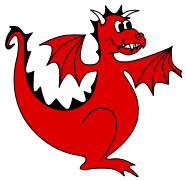


Firedrake

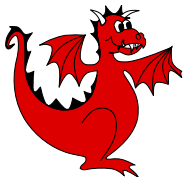
Specifying and solving PDEs

Lawrence Mitchell[1,*] + a cast of tens

18th May 2017

[1]Departments of Computing and Mathematics, Imperial College London
*lawrence.mitchell@imperial.ac.uk

Imperial College London



Firedrake

Specifying and solving PDEs

. . . easily?

Lawrence Mitchell[1,*] + a cast of tens

18th May 2017

[1]Departments of Computing and Mathematics, Imperial College London
*lawrence.mitchell@imperial.ac.uk

Imperial College London

Introduction

A barrier to progress

Lemma

Developing sophisticated numerical models "from scratch" is a lot of work.

Corollary

Many new advances in methods are only tried on simple problems.

Better numerical methods take a long time to move into "real world" application domains.

A play in two acts

- “I need to solve a PDE”

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)
- “That was easy”

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)
- “That was easy”
- “Oh wait, I need parallel/adaptivity/high-order/…”

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)
- “That was easy”
- “Oh wait, I need parallel/adaptivity/high-order/...”
- (3 years writing code)

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)
- “That was easy”
- “Oh wait, I need parallel/adaptivity/high-order/…”
- (3 years writing code)
- “That was easy”

A play in two acts

- “I need to solve a PDE”
- (search for software that solves my problem)
- “All these libraries are too complicated”
- (2 weeks writing code)
- “That was easy”
- “Oh wait, I need parallel/adaptivity/high-order/...”
- (3 years writing code)
- “That was easy”

We should be able to do better.

Why might this occur

"Mongolian is hard, I need to learn all these new words"

Why might this occur

- Evidence that the *abstraction level* is wrong?
- Abstraction level is right, but the library only works for toy problems.

Why might this occur

- Evidence that the *abstraction level* is wrong?
- Abstraction level is right, but the library only works for toy problems.
- Well-designed abstractions present low barrier to entry, without only being useful for toy problems.
- Then, we can build on them, rather than starting from scratch every time.

In the beginning

Compute $y \leftarrow \nabla^2 x$ using finite differences.

$$y_{i,j} = x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j}$$

In the beginning

Compute $y \leftarrow \nabla^2 x$ using finite differences.

$$y_{i,j} = x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j}$$

Before 1953

```
...
faddp   %st, %st(1)
movl    -8(%ebp), %edx
movl    %edx, %eax
sall    $2, %eax
addl    %edx, %eax
leal    0(,%eax,4), %edx
addl    %edx, %eax
sall    $2, %eax
movl    %eax, %edx
movl    -4(%ebp), %eax
addl    %edx, %eax
subl    $101, %eax
flds    x.3305(,%eax,4)
flds    .LC0
fmulp   %st, %st(1)
faddp   %st, %st(1)
fstps   y.3307(,%ecx,4)
...
```

In the beginning

Compute $y \leftarrow \nabla^2 x$ using finite differences.

$$y_{i,j} = x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j}$$

After 1953

```
      PROGRAM MAIN
      PARAMETER (N=100)
      REAL X(N,N), Y(N,N)
      DO 10 J=2,N-1
         DO 20 I=2,N-1
            Y(I,J)=X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1)+4*X(I,J)
20       CONTINUE
10    CONTINUE
      DO 30 I=1,N
         Y(I,1) = 0.0
         Y(I,N) = 0.0
         Y(1,I) = 0.0
         Y(N,I) = 0.0
30    CONTINUE
      END
```

Pros

- Not tied to a particular piece of hardware
- Easier to understand
- Composable

Cons

- Lost full control over program execution
- Compiler may do a bad job optimising

Describe, don’t implement

Solving PDEs

$$F(u) = 0 \text{ in } \Omega$$
$$u = g \text{ on } \Gamma_1$$
$$\frac{\partial u}{\partial n} = h \text{ on } \Gamma_2$$

Seek *weak* solution in some space of functions $V(\Omega)$.

Find $u \in V$ s.t.

$$\int_\Omega F(u) v \, \mathrm{d}x = 0 \quad \forall \, v \in V$$

Choose discrete $V_h \subset V$, and seek $u_h \in V_h$.

Pick *basis* for V_h with finite support.

Divide domain Ω into triangulation $\mathcal{T}$.

Integrals become sum over element integrals

$$\int_\Omega F(u_h)v_h \,\mathrm{d}x = \sum_{e\in\mathcal{T}} \int_e F(u_h)v_h \,\mathrm{d}x$$

Perform element integrals with numerical quadrature

$$\int_e F(u_h)v_h \,\mathrm{d}x = \sum_q w_q F(u_h(q))v_h(q) \,\mathrm{d}x$$

$F(u_h(q))$ is "user-specified". Variability in innermost loop.

Added complexity: more code

Compared to the simple finite difference code, I need much more code

- Numerical quadrature
- Orthogonal polynomials
- Indirections from mesh topology to data

I can’t fit the action of the Laplacian on a slide any more.

Added complexity: hardware is terrible

Too many cores

Chip	Cores	TF/s	GB/s	F/B
NVidia P100	56 (3584)	5.3	730	7.2
Xeon Phi 7290F	72	3.5	450 + 100	5.4
Broadwell	22	0.78	150	5.2

Fine grained concurrency

Chip	Cores	Clock	Vector width
P100	56	1.5	32
Broadwell	22	2.2	4
Phi	72	1.5	8
Skylake	32	2.2	8
ARMv8 (Cavium)	54?	2?	4-32?

What about solving PDEs

Simple finite difference models can be programmed from scratch.

No-one should attempt to write a finite element model from scratch. Pick a library:

deal.II	`www.dealii.org`
DUNE	`www.dune-project.org`
Feel++	`www.feelpp.org`
FEniCS	`www.fenicsproject.org`
Firedrake	`www.firedrakeproject.org`
FreeFEM++	`www.freefem.org`
MFEM	`mfem.org`
NGSolve	`ngsolve.org`
oomph-lib	`oomph-lib.maths.man.ac.uk`

…

Computational model

Abstract model for computing a finite element integral:

1. *gather* from global to local
2. *compute* on local data
3. *scatter* from local to global

Software libraries provide APIs for each of these steps.

You might write code like this.

```
template<typename EG, typename LFSU, typename X, typename LFSV, typename M>
void jacobian_volume(const EG& eg, const LFSU& lfsu, const X& x,
                     const LFSV& lfsv, M& mat) const {
  const auto geo = eg.geometry();
  const auto S = geo.jacobianInverseTransposed(qp);
  RF factor = weight*geo.integrationElement(qp);
  double grad[dim][n] = {{0.0}};
  for (int i=0; i<dim; i++)
    for (int k=0; k<dim; k++)
      for (int j=0; j<n; j++)
        grad[i][j] += S[i][k] * gradhat[k][j];
  double A[n][n] = {{0.0}};
  for (int i=0; i<n; i++)
    for (int k=0; k<dim; k++)
      for (int j=0; j<n; j++)
        A[i][j] += grad[k][i]*grad[k][j];
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      mat.accumulate(lfsu,i,lfsu,j,A[i][j]*factor);
}
```

Pros

- Not tied to particular discretisation
- Relatively easy to understand
- Extensible once code is understood
- Free to “do what you want”, e.g. manually vectorise

Cons

- Lost full control over data layout/execution (e.g. `geo.integrationElement(qp)`)
- Compiler may do a bad job optimising
- Need to learn more nouns and verbs
- May do a bad job of producing high performance code

Assertion

Once we pick the discretisation, writing the element integral is mechanical.

Corollary

Computers are good at mechanical things, why don't we get the computer to write the element integral?

A descriptive approach

Experimentation should be easy

Once I've set up my model, how much code do I have to change to:

- Change preconditioner?
- Drop terms in preconditioning operator?
- Use a different operator as preconditioner?
- Run quasi-Newton?
- Use matrix-free actions?
- Precondition matrix-free problems?

There is no prototype code

- Research groups don’t typically have the resource to rewrite models
- Need to have efficient code first time round
- Do you have the expertise to do this?

Firedrake

An automated finite element system.

`www.firedrakeproject.org`

Rathgeber et al. 2016 `arXiv:1501.01809[cs.MS]`

Development team

IC Thomas Gibson, David A. Ham, Miklós Homolya, Lawrence Mitchell, Fabio Luporini, Tianjiao Sun, Paul H. J. Kelly

Baylor Robert C. Kirby

Bath Andrew T. T. McRae

ECMWF Florian Rathgeber

IBM Gheorghe-Teodor Bercea

Firedrake

An automated finite element system.

`www.firedrakeproject.org`

Rathgeber et al. 2016 `arXiv:1501.01809[cs.MS]`

Users at

Imperial, Bath, Leeds, Kiel, Rice, Houston, Oregon Health & Science, Exeter, ETH, Waterloo, Minnesota, Baylor ...

Exploiting abstractions

- Firedrake builds on, and extends, embedded DSLs developed in the FEniCS project
  `www.fenicsproject.org`
- The *Unified Form Language* (Alnæs et al. 2014) to specify variational forms
- A symbolic problem description (generic) is woven together with problem-specific data, and executed by a runtime Python library that does JIT code compilation.

A descriptive approach

Firedrake, and FEniCS, mirror the mathematical abstraction of the finite element method in a *domain specific language*.

```
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(grad(u), grad(v))*dx
A = assemble(a)
```

- Symbolic description, implementation of “maths”, not “numerics”.
- Can concentrate on numerical aspects, not low-level code.
- Don’t control everything, but did I need to?

```
void cell_integral(double A[4][4], const double *const restrict *restrict coords) {
  static const double  t23[4]  = {-1.0, 0.0, 0.0, 1.0};
  static const double  t25[4]  = {-1.0, 0.0, 1.0, 0.0};
  static const double  t27[4]  = {-1.0, 1.0, 0.0, 0.0};
  ... /* Part of unrolled Jacobian computation */
  double t0  = (-1 * coords[0][0]);
  double t1  = (t0 + coords[1][0]);
  double t2  = (-1 * coords[0][1]);
  double t3  = (t2 + coords[2][1]);
  double t4  = (-1 * coords[0][2]);
  double t5  = (t4 + coords[3][2]);
  double t6  = (t2 + coords[3][1]);
  double t7  = (t4 + coords[2][2]);
  ...
  double t31  = (0.166666666666667 * fabs(t15));
  for (int k0 = 0; k0 < 4; k0++) {
    t28[k0] = ((t26 * t27[k0]) + (t24 * t25[k0])) + (t22 * t23[k0]);
    t29[k0] = ((t21 * t27[k0]) + (t20 * t25[k0])) + (t19 * t23[k0]);
    t30[k0] = ((t18 * t27[k0]) + (t17 * t25[k0])) + (t16 * t23[k0]);
  }
  for (int j0 = 0; j0 < 4; j0++) {
    double t32  = (((t26 * t27[j0]) + (t24 * t25[j0])) + (t22 * t23[j0]));
    double t33  = (((t21 * t27[j0]) + (t20 * t25[j0])) + (t19 * t23[j0]));
    double t34  = (((t18 * t27[j0]) + (t17 * t25[j0])) + (t16 * t23[j0]));
    for (int  k0  = 0; k0 < 4; k0 += 1) {
      A[j0][k0] += t31 * (((t34 * t30[k0]) + (t33 * t29[k0])) + (t32 * t28[k0]));
    }
  }
}
```

A job for an expert

Hardware-aware optimsation of finite element kernels is a job for:

- A numerical analyst?
- A geodynamicist?
- A computational chemist?
- A computational scientist?
- A computer scientist?

Form compilers

Translation of mathematical description of finite element problems carried out by a *form compiler*.

There are a few of these:

- DUNE (PDELAB): form compiler in development
- FFC: the FEniCS form compiler
- TSFC: the two stage form compiler (Firedrake)

Optimising compilers

- Both code snippets are independent of polynomial degree. *But*, the naive implementation I showed is algorithmically suboptimal.
- *Sum factorisation* (e.g. Karniadakis and Sherwin (2005)) reduces operation count for residual evaluation from $\mathcal{O}(p^{2d})$ to $\mathcal{O}(dp^{d+1})$.

Optimising compilers

- Both code snippets are independent of polynomial degree. *But*, the naive implementation I showed is algorithmically suboptimal.
- *Sum factorisation* (e.g. Karniadakis and Sherwin (2005)) reduces operation count for residual evaluation from $\mathcal{O}(p^{2d})$ to $\mathcal{O}(dp^{d+1})$.
- Computer science translation: “sum factorisation” is “loop invariant code motion”.

Optimising compilers

- Both code snippets are independent of polynomial degree. *But*, the naive implementation I showed is algorithmically suboptimal.
- *Sum factorisation* (e.g. Karniadakis and Sherwin (2005)) reduces operation count for residual evaluation from $\mathcal{O}(p^{2d})$ to $\mathcal{O}(dp^{d+1})$.
- Computer science translation: “sum factorisation” is “loop invariant code motion”.

```
V = FiniteElement("Lagrange", hexahedron, 7)
u = Coefficient(V)
v = TestFunction(V)
F = inner(grad(u), grad(v))*dx
count_flops(F, mode="vanilla") => 6668201
count_flops(F, mode="spectral") => 185257
```

Automating expertise

You can perform these optimisations by hand, but

- "In-person" case-by-case optimisation *does not scale*
- Code generation allows us to package expertise and provide it to everyone
- Done by a domain compiler
- Anecdotally, research groups that have implemented "high performance" finite element code by hand, are now turning to domain compilers.

Complex models

Still fit on a slide!

Rayleigh-Bénard convection

$$-\Delta u + u \cdot \nabla u + \nabla p + \frac{\mathrm{Ra}}{\mathrm{Pr}} \hat{g} T = 0$$

$$\nabla \cdot u = 0$$

$$-\frac{1}{\mathrm{Pr}} \Delta T + u \cdot \nabla T = 0$$

```
from firedrake import *
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
W = FunctionSpace(mesh, "CG", 1)
Q = FunctionSpace(mesh, "CG", 1)
Z = V * W * Q
upT = Function(Z)
u, p, T = split(upT)
v, q, S = TestFunctions(Z)
bcs = [...] # no-flow + temp gradient
nullspace = MixedVectorSpaceBasis(
   Z, [Z.sub(0), VectorSpaceBasis(constant=True),
       Z.sub(2)])
F = (inner(grad(u), grad(v))
     + inner(dot(grad(u), u), v)
     - inner(p, div(v))
     + (Ra/Pr)*inner(T*g, v)
     + inner(div(u), q)
     + inner(dot(grad(T), u), S)
     + (1/Pr) * inner(grad(T), grad(S)))*dx

solve(F == 0, upT, bcs=bcs, nullspace=nullspace)
```

More than a pretty face

Library usability

- High-level language enables rapid model development
- Ease of experimentation
- Small model code base

Library development

- Automation of complex optimisations
- Exploit expertise across disciplines
- Small library code base

Simple model development and experimentation

Fast prototyping is good, “but I have to rewrite for performance”.

Simple model development and experimentation

Fast prototyping is good, “but I have to rewrite for performance”.

- Firedrake provides computational performance often >50% achievable peak (Luporini, Varbanescu, et al. 2015; Luporini, Ham, and Kelly 2016; Bercea et al. 2016).
- Hero-coding necessary if you want the last 10-20%
- …but at what (person) cost?

Maintainability

Core Firedrake

Component	LOC
Firedrake	11500
PyOP2	6000
TSFC	3700
finat	600
Total	21800

Shared with FEniCS

Component	LOC
FIAT	4000
UFL	13000
Total	17000

Maintainable models

Thetis

`github.com/thetisproject/thetis`

- 3D unstructured coastal ocean model written with Firedrake
- 6400 LOC
- 4-8x faster than previous code in group (same numerics)

Maintainable models

Gusto

`www.firedrakeproject.org/gusto/`

- 3D atmospheric dynamical core using compatible FE
- Implements Met Office “Gung Ho” numerics
- 1600 LOC

Solving implicit systems

So I can write down a residual

- It is easy to write down complex implicit systems.
- But the challenge is usually in solving them
- Firedrake provides access to PETSc for solving
- “Any” algebraic solver you like, plus composition of block eliminations “fieldsplit” solvers.

