Introduction to PETSc

Victor Eijkhout
Outline

- Introduction
- Getting started
- PETSc objects
- SNES: Nonlinear solvers
- TS: Time stepping
- Profiling, debugging
Introduction
To set the stage

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort. PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not black-box PDE solver, nor a silver bullet.

Barry Smith
More specifically. . .

Portable Extendable Toolkit for Scientific Computations

- Scientific Computations: parallel linear algebra, in particular linear and nonlinear solvers
- Toolkit: Contains high level solvers, but also the low level tools to roll your own.
- Portable: Available on many platforms, basically anything that has MPI

Why use it? It’s big, powerful, well supported.
What does PETSc target?

• Serial and Parallel
• Linear and nonlinear
• Finite difference and finite element
• Structured and unstructured
What is in PETSc?

- Linear system solvers (sparse/dense, iterative/direct)
- Nonlinear system solvers
- Tools for distributed matrices
- Support for profiling, debugging, graphical output
Documentation and help

- Web page: http://tinyurl.com/PETSc-man-page
- PETSc on TACC clusters: consulting through TACC/XSEDE user portal
- General questions about PETSc: petsc-maint@mcs.anl.gov
- Follow-up to this tutorial: eijkhout@tacc.utexas.edu
### Parallel Numerical Components of PETSc

#### Nonlinear Solvers

<table>
<thead>
<tr>
<th>Newton–based Methods</th>
<th>Line Search</th>
<th>Trust Region</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Krylov Subspace Methods

<table>
<thead>
<tr>
<th>GMRES</th>
<th>CG</th>
<th>CGS</th>
<th>Bi–CG–Stab</th>
<th>TFQMR</th>
<th>Richardson</th>
<th>Chebychev</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Time Steppers

<table>
<thead>
<tr>
<th>Euler</th>
<th>Backward Euler</th>
<th>Pseudo–Time Stepping</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Preconditioners

<table>
<thead>
<tr>
<th>Additive Schwarz</th>
<th>Block Jacobi</th>
<th>Jacobi</th>
<th>ILU</th>
<th>ICC</th>
<th>LU (sequential only)</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Matrices

<table>
<thead>
<tr>
<th>Compressed Sparse Row (AIJ)</th>
<th>Block Compressed Sparse Row (BAIJ)</th>
<th>Block Diagonal (BDiag)</th>
<th>Dense</th>
<th>Other</th>
</tr>
</thead>
</table>

#### Vectors

#### Index Sets

<table>
<thead>
<tr>
<th>Indices</th>
<th>Block Indices</th>
<th>Stride</th>
<th>Other</th>
</tr>
</thead>
</table>
External packages

PETSc does not do everything, but it interfaces to other software:

- Dense linear algebra: Scalapack, Plapack
- Grid partitioning software: ParMetis, Jostle, Chaco, Party
- ODE solvers: PVODE
- Eigenvalue solvers (including SVD): SLEPc
- Optimization: TAO
PETSc and parallelism

PETSc is layered on top of MPI

MPI has basic tools: send elementary datatypes between processors

PETSc has intermediate tools:
- insert matrix element in arbitrary location,
- do parallel matrix-vector product

⇒ you do not need to know much MPI when you use PETSc
PETSc and parallelism

All objects in Petsc are defined on a communicator; can only interact if on the same communicator

Parallelism through MPI

No OpenMP used; user can use shared memory programming

Transparent: same code works sequential and parallel
Object oriented design

Petsc uses objects: vector, matrix, linear solver, nonlinear solver

Overloading:

\[ \text{MATMult}(A,x,y); \quad \text{// } y \leftarrow A \times \]

same for sequential, parallel, dense, sparse
Data hiding

To support this uniform interface, the implementation is hidden:

```
MatSetValue(A,i,j,v,INSERT_VALUES); // A[i,j] <- v
```

There are some direct access routines, but most of the time you don’t need them.
Getting started
Program header, C

```c
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"

int main(int argc, char **argv)
```

- Petsc include file: one at the top of the file
- Declare the name of each routine: helps with traceback
- Can also include `petscmat.h` if no higher functionality needed.
Program header, F

program init

implicit none
#include "finclude/petsc.h"

Include file once per subprogram

#include "finclude/petscsys.h"
#include "finclude/petscvec.h"
#include "finclude/petscmat.h"
Variable declarations, C

KSP solver;
Mat A;
Vec x, y;
PetscInt n = 20;
PetscScalar v;
PetscReal nrm;

Note Scalar vs Real
Variable declarations, F

KSP :: solver
Mat :: A
Vec :: x, y
PetscInt :: j(3)
PetscScalar :: mv
PetscReal :: nrm

Much like in C; uses cpp
Routine start/end, C

PetscFunctionBegin;
// all statements
PetscFunctionReturn(0);

only in C, not in Fortran
Library setup, C

```c
ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);
// all the petsc work
ierr = PetscFinalize(); CHKERRQ(ierr);
```

Can replace MPI_Init

General: Every routine has an error return. Catch that value!
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRQ(ierr)

// all the petsc work

call PetscFinalize(ierr)
CHKERRQ(ierr)

Error code is now final parameter. This holds for every PETSc routine
Note to self

PetscInitialize
    (&argc,&args,0,"Usage: prog -o1 v1 -o2 v2\n");

run as

./program -help

This displays the usage note, plus all available petsc options.

