Across the great divide

composable block preconditioning from UFL

Lawrence Mitchell[1,*] Rob Kirby[2]

12th June 2017

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

[2]Department of Mathematics, Baylor University

UFL makes it easy to write complex PDEs

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

$$\begin{bmatrix} F & B^T & M_1 \\ C & 0 & 0 \\ M_2 & 0 & K \end{bmatrix} \begin{bmatrix} \delta u \\ \delta p \\ \delta T \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

```
from firedrake import *
mesh = Mesh(...)
V = VectorFunctionSpace(mesh, "CG", 2)
W = FunctionSpace(mesh, "CG", 1)
Q = FunctionSpace(mesh, "CG", 1)
Z = V * W * Q
Ra = Constant(200)
Pr = Constant(6.18)
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)
```

Imperial College London

Cahn-Hilliard

$$\phi_t - \nabla \cdot \rho \nabla \mu = 0$$
$$\mu + \lambda \Delta \phi - \phi(\phi^2 - 1) = 0$$

Implicit timestepping + Newton

$$\begin{bmatrix} M & \Delta t \theta \rho K \\ -\lambda K - M_\phi & M \end{bmatrix} \begin{bmatrix} \delta \phi \\ \delta \mu \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

```
from firedrake import *
mesh = Mesh(...)
V = FunctionSpace(mesh, "CG", 1)
Z = V*V
θ = 0.5
λ = Constant(0.005)
M = Constant(10)
q, v = TestFunctions(Z)
z = Function(Z)
z0 = Function(Z)
φ_{n+1}, μ_{n+1} = split(z)
φ_n, μ_n = split(z0)
φ_{n+1} = variable(φ_{n+1})
f = (1.0/4)*(φ_{n+1}**2 - 1)**2
dfdphi = diff(f, φ_{n+1})
μ_{n+θ} = (1-θ)*μ_n + θ*μ_{n+1}
dt = 5e-6
F = ((φ_{n+1} - φ_n)*q
     + dt*M*dot(grad(μ_{n+θ}), grad(q))
     + (μ_{n+1} - dfdphi)*v
     -λ*dot(grad(φ_{n+1}), grad(v)))*dx
while t < ...:
    z0.assign(z)
    solve(F == 0, z)
```

Ohta-Kawasaki

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

Implicit timestepping + Newton

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

```
from firedrake import *
mesh = Mesh(...)
V = FunctionSpace(mesh, "CG", 1)
Z = V*V
ϵ = Constant(0.02)
σ = Constant(100)
dt = Constant(eps**2)
θ = Constant(0.5)
v, q = TestFunctions(Z)
z = Function(Z)
z0 = Function(Z)
u, w = split(z)
u0, w0 = split(z0)
u_θ = (1 - θ)*u0 + θ*u
w_θ = (1 - θ)*w0 + θ*w
dfdu = u**3 - u
F = ((u - u0)*v
     + dt*dot(grad(w_θ), grad(v))
     + dt*σ*(u_θ - m)*v
     + w*q - dfdu*q
     - ϵ**2*dot(grad(u), grad(q)))*dx
while t < ...:
    z0.assign(z)
    solve(F == 0, z)
```

What about the solvers?

- LU is alright for small problems
- …but quickly becomes untenable in 3D.
- Instead we use iterative methods (e.g. Krylov methods)

What about the solvers?

- LU is alright for small problems
- ...but quickly becomes untenable in 3D.
- Instead we use iterative methods (e.g. Krylov methods)
- ...but Krylov methods are *not* solvers

What about the solvers?

- LU is alright for small problems
- …but quickly becomes untenable in 3D.
- Instead we use iterative methods (e.g. Krylov methods)
- …but Krylov methods are *not* solvers
- so we need *preconditioners*.

Block preconditioning

- Coupled problems are (typically) not amenable to black box preconditioning.

Block preconditioning

- Coupled problems are (typically) not amenable to black box preconditioning.
- Solution: block preconditioning

Schur complement

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

has minimal polynomial $T(T-I)(T^2 - T - I) = 0$.

