Introduction to PETSc

Victor Eijkhout

Outline

- Introduction
- An example program
- `Vec` datatype: vectors
- `Mat` Datatype: matrix
- `KSP & PC`: Iterative solvers
- Making and running PETSc programs
- Profiling, debugging
- Remaining topics

Introduction

What is PETSc? Why should you use PETSc?

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 is in PETSc?

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

An example program

Include files

Multiple include files; in C only the highest level

```
#include "petscksp.h"
```

In Fortran sequence of include files *in the subprogram*

```
#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
#include "include/finclude/petscpc.h"
```

Program/subprogram heading

CPP definition of (sub)program name:
will be used in traceback

```
#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  PetscFunctionBegin;
  ...
  PetscFunctionReturn(0);
}
```

Use this for every subprogram.

Not available in Fortran

Petsc initialize / finalize

One-time initialization, includes MPI if not already initialized:

```
ierr = PetscInitialize(&argc,&args,0,0);
....
ierr = PetscFinalize();
```

Fortran:

```
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
....
call PetscFinalize(ierr)
```

Petsc function return values

Every PETSc function returns an error code, zero is success.

C:

```
ierr = SomePetscFunction(....); CHKERRQ(ierr);
```

Fortran:

```
call SomePetscFunction(...., ierr )
CHKERRQ(ierr);
```

Variable declarations

Everything is an object

```
MPI_Comm comm;
PetscErrorCode ierr; PetscTruth flag;
KSP Solver; Mat A; Vec Rhs,Sol;
PetscScalar one; PetscInt its; PetscReal norm;

PetscErrorCode ierr
PetscTruth flag
KSP Solver
PC Prec
Mat A
PetscInt its
....
```

(note scalar vs real)

Read input parameter

Read user commandline argument (`a.out -n 55`)

```
  ierr = PetscOptionsGetInt
    (PETSC_NULL,"-n",&dom_size,&flag); CHKERRQ(ierr);
  if (!flag) dom_size = 10;
  matrix_size = dom_size*dom_size;
```

```
  call PetscOptionsGetInt(PETSC_NULL_CHARACTER,
 >     "-n",dom_size,flag,ierr)
  CHKERRQ(ierr)
  if (flag==0) dom_size = 10
  matrix_size = dom_size*dom_size;
```

