PETSc Course

Victor Eijkhout

2024 COE 379L / CSE 392

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 diffculties 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

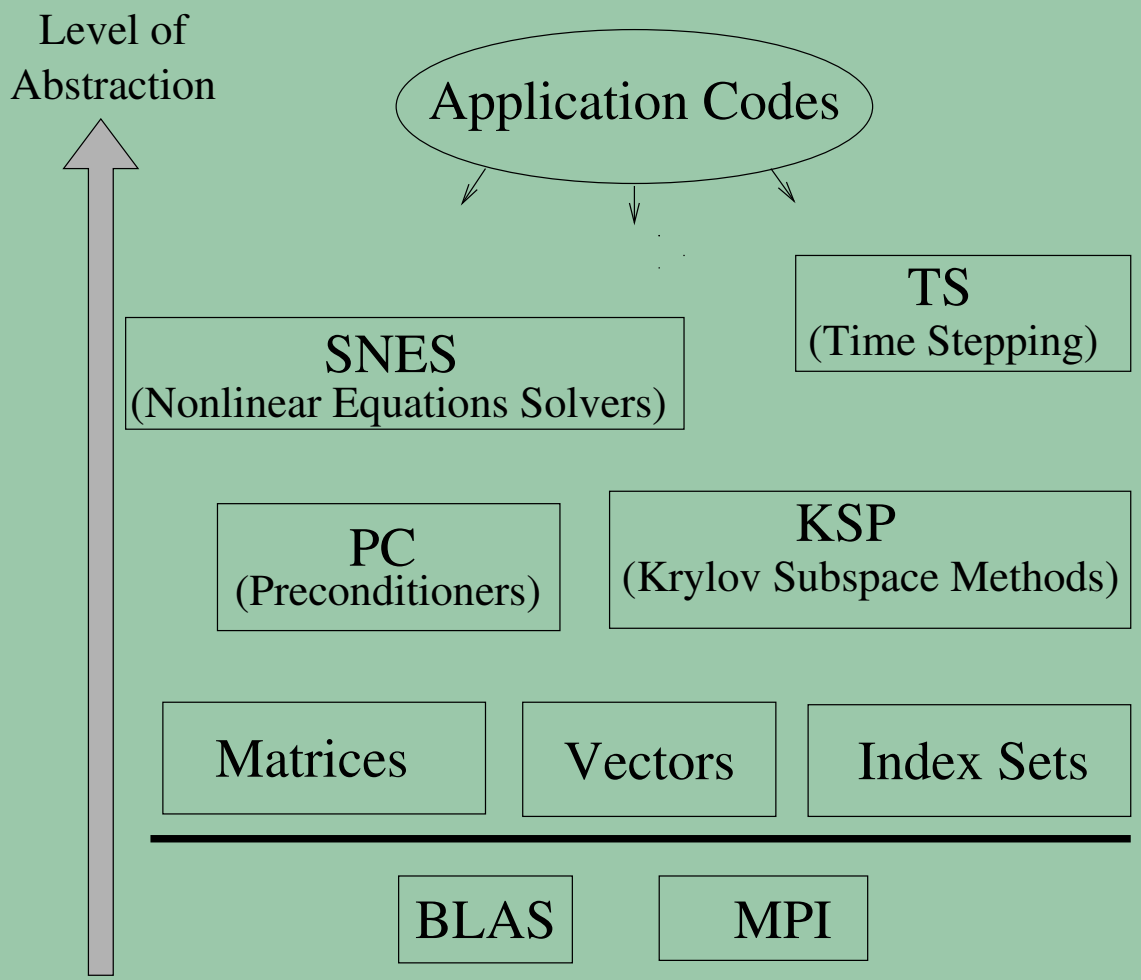
- Scientific Computations: parallel linear algebra, in particular linear and nonlinear solvers
- Toolkit: Contains high level solvers, but also the low level tools to roll your own.
- Portable: Available on many platforms, basically anything that has MPI

Why use it? It's big, powerful, well supported.

What is in PETSc?

- Linear algebra data structures, all serial/parallel
- Linear system solvers (sparse/dense, iterative/direct)
- Nonlinear system solvers
- Optimization: `TAO` (used to be separate library)
- Tools for distributed matrices
- Support for profiling, debugging, graphical output

Structure of a PETSc application



TACC

Parallel Numerical Components of PETSc

Nonlinear Solvers		
Newton–based Methods		Other
Line Search	Trust Region	

Time Steppers			
Euler	Backward Euler	Pseudo–Time Stepping	Other

Krylov Subspace Methods							
GMRES	CG	CGS	Bi–CG–Stab	TFQMR	Richardson	Chebychev	Other

Preconditioners						
Additive Schwarz	Block Jacobi	Jacobi	ILU	ICC	LU (sequential only)	Other

Matrices				
Compressed Sparse Row (AIJ)	Block Compressed Sparse Row (BAIJ)	Block Diagonal (BDiag)	Dense	Other

Index Sets

Documentation and help

- Web page: `https://petsc.org/`
- Documentation (pdf/html): `https://petsc.org/release/docs/`
- Follow-up to this tutorial: eijkhout@tacc.utexas.edu
- PETSc on your local cluster: ask your local support
- General questions about PETSc: petsc-maint@mcs.anl.gov
- Example codes, found online, and in `$PETSC_DIR/src/mat/examples` et cetera
- Sometimes consult include files, for instance `$PETSC_DIR/include/petscmat.h`

