] If True, use the semi-smooth algorithm. Otherwise a non-smooth equation is assumed (use a brute force).

**i_max**
- [int (default: 1)] The maximum number of iterations.

**eps_a**
- [float (default: 1e-10)] The absolute tolerance for the residual, i.e. \( ||f(x^i)|| \).

**eps_r**
- [float (default: 1.0)] The relative tolerance for the residual, i.e. \( \frac{||f(x^i)||}{||f(x^0)||} \).

**macheps**
- [float (default: 2.220446049250313e-16)] The float considered to be machine “zero”.

**lin_red**
- [float (default: 1.0)] The linear system solution error should be smaller than \( (eps_a \times lin_red) \), otherwise a warning is printed.

**ls_on**
- [float (default: 0.99999)] Start the backtracking line-search by reducing the step, if \( \frac{||f(x^i)||}{||f(x^{i-1})||} \) is larger than \( ls_on \).

**ls_red**
- [dict (default: {‘regular’: 0.1, ‘steepest_descent’: 0.01})] The step reduction factor in case of correct residual assembling for regular and steepest descent modes.

**ls_red_warp**
- [0.0 < float < 1.0 (default: 0.001)] The step reduction factor in case of failed residual assembling (e.g. the “warp violation” error caused by a negative volume element resulting from too large deformations).

**ls_min**
- [0.0 < float < 1.0 (default: 1e-05)] The minimum step reduction factor.

**compute_jacobian**
- (vec_x, fun_smooth_grad, fun_a_grad, fun_b_grad, vec_smooth_r, vec_a_r, vec_b_r)

**name** = 'nls.semismooth_newton'

### sfepy.solvers.solvers module

Base (abstract) solver classes.

**class sfepy.solvers.solvers.EigenvalueSolver**

Base (abstract) solver classes.

**class sfepy.solvers.solvers.LinearSolver**

Abstract eigenvalue solver class.

**class sfepy.solvers.solvers.LinearSolver**

Abstract linear solver class.

**clear()**

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, status=None, context=None, **kwargs)

Abstract nonlinear solver class.

class sfepy.solvers.solvers.OptimizationSolver(conf, obj_fun=None, obj_fun_grad=None, status=None, obj_args=None, context=None, **kwargs)

Abstract optimization solver class.

class sfepy.solvers.solvers.QuadraticEVPSolver(conf, mtx_m=None, mtx_d=None, mtx_k=None, n_eigs=None, eigenvectors=None, status=None, context=None, **kwargs)

Abstract quadratic eigenvalue problem solver class.

class sfepy.solvers.solvers.Solver(conf=None, context=None, **kwargs)

Base class for all solver kinds. Takes care of processing of common configuration options.
The factory method any_from_conf() can be used to create an instance of any subclass.
The subclasses have to reimplement __init__() and __call__().

All solvers use the following configuration parameters:

Parameters

name

[str] The name referred to in problem description options.

kind

[str] The solver kind, as given by the name class attribute of the Solver subclasses.

verbose

[bool (default: False)] If True, the solver can print more information about the solution.

static any_from_conf(conf, **kwargs)

Create an instance of a solver class according to the configuration.

build_solver_kwargs(conf)

Build the kwargs dict for the underlying solver function using the extra options (marked by ‘*’ in _parameters) in conf. The declared parameters are omitted.

classmethod process_conf(conf, kwargs)

Process configuration parameters.

set_field_split(field_ranges, **kwargs)

class sfepy.solvers.solvers.SolverMeta(name, bases, ndict)

Metaclass for solver classes that automatically adds configuration parameters to the solver class docstring from the _parameters class attribute.

class sfepy.solvers.solvers.TimeStepController(conf, **kwargs)

Abstract time step controller class.

get_initial_dt(ts, vec, **kwargs)

class sfepy.solvers.solvers.TimeSteppingSolver(conf, nls=None, status=None, tsc=None, context=None, **kwargs)

Abstract time stepping solver class.
set_dof_info(di)

sfepy.solvers.solvers.format_next(text, new_text, pos, can_newline, width, ispaces)

sfepy.solvers.solvers.make_get_conf(conf, kwargs)

sfepy.solvers.solvers.make_option_docstring(name, kind, default, required, doc)

sfepy.solvers.solvers.typeset_to_indent(txt, indent, width)

sfepy.solvers.solvers.use_first_available(solver_list, context=None, **kwargs)

Use the first available solver from `solver_list`.

Parameters

- `solver_list`
  [list of str or Struct] The list of solver names or configuration objects.

- `context`
  [object, optional] An optional solver context to pass to the solver.

- `**kwargs`
  [keyword arguments] Additional solver options, see the particular `__init__()` methods.

Returns

- `out`
  [Solver] The first available solver.

**sfepy.solvers.ts module**

**class sfepy.solvers.ts.TimeStepper**(t0, t1, dt=None, n_step=None, step=None, is_quasistatic=False)

Time stepper class.

- `advance()`
- `static from_conf(conf)`
- `get_state()`
- `iter_from(step)`
- `normalize_time()`
- `restore_step_time()`
- `set_from_data(t0, t1, dt=None, n_step=None, step=None)`
- `set_from_ts(ts, step=None)`
- `set_state(step=0, **kwargs)`
- `set_step(step=0, nt=0.0)`
- `set_substep_time(sub_dt)`

**class sfepy.solvers.ts.VariableTimeStepper**(t0, t1, dt=None, n_step=None, step=None, is_quasistatic=False)

Time stepper class with a variable time step.
advance()

static from_conf(conf)

get_default_time_step()

get_state()

iter_from(step)

iter_from_current()
    ts.step, ts.time is consistent with step, time returned here ts.nt is normalized time in [0, 1].

set_from_data(t0, t1, dt=None, n_step=None, step=None)

set_from_ts(ts, step=None)

set_n_digit_from_min_dt(dt)

set_state(step=0, dt=None, times=None, **kwargs)

set_step(step=0, nt=0, 0)

set_time_step(dt, update_time=False)

sfepy.solvers.ts.get_print_info(n_step)

**sfepy.solvers.ts_controllers module**

Time step controllers.

class sfepy.solvers.ts_controllers.ElastodynamicsBasicTSC(conf, **kwargs)

Adaptive time step I-controller for elastodynamics.

The implementation is based on [1]. The default parameters correspond to the PID-Controller as implemented in tsc.ed_pid with P=D=0, I=1.


Kind: ‘tsc.ed_basic’

For common configuration parameters, see Solver.

Specific configuration parameters:

**Parameters**

eps_r
    [list of floats or float] Relative tolerance(s).

eps_a
    [list of floats or float] Absolute tolerance(s).

fmin
    [float (default: 0.3)] Minimum step size change factor on step rejection.

fmax
    [float (default: 2.5)] Maximum step size change factor on step acceptance.
fsafety
  [float (default: 0.8)] Step size change safety factor.

error_order
  [float (default: 2)] The order of the solver error estimate.

guess_dt0
  [bool (default: False)] Guess a good initial step size from initial conditions.

get_initial_dt(ts, vec, unpack, **kwargs)
  Adapted from [1] for second order ODEs.

[1] Hairer, Ernst, Gerhard Wanner, and Syvert P. Nørsett. Solving Ordinary Differential Equations I: Non-
https://doi.org/10.1007/978-3-540-78862-1.

static get_scaled_errors(dt, vec0, vec1, eps_r, eps_a, eps_rs, unpack)

name = 'tsc.ed_basic'

class sfepy.solvers.ts_controllers.ElastodynamicsLinearTSC(conf, **kwargs)
  Adaptive time step controller for elastodynamics and linear problems.

Simple heuristics around ElastodynamicsBasicTSC that increases the step size only after a sufficient number
of accepted iterations passed and the increase is large enough. In particular:

• Let new_dt be the step size proposed by tsc.ed_basic and dt the current step size.
• If the current step is rejected, the count attribute is reset to zero and fred * new_dt is returned.
• If the current step is accepted:
  – If the count is lower than inc_wait, it is incremented and dt is returned.
  – Otherwise, if (new_dt / dt) >= min_finc (>= 1), the count is reset to zero and new_dt is returned.
  – Else, if (new_dt / dt) < min_finc, dt is returned.

Kind: ‘tsc.ed_linear’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters

eps_r
  [list of floats or float] Relative tolerance(s).

eps_a
  [list of floats or float] Absolute tolerance(s).

fmin
  [float (default: 0.3)] Minimum step size change factor on step rejection.

fmax
  [float (default: 2.5)] Maximum step size change factor on step acceptance.

fsafety
  [float (default: 0.8)] Step size change safety factor.

error_order
  [float (default: 2)] The order of the solver error estimate.

guess_dt0
  [bool (default: False)] Guess a good initial step size from initial conditions.
fred

- Additional step size reduction factor w.r.t. `tsc.ed_basic`.

`inc_wait`

- The number of consecutive accepted steps to wait before increasing the step size.

`min_finc`

- Minimum step size increase factor.

- Minimum step size increase factor.

`name = 'tsc.ed_linear'`

```python
class sfepy.solvers.ts_controllers.ElastodynamicsPIDTSC(conf, **kwargs)
```

Adaptive time step PID controller for elastodynamics.

The implementation is based on [1], [2] (PI Controller) and [3] (PID). The default parameters correspond to the I-Controller as implemented in `tsc.ed_basic`.


Kind: ‘tsc.ed_pid’

For common configuration parameters, see `Solver`.

Specific configuration parameters:

**Parameters**

- `eps_r`
  - Relative tolerance(s).
- `eps_a`
  - Absolute tolerance(s).
- `fmin`
  - Minimum step size change factor on step rejection.
- `fmax`
  - Maximum step size change factor on step acceptance.
- `fsafety`
  - Step size change safety factor.
- `error_order`
  - The order of the solver error estimate.
- `guess_dt0`
  - Guess a good initial step size from initial conditions.
- `pcoef`
  - Proportional (P) coefficient of the step size control.
- `icoef`
  - Integral (I) coefficient of the step size control.
- `dcoef`
  - Derivative (D) coefficient of the step size control.
name = 'tsc.ed_pid'

class sfepy.solvers.ts_controllers.FixedTSC(conf, **kwargs)
    Fixed (do-nothing) time step controller.
    Kind: ‘tsc.fixed’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    name = 'tsc.fixed'

class sfepy.solvers.ts_controllers.TimesSequenceTSC(conf, **kwargs)
    Given times sequence time step controller.
    Kind: ‘tsc.time_sequence’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    Parameters
    times
        [iterable (default: range(1, 6))] A sequence of times to generate.
    get_initial_dt(ts, vec, **kwargs)
    name = 'tsc.time_sequence'

sfepy.solvers.ts_controllers.eval_scaled_norm(terr, eps_a, eps_r)

sfepy.solvers.ts_solvers module

Time stepping solvers.

class sfepy.solvers.ts_solvers.AdaptiveTimeSteppingSolver(conf, nls=None, context=None, **kwargs)
    Implicit time stepping solver with an adaptive time step.
    Either the built-in or user supplied function can be used to adapt the time step.
    Kind: ‘ts.adaptive’
    For common configuration parameters, see Solver.
    Specific configuration parameters:
    Parameters
    t0
        [float (default: 0.0)] The initial time.
    t1
        [float (default: 1.0)] The final time.
    dt
        [float] The time step. Used if n_step is not given.
    n_step
        [int (default: 10)] The number of time steps. Has precedence over dt.
quasistatic
   [bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver
   is invoked also for the initial time.

adapt_fun
   [callable(ts, status, adt, context, verbose)] If given, use this function to set the time step in
   ts. The function return value is a bool - if True, the adaptivity loop should stop. The other
   parameters below are collected in adt, status is the nonlinear solver status, context is a user-
   defined context and verbose is a verbosity flag. Solvers created by Problem use the Problem
   instance as the context.

dt_red_factor
   [float (default: 0.2)] The time step reduction factor.

dt_red_max
   [float (default: 0.001)] The maximum time step reduction factor.

dt_inc_factor
   [float (default: 1.25)] The time step increase factor.

dt_inc_on_iter
   [int (default: 4)] Increase the time step if the nonlinear solver converged in less than this
   amount of iterations for dt_inc_wait consecutive time steps.

dt_inc_wait
   [int (default: 5)] The number of consecutive time steps, see dt_inc_on_iter.

name = 'ts.adaptive'

output_step_info(ts)

solve_step(ts, nls, vec, prestep_fun)
   Solve a single time step.

class sfepy.solvers.ts_solvers.BatheTS(conf, nls=None, context=None, **kwargs)
   Solve elastodynamics problems by the Bathe method.

   The method was introduced in [1].

   [1] Klaus-Juergen Bathe, Conserving energy and momentum in nonlinear dynamics: A simple implicit time
   https://doi.org/10.1016/j.compstruc.2006.09.004.

   Kind: ‘ts.bathe’

   For common configuration parameters, see Solver.

   Specific configuration parameters:

   Parameters

   t0
      [float (default: 0.0)] The initial time.

   t1
      [float (default: 1.0)] The final time.

   dt
      [float] The time step. Used if n_step is not given.

   n_step
      [int (default: 10)] The number of time steps. Has precedence over dt.
**is_linear**

[bool (default: False)] If True, the problem is considered to be linear.

**has_time_derivatives**

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

**var_names**

[dict] The mapping of variables with keys ‘u’, ‘du’, ‘ddu’ and ‘extra’, and values corresponding to the names of the actual variables. See var_names returned from transform_equations_ed()

**clear_lin_solver**

(\texttt{clear\_constant\_matrices=True})

**create_nlst1**

(\texttt{nls, dt, u0, v0, a0})

The first sub-step: the trapezoidal rule.

**create_nlst2**

(\texttt{nls, dt, u0, u1, v0, v1})

The second sub-step: the three-point Euler backward method.

**name** = \texttt{’ts.bathe’}

**step**

(\texttt{ts, vec, nls, pack, unpack, prestep\_fun})

Solve a single time step.

**class sfepy.solvers.ts_solvers.CentralDifferenceTS**

Solve elastodynamics problems by the explicit central difference method.

It is the same method as obtained by using NewmarkTS with $\beta = 0, \gamma = 1/2$, but uses a simpler code.

It is also mathematically equivalent to the VelocityVerletTS method. The current implementation code is essentially the same.

This solver supports, when used with RMMSolver, the reciprocal mass matrix algorithm, see MassTerm.

Kind: ‘ts.central\_difference’

For common configuration parameters, see Solver.

Specific configuration parameters:

**Parameters**

\texttt{t0}

[float (default: 0.0)] The initial time.

\texttt{t1}

[float (default: 1.0)] The final time.

\texttt{dt}

[float] The time step. Used if \texttt{n\_step} is not given.

\texttt{n\_step}

[int (default: 10)] The number of time steps. Has precedence over \texttt{dt}.

\texttt{is\_linear}

[bool (default: False)] If True, the problem is considered to be linear.
has_time_derivatives
[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names
[dict] The mapping of variables with keys ‘u’, ‘du’, ‘ddu’ and ‘extra’, and values corresponding to the names of the actual variables. See var_names returned from transform_equations_ed()

create_nlst(nls, dt, u0, v0, a0)

name = 'ts.central_difference'

step(ts, vec, nls, pack, unpack, **kwargs)
Solve a single time step.

class sfepy.solvers.ts_solvers.ElastodynamicsBaseTS(conf, nls=None, tsc=None, context=None, **kwargs)
Base class for elastodynamics solvers.
Assumes block-diagonal matrix in u, v, a.

clear_lin_solver(clear_constant_matrices=True)

get_a0(nls, u0, e0, v0, unpack)

get_initial_vec(nls, vec0, init_fun, prestep_fun, poststep_fun)

get_matrices(nls, vec, unpack=None)

class sfepy.solvers.ts_solvers.GeneralizedAlphaTS(conf, nls=None, tsc=None, context=None, **kwargs)
Solve elastodynamics problems by the generalized α method.

• The method was introduced in [1].
• The method is unconditionally stable provided \( \alpha_m \leq \alpha_f \leq \frac{1}{2} \), \( \beta \geq \frac{1}{4} + \frac{1}{2}(\alpha_f - \alpha_m) \).
• The method is second-order accurate provided \( \gamma = \frac{1}{2} - \alpha_m + \alpha_f \). This is used when gamma is None.
• High frequency dissipation is maximized for \( \beta = \frac{1}{4}(1 - \alpha_m + \alpha_f)^2 \). This is used when beta is None.
• The default values of \( \alpha_m \), \( \alpha_f \) (if alpha_m or alpha_f are None) are based on the user specified high-frequency dissipation parameter rho_inf.

Special settings:
• \( \alpha_m = 0 \) corresponds to the HHT-α method.
• \( \alpha_f = 0 \) corresponds to the WBZ-α method.
• \( \alpha_m = 0, \alpha_f = 0 \) produces the Newmark method.


Kind: ‘ts.generalized_alpha’

For common configuration parameters, see Solver.

Specific configuration parameters:

Parameters
rho_inf
[ float (default: 0.5)] The spectral radius in the high frequency limit (user specified high-frequency dissipation) in \([0, 1]\): 1 = no dissipation, 0 = asymptotic annihilation.

alpha_m
[ float] The parameter \(\alpha_m\).

alpha_f
[ float] The parameter \(\alpha_f\).

beta
[ float] The Newmark-like parameter \(\beta\).

gamma
[ float] The Newmark-like parameter \(\gamma\).

t0
[ float (default: 0.0)] The initial time.

t1
[ float (default: 1.0)] The final time.

dt
[ float] The time step. Used if n_step is not given.

n_step
[ int (default: 10)] The number of time steps. Has precedence over dt.

is_linear
[ bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives
[ bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names
[ dict] The mapping of variables with keys ‘u’, ‘du’, ‘ddu’ and ‘extra’, and values corresponding to the names of the actual variables. See var_names returned from transform_equations_ed()

create_nlst(nls, dt, alpha_m, alpha_f, gamma, beta, u0, v0, a0)

name = 'ts.generalized_alpha'

step(ts, vec, nls, pack, unpack, **kwargs)
Solve a single time step.

class sfepy.solvers.ts_solvers.NewmarkTS(conf, nls=None, tsc=None, context=None, **kwargs)
Solve elastodynamics problems by the Newmark method.

The method was introduced in [1]. Common settings [2]:

<table>
<thead>
<tr>
<th>name</th>
<th>kind</th>
<th>beta</th>
<th>gamma</th>
<th>Omega_crit</th>
</tr>
</thead>
<tbody>
<tr>
<td>trapezoidal rule:</td>
<td>implicit</td>
<td>1/4</td>
<td>1/2</td>
<td>unconditional</td>
</tr>
<tr>
<td>linear acceleration:</td>
<td>implicit</td>
<td>1/6</td>
<td>1/2</td>
<td>2√3</td>
</tr>
<tr>
<td>Fox-Goodwin:</td>
<td>implicit</td>
<td>1/12</td>
<td>1/2</td>
<td>√6</td>
</tr>
<tr>
<td>central difference:</td>
<td>explicit</td>
<td>0</td>
<td>1/2</td>
<td>2</td>
</tr>
</tbody>
</table>

All of these methods are 2-order of accuracy.

Kind: ‘ts.newmark’
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters

- **beta**
  - [float (default: 0.25)] The Newmark method parameter beta.

- **gamma**
  - [float (default: 0.5)] The Newmark method parameter gamma.

- **t0**
  - [float (default: 0.0)] The initial time.

- **t1**
  - [float (default: 1.0)] The final time.

- **dt**
  - [float] The time step. Used if n_step is not given.

- **n_step**
  - [int (default: 10)] The number of time steps. Has precedence over dt.

- **is_linear**
  - [bool (default: False)] If True, the problem is considered to be linear.

- **has_time_derivatives**
  - [bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

- **var_names**
  - [dict] The mapping of variables with keys ‘u’, ‘du’, ‘ddu’ and ‘extra’, and values corresponding to the names of the actual variables. See var_names returned from transform_equations_ed()

create_nlst(nls, dt, gamma, beta, u0, e0, v0, a0, pack, unpack)

extra_variables = True

name = 'ts.newmark'

step(ts, vec, nls, pack, unpack, **kwargs)
Solve a single time step.

class sfepy.solvers.ts_solvers.SimpleTimeSteppingSolver(conf, nls=None, context=None, **kwargs)
Implicit time stepping solver with a fixed time step.
Kind: ‘ts.simple’
For common configuration parameters, see Solver.
Specific configuration parameters:

Parameters
Parameters

t0
[Float (default: 0.0)] The initial time.

t1
[Float (default: 1.0)] The final time.

dt
[Float] The time step. Used if \textit{n\_step} is not given.

\textbf{n\_step}
[int (default: 10)] The number of time steps. Has precedence over \textit{dt}.

\textbf{quasistatic}
[bool (default: False)] If True, assume a quasistatic time-stepping. Then the non-linear solver is invoked also for the initial time.

name = 'ts.simple'

output_step_info\((ts)\)

solve_step\((ts, nls, vec, prestep\_fun=None)\)

solve_step0\((nls, vec0)\)

class \texttt{sfepy.solvers.ts\_solvers.StationarySolver}(conf, nls=None, context=None, **kwargs)
Solver for stationary problems without time stepping.
This class is provided to have a unified interface of the time stepping solvers also for stationary problems.
Kind: ‘ts.stationary’
For common configuration parameters, see \texttt{Solver}.
Specific configuration parameters:
name = 'ts.stationary'

class \texttt{sfepy.solvers.ts\_solvers.VelocityVerletTS}(conf, nls=None, tsc=None, context=None, **kwargs)
Solve elastodynamics problems by the explicit velocity-Verlet method.
The algorithm can be found in [1].
It is mathematically equivalent to the \texttt{CentralDifferenceTS} method. The current implementation code is essentially the same, as the mid-time velocities are not used for anything other than computing the new time velocities.
Kind: ‘ts.velocity\_verlet’
For common configuration parameters, see \texttt{Solver}.
Specific configuration parameters:

Parameters

t0
[Float (default: 0.0)] The initial time.

t1
[Float (default: 1.0)] The final time.

dt
[Float] The time step. Used if \textit{n\_step} is not given.

\textbf{n\_step}
[int (default: 10)] The number of time steps. Has precedence over \textit{dt}.
is_linear

[bool (default: False)] If True, the problem is considered to be linear.

has_time_derivatives

[bool (default: False)] If True, the problem equations contain time derivatives of other variables besides displacements. In that case the cached constant matrices must be cleared on time step changes.

var_names

[dict] The mapping of variables with keys ‘u’, ‘du’, ‘ddu’ and ‘extra’, and values corresponding to the names of the actual variables. See var_names returned from transform_equations_ed()

create_nlst(nls, dt, u0, v0, a0)

name = 'ts.velocity_verlet'

step(ts, vec, nls, pack, unpack, **kwargs)

Solve a single time step.

sfepy.solvers.ts_solvers.adapt_time_step(ts, status, adt, context=None, verbose=False)

Adapt the time step of ts according to the exit status of the nonlinear solver.

The time step dt is reduced, if the nonlinear solver did not converge. If it converged in less than a specified number of iterations for several time steps, the time step is increased. This is governed by the following parameters:

- red_factor : time step reduction factor
- red_max : maximum time step reduction factor
- inc_factor : time step increase factor
- inc_on_iter : increase time step if the nonlinear solver converged in less than this amount of iterations...
- inc_wait : ...for this number of consecutive time steps

Parameters

ts

[VariableTimeStepper instance] The time stepper.

status

[IndexedStruct instance] The nonlinear solver exit status.

adt

[Struct instance] The object with the adaptivity parameters of the time-stepping solver such as red_factor (see above) as attributes.

context

[object, optional] The context can be used in user-defined adaptivity functions. Not used here.

Returns

is_break

[bool] If True, the adaptivity loop should stop.

sfepy.solvers.ts_solvers.gen_multi_vec_packing(di, names, extra_variables=False)

Return DOF vector (un)packing functions for nlst. For multiphysical problems (non-empty ie slice for extra variables) the unpack() function accepts an additional argument mode that can be set to ‘full’ or ‘nls’.

The following DOF ordering must be obeyed:
• The full DOF vector:

---iue---|-iv-|-ia-
-iu-|-ie-|

`sfepy.solvers.ts_solvers.get_min_dt(adt)`

`sfepy.solvers.ts_solvers.standard_ts_call(call)`
Decorator handling argument preparation and timing for time-stepping solvers.

`sfepy.solvers.ts_solvers.transform_equations_ed(equations, materials)`
Transform equations and variables for `ElastodynamicsBaseTS`-based time stepping solvers. The displacement variable name is automatically detected by seeking the second time derivative, i.e. the ‘dd’ prefix in variable names.

`sfepy.terms package`

`sfepy.terms.terms module`

`class sfepy.terms.terms.ConnInfo(**kwargs)`

`get_region(can_trace=True)`
`get_region_name(can_trace=True)`

`class sfepy.terms.terms.Term(name, arg_str, integral, region, **kwargs)`

`advance(ts)`
Advance to the next time step. Implemented in subclasses.

`arg_shapes = {}`
`arg_types = []`

`assemble_to(asm_obj, val, iels, mode='vector', diff_var=None)`
Assemble the results of term evaluation.

For standard terms, assemble the values in `val` corresponding to elements/cells `iels` into a vector or a CSR sparse matrix `asm_obj`, depending on `mode`.

For terms with a dynamic connectivity (e.g. contact terms), in ‘matrix’ mode, return the extra COO sparse matrix instead. The extra matrix has to be added to the global matrix by the caller. By default, this is done in `Equations.evaluate()`.

`assign_args(variables, materials, user=None)`
Check term argument existence in variables, materials, user data and assign the arguments to terms. Also check compatibility of field and term regions.

`call_function(out, fargs)`

`call_get_fargs(args, kwargs)`
check_args()
    Common checking to all terms.
    Check compatibility of field and term regions.
check_shapes(*args, **kwargs)
    Check term argument shapes at run-time.
classify_args()
    Classify types of the term arguments and find matching call signature.
    A state variable can be in place of a parameter variable and vice versa.
diff_info = {}
eval_complex(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
evaluate(mode='eval', diff_var=None, standalone=True, ret_status=False, **kwargs)
    Evaluate the term.
    Parameters
    mode
        ['eval' (default), or 'weak'] The term evaluation mode.
    Returns
    val
        [float or array] In ‘eval’ mode, the term returns a single value (the integral, it does not need to be a scalar), while in ‘weak’ mode it returns an array for each element.
    status
        [int, optional] The flag indicating evaluation success (0) or failure (nonzero). Only provided if ret_status is True.
    iels
        [array of ints, optional] The local elements indices in ‘weak’ mode. Only provided in non-‘eval’ modes.
static from_desc(constructor, desc, region, integrals=None)
geometries = ['1_2', '2_3', '2_4', '3_4', '3_8']
get(variable, quantity_name, bf=None, integration=None, step=None, time_derivative=None)
    Get the named quantity related to the variable.

Notes
This is a convenience wrapper of Variable.evaluate() that initializes the arguments using the term data.
get_arg_name(arg_type, full=False, join=None)
    Get the name of the argument specified by arg_type.
    Parameters
    arg_type
        [str] The argument type string.
full
[bool] If True, return the full name. For example, if the name of a variable argument is ‘u’ and its time derivative is requested, the full name is ‘du/dt’.

join
[str, optional] Optionally, the material argument name tuple can be joined to a single string using the join string.

Returns

name
[str] The argument name.

get_args(arg_types=None, **kwargs)
Return arguments by type as specified in arg_types (or self.ats). Arguments in **kwargs can override the ones assigned at the term construction - this is useful for passing user data.

get_args_by_name(arg_names)
Return arguments by name.

get_assembling_cells(shape=None)
Return the assembling cell indices into a DOF connectivity.

get_conn_info()

get_conn_key()
The key to be used in DOF connectivity information.

get_data_shape(variable)
Get data shape information from variable.

Notes
This is a convenience wrapper of FieldVariable.get_data_shape() that initializes the arguments using the term data.

get_dof_conn_type(var_name)

get_kargs(keys, **kwargs)
Extract arguments from **kwargs listed in keys (default is None).

get_mapping(variable, get_saved=False, return_key=False)
Get the reference mapping from a variable.

Notes
This is a convenience wrapper of Field.get_mapping() that initializes the arguments using the term data.

get_material_names(part=0)

get_materials(join=False)

get_parameter_names()

get_parameter_variables()
get_physical_qps()  
Get physical quadrature points corresponding to the term region and integral.

get_qp_key()  
Return a key identifying uniquely the term quadrature points.

g地区ion()  

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)  

g地区str()  

get_user_names()  

get_variable_names()  

get_variables(as_list=True)  

g地区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()  

g地区virtual_variable()  

integration = 'cell'  

name = ''  

static new(name, integral, region, **kwargs)  

set_arg_types()  

set_integral(integral)  
Set the term integral.

setup(allow_derivatives=False)  

setup_args(**kwargs)  

setup_formal_args(allow_derivatives=False)  

setup_geometry_types()  

setup_integration()  

standalone_setup()  

static tile_mat(mat, nel)  

static time_update(ts)  

static translate_fargs_mapping(function, fargs, force=False)  

class sfepy.terms.terms.Terms(objs=None)
append(obj)

assign_args(variables, materials, user=None)
    Assign all term arguments.

static from_desc(term_descs, regions, integrals=None)
    Create terms, assign each term its region.

get_material_names()

get_user_names()

get_variable_names()

insert(ii, obj)

setup(**kwargs)

update_expression()

sfepy.terms.terms.create_arg_parser(allow_derivatives=False)

sfepy.terms.terms.get_arg_kinds(arg_types)
    Translate arg_types of a Term to a canonical form.

Parameters

arg_types
    [tuple of strings] The term argument types, as given in the arg_types attribute.

Returns

arg_kinds
    [list of strings] The argument kinds - one of 'virtual_variable', 'state_variable', 'parameter_variable', 'opt_material', 'ts', 'user'.

sfepy.terms.terms.get_shape_kind(integration)
    Get data shape kind for given integration type.

sfepy.terms.terms.split_complex_args(args)
    Split complex arguments to real and imaginary parts.

Returns

newargs
    [dictionary] Dictionary with lists corresponding to args such that each argument of numpy.complex128 data type is split to its real and imaginary part. The output depends on the number of complex arguments in 'args':
    • 0: list (key 'r') identical to input one
    • 1: two lists with keys 'r', 'i' corresponding to real and imaginary parts
    • 2: output dictionary contains four lists:
        – 'r' - real(arg1), real(arg2)
        – 'i' - imag(arg1), imag(arg2)
        – 'ri' - real(arg1), imag(arg2)
        – 'ir' - imag(arg1), real(arg2)
sfepy.terms.terms_adj_navier_stokes module

class sfepy.terms.terms_adj_navier_stokes.AdjConvect1Term

The first adjoint term to nonlinear convective term \( dw_{\text{convect}} \).

Definition

\[
\int_\Omega \left((v \cdot \nabla) u\right) \cdot w
\]

Call signature

\[
\text{dw_adj_convect1} \ (\text{virtual}, \text{state}, \text{parameter})
\]

Arguments

- virtual : \( v \)
- state : \( w \)
- parameter : \( u \)

\[ \text{arg_shapes} = \{ \text{'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')} \} \]
\[ \text{arg_types} = \{ \text{'virtual', 'state', 'parameter'} \} \]

static function

get_fargs(\( \text{virtual}, \text{state}, \text{parameter}, \text{mode}=\text{None}, \text{term_mode}=\text{None}, \text{diff_var}=\text{None}, **\text{kwargs} \))

name = 'dw_adj_convect1'

class sfepy.terms.terms_adj_navier_stokes.AdjConvect2Term

The second adjoint term to nonlinear convective term \( dw_{\text{convect}} \).

Definition

\[
\int_\Omega \left((u \cdot \nabla)v\right) \cdot w
\]

Call signature

\[
\text{dw_adj_convect2} \ (\text{virtual}, \text{state}, \text{parameter})
\]

Arguments

- virtual : \( v \)
- state : \( w \)
- parameter : \( u \)

\[ \text{arg_shapes} = \{ \text{'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')} \} \]
\[ \text{arg_types} = \{ \text{'virtual', 'state', 'parameter'} \} \]
static function()
get_fargs(virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_convect2'

class sfepy.terms.terms_adj_navier_stokes.AdjDivGradTerm(name, arg_str, integral, region, **kwargs)
Gateaux differential of $\Psi(u) = \int_\Omega \nu \nabla v : \nabla u$ w.r.t. $u$ in the direction $v$ or adjoint term to $dw_{\text{div_grad}}$.
Definition
$$w \delta_v \Psi(u) \circ v$$

Call signature

`dw_adj_div_grad` (material_1, material_2, virtual, parameter)

Arguments
- material_1 : $w$ (weight)
- material_2 : $\nu$ (viscosity)
- virtual : $v$
- state : $u$

arg_shapes = {'material_1': '1', 'material_2': '1', 'parameter': 'D', 'virtual': ('D', None)}
arg_types = ('material_1', 'material_2', 'virtual', 'parameter')

static function()
get_fargs(mat1, mat2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_adj_div_grad'

class sfepy.terms.terms_adj_navier_stokes:NSOFMinGradTerm(name, arg_str, integral, region, **kwargs)

Call signature

`d_of_ns_min_grad` (material_1, material_2, parameter)

arg_shapes = {'material_1': '1', 'material_2': '1', 'parameter': 1}
arg_types = ('material_1', 'material_2', 'parameter')

static function()
get_eval_shape(weight, mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(weight, mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'd_of_ns_min_grad'
class sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressDiffTerm(name, arg_str, integral, region, **kwargs)

Gateaux differential of $\Psi(p)$ w.r.t. $p$ in the direction $q$.

Definition

$$w\delta_p \Psi(p) \circ q$$

Call signature

```
dw_of_ns_surf_min_d_press_diff (material, virtual)
```

Arguments

- material : $w$ (weight)
- virtual : $q$

```
arg_shapes = {'material': 1, 'virtual': (1, None)}
arg_types = ('material', 'virtual')
```

gateaux differential

get_fargs(weight, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

```
name = 'dw_of_ns_surf_min_d_press_diff'
```

call signature

class sfepy.terms.terms_adj_navier_stokes.NSOFSurfMinDPressTerm(name, arg_str, integral, region, **kwargs)

Sensitivity of $\Psi(p)$.

Definition

$$\delta \Psi(p) = \delta \left( \int_{\Gamma_{in}} p - \int_{\Gamma_{out}} b_{press} \right)$$

Call signature

```
ev_of_ns_surf_min_d_press (material_1, material_2, parameter)
```

Arguments

- material_1 : $w$ (weight)
- material_2 : $b_{press}$ (given pressure)
- parameter : $p$

```
arg_shapes = {'material_1': 1, 'material_2': 1, 'parameter': 1}
arg_types = ('material_1', 'material_2', 'parameter')
```

get_fargs

```
get_eval_shape(weight, bpress, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(weight, bpress, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
```

integration = 'facet'
name = 'ev_of_ns_surf_min_d_press'

class sfepy.terms.terms_adj_navier_stokes.SDConvectTerm(name, arg_str, integral, region, **kwargs)
Sensitivity (shape derivative) of convective term $dw_{\text{convect}}$. Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition
\[
\int_\Omega \left[ u_k \frac{\partial u_i}{\partial x_k} w_i (\nabla \cdot \mathbf{v}) - u_k \frac{\partial \mathbf{v}_j}{\partial x_k} \frac{\partial u_i}{\partial x_j} w_i \right]
\]

Call signature

```
ev_sd_convect (parameter_u, parameter_w, parameter_mv)
```

Arguments

- parameter_u : $u$
- parameter_w : $w$
- parameter_mv : $\mathbf{v}$

arg_shapes = {'parameter_mv': 'D', 'parameter_u': 'D', 'parameter_w': 'D'}
arg_types = ('parameter_u', 'parameter_w', 'parameter_mv')

static function()

get_eval_shape(par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_convect'

class sfepy.terms.terms_adj_navier_stokes.SDDivGradTerm(name, arg_str, integral, region, **kwargs)
Sensitivity (shape derivative) of diffusion term $dw_{\text{div}\mathbf{grad}}$. Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition
\[
\int_\Omega \hat{I} \nabla v : \nabla u, \int_\Omega \nu \hat{I} \nabla v : \nabla u \\
\hat{I}_{ijkl} = \delta_{ik}\delta_{jl} \nabla \cdot \mathbf{v} - \delta_{ik}\delta_{js} \frac{\partial \mathbf{v}_l}{\partial x_s} - \delta_{is}\delta_{jl} \frac{\partial \mathbf{v}_k}{\partial x_s}
\]

Call signature

```
ev_sd_div_grad (opt_material, parameter_u, parameter_w, parameter_mv)
```

Arguments

- material : $\nu$ (viscosity, optional)
- parameter_u : $u$
- parameter_w : $w$
• parameter_mv : $V$

arg_shapes = [{'opt_material': '1', '1', 'parameter_u': 'D', 'parameter_w': 'D',
        'parameter_mv': 'D'}, {'opt_material': None}]

arg_types = ('opt_material', 'parameter_u', 'parameter_w', 'parameter_mv')

static function()

get_eval_shape(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_div_grad'

class sfepy.terms.terms_adj_navier_stokes.SDDivTerm(name, arg_str, integral, region, **kwargs)
Sensitivity (shape derivative) of Stokes term $dw_{stokes}$ in ‘div’ mode.
Supports the following term modes: 1 (sensitivity) or 0 (original term value).

Definition

$$\int_\Omega p((\nabla \cdot w)(\nabla \cdot V) - \frac{\partial V_k}{\partial x_i} \frac{\partial w_i}{\partial x_k})$$

Call signature

ev_sd_div (parameter_u, parameter_p, parameter_mv)

Arguments

• parameter_u : u
• parameter_p : p
• parameter_mv : $V$

arg_shapes = {'parameter_mv': 'D', 'parameter_p': 1, 'parameter_u': 'D'}

arg_types = ('parameter_u', 'parameter_p', 'parameter_mv')

static function()

get_eval_shape(par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_div'

class sfepy.terms.terms_adj_navier_stokes.SDDotTerm(name, arg_str, integral, region, **kwargs)
Sensitivity (shape derivative) of dot product of scalars or vectors.

Definition

$$\int_\Omega pq(\nabla \cdot V), \int_\Omega (u \cdot w)(\nabla \cdot V)$$

Call signature

ev_sd_dot (parameter_1, parameter_2, parameter_mv)
Arguments

- parameter_1: $p$ or $u$
- parameter_2: $q$ or $w$
- parameter_mv: $\mathcal{V}$

```
arg_shapes = [   
    {'parameter_1': 'D', 'parameter_2': 'D', 'parameter_mv': 'D'},  
    {'parameter_1': 1, 'parameter_2': 1}  
]
```

```
arg_types = ('parameter_1', 'parameter_2', 'parameter_mv')
```

```
static function()
```

```
get_eval_shape(par1, par2, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(par1, par2, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_dot'
```

class sfepy.terms.terms_adj_navier_stokes.SDGradDivStabilizationTerm(name, arg_str, integral,  
region, **kwargs)

Sensitivity (shape derivative) of stabilization term $dw_{st\_grad\_div}$.

Definition

$$
\gamma \int_{\Omega} \left[ (\nabla \cdot u)(\nabla \cdot w)(\nabla \cdot \mathcal{V}) - \frac{\partial u_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} (\nabla \cdot w) - (\nabla \cdot u) \frac{\partial w_i}{\partial x_k} \frac{\partial \mathcal{V}_k}{\partial x_i} \right]
$$

Call signature

```
ev_sd_st_grad_div (material, parameter_u, parameter_w, parameter_mv)
```

Arguments

- material: $\gamma$
- parameter_u: $u$
- parameter_w: $w$
- parameter_mv: $\mathcal{V}$
- mode: 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_mv': 'D', 'parameter_u': 'D',  
'parameter_w': 'D'}
```

```
arg_types = ('material', 'parameter_u', 'parameter_w', 'parameter_mv')
```

```
static function()
```

```
get_eval_shape(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
name = 'ev_sd_st_grad_div'
```
class sfepy.terms.terms_adj_navier_stokes.SDPSPGPStabilizationTerm(name, arg_str, integral, region, **kwargs)

Sensitivity (shape derivative) of stabilization terms $dw_{st\_supg\_p}$ or $dw_{st\_pspg\_c}$.

Definition

$$\sum_{K \in \Omega} h \int_{T_K} [ \delta_K \left( \nabla \cdot \nabla \cdot (\mathbf{b} \cdot \nabla u_i)(\nabla \cdot \mathbf{V}) - \frac{\partial r}{\partial x_k} \partial V_k \nabla u_i - \frac{\partial r}{\partial x_k} (\mathbf{b} \cdot \nabla V_k) \frac{\partial u_i}{\partial x_k} \right) ]$$

Call signature

```
ev_sd_st_pspg_c  (material, parameter_b, parameter_u, parameter_r, parameter_mv)
```

Arguments

- material : $\delta_K$
- parameter_b : $\mathbf{b}$
- parameter_u : $u$
- parameter_r : $r$
- parameter_mv : $\mathbf{V}$
- mode : 1 (sensitivity) or 0 (original term value)

```
arg_shapes = {'material': '1, 1', 'parameter_b': 'D', 'parameter_mv': 'D', 'parameter_r': '1', 'parameter_u': 'D'}
arg_types = ('material', 'parameter_b', 'parameter_u', 'parameter_r', 'parameter_mv')
```

static function()

get_eval_shape(mat, par_b, par_u, par_r, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, par_b, par_u, par_r, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_st_pspg_c'

class sfepy.terms.terms_adj_navier_stokes.SDPSPGCStabilizationTerm(name, arg_str, integral, region, **kwargs)

Sensitivity (shape derivative) of stabilization term $dw_{st\_pspg\_p}$.

Definition

$$\sum_{K \in \Omega} h \int_{T_K} \tau_K \left[ (\nabla \cdot \nabla \cdot \mathbf{V})(\nabla \cdot \mathbf{V}) - \frac{\partial r}{\partial x_k} (\nabla V_k \cdot \nabla \mathbf{V}) - (\nabla r \cdot \nabla V_k) \frac{\partial \mathbf{V}}{\partial x_k} \right]$$

Call signature

```
ev_sd_st_pspg_p  (material, parameter_r, parameter_p, parameter_mv)
```

Arguments

- material : $\tau_K$
• parameter_r : \( r \)
• parameter_p : \( p \)
• parameter_mv : \( \nu \)
• mode : 1 (sensitivity) or 0 (original term value)

\[
\text{arg_shapes} = \{ \text{'material': '1, 1', 'parameter_mv': 'D', 'parameter_p': 1, 'parameter_r': 1} \}
\]

\[
\text{arg_types} = (\text{'material'}, \text{'parameter_r'}, \text{'parameter_p'}, \text{'parameter_mv'})
\]

static function()

\[
\text{get_eval_shape} (\text{mat, par_r, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs})
\]

\[
\text{get_fargs} (\text{mat, par_r, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs})
\]

name = 'ev_sd_st_psgp_p'

class sfepy.terms.terms_adj_navier_stokes.SDSUPGCStabilizationTerm(name, arg_str, integral, region, **kwargs)

Sensitivity (shape derivative) of stabilization term \( dw_{st\_supg\_c} \).

Definition

\[
\sum_{K \in \mathcal{I}} \int_{T_K} \delta_K \left[ (\mathbf{b} \cdot \nabla u_k)(\mathbf{b} \cdot \nabla w_k)(\nabla \cdot \nu) - (\mathbf{b} \cdot \nabla \nu_i) \frac{\partial u_k}{\partial x_i} (\mathbf{b} \cdot \nabla w_k) - (u \cdot \nabla u_k)(\mathbf{b} \cdot \nabla \nu_i) \frac{\partial w_k}{\partial x_i} \right]
\]

Call signature

\[
\text{ev_sd_st_supg_c} \ (\text{material, parameter_b, parameter_u, parameter_w, parameter_mv})
\]

Arguments

• material : \( \delta_K \)
• parameter_b : \( \mathbf{b} \)
• parameter_u : \( u \)
• parameter_w : \( w \)
• parameter_mv : \( \nu \)
• mode : 1 (sensitivity) or 0 (original term value)

\[
\text{arg_shapes} = \{ \text{'material': '1, 1', 'parameter_b': 'D', 'parameter_mv': 'D', 'parameter_u': 'D', 'parameter_w': 'D'} \}
\]

\[
\text{arg_types} = (\text{'material'}, \text{'parameter_b'}, \text{'parameter_u'}, \text{'parameter_w'}, \text{'parameter_mv'})
\]

static function()

\[
\text{get_eval_shape} (\text{mat, par_b, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs})
\]

2.3. Developer Guide
get_fargs(mat, par_b, par_u, par_w, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev_sd_st_supg_c'

class sfepy.terms.terms_adj_navier_stokes.SUPGCAdjStabilizationTerm(name, arg_str, integral, region, **kwargs)

Adjoint term to SUPG stabilization term \( dw_{st\_supg\_c} \).

Definition

\[
\sum_{K \in I_h} \int_{T_K} \delta_K \left[ ((v \cdot \nabla)u)((u \cdot \nabla)w) + ((u \cdot \nabla)u)((v \cdot \nabla)w) \right]
\]

Call signature

\[
\text{dw_st_adj_supg_c (material, virtual, parameter, state)}
\]

Arguments

• material : \( \delta_K \)
• virtual : \( v \)
• state : \( w \)
• parameter : \( u \)

arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state'))

arg_types = ('material', 'virtual', 'parameter', 'state')

static function()

get_fargs(mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_st_adj_supg_c'

class sfepy.terms.terms_adj_navier_stokes.SUPGPAdj1StabilizationTerm(name, arg_str, integral, region, **kwargs)

The first adjoint term to SUPG stabilization term \( dw_{st\_supg\_p} \).

Definition

\[
\sum_{K \in I_h} \int_{T_K} \delta_K \nabla p(v \cdot \nabla w)
\]

Call signature

\[
\text{dw_st_adj1_supg_p (material, virtual, state, parameter)}
\]

Arguments

• material : \( \delta_K \)
• virtual : \( v \)
• state : \( w \)
• parameter: \( p \)

\[
\text{arg_shapes} = \{ \text{'material': '1, 1', 'parameter': 1, 'state': 'D', 'virtual': ('D', 'state')} \}
\]

\[
\text{arg_types} = (\text{'material', 'virtual', 'state', 'parameter'})
\]

static function()

get_fargs(mat, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_st_adj1_supg_p'

class sfepy.terms.terms_adj_navier_stokes.SUPGPAdj2StabilizationTerm(name, arg_str, integral, region, **kwargs)

The second adjoint term to SUPG stabilization term \( dw_{st\_supg\_p} \) as well as adjoint term to PSPG stabilization term \( dw_{st\_pspg\_c} \).

Definition

\[
\sum_{K \in T_h} \int_{T_K} \tau_K \nabla r (\mathbf{v} \cdot \nabla \mathbf{u})
\]

Call signature

\[
dw_{st\_adj2\_supg\_p} \quad (\text{material, virtual, parameter, state})
\]

Arguments

• material: \( \tau_K \)
• virtual: \( \mathbf{v} \)
• parameter: \( \mathbf{u} \)
• state: \( r \)

\[
\text{arg_shapes} = \{ \text{'material': '1, 1', 'parameter': 'D', 'state': 1, 'virtual': ('D', 'state')} \}
\]

\[
\text{arg_types} = (\text{'material', 'virtual', 'parameter', 'state'})
\]

static function()

get_fargs(mat, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_st_adj2_supg_p'

sfepy.terms.terms_adj_navier_stokes.grad_as_vector(grad)
sfepy.terms.terms_basic module

class sfepy.terms.terms_basic.IntegrateMatTerm(name, arg_str, integral, region, **kwargs)

Evaluate material parameter $m$ in a volume region.
Depending on evaluation mode, integrate a material parameter over a volume region (‘eval’), average it in elements (‘el_avg’) or interpolate it into volume quadrature points (‘qp’).
Uses reference mapping of $y$ variable.
Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

\[ \int_D c \]

Call signature

```
ev_integrate_mat (material, parameter)
```

Arguments

- material : $c$ (can have up to two dimensions)
- parameter : $y$

arg_shapes = [{'material': 'N, N', 'parameter': 'N'}]
arg_types = ('material', 'parameter')

static function(out, mat, geo, fmode)

get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet')

name = 'ev_integrate_mat'

class sfepy.terms.terms_basic.IntegrateOperatorTerm(name, arg_str, integral, region, **kwargs)

Integral of a test function weighted by a scalar function $c$.

Definition

\[ \int_D q \text{ or } \int_D cq \]

Call signature

```
dw_integrate (opt_material, virtual)
```

Arguments

- material : $c$ (optional)
- virtual : $q$
arg_shapes = [{'opt_material': '1, 1', 'virtual': (1, None)}, {'opt_material': None}]
arg_types = ('opt_material', 'virtual')

static function(out, material, bf, geo)
get_fargs(material, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet')
name = 'dw_integrate'

class sfepy.terms.terms_basic.IntegrateTerm

Evaluate (weighted) variable in a region. Depending on evaluation mode, integrate a variable over a region ('eval'), average it in elements ('el_avg') or interpolate it into quadrature points ('qp'). For a surface region and vector variables, setting term_mode to 'flux' leads to computing corresponding fluxes for the three modes instead.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition

\[ \int_{\Omega} y, \int_{\Omega} y \cdot n, \int_{\Gamma} y \cdot n \\]
\[ \int_{\Omega} cy, \int_{\Omega} cy \cdot n, \int_{\Gamma} cy \cdot n \text{ flux} \]

Call signature

\[ \text{ev_integrate} \quad \text{(opt_material, parameter)} \]

Arguments

• material : c (optional)
• parameter : y or \( y \)

arg_shapes = [{'opt_material': '1, 1', 'parameter': 'N'}, {'opt_material': None}]
arg_types = ('opt_material', 'parameter')

static function(out, val_qp, vg, fmode)
get_eval_shape(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet')
name = 'ev_integrate'

class sfepy.terms.terms_basic.SumNodalValuesTerm

Sum nodal values.

Call signature

\[ \text{ev_sum_vals} \quad \text{(parameter)} \]
Arguments

- parameter : $p$ or $u$

```python
define_function(
    arg_shapes = {'parameter': 'N'},
    arg_types = ('parameter',)
)
def get_eval_shape(
    parameter, mode=None, term_mode=None, diff_var=None, **kwargs
)
def get_fargs(
    parameter, mode=None, term_mode=None, diff_var=None, **kwargs
)

def name = 'ev_sum_vals'
class SurfaceMomentTerm:
    Surface integral of the outer product of the unit outward normal $n$ and the coordinate $x$ shifted by $x_0$

    Definition
    \[
    \int_{\Gamma} n(x - x_0)
    \]

    Call signature

    | ev_surface_moment | (material, parameter) |
    |-------------------|-----------------------|

Arguments

- material : $x_0$ (special)
- parameter : any variable

```python
def fct:
    define_function(
        arg_shapes = {'material': ': D', 'parameter': 'N'},
        arg_types = ('material', 'parameter')
    )
def get_eval_shape(
    material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs
)
def get_fargs(
    material, parameter, mode=None, term_mode=None, diff_var=None, **kwargs
)

def name = 'ev_surface_moment'
class VolumeSurfaceTerm:
    Volume of a $D$-dimensional domain, using a surface integral. Uses approximation of the parameter variable.

    Definition
    \[
    1/D \int_{\Gamma} \hat{x} \cdot n
    \]

    Call signature

    | ev_volume_surface | (parameter) |
    |-------------------|-------------|
Arguments

- parameter: any variable

arg_shapes = {'parameter': 'N'}
arg_types = ('parameter',)
static function()

get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'
name = 'ev_volume_surface'

class sfepy.terms.terms_basic.VolumeTerm(name, arg_str, integral, region, **kwargs)
Volume or surface of a domain. Uses approximation of the parameter variable.

Definition

\[ \int_\Omega 1 \]

Call signature

\[ \text{ev_volume} \text{ (parameter)} \]

Arguments

- parameter: any variable

arg_shapes = [{'parameter': 'N'}]
arg_types = ('parameter',)
static function(out, geo)

get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet')
name = 'ev_volume_surface'

class sfepy.terms.terms_basic.ZeroTerm(name, arg_str, integral, region, **kwargs)
A do-nothing term useful for introducing additional variables into the equations.

Definition

0

Call signature

\[ \text{dw_zero} \text{ (virtual, state)} \]
Arguments

- `virtual`: `q` or `v`
- `state`: `p` or `u`

```python
derived = {'state': 'N', 'virtual': ('N', None)}
```

```python
derived_types = ('virtual', 'state')
```

```python
def get_fargs(vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
    name = 'dw_zero'
```

**sfepy.terms.terms_biot module**

```python
class sfepy.terms.terms_biot.BiotETHTerm(name, arg_str, integral, region, **kwargs)
```

This term has the same definition as `dw_biot_th`, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

**Definition**

\[
\int_\Omega \left[ \int_0^t \alpha_{ij}(t-\tau) p(\tau) \, d\tau \right] e_{ij}(v) , \\
\int_\Omega \left[ \int_0^t \alpha_{ij}(t-\tau) e_{kl}(u(\tau)) \, d\tau \right] q
\]

**Call signature**

```python

dw_biot_eth (ts, material_0, material_1, virtual, state)  
(ts, material_0, material_1, state, virtual)
```

**Arguments 1**

- `ts`: TimeStepper instance
- `material_0`: `\alpha_{ij}(0)`
- `material_1`: `\exp(-\lambda \Delta t)` (decay at `t_1`)
- `virtual`: `v`
- `state`: `p`

**Arguments 2**

- `ts`: TimeStepper instance
- `material_0`: `\alpha_{ij}(0)`
- `material_1`: `\exp(-\lambda \Delta t)` (decay at `t_1`)
- `state`: `u`
- `virtual`: `q`

```python
derived = {'material_0': 'S', 1, 'material_1': '1', 1, 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None)}
```
arg_types = ('ts', 'material_0', 'material_1', 'virtual', 'state'), ('ts', 'material_0', 'material_1', 'state', 'virtual'))

get_fargs(ts, mat0, mat1, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad', 'div')

name = 'dw_biot_eth'

class sfepy.terms.terms_biot.BiotStressTerm(name, arg_str, integral, region, **kwargs)
Evaluate Biot stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

\[-\int_{\Omega} \alpha_{ij} p\]

Call signature

\[
\text{ev_biot_stress} \quad (\text{material}, \text{parameter})
\]

Arguments

- material : \(\alpha_{ij}\)
- parameter : \(p\)

arg_shapes = {'material': 'S', 1', 'parameter': 1'}

arg_types = ('material', 'parameter')

static function(out, val_qp, mat, vg, fmode)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'cell'

name = 'ev_biot_stress'

class sfepy.terms.terms_biot.BiotTHTerm(name, arg_str, integral, region, **kwargs)
Fading memory Biot term. Can use derivatives.

Definition

\[
\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) p(\tau) \, d\tau \right] e_{ij}(\nu), \\
\int_{\Omega} \left[ \int_0^t \alpha_{ij}(t-\tau) e_{kl}(u(\tau)) \, d\tau \right] q
\]

Call signature

\[
\text{dw_biot_th} \quad (\text{ts}, \text{material}, \text{virtual}, \text{state}) \\
(\text{ts}, \text{material}, \text{state}, \text{virtual})
\]

Arguments 1
• ts : TimeStepper instance
• material : $\alpha_{ij}(\tau)$
• virtual : $v$
• state : $p$

Arguments 2
• ts : TimeStepper instance
• material : $\alpha_{ij}(\tau)$
• state : $u$
• virtual : $q$

arg_shapes = {"material": ': N, S, 1', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None)}

arg_types = ("ts", 'material', 'virtual', 'state'), ('ts', 'material', 'state', 'virtual'))

get_fargs(ts, mats, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad', 'div')

name = 'dw_biot_th'

class sfepy.terms.terms_biot.BiotTerm(name, arg_str, integral, region, **kwargs)

Biot coupling term with $\alpha_{ij}$ given in:

• vector form exploiting symmetry - in 3D it has the indices ordered as $[11, 22, 33, 12, 13, 23]$, in 2D it has the indices ordered as $[11, 22, 12]$,

• matrix form - non-symmetric coupling parameter.

Corresponds to weak forms of Biot gradient and divergence terms. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} p \alpha_{ij} e_{ij}(v), \int_{\Omega} q \alpha_{ij} e_{ij}(u)$$

Call signature

<table>
<thead>
<tr>
<th>dw_biot</th>
<th>(material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material, state, virtual)</td>
</tr>
<tr>
<td></td>
<td>(material, parameter_v, parameter_s)</td>
</tr>
</tbody>
</table>

Arguments 1
• material : $\alpha_{ij}$
• virtual : $v$
• state : $p$

Arguments 2
• material : $\alpha_{ij}$
• state : $u$
• virtual : $q$

Arguments 3
  • material : $\alpha_{ij}$
  • parameter_v : $u$
  • parameter_s : $p$

```
arg_shapes = [{'material': 'S, 1', 'virtual/grad': ('D', None), 'state/grad': 1, 'virtual/div': (1, None), 'state/div': 'D', 'parameter_v': 'D', 'parameter_s': 1}, {'material': 'D, D'}]
```

```
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter_v', 'parameter_s'))
```

```
get_eval_shape(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'dw_biot'
```

```
set_arg_types()
```

---

**sfepy.terms.terms_compat module**

**class** sfepy.terms.terms_compat.CauchyStrainSTerm (name, arg_str, integral, region, **kwargs)

Call signature

```
ev_cauchy_strain_s (parameter)
```

```
name = 'ev_cauchy_strain_s'
```

**class** sfepy.terms.terms_compat.DSumNodalValuesTerm (name, arg_str, integral, region, **kwargs)

Call signature

```
d_sum_vals (parameter)
```

```
name = 'd_sum_vals'
```

**class** sfepy.terms.terms_compat.DSurfaceFluxTerm (name, arg_str, integral, region, **kwargs)

Call signature

```
d_surface_flux (material, parameter)
```

```
name = 'd_surface_flux'
```
class sfepy.terms.terms_compat.DSurfaceMomentTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    d_surface_moment (material, parameter)

    name = 'd_surface_moment'

class sfepy.terms.terms_compat.DVolumeSurfaceTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    d_volume_surface (parameter)

    name = 'd_volume_surface'

class sfepy.terms.terms_compat.DotSurfaceProductTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    dw_surface_dot (opt_material, virtual, state)
                    (opt_material, parameter_1, parameter_2)

    name = 'dw_surface_dot'

class sfepy.terms.terms_compat.DotVolumeProductTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    dw_volume_dot (opt_material, virtual, state)
                    (opt_material, parameter_1, parameter_2)

    name = 'dw_volume_dot'

class sfepy.terms.terms_compat.IntegrateSurfaceMatTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    ev_surface_integrate_mat (material, parameter)

    name = 'ev_surface_integrate_mat'

class sfepy.terms.terms_compat.IntegrateSurfaceOperatorTerm(name, arg_str, integral, region, **kwargs)

    Call signature

    dw_surface_integrate (opt_material, virtual)

    name = 'dw_surface_integrate'
class sfepy.terms.terms_compat.IntegrateSurfaceTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   ev_surface_integrate  (opt_material, parameter)
```

name = 'ev_surface_integrate'

class sfepy.terms.terms_compat.IntegrateVolumeMatTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   ev_volume_integrate_mat  (material, parameter)
```

name = 'ev_volume_integrate_mat'

class sfepy.terms.terms_compat.IntegrateVolumeOperatorTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   dw_volume_integrate  (opt_material, virtual)
```

name = 'dw_volume_integrate'

class sfepy.terms.terms_compat.IntegrateVolumeTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   ev_volume_integrate  (opt_material, parameter)
```

name = 'ev_volume_integrate'

class sfepy.terms.terms_compat.SDVolumeDotTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   ev_sd_volume_dot  (parameter_1, parameter_2, parameter_mv)
```

name = 'ev_sd_volume_dot'

class sfepy.terms.terms_compat.SurfaceDivTerm(name, arg_str, integral, region, **kwargs)

Call signature

```
   ev_surface_div  (opt_material, parameter)
```

name = 'ev_surface_div'
class sfepy.terms.terms_compat.SurfaceGradTerm(name, arg_str, integral, region, **kwargs)

    Call signature
    
    ev_surface_grad  (opt_material, parameter)

    name = 'ev_surface_grad'

class sfepy.terms.terms_compat.SurfaceTerm(name, arg_str, integral, region, **kwargs)

    Call signature
    
    d_surface  (parameter)

    name = 'd_surface'

class sfepy.terms.terms_compat.VolumeXTerm(name, arg_str, integral, region, **kwargs)

    Call signature
    
    d_volume  (parameter)

    name = 'd_volume'

sfepy.terms.terms_constraints module

class sfepy.terms.terms_constraints.NonPenetrationPenaltyTerm(name, arg_str, integral, region, **kwargs)

    Non-penetration condition in the weak sense using a penalty.

    Definition
    \[ \int_\Gamma c (\mathbf{n} \cdot \mathbf{v}) (\mathbf{n} \cdot \mathbf{u}) \]

    Call signature
    
    dw_non_penetration_p  (material, virtual, state)

    Arguments
    
    - material : c
    - virtual : v
    - state : u

    arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
    arg_types = ('material', 'virtual', 'state')

    static function(out, val_qp, ebf, mat, sg, diff_var)
get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'dw_non_penetration_p'

class sfepy.terms.terms_constraints.NonPenetrationTerm(name, arg_str, integral, region, **kwargs)

Non-penetration condition in the weak sense.

Definition

\[\int_{\Gamma} c\lambda n \cdot \nu, \int_{\Gamma} c\hat{\lambda} n \cdot u\]
\[\int_{\Gamma} \lambda n \cdot \nu, \int_{\Gamma} \hat{\lambda} n \cdot u\]

Call signature

<table>
<thead>
<tr>
<th>dw_non_penetration</th>
<th>(opt_material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(opt_material, state, virtual)</td>
</tr>
</tbody>
</table>

Arguments 1

- material : c (optional)
- virtual : \(\nu\)
- state : \(\lambda\)

Arguments 2

- material : c (optional)
- state : \(u\)
- virtual : \(\hat{\lambda}\)

arg_shapes = [{\'opt_material\': '1, 1', \'virtual/grad\': ('D', None), \'state/grad\': 1, \'virtual/div\': (1, None), \'state/div\': 'D'}, {\'opt_material\': None}]

arg_types = ((\'opt_material\', \'virtual\', \'state\'), (\'opt_material\', \'state\', \'virtual\'))

static function(out, val_qp, ebf, bf, mat, sg, diff_var, mode)

\(ebf\) belongs to vector variable, \(bf\) to scalar variable.

going to get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('grad', 'div')

name = 'dw_non_penetration'
sfepy.terms.terms_contact module

class sfepy.terms.terms_contact.ContactInfo(region, integral, geo, state)
    Various contact-related data of contact terms.

    update(xx)
        A dict-like update for Struct attributes.

class sfepy.terms.terms_contact.ContactTerm(*args, **kwargs)
    Contact term with a penalty function.

    The penalty function is defined as $\varepsilon N \langle g_N(u) \rangle$, where $\varepsilon N$ is the normal penalty parameter and $\langle g_N(u) \rangle$ are the Macaulay's brackets of the gap function $g_N(u)$.

    This term has a dynamic connectivity of DOFs in its region.

    Definition

    $\int_{\Gamma_c} \varepsilon N \langle g_N(u) \rangle n v$

    Call signature

    dw_contact (material, virtual, state)

    Arguments

    • material : $\varepsilon N$
    • virtual : $v$
    • state : $u$

    arg_shapes = {'material': ': 1', 'state': 'D', 'virtual': ('D', 'state')}

    arg_types = ('material', 'virtual', 'state')

    call_function(out, fargs)

    eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

    static function(out, fun, *args)

    static function_weak(out, out_cc)

    get_contact_info(geo, state, init_gps=False)

    get_eval_shape(epss, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

    get_fargs(epss, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

    static integrate(out, val_qp, geo, fmode)

    integration = 'facet'

    name = 'dw_contact'
**sfepy.terms.terms_dg module**

Discontinuous Galekrin method specific terms

**Note**

In einsum calls the following convention is used:

- \( i \) represents iterating over all cells of a region;
- \( n \) represents iterating over selected cells of a region, for example over cells on boundary;
- \( b \) represents iterating over basis functions of state variable;
- \( d \) represents iterating over basis functions of test variable;
- \( k, l, m \) represent iterating over geometric dimensions, for example coordinates of velocity or facet normal vector or rows and columns of diffusion tensor;
- \( q \) represents iterating over quadrature points;
- \( f \) represents iterating over facets of cell;

```python
class sfepy.terms.terms_dg.AdvectionDGFluxTerm(name, arg_str, integral, region, **kwargs)
```

Lax-Friedrichs flux term for advection of scalar quantity \( p \) with the advection velocity \( a \) given as a material parameter (a known function of space and time).

**Definition**

\[
\int_{\partial T_K} n \cdot f^*(p_{in}, p_{out}) q
\]

where

\[
f^*(p_{in}, p_{out}) = a \frac{p_{in} + p_{out}}{2} + (1 - \alpha) n C p_{in} - p_{out} \frac{2}{2},
\]

\( \alpha \in [0, 1] \); \( \alpha = 0 \) for upwind scheme, \( \alpha = 1 \) for central scheme, and

\[
C = \max_{p \in [\cdot, \cdot]} |n_x a_1 + n_y a_2| = \max_{p \in [\cdot, \cdot]} |n \cdot a|
\]

the \( p_{in} \) resp. \( p_{out} \) is solution on the boundary of the element provided by element itself resp. its neighbor and \( a \) is advection velocity.

**Call signature**

```python
dw_dg_advect_laxfrie_flux (opt_material, material_advelo, virtual, state)
```

**Arguments 1**

- \( \text{material} : a \)
- \( \text{virtual} : q \)
- \( \text{state} : p \)

**Arguments 3**

- \( \text{material} : a \)
- \( \text{virtual} : q \)
• state : $p$
• opt_material : $\alpha$

$\alpha = 0$

arg_shapes = [{'opt_material': '.', 'material_advelo': 'D, 1', 'virtual': (1, 'state'), 'state': 1}, {'opt_material': None}]

arg_types = ('opt_material', 'material_advelo', 'virtual', 'state')

function(out, state, diff_var, field, region, advelo)

going_fars(alpha, advelo, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'cell'

modes = ('weak',)

name = 'dw_dg_advect_laxfrie_flux'

symbolic = {'expression': 'div(a*p)*w', 'map': {'a': 'material', 'p': 'state', 'v': 'virtual'}}

class sfepy.terms.terms_dg.DGTerm(name, arg_str, integral, region, **kwargs)

Abstract base class for DG terms, provides alternative call_function and eval_real methods to accommodate returning iels and vals.

call_function(out, fargs)

eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

class sfepy.terms.terms_dg.DiffusionDGFluxTerm(name, arg_str, integral, region, **kwargs)

Basic DG diffusion flux term for scalar quantity.

Definition

$$\int_{\partial T_K} D(\nabla p)[q] . \int_{\partial T_K} D(\nabla q)[p]$$

where

$$\langle \nabla \phi \rangle = \frac{\nabla \phi_{in} + \nabla \phi_{out}}{2}$$

$$[\phi] = \phi_{in} - \phi_{out}$$

Math

The $p_{in}$ resp. $p_{out}$ is solution on the boundary of the element provided by element itself resp. its neighbour.

Call signature

<table>
<thead>
<tr>
<th>dw_dg_diffusion_flux</th>
<th>(material, state, virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_dg_diffusion_flux</td>
<td>(material, virtual, state)</td>
</tr>
</tbody>
</table>

Arguments 1

• material : $D$
• state : p
• virtual : q

Arguments 2
• material : D
• virtual : q
• state : p

arg_shapes = [{'material': '1, 1', 'virtual/avg_state': (1, None), 'state/avg_state': 1, 'virtual/avg_virtual': (1, None), 'state/avg_virtual': 1}]

arg_types = (('material', 'state', 'virtual'), ('material', 'virtual', 'state'))

function(out, state, diff_var, field, region, D)

get_fargs(diff_tensor, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'cell'

modes = ('avg_state', 'avg_virtual')

name = 'dw DG diffusion flux'

class sfepy.terms.terms_dg.DiffusionInteriorPenaltyTerm(name, arg_str, integral, region, **kwargs)
Penalty term used to counteract discontinuity arising when modeling diffusion using Discontinuous Galerkin schemes.

Definition

$$\int_{\partial T_K} \bar{D} C_w \frac{O r d^2}{d(\partial T_K)} [p][q]$$

where

$$[\phi] = \phi_{in} - \phi_{out}$$

Math

the \( p_{in} \) resp. \( p_{out} \) is solution on the boundary of the element provided by element itself resp. its neighbour.

Call signature

\[\text{dw DG interior penalty} \ (\text{material, material}_Cw, \text{virtual, state})\]

Arguments
• material : D
• material : \( C_w \)
• state : p
• virtual : q

arg_shapes = [{'material': '1, 1', 'material_Cw': '.', 'virtual': (1, 'state'), 'state': 1}]

2.3. Developer Guide 477
arg_types = ('material', 'material_Cw', 'virtual', 'state')

function(out, state, diff_var, field, region, Cw, diff_tensor)

get_fargs(diff_tensor, Cw, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)

name = 'dw_dg_interior_penalty'

class sfepy.terms.terms_dg.NonlinearHyperbolicDGFluxTerm(name, arg_str, integral, region, **kwargs)

Lax-Friedrichs flux term for nonlinear hyperbolic term of scalar quantity $p$ with the vector function $f$ given as a material parameter.

Definition

$$\int_{\partial T_K} \mathbf{n} \cdot f^*(p_{\text{in}}, p_{\text{out}}) q$$

where

$$f^*(p_{\text{in}}, p_{\text{out}}) = \frac{f(p_{\text{in}}) + f(p_{\text{out}})}{2} + (1 - \alpha)\mathbf{n}C p_{\text{in}} - p_{\text{out}}$$

$\alpha \in [0, 1]; \alpha = 0$ for upwind scheme, $\alpha = 1$ for central scheme, and

$$C = \max_{p \in [?, ?]} \left| n_x \frac{df_1}{dp} + n_y \frac{df_2}{dp} + \cdots \right| = \max_{p \in [?, ?]} \left| \mathbf{n} \cdot \frac{df}{dp}(p) \right|$$

the $p_{\text{in}}$ resp. $p_{\text{out}}$ is solution on the boundary of the element provided by element itself resp. its neighbor.

Call signature

```
dw_dg_nonlinear_laxfrie_flux (opt_material, fun, fun_d, virtual, state)
```

Arguments 1

- material : $f$
- material : $\frac{df}{dp}$
- virtual : $q$
- state : $p$

Arguments 3

- material : $f$
- material : $\frac{df}{dp}$
- virtual : $q$
- state : $p$
- opt_material : $\alpha$
alf = 0

arg_shapes = [{'opt_material': '.', 'material_fun': '.', 'material_fun_d': '.', 'virtual': (1, 'state'), 'state': 1}, {'opt_material': None}]

arg_types = ('opt_material', 'fun', 'fun_d', 'virtual', 'state')

function(out, state, field, region, f, df)

get_fargs(alpha, fun, dfun, test, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'cell'

modes = ('weak',)

name = 'dw_dg_nonlinear_laxfrie_flux'

symbolic = {'expression': 'div(f(p))*w', 'map': {'f': 'function', 'p': 'state', 'v': 'virtual'}}

class sfepy.terms.terms_dg.NonlinearScalarDotGradTerm(name, arg_str, integral, region, **kwargs)

Product of virtual and divergence of vector function of state or volume dot product of vector function of state and gradient of scalar virtual.

Definition

\[
\int_{\Omega} q \cdot \nabla \cdot f(p) = \int_{\Omega} q \cdot \text{div} f(p) = \int_{\Omega} f(p) \cdot \nabla q
\]

Call signature

dw_ns_dot_grad_s (fun, fun_d, virtual, state)
(fun, fun_d, state, virtual)

Arguments 1

• function : \( f \)
• virtual : \( q \)
• state : \( p \)

Arguments 2

• function : \( f \)
• state : \( p \)
• virtual : \( q \)

TODO maybe this term would fit better to terms_dot?

arg_shapes = [{'material_fun': '.', 'material_fun_d': '.', 'virtual/grad_state': (1, None), 'state/grad_state': 1, 'virtual/grad_virtual': 1, 'state/grad_virtual': 1}]

arg_types = ('fun', 'fun_d', 'virtual', 'state'), ('fun', 'fun_d', 'state', 'virtual'))
static function(out, out_qp, geo, fmode)

get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad_state', 'grad_virtual')

name = 'dw_ns_dot_grad_s'

**sfepy.terms.terms_diffusion module**

class sfepy.terms.terms_diffusion.AdvectDivFreeTerm(name, arg_str, integral, region, **kwargs)

Advection of a scalar quantity \(p\) with the advection velocity \(y\) given as a material parameter (a known function of space and time).

The advection velocity has to be divergence-free!

Definition

\[
\int_{\Omega} \nabla \cdot (yp) q = \int_{\Omega} \left( (\nabla \cdot y) + y \cdot \nabla \right) p q
\]

Call signature

```
dw_advect_div_free  (material, virtual, state)
```

Arguments

- material : \(y\)
- virtual : \(q\)
- state : \(p\)

arg_shapes = {'material': 'D, 1', 'state': '1', 'virtual': ('1', 'state')}

arg_types = ('material', 'virtual', 'state')

mode = 'grad_state'

name = 'dw_advect_div_free'

class sfepy.terms.terms_diffusion.ConvectVGradSTerm(name, arg_str, integral, region, **kwargs)

Scalar gradient term with convective velocity.

Definition

\[
\int_{\Omega} q (u \cdot \nabla p)
\]

Call signature

```
dw_convect_v_grad_s   (virtual, state_v, state_s)
```

Arguments

- virtual : \(q\)
• state_v : \(u\)
• state_s : \(p\)

arg_shapes = [['virtual': (1, 'state_s'), 'state_v': 'D', 'state_s': 1]]
arg_types = ('virtual', 'state_v', 'state_s')

function()
get_fargs(virtual, state_v, state_s, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_convect_v_grad_s'

class sfepy.terms.terms_diffusion.DiffusionCoupling(name, arg_str, integral, region, **kwargs)
Diffusion coupling term with material parameter \(K_j\).

Definition
\[
\int_{\Omega} pK_j \nabla_j q \cdot \int_{\Omega} qK_j \nabla_j p
\]

Call signature

<table>
<thead>
<tr>
<th>dw_diffusion_coupling</th>
<th>(material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material, state, virtual)</td>
</tr>
<tr>
<td></td>
<td>(material, parameter_1, parameter_2)</td>
</tr>
</tbody>
</table>

Arguments
• material : \(K_j\)
• virtual : \(q\)
• state : \(p\)

arg_shapes = {'material': 'D', 1', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}
arg_types = ('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter_1', 'parameter_2')

static d_fun(out, mat, val, grad, vg)
static dw_fun(out, val, mat, bf, vg, fmode)
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak0', 'weak1', 'eval')
name = 'dw_diffusion_coupling'
set_arg_types()
class sfepy.terms.terms_diffusion.DiffusionRTerm(name, arg_str, integral, region, **kwargs)

Diffusion-like term with material parameter $K_j$ (to use on the right-hand side).

Definition

$$\int_{\Omega} K_j \nabla_j q$$

Call signature

``dw_diffusion_r`` (material, virtual)

Arguments

- material : $K_j$
- virtual : $q$

arg_shapes = {'material': 'D, 1', 'virtual': (1, None)}
arg_types = ('material', 'virtual')

static function()

def get_fargs(mat, virtual, state)

def get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

def get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

class sfepy.terms.terms_diffusion.DiffusionTerm(name, arg_str, integral, region, **kwargs)

General diffusion term with permeability $K_{ij}$. Can be evaluated. Can use derivatives.

Definition

$$\int_{\Omega} K_{ij} \nabla_i q \nabla_j p$$

Call signature

``dw_diffusion`` (material, virtual, state)
(material, parameter_1, parameter_2)

Arguments

- material: $K_{ij}$
- virtual/parameter_1: $q$
- state/parameter_2: $p$

arg_shapes = {'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}
arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))

def get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

def get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('weak', 'eval')
name = 'dw_diffusion'
set_arg_types()
symbolic = {'expression': 'div(K * grad(u))', 'map': {'K': 'material', 'u': 'state'}}

class sfepy.terms.terms_diffusion.DiffusionVelocityTerm(name, arg_str, integral, region, **kwargs)
Evaluate diffusion velocity.
Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition
\[- \int_D K_{ij} \nabla_j p\]

Call signature

ev_diffusion_velocity (material, parameter)

Arguments
• material : $K_{ij}$
• parameter : $p$

arg_shapes = {'material': 'D, D', 'parameter': 1}
arg_types = ('material', 'parameter')

static function(out, grad, mat, vg, fmode)
get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = ('cell', 'facet_extra')
name = 'ev_diffusion_velocity'

class sfepy.terms.terms_diffusion.LaplaceTerm(name, arg_str, integral, region, **kwargs)
Laplace term with $c$ coefficient. Can be evaluated. Can use derivatives.

Definition
\[\int_\Omega c \nabla q \cdot \nabla p\]

Call signature

dw_laplace (opt_material, virtual, state)
(opt_material, parameter_1, parameter_2)

Arguments 1
• material: $c$
• virtual/parameter_1: q
• state/parameter_2: p

arg_shapes = [['opt_material': '1', 'virtual': (1, 'state'), 'state': 1, 'parameter_1': 1, 'parameter_2': 1}, {'opt_material': None}]

arg_types = ('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))

modes = ('weak', 'eval')

name = 'dw_laplace'

set_arg_types()

symbolic = {'expression': 'c * div( grad( u ) )', 'map': {'c': 'opt_material', 'u': 'state'}}

class sfepy.terms.terms_diffusion.NonlinearDiffusionTerm(name, arg_str, integral, region, **kwargs)

The diffusion term with a scalar coefficient given by a user supplied function of the state variable.

Definition
\[ \int_\Omega \nabla q \cdot \nabla p f(p) \]

Call signature

\[ \text{dw_nl_diffusion} \quad (\text{fun, dfun, virtual, state}) \]

Arguments
• fun : f(p)
• dfun : \partial f(p)/\partial p
• virtual : q
• state : p

arg_shapes = {'dfun': <function NonlinearDiffusionTerm.<lambda>>, 'fun': <function NonlinearDiffusionTerm.<lambda>>, 'state': 1, 'virtual': (1, 'state')}

arg_types = ('fun', 'dfun', 'virtual', 'state')

static function(out, out_qp, geo)

get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_nl_diffusion'

class sfepy.terms.terms_diffusion.SDDiffusionTerm(name, arg_str, integral, region, **kwargs)

Diffusion sensitivity analysis term.

Definition
\[\int_{\Omega} \hat{K}_{ij} \nabla_i q \nabla_j p\]

\[\hat{K}_{ij} = K_{ij} \left( \delta_{ik} \delta_{jl} \nabla \cdot \mathbf{V} - \delta_{ik} \frac{\partial V_j}{\partial x_l} - \delta_{jl} \frac{\partial V_i}{\partial x_k} \right)\]

Call signature

\[
\text{ev_sd_diffusion} \quad (\text{material}, \text{parameter}_q, \text{parameter}_p, \text{parameter}_mv)
\]

Arguments

- material: \(K_{ij}\)
- parameter\(_q\): \(q\)
- parameter\(_p\): \(p\)
- parameter\(_mv\): \(\mathbf{V}\)

arg\_shapes = {'material': 'D, D', 'parameter\_mv': 'D', 'parameter\_p': 1, 'parameter\_q': 1}

arg\_types = ('material', 'parameter\_q', 'parameter\_p', 'parameter\_mv')

static function()

get\_eval\_shape(mat, parameter\_q, parameter\_p, parameter\_mv, mode=None, term\_mode=None, diff\_var=None, **kwargs)

get\_fargs(mat, parameter\_q, parameter\_p, parameter\_mv, mode=None, term\_mode=None, diff\_var=None, **kwargs)

name = 'ev_sd_diffusion'

class sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm(name, arg\_str, integral, region, **kwargs)

Surface flux operator term.

Definition

\[\int_{\Gamma} q_n \cdot K \cdot \nabla p\]

Call signature

\[
\text{dw_surface_flux} \quad (\text{opt\_material}, \text{virtual}, \text{state})
\]

Arguments

- material : \(K\)
- virtual : \(q\)
- state : \(p\)

arg\_shapes = [{'opt\_material': 'D, D', 'virtual': (1, 'state'), 'state': 1}, {'opt\_material': None}]
arg_types = ('opt_material', 'virtual', 'state')

definition()

def get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet_extra'
name = 'dw_surface_flux'

class sfepy.terms.terms_diffusion.SurfaceFluxTerm(name, arg_str, integral, region, **kwargs)
Surface flux term.
Supports 'eval', 'el_eval' and 'el_avg' evaluation modes.
Definition
\[
\int_{\Gamma} n \cdot K_{ij} \nabla_j p
\]
Call signature

<table>
<thead>
<tr>
<th>ev_surface_flux</th>
<th>(material, parameter)</th>
</tr>
</thead>
</table>

Arguments

- material: \( K \)
- parameter: \( p \).

arg_shapes = {'material': 'D, D', 'parameter': 1}
arg_types = ('material', 'parameter')

static function()

def get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
def get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet_extra'
name = 'dw_surface_flux'

sfepy.terms.terms_dot module

class sfepy.terms.terms_dot.BCNewtonTerm(name, arg_str, integral, region, **kwargs)
Newton boundary condition term.
Definition
\[
\int_{\Gamma} \alpha q (p - p_{outer})
\]
Call signature

| dw_bc_newton | (material_1, material_2, virtual, state) |
Arguments

- material_1: \( \alpha \)
- material_2: \( p_{\text{outer}} \)
- virtual: \( q \)
- state: \( p \)

\[
\text{arg_shapes} = \{ \text{'material}_1\': 1, 1', \text{'material}_2\': 1, 1', \text{'state'}: 1, \text{'virtual'}: (1, 'state') \} \\
\text{arg_shapes_dict} = \text{None} \\
\text{arg_types} = (\text{'material}_1', \text{'material}_2', \text{'virtual'}, \text{'state'})
\]

get_fargs(alpha, p_out, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

mode = 'weak'

name = 'dw_bc_newton'

class sfepy.terms.terms_dot.DotProductTerm(name, arg_str, integral, region, **kwargs)

Volume and surface \( L^2() \) weighted dot product for both scalar and vector fields. If the region is a surface and either virtual or state variable is a vector, the orientation of the normal vectors is outwards to the parent region of the virtual variable. Can be evaluated. Can use derivatives.

Definition

\[
\int_D qp , \int_D v \cdot u , \\
\int_I v \cdot np , \int_I qn \cdot u , \\
\int_D cqp , \int_D cv \cdot u , \int_D v \cdot c \cdot u 
\]

Call signature

\[
\text{dw_dot} \begin{pmatrix} \text{(opt_material, virtual, state)} \\
\text{(opt_material, parameter}_1, \text{parameter}_2) \end{pmatrix}
\]

Arguments

- material: \( c \) or \( \underline{c} \) (optional)
- virtual/parameter_1: \( q \) or \( \underline{v} \)
- state/parameter_2: \( p \) or \( \underline{u} \)
arg_shapes_dict = {'cell': [{'opt_material': '1, 1', 'virtual': '1, state'}, 'state': 1, 'parameter_1': 1, 'parameter_2': 1}, {'opt_material': None},
{'opt_material': '1, 1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter_1': 'D', 'parameter_2': 'D'}, {'opt_material': 'D, D'}, {'opt_material': None}],

arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))

class sfepy.terms.terms_dot.DotSProductVolumeOperatorWETHTerm(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
\[
\int_{\Omega} \left[ \int_0^t G(t-\tau)p(\tau) \, d\tau \right] q
\]

Call signature

dw_volume_dot_w_scalar_eth (ts, material_0, material_1, virtual, state)
arg_shapes = {'material_0': '1, 1', 'material_1': '1, 1', 'state': '1', 'virtual': '1, 'state')}
arg_types = ('ts', 'material_0', 'material_1', 'virtual', 'state')
static function()
get_fargs(ts, mat0, mat1, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_volume_dot_w_scalar_eth'

class sfepy.terms.terms_dot.DotSProductVolumeOperatorWTHTerm(name, arg_str, integral, region, **kwargs)
Fading memory volume $L^2(\Omega)$ weighted dot product for scalar fields. Can use derivatives.

Definition
\[
\int_{\Omega} \left[ \int_0^t G(t-\tau)p(\tau) \, d\tau \right] q
\]

Call signature

```
dw_volume_dot_w_scalar_th (ts, material, virtual, state)
```

Arguments
- ts : TimeStepper instance
- material : $G(\tau)$
- virtual : $q$
- state : $p$

arg_shapes = {'material': 'N, 1, 1', 'state': '1, 'state')
arg_types = ('ts', 'material', 'virtual', 'state')
static function()
get_fargs(ts, mats, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_volume_dot_w_scalar_th'

class sfepy.terms.terms_dot.ScalarDotGradIScalarTerm(name, arg_str, integral, region, **kwargs)
Dot product of a scalar and the $i$-th component of gradient of a scalar. The index should be given as a `special_constant` material parameter.

Definition
\[
Z^i = \int_{\Omega} q \nabla_i p
\]

Call signature

```
dw_s_dot_grad_i_s (material, virtual, state)
```

Arguments
• material : \( i \)
• virtual : \( q \)
• state : \( p \)

\[
\text{arg_shapes} = \{ 'material': ': 1, 1', 'state': 1, 'virtual': (1, 'state') \}
\]

\[
\text{arg_types} = ( 'material', 'virtual', 'state')
\]

static \text{dw_fun}(\text{out, bf, vg, grad, idx, fmode})

\text{get_fargs}(\text{material, virtual, state, mode=\text{None, term_mode=\text{None, diff_var=\text{None, **kwargs}}})

name = 'dw_s_dot_grad_i_s'

set_arg_types()

class \text{sfePy.terms.terms_dot}.\text{ScalarDotMGradScalarTerm}(\text{name, arg_str, integral, region, **kwargs})

Volume dot product of a scalar gradient dotted with a material vector with a scalar.

\[
\int_{\Omega} qy \cdot \nabla p, \int_{\Omega} py \cdot \nabla q
\]

Call signature

\[
\text{dw_s_dot_mgrad_s} \quad (\text{material, virtual, state})
\]

\[
\text{dw_s_dot_mgrad_s} \quad (\text{material, state, virtual})
\]

Arguments 1
• material : \( y \)
• virtual : \( q \)
• state : \( p \)

Arguments 2
• material : \( y \)
• state : \( p \)
• virtual : \( q \)

\[
\text{arg_shapes} = \{ ['material': 'D, 1', 'virtual/grad_state': (1, \text{\text{None})},
\text{state/grad_state': 1, 'virtual/grad_virtual': (1, \text{\text{None})}, 'state/grad_virtual': 1] \}
\]

\[
\text{arg_types} = ( ['material', 'virtual', 'state'), ('material', 'state', 'virtual'))
\]

static \text{function}(\text{out, out_qp, geo, fmode})

\text{get_fargs}(\text{mat, var1, var2, mode=\text{None, term_mode=\text{None, diff_var=\text{None, **kwargs}}})

\text{modes} = ( 'grad_state', 'grad_virtual'))

name = 'dw_s_dot_mgrad_s'
class sfepy.terms.terms_dot.VectorDotGradScalarTerm(name, arg_str, integral, region, **kwargs)
Volume dot product of a vector and a gradient of scalar. Can be evaluated.

Definition
\[
\int_\Omega \mathbf{v} \cdot \nabla p, \quad \int_\Omega u \cdot \nabla q \\
\int_\Omega c \mathbf{v} \cdot \nabla p, \quad \int_\Omega c u \cdot \nabla q \\
\int_\Omega \mathbf{v} \cdot (c \nabla p), \quad \int_\Omega u \cdot (c \nabla q)
\]

Call signature

<table>
<thead>
<tr>
<th>dw_v_dot_grad_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(opt_material, virtual, state)</td>
</tr>
<tr>
<td>(opt_material, state, virtual)</td>
</tr>
<tr>
<td>(opt_material, parameter_v, parameter_s)</td>
</tr>
</tbody>
</table>

Arguments 1
- material: $c$ or $z$ (optional)
- virtual/parameter_v: $v$
- state/parameter_s: $p$

Arguments 2
- material: $c$ or $z$ (optional)
- state: $u$
- virtual: $q$

arg_shapes = [['opt_material': '1', '1', 'virtual/v_weak': ('D', None), 'state/v_weak': 1, 'virtual/s_weak': (1, None), 'state/s_weak': 'D', 'parameter_v': 'D', 'parameter_s': 1}, {'opt_material': 'D', 'D'}, {'opt_material': None}]

arg_types = [('opt_material', 'virtual', 'state'), ('opt_material', 'state', 'virtual'), ('opt_material', 'parameter_v', 'parameter_s')]

globals_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
globals_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('v_weak', 's_weak', 'eval')

name = 'dw_v_dot_grad_s'

set_arg_types()

class sfepy.terms.terms_dot.VectorDotScalarTerm(name, arg_str, integral, region, **kwargs)
Volume dot product of a vector and a scalar. Can be evaluated.

Definition
\[
\int_\Omega \mathbf{v} \cdot c p, \quad \int_\Omega u \cdot c q
\]
Call signature

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_vm_dot_s</td>
<td>(material, virtual, state)</td>
</tr>
<tr>
<td></td>
<td>(material, state, virtual)</td>
</tr>
<tr>
<td></td>
<td>(material, parameter_v, parameter_s)</td>
</tr>
</tbody>
</table>

Arguments 1
- material : \(c\)
- virtual/parameter_v: \(v\)
- state/parameter_s: \(p\)

Arguments 2
- material : \(c\)
- state : \(u\)
- virtual : \(q\)

arg_shapes = [[{'material': 'D', 'virtual/v_weak': ('D', None), 'state/v_weak': 1, 'virtual/s_weak': (1, None), 'state/s_weak': 'D', 'parameter_v': 'D', 'parameter_s': 1}]

arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter_v', 'parameter_s'))

static d_dot(out, mat, val1_qp, val2_qp, geo)
static dw_dot(out, mat, val_qp, bfve, bfsc, geo, fmode)
get_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('v_weak', 's_weak', 'eval')
name = 'dw_vm_dot_s'
set_arg_types()

sfepy.terms.terms_elastic module

class sfepy.terms.terms_elastic.CauchyStrainTerm(name, arg_str, integral, region, **kwargs)

Evaluate Cauchy strain tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12]. The last three (non-diagonal) components are doubled so that it is energetically conjugate to the Cauchy stress tensor with the same storage.

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

\[
\int_D \varepsilon(w)
\]
Call signature

```
ev_cauchy_strain (parameter)
```

Arguments

- parameter : \( w \)

arg_shapes = {'parameter': 'D'}
 arg_types = ('parameter',)

```python
static function(out, strain, vg, fmode)
```

get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet_extra')

name = 'ev_cauchy_strain'

class sfepy.terms.terms_elastic.CauchyStressETHTerm(name, arg_str, integral, region, **kwargs)

Evaluate fading memory Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as
\([11, 22, 33, 12, 13, 23]\), in 2D it has 3 components with the indices ordered as \([11, 22, 12]\).

Assumes an exponential approximation of the convolution kernel resulting in much higher efficiency.

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

```
\int_0^t \int_0^t \mathcal{H}_{ijkl}(t - \tau) e_{kl}(w(\tau)) d\tau
```

Call signature

```
ev_cauchy_stress_eth (ts, material_0, material_1, parameter)
```

Arguments

- ts : TimeStepper instance
- material_0 : \( \mathcal{H}_{ijkl}(0) \)
- material_1 : \( \exp(-\lambda \Delta t) \) (decay at \( t_1 \))
- parameter : \( w \)

arg_shapes = {'material_0': 'S, S', 'material_1': '1, 1', 'parameter': 'D'}
 arg_types = ('ts', 'material_0', 'material_1', 'parameter')

get_eval_shape(ts, mat0, mat1, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(ts, mat0, mat1, state, mode=None, term_mode=None, diff_var=None, **kwargs)
class sfepy.terms.terms_elastic.CauchyStressTHTerm(name, arg_str, integral, region, **kwargs)
Evaluate fading memory Cauchy stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition
\[ \int_{t_1}^{t} \int_{0}^{t} H_{ijkl}(t-\tau) \varepsilon_{kl}(w(\tau)) d\tau \]

Call signature
```
\text{ev\_cauchy\_stress\_th} \ (ts, \ material, \ parameter)
```

Arguments
- \(ts\) : TimeStepper instance
- \(material\) : \(H_{ijkl}(\tau)\)
- \(parameter\) : \(w\)

arg_shapes = {'material': 'N, S, S', 'parameter': 'D'}
arg_types = ('ts', 'material', 'parameter')

get_eval_shape(ts, mats, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(ts, mats, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'ev\_cauchy\_stress\_th'

class sfepy.terms.terms_elastic.CauchyStressTerm(name, arg_str, integral, region, **kwargs)
Evaluate Cauchy stress tensor.

It is given in the usual vector form 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
\[ \int_{\Omega} D_{ijkl} \varepsilon_{kl}(w) \]

Call signature
```
\text{ev\_cauchy\_stress} \ (material, \ parameter)
```

Arguments
- \(material\) : \(D_{ijkl}\)
- \(parameter\) : \(w\)
arg_shapes = {'material': 'S, S', 'parameter': 'D'}

arg_types = ('material', 'parameter')

static function(out, coef, strain, mat, vg, fmode)

get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet_extra')

name = 'ev_cauchy_stress'

class sfepy.terms.terms_elastic.ElasticWaveCauchyTerm(name, arg_str, integral, region, **kwargs)

Elastic dispersion term involving the wave strain \( g_{ij}(u) = \frac{1}{2}(u_i \kappa_j + \kappa_i u_j) \), with the wave vector \( \kappa \) and the elastic strain \( e_{ij} \). \( D_{ijkl} \) is given in the usual matrix form exploiting symmetry: in 3D it is \( 6 \times 6 \) with the indices ordered as \([11, 22, 33, 12, 13, 23]\), in 2D it is \( 3 \times 3 \) with the indices ordered as \([11, 22, 12]\).

\[
\int_\Omega D_{ijkl} g_{ij}(u) c_{kl}(v) \\
\int_\Omega D_{ijkl} g_{ij}(u) c_{kl}(v)
\]

Call signature

dw_elastic_wave_cauchy (material_1, material_2, virtual, state)
                  (material_1, material_2, state, virtual)

Arguments 1

• material_1 : \( D_{ijkl} \)
• material_2 : \( \kappa \)
• virtual : \( v \)
• state : \( u \)

Arguments 2

• material_1 : \( D_{ijkl} \)
• material_2 : \( \kappa \)
• state : \( u \)
• virtual : \( v \)

arg_shapes = {'material_1': 'S, S', 'material_2': '.: D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material_1', 'material_2', 'virtual', 'state'), ('material_1', 'material_2', 'state', 'virtual'))

static function(out, out_qp, geo, fmode)

geometries = ['2_3', '2_4', '3_4', '3_8']
def get_fargs(mat, kappa, gvar, evar, mode=None, term_mode=None, diff_var=None, **kwargs):
    modes = ('ge', 'eg')
    name = 'dw_elastic_wave_cauchy'

class sfepy.terms.terms_elastic.ElasticWaveTerm(name, arg_str, integral, region, **kwargs):
    Elastic dispersion term involving the wave strain $g_{ij}$, $g_{ij}(u) = \frac{1}{2}(u_ig_i + \kappa_iu_i)$, with the wave vector $\kappa$. $D_{ijkl}$ is given in the usual matrix form exploiting symmetry: in 3D it is $6 \times 6$ with the indices ordered as $[11, 22, 33, 12, 13, 23]$, in 2D it is $3 \times 3$ with the indices ordered as $[11, 22, 12]$.

    Definition
    $\int_{\Omega} D_{ijkl} g_{ij}(u) g_{kl}(u)$

    Call signature

    dw_elastic_wave (material_1, material_2, virtual, state)

    Arguments
    • material_1 : $D_{ijkl}$
    • material_2 : $\kappa$
    • virtual : $v$
    • state : $u$

    arg_shapes = {'material_1': 'S, S', 'material_2': '.', 'virtual': 'D', 'state': 'D', 'virtual': ('D', 'state')}

    arg_types = ('material_1', 'material_2', 'virtual', 'state')

    static function(out, out_qp, geo, fmode)

    geometries = ['2_3', '2_4', '3_4', '3_8']

    get_fargs(mat, kappa, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

    name = 'dw_elastic_wave'

class sfepy.terms.terms_elastic.LinearElasticETHTerm(name, arg_str, integral, region, **kwargs):
    This term has the same definition as dw_lin_elastic_th, but assumes an exponential approximation of the convolution kernel resulting in much higher efficiency. Can use derivatives.

    Definition
    $\int_{\Omega} \left[ \int_0^t H_{ijkl}(t - \tau) e_{kl}(u(\tau)) \, d\tau \right] e_{ij}(u)$

    Call signature

    dw_lin_elastic_eth (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: $v$
• state: $u$

arg_shapes = {'material_0': 'S, S', 'material_1': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('ts', 'material_0', 'material_1', 'virtual', 'state')

static function()

get_fargs(ts, mat0, mat1, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_lin_elastic_eth'

class sfepy.terms.terms_elastic.LinearElasticIsotropicTerm(name, arg_str, integral, region, **kwargs)

Isotropic linear elasticity term.

Definition

$$\int_{\Omega} D_{ijkl} e_{ij}(v)e_{kl}(u)$$

with

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

Call signature

<table>
<thead>
<tr>
<th>dw_lin_elastic_iso</th>
<th>(material_1, material_2, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material_1, material_2, parameter_1, parameter_2)</td>
</tr>
</tbody>
</table>

Arguments

• material_1: $\lambda$
• material_2: $\mu$
• virtual/parameter_1: $v$
• state/parameter_2: $u$

arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('material_1', 'material_2', 'virtual', 'state'), ('material_1', 'material_2', 'parameter_1', 'parameter_2'))

geometries = ['2_3', '2_4', '3_4', '3_8']

get_eval_shape(mat1, mat2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(lam, mu, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_lin_elastic_iso'
class sfepy.terms.terms_elastic.LinearElasticTHTerm(name, arg_str, integral, region, **kwargs)

Fading memory linear elastic (viscous) term. Can use derivatives.

Definition

\[ \int_{\Omega} \left[ \int_{0}^{t} \mathcal{H}_{ijkl}(t - \tau) e_{kl}(u(\tau)) d\tau \right] e_{ij}(v) \]

Call signature

\texttt{dw\_lin\_elastic\_th} (ts, material, virtual, state)

Arguments

\begin{itemize}
  \item ts : TimeStepper instance
  \item material : \( \mathcal{H}_{ijkl}(\tau) \)
  \item virtual : \( v \)
  \item state : \( u \)
\end{itemize}

arg_shapes = {'material': 'N, S, S', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('ts', 'material', 'virtual', 'state')

static function()

get_fargs(ts, mats, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw\_lin\_elastic\_th'

class sfepy.terms.terms_elastic.LinearElasticTerm(name, arg_str, integral, region, **kwargs)

General linear elasticity term, with \( D_{ijkl} \) given in the usual matrix form exploiting symmetry: in 3D it is \( 6 \times 6 \) with the indices ordered as \([11, 22, 33, 12, 13, 23]\), in 2D it is \( 3 \times 3 \) with the indices ordered as \([11, 22, 12]\). Can be evaluated. Can use derivatives.

Definition

\[ \int_{\Omega} D_{ijkl} e_{ij}(v) e_{kl}(u) \]

Call signature

\texttt{dw\_lin\_elastic} (material, virtual, state)

\texttt{dw\_lin\_elastic} (material, parameter_1, parameter_2)

Arguments 1

\begin{itemize}
  \item material : \( D_{ijkl} \)
  \item virtual : \( v \)
  \item state : \( u \)
\end{itemize}

Arguments 2

\begin{itemize}
  \item material : \( D_{ijkl} \)
  \item parameter_1 : \( w \)
• parameter_2 : \( u \)

\[
\text{arg\_shapes} = \{ \text{'material': 'S', 'S'}, \text{'parameter\_1': 'D'}, \text{'parameter\_2': 'D'},
\text{'state': 'D'}, \text{'virtual': ('D', 'state')} \}
\]

\[
\text{arg\_types} = (\text{('material', 'virtual', 'state')}, (\text{'material', 'parameter\_1'}, \text{'parameter\_2'})
\]

\[
\text{get\_eval\_shape}(\text{mat, virtual, state, mode=\text{None, term\_mode=\text{None, diff\_var=\text{None, **kwargs}})}
\]

\[
\text{get\_fargs}(\text{mat, virtual, state, mode=\text{None, term\_mode=\text{None, diff\_var=\text{None, **kwargs}})}
\]

\[
\text{modes} = (\text{'weak', 'eval'})
\]

\[
\text{name} = \text{'dw\_lin\_elastic'}
\]

\[
\text{set\_arg\_types}()
\]

class sfepy.terms.terms_elastic.LinearPrestressTerm(name, arg\_str, integral, region, **kwargs)

Linear prestress term, with the prestress \( \sigma_{ij} \) given either in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as \([11, 22, 33, 12, 13, 23]\), in 2D it has 3 components with the indices ordered as \([11, 22, 12]\), or in the matrix (possibly non-symmetric) form. Can be evaluated.

**Definition**

\[
\int_{\Omega} \sigma_{ij} e_{ij}(\nu)
\]

**Call signature**

<table>
<thead>
<tr>
<th>dw_lin_prestress</th>
<th>(material, virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dw_lin_prestress</td>
<td>(material, parameter)</td>
</tr>
</tbody>
</table>

**Arguments 1**

- material : \( \sigma_{ij} \)
- virtual : \( \nu \)

**Arguments 2**

- material : \( \sigma_{ij} \)
- parameter : \( u \)

\[
\text{arg\_shapes} = [\{ \text{'material': 'S', '1'}, \text{'virtual': ('D', \text{None}), \text{'parameter': 'D'}},
\{ \text{'material': 'D', 'D'} \}
\]

\[
\text{arg\_types} = (\text{('material', 'virtual')}, (\text{'material', 'parameter'})
\]

\[
\text{d\_lin\_prestress}(\text{out, strain, mat, vg, fmode})
\]

\[
\text{get\_eval\_shape}(\text{mat, virtual, mode=\text{None, term\_mode=\text{None, diff\_var=\text{None, **kwargs}})}
\]

\[
\text{get\_fargs}(\text{mat, virtual, mode=\text{None, term\_mode=\text{None, diff\_var=\text{None, **kwargs}})}
\]

\[
\text{modes} = (\text{'weak', 'eval'})
\]

\[
\text{name} = \text{'dw\_lin\_prestress'}
\]
**Linear Spring Element**

**Definition**

\[ f^{(i)} = -f^{(j)} = k(u^{(j)} - u^{(i)}) \]

in a region connecting nodes \( i, j \)

**Call signature**

```
dw_lin_spring (material, virtual, state)
```

**Arguments**

- `material`: \( k \)
- `virtual`: \( v \)
- `state`: \( u \)

**Static function**

```
def get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

**Name**

`'dw_lin_spring'`

**Linear (Pre)strain Fiber Term**

**Definition**

\[ \int_{\Omega} D_{ijkl} e_{ij}(v) (d_k d_l) \]

**Call signature**

```
dw_lin_strain_fib (material_1, material_2, virtual)
```

**Arguments**

- `material_1`: \( D_{ijkl} \)
- `material_2`: \( d \)
- `virtual`: \( v \)

**Static function**

```
static function()
```

---

**Additional Information**

If you need further assistance or have specific questions about this content, feel free to ask!
get_fargs(mat1, mat2, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_lin_strain_fib'

class sfepy.terms.terms_elastic.LinearTrussInternalForceTerm(name, arg_str, integral, region, **kwargs)

Evaluate internal force in the element direction. To be used with 'el_avg' or 'qp' evaluation modes which give the same results. The material parameter $E_A$ is equal to Young modulus times element cross-section. The internal force is given by $F(i) = -F(j) = E_A/l(U(j) - U(i))$, where $l$ is the element length and $U, F$ are the nodal displacements and the nodal forces in the element direction.

Definition

$$F = E_A/l(U(j) - U(i))$$

$\forall$ elements $T_{K}^{i,j}$
in a region connecting nodes $i, j$

Call signature

dw_lin_strain_fib (material, virtual, state)

Arguments

- material : $EA$
- parameter : $w$

arg_shapes = {'material': '1, 1', 'parameter': 'D'}

arg_types = ('material', 'parameter')

static function(out, mat, vec, mtx_t)

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 = 'dw_lin_strain_fib'

class sfepy.terms.terms_elastic.LinearTrussTerm(name, arg_str, integral, region, **kwargs)

Evaluate internal force in the element direction. To be used with 'el_avg' or 'qp' evaluation modes which give the same results. The material parameter $E_A$ is equal to Young modulus times element cross-section. The internal force is given by $F(i) = -F(j) = E_A/l(U(j) - U(i))$, where $l$ is the element length and $U, F$ are the nodal displacements and the nodal forces in the element direction.

Definition

$$F(i) = -F(j) = E_A/l(U(j) - U(i))$$

$\forall$ elements $T_{K}^{i,j}$
in a region connecting nodes $i, j$

Call signature

dw_lin_truss (material, virtual, state)

Arguments
• material : $EA$
• parameter : $w$

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')

static function(out, mat, vec, mtx_t, length, diff_var)
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
static get_mtx_t_and_length(coors)

name = 'dw_lin_truss'

class sfepy.terms.terms_elastic.NonsymElasticTerm(name, arg_str, integral, region, **kwargs)
Elasticity term with non-symmetric gradient. The indices of matrix $D_{ijkl}$ are ordered as $[11, 12, 13, 21, 22, 23, 31, 32, 33]$ in 3D and as $[11, 12, 21, 22]$ in 2D.

Definition

$$\int_\Omega D \nabla u : \nabla v$$

Call signature

<table>
<thead>
<tr>
<th>dw_nonsym_elastic</th>
<th>(material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material, parameter_1, parameter_2)</td>
</tr>
</tbody>
</table>

Arguments 1

• material : $D$
• virtual : $v$
• state : $u$

Arguments 2

• material : $D$
• parameter_1 : $w$
• parameter_2 : $u$

arg_shapes = {'material': 'D2, D2', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state'), ('material', 'parameter_1', 'state'), ('parameter_2'))

geometries = ['2_3', '2_4', '3_4', '3_8']

generate_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
generate_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('weak', 'eval')
name = 'dw_nonsym_elastic'

def set_arg_types():

class sfepy.terms.terms_elastic.SDLinearElasticTerm(name, arg_str, integral, region, **kwargs):
    """Sensitivity analysis of the linear elastic term."
    ""
    Definition
    \[
    \int_\Omega \hat{D}_{ijkl} \varepsilon_{ij}(\nu) \varepsilon_{kl}(\nu) \\
    \hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \nu) - D_{ijkm} \frac{\partial \nu_m}{\partial x_i} - D_{ijmk} \frac{\partial \nu_k}{\partial x_m}
    \]
    """
    Call signature
    `ev_sd_lin_elastic` (material, parameter_w, parameter_u, parameter_mv)

    Arguments
    • material : \( D_{ijkl} \)
    • parameter_w : \( \nu \)
    • parameter_u : \( u \)
    • parameter_mv : \( \nu \)

    arg_shapes = {'material': 'S, S', 'parameter_mv': 'D', 'parameter_u': 'D', 'parameter_w': 'D'}

    arg_types = ('material', 'parameter_w', 'parameter_u', 'parameter_mv')

    function()

def get_eval_shape(mat, par_w, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs):

def get_fargs(mat, par_w, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs):

    name = 'ev_sd_lin_elastic'

sfepy.terms.terms_electric module

class sfepy.terms.terms_electric.ElectricSourceTerm(name, arg_str, integral, region, **kwargs):
    """Electric source term."
    ""
    Definition
    \[
    \int_\Omega c_s (\nabla \phi)^2
    \]
    """
    Call signature
    `dwElectricSource` (material, virtual, parameter)
Arguments

- material: \( c \) (electric conductivity)
- virtual: \( s \) (test function)
- parameter: \( \phi \) (given electric potential)

\[
\text{arg_shapes} = \{ \text{'material': '1, 1'}, \text{'parameter': 1}, \text{'virtual': (1, None)} \}
\]
\[
\text{arg_types} = (\text{'material'}, \text{'virtual'}, \text{'parameter'})
\]

static function()

get_fargs(mat, virtual, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_electric_source'

sfepy.terms.terms_fibres module

class sfepy.terms.terms_fibres.FibresActiveTLTerm(*args, **kwargs)

Hyperelastic active fibres term. Effective stress \( S_{ij} = Af_{\text{max}} \exp \left\{ -\left( \frac{\epsilon - \epsilon_{\text{opt}}}{s} \right)^2 \right\} d_id_j \), where \( \epsilon = E_{ij}d_id_j \) is the Green strain projected to the fibre direction \( d \).

Definition

\[
\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)
\]

Call signature

dw_tl_fib_a (material_1, material_2, material_3, material_4, material_5, virtual, state)

Arguments

- material_1: \( f_{\text{max}} \)
- material_2: \( \epsilon_{\text{opt}} \)
- material_3: \( s \)
- material_4: \( d \)
- material_5: \( A \)
- virtual: \( v \)
- state: \( u \)

\[
\text{arg_shapes} = \{ \text{'material_1': '1, 1'}, \text{'material_2': '1, 1'}, \text{'material_3': '1, 1'}, \text{'material_4': 'D, 1'}, \text{'material_5': '1, 1'}, \text{'state': 'D'}, \text{'virtual': ('D', 'state')} \}
\]
\[
\text{arg_types} = (\text{'material_1'}, \text{'material_2'}, \text{'material_3'}, \text{'material_4'}, \text{'material_5'}, \text{'virtual'}, \text{'state'})
\]

family_data_names = ['green_strain']
get_eval_shape(mat1, mat2, mat3, mat4, mat5, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat1, mat2, mat3, mat4, mat5, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_tl_fib_a'

static stress_function(out, pars, green_strain, fibre_data=None)

static tan_mod_function(out, pars, green_strain, fibre_data=None)

sfepy.terms.terms_fibres.compute_fibre_strain(green_strain, omega)

Compute the Green strain projected to the fibre direction.

sfepy.terms.terms_fibres.create_omega(fdir)

Create the fibre direction tensor \( \omega_{ij} = d_i d_j \).

sfepy.terms.terms_hyperelastic_base module

class sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm(name, arg_str, integral, region, **kwargs)

Deformation gradient \( F \) in quadrature points for \( \text{term_mode}='\text{def_grad}' \) (default) or the jacobian \( J \) if \( \text{term_mode}='\text{jacobian}' \).

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

\[
F = \frac{\partial x}{\partial X} |_{qp} = I + \frac{\partial u}{\partial X} |_{qp},
\]

\[
x = X + u, J = \det (F)
\]

Call signature

\[
\text{ev_def_grad} \ (\text{parameter})
\]

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 (→ residual) or tangent modulus (→ tangent stiffness matrix), i.e. constitutive relation type (CRT) related data. The CRT data are computed in subclasses implementing particular CRT (e.g. neo-Hookean material), in self.compute_crt_data().

Modes:
• 0: total formulation
• 1: updated formulation

Notes
This is not a proper Term!

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')
compute_stress(mat, family_data, **kwargs)
compute_tan_mod(mat, family_data, **kwargs)
static function(out, fun, *args)
get_eval_shape(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
static integrate(out, val_qp, vg, fmode)

class sfepy.terms.terms_hyperelastic_base.HyperElasticFamilyData(**kwargs)
Base class for hyperelastic family data.

The common (family) data are cached in the evaluate cache of state variable.

data_shapes = {'det_f': ('n_el', 'n_qp', 1, 1), 'green_strain': ('n_el', 'n_qp', 'sym', 1), 'in2_b': ('n_el', 'n_qp', 1, 1), 'inv2_c': ('n_el', 'n_qp', 1, 1),
inv_f: ('n_el', 'n_qp', 'dim', 'dim'), mtx_f: ('n_el', 'n_qp', 'dim', 'dim'),
sym_b: ('n_el', 'n_qp', 'sym', 1), sym_c: ('n_el', 'n_qp', 'sym', 1),
sym_inv_c: ('n_el', 'n_qp', 'sym', 1), 'tr_b': ('n_el', 'n_qp', 1, 1), 'tr_c': ('n_el', 'n_qp', 1, 1)}
init_data_struct(state_shape, name='family_data')

sfepy.terms.terms_hyperelastic_tl module

class sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm(*args, **kwargs)
Hyperelastic bulk active term. Stress \( S_{ij} = A J C_{ij}^{-1} \), where \( A \) is the activation in \([0, F_{\text{max}}]\).

Definition

\[
\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; u)
\]

Call signature
### dw_tl_bulk_active (material, virtual, state)

**Arguments**

- material : $A$
- virtual : $v$
- state : $u$

**family_data_names** = ['det_f', 'sym_inv_c']

**name** = 'dw_tl_bulk_active'

**static stress_function**()

**static tan_mod_function**()

```python
class sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm(*args, **kwargs)
```

Hyperelastic bulk penalty term. Stress $S_{ij} = K(J - 1)JC_{ij}^{-1}$.

**Definition**

$$\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)$$

**Call signature**

### dw_tl_bulk_penalty (material, virtual, state)

**Arguments**

- material : $K$
- virtual : $v$
- state : $u$

**family_data_names** = ['det_f', 'sym_inv_c']

**name** = 'dw_tl_bulk_penalty'

**static stress_function**()

**static tan_mod_function**()

```python
class sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm(*args, **kwargs)
```

Hyperelastic bulk pressure term. Stress $S_{ij} = -pJC_{ij}^{-1}$.

**Definition**

$$\int_{\Omega} S_{ij}(p) \delta E_{ij}(u; v)$$

**Call signature**

### dw_tl_bulk_pressure (virtual, state, state_p)
Arguments

- virtual : \( v \)
- state : \( u \)
- state_p : \( p \)

arg_shapes = {'state': 'D', 'state_p': 1, 'virtual': ('D', 'state')}

arg_types = ('virtual', 'state', 'state_p')

calculate_data(family_data, mode, **kwargs)

family_data_names = ['det_f', 'sym_inv_c']

generate_eval_shape(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

generate_fargs(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_tl_bulk_pressure'

static stress_function()

static tangent_mod_u_function()

static weak_dp_function()

static weak_function()

class sfepy.terms.terms_hyperelastic_tl.DiffusionTLTerm(*args, **kwargs)

Diffusion term in the total Lagrangian formulation with linearized deformation-dependent permeability \( K(u) = J F^{-1} k f(J) F^{-T} \), where \( u \) relates to the previous time step \( (n-1) \) and \( f(J) = \max \left( 0, \left( 1 + \frac{J-1}{N_f} \right)^2 \right) \) expresses the dependence on volume compression/expansion.

Definition

\[
\int_\Omega K(u^{(n-1)}) : \frac{\partial q}{\partial X} \frac{\partial p}{\partial X}
\]

Call signature

dw_tl_diffusion (material_1, material_2, virtual, state, parameter)
family_data_names = ['mtx_f', 'det_f']

static function()

get_eval_shape(perm, ref_porosity, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(perm, ref_porosity, virtual, state, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_tl_diffusion'

class sfepy.terms.terms_hyperelastic_tl.GenYeohTLTerm(*args, **kwargs)

Hyperelastic generalized Yeoh term [1]. Effective stress \( S_{ij} = 2pK(I_1 - 3)J^{-\frac{3}{2}}(\delta_{ij} - \frac{1}{3}C_{kk}C_{ij}^{-1}) \).

Definition

\[
\int_{\Omega} S_{ij}(\mathbf{u}) \delta E_{ij}(\mathbf{u}; \mathbf{v})
\]

Call signature

\[
\text{dw_tl_he_genyeoh} \quad (\text{material, virtual, state})
\]

Arguments

- material : \( p, K \)
- virtual : \( \mathbf{v} \)
- state : \( \mathbf{u} \)


arg_shapes = {'material': '1, 2', 'state': 'D', 'virtual': ('D', 'state')}

family_data_names = ['det_f', 'tr_c', 'sym_inv_c']

geometries = ['3_4', '3_8']

name = 'dw_tl_he_genyeoh'

stress_function(out, mat, *fargs, **kwargs)

tan_mod_function(out, mat, *fargs, **kwargs)

class sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTLBase(*args, **kwargs)

Base class for all hyperelastic surface terms in TL formulation family.

get_family_data = HyperElasticSurfaceTFLFamilyData

class sfepy.terms.terms_hyperelastic_tl.HyperElasticSurfaceTFLFamilyData(**kwargs)

Family data for TL formulation applicable for surface terms.

cache_name = 'tl_surface_common'

data_names = ('mtx_f', 'det_f', 'inv_f')
static family_function()

class sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase(*args, **kwargs)

Base class for all hyperelastic terms in TL formulation family.

The subclasses should have the following static method attributes:
- `stress_function()` (the stress)
- `tan_mod_function()` (the tangent modulus)

The common (family) data are cached in the evaluate cache of state variable.

get_family_data = HyperElasticTLFamilyData

hyperelastic_mode = 0

static weak_function()

class sfepy.terms.terms_hyperelastic_tl.HyperElasticTLFamilyData(**kwargs)

Family data for TL formulation.

cache_name = 'tl_common'

data_names = ('mtx_f', 'det_f', 'sym_c', 'tr_c', 'in2_c', 'sym_inv_c', 'green_strain')

static family_function()

class sfepy.terms.terms_hyperelastic_tl.MooneyRivlinTLTerm(*args, **kwargs)

Hyperelastic Mooney-Rivlin term. Effective stress
\[ S_{ij} = \kappa J \frac{1}{3} (C_{kk} \delta_{ij} - C_{ij} - \frac{2}{3} I_2 C_{ij}^{-1}). \]

Definition

\[ \int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v) \]

Call signature

**dw_tl_he_mooney_rivlin** (material, virtual, state)

Arguments

- material : \( \kappa \)
- virtual : \( v \)
- state : \( u \)

family_data_names = ['det_f', 'tr_c', 'sym_inv_c', 'sym_c', 'in2_c']

name = 'dw_tl_he_mooney_rivlin'

static stress_function()

static tan_mod_function()

class sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm(*args, **kwargs)

Hyperelastic neo-Hookean term. Effective stress
\[ S_{ij} = \mu J \frac{1}{3} (\delta_{ij} - \frac{1}{3} C_{kk} C_{ij}^{-1}). \]

Definition
\[
\int_\Omega S_{ij}(u) \delta E_{ij}(u; v)
\]

Call signature

\begin{verbatim}
dw_tl_he_neohook (material, virtual, state)
\end{verbatim}

Arguments

- material : \(\mu\)
- virtual : \(v\)
- state : \(u\)

family_data_names = ['det_f', 'tr_c', 'sym_inv_c']

name = 'dw_tl_he_neohook'

static stress_function()

static tan_mod_function()

class sfepy.terms.terms_hyperelastic_tl.OgdenTLTerm(*args, **kwargs)

Single term of the hyperelastic Ogden model [1] with the strain energy density

\[
W = \frac{\mu}{\alpha} \left( \lambda_1^\alpha + \lambda_2^\alpha + \lambda_3^\alpha - 3 \right),
\]

where \(\lambda_k, k = 1, 2, 3\) are the principal stretches, whose squares are the principal values of the right Cauchy-Green deformation tensor \(C\).

Effective stress (2nd Piola-Kirchhoff) is [2]

\[
S_{ij} = 2 \frac{\partial W}{\partial C_{ij}} = \sum_{k=1}^{3} S^{(k)} N_i^{(k)} N_j^{(k)},
\]

where the principal stresses are

\[
S^{(k)} = J^{-2/3} \left( \mu \lambda_k^{\alpha-2} - \sum_{j=1}^{3} \mu \frac{\lambda_j^{\alpha}}{3 \lambda_k^{2}} \right), \quad k = 1, 2, 3.
\]

and \(N^{(k)}, k = 1, 2, 3\) are the eigenvectors of \(C\).

Definition

\[
\int_\Omega S_{ij}(u) \delta E_{ij}(u; v)
\]

Call signature

\begin{verbatim}
dw_tl_he_ogden (material, virtual, state)
\end{verbatim}

Arguments

- material : \(p, K\)
- virtual : \(v\)


det_f, sym_c, tr_c, sym_inv_c

geometries = ['3_4', '3_8']

name = 'dw_tl_he_ogden'

stress_function(out, mat, *fargs, **kwargs)
tan_mod_function(out, mat, *fargs, **kwargs)
class sfepy.terms.terms_hyperelastic_tl.SurfaceFluxTLTerm(*args, **kwargs)

Surface flux term in the total Lagrangian formulation, consistent with DiffusionTLTerm.

Definition

\[ \int_{\Gamma} \nu \cdot K(u^{(n-1)}) \frac{\partial p}{\partial X} \]

Call signature

ev_tl_surface_flux (material_1, material_2, parameter_1, parameter_2)

Arguments

- material_1 : \( k \)
- material_2 : \( N_f \)
- parameter_1 : \( p \)
- parameter_2 : \( u^{(n-1)} \)

arg_shapes = {'material_1': 'D', 'material_2': '1, 1', 'parameter_1': 1, 'parameter_2': 'D'}

arg_types = ('material_1', 'material_2', 'parameter_1', 'parameter_2')

family_data_names = ['det_f', 'inv_f']

static function()

get_eval_shape(perm, ref_porosity, pressure, displacement, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(perm, ref_porosity, pressure, displacement, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet_extra'
name = 'ev_tl_surface_flux'

class sfepy.terms.terms_hyperelastic_tl.SurfaceTractionTLTerm(*args, **kwargs)
Surface traction term in the total Lagrangian formulation, expressed using $\nu$, the outward unit normal vector w.r.t. the undeformed surface, $F(u)$, the deformation gradient, $J = \det(F)$, and $\sigma$ a given traction, often equal to a given pressure, i.e. $\sigma = \pi F$.

Definition

$$\int_{\Gamma} \nu \cdot F^{-1} \cdot \sigma \cdot v J$$

Call signature

`dw_tl_surface_traction` (opt_material, virtual, state)

Arguments

- material : $\sigma$
- virtual : $v$
- state : $u$

arg_shapes = [{'opt_material': 'D', 'D'}, 'virtual': ('D', 'state'), 'state': 'D'}, {'opt_material': None}]
arg_types = ('opt_material', 'virtual', 'state')
family_data_names = ['det_f', 'inv_f']
static function()
get_fargs(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet_extra'
name = 'ev_tl_surface_flux'

class sfepy.terms.terms_hyperelastic_tl.VolumeSurfaceTLTerm(*args, **kwargs)
Volume of a $D$-dimensional domain, using a surface integral in the total Lagrangian formulation, expressed using $\nu$, the outward unit normal vector w.r.t. the undeformed surface, $F(u)$, the deformation gradient, and $J = \det(F)$. Uses the approximation of $u$ for the deformed surface coordinates $x$.

Definition

$$\frac{1}{D} \int_{\Gamma} \nu \cdot F^{-1} \cdot x J$$

Call signature

`ev_tl_volume_surface` (parameter)

Arguments

- parameter : $u$
arg_shapes = {'parameter': 'D'}
arg_types = ('parameter',)
family_data_names = ['det_f', 'inv_f']
static function()
get_eval_shape(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet_extra'
name = 'ev_tl_volume_surface'

class sfepy.terms.terms_hyperelastic_tl.VolumeTLTerm(*args, **kwargs)
Volume term (weak form) in the total Lagrangian formulation.

Definition
\[ \int_{\Omega} q J(u) \]
volume mode: vector for \( K \leftarrow T_h : \int_{T_K} J(u) \)

rel_volume mode: vector for \( K \leftarrow T_h : \int_{T_K} J(u) / \int_{T_K} 1 \)

Call signature

**dw_tl_volume** (virtual, state)

Arguments
- virtual : q
- state : u

arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f', 'sym_inv_c']
static function()
get_eval_shape(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_tl_volume'
**sfepy.terms.terms_hyperelastic_ul module**

*class* `sfepy.terms.terms_hyperelastic_ul.BulkPenaltyULTerm(*args, **kwargs)`

Hyperelastic bulk penalty term. Stress \( \tau_{ij} = K(J - 1)J \delta_{ij} \).

**Definition**

\[
\int_{\Omega} L \tau_{ij}(u) e_{ij}(\delta v) / J
\]

**Call signature**

```python
dw_ul_bulk_penalty (material, virtual, state)
```

**Arguments**

- `material`: \( K \)
- `virtual`: \( v \)
- `state`: \( u \)

**family_data_names** = ["det_f"]

**name** = 'dw Ul bulk penalty'

**static stress_function**()

**static tan_mod_function**()

*class* `sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm(*args, **kwargs)`

Hyperelastic bulk pressure term. Stress \( S_{ij} = -pJ \delta_{ij} \).

**Definition**

\[
\int_{\Omega} L \tau_{ij}(u) e_{ij}(\delta v) / J
\]

**Call signature**

```python
dw_ul_bulk_pressure (virtual, state, state_p)
```

**Arguments**

- `virtual`: \( v \)
- `state`: \( u \)
- `state_p`: \( p \)

**arg_shapes** = {
    'state': 'D',
    'state_p': 1,
    'virtual': ('D', 'state')
}

**arg_types** = ('virtual', 'state', 'state_p')

**compute_data**(family_data, mode, **kwargs)

**family_data_names** = ['det_f', 'sym_b']
static family_function()

get_eval_shape(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(virtual, state, state_p, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_ul_bulk_pressure'

static stress_function()

static tan_mod_u_function()

static weak_dp_function()

static weak_function()

class sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm(*args, **kwargs)

Compressibility term for the updated Lagrangian formulation

Definition

\[ \int_Ω \frac{1}{\gamma p q} \]

Call signature

```
dw_ul_compressible  (material, virtual, state, parameter_u)
```

Arguments

- material : \( \gamma \)
- virtual : \( q \)
- state : \( p \)
- parameter_u : \( u \)

arg_shapes = { 'material': '1, 1', 'parameter_u': 'D', 'state': 1, 'virtual': (1, 'state') }

arg_types = ('material', 'virtual', 'state', 'parameter_u')

family_data_names = ['mtx_f', 'det_f']

static function()

get_fargs(bulk, virtual, state, parameter_u, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_ul_compressible'

class sfepy.terms.terms_hyperelastic_ul.HyperElasticULBase(*args, **kwargs)

Base class for all hyperelastic terms in UL formulation family.

The subclasses should have the following static method attributes:

- stress_function() (the stress)
- tan_mod_function() (the tangent modulus)

get_family_data = HyperElasticULFamilyData

hyperelastic_mode = 1
class sfepy.terms.terms_hyperelastic_ul.HyperElasticULFamilyData(**kwargs)

Family data for UL formulation.

    cache_name = 'ul_common'
    data_names = ('mtx_f', 'det_f', 'sym_b', 'tr_b', 'in2_b', 'green_strain')

static family_function()

class sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm(*args, **kwargs)

Hyperelastic Mooney-Rivlin term.

Definition

\[ \int_{\Omega} L \tau_{ij}(u) e_{ij}(\delta v) / J \]

Call signature

    dw_ul_he_mooney_rivlin (material, virtual, state)

Arguments

- material : \( \kappa \)
- virtual : \( v \)
- state : \( u \)

family_data_names = ['det_f', 'tr_b', 'sym_b', 'in2_b']

name = 'dw_ul_he_mooney_rivlin'

static stress_function()

static tan_mod_function()

class sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm(*args, **kwargs)

Hyperelastic neo-Hookean term. Effective stress \( \tau_{ij} = 2 \mu J^{-\frac{3}{2}} (b_{ij} - \frac{1}{3} b_{kk} \delta_{ij}) \).

Definition

\[ \int_{\Omega} L \tau_{ij}(u) e_{ij}(\delta v) / J \]

Call signature

    dw_ul_he_neohook (material, virtual, state)

Arguments

- material : \( \mu \)
- virtual : \( v \)
- state : \( u \)
family_data_names = ['det_f', 'tr_b', 'sym_b']
name = 'dw_ul_he_neohook'
static stress_function()
static tan_mod_function()

class sfepy.terms.terms_hyperelastic_ul.VolumeULTerm(*args, **kwargs)
    Volume term (weak form) in the updated Lagrangian formulation.
    Definition
    \[ \int_{\Omega} qJ(\mathbf{u}) \]
    volume mode: vector for \( K \leftarrow \mathcal{I}_h : \int_{T_K} J(\mathbf{u}) \)
    \[ \text{rel_volume mode: vector for } K \leftarrow \mathcal{I}_h : \frac{\int_{T_K} J(\mathbf{u})}{\int_{T_K} 1} \]
    Call signature
    dw_ul_volume (virtual, state)

Arguments
- virtual : \( q \)
- state : \( u \)

arg_shapes = {'state': 'D', 'virtual': (1, None)}
arg_types = ('virtual', 'state')
family_data_names = ['mtx_f', 'det_f']
static function()
get_eval_shape(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
get_fargs(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_ul_volume'

sfepy.terms.terms_jax module

Proof-of-concept JAX-based terms supporting automatic differentiation.
class sfepy.terms.terms_jax.LinearElasticLADTerm(name, arg_str, integral, region, **kwargs)
    Homogeneous isotropic linear elasticity term differentiable w.r.t. material parameters \( \lambda, \mu \) (Lamé’s parameters).
    Definition
    \[ \int_{\Omega} D_{ijkl} e_{ij}(\mathbf{u}) e_{kl}(\mathbf{u}) \]
    with
    \[ D_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda \delta_{ij}\delta_{kl} \]
    Call signature
**Definition**

\[
\int_{\Omega} D_{ijkl} \varepsilon_{ij}(v) \varepsilon_{kl}(u)
\]

with

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

where

\[
\lambda = E\nu/((1 + \nu)(1 - 2\nu)),
\]

\[
\mu = E/2(1 + \nu)
\]
arg_shapes = {'material_1': '1, 1', 'material_2': '1, 1', 'state': 'D',
'virtual': ('D', 'state')}
arg_types = (('material_1', 'material_2', 'virtual', 'state'),)
diff_info = {'material_1': 1, 'material_2': 1}

static function(out, fun, fargs)

get_fargs(material1, material2, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)
name = 'dw_lin_elastic_yad'

class sfepy.terms.terms_jax.MassADTerm(name, arg_str, integral, region, **kwargs)
Homogeneous mass term differentiable w.r.t. the material parameter.

Definition
\[ \int_D \rho \mathbf{v} \cdot \mathbf{u} \]

Call signature

dw_mass_ad (material, virtual, state)

Arguments
- material_1: \( \rho \)
- virtual: \( \mathbf{v} \)
- state: \( \mathbf{u} \)

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = (('material', 'virtual', 'state'),)
diff_info = {'material': 1}

static function(out, fun, fargs)

get_fargs(material_density, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)
name = 'dw_mass_ad'

class sfepy.terms.terms_jax.NeoHookeanTLADTerm(name, arg_str, integral, region, **kwargs)
Homogeneous Hyperelastic neo-Hookean term differentiable w.r.t. the material parameter. Effective stress \( S_{ij} = \mu J^{-\frac{2}{3}} (\delta_{ij} - \frac{1}{3} C_{kk} C^{-1}_{ij}) \).

Definition
\[ \int_{\Omega} S_{ij}(\mathbf{u}) \delta E_{ij}(\mathbf{u}; \mathbf{v}) \]

Call signature
### dw_tl_he_neohook_ad (material, virtual, state)

**Arguments**
- material : \( \mu \)
- virtual : \( v \)
- state : \( u \)

\[
\text{arg_shapes} = \{ 'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state') \}
\]
\[
\text{arg_types} = ( 'material', 'virtual', 'state' )
\]
\[
\text{diff_info} = \{ 'material': 1 \}
\]

**Static function**

```
static function(out, fun, fargs)
```

**Geometries**

- \[ [ '2_3', '2_4', '3_4', '3_8' ] \]

**Get Fargs**

```
get_fargs(material, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

**Modes**

- \[ 'weak' \]

**Name**

- \[ 'dw_tl_he_neohook_ad' \]

---

### dw_tl_he_ogden_ad (material_mu, material_alpha, virtual, state)

**Arguments**
- material_1 : \( \mu \)
- material_2 : \( \alpha \)
- virtual : \( v \)

---

### Definition

\[
\int_{\Omega} S_{ij}(u) \delta E_{ij}(u; v)
\]

### Call signature

---

2.3. Developer Guide
state : \( u \)

\[
\text{arg_shapes} = \{'\text{material\_alpha}' : '1, 1', '\text{material\_mu}' : '1, 1', '\text{state}' : 'D', '\text{virtual}' : ('D', 'state')\}
\]

\[
\text{arg_types} = ('\text{material\_mu}', '\text{material\_alpha}', '\text{virtual}', '\text{state}')
\]

diff_info = \{'\text{material\_alpha}' : 1, '\text{material\_mu}' : 1\}

static function(out, fun, fargs)

geometries = ['3_4', '3_8']

get_fargs(material_mu, material_alpha, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak',)

name = 'dw_tl_he_ogden_ad'

sfepy.terms.terms_jax.ceval_elasticity_l(lam, mu, vbfg, ubfg, det, cu)

sfepy.terms.terms_jax.ceval_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)

sfepy.terms.terms_jax.ceval_mass(density, vb, ubf, det, cu)

sfepy.terms.terms_jax.ceval_neohook(mu, vbfg, ubfg, det, cu)

sfepy.terms.terms_jax.ceval_ogden(mu, alpha, vbfg, ubfg, det, cu)

sfepy.terms.terms_jax.eval_alpha_ogden(mu, alpha, vbfg, ubfg, det, cu)

Vectorized version of ceval_ogden. Takes similar arguments as ceval_ogden but with additional array axes over which ceval_ogden is mapped.

Original documentation:

Jacobian of ceval_ogden with respect to positional argument(s) 1. Takes the same arguments as ceval_ogden but returns the jacobian of the output with respect to the arguments at positions 1.

sfepy.terms.terms_jax.eval_density_mass(density, vb, ubf, det, cu)

Vectorized version of ceval_mass. Takes similar arguments as ceval_mass but with additional array axes over which ceval_mass is mapped.

Original documentation:

Jacobian of ceval_mass with respect to positional argument(s) 0. Takes the same arguments as ceval_mass but returns the jacobian of the output with respect to the arguments at positions 0.

sfepy.terms.terms_jax.eval_elasticity_l(lam, mu, vbfg, ubfg, det, cu)

Vectorized version of ceval_elasticity_l. Takes similar arguments as ceval_elasticity_l but with additional array axes over which ceval_elasticity_l is mapped.

sfepy.terms.terms_jax.eval_elasticity_yp(young, poisson, plane, vhfg, ubfg, det, cu)

Vectorized version of ceval_elasticity_yp. Takes similar arguments as ceval_elasticity_yp but with additional array axes over which ceval_elasticity_yp is mapped.
Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) -1. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions -1.

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) -1. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions -1.

Vectorized version of `ceval_mass`. Takes similar arguments as `ceval_mass` but with additional array axes over which `ceval_mass` is mapped.

Original documentation:

Jacobian of `ceval_mass` with respect to positional argument(s) -1. Takes the same arguments as `ceval_mass` but returns the jacobian of the output with respect to the arguments at positions -1.

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

Original documentation:

Jacobian of `ceval_neohook` with respect to positional argument(s) -1. Takes the same arguments as `ceval_neohook` but returns the jacobian of the output with respect to the arguments at positions -1.

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

Original documentation:

Jacobian of `ceval_ogden` with respect to positional argument(s) -1. Takes the same arguments as `ceval_ogden` but returns the jacobian of the output with respect to the arguments at positions -1.

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) 0. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions 0.

Vectorized version of `ceval_mass`. Takes similar arguments as `ceval_mass` but with additional array axes over which `ceval_mass` is mapped.

Vectorized version of `ceval_elasticity_l`. Takes similar arguments as `ceval_elasticity_l` but with additional array axes over which `ceval_elasticity_l` is mapped.
Original documentation:

Jacobian of `ceval_elasticity_l` with respect to positional argument(s) 1. Takes the same arguments as `ceval_elasticity_l` but returns the jacobian of the output with respect to the arguments at positions 1.

```python
sfepy.terms/terms_jax.eval_mu_neohook(mu, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

Original documentation:

Jacobian of `ceval_neohook` with respect to positional argument(s) 0. Takes the same arguments as `ceval_neohook` but returns the jacobian of the output with respect to the arguments at positions 0.

```python
sfepy.terms/terms_jax.eval_mu_ogden(mu, alpha, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

Original documentation:

Jacobian of `ceval_ogden` with respect to positional argument(s) 0. Takes the same arguments as `ceval_ogden` but returns the jacobian of the output with respect to the arguments at positions 0.

```python
sfepy.terms/terms_jax.eval_neohook(mu, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_neohook`. Takes similar arguments as `ceval_neohook` but with additional array axes over which `ceval_neohook` is mapped.

```python
sfepy.terms/terms_jax.eval_ogden(mu, alpha, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_ogden`. Takes similar arguments as `ceval_ogden` but with additional array axes over which `ceval_ogden` is mapped.

```python
sfepy.terms/terms_jax.eval_poisson_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) 1. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions 1.

```python
sfepy.terms/terms_jax.eval_young_elasticity_yp(young, poisson, plane, vbfg, ubfg, det, cu)
```

Vectorized version of `ceval_elasticity_yp`. Takes similar arguments as `ceval_elasticity_yp` but with additional array axes over which `ceval_elasticity_yp` is mapped.

Original documentation:

Jacobian of `ceval_elasticity_yp` with respect to positional argument(s) 0. Takes the same arguments as `ceval_elasticity_yp` but returns the jacobian of the output with respect to the arguments at positions 0.

```python
sfepy.terms/terms_jax.get_neohook_strain_energy(mu, C)
```

```python
sfepy.terms/terms_jax.get_neohook_strain_energy_f(mu, F)
```

```python
sfepy.terms/terms_jax.get_neohook_stress_1pk(mu, gu)
```

```python
sfepy.terms/terms_jax.get_neohook_stress_2pk(mu, gu)
```

```python
sfepy.terms/terms_jax.get_ogden_strain_energy_f(mu, alpha, F)
```

```python
sfepy.terms/terms_jax.get_ogden_stress_1pk(mu, alpha, gu)
```
sfepy.terms.terms_jax.get_strain(gu)
sfepy.terms.terms_jax.get_stress(lam, mu, gu)

sfepy.terms.terms_mass module

class sfepy.terms.terms_mass.MassTerm(*args, **kwargs)

Mass term with lumping and RMM support [1].

The lumping parameter can be ‘row_sum’, ‘hrz’ or ‘none’ (default). It applies for $\beta > 0$:
- $\beta = 0$ corresponds to the consistent mass matrix $M^C$;
- $0 < \beta < 1$ corresponds to the averaged mass matrix $M^A$.
- $\beta = 1$ corresponds to the lumped mass matrix $M^L$;

term_mode can be None (default), ‘DPM’ (diagonal projection matrix $A$), or ‘RMM’ (reciprocal mass matrix $C$).

Definition

\[
M^C = \int_D \rho \mathbf{u} \cdot \mathbf{u} \\
M^L = \text{lumping}(M^C) \\
M^A = (1 - \beta)M^C + \beta M^L \\
A = \sum_e A_e \\
C = \sum_e A_e^T (M^A_e)^{-1} A_e
\]

Call signature

```
de_mass (material_rho, material_lumping, material_beta, virtual, state)
```

Arguments

- material: $\rho$
- material: lumping
- material: $\beta$
- virtual/parameter_1: $\mathbf{u}$
- state/parameter_2: $\mathbf{u}$

```
arg_shapes = {'material_beta': ': 1', 'material_lumping': ': str', 'material_rho': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material_rho', 'material_lumping', 'material_beta', 'virtual', 'state')

get_function(rho, lumping, beta, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

modes = ('weak', 'eval')

name = 'de_mass'
**sfepy.terms.terms_membrane module**

**class** sfepy.terms.terms_membrane.TLMembraneTerm(*args, **kwargs)**

Mooney-Rivlin membrane with plain stress assumption.

The membrane has a uniform initial thickness $h_0$ and obeys a hyperelastic material law with strain energy by Mooney-Rivlin: $\Psi = a_1(I_1 - 3) + a_2(I_2 - 3)$.

**Call signature**

```
dw_tl_membrane (material_a1, material_a2, material_h0, virtual, state)
```

**Arguments**

- **material_a1** : $a_1$
- **material_a2** : $a_2$
- **material_h0** : $h_0$
- **virtual** : $\nu$
- **state** : $\nu$

**arg_shapes** = {'material_a1': '1, 1', 'material_a2': '1, 1', 'material_h0': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}

**arg_types** = ('material_a1', 'material_a2', 'material_h0', 'virtual', 'state')

**static eval_function**(out, a1, a2, h0, mtx_c, c33, mtx_b, mtx_t, geo, term_mode, fmode)

**static function**(out, fun, *args)

**Notes**

`fun` is either `weak_function` or `eval_function` according to evaluation mode.

**geometries** = ['3_4', '3_8']

**get_eval_shape**(a1, a2, h0, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

**get_fargs**(a1, a2, h0, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'dw_tl_membrane '

**static weak_function**(out, a1, a2, h0, mtx_c, c33, mtx_b, mtx_t, bfg, geo, fmode)

sfepy.terms.terms_membrane.eval_membrane_mooney_rivlin(a1, a2, mtx_c, c33, mode)

Evaluate stress or tangent stiffness of the Mooney-Rivlin membrane.

sfepy.terms.terms_multilinear module

class sfepy.terms.terms_multilinear.ECauchyStressTerm(*args, **kwargs)
    Evaluate Cauchy stress tensor.

    It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

    Definition

        \[ \int_\Omega D_{ijkl} e_{kl}(w) \]

    Call signature

    de_cauchy_stress (material, parameter)

Arguments

- material : \( D_{ijkl} \)
- parameter : \( w \)

arg_shapes = {'material': 'S', 'parameter': 'D'}
arg_types = ('material', 'parameter')

get_function(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'de_cauchy_stress'

class sfepy.terms.terms_multilinear.EConvectTerm(*args, **kwargs)
    Nonlinear convective term.

    Definition

        \[ \int_\Omega ((u \cdot \nabla)u) \cdot v \]

    Call signature

    de_convect (virtual, state)
    (parameter_1, parameter_2)

Arguments

- virtual/parameter_1: \( v \)
- state/parameter_2: \( u \)

arg_shapes = {'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = (['virtual', 'state'], ['parameter_1', 'parameter_2'])

get_function(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('weak', 'eval')
name = 'de_convect'

class sfepy.terms.terms_multilinear.EDiffusionTerm(*args, **kwargs)
   General diffusion term.
   
   Definition
   \[ \int_{\Omega} K_{ij} \nabla_i q \nabla_j p \]

   Call signature
   
   **de_diffusion**  (material, virtual, state)
   (material, parameter_1, parameter_2)

   Arguments
   
   • material: \( K_{ij} \)
   • virtual/parameter_1: \( q \)
   • state/parameter_2: \( p \)

   arg_shapes = {'material': 'D, D', 'parameter_1': 1, 'parameter_2': 1, 'state': 1, 'virtual': (1, 'state')}
   arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))
   get_function(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)

   modes = ('weak', 'eval')
   name = 'de_diffusion'

class sfepy.terms.terms_multilinear.EDivGradTerm(*args, **kwargs)
   Vector field diffusion term.
   
   Definition
   \[ \int_{\Omega} \nabla v : \nabla u, \int_{\Omega} \nu \nabla v : \nabla u \]

   Call signature
   
   **de_div_grad**  (opt_material, virtual, state)
   (opt_material, parameter_1, parameter_2)

   Arguments
   
   • material: \( \nu \) (viscosity, optional)
   • virtual/parameter_1: \( v \)
   • state/parameter_2: \( u \)
arg_shapes = [{'opt_material': '1, 1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter_1': 'D', 'parameter_2': 'D'}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))

def get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs):
    modes = ('weak', 'eval')
    name = 'de_div_grad'

class sfepy.terms.terms_multilinear.EDivTerm(*args, **kwargs):
    Weighted divergence term.

    Definition

    \[ \int_\Omega \nabla \cdot \psi, \int_\Omega c \nabla \cdot \psi \]

    Call signature

    de_div (opt_material, virtual)
           (opt_material, parameter)

    Arguments

    - material: c (optional)
    - virtual/parameter: \psi

arg_shapes = [{'opt_material': '1, 1', 'virtual': ('D', None), 'parameter': 'D'}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual'), ('opt_material', 'parameter'))

def get_function(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs):
    modes = ('weak', 'eval')
    name = 'de_div'

class sfepy.terms.terms_multilinear.EDotTerm(*args, **kwargs):
    Volume and surface \( L^2(\Omega) \) weighted dot product for both scalar and vector fields. Can be evaluated. Can use derivatives.

    Definition

    \[ \int_D q \cdot p, \int_D \psi \cdot u \\
    \int_D c q \cdot p, \int_D c \psi \cdot u \\
    \int_D \psi \cdot (c u) \]

    Call signature
Arguments

- material: \( c \) or \( c \) (optional)
- virtual/parameter_1: \( q \) or \( q \)
- state/parameter_2: \( p \) or \( p \)

\[
\begin{align*}
\text{arg_shapes} &= [\{\text{opt_material: '1', 'virtual': (1, 'state'), 'state': 1, 'parameter_1': 1, 'parameter_2': 1}, \{\text{opt_material: '1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter_1': 'D', 'parameter_2': 'D'}, \{\text{opt_material: 'None'}\}]
\end{align*}
\]

\[
\begin{align*}
\text{arg_types} &= (\{\text{opt_material', 'virtual', 'state'}, ('opt_material', 'parameter_1', 'parameter_2')\})
\end{align*}
\]

\[
\begin{align*}
\text{get_function}(\text{mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs})
\end{align*}
\]

\[
\begin{align*}
\text{integration} &= ('cell', 'facet')
\end{align*}
\]

\[
\begin{align*}
\text{modes} &= ('\text{weak}', '\text{eval}')
\end{align*}
\]

\[
\begin{align*}
\text{name} &= '\text{de_dot}'
\end{align*}
\]

class sfepy.terms.terms_multilinear.EGradTerm(*args, **kwargs)

Weighted gradient term.

**Definition**

\[
\begin{align*}
\int_\Omega \nabla q, \int_\Omega c \nabla q, \int_\Omega c \cdot \nabla q, \int_\Omega c \cdot \nabla q
\end{align*}
\]

**Call signature**

\[
\begin{align*}
\text{de_grad} \ (\text{opt_material, parameter})
\end{align*}
\]

Arguments

- material: \( c \) (optional)
- virtual/parameter: \( \overline{v} \)

\[
\begin{align*}
\text{arg_shapes} &= [\{\text{opt_material: '1', 'parameter': 'N'}, \{\text{opt_material: 'N', 1}, \{\text{opt_material: 'N', 'N'}, \{\text{opt_material: 'None'}\}]
\end{align*}
\]

\[
\begin{align*}
\text{arg_types} &= (\{'\text{opt_material'}', '\text{parameter}'\})
\end{align*}
\]

\[
\begin{align*}
\text{get_function}(\text{mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs})
\end{align*}
\]

\[
\begin{align*}
\text{name} &= '\text{de_grad}'
\end{align*}
\]
class sfepy.terms.terms_multilinear.EIntegrateOperatorTerm(*args, **kwargs)

Volume and surface integral of a test function weighted by a scalar function \( c \).

**Definition**

\[
\int_D q \text{ or } \int_D cq
\]

**Call signature**

```
de_integrate (opt_material, virtual)
```

**Arguments**

- material : \( c \) (optional)
- virtual : \( q \)

```
arg_shapes = [{'opt_material': '1, 1', 'virtual': (1, None)}, {'opt_material': None}]
arg_types = ('opt_material', 'virtual')
```

```
get_function(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = ('cell', 'facet')
name = 'de_integrate'
```

class sfepy.terms.terms_multilinear.ELaplaceTerm(*args, **kwargs)

Laplace term with \( c \) coefficient. Can be evaluated. Can use derivatives.

**Definition**

\[
\int_\Omega \nabla q \cdot \nabla p, \int_\Omega c \nabla q \cdot \nabla p
\]

**Call signature**

```
de_laplace (opt_material, virtual, state)
```

```
de_laplace (opt_material, parameter_1, parameter_2)
```

**Arguments**

- material: \( c \)
- virtual/parameter_1: \( q \)
- state/parameter_2: \( p \)

```
arg_shapes = [{'opt_material': '1, 1', 'virtual': (1, 'state'), 'state': 1, 'parameter_1': 1, 'parameter_2': 1}, {'opt_material': None}]
arg_types = ('opt_material', 'virtual', 'state'), ('opt_material', 'parameter_1', 'parameter_2'))
```

```
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```
modes = ('weak', 'eval')
name = 'de_laplace'
class sfepy.terms.terms_multilinear.ELinearConvectTerm(*args, **kwargs)
Linearized convective term.

Definition
\[ \int_{\Omega} ((w \cdot \nabla)u) \cdot v \]

Call signature

```
de_lin_convect (virtual, parameter, state)
(parameter_1, parameter_2, parameter_3)
```

Arguments
- virtual/parameter_1: \( v \)
- parameter/parameter_2: \( w \)
- state/parameter_3: \( u \)

arg_shapes = {'parameter': 'D', 'parameter_1': 'D', 'parameter_2': 'D',
              'parameter_3': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('virtual', 'parameter', 'state'), ('parameter_1', 'parameter_2',
             'parameter_3'))

get_function(virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('weak', 'eval')
name = 'de_lin_convect'
class sfepy.terms.terms_multilinear.ELinearElasticTerm(*args, **kwargs)
General linear elasticity term, with \( D_{ijkl} \) given in the usual matrix form exploiting symmetry: in 3D it is \( 6 \times 6 \)
with the indices ordered as \([11, 22, 33, 12, 13, 23] \), in 2D it is \( 3 \times 3 \) with the indices ordered as \([11, 22, 12] \).

Definition
\[ \int_{\Omega} D_{ijkl} e_{ij}(v) e_{kl}(u) \]

Call signature

```
de_lin_elastic (material, virtual, state)
(material, parameter_1, parameter_2)
```

Arguments
- material: \( D_{ijkl} \)
- virtual/parameter_1: \( v \)
- state/parameter_2: \( u \)
arg_shapes = {'material': 'S, S', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))

get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_lin_elastic'

class sfepy.terms.terms_multilinear.ELinearTractionTerm(*args, **kwargs)

Linear traction term. The material parameter can have one of the following shapes:

- 1 or (1, 1) - a given scalar pressure
- (D, 1) - a traction vector
- (S, 1) or (D, D) - a given stress in symmetric or non-symmetric tensor storage (in symmetric storage indices are order as follows: 2D: [11, 22, 12], 3D: [11, 22, 33, 12, 13, 23])

Definition

\[
\int_T v \cdot n, \int_T c v \cdot n, \\
\int_T v \cdot (\sigma n), \int_T v \cdot f
\]

Call signature

```
de_surface_ltr (opt_material, virtual)
(opt_material, parameter)
```

Arguments

- material: c, f, σ or \( \sigma \)
- virtual/parameter: v

arg_shapes = [{'opt_material': 'S, 1', 'virtual': ('D', None), 'parameter': 'D'}, {'opt_material': None}, {'opt_material': '1, 1'}, {'opt_material': 'D, 1'}, {'opt_material': 'D, D'}]

arg_types = (('opt_material', 'virtual'), ('opt_material', 'parameter'))

get_function(traction, vvar, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'de_surface_ltr'
class sfepy.terms.terms_multilinear.ENonPenetration PenaltyTerm(*args, **kwargs)

Non-penetration condition in the weak sense using a penalty.

Definition

\[ \int_\Gamma c(n \cdot v)(n \cdot u) \]

Call signature

```
def non_penetration_p(material, virtual, state)
```

Arguments

- *material*: \(c\)
- *virtual*: \(v\)
- *state*: \(u\)

`arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}`

`arg_types = ('material', 'virtual', 'state')`

get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'de_non_penetration_p'

class sfepy.terms.terms_multilinear.ENonSymElasticTerm(*args, **kwargs)

Elasticity term with non-symmetric gradient. The indices of matrix \(D_{ijkl}\) are ordered as \([11, 12, 13, 21, 22, 23, 31, 32, 33]\) in 3D and as \([11, 12, 21, 22]\) in 2D.

Definition

\[ \int_\Omega D \nabla v : \nabla u \]

Call signature

```
def nonsym_elastic(mat, virtual, state)
```

Arguments

- *material*: \(D\)
- *virtual*: \(v\)
- *state*: \(u\)

`arg_shapes = {'material': 'D2, D2', 'parameter_1': 'D', 'parameter_2': 'D', 'state': 'D', 'virtual': ('D', 'state')}}`

`arg_types = (('material', 'virtual', 'state'), ('material', 'parameter_1', 'parameter_2'))`
get_function(mat, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de nonsym elastic'

class sfepy.terms.terms_multilinear.EScalarDotMGradScalarTerm(*args, **kwargs)

Volume dot product of a scalar gradient dotted with a material vector with a scalar.

Definition

\[
\int_{\Omega} q y \cdot \nabla p, \int_{\Omega} p y \cdot \nabla q
\]

Call signature

<table>
<thead>
<tr>
<th>de_s_dot_mgrad_s</th>
<th>(material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material, state, virtual)</td>
</tr>
<tr>
<td></td>
<td>(material, parameter_1, parameter_2)</td>
</tr>
</tbody>
</table>

Arguments 1

• material : y
• virtual : q
• state : p

Arguments 2

• material : y
• state : p
• virtual : q

arg_shapes = [{'material': 'D, 1', 'virtual/grad_state': (1, None), 'state/grad_state': 1, 'virtual/grad_virtual': (1, None), 'state/grad_virtual': 1, 'parameter_1': 1, 'parameter_2': 1}]

arg_types = (['material', 'virtual', 'state'], ['material', 'state', 'virtual'], ['material', 'parameter_1', 'parameter_2'])

get_function(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad_state', 'grad_virtual', 'eval')

name = 'de_s_dot_mgrad_s'

class sfepy.terms.terms_multilinear.EStokesTerm(*args, **kwargs)

Stokes problem coupling term. Corresponds to weak forms of gradient and divergence terms.

Definition

\[
\int_{\Omega} p \nabla \cdot v, \int_{\Omega} q \nabla \cdot u
\]

Call signature
### de_stokes

<table>
<thead>
<tr>
<th>Arguments 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>• material: $c$ (optional)</td>
</tr>
<tr>
<td>• virtual/parameter_v: $v$</td>
</tr>
<tr>
<td>• state/parameter_s: $p$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• material : $c$ (optional)</td>
</tr>
<tr>
<td>• state : $u$</td>
</tr>
<tr>
<td>• virtual : $q$</td>
</tr>
</tbody>
</table>

```
arg_shapes = ["opt_material": '1, 1', 'virtual/grad': ('D', None), 'state/grad': 1, 'virtual/div': (1, None), 'state/div': 'D', 'parameter_v': 'D', 'parameter_s': 1}, {'opt_material': None}]
```

```
arg_types = (('opt_material', 'virtual', 'state'), ('opt_material', 'state', 'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))
```

```
get_function(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
modes = ('grad', 'div', 'eval')
```

```
name = 'de_stokes'
```

```python
class sfepy.terms.terms_multilinear.ETermBase(*args, **kwargs)
```

Reserved letters:

- c .. cells
- q .. quadrature points
- d-h .. DOFs axes
- r-z .. auxiliary axes

Layout specification letters:

- c .. cells
- q .. quadrature points
- v .. variable component
- m .. matrix form
- g .. gradient component
- d .. DOF (basis, node)
- 0 .. all material axes

```
built_expression(texpr, *eargs, diff_var=None)
```
can_backend = {'dask_single': <module 'dask.array' from '/home/eldaran/.local/lib/python3.8/site-packages/dask/array/__init__.py'>,
'dask_threads': <module 'dask.array' from '/home/eldaran/.local/lib/python3.8/site-packages/dask/array/__init__.py'>,
'jax': <module 'jax.numpy' from '/home/eldaran/.local/lib/python3.8/site-packages/jax/numpy/__init__.py'>,
'jax_vmap': <module 'jax.numpy' from '/home/eldaran/.local/lib/python3.8/site-packages/jax/numpy/__init__.py'>,
'numpy': <module 'numpy' from '/home/eldaran/.local/lib/python3.8/site-packages/numpy/__init__.py'>,
'numpy_loop': <module 'numpy' from '/home/eldaran/.local/lib/python3.8/site-packages/numpy/__init__.py'>,
'numpy_qloop': <module 'numpy' from '/home/eldaran/.local/lib/python3.8/site-packages/numpy/__init__.py'>,
'opt_einsum': <module 'opt_einsum' from '/home/eldaran/.local/lib/python3.8/site-packages/opt_einsum/__init__.py'>,
'opt_einsum_dask_single': <module 'dask.array' from '/home/eldaran/.local/lib/python3.8/site-packages/dask/array/__init__.py'>,
'opt_einsum_dask_threads': <module 'dask.array' from '/home/eldaran/.local/lib/python3.8/site-packages/dask/array/__init__.py'>,
'opt_einsum_loop': <module 'opt_einsum' from '/home/eldaran/.local/lib/python3.8/site-packages/opt_einsum/__init__.py'>,
'opt_einsum_qloop': <module 'opt_einsum' from '/home/eldaran/.local/lib/python3.8/site-packages/opt_einsum/__init__.py'>}

clear_cache()

eval_complex(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)
eval_real(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

static function_silent(out, eval_einsum, *args)
static function_timer(out, eval_einsum, *args)

get_eval_shape(*args, **kwargs)
get_fargs(*args, **kwargs)
get_normals(arg)
get_operands(diff_var)
get_paths(expressions, operands)

layout_letters = 'cqgd0'

make_function(texpr, *args, diff_var=None)

set_backend(backend='numpy', optimize=True, layout=None, **kwargs)

set_verbosity(verbosity=None)

verbosity = 0

class sfepy.terms.terms_multilinear.ExpressionArg(**kwargs)
static from_term_arg(arg, term)
get_bf(expr_cache)
get_dofs(cache, expr_cache, oname)

class sfepy.terms.terms_multilinear.ExpressionBuilder(n_add, cache)
    add_arg_dofs(iin, ein, name, n_components, iia=None)
    add_bf(iin, ein, name, cell_dependent=False)
    add_bfg(iin, ein, name)
    add_constant(name, cname)
    add_eye(iic, ein, name, iia=None)
    add_material_arg(arg, ii, ein)
    add_psg(iic, ein, name, iia=None)
    add_pvg(iic, ein, name, iia=None)
    add_state_arg(arg, ii, ein, modifier, diff_var)
    add_virtual_arg(arg, ii, ein, modifier)
    apply_layout(layout, operands, defaults=None, verbosity=0)
    build(texpr, *args, diff_var=None)
    get_expressions(subscripts=None)
static join_subscripts(subscripts, out_subscripts)

letters = 'defgh'
make_eye(size)
make_psg(dim)
make_pvg(dim)
print_shapes(subscripts, operands)
transform(subscripts, operands, transformation='loop', **kwargs)

class sfepy.terms.terms_multilinear.SurfaceFluxOperatorTerm(*args, **kwargs)
Surface flux operator term.

Definition
\[
\int_{\Gamma} q n \cdot K \cdot \nabla p, \int_{\Gamma} p n \cdot K \cdot \nabla q
\]

Call signature

<table>
<thead>
<tr>
<th>de_surface_flux</th>
<th>(material, virtual, state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(material, state, virtual)</td>
</tr>
<tr>
<td></td>
<td>(material, parameter_1, parameter_2)</td>
</tr>
</tbody>
</table>
Arguments 1

• material : $K$
• virtual : $q$
• state : $p$

Arguments 2

• material : $K$
• state : $p$
• virtual : $q$

arg_shapes = [{'material': 'D, D', 'virtual/grad_state': (1, None), 'state/grad_state': 1, 'parameter_1': 1, 'parameter_2': 1}]

arg_types = ('material', 'virtual'), ('material', 'state', 'virtual'), ('material', 'parameter_1', 'parameter_2'))

get_function(mat, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet_extra'

modes = ('grad_state', 'grad_virtual', 'eval')

name = 'de_surface_flux'

sfepy.terms.terms_multilinear.append_all(seqs, item, ii=None)

sfepy.terms.terms_multilinear.collect_modifiers(modifiers)

sfepy.terms.terms_multilinear.find_free_indices(indices)

sfepy.terms.terms_multilinear.get_einsum_ops(eargs, ebuilder, expr_cache)

sfepy.terms.terms_multilinear.get_loop_indices(subs, loop_index)

sfepy.terms.terms_multilinear.get_output_shape(out_subscripts, subscripts, operands)

sfepy.terms.terms_multilinear.get_sizes(indices, operands)

sfepy.terms.terms_multilinear.get_slice_ops(subs, ops, loop_index)

sfepy.terms.terms_multilinear.parse_term_expression(texpr)

sfepy.terms.terms_multilinear.sym2nonsym(sym_obj, axes=[3])

**sfepy.terms.terms_navier_stokes module**

**class sfepy.terms.terms_navier_stokes.ConvectTerm**(name, arg_str, integral, region, **kwargs)

Nonlinear convective term.

**Definition**

\[\int_{\Omega} ((u \cdot \nabla) u) \cdot \nu\]
Call signature

\[
dw_{\text{convect}} \ (\text{virtual, state})
\]

Arguments

- virtual : \(v\)
- state : \(u\)

arg_shapes = {'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('virtual', 'state')

static function()

get_fargs(virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

ame = 'dw_{\text{convect}}'

class sfepy.terms.terms_navier_stokes.DivGradTerm(name, arg_str, integral, region, **kwargs)

Definition

\[
\int_\Omega \nu \nabla v : \nabla \nabla u, \quad \int_\Omega \nabla v : \nabla \nabla u
\]

Call signature

\[
dw_{\text{div grad}} \ (\text{opt_material, virtual, state})
\]

\[
dw_{\text{div grad}} \ (\text{opt_material, parameter}_1, \text{parameter}_2)
\]

Arguments

- material: \(\nu\) (viscosity, optional)
- virtual: \(v\)
- state/parameter_1: \(u\)
- state/parameter_2: \(u\)

arg_shapes = [{'opt_material': '1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter_1': 'D', 'parameter_2': 'D'}, {'opt_material': None}]

arg_types = (['opt_material', 'virtual', 'state'], ['opt_material', 'parameter_1', 'parameter_2'])

d_div_grad(out, grad1, grad2, mat, vg, fmode)

static function()

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_{\text{div grad}}'
set_arg_types()

class sfepy.terms.terms_navier_stokes.DivOperatorTerm(name, arg_str, integral, region, **kwargs)
Weighted divergence term of a test function.

Definition
\[ \int_{\Omega} \nabla \cdot \mathbf{v} \text{ or } \int_{\Omega} c \nabla \cdot \mathbf{v} \]

Call signature

| dw_div | (opt_material, virtual) |

Arguments

- material : c (optional)
- virtual : \( \mathbf{v} \)

arg_shapes = [{'opt_material': '1, 1', 'virtual': ('D', None)}, {'opt_material': None}]

arg_types = ('opt_material', 'virtual')

static function(out, mat, vg)

get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_div'

class sfepy.terms.terms_navier_stokes.DivTerm(name, arg_str, integral, region, **kwargs)
Evaluate divergence of a vector field.

Supports 'eval', 'el_avg' and 'qp' evaluation modes.

Definition
\[ \int_{\mathcal{D}} \nabla \cdot \mathbf{u} \text{, } \int_{\mathcal{D}} c \nabla \cdot \mathbf{u} \]

Call signature

| ev_div | (opt_material, parameter) |

Arguments

- parameter : \( \mathbf{u} \)

arg_shapes = [{'opt_material': '1, 1', 'parameter': 'D'}, {'opt_material': None}]

arg_types = ('opt_material', 'parameter')

static function(out, mat, div, vg, fmode)

get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = ('cell', 'facet_extra')
name = 'ev_div'

class sfepy.terms.terms_navier_stokes.GradDivStabilizationTerm(name, arg_str, integral, region, **kwargs)
Grad-div stabilization term (γ is a global stabilization parameter).
Definition
\[ \gamma \int_\Omega (\nabla \cdot u) : (\nabla \cdot v) \]
Call signature

\[
\text{dw_st_grad_div} \ (\text{material}, \text{virtual}, \text{state})
\]
Arguments

- material : γ
- virtual : v
- state : u

arg_shapes = {'material': '1, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')

static function()
get_fargs(gamma, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_st_grad_div'

class sfepy.terms.terms_navier_stokes.GradTerm(name, arg_str, integral, region, **kwargs)
Evaluate gradient of a scalar or vector field.
Supports 'eval', 'el_avg' and 'qp' evaluation modes.
Definition
\[ \int_D \nabla p \quad \text{or} \quad \int_D \nabla u \]
\[ \int_D c \nabla p \quad \text{or} \quad \int_D c \nabla u \]
Call signature

\[
\text{ev_grad} \ (\text{opt_material}, \text{parameter})
\]
Arguments

- parameter : p or u

arg_shapes = [{'opt_material': '1, 1', 'parameter': 'N'}, {'opt_material': None}]
arg_types = ('opt_material', 'parameter')
static function(out, mat, grad, vg, fmode)

get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = ('cell', 'facet_extra')

name = 'ev_grad'

class sfepy.terms.terms_navier_stokes.LinearConvect2Term(name, arg_str, integral, region, **kwargs)
Linearized convective term with the convection velocity given as a material parameter.

Definition

\[ \int_{\Omega} ((c \cdot \nabla) u) \cdot v \]

\[ ((c \cdot \nabla) u)|_{qp} \]

Call signature

dw_lin_convect2 (material, virtual, state)

Arguments

• material : c
• virtual : v
• state : u

arg_shapes = {'material': 'D, 1', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('material', 'virtual', 'state')

static function()

get_fargs(material, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_lin_convect2'

class sfepy.terms.terms_navier_stokes.LinearConvectTerm(name, arg_str, integral, region, **kwargs)
Linearized convective term.

Definition

\[ \int_{\Omega} ((w \cdot \nabla) u) \cdot v \]

\[ ((w \cdot \nabla) u)|_{qp} \]

Call signature

dw_lin_convect (virtual, parameter, state)

Arguments

• virtual : v
• parameter : \( w \)
• state : \( u \)

```python
arg_shapes = {'parameter': 'D', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = ('virtual', 'parameter', 'state')
```

```python
static function()
get_fargs(virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_lin_convect'
```

```
class sfepy.terms.terms_navier_stokes.PSPGCStabilizationTerm(name, arg_str, integral, region, **kwargs)
```

PSPG stabilization term, convective part \((\tau\text{ is a local stabilization parameter)}\).

**Definition**

\[
\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K ((b \cdot \nabla)u) \cdot \nabla q
\]

**Call signature**

\[
\text{dw_st_pspg_c} \ (\text{material, virtual, parameter, state})
\]

**Arguments**

- material : \( \tau_K \)
- virtual : \( q \)
- parameter : \( b \)
- state : \( u \)

```python
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'virtual': (1, None)}
arg_types = ('material', 'virtual', 'parameter', 'state')
```

```python
static function()
get_fargs(tau, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_st_pspg_c'
```

```
class sfepy.terms.terms_navier_stokes.PSPGPStabilizationTerm(name, arg_str, integral, region, **kwargs)
```

PSPG stabilization term, pressure part \((\tau\text{ is a local stabilization parameter)}\), alias to Laplace term \( \text{dw_laplace} \).

**Definition**

\[
\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q
\]

**Call signature**

\[
\text{dw_st_pspg_p} \ (\text{material, pressure, parameter, state})
\]

**Arguments**

- material : \( \tau_K \)
- pressure : \( p \)
- parameter : \( b \)
- state : \( u \)

```python
arg_shapes = {'material': '1, 1', 'parameter': 'D', 'state': 'D', 'pressure': (1, None)}
arg_types = ('material', 'pressure', 'parameter', 'state')
```

```python
static function()
get_fargs(tau, pressure, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_st_pspg_p'
```

```
class sfepy.terms.terms_navier_stokes.PSPGPStabilizationTerm(name, arg_str, integral, region, **kwargs)
```

PSPG stabilization term, pressure part \((\tau\text{ is a local stabilization parameter)}\), alias to Laplace term \( \text{dw_laplace} \).

**Definition**

\[
\sum_{K \in \mathcal{I}_h} \int_{T_K} \tau_K \nabla p \cdot \nabla q
\]

**Call signature**
**dw_st_pspg_p** (opt_material, virtual, state)  
(opt_material, parameter_1, parameter_2)

**Arguments**
- material : $\tau_K$
- virtual : $q$
- state : $p$

**name = 'dw_st_pspg_p'**

```python
class sfepy.terms.terms_navier_stokes.SUPGCStabilizationTerm(name, arg_str, integral, region, **kwargs):
```

SUPG stabilization term, convective part ( $\delta$ is a local stabilization parameter).

**Definition**

$$\sum_{K \in I_h} \int_{T_K} \delta_K ((b \cdot \nabla) \mathbf{u}) \cdot ((b \cdot \nabla) \mathbf{v})$$

**Call signature**

**dw_st_supg_c** (material, virtual, parameter, state)

**Arguments**
- material : $\delta_K$
- virtual : $\mathbf{v}$
- parameter : $b$
- state : $\mathbf{u}$

**arg_shapes = { material: 'l, l', parameter: 'D', state: 'D', virtual: ('D', 'state') }**

**arg_types = (material, virtual, parameter, state)**

**static function()**

```python
get_fargs(delta, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

**name = 'dw_st_supg_c'**

```python
class sfepy.terms.terms_navier_stokes.SUPGPStabilizationTerm(name, arg_str, integral, region, **kwargs):
```

SUPG stabilization term, pressure part ( $\delta$ is a local stabilization parameter).

**Definition**

$$\sum_{K \in I_h} \int_{T_K} \delta_K \nabla p \cdot ((b \cdot \nabla) \mathbf{v})$$

**Call signature**
**Arguments**

- **material**: $\delta_K$
- **virtual**: $v$
- **parameter**: $b$
- **state**: $p$

```python
dw_st_supg_p = (material, virtual, parameter, state)
```

```python
arg_shapes = {'material': '1', 'parameter': 'D', 'state': '1', 'virtual': ('D', None)}
arg_types = ('material', 'virtual', 'parameter', 'state')
static function()
get_fargs(delta, virtual, parameter, state, mode=None, term_mode=None, diff_var=None, **kwargs)
name = 'dw_st_supg_p'
```

**Class** `sfepy.terms.terms_navier_stokes.StokesTerm`  
Corresponds to weak forms of gradient and divergence terms. Can be evaluated.

**Definition**

\[
\int_\Omega p \nabla \cdot v, \int_\Omega q \nabla \cdot u
\]

or

\[
\int_\Omega c p \nabla \cdot v, \int_\Omega c q \nabla \cdot u
\]

**Call signature**

```python
dw_stokes (opt_material, virtual, state)
(opt_material, state, virtual)
(opt_material, parameter_v, parameter_s)
```

**Arguments 1**

- **material**: $c$ (optional)
- **virtual/parameter_v**: $v$
- **state/parameter_s**: $p$

**Arguments 2**

- **material**: $c$ (optional)
- **state**: $u$
- **virtual**: $q$

```python
arg_shapes = [{'opt_material': '1', 'virtual/grad': ('D', None), 'state/grad': '1', 'virtual/div': (1, None), 'state/div': 'D', 'parameter_v': 'D', 'parameter_s': '1}, {'opt_material': None}]
```
arg_types = (('opt_material', 'virtual', 'state'),
('opt_material', 'state',
'virtual'), ('opt_material', 'parameter_v', 'parameter_s'))

d_eval(static, out, coef, vec_qp, div, vvg)

def get_eval_shape(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs):

def get_fargs(coef, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs):

def modes = ('grad', 'div', 'eval')

name = 'dw_stokes'

class sfepy.terms.terms_navier_stokes.StokesWaveDivTerm(name, arg_str, integral, region, **kwargs):

Stokes dispersion term with the wave vector $\kappa$ and the divergence operator.

**Definition**

\[
\int_{\Omega} (\kappa \cdot \mathbf{v}) (\nabla \cdot \mathbf{u}) , \int_{\Omega} (\kappa \cdot \mathbf{u}) (\nabla \cdot \mathbf{v})
\]

**Call signature**

\[
\text{dw_stokes_wave_div} \quad \text{(material, virtual, state)}
\]

\[
\text{dw_stokes_wave_div} \quad \text{(material, state, virtual)}
\]

**Arguments 1**

- material : $\kappa$
- virtual : $\mathbf{v}$
- state : $\mathbf{u}$

**Arguments 2**

- material : $\kappa$
- state : $\mathbf{u}$
- virtual : $\mathbf{v}$

arg_shapes = {'material': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = (('material', 'virtual', 'state'),
('material', 'state',
'virtual'))

static function(out, out_qp, geo, fmode)

geometries = ['2_3', '2_4', '3_4', '3_8']

get_fargs(kappa, kvar, dvar, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('kd', 'dk')

name = 'dw_stokes_wave_div'
class sfepy.terms.terms_navier_stokes.StokesWaveTerm(name, arg_str, integral, region, **kwargs)

Stokes dispersion term with the wave vector $\kappa$.

Definition
\[
\int_{\Omega} (\kappa \cdot \mathbf{v})(\kappa \cdot \mathbf{u})
\]

Call signature

\[
\texttt{dw_stokes_wave} \quad \text{(material, virtual, state)}
\]

Arguments
- material : $\kappa$
- virtual : $\mathbf{v}$
- state : $\mathbf{u}$

arg_shapes = {'material': 'D', 'state': 'D', 'virtual': ('D', 'state')}

arg_types = ('material', 'virtual', 'state')

static function(out, out_qp, geo, fmode)

generations = ['2_3', '2_4', '3_4', '3_8']

get_fargs(kappa, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_stokes_wave'

sfepy.terms.terms_piezo module

class sfepy.terms.terms_piezo.PiezoCouplingTerm(name, arg_str, integral, region, **kwargs)

Piezoelectric coupling term. Can be evaluated.

Definition
\[
\int_{\Omega} g_{kij} e_{ij}(\mathbf{v})\nabla k\mathbf{p}
\]
\[
\int_{\Omega} g_{kij} e_{ij}(\mathbf{u})\nabla k\mathbf{q}
\]

Call signature

\[
\texttt{dw_piezo_coupling} \quad \text{(material, virtual, state)}
\]
\[
\text{(material, state, virtual)}
\]
\[
\text{(material, parameter_v, parameter_s)}
\]

Arguments
- material: $g_{kij}$
- virtual/parameter_v: $\mathbf{v}$
- state/parameter_s: $\mathbf{p}$
Arguments

- material : $g_{kij}$
- state : $u$
- virtual : $q$

```python
arg_shapes = { 'material': 'D, S', 'parameter_s': 1, 'parameter_v': 'D', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None) }
arg_types = (('material', 'virtual', 'state'), ('material', 'state', 'virtual'), ('material', 'parameter_v', 'parameter_s'))
```

```python
def get_eval_shape(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs):
    pass

def get_fargs(mat, vvar, svar, mode=None, term_mode=None, diff_var=None, **kwargs):
    pass
```

```python
def modes():
    return ('grad', 'div', 'eval')
```

```python
def name:
    return 'ev_piezo_strain'
```

class sfepy.terms.terms_piezo.PiezoStrainTerm(name, arg_str, integral, region, **kwargs):
    Evaluate piezoelectric strain tensor.

    It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as $[11, 22, 33, 12, 13, 23]$, in 2D it has 3 components with the indices ordered as $[11, 22, 12]$.

    Supports 'eval', 'el_avg' and 'qp' evaluation modes.