Not available in Fortran
Let’s do something useful, C

```c
ierr = PetscOptionsGetInt
     (PETSC_NULL,"-n",&n,PETSC_NULL); CHKERRQ(ierr);
ierr = PetscPrintf
     (comm,"Input parameter: %d\n",n); CHKERRQ(ierr);
```

Read commandline argument, print out from processor zero
Let’s do something useful, F

```fortran
character*80    msg
call PetscOptionsGetInt(PETSC_NULL_CHARACTER,
   >       "-n",n,PETSC_NULL_CHARACTER,ierr)
CHKERRQ(ierr)
write(msg,10) n
10 format("Input parameter:",i5)
call PetscPrintf(PETSC_COMM_WORLD,msg,ierr)
CHKERRQ(ierr)
```

Note the PETSC_NULL_CHARACTER, note that PetscPrintf has only one string argument
Lab 1

init.c/init.F
Vec `datatype: vectors`
Create calls

Everything in PETSc is an object, with create and destroy calls:

VecCreate(MPI_Comm comm, Vec *v);
VecDestroy(Vec v);

/* C */
Vec V;
ierr = VecCreate(MPI_COMM_SELF, &V); CHKERRQ(ierr);
ierr = VecDestroy(&V); CHKERRQ(ierr);

! Fortran
Vec V
call VecCreate(MPI_COMM_SELF, V, e)
CHKERRQ(ierr)
call VecDestroy(V, e)
CHKERRQ(ierr);

Note: in Fortran there are no “star” arguments
More about vectors

A vector is a vector of PetscScalars: there are no vectors of integers (see the IS datatype later)

The vector object is not completely created in one call:

VecSetSizes(Vec v, int m, int M);

Other ways of creating: make more vectors like this one:

VecDuplicate(Vec v, Vec *w);
Parallel layout

Local or global size in

VecSetSizes(Vec v, int m, int M);

Global size can be specified as PETSC_DECIDE.

VecSetSizes(V,2,5)
VecSetSizes(V,3,5)
VecSetSizes(V,2,PETSC_DECIDE)
VecSetSizes(V,3,PETSC_DECIDE)
Parallel layout up to PETSc

VecSetSizes(Vec v, int m, int M);

Local size can be specified as PETSC_DECIDE.

VecSetSizes(V,PETSC_DECIDE,8)

VecSetSizes(V,PETSC_DECIDE,8)

VecSetSizes(V,PETSC_DECIDE,8)
Query parallel layout

Query vector layout:

\[
\text{VecGetOwnershipRange}(\text{Vec } x, \text{PetscInt } *\text{low}, \text{PetscInt } *\text{high})
\]

\[
\text{VecGetOwnershipRange}(x, \text{low}, \text{high}, \text{ierr}) ! F
\]

On 2nd processor:

```
0 1
2 3
4 5
6 7
```

\[
\text{On 2nd processor:}
\]

\[
\text{low} = 3
\]

\[
\text{high} = 6
\]

Query general layout:

\[
\text{PetscSplitOwnership}(	ext{MPI_Comm } \text{comm}, \text{PetscInt } *\text{n}, \text{PetscInt } *\text{N})
\]

\[
\text{PetscSplitOwnership}(\text{comm}, \text{n}, \text{N}, \text{ierr}) ! F
\]

\[
\text{(get local/global given the other)}
\]
Setting values

Set vector to constant value:

\[
\text{VecSet}(\text{Vec } x, \text{PetscScalar value});
\]

Set individual elements (global indexing!):

\[
\text{VecSetValues}(\text{Vec } x, \text{int } n, \text{int } *\text{indices}, \text{PetscScalar } *\text{values}, \\
\quad \text{INSERT\_VALUES}); /* or ADD\_VALUES */
\]

\[
i = 1; v = 3.14;
\]
\[
\text{VecSetValues}(x, 1, &i, &v, \text{INSERT\_VALUES});
\]
\[
\text{ii}[0] = 1; \text{ii}[1] = 2; \text{vv}[0] = 2.7; \text{vv}[1] = 3.1;
\]
\[
\text{VecSetValues}(x, 2, \text{ii}, \text{vv}, \text{INSERT\_VALUES});
\]

\[
call \text{VecSetValues}(x, 1, i, v, \text{INSERT\_VALUES}, \text{ierr}, e)
\]
\[
i(1) = 1; i(2) = 2; v(1) = 2.7; v(2) = 3.1
\]
\[
call \text{VecSetValues}(x, 2, \text{ii}, \text{vv}, \text{INSERT\_VALUES}, \text{ierr}, e)
\]
Setting values

No restrictions on parallelism; after setting, move values to appropriate processor:

VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
Getting values (C)

Setting values is done without user access to the stored data. Getting values is often not necessary: many operations provided.

what if you do want access to the data?

- Create vector from user provided array:

  VecCreateSeqWithArray(MPI_Comm comm, PetscInt n, const PetscScalar array[], Vec *V)
  VecCreateMPIWithArray(MPI_Comm comm, PetscInt n, PetscInt N, const PetscScalar array[], Vec *vv)

- Get the internal array (local only; see VecScatter for more general mechanism):

  VecGetArray(Vec x, PetscScalar *a[])
  /* do something with the array */
  VecRestoreArray(Vec x, PetscScalar *a[])
Getting values example

```c
int localsize, first, i;
PetscScalar *a;
VecGetLocalSize(x, &localsize);
VecGetOwnershipRange(x, &first, PETSC_NULL);
VecGetArray(x, &a);
for (i = 0; i < localsize; i++)
    printf("Vector element %d : %e\n", first + i, a[i]);
VecRestoreArray(x, &a);
```
Array handling in F90

PetscScalar, pointer :: xx_v(:)

....
call VecGetArrayF90(x, xx_v, ierr)
a = xx_v(3)
call VecRestoreArrayF90(x, xx_v, ierr)

More separate F90 versions for ‘Get’ routines
(there are some ugly hacks for F77)
Basic operations

VecAXPY(Vec y, PetscScalar a, Vec x); /* y <- y + a x */
VecAYPX(Vec y, PetscScalar a, Vec x); /* y <- a y + x */
VecScale(Vec x, PetscScalar a);
VecDot(Vec x, Vec y, PetscScalar *r); /* several variants */
VecMDot(Vec x, int n, Vec y[], PetscScalar *r);
VecNorm(Vec x, NormType type, double *r);
VecSum(Vec x, PetscScalar *r);
VecCopy(Vec x, Vec y);
VecSwap(Vec x, Vec y);
VecPointwiseMult(Vec w, Vec x, Vec y);
VecPointwiseDivide(Vec w, Vec x, Vec y);
VecMAXPY(Vec y, int n, PetscScalar *a, Vec x[]);
VecMax(Vec x, int *idx, double *r);
VecMin(Vec x, int *idx, double *r);
VecAbs(Vec x);
VecReciprocal(Vec x);
VecShift(Vec x, PetscScalar s);
Mat Datatype: matrix
Matrix creation

The usual create/destroy calls:

```c
MatCreate(MPI_Comm comm, Mat *A)
MatDestroy(Mat A)
```

Several more aspects to creation:

```c
MatSetType(A, MATSEQAIJ) /* or MATMPIAIJ or MATAIJ */
MatSetSizes(Mat A, int m, int n, int M, int N)
MatSeqAIJSetPreallocation /* more about this later*/
   (Mat B, PetscInt nz, const PetscInt nnz[])
```

Local or global size can be PETSC_DECIDE (as in the vector case)
Matrix creation all in one

MatCreateSeqAIJ(MPI_Comm comm, PetscInt m, PetscInt n, PetscInt nz, const PetscInt nnz[], Mat *A)
MatCreateMPIAIJ(MPI_Comm comm, PetscInt m, PetscInt n, PetscInt M, PetscInt N, PetscInt d_nz, const PetscInt d_nnz[], PetscInt o_nz, const PetscInt o_nnz[], Mat *A)
If you already have a CRS matrix

PetscErrorCode MatCreateSeqAIJWithArrays

(MPI_Comm comm,PetscInt m,PetscInt n,
PetscInt* i,PetscInt* j,PetscScalar *a,Mat *mat)

(Also from triplets)

Do not use this unless you interface to a legacy code. And even then...
Matrix Preallocation

- PETSc matrix creation is very flexible:
- No preset sparsity pattern
- any processor can set any element
  ⇒ potential for lots of malloc calls
- malloc is very expensive: (run your code with -memory_info, -malloc_log)
- tell PETSc the matrix’ sparsity structure
  (do construction loop twice: once counting, once making)
Sequential matrix structure

MatSeqAIJSetPreallocation
  (Mat B,PetscInt nz,const PetscInt nnz[])/* or */
MatCreateSeqAIJ(comm,int m,int n,
    int nz,int *nnz,Mat *A);

- nz number of nonzeros per row
  (or slight overestimate)
- nnz array of row lengths (or overestimate)
- considerable savings over dynamic allocation!

In Fortran use PETSC_NULL_INTEGER if not specifying nnz array
Parallel matrix structure

Off–diagonal block has off–processor connections

Diagonal block has on–processor connections
(why does it do this?)

• \( y \leftarrow Ax_A + Bx_b \)
• \( x_B \) needs to be communicated; \( Ax_A \) can be computed in the meantime

**Algorithm**
- Initiate asynchronous sends/receives for \( x_b \)
- compute \( Ax_A \)
- make sure \( x_b \) is in
- compute \( Bx_B \)

• so by splitting matrix storage into \( A, B \) part, code for the sequential case can be reused.

