SfePy - Simple Finite Elements in Python

Short Introduction . . .

Robert Cimrman\textsuperscript{1} Ondřej Čertík\textsuperscript{2}

\textsuperscript{1}Department of Mechanics & New Technologies Research Centre
University of West Bohemia in Plzeň, Czech Republic

\textsuperscript{2}Institute of Physics, Academy of Sciences of the Czech Republic
& Charles University in Prague, Czech Republic

Introduction

- SfePy = simple finite elements in Python
  - general finite element analysis software
  - solving systems of PDEs
- BSD open-source license
- available at
  - http://sfepy.org (developers)
    - mailing lists, issue (bug) tracking
    - we encourage and support everyone who joins!
  - http://sfepy.kme.zcu.cz (project information)
- selected applications:
  - homogenization of porous media (parallel flows in a deformable porous medium)
  - acoustic band gaps (homogenization of a strongly heterogenous elastic structure: phononic materials)
  - shape optimization in incompressible flow problems
  - finite element formulation of Schrödinger equation
## Introduction

- **SfePy** = simple finite elements in Python
  - general finite element analysis software
  - solving systems of PDEs
- BSD open-source license
- available at
  - [http://sfepy.org](http://sfepy.org) (developers)
    - mailing lists, issue (bug) tracking
    - we encourage and support everyone who joins!
  - [http://sfepy.kme.zcu.cz](http://sfepy.kme.zcu.cz) (project information)
- selected applications:
  - homogenization of porous media (parallel flows in a deformable porous medium)
  - acoustic band gaps (homogenization of a strongly heterogenous elastic structure: phononic materials)
  - shape optimization in incompressible flow problems
  - finite element formulation of Schrödinger equation
### Notes on Programming Languages

#### Rough Division

- **compiled (fortran, C, C++, Java, ...)**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• speed</td>
<td>• (often) complicated build process, recompile after any change</td>
</tr>
<tr>
<td>• large code base (legacy codes)</td>
<td>• low-level ⇒ lots of lines to get basic stuff done</td>
</tr>
<tr>
<td>• tradition</td>
<td>• code size ⇒ maintenance problems</td>
</tr>
<tr>
<td></td>
<td>• static!</td>
</tr>
</tbody>
</table>

- **interpreted or scripting (sh, tcl, matlab, perl, ruby, python, ...)**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• no compiling</td>
<td>• many are relatively new</td>
</tr>
<tr>
<td>• (very) high-level ⇒ a few of lines to get (complex) stuff done</td>
<td>• not known as useful in many scientific communities</td>
</tr>
<tr>
<td>• code size ⇒ easy maintenance</td>
<td>• lack of speed</td>
</tr>
<tr>
<td></td>
<td>• dynamic</td>
</tr>
<tr>
<td></td>
<td>• (often) large code base</td>
</tr>
</tbody>
</table>
Notes on Programming Languages

Rough Division

- compiled (fortran, C, C++, Java, ...)

**Pros**
- speed
- large code base (legacy codes)
- tradition

**Cons**
- (often) complicated build process, recompile after any change
- low-level ⇒ lots of lines to get basic stuff done
- code size ⇒ maintenance problems
- static!

- interpreted or scripting (sh, tcl, matlab, perl, ruby, python, ...)

**Pros**
- no compiling
- (very) high-level ⇒ a few of lines to get (complex) stuff done
- code size ⇒ easy maintenance
- dynamic!
- (often) large code base

**Cons**
- many are relatively new
- not known as useful in many scientific communities
- lack of speed
- lack of speed
# Notes on Programming Languages

## Rough Division

- **compiled** (fortran, C, C++, Java, ...)

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- speed</td>
<td>- (often) complicated build process, recompile after any change</td>
</tr>
<tr>
<td>- large code base (legacy codes)</td>
<td>- low-level ⇒ lots of lines to get basic stuff done</td>
</tr>
<tr>
<td>- tradition</td>
<td>- code size ⇒ maintenance problems</td>
</tr>
</tbody>
</table>

- **interpreted or scripting** (sh, tcl, matlab, perl, ruby, python, ...)