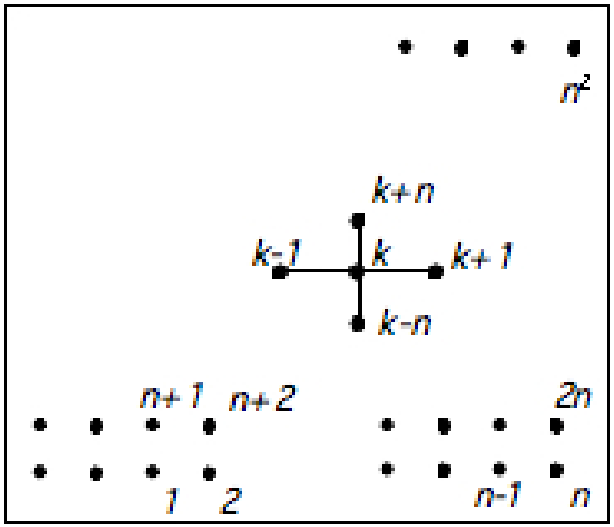
Create matrix

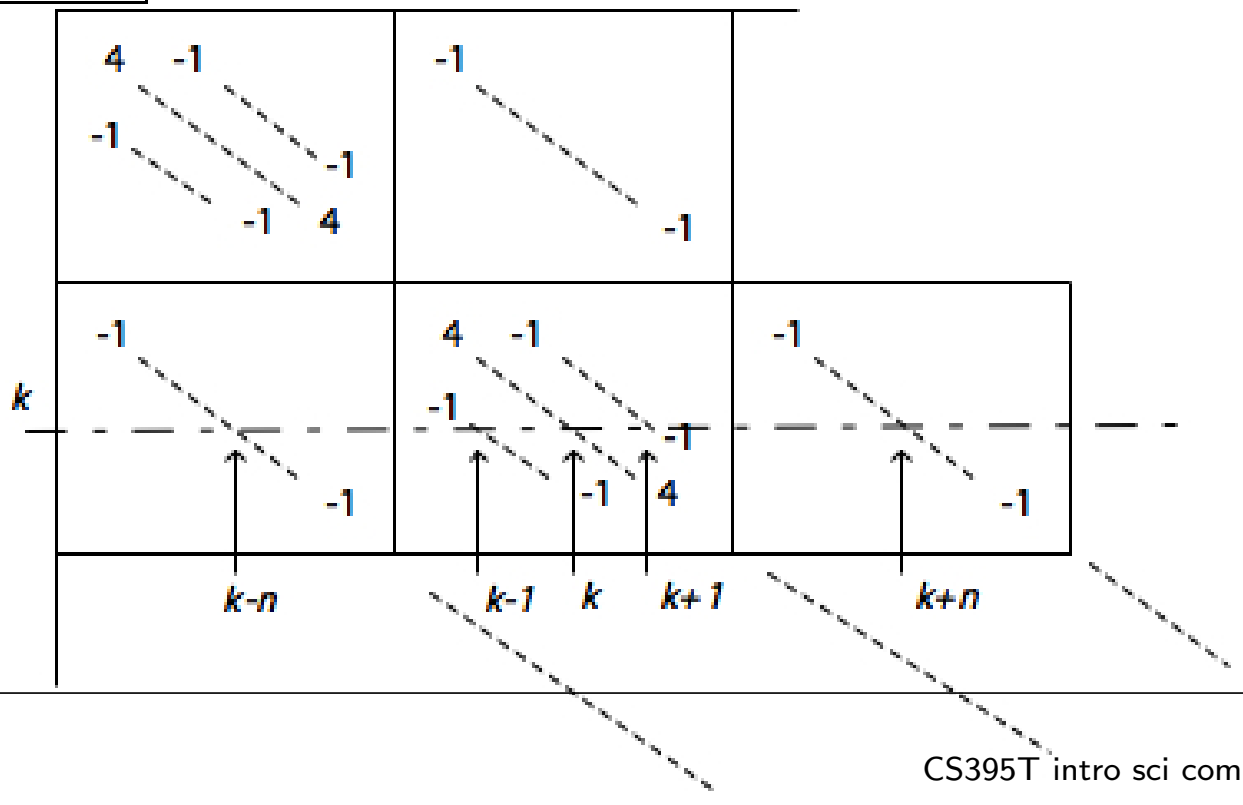
Create distributed matrix on communicator, set local and global size.

```
  ierr = MatCreate(comm,&A); CHKERRQ(ierr);
  ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,
    matrix_size,matrix_size); CHKERRQ(ierr);
  ierr = MatSetType(A,MATMPIAIJ); CHKERRQ(ierr);

  call MatCreate(comm,A,ierr)
  CHKERRQ(ierr)
  call MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,
 >     matrix_size,matrix_size,ierr)
  CHKERRQ(ierr)
  call MatSetType(A,MATMPIAIJ,ierr)
  CHKERRQ(ierr)
```

Five-point Laplacian





Fill in matrix elements (C)

```
ierr = MatGetOwnershipRange(A,&low,&high); CHKERRQ(ierr);
for ( i=0; i<m; i++ ) {
  for ( j=0; j<n; j++ ) {
    I = j + n*i;
    if (I>=low && I<high) {
      J = I-1; v = -1.0;
      ierr = MatSetValues
          (mat,1,&I,1,&J,&v,INSERT_VALUES); CHKERRQ(ierr)
      ....
    }
  }
}
ierr = MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY); CHKERRQ
ierr = MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY); CHKERRQ(ie
```

Fill in matrix elements (F)

```
call MatGetOwnershipRange(A,low,high,ierr)
do i=0,m-1
   do j=0,n-1
      ii = j + n*i
      if (ii>=low .and. ii<high) then
         jj = ii - n
         call MatSetValues(mat,1,ii,1,jj,v,INSERT_VALUES,
         CHKERRQ(ierr)
         ...
call MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY,ierr)
CHKERRQ(ierr)
call MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY,ierr)
CHKERRQ(ierr)
```

Create solver

```
ierr = KSPCreate(comm,&Solver);
ierr = KSPSetOperators(Solver,A,A,0); CHKERRQ(ierr);
ierr = KSPSetType(Solver,KSPCGS); CHKERRQ(ierr);
```

```
call KSPCreate(comm,Solve,ierr)
call KSPSetOperators(Solve,A,A,0,ierr)
CHKERRQ(ierr)
call KSPSetType(Solve,KSPCGS,ierr)
CHKERRQ(ierr)
```

Create preconditioner

```
{
  PC Prec;
  ierr = KSPGetPC(Solver,&Prec); CHKERRQ(ierr);
  ierr = PCSetType(Prec,PCJACOBI); CHKERRQ(ierr);
}
```

```
call KSPGetPC(Solve,Prec,ierr)
CHKERRQ(ierr)
call PCSetType(Prec,PCJACOBI,ierr)
CHKERRQ(ierr)
```

Input and output vectors

```
ierr = VecCreateMPI(comm,PETSC_DECIDE,matrix_size,&Rhs);
ierr = VecDuplicate(Rhs,&Sol); CHKERRQ(ierr);
ierr = VecSet(Rhs,one); CHKERRQ(ierr);
```

```
call VecCreateMPI(comm,PETSC_DECIDE,matrix_size,Rhs,ierr)
CHKERRQ(ierr)
call VecDuplicate(Rhs,Sol,ierr)
CHKERRQ(ierr)
call VecSet(Rhs,one,ierr)
CHKERRQ(ierr)
```

Solve! (C)

```
ierr = KSPSolve(Solver,Rhs,Sol); CHKERRQ(ierr);
{
  PetscInt its; KSPConvergedReason reason;
  Vec Res; PetscReal norm;
  ierr = KSPGetConvergedReason(Solver,&reason);
  if (reason<0) {
    PetscPrintf(comm,"Failure to converge\n");
  } else {
    ierr = KSPGetIterationNumber(Solver,&its); CHKERRQ(ie
    PetscPrintf(comm,"Number of iterations: %d\n",its);
  }
}
```

Solve! (F)

```
  call KSPSolve(Solve,Rhs,Sol,ierr)
  CHKERRQ(ierr)

  call KSPGetConvergedReason(Solve,reason,ierr)
  if (reason<0) then
     call PetscPrintf(comm,"Failure to converge\n")
  else
     call KSPGetIterationNumber(Solve,its,ierr)
     CHKERRQ(ierr)
     write(msg,10) its
10   format('Number of iterations: i4")
     call PetscPrintf(comm,msg,ierr)
  end if
```

Residual calculation

```
ierr = VecDuplicate(Rhs,&Res); CHKERRQ(ierr);
ierr = MatMult(A,Sol,Res); CHKERRQ(ierr);
ierr = VecAXPY(Res,-1,Rhs); CHKERRQ(ierr);
ierr = VecNorm(Res,NORM_2,&norm); CHKERRQ(ierr);
ierr = PetscPrintf(MPI_COMM_WORLD,"residual norm: %e\n",n
```

```
call VecDuplicate(Rhs,Res,ierr)
CHKERRQ(ierr)
call MatMult(A,Sol,Res,ierr)
CHKERRQ(ierr)
call VecAXPY(Res,mone,Rhs,ierr)
CHKERRQ(ierr)
call VecNorm(Res,NORM_2,norm,ierr)
CHKERRQ(ierr)
if (mytid==0) print *,"residual norm:",norm
```

Clean up

```
ierr = VecDestroy(Res); CHKERRQ(ierr);
ierr = KSPDestroy(Solver); CHKERRQ(ierr);
ierr = VecDestroy(Rhs); CHKERRQ(ierr);
ierr = VecDestroy(Sol); CHKERRQ(ierr);

call MatDestroy(A,ierr)
CHKERRQ(ierr)
call KSPDestroy(Solve,ierr)
CHKERRQ(ierr)
call VecDestroy(Rhs,ierr)
CHKERRQ(ierr)
call VecDestroy(Sol,ierr)
CHKERRQ(ierr)
call VecDestroy(Res,ierr)
CHKERRQ(ierr)
```

`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)
CHKERRQ(ierr)
call VecDestroy(V)
CHKERRQ(ierr);
```

Note: in Fortran there are no "star" arguments

More about vectors

A vector is a vectors 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:

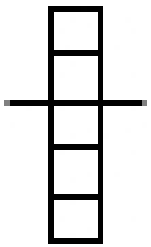
```
VecDuplicate(Vec v,Vec *w);
```

Parallel layout

Local or global size in

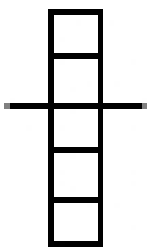
```
VecSetSizes(Vec v, int m, int M);
```

Global size can be specified as `PETSC_DECIDE`.



VecSetSizes(V,2,5)

VecSetSizes(V,3,5)



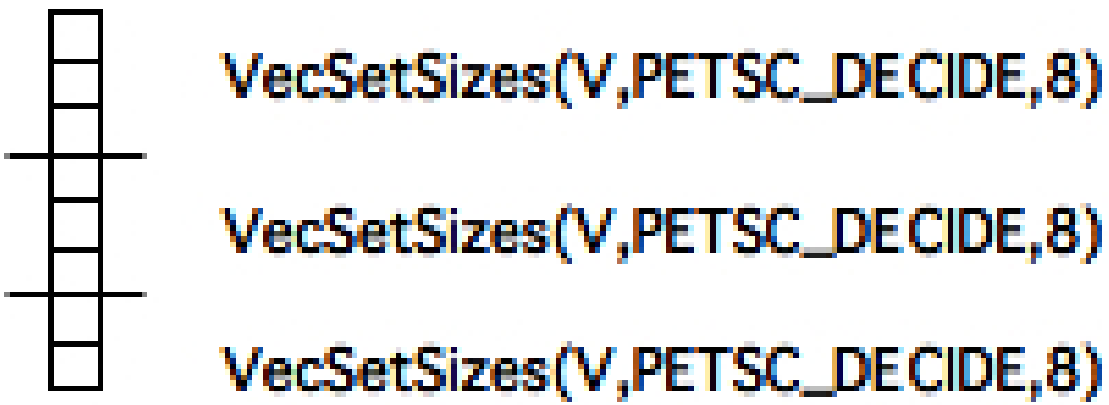
VecSetSizes(V,2,PETSC_DECIDE)

VecSetSizes(V,3,PETSC_DECIDE)

Parallel layout up to PETSc

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

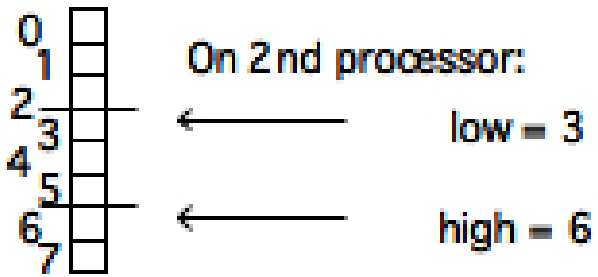
Local size can be specified as `PETSC_DECIDE`.



Query parallel layout

Query vector layout:

```
VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)
```



Query general layout:

```
PetscSplitOwnership(MPI_Comm comm,PetscInt *n,PetscInt *N);
```

(get local/global given the other)

Setting values

Set vector to constant value:

```
VecSet(Vec x,PetscScalar value);
```

Set individual elements (global indexing!):

```
VecSetValues(Vec x,int n,int *indices,PetscScalar *values,
    INSERT VALUES); /* or ADD_VALUES */

i = 1; v = 3.14;
VecSetValues(x,1,&i,&v,INSERT_VALUES);
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
VecSetValues(x,2,ii,vv,INSERT_VALUES);

call VecSetValues(x,1,i,v,INSERT_VALUES,ierr)
call VecSetValues(x,2,ii,vv,INSERT_VALUES,ierr)
```

Setting values'

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

```
VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
```

Getting values

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

```
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 seperate F90 versions for 'Get' routines

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:

```
MatCreate(MPI Comm comm,Mat *A)
MatDestroy(Mat A)
```

Several more aspects to creation:

```
MatSetType(A,MATSEQAIJ) /* or MATMPIAIJ */
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

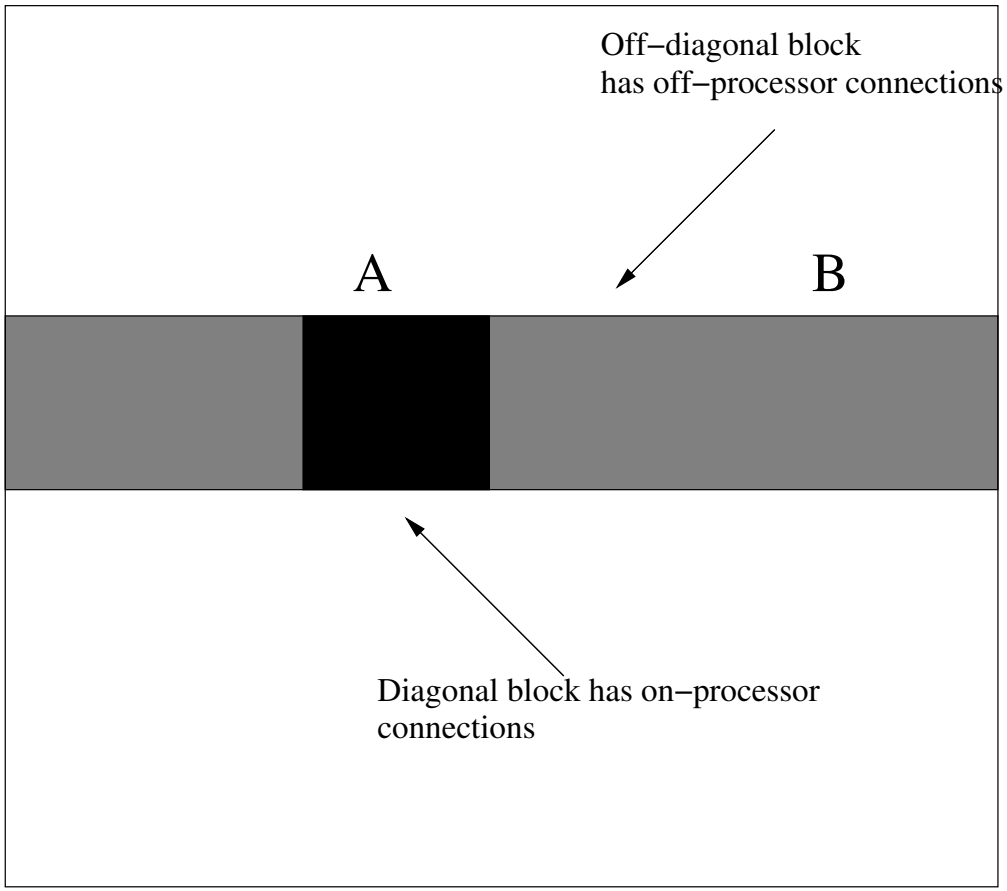
```
MatCreateSeqAIJ(MPI_Comm comm,PetscInt m,PetscInt n,
  PetscInt nz,const PetscInt nnz[],Mat *A)
MatCreateMPIAIJ(MPI_Comm comm,
  PetscInt m,PetscInt n,PetscInt M,PetscInt N,
  PetscInt d_nz,const PetscInt d_nnz[],
  PetscInt o_nz,const PetscInt o_nnz[],
  Mat *A)
```

Sequential matrix structure

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

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

Parallel matrix structure



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

```
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);
```

Setting values

Setting is independent of parallelism

Set block of values:

```
MatSetValues(Mat A,int m,const int idxm[],
    int n,const int idxn[],const PetscScalar values[],
    INSERT VALUES); /* or ADD_VALUES */
MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

(v is row-oriented)

Common case:

```
MatSetValues(A,1,&i,1,&j,&v,INSERT_VALUES);
```

Getting values

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

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

Note: for inspection only; possibly expensive.

Other matrix types

`MATBAIJ` : blocked matrices (dof per node)

`MATAIJMUMPS` : for the Mumps solver (explained later)

(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)`
- also invoked by `-mat_view`
- Sparse: only allocated positions listed
- other viewers: for instance `-mat_view_draw` (X terminal)

More matrix topics

- Viewers are also for binary dump, plotting with X
- Shell matrices: matrix-free operation
- Submatrix extraction
- Matrix partitioning for load balancing

KSP & PC: **Iterative solvers**

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

KSP: Krylov Space objects

Iterative solver basics

```
KSPCreate(comm,&solver); KSPDestroy(solver);
KSPSetOperators(solver,A,B,MAT_SAME_STRUCTURE);
/* optional */ KSPSetup(solver);
KSPSolve(solver,rhs,sol);
```

Settings in general

- Other settings, both as command and runtime option
- option `-ksp_view`
- (preconditioners discussed below)

Solver type and tolerances

```
KSPSetType(solver,KSPGMRES);
KSPSetTolerances(solver,rtol,atol,dtol,maxit);
```

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`
- tolerances: `-ksp_max_it 50`

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);
}
```

More KSP topics

- Custom monitors and convergence tests
- Method-specific options (especially GMRES)
- Null spaces

PC: Preconditioner objects

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`

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`:
- different solvers triggered by matrix type, for instance `MATAIJMUMPS`

```
myprog -mat_type aijmumps -ksp_type preonly -pc_type lu
```

Making and running PETSc programs

Installation

```
cd petsc-2.3.3
python config/configure.py
make ; make install
```

`configure.py --help` gives

```
PETSc:
  --prefix=<path>                         : Specifiy location to install PETSc (eg. /usr/local)
  --with-sudo=sudo                        : Use sudo when installing packages
  --with-default-arch=<bool>              : Allow using the last configured arch without setting PETSC_AR
  --PETSC_ARCH                            : The configuration name
  --with-petsc-arch                       : The configuration name
  --PETSC_DIR                             : The root directory of the PETSc installation
  --with-installation-method=<method>     : Method of installation, e.g. tarball, clone, etc.
```

Compiling

Petsc compile and link lines are very long!
Use PETSc include file with variables and rules:

```
include ${PETSC_DIR}/bmake/common/base
prog : ${OBJS}
        ${CLINKER} -o prog ${OBJS} ${PETSC_LIB}
```

My preference:

```
include ${PETSC_DIR}/bmake/common/variables
CFLAGS = ${PETSC_INCLUDE}
prog : ${OBJS}
        ${CLINKER} -o prog ${OBJS} ${PETSC_LIB}
FFLAGS = ${PETSC_INCLUDE}
fprog : ${OBJS}
${FLINKER} -o prog ${OBJS} ${PETSC_LIB}
```

Environment variables

- `PETSC_DIR` : different for different version numbers
- `PETSC_ARCH` : for one version, this controls real/complex or opt/debug variants

Versions available at TACC

