Optimised finite difference computation from symbolic equations

M. Lange[1] N. Kukreja[1] F. Luporini[1] M. Louboutin[2] C. Yount[3]
J. Hückelheim[1] G. Gorman[1]

June 13, 2017

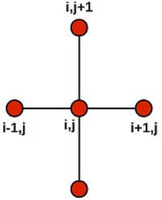
[1]Department of Earth Science and Engineering, Imperial College London, UK
[2]Seismic Lab. for Imaging and Modeling, The University of British Columbia, Canada
[3]Intel Corporation

Imperial College London

Solving simple PDEs is (kind of) easy...

First-order diffusion equation:

```
for ti in range(timesteps):
    t0 = ti % 2
    t1 = (ti + 1) % 2
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            uxx = (u[t0, i+1, j] -2 * u[t0, i, j] + u[t0, i-1, j]) / dx2
            uyy = (u[t0, i, j+1] -2 * u[t0, i, j] + u[t0, i, j-1]) / dy2
            u[t1, i, j] = u[t0, i, j] + dt * a * (uxx + uyy)
```



Symbolic computation is a powerful tool

Solving complicated PDEs is not easy!

12th-order acoustic wave equation:

```
for (int i4 = 0; i4<149; i4+=1) {
  for (int i1 = 6; i1<64; i1++) {
    for (int i2 = 6; i2<64; i2++) {
      for (int i3 = 6; i3<64; i3++) {
        u[i4][i1][i2][i3] = 6.01250601250601e-9F*(2.80896e+8F*damp[i1][i2][i3]*u[i4-2][i1
              ][i2][i3]-3.3264e+8F*m[i1][i2][i3]*u[i4-2][i1][i2][i3]+6.6528e+8F*m[i1][i2][
              i3]*u[i4-1][i1][i2][i3]-2.12255421155556e+7F*u[i4-1][i1][i2][i3
              ]-1.42617283950617e+2F*u[i4-1][i1][i2][i3-6]+2.46442666666667e+3F*u[i4-1][i1
              ][i2][i3-5]-2.11786666666667e+4F*u[i4-1][i1][i2][i3-4]+1.25503209876543e+5F*
              u[i4-1][i1][i2][i3-3]-6.3536e+5F*u[i4-1][i1][i2][i3-2]+4.066304e+6F*u[i4-1][
              i1][i2][i3-1]+4.066304e+6F*u[i4-1][i1][i2][i3+1]-6.3536e+5F*u[i4-1][i1][i2][
              i3+2]+1.25503209876543e+5F*u[i4-1][i1][i2][i3+3]-2.11786666666667e+4F*u[i4
              -1][i1][i2][i3+4]+2.46442666666667e+3F*u[i4-1][i1][i2][i3
              +5]-1.42617283950617e+2F*u[i4-1][i1][i2][i3+6]-1.42617283950617e+2F*u[i4-1][
              i1][i2-6][i3]+2.46442666666667e+3F*u[i4-1][i1][i2-5][i3]-2.11786666666667e+4
              F*u[i4-1][i1][i2-4][i3]+1.25503209876543e+5F*u[i4-1][i1][i2-3][i3]-6.3536e+5
              F*u[i4-1][i1][i2-2][i3]+4.066304e+6F*u[i4-1][i1][i2-1][i3]+4.066304e+6F*u[i4
              -1][i1][i2+1][i3]-6.3536e+5F*u[i4-1][i1][i2+2][i3]+1.25503209876543e+5F*u[i4
              -1][i1][i2+3][i3]-2.11786666666667e+4F*u[i4-1][i1][i2+4][i3
              ]+2.46442666666667e+3F*u[i4-1][i1][i2+5][i3]-1.42617283950617e+2F*u[i4-1][i1
              ][i2+6][i3]-1.42617283950617e+2F*u[i4-1][i1-6][i2][i3]+2.46442666666667e+3F*
              u[i4-1][i1-5][i2][i3]-2.11786666666667e+4F*u[i4-1][i1-4][i2][i3
              ]+1.25503209876543e+5F*u[i4-1][i1-3][i2][i3]-6.3536e+5F*u[i4-1][i1-2][i2][i3
              ]+4.066304e+6F*u[i4-1][i1-1][i2][i3]+4.066304e+6F*u[i4-1][i1+1][i2][i3
              ]-6.3536e+5F*u[i4-1][i1+2][i2][i3]+1.25503209876543e+5F*u[i4-1][i1+3][i2][i3
              ]-2.11786666666667e+4F*u[i4-1][i1+4][i2][i3]+2.46442666666667e+3F*u[i4-1][i1
              +5][i2][i3]-1.42617283950617e+2F*u[i4-1][i1+6][i2][i3])/(1.68888888888889F*
              damp[i1][i2][i3]+2*m[i1][i2][i3]);
    }
  }
```