    Definition

    $\int_{\Omega} g_{kij} \epsilon_{ij}(u)$

    Call signature

    ```python
ev_piezo_strain  (material, parameter)
```

Arguments

- material : $g_{kij}$
- parameter : $u$

```python
arg_shapes = { 'material': 'D, S', 'parameter': 'D' }
```

```python
def get_eval_shape(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs):
    pass

def get_fargs(mat, parameter, mode=None, term_mode=None, diff_var=None, **kwargs):
    pass
```

```python
def name:
    return 'ev_piezo_strain'
```
class sfepy.terms.terms_piezo.PiezostressTerm(name, arg_str, integral, region, **kwargs)

Evaluate piezoelectric stress tensor.

It is given in the usual vector form exploiting symmetry: in 3D it has 6 components with the indices ordered as [11, 22, 33, 12, 13, 23], in 2D it has 3 components with the indices ordered as [11, 22, 12].

Supports ‘eval’, ‘el_avg’ and ‘qp’ evaluation modes.

Definition

\[
\int_{\Omega} g_{kij} \nabla k p
\]

Call signature

ev_piezo_stress (material, parameter)

Arguments

- material : \(g_{kij}\)
- parameter : \(p\)

arg_shapes = {'material': 'E, S', 'parameter': '1'}

arg_types = ('material', 'parameter')

class sfepy.terms.terms_piezo.SDPiezoCouplingTerm(*args, **kwargs)

Sensitivity (shape derivative) of the piezoelectric coupling term.

Definition

\[
\int_{\Omega} \hat{g}_{kij} E_{ij}(u) \nabla k p
\]

\[
\hat{g}_{kij} = g_{kij} (\nabla \cdot \Psi) - g_{kli} \frac{\partial \Psi_j}{\partial x_l} - g_{lki} \frac{\partial \Psi_j}{\partial x_l}
\]

Call signature

ev_sd_piezo_coupling (material, parameter_u, parameter_p, parameter_mv)

Arguments

- material : \(g_{kij}\)
- parameter_u : \(u\)
- parameter_p : \(p\)
- parameter_mv : \(\Psi\)
arg_shapes = {'material': 'D, S', 'parameter_mv': 'D', 'parameter_p': 1, 'parameter_u': 'D'}
arg_types = ('material', 'parameter_u', 'parameter_p', 'parameter_mv')
geometries = ['2_3', '2_4', '3_4', '3_8']

def get_function(mat, par_u, par_p, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
    name = 'ev_sd_piezo_coupling'

sfepy.terms.terms_point module

class ConcentratedPointLoadTerm
    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
    \[ f_i = \bar{f}_i \quad \forall \text{FE node } i \text{ in a region} \]

    Call signature
    ```
    dw_point_load (material, virtual)
    ```

    Arguments
    - material : \( \bar{f}_i \)
    - virtual : \( v \).

    arg_shapes = {'material': '.', 'virtual': ('N', None)}
    arg_types = ('material', 'virtual')

class LinearPointSpringTerm
    Linear springs constraining movement of FE nodes in a region; to use as a relaxed Dirichlet boundary conditions.

    Definition
    \[ f_i = -k u_i \quad \forall \text{FE node } i \text{ in a region} \]

    Call signature
Arguments

- material: \( k \)
- virtual: \( v \)
- state: \( u \)

\[
\text{arg_shapes} = \{'\text{material}': ': 1', '\text{state}': 'D', '\text{virtual}': ('D', 'state')\}
\]

\[
\text{arg_types} = ('\text{material}', '\text{virtual}', '\text{state}')
\]

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'

---

dsfePy.terms.terms_sensitivity module

class sfepy.terms.terms_sensitivity.ESDDiffusionTerm(*args, **kwargs)

Diffusion sensitivity analysis term.

Definition

\[
\int_\Omega \hat{K}_{ij} \nabla_i q \nabla_j p
\]

\[
\hat{K}_{ij} = K_{ij} \left( \delta_{ik} \delta_{jl} \nabla \cdot \mathbf{Y} - \delta_{ik} \frac{\partial V_j}{\partial x_l} - \delta_{jl} \frac{\partial V_i}{\partial x_k} \right)
\]

Call signature

\[
\text{de_sd_diffusion} \quad (\text{material, virtual, state, parameter_mv})
\]

\[
(\text{material, parameter_1, parameter_2, parameter_mv})
\]

Arguments

- material: \( K_{ij} \)
- virtual/parameter_1: \( q \)
- state/parameter_2: \( p \)
- parameter_mv: \( \mathbf{Y} \)

\[
\text{arg_shapes} = \{'\text{material}': 'D, D', '\text{parameter_1}': 1, '\text{parameter_2}': 1, '\text{parameter_mv}': 'D', '\text{state}': 1, '\text{virtual}': (1, 'state')\}
\]

\[
\text{arg_types} = ('\text{material}', '\text{virtual}', '\text{state}', '\text{parameter_mv}'), ('\text{material}', '\text{parameter_1}', '\text{parameter_2}', '\text{parameter_mv}')
\]
get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_sd_diffusion'

class sfepy.terms.terms_sensitivity.ESDDivGradTerm(*args, **kwargs)

Sensitivity (shape derivative) of diffusion term de_div_grad.

Definition

$$
\int_\Omega \hat{I} \nabla \varepsilon \cdot \nabla u \cdot \nabla \varepsilon - \delta_{ij} \delta_{kl} \frac{\partial \varepsilon}{\partial x_i} - \delta_{is} \delta_{jk} \frac{\partial \varepsilon}{\partial x_s}
$$

Call signature

de_sd_div_grad (opt_material, virtual, state, parameter_mv)
            (opt_material, parameter_1, parameter_2, parameter_mv)

Arguments

- material: \( \nu \) (viscosity, optional)
- virtual/parameter_1: \( \varepsilon \)
- state/parameter_2: \( u \)
- parameter_mv: \( \varepsilon \)

arg_shapes = [{'opt_material': '1, 1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter_1': 'D', 'parameter_2': 'D', 'parameter_mv': 'D'}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual', 'state', 'parameter_mv'), ('opt_material', 'parameter_1', 'parameter_2', 'parameter_mv'))

get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_sd_div_grad'

class sfepy.terms.terms_sensitivity.ESDDotTerm(*args, **kwargs)

Sensitivity (shape derivative) of dot product of scalars or vectors.

Definition

$$
\int_\Omega q p (\nabla \cdot \varepsilon \cdot \varepsilon - \delta_{ij} \delta_{kl} \frac{\partial \varepsilon}{\partial x_i} - \delta_{is} \delta_{jk} \frac{\partial \varepsilon}{\partial x_s})
$$

Call signature
### de_sd_dot

<table>
<thead>
<tr>
<th>(opt_material, virtual, state, parameter_mv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(opt_material, parameter_1, parameter_2, parameter_mv)</td>
</tr>
</tbody>
</table>

**Arguments**

- material: $c$ or $M$ (optional)
- virtual/parameter$_{1}$: $q$ or $v$
- state/parameter$_{2}$: $p$ or $u$
- parameter$_{mv}$: $\mathcal{V}$

**arg_shapes** = [{"opt_material": '1, 1', 'virtual': '1', 'state': 1, 'parameter$_{1}$': 1, 'parameter$_{mv}$': 'D'}, {'opt_material': '1, 1', 'virtual': ('D', 'state'), 'state': 'D', 'parameter$_{1}$': 'D', 'parameter$_{2}$': 'D', 'parameter$_{mv}$': 'D'}, {'opt_material': 'D, D'}, {'opt_material': None}]

**arg_types** = (("opt_material", "virtual", "state", "parameter$_{mv}$"), ("opt_material", "parameter$_{1}$", "parameter$_{2}$", "parameter$_{mv}$"))

**get_function** *(mat, vvar, svar, par$_{mv}$, mode=None, term_mode=None, diff_var=None, **kwargs)*

**modes** = ('weak', 'eval')

**name** = 'de_sd_dot'

### class sfepy.terms.terms_sensitivity.ESDLinearElasticTerm(*args, **kwargs)*

Sensitivity analysis of the linear elastic term.

**Definition**

\[
\int_{\Omega} \hat{D}_{ijkl} e_{ij}(v) e_{kl}(u) = \hat{D}_{ijkl} = D_{ijkl}(\nabla \cdot \mathcal{V}) - D_{ijklq} \frac{\partial V_i}{\partial x_q} - D_{ijklq} \frac{\partial V_j}{\partial x_q}
\]

**Call signature**

<table>
<thead>
<tr>
<th>de_sd_lin_elastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>(material, virtual, state, parameter$_{mv}$)</td>
</tr>
<tr>
<td>(material, parameter$<em>{1}$, parameter$</em>{2}$, parameter$_{mv}$)</td>
</tr>
</tbody>
</table>

**Arguments 1**

- material : $D$
- virtual/parameter$_{v}$ : $v$
- state/parameter$_{s}$ : $u$
- parameter$_{mv}$ : $\mathcal{V}$

**arg_shapes** = {'material': 'S, S', 'parameter$_{1}$': 'D', 'parameter$_{2}$': 'D', 'parameter$_{mv}$': 'D', 'state': 'D', 'virtual': ('D', 'state')}
arg_types = (('material', 'virtual', 'state', 'parameter_mv'), ('material', 'parameter_1', 'parameter_2', 'parameter_mv'))

geometries = ['2_3', '2_4', '3_4', '3_8']

get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('weak', 'eval')

name = 'de_sd_lin_elastic'

class sfepy.terms.terms_sensitivity.ESDLinearTractionTerm(*args, **kwargs)

Sensitivity of the linear traction term.

Definition

\[ \int_{\Gamma} \mathbf{v} \cdot (\hat{\sigma} \nabla \cdot \mathbf{V} - \hat{\sigma} \nabla \mathbf{V}) n \]

\[ \hat{\sigma} = I, \hat{\sigma} = cI \text{ or } \hat{\sigma} = \sigma \]

Call signature

<table>
<thead>
<tr>
<th>de_sd_surface_ltr</th>
<th>(opt_material, virtual, parameter_mv)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(opt_material, parameter, parameter_mv)</td>
</tr>
</tbody>
</table>

Arguments

- material: \( c, \sigma, \hat{\sigma} \)
- virtual/parameter: \( \mathbf{v} \)
- parameter_mv: \( \mathbf{V} \)

arg_shapes = [\{ 'opt_material': 'S', 1 \}, 'virtual': ('D', None), 'parameter_mv': 'D', 'parameter': 'D'], \{ 'opt_material': None \}, \{ 'opt_material': '1, 1' \}, \{ 'opt_material': 'D, D' \}]

arg_types = (('opt_material', 'virtual', 'parameter_mv'), ('opt_material', 'parameter', 'parameter_mv'))

get_function(traction, vvar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'de_sd_surface_ltr'

class sfepy.terms.terms_sensitivity.ESDPiezoCouplingTerm(*args, **kwargs)

Sensitivity (shape derivative) of the piezoelectric coupling term.

Definition

\[ \int_{\Omega} \hat{g}_{kij} e_{ij}(\mathbf{v}) \nabla k p, \int_{\Omega} \hat{g}_{kij} e_{ij}(\mathbf{u}) \nabla k q \]

\[ \hat{g}_{kij} = g_{kij}(\nabla \cdot \mathbf{V}) - g_{kil} \frac{\partial V_j}{\partial x_l} - g_{lij} \frac{\partial V_k}{\partial x_l} \]

Call signature
### de_sd_piezo_coupling

<table>
<thead>
<tr>
<th>Arguments 1</th>
<th>Arguments 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(material, virtual, state, parameter_mv)</td>
<td>(material, state, virtual, parameter_mv)</td>
</tr>
<tr>
<td>(material, state, virtual, parameter_mv)</td>
<td>(material, parameter_v, parameter_s, parameter_mv)</td>
</tr>
</tbody>
</table>

**Arguments 1**

- material : \(g_{kij}\)
- virtual/parameter_v : \(\nu\)
- state/parameter_s : \(p\)
- parameter_mv : \(\mathcal{V}\)

**Arguments 2**

- material : \(g_{kij}\)
- state : \(u\)
- virtual : \(q\)
- parameter_mv : \(\mathcal{V}\)

```
arg_shapes = {'material': 'D, S', 'parameter_mv': 'D', 'parameter_s': 1, 'parameter_v': 'D', 'state/div': 'D', 'state/grad': 1, 'virtual/div': (1, None), 'virtual/grad': ('D', None)}
arg_types = (('material', 'virtual', 'state', 'parameter_mv'), ('material', 'state', 'virtual', 'parameter_mv'), ('material', 'parameter_v', 'parameter_s', 'parameter_mv'))
geometries = ['2_3', '2_4', '3_4', '3_8']
get_function(mat, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
modes = ('grad', 'div', 'eval')
name = 'de_sd_piezo_coupling'
```

class sfepy.terms.terms_sensitivity.ESDStokesTerm(*args, **kwargs)

Stokes problem coupling term. Corresponds to weak forms of gradient and divergence terms.

**Definition**

\[
\int_{\Omega} p \hat{I}_{ij} \frac{\partial v_i}{\partial x_j} \, \int_{\Omega} q \hat{I}_{ij} \frac{\partial u_i}{\partial x_j} \\
\hat{I}_{ij} = \delta_{ij} \nabla \cdot \mathcal{V} - \frac{\partial \mathcal{V}_j}{\partial x_i}
\]

**Call signature**

<table>
<thead>
<tr>
<th>class</th>
<th>sfepy.terms.terms_sensitivity.ESDStokesTerm(*args, **kwargs)</th>
</tr>
</thead>
</table>

### de_sd_stokes

<table>
<thead>
<tr>
<th>Arguments 1</th>
<th>Arguments 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(opt_material, virtual, state, parameter_mv)</td>
<td>(opt_material, state, virtual, parameter_mv)</td>
</tr>
<tr>
<td>(opt_material, parameter_v, parameter_s, parameter_mv)</td>
<td>(opt_material, state, virtual, parameter_mv)</td>
</tr>
</tbody>
</table>

**Arguments 1**

- virtual/parameter_v: \(\nu\)
state/parameter_s: \( p \)

parameter_mv: \( \mathcal{Y} \)

**Arguments 2**

- state: \( u \)
- virtual: \( q \)
- parameter_mv: \( \mathcal{Y} \)

arg_shapes = [{'opt_material': '1, 1', 'virtual/grad': ('D', None), 'state/grad': 1, 'virtual/div': (1, None), 'state/div': 'D', 'parameter_v': 'D', 'parameter_s': 1, 'parameter_mv': 'D'}, {'opt_material': None}]

arg_types = (('opt_material', 'virtual', 'state', 'parameter_mv'), ('opt_material', 'state', 'virtual', 'parameter_mv'), ('opt_material', 'parameter_v', 'parameter_s', 'parameter_mv'))

get_function(coef, vvar, svar, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)

modes = ('grad', 'div', 'eval')

name = 'de_sd_stokes'

texpr = 'ij,ij,0'

class sfepy.terms.terms_sensitivity.ESDVectorDotGradScalarTerm(*args, **kwargs)

Sensitivity of volume dot product of a vector and a gradient of scalar.

**Definition**

\[
\int_\Omega \hat{I}_{ij} \frac{\partial p}{\partial x_j} v_i + \int_\Omega \hat{I}_{ij} \frac{\partial \eta}{\partial x_j} u_i \\
\hat{I}_{ij} = \delta_{ij} \nabla \cdot \mathcal{Y} - \frac{\partial v_j}{\partial x_i}
\]

**Call signature**

<table>
<thead>
<tr>
<th>de_sd_v_dot_grad_s</th>
<th>(opt_material, virtual, state, parameter_mv)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(opt_material, state, virtual, parameter_mv)</td>
</tr>
<tr>
<td></td>
<td>(opt_material, parameter_v, parameter_s, parameter_mv)</td>
</tr>
</tbody>
</table>

**Arguments 1**

- virtual/parameter_v: \( v \)
- state/parameter_s: \( p \)
- parameter_mv: \( \mathcal{Y} \)

**Arguments 2**

- state: \( u \)
- virtual: \( q \)
- parameter_mv: \( \mathcal{Y} \)
name = 'de_sd_v_dot_grad_s'
texpr = 'ij,i,0.j'
sfepy.terms.terms_sensitivity.get_nonsym_grad_op(sgrad)

sfepy.terms.terms_shells module

Terms implementing shell elements.

class sfepy.terms.terms_shells.Shell10XTerm(name, arg_str, integral, region, **kwargs)

The shell10x element term based on the Reissner-Mindlin theory [1], [2], corresponding to a shell of thickness \( t \).

The term requires a custom 3D quadrature, where the \( z \) components of quadrature point coordinates are transformed from \([0, 1]\) to \([-t/2, t/2]\), and the quadrature weights are multiplied by \( t \). The variables \( u \) and \( v \) have to use Shell10XField and have six components. The reference element mapping is implemented by Shell10XMapping. The term does not implement the piezo-electric components of the shell10x element yet.

The term has to be used with quadrilateral cells in 3D and should behave as the linear elastic term, but with fewer degrees of freedom for the same accuracy for shell-like structures. The shell has six degrees of freedom in each of the four nodes:

\[
\begin{bmatrix}
  u_i' \\
  v_i' \\
  w_i' \\
  \alpha_i' \\
  \beta_i' \\
  \gamma_i'
\end{bmatrix},
\]

where

\[
\begin{bmatrix}
  \tilde{u}_i \\
  \tilde{v}_i \\
  \tilde{w}_i
\end{bmatrix} = \tilde{\zeta} \begin{bmatrix}
  \tilde{e}_1 & \tilde{e}_2 \\
  \tilde{e}_1^T & \tilde{e}_2^T
\end{bmatrix} \begin{bmatrix}
  \alpha_i' \\
  \beta_i' \\
  \gamma_i'
\end{bmatrix}
\]

where \( \tilde{\zeta} = (t/2)\zeta \). The local nodal axes \( \tilde{e}_i \) are constructed in order to describe the behavior of warped (non-planar) elements adequately.

The term employs three shell element enhancements:

- DSG method
• EAS method
• drilling rotations lock (parameter $\chi$ - a good value is about $10^{-7}$)

For detailed theoretical information see the references.

High-Performance 4-Node Shell Element with Piezoelectric Coupling Mechanics of Advanced Materials and Structures Vol. 13, Iss. 5, doi:10.1080/15376490600777657


**Definition**

$$\int_{\Omega} D_{ijkl} \varepsilon_{ij}(v) \varepsilon_{kl}(u)$$

**Call signature**

```python
def dw_shell10x(material_d, material_drill, virtual, state)
```

**Arguments**

- `material_d`: $D$
- `material_drill`: $\chi$
- `virtual`: $v$
- `state`: $u$

```python
arg_shapes = {'material_d': '6', 'material_drill': '1', 'state': 6, 'virtual': (6, 'state')}
```

```python
arg_types = ('material_d', 'material_drill', 'virtual', 'state')
```

**static function**

```python
def get_fargs(mtx_d, drill, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
```

**get_physical_qps**

Get physical quadrature points corresponding to the term region and integral.

```python
integration = 'custom'
```

```python
name = 'dw_shell10x'
```

```python
poly_space_base = 'shell10x'
```

**set_integral**

Set the term integral.
**sfepy.terms.terms_surface module**

class sfepy.terms.terms_surface.ContactPlaneTerm(*args, **kwargs)

Small deformation elastic contact plane term with penetration penalty.

The plane is given by an anchor point $A$ and a normal $n$. The contact occurs in points that orthogonally project onto the plane into a polygon given by orthogonal projections of boundary points $\{B_i\}, i = 1, \ldots, N_B$ on the plane. In such points, a penetration distance $d(u) = (X + u - A, n)$ is computed, and a force $f(d(u))n$ is applied. The force depends on the non-negative parameters $k$ (stiffness) and $f_0$ (force at zero penetration):

- If $f_0 = 0$:
  
  $$
  f(d) = 0 \text{ for } d \leq 0 , \\
  f(d) = kd \text{ for } d > 0 .
  $$

- If $f_0 > 0$:
  
  $$
  f(d) = 0 \text{ for } d \leq -\frac{2r_0}{k} , \\
  f(d) = \frac{k^2}{4r_0} d^2 + kd + r_0 \text{ for } -\frac{2r_0}{k} < d \leq 0 , \\
  f(d) = kd + f_0 \text{ for } d > 0 .
  $$

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**

$$
\int_{\Gamma} v \cdot f(d(u)) n
$$

**Call signature**

```
dw_contact_plane (material_f, material_n, material_a, material_b, virtual, state)
```

**Arguments**

- **material_f**: $[k, f_0]$
- **material_n**: $n$ (special)
- **material_a**: $A$ (special)
- **material_b**: $\{B_i\}, i = 1, \ldots, N_B$ (special)
- **virtual**: $v$
- **state**: $u$

**arg_shapes** = {'material_a': ': D', 'material_b': ': N, D', 'material_f': '1, 2', 'material_n': ': D', 'state': ': D', 'virtual': (': D', 'state')}

**arg_types** = ('material_f', 'material_n', 'material_a', 'material_b', 'virtual', 'state')

```
static function(out, force, normal, geo, fmode)
```

560 Chapter 2. Development
geometries = ['3_4', '3_8']

get_fargs(force_pars, normal, anchor, bounds, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'dw_contact_plane'

static smooth_f(d, k, f0, a, eps, diff)

class sfepy.terms.terms_surface.ContactSphereTerm(*args, **kwargs)

Small deformation elastic contact sphere term with penetration penalty.

The sphere is given by a centre point $C$ and a radius $R$. The contact occurs in points that are closer to $C$ than $R$. In such points, a penetration distance $d(u) = R - ||X + u - C||$ is computed, and a force $f(d(u))n(u)$ is applied, where $n(u) = (X + u - C)/||X + u - C||$. The force depends on the non-negative parameters $k$ (stiffness) and $f_0$ (force at zero penetration):

- If $f_0 = 0$:
  
  $f(d) = 0$ for $d \leq 0$ ,
  
  $f(d) = kd$ for $d > 0$ .

- If $f_0 > 0$:
  
  $f(d) = 0$ for $d \leq -\frac{2r_0}{k}$ ,
  
  $f(d) = \frac{k^2}{4r_0}d^2 + kd + r_0$ for $-\frac{2r_0}{k} < d \leq 0$ ,
  
  $f(d) = kd + f_0$ for $d > 0$ .

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

$$\int_\Gamma v \cdot f(d(u))n(u)$$

Call signature

dw_contact_sphere (material_f, material_c, material_r, virtual, state)

Arguments

- material_f : $[k, f_0]$
- material_c : $C$ (special)
- material_r : $R$ (special)
- virtual : $v$
- state : $u$
arg_shapes = {'material_c': 'D', 'material_f': '1, 2', 'material_r': '1', 'state': 'D', 'virtual': '(D', 'state')}
arg_types = ('material_f', 'material_c', 'material_r', 'virtual', 'state')

static function(out, force, normals, fd, geo, fmode)

geometries = ['3_4', '3_8']

get_fargs(force_pars, centre, radius, virtual, state, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet'

name = 'dw_contact_sphere'

class sfepy.terms.terms_surface.LinearTractionTerm(name, arg_str, integral, region, **kwargs)

Linear traction forces, where, depending on dimension of 'material' argument, \( \sigma \cdot n \) is \( \bar{p} \cdot n \) for a given scalar pressure, \( f \) for a traction vector, and itself for a stress tensor.

The material parameter can have one of the following shapes: 1 or (1, 1), (D, 1), (S, 1) in all modes, or (D, D) in the eval mode only. The symmetric tensor storage (S, 1) is as follows: in 3D S = 6 and the indices ordered as
\[ [11, 22, 33, 12, 13, 23] \], in 2D S = 3 and the indices ordered as [11, 22, 12].

Definition

\[
\int_{\Gamma} v \cdot \sigma \cdot n, \int_{\Gamma} v \cdot n,
\]

Call signature

<table>
<thead>
<tr>
<th>dw_surface_ltr</th>
<th>(opt_material, virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(opt_material, parameter)</td>
</tr>
</tbody>
</table>

Arguments

- material : \( \sigma \)
- virtual : \( v \)

arg_shapes = [\{'opt_material': 'S', 1', 'virtual': '(D', None), 'parameter': 'D'}],
\{ 'opt_material': 'D, 1'\}, [ 'opt_material': '1, 1'], \{ 'opt_material': 'D, D'\},
\{ 'opt_material': None\}]

arg_types = ('opt_material', 'virtual'), ('opt_material', 'parameter'))

static d_fun(out, traction, val, sg)

get_eval_shape(traction, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(traction, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)
integration = 'facet'

modes = ('weak', 'eval')

name = 'dw_surface_ltr'
Sensitivity of the linear traction term.

Definition
\[ \int_{\Gamma} \vec{v} \cdot (\sigma \cdot n), \int_{\Gamma} \vec{v} \cdot n. \]

Call signature

```
ev_sd_surface_ltr (opt_material, parameter, parameter_mv)
```

Arguments
- material : \( \sigma \)
- parameter : \( \vec{v} \)

```
arg_shapes = [{'opt_material': 'S', 'parameter': 'D'},
               {'opt_material': 'D'},
               {'opt_material': '1', 'parameter': 'D'},
               {'opt_material': 'D'},
               {'opt_material': None}]
```

```
arg_types = ('opt_material', 'parameter', 'parameter_mv')
```

```
static d_fun(out, traction, val, grad_mv, div_mv, sg)
```

```
get_eval_shape(traction, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
get_fargs(traction, par_u, par_mv, mode=None, term_mode=None, diff_var=None, **kwargs)
```

```
integration = 'facet'
```

```
name = 'ev_sd_surface_ltr'
```

```
set_arg_types()
```

Sensitivity of scalar traction.

Definition
\[ \int_{\Gamma} p \nabla \cdot \nabla \]

Call signature

```
ev_sd_surface_integrate (parameter, parameter_mv)
```

Arguments
- parameter : \( p \)
- parameter_mv : \( \nabla \)

```
arg_shapes = {'parameter': 1, 'parameter_mv': 'D'}
```

```
arg_types = ('parameter', 'parameter_mv')

static function(out, val_p, div_v, sg)

get_eval_shape(par, par_v, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(par, par_v, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

name = 'ev_sd_surface_integrate'

class sfepy.terms.terms_surface.SurfaceNormalDotTerm(name, arg_str, integral, region, **kwargs)

“Scalar traction” term, (weak form).

Definition

\[ \int_{\Gamma} q_{c} \cdot n \]

Call signature

<table>
<thead>
<tr>
<th>dw_surface_ndot</th>
</tr>
</thead>
<tbody>
<tr>
<td>(material, virtual)</td>
</tr>
<tr>
<td>(material, parameter)</td>
</tr>
</tbody>
</table>

Arguments

- material : \( c \)
- virtual : \( q \)

arg_shapes = {'material': 'D, 1', 'parameter': 1, 'virtual': (1, None)}

arg_types = (('material', 'virtual'), ('material', 'parameter'))

static d_fun(out, material, val, sg)

static dw_fun(out, material, bf, sg)

get_eval_shape(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

integration = 'facet'

modes = ('weak', 'eval')

name = 'dw_surface_ndot'

set_arg_types()

class sfepy.terms.terms_surface.SurfaceJumpTerm(name, arg_str, integral, region, **kwargs)

Interface jump condition.

Definition

\[ \int_{\Gamma} c q (p_1 - p_2) \]

Call signature
**dw_jump** (opt_material, virtual, state_1, state_2)

**Arguments**

- **material** : \( c \)
- **virtual** : \( q \)
- **state_1** : \( p_1 \)
- **state_2** : \( p_2 \)

**arg_shapes** = [{'opt_material': '1, 1', 'virtual': (1, None), 'state_1': 1, 'state_2': 1}, {'opt_material': None}]

**arg_types** = ('opt_material', 'virtual', 'state_1', 'state_2')

**static function***(out, jump, mul, bf1, bf2, sg, fmode)***

**get_fargs**(coef, virtual, state1, state2, mode=None, term_mode=None, diff_var=None, **kwargs)

**integration** = 'facet'

**name** = 'dw_jump'

**sfepy.terms.terms_th module**

**class** *sfepy.terms.terms_th.ETHTerm*(name, arg_str, integral, region, **kwargs)*

Base class for terms depending on time history with exponential convolution kernel (fading memory terms).

**advance_eth_data**(ts, data)

**get_eth_data**(key, state, decay, values)

**class** *sfepy.terms.terms_th.THTerm*(name, arg_str, integral, region, **kwargs)*

Base class for terms depending on time history (fading memory terms).

**eval_real**(shape, fargs, mode='eval', term_mode=None, diff_var=None, **kwargs)

**sfepy.terms.terms_volume module**

**class** *sfepy.terms.terms_volume.LinearVolumeForceTerm*(name, arg_str, integral, region, **kwargs)*

Vector or scalar linear volume forces (weak form) — a right-hand side source term.

**Definition**

\[
\int_{\Omega} f \cdot v \quad \text{or} \quad \int_{\Omega} f q
\]

**Call signature**

**dw_volume_lvf** (material, virtual)

**Arguments**
• material : $f$ or $f$
• virtual : $v$ or $q$

arg_shapes = [{'material': 'D, 1', 'virtual': ('D', None)}, {'material': '1, 1', 'virtual': (1, None)}]

arg_types = ('material', 'virtual')

static function()

get_fargs(mat, virtual, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_volume_lvf'

class sfepy.terms.terms_volume.NonlinearVolumeForceTerm(name, arg_str, integral, region, **kwargs)
The volume force term with the force given by a user supplied function of the state variable.

Definition

\[ \int_{\Omega} q f(p) \]

Call signature

| dw_volume_nvf | (fun, dfun, virtual, state) |

Arguments
• fun : $f(p)$
• dfun : $\partial f(p)/\partial p$
• virtual : $q$
• state : $p$

arg_shapes = {'dfun': <function NonlinearVolumeForceTerm.<lambda>>, 'fun':
<function NonlinearVolumeForceTerm.<lambda>>, 'state': 1, 'virtual': (1, 'state')}

arg_types = ('fun', 'dfun', 'virtual', 'state')

static function(out, out_qp, geo)

get_fargs(fun, dfun, var1, var2, mode=None, term_mode=None, diff_var=None, **kwargs)

name = 'dw_volume_nvf'

sfepy.terms.utils module

sfepy.terms.utils.check_finiteness(data, info)

sfepy.terms.utils.get_range_indices(num)

Return indices and slices in given range.

Returns

index

[list of tuples] The list of (ii, slice(ii, ii + 1)) of the indices. The first item is the index itself, the second item is a convenience slice to index components of material parameters.
sfepy.terms.extmods.terms module

Low level term evaluation functions.
sfepy.terms.extmods.terms.actBfT()
sfepy.terms.extmods.terms.d_biot_div()
sfepy.terms.extmods.terms.d_diffusion()
sfepy.terms.extmods.terms.d_laplace()
sfepy.terms.extmods.terms.d_lin_elastic()
sfepy.terms.extmods.terms.d_of_nsMinGrad()
sfepy.terms.extmods.terms.d_of_nsSurfMinDPress()
sfepy.terms.extmods.terms.d_piezo_coupling()
sfepy.terms.extmods.terms.d_sd_convect()
sfepy.terms.extmods.terms.d_sd_diffusion()
sfepy.terms.extmods.terms.d_sd_div()
sfepy.terms.extmods.terms.d_sd_div_grad()
sfepy.terms.extmods.terms.d_sd_lin_elastic()
sfepy.terms.extmods.terms.d_sd_st_grad_div()
sfepy.terms.extmods.terms.d_sd_st_pspg_c()
sfepy.terms.extmods.terms.d_sd_st_pspg_p()
sfepy.terms.extmods.terms.d_sd_st_supg_c()
sfepy.terms.extmods.terms.d_sd_volume_dot()
sfepy.terms.extmods.terms.d_surface_flux()
sfepy.terms.extmods.terms.d_tl_surface_flux()
sfepy.terms.extmods.terms.d_tl_volume_surface()
sfepy.terms.extmods.terms.d_volume_surface()
sfepy.terms.extmods.terms.de_cauchy_strain()
sfepy.terms.extmods.terms.de_cauchy_stress()
sfepy.terms.extmods.terms.de_he_rtm()
sfepy.terms.extmods.terms.di_surface_moment()
sfepy.terms.extmods.terms.dq_cauchy_strain()
sfepy.terms.extmods.terms.dq_def_grad()
sfepy.terms.extmods.terms.dq_div_vector()
sfepy.terms.extmods.terms.dq_finite_strain_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_convevt1_v_grad_s()
sfepy.terms.extmods.terms.dw_diffusion()
sfepy.terms.extmods.terms.dw_diffusion_r()
sfepy.terms.extmods.terms.dw_div()
sfepy.terms.extmods.terms.dw_electric_source()
sfepy.terms.extmods.terms.dw_grad()
sfepy.terms.extmods.terms.dw_he_rtm()
sfepy.terms.extmods.terms.dw_laplace()
sfepy.terms.extmods.terms.dw_lin_conveet()
sfepy.terms.extmods.terms.dw_lin_elastic()
sfepy.terms.extmods.terms.dw_lin_prestress()
sfepy.terms.extmods.terms.dw_lin_strain_fib()
sfepy.terms.extmods.terms.dw_nonsym_elastic()
sfepy.terms.extmods.terms.dw_piezo_coupling()
sfepy.terms.extmods.terms.dw_st_adj1_supg_p()
sfepy.terms.extmods.terms.dw_st_adj2_supg_p()
sfepy.terms.extmods.terms.dw_st_adj_supg_c()
sfepy.terms.extmods.terms.dw_st_grad_div()
sfepy.terms.extmods.terms.dw_st_pspg_c()
sfepy.terms.extmods.terms.dw_st_supg_c()
sfepy.terms.extmods.terms.dw_st_supg_p()
sfepy.terms.extmods.terms.dw_surface_flux()
sfepy.terms.extmods.terms.dw_surface_ltr()
sfepy.terms.extmods.terms.dw_surface_s_v_dot_n()
sfepy.terms.extmods.terms.dw_surface_v_dot_n_s()
sfepy.terms.extmods.terms.dw_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.mulAB_integrate()
sfepy.terms.extmods.terms.sym2nonsym()
sfepy.terms.extmods.terms.term_ns_asm_conve()
sfepy.terms.extmods.terms.term_ns_asm_div_grad()

**Scripts**

**sfepy.scripts.blockgen module**

Block mesh generator.

sfepy.scripts.blockgen.add_args(parser)
sfepy.scripts.blockgen.gen_block(options)
sfepy.scripts.blockgen.main()

**sfepy.scripts.convert_mesh module**

Convert a mesh file from one SfePy-supported format to another.

sfepy.scripts.convert_mesh.main()

**sfepy.scripts.cylindergen module**

Cylinder mesh generator.

sfepy.scripts.cylindergen.add_args(parser)
sfepy.scripts.cylindergen.gen_cylinder(options)
sfepy.scripts.cylindergen.main()

**sfepy.scripts.gen_iga_patch module**

Generate a single IGA patch block in 2D or 3D of given degrees and continuity using igakit.

The grid has equally-spaced knot vectors.

sfepy.scripts.gen_iga_patch.add_args(parser)
sfepy.scripts.gen_iga_patch.gen_iga_patch(options)
sfepy.scripts.gen_iga_patch.main()
**sfepy.scripts.gen_mesh module**

Simple mesh generators and statistics.

`sfepy.scripts.gen_mesh.main()`

**sfepy.scripts.gen_mesh_prev module**

Mesh Preview Generator.

**Examples**

```
$ ./script/gen_mesh_prev.py meshes/2d/
```

`sfepy.scripts.gen_mesh_prev.gen_shot(vtk_filename, png_filename)`

Generate PNG image of the FE mesh.

**Parameters**

- **vtk_filename**
  - [str] The input mesh filename (file in VTK format).

- **png_filename**
  - [str] The name of the output PNG file.

`sfepy.scripts.gen_mesh_prev.main()`

**sfepy.scripts.plot_condition_numbers module**

Plot conditions numbers w.r.t. polynomial approximation order of reference element matrices for various FE polynomial spaces (bases).

`sfepy.scripts.plot_condition_numbers.main()`

**sfepy.scripts.plot_logs module**

Plot logs of variables saved in a text file by sfepy.base.log.Log class.

The plot should be almost the same as the plot that would be generated by the Log directly.

```python
class sfepy.scripts.plot_logs.ParseRc(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None)
```

`sfepy.scripts.plot_logs.main()`
**sfepy.scripts.plot_mesh module**

Plot mesh connectivities, facet orientations, global and local DOF ids etc.
To switch off plotting some mesh entities, set the corresponding color to `None`.

```python
sfepy.scripts.plot_mesh.main()
```

**sfepy.scripts.plot_quadratures module**

Plot quadrature points for the given geometry and integration order.

```python
sfepy.scripts.plot_quadratures.main()
```

**sfepy.scripts.plot_times module**

Plot time steps, times of time steps and time deltas in a HDF5 results file.

```python
sfepy.scripts.plot_times.main()
```

**sfepy.scripts.probe module**

Probe finite element solutions in points defined by various geometrical probes.

In the examples below it is supposed that sfepy is installed. When using the in-place build, replace `sfepy-probe` by `python3 sfepy/scripts/probe.py`.

**Generation mode**

```bash
sfepy-probe [generation options] <input file> <results file>
```

Probe the data in the results file corresponding to the problem defined in the input file. The input file options must contain 'gen_probes' and 'probe_hook' keys, pointing to proper functions accessible from the input file scope.

For each probe returned by `gen_probes()` a data plot figure and a text file with the data plotted are saved, see the options below.

**Generation options**

- `-o, --auto-dir, --same-dir, -f, --only-names, -s`

**Postprocessing mode**

```bash
sfepy-probe [postprocessing options] <probe file> <figure file>
```

Read a previously probed data from the probe text file, re-plot them, and integrate them along the probe.
Postprocessing options

--postprocess, --radial, --only-names

Notes

For extremely thin hexahedral elements the Newton’s iteration for finding the reference element coordinates might converge to a spurious solution outside of the element. To obtain some values even in this case, try increasing the --close-limit option value.

sfepy.scripts.probe.generate_probes(filename_input, filename_results, options, conf=None, problem=None, probes=None, labels=None, probe_hooks=None)

Generate probe figures and data files.

sfepy.scripts.probe.integrate_along_line(x, y, is_radial=False)

Integrate numerically (trapezoidal rule) a function \( y = y(x) \).
If is_radial is True, multiply each \( y \) by \( 4\pi x^2 \).

sfepy.scripts.probe.main()

sfepy.scripts.probe.postprocess(filename_input, filename_results, options)

Postprocess probe data files - replot, integrate data.

sfepy.scripts.resview module

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

In the examples below it is supposed that sfepy is installed. When using the in-place build, replace sfepy-view by python3 sfepy/scripts/resview.py.

Examples

The examples assume that python -c "import sfepy; sfepy.test('--output-dir=output-tests')" has been run successfully and the resulting data files are present.

- View data in output-tests/test_navier_stokes.vtk:

  sfepy-view output-tests/navier_stokes-navier_stokes.vtk

- Customize the above output: plot0: field “p”, switch on edges, plot1: field “u”, surface with opacity 0.4, glyphs scaled by factor 2e-2:

  sfepy-view output-tests/navier_stokes-navier_stokes.vtk -f p:e:p0 u:o.4:p1 u:g:f2e-2:p1

- As above, but glyphs are scaled by the factor determined automatically as 20% of the minimum bounding box size:

  sfepy-view output-tests/navier_stokes-navier_stokes.vtk -f p:e:p0 u:o.4:p1 u:g:f10-%:p1

- View data and take a screenshot:
• Take a screenshot without a window popping up:
  `sfepy-view output-tests/diffusion-poisson.vtk -o image.png --off-screen`

• Create animation from output-tests/diffusion-time_poisson.*.vtk:
  `sfepy-view output-tests/diffusion-time_poisson.*.vtk -a mov.mp4`

• Create animation from output-tests/test_hyperelastic.*.vtk, set frame rate to 3, plot displacements and mooney_rivlin_stress:
  `sfepy-view output-tests/test_hyperelastic_TL.*.vtk -f u:wu:e:p0 mooney_rivlin_stress:p1 -a mov.mp4 -r 3`

```python
class sfepy.scripts.resview.FieldOptsToListAction(
    option_strings, dest, nargs=None, const=None,
    default=None, type=None, choices=None,
    required=False, help=None, metavar=None
)

    separator = ':

class sfepy.scripts.resview.OptsToListAction(
    option_strings, dest, nargs=None, const=None,
    default=None, type=None, choices=None,
    required=False, help=None, metavar=None
)

    separator = '='

class sfepy.scripts.resview.StoreNumberAction(
    option_strings, dest, nargs=None, const=None,
    default=None, type=None, choices=None,
    required=False, help=None, metavar=None
)
```

```python
sfepy.scripts.resview.add_mat_id_to_grid(grid, cell_groups)
```

```python
sfepy.scripts.resview.get_camera_position(bounds, azimuth, elevation, distance=None, zoom=1.0)
```

```python
sfepy.scripts.resview.main()
```

```python
sfepy.scripts.resview.make_cells_from_conn(conns, convert_to_vtk_type)
```

```python
sfepy.scripts.resview.make_grid_from_mesh(mesh, add_mat_id=False)
```

```python
sfepy.scripts.resview.make_title(filenames)
```

```python
sfepy.scripts.resview.parse_options(opts, separator=':')
```

```python
sfepy.scripts.resview.print_camera_position(plotter)
```

```python
sfepy.scripts.resview.pv_plot(filenames, options, plotter=None, step=None, scalar_bar_limits=None,
    ret_scalar_bar_limits=False, step_inc=None, use_cache=True)
```

```python
sfepy.scripts.resview.read_mesh(filenames, step=None, print_info=True, ret_n_steps=False,
    use_cache=True)
```
**sfepy.scripts.run_tests module**

Run SfePy tests. All arguments are passed to pytest.

`sfepy.scripts.run_tests.main()`

**sfepy.scripts.simple module**

Solve partial differential equations given in a SfePy problem definition file.

Example problem definition files can be found in `sfepy/examples/` directory of the SfePy top-level directory.

In the examples below it is supposed that `sfepy` is installed. When using the in-place build, replace `sfepy-run` by `python3 sfepy/scripts/simpepy.run`.

The supported application kinds (`--app` option) are:

- **bvp** - boundary value problem. Example:
  ```
  sfepy-run sfepy/examples/diffusion/poisson.py
  ```

- **homogen** - calculation of local microscopic problems (correctors) and homogenized coefficients. Example:
  ```
  sfepy-run sfepy/examples/homogenization/perfusion_micro.py
  ```

- **bvp-mM** - micro-macro boundary value problem. Solve a coupled two-scale problem in parallel using MPI. One computational node is solving a macroscopic equation while the others are solving local microscopic problems and homogenized coefficients. The `--app` option is required in this case. Example:
  ```
  mpiexec -n 4 sfepy-run --app=bvp-mM --debug-mpi sfepy/examples/homogenization/
  →nonlinear_hyperelastic_mM.py
  ```

- **evp** - eigenvalue problem. Example:
  ```
  sfepy-run sfepy/examples/quantum/well.py
  ```

- **phonon** - phononic band gaps. Example:
  ```
  sfepy-run sfepy/examples/phononic/band_gaps.py --phonon-plot
  ```

Both normal and parametric study runs are supported. A parametric study allows repeated runs for varying some of the simulation parameters - see `sfepy/examples/diffusion/poisson_parametric_study.py` file.

`sfepy.scripts.simple.main()`

`sfepy.scripts.simple.print_solvers()`

`sfepy.scripts.simple.print_terms()`
Tests

**sfepy.tests.conftest module**

*sfepy.tests.conftest.output_dir*(request, tmpdir_factory)*

Output directory for tests.

