PETSc in CitcomS

Rajesh Kommu

September 29, 2014

PETSc is used for solving the Stokes equations in CitcomS. Currently, this is handled in the **`solve_Ahat_p_fhat`** function. This function lets the user choose between either the Conjugate Gradient (CG) or the BiConjugate Gradient Stabilized (BiCGStab) Krylov methods to solve the Stokes equations, or the Schur complement reduction method.

1 Stokes Equation

Our primary goal is to solve the following set of equations:

$$\begin{pmatrix} K & G \\ G^T & 0 \end{pmatrix} \begin{pmatrix} V \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix} \tag{1}$$

Here u is the velocity field and p is the pressure field.

K corresponds to the discrete Laplacian of the velocity field, and is currently implemented by the function **`assemble_del2_u`** in CitcomS. **`assemble_del2_u`** takes a $(neq \times 1)$ velocity vector u and returns a $(neq \times 1)$ vector $\nabla^2 u$. Thus K is a $(neq \times neq)$ matrix operator. neq is the number of equations, which is equal to the number of nodes, nno times the number of spatial dimensions, nsd.

G corresponds to the discrete gradient of the pressure field, and is currently implemented by the function **`assemble_grad_p`** in CitcomS. **`assemble_grad_p`** takes a $(nel \times 1)$ pressure vector p and returns a $(neq \times 1)$ vector ∇p. Thus G is a $(neq \times nel)$ matrix operator. nel is the number of elements.

G^T corresponds to the discrete divergence of the velocity field, and is currently implemented by the function **`assemble_div_u`** in CitcomS. **`assemble_div_u`** takes a $(neq \times 1)$ velocity vector u and returns a $(nel \times 1)$ vector $\nabla \cdot u$. Thus G^T is a $(nel \times neq)$ matrix operator.

Consistency requires that the 0 block be of dimension $(nel \times nel)$.

1.1 Uzawa Algorithm

Equation (1) can be solved using the *Uzawa algorithm*, in which the block matrix equation is broken into two coupled systems of equations

$$KV + GP = F \tag{2}$$

$$G^T V = 0 \tag{3}$$

The Schur complement system for pressure is formed by combining the two equations and eliminating V using equation (3)

$$\hat{K}P = G^T K^{-1} F \tag{4}$$

where $\hat{K} = G^T K^{-1} G$. The presence of K^{-1} makes the direct solution of equation unfeasible. However Krylov (iterative) methods can be used to solve the system of equations without computing any inverses.

First, with an initial guess pressure $P_0 = 0$, the initial velocity V_0 can be obtained by solving

$$KV_0 = F \tag{5}$$

and the initial pressure residual is computed as

$$r_0 = G^T K^{-1} F = G^T V_0 \tag{6}$$

This is also the initial "search direction" ($s_1 = r_0$).

To compute the search step α_k in the conjugate gradient algorithm, we need to compute the product of search direction s_k and $\hat{K}$, $s_k^T \hat{K} s_k$. This product can be computed without explicitly evaluating $\hat{K}$ as follows:

$$s_k^T \hat{K} s_k = s_k^T G^T K^{-1} G s_k = (G s_k)^T K^{-1} G s_k \tag{7}$$

Next we define u_k as the solution of

$$K u_k = G s_k \tag{8}$$

which means $u_k = K^{-1} G s_k$. Substituting this in equation (7), we get

$$s_k^T \hat{K} s_k = (G s_k)^T K^{-1} G s_k = (G s_k)^T u_k \tag{9}$$

Similarly, the product $\hat{K} s_k$, needed to update the residual r_k, can be computed as

$$\hat{K} s_k = G^T K^{-1} G s_k = G^T u_k \tag{10}$$

Pressure and velocity are updated as:

$$P_k = P_{k-1} + \alpha_k s_k \tag{11}$$

$$V_k = V_{k-1} - \alpha_k u_k \tag{12}$$

All of the above steps are summarized below

Listing 1: Uzawa algorithm

$k = 0; P_0 = 0$
`Solve` $KV_0 = F$
$s_1 = r_0 = G^T V_0$
`while` $|r_k| > \epsilon$

$k = k + 1$
if $k > 1$
$\beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}$
$s_k = r_{k-1} + \beta_k s_{k-1}$
end
Solve $Ku_k = Gs_k$
$\alpha_k r_{k-1}^T r_{k-1} / (Gs_k)^T u_k$
$P_k = P_{k-1} + \alpha_k s_k$
$V_k = V_{k-1} - \alpha_k u_k$
$r_k = r_{k-1} - \alpha_k G^T u_k$
end
$P = P_k; V = V_k$