Symbolic computation is a powerful tool

We can solve PDEs symbolically

- Domain-specific languages provide high levels of abstraction
- Separation of concerns between scientists and computational experts

SymPy: Symbolic computer algebra system in pure Python[1]

Enables automation of stencil generation

- Complex symbolic expressions as Python object trees
- Programmatic manipulation of symbolic expressions
- Built-in code generation for variety of languages
- For a great overview see A. Meurer's talk at SciPy 2016

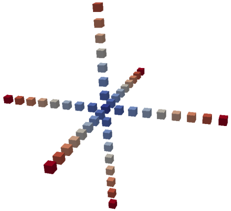
[1] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017

Devito - Automated finite difference propagators

Devito: Finite difference DSL based on SymPy

Devito generates highly optimized stencil code...

- OpenMP threading and vectorisation pragmas
- Cache blocking and auto-tuning
- Symbolic stencil optimization



... from concise mathematical syntax!

Example: acoustic wave equation with dampening

$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \nabla u = 0$$

can be written as

```
eqn = m * u.dt2 + eta * u.dt - u.laplace
```

CFD Python: Step 5 - Linear convection

Governing equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0$$

Discretized:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n})$$

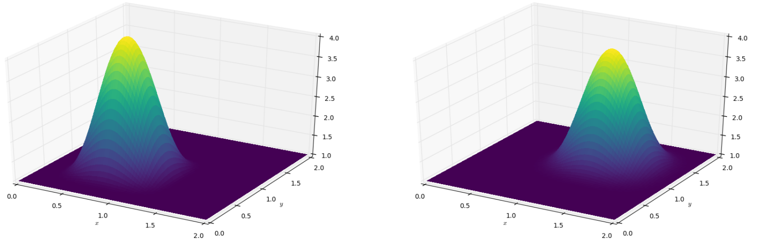
SymPy stencil (assume $\Delta t = s$, $\Delta x = \Delta y = h$):

```
from devito import *
from sympy import solve

c = 1.
grid = Grid(shape=(nx, ny))
u = TimeFunction(name='u', grid=grid)
eq = Eq(u.dt + c * u.dxl + c * u.dyl)
stencil = solve(eq, u.forward)[0]

[In] print(stencil)
[Out] (h*u(t, x, y) - 2.0*s*u(t, x, y)
     + s*u(t, x, y - h) + s*u(t, x - h, y))/h
```

CFD Python: Step 5 - Linear convection

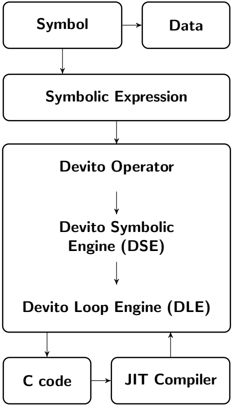
Simple advection example:

```
op = Operator(Eq(u.forward, stencil))

# Set initial condition as a smooth bump
init_smooth(u.data, dx, dy)

op(u=u, time=100, dt=dt)  # Apply for 100 timesteps
```



http://nbviewer.jupyter.org/github/barbagroup/CFDPython/blob/master/lessons/07_Step_5.ipynb

Devito - Automated code optimizations




```
u = TimeFunction(name='u', grid=grid)
m = Function(name='m', grid=grid)
```

High-level function symbols associated with user data

```
eqn = m * u.dt2 - u.laplace
```

Symbolic equations that expand finite difference stencils

```
op = Operator(expressions)
op.apply(time=ntime)
```

Automatic code generation and execution from high-level expressions

Symbolic optimization to reduce computation per stencil point

Loop-level optimization for efficient parallel execution

Just-in-time compilation of optimized C code

Motivation: Inversion problems for seismic imaging

Seismic imaging is a challenging problem for HPC

Big data meets big compute

- Very large amounts of data, huge amount of compute
- HPC architectures, often with accelerators (eg. Intel®Xeon Phi)
- Require highly optimized solvers code

Often use complex finite difference operators

- Different high-order formulations of wave equations
- Unknown topology and high wave frequencies
- Large, complicated stencils, often **written by hand!**

