# PETSCEXT-V3.0.0: BLOCK EXTENSIONS TO PETSC

DAVE A. MAY

## 1. Overview

The discrete form of coupled partial differential equations require some ordering of the unknowns. For example, fluid flow problems involving velocity $\boldsymbol{v}$ and pressure $p$, might have their associated approximate nodal values ordered like $(u_1, v_1, p_1, u_2, v_2, p_2, \dots, u_N, v_N, p_N)^T$, where $u_i, v_i$ represent the x-component and y-component of the velocity field at node $i$. Implicit in this ordering is the assumption that we have the same number of unknowns for velocity and pressure. If we break this assumption, a more natural ordering may be to keep the like quantities grouped together, e.g. $(u_1, v_1, \dots, u_N, v_N, p_1, \dots p_N)^T$. In this case, we can separate the discrete solution into two vectors $\boldsymbol{x} = (\boldsymbol{u}, \boldsymbol{p})^T$.

Physcially motivated splitting such as this can be desirable in solving coupled, multi-physics problems as they allow application programmers to apply efficient solution techniques to the individual operators or blocks. The block objects introduced in this library are designed to enable systems of linear equations to be described and solved at the block level. The block objects introduced are built within the PETSc [1] library.

## 2. Block vectors

A block vector, $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N)^T$, where $\boldsymbol{x} \in \mathcal{R}^N$ and $\boldsymbol{x}_1 \in \mathcal{R}^{n_1}, \dots, \boldsymbol{x}_N \in \mathcal{R}^{n_N}$ can be defined. Each of the sub vectors $\boldsymbol{x}_i$, $i = 1, \dots, N$ may be any vector type supported by PETSc (VECSEQ, VECMPI, etc). This may include sub vectors which are themselves blocks, however not all vector operations support nested block vectors. There are no restrictions on the length of each sub vector. The creation of a block vector is shown below

```
Vec xb;

VecCreate( PETSC_COMM_WORLD, &xb );
/* Define a 3 block vector */
VecSetSizes( xb, 3, 3 );
VecSetType( xb, "block" );
```

The length of a block vector corresponds to the number of sub vectors within the block. When defining vectors of type "block", you must always specify the local ($n$) and global ($N$) sizes to be the same.

By default when a block vector of size $N$ is created, $N$ sub vectors will also be created. The $i^{th}$ sub vectors can be extracted using

```
PetscErrorCode VecBlockGetSubVector( Vec X, PetscInt idxm, Vec *sx )
```

where `X` is the block vector, `idxm` is index of the sub vector and `*sx` is the pointer to sub vector object. Note that `idxm` assumed a 0-based indexing system. Alternatively one can fetch all the sub vectors using

```
PetscErrorCode VecBlockGetSubVectors( Vec X, Vec **sx )
```

The array of sub vectors `*sx` should never be freed by the user. Whenever the sub vectors are no longer in use the user should always call

```
PetscErrorCode VecBlockRestoreSubVectors( Vec x )
```

on the block vector

The sub vectors of a block vector can be specified by fetching the sub vector using `VecBlockGetSubVector()` and then configuring (i.e. setting its size and type) the returned vector. Alternatively one can call

```
PetscErrorCode VecBlockSetValue( Vec x, PetscInt idxm, Vec vec, InsertMode addv )
```

*Date*: February 3, 2009.

to set a single sub vector `vec` into the `idxm` location within the block vector `x`. When `addv = INSERT_VALUES`, the sub vector at location `idxm` will be destroyed and then be replaced by `vec`. Multiple sub vectors can be specified at one using

```
PetscErrorCode VecBlockSetValues( Vec V, PetscInt m, const PetscInt idxm[],
    const Vec vec[], InsertMode addv )
```

The block vector object does not support sub vectors which are null. Once the sub vectors have been set, one should call `VecAssemblyBegin()` and `VecAssemblyEnd()` on the block vector to indicate that no more entries (which in this case are vectors) are to be inserted.

2.1. **Example.** In the following example, we will describe the block vector $\boldsymbol{x}_b = (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)^T$.

```
Vec xb;
Vec x1,x2,x3;

/* user defines x1,x2,x3 */
.
.
.

VecCreate( PETSC_COMM_WORLD, &xb );
VecSetSizes( xb, 3, 3 );
VecSetType( xb, "block" );
VecBlockSetValue( xb, 0, x1, INSERT_VALUES );
VecBlockSetValue( xb, 1, x2, INSERT_VALUES );
VecBlockSetValue( xb, 2, x3, INSERT_VALUES );
VecAssemblyBegin( xb );
VecAssemblyEnd( xb );

/* pass control of destroying sub vectors to the block */
VecDestroy( x1 );
VecDestroy( x2 );
VecDestroy( x3 );

/* do some operations with the block vector xb */
.
.
.

VecDestroy( xb );
```

## 3. Block matrices

We define a block matrix $\boldsymbol{A}$ as

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \dots & \boldsymbol{A}_{1N} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & \dots & \boldsymbol{A}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{A}_{M1} & \boldsymbol{A}_{M2} & \dots & \boldsymbol{A}_{MN} \end{pmatrix}, \qquad \boldsymbol{A} \in \mathcal{R}^{M\times N}. \tag{1}$$

Each of the sub matrices $\boldsymbol{A}_{ij}$, $i = 1, \dots, M$, $j = 1, \dots, N$ may be any matrix type supported by PETSc. This may include sub matrices which are themselves blocks, however not all matrix operations support nested block matrices.

The creation of a block matrix is shown below

```
Mat Ab;
```

```
MatCreate( PETSC_COMM_WORLD, &Ab );
/* Define a 3x3 block matrix */
MatSetSizes( Ab, 3,3, 3,3 );
MatSetType( Ab, "block" );
```

The dimension $M \times N$ of a block matrix corresponds to the number of rows and columns of sub matrices within the block. When defining matrices of type "block", you must always specify the local rows ($m$) and global rows ($M$) to be the same size. Similarly the local ($n$) and global ($N$) columns must also be the same size.

Upon initial construction of a block matrix, all the $M \times N$ sub matrices are null. The block matrix may be sparse (i.e. contain null blocks), but the user is required to set all non-zero sub matrices to define the sparsity pattern of the block matrix. Sub matrices can be set into a block matrix using either

```
PetscErrorCode MatBlockSetValues( Mat A,
    PetscInt m, const PetscInt idxm[],
    PetscInt n ,const PetscInt idxn[],
    const Mat mat[], MatStructure str, InsertMode addv )

PetscErrorCode MatBlockSetValue( Mat A, PetscInt idxm, PetscInt idxn,
    Mat mat, MatStructure str, InsertMode addv )
```

The argument `str` is only used for `addv=ADD_VALUES`. When each sub matrix $\boldsymbol{A}_{ij}$ is inserted into the block matrix $\boldsymbol{A}$, we check that the dimension of $\boldsymbol{A}_{ij}$ (global row and column size) is compatible with the other sub matrices already existing within the block structure. Once the sub matrices have been set, the user should call `MatAssemblyBegin()` and `MatAssemblyEnd()`.

The sub matrices can be extracted using the following functions;

```
PetscErrorCode MatBlockGetSubMatrix( Mat A, PetscInt ridx, PetscInt cidx, Mat *sa )

PetscErrorCode MatBlockGetSubMatrices( Mat A, Mat ***sa )
```

When the user is finished manipulating the sub blocks, one should always call

```
PetscErrorCode MatBlockRestoreSubMatrices( Mat A )
```

The block matrix can generate a new compatible sub matrix for a particular location within the block using

```
PetscErrorCode MatBlockCreateSubMatrix( Mat A,
    MatType mtype, PetscInt r, PetscInt c, Mat *sA )
```

The sub matrix `sA` can only be generated if the block already contains at least one sub matrix within row `r`, and column `c`.

## 4. Block preconditioners

We allow three basic types of block preconditioners to be defined; diagonal, upper triangular and lower triangular, i.e.

$$\hat{\boldsymbol{A}}_D = \begin{pmatrix} \hat{\boldsymbol{A}}_{11} & & \\ & \ddots & \\ & & \hat{\boldsymbol{A}}_{MN} \end{pmatrix}, \quad \hat{\boldsymbol{A}}_U = \begin{pmatrix} \hat{\boldsymbol{A}}_{11} & \dots & \boldsymbol{A}_{1N} \\ & \ddots & \vdots \\ & & \hat{\boldsymbol{A}}_{MN} \end{pmatrix}, \quad \hat{\boldsymbol{A}}_L = \begin{pmatrix} \hat{\boldsymbol{A}}_{11} & & \\ \vdots & \ddots & \\ \boldsymbol{A}_{M1} & \dots & \hat{\boldsymbol{A}}_{MN} \end{pmatrix}. \tag{2}$$

Assuming we have the block matrix `Ab` available, a block preconditioner can be created via

```
PC pcA;

PCCreate( PETSC_COMM_WORLD, &pcA );
PCSetOperators( pcA, Ab, Ab, SAME_NONZERO_PATTERN );
PCSetType( pcA, "block" );
```

To facilitate inversion of the diagonal, blocks, the block preconditioner contains a `KSP` for each diagonal entry. By default, the operator used by the $i^{th}$ Krylov method is taken from $(i, i)$ sub matrix of the preconditioner matrix passed to the call `PCSetOperators()` used to configure the block preconditioner. Each Krylov method on the diagonal is by default, set to be of type "preonly" and will use the option prefix `pc_block_Qii` where $i$ corresponds to the row index. Note that the $i$ used in the options prefix uses indices starting from 1. For example `-pc_block_Q11_pc_type ilu` will configure the first KSP in (2) to be $ILU(0)$. The default ksp type can easily be over ridden from the command line via `-pc_block_Q11_ksp_type cg` for example.

The `KSP` along the diagonal can be manipulated using the functions

```
PetscErrorCode PCBlockGetSubKSP( PC pc, PetscInt i, KSP *sub_ksp )
PetscErrorCode PCBlockSetSubKSP( PC pc, PetscInt i, KSP sub_ksp )
```

Should the $(i, i)$ sub matrix in the preconditioner matrix be null, we can use `PCBlockSetSubKSP()` to define a suitable preconditioner for the $(i, i)$ component of the diagonal.

The default block preconditioner is diagonal. This can be changed

```
PetscErrorCode PCBlockSetBlockType( PC pc, PCBlockType bt )
```

where

```
bt = { PC_BLOCK_DIAGONAL, PC_BLOCK_UPPER, PC_BLOCK_LOWER }
```

The textual name of the block type can be obtained using

```
PetscErrorCode PCBlockGetBlockType( PC pc, const char **bt )
```

The block type can be changed from via the command line argument `-pc_block_type TYPE`, where `TYPE` is one of `<diagonal, upper, lower>`.

## 5. Schur complement

Given a $2 \times 2$ block system,

$$\begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{pmatrix}, \tag{3}$$

the following schur complement systems can be constructed

$$\boldsymbol{S}_{11}\boldsymbol{x}_2 = \hat{\boldsymbol{f}}_1, \quad \text{where } \boldsymbol{S}_{11} = \boldsymbol{A}_{21}\boldsymbol{A}_{11}^{-1}\boldsymbol{A}_{12} - \boldsymbol{A}_{22}, \quad \hat{\boldsymbol{f}}_1 = \boldsymbol{A}_{21}\boldsymbol{A}_{11}^{-1}\boldsymbol{b}_1 - \boldsymbol{b}_2, \tag{4}$$

and

$$\boldsymbol{S}_{22}\boldsymbol{x}_1 = \hat{\boldsymbol{f}}_2, \quad \text{where } \boldsymbol{S}_{22} = \boldsymbol{A}_{12}\boldsymbol{A}_{22}^{-1}\boldsymbol{A}_{21} - \boldsymbol{A}_{11}, \quad \hat{\boldsymbol{f}}_2 = \boldsymbol{A}_{12}\boldsymbol{A}_{22}^{-1}\boldsymbol{b}_2 - \boldsymbol{b}_1. \tag{5}$$

We note that this definition of the Schur complement negative of what is commonly used, thus we allow the system in (4) and (5) to be scaled by a parameter $\alpha$.

Given the four matrices `A11, A12, A21, A22`, we define a matrix of type "schur" via

```
PetscErrorCode MatCreateSchur( MPI_Comm comm,
    Mat A11, Mat A12, Mat A21, Mat A22,
    PetscScalar alpha, MatSchurComplementType type, Mat *A )
```

where the type of Schur complement ($\boldsymbol{S}_{11}$ or $\boldsymbol{S}_{22}$) is selected by selecting from

```
MatSchurComplementType = { "MatSchur_A11", "MatSchur_A22" }
```

A Schur complement matrix can also be constructed from a $2 \times 2$ block matrix (rather than having to specify the individual operators) via;

```
PetscErrorCode MatCreateSchurFromBlock( Mat bmat, PetscScalar alpha,
    MatSchurComplementType type, Mat *A )
```

We support a null $(2, 2)$ block when using "MatSchur_A11", and a null $(1, 1)$ when using "MatSchur_A22".