External packages

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

- Dense linear algebra: `Scalapack`, `Plapack`, `Elemental`
- Grid partitioning software: `ParMetis`, `Jostle`, `Chaco`, `Party`
- ODE solvers: `PVODE`
- Optimization: `TAO` (now integrated)
- Eigenvalue solvers (including SVD): `SLEPc` (integrated)

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
- Transparent: same code works sequential and parallel.
  (Some objects explicitly declared `Seq/MPI`)
- $\Rightarrow$ you do not need to know much MPI when you use PETSc
- All objects in Petsc are defined on a communicator;
  can only interact if on the same communicator
- No OpenMP used in the library:
  user can use shared memory programming.
- Likewise, threading is kept outside of PETSc code.
- Limited Graphics Processing Unit (GPU) support: know what

Object oriented design

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

Overloading:

```
MATMult(A,x,y); // y <- A x
```

same for sequential, parallel, dense, sparse, FFT

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.

(And don't worry about function call overhead.)

Getting started

Include files, C

```
1 #include "petsc.h"
2 int main(int argc,char **argv)
```


Include files, Fortran

Include file for preprocessor definitions,
module for library definitions

```
1  program basic
2  #include <petsc/finclude/petsc.h>
3  use petsc
4  implicit none
```

Include files, Python

```
1  from petsc4py import PETSc
```

Variable declarations, C

```
1    KSP            solver;
2    Mat            A;
3    Vec            x,y;
4    PetscInt       n = 20;
5    PetscScalar    v;
6    PetscReal      nrm;
```

Note Scalar vs Real

Variable declarations, F

```
1    KSP              :: solver
2    Mat              :: A
3    Vec              :: x,y
4    PetscInt         :: j(3)
5    PetscScalar      :: mv
6    PetscReal        :: nrm
```

Much like in C

Library setup, C

```c
// init.c
PetscCall( PetscInitialize
  (&argc,&argv,(char*)0,help) );
int flag;
MPI_Initialized(&flag);
if (flag)
  printf("MPI was initialized by PETSc\n");
else
  printf("MPI not yet initialized\n");
```

Can replace `MPI_Init`

General: Every routine has an error return. Catch that value!

Library setup, F

```
1     call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
2         CHKERRQ(ierr)
3   // all the petsc work
4     call PetscFinalize(ierr); CHKERRQ(ierr)
```

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

A word about datatypes

PETSc programs can not mix single and double precision, nor real/complex:
`PetscScalar` is single/double/complex depending on the installation.
`PetscReal` is always real, even in complex installations.

Similarly, `PetscInt` is 32/64 bit depending.

Other scalar data types: `PetscBool`, `PetscErrorCode`

TACC note.

```
module spider petsc
module avail petsc

module load petsc/3.16-i64 # et cetera
```

Debug and production

While you are developing your code:

```
module load petsc/3.16-debug
# or 3.16-complexdebug, i64debug, rtxdebug &c
```

This does bounds tests and other time-wasting error checking.

Production:

```
module load petsc/3.16
```

This will just bomb if your program is not correct.

Every petsc configuration is available as debug and non-debug.

Exercise 1 (`hello`)

Look up the function `PetscPrintf` and print a message
'This program runs on 27 processors'
from process zero.

- Start with the template code `hello.c`/`hello.F`
- (or see slide **??**)
- Compile with `make hello`
- Part two: use `PetscSynchronizedPrintf`

PetscPrintf

```
C:
PetscErrorCode  PetscPrintf(MPI_Comm comm,const char format[],...)

Fortran:
PetscPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.Print(type cls, *args, **kwargs)
kwargs:
comm : communicator object
```

PetscPrintf in Fortran

Can only print character buffer:

```
1     character*80       msg
2     write(msg,10) n
3    10 format("Input parameter:",i5)
4     call PetscPrintf(PETSC_COMM_WORLD,msg,ierr)
```

Less elegant than `PetscPrintf` in C

About routine prototypes: C/C++

Prototype:

```
1  PetscErrorCode VecCreate(MPI_Comm comm,Vec *v);
```

Use:

```
1  PetscErrorCode ierr;
2  MPI_Comm comm = MPI_COMM_WORLD;
3  Vec v;
4  ierr = VecCreate( comm,&vec ); CHKERRQ(ierr).
```

(always good idea to catch that error code)

About routine prototypes: Fortran

TACC

Prototype

```
1  Subroutine VecCreate
2      ( comm,v,ierr )
3  Type(MPI_Comm) :: comm
4  Vec            :: v
5  PetscErrorCode :: ierr
```

Use:

```
1  Type(MPI_Comm) :: &
2      comm = MPI_COMM_WORLD
3  Vec            :: v
4  PetscErrorCode :: ierr
5
6  call VecCreate(comm,v,ierr)
```

- Final parameter always error parameter. Do not forget!
- MPI types are of often `Type(MPI_Comm)` and such,
- PETSc datatypes are handled through the preprocessor.

About routine prototypes: Python

TACC

Object methods:

```
# definition
PETSc.Mat.setSizes(self, size, bsize=None)

# use
A = PETSc.Mat().create(comm=comm)
A.setSizes( ( (None,matrix_size), (None,matrix_size) ) )
```

Class methods:

```
# definition
PETSc.Sys.Print(type cls, *args, **kwargs)

# use
PETSc.Sