Introduction to PETSc

Victor Eijkhout

Outline

- Introduction
- An example program
- Making and running PETSc programs
- `Vec` datatype: vectors
- `Mat` Datatype: matrix
- `KSP & PC`: Iterative solvers
- Grid manipulation
- Nonlinear example
- `SNES`: Nonlinear solvers
- `TS`: Time stepping
- Profiling, debugging

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

External packages

- 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

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

An example program

note

This example is simplified, it is intended to convey the 'taste' of PETSc, not all the details.

However, there will be several complete examples in the lab exercises.

Also the PETSc source tree comes with many examples.

Read input parameter

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

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

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

Create matrix

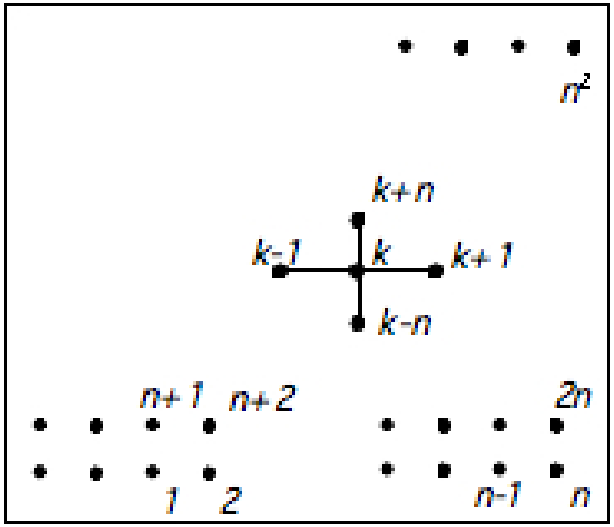
Create distributed matrix on communicator, set local and global size.

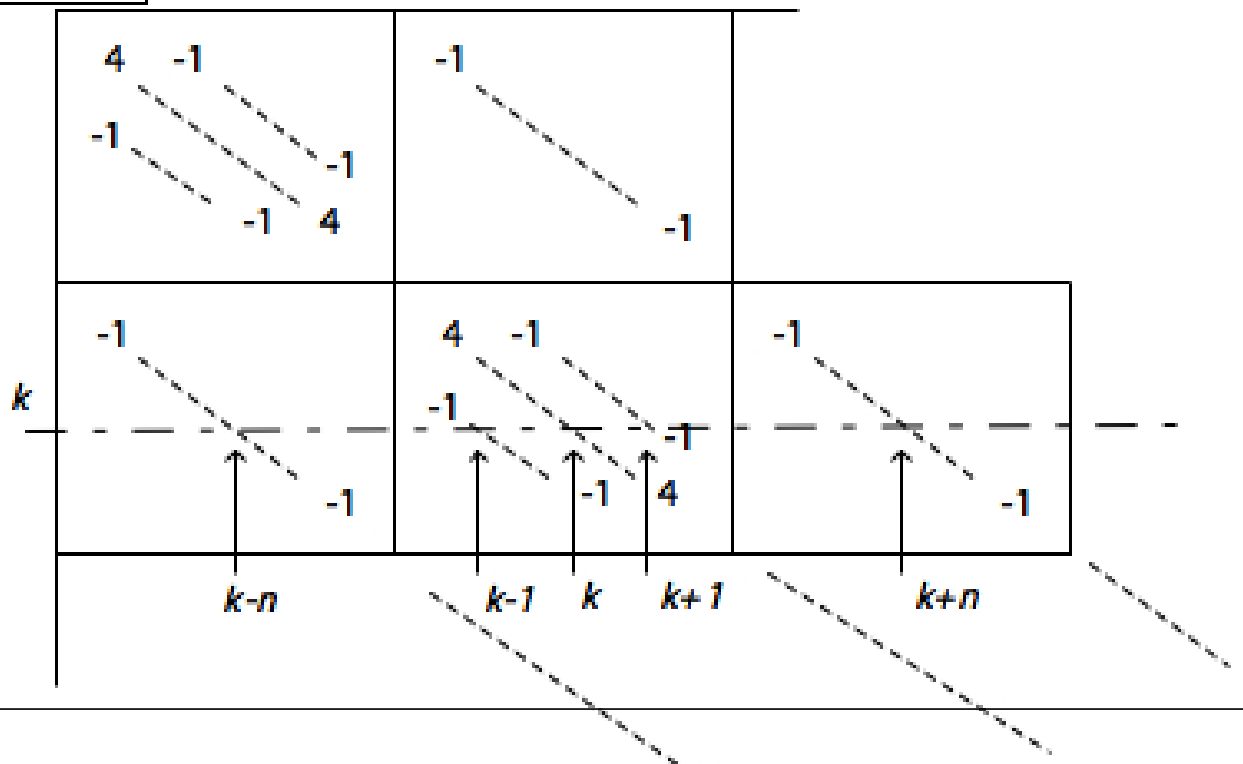
```
  comm = MPI_COMM_WORLD; // or subset
  MatCreate(comm,&A);
  MatSetType(A,MATMPIAIJ);
  MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,
    matrix_size,matrix_size);

  comm = MPI_COMM_WORLD ! or subset
  call MatCreate(comm,A)
  call MatSetType(A,MATMPIAIJ)
  call MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,
 >      matrix_size,matrix_size)
```

Parallel layout of matrices and vectors

- local size and global size
- local rows contiguous
- either one can be set by the user or determined by PETSc
- matrix/vector local sizes must fit

Five-point Laplacian





Fill in matrix elements (C)

```
MatGetOwnershipRange(A,&low,&high);
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;
      if (i>0) MatSetValues
          (mat,1,&I,1,&J,&v,INSERT_VALUES);
      J = I+1 // et cetera
    }
  }
}
MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY);
```

Fill in matrix elements (F)

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

Finite Element Matrix assembly

```
for (e=myfirstelement; e<mylastelement; e++) {
  for (i=0; i<nlocalnodes; i++) {
    I = localtoglobal(e,i);
    for (j=0; j<nlocalnodes; j++) {
      J = localtoglobal(e,j);
      v = integration(e,i,j);
      MatSetValues
          (mat,1,&I,1,&J,&v,ADD_VALUES);
      ....
    }
  }
}
MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY);
```

Create solver

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

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

Create preconditioner

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

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

Input and output vectors

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

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

Another way to determine the vector size:

```
MatGetLocalSize(A,&isize,PETSC_NULL);
VecCreateMPI(comm,isize,PETSC_DECIDE,&Rhs);
```

Solve! (C)

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

Solve! (F)

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

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

Quick experimentation

Set iterative solver and preconditioner from the commandline:

```
yourprog -ksp_type gmres -ksp_gmres_restart 25
    -pc_type ilu -pc_factor_levels 3
```

requires in your code:

```
KSPSetFromOptions(solver);
```

Residual calculation

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

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

Clean up

Free all objects, both the obvious ones (matrix) and the non-obvious ones (solver)

```
VecDestroy(Res);
KSPDestroy(Solver);
VecDestroy(Rhs);
VecDestroy(Sol);
```

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

Making and running PETSc programs

Installation

```
cd petsc-3.0.0
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,
  et cetera: many more
```

Include files

Multiple include files; in C only the highest level

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

In Fortran sequence of include files *in the subprogram*

```
#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscmat.h"
#include "finclude/petscksp.h"
#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 function return values

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

C:

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

Fortran:

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

Petsc initialize / finalize

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

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

Fortran:

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

Compiling

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

```
include ${PETSC_DIR}/conf/variables
include ${PETSC_DIR}/conf/rules
prog : ${OBJS}
        ${CLINKER} -o prog ${OBJS} ${PETSC_LIB}
```

My preference:

```
include ${PETSC_DIR}/conf/variables
CFLAGS = ${PETSC_INCLUDE}
FFLAGS = ${PETSC_INCLUDE}
```

and write your own compile rules

Find out what PETSC_INCLUDE and PETSC_LIB are:

```
cd $PETSC_DIR ; make getincludedirs getlinklibs
```

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: 2.3.3 and 3.0.0

```
petsc/3.0.0                    petsc/3.0.0-debug
petsc/3.0.0-complex            petsc/3.0.0-complexdebug
petsc/3.0.0-cxx                petsc/3.0.0-cxxdebug
petsc/3.0.0-uni                petsc/3.0.0-unidebug
```

```
%% module load petsc/3.0.0-cxxdebug
%% echo $PETSC_DIR
/opt/apps/intel10_1/mvapich1_1_0_1/petsc/3.0.0/
%% echo $PETSC_ARCH
barcelona-cxxdebug
```

Running

Parallel invocation. On your own machine:

`mpirun -np 3 petscprog <bunch of runtime options>`

On lonestar and ranger: using `ibrun`

Petsc has lots of runtime options.

- `-log_summary` : give runtime statistics
- `-malloc_dump -memory_info` : memory statistics
- `-start_in_debugger` : parallel debugging (not on our clusters, but very convenient on your laptop)
- `-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-3.0.0/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 (already mentioned):

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

More Fortran

- Fortran/C interoperability:

```
#if defined(PETSC HAVE FORTRAN CAPS)
#define dabsc DABSC
#elif !defined(PETSC HAVE FORTRAN UNDERSCORE)
#define dabsc dabsc
#endif
.....
dabsc (&n,x,y); /* call the Fortran function */
```

- Indexing in `MatSetValues` and such is always zero-based
- Function pointers: routine is assumed to be in the same language

`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 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:

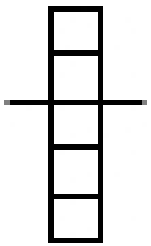
```
VecDuplicate(Vec v,Vec *w);
```

Parallel layout

Local or global size in

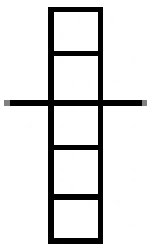
```
VecSetSizes(Vec v, int m, int M);
```

Global size can be specified as `PETSC_DECIDE`.



VecSetSizes(V,2,5)

VecSetSizes(V,3,5)



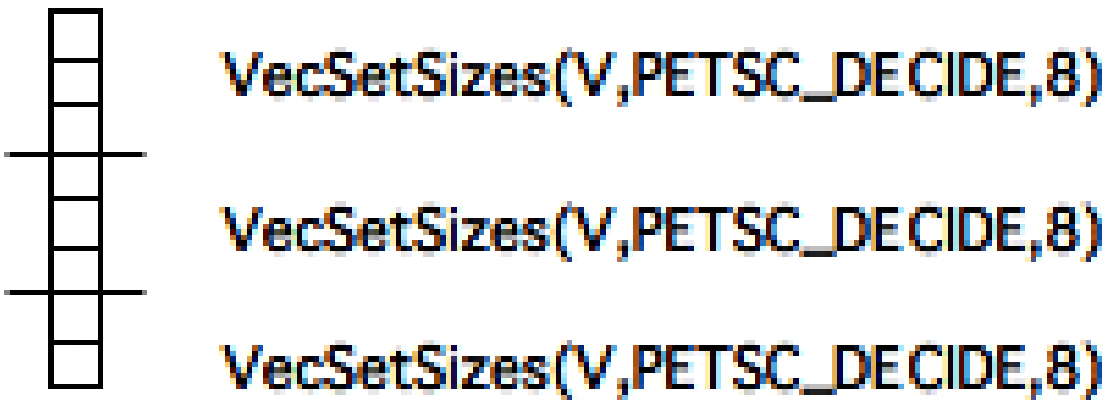
VecSetSizes(V,2,PETSC_DECIDE)

VecSetSizes(V,3,PETSC_DECIDE)

Parallel layout up to PETSc

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

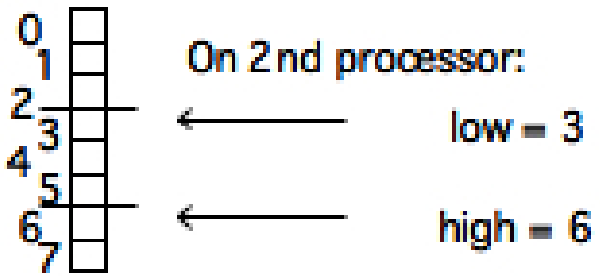
Local size can be specified as `PETSC_DECIDE`.



Query parallel layout

Query vector layout:

```
VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)
VecGetOwnershipRange(x,low,high,ierr) ! F
```



Query general layout:

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

(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 (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

