Devito: Towards a generic Finite Difference DSL using Symbolic Python

M. Lange[1] N. Kukreja[2] M. Louboutin[3] F. Luporini[4] F. Vieira[2]
V. Pandolfo[4] P. Velesko[5] P. Kazakas[6] G. Gorman[1]

November 14, 2016

[1]Department of Earth Science and Engineering, Imperial College London, UK
[2]SENAI CIMATEC, Salvador, Brazil
[3]Seismic Lab. for Imaging and Modeling, The University of British Columbia, Canada
[4]Department of Computing, Imperial College London, UK
[5]College of Electrical and Computer Engineering, University of Oklahoma, USA
[6]Department of Computer Science, University of York, UK

Symbolic computation is a powerful tool

OPESCI

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

OPESCI

Solving complicated PDEs is not easy!

12th-order acoustic wave equation:

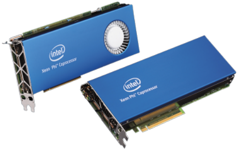
```
for (int i4 = 0; i4<149; i4+=1) {
  for (int i1 = 6; i1<64; i1++) {
    for (int i2 = 6; i2<64; i2++) {
      for (int i3 = 6; i3<64; i3++) {
        u[i4][i1][i2][i3] = 6.01250601250601e-9F*(2.80896e+8F*damp[i1][i2][i3]*u[i4-2][i1
              ][i2][i3]-3.3264e+8F*m[i1][i2][i3]*u[i4-2][i1][i2][i3]+6.6528e+8F*m[i1][i2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

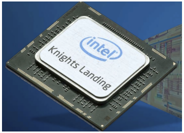
Imperial College London

Symbolic computation is a powerful tool

OPESCI

- **Getting performance on modern hardware is not easy!**
  - Functioning code exists but is not optimised for current hardware
  - **Evolution vs. revolution?**
- **Domain-specific languages (DSL) make revolution easy**
  - **Separate problem definition from implementation**
  - Creates a separate of concerns between scientists and computation experts
- **Performance portability through code-generation**
  - Code is auto-generated and optimised at run-time
  - Platform-specific optimisation for target hardware





Imperial College London

Symbolic computation is a powerful tool

- **Symbolic DSLs for solving PDEs have proven successful**

  FEniCS / Firedrake - Finite element DSL packages

  Velocity-stress formulation of elastic wave equation, with isotropic stress:

$$\rho \frac{\partial \mathbf{u}}{\partial t} = \nabla \cdot \mathbb{T}$$

$$\frac{\partial \mathbb{T}}{\partial t} = \lambda \left(\nabla \cdot \mathbf{u} \right) \mathbb{I} + \mu \left(\nabla \mathbf{u} + \nabla \mathbf{u}^{\mathrm{T}} \right)$$

  Weak form of equations written in UFL[1]:

```
F_u = density*inner(w, (u - u0)/dt)*dx - inner(w, div(s0))*dx
solve(lhs(F_u) == rhs(F_u), u)
```

[1] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer Publishing Company, Incorporated, 2012

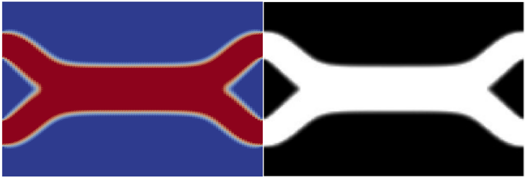
Symbolic computation is a powerful tool

- **Symbolic DSLs for solving PDEs have proven successful**

  Dolfin-Adjoint: Symbolic adjoints from symbolic PDEs[1]

  - Solves complex optimisation problems
  - 2015 Wilkinson prize winner

  Below is the optimal design of a double pipe that minimises the dissipated power in the fluid.



[1] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, 35(4):C369–C393, 2013

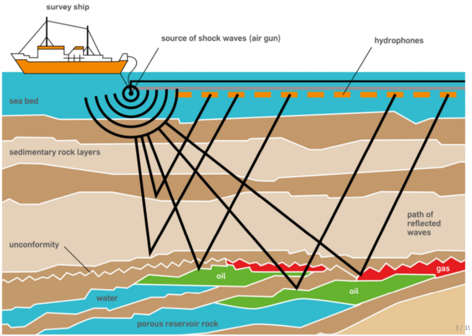
Inversion problems for seismic imaging

For Seismic imaging we need to solve inversion problems

- Finite Difference solvers for forward and adjoint runs
- Different types of wave equations with large complicated stencils

Many stencil languages exist, but few are practical

- Stencil still written by hand!



SymPy - Symbolic computation in Python

- Symbolic computer algebra system (CAS) written in pure Python[1]
- *Features:*
  - Complex symbolic expressions as Python object trees
  - Symbolic manipulation routines and interfaces
  - Convert symbolic expressions to numeric functions
    - Python or NumPy functions
    - C or Fortran kernels
  - For a great overview see A. Meuer's talk at SciPy 2016

For specialised domains generating C code is not enough!

[1] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, Thilina Rathnayake, et al. Sympy: Symbolic computing in python. Technical report, PeerJ Preprints, 2016

OPESCI

Devito - A Finite Difference DSL for seismic imaging

- Aimed at creating fast high-order inversion kernels
- Development is driven by "real-world" problems

Devito is based on SymPy expressions

- Acoustic wave equation:

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

can be defined symbolically as

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

Devito auto-generates optimised C kernel code

- OpenMP threading and vectorisation pragmas
- Cache blocking and auto-tuning
- Symbolic stencil optimisation (eg. CSE, hoisting)

Imperial College London

Devito - A prototype Finite Difference DSL

Devito Data Objects