• This is one of the few places where PETSc’s design is visible to the user.
Parallel matrix structure description

- \( d_{nz} \): number of nonzeros per row in diagonal part
- \( o_{nz} \): number of nonzeros per row in off-diagonal part
- \( d_{nnz} \): array of numbers of nonzeros per row in diagonal part
- \( o_{nnz} \): array of numbers of nonzeros per row in off-diagonal part

```c
MatCreateMPIAIJ(MPI Comm comm, int m, int n, int M, int N,
                int d_nz, int *d_nnz, int o_nz, int *o_nnz, Mat *A);
```

In Fortran use `PETSC_NULL_INTEGER` if not specifying arrays
Querying parallel structure

Matrix partitioned by block rows:

MatGetSize(Mat mat, PetscInt *M, PetscInt *N);
MatGetLocalSize(Mat mat, PetscInt *m, PetscInt *n);
MatGetOwnershipRange(Mat A, int *first row, int *last row);

If the matrix is square, $m, n$ will probably be equal, even though
distribution by block rows
Setting values

Set one value:

MatSetValue(Mat v,
  PetscInt i,PetscInt j,PetscScalar va,InsertMode mode)

where insert mode is INSERT_VALUES, ADD_VALUES

Set block of values:

MatSetValues(Mat A,int m,const int idxm[],
  int n,const int idxn[],const PetscScalar values[],
  InsertMode mode)

(v is row-oriented)
Special case of the general case:

MatSetValues(A, 1, &i, 1, &j, &v, INSERT_VALUES); // C
MatSetValues(A, 1, i, 1, j, v, INSERT_VALUES, e); ! F
Assembling the matrix

Setting is independent of parallelism

MatAssemblyBegin(Mat A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A, MAT_FINAL_ASSEMBLY);

Cannot mix inserting/adding values: need to do assembly in between
Getting values (C)

- Values are often not needed: many matrix operations supported
- Matrix elements can only be obtained locally.

```c
PetscErrorCode MatGetRow(Mat mat,
   PetscInt row,PetscInt *ncols,const PetscInt *cols[],
   const PetscScalar *vals[])
PetscErrorCode MatRestoreRow(/* same parameters */

Note: for inspection only; possibly expensive.
```
Getting values (F)

MatGetRow(A,row,ncols,cols,vals,ierr)

where cols(maxcols), vals(maxcols) are long enough arrays (allocated by the user)
Other matrix types

MATBAIJ : blocked matrices (dof per node)

(see PETSC_DIR/include/petscmat.h)

Dense:

MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,
     PetscScalar *data,Mat *A);
MatCreateMPIDense(MPI Comm comm,int m,int n,int M,int N,
     PetscScalar *data,Mat *A)

Data argument optional
Matrix operations

Main operations are matrix-vector:

MatMult(Mat A, Vec in, Vec out);
MatMultAdd
MatMultTranspose
MatMultTransposeAdd

Simple operations on matrices:

MatNorm

MatScale
MatDiagonalScale
Matrix viewers

MatView(A,0);

row 0: (0, 1) (2, 0.333333) (3, 0.25) (4, 0.2)
row 1: (0, 0.5) (1, 0.333333) (2, 0.25) (3, 0.2)
....

- Shorthand for MatView(A,PETSC_VIEWER_STDOUT_WORLD);
  or even MatView(A,0) (Fortran: PETSC_NULL_INTEGER)
- also invoked by -mat_view
- Sparse: only allocated positions listed
- other viewers: for instance -mat_view_draw (X terminal)
General viewers

Any PETSc object can be viewed
binary dump is a view:

PetscViewer fd;
PetscViewerBinaryOpen
   (PETSC_COMM_WORLD,"matdata",FILE_MODE_WRITE,&fd);
MatView(A,fd);
PetscViewerDestroy(fd);
Shell matrices

What if the matrix is a user-supplied operator, and not stored?

MatSetType(A,MATSHELL); /* or */
MatCreateShell(MPI Comm comm,
               int m,int n,int M,int N,void *ctx,Mat *mat);

PetscErrorCode UserMult(Mat mat,Vec x,Vec y);

MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,
                      (void(*)(void)) PetscErrorCode (*UserMult)(Mat,Vec,Vec));

Inside iterative solvers, PETSc calls MatMult(A,x,y):
no difference between stored matrices and shell matrices
Shell matrix context

Shell matrices need custom data

MatShellSetContext(Mat mat, void *ctx);
MatShellGetContext(Mat mat, void **ctx);

(This does not work in Fortran: use Common or Module)

User program sets context, matmulf routine accesses it
Shell matrix example

...
MatSetType(A,MATSHELL);
MatShellSetOperation(A,MATOP_MULT,(void*)&mymatmult);
MatShellSetContext(A,(void*)&mystruct);
...

PetscErrorCode mymatmult(Mat mat,Vec in,Vec out)
{
    PetscFunctionBegin;
    MatShellGetContext(mat,(void**)&mystruct);
    /* compute out from in, using mystruct */
    PetscFunctionReturn(0);
}
Submatrices

Extract one parallel submatrix:

MatGetSubMatrix(Mat mat,
    IS isrow,IS iscol,PetscInt csize,MatReuse cll,
    Mat *newmat)

Extract multiple single-processor matrices:

MatGetSubMatrices(Mat mat,
    PetscInt n,const IS irow[],const IS icol[],MatReuse scall,
    Mat *submat[])

Collective call, but different index sets per processor
Load balancing

MatPartitioningCreate
(MPI Comm comm, MatPartitioning *part);

Various packages for creating better partitioning: Chaco, ParMETIS
KSP & PC: Iterative solvers
What are iterative solvers?

Solving a linear system $Ax = b$ with Gaussian elimination can take lots of time/memory.

Alternative: iterative solvers use successive approximations of the solution:

- Convergence not always guaranteed
- Possibly much faster / less memory
- Basic operation: $y \leftarrow Ax$ executed once per iteration
- Also needed: preconditioner $B \approx A^{-1}$
Basic concepts

- All linear solvers in PETSc are iterative (see below)
- Object oriented: solvers only need matrix action, so can handle shell matrices
- Preconditioners
- Fargoing control through commandline options
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests
Iterative solver basics

KSPCreate(comm,&solver); KSPDestroy(solver);

// general:
KSPSetOperators(solver,A,B,DIFFERENT_NONZERO_PATTERN);
// common:
KSPSetOperators(solver,A,A,DIFFERENT_NONZERO_PATTERN);
// also SAME_NONZERO_PATTERNS and SAME_PRECONDITIONER

KSPSolve(solver,rhs,sol);
/* optional */ KSPSetup(solver);
Solver type

KSPSetType(solver, KSPGMRES);

KSP can be controlled from the commandline:

KSPSetFromOptions(solver);
/* right before KSPSolve or KSPSetUp */

then options -ksp.... are parsed.

- type: -ksp_type gmres -ksp_gmres_restart 20
- -ksp_view
Convergence

Iterative solvers can fail

- Solve call itself gives no feedback: solution may be completely wrong
- `KSPGetConvergedReason(solver,&reason)`: positive is convergence, negative divergence
  (`${PETSC_DIR}/include/petscksp.h` for list)
- `KSPGetIterationNumber(solver,&nits)`: after how many iterations did the method stop?
KSPSolve(solver,B,X);
KSPGetConvergedReason(solver,&reason);
if (reason<0) {
    printf("Divergence.\n");
} else {
    KSPGetIterationNumber(solver,&its);
    printf("Convergence in %d iterations.\n",(int)its);
}
Monitors and convergence tests

KSPSetTolerances(solver, rtol, atol, dtol, maxit);

Monitors can also be set in code, but easier:

• -ksp_monitor
• -ksp_monitor_true_residual
Monitors and convergence tests (adv)

KSPMonitorSet(KSP ksp,
        PetscErrorCode (*monitor)
        (KSP,PetscInt,PetscReal,void*),
    void *mctx,
    PetscErrorCode (*monitordestroy)(void*));

KSPSetConvergenceTest(KSP ksp,
        PetscErrorCode (*converge)
        (KSP,PetscInt,PetscReal,KSPConvergedReason*,void*),
    void *cctx,
    PetscErrorCode (*destroy)(void*))
Example of convergence tests

PetscErrorCode resconverge
(KSP solver,PetscInt it,PetscReal res,
KSPConvergedReason *reason,void *ctx)
{
    MPI_Comm comm; Mat A; Vec X,R; PetscErrorCode ierr;
PetscFunctionBegin;
    KSPGetOperators(solver,&A,PETSC_NULL,PETSC_NULL);
PetscObjectGetComm((PetscObject)A,&comm);
    KSPBuildResidual(solver,PETSC_NULL,PETSC_NULL,&R);
    KSPBuildSolution(solver,PETSC_NULL,&X);
    /* stuff */
    if (sometest) *reason = 15;
    else *reason = KSP_CONVERGED_ITERATING;
PetscFunctionReturn(0);
Advanced options

Many options for the (mathematically) sophisticated user
some specific to one method

KSPSetInitialGuessNonzero
KSPGMRESSetRestart
KSPSetPreconditionerSide
KSPSetNormType
Null spaces

MatNullSpace sp;
MatNullSpaceCreate /* constant vector */
   (PETSC_COMM_WORLD,PETSC_TRUE,0,PETSC_NULL,&sp);
MatNullSpaceCreate /* general vectors */
   (PETSC_COMM_WORLD,PETSC_FALSE,5,vecs,&sp);
KSPSetNullSpace(ksp,sp);

The solver will now properly remove the null space at each iteration.
PC basics

- PC usually created as part of KSP: separate create and destroy calls exist, but are (almost) never needed