Function space

If $\mathcal{A} : W \to W^*$, then

$$\langle u, v \rangle_W^{-1} \mathcal{A}$$

has mesh independent condition number.

A problem

- Such preconditioners often need auxiliary matrices not appearing in the original operator.
- How do we provide these to the solver library in a composable manner?

A problem

- Such preconditioners often need auxiliary matrices not appearing in the original operator.
- How do we provide these to the solver library in a composable manner?

Firedrake & PETSc to the rescue

- PETSc already provides a highly runtime-configurable library for *algebraically* composing solvers.
- 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.

$$y \leftarrow Ax$$

```
A = assemble(a, mat_type="matfree")
```

Custom preconditioners

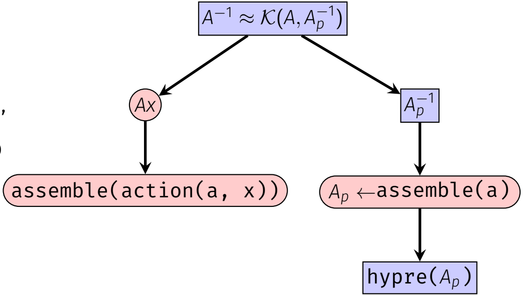
Such matrices do not have entries, we create preconditioners that inspect the UFL and do the appropriate thing.

$$y \leftarrow \tilde{A}^{-1}x$$

```
solve(a == L, x,
      {"pc_type": "python",
       "pc_python_type": "AssembledPC"})
```

Matrix-free actions with AMG on the assembled operator.

```
a = u*v*dx + dot(grad(u), grad(v)*dx)
opts = {"ksp_type": "cg",
        "mat_type": "matfree",
        "pc_type": "python",
        "pc_python_type": "AssembledPC",
        "assembled_pc_type": "hypre"}
solve(a == L, x, solver_parameters=opts)
```



Imperial College London

Something more complicated I

A preconditioner for the Ohta–Kawasaki equation (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

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

Something more complicated II

Preconditioning strategy:

$$\begin{bmatrix} [(1+\Delta t\theta\sigma)M]^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ (\epsilon^2 K + M_E)\,[(1+\Delta t\theta\sigma)M]^{-1} & I \end{bmatrix} \begin{bmatrix} (1+\Delta t\theta\sigma)M & \Delta t\theta K \\ -\epsilon^2 K - M_E & M \end{bmatrix}.$$

Where

$$S = M + (\epsilon^2 K + M_E)\,[(1+\Delta t\theta\sigma)M]^{-1}\,\Delta t\theta K$$

is inverted iteratively, preconditioned by

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

with

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

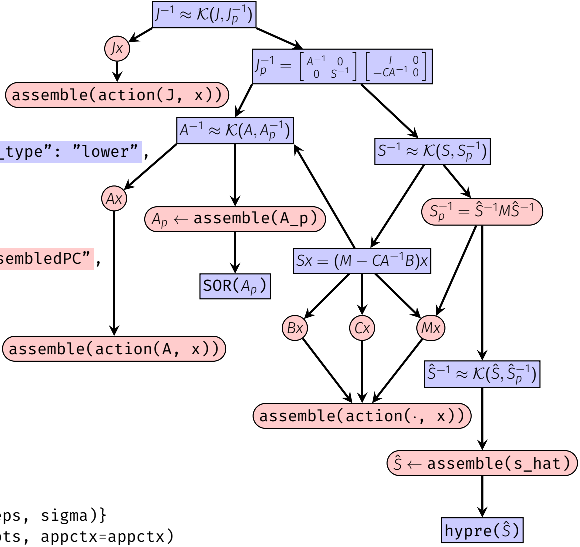
Implementation

```
class OKPC(PCBase):
    def initialize(self, pc):
        _, P = pc.getOperators()
        ctx = P.getPythonContext()
        # User information about Δt, θ, etc...
        dt, θ, ϵ, σ = ctx.appctx["parameters"]
        V = ctx.a.arguments()[0].function_space()
        c = (dt * θ)/(1 + dt * θ * σ)
        w = TrialFunction(V)
        q = TestFunction(V)
        # Ŝ = ⟨q, w⟩ + ϵ√c ⟨∇q, ∇w⟩, c = Δtθ/(1+Δtθσ)
        op = assemble(inner(w, q)*dx + ϵ*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)
        self.ksp.setFromOptions()
        mass = assemble(w*q*dx)
        self.mass = mass.petscmat
        work = self.mass.createVecLeft()
        self.work = (work, work.duplicate())
    def apply(self, pc, x, y):
        t1, t2 = self.work
        # t1 ← Ŝ^{-1}x
        self.ksp.solve(x, t1)
        # t2 ← Mt1
        self.mass.mult(t1, t2)
        # y ← Ŝ^{-1}t2 = Ŝ^{-1}MŜ^{-1}x
        self.ksp.solve(t2, y)
```

Usage

```
opts = {
  "snes_lag_preconditioner": -1,
  "mat_type": "matfree",
  "ksp_type": "gmres",
  "pc_type": "fieldsplit",
  "pc_fieldsplit_type": "schur",
  "pc_fieldsplit_schur_factorization_type": "lower",
  "fieldsplit_0": {
      "ksp_type": "chebyshev",
      "ksp_max_it": 10,
      "pc_type": "python",
      "pc_python_type": "firedrake.AssembledPC",
      "assembled_pc_type": "sor"},
  "fieldsplit_1": {
      "ksp_type": "richardson",
      "ksp_max_it": 2,
      "pc_type": "python",
      "pc_python_type": "OKPC",
      "shat_ksp_type": "richardson",
      "shat_ksp_max_it": 5,
      "shat_pc_type": "hypre"
 }

appctx = {"parameters": (dt, theta, eps, sigma)}
solve(L == 0, z, solver_parameters=opts, appctx=appctx)
```



Performance: Rayleigh-Bénard

Limited by performance of algebraic solvers on subblocks.

DoFs ($\times 10^6$)	MPI processes	Newton its	Krylov its	Time (s)
0.7405	24	3	16	31.7
2.973	96	3	17	43.9
11.66	384	3	17	56
45.54	1536	3	18	85.2
185.6	6144	3	19	167

Performance: Rayleigh-Bénard

Limited by performance of algebraic solvers on subblocks.

DoFs ($\times 10^6$)	MPI processes	Newton its	Krylov its	Time (s)
0.7405	24	3	16	31.7
2.973	96	3	17	43.9
11.66	384	3	17	56
45.54	1536	3	18	85.2
185.6	6144	3	19	167

DoFs ($\times 10^6$)	Navier-Stokes iterations		Temperature iterations	
	Total	per solve	Total	per solve
0.7405	329	20.6	107	6.7
2.973	365	21.5	132	7.8
11.66	373	21.9	137	8.1
45.54	403	22.4	151	8.4
185.6	463	24.4	174	9.2

Conclusions

- Composable, extensible method using UFL to build block preconditioners.
- Model formulation doesn't need to know about the solver configuration.
- Composes with nonlinear solvers that need linearisations: no need to write your own special Newton iteration to get data in.
- Automatically takes advantage of any improvements in Firedrake (fast matrix actions, etc...)

Kirby and Mitchell (2017) `arXiv: 1706.01346 [cs.MS]`

`www.firedrakeproject.org`

EPSRC
Engineering and Physical Sciences Research Council

References

Benzi, M., G. H. Golub, and J. Liesen (2005). “Numerical solution of saddle point problems”. *Acta Numerica* 14. doi:10.1017/S0962492904000212.

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.

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

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

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

Kirby, R. C. and L. Mitchell (2017). *Solver composition across the PDE/linear algebra barrier*. arXiv: 1706.01346 [cs.MS].

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.

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.