```
u = TimeData('u', shape=(nx, ny))
m = DenseData('m', shape=(nx, ny))
```

High-level function symbols associated with user data

Stencil Equation

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

Symbolic equations that expand Finite Difference stencils

Devito Operator

```
op = Operator(eqn)
```

Transform stencil expressions into explicit array accesses

Devito Propagator

```
u = op.apply(u.data, m.data)
```

Generate low-level optimized kernel code and apply to data

Devito Compiler

GCC — Clang — Intel®— Intel®Xeon Phi™

```
op.compiler = IntelMIC
```

Compiles and loads Platform specific executable function

Imperial College London

OPESCI

Real-world applications need more than PDE solvers

- File I/O and support for large data sets
- Non-PDE kernel code, eg. sparse point interpolation

Devito follows the principle of Graceful Degradation

- Circumvent restrictions to the high-level API by customisation
- Devito translates high-level PDE-based stencils into "matrix index" format

```
# High-level expression equivalent to f.dx2
(-2*f(x, y) + f(x - h, y) + f(x + h, y)) / h**2

# Low-level expression with explicit indexing
(-2*f[x, y] + f[x - 1, y] + f[x + 1, y]) / h**2
```

- **Allows custom functionality in auto-generated kernels**

Imperial College London

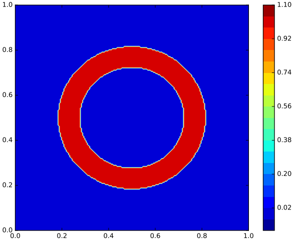
Example - 2D diffusion equation

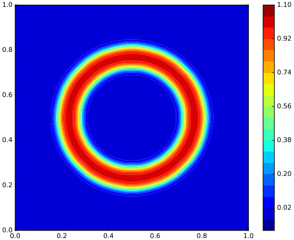
To illustrate let's consider the 2D diffusion equation:

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Example: Smoothing a sharp-edged ring

- Finite difference with 5-point stencil





Example - 2D diffusion equation

We can solve this using Python (slow) . . .

```
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)
```

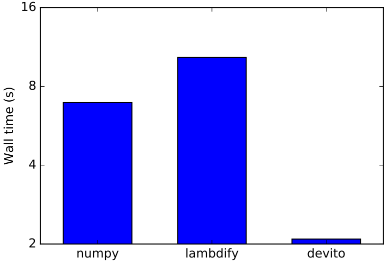
Vectorised NumPy (faster) . . .

```
for ti in range(timesteps):
  t0 = ti % 2
  t1 = (ti + 1) % 2
  # Vectorised version of the diffusion stencil
  uxx = (u[t0,2:,1:-1]-2*u[t0,1:-1,1:-1]+u[t0,:-2,1:-1])/dx2
  uyy = (u[t0,1:-1,2:]-2*u[t0,1:-1,1:-1]+u[t0,1:-1,:-2])/dy2
  u[t1,1:-1,1:-1] = u[t0,1:-1,1:-1] + a * dt * (uxx + uyy)
```

Solve symbolically in Devito:

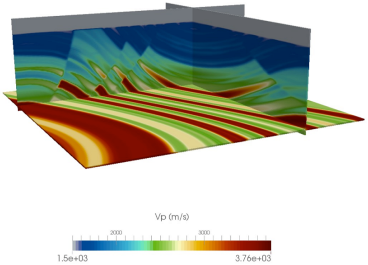
```
u = TimeData(name='u', shape=(nx, ny),
             time_order=1, space_order=2)
u.data[0, :] = ring_initial(spacing=dx)
eqn = Eq(u.dt, a * (u.dx2 + u.dy2))
stencil = solve(eqn, u.forward)[0]
op = Operator(stencils=Eq(u.forward, stencil),
              subs={h: dx, s: dt}, nt=timesteps)
op.apply()
```

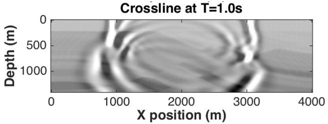
Single core benchmark:

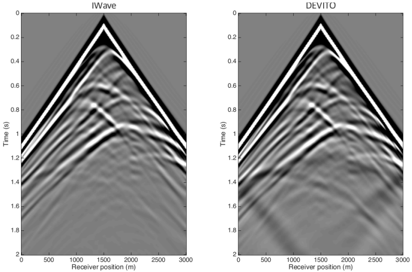


Full Waveform Inversion models

- Acoustic and TTI wave equations of varying spatial order
- Validated against industry data set
- Achieve performance similar to industry leading production code







Example - Full Waveform Inversion

