Solver composition across the PDE/linear algebra divide

Lawrence Mitchell\textsuperscript{1,*} Rob Kirby\textsuperscript{2,†}

28th May 2018

\textsuperscript{1}Departments of Computing and Mathematics, Imperial College London
\textsuperscript{*}lawrence.mitchell@imperial.ac.uk

\textsuperscript{2}Department of Mathematics, Baylor University
\textsuperscript{†}robert_kirby@baylor.edu
Rayleigh-Bénard convection

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

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(...)  # 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

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

```python
from firedrake import *

mesh = Mesh(...)
V = FunctionSpace(mesh, "CG", 1)
Z = V*V
epsilon = Constant(0.02)
sigma = Constant(100)
dt = Constant(epsilon**2)
theta = Constant(0.5)
v, q = TestFunctions(Z)
z = Function(Z)
z0 = Function(Z)

u, w = split(z)
u0, w0 = split(z0)

u_\theta = (1 - \theta)u0 + \theta*u
w_\theta = (1 - \theta)w0 + \theta*w

dfdu = u**3 - u
F = ((u - u_\theta)*v +
    + dt*dot(grad(w_\theta), grad(v)) +
    + dt*sigma*(u_\theta - m)*v +
    + w*q - dfdu*q -
    - epsilon**2*dot(grad(u), grad(q))))*dx

while t < ...:
z0.assign(z)
solve(F == 0, z)
```

2
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.

```python
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
        # $e \leftarrow A^{-1}r$

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.

\[ y \leftarrow A x \quad \text{A} = \text{assemble}(a, \text{mat_type} = \text{"matfree"}) \]

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.

\[ y \leftarrow A^{-1} x \quad \text{solve}(a == L, x, \{ \text{"mat_type"}: \text{"matfree"}, \text{"pc_type"}: \text{"python"}, \text{"pc_python_type"}: \text{"AssembledPC"} \}) \]
A simple example

Matrix-free actions with AMG on the assembled operator.

\[ a = u^*v^*dx + \text{dot}(\text{grad}(u), \text{grad}(v))^*dx \]

\[
\text{opts} = \{
    \text{"ksp_type": "cg"},
    \text{"mat_type": "matfree"},
    \text{"pc_type": "python"},
    \text{"pc_python_type": "AssembledPC"},
    \text{"assembled_pc_type": "hypre"}
\} 
\]

\[
\text{solve}(a \equiv L, x, \text{solver_parameters=opts}) 
\]

\[ A^{-1} \approx \mathcal{K}(A, A_p^{-1}) \]

\[ A^{-1} p \approx A_p \]

\[
\text{assemble}(\text{action}(a, x)) 
\]

\[
\text{hypre}(A_p) 
\]
A more complicated example

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

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

Newton iteration at each timestep solves

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

Preconditioning strategy:

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

Where

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

is inverted iteratively, preconditioned by

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

with

\[
\hat{S} = M + \epsilon\sqrt{(\Delta t\theta)/(1 + \Delta t\theta\sigma)}K.
\]
class OKPC(PCBase):
    def initialize(self, pc):
        _, P = pc.getOperators()
        ctx = P.getPythonContext()
        # User information about ∆t, 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)
        # \hat{S} = ⟨q, w⟩ + ε\sqrt{c} ⟨\nabla q, \nabla w⟩, c = \frac{\Delta t \theta}{1 + \Delta t \theta \sigma}
        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 ← \hat{S}^{-1}x
        self.ksp.solve(x, t1)
        # t2 ← M_{t1}
        self.mass.mult(t1, t2)
        # y ← \hat{S}^{-1}t_2 = \hat{S}^{-1}M\hat{S}^{-1}x
        self.ksp.solve(t2, y)
Rayleigh-Bénard solver

For each Newton step, solve

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

using a preconditioner from Howle and Kirby (2012):

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

with

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

and

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

Kirby and Mitchell (2018, §B.4) shows full solver configuration.
Limited by performance of algebraic solvers on subblocks.

<table>
<thead>
<tr>
<th>DoFs ($\times 10^6$)</th>
<th>MPI processes</th>
<th>Newton its</th>
<th>Krylov its</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7405</td>
<td>24</td>
<td>3</td>
<td>16</td>
<td>31.7</td>
</tr>
<tr>
<td>2.973</td>
<td>96</td>
<td>3</td>
<td>17</td>
<td>43.9</td>
</tr>
<tr>
<td>11.66</td>
<td>384</td>
<td>3</td>
<td>17</td>
<td>56</td>
</tr>
<tr>
<td>45.54</td>
<td>1536</td>
<td>3</td>
<td>18</td>
<td>85.2</td>
</tr>
<tr>
<td>185.6</td>
<td>6144</td>
<td>3</td>
<td>19</td>
<td>167</td>
</tr>
</tbody>
</table>
Weak scaling

Limited by performance of algebraic solvers on subblocks.

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

<table>
<thead>
<tr>
<th>DoFs ($\times 10^6$)</th>
<th>Navier-Stokes iterations per solve</th>
<th>Temperature iterations per solve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Total</td>
</tr>
<tr>
<td>0.7405</td>
<td>329</td>
<td>107</td>
</tr>
<tr>
<td>2.973</td>
<td>365</td>
<td>132</td>
</tr>
<tr>
<td>11.66</td>
<td>373</td>
<td>137</td>
</tr>
<tr>
<td>45.54</td>
<td>403</td>
<td>151</td>
</tr>
<tr>
<td>185.6</td>
<td>463</td>
<td>174</td>
</tr>
</tbody>
</table>
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
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.

**Built-in**

Specify patches by selecting:

1. Entities to iterate over (vertices, cells, ...);
2. Adjacency relation that gathers “free” dofs. Some built-in:
   - **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](https://github.com/wence-/ssc), available in PETSc RSN.
Monolithic multigrid with Vanka smoother on each level.

```python
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)
```
Schur complement only requires change of options.

```python
solver_parameters = {
    "mat_type": "matfree",
    "ksp_type": "gmres",
    "ksp_monitor": None,
    "pc_type": "fieldsplit",
    # Use $\text{diag}(A^{-1}, S^{-1})$ 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