*sfepy.tests.conftest.pytest_addoption*(parser)*

*sfepy.tests.conftest.pytest_configure*(config)*

**sfepy.tests.test_assembling module**

*sfepy.tests.test_assembling.data()*

*sfepy.tests.test_assembling.test_assemble_matrix*(data)*

*sfepy.tests.test_assembling.test_assemble_matrix_complex*(data)*

*sfepy.tests.test_assembling.test_assemble_vector*(data)*

*sfepy.tests.test_assembling.test_assemble_vector_complex*(data)*

**sfepy.tests.test_base module**

*sfepy.tests.test_base.test_container_add()*

*sfepy.tests.test_base.test_parse_conf()*

*sfepy.tests.test_base.test_resolve_deps()*

*sfepy.tests.test_base.test_struct_add()*

*sfepy.tests.test_base.test_struct_i_add()*

*sfepy.tests.test_base.test_verbose_output()*

**sfepy.tests.test_cmesh module**

*sfepy.tests.test_cmesh.filename_meshes()*

*sfepy.tests.test_cmesh.test_cmesh_counts*(filename_meshes)*

*sfepy.tests.test_cmesh.test_entity_volumes()*
sfepy.tests.test_conditions module

sfepy.tests.test_conditions.check_vec(vec, ii, ok, conds, variables)
sfepy.tests.test_conditions.data()
sfepy.tests.test_conditions.init_vec(variables)
sfepy.tests.test_conditions.test_ebcs(data)
sfepy.tests.test_conditions.test_epbcs(data)
sfepy.tests.test_conditions.test_ics(data)
sfepy.tests.test_conditions.test_save_ebc(data, output_dir)

sfepy.tests.test_declarative_examples module

sfepy.tests.test_declarative_examples.inedir(filename)
sfepy.tests.test_declarative_examples.test_examples(ex_filename, output_dir)
sfepy.tests.test_declarative_examples.test_examples_dg(ex_filename, output_dir)

sfepy.tests.test_dg_field module

class sfepy.tests.test_dg_field.TestDGField
    test_create_output1D()
    test_create_output2D()
    test_get_bc_facet_values_1D()
    test_get_bc_facet_values_2D()
    test_get_bc_facet_values_2D_const()
    test_get_facet_idx1D()
    test_get_facet_idx2D()
    test_get_facet_neighbor_idx_1d()
    test_get_facet_neighbor_idx_2d()
    test_set_dofs_1D()
    test_set_dofs_2D()

sfepy.tests.test_dg_field.prepare_dgfield(approx_order, mesh)
sfepy.tests.test_dg_field.prepare_dgfield_1D(approx_order)
sfepy.tests.test_dg_field.prepare_field_2D(approx_order)
sfepy.tests.test_dg_terms_calls module

Test all terms in terms_dg. Performs numerical test on simple mesh.

class sfepy.tests.test_dg_terms_calls.DGTermTestEnvironment(dim, approx_order, **kwargs)

Class for easy creation of all the data needed for testing terms.

burg_fun(u)

burg_fun_d(u)

prepare_materials(field, velo=1.0, diffusion=0.1, penalty=100)

Crate material objects with data attribute, containing properly shaped data to pass to terms

Parameters

• field – DGField
• velo – optional values for velocity a
• diffusion – optional value for diffusion tensor D
• penalty – optional value for diffusion penalty Cw

Returns

a, D, Cw

prepare_variables(field)

Prepares state and test variables, adds empty eq_map to state variable

Parameters

field –

Returns

state, test

class sfepy.tests.test_dg_terms_calls.TestAdvectDGFluxTerm

test_function_explicit_1D(dg_test_env)

test_function_implicit_1D(dg_test_env)

class sfepy.tests.test_dg_terms_calls.TestDiffusionDGFluxTerm

test_function_explicit_left_1D(dg_test_env)

test_function_explicit_right_1D(dg_test_env)

test_function_implicit_left_1D(dg_test_env)

test_function_implicit_right_1D(dg_test_env)

class sfepy.tests.test_dg_terms_calls.TestDiffusionInteriorPenaltyTerm

test_function_explicit_1D(dg_test_env)

test_function_implicit_1D(dg_test_env)

class sfepy.tests.test_dg_terms_calls.TestNonlinScalarDotGradTerm

test_function_explicit_1D(dg_test_env)
class sfepy.tests.test_dg_terms_calls.TestNonlinearHyperDGFluxTerm

    test_function_explicit_1D(dg_test_env)

dfpy.tests.test_dg_terms_calls.dg_test_env(request)

sfepy.tests.test_domain module

sfepy.tests.test_domain.compare_mesh(geo_name, coors, conn)
sfepy.tests.test_domain.domain()
sfepy.tests.test_domain.refine(domain, out_dir, level=3)
sfepy.tests.test_domain.test_facets(domain)
sfepy.tests.test_domain.test_refine_2_3(output_dir)
sfepy.tests.test_domain.test_refine_2_4(output_dir)
sfepy.tests.test_domain.test_refine_3_4(output_dir)
sfepy.tests.test_domain.test_refine_3_8(output_dir)
sfepy.tests.test_domain.test_refine_hexa(output_dir)
sfepy.tests.test_domain.test_refine_tetra(domain, output_dir)

sfepy.tests.test_ed_solvers module

sfepy.tests.test_ed_solvers.define(t1=1.5e-05, dt=1e-06, dims=(0.1, 0.02, 0.005), shape=(11, 3, 3),
young=7000000000.0, poisson=0.3, density=2700,
mass_lumping='row_sum', mass_beta=0.2)

sfepy.tests.test_ed_solvers.problem()

sfepy.tests.test_ed_solvers.test_active_only(output_dir)

    Note: with tsc the results would differ, as eval_scaled_norm() depends on the vector length.

sfepy.tests.test_ed_solvers.test_ed_solvers(problem, output_dir)

sfepy.tests.test_ed_solvers.test_rmm_solver(problem, output_dir)

sfepy.tests.test_eigenvalue_solvers module

sfepy.tests.test_eigenvalue_solvers.data()

sfepy.tests.test_eigenvalue_solvers.mesh_hook(mesh, mode)

    Generate the block mesh.

sfepy.tests.test_eigenvalue_solvers.test_eigenvalue_solvers(data)
sfepy.tests.test_elasticity_small_strain module

sfepy.tests.test_elasticity_small_strain.get_pars(dim, full=False)
sfepy.tests.test_elasticity_small_strain.solutions(output_dir)
sfepy.tests.test_elasticity_small_strain.test_converged(solutions)
sfepy.tests.test_elasticity_small_strain.test_linear_terms(solutions)

sfepy.tests.test_fem module

sfepy.tests.test_fem.gels()
sfepy.tests.test_fem.test_base_functions_delta(gels)
    Test δ property of base functions evaluated in the reference element nodes.
sfepy.tests.test_fem.test_base_functions_values(gels)
    Compare base function values and their gradients with correct data. Also test that sum of values over all element nodes gives one.

sfepy.tests.test_functions module

sfepy.tests.test_functions.get_circle(coors, domain=None)
sfepy.tests.test_functions.get_p_edge(ts, coors, bc=None, **kwargs)
sfepy.tests.test_functions.get_pars(ts, coors, mode=None, extra_arg=None, equations=None, term=None, problem=None, **kwargs)
sfepy.tests.test_functions.get_u_edge(ts, coors, bc=None, **kwargs)
sfepy.tests.test_functions.problem()
sfepy.tests.test_functions.test_ebc_functions(problem, output_dir)
sfepy.tests.test_functions.test_material_functions(problem)
sfepy.tests.test_functions.test_region_functions(problem, output_dir)

sfepy.tests.test_high_level module

sfepy.tests.test_high_level.data()
sfepy.tests.test_high_level.fix_u_fun(ts, coors, bc=None, problem=None, extra_arg=None)
sfepy.tests.test_high_level.test_solving(data, output_dir)
sfepy.tests.test_high_level.test_term_arithmetics(data)
sfepy.tests.test_high_level.test_term_evaluation(data)
sfepy.tests.test_high_level.test_variables(data)
sfepy.tests.test_homogenization_engine module

sfepy.tests.test_homogenization_engine.test_chunk_micro()

sfepy.tests.test_homogenization_engine.test_dependencies()

sfepy.tests.test_homogenization_perfusion module

sfepy.tests.test_homogenization_perfusion.compare_scalars(s1, s2, l1='s1', l2='s2', allowed_error=1e-08)

sfepy.tests.test_homogenization_perfusion.test_solution(output_dir)

sfepy.tests.test_hyperelastic_tlul module

sfepy.tests.test_hyperelastic_tlul.test_solution(output_dir)

sfepy.tests.test_io module

sfepy.tests.test_io.test_recursive_dict_hdf5(output_dir)

sfepy.tests.test_io.test_sparse_matrix_hdf5(output_dir)

sfepy.tests.test_laplace_unit_disk module

sfepy.tests.test_laplace_unit_disk.data()

sfepy.tests.test_laplace_unit_disk.test_boundary_fluxes(data)

sfepy.tests.test_laplace_unit_square module

sfepy.tests.test_laplace_unit_square.data()

sfepy.tests.test_laplace_unit_square.linear(bc, ts, coor, which)

sfepy.tests.test_laplace_unit_square.linear_x(bc, ts, coor)

sfepy.tests.test_laplace_unit_square.linear_y(bc, ts, coor)

sfepy.tests.test_laplace_unit_square.linear_z(bc, ts, coor)

sfepy.tests.test_laplace_unit_square.test_boundary_fluxes(data, output_dir)

sfepy.tests.test_laplace_unit_square.test_solution(data)
sfepy.tests.test_lcbcs module

sfepy.tests.test_lcbcs.test_elasticity_rigid(mesh_filename, output_dir)
sfepy.tests.test_lcbcs.test_laplace_shifted_periodic(output_dir)
sfepy.tests.test_lcbcs.test_stokes_slip_bc(output_dir)

sfepy.tests.test_linalg module

dsfePy.tests.test_linalg.test_assemble1d()
dsfePy.tests.test_linalg.test_geometry()
dsfePy.tests.test_linalg.test_get_blocks_stats()
dsfePy.tests.test_linalg.test_tensors()
dsfePy.tests.test_linalg.test_unique_rows()

sfepy.tests.test_linear_solvers module

class sfepy.tests.test_linear_solvers.DiagPC
    Diagonal (Jacobi) preconditioner.
    Equivalent to setting ‘precond’ : ‘jacobi’.
    apply(pc, x, y)
    setUp(pc)
dsfePy.tests.test_linear_solvers.problem()
dsfePy.tests.test_linear_solvers.setup_petsc_precond(mtx, problem)
dsfePy.tests.test_linear_solvers.test_ls_reuse(problem)
dsfePy.tests.test_linear_solvers.test_solvers(problem, output_dir)

sfepy.tests.test_linearization module

sfepy.tests.test_linearization.test_linearization(output_dir)

sfepy.tests.test_log module

sfepy.tests.test_log.log(log_filename)
sfepy.tests.test_log.log_filename(output_dir)
sfepy.tests.test_log.test_log_rw(log_filename, log, output_dir)
**sfepy.tests.test_matcoefs module**

- `test_conversion_functions()`
- `test_elastic_constants()`
- `test_stiffness_tensors()`
- `test_wave_speeds()`

**sfepy.tests.test_mesh_expand module**

- `test_mesh_expand()`

**sfepy.tests.test_mesh_generators module**

- `test_gen_block_mesh(output_dir)`
- `test_gen_cylinder_mesh(output_dir)`
- `test_gen_extended_block_mesh(output_dir)`
- `test_gen_mesh_from_geom(output_dir)`
- `test_gen_mesh_from_voxels(output_dir)`
- `test_gen_tiled_mesh(output_dir)`

**sfepy.tests.test_mesh_interp module**

- `do_interpolation(m2, m1, data, field_name, force=False)`
  
  Interpolate data from m1 to m2.
- `gen_datas(meshes)`
- `in_dir(adir)`
- `prepare_variable(filename, n_components)`
- `test_evaluate_at()`
- `test_field_gradient()`
- `test_interpolation(output_dir)`
- `test_interpolation_two_meshes(output_dir)`
- `test_invariance()`
- `test_invariance_qp()`
sfepy.tests.test_mesh_smoothing module

sfepy.tests.test_mesh_smoothing.get_volume(el, nd)
sfepy.tests.test_mesh_smoothing.test_mesh_smoothing(output_dir)

sfepy.tests.test_meshio module

sfepy.tests.test_meshio.mesh_hook(mesh, mode)
   Define a mesh programmatically.
sfepy.tests.test_meshio.test_compare_same_meshes()
   Compare same meshes in various formats.
sfepy.tests.test_meshio.test_hdf5_meshio()
sfepy.tests.test_meshio.test_read_dimension()
sfepy.tests.test_meshio.test_read_meshes()
   Try to read all listed meshes.
sfepy.tests.test_meshio.test_write_read_meshes(output_dir)
   Try to write and then read all supported formats.

sfepy.tests.test_msm_laplace module

sfepy.tests.test_msm_laplace.ebc(ts, coor, **kwargs)
sfepy.tests.test_msm_laplace.problem()
sfepy.tests.test_msm_laplace.rhs(ts, coor, mode=None, expression=None, **kwargs)
sfepy.tests.test_msm_laplace.test_msm_laplace(problem, output_dir)

sfepy.tests.test_msm_symbolic module

sfepy.tests.test_msm_symbolic.ebc(ts, coor, solution=None)
sfepy.tests.test_msm_symbolic.problem()
sfepy.tests.test_msm_symbolic.rhs(ts, coor, mode=None, expression=None, **kwargs)
sfepy.tests.test_msm_symbolic.test_msm_symbolic_diffusion(problem, output_dir)
sfepy.tests.test_msm_symbolic.test_msm_symbolic_laplace(problem, output_dir)
sfepy.tests.test_normals module

sfepy.tests.test_normals.test_normals()

Check orientations of surface normals on the reference elements.

sfepy.tests.test_parsing module

sfepy.tests.test_parsing.test_parse_equations()
sfepy.tests.test_parsing.test_parse_regions()

sfepy.tests.test_poly_spaces module

Test continuity of polynomial basis and its gradients along an edge on y line (2D) or on a face in x-y plane (3D) between two elements aligned with the coordinate system, stack one on top of the other. The evaluation occurs in several points shifted by a very small amount from the boundary between the elements into the top and the bottom element.

For H1 space, the basis should be continuous. The components of its gradient parallel to the edge/face should be continuous as well, while the perpendicular component should have the same absolute value, but different sign in the top and the bottom element.

All connectivity permutations of the two elements are tested.

The serendipity basis implementation is a pure python proof-of-concept. Its order in continuity tests is limited to 2 on 3_8 elements to decrease the tests run time.

sfepy.tests.test_poly_spaces.gels()
sfepy.tests.test_poly_spaces.test_continuity(gels)
sfepy.tests.test_poly_spaces.test_gradients(gels)
sfepy.tests.test_poly_spaces.test_hessians(gels)

Test the second partial derivatives of basis functions using finite differences.

sfepy.tests.test_poly_spaces.test_partition_of_unity(gels)

sfepy.tests.test_projections module

sfepy.tests.test_projections.data()
sfepy.tests.test_projections.test_mass_matrix(data)
sfepy.tests.test_projections.test_project_tensors(data)
sfepy.tests.test_projections.test_projection_iga_fem()
sfepy.tests.test_projections.test_projection_tri_quad(data, output_dir)
sfepy.tests.test_quadratures module

sfepy.tests.test_quadratures.get_poly(order, dim, is_simplex=False)
    Construct a polynomial of given order in space dimension dim, and integrate it symbolically over a rectangular or simplex domain for coordinates in [0, 1].

sfepy.tests.test_quadratures.symarray(prefix, shape)
    Copied from SymPy so that the tests pass for its different versions.

sfepy.tests.test_quadratures.test_quadratures()
    Test if the quadratures have orders they claim to have, using symbolic integration by sympy.

sfepy.tests.test_quadratures.test_weight_consistency()
    Test if integral of 1 (= sum of weights) gives the domain volume.

sfepy.tests.test_ref_coors module

sfepy.tests.test_ref_coors.test_ref_coors_fem()

sfepy.tests.test_ref_coors.test_ref_coors_iga()

sfepy.tests.test_refine_hanging module

Test continuity along a boundary with hanging nodes due to a mesh refinement.

sfepy.tests.test_refine_hanging.eval_fun(ts, coors, mode, **kwargs)

sfepy.tests.test_refine_hanging.gels()

sfepy.tests.test_refine_hanging.test_continuity(gels, output_dir)

sfepy.tests.test_refine_hanging.test_preserve_coarse_entities(output_dir)

sfepy.tests.test_regions module

sfepy.tests.test_regions.data()

sfepy.tests.test_regions.get_cells(coors, domain=None)

sfepy.tests.test_regions.get_vertices(coors, domain=None)

sfepy.tests.test_regions.test_operators(data)
    Test operators in region selectors.

sfepy.tests.test_regions.test_selectors(data)
    Test basic region selectors.
sfepy.tests.test_semismooth_newton module

sfepy.tests.test_semismooth_newton.convert_to_csr(m_in)
sfepy.tests.test_semismooth_newton.define_matrices()
sfepy.tests.test_semismooth_newton.eval_matrix(mtx, **kwargs)
sfepy.tests.test_semismooth_newton.test_semismooth_newton()

sfepy.tests.test_sparse module

sfepy.tests.test_sparse.test_compose_sparse()

sfepy.tests.test_splinebox module

sfepy.tests.test_splinebox.test_spbox_2d()  
Check position of a given vertex in the deformed mesh.
sfepy.tests.test_splinebox.test_spbox_3d()  
Check volume change of the mesh which is deformed using the SplineBox functions.
sfepy.tests.test_splinebox.test_spbox_field()  
‘Field’ vs. ’coors’.
sfepy.tests.test_splinebox.test_spregion2d()  
Check position of a given vertex in the deformed mesh.
sfepy.tests.test_splinebox.tetravolume(cells, vertices)

sfepy.tests.test_tensors module

sfepy.tests.test_tensors.get_ortho_d(phi1, phi2)
sfepy.tests.test_tensors.test_stress_transform()
sfepy.tests.test_tensors.test_tensors()
sfepy.tests.test_tensors.test_transform_data()
sfepy.tests.test_tensors.test_transform_data4()

sfepy.tests.test_term_call_modes module

sfepy.tests.test_term_call_modes.data()
sfepy.tests.test_term_call_modes.make_term_args(arg_shapes, arg_kinds, arg_types, ats_mode, domain, material_value=None, poly_space_base=None)
sfepy.tests.test_term_call_modes.test_term_call_modes(data)
**SfePy Documentation, Release version: 2023.4+git.4f968b9d**

**sfepy.tests.test_term_consistency module**

`sfepy.tests.test_term_consistency.get_pars(ts, coor=None, mode=None, term=None, **kwargs)`

`sfepy.tests.test_term_consistency.problem()`

`sfepy.tests.test_term_consistency.test_consistency_d_dw(problem)`

`sfepy.tests.test_term_consistency.test_ev_div(problem)`

`sfepy.tests.test_term_consistency.test_ev_grad(problem)`

`sfepy.tests.test_term_consistency.test_eval_matrix(problem)`

`sfepy.tests.test_term_consistency.test_surface_evaluate(problem)`

`sfepy.tests.test_term_consistency.test_vector_matrix(problem)`

**sfepy.tests.test_term_sensitivity module**

`sfepy.tests.test_term_sensitivity.modify_mesh(val, spbox, dv_mode, cp_pos)`

`sfepy.tests.test_term_sensitivity.problem()`

`sfepy.tests.test_term_sensitivity.test_sensitivity(problem)`

**sfepy.tests.test_units module**

`sfepy.tests.test_units.test_consistent_sets()`

`sfepy.tests.test_units.test_units()`

**sfepy.tests.test_volume module**

Test computing volumes by volume or surface integrals.

`sfepy.tests.test_volume.problem()`

`sfepy.tests.test_volume.test_volume(problem)`

`sfepy.tests.test_volume.test_volume_tl(problem)`

**Tools**

**tools/build_helpers.py script**

Build helpers for setup.py.
Notes

The original version of this file was adapted from NiPy project [1].

class build_helpers.Clean(dist, **kw)
    Command class to clean, enhanced to clean also files generated during python setup.py build_ext –inplace.
    run()
        After calling the super class implementation, this function removes the directories specific to scikit-build.

class build_helpers.DoxygenDocs(dist, **kw)
    description = 'generate docs by Doxygen'
    run()
        A command’s raison d’etre: carry out the action it exists to perform, controlled by the options initialized in ‘initialize_options()’, customized by other commands, the setup script, the command-line, and config files, and finalized in ‘finalize_options()’. All terminal output and filesystem interaction should be done by ‘run()’.
        This method must be implemented by all command classes.

class build_helpers.NoOptionsDocs(dist, **kw)
    finalize_options()
        Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if ‘foo’ depends on ‘bar’, then it is safe to set ‘foo’ from ‘bar’ as long as ‘foo’ still has the same value it was assigned in ‘initialize_options()’.
        This method must be implemented by all command classes.

    initialize_options()
        Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, ‘initialize_options()’ implementations are just a bunch of “self.foo = None” assignments.
        This method must be implemented by all command classes.

    user_options = [({'None', None, 'this command has no options'})]

class build_helpers.SphinxHTMLDocs(dist, **kw)
    description = 'generate html docs by Sphinx'
    run()
        A command’s raison d’etre: carry out the action it exists to perform, controlled by the options initialized in ‘initialize_options()’, customized by other commands, the setup script, the command-line, and config files, and finalized in ‘finalize_options()’. All terminal output and filesystem interaction should be done by ‘run()’.
        This method must be implemented by all command classes.

class build_helpers.SphinxPDFDocs(dist, **kw)
    description = 'generate pdf docs by Sphinx'
**run()**

A command’s raison d’etre: carry out the action it exists to perform, controlled by the options initialized in ‘initialize_options()’, customized by other commands, the setup script, the command-line, and config files, and finalized in ‘finalize_options()’. All terminal output and filesystem interaction should be done by ‘run()’.

This method must be implemented by all command classes.

```python
build_helpers.get_sphinx_make_command()
build_helpers.have_good_cython()
build_helpers.package_check(pkg_name, version=None, optional=False, checker=<function parse_version>, version_getter=None, messages=None, show_only=False)
```

Check if package *pkg_name* is present, and in correct version.

**Parameters**

- **pkg_name**
  - [str or sequence of str] The name of the package as imported into python. Alternative names (e.g. for different versions) may be given in a list.

- **version**
  - [str, optional] The minimum version of the package that is required. If not given, the version is not checked.

- **optional**
  - [bool, optional] If False, raise error for absent package or wrong version; otherwise warn

- **checker**
  - [callable, optional] If given, the callable with which to return a comparable thing from a version string. The default is `pkg_resources.parse_version`.

- **version_getter**
  - [callable, optional:] If given, the callable that takes *pkg_name* as argument, and returns the package version string - as in:

    ```
    version = version_getter(pkg_name)
    ```

    The default is equivalent to:

    ```
    mod = __import__(pkg_name); version = mod.__version__
    ```

- **messages**
  - [dict, optional] If given, the dictionary providing (some of) output messages.

- **show_only**
  - [bool] If True, do not raise exceptions, only show the package name and version information.

```python
build_helpers.recursive_glob(top_dir, pattern)
```

Utility function working like `glob.glob()`, but working recursively and returning generator.

**Parameters**

- **topdir**
  - [str] The top-level directory.

- **pattern**
  - [str or list of str] The pattern or list of patterns to match.
tools/gen_gallery.py script

Generate the images and rst files for gallery of SfePy examples.

The following steps need to be made to regenerate the documentation with the updated example files:

1. remove doc/examples/*:

   $ rm -rf doc/examples/*

2. generate the files:

   $ ./tools/gen_gallery.py

3. regenerate the documentation:

   $ python setup.py htmldocs

gen_gallery.apply_view_options(views, default)

gen_gallery.ebase2fbase(ebase)

gen_gallery.generate_gallery(examples_dir, output_filename, doc_dir, rst_dir, thumbnails_dir, dir_map, n_col=3)

Generate the gallery rst file with thumbnail images and links to examples.

Parameters

output_filename
  [str] The output rst file name.

doc_dir
  [str] The top level directory of gallery files.

rst_dir
  [str] The full path to rst files of examples within doc_dir.

thumbnails_dir
  [str] The full path to thumbnail images within doc_dir.

dir_map
  [dict] The directory mapping returned by generate_rst_files()

n_col
  [int] The number of columns in the gallery table.

gen_gallery.generate_images(images_dir, examples_dir, pattern='*.py')

Generate images from results of running examples found in examples_dir directory.

The generated images are stored to images_dir,

gen_gallery.generate_rst_files(rst_dir, examples_dir, images_dir, pattern='*.py')

Generate Sphinx rst files for examples in examples_dir with images in images_dir and put them into rst_dir.

Returns

dir_map
  [dict] The directory mapping of examples and corresponding rst files.

gen_gallery.generate_thumbnails(thumbnails_dir, images_dir, scale=0.3)

Generate thumbnails into thumbnails_dir corresponding to images in images_dir.
gen_gallery.main()
gen_gallery.resview_plot(filename, filename_out, options)
gen_gallery.run_resview_plot(*args)
    A fix for the problem that calling resview_plot() directly often terminates the program.

tools/gen_legendre_simplex_base.py script

Generate simplex legendre 2D basis coefficients and exponents matrices and save them to legendre2D_simplex_coefs.txt and legendre2D_simplex_expos.txt
gen_legendre_simplex_base.main()

tools/gen_lobatto1d_c.py script

Generate lobatto1d.c and lobatto1h.c files.
gen_lobatto1d_c.append_declarations(out, cpolys, comment, cvar_name, shift=0)
gen_lobatto1d_c.append_lists(out, names, length)
gen_lobatto1d_c.append_polys(out, cpolys, comment, cvar_name, var_name='x', shift=0)
gen_lobatto1d_c.gen_lobatto(max_order)
gen_lobatto1d_c.main()
gen_lobatto1d_c.plot_polys(fig, polys, var_name='x')

tools/gen_release_notes.py script

Generate release notes using git log starting from the given version.
gen_release_notes.main()

tools/gen_serendipity_basis.py script

python3 tools/gen_serendipity_basis.py > sfepy/discrete/fem/_serendipity.py
gen_serendipity_basis.main()

tools/gen_solver_table.py script

Generate available solvers table for ReST documentation.
gen_solver_table.gen_solver_table(app)
gen_solver_table.main()
gen_solver_table.setup(app)
gen_solver_table.trim(docstring)
    Trim and split (doc)string.

gen_solver_table.typeset(fd)
    Utility function called by Sphinx.

gen_solver_table.typeset_solvers_table(fd, solver_table)
    Generate solvers table ReST output.

**tools/gen_term_table.py script**

Generate the table of all terms for the sphinx documentation.

gen_term_table.create_parser(slist, current_section)
gen_term_table.format_next(text, new_text, pos, can_newline, width, ispaces)
gen_term_table.gen_term_table(app)
gen_term_table.get_examples(table)
gen_term_table.main()
gen_term_table.set_section(sec)
gen_term_table.setup(app)
gen_term_table.to_list(slist, sec)
gen_term_table.typeset(filename)
    Utility function called by sphinx.
gen_term_table.typeset_examples(term_class, term_use)
gen_term_table.typeset_term_syntax(term_class)
gen_term_table.typeset_term_table(fd, keys, table, title)
    Terms are sorted by name without the d*_ prefix.
gen_term_table.typeset_term_tables(fd, table)
    Generate tables: basic, sensitivity, special.
gen_term_table.typeset_to_indent(txt, indent0, indent, width)

**tools/install_data.py script**

class install_data.install_data(dist)

    finalize_options()
        Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if ‘foo’ depends on ‘bar’, then it is safe to set ‘foo’ from ‘bar’ as long as ‘foo’ still has the same value it was assigned in ‘initialize_options()’.

        This method must be implemented by all command classes.
tools/show_authors.py script

show_authors.main()

tools/show_terms_use.py script

Show terms use in problem description files in the given directory.
show_terms_use.main()

tools/sync_module_docs.py script

Synchronize the documentation files in a given directory doc_dir with the actual state of the SfePy sources in top_dir. Missing files are created, files with no corresponding source file are removed, other files are left untouched.

Notes

The developer guide needs to be edited manually to reflect the changes.

sync_module_docs.main()


PYTHON MODULE INDEX

b
build_helpers, 588

g
gen_gallery, 591
gen_legendre_simplex_base, 592
gen_lobattoid_c, 592
gen_release_notes, 592
gen_serendipity_basis, 592
gen_solver_table, 592
gen_term_table, 593

i
install_data, 593

S
sfepy.applications.application, 160
sfepy.applications.evp_solver_app, 160
sfepy.applications.pde_solver_app, 161
sfepy.base.base, 162
sfepy.base.compat, 168
sfepy.base.conf, 172
sfepy.base.getch, 174
sfepy.base.options, 175
sfepy.base.iutils, 175
sfepy.base.log, 180
sfepy.base.log_plotter, 182
sfepy.base.mem_usage, 182
sfepy.base.multproc, 183
sfepy.base.multproc_mpi, 183
sfepy.base.multproc_proc, 186
sfepy.base.parse_conf, 187
sfepy.base.plotutils, 188
sfepy.base.reader, 188
sfepy.base.resolve_deps, 188
sfepy.base.testing, 189
sfepy.base.timing, 189
sfepy.config, 159
sfepy.discrete.common.dof_info, 237
sfepy.discrete.common.domain, 240
sfepy.discrete.common.extmods.fmfield, 241
sfepy.discrete.common.extmods.geommech, 241
sfepy.discrete.common.extmods.assemble, 241
sfepy.discrete.common.extmods.cmapping, 241
sfepy.discrete.common.extmods.cmesh, 241
sfepy.discrete.common.extmods.crefcoors, 244
sfepy.discrete.common.fields, 245
sfepy.discrete.common.global_interp, 248
sfepy.discrete.common.mappings, 251
sfepy.discrete.common.poly_spaces, 253
sfepy.discrete.common.region, 254
sfepy.discrete.conditions, 190
sfepy.discrete.dg.dg_1D_visualizer, 293
sfepy.discrete.dg.fields, 298
sfepy.discrete.dg.limiters, 310
sfepy.discrete.dg.poly_spaces, 307
sfepy.discrete.equations, 192
sfepy.discrete.evaluate, 198
sfepy.discrete.evaluate_variable, 202
sfepy.discrete.fem._serendipity, 291
sfepy.discrete.fem.domain, 257
sfepy.discrete.fem.extmods.bases, 258
sfepy.discrete.fem.extmods.lobatto_bases, 259
sfepy.discrete.fem.facets, 259
sfepy.discrete.fem.fe_surface, 261
sfepy.discrete.fem.fields_base, 261
sfepy.discrete.fem.fields_hierarchic, 267
sfepy.discrete.fem.fields_12, 267
sfepy.discrete.fem.fields_nodal, 269
sfepy.discrete.fem.fields_positive, 271
sfepy.discrete.fem.geometry_element, 271
sfepy.discrete.fem.history, 272
sfepy.discrete.fem.lcbc_operators, 272
sfepy.discrete.fem.linearizer, 275
sfepy.discrete.fem.mappings, 275
sfepy.discrete.fem.mesh, 277
sfepy.discrete.fem.meshio, 279
sfepy.discrete.fem.periodic, 286
sfepy.discrete.fem.poly_spaces, 287
sfepy.discrete.fem.refine, 290
sfepy.discrete.fem.refine_hanging, 291
sfepy.discrete.fem.utils, 291
sfepy.discrete.functions, 202
sfepy.discrete.iga.domain, 313
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfepy.terms.terms_jax</td>
<td>518</td>
</tr>
<tr>
<td>sfepy.terms.terms_mass</td>
<td>525</td>
</tr>
<tr>
<td>sfepy.terms.terms_membrane</td>
<td>526</td>
</tr>
<tr>
<td>sfepy.terms.terms_multilinear</td>
<td>527</td>
</tr>
<tr>
<td>sfepy.terms.terms_navier_stokes</td>
<td>539</td>
</tr>
<tr>
<td>sfepy.terms.terms_piezo</td>
<td>548</td>
</tr>
<tr>
<td>sfepy.terms.terms_point</td>
<td>551</td>
</tr>
<tr>
<td>sfepy.terms.terms_sensitivity</td>
<td>552</td>
</tr>
<tr>
<td>sfepy.terms.terms_shells</td>
<td>558</td>
</tr>
<tr>
<td>sfepy.terms.terms_surface</td>
<td>560</td>
</tr>
<tr>
<td>sfepy.terms.terms_th</td>
<td>565</td>
</tr>
<tr>
<td>sfepy.terms.terms_volume</td>
<td>565</td>
</tr>
<tr>
<td>sfepy.terms.utils</td>
<td>566</td>
</tr>
<tr>
<td>sfepy.tests.conf-test</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_assembling</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_base</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_cmesh</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_conditions</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_declarative_examples</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_dg_field</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_dg_terms_calls</td>
<td>578</td>
</tr>
<tr>
<td>sfepy.tests.test_domain</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_ed_solvers</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_eigenvalue_solvers</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_elasticity_small_strain</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_fem</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_functions</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_high_level</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_homogenization_engine</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_homogenization_perfusion</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_hyperelastic_tlul</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_i0</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_laplace_unit_disk</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_laplace_unit_square</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_lcbcs</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linalg</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linear_solvers</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linearization</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_log</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_matcoefs</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_expand</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_generators</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_interp</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_smoothing</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_meshio</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_msm_laplace</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_msm_symbolic</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_normals</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_parsing</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_poly_spaces</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_projections</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_quadratures</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_ref_coors</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_refine_hanging</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_regions</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_semismooth_newton</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_sparse</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_splinebox</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_tensors</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_term_call_modes</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_term_consistency</td>
<td>588</td>
</tr>
<tr>
<td>sfepy.tests.test_term_sensitivity</td>
<td>588</td>
</tr>
<tr>
<td>sfepy.tests.test_term_volume</td>
<td>588</td>
</tr>
<tr>
<td>sfepy.version</td>
<td>160</td>
</tr>
<tr>
<td>show_authors</td>
<td>594</td>
</tr>
<tr>
<td>show_terms_use</td>
<td>594</td>
</tr>
<tr>
<td>sync_module_docs</td>
<td>594</td>
</tr>
</tbody>
</table>
Symbols

__call__() (sfepy.solvers.nls.Newton method), 426
__call__() (sfepy.solvers.nls.PETScNonlinearSolver method), 427
__call__() (sfepy.solvers.nls.ScipyBroyden method), 428
__init__() (sfepy.discrete.materials.Material method), 204
__init__() (sfepy.solvers.nls.Newton method), 426
__init__() (sfepy.solvers.nls.PETScNonlinearSolver method), 427
__init__() (sfepy.solvers.nls.ScipyBroyden method), 428
__module__ (sfepy.solvers.nls.Newton attribute), 426
__module__ (sfepy.solvers.nls.PETScNonlinearSolver attribute), 427
__module__ (sfepy.solvers.nls.ScipyBroyden attribute), 428

A

a (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 406
a (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 409
a (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 413
a (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 416
a (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 420
a_elt (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 406
a_elt (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 409
a_elt (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 413
a_elt (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 416
a_elt (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 420
a_loc (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 406
Index
arg_types (sfepy.terms.termsargin_types (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm
attribute), 479
attribute), 479
arg_types (sfepy.terms.terms_diffusion.AdvectDivFreeTermarg_types (sfepy.terms.terms_elastic.LinearElasticTerm
attribute), 480
attribute), 480
arg_types (sfepy.terms.terms_diffusion.ConvectVGradSTermarg_types (sfepy.terms.terms_elastic.LinearElasticTHTerm
attribute), 481
attribute), 481
arg_types (sfepy.terms.terms_diffusion.DiffusionCoupling arg_types (sfepy.terms.terms_elastic.LinearPrestressTerm
attribute), 482
attribute), 482
arg_types (sfepy.terms.terms_diffusion.DiffusionVelocityTarg_types (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm
attribute), 483
attribute), 483
arg_types (sfepy.terms.terms_diffusion.LaplaceTerm attribute), 484
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 484
attribute), 484
attribute), 484
arg_types (sfepy.terms.terms_diffusion.SDDiffusionTerm arg_types (sfepy.terms.terms_elastic.NonsymElasticIsotropicTerm
attribute), 485
attribute), 485
arg_types (sfepy.terms.terms_diffusion.SurfaceFluxOperatorTerm attribute), 485
arg_types (sfepy.terms.terms_elastic.NonsymElasticTerm
attribute), 485
arg_types (sfepy.terms.terms_elastic.LinearPrestressTerm
attribute), 500
arg_types (sfepy.terms.terms_elastic.LinearStrainFiberTerm
attribute), 500
arg_types (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm
attribute), 501
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 502
arg_types (sfepy.terms.terms_elastic.LinearStrainFiberTerm
attribute), 502
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 503
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 504
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 505
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 506
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 507
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 508
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 508
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 509
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 510
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 511
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 512
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 513
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 514
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 514
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 515
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 516
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 517
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 519
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 520
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 520
arg_types (sfepy.terms.terms_elastic.LinearTrussTerm
attribute), 521
### Index

<table>
<thead>
<tr>
<th>Argument Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg_types</code></td>
<td>(sfepy.terms.terms_jax.OgdenTLADTerm attribute), 522</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_mass.MassTerm attribute), 525</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_membrane.TLMembraneTerm attribute), 526</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ECauchyStressTerm attribute), 527</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EConvectTerm attribute), 527</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EDiffusionTerm attribute), 528</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EDivGradTerm attribute), 529</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EDivTerm attribute), 529</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EDotTerm attribute), 530</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EGradTerm attribute), 530</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELaplaceTerm attribute), 531</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELinearConveTerm attribute), 532</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELinearConvectTerm attribute), 533</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELinearElasticTerm attribute), 533</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELinearTractionTerm attribute), 534</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELNonPenetrationTerm attribute), 534</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ELNonSymElasticTerm attribute), 535</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.EStokesTerm attribute), 536</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_multilinear.ESurfaceFluxOperatorTerm attribute), 538</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_piezo.PiezozCouplingTerm attribute), 540</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_piezo.PiezozStressTerm attribute), 540</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDStokesTerm attribute), 544</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDPiezozCouplingTerm attribute), 545</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDDiffusionTerm attribute), 547</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDDivGradTerm attribute), 550</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDdotTerm attribute), 553</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDLinearElasticTerm attribute), 554</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDLinearTractionTerm attribute), 555</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_sensitivity.ESDPiezozCouplingTerm attribute), 556</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms.surfaceshells.Shell10XTerm attribute), 557</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.ContactPlaneTerm attribute), 560</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.ContactSphereTerm attribute), 562</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.LayerTerm attribute), 562</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.SDLinearTractionTerm attribute), 563</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.SDSurfaceIntegrateTerm attribute), 563</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.SDNormalTractionTerm attribute), 564</td>
</tr>
<tr>
<td></td>
<td>(sfepy.terms.terms_surface.SurfaceJumpTerm attribute), 565</td>
</tr>
</tbody>
</table>
arg_types (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 566
arg_types (sfepy.terms.terms_volume.NonlinearVolumeForceTerm attribute), 566
argsort_rows() (in module sfepy.linalg.utils), 355
as_dict() (sfepy.base.base.Container method), 162
as_float_or_complex() (in module sfepy.base.base), 164
assemble1d() (in module sfepy.base.base), 164
assemble_mtx_to_petsc() (in module sfepy.linalg.utils), 355
assemble_contact_residual_and_stiffness() (in module sfepy.mechanics.extmods.ccontres), 372
assemble_matrix() (in module sfepy.discrete.common.exmodsexamine), 241
assemble_matrix_complex() (in module sfepy.discrete.common.exmodsexamine), 241
assemble_matrix_complex() (in module sfepy.discrete.common.exmodsexamine), 241
assemble_mtx_to_petsc() (in module sfepy.parallel.parallel), 386
assemble_rhs_to_petsc() (in module sfepy.parallel.parallel), 386
assemble_to() (sfepy.terms.terms.Term method), 447
assemble_vector() (in module sfepy.discrete.common.exmodsexamine), 241
assemble_vector_complex() (in module sfepy.discrete.common.exmodsexamine), 241
assert() (in module sfepy.base.base), 164
assert_equal() (in module sfepy.base.base.testing), 189
assign_args() (sfepy.terms.terms.Term method), 447
assign_args() (sfepy.terms.terms.Term method), 451
assign_standard_hooks() (in module sfepy.applications.pde_solver_app), 161
AutoDirect (class in sfepy.solvers.auto_fallback), 395
AutoFallbackSolver (class in sfepy.solvers.auto_fallback), 395
AutoIterative (class in sfepy.solvers.auto_fallback), 395
aux (sfepy.solvers.ls_mumps.mumps_struc_c_x attribute), 424
average_qp_to_vertices() (sfepy.discrete.fem.fields_base.FEField method), 261
average_to_vertices() (sfepy.discrete.fem.fields_nodal.H1DiscontinuousField method), 269
average_vertex_var_in_cells() (in module sfepy.postprocess.time_history), 392

B
BandGaps (class in sfepy.homogenization.coefs_phononic),
CoefNonSym (class in sfepy.homogenization.coefs_base), 331
CoefNonSymNonSym (class in sfepy.homogenization.coefs_base), 331
CoefOne (class in sfepy.homogenization.coefs_base), 331
CoefRegion (class in sfepy.homogenization.coefs_perfusion), 331
CoefSum (class in sfepy.homogenization.coefs_base), 332
CoefSym (class in sfepy.homogenization.coefs_base), 332
CoefSymSym (class in sfepy.homogenization.coefs_base), 332
CoefVolume (class in sfepy.homogenization.engine), 340
collect_conn_info () (sfepy.discrete.equations.Equation method), 192
collect_conn_info () (sfepy.discrete.equations.Equations method), 193
collect_materials () (sfepy.discrete.equations.Equation method), 192
collect_materials () (sfepy.discrete.equations.Equations method), 193
collect_modifiers () (in module sfepy/terms/terms_multilinear), 539
collect_term () (in module sfepy.discrete.parse_equations), 207
collect_variables () (sfepy.discrete.equations.Equation method), 192
collect_variables () (sfepy.discrete.equations.Equations method), 193
colsca (sfepy.solvers.ls_mumps.mumps_struc_c_4.attribute), 406
colsca (sfepy.solvers.ls_mumps.mumps_struc_c_5_0.attribute), 409
colsca (sfepy.solvers.ls_mumps.mumps_struc_c_5_1.attribute), 413
colsca (sfepy.solvers.ls_mumps.mumps_struc_c_5_2.attribute), 416
colsca (sfepy.solvers.ls_mumps.mumps_struc_c_5_3.attribute), 420
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_tl.BulkPressureTLTerm.method), 508
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 516
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 515
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 515
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 515
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 515
colsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_computex.data) (sfepy/terms/terms_hyperelastic_uf.BulkPressureULTerm.method), 515
combine() (in module sfepy.linalg.util), 355
combine_bezier_extraction() (in module sfepy.discrete.iga.iga), 319
combine_scalar_grad() (in module sfepy.homogenization.recovery), 344
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_4.attribute), 406
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_5_0.attribute), 410
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_5_1.attribute), 413
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_5_2.attribute), 416
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_5_3.attribute), 420
comm_fortran (sfepy.solvers.ls_mumps.mumps_struc_c_x.attribute), 424
compare_mesh() (in module sfepy/tests/test_domain), 579
compare_scalars() (in module sfepy/tests/test_homogenization_perfusion), 581
compare_vectors() (in module sfepy/base/testing), 189
compose_flags() (sfepy/config/Config method), 159
compose_sparse() (in module sfepy/linalg/sparse), 353
compose_system_compile_flags() (in module sfepy/config), 160
ComposedLimiter (class in sfepy/discrete/dg/limiters), 310
CompressibilityULTerm (class in sfepy/terms/terms_hyperelastic_uf), 516
compute_bezier_control() (in module sfepy/discrete/iga.iga), 319
compute_bezier_extraction() (in module sfepy/discrete/iga.iga), 320
compute_bezier_extraction lid() (in module sfepy/discrete/iga.iga), 320
compute_cat_dim_dim() (in module sfepy/homogenization/coefs/phononic), 337
compute_cat_dim_sym() (in module sfepy/homogenization/coefs/phononic), 337
compute_cat_sym_sym() (in module sfepy/homogenization/coefs/phononic), 337
compute_correction() (sfepy/homogenization/coefs_base/TCorrectorsViaPressureEVP method), 333
compute_eigenmomenta() (in module sfepy/homogenization/coefs/phononic), 337
compute_fibre_strain() (in module sfepy/terms/terms_fibres), 505
compute_jacobian() (sfepy.solvers.semismooth_newton/SemismoothNewton method), 433