The vectors $\hat{\boldsymbol{f}}_1$ and $\hat{\boldsymbol{f}}_2$ can be constructed using the functions

```
PetscErrorCode MatSchurApplyReductionToVec( Mat A, Vec f1, Vec f2, Vec subb )
PetscErrorCode MatSchurApplyReductionToVecFromBlock( Mat A, Vec F, Vec subb )
```

The vector `subb` needs to be create by the user. The simplest way to do this is to call `MatGetVecs(schur, PETSC_NULL, &subb )` on the schur complement matrix. Presently these functions do not support using null vectors. In the future I will add support to allow a null $b_2$ block when using "MatSchur_A11", and a null $b_1$ when using "MatSchur_A22".

The matrix type "schur" is a matrix free representation of the Schur complement. This representation avoids the need to explicitly construct the inverse operator $\boldsymbol{A}_{11}^{-1}$ or $\boldsymbol{A}_{22}^{-1}$. To define the operation $\boldsymbol{y} = \boldsymbol{S}\boldsymbol{x}$, we define the inverse via a Krylov method. The `KSP` used to define the inverse operator acting on a vector ($\boldsymbol{y} = \boldsymbol{A}_{11}^{-1}\boldsymbol{x}$ or $\boldsymbol{y} = \boldsymbol{A}_{22}^{-1}\boldsymbol{x}$) can be manipulated using the following functions.

```
PetscErrorCode MatSchurGetKSP( Mat A, KSP *ksp )
PetscErrorCode MatSchurSetKSP( Mat A, KSP ksp )
```

By default, the `KSP` used to define the inverse operator will have the option prefix `"mat_schur_"`. Thus to configure this object from the command line to be of type "cg" for example, one would do `-mat_schur_ksp_type cg`.

Other interface to the Schur complement matrix are shown below

```
PetscErrorCode MatSchurGetSchurComplementType( Mat A,
    MatSchurComplementType *type )
PetscErrorCode MatSchurSetSchurComplementType( Mat A,
    MatSchurComplementType type )

PetscErrorCode MatSchurGetScalar( Mat A, PetscScalar *alpha )
PetscErrorCode MatSchurSetScalar( Mat A, PetscScalar alpha )
```

## 6. Miscellaneous Extensions

**MatSymTrans.** Description: Allows for the symbolic definition of the transpose of a matrix.

- `PetscErrorCode MatCreateSymTrans( MPI_Comm comm, Mat A, Mat *symAt )`.
  Given $\boldsymbol{A}$, this will create a new operator which symbolically defines $\boldsymbol{A}^T$. Note that the operation `MatMult(symAt,x,y)` is identical to `MatMultTranspose(A,x,y)`.
- `PetscErrorCode MatSymTransGetOperator( Mat symAt, Mat *A )`
  Returns the original operator in `A`
- `PetscErrorCode MatSymTransGetExplicitOperator( Mat symAt, Mat *A )`
  Numerically constructs the transpose operator. This will probably never be used. If you actually require the $(i, j)$ components of the transpose operator, you might have been better off just using `MatTranspose`.

## 7. Examples

### 7.1. Schur Complement Reduction (SCR) to solve Stokes equations.

The discrete form of the Stokes equations will be denoted as

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{G} \\ \boldsymbol{G}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{h} \end{pmatrix}. \tag{6}$$