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- no compiling</td>
<td>- many are relatively new</td>
</tr>
<tr>
<td>- (very) high-level ⇒ a few of lines to get (complex) stuff done</td>
<td>- not known as useful in many scientific communities</td>
</tr>
<tr>
<td>- code size ⇒ easy maintenance</td>
<td>- lack of speed</td>
</tr>
<tr>
<td>- dynamic!</td>
<td></td>
</tr>
</tbody>
</table>
Notes on Programming Languages
Rough Division

- compiled (fortran, C, C++, Java, ...)

**Pros**
- speed
- large code base (legacy codes)
- tradition

**Cons**
- (often) complicated build process, recompile after any change
- low-level ⇒ lots of lines to get basic stuff done
- code size ⇒ maintenance problems
- static!

- interpreted or scripting (sh, tcl, matlab, perl, ruby, python, ...)

**Pros**
- no compiling
- (very) high-level ⇒ a few of lines to get (complex) stuff done
- code size ⇒ easy maintenance
- dynamic!
- (often) large code base

**Cons**
- many are relatively new
- not known as useful in many scientific communities
- lack of speed
Mixing Languages — Best of Both Worlds

- **low level code** (C or fortran): element matrix evaluations, costly mesh-related functions, . . .
- **high level code** (Python): logic of the code, particular applications, configuration files, problem description files

www.python.org

SfePy = Python + C (+ fortran)

- **notable features:**
  - small size (complete sources are just about 1.3 MB, July 2008)
  - fast compilation
  - problem description files in pure Python
  - problem description form similar to mathematical description “on paper”
Mixing Languages — Best of Both Worlds

- **low level code** (C or fortran): element matrix evaluations, costly mesh-related functions, ...
- **high level code** (Python): logic of the code, particular applications, configuration files, problem description files

www.python.org

SfePy = Python + C (+ fortran)

- notable features:
  - small size (complete sources are just about 1.3 MB, July 2008)
  - fast compilation
  - problem description files in pure Python
  - problem description form similar to mathematical description “on paper”
Software Dependencies

- to install and use SfePy, several other packages or libraries are needed:
  - **NumPy and SciPy**: free (BSD license) collection of numerical computing libraries for Python
    - enables Matlab-like array/matrix manipulations and indexing
  - other: UMFPACK, Pyparsing, Matplotlib, Pytables (+ HDF5), swig
  - visualization of results: ParaView, MayaVi2, or any other VTK-capable viewer

- **missing**:
  - free (BSD license) 3D mesh generation and refinement tool
  - ... can use netgen, tetgen
Software Dependencies

- to install and use SfePy, several other packages or libraries are needed:
  - **NumPy and SciPy**: free (BSD license) collection of numerical computing libraries for Python
    - enables Matlab-like array/matrix manipulations and indexing
  - other: UMFPACK, Pyparsing, Matplotlib, Pytables (+ HDF5), swig
  - visualization of results: ParaView, MayaVi2, or any other VTK-capable viewer

- **missing**:
  - free (BSD license) 3D mesh generation and refinement tool
  - ... can use netgen, tetgen
Introduction

- problem description file is a regular Python module, i.e. all Python syntax and power is accessible
- consists of entities defining:
  - fields of various FE approximations, variables
  - equations in the weak form, quadratures
  - boundary conditions (Dirichlet, periodic, “rigid body”)
  - FE mesh file name, options, solvers, . . .
- simple example: the Laplace equation:

\[ c \Delta u = 0 \text{ in } \Omega, \quad u = \bar{u} \text{ on } \Gamma, \text{ weak form: } \int_{\Omega} c \nabla u \cdot \nabla v = 0, \quad \forall v \in V_0 \]
Problem Description File
Solving Laplace Equation — FE Approximations

- **mesh** → define FE approximation to $\Omega$:
  
  $\text{filename\_mesh} = \text{'simple.mesh'}$

- **fields** → define space $V_h$:
  
  $$\text{field\_1} = \{$$
  
  'name': 'temperature',
  'dim': (1,1),
  'domain': 'Omega',
  'bases': 'Omega': '3_4_P1'

  $$\}$$
  
  '3_4_P1' means P1 approximation, in 3D, on 4-node FEs (tetrahedra)

- **variables** → define $u_h, v_h$:
  
  $$\text{variables} = \{$$

  'u': ('unknown field', 'temperature', 0),
  'v': ('test field', 'temperature', 'u')

  $$\}$$
Problem Description File
Solving Laplace Equation — FE Approximations

- **mesh** → define FE approximation to \( \Omega \):
  
  ```
  filename_mesh = 'simple.mesh'
  ```

- **fields** → define space \( V_h \):
  ```
  field_1 = {
    'name': 'temperature',
    'dim': (1,1),
    'domain': 'Omega',
    'bases': 'Omega': '3_4_P1'
  }
  
  '3_4_P1' means P1 approximation, in 3D, on 4-node FEs (tetrahedra)

- **variables** → define \( u_h, v_h \):
  ```
  variables = {
    'u': ('unknown field', 'temperature', 0),
    'v': ('test field', 'temperature', 'u'),
  }
  ```
Problem Description File
Solving Laplace Equation — FE Approximations

- **mesh**: define FE approximation to $\Omega$:
  
  ```python
  filename_mesh = 'simple.mesh'
  ```

- **fields**: define space $V_h$:
  
  ```python
  field_1 = {
    'name': 'temperature',
    'dim': (1,1),
    'domain': 'Omega',
    'bases': 'Omega': '3_4_P1'
  }

  '3_4_P1' means P1 approximation, in 3D, on 4-node FEs (tetrahedra)

- **variables**: define $u_h$, $v_h$:
  
  ```python
  variables = {
    'u': ('unknown field', 'temperature', 0),
    'v': ('test field', 'temperature', 'u'),
  }
  ```
Problem Description File
Solving Laplace Equation — Boundary Conditions

- **regions** → define domain $\Omega$, regions $\Gamma_{\text{left}}$, $\Gamma_{\text{right}}$, $\Gamma = \Gamma_{\text{left}} \cup \Gamma_{\text{right}}$:
  - $h$ omitted from now on ...

```python
regions = {
    'Omega': ('all', {}),
    'Gamma_Left': ('nodes in (x < 0.0001)', {}),
    'Gamma_Right': ('nodes in (x > 0.0999)', {}),
}
```

- **Dirichlet BC** → define $\bar{u}$ on $\Gamma_{\text{left}}$, $\Gamma_{\text{right}}$:

```python
ebcs = {
    't_left': ('Gamma_Left', 'u.0': 2.0),
    't_right': ('Gamma_Right', 'u.all': -2.0),
}
```
Problem Description File
Solving Laplace Equation — Boundary Conditions

- **regions** → define domain Ω, regions Γ_left, Γ_right, Γ = Γ_left ∪ Γ_right:
  - h omitted from now on . . .

regions = {
  'Omega': ('all', {}),
  'Gamma_Left': ('nodes in (x < 0.0001)', {}),
  'Gamma_Right': ('nodes in (x > 0.0999)', {}),
}

- **Dirichlet BC** → define ¯u on Γ_left, Γ_right:

ebcs = {
  't_left': ('Gamma_Left', 'u.0': 2.0),
  't_right': ('Gamma_Right', 'u.all': -2.0),
}
Problem Description File
Solving Laplace Equation — Equations

- **materials** → define \( c \):
  
  ```python
  material_1 = {
    'name' : 'm',
    'mode' : 'here',
    'region' : 'Omega',
    'c' : 1.0,
  }
  ```

- **integrals** → define numerical quadrature:
  
  ```python
  integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'quadrature' : 'gauss_o1_d3',
  }
  ```

- **equations** → define what and where should be solved:
  
  ```python
  equations = {
    'eq' : 'dw_laplace.i1.Omega( m.c, v, u ) = 0'
  }
  ```
Problem Description File
Solving Laplace Equation — Equations