Index
Index
Index
dw_electric_source() (in module sfepy.terms.extmods.terms), 569  
dw_fun() (sfepy.terms.terms_diffusion.DiffusionCoupling static method), 481  
dw_fun() (sfepy.terms.terms_dot.ScalarDotGradIScalarTerm static method), 490  
dw_fun() (sfepy.terms.terms_surface.SurfaceNormalDotTerm static method), 564  
dw_grad() (in module sfepy.terms.extmods.terms), 569  
dw_he_rtm() (in module sfepy.terms.extmods.terms), 569  
dw_laplace() (in module sfepy.terms.extmods.terms), 569  
dw_lin_conve() (in module sfepy.terms.extmods.terms), 569  
dw_lin_elastic() (in module sfepy.terms.extmods.terms), 569  
dw_lin_prestress() (in module sfepy.terms.extmods.terms), 569  
dw_lin_strain_fib() (in module sfepy.terms.extmods.terms), 569  
dw_nonsym_elastic() (in module sfepy.terms.extmods.terms), 569  
dw_piezoe() (in module sfepy.terms.extmods.terms), 569  
dw_st_adj1_supg_p() (in module sfepy.terms.extmods.terms), 569  
dw_st_adj1_supg_c() (in module sfepy.terms.extmods.terms), 569  
dw_st_adj2_supg_p() (in module sfepy.terms.extmods.terms), 569  
dw_st_asp() (in module sfepy.terms.extmods.terms), 569  
dw_st_grad_div() (in module sfepy.terms.extmods.terms), 569  
dw_st_pspg_c() (in module sfepy.terms.extmods.terms), 569  
dw_st_supp() (in module sfepy.terms.extmods.terms), 569  
dw_st_supp_p() (in module sfepy.terms.extmods.terms), 569  
dw_surface_flux() (in module sfepy.terms.extmods.terms), 569  
dw_surface_ltrim() (in module sfepy.terms.extmods.terms), 569  
dw_surface_s_v_dot_n() (in module sfepy.terms.extmods.terms), 569  
dw_surface_v_dot_n() (in module sfepy.terms.extmods.terms), 569  
dw_surface_v_dot_n_s() (in module sfepy.terms.extmods.terms), 569  
dw_tl_diffusion() (in module sfepy.terms.extmods.terms), 569  
dw_tl_surface_traction() (in module sfepy.terms.extmods.terms), 569  
dw_tl_volume() (in module sfepy.terms.extmods.terms), 569  
dw_v_dot_grad_s_sw() (in module sfepy.terms.extmods.terms), 569  
dw_v_dot_grad_s_vw() (in module sfepy.terms.extmods.terms), 569  
dw_volume_dot_scalar() (in module sfepy.terms.extmods.terms), 569  
dw_volume_dot_vector() (in module sfepy.terms.extmods.terms), 569  
dw_volume_lvf() (in module sfepy.terms.extmods.terms), 569  
ed_coors_max() (class in sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), 258  
ed_coors_max() (class in sfepy.discrete.iga.extmods.igac.CNURBSContext attribute), 315  
ebase2fbase() (in module gen_gallery), 591  
ebc() (in module sfepy.tests.test_msm_laplace), 584  
ebc() (in module sfepy.tests.test_msm_symbolic), 584  
ECauchyStressTerm (class in module sfepy.terms.extmods.terms), 527  
EConve() (class in module sfepy.terms.extmods.terms), 527  
edge_oris() (class in sfepy.discrete.common.mods.cmsh.CMesh attribute), 242  
EdgeDirectionOperator (class in module sfepy.discrete.fem.lcbc_operators), 272  
edges (class in module sfepy.discrete.common.region.Region property), 254  
EDiffu() (class in module sfepy.terms.extmods.terms), 528  
edict() (class in module sfepy.base.conf.ProblemConf method), 172  
edict_strings() (in module sfepy.base.base), 165  
edict_filename() (in module sfepy.base.ioutils), 177  
edict_tuple_strings() (in module sfepy.base.base), 165  
EDivGradTerm (class in module sfepy.terms.extmods.terms), 528  
EDivTerm (class in module sfepy.terms.extmods.terms), 529  
EDotTerm (class in module sfepy.terms.extmods.terms), 529  
EGradTerm (class in module sfepy.terms.extmods.terms), 530  
eig() (in module sfepy.solvers.eigen), 398  
Eigenmomenta (class in module sfepy.homogenization.coefs_phononic), 336  
EigenvalueSolver (class in module sfepy.solvers.solvers), 433  
IntegrateOperatorTerm (class in module sfepy.terms.extmods.terms), 530  
ELaplaceTerm (class in module sfepy.terms.extmods.terms), 531  
ElasticConstants (class in module sfepy.mechanics.matcoefs), 359  
ElasticWaveCauchyTerm (class in module sfepy.terms.extmods.terms), 495
ElasticWaveTerm (class in sfepy/terms/terms_elastic), 496
ElastodynamicsBaseTS (class in sfepy/solvers/ts_solvers), 442
ElastodynamicsBasicTSC (class in sfepy/solvers/ts_controllers), 436
ElastodynamicsLinearTSC (class in sfepy/solvers/ts_controllers), 437
ElastodynamicsPIDTSC (class in sfepy/solvers/ts_controllers), 438
ElectricSourceTerm (class in sfepy/terms/terms_electric), 503
elems_q2t() (in module sfepy/mesh/mesh_tools), 383
elevate() (sfepy/discrete/iga/domain/NurbsPatch method), 313
ELinearConvectTerm (class in sfepy/terms/terms_multilinear), 532
ELinearElasticTerm (class in sfepy/terms/terms_multilinear), 532
ELinearTractionTerm (class in sfepy/terms/terms_multilinear), 533
eltptr (sfepy/solvers ls_mumps.mumps_struc_c_4 attribute), 407
eltptr (sfepy/solvers ls_mumps.mumps_struc_c_5_0 attribute), 410
eltptr (sfepy/solvers ls_mumps.mumps_struc_c_5_1 attribute), 413
eltptr (sfepy/solvers ls_mumps.mumps_struc_c_5_2 attribute), 417
eltptr (sfepy/solvers ls_mumps.mumps_struc_c_5_3 attribute), 421
eltvar (sfepy/solvers ls_mumps.mumps_struc_c_4 attribute), 407
eltvar (sfepy/solvers ls_mumps.mumps_struc_c_5_0 attribute), 410
eltvar (sfepy/solvers ls_mumps.mumps_struc_c_5_1 attribute), 413
eltvar (sfepy/solvers ls_mumps.mumps_struc_c_5_2 attribute), 417
eltvar (sfepy/solvers ls_mumps.mumps_struc_c_5_3 attribute), 421
emit() (sfepy/base/multiproc_mpi.MPIFileHandler method), 184
eqcon() (in module sfepy/base/ioutils), 177
ENonPenetrationPenaltyTerm (class in sfepy/terms/terms_multilinear), 533
ENonSymElasticTerm (class in sfepy/terms/terms_multilinear), 534
ensure_path() (in module sfepy/base/ioutils), 177
entities (sfepy/discrete/common.extmods.cmesh.CMesh attribute), 242
enum() (in module sfepy/base/multiproc_mpi), 185
Equation (class in sfepy/discrete.equations), 192
equation_mapping() (sfepy/discrete.variables.FieldVariable method), 228
EquationMap (class in sfepy/discrete.common.dof_info), 234
Equations (class in sfepy/discrete.equations), 238
errclear() (in module sfepy/terms/terms_extmods/terms), 569
EScalarDotMGradScalarTerm (class in sfepy/terms/terms_multilinear), 535
ESDiffusionTerm (class in sfepy/terms/terms_sensitivity), 552
ESDivGradTerm (class in sfepy/terms/terms_sensitivity), 553
ESDotTerm (class in sfepy/terms/terms_sensitivity), 553
ESLinearElasticTerm (class in sfepy/terms/terms_sensitivity), 554
ESLinearElasticTerm (class in sfepy/terms/terms_sensitivity), 555
ESDiffusionTerm (class in sfepy/terms/terms_sensitivity), 555
ESDiffusionTerm (class in sfepy/terms/terms_sensitivity), 556
ESDVectorDotGradScalarTerm (class in sfepy/terms/terms_sensitivity), 557
EssentialBC (class in sfepy/discrete.conditions), 190
ESStokesTerm (class in sfepy/terms/terms_multilinear), 535
ETermBase (class in sfepy/terms/terms_multilinear), 536
ETermBase (class in sfepy/terms/terms_th), 565
EulerStepSolver (class in sfepy/solvers/ts_dg_solvers), 312
eval() (sfepy/mesh.bspline.BSpline method), 373
eval() (sfepy/mesh.bspline.BSplineSurf method), 375
eval_alpha_ogden() (in module sfepy/terms/terms_jax), 522
eval_base() (sfepy/discrete.common.poly_spaces.PolySpace method), 253
eval_basis() (sfepy/mesh.bspline.BSpline method), 374
eval_bernstein_basis() (in module sfepy/discrete.iga.extmods.iga), 315
eval_bernstein_basis() (in module sfepy/discrete.iga.iga), 321
eval_complex() (in module sfepy/discrete.evaluate_variable), 202
eval_complex() (class in sfepy/terms/terms_multilinear.ETermBase method), 448
eval_complex() (class in sfepy/terms/terms_multilinear.ETermBase method), 537
eval_coer_expression() (in module sfepy/base/testing), 189
eval_density_mass() (in module sfepy/terms/terms_fax), 522
eval_elasticity_l() (in module sfepy/terms/terms_fax), 522
Function (class in sfepy.discrete.functions), 202
function() (sfepy.terms.terms_adj_navier_stokes.AdjConvectTerm function), 452
function() (sfepy.terms.terms_adj_navier_stokes.AdjConvectTerm static method), 452
function() (sfepy.terms.terms_adj_navier_stokes.AdjConvectTerm function), 452
function() (sfepy.terms.terms_contact.ContactTerm function), 473
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact.ContactTerm static method), 474
function() (sfepy.terms.terms_contact.ContactTerm function), 474
function() (sfepy.terms.terms_contact>ContactTerm static method), 474
function() (sfepy.terms.terms_electric.ElectricSourceTerm static method), 504
function() (sfepy.terms.terms_hyperelastic_base.DeformationTerm static method), 505
function() (sfepy.terms.terms_hyperelastic_base.HyperElasticBase static method), 506
function() (sfepy.terms.terms_hyperelastic_base.HyperElasticStressTerm static method), 507
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 510
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 511
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 512
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 513
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 514
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 515
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 516
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 517
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 518
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 519
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 520
function() (sfepy.terms.terms_hyperelastic_base.VolumeSurfaceTerm static method), 521
function() (sfepy.terms.terms_jax.LinearElasticLADTerm static method), 522
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 523
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 524
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 525
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 526
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 527
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 528
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 529
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 530
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 531
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 532
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 533
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 534
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 535
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 536
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 537
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 538
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 539
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 540
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 541
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 542
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 543
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 544
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 545
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 546
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 547
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 548
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 549
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 550
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 551
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 552
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 553
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 554
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 555
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 556
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 557
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 558
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 559
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 560
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 561
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 562
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 563
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 564
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 565
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 566
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 567
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 568
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 569
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 570
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 571
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 572
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 573
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 574
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 575
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 576
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 577
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 578
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 579
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 580
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 581
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 582
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 583
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 584
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 585
function() (sfepy.terms.terms_jax.LinearElasticPADTerm static method), 586

G:

...
get_state_in_region()
(sfname: discrete: variables: FieldVariable method), 231
get_state_names()
(sfname: terms: terms: Term method), 450
get_state_parts()
(sfname: discrete: variables: Variables method), 234
get_state_variables()
(sfname: terms: terms: Term method), 450
get_str()
(sfname: terms: terms: Term method), 450
get_strain()
in module sfepy: terms: terms_jax, 524
get_stress()
in module sfepy: terms: terms_jax, 525
get_subdict()
in module sfepy: base: base, 166
get_subset_info()
(sfname: discrete: common: dof_info: DofInfo method), 238
get_surface_basis()
(sfname: discrete: fem: fields: nodal: GlobalNodalLikeBasis method), 269
get_surface_degrees()
in module sfepy: discrete: iga: fields: IGField, 319
get_surface_entities()
get_surface_faces()
in module sfepy: mesh: mesh_tools, 383
get_surface_facets()
(sfname: discrete: fem: mesh: meshio: MeshIO method), 284
get_sym_indices()
in module sfepy: mechanics: tensors, 367
get_t4_from_t2s()
in module sfepy: mechanics: tensors, 367
get_tangent_stress_matrix()
in module sfepy: mechanics: membranes, 364
get_tensor_product_conn()
in module sfepy: mesh: mesh_generators, 382
get_timestepper()
(sfname: discrete: problem: Problem method), 217
get_tolerance()
(sfname: solvers: solvers: LinearSolver method), 433
get_trace()
in module sfepy: mechanics: tensors, 368
get_true_order()
(sfname: discrete: fem: fields: base: FEField method), 264
get_true_order()
(sfname: discrete: iga: fields: IGField method), 319
get_trunk()
in module sfepy: base: ioutils, 178
get_ts_val()
(sfname: homogenization: coefs: base: CorrSolution method), 533
get_tss()
(sfname: discrete: problem: Problem method), 217
get_tss_functions()
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HomogenizationApp</code></td>
<td>(sfepy.discrete.variables.Variables method), 235</td>
</tr>
<tr>
<td><code>have_good_cython()</code></td>
<td>(in module build_helpers), 590</td>
</tr>
<tr>
<td><code>HDF5BaseData</code></td>
<td>(class in sfepy.base.ioutils), 176</td>
</tr>
<tr>
<td><code>HDF5ContextManager</code></td>
<td>(class in sfepy.base.ioutils), 176</td>
</tr>
<tr>
<td><code>HDF5Data</code></td>
<td>(class in sfepy.base.ioutils), 176</td>
</tr>
<tr>
<td><code>HDF5MeshIO</code></td>
<td>(class in sfepy.discrete.fem.meshio), 282</td>
</tr>
<tr>
<td><code>HDF5XdmfMeshIO</code></td>
<td>(class in sfepy.discrete.fem.meshio), 283</td>
</tr>
<tr>
<td><code>he_eval_from_mtx()</code></td>
<td>(in module sfepy.terms.extmods.terms), 570</td>
</tr>
<tr>
<td><code>he_residuum_from_mtx()</code></td>
<td>(in module sfepy.terms.extmods.terms), 570</td>
</tr>
<tr>
<td><code>head()</code></td>
<td>(in module sfepy.discrete.dg.dg_1D_vizualizer), 294</td>
</tr>
<tr>
<td><code>Histories</code></td>
<td>(class in sfepy.discrete.fem.history), 272</td>
</tr>
<tr>
<td><code>History</code></td>
<td>(class in sfepy.discrete.fem.history), 272</td>
</tr>
<tr>
<td><code>HomogenizationApp</code></td>
<td>(class sfepy.homogenization.homogen_app), 343</td>
</tr>
<tr>
<td><code>HomogenizationEngine</code></td>
<td>(class sfepy.homogenization.engine), 340</td>
</tr>
<tr>
<td><code>HomogenizationWorker</code></td>
<td>(class sfepy.homogenization.engine), 340</td>
</tr>
<tr>
<td><code>HomogenizationWorkerMulti</code></td>
<td>(class sfepy.homogenization.engine), 341</td>
</tr>
<tr>
<td><code>HomogenizationWorkerMultiMPI</code></td>
<td>(class sfepy.homogenization.engine), 342</td>
</tr>
<tr>
<td><code>hyperelastic_mode</code></td>
<td>(sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase, 510)</td>
</tr>
<tr>
<td><code>hyperelastic_mode</code></td>
<td>(sfepy.terms.terms_hyperelastic_tl.HyperElasticTLBase, 516)</td>
</tr>
<tr>
<td><code>HyperElasticBase</code></td>
<td>(class sfepy.terms.terms_hyperelastic_base), 505</td>
</tr>
<tr>
<td><code>HyperElasticFamilyData</code></td>
<td>(class sfepy.terms.terms_hyperelastic_base), 506</td>
</tr>
<tr>
<td><code>HyperElasticSurfaceTLBase</code></td>
<td>(class sfepy.terms.terms_hyperelastic_tl), 509</td>
</tr>
<tr>
<td><code>HyperElasticSurfaceTLFamilyData</code></td>
<td>(class sfepy.terms.terms_hyperelastic_tl), 509</td>
</tr>
<tr>
<td><code>HyperElasticTLBase</code></td>
<td>(class sfepy.terms.terms_hyperelastic_tl), 510</td>
</tr>
<tr>
<td><code>HyperElasticTLFamilyData</code></td>
<td>(class sfepy.terms.terms_hyperelastic_tl), 510</td>
</tr>
<tr>
<td><code>HyperElasticULBase</code></td>
<td>(class sfepy.terms.terms_hyperelastic_ul), 516</td>
</tr>
<tr>
<td><code>HyperElasticULFamilyData</code></td>
<td>(class sfepy.terms.terms_hyperelastic_ul), 517</td>
</tr>
<tr>
<td><code>HypermeshAsciiMeshIO</code></td>
<td>(class sfepy.discrete.fem.meshio), 283</td>
</tr>
<tr>
<td><code>icntl1</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 407</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IdentityLimiter</code></td>
<td>(class in sfepy.discrete.dg.limiters), 310</td>
</tr>
<tr>
<td><code>iel</code></td>
<td>(sfepy.discrete.fem.extmods.bases.CLagrangeContext attribute), 259</td>
</tr>
<tr>
<td><code>ICDomain</code></td>
<td>(class in sfepy.discrete.iga.domain), 313</td>
</tr>
<tr>
<td><code>IGField</code></td>
<td>(class in sfepy.discrete.iga.fields), 317</td>
</tr>
<tr>
<td><code>IGMapping</code></td>
<td>(class in sfepy.discrete.iga.mappings), 326</td>
</tr>
<tr>
<td><code>import_file()</code></td>
<td>(in module sfepy.base.base), 166</td>
</tr>
<tr>
<td><code>in1d()</code></td>
<td>(in module sfepy.base.compat), 168</td>
</tr>
<tr>
<td><code>in_dir()</code></td>
<td>(in module sfepy.tests.test_mesh_interp), 583</td>
</tr>
<tr>
<td><code>IndexedStruct</code></td>
<td>(class in sfepy.base.base), 162</td>
</tr>
<tr>
<td><code>infinity_norm()</code></td>
<td>(in module sfepy.linalg.sparse), 353</td>
</tr>
<tr>
<td><code>HyperElasticSolver</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 407</td>
</tr>
<tr>
<td><code>info</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 410</td>
</tr>
<tr>
<td><code>icntl1</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 413</td>
</tr>
<tr>
<td><code>icntl1</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 417</td>
</tr>
<tr>
<td><code>icntl1</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 421</td>
</tr>
<tr>
<td><code>icntl</code></td>
<td>(sfepy.solvers.ls_mumps.mumps_struc_c_x attribute), 424</td>
</tr>
<tr>
<td><code>init()</code></td>
<td>(sfepy.mechanics.matcoefs.EulerianElasticConstants method), 360</td>
</tr>
<tr>
<td><code>init_data()</code></td>
<td>(sfepy.discrete.variables.Variable method), 232</td>
</tr>
<tr>
<td><code>init_data_struct()</code></td>
<td>(sfepy.terms.terms_hyperelastic_base.HyperElasticFamilyData, 310)</td>
</tr>
</tbody>
</table>
method), 506
init_global_search() (in module sfepy.mechanics.extmods.ccontres), 372
init_history() (sfepy.discrete.variables.Variable method), 232
init_history() (sfepy.discrete.variables.Variables method), 235
init_petsc_args() (in module sfepy.parallel.parallel), 388
init_rmm() (sfepy.solvers.ls.RMMSolver method), 402
init_slepc_args() (in module sfepy.solvers.eigen), 398
init_solvers() (sfepy.discrete.problem.Problem method), 218
init_solvers() (sfepy.homogenization.coefs_base.MiniAppBase method), 333
init_state() (sfepy.discrete.equations.Equations method), 196
init_state() (sfepy.discrete.variables.Equations method), 196
init_subproblems() (sfepy.solvers.ls.MultiProblem method), 400
init_time() (sfepy.discrete.equations.Equations method), 196
init_time() (sfepy.discrete.problem.Problem method), 218
init_vec() (in module sfepy.tests.test_conditions), 577
InitialCondition (class in sfepy.discrete.conditions), 191
initialize_options() (build_helpers.NoOptionsDocs method), 589
insert() (sfepy.base.base.Container method), 162
insert() (sfepy.terms.terms.Terms method), 451
insert_as_static_method() (in module sfepy.base.base), 166
insert_knot() (sfepy.mesh.bspline.BSpline method), 374
insert_method() (in module sfepy.base.base), 166
insert_sparse_to_csr() (in module sfepy.linalg.sparse), 354
insert_static_method() (in module sfepy.base.base), 166
insert_strided_axis() (in module sfepy.linalg.utils), 356
insert_sub_reqs() (in module sfepy.homogenization.engine), 343
install_data module, 593
install_data (class in install_data), 593
instance_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 413
instance_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 417
instance_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 421
int_dt() (sfepy.homogenization.convolutions.ConvolutionKernel method), 339
Integral (class in sfepy.discrete.integrals), 203
IntegralMeanValueOperator (class in sfepy.discrete.fem.lcbc_operators), 272
IntegralProbe (class in sfepy.discrete.probes), 208
Integrals (class in sfepy.discrete.integrals), 203
integrate() (sfepy.discrete.common.mappings.PyCMapping method), 251
integrate() (sfepy.discrete.integrals.Integral method), 203
integrate() (sfepy.terms.terms_contact.ContactTerm static method), 474
integrate() (sfepy.terms.terms_hyperelastic_base.HyperElasticBase static method), 506
integrate_along_line() (in module sfepyscripts.probe), 573
integrate_in_time() (in module sfepy.homogenization.utils), 348
IntegrateMatTerm (class in sfepy.terms.terms_basic), 462
IntegrateOperatorTerm (class in sfepy.terms.terms_basic), 462
IntegrateSurfaceMatTerm (class in sfepy.terms.terms_basic), 471
IntegrateSurfaceOperatorTerm (class in sfepy.terms.terms_basic), 470
IntegrateSurfaceTerm (class in sfepy.terms.terms_basic), 470
IntegrateTerm (class in sfepy.terms.terms_basic), 463
IntegrateVolumeMatTerm (class in sfepy.terms.terms_basic), 471
IntegrateVolumeOperatorTerm (class in sfepy.terms.terms_basic), 471
IntegrateVolumeTerm (class in sfepy.terms.terms_basic), 471
integrate() (sfepy.terms.terms_contact.ContactTerm static method), 474
integrate() (sfepy.terms.terms_hyperelastic_base.HyperElasticBase static method), 506
integration (class in sfepy.terms.terms_basic), 450
integration (sfepy.terms.terms_adf_navier_stokes.NSOFsSurfaceMinDPressTerm attribute), 454
integration (sfepy.terms.terms_basic.IntegrateMatTerm attribute), 462
integration (sfepy.terms.terms_basic.IntegrateOperatorTerm attribute), 463
integration (sfepy.terms.terms_basic.IntegrateTerm attribute), 463
integration (sfepy.terms.terms_basic.SurfaceMomentTerm attribute), 464
integration (sfepy.terms.terms_basic.VolumeSurfaceTerm attribute), 464
L2ConstantSurfaceField (class in sfepy.discrete.fem.fields_l2), 267
L2ConstantVolumeField (class in sfepy.discrete.fem.fields_l2), 267
label_dofs() (in module sfepy.parallel.plot_parallel_dofs), 388
label_global_entities() (in module sfepy.postprocess.plot_cmesh), 389
label_local_entities() (in module sfepy.postprocess.plot_cmesh), 389
label_points() (in module sfepy.postprocess.plot_quadrature), 390
LagrangeNodes (class in sfepy.discrete.fem.poly_spaces), 287
LagrangePolySpace (class in sfepy.discrete.fem.poly_spaces), 287
LagrangeSimplexBPolySpace (class in sfepy.discrete.fem.poly_spaces), 287
LagrangeSimplexPolySpace (class in sfepy.discrete.fem.poly_spaces), 288
LagrangeTensorProductPolySpace (class in sfepy.discrete.fem.poly_spaces), 288
lame_from_stiffness() (in module sfepy.mechanics.matcoefs), 360
lame_from_youngpoisson() (in module sfepy.mechanics.matcoefs), 360
laplace() (in module sfepy.linalg.sympy_operators), 354
LaplaceTerm (class in sfepy.terms.terms_diffusion), 483
layout_letters (class in sfepy.discrete.fem.fields_base.FEField), 537
LCBCOperator (class in sfepy.discrete.fem.lcbc_operators), 272
leaveonlyphysicalsurfaces() (class in sfepy.mesh.geom_tools.geometry method), 378
leaveonlyphysicalvolumes() (class in sfepy.mesh.geom_tools.geometry method), 378
generate_poly_spaces() (class in sfepy.discrete.dg.poly_spaces), 378
LegendrePolySpace (class in sfepy.discrete.dg.poly_spaces), 308
LegendreTensorProductPolySpace (class in sfepy.discrete.dg.poly_spaces), 309
letters (class in sfepy.terms.terms_multilinear.ExpressionBuilder), 538
light_copy() (class in sfepy.discrete.common.region.Region method), 257
line (in module sfepy.tests.test_laplace_unit_square), 581
linear() (in module sfepy.terms.terms_elastic), 378
linear_x() (in module sfepy.tests.test_laplace_unit_square), 581
linear_y() (in module sfepy.tests.test_laplace_unit_square), 581
linear_z() (in module sfepy.tests.test_laplace_unit_square), 581
LinearCombinationBC (class in sfepy.discrete.conditions), 191
LinearConvevt2Term (class in sfepy.terms.terms_navier_stokes), 543
LinearConvevtTerm (class in sfepy.terms.terms_navier_stokes), 543
LinearElasticETHTerm (class in sfepy.terms.terms_elastic), 496
LinearElasticIsotropicTerm (class in sfepy.terms.terms_elastic), 497
LinearElasticLADTerm (class in sfepy.terms.terms_elastic), 497
LinearElasticTerm (class in sfepy.terms.terms_elastic), 497
LinearElasticTHTerm (class in sfepy.terms.terms_elastic), 497
LinearElasticYPADTerm (class in sfepy.terms.terms_jax), 519
linearize() (class in sfepy.terms.terms_base.FEField method), 265
LinearPointSpringTerm (class in sfepy.terms.terms_point), 551
LinearPrestressTerm (class in
match_coors() (in module sfepy.discrete.fem.periodic), 286
match_grid_line() (in module sfepy.discrete.fem.periodic), 286
match_grid_plane() (in module sfepy.discrete.fem.periodic), 286
match_plane_by_dir() (in module sfepy.discrete.fem.periodic), 286
match_x_line() (in module sfepy.discrete.fem.periodic), 286
match_x_plane() (in module sfepy.discrete.fem.periodic), 286
match_y_line() (in module sfepy.discrete.fem.periodic), 286
match_y_plane() (in module sfepy.discrete.fem.periodic), 286
match_z_line() (in module sfepy.discrete.fem.periodic), 286
match_z_plane() (in module sfepy.discrete.fem.periodic), 287
MatchDofsOperator (class sfepy.discrete.fem.ibc_operators), 273
Material (class in sfepy.discrete.materials), 204
Materials (class in sfepy.discrete.materials), 206
MatlabEigenvalueSolver (class sfepy.solvers.eigen), 396
MatrixAction (class in module sfepy.linalg.utils), 358
max_diff_csr() (in module sfepy.linalg.utils), 358
mbfg (module sfepy.discrete.fem.extmods.bases), 283
mblock (module sfepy.solvers.ls_mumps.mumps_struc_c_4), 408
mblock (module sfepy.solvers.ls_mumps.mumps_struc_c_5_0), 411
mblock (module sfepy.solvers.ls_mumps.mumps_struc_c_5_1), 414
mblock (module sfepy.solvers.ls_mumps.mumps_struc_c_5_2), 418
mblock (module sfepy.solvers.ls_mumps.mumps_struc_c_5_3), 422
merge_lines() (in module sfepy.mesh.mesh_tools), 383
merge_mesh() (in module sfepy.discrete.fem.mesh), 279
Mesh (class in module sfepy.discrete.fem.mesh), 277
Mesh3DMeshIO (class in module sfepy.discrete.fem.meshio), 283
mesh_conn (module sfepy.discrete.fem.extmods.bases), 259
mesh_coors (module sfepy.discrete.fem.extmods.bases), 283
mesh_from_groups() (module sfepy.discrete.fem.meshio), 286
mesh_hook() (module sfepy.tests.test_eigenvalue_solvers), 579
mesh_hook() (module sfepy.tests.test_meshio), 584
MeshIO (class in module sfepy.discrete.fem.meshio), 283
MeshLibIO (module sfepy.discrete.fem.meshio), 285
metis_options (module sfepy.solvers.ls_mumps.mumps_struc_c_5_2), 418
metis_options (module sfepy.solvers.ls_mumps.mumps_struc_c_5_3), 422
mini_newton() (module sfepy.linalg.utils), 358
MiniAppBase (class in module sfepy.homogenization.coefs_base), 333
minmod() (module sfepy.discrete.dg.limiters), 310
minmod_seq() (module sfepy.discrete.dg.limiters), 311
mode (module sfepy.terms.terms_diffusion.AdvectDivFreeTerm), 480
mode (module sfepy.terms.terms_dot.BCNewtonTerm), 487
modes (module sfepy.terms.terms_biot.BiotETHTerm), 467
modes (module sfepy.terms.terms_biot.BiotTerm), 469
modes (module sfepy.terms.terms.BioTHTerm), 468
modes (module sfepy.terms.terms_constraints.NonPenetrationTerm), 473
modes (module sfepy.terms.terms_diffusion.AdvectionDGFluxTerm), 476
modes (module sfepy.terms.terms_diffusion.DiffusionDGFluxTerm), 477
modes (module sfepy.terms.terms_diffusion.DiffusionInteriorPenaltyTerm), 478
modes (module sfepy.terms.terms_diffusion.NonlinearHyperbolicDGFluxTerm), 479
modes (module sfepy.terms.terms_diffusion.NonlinearScalarDGFluxTerm), 480
modes (module sfepy.terms.terms_diffusion.DiffusionCoupling), 481
modes (module sfepy.terms.terms_diffusion.DiffusionTerm), 482
modes (module sfepy.terms.terms_diffusion.LaplaceTerm), 484
modes (module sfepy.terms.terms_dot.DotProductTerm), 488
modes (module sfepy.terms.terms_dot.ScalarDotMGradScalarTerm), 490
modes (module sfepy.terms.terms_dot.VectorDotGradScalarTerm), 491
modes (module sfepy.terms.terms_dot.VectorDotScalarTerm), 492
modes (module sfepy.terms.terms_elastic.ElasticWaveCauchyTerm), 496
modes (module sfepy.terms.terms_elastic.LinearElasticTerm), 499
modes (module sfepy.terms.terms_elastic.LinearPrestressTerm), 499
modes (module sfepy.terms.terms_elastic.NonSymElasticTerm), 502
modes (sfepy.terms/terms_jax.LinearElasticLADTerm attribute), 519
modes (sfepy.terms/terms_jax.LinearElasticYPADTerm attribute), 520
modes (sfepy.terms/terms_jax.MassADTerm attribute), 520
modes (sfepy.terms/terms_jax.NeoHookeanTLADTerm attribute), 521
modes (sfepy.terms/terms_jax.OgdenTLADTerm attribute), 522
modes (sfepy.terms/terms_mass.MassTerm attribute), 525
modes (sfepy.terms/terms_multilinear.EConvectTerm attribute), 527
modes (sfepy.terms/terms_multilinear.EDiffusionTerm attribute), 528
modes (sfepy.terms/terms_multilinear.EDivGradTerm attribute), 529
modes (sfepy.terms/terms_multilinear.EDivTerm attribute), 529
modes (sfepy.terms/terms_multilinear.EDotTerm attribute), 530
modes (sfepy.terms/terms_multilinear.ELaplaceTerm attribute), 531
modes (sfepy.terms/terms_multilinear.ELinearConvectTerm attribute), 532
modes (sfepy.terms/terms_multilinear.ELinearElasticTerm attribute), 533
modes (sfepy.terms/terms_multilinear.ELinearTractionTerm attribute), 533
modes (sfepy.terms/terms_multilinear.ENonSymElasticTerm attribute), 535
modes (sfepy.terms/terms_multilinear.EScalarDotMGradScalarTerm attribute), 535
modes (sfepy.terms/terms_multilinear.EStokesTerm attribute), 536
modes (sfepy.terms/terms_multilinear.SurfaceFluxOperatorTerm attribute), 539
modes (sfepy.terms/terms_navier_stokes.DivGradTerm attribute), 540
modes (sfepy.terms/terms_navier_stokes.StokesTerm attribute), 547
modes (sfepy.terms/terms_navier_stokes.StokesWaveDivTerm attribute), 547
modes (sfepy.terms/terms_piezo.PiezocouplingTerm attribute), 549
modes (sfepy.terms/terms_sensitivity.ESDDiffusionTerm attribute), 553
modes (sfepy.terms/terms_sensitivity.ESDDivGradTerm attribute), 553
modes (sfepy.terms/terms_sensitivity.ESDDotTerm attribute), 554
modes (sfepy.terms/terms_sensitivity.ESDLLinearElasticTerm attribute), 555
modes (sfepy.terms/terms_sensitivity.ESDLLinearTractionTerm attribute), 555

modify_mesh() (in module sfepy.test_term_sensitivity), 588
module
build_helpers, 588
gen_gallery, 591
gen_legendre_simplex_base, 592
gen_lobatto1d_c, 592
gen_release_notes, 592
gen_serendipity_basis, 592
gen_solver_table, 592
gen_term_table, 593
install_data, 593

sfepy.applications.application, 160
sfepy.applications.evp_solver_app, 160
sfepy.applications.pde_solver_app, 161
sfepy.base.base, 162
sfepy.base.compact, 168
sfepy.base.conf, 172
sfepy.base.getch, 174
sfepy.base.goptions, 175
sfepy.base.ioutils, 175
sfepy.base.log, 180
sfepy.base.log.plotter, 182
sfepy.base.mem_usage, 182
sfepy.base.multiproc, 183
sfepy.base.multiproc_mpi, 183
sfepy.base.parse_conf, 187
sfepy.base.pltutils, 188
sfepy.base.reader, 188
sfepy.base.resolve_deps, 188
sfepy.base.testing, 189
sfepy.config, 159
sfepy.discrete.common.dof_info, 237
sfepy.discrete.common.domain, 240
sfepy.discrete.common.extmods._fmfield, 241
sfepy.discrete.common.extmods._geommech, 241
sfepy.discrete.common.extmods.assemble, 241
sfepy.discrete.common.extmods.cmapping, 241
sfepy.discrete.common.extmods.cmesh, 241

Index 649
sfepy.tests.test_volume, 588
sfepy.version, 160
show_authors, 594
show_terms_use, 594
sync_module_docs, 594

MomentLimiter1D (class in sfepy.discrete.dg.limiters), 310
MomentLimiter2D (class in sfepy.discrete.dg.limiters), 310
MooneyRivlinTLTerm (class in sfepy.terms.terms_hyperelastic_tl), 510
MooneyRivlinULTerm (class in sfepy.terms.terms_hyperelastic_ul), 517
move_control_point() (sfepy.mesh.splinebox.SplineBox method), 385

MPIFileHandler (class in sfepy.base.multiproc_mpi), 183

MPILEgLogFile (class in sfepy.base.multiproc_mpi), 184

MRLCBCOperator (class in sfepy.discrete.fem.lcbc_operators), 273
mulAB_integrate() (in module sfepy.terms.extmods, 570

MultiProblem (class in sfepy.solvers.ls), 399
mumps_parallel_solve() (in module sfepy.solvers.ls.mumps_parallel), 424
mumps_pcomplex (in module sfepy.solvers.ls.mumps), 406
mumps_preal (in module sfepy.solvers.ls.mumps), 406
mumps_struc_c_4 (class in sfepy.solvers.ls.mumps), 406
mumps_struc_c_5_0 (class in sfepy.solvers.ls.mumps), 409
mumps_struc_c_5_1 (class in sfepy.solvers.ls.mumps), 413
mumps_struc_c_5_2 (class in sfepy.solvers.ls.mumps), 416
mumps_struc_c_5_3 (class in sfepy.solvers.ls.mumps), 420
mumps_struc_c_x (class in sfepy.solvers.ls.mumps), 424

MUMPSParallelSolver (class in sfepy.solvers.ls), 399

MUMPS SylvesterSolver (class in sfepy.solvers.ls), 399

MyQueue (class in sfepy.base.multiproc_proc), 186

n (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418
n (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422
n_coor (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 243
n_el (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 243
n_incident (sfepy.discrete.common.extmods.cmesh.CConnectivity attribute), 241
name (sfepy.discrete.dg.limiters.DGLimiter attribute), 310
name (sfepy.discrete.dg.limiters.IdentityLimiter attribute), 310
name (sfepy.discrete.dg.limiters.MomentLimiter1D attribute), 310
name (sfepy.discrete.dg.limiters.MomentLimiter2D attribute), 310
name (sfepy.discrete.dg.poly_spaces.LegendreSimplexPolySpace attribute), 390
name (sfepy.discrete.dg.poly_spaces.LegendreTensorProductPolySpace attribute), 390
name (sfepy.discrete.dg.poly_spaces.BernsteinSimplexPolySpace attribute), 287
name (sfepy.discrete.dg.poly_spaces.BernsteinTensorProductPolySpace attribute), 287
name (sfepy.discrete.dg.poly_spaces.LagrangeSimplexBPolySpace attribute), 287
name (sfepy.discrete.dg.poly_spaces.LagrangeSimplexPolySpace attribute), 287
name (sfepy.discrete.dg.poly_spaces.LagrangeTensorProductPolySpace attribute), 288
name (sfepy.discrete.dg.poly_spaces.LobattoTensorProductPolySpace attribute), 288
name (sfepy.discrete.dg.poly_spaces.SEMTensorProductPolySpace attribute), 288
name (sfepy.discrete.dg.poly_spaces.SerendipityTensorProductPolySpace attribute), 290
name (sfepy.solvers.auto_fallback.AutoDirect attribute), 395
name (sfepy.solvers.auto_fallback.AutoIterative attribute), 395
name (sfepy.solvers.solvers.eigen.LOBPCGEigenvalueSolver attribute), 396
name (sfepy.solvers.solvers.eigen.MatlabEigenvalueSolver attribute), 396
name (sfepy.solvers.solvers.eigen.PrimeEigenvalueSolver attribute), 397
name (sfepy.solvers.solvers.eigen.ScipyEigenvalueSolver attribute), 398
name (sfepy.solvers.solvers.eigen.ScipySGEigenvalueSolver attribute), 398
name (sfepy.solvers.solvers.SLEPcEigenvalueSolver attribute), 398
name (sfepy.solvers.ls.MultiProblem attribute), 400
name (sfepy.solvers.ls.MUMPSParallelSolver attribute), 399
name (sfepy.solvers.ls.MUMPSSolver attribute), 399
name (sfepy.solvers.ls.PETScKrylovSolver attribute), 401
name (sfepy.solvers.ls.PyAMGKrylovSolver attribute), 401
name (sfepy.solvers.ls.PyAMGSolver attribute), 402
name (sfepy.solvers.ls.RMMSSolver attribute), 402
name (sfepy.solvers.ls.SchurMumps attribute), 403
name (sfepy.solvers.ls.ScipyDirect attribute), 403
name (sfepy.solvers.ls.ScipyIterative attribute), 404
name (sfepy.solvers.ls.ScipySuperLU attribute), 404
name (sfepy.solvers.ls.ScipyUmfpack attribute), 405
name (sfepy.solvers.nls.Newton attribute), 426
name (sfepy.solvers.nls.PETScNonlinearSolver attribute), 427
name (sfepy.solvers.nls.ScipyBroyden attribute), 428
name (sfepy.solvers.optimize.FMinSteepestDescent attribute), 429
name (sfepy.solvers.optimize.ScipyFMinSolver attribute), 430
name (sfepy.solvers.oseen.Oseen attribute), 431
name (sfepy.solvers.geigen.LQuadraticEVPSolver attribute), 432
name (sfepy.solvers.semsmooth_newton.SemsmoothNewton attribute), 433
name (sfepy.solvers.ts_controllers.ElastodynamicsBasicTSC attribute), 437
name (sfepy.solvers.ts_controllers.ElastodynamicsLinearTSC attribute), 438
name (sfepy.solvers.ts_controllers.ElastodynamicsPIDTSC attribute), 438
name (sfepy.solvers.ts_controllers.FixedTSC attribute), 439
name (sfepy.solvers.ts_controllers.TimesSequenceTSC attribute), 439
name (sfepy.solvers.ts_dg_solvers.DGMultiStageTSS attribute), 311
name (sfepy.solvers.ts_dg_solvers.EulerStepSolver attribute), 312
name (sfepy.solvers.ts_dg_solvers.RK4StepSolver attribute), 312
name (sfepy.solvers.ts_dg_solvers.TVDRK3StepSolver attribute), 312
name (sfepy.solvers.ts_solver.AdaptiveTimeSteppingSolver attribute), 440
name (sfepy.solvers.ts_solver.BatheTS attribute), 441
name (sfepy.solvers.ts_solver.CentralDifferenceTS attribute), 442
name (sfepy.solvers.ts_solver.GeneralizedAlphaTS attribute), 443
name (sfepy.solvers.ts_solver.NewmarkTS attribute), 444
name (sfepy.solvers.ts_solver.SimpleTimeSteppingSolver attribute), 445
name (sfepy.solvers.ts_solvers.StationarySolver attribute), 445
name (sfepy.solvers.ts_solvers.VelocityVerletTS attribute), 446
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.AdjustConvect1Term attribute), 452
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.AdjustConvect2Term attribute), 453
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.AdjustDivGradTerm attribute), 453
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFDivergenceTerm attribute), 453
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFStabilizationTerm attribute), 454
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFStabilizationTerm attribute), 454
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFDivergenceTerm attribute), 455
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFDivergenceTerm attribute), 456
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFDivergenceTerm attribute), 457
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFDivergenceTerm attribute), 457
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFSurfaceTerm attribute), 458
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.NSOFSurfaceTerm attribute), 459
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SDSUPGCStabilizationTerm attribute), 460
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SDSUPGCStabilizationTerm attribute), 460
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 461
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 461
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 461
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 462
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 463
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 463
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 464
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 464
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 465
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 465
name (sfepy.solvers.ts_solvers.AdjustNavierStokes.SUPGStabilizationTerm attribute), 466
name (sfepy.terms.terms_elastic.LinearElasticETHTerm attribute), 497
name (sfepy.terms.terms_elastic.LinearElasticIsotropicTerm attribute), 497
name (sfepy.terms.terms_elastic.LinearElasticTHTerm attribute), 499
name (sfepy.terms.terms_elastic.LinearPrestressTerm attribute), 499
name (sfepy.terms.terms_elastic.LinearSpringTerm attribute), 500
name (sfepy.terms.terms_elastic.LinearStrainFiberTerm attribute), 501
name (sfepy.terms.terms_elastic.LinearTrussInternalForceTerm attribute), 501
name (sfepy.terms.terms_elastic.LinearTrussTerm attribute), 502
name (sfepy.terms.terms_elastic.NonsymElasticTerm attribute), 502
name (sfepy.terms.terms_elastic.SDLinearElasticTerm attribute), 503
name (sfepy.terms.terms_elastic.ElectricSourceTerm attribute), 505
name (sfepy.terms.terms_fibres.FibresActiveTLTerm attribute), 505
name (sfepy.terms.terms_hyperelastic_base.DeformationGradientTerm attribute), 505
name (sfepy.terms.terms_hyperelastic_tl.BulkActiveTLTerm attribute), 507
name (sfepy.terms.terms_hyperelastic_tl.BulkPenaltyTLTerm attribute), 507
name (sfepy.terms.terms_hyperelastic_tl.BulkPressureTLTerm attribute), 508
name (sfepy.terms.terms_hyperelastic_tl.BulkPenaltyULTerm attribute), 516
name (sfepy.terms.terms_hyperelastic_tl.CompressibilityULTerm attribute), 516
name (sfepy.terms.terms_hyperelastic_tl.MooneyRivlinULTerm attribute), 517
name (sfepy.terms.terms_hyperelastic_tl.NeoHookeanULTerm attribute), 518
name (sfepy.terms.terms_hyperelastic_tl.VolumeULTerm attribute), 518
name (sfepy.terms.terms_hyperelastic_ul.BulkActiveULTerm attribute), 497
name (sfepy.terms.terms_hyperelastic_ul.BulkPressureULTerm attribute), 497
name (sfepy.terms.terms_hyperelastic_ul.CompressibilityULTerm attribute), 497
name (sfepy.terms.terms_hyperelastic_ul.MooneyRivlinULTerm attribute), 498
name (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm attribute), 499
name (sfepy.terms.terms_hyperelastic_ul.VolumeULTerm attribute), 499
name (sfepy.terms.terms_jax.LinearElasticLADTerm attribute), 519
name (sfepy.terms.terms_jax.LinearElasticYPADTerm attribute), 520
name (sfepy.terms.terms_jax.MassADTerm attribute), 520
name (sfepy.terms.terms_jax.NeoHookeanTLADTerm attribute), 521
name (sfepy.terms.terms_jax.OgdenTLADTerm attribute), 522
name (sfepy.terms.terms_mass.MassTerm attribute), 525
name (sfepy.terms.terms_membrane.TLMembraneTerm attribute), 526
name (sfepy.terms.terms_multilinear.ECauchyStressTerm attribute), 527
name (sfepy.terms.terms_multilinear.EConve..
name (sfepy.terms.terms_multilinear.SurfaceFluxOperator attribute), 539
name (sfepy.terms.terms_hyperelastic_tl.NeoHookeanULTerm attribute), 550
name (sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLADTerm attribute), 553
name (sfepy.terms.terms_hyperelastic_tl.NeoHookeanTLTerm attribute), 554
name (sfepy.terms.terms_point.ConcentratedPointLoadTerm attribute), 551
name (sfepy.terms.terms_point.LinearPointSpringTerm attribute), 552
name (sfepy.terms.terms_sensitivity.ESDDiffusionTerm attribute), 553
name (sfepy.terms.terms_sensitivity.ESDVectorDotGradScalarTerm attribute), 557
name (sfepy.terms.terms_sensitivity.ESDStokesTerm attribute), 558
name (sfepy.terms.terms_sensitivity.ESDVectorStressTerm attribute), 559
name (sfepy.terms.terms_sensitivity.ESDVolumeForceTerm attribute), 561
name (sfepy.terms.terms_sensitivity.ESDSurfaceIntegrateTerm attribute), 562
name (sfepy.terms.terms_sensitivity.ESDGradientTerm attribute), 563
name (sfepy.terms.terms_sensitivity.ESDCouplingTerm attribute), 564
name (sfepy.terms.terms_sensitivity.ESDNormalDotTerm attribute), 565
name (sfepy.terms.terms_piezo.PiezoStressTerm attribute), 566
name (sfepy.terms.terms_piezo.PiezoCouplingTerm attribute), 567
name (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm attribute), 568
name (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTADTerm attribute), 570
name (sfepy.terms.terms_hyperelastic_ul.NeoHookeanULTerm attribute), 571
name (sfepy.terms.terms_piezo.PiezoStrainTerm attribute), 572
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 408
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 411
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 414
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422
name (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 408
name (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 422
name (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 566
name (sfepy.terms.terms_surface.SurfaceJumpTerm attribute), 567
name (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 566
name (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 566
name (sfepy.terms.terms_volume.LinearVolumeForceTerm attribute), 566
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
name (sfepy.terms.terms_surface.LinearTractionTerm attribute), 562
NodalLCOperator