  KSP solver; PC precon;
  KSPCreate(comm,&solver);
  KSPGetPC(solver,&precon);
  PCSetType(precon,PCJACOBI);

- PCJACOBI, PCILU (only sequential), PCASM, PCBJACOBI, PCMG, et cetera

- Controllable through commandline options:
  -pc_type ilu -pc_factor_levels 3
Preconditioner reuse

In context of nonlinear solvers, the preconditioner can sometimes be reused:

- If the jacobian doesn’t change much, reuse the preconditioner completely
- If the preconditioner is recomputed, the sparsity pattern probably stays the same

\[
\text{KSPSetOperators}(\text{solver}, A, B, \text{structureflag})
\]

- B is basis for preconditioner, need not be A
- structureflag can be SAME_PRECONDITIONER, SAME_NONZERO_PATTERN, DIFFERENT_NONZERO_PATTERN: avoid recomputation of preconditioner (sparsity pattern) if possible
Factorization preconditioners

Exact factorization: $A = LU$

Inexact factorization: $A \approx M = LU$ where $L, U$ obtained by throwing away ‘fill-in’ during the factorization process.

Exact:

$\forall i, j: a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$

Inexact:

$\forall i, j: \text{if } a_{ij} \neq 0 \ a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$

Application of the preconditioner (that is, solve $Mx = y$) approx same cost as matrix-vector product $y \leftarrow Ax$

Factorization preconditioners are sequential
ILU

PCICC: symmetric, PCILU: nonsymmetric

PCFactorSetLevels(PC pc,int levels);
   -pc_factor_levels <levels>

et cetera
Prevent indefinite preconditioners:

PCFactorSetShiftPd(PC pc,MatFactorShiftType type);

value MAT_SHIFT_POSITIVE_DEFINITE et cetera
Block Jacobi and Additive Schwarz

- Factorization preconditioners are sequential;
- can be made parallel by use in Block Jacobi or Additive Schwarz methods
- each processor has its own block(s) to work with
Block Jacobi and Additive Schwarz, theory

- Both methods parallel
- Jacobi fully parallel
  - Schwarz local communication between neighbours
- Both require sequential local solver
- Jacobi limited reduction in iterations
  - Schwarz can be optimal
Block Jacobi and Additive Schwarz, coding

KSP *ksps; int nlocal, firstlocal; PC pc;
PCBJacobiGetSubKSP(pc,&nlocal,&firstlocal,&ksps);
for (i=0; i<nlocal; i++) {
    KSPSetType( ksps[i], KSPGMRES );
    KSPGetPC( ksps[i], &pc );
    PCSetType( pc, PCILU );
}

Much shorter: commandline options -sub_ksp_type and -sub_pc_type (subksp is PREONLY by default)

PCASMSSetOverlap(PC pc, int overlap);
Matrix-free solvers

Shell matrix requires shell preconditioner (or use different operators in `KSPSetOperators`):

```c
PCSetType(pc,PCSHELL);
PCShellSetContext(PC pc,void *ctx);
PCShellGetContext(PC pc,void **ctx);
PCShellSetApply(PC pc,
    PetscErrorCode (*apply)(void*,Vec,Vec)));
PCShellSetSetUp(PC pc,
    PetscErrorCode (*setup)(void*))
```

similar idea to shell matrices
Direct methods

• Iterative method with direct solver as preconditioner would converge in one step

• Direct methods in PETSc implemented as special iterative method: KSPPREONLY only apply preconditioner

• All direct methods are preconditioner type PCLU:

```
myprog -pc_type lu -ksp_type preonly \
    -pc_factor_mat_solver_package mumps
```
Other external PCs

If installed, other parallel preconditioner are available:

- From Hypre: PCHYPRE with subtypes boomeramg, parasails, euclid, pilut:
  PCHYPRESetType(pc,parasails) or -pc_hypre_type parasails
- PCSPAI for Sparse Approximate Inverse
- PCPROMETHEUS
- External packages’ existence can be tested:
  ```
  % grep hypre $PETSC_DIR/$PETSC_ARCH/include/petscconf.h
  ifndef PETSC_HAVE_HYPRE
  define PETSC_HAVE_HYPRE 1
  ifndef PETSC_HAVE_LIBHYPRE
  define PETSC_HAVE_LIBHYPRE 1
  ```
Grid manipulation
Regular grid: DMDA

DMDAs are for storing vector field, not matrix.