- **materials** → define $c$:
  
  ```python
  material_1 = {
    'name' : 'm',
    'mode' : 'here',
    'region' : 'Omega',
    'c' : 1.0,
  }
  ```

- **integrals** → define numerical quadrature:
  
  ```python
  integral_1 = {
    'name' : 'i1',
    'kind' : 'v',
    'quadrature' : 'gauss_o1_d3',
  }
  ```

- **equations** → define what and where should be solved:
  
  ```python
  equations = {
    'eq' : 'dw_laplace.i1.Omega( m.c, v, u ) = 0'
  }
  ```
Problem Description File
Solving Laplace Equation — Equations

- **materials** → define \( c \):
  
  ```python
  material_1 = {
    'name': 'm',
    'mode': 'here',
    'region': 'Omega',
    'c': 1.0,
  }
  ```

- **integrals** → define numerical quadrature:
  
  ```python
  integral_1 = {
    'name': 'i1',
    'kind': 'v',
    'quadrature': 'gauss_o1_d3',
  }
  ```

- **equations** → define what and where should be solved:
  
  ```python
  equations = {
    'eq': 'dw_laplace.i1.Omega( m.c, v, u ) = 0'
  }
  ```
$ . / s i m p l e . p y i n p u t / p o i s s o n . p y$

sfepy: reading mesh...
sfepy: ...done in 0.02 s
sfepy: setting up domain edges...
sfepy: ...done in 0.02 s
sfepy: setting up domain faces...
sfepy: ...done in 0.02 s
sfepy: creating regions...
sfepy: leaf Gamma_Right region_4
sfepy: leaf Omega region_1000
sfepy: leaf Gamma_Left region_03
sfepy: ...done in 0.07 s
sfepy: equation "Temperature":
sfepy: dw_laplace.i1.Omega( coef.val, s, t ) = 0
sfepy: describing geometries...
sfepy: ...done in 0.01 s
sfepy: setting up dof connectivities...
sfepy: ...done in 0.00 s
sfepy: using solvers:

    nls: newton
    ls: ls
sfepy: matrix shape: (300, 300)
sfepy: assembling matrix graph...
sfepy: ...done in 0.01 s
sfepy: matrix structural nonzeros: 3538 (3.93e−02% fill)
sfepy: updating materials...
sfepy: coef
sfepy: ...done in 0.00 s
sfepy: nls: iter: 0, residual: 1.176265e−01 (rel: 1.000000e+00)
sfepy: reidual: 0.00 [s]
sfepy: solve: 0.01 [s]
sfepy: matrix: 0.00 [s]
sfepy: nls: iter: 1, residual: 9.921082e−17 (rel: 8.434391e−16)

- The top level of SfePy code is a collection of executable scripts tailored for various applications.
- `simple.py` is a dumb script of brute force, attempting to solve any equations it finds by the Newton method.
- ...exactly what we need here (solver options were omitted in previous slides).
### Top-level Scripts

**Main scripts / applications:**

- `runTests.py` ... run all/selected unit tests
- `simple.py` ... generic problem solver, both for stationary and time-dependent problems
- `eigen.py` ... application: acoustic band gaps in strongly heterogenous media
- `schroedinger.py` ... application: Schrödinger equation solver

**Auxiliary:**

- `extractor.py` ... extract results stored in a HDF5 file, dump results to VTK
- `findSurf.py` ... extract a mesh surface, mark its components
- `gen` ... (re-)generate documentation, found in `doc/sfepy_manual.pdf`, requires additional packages: pexpect, lxml
- `genPerMesh.py` ... scale and periodically repeat a reference volume mesh
Top-level Scripts

Main scripts / applications:

- **runTests.py** ... run all/selected unit tests
- **simple.py** ... generic problem solver, both for stationary and time-dependent problems
- **eigen.py** ... application: acoustic band gaps in strongly heterogenous media
- **schroedinger.py** ... application: Schrödinger equation solver

Auxiliary:

- **extractor.py** ... extract results stored in a HDF5 file, dump results to VTK
- **findSurf.py** ... extract a mesh surface, mark its components
- **gen** ... (re-)generate documentation, found in doc/sfepy_manual.pdf, requires additional packages: pexpect, lxml
- **genPerMesh.py** ... scale and periodically repeat a reference volume mesh
Verification of Numerical Results

- to verify numerical results we use method of **manufactured solutions**: for example, for Poisson’s equation \( \text{div}(\text{grad}(u)) = f \):
  1. make up a solution, e.g. \( u = \sin 3x \cos 4y \)
  2. compute corresponding \( f \), here \( f = 25 \sin 3x \cos 4y \), and boundary conditions by substituting \( u \) into the equation
  3. solve numerically and compare the **exact** solution of the strong problem with the numerical solution of the weak problem

\[ \rightarrow \] allows to assess both the discretization and numerical errors

- manual derivation of \( f \) tedious \( \rightarrow \) SymPy
  - each term class annotated by a corresponding symbolic expression
  - example: anisotropic diffusion term

\[
\text{symbolic} = \{ \text{'expression': 'div( K * grad( u ) )'}, \\
\text{'map' : {'u' : 'state', 'K' : 'material'}}\}
\]

- \( f \) is built by substituting the manufactured solution into the expressions and subsequent evaluation in FE nodes
- work in progress
Verification of Numerical Results

- to verify numerical results we use method of manufactured solutions: for example, for Poisson’s equation \( \text{div}(\text{grad}(u)) = f \):
  1. make up a solution, e.g. \( u = \sin 3x \cos 4y \)
  2. compute corresponding \( f \), here \( f = 25 \sin 3x \cos 4y \), and boundary conditions by substituting \( u \) into the equation
  3. solve numerically and compare the exact solution of the strong problem with the numerical solution of the weak problem
     → allows to assess both the discretization and numerical errors
- manual derivation of \( f \) tedious → SymPy
  - each term class annotated by a corresponding symbolic expression
  - example: anisotropic diffusion term

\[
\text{symbolic} = \{ \ 'expression': \ '\text{div}( K * \text{grad}( u ))', \\
            'map': \ { 'u': \ 'state', 'K': \ 'material' } \}
\]

- \( f \) is built by substituting the manufactured solution into the expressions and subsequent evaluation in FE nodes
- work in progress
Optimal Flow Problem

Problem Setting

Objective Function

\[ \Psi(u) \equiv \frac{\nu}{2} \int_{\Omega_c} |\nabla u|^2 \rightarrow \text{min} \]

- minimize gradients of solution (e.g. losses) in \( \Omega_c \subset \Omega \)
- by moving design boundary \( \Gamma \subset \partial \Omega \)
- perturbation of \( \Gamma \) by vector field \( \mathcal{V} \)

\[ \Omega(t) = \Omega + \{ t \mathcal{V}(x) \}_{x \in \Omega} \quad \text{where} \quad \mathcal{V} = 0 \text{ in } \overline{\Omega_c} \cup \partial \Omega \setminus \Gamma \]
Optimal Flow Problem

Example Results

- flow and domain control boxes, left: initial, right: final

- $\Omega_C$ between two grey planes

- work in progress . . .
Direct Problem

... paper ↔ input file

- **weak form** of Navier-Stokes equations: \( \mathbf{u} \in V_0(\Omega), p \in L^2(\Omega) \)
  such that

\[
a_\Omega (\mathbf{u}, \mathbf{v}) + c_\Omega (\mathbf{u}, \mathbf{u}, \mathbf{v}) - b_\Omega (\mathbf{v}, p) = g_{\Gamma_{\text{out}}} (\mathbf{v}) \quad \forall \mathbf{v} \in V_0, \\
b_\Omega (\mathbf{u}, q) = 0 \quad \forall q \in L^2(\Omega).
\]