```
petsc/2.3.2                    petsc/2.3.2-debug
petsc/2.3.2-cxx                petsc/2.3.2-cxxdebug
petsc/2.3.3(default)           petsc/2.3.3-debug
petsc/2.3.3-complex            petsc/2.3.3-complexdebug
petsc/2.3.3-cxx                petsc/2.3.3-cxxdebug
```

```
%% module load petsc/2.3.3-cxx
%% echo $PETSC_DIR
/opt/apps/petsc-intel9-2.3.3/2.3.3
%% echo $PETSC_ARCH
em64t-cxx
%%
```

Running

`mpirun -np 3 petscprog <bunch of runtime options>`
(or `ibrun` on Lonestar)

- `-log_summary` : give runtime statistics
- `-malloc_dump -memory_info` : memory statistics
- `-start_in_debugger` : parallel debugging
- `-options_left` : check for mistyped options
- `-ksp_type gmres` (et cetera) : program control

more later

Documentation and examples

- Manual in pdf form
- All man pages online

  `http://www-unix.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-2.3.3/docs/manualpages/singleindex.html`

  start at `http://www.mcs.anl.gov/petsc/`
- Example codes, found online, and in `$PETSC_DIR/src/mat/examples` et cetera
- Sometimes consult include files, for instance `$PETSC_DIR/include/petscmat.h`

Fortran

- Include files:

```
#include "include/finclude/petsc.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
```

- Separate F90 version of various 'Get' routines
- Null pointers: C is tolerant for 0 or `PETSC_NULL`, Fortran use `PETSC_NULL_CHARACTER`, `PETSC_NULL_INTEGER` et cetera. Example:

```
call PetscOptionsGetInt(PETSC_NULL_CHARACTER,"-name",
    N,flg,ierr)
```

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

	Max	Max/Min	Avg	Total
Time (sec):	5.493e-01	1.00006	5.493e-01	
Objects:	2.900e+01	1.00000	2.900e+01	
Flops:	1.373e+07	1.00000	1.373e+07	2.746e+07
Flops/sec:	2.499e+07	1.00006	2.499e+07	4.998e+07
Memory:	1.936e+06	1.00000		3.871e+06
MPI Messages:	1.040e+02	1.00000	1.040e+02	2.080e+02
MPI Msg Lengths:	4.772e+05	1.00000	4.588e+03	9.544e+05
MPI Reductions:	1.450e+02	1.00000		

Log summary: details

	Max	Ratio	Max	Ratio	Max	Ratio	Avg len	%T	%F	%M	%L	%R	%T	%F	%M	%L	%R	Mflop/s
MatMult	100	1.0	3.4934e-02	1.0	1.28e+08	1.0	8.0e+02	6	32	96	17	0	6	32	96	17	0	255
MatSolve	101	1.0	2.9381e-02	1.0	1.53e+08	1.0	0.0e+00	5	33	0	0	0	5	33	0	0	0	305
MatLUFactorNum	1	1.0	2.0621e-03	1.0	2.18e+07	1.0	0.0e+00	0	0	0	0	0	0	0	0	0	0	43
MatAssemblyBegin	1	1.0	2.8350e-03	1.1	0.00e+00	0.0	1.3e+05	0	0	3	83	1	0	0	3	83	1	0
MatAssemblyEnd	1	1.0	8.8258e-03	1.0	0.00e+00	0.0	4.0e+02	2	0	1	0	3	2	0	1	0	3	0
VecDot	101	1.0	8.3244e-03	1.2	1.43e+08	1.2	0.0e+00	1	7	0	0	35	1	7	0	0	35	243
KSPSetup	2	1.0	1.9123e-02	1.0	0.00e+00	0.0	0.0e+00	3	0	0	0	2	3	0	0	0	2	0
KSPSolve	1	1.0	1.4158e-01	1.0	9.70e+07	1.0	8.0e+02	26	100	96	17	92	26	100	96	17	92	194

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)