Block preconditioners in DOLFIN [inf9681]

Joachim Berdal Haga

March 17, 2011

This report describes block preconditioners based on the Python layer of DOLFIN. The block preconditioners are shown applied to the mixed Poisson problem as well as Biot's problem.

A number of patches has been submitted to DOLFIN, Viper and Dorsal to lay the foundation for the work described here. These patches can be reviewed on the appropriate mailing lists or Mercurial repositories (for January 2011).

1 Rationale

Block-structured systems of equations are a common occurence in the discrete solution of coupled partial differential equations (PDEs). Such block-structured systems may pose difficulties for iterative solvers, because they are often indefinite or saddle-point problems, and because the different blocks prefer different preconditioning.

For large-scale problems, in particular those that require the use of massively parallel computers with distributed memory, direct solvers are not a feasible option, and iterative methods must be applied. To achieve satisfactory convergence, these iterative methods require effective and robust preconditioning.

Block preconditioners are a practical solution to the problem of preconditioning the discretisation of coupled PDEs, but are often difficult to specify and implement. Implementing the block structure in Python promises a general and easy-to-use solution.

Rather than showing the implementation details (which — apart from the patches mentioned above — are about 450 lines of Python code for the framework, and another few hundred for the demos), I will just show examples of

how it is used. Since the usage is very close to the mathematical notation, it should be easy to follow.

2 Example I: Mixed Poisson

The reader will be familiar with the Mixed Poisson demo in DOLFIN. I will demonstrate how it is converted to a block matrix structure, and then construct a block preconditioner.

The major change is that MixedFunctionSpace is not used, just plain function spaces:

Mixed Poisson

```
from dolfin import *

mesh = UnitSquare(32, 32)

# Define function spaces and mixed (product) space
BDM = FunctionSpace(mesh, "BDM", 1)
DG = FunctionSpace(mesh, "DG", 0)
W = BDM * DG

# Define trial and test functions
(sigma, u) = TrialFunctions(W)
(tau, v) = TestFunctions(W)
```

Block Poisson

```
import PyTrilinos     # must be imported before dolfin
from dolfin import *
from block import *
from block.iterative import *
from block.algebraic.trilinos import ML

# Create mesh
mesh = UnitSquare(32,32)

# Define function spaces
BDM = FunctionSpace(mesh, "BDM", 1)
DG = FunctionSpace(mesh, "DG", 0)

sigma, tau = TrialFunction(BDM), TestFunction(BDM)
u, v       = TrialFunction(DG),  TestFunction(DG)
```

This is reflected in the definition of the form(s). In Mixed Poisson, a single form is used for each side of the equation, while Block Poisson creates one form per block. The same structure is found in the assembled matrices.

Mixed Poisson

```
# Define variational form
a = (dot(sigma, tau) + div(tau)*u + div(sigma)*v)*dx
L = - f*v*dx

# Assemble matrice and vector
A = assemble(a)
b = assemble(L)
```

Block Poisson

```
# Define variational forms separately, one per block
a00 = dot(sigma,tau)*dx
a01 = div(tau)*u*dx
a10 = div(sigma)*v*dx
L1  = -f*v*dx

# Assemble matrices and vector
A = assemble(a00)
B = assemble(a01)
C = assemble(a10)
b = assemble(L1)

# Create the block structured matrix and vector
from block import *
AA = block_mat([[A, B],
                [C, 0]])
bb = block_vec([0, b])
```

The zero block in the block coefficient structure matrix is not a problem; scalars are treated as either empty blocks or (scaled) identities, and uninitialised blocks (`None`) are treated as zero. In the RHS vector, uninitialised blocks are allowed, and treated as zero. Note that `block_mat` and `block_vec` are pure python objects.

We skip the definition of the source and boundary, which are the same in the two implementations. The definition and application of the Dirichlet boundary condition differ, however. It is defined on a different function space; in Mixed Poisson, the function space must be a subspace of the mixed space, but in Block Poisson the original space is used.

Mixed Poisson

```
# Create and apply Dirichlet BC for flux (BDM subspace)
bc = DirichletBC(W.sub(0), source, boundary)
bc.apply(A, b)
```

Block Poisson

```
# Create the Dirichlet BC on the first block
bc = BlockBC([DirichletBC(BDM, source, boundary), None])
bc.apply(AA, bb)
```

For iterative solvers the symmetry of the coefficient matrix is often important, and thus the matrix is modified in a symmetric fashion even though this is a quite expensive operation in the current implementation. In addition, the sign of the matrix is taken into account so that a negative definite matrix is modified with -1 on the diagonal instead of 1, so that it stays definite.

Now for the most interesting part: The algebraic solver. Mixed Poisson uses the default direct solver (boring but safe). For Block Poisson, however, I use preconditioned Conjucated Gradients with a block Jacobi preconditioner.

Mixed Poisson

```
x = Function(W)
solve(A, x.vector(), b)
```

Block Poisson

```
bJac = block_mat([[ML(A), 0],
                  [0,     1]])
AAinv = ConjGrad(AA, precond=bJac, maxiter=1000)
x = AAinv * bb
```

Since the mixed Poisson problem is a saddle-point problem, the (approximated) inverse of the $(2,2)$ block can of course not be used directly. Instead, the block Jacobi preconditioner uses an ML preconditioner for the first block, and the identity matrix for the second block. This is not a very strong preconditioner, but it converges to residual 10^{-5} in 867 iterations (4.6 seconds). For comparison, the LU solver for Mixed Poisson returns in about 1.1 second.

Now, the "proper" identity matrix is in this case the mass matrix, which we can create like so:

```
I = assemble(u*v*dx)
bJac = block_mat([[ML(A), 0],
                  [0,     I]])
```

With this change, convergence is in 503 iterations (2.8 seconds). But the real potential lies in that the operators can be combined almost arbitrarily: For example, `C*B` returns a composite object that, when left-multiplied with a vector, applies first B, then C.

Now, the inverse of the Laplacian is a good choice for the $(2,2)$ block, but since the DG(0) space has zero gradients it is not so easy to create directly. However, replacing the identity block in `bJac` with `ConjGrad(C*B)`,

```
bJac = BlockOperator([[ML(A),          0     ],
                      [0,     ConjGrad(C*B)]])
```

creates an (unpreconditioned) inner solver for the Laplacian, and with only this change the system converges in 44 outer iterations (1.5 seconds, with each outer iteration requiring up to 150 inner CG iterations). Further improvement can be had by using a Block Gauß-Seidel scheme instead of Block Jacobi (not shown): 31 iterations in 1.4 seconds. This can obviously be improved by using a cheaper approximation to the inverse Laplacian, either directly or as a preconditioner for the inner CG solve.

A word of warning: It is not generally recommended to use Conjugated Gradients as both inner and outer solver, because it's not linear, and because

they tend to reduce the same parts (and to miss the same parts) of the error. In this case, however, it seems to work fine. Caveat lector.

3 Performance

The bulk of the work is done in compiled code, and the python interface is just glue. Thus, for sufficiently large blocks, the impact is small. The main overhead compared to a C implementation is that a lot of temporary vectors are created. This may be fixable, at least partly, for example by using pools of scratch vectors.

Again, since all the real work is passed to parallel-aware linear algebra libraries, it should in principle work in parallel. It currently does not, because the application of Dirichlet boundary conditions is not parallelised. It only works in parallel in the rare case where no Dirichlet boundaries are used.