Support for different stencil types:

- Star stencil
- Box stencil
Ghost regions around processors

A DMDA defines a global vector, which contains the elements of the grid, and a local vector for each processor which has space for "ghost points".
DMDA construction

DMDACreate2d(comm, bndx, bndy, type, M, N, m, n, 
dof, s, lm[], ln[], DMDA *da)

bndx, bndy boundary behaviour: none/ghost/periodic

type: Specifies stencil
DMDA_STENCIL_BOX or DMDA_STENCIL_STAR

M/N: Number of grid points in x/y-direction
m/n: Number of processes in x/y-direction
dof: Degrees of freedom per node
s: The stencil width (for instance, 1 for 2D five-point stencil)

lm/n: array of local sizes (optional; Use PETSC_NULL for the default)
**Associated vectors**

DMCreateGlobalVector(DMDA da, Vec *g);
DMCreateLocalVector(DMDA da, Vec *l);

global -> local
DMGlobalToLocalBegin/End
   (DMDA da, Vec g, InsertMode iora, Vec l);

local -> global
DMLocalToGlobalBegin/End
   (DMDA da, Vec l, InsertMode mode, Vec g);

local -> global -> local :
DMLocalToLocalBegin/End
   (DMDA da, Vec l1, InsertMode iora, Vec l2);
Irregular grid: IS & VecScatter

Index Set is a set of indices (more later about their uses)

```c
ISCreateGeneral(comm,n,indices,&is);
    /* indices can now be freed */
ISCreateGeneralWithArray(comm,n,indices,&is);
    /* indices are stored */
ISCreateStride (comm,n,first,step,&is);
ISCreateBlock    (comm,bs,n,indices,&is);

ISDestroy(is);
```

Various manipulations: ISSum, ISDifference, ISInvertPermutations et cetera.
Retrieving information

ISGetIndices / ISRestoreIndices ISGetSize
VecScatter

VecScatterCreate(Vec, IS, Vec, IS, VecScatter*) & Destroy
VecScatterBegin
  (VecScatter, Vec, Vec, InsertMode mode, ScatterMode direction)
VecScatterEnd
  (VecScatter, Vec, Vec, InsertMode mode, ScatterMode direction)

Example: collect distributed boundary onto a single processor:
SNES: Nonlinear solvers
Nonlinear problems

Basic equation

\[ f(u) = 0 \]

where \( u \) can be big, for instance nonlinear PDE.

Typical solution method:

\[ u_{n+1} = u_n - J(u_n)^{-1}f(u_n) \]

Newton iteration.

Needed: function and Jacobian.
Basic SNES usage

User supplies function and Jacobian:

```c
SNES snes;
SNESCreate(PETSC_COMM_WORLD,&snes);
SNESSetType(snes,type);
SNESSetFromOptions(snes);
SNESDestroy(SNES snes);
```

where type:

- SNESLS Newton with line search
- SNESTR Newton with trust region
- several specialized ones
SNES specification

VecCreate(PETSC_COMM_WORLD,&r)
SNESSetFunction(snes,r,FormFunction,*ctx)

MatCreate(PETSC_COMM_WORLD,&J)
SNESSetJacobian(snes,J,J,FormJacobian,*ctx)

SNESolve(snes,PETSC_NULL,x)
SNESGetIterationNumber(snes,&its)
Target function

PetscErrorCode FormFunction
  (SNES snes, Vec x, Vec f, void *dummy)
{
  VecGetArray(x, &xx); VecGetArray(f, &ff);

  ff[0] = PetscSinScalar(3.0*xx[0]) + xx[0];
  ff[1] = xx[1];

  VecRestoreArray(x, &xx); VecRestoreArray(f, &ff);
  return 0;
}
Jacobian

PetscErrorCode FormJacobian
(SNES snes, Vec x, Mat *jac, Mat *prec, MatStructure *flag, void *)
{
    PetscScalar A[];
    VecGetArray(x, &xx)
    A[0] = ... ; /* et cetera */
    MatSetValues(*jac, ..., INSERT_VALUES)
    MatSetValues(*prec, ..., INSERT_VALUES)
    *flag = SAME_NONZERO_PATTERN;
    VecRestoreArray(x, &xx)
    MatAssemblyBegin(*prec, MAT_FINAL_ASSEMBLY)
    MatAssemblyEnd(*prec, MAT_FINAL_ASSEMBLY)
    MatAssemblyBegin(*jac, MAT_FINAL_ASSEMBLY)
    MatAssemblyEnd(*jac, MAT_FINAL_ASSEMBLY)
    return 0;
}
Further possibilities

SNESSetTolerances(SNES snes, double atol, double rtol, double stol, int its, int fcts);

convergence test and monitoring, specific options for line search and trust region

adaptive convergence: -snes_ksp_ew_conv (Eisenstat Walker)
Solve customization

SNESSetType(snes, SNESTR); /* newton with trust region */
SNESGetKSP(snes, &ksp)
KSPGetPC(ksp, &pc)
PCSetType(pc, PCNONE)
KSPSetTolerances(ksp, 1.e-4, PETSC_DEFAULT, PETSC_DEFAULT, 20)
sophisticated stuff

- Jacobian through finite difference:
  
  SNESDefaultComputeJacobian or -snes_fd

- Matrix-free operation
TS: Time stepping
Profiling, debugging
Basic profiling

- log_summary flop counts and timings of all PETSc events
- info all sorts of information, in particular
  
  ```
  %% mpiexec yourprogram -info | grep malloc
  [0] MatAssemblyEnd_SeqAIJ():
  Number of mallocs during MatSetValues() is 0
  ```
- log_trace start and end of all events: good for hanging code
Log summary: overall

