SfePy - Write Your Own FE Application

Robert Cimrman∗†

Abstract—SfePy (Simple Finite Elements in Python) is a framework for solving
various kinds of problems (mechanics, physics, biology, ...) described by partial
differential equations in two or three space dimensions by the finite element
method. The paper illustrates its use in an interactive environment or as a
framework for building custom finite-element based solvers.

Index Terms—partial differential equations, finite element method, Python

1 INTRODUCTION

SfePy (Simple Finite Elements in Python) is a multi-platform
(Linux, Mac OS X, Windows) software released under the
New BSD license, see http://sfepy.org. It implements one of
the standard ways of discretizing partial differential equations
(PDEs) which is the finite element method (FEM) [R1].

So far, SfePy has been employed in modeling in materi-
als science, including, for example, multiscale biomechanical
modeling (bone, muscle tissue with blood perfusion) [R2],
[R3], [R4], [R5], [R6], [R7], computation of acoustic transmis-
sion coefficients across an interface of arbitrary microstructure
geometry [R8], computation of phononic band gaps [R9],
[R10], finite element formulation of Schrödinger equation
[R11], and other applications, see also Figure 3. The software
can be used as

• a collection of modules (a library) for building custom or
domain-specific applications,
• a highly configurable “black box” PDE solver with prob-
lem description files in Python.

In this paper we focus on illustrating the former use by
using a particular example. All examples presented below were
tested to work with the version 2013.3 of SfePy.

2 DEVELOPMENT

The SfePy project uses Git for source code management and
GitHub web site for the source code hosting and developer
interaction, similarly to many other scientific python tools.
The version 2013.3 has been released in September 18, 2013,
and its git-controlled sources contained 761 total files, 138191
total lines (725935 added, 587744 removed) and 3857 commits
done by 15 authors. However, about 90% of all commits were
done by the main developer as most authors contributed only a
few commits. The project seeks and encourages more regular
contributors, see http://sfepy.org/doc-devel/development.html.

3 SHORT DESCRIPTION

The code is written mostly in Python. For speed in general, it
relies on fast vectorized operations provided by NumPy [R12]
arrays, with heavy use of advanced broadcasting and "index
tricks" features. C and Cython [R13] are used in places where
vectorization is not possible, or is too difficult/unreadable.
Other components of the scientific Python software stack
are used as well, among others: SciPy [R14] solvers and
algorithms, Matplotlib [R15] for 2D plots, Mayavi [R16] for
3D plots and simple postprocessing GUI, IPython [R17] for
a customized shell, SymPy [R18] for symbolic operations/code
generation etc.

The basic structure of the code allows a flexible definition of
various problems. The problems are defined using components
directly corresponding to mathematical counterparts present in
a weak formulation of a problem in the finite element setting:
a solution domain and its sub-domains (regions), variables be-
longing to suitable discrete function spaces, equations as sums
of terms (weak form integrals), various kinds of boundary
conditions, material/constitutive parameters etc.

The key notion in SfePy is a term, which is the smallest
unit that can be used to build equations (linear combinations of
terms). It corresponds to a weak formulation integral and takes
usually several arguments: (optional) material parameters, a
single virtual (or test) function variable and zero or more state
(or unknown) variables. The available terms (currently 105)
are listed at our web site (http://sfepy.org/doc-devel/terms_
overview.html). The already existing terms allow to solve
problems from many scientific domains, see Figure 3. Those
terms cover many common PDEs in continuum mechanics,
poromechanics, biomechanics etc. with a notable exception of
electromagnetism (work in progress, see below).

SfePy discretizes PDEs using a continuous Galerkin ap-
proximation with finite elements as defined in [R1]. Discon-
tinuous element-wise constant approximation is also possible.
Currently the code supports the 2D area (triangle, rectangle)
and 3D volume (tetrahedron, hexahedron) elements. Structural
elements like shells, plates, membranes or beams are not
supported with a single exception of a hyperelastic Mooney-
Rivlin membrane.