Krylov methods are not solvers

Coupled problems make this hard

- Coupled problems are (typically) not amenable to black box solution methods.
- For small problems, can just use LU factorisation.
- For large problems, often use approximate block factorisations.
- Many configuration options, may require problem-specific auxiliary operators.
- Important to capture the abstraction so that automated model manipulation is still possible.

Block preconditioning

State of the art preconditioning for multi-variable problems is typically based on block LU factorisations.

Block preconditioning

State of the art preconditioning for multi-variable problems is typically based on block LU factorisations.

Rationale is that

$$T = \begin{bmatrix} A & 0 \\ 0 & D - CA^{-1}B^T \end{bmatrix}^{-1} \begin{bmatrix} A & B^T \\ C & D = 0 \end{bmatrix}$$

has minimal polynomial $T(T-I)(T^2-T-I) = 0$ (Murphy, Golub, and A. J. Wathen 2000). And thus a Krylov method converges in at most 4 iterations.

Ipsen (2001) treats case of $D \neq 0$.

Block preconditioning

Or “function space” preconditioning (Kirby 2010; Mardal and Winther 2011; Málek and Strakoš 2014).

Only uses block-diagonal operators.

e.g. Implicit timestepping for time-dependent Stokes

$$\begin{bmatrix} \mathbb{I} - \Delta t \nabla^2 & -\nabla \\ \nabla \cdot & 0 \end{bmatrix}$$

Can be preconditioned by

$$\begin{bmatrix} (\mathbb{I} - \Delta t \nabla^2)^{-1} & 0 \\ 0 & (-\nabla^2)^{-1} + \Delta t \mathbb{I}^{-1} \end{bmatrix}$$

Rayleigh-Bénard convection

$$-\Delta u + u \cdot \nabla u + \nabla p + \frac{\mathrm{Ra}}{\mathrm{Pr}} \hat{g} T = 0$$
$$\nabla \cdot u = 0$$
$$-\frac{1}{\mathrm{Pr}} \Delta T + u \cdot \nabla T = 0$$

Newton updates need inverse of Jacobian:

$$J = \begin{bmatrix} F & B^T & M_1 \\ C & 0 & 0 \\ M_2 & 0 & K \end{bmatrix}.$$

- Navier-Stokes (top left)
- Convection-diffusion for temperature (bottom right)
- Coupling in M_1 and M_2 (non-symmetric).

Preconditioning

We will invert J with a Krylov method, so we need a preconditioner. Let

$$N = \begin{bmatrix} F & B^T \\ C & 0 \end{bmatrix} \quad \tilde{M}_1 = \begin{bmatrix} M_1 \\ 0 \end{bmatrix} \quad \tilde{M}_2 = \begin{bmatrix} M_2 & 0 \end{bmatrix}$$

and block eliminate N in J, giving system for temperature:

$$S_T = K - \tilde{M}_2 N^{-1} \tilde{M}_1.$$

Howle and Kirby (2012) show that K^{-1} is a good preconditioner for S_T.

Newton update

Solve for the update

$$\delta x = J^{-1} F(x).$$

$$x \leftarrow x + \delta x$$

Write $\mathcal{K}(J, \mathbb{J})$ to denote approximating J^{-1} using an iteration $\mathcal{K}$ on J using $\mathbb{J}$ as a preconditioner. Then the iteration

$$\mathcal{K}\left(J, \begin{bmatrix} \mathcal{K}(N, \mathbb{N}) & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & -\tilde{M}_1 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & \mathcal{K}(K, \mathbb{K}) \end{bmatrix}\right)$$

empirically converges swiftly, and requires only $\mathbb{N}$ and $\mathbb{K}$.