- in **SfePy** syntax:

```python
equations = {
    'balance': "
    dw_div_grad.i2.Omega( fluid.viscosity, v, u )
    + dw_convevt.i2.Omega( v, u )
    - dw_grad.i1.Omega( v, p ) = 0""",
    'incompressibility': "
    dw_div.i1.Omega( q, u ) = 0""",
}
```
Direct Problem

...paper ↔ input file

- **weak form** of Navier-Stokes equations: \( u \in V_0(\Omega), \, p \in L^2(\Omega) \) such that

\[
a_\Omega (u, v) + c_\Omega (u, u, v) - b_\Omega (v, p) = g_{\Gamma_{\text{out}}} (v) \quad \forall v \in V_0, \\
b_\Omega (u, q) = 0 \quad \forall q \in L^2(\Omega).
\]

- in SfePy syntax:

```python
equations = {
    'balance': """"
    dw_div_grad.i2.Omega( fluid.viscosity, v, u )
    + dw_convevt.i2.Omega( v, u )
    - dw_grad.i1.Omega( v, p ) = 0""",
    'incompressibility': """"
    dw_div.i1.Omega( q, u ) = 0""",
}
```
Adjoint Problem

... paper ↔ input file

- **KKT conditions** $\delta_{u,p} \mathcal{L} = 0$ yield adjoint state problem for $w, r$:

  $$
  \delta_u \mathcal{L} \circ v = 0 = \delta_u \Psi(u, p) \circ v \\
  + a_\Omega(v, w) + c_\Omega(v, u, w) + c_\Omega(u, v, w) + b_\Omega(v, r),
  $$

  $$
  \delta_p \mathcal{L} \circ q = 0 = \delta_p \Psi(u, p) \circ q - b_\Omega(w, q), \forall v \in V_0, \text{ and } \forall q \in L^2(\Omega).
  $$

- in **SfePy** syntax:

  ```python
  equations = {
    'balance': ""
    dw_div_grad.i2.Omega( fluid.viscosity, v, w ) 
    + dw_adj_conve...t2.i2.Omega( v, w, u ) 
    + dw_grad.i1.Omega( v, r ) 
    = - '\delta_u \Psi(u, p) \circ v',

    'incompressibility': ""
    dw_div.i1.Omega( q, w ) = 0"",
  }
  ```
KKT conditions $\delta_{u,p}\mathcal{L} = 0$ yield adjoint state problem for $w, r$:

$$\delta_u \mathcal{L} \circ v = 0 = \delta_u \Psi(u, p) \circ v$$

$$+ a_\Omega(v, w) + c_\Omega(v, u, w) + c_\Omega(u, v, w) + b_\Omega(v, r) ,$$

$$\delta_p \mathcal{L} \circ q = 0 = \delta_p \Psi(u, p) \circ q - b_\Omega(w, q) , \forall v \in V_0, \text{ and } \forall q \in L^2(\Omega).$$

in SfePy syntax:

```python
equations = {
    'balance': """"""
    dw_div_grad.i2.Omega( fluid.viscosity, v, w )
    + dw_adj_convecept1.i2.Omega( v, w, u )
    + dw_adj_convecept2.i2.Omega( v, w, u )
    + dw_grad.i1.Omega( v, r )
    = - 'δuΨ(u,p)∘v',""",
    'incompressibility': """"""
    dw_div.i1.Omega( q, w ) = 0""",
}
```
Finite Element Formulation of Schrödinger Equation

One particle Schrödinger equation:

$$\left(-\frac{\hbar^2}{2m} \nabla^2 + V\right) \psi = E\psi.$$  

FEM:

$$(K_{ij} + V_{ij}) q_j = E M_{ij} q_j + F_i,$$

$$V_{ij} = \int \phi_i V \phi_j \, dV,$$

$$M_{ij} = \int \phi_i \phi_j \, dV,$$

$$K_{ij} = \frac{\hbar^2}{2m} \int \nabla \phi_i \cdot \nabla \phi_j \, dV,$$

$$F_i = \frac{\hbar^2}{2m} \oint \frac{d\psi}{dn} \phi_i \, dS.$$  

Usually we set $F_i = 0.$
Particle in the Box

\[ V(x) = \begin{cases} 0, & \text{inside the box } a \times a \times a \\ \infty, & \text{outside} \end{cases} \]