Several kinds of basis or shape functions can be used for
the finite element approximation of the physical fields:
• the classical nodal (Lagrange) basis can be used with all element types;
• the hierarchical (Lobatto) basis can be used with tensor-product elements (rectangle, hexahedron).

Although the code can provide the basis function polynomials of a high order (10 or more), orders greater than 4 are not practically usable, especially in 3D. This is caused by the way the code assembles the element contributions into the global matrix - it relies on NumPy vectorization and evaluates the element matrices all at once and then adds to the global matrix - this allows fast term evaluation and assembling but its drawback is a very high memory consumption for high polynomial orders. Also, the polynomial order has to be uniform over the whole (sub)domain where a field is defined. Related to polynomial orders, tables with quadrature points for numerical integration are available or can be generated as needed.

We are now working on implementing Nédélec and Raviart-Thomas vector bases in order to support other kinds of PDEs, such as the Maxwell equations of electromagnetism.

Once the equations are assembled, a number of solvers can be used to solve the problem. SfePy provides a unified interface to many standard codes, for example UMFPACK [R19], PETSc [R20], Pysparse [R21] as well as the solvers available in SciPy. Various solver classes are supported: linear, nonlinear, eigenvalue, optimization, and time stepping.

Besides the external solvers mentioned above, several solvers are implemented directly in SfePy. Nonlinear/optimization solvers:
• The Newton solver with a backtracking line-search is the default solver for all problems. For simplicity we use a unified approach to solve both the linear and non-linear problems - former (should) converge in a single nonlinear solver iteration.
• The steepest descent optimization solver with a backtracking line-search can be used as a fallback optimization solver when more sophisticated solvers fail.

Time-stepping-related solvers:
• The stationary solver is used to solve time-independent problems.
• The equation sequence solver is a stationary solver that analyzes the dependencies among equations and solves smaller blocks first. An example would be the thermoelasticity problem described below, where the elasticity equation depends on the load given by temperature distribution, but the temperature distribution (Poisson equation) does not depend on deformation. Then the temperature distribution can be found first, followed by the elasticity problem with the already known temperature load. This greatly reduces memory usage and improves speed of solution.
• The simple implicit time stepping solver is used for (quasistatic) time-dependent problems, using a fixed time step.
• The adaptive implicit time stepping solver can change the time step according to a user provided function. The default function adapt_time_step() decreases the step in case of bad Newton convergence and increases the step (up to a limit) when the convergence is fast. It is convenient for large deformation (hyperelasticity) problems.
• The explicit time stepping solver can be used for dynamic problems.

4 Thermoelasticity Example

This example involves calculating a temperature distribution in an object followed by an elastic deformation analysis of the object loaded by the thermal expansion and boundary displacement constraints. It shows how to use SfePy in a script/interactively. The actual equations (weak form) are described below. The entire script consists of the following steps:

Import modules. The SfePy package is organized into several sub-packages. The example uses:
• sfepy.fem: the finite element method (FEM) modules
• sfepy.terms: the weak formulation terms - equations building blocks
• sfepy.solvers: interfaces to various solvers (SciPy, PETSc, ...)
• sfepy.postprocess: post-processing & visualization based on Mayavi

```python
import numpy as np
from sfepy.fem import (Mesh, Domain, Field, FieldVariable, Material, Integral, Equation, Equations, ProblemDefinition)
from sfepy.terms import Term
from sfepy.fem.conditions import Conditions, EssentialBC
from sfepy.solvers.ls import ScipyDirect
from sfepy.solvers.nls import Newton
from sfepy.postprocess import Viewer

Load a mesh file defining the object geometry.
mesh = Mesh.from_file('meshes/2d/square_tri2.mesh')
domain = Domain(('domain', mesh))

Define solution and boundary conditions domains, called regions.
omega = domain.create_region('Omega', 'all')
left = domain.create_region('Left',
    'vertices in x < -0.999',
    'facet')
right = domain.create_region('Right',
    'vertices in x > 0.999',
    'facet')
bottom = domain.create_region('Bottom',
    'vertices in y < -0.999',
    'facet')
top = domain.create_region('Top',
    'vertices in y > 0.999',
    'facet')

Save regions for visualization.
domain.save_regions_as_groups('regions.vtk')
```

