Solver composition across the PDE/linear algebra divide

Lawrence Mitchell[1,*] Rob Kirby[2,†]

28th May 2018

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

[2]Department of Mathematics, Baylor University
[†]robert_kirby@baylor.edu

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

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

```python
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)
```

UFL makes it easy to write complex PDEs

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)
uθ = (1 - θ)*u0 + θ*u
wθ = (1 - θ)*w0 + θ*w
dfdu = u**3 - u
F = ((u - u0)*v
     + dt*dot(grad(wθ), grad(v))
     + dt*σ*(uθ - 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?

Direct

- Great for small problems;
- wait forever for large problems.

Iterative

- Need good preconditioners;
- For many problems, algebraic manipulation of the operator is insufficient.

A wishlist

- Access to standard block preconditioners
- Easy specification of auxiliary operators
- “Simple” configuration
- Arbitrary nesting: smoothers inside splits, multigrid, etc…
- *Easy* to extend

Claim

All of these things are possible (straightforward?) if the solver library can call back to the PDE library to create operators.

Idea

- Endow discretised operators with PDE-level information:
    - what bilinear form
    - which function spaces
    - boundary conditions
- Enable standard fieldsplitting on these operators.
- Write custom preconditioners that can utilise the information in appropriately.

Idea

- Endow discretised operators with PDE-level information:
  - what bilinear form
  - which function spaces
  - boundary conditions
- Enable standard fieldsplitting on these operators.
- Write custom preconditioners that can utilise the information in appropriately.

Extend PETSc with Firedrake-level PCs

- PETSc already provides *algebraic* composition of solvers.
- Firedrake can provide auxiliary operators
- We just need to combine these appropriately.

This sounds like hard work

Fortunately, `petsc4py` makes it easy to write these PCs.

```
class MyPC(object):
    def setUp(self, pc):
        A, P = pc.getOperators()
        # A and P are shell matrices, carrying the symbolic
        # discretisation information.
        # So I have access to the mesh, function spaces, etc...
        # Can inspect options dictionary here
        # do whatever
    def apply(self, pc, r, e):
        # Compute approximation to error given current residual
        # e ← A^{-1} r

solve(..., solver_parameters={"pc_type": "python",
                              "pc_python_type": "MyPC"})
```

PETSc manages all the splitting and nesting already. So this does the right thing *inside* multigrid, etc...

Implementation: two parts

A new matrix type

A shell matrix that implements matrix-free actions, and contains the symbolic information about the bilinear form.

$$y \leftarrow Ax$$

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

Could do this all with assembled matrices if desired.

Custom preconditioners

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

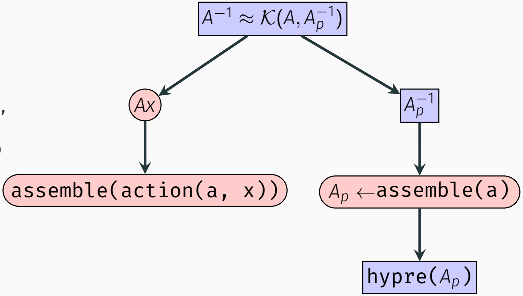
$$y \leftarrow \tilde{A}^{-1}x$$

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

A simple example

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)
```



A more complicated example

A preconditioner for the Ohta–Kawasaki equation (Farrell and Pearson 2017)

$$
\begin{aligned}
u_t - \Delta w + \sigma(u - m) &= 0 \\
w + \epsilon^2 \Delta u - u(u^2 - 1) &= 0
\end{aligned}
$$

Newton iteration at each timestep solves

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

A more complicated example

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 + "hats_")
        self.ksp.setOperators(op.petscmat, op.petscmat)
        self.ksp.setFromOptions()
        mass = assemble(w*q*dx)
        self.mass = mass.petscmat
        ...
    def apply(self, pc, x, y):
        t1, t2 = self.work
        # t₁ ← Ŝ⁻¹x
        self.ksp.solve(x, t1)
        # t₂ ← Mt₁
        self.mass.mult(t1, t2)
        # y ← Ŝ⁻¹t₂ = Ŝ⁻¹MŜ⁻¹x
        self.ksp.solve(t2, y)
```

Rayleigh-Bénard solver

For each Newton step, solve

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

using a preconditioner from Howle and Kirby (2012):

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

with

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

and

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

Kirby and Mitchell (2018, §B.4) shows full solver configuration.

Weak scaling

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

Weak scaling

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

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches
- Operators on each patch
- Solvers for each patch
- Boundary conditions

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches: PETSc
- Operators on each patch
- Solvers for each patch
- Boundary conditions

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches: PETSc
- Operators on each patch: Firedrake
- Solvers for each patch
- Boundary conditions

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches: PETSc
- Operators on each patch: Firedrake
- Solvers for each patch: PETSc
- Boundary conditions

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches: PETSc
- Operators on each patch: Firedrake
- Solvers for each patch: PETSc
- Boundary conditions: Homogeneous Dirichlet only for now

Schwarz smoothers

This approach works well for block solvers, what else can I do?

Building blocks

- Decomposition of mesh into patches: PETSc
- Operators on each patch: Firedrake
- Solvers for each patch: PETSc
- Boundary conditions: Homogeneous Dirichlet only for now

Idea

- PETSc PC using `DMPlex` to provide patches
- Callback interface provides operator on each patch to `PCApply`
- Normal `KSP` on each patch to do the solve

Patch definition

Patch described by set of entities on which dofs are free.

Builtin

Specify patches by selecting:

1. Entities to iterate over (vertices, cells, …);
2. Adjacency relation that gathers "free" dofs. Some builtin:

`star` all dofs in star of entity

`vanka` all dofs in closure of star of entity

User-defined

Write short function to define patches "by hand".

Implementation

`github.com/wence-/ssc`, available in PETSc RSN.

Example: P2-P1 Stokes

Monolithic multigrid with Vanka smoother on each level.

```
solver_parameters = {
    "mat_type": "matfree",
    # Flex-gmres due to nonlinear PC (gmres as smoother)
    "ksp_type": "fgmres",
    "pc_type": "mg",
    "mg_levels": {
        "ksp_type": "gmres",
        "ksp_max_it": 2,
        "pc_type": "python",
        "pc_python_type": "ssc.PatchPC",
        "patch_pc_patch_construction_type": "vanka",
        "patch_pc_patch_construction_dim": 0,  # patches over vertices
        "patch_pc_patch_vanka_dim": 0,         # what entities are in the constraint space?
        "patch_pc_patch_exclude_subspace": 1,  # which subspace to exclude?
        "patch_pc_patch_sub_mat_type": "seqaij",
        "patch_sub_ksp_type": "preonly",
        "patch_sub_pc_type": "lu",
        "patch_sub_pc_factor_shift_type": "nonzero"
    },
    "mg_coarse_pc_type": "lu",
}
solve(F == 0, u, solver_parameters=solver_parameters)
```

Example: P2-P1 Stokes

Schur complement only requires change of options.

```
solver_parameters = {
    "mat_type": "matfree",
    "ksp_type": "gmres",
    "ksp_monitor": None,
    "pc_type": "fieldsplit",
    # Use diag(A^-1, S^-1) as PC
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_factorization_type": "diag",
    "fieldsplit_0": {
        # AMG on velocity block
        "ksp_type": "preonly",
        "pc_type": "python",
        "pc_python_type": "firedrake.AssembledPC",
        "assembled_pc_type": "hypre",
    },
    "fieldsplit_1": {
        # Inverse mass matrix to precondition S
        "ksp_type": "richardson",
        "pc_type": "firedrake.MassInvPC",
    }
}
solve(F == 0, u, solver_parameters=solver_parameters)
```

Conclusions

- Composable solvers, using PDE library to easily develop complex block preconditioners.
- Model formulation decoupled from solver configuration.
- Automatically takes advantage of any improvements in both PETSc and Firedrake.
- Same approach works for Schwarz-like methods.

`www.firedrakeproject.org`

Kirby and Mitchell (2018) `arXiv:1706.01346 [cs.MS]`

EPSRC
Engineering and Physical Sciences
Research Council

References

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 (2017). "A preconditioner for the Ohta-Kawasaki equation". *SIAM Journal on Matrix Analysis and Applications* 38. 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.

Kirby, R. C. and L. Mitchell (2018). "Solver composition across the PDE/linear algebra barrier". *SIAM Journal on Scientific Computing* 40. doi:10.1137/17M1133208. arXiv: 1706.01346 [cs.MS].