Analytic solution:

\[ E_{n_1n_2n_3} = \frac{\pi^2}{2a^2} \left( n_1^2 + n_2^2 + n_3^2 \right) \]

where \( n_i = 1, 2, 3, \ldots \) are independent quantum numbers. We chose \( a = 1 \), i.e.: \( E_{111} = 14.804 \), \( E_{211} = E_{121} = E_{112} = 29.608 \),
\( E_{122} = E_{212} = E_{221} = 44.413 \), \( E_{311} = E_{131} = E_{113} = 54.282 \)
\( E_{222} = 59.217 \), \( E_{123} = E_{\text{perm.}} = 69.087 \).

Numerical solution (\( a = 1 \), 24702 nodes):

<table>
<thead>
<tr>
<th>E</th>
<th>1</th>
<th>2-4</th>
<th>5-7</th>
<th>8-10</th>
<th>11</th>
<th>12-</th>
</tr>
</thead>
<tbody>
<tr>
<td>theory</td>
<td>14.804</td>
<td>29.608</td>
<td>44.413</td>
<td>54.282</td>
<td>59.217</td>
<td>69.087</td>
</tr>
<tr>
<td>FEM</td>
<td>14.861</td>
<td>29.833</td>
<td>44.919</td>
<td>55.035</td>
<td>60.123</td>
<td>70.305</td>
</tr>
<tr>
<td></td>
<td>29.834</td>
<td>44.920</td>
<td>55.042</td>
<td>70.310</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29.836</td>
<td>44.925</td>
<td>55.047</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3D Harmonic Oscillator

\[ V(r) = \begin{cases} \frac{1}{2} \omega^2 r^2, & \text{inside the box} \quad a \times a \times a \\ \infty, & \text{outside} \end{cases} \]

Analytic solution in the limit \( a \to \infty \):

\[ E_{nl} = \left( 2n + l + \frac{3}{2} \right) \omega \]

where \( n, l = 0, 1, 2, \ldots \). Degeneracy is \( 2l + 1 \), so: \( E_{00} = \frac{3}{2} \), triple \( E_{01} = \frac{5}{2}, E_{10} = \frac{7}{2} \), quintuple \( E_{02} = \frac{7}{2} \) triple \( E_{11} = \frac{9}{2} \), quintuple \( E_{12} = \frac{11}{2} \):

Numerical solution (\( a = 15, \omega = 1, 290620 \) nodes):

<table>
<thead>
<tr>
<th>E</th>
<th>1</th>
<th>2-4</th>
<th>5-10</th>
<th>11-</th>
</tr>
</thead>
<tbody>
<tr>
<td>theory</td>
<td>1.5</td>
<td>2.5</td>
<td>3.5</td>
<td>4.5</td>
</tr>
<tr>
<td>FEM</td>
<td>1.522</td>
<td>2.535</td>
<td>3.554</td>
<td>4.578</td>
</tr>
<tr>
<td></td>
<td>2.536</td>
<td>3.555</td>
<td>4.579</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.536</td>
<td>3.555</td>
<td>4.579</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.555</td>
<td>3.555</td>
<td>4.579</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.556</td>
<td>3.556</td>
<td>4.579</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.556</td>
<td>3.556</td>
<td>4.579</td>
<td></td>
</tr>
</tbody>
</table>
3D Harmonic Oscillator

Eigenvalues:

- 0th
- 10th
- 12th
Hydrogen Atom

\[ V(r) = \begin{cases} 
-\frac{1}{r}, & \text{inside the box \quad } a \times a \times a \\
\infty, & \text{outside}
\end{cases} \]

Analytic solution in the limit \( a \to \infty \):

\[ E_n = -\frac{1}{2n^2} \]

where \( n = 1, 2, 3, \ldots \). Degeneracy is \( n^2 \), so: \( E_1 = -\frac{1}{2} = -0.5 \), \( E_2 = -\frac{1}{8} = -0.125 \), \( E_3 = -\frac{1}{18} = -0.055 \), \( E_4 = -\frac{1}{32} = -0.031 \).