We will solve $\boldsymbol{G}^T\boldsymbol{A}^{-1}\boldsymbol{G}\boldsymbol{p} = \boldsymbol{G}^T\boldsymbol{A}^{-1}\boldsymbol{f} - \boldsymbol{h}$ for the pressure and then solve $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f} - \boldsymbol{G}\boldsymbol{p}$ for velocity. In this example we will assume that the matrices `A,G` and vectors `u,p,f,h` have been provided by the user. We will assume that $u, p$ may contain a nonzero value which we will use as our initial guess.

```
Mat S, Gtrans;
Vec fhat, fstar;
KSP S_ksp, inner_ksp;
PC S_pc, inner_pc;

/* Represent the (2,1) block as symbolic, G^t */
MatCreateSymTrans( PETSC_COMM_WORLD, G, &Gtrans );
```

```
/* Define a schur complement matrix */
MatCreateSchur( PETSC_COMM_WORLD, A, G, Gtrans, PETSC_NULL,
    PETSC_NULL, "MatSchur_A11", &S );
 MatAssemblyBegin( S, MAT_FINAL_ASSEMBLY );
 MatAssemblyEnd( S, MAT_FINAL_ASSEMBLY );

/* Configure definition of A^{-1} */
MatSchurGetKSP( S, &inner_ksp );
KSPSetType( inner_ksp, "preonly" );
KSPGetPC( inner_ksp, &inner_pc );
PCSetType( inner_pc, "lu" );

/* Build the SCR rhs */
MatGetVecs( S, PETSC_NULL, &fhat );
MatSchurApplyReductionToVec( S, f,h, fhat );

/* Build solver for schur complement */
KSPCreate( PETSC_COMM_WORLD, &S_ksp );
KSPSetOperators( S_ksp, S, S, SAME_NONZERO_PATTERN );
KSPGetPC( S_ksp, &S_pc );
/* configure solver */
KSPSetType( S_ksp, "cg" );
PCSetType( S_pc, "none" );
KSPSetInitialGuessNonzero( S_ksp, PETSC_TRUE );

/* Solve for pressure */
KSPSolve( S_ksp, fhat, p );

/* Solve for velocity */
MatGetVecs( A, PETSC_NULL, &fstar );
MatMult( G, p, fstar );
VecAYPX( fstar, -1.0, f );  /* fstar <- -fstar + f */

MatSchurGetKSP( S, &inner_ksp );
KSPSetInitialGuessNonzero( inner_ksp, PETSC_TRUE );
KSPSolve( inner_ksp, fstar, u );

/* tidy up */
VecDestroy( fstar );
VecDestroy( fhat );
MatDestroy( S );
KSPDestroy( S_ksp );
```

7.2. **Elman style coupled block solve for Stokes equations.** The discrete form of the Stokes equations will be denoted as

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{G} \\ \boldsymbol{G}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{h} \end{pmatrix}. \tag{7}$$

We wish to solve the stokes flow equations in a fully coupled fashion. The system will be right preconditioned with an upper triangular matrix denoted as

$$\hat{\mathcal{A}}^{-1} = \begin{pmatrix} \hat{\boldsymbol{A}} & \boldsymbol{G} \\ \boldsymbol{0} & -\hat{\boldsymbol{S}} \end{pmatrix}^{-1}, \tag{8}$$

where $\hat{\boldsymbol{S}}$ is an approximate Schur complement.

In this example we will assume that the matrices `A, G` and vectors `u,p,f,h` have been provided by the user. We also assume that a matrix approximating the Schur complement `s_hat` has been provided and the both $u, p$ may contain a nonzero value which we will use our initial guess.

```
Mat stokes_A, Gtrans;
Vec stokes_x, stokes_b;
KSP stokes_ksp, sub_ksp, ksp_Q22;
PC stokes_pc, sub_pc, pc_Q22;

/* Build a block system for the Stokes operator */
MatCreate( PETSC_COMM_WORLD, &stokes_A );
MatSetSizes( stokes_A, 2,2, 2,2 );
MatSetType( stokes_A, "block" );

/* Represent the (2,1) block as symbolic, G^t */
MatCreateSymTrans( PETSC_COMM_WORLD, G, &Gtrans );

/* Prescribe sub matrices within the block */
MatBlockSetValue( stokes_A, 0,0, A,  DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );
MatBlockSetValue( stokes_A, 0,1, G,  DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );
MatBlockSetValue( stokes_A, 1,0, Gtrans, DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );
/* Assemble */
MatAssemblyBegin( stokes_A, MAT_FINAL_ASSEMBLY );
MatAssemblyEnd( stokes_A, MAT_FINAL_ASSEMBLY );

/* Create compatible block vectors for solution and rhs */
MatGetVecs( stokes_A, &stokes_x, &stokes_b );

/* Prescribe sub vectors within the rhs */
VecBlockSetValue( stokes_b, 0, f, INSERT_VALUES );
VecBlockSetValue( stokes_b, 1, h, INSERT_VALUES );
VecAssemblyBegin( stokes_b );
VecAssemblyEnd( stokes_b );

/* Prescribe sub vectors within the solution */
VecBlockSetValue( stokes_x, 0, u, INSERT_VALUES );
VecBlockSetValue( stokes_x, 1, p, INSERT_VALUES );
VecAssemblyBegin( stokes_x );
VecAssemblyEnd( stokes_x );

/* Create a solver */
KSPCreate( PETSC_COMM_WORLD, &stokes_ksp );
/* Name the ksp for the fully coupled block operator */
KSPSetOptionsPrefix( stokes_ksp, "fc_");
KSPSetOperators( stokes_ksp, stokes_A, stokes_A, SAME_NONZERO_PATTERN );
KSPSetType( stokes_ksp, "fgmres" );

/* configure stokes pc to be block */
KSPGetPC( stokes_ksp, &stokes_pc );
PCSetType( stokes_pc, "block" );
KSPSetInitialGuessNonzero( stokes_ksp, PETSC_TRUE );
KSPSetFromOptions( stokes_ksp );
```

```
/* Configure sub ksp's on the block pc */

/* Q11 will be an iterative solve */
PCBlockGetSubKSP( stokes_pc, 0, &sub_ksp );
KSPGetPC( sub_ksp, &sub_pc );
KSPSetType( sub_ksp, "cg" );
PCSetType( sub_pc, "ilu" );

/* Q22 will be defined as LU applied to -s_hat */
MatScale( s_hat, -1.0 );
KSPCreate( PETSC_COMM_WORLD, &ksp_Q22 );
/* give Q22 a name to enable unique cmd. line configuration */
KSPSetOptionsPrefix( ksp_Q22, "Q22_);
KSPSetOperators( ksp_Q22, s_hat, s_hat, SAME_NONZERO_PATTERN );
KSPSetType( ksp_Q22, "preonly" );
KSPGetPC( ksp_Q22, &pc_Q22 );
PCSetType( pc_Q22, "lu" );
KSPSetFromOptions( ksp_Q22 );

PCBlockSetSubKSP( stokes_pc, 1, &sub_ksp );
/* Hand control back to stokes_pc */
KSPDestroy( ksp_Q22 );

/* perform solve */
KSPSolve( stokes_ksp, stokes_b, stokes_x );

/* tidy up */
VecDestroy( stokes_x );
VecDestroy( stokes_b );
MatDestroy( stokes_A );
MatDestroy( Gtrans );
KSPDestroy( stokes_ksp );
```