Pure stencil DSLs are not enough

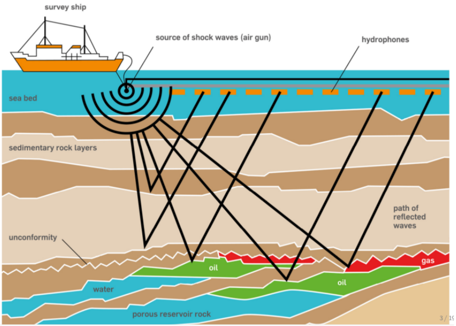
- Generating stencils still has to be done by hand
- Many special-cases that do not fit the "stencil" abstraction

High-performance wave propagators for seismic imaging

The aim is to derive an image of the earth's subsurface

Solve a PDE-constrained optimization problem

- Using wave propagation operators and their adjoints
- Wave is inserted and read at unaligned points
  Inject sparse point interpolation into kernels!



High-performance wave propagators for seismic imaging

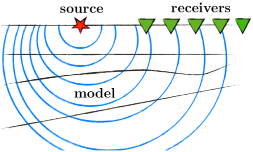
```
def forward(model, m, eta, src, rec, order=2):
    # Create the wavefeld function
    u = TimeFunction(name='u', grid=model.grid,
                     time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)[0]
    update_u = [Eq(u.forward, stencil)]

    # Inject wave as source term
    src_term = src.inject(field=u, expr=src * dt**2 / m)

    # Interpolate wavefield onto receivers
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term)
```

Acoustic wave equation:

$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \nabla u = 0$$

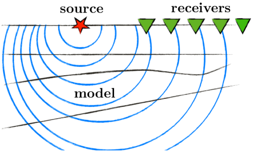
High-performance wave propagators for seismic imaging

```
def forward(model, m, eta, src, rec, order=2):
    # Create the wavefeld function
    u = TimeFunction(name='u', grid=model.grid,
                     time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)[0]
    update_u = [Eq(u.forward, stencil)]

    # Inject wave as source term
    src_term = src.inject(field=u, expr=src * dt**2 / m)

    # Interpolate wavefield onto receivers
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term)
```



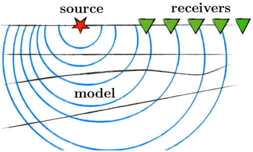
High-performance wave propagators for seismic imaging

```
def gradient(model, m, eta, srca, rec, order=2):
    # Create the adjoint wavefeld function
    v = TimeFunction(name='v', grid=model.grid,
                     time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    eqn = m * v.dt2 - v.laplace — eta * v.dt
    stencil = solve(eqn, u.forward)[0]
    update_v = [Eq(v.backward, stencil)]

    # Inject the previous receiver readings
    rec_term = rec.inject(field=v, expr=rec * dt**2 / m)

    # Gradient update terms
    grad = Function(name='grad', grid=model.grid)
    grad_update = Eq(grad, grad — u.dt2 * v)

    # Create operator with source and receiver terms
    return Operator(update_v + [grad_update] + rec_term,
                    time_axis=Backward)
```



Reverse time migration in $<$ 100 lines of Python

```
# Define acquisition geometry and timestepping
model = Model(...)
dt, nt = <timestepping parameters>
src = RickerSource(...)
rec = Receiver(...)

# Create forward and gradient operators
op_fwd = forward(model, src, rec, order)
op_grad = gradient(model, rec, order)

grad = Function(name='grad', grid=model.grid)

for shot in shots:
    # Create wavefield for forward propagation
    u = TimeFunction(name='u', grid=model.grid,
                     space_order=order)

    # Update source location and compute forward
    src.coordinates.data[0. :] = source_loc[i]
    op_forward(u=u, src=src, rec=rec, m=model.m)

    # Compute gradient update from the residual
    residual = measurement_data - rec.data[:]
    op_gradient(u=u, grad=grad, rec=residual,
                m=model.m)
```

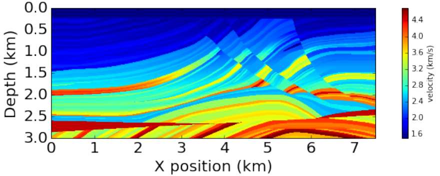


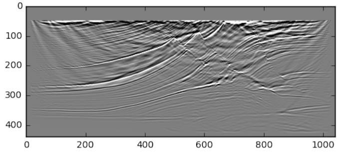
High-performance wave propagators for seismic imaging

Efficient development

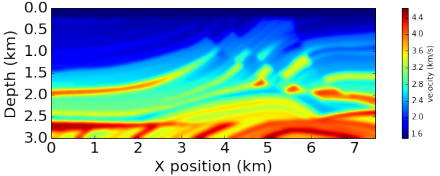
- Test and verify in Python
- Operators in < 20 lines
- RTM loop in < 100 lines
- Variable stencil order