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Max/Min</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec):</td>
<td>5.493e-01</td>
<td>1.00006</td>
<td>5.493e-01</td>
<td></td>
</tr>
<tr>
<td>Objects:</td>
<td>2.900e+01</td>
<td>1.00000</td>
<td>2.900e+01</td>
<td></td>
</tr>
<tr>
<td>Flops:</td>
<td>1.373e+07</td>
<td>1.00000</td>
<td>1.373e+07</td>
<td>2.746e+07</td>
</tr>
<tr>
<td>Flops/sec:</td>
<td>2.499e+07</td>
<td>1.00006</td>
<td>2.499e+07</td>
<td>4.998e+07</td>
</tr>
<tr>
<td>Memory:</td>
<td>1.936e+06</td>
<td>1.00000</td>
<td>2.746e+07</td>
<td>3.871e+06</td>
</tr>
<tr>
<td>MPI Messages:</td>
<td>1.040e+02</td>
<td>1.00000</td>
<td>1.040e+02</td>
<td>2.080e+02</td>
</tr>
<tr>
<td>MPI Msg Lengths:</td>
<td>4.772e+05</td>
<td>1.00000</td>
<td>4.588e+03</td>
<td>9.544e+05</td>
</tr>
<tr>
<td>MPI Reductions:</td>
<td>1.450e+02</td>
<td>1.00000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Log summary: details

<table>
<thead>
<tr>
<th>Function</th>
<th>Max Ratio</th>
<th>Max Ratio</th>
<th>Avg len</th>
<th>%T</th>
<th>%F</th>
<th>%M</th>
<th>%L</th>
<th>%R</th>
<th>Mflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatMult</td>
<td>100</td>
<td>1.0</td>
<td>3.4934e-02</td>
<td>1.0</td>
<td>1.28e+08</td>
<td>1.0</td>
<td>8.0e+02</td>
<td>6 32 96 17 0</td>
<td>6 32 96 17 0</td>
</tr>
<tr>
<td>MatSolve</td>
<td>101</td>
<td>1.0</td>
<td>2.9381e-02</td>
<td>1.0</td>
<td>1.53e+08</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>5 33 0 0 0</td>
<td>5 33 0 0 0</td>
</tr>
<tr>
<td>MatLUFactorNum</td>
<td>1</td>
<td>1.0</td>
<td>2.0621e-03</td>
<td>1.0</td>
<td>2.18e+07</td>
<td>1.0</td>
<td>0.0e+00</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>MatAssemblyBegin</td>
<td>1</td>
<td>1.0</td>
<td>2.8350e-03</td>
<td>1.1</td>
<td>0.00e+00</td>
<td>0.0</td>
<td>1.3e+05</td>
<td>0 0 3 83 1</td>
<td>0 0 3 83 1</td>
</tr>
<tr>
<td>MatAssemblyEnd</td>
<td>1</td>
<td>1.0</td>
<td>8.8258e-03</td>
<td>1.0</td>
<td>0.00e+00</td>
<td>0.0</td>
<td>4.0e+02</td>
<td>2 0 1 0 3</td>
<td>2 0 1 0 3</td>
</tr>
<tr>
<td>VecDot</td>
<td>101</td>
<td>1.0</td>
<td>8.3244e-03</td>
<td>1.2</td>
<td>1.43e+08</td>
<td>1.2</td>
<td>0.0e+00</td>
<td>1 7 0 0 35</td>
<td>1 7 0 0 35</td>
</tr>
<tr>
<td>KSPSetup</td>
<td>2</td>
<td>1.0</td>
<td>1.9123e-02</td>
<td>1.0</td>
<td>0.00e+00</td>
<td>0.0</td>
<td>0.0e+00</td>
<td>3 0 0 0 2</td>
<td>3 0 0 0 2</td>
</tr>
<tr>
<td>KSPSolve</td>
<td>1</td>
<td>1.0</td>
<td>1.4158e-01</td>
<td>1.0</td>
<td>9.70e+07</td>
<td>1.0</td>
<td>8.0e+02</td>
<td>26100 96 17 92</td>
<td>26100 96 17 92</td>
</tr>
</tbody>
</table>
User events

#include "petsclog.h"
int USER EVENT;
PetscLogEventRegister(&USER EVENT,"User event name",0);
PetscLogEventBegin(USER EVENT,0,0,0,0,0);
/* application code segment to monitor */
PetscLogFlops(number of flops for this code segment);
PetscLogEventEnd(USER EVENT,0,0,0,0,0);
Program stages

PetscLogStagePush(int stage); /* 0 <= stage <= 9 */
PetscLogStagePop();
PetscLogStageRegister(int stage, char *name)
Debugging

- Use of CHKERRQ and SETERRQ for catching and generating error
- Use of PetscMalloc and PetscFree to catch memory problems;
  CHKMEMQ for instantaneous memory test (debug mode only)