```
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 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)
```

Matrix Preallocation

- PETSc matrix creation is very flexible:
- No preset sparsity pattern
- any processor can set any element
  $\Rightarrow$ potential for lots of malloc calls
- (run your code with `-memory_info`, `-malloc_log`)

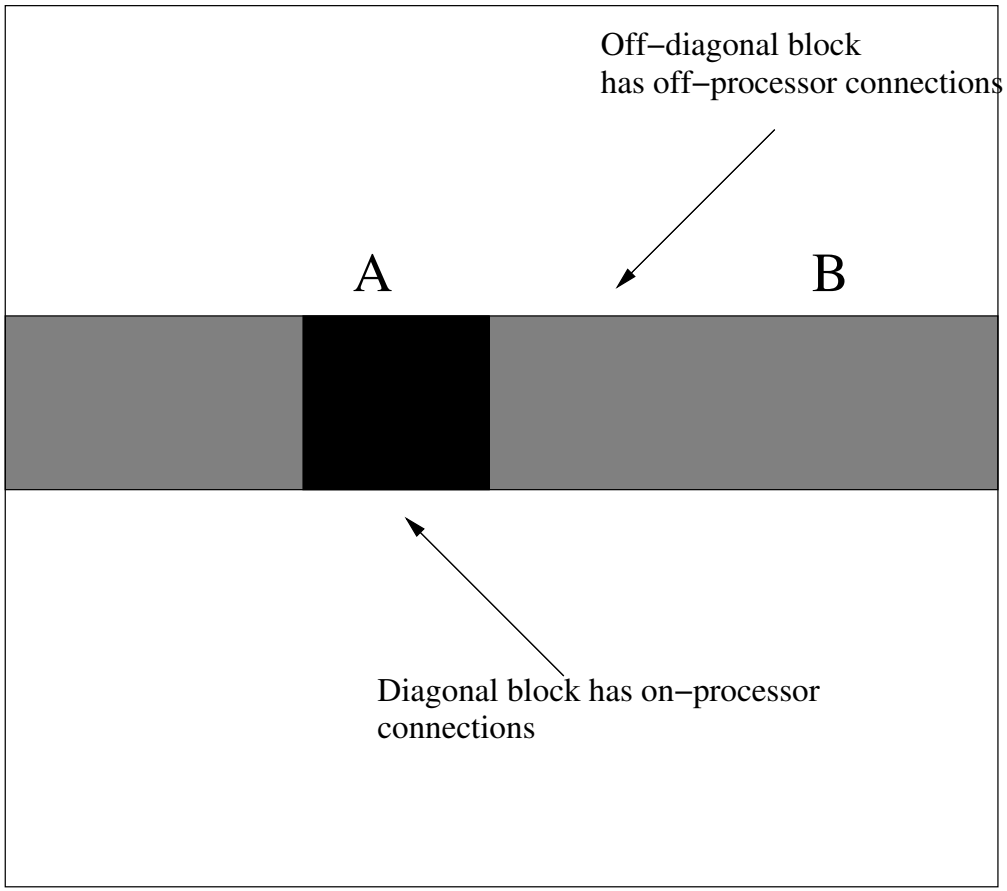
malloc is very expensive:
tell PETSc the matrix' sparsity structure
(do construction loop twice: once counting, once making)

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 (C)

- 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.

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)`
- 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, matmult 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

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);
}
```

Monitors and convergence tests

```
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 fpr 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: 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`

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

`KSPSetOperators(solver,A,B,structureflag)`

- `B` is basis for preconditioner, need not be `A`
- `structureflag` can be `SAME_PRECONDITIONER`, `SAME_NONZERO_PATTERN`, `DIFFERENT_PRECONDITIONER`: 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);
PCFactorSetUseDropTolerance
    (PC pc,double dt,double dtcol,int dtcount);
PCFactorSetReuseOrdering(PC pc,PetscTruth flag);
PCFactorSetReuseFill(PC pc,PetscTruth flag);
PCFactorSetAllowDiagonalFill(PC pc);
```

Prevent indefinite preconditioners:

```
PCFactorSetShiftPd(PC pc,PetscTruth shift)
```

option `-pc_shift_positive_definite`

SOR

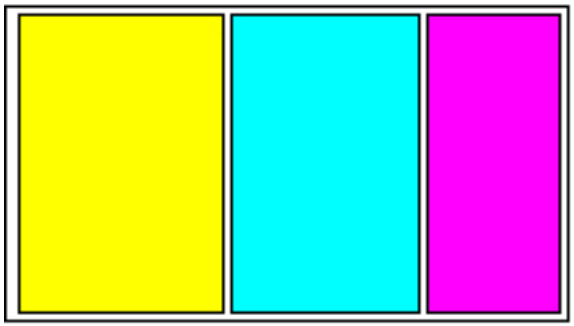
PCSOR preconditioner

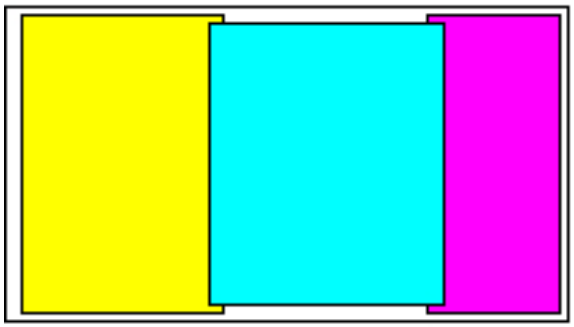
```
PCSORSetOmega(PC pc,double omega);
PCSORSetIterations(PC pc,int its,int lits);
PCSORSetSymmetric(PC pc,MatSORType type);
```

Block Jacobi and Additive Schwarz

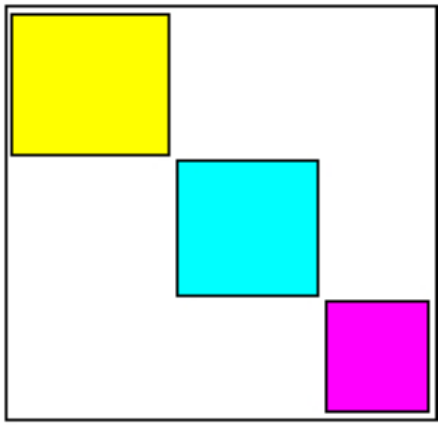
- Factorization preconditioners are sequential;
- can be made parallel by use in Block Jacobi or Additive Schwarz methods
- each processor has its own block(s) to work with

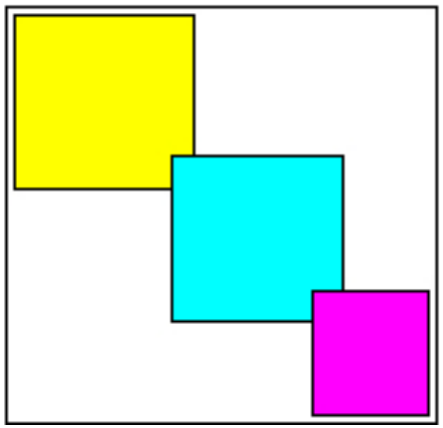
Domain partitioning





Matrix blocks





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)

```
PCASMSetOverlap(PC pc,int overlap);
```

Matrix-free solvers

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

```
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_factor_mat_solve_package mumps -ksp_type preonly
```

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:

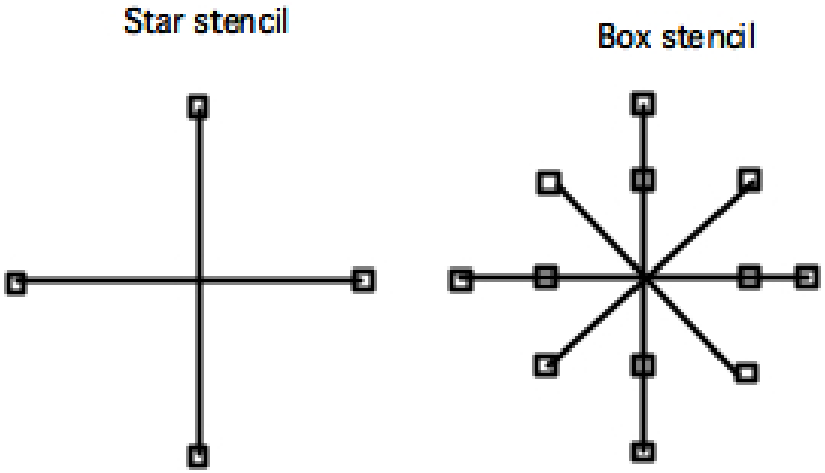
```
%% grep hypre petsc-2.3.3/bmake/$PETSC_ARCH/petscconf.h
#ifndef PETSC_HAVE_HYPRE
#define PETSC_HAVE_HYPRE 1
#ifndef PETSC_HAVE_LIBHYPRE
#define PETSC_HAVE_LIBHYPRE 1
```

Grid manipulation

Regular grid: DA

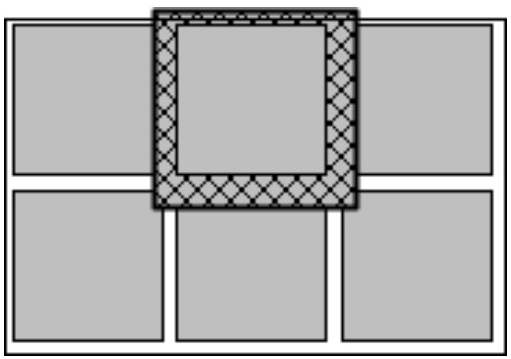
DAs are for storing vector field, not matrix.