Navier-Stokes block (Elman, Silvester, and A. Wathen 2014)

A lower Schur complement factorisation of N is a good option. Requires $\mathcal{K}(F, \mathbb{F})$ and $\mathcal{K}(S_p, \mathbb{S})$ where $S_p = -CF^{-1}B^T$.

One option is the *pressure convection-diffusion* approximation:

$$\mathbb{S} = \mathcal{K}(L_p, \mathbb{L})\, F_p \mathcal{K}(M_p, \mathbb{M}),$$

so our recipe for $\mathcal{K}(N, \mathbb{N})$ is:

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\ F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right).$$

Things to note

- We only ever need inverses of diagonal blocks.
- Can save memory by applying operators matrix-free.
- The inverses are nested, we need ways of controlling the inner iterations.

PCD

Needs the auxiliary discretised operator F_p and approximate inverses of the auxiliary operators L_p and M_p.

Communication between PDE and solver libraries can no longer be *unidirectional*.

Firedrake & PETSc to the rescue

- PETSc already provides a highly runtime-configurable library for *algebraically* composing solvers (Brown et al. 2012).
- Firedrake makes it straightforward to build auxiliary operators.
- We combine these to allow simple development of complex preconditioners.

Two new pieces

A new matrix type

A PETSc shell matrix that implements matrix-free actions using Firedrake, and contains the UFL of the bilinear form.

Custom preconditioners

These matrices do not have entries, so we create custom preconditioners that can inspect the UFL and do the appropriate thing.

PCD

```python
class PCDPC(PCBase):
    def initialize(self, pc):
        _, P = pc.getOperators()
        ctx = P.getContext()
        appctx = ctx.appctx
        p, q = ctx.arguments()
        [...] # Handling of boundary conditions elided
        M_p = assemble(p*q*dx)
        L_p = assemble(inner(grad(p), grad(q))*dx
        M_ksp = KSP().create()
        M_ksp.setOperators(M_p)
        L_ksp = KSP().create()
        L_ksp.setOperators(L_p)
        [...] # Some boilerplate elided
        u0 = split(appctx["state"])[appctx["velocity_space"]]
        F_p = assemble(inner(grad(p), grad(q))*dx + inner(u0, grad(p))*q*dx)

    def apply(self, pc, x, y):
        # y ← 𝒦(L_p, 𝕃) F_p 𝒦(M_p, 𝕄) x
        a, b = self.workspace
        self.M_ksp.solve(x, a)
        self.F_p.mult(a, b)
        self.L_ksp.solve(b, y)
```

How to configure things

PETSc provides a “programming language” for configuring objects at runtime. It has two operations

1. Value assignment
2. String concatenation

Every object has an *options prefix* which controls where in the options database it looks for configuration values.

This is verbose, but a very powerful idea. We can control the types of individual solves by ensuring that they have different prefixes.

An overview of the full solver

We are solving

$$\mathcal{K}\left(\begin{bmatrix} F & B^T & M_1 \\ C & 0 & 0 \\ M_2 & 0 & K \end{bmatrix}, \mathbb{J}\right)$$

using

$$\mathbb{J} = \begin{bmatrix} \mathcal{K}\left(\begin{bmatrix} F & B^T \\ C & 0 \end{bmatrix}, \mathbb{N}\right) & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & 0 & -M_1 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & \mathcal{K}(K, \mathbb{K}) \end{bmatrix}$$

with

$$\mathbb{N} = \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\ F_p\ \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}$$

and

$$S_p = -C\mathcal{K}(F, \mathbb{F})\, B^T.$$

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(K, \mathbb{K}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\ F_p\ \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type aij
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\ F_p\ \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\ F_p\ \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Let's configure the Navier-Stokes solve

$$\mathcal{K}\left(N, \begin{bmatrix} F & 0 \\ 0 & \mathcal{K}(S_p, \mathcal{K}(L_p, \mathbb{L})\, F_p\, \mathcal{K}(M_p, \mathbb{M})) \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} \mathcal{K}(F, \mathbb{F}) & 0 \\ 0 & I \end{bmatrix}\right)$$

```
-fieldsplit_0_ksp_type gmres
-fieldsplit_0_pc_type fieldsplit
-fieldsplit_0_pc_fieldsplit_type schur
-fieldsplit_0_pc_fieldsplit_schur_fact_type lower
-fieldsplit_0_fieldsplit_0_ksp_type preonly
-fieldsplit_0_fieldsplit_0_pc_type python
-fieldsplit_0_fieldsplit_0_pc_python_type firedrake.AssembledPC
-fieldsplit_0_fieldsplit_0_assembled_pc_type hypre
-fieldsplit_0_fieldsplit_1_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pc_type python
-fieldsplit_0_fieldsplit_1_pc_python_type firedrake.PCDPC
-fieldsplit_0_fieldsplit_1_pcd_Fp_mat_type matfree
-fieldsplit_0_fieldsplit_1_pcd_Mp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Mp_pc_type ilu
-fieldsplit_0_fieldsplit_1_pcd_Kp_ksp_type preonly
-fieldsplit_0_fieldsplit_1_pcd_Kp_pc_type hypre
```

Reasonable performance

DoFs ($\times 10^6$)	Cores	Newton its	Krylov its	Time (s)
0.741	24	3	16	32
1.488	48	3	16	36
2.973	96	3	17	44
5.769	192	3	17	47
11.66	384	3	17	56
23.39	768	3	17	65
45.54	1536	3	18	85
92.28	3072	3	18	120
185.6	6144	3	19	167

Runtime composability

- Can tune implicit solve for Navier-Stokes on its own, then drop in where-ever such a block wants inverted.
- Model formulation doesn't care about variable splittings.
- Composes with nonlinear solvers that need linearisations.
- Automatically take advantage of any improvements in Firedrake (fast matrix actions, etc...)
- No need to worry about parallel!

Easy extensibility I

A preconditioner for the Ohta–Kawasaki equation (P. E. Farrell and Pearson 2016)

$$u_t - \Delta w + \sigma(u - m) = 0$$
$$w + \epsilon^2 \Delta u - u(u^2 - 1) = 0$$

Newton iteration at each timestep solves

$$\begin{bmatrix} (1 + \Delta t \theta \sigma) M & \Delta t \theta K \\ -\epsilon^2 K - M_E & M \end{bmatrix} \begin{bmatrix} \delta u \\ \delta w \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

Easy extensibility II

Preconditioning

$$P^{-1} = \begin{bmatrix} (1+\Delta t\theta\sigma)M & 0 \\ -\epsilon^2 K - M_E & S \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix}.$$

Use

$$S^{-1} \approx \hat{S}^{-1} M \hat{S}^{-1}$$

where

$$\hat{S} = M + \epsilon\sqrt{(\Delta t\theta)/(1+\Delta t\theta\sigma)}K.$$

Implementation

```python
class OKPC(PCBase):

    def initialize(self, pc):
        # Approximate S^{-1} ~ Ŝ^{-1} M Ŝ^{-1} where Ŝ = ⟨q, w⟩ + ε√c ⟨∇q, ∇w⟩
        _, P = pc.getOperators()
        ctx = P.getPythonContext()
        # User information about Δt, θ, etc...
        dt, theta, eps, sigma = ctx.appctx["parameters"]
        V = ctx.a.arguments()[0].function_space()
        c = (dt * theta * eps**2)/(1 + dt * theta * sigma)
        w = TrialFunction(V)
        q = TestFunction(V)
        op = assemble(inner(w, q)*dx + sqrt(c) * inner(grad(w), grad(q))*dx)
        self.ksp = KSP().create(comm=pc.comm)
        self.ksp.setOptionsPrefix(pc.getOptionsPrefix + "shat_")
        self.ksp.setOperators(op.petscmat, op.petscmat)
        [...] # boilerplate elided
        mass = assemble(w*q*dx)
        self.mass = mass.petscmat
        work = self.mass.createVecLeft()
        self.work = (work, work.duplicate())

    def apply(self, pc, x, y):
        tmp1, tmp2 = self.work
        self.ksp.solve(x, tmp1)
        self.mass.mult(tmp1, tmp2)
        self.ksp.solve(tmp2, y)
```

Wrapping up

Answering some questions

Once I've set up my model, how much code do I have to change to:

- Change preconditioner? 1-10 lines
- Drop terms in preconditioning operator? 1-10 lines
- Use a different operator as preconditioner? 1-10 lines
- Run quasi-Newton? 1-10 lines
- Use matrix-free actions? 1-10 lines
- Precondition matrix-free problems? 10-100 lines

Summing up

- Firedrake is high performance for many problems in both the programmer and computer time metrics
- A good choice for experimenting with new algorithms: you can run on your laptop and scale up easily
- I can't solve your maths problems, I can give you tools to write down the solution more easily
- You can't do everything in the framework, but we do give you some mechanisms for escaping the walled garden

Open challenges

- How should we compare methods? What is fair? Work precision?
- I haven't really said anything about timestepping.
- Problem, I think, is that there is no clean mathematical abstraction that captures a large enough class of methods. But note the work in Maddison and P. Farrell (2014).
- Some formulations (e.g. space-time DG) are amenable, but no-one has done the work.
- What about complex (multilevel) timesteppers?

References I

Alnæs, M. S. et al. (2014). "Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations". *ACM Trans. Math. Softw.* **40**. doi:10.1145/2566630.

Bercea, G.-T. et al. (2016). "A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake". *Geoscientific Model Development* **9**. doi:10.5194/gmd-9-3803-2016. arXiv: 1604.05937 [cs.MS].

Brown, J. et al. (2012). "Composable Linear Solvers for Multiphysics". *Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*. ISPDC '12. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ISPDC.2012.16.

Elman, H., D. Silvester, and A. Wathen (2014). *Finite elements and fast iterative solvers*. Second edition. Oxford University Press.

References II

Farrell, P. E. and J. W. Pearson (2016). *A preconditioner for the Ohta–Kawasaki equation*. arXiv: 1603.04570 [ma.NA].

Howle, V. E. and R. C. Kirby (2012). "Block preconditioners for finite element discretization of incompressible flow with thermal convection". *Numerical Linear Algebra with Applications* **19**. doi:10.1002/nla.1814.

Ipsen, I. C. F. (2001). "A Note on Preconditioning Nonsymmetric Matrices". *SIAM Journal on Scientific Computing* **23**. doi:10.1137/S1064827500377435.

Karniadakis, G. and S. Sherwin (2005). *Spectral/hp element methods for computational fluid dynamics*. 2nd ed. Oxford University Press.

Kirby, R. C. (2010). "From Functional Analysis to Iterative Methods". *SIAM Review* **52**. doi:10.1137/070706914.

References III

Luporini, F., D. A. Ham, and P. H. J. Kelly (2016). *An algorithm for the optimization of finite element integration loops*. Submitted. arXiv: 1604.05872 [cs.MS].

Luporini, F., A. L. Varbanescu, et al. (2015). "Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly". *ACM Trans. Archit. Code Optim.* **11**. doi:10.1145/2687415.

Maddison, J. and P. Farrell (2014). "Rapid development and adjoining of transient finite element models". *Computer Methods in Applied Mechanics and Engineering* **276**. doi:10.1016/j.cma.2014.03.010.

Málek, J. and Z. Strakoš (2014). *Preconditioning and the Conjugate Gradient Method in the Context of Solving PDEs*. Philadelphia, PA: Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973846.

References IV

Mardal, K.-A. and R. Winther (2011). “Preconditioning discretizations of systems of partial differential equations”. *Numerical Linear Algebra with Applications* **18**. doi:10.1002/nla.716.

Murphy, M. F., G. H. Golub, and A. J. Wathen (2000). “A Note on Preconditioning for Indefinite Linear Systems”. *SIAM Journal on Scientific Computing* **21**. doi:10.1137/S1064827599355153.

Rathgeber, F. et al. (2016). “Firedrake: automating the finite element method by composing abstractions”. *ACM Transactions on Mathematical Software* **43**. doi:10.1145/2998441. arXiv: 1501.01809 [cs.MS].