Imperial College
London

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

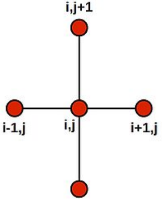
Imperial College London

Symbolic computation is a powerful tool

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]);
    }
  }
```

Inversion problems for seismic imaging

We can solve PDEs symbolically...

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

For high-performance kernels in seismic imaging

Large scale inversion problems

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

Most algorithms use finite difference operators

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

Symbolic computation is a powerful tool

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

Enables automation of stencil generation

- Complex symbolic expressions as Python object trees
- Symbolic manipulation routines and interfaces
- Convert symbolic expressions to numeric functions
  - Python (NumPy) functions; C or Fortran kernels
- For a great overview see A. Meurer's talk at SciPy 2016

For specialised domains generating C code is not enough!

- Compiler-level optimimizaton to leverage performance
- Stencil optimization is a research field of its own

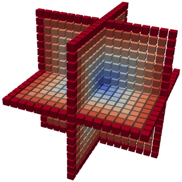
[1] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, 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 optimisation



... 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
```

Computational Fluid Dynamics examples:

http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/

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.
u = TimeData(name='u', shape=(nx, ny))
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
```

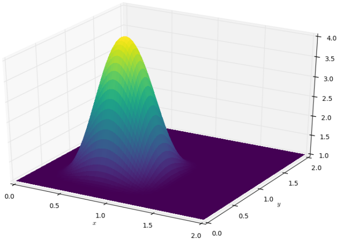
CFD Python: Step 5 - Linear convection

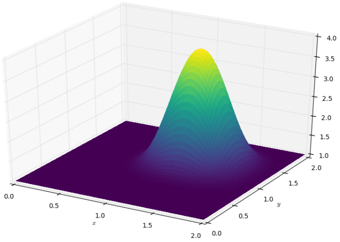
Simple advection example:

```
op = Operator(Eq(u.forward, stencil), subs={h: dx, s:dt})

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

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





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

Imperial College London

CFD Python: Step 9 - Laplace equation

Governing equation:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0$$

Discretized:

$$p^n_{i,j} = \frac{\Delta y^2 (p^n_{i+1,j} + p^n_{i-1,j}) + \Delta x^2 (p^n_{i,j+1} + p^n_{i,j-1})}{2(\Delta x^2 + \Delta y^2)}$$

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

```
# Create two separate symbols with space dimensions
p = DenseData(name='p', shape=(nx, ny), space_order=2)
pn = DenseData(name='pn', shape=(nx, ny), space_order=2)

# Define equation and solve for center point in 'pn'
eq = Eq(pn.dx2 + pn.dy2)
stencil = solve(eq, pn)[0]
# The update expression to populate buffer 'p'
eq_stencil = Eq(p, stencil)
```

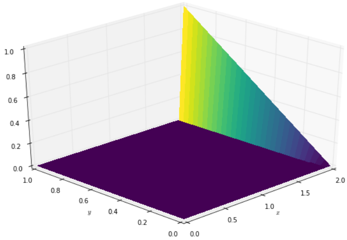
CFD Python: Step 9 - Laplace equation

Boundary conditions:

$$p = 0 \text{ at } x = 0$$

$$p = y \text{ at } x = 2$$

$$\frac{\partial p}{\partial y} = 0 \text{ at } y = 0,\ 1$$



Explicit BCs via expressions:

```
# Create symbol for prescibed BC
bc_r = DenseData(name='bc_r', shape=(nx, ),
                 dimensions=(x, ))
bc_r.data[:] = np.linspace(0, 1, nx)

# Create explicit BC expressions
bc = [Eq(p.indexed[x, 0], 0.)]
bc += [Eq(p.indexed[x, ny-1], bc_r.indexed[x])]
bc += [Eq(p.indexed[0, y], p.indexed[1, y])]
bc += [Eq(p.indexed[nx-1, y], p.indexed[nx-2, y])]

# Build operator with multple expressions
op = Operator([eq_stencil] + bc, subs={h: dx, a: 1.})
```

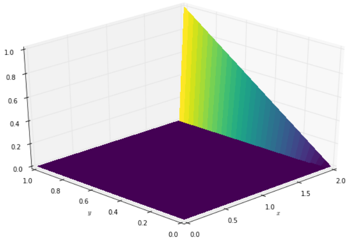
CFD Python: Step 9 - Laplace equation

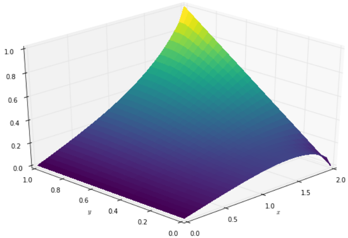
Convergence loop:

```
l1norm = 1; counter = 0
while l1norm > 1.e-4:
    # Determine buffer order
    if counter % 2 == 0:
        _p, _pn = p, pn
    else:
        _p, _pn = pn, p

    # Apply operator
    op(p=_p, pn=_pn)

    l1norm = (np.sum(np.abs(_p.data[:])
             - np.abs(_pn.data[:]))
             / np.sum(np.abs(_pn.data[:])))
    counter += 1
```





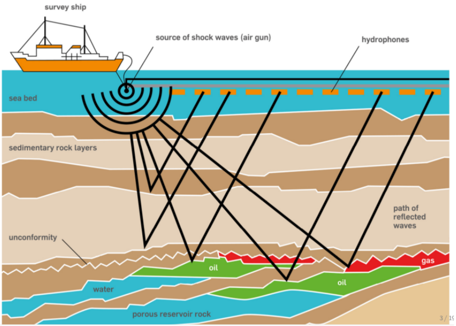
http://nbviewer.jupyter.org/github/barbagroup/CFDPython/blob/master/lessons/12_Step_9.ipynb

High-performance wave propagators for seismic imaging

The aim: Derive image of the earth's sub-surface

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!