Numerical solution (\( a = 15 \), 160000 nodes):

<table>
<thead>
<tr>
<th>E</th>
<th>1</th>
<th>2.5</th>
<th>6-14</th>
<th>15-</th>
</tr>
</thead>
<tbody>
<tr>
<td>theory</td>
<td>-0.5</td>
<td>-0.125</td>
<td>-0.055</td>
<td>-0.031</td>
</tr>
<tr>
<td>FEM</td>
<td>-0.481</td>
<td>-0.118</td>
<td>-0.006</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Hydrogen Atom

11th eigenvalue (calculated: -0.04398532, exact: -0.056), on the mesh with 976 691 tetrahedrons and 163 666 nodes, for the hydrogen atom (V=-1/r).
Conclusion

What is done

- basic FE element engine:
  - finite-dimensional approximations of continuous fields
  - variables, boundary conditions, FE assembling
  - equations, terms, regions
  - materials, material caches
- various solvers accessed via abstract interface
- unit tests, automatic documentation generation
- mostly linear problems, but multiphysical

What is not done

- general FE engine, possibly with symbolic evaluation (SymPy)
- good documentation
- fast problem-specific solvers (!)
- adaptive mesh refinement (!)
- parallelization (petsc4py)

What will not be done (?)

- GUI
- real symbolic parsing/evaluation of equations

http://sfepy.org
Conclusion

What is done

- basic FE element engine:
  - finite-dimensional approximations of continuous fields
  - variables, boundary conditions, FE assembling
  - equations, terms, regions
  - materials, material caches
- various solvers accessed via abstract interface
- unit tests, automatic documentation generation
- mostly linear problems, but multiphysical

What is not done

- general FE engine, possibly with symbolic evaluation (SymPy)
- good documentation
- fast problem-specific solvers (!)
- adaptive mesh refinement (!)
- parallelization (petsc4py)

What will not be done (?)

- GUI
- real symbolic parsing/evaluation of equations

http://sfepy.org
Conclusion

What is done

- basic FE element engine:
  - finite-dimensional approximations of continuous fields
  - variables, boundary conditions, FE assembling
  - equations, terms, regions
  - materials, material caches

- various solvers accessed via abstract interface

- unit tests, automatic documentation generation

- mostly linear problems, but multiphysical

What is not done

- general FE engine, possibly with symbolic evaluation (SymPy)

- good documentation

- fast problem-specific solvers (!)

- adaptive mesh refinement (!)

- parallelization (petsc4py)

What will not be done (?)

- GUI

- real symbolic parsing/evaluation of equations

http://sfepy.org
## Conclusion

### What is done

- basic FE element engine:
  - finite-dimensional approximations of continuous fields
  - variables, boundary conditions, FE assembling
  - equations, terms, regions
  - materials, material caches

- various solvers accessed via abstract interface

- unit tests, automatic documentation generation

- mostly linear problems, but multiphysical

### What is not done

- general FE engine, possibly with symbolic evaluation (SymPy)

- good documentation

- fast problem-specific solvers (!)

- adaptive mesh refinement (!)

- parallelization (petsc4py)

### What will not be done (?)

- GUI

- real symbolic parsing/evaluation of equations

http://sfepy.org
Acknowledgements

The work on various parts of SfePy has been supported by the following grants and research projects funded by several agencies in the Czech Republic:

- **Robert Cimrman:**
  - grant project GAČR 101/07/1471, entitled “Finite element modelling of linear, non-linear and multiscale effects in wave propagation in solids and heterogeneous media”
  - research project MŠMT 1M06031
  - research project MŠMT 4977751303

- **Ondřej Čertík:**
  - research center project LC06040
  - grant project GAČR IAA100100637
This is not a slide!

1Do you like Monty Python’s Flying Circus? It helps! (Python FAQ 1.1.17)