Support for different stencil types:



Ghost regions around processors

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



DA construction

```
DACreate2d(comm, wrap, type, M, N, m, n, dof, s, lm[],
ln[], DA *da)
```

wrap: Specifies periodicity
`DA_NONPERIODIC`, `DA_XPERIODIC`, `DA_YPERIODIC`, or `DA_XYPERIODIC`
type: Specifies stencil
DA STENCIL BOX or DA 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: Alternative array of local sizes

Use PETSC NULL for the default

Associated vectors

```
DACreateGlobalVector(DA da,Vec *g);
DACreateLocalVector(DA da,Vec *l);

DAGlobalToLocalBegin(DA da,Vec g,InsertMode iora,Vec l);
DAGlobalToLocalEnd(DA da,Vec g,InsertMode iora,Vec l);

purely local:
DALocalToGlobal(DA da,Vec l,InsertMode mode,Vec g);

local -> global -> local :
DALocalToLocalBegin(DA da,Vec l1,InsertMode iora,Vec l2);
DALocalToLocalEnd(DA da,Vec l1,InsertMode iora,Vec l2);
```

Irregular grid: IS & VecScatter

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

```
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.

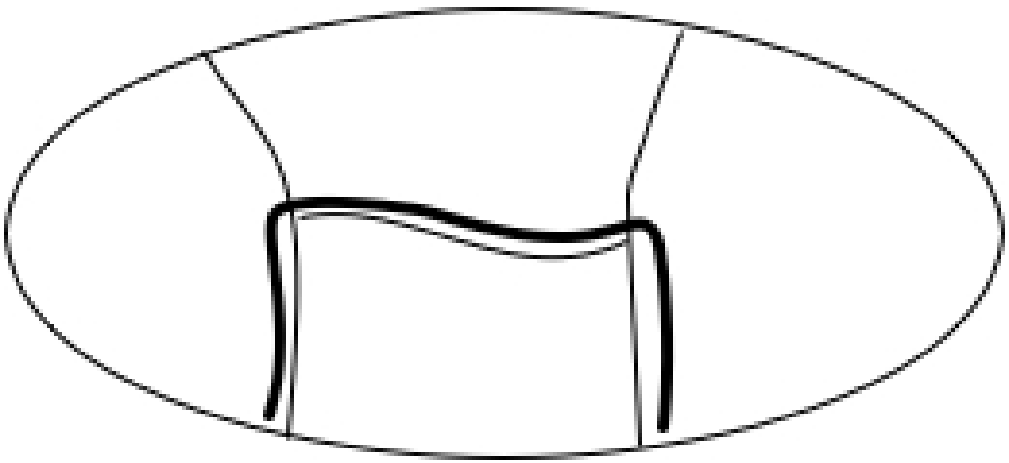
Retrieving information

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:



Nonlinear example

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:

```
SNES           snes;

SNESCreate(PETSC_COMM_WORLD,&snes)

VecCreate(PETSC_COMM_WORLD,&r)
SNESSetFunction(snes,r,FormFunction,PETSC_NULL)

MatCreate(PETSC_COMM_WORLD,&J)
SNESSetJacobian(snes,J,J,FormJacobian,PETSC_NULL)

SNESSolve(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,voi
{
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;
}
```

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)
```

SNES: Nonlinear solvers

Basics

```
SNESCreate(MPI Comm comm,SNES *snes);
SNESSetType(SNES snes,SNESType method);
SNESSetFromOptions(snes);
SNESDestroy(SNES snes);

SNESSetFunction(SNES snes, Vec r, residualFunc, void *ctx)
SNESSetJacobian(SNES snes, Mat A, Mat M, jacFunc, void *ctx
SNESSolve(SNES snes, Vec b, Vec x)
SNESGetIterationNumber(snes,&its)
SNESGetKSP(SNES snes, KSP *ksp)
```

lots more options: 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 converngence: `-snes_ksp_ew_conv` (Eisenstat Walker)

sophisticated stuff

- Jacobian through finite difference
- 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

	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

User events

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