```python
def forward(model, nt, dt, h, order=2):
    shape = model.shape
    m = DenseData(name="m", shape=shape,
                  space_order=order)
    m.data[:] = model
    u = TimeData(name='u', shape=shape,
                 time_dim=nt, time_order=2,
                 space_order=order)
    eta = DenseData(name='eta', shape=shape,
                    space_order=order)

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

    op = Operator(stencils=Eq(u.forward, stencil),
                  nt=nt, subs={s: dt, h: h},
                  shape=shape, forward=True)
    # Source injection code omitted for brevity

    op.apply()
```

Example - Full Waveform Inversion

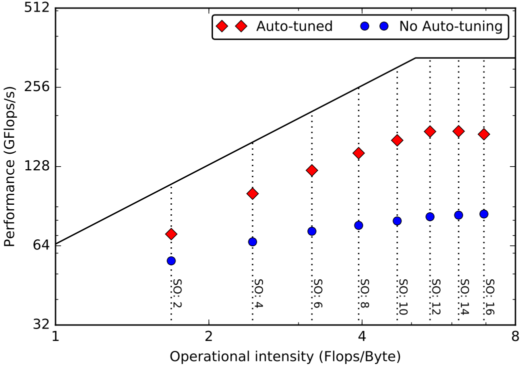
```
def adjoint(model, nt, dt, h, order=2):
    shape = model.shape
    m = DenseData(name="m", shape=shape,
                  space_order=order)
    m.data[:] = model
    v = TimeData(name='v', shape=shape,
                 time_dim=nt, time_order=2,
                 space_order=order)
    eta = DenseData(name='eta', shape=shape,
                    space_order=order)

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

    op = Operator(stencils=Eq(u.backward, stencil),
                  nt=nt, subs={s: dt, h: h},
                  shape=shape, forward=False)
    # Receiver interpolation omitted for brevity

    op.apply()
```

Example - Full Waveform Inversion

Performance of acoustic forward operator

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

- E5-2697 v4 (Broadwell) @ 2.3GHz
- Single socket with 16 cores



Automated code optimisations:

- OpenMP and vectorisation pragmas
- Loop blocking and auto-tuning for block size
- Automated roofline plotting for performance analysis

Symbolic optimisations:

- Common sub-expression elimination:
  - Reduces compilation time from hours to seconds for large stencils
  - Enables further factorisation techniques to reduce flops

Potential future optimisations:

- Polyhedral compilation (time blocking)
- Automated data layout optimisations

Motivation

Devito - A prototype Finite Difference DSL

Example - 2D diffusion equation

Example - Seismic Imaging

Conclusion

Devito - A prototype Finite Difference DSL

- **Devito: A finite difference DSL for seismic imaging**
  - Symbolic problem description (PDEs) via SymPy
  - Low-level API for kernel customisation
  - Automated performance optimisation
- **Devito is driven by real-world scientific problems**
  - **Not "yet another stencil compiler"**
  - **Bridge the gap between stencil compilers and real world applications**
- **Future work includes:**
  - Extend feature range to facilitate more science
  - MPI parallelism for larger models
  - Integrate stencil or polyhedral compiler backends
  - Additional symbolic optimisation (factorisation, hoisting, etc.)
  - Integrate automated verification tools to catch compiler bugs

Thank You

Links:

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

Poster in Lower Lobby Concourse:

- Programming Systems - 39

OPESCI

Intel PCC

Sistema FIEB SENAI
Federação das Indústrias do Estado da Bahia

EPSRC
Engineering and Physical Sciences Research Council

BG GROUP

SINBAD

Imperial College London