Use a quadratic approximation for temperature field, define unknown \( T \) and test \( s \) variables.
```python
field_t = Field.from_args('temperature', np.float64,
    'scalar', omega, 2)
t = FieldVariable('t', 'unknown', field_t, 1)
s = FieldVariable('s', 'test', field_t, 1,
    primary_var_name='t')
```
Define numerical quadrature for the approximate integration rule.

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

define Laplace equation governing the temperature distribution:

\[ \int_{\Omega} \nabla^2 T = 0. \]

term = Term.new('dw_laplace(s, t)', integral, omega, s-s, s=t)

eq = Equation('temperature', term)
eqs = Equations([eq])

Set boundary conditions for the temperature: \( T = 10 \) on \( \Gamma_{\text{left}} \), \( T = 30 \) on \( \Gamma_{\text{right}} \).

t_left = EssentialBC('t_left',
    'left', {'t.0' : 10.0})
t_right = EssentialBC('t_right',
    'right', {'t.0' : 30.0})

Create linear (ScipyDirect) and nonlinear solvers (Newton).

ls = ScipyDirect({})
nls = Newton({})

Combine the equations, boundary conditions and solvers to form a full problem definition.

pb = ProblemDefinition('temperature', equations=eqs, forms=terms, nls=nls, ls=ls)

Solve the temperature distribution problem to get \( T \).

pb.solve()

Use a linear approximation for displacement field, define unknown \( u \) and test \( v \) variables. The variables are vectors with two components in any point, as we are solving on a 2D domain.

field_u = Field.from_args('displacement', np.float64, 'vector', omega, 1)
u = FieldVariable('u', 'unknown', field_u, mesh.dim, primary_var_name='u')
v = FieldVariable('v', 'test', field_u, mesh.dim, primary_var_name='v')

Set Lamé parameters of elasticity \( \lambda, \mu \), thermal expansion coefficient \( \alpha \), and background temperature \( T_0 \). Constant values are used here. In general, material parameters can be given as functions of space and time.

lam = 10.0 # Lamé parameters.
m = Material('m', lam=lam, mu=mu,
    m=mu, alpha=te * eye_sym)

Define and set the temperature load variable to \( T - T_0 \).

t2 = FieldVariable('t', 'parameter', field_u, 1,
    primary_var_name='(set-to-None)')
t2.set_data(t() - T0)

Define the thermoelasticity equation governing structure deformation:

\[ \int_{\Omega} D_{ijkl} \epsilon_{ij}(u) \epsilon_{kl}(u) - \int_{\Gamma} \left( T - T_0 \right) \alpha \epsilon_{ij}(u) = 0, \quad \forall \epsilon, \]

where \( D_{ijkl} = \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) + \lambda \delta_{ij} \delta_{kl} \) is the homogeneous isotropic elasticity tensor and \( \epsilon_{ij}(u) = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \) is the small strain tensor. The equations can be built as linear combinations of terms.

term1 = Term.new('dw_lin_elastic_iso(m.lam, m.mu, v, u)
    integral, omega, m=m, v-v, u-u, u-u)
supporting SfePy modules) is available when needed. The definitions are used to construct and initialize in an automatic way the corresponding objects, similarly to what was presented in the example above, and the problem is solved. The main script for running a simulation described in a PDF is called simple.py.

