

PySE – Python Stencil Environment

Åsmund Ødegård

Simula Research Laboratory

FEniCS'05, Chicago, October 19-20, 2005

Outline

- 1 General Ideas of PySE
- 2 A simple example
- 3 Some details on the interfaces
- 4 Performance of PySE

Outline

- 1 General Ideas of PySE
- 2 A simple example
- 3 Some details on the interfaces
- 4 Performance of PySE

Basic features of PySE

High-level tool for rapid development of FDM solvers.

- High-level syntax, Matlab-like.
- Code close to the math or pseudo code.
- Easy deployment on parallel computers.
- Written in python, uses extension modules for better performance.
- Available at <http://pyfdm.sourceforge.net>.
- Former know as paraStencils and pyFDM.

Priorities: 1. Abstractions, 2. Parallelization, 3. Efficiency

Some related works

PySE use ideas and concepts from many other tools:

- Diffpack
- *hypr*
- A++/P++
- cogito

- PETSc
- Trilinos
- Chombo

The abstractions

PySE defines the following abstractions.

- Grid; for the domain and FDM mesh.
- Field; for scalar fields over a Grid.
- Stencil; the action of the PDE in a point.
- StencilSet; set of stencils for a problem.

The first three abstractions are quite common. Stencil and StencilSet are the most important abstractions in PySE.

Outline

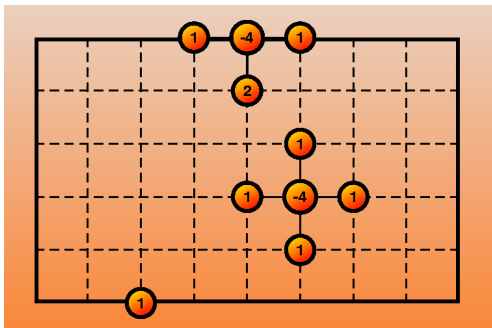
- 1 General Ideas of PySE
- 2 A simple example**
- 3 Some details on the interfaces
- 4 Performance of PySE

Consider a simple Heat equation:

$$\begin{aligned}
 u_t &= \nabla^2 u & x \in \Omega \\
 u(x, 0) &= f(x), & x \in \Omega \\
 \frac{\partial u(x, t)}{\partial n} &= g_n(x, t), & x \in \partial\Omega_n \\
 u(x, t) &= g_d(x, t), & x \in \partial\Omega_d
 \end{aligned}$$

Assume further that we want to solve this on the unit square with f and g given as initialfunc and neumannfunc, respectively.

A simple example, graphically



- Assume A is the StencilSet.
- One explicit step: $u_{n+1} = A(u_n)$.

Example code

This short code solves the problem on the previous slide:

```
from pyFDM import *

def neumannfunc(x,y): return sin(x)*cos(y)

def initialfunc(x,y): return sin(x)*cos(y)

g = Grid(domain=( [0,1, [0,1]], division=(100,100))
u = Field(g)
t = 0; dt = T/n;
A = StencilSet(g)
innerstencil = Identity(g.nsd) + dt*Laplace(g)
innerind = A.addStencil(innerstencil, g.innerPoints())
A += createNeumannBoundary(innerstencil, g, neumannfunc)
u.fill(initialfunc)
for t < T:
    u = A(u)
    t += dt
plot(u)
```

Example code

Some remarks on the code

- Laplace and Identity are stencils defined in PySE
- The Neumann condition function is not time dependent
- It can be made time dependent by wrapping into a lambda:

```
def neumannfunc(x,y,t): return x*y*t

rt = 0
neumanncall = lambda x,y: neumannfunc(x,y,rt)
A += createNeumannBoundary(innerstencil, g, neumanncall)
while rt < T:
    u = A(u)
    t += dt
    A.updateDataStructures()
```

Outline

- 1 General Ideas of PySE
- 2 A simple example
- 3 Some details on the interfaces**
- 4 Performance of PySE

Stencil

You can easily build your own stencils

```
h = (dt**2/g.dx**2)
lap_5pt = Stencil(nsd=2, varcoeff=False,\
                 nodes={
                     (0,1): 1.,\
                     (-1,0): 1., (0,0): -4., (1,0): 1.,\
                     (0,-1): 1.})
id = Stencil(nsd=2, varcoeff=False, nodes={(0,0): 1.0})
inner = id + h*lap_5pt
```

Stencils can be added together, scaled, and evaluated

StencilSet

- Stencils are added to a StencilSet together with an iterator for nodes.
- Grid have methods for various sets of nodes:
 - allPoints
 - innerPoints
 - boundary
 - corners
- innerPoints and boundary take an optional *region* argument:

```
A.addStencil(diricond, \  
             grid.boundary(region=((-1,1),(-1,1)), \  
                           type='circle', center=(0,0), \  
                           radius=1, direction='in'))
```

StencilSet

During the first call to the call-operator $A(u)$ in StencilSet, more efficient datastructures are build:

- Why:
 - Walking the iterators is time-consuming (in pure Python).
- Stencil-coefficients are assembled in a sparse matrix.
- Source information is assembled in a vector.
- We need to provide hooks to update for changes in coefficients and source:
 - `updateDataStructures`
 - `updateSourceDatastructures`
- These methods trigger reassembling of all or parts of the data.

StencilSet

- If present in StencilSet instances, the sparse matrix and vector will be used on subsequent calls to $A(u)$ and A^*u .
- `A.direct_matvec(x)`: operate on a NumPy vector, returns a NumPy vector.
 - Less overhead (no Field creation), hence more efficient.
 - The interface for updating datastructures is (at least for now) less convenient in this case.
 - A Field `u` stores its data as `u.data`, a NumPy vector.
 - Remark, a dummy $A(u)$ must be inserted to build datastructures.

Neumann boundary conditions

Creating Neumann boundary conditions can be tricky in the multidimensional case.

- The function `createNeumanBoundary` function can be used:

```
nSet = createNeumanBoundary(stencil, grid, condition)
```

- The Neumann creator also accept a region specification:

```
nSet = createNeumanBoundary(stencil, grid, condition, \
    region=((-1,1),(-1,1)), \
    type='circle', center=(0,0), \
    raduis=1, direction='out')
```

Parallel computations with PySE

All parts of PySE are inherently parallel.

- Parallelism is initiated with
`grid.partition(StencilSet)`
- The StencilSet supplied should be “ready”
- All Fields created on the grid, will be converted.
- Other StencilSets in the grid get the same partitioning with
`StencilSet.doInitParallel()`

Outline

- 1 General Ideas of PySE
- 2 A simple example
- 3 Some details on the interfaces
- 4 Performance of PySE**

A more involved example

Consider the following problem:

$$\begin{aligned}u_t &= \nabla \cdot (k(x, y) \nabla u) + f(x, y, t), & (x, y) \in \Omega, & \quad t \in \mathbb{R}^+, \\u(x, y, t) &= h(x, y, t), & (x, y) \in \partial\Omega, & \quad t \in \mathbb{R}^+, \\u(x, y, 0) &= g(x, y), & (x, y) \in \Omega.\end{aligned}$$

- We chose $f(x, y, t)$, $k(x, y)$, $h(x, y, t)$, and $g(x, y)$ such that

$$u(x, y, t) = e^{-t} \sin(\pi x) \cos(\pi y)$$

- Implementation follow the simple example.

Timing of the solver

# cpus:	1	4	16	24	32
1000 × 1000, 160 step:	7984	1998	498.5	332.0	249.3
speed-up:	1	3.99	16.0	24.0	32.0
1500 × 1500, 240 step:	26820	6728	1681	1125	838.8
speed-up:	1	3.98	15.9	23.8	31.9

CPU time in seconds and corresponding speed-up numbers.

- The solver uses the `direct_matvec` trick

Comparison with a C solver

We have created a (less flexible) solver in C:

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	74.4	2.32
1500 × 1500, 240 steps:	362.4	74.0	2.31

CPU time in seconds for the solver implemented in C, as well as speed– relative to the Python solver running on one and 32 processors.

- For certain applications, this is just fine
- If we do not assemble in sparse matrix and vector, multiply P/C numbers by $O(10)$ (update source vs. all)
- Where do we loose that much?

Comparison with a C solver

We have created a (less flexible) solver in C:

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	74.4	2.32
1500 × 1500, 240 steps:	362.4	74.0	2.31

CPU time in seconds for the solver implemented in C, as well as speed– relative to the Python solver running on one and 32 processors.

- For certain applications, this is just fine
- If we do not assemble in sparse matrix and vector, multiply P/C numbers by O(10) (update source vs. all)
- Where do we lose that much?

Comparison with a C solver

We have created a (less flexible) solver in C:

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	74.4	2.32
1500 × 1500, 240 steps:	362.4	74.0	2.31

CPU time in seconds for the solver implemented in C, as well as speed– relative to the Python solver running on one and 32 processors.