1.2 Schur Complement Reduction

Applying block Gaussian elimination to equation (1), we get

$$\begin{pmatrix} K & G \\ 0 & S \end{pmatrix} \begin{pmatrix} V \\ P \end{pmatrix} = \begin{pmatrix} F \\ H \end{pmatrix} \tag{13}$$

where $H = G^T K^{-1} F$ and the Schur complement S is given by $S = G^T K^{-1} G$. V and P are obtained by solving the following systems for P and V, respectively

$$SP = H \tag{14}$$

$$KV = F - GP \tag{15}$$

The explicit construction of S is difficult, due to difficulties computing K^{-1}. Hence we adopt a **matrix-free** representation for S, which basically means the matrix-vector product $y = Sx$ needs to be somehow defined without actually computing the elements of S. To compute the matrix-vector product $y = Sx$, the following steps are carried out:

Compute

$$\hat{f} = Gx \tag{16}$$

Solve

$$K\hat{u} = \hat{f} \tag{17}$$

which implies $\hat{u} = K^{-1}\hat{f}$

Compute

$$y = G^T \hat{u} \tag{18}$$

which implies $y = G^T K^{-1} \hat{f} = G^T K^{-1} G x = Sx$ as desired.

The method used to obtain $\hat{u}$ in equation (17) is called the **inner solver** while the method applied to obtain P in equation (14) is called the **outer solver**.

Preconditioning equation (14) is essential to minimize the number of outer iterations required for convergence.

2 PETSc Implementation

2.1 The Uzawa algorithm

The PETSc version of the Uzawa algorithm can be turned on by the following settings

```
use_petsc=on
petsc_schur=off
```

The following listing shows an overview of this implementation:

Listing 2: Uzawa Algorithm

```
  // Create the force Vec
  ierr = VecCreateMPI( PETSC_COMM_WORLD, neq, PETSC_DECIDE, &FF );
  CHKERRQ(ierr);
  double *F_tmp;
  ierr = VecGetArray( FF, &F_tmp ); CHKERRQ( ierr );
  for( i = 0; i < neq; i++ )
    F_tmp[i] = F[i];
  ierr = VecRestoreArray( FF, &F_tmp ); CHKERRQ( ierr );

  // create the pressure vector and initialize it to zero
  ierr = VecCreateMPI( PETSC_COMM_WORLD, nel, PETSC_DECIDE, &P_k );
  CHKERRQ(ierr);
  ierr = VecSet( P_k, 0.0 ); CHKERRQ( ierr );

  // create the velocity vector
  ierr = VecCreateMPI( PETSC_COMM_WORLD, neq, PETSC_DECIDE, &V_k );
  CHKERRQ(ierr);

  // Copy the contents of V into V_k
  PetscScalar *V_k_tmp;
  ierr = VecGetArray( V_k, &V_k_tmp ); CHKERRQ( ierr );
  for( i = 0; i < neq; i++ )
    V_k_tmp[i] = V[i];
  ierr = VecRestoreArray( V_k, &V_k_tmp ); CHKERRQ( ierr );

 /* initial residual r1 = div(V) */
  ierr = MatMult( E->D, V_k, r_1 ); CHKERRQ( ierr );
```

```
/* add the contribution of compressibility to the initial residual */
if( E->control.inv_gruneisen != 0 ) {
  // r_1 += cu
  ierr = VecAXPY( r_1, 1.0, cu ); CHKERRQ( ierr );
}

E->monitor.vdotv = global_v_norm2_PETSc( E, V_k );
E->monitor.incompressibility = sqrt( global_div_norm2_PETSc( E, r_1 ) / (1e-32 + E->monitor.vdotv) );

r0dotz0 = 0;
ierr = VecNorm( r_1, NORM_2, &r_1_norm ); CHKERRQ( ierr );

while( (r_1_norm > E->control.petsc_uzawa_tol) && (count < *steps_max) )
  {

    /* preconditioner BPI ~= inv(K), z1 = BPI*r1 */
    ierr = VecPointwiseMult( z_1, BPI, r_1 ); CHKERRQ( ierr );

    /* r1dotz1 = <r1, z1> */
    ierr = VecDot( r_1, z_1, &r1dotz1 ); CHKERRQ( ierr );
    assert( r1dotz1 != 0.0  /* Division by zero in head of incompressibility
                               iteration */);

    /* update search direction */
    if( count == 0 )
          {
            // s_2 = z_1
            ierr = VecCopy( z_1, s_2 ); CHKERRQ( ierr ); // s2 = z1
          }
    else
          {
            // s2 = z1 + s1 * <r1,z1>/<r0,z0>
            delta = r1dotz1 / r0dotz0;
            ierr = VecWAXPY( s_2, delta, s_1, z_1 ); CHKERRQ( ierr );
          }

    // Solve K*u_k = grad(s_2) for u_k
    ierr = MatMult( E->G, s_2, Gsk ); CHKERRQ( ierr );
    ierr = KSPSolve( E->ksp, Gsk, u_k ); CHKERRQ( ierr );
```

```
strip_bcs_from_residual_PETSc( E, u_k, lev );

// Duk = D*u_k ( D*u_k is the same as div(u_k) )
ierr = MatMult( E->D, u_k, Duk ); CHKERRQ( ierr );

// alpha = <r1,z1> / <s2,F>
ierr = VecDot( s_2, Duk, &s_2_dot_F ); CHKERRQ( ierr );
alpha = r1dotz1 / s_2_dot_F;

// r2 = r1 - alpha * div(u_k)
ierr = VecWAXPY( r_2, -1.0*alpha, Duk, r_1 ); CHKERRQ( ierr );

// P = P + alpha * s_2
ierr = VecAXPY( P_k, 1.0*alpha, s_2 ); CHKERRQ( ierr );

// V = V - alpha * u_1
ierr = VecAXPY( V_k, -1.0*alpha, u_k ); CHKERRQ( ierr );
//strip_bcs_from_residual_PETSc( E, V_k, E->mesh.levmax );

/* compute velocity and incompressibility residual */
E->monitor.vdotv = global_v_norm2_PETSc( E, V_k );
E->monitor.pdotp = global_p_norm2_PETSc( E, P_k );
v_norm = sqrt( E->monitor.vdotv );
p_norm = sqrt( E->monitor.pdotp );
dvelocity = alpha * sqrt( global_v_norm2_PETSc( E, u_k ) / (1e-32 + E->monitor.vdotv) );
dpressure = alpha * sqrt( global_p_norm2_PETSc( E, s_2 ) / (1e-32 + E->monitor.pdotp) );

// compute the updated value of z_1, z1 = div(V)
ierr = MatMult( E->D, V_k, z_1 ); CHKERRQ( ierr );
if( E->control.inv_gruneisen != 0 )
      {
  // z_1 += cu
  ierr = VecAXPY( z_1, 1.0, cu ); CHKERRQ( ierr );
}

E->monitor.incompressibility = sqrt( global_div_norm2_PETSc( E, z_1 ) / (1e-32 + E->monitor.vdotv) );

count++;
```

```
    if( E->control.print_convergence && E->parallel.me == 0 ) {
      print_convergence_progress( E, count, time0,
                                  v_norm, p_norm,
                                  dvelocity, dpressure,
                                  E->monitor.incompressibility );
    }

    /* shift array pointers */
    ierr = VecSwap( s_2, s_1 ); CHKERRQ( ierr );
    ierr = VecSwap( r_2, r_1 ); CHKERRQ( ierr );

    /* shift <r0, z0> = <r1, z1> */
    r0dotz0 = r1dotz1;

    // recompute the norm
    ierr = VecNorm( r_1, NORM_2, &r_1_norm ); CHKERRQ( ierr );

  } /* end loop for conjugate gradient */
```

2.2 Schur Complement Reduction

Schur complement reduction be turned on by the following settings

```
use_petsc=on
petsc_schur=on
```

The following listing shows an overview of Schur complement reduction implementation:

Listing 3: Schur Complement Reduction

```
  /*---------------------------------*/
  /* Define a Schur complement matrix */
  /*---------------------------------*/
  ierr = MatCreateSchurComplement(E->K,E->K,E->G,E->D,PETSC_NULL, &S);
  CHKERRQ(ierr);
  ierr = MatSchurComplementGetKSP(S, &inner_ksp); CHKERRQ(ierr);
  ierr = KSPGetPC(inner_ksp, &inner_pc); CHKERRQ(ierr);

  /*-------------------------------------------------*/
  /* Build the RHS of the Schur Complement Reduction */
```

```
  /*-----------------------------------------------*/
  ierr = MatGetVecs(S, PETSC_NULL, &fhat); CHKERRQ(ierr);
  ierr = KSPSolve(inner_ksp, FF, t1); CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(inner_ksp, &inner1); CHKERRQ(ierr);
  ierr = MatMult(E->D, t1, fhat); CHKERRQ(ierr);

  /*-----------------------------------------*/
  /* Build the solver for the Schur complement */
  /*-----------------------------------------*/
  ierr = KSPCreate(PETSC_COMM_WORLD, &S_ksp); CHKERRQ(ierr);
  ierr = KSPSetOperators(S_ksp, S, S); CHKERRQ(ierr);
  ierr = KSPSetType(S_ksp, "cg"); CHKERRQ(ierr);
  ierr = KSPSetInitialGuessNonzero(S_ksp, PETSC_TRUE); CHKERRQ(ierr);

  /*--------------------*/
  /* Solve for pressure */
  /*--------------------*/
  ierr = KSPSolve(S_ksp, fhat, PVec); CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(inner_ksp, &outer); CHKERRQ(ierr);

  /*--------------------*/
  /* Solve for velocity */
  /*--------------------*/
  ierr = MatGetVecs(E->K, PETSC_NULL, &fstar); CHKERRQ(ierr);
  ierr = MatMult(E->G, PVec, fstar); CHKERRQ(ierr);
  ierr = VecAYPX(fstar, 1.0, FF); CHKERRQ(ierr);
  ierr = KSPSetInitialGuessNonzero(inner_ksp, PETSC_TRUE); CHKERRQ(ierr);
  ierr = KSPSolve(inner_ksp, fstar, VVec); CHKERRQ(ierr);
  ierr = KSPGetIterationNumber(inner_ksp, &inner2); CHKERRQ(ierr);

}
```

2.3 Shell Matrices in PETSc

PETSc allows us to define matrices of type **MATSHELL** which are matrices for which various matrix operations are defined without first actually having to compute each element of the matrix. This is identical to the matrix-free approach we discussed earlier.

For example, K corresponds to the discrete Laplacian of the velocity field, and is currently implemented by the function **assemble_del2_u** in CitcomS. The matrix corresponding to K will be implemented as follows in CitcomS.

First we create a shell matrix:

Listing 4: Creating a shell matrix

```
// stores all the matrix specific information
struct MatMultShell mtx;
mtx.E = E;
mtx.level = E->mesh.levmax;
mtx.neq = E->lmesh.neq;
mtx.nel = E->lmesh.nel;

Mat K;
// create K as a shell matrix
MatCreateShell( PETSC_COMM_WORLD,
                neq, neq, // local dims
                PETSC_DETERMINE, PETSC_DETERMINE, // global dims
                (void *)&mtx, &K );
```

Next we define all the desired operations for our shell matrix. In CitcomS, we only need to define the multiplication operation.

Listing 5: Matrix operations for shell matrix

```
MatShellSetOperation( K, MATOP_MULT,
                      (void(*)(void))MatShellMult_del2_u );
```

Here **MATOP_MULT** means we are defining the matrix multiplication operation for K, and **MatShellMult_del2_u** is the actual function that defines the matrix multiplication. Anytime $y = Kx$ needs to be computed, PETSc will call **MatShellMult_del2_u(K,x,y)** where **x** and **y** are PETSc vectors (**Vec**). The definition of **MatShellMult_del2_u** is as follows

Listing 6: MatShellMult_del2_u

```
PetscErrorCode MatShellMult_del2_u( Mat K, Vec X, Vec Y )
{
  int i, j, neq, nel;
  PetscErrorCode ierr;

  // recover the matrix specific information
  struct MatMultShell *ctx;
  MatShellGetContext( K, (void **)&ctx );
  neq = ctx->neq;
  nel = ctx->nel;
```

```
  // transfer data from PETSc to CitcomS
  for( i = 1; i <= ctx->E->sphere.caps_per_proc; i++ ) {
    // first neq elements of X are copied into ctx->u
    for( j = 0; j < neq; j++ ) {
      ierr = VecGetValues( X, 1, &j, &ctx->u[i][j] );
      CHKERRQ( ierr );
    }
  }

  // carry out the actual CitcomS operation
  assemble_del2_u( ctx->E, ctx->u, ctx->Ku, ctx->level, 1 );

  // Transfer data from CitcomS to PETSc
  for( j = 0; j < neq; j++ ) {
    ierr = VecSetValues( Y, 1, &j, &ctx->Ku[1][j], INSERT_VALUES );
    CHKERRQ( ierr );
  }
  VecAssemblyBegin( Y );
  VecAssemblyEnd( Y );

  PetscFunctionReturn(0);
}
```

We can similarly define **MatShellMult_grad_p** for G and **MatShellMult_div_u** for G^T. Once these have been defined, we can implement the Uzawa algorithm and the Schur complement reduction approach using PETSc.