5.1 Example: Temperature Distribution

This example defines the problem of temperature distribution on a 2D rectangular domain. It directly corresponds to the temperature part of the thermoelasticity example, only for the sake of completeness a definition of a material coefficient is shown as well.

```python
from sfepy import data_dir
filename_mesh = data_dir + '/meshes/2d/square_tri2.mesh'

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

regions = {
    'Omega' : 'all',
    'Left' : ('vertices in (x < -0.999)', 'facet'),
    'Right' : ('vertices in (x > 0.999)', 'facet'),
}

fields = {
    'temperature' : ('real', 1, 'Omega', 2),
}

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

ebcs = {
    't_left' : ('Left', {'t.0' : 10.0}),
    't_right' : ('Right', {'t.0' : 30.0}),
}

integrals = {
    'il' : ('v', 2),
}

equations = {
    'eq' : 'dw_laplace.ii.Omega(coef.val, s, t) = 0'
}

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

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

Many more examples can be found at http://docs.sfepy.org/gallery/gallery or http://sfepy.org/doc-devel/examples.html.

6 Conclusion

We briefly introduced the open source finite element package SfePy as a tool for building domain-specific FE-based solvers as well as a black-box PDE solver.

6.1 Support

Work on SfePy is partially supported by the Grant Agency of the Czech Republic, projects P108/11/0853 and 101/09/1630.

References

[R1] Thomas J. R. Hughes, The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Dover Publications, 2000.
[R2] R.-Cimrman and E.-Rohan. Two-scale modeling of tissue perfusion problem using homogenization of porous media. International Journal for Multiscale Computational Engineering, 8(1):81–102, 2010.
[R3] E.-Rohan, R.-Cimrman, S.-Naili, and T.-Lemaire. Multiscale modeling of compact bone based on homogenization of double porous medium. In Computational Plasticity X - Fundamentals and Applications, 2009.
[R4] E.-Rohan and R.-Cimrman. Multiscale finite element simulation of diffusion-deformation processes in homogenized dual-porous media. Mathematics and Computers in Simulation, 82(10):1744–1772, 2012.
[R5] R.-Cimrman and E.-Rohan. On modelling the parallel diffusion flow in deforming porous media. Mathematics and Computers in Simulation, 76(1-3):34–43, 2007.
[R6] E.-Rohan and R.-Cimrman. Multiscale finite element simulation of diffusion-deformation processes in homogenized dual-porous media. Mathematics and Computers in Simulation, 82(10):1744–1772, 2012.
[R7] E.-Rohan, S.-Naili, R.-Cimrman, and T.-Lemaire. Hierarchical homogenization of fluid saturated porous solid with multiple porosity scales. Comptes Rendus - Mecanique, 340(10):688–694, 2012.
[R8] E.-Rohan and V.-Lukeš. Homogenization of the acoustic transmission through a perforated layer. Journal of Computational and Applied Mathematics, 234(6):1876–1885, 2010.
[R9] E.-Rohan, B.-Miara, and F.-Seifrit. Numerical simulation of acoustic band gaps in homogenized elastic composites. International Journal of Engineering Science, 47(4):573–594, 2009.
[R10] E.-Rohan and B.-Miara. Band gaps and vibration of strongly heterogeneous reissner-mindlin elastic plates. Comptes Rendus Mathematique, 349(13-14):777–781, 2011.
[R11] R.-Cimrman, J.-Vackář, M.-Novák, O.-Čertík, E.-Rohan, and M.-Tůma. Finite element code in python as a universal and modular tool applied to kohn-sham equations. In ECCOMAS 2012 - European Congress on Computational Methods in Applied Sciences and Engineering, e-Book Full Papers, pages 5212–5221, 2012.
[R12] T. E. Oliphant. Python for scientific computing. Computing in Science & Engineering, 9(3):10–20, 2007. http://www.numpy.org.
[R13] R. Bradshaw, S. Behnel, D. S. Seljebotn, G. Ewing, et al. The Cython compiler. http://cython.org.
[R14] E. Jones, T. E. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001-. http://www.scipy.org.
Fig. 3: Gallery of applications. Perfusion and acoustic images by Vladimír Lukeš.