Imperial College London

High-performance wave propagators for seismic imaging

```
def forward(model, m, eta, src, rec, order=2):
    # Create the wavefeld function
    u = TimeData(name='u', shape=model.shape
                 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,
                    subs={s: dt, h: model.spacing})
```

High-performance wave propagators for seismic imaging

```
def forward(model, m, eta, src, rec, order=2):
    # Create the wavefeld function
    u = TimeData(name='u', shape=model.shape
                 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,
                    subs={s: dt, h: model.spacing})
```

High-performance wave propagators for seismic imaging

```
def gradient(model, m, eta, srca, rec, order=2):
    # Create the adjoint wavefeld function
    v = TimeData(name='v', shape=model.shape,
                 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 = DenseData(name='grad', shape=model.shape)
    grad_update = Eq(grad, grad - u.dt2 * v)

    # Create operator with source and receiver terms
    return Operator(update_v + [grad_update] + rec_term,
                    subs={s: dt, h: model.spacing},
                    time_axis=Backward)
```

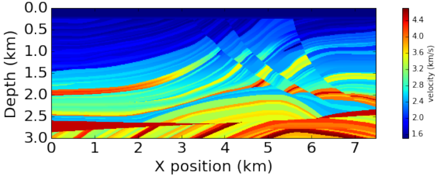
Imperial College London

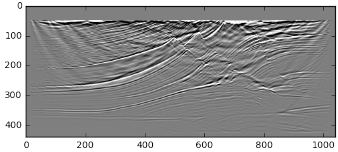
High-performance wave propagators for seismic imaging

Reverse Time Migration

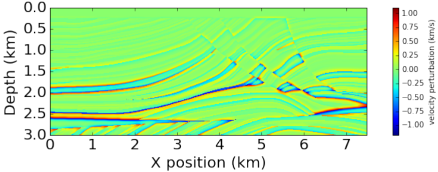
- Synthetic “true” model and a smoothed boundary to invert for
- Use `forward` on synthetic and smooth data to compute residual
- Compute `gradient` to form image

“True” sub-surface model (Marmousi 2D)





Inverted subsurface image



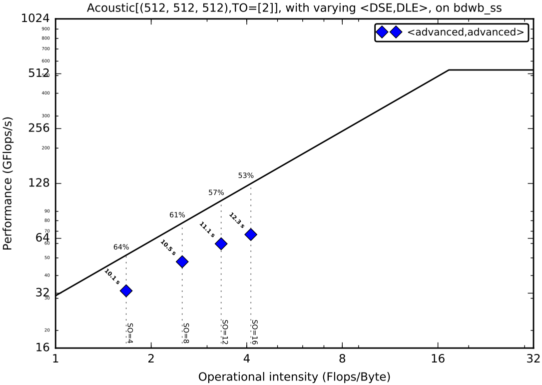
Model pertubation (Gaussian filter)

High-performance wave propagators for seismic imaging

Performance benchmark:

- Second order in time with boundary dampening
- 3D domain ($512 \times 512 \times 512$), grid spacing $= 20$.
- Varying space order (SO)

- Xeon E5-2620 v4 2.1Ghz (Broadwell) 8 cores @ 2.1GHz, single socket



Devito - Automated code optimisations

Devito Symbolic Engine (DSE)

- Common sub-expression elemination (CSE)
- Factorization and time invariant hoisting
- Alias detection (WIP)

Devito Loop Engine (DLE)

- OpenMP and vectorisation via pragmas
- Loop blocking and auto-tuning for block size

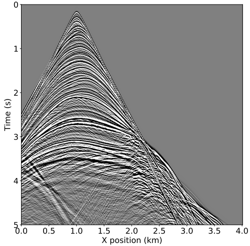
YASK integration (ongoing):

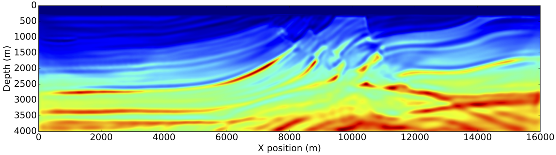
- Yet Another Stencil Kernel
- **Superior performance!**
  - Stencil folding
  - Nested OpenMP and MPI support
- Integrated with DLE as alternative backend

Summary

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

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





Imperial College London

Thank You

Useful links:

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

Tutorials:

- The original CFD Python tutorial: http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/
- Devito implementation: http://www.opesci.org/devito/tutorials.html

Intel PCC

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.