**new** (sfepy.terms.terms.Term static method), 450

**new_ulf_iteration**

(sfepy.discrete.evaluate.Evaluator static method), 198

**new_vtk_polyline** (sfepy.postprocess.probes_vtk.Probe method), 392

NewtonTS (class in sfepy.solvers.ts_solvers), 443

Nloc_rhs (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

Nloc_rhs (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

NLSStatus (class in sfepy.base.testing), 189

nnz (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 414

nnz (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

nnz (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 414

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

NodalLCOperator (class in sfepy.discrete.fem.lcbc_operators), 274

NodeDescription (class in sfepy.discrete.fem.poly_spaces), 288

NonlinearDiffusionTerm (class in sfepy.terms.terms_diffusion), 484

NonlinearHyperbolicDGFluxTerm (class in sfepy.terms.terms_dg), 478

NonlinearScalarDotGradTerm (class in sfepy.terms.terms_dg), 479

NonlinearSolver (class in sfepy.solvers.solvers), 434

NonlinearVolumeForceTerm (class in sfepy.terms.terms_volume), 566

NonPenetrationPenaltyTerm (class in sfepy.terms.terms_constraints), 472

NonPenetrationTerm (class in sfepy.terms.terms_constraints), 473

NonsymElasticTerm (class in sfepy.terms.terms_elastic), 502

NoOptionsDocs (class in build_helpers), 589

NoPenetrationOperator (class in sfepy.discrete.fem.lcbc_operators), 274

**normalize_vectors** (in module sfepy.linalg.utils), 358

npcol (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 408

npcol (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 411

npcol (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 414

npcol (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

npcol (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

nprow (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 408

nprow (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 411

nprow (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 415

nprow (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

nprow (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 414

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 418

nnz_loc (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 422

da具体内容。
problem() (in module sfepy.tests.test_term_consistency), 588
problem() (in module sfepy.tests.test_term_sensitivity), 588
problem() (in module sfepy.tests.test_volume), 588
ProblemConf (class in sfepy.base.conf), 172
process_command() (sfepy.base.log_plotter.LogPlotter method), 182
process_conf() (sfepy.solvers.solvers.Solver class method), 434
process_options() (sfepyapplications.evpsolver_app.PDESolverApp static method), 160
process_options() (sfepyapplications.pdesolver_app.PDESolverApp static method), 161
process_options() (sfepyhomogenization.bandgaps_app.AcousticBandGapsApp static method), 329
process_options() (sfepyhomogenization.coefs_base.MinAppBase class method), 333
process_options() (sfepyhomogenization.coefs_phononic.BandGaps method), 335
process_options() (sfepyhomogenization.coefs_phononic.ChristoffelAcousticTensor method), 336
process_options() (sfepyhomogenization.coefs_phononic.Eigenmomenta method), 336
process_options() (sfepyhomogenization.coefs_phononic.PolarizationAngles method), 337
process_options() (sfepyhomogenization.coefs_phononic.SimpleEVPSolver method), 337
process_options() (sfepyhomogenization.engine.HomogenizationApp static method), 340
process_options() (sfepyhomogenization.homogen_app.HomogenizationApp static method), 343
process_options_pr() (sfepyhomogenization.bandgaps_app.AcousticBandGapsApp static method), 329
process_reqs_coeffs() (sfepyhomogenization.engine.HomogenizationWorkerMulti static method), 342
project_by_component() (in module sfepy.discrete.projections), 225
project_to_facets() (in module sfepy.discrete.projections), 225
PSPGStabilizationTerm (class in sfepyterms.terms.navier_stokes), 544
PSPGPStabilizationTerm (class in sfepyterms.terms.navier_stokes), 544
put() (sfepy.base.multiproc_mpi.RemoteQueue method), 184
put() (sfepy.base.multiproc_mpi.RemoteQueueMaster method), 185
put() (sfepy.base.multiproc_proc.MyQueue method), 186
pv_plot() (in module sfepy.scripts.resview), 574
PyAMGKrylovSolver (class in sfepy.solvers.ls), 401
PyAMGSolver (class in sfepy.solvers.ls), 401
PyCMapping (class in sfepy.discrete.common.mappings), 251
pytest_adoption() (in module sfepy.tests.conftest), 576
pytest_configure() (in module sfepy.tests.conftest), 576
python_include() (sfepy.config.Config method), 159
pythonshell() (in module sfepy.base.base), 167
python_version() (sfepy.config.Config method), 159
Q
QuadraticWeightedSolver (class in sfepy.solvers.solvers), 434
quadraturePoints (class in sfepy.discrete.quadratures), 226
Quantity (class in sfepy.mechanics.units), 369
R
RayProbe (class in sfepy.discrete.probes), 210
R
RayProbe (class in sfepy.discrete.probes), 211
Read() (sfepydiscrete.fem.meshio.ANSYSCDBMeshIO method), 279
Read() (sfepydiscrete.fem.meshio.HDF5MeshIO method), 279
Read() (sfepydiscrete.fem.meshio.HDF5MeshIO method), 282
Read() (sfepydiscrete.fem.meshio.Mesh3DMeshIO method), 283
Read() (sfepydiscrete.fem.meshio.MeshIO method), 284
Read() (sfepydiscrete.fem.meshio.MeshLibIO method), 285
Read() (sfepydiscrete.fem.meshio.NEUMeshIO method), 285
Read() (sfepydiscrete.fem.meshio.UserMeshIO method), 285
Read() (sfepydiscrete.fem.meshio.XYZMeshIO method), 286
read_array() (in module sfepybase.ioutils), 178
read_bounding_box() (sfepy.discrete.fem.meshio.ANSYSCDBMeshIO method), 279
read_bounding_box() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 282
read_bounding_box()  
(sfepy.discrete.fem.meshio.MeshLibIO  
method), 282  

read_bounding_box()  
(sfepy.discrete.fem.meshio.XYZMeshIO  
method), 286  

read_data()  
(sfepy.discrete.fem.meshio.GmshIO  
method), 280  

read_data()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_data()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_data()  
(sfepy.discrete.fem.meshio.MeshIO  
method), 282  

read_data()  
(sfepy.discrete.fem.meshio.MeshLibIO  
method), 285  

read_data()  
(sfepy.discrete.fem.meshio.MeshLibIO  
method), 285  

read_data()  
(sfepy.discrete.fem.meshio.Mesh3DMeshIO  
method), 283  

read_data()  
(sfepy.discrete.fem.meshio.MeshIO  
method), 282  

read_dict_hdf5()  
in module sfepy.base.ioutils, 178  

read_dimension()  
(sfepy.discrete.fem.meshio.ANSYSCDMeshIO  
method), 279  

read_dimension()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_dimension()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_dimension()  
(sfepy.discrete.fem.meshio.Mesh3DMeshIO  
method), 283  

read_dimension()  
(sfepy.discrete.fem.meshio.MeshLibIO  
method), 285  

read_dimension()  
(sfepy.discrete.fem.meshio.MeshIO  
method), 282  

read_domain_from_hdf5()  
(sfepy.discrete.iga.domain.IGDomain  
static method), 313  

read_from_hdf5()  
in module sfepy.discrete.fem.probes, 211  

read_header()  
in module sfepy.discrete.fem.probes, 211  

read_iga_data()  
in module sfepy.discrete.iga.io, 325  

read_last_step()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_last_step()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 284  

read_list()  
in module sfepy.base.ioutils, 178  

read_log()  
in module sfepy.base.log, 181  

read_mesh()  
in module sfepy.scripts.resview, 574  

read_mesh_from_hdf5()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
static method), 282  

read_results()  
in module sfepy.discrete.probes, 211  

read_sparse_matrix_from_hdf5()  
in module sfepy.base.ioutils, 179  

read_sparse_matrix_hdf5()  
in module sfepy.base.ioutils, 179  

read_time_history()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_time_step()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_times()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_times()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 282  

read_token()  
in module sfepy.base.ioutils, 179  

read_variables_time_history()  
(sfepy.discrete.fem.meshio.HDF5MeshIO  
method), 283  

Reader  
(class in sfepy.base.reader), 188  

reconstruct_legendre_dofs()  
in module sfepy.homogenization.recovery, 346  

recover_bones()  
in module sfepy.homogenization.recovery, 345  

recover_paraflow()  
in module sfepy.homogenization.recovery, 346  

reduce_vec()  
sfepy.solvers.ls_mumps.mumps_struc_c_4 at-

redrhs  
sfepy.solvers.ls_mumps.mumps_struc_c_5_0 at-

redrhs  
sfepy.solvers.ls_mumps.mumps_struc_c_5_1 at-

redrhs  
sfepy.solvers.ls_mumps.mumps_struc_c_5_2 at-

redrhs  
sfepy.solvers.ls_mumps.mumps_struc_c_5_3 at-

reduce_on_datas()  
sfepy.discrete.materials.Material  
method), 205  

reduce_vec()  
sfepy.discrete.equations.Equations  
method), 196  

reduce_vec()  
sfepy.discrete.variables.Variables  
method), 235  

refine()  
in module sfepy.discrete.fem.refine_hanging, 291  

refine()  
in module sfepy.discrete.fem.refine, 290  

refine()  
in module sfepy.discrete.fem.refine, 290  

refine()  
in module sfepy.tests.test_domain, 579  

refine()  
sfepy.discrete.fem.FEDomain  
method), 258  

refine_1_2()  
in module sfepy.discrete.fem.refine, 290  

refine_2_3()  
in module sfepy.discrete.fem.refine, 290  

refine_2_4()  
in module sfepy.discrete.fem.refine, 290  

refine_3_4()  
in module sfepy.discrete.fem.refine, 290  

refine_3_8()  
in module sfepy.discrete.fem.refine, 290  

refine_mes()  
in module sfepy.discrete.fem.utilities, 292  

refine_pars()  
sfepy.discrete.probes.Probe  
static method), 209  

refine_points()  
sfepy.discrete.probes.PointsProbe  
method), 209  

refine_points()  
sfepy.discrete.probes.Probe
rowsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
rowsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 415
rowsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
rowsca_from_mumps (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
run () (build_helpers.Clean method), 589
run () (build_helpers.DoxygenDocs method), 589
run () (build_helpers.SphinxHTMLDocs method), 589
run () (build_helpers.SphinxPDFDocs method), 589
run_declarative_example () (in module sfepy.base.testing), 189
run_resview_plot () (in module gen_gallery), 592

S
save () (sfepy.homogenization.coefs_base.CorrMiniApp method), 332
save () (sfepy.homogenization.coefs_base.TCorrectorsViaPressureEVP method), 333
save () (sfepy.homogenization.coefs_phononic.SimpleEVP method), 337
save_animation () (in module sfepy.discrete.dg.dg_ID_visualizer), 296
save_as_mesh () (sfepy.discrete.variables.FieldVariable method), 231
save_basis () (in module sfepy.discrete.iga.utils), 328
save_dict () (sfepy.applications.pde_solver_app.PDESolverApp method), 161
save_dir () (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 415
save_dir () (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
save_dir () (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
save_ebc () (sfepy.discrete.problem.Problem method), 219
save_log () (sfepy.homogenization.coefs_phononic.BandGaps static method), 336
save_mappings () (sfepy.discrete.common.fields.Field method), 247
save_only () (in module sfepy.applications.pde_solver_app), 161
save_options () (in module sfepy.base.ioutils), 179
save_prefix () (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 415
save_prefix () (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
save_prefix () (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
save_raw_bg_logs () (in module sfepy.homogenization.band_gaps_app), 330

radians (sfepy.base.ioutils), 327
radians () (in module sfepy.base.ioutils), 327
RKMMSolver (class in sfepy.solvers.ts_dg_solvers), 348
RMMsolver (class in sfepy.solvers.ls), 402
rotate_elastic_tensor () (in module sfepy.mechanics.shell10x), 367
rotation_matrix2d () (in module sfepy.linalg.geometry), 352
rowsca () (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
rowsca () (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
rowsca () (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 415
rowsca () (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
rowsca () (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
RigidOperator (in module sfepy.discrete.fem.lcbc_operators), 274
RK4StepSolver (class in sfepy.solvers.ts_dg_solvers), 312
rm_multi () (in module sfepy.homogenization.utils), 348

save_recovery_region() (in module sfepy.homogenization.recovery), 347
save_regions() (sfepy.discrete.common.domain.Domain method), 240
save_regions() (sfepy.discrete.problem.Problem method), 219
save_regions_as_groups() (sfepy.discrete.common.domain.Domain method), 240
save_regions_as_groups() (sfepy.discrete.problem.Problem method), 219
save_restart() (sfepy.discrete.problem.Problem method), 220
save_results() (sfepy.applications.evp_solver_app.EVPSolverApp method), 160
save_sol_snap() (in module sfepy.discrete.dg.dg_1D_vizualizer), 296
save_sparse_txt() (in module sfepy.linalg.sparse), 354
save_state() (sfepy.discrete.problem.Problem method), 220
save_time_history() (in module sfepy.postprocess.time_history), 393
ScalarDotGradScalarTerm (class in sfepy.terms.terms_adj_navier_stokes), 489
ScalarDotMGradScalarTerm (class in sfepy.terms.terms_adj_navier_stokes), 490
scale_matrix() (in module sfepy.solvers.oseen), 431
schur (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
schur (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
schur (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
schur (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
schur (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
schur_lld (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
schur_lld (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
schur_lld (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
schur_lld (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
schur_lld (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 423
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 419
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 419
schur_mloc (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 419
SchurEVP (class in sfepy.solvers.oseen), 337
ScipyBroyden (class in sfepy.solvers.nls), 427
ScipyDirect (class in sfepy.solvers.ls), 403
ScipyEigenvalueSolver (class in sfepy.solvers.eigen), 398
ScipyFMinSolver (class in sfepy.solvers.optimize), 429
ScipyIterative (class in sfepy.solvers.ls), 403
ScipySGEigenvalueSolver (class in sfepy.solvers.eigen), 398
ScipySuperLU (class in sfepy.solvers.ls), 404
ScipyUmfpack (class in sfepy.solvers.ls), 404
SDConvectTerm (class in sfepy.terms.terms_adj_navier_stokes), 455
SDDiffusionTerm (class in sfepy.terms.terms_diffusion), 484
SDDivGradTerm (class in sfepy.terms.terms_adj_navier_stokes), 455
SDDivTerm (class in sfepy.terms.terms_adj_navier_stokes), 456
SDDotTerm (class in sfepy.terms.terms_adj_navier_stokes), 456
SDGradDivStabilizationTerm (class in sfepy.terms.terms_adj_navier_stokes), 457
SDLLinearElasticTerm (class in sfepy.terms.terms_elastic), 503
SDLlinearTractiveTerm (class in sfepy.terms.terms_surface), 563
SDPiezoCouplingTerm (class in sfepy.terms.terms_piezo), 550
SDPSGCGStabilizationTerm (class in sfepy.terms.terms_adj_navier_stokes), 457
SDPSGPGStabilizationTerm (class in sfepy.terms.terms_adj_navier_stokes), 458
SDSufaceIntegrateTerm (class in sfepy.terms.terms_surface), 563
SDSUPGCStabilizationTerm (class in sfepy.terms.terms_adj_navier_stokes), 459
set_defaults() (in module sfepy.base.base), 167
set_dim() (in module sfepy.linalg.sympy_operators), 354
set_dof_info() (sfepy.solvers.solvers.TimeSteppingSolver method), 434
set_dofs() (sfepy.discrete.common.fields.Field method), 247
set_dofs() (sfepy.discrete.dg.fields.DGField method), 305
set_dofs() (sfepy.discrete.fem.fields_hierarchic.H1HierarchicVolumeField method), 267
set_dofs() (sfepy.discrete.fem.fields_nodal.H1NodalMixin method), 269
set_equations() (sfepy.discrete.problem.Problem method), 220
set_equations_instance() (sfepy.discrete.problem.Problem method), 221
set_extra_args() (sfepy.discrete.functions.Function method), 202
set_extra_args() (sfepy.discrete.materials.Material method), 205
set_facet_dofs() (sfepy.discrete.dg.fields.DGField method), 305
set_field_split() (sfepy.solvers.ls.PETScKrylovSolver method), 401
set_field_split() (sfepy.solvers.solvers.Solver method), 434
set_fields() (sfepy.discrete.problem.Problem method), 221
set_function() (sfepy.discrete.materials.Material method), 202
set_function() (sfepy.discrete.functions.Function method), 202
set_function() (sfepy.discrete.materials.Material method), 205
set_integ() (sfepy.discrete.problem.Problem method), 221
set_integral() (sfepy.terms.terms.Term method), 450
set_integral() (sfepy.terms.terms_shells.Shell10XTerm method), 559
set_kind() (sfepy.discrete.common.region.Region method), 257
set_kind_field() (sfepy.discrete.common.region.Region method), 257
set_knot_vector() (sfepy.mesh.bspline.BSpline method), 375
set_linear() (sfepy.discrete.problem.Problem method), 375
set_local_entities() (sfepy.discrete.common.extmods.cmesh.CMesh method), 243
set_logging_level() (in module sfepy.base.multiproc_mpi), 185
set_materials() (sfepy.discrete.problem.Problem method), 221
set_mesh_coors() (in module sfepy.discrete.fem.fields_base), 267
set_micro_states() (sfepy.homogenization.engine.HomogenizationEngine method), 340
set_mtx_centralized() (sfepy.solvers.ls.mumps.MumpsSolver method), 405
set_n_digit_from_min_dt() (sfepy.discrete.probes.Probe method), 436
set_n_point() (sfepy.discrete.probes.Probe method), 210
set_nonlin_states() (in module sfepy.homogenization.utils), 348
set orientation_map() (sfepy.discrete.fem.fe_surface.FESurface method), 261
set_output() (sfepy.base.base.Output method), 163
set_output_dir() (sfepy.discrete.problem.Problem method), 221
set_output_prefix() (sfepy.base.base.Output method), 163
set_param() (sfepy.mesh.bspline.BSpline method), 375
set_param() (sfepy.mesh.bspline.BSpline method), 375
set_param() (sfepy.mesh.bspline.BSpline method), 375
set_param() (sfepy.mesh.bspline.BSpline method), 375
<p>| sfepy.base.testing             | sfepy.discrete.fem.extmods.lobatto_bases             |
|                               | module, 189                                         |
| sfepy.base.timing             | sfepy.discrete.fem.facets                           |
| module, 189                   | module, 259                                         |
| sfepy.config                  | sfepy.discrete.fem.fe_surface                        |
| module, 159                   | module, 261                                         |
| sfepy.discrete.common.dof_info| sfepy.discrete.fem.fields_base                       |
| module, 237                   | module, 261                                         |
| sfepy.discrete.common.domain  | sfepy.discrete.fem.fields_hierarchic                 |
| module, 240                   | module, 267                                         |
| sfepy.discrete.common.extmods._fmfield |                 |
| module, 241                   | sfepy.discrete.fem.fields_12                         |
| sfepy.discrete.common.extmods._geommech |                 |
| module, 241                   | module, 267                                         |
| sfepy.discrete.common.extmods.assemble |                 |
| module, 241                   | sfepy.discrete.fem.fields_nodal                      |
| sfepy.discrete.common.extmods.cmapping |                 |
| module, 241                   | module, 269                                         |
| sfepy.discrete.common.extmods.cmesh |                 |
| module, 241                   | sfepy.discrete.fem.linearizer                        |
| sfepy.discrete.common.extmods.crefcoors |                 |
| module, 244                   | module, 275                                         |
| sfepy.discrete.common.fields  | sfepy.discrete.fem.mappings                         |
| module, 245                   | module, 275                                         |
| sfepy.discrete.common.global_interp |                 |
| module, 248                   | sfepy.discrete.fem.mesh                              |
| sfepy.discrete.common.mappings | module, 277                                         |
| module, 251                   | sfepy.discrete.fem.meshio                           |
| sfepy.discrete.common.poly_spaces | module, 279                                       |
| module, 253                   | sfepy.discrete.fem.periodic                         |
| sfepy.discrete.common.region  | module, 286                                         |
| module, 254                   | sfepy.discrete.fem.polymesh                         |
| sfepy.discrete.conditions     | module, 287                                         |
| module, 190                   | sfepy.discrete.fem.refine                           |
| sfepy.discrete.dg.dg_1D_vizualizer | module, 290                                   |
| module, 293                   | sfepy.discrete.fem.refine_hanging                   |
| sfepy.discrete.dg.dg_fields   | module, 291                                         |
| module, 298                   | sfepy.discrete.fem.utils                            |
| sfepy.discrete.dg.limiters    | module, 291                                         |
| module, 310                   | sfepy.discrete.functions                            |
| sfepy.discrete.dg.poly_spaces | module, 291                                         |
| module, 307                   | sfepy.discrete.iga.domain                           |
| sfepy.discrete.dg.equations   | module, 313                                         |
| module, 192                   | sfepy.discrete.iga.domain_generators                |
| sfepy.discrete.evaluate       | module, 314                                         |
| module, 198                   | sfepy.discrete.iga.extmods.gac                      |
| sfepy.discrete.evaluate_variable | module, 315                               |
| module, 202                   | sfepy.discrete.iga.fields                           |
| sfepy.discrete.fem._serendipity | module, 317                                   |
| module, 291                   | sfepy.discrete.iga.gac                              |
| sfepy.discrete.fem.domain     | module, 319                                         |
| module, 257                   | sfepy.discrete.iga.io                               |
| sfepy.discrete.fem.extmods.bases | module, 325                         |</p>
<table>
<thead>
<tr>
<th>sfepy.discrete.iga.mappings module, 326</th>
<th>sfepy.linalg.check_derivatives module, 349</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfepy.discrete.iga.plot_nurbs module, 327</td>
<td>sfepy.linalg.eigen module, 349</td>
</tr>
<tr>
<td>sfepy.discrete.iga.utils module, 327</td>
<td>sfepy.linalg.geometry module, 350</td>
</tr>
<tr>
<td>sfepy.discrete.integrals module, 203</td>
<td>sfepy.linalg.sparse module, 353</td>
</tr>
<tr>
<td>sfepy.discrete.materials module, 204</td>
<td>sfepy.linalg.sympy_operators module, 354</td>
</tr>
<tr>
<td>sfepy.discrete.parse_equations module, 207</td>
<td>sfepy.linalg.utils module, 354</td>
</tr>
<tr>
<td>sfepy.discrete.parse_regions module, 207</td>
<td>sfepy.mechanics.contact_bodies module, 358</td>
</tr>
<tr>
<td>sfepy.discrete.probes module, 208</td>
<td>sfepy.mechanics.elastic_constants module, 359</td>
</tr>
<tr>
<td>sfepy.discrete.problem module, 211</td>
<td>sfepy.mechanics.extmods.ccontres module, 372</td>
</tr>
<tr>
<td>sfepy.discrete.projections module, 224</td>
<td>sfepy.mechanics.matcoefs module, 359</td>
</tr>
<tr>
<td>sfepy.discrete.quadratures module, 225</td>
<td>sfepy.mechanics.units module, 369</td>
</tr>
<tr>
<td>sfepy.discrete.simplex_cubature module, 227</td>
<td>sfepy.mesh.bspline module, 372</td>
</tr>
<tr>
<td>sfepy.discrete.structural.fields module, 328</td>
<td>sfepy.mesh.geom_tools module, 377</td>
</tr>
<tr>
<td>sfepy.discrete.structural.mappings module, 329</td>
<td>sfepy.mesh.mesh_generators module, 379</td>
</tr>
<tr>
<td>sfepy.discrete.variables module, 227</td>
<td>sfepy.mesh.mesh_tools module, 383</td>
</tr>
<tr>
<td>sfepy.homogenization.band_gaps_app module, 329</td>
<td>sfepy.mesh.splinebox module, 384</td>
</tr>
<tr>
<td>sfepy.homogenization.coefficients module, 330</td>
<td>sfepy.parallel.evaluate module, 386</td>
</tr>
<tr>
<td>sfepy.homogenization.coefs_base module, 331</td>
<td>sfepy.parallel.parallel module, 386</td>
</tr>
<tr>
<td>sfepy.homogenization.coefs_elastic module, 334</td>
<td>sfepy.parallel.plot_parallel_dofs module, 388</td>
</tr>
<tr>
<td>sfepy.homogenization.coefs_perfusion module, 334</td>
<td>sfepy.postprocess.plot_cmesh module, 389</td>
</tr>
<tr>
<td>sfepy.homogenization.coefs_phononic module, 334</td>
<td>sfepy.postprocess.plot_dofs module, 390</td>
</tr>
<tr>
<td>sfepy.homogenization.convolutions module, 339</td>
<td>sfepy.postprocess.plot_facets module, 390</td>
</tr>
<tr>
<td>sfepy.homogenization.engine module, 340</td>
<td>sfepy.postprocess.plot_quadrature module, 390</td>
</tr>
<tr>
<td>sfepy.homogenization.homogen_app module, 343</td>
<td>sfepy.postprocess.probes_vtk module, 391</td>
</tr>
<tr>
<td>sfepy.homogenization.micmac module, 343</td>
<td></td>
</tr>
<tr>
<td>sfepy.homogenization.recovery module, 344</td>
<td></td>
</tr>
<tr>
<td>sfepy.homogenization.utils module, 347</td>
<td></td>
</tr>
</tbody>
</table>
sfepy.postprocess.time_history
   module, 392
sfepy.postprocess.utils_vtk
   module, 394
sfepy.scripts.blockgen
   module, 570
sfepy.scripts.convert_mesh
   module, 570
sfepy.scripts.cylindergen
   module, 570
sfepy.scripts.gen_iga_patch
   module, 570
sfepy.scripts.gen_mesh
   module, 571
sfepy.scripts.gen_mesh_prev
   module, 571
sfepy.scripts.plot_condition_numbers
   module, 571
sfepy.scripts.plot_logs
   module, 571
sfepy.scripts.plot_mesh
   module, 572
sfepy.scripts.plot_quadratures
   module, 572
sfepy.scripts.plot_times
   module, 572
sfepy.scripts.probe
   module, 572
sfepy.scripts.resview
   module, 573
sfepy.scripts.run_tests
   module, 575
sfepy.scripts.simple
   module, 575
sfepy.solvers.auto_fallback
   module, 395
sfepy.solvers.eigen
   module, 396
sfepy.solvers.ls
   module, 399
sfepy.solvers.ls_mumps
   module, 405
sfepy.solvers.ls_mumps_parallel
   module, 424
sfepy.solvers.nls
   module, 425
sfepy.solvers.optimize
   module, 428
sfepy.solvers.oseen
   module, 430
sfepy.solvers.qeigen
   module, 432
sfepy.solvers.semismooth_newton
   module, 432
sfepy.solvers.solvers
   module, 433
sfepy.solvers.ts
   module, 435
sfepy.solvers.ts_controllers
   module, 436
sfepy.solvers.ts_dg_solvers
   module, 311
sfepy.solvers.ts_solvers
   module, 439
sfepy.terms.extmods.terms
   module, 567
sfepy.terms.terms
   module, 447
sfepy.terms.terms_adj_navier_stokes
   module, 452
sfepy.terms.terms_basic
   module, 462
sfepy.terms.terms_biot
   module, 466
sfepy.terms.terms_compact
   module, 469
sfepy.terms.terms_constraints
   module, 472
sfepy.terms.terms_contact
   module, 474
sfepy.terms.terms_dg
   module, 475
sfepy.terms.terms_diffusion
   module, 480
sfepy.terms.terms_dot
   module, 486
sfepy.terms.terms_elastic
   module, 492
sfepy.terms.terms_electric
   module, 503
sfepy.terms.terms_fibres
   module, 504
sfepy.terms.terms_hyperelastic_base
   module, 505
sfepy.terms.terms_hyperelastic_tl
   module, 506
sfepy.terms.terms_hyperelastic_ul
   module, 515
sfepy.terms.terms_jax
   module, 518
sfepy.terms.terms_mass
   module, 525
sfepy.terms.terms_membrane
   module, 526
sfepy.terms.terms_multilinear
   module, 527
sfepy.terms.terms_navier_stokes
   module, 539
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfepy.terms.terms_piezo</td>
<td>548</td>
</tr>
<tr>
<td>sfepy.terms.terms_point</td>
<td>551</td>
</tr>
<tr>
<td>sfepy.terms.terms_sensitivity</td>
<td>552</td>
</tr>
<tr>
<td>sfepy.terms.terms_shells</td>
<td>558</td>
</tr>
<tr>
<td>sfepy.terms.terms_surface</td>
<td>560</td>
</tr>
<tr>
<td>sfepy.terms.terms_th</td>
<td>565</td>
</tr>
<tr>
<td>sfepy.terms.terms_volume</td>
<td>565</td>
</tr>
<tr>
<td>sfepy.terms.utils</td>
<td>566</td>
</tr>
<tr>
<td>sfepy.tests.confest</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_assembling</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_base</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_cmesh</td>
<td>576</td>
</tr>
<tr>
<td>sfepy.tests.test_conditions</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_declarative_examples</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_dg_field</td>
<td>577</td>
</tr>
<tr>
<td>sfepy.tests.test_dg_terms_calls</td>
<td>578</td>
</tr>
<tr>
<td>sfepy.tests.test_domain</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_ed_solvers</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_eigenvalue_solvers</td>
<td>579</td>
</tr>
<tr>
<td>sfepy.tests.test_elasticity_small_strain</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_fem</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_functions</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_high_level</td>
<td>580</td>
</tr>
<tr>
<td>sfepy.tests.test_homogenization_engine</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_homogenization_perfusion</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_hyperelastic_tlul</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_io</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_laplace_unit_disk</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_laplace_unit_square</td>
<td>581</td>
</tr>
<tr>
<td>sfepy.tests.test_lcbcs</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linalg</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linear_solvers</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_linearization</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_log</td>
<td>582</td>
</tr>
<tr>
<td>sfepy.tests.test_matcoefs</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_expand</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_generators</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_interp</td>
<td>583</td>
</tr>
<tr>
<td>sfepy.tests.test_mesh_smoothing</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_meshio</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_msm_laplace</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_msm_symbolic</td>
<td>584</td>
</tr>
<tr>
<td>sfepy.tests.test_normals</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_parsing</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_poly_spaces</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_projections</td>
<td>585</td>
</tr>
<tr>
<td>sfepy.tests.test_quadratures</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_ref_coors</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_refine_hanging</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_regions</td>
<td>586</td>
</tr>
<tr>
<td>sfepy.tests.test_semismooth_newton</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_sparse</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_splinebox</td>
<td>587</td>
</tr>
<tr>
<td>sfepy.tests.test_tensors</td>
<td>587</td>
</tr>
</tbody>
</table>
SurfaceDivTerm (class in sfepy.terms.terms_compact), 471
SurfaceFluxOperatorTerm (class in sfepy.terms.terms_diffusion), 485
SurfaceFluxOperatorTerm (class in sfepy.terms_multilinear), 538
SurfaceFluxTerm (class in sfepy.terms_diffusion), 486
SurfaceFluxTLTerm (class in sfepy.terms_hyperelastic_fld), 512
SurfaceGradTerm (class in sfepy.terms_surface), 471
SurfaceJumpTerm (class in sfepy.terms_surface), 564
SurfaceMomentTerm (class in sfepy.terms_basic), 464
SurfaceTerm (class in sfepy.terms_compact), 472
SurfaceTractionTLTerm (class in sfepy.terms_hyperelastic_fld), 513
sym (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
sym (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
sym (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
sym (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 420
sym (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 424
sym (sfepy.solvers.ls_mumps.mumps_struc_c_x attribute), 424
sym2dim() (in module sfepy.mechanics.tensors), 368
sym2nonsym() (in module sfepy.terms.extmods.terms), 570
sym2nonsym() (in module sfepy.terms.extmods.terms), 570
sym_perm (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
sym_perm (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 412
sym_perm (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
sym_perm (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 420
sym_perm (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 424
sym_tri_eigen() (in module sfepy.linalg.eigen), 350
symarray() (in module sfepy.terms.test_quadratures), 586
symbolic (sfepy.terms_dg.AdvectionDGFluxTerm attribute), 476
symbolic (sfepy.terms_dg.NonlinearHyperbolicDGFluxTerm attribute), 479
symbolic (sfepy.terms_diffusion.DiffusionTerm attribute), 483
symbolic (sfepy.terms_diffusion.LaplaceTerm attribute), 484
sync() (sfepy.base.multproc_mpi.MPILogFile method), 184
sync_module_docs module, 594
system() (sfepy.config.Config method), 159
T tags (in module sfepy.base.multproc_mpi), 185
tan_mod_function() (sfepy.terms_fibres.FibresActiveTLTerm static method), 505
tan_mod_function() (sfepy.terms_hyperelastic_tl.BulkActiveTLTerm static method), 507
tan_mod_function() (sfepy.terms_hyperelastic_tl.BulkPenaltyTLTerm static method), 507
tan_mod_function() (sfepy.terms_hyperelastic_tl.GenYeohTLTerm method), 509
tan_mod_function() (sfepy.terms_hyperelastic_tl.MooneyRivlinTLTerm static method), 510
tan_mod_function() (sfepy.terms_hyperelastic_tlNeoHookeanTLTerm static method), 511
tan_mod_function() (sfepy.terms_hyperelastic_tl.OgdenTLTerm method), 512
tan_mod_function() (sfepy.terms_hyperelastic_ul.BulkPenaltyULTerm method), 515
tan_mod_function() (sfepy.terms_hyperelastic_ul.MooneyRivlinULTerm static method), 517
tan_mod_function() (sfepy.terms_hyperelastic_ul.NeoHookeanULTerm static method), 518
tan_mod_u_function() (sfepy.terms_hyperelastic_tl.BulkPressureTLTerm static method), 508
tan_mod_u_function() (sfepy.terms_hyperelastic_tl.BulkPressureULTerm static method), 516
TCorrectorsPressureViaPressureEVP (class in sfepy.homogenization.coefs_elastic), 334
TCorrectorsRSViaPressureEVP (class in sfepy.homogenization.coefs_elastic), 334
TCorrectorsViaPressureEVP (class in sfepy.homogenization.coefs_base), 333
tdim (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 243
tensor_plane_stress() (sfepy.mechanics.matcoefs.TransformToPlane method), 360
tensor_product() (in module sfepy.discrete.iga.iga), 325
Term (class in sfepy.terms), 447
term_ns_asm_convect() (in module sfepy.terms_extmods.terms), 570
Index
update_supported_formats() (in module sfepy.discrete.fem.meshio), 286

update_time_stepper() (sfepy.discrete.problem.Problem method), 224

use_first_available() (in module sfepy.solvers.solvers), 435

use_method_with_name() (in module sfepy.base.base), 168

user_options (build_helpers.NoOptionsDocs attribute), 589

UserMeshIO (in module sfepy.discrete.fem.meshio), 285

V

validate (sfepy.base.goptions.ValidatedDict attribute), 175

validate() (sfepy.base.conf.ProblemConf method), 173

validate_bool() (in module sfepy.base.goptions), 175

ValidatedDict (in module sfepy.base.goptions), 175

value (sfepy.base.multiproc_mpi.RemoteInt attribute), 184

values() (sfepy.base.goptions.ValidatedDict method), 175

var (in module sfepy.discrete.fem.meshio), 286

Variable (class in sfepy.discrete.variables), 232

Variables (class in sfepy.discrete.variables), 233

VariableTimeStepper (class in sfepy.solvers.ts), 435

VectorDotGradScalarTerm (class in module sfepy.terms.terms_dot), 490

VectorDotScalarTerm (in module sfepy.terms.terms_dot), 491

VelocityVerletTS (class in sfepy.solvers.ts_solvers), 445

verbosity (sfepy.terms.terms_multilinear.ETermBase attribute), 537

verify_task_dof_maps() (in module sfepy.parallel.parallel), 388

version_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_0_user attribute), 409

version_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_1_user attribute), 412

version_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_2_user attribute), 416

version_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_3_user attribute), 420

version_number (sfepy.solvers.ls_mumps.mumps_struc_c_5_0_user attribute), 424

vertex_groups (sfepy.discrete.common.extmods.cmesh.CMesh attribute), 243

vertices (sfepy.discrete.common.region.Region property), 257

view_petsc_local() (in module sfepy.parallel.parallel), 388

visit_stack() (in module sfepy.discrete.parse_regions), 207

volume (class in sfepy.mesh.geom_tools), 379

VolumeFractions (class in sfepy.homogenization.coefs_base), 333

VolumeSurfaceTerm (class in sfepy.terms.terms_basic), 464

VolumeSurfaceTLTerm (class in sfepy.terms.hyperelastic_tl), 513

VolumeTerm (class in sfepy.terms.terms_basic), 465

VolumeTLTerm (class in sfepy.terms.hyperelastic_tl), 514

VolumeULTerm (class in sfepy.terms.hyperelastic_ul), 518

VolumeXTerm (class in sfepy.terms.termsCompat), 472

W

wait_for_tag() (in module sfepy.base.multiproc_mpi), 185

wandering_element() (in module sfepy.discrete.simplex_cubature), 227

wave_speeds_from_youngpoisson() (in module sfepy.mechanics.matcoefs), 361

weak_dp_function() (sfepy.terms.hyperelastic_tl.BulkPressureTLTerm static method), 508

weak_dp_function() (sfepy.terms.hyperelastic_ul.BulkPressureULTerm static method), 516

weak_function() (sfepy.terms.hyperelastic_tl.BulkPressureTLTerm static method), 508

weak_function() (sfepy.terms.hyperelastic_tl.HyperElasticTLBase static method), 510

weak_function() (sfepy.terms.hyperelastic_ul.BulkPressureULTerm static method), 516

weak_function() (sfepy.terms.hyperelastic_ul.HyperElasticULBase static method), 516

weak_function() (sfepy.terms.membrane.TLMembraneTerm static method), 526

weak_function() (sfepy.terms.termsCompat.TLMembraneTerm static method), 526

write() (sfepy.base.ioutils.HDF5Data method), 176

write() (sfepy.base.ioutils.SoftLink method), 177

write() (sfepy.base.multiproc_mpi.MPILogFile method), 184

write() (sfepy.discrete.fem.mesh.Mesh method), 278
write() (sfepy.discrete.fem.meshio.ANSYSCDBMeshIO method), 279
write() (sfepy.discrete.fem.meshio.ComsolMeshIO method), 279
write() (sfepy.discrete.fem.meshio.GmshIO method), 281
write() (sfepy.discrete.fem.meshio.HDF5MeshIO method), 283
write() (sfepy.discrete.fem.meshio.HDF5XdmfMeshIO method), 283
write() (sfepy.discrete.fem.meshio.HypermeshAsciiMeshIO method), 283
write() (sfepy.discrete.fem.meshio.MeshIO method), 285
write() (sfepy.discrete.fem.meshio.MeshLibIO method), 285
write() (sfepy.discrete.fem.meshio.NEUMeshIO method), 285
write() (sfepy.discrete.fem.meshio.UserMeshIO method), 286
write() (sfepy.discrete.fem.meshio.XYZMeshIO method), 286
write_control_net() (sfepy.mesh.splinebox.SplineBox method), 385
write_control_polygon_vtk() (sfepy.mesh.bspline.BSplineSurf method), 376
write_data() (sfepy.base.ioutils.DataSoftLink method), 176
write_data() (sfepy.base.ioutils.HDF5Data method), 176
write_dict_hdf5() (in module sfepy.base.ioutils), 179
write_domain_to_hdf5() (sfepy.discrete.iga.domain.IGDoman method), 313
write_iga_data() (in module sfepy.discrete.iga.io), 326
write_log() (in module sfepy.base.log), 182
write_mesh_to_hdf5() (sfepy.discrete.fem.meshio.HDF5MeshIO static method), 283
write_problem (sfepy.solvers.ls_mumps.mumps_struc_c_4 attribute), 409
write_problem (sfepy.solvers.ls_mumps.mumps_struc_c_5_0 attribute), 413
write_problem (sfepy.solvers.ls_mumps.mumps_struc_c_5_1 attribute), 416
write_problem (sfepy.solvers.ls_mumps.mumps_struc_c_5_2 attribute), 420
write_problem (sfepy.solvers.ls_mumps.mumps_struc_c_5_3 attribute), 424
write_results() (in module sfepy.discrete.probes), 211
write_sparse_matrix_hdf5() (in module sfepy.base.ioutils), 179
write_sparse_matrix_to_hdf5() (in module sfepy.base.ioutils), 179
write_surface_vtk() (sfepy.mesh.bspline.BSplineSurf method), 376
write_to_hdf5() (in module sfepy.base.ioutils), 180
write_vtk_to_file() (in module sfepy.postprocess.utils_vtk), 395
write_xdmf_file() (sfepy.discrete.fem.meshio.HDF5MeshIO static method), 283
X
XYZMeshIO (class in sfepy.discrete.fem.meshio), 286
Y
youngpoisson_from_stiffness() (in module sfepy.mechanics.matcoefs), 362
youngpoisson_from_wave_speeds() (in module sfepy.mechanics.matcoefs), 362
Z
zerodofs() (sfepy.discrete.conditions.Conditions method), 190
zero_dofs() (sfepy.discrete.conditions.EssentialBC method), 191
ZeroTerm (class in sfepy.terms.terms_basic), 465
Index 683