Subsurface model (Marmousi 2D)





RTM subsurface image



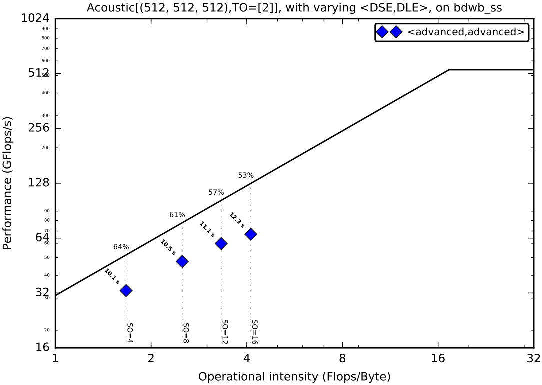
Initial pertubation (Gaussian filter)

http://www.opesci.org/devito/tutorials.html

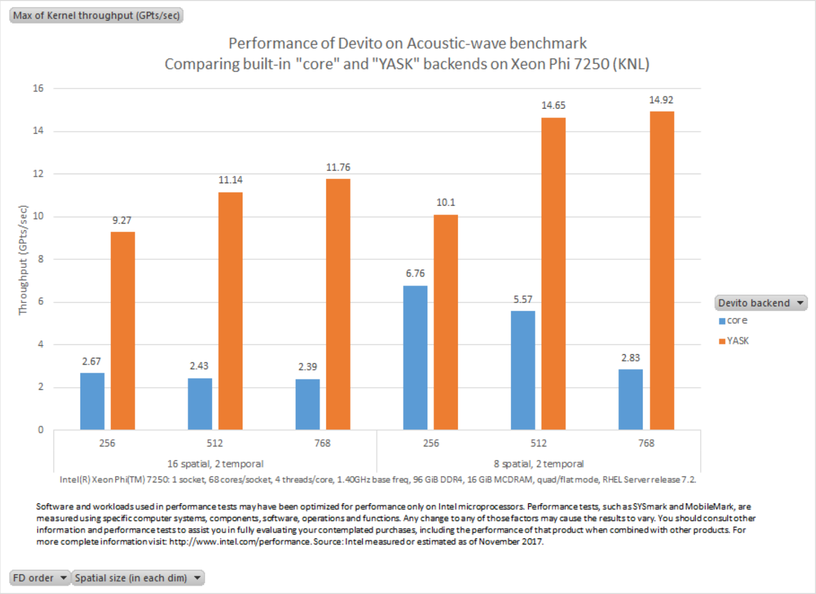
Devito - Performance of acoustic operators

Performance benchmark:

- Second order in time with boundary dampening
- 3D domain (512 × 512 × 512), grid spacing = 20.
- Varying space order (SO)
- Xeon E5-2620 v4 2.1Ghz (Broadwell) 8 cores @ 2.1GHz, single socket



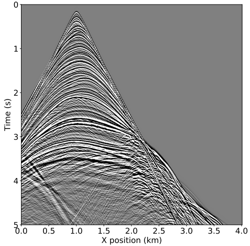
Devito - YASK integration

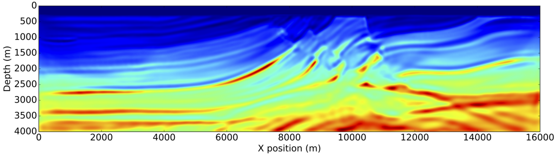


Summary

- **Devito: High-performance finite difference DSL**
  - Symbolic finite difference stencils via SymPy
  - Fully executable via JIT compilation
  - **Increased productivity through high-level API**
  - **Fully composable with scientific Python ecosystem**

- **Fast wave propagators for inversion problems**
  - Seismic inversion operators in < 20 lines
  - Complete problem setups in 200 lines
  - **Automated performance optimisation!**





Thank You

Useful links:

- http://www.opesci.org
- https://github.com/opesci/devito
- http://www.sympy.org

Tutorials:

- Recorded version of this talk given at SciPy17
- Devito tutorials: http://www.opesci.org/devito/tutorials.html
- CFD Python tutorial: http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/

OPESCI

Intel PCC

Y*A*S*K Yet Another Stencil Kernel

EPSRC Engineering and Physical Sciences Research Council

SINBAD

Part of this work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract number DE-AC02-06CH11357.

Imperial College London