- For certain applications, this is just fine
- If we do not assemble in sparse matrix and vector, multiply P/C numbers by O(10) (update source vs. all)
- Where do we lose that much?

The bottleneck...

In this problem, the Dirichlet boundary condition and the source function are time dependent.

- For each timestep, we walk the iterators to update data.
- If we remove the time dependency (and hence the need for update of data), we get:

Problem size	C	Python
1000 × 1000, 160 time steps:	34.5	30.5

The modules we're using from Python for `mat*vec`, `vec*vec` are obviously smarter than my C program

The bottleneck can be removed!

- NumPy can fill an array with values from a function very fast!
- We can put source and boundary information in Fields, and use additional (static) StencilSet operators.
- Rewrite the explicit update as
$$u = A(u) + S(F) + B(H)$$
- The Fields F and H can be filled quickly with `F.fill_vec(function)`.
- The `direct_matvec` trick improve performance further.

The bottleneck can be removed!

- NumPy can fill an array with values from a function very fast!
- We can put source and boundary information in Fields, and use additional (static) StencilSet operators.

- Rewrite the explicit update as

$$u = A(u) + S(F) + B(H)$$

- The Fields F and H can be filled quickly with `F.fill_vec(function)`.
- The `direct_matvec` trick improve performance further.

The bottleneck can be removed!

- NumPy can fill an array with values from a function very fast!
- We can put source and boundary information in Fields, and use additional (static) StencilSet operators.

- Rewrite the explicit update as

$$\mathbf{u} = \mathbf{A}(\mathbf{u}) + \mathbf{S}(\mathbf{F}) + \mathbf{B}(\mathbf{H})$$

- The Fields \mathbf{F} and \mathbf{H} can be filled quickly with `F.fill_vec(function)`.
- The `direct_matvec` trick improve performance further.

The bottleneck can be removed!

- NumPy can fill an array with values from a function very fast!
- We can put source and boundary information in Fields, and use additional (static) StencilSet operators.

- Rewrite the explicit update as

$$u = A(u) + S(F) + B(H)$$

- The Fields F and H can be filled quickly with `F.fill_vec(function)`.
- The `direct_matvec` trick improve performance further.

The bottleneck can be removed!

- NumPy can fill an array with values from a function very fast!
- We can put source and boundary information in Fields, and use additional (static) StencilSet operators.

- Rewrite the explicit update as

$$u = A(u) + S(F) + B(H)$$

- The Fields F and H can be filled quickly with `F.fill_vec(function)`.
- The `direct_matvec` trick improve performance further.

When the bottleneck is gone, we get good performance

# cpus:	1	4	16	24	32
1000 × 1000, 160 steps:	365.0	93.90	23.51	16.05	12.52
speed-up:	1	3.89	15.5	22.7	29.2
1500 × 1500, 240 steps:	1226	315.7	78.42	52.33	40.73
speed-up:	1	3.88	15.6	23.4	30.1

CPU time in seconds and corresponding speed-up numbers for the improved Python solver.

Comparison with C of the faster solver

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	3.40	0.12
1500 × 1500, 240 steps:	362.4	3.38	0.11

- Is this fast enough?
 - What did we loose? Only nice syntax.
 - What's not there: limited support for higher order methods, no support for non-linear problems, only simple grids
 - How can it be usefull in FEniCS?

Comparison with C of the faster solver

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	3.40	0.12
1500 × 1500, 240 steps:	362.4	3.38	0.11

- Is this fast enough?
- What did we loose? Only nice syntax.
- What's not there: limited support for higher order methods, no support for non-linear problems, only simple grids
- How can it be usefull in FEniCS?

Comparison with C of the faster solver

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	3.40	0.12
1500 × 1500, 240 steps:	362.4	3.38	0.11

- Is this fast enough?
- What did we loose? Only nice syntax.
- What's not there: limited support for higher order methods, no support for non-linear problems, only simple grids
- How can it be usefull in FEniCS?

Comparison with C of the faster solver

Problem size	runtime	1-cpu P/C	32-cpu P/C
1000 × 1000, 160 steps:	107.3	3.40	0.12
1500 × 1500, 240 steps:	362.4	3.38	0.11

- Is this fast enough?
- What did we loose? Only nice syntax.
- What's not there: limited support for higher order methods, no support for non-linear problems, only simple grids
- How can it be usefull in FEniCS?