It should be stressed that when the block preconditioner is created the operators (`Amat,Pmat`) are used to construct each the operators for each $(i, i)$ KSP. Thus in this example, initially the KSP at (2,2) block within `stokes_pc` will have NULL operators for both (`Amat,Pmat`). This is not an issue provided that a KSP is created and suitable operators are set for `Amat,Pmat` and this KSP replaces the NULL operator KSP within the block preconditioner. This was the approach adopted here. When you do not have an explicit representation of $\hat{\boldsymbol{S}}$, this approach should be followed.

When $\hat{\boldsymbol{S}}$ is an explicitly formed, a cleaner approach would be to define a different block matrix for the block preconditioner. Such was the situation in the example above. In this case, assuming we still have access to `s_hat` and `G`, in the above example one could have done;

```
Mat stokes_Pmat;

/* Build a block system for the Stokes block precondioning operator */
MatCreate( PETSC_COMM_WORLD, &stokes_Pmat );
MatSetSizes( stokes_Pmat, 2,2, 2,2 );
MatSetType( stokes_Pmat, "block" );

/* Prescribe sub matrices within the block */
MatBlockSetValue( stokes_Pmat, 0,0, A,  DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );
MatBlockSetValue( stokes_Pmat, 0,1, G,  DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );
```

```
/* set the operator defining the matrix for the Schur complement preconditioner */
MatScale( s_hat, -1.0 );
MatBlockSetValue( stokes_Pmat, 1,1, s_hat, DIFFERENT_NONZERO_PATTERN, INSERT_VALUES );

/* Assemble */
MatAssemblyBegin( stokes_Pmat, MAT_FINAL_ASSEMBLY );
MatAssemblyEnd( stokes_Pmat, MAT_FINAL_ASSEMBLY );
.
.
.
/* Create a solver for the fully coupled block system*/
KSPCreate( PETSC_COMM_WORLD, &stokes_ksp );
/* Name the ksp for the fully coupled block operator */
KSPSetOptionsPrefix( stokes_ksp, "fc_" );
KSPSetOperators( stokes_ksp, stokes_A, stokes_Pmat, DIFFERENT_NONZERO_PATTERN );
.
.
```

I

## References

[1] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficienct management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhauser Press, pp. 163–202.

School of Mathematical Sciences, Monash University, Clayton Victoria 3800, Australia
*URL*: http://www.maths.monash.edu.au/~dmay/PetscExt
*E-mail address*: dave.mayhem23@gmail.com, david.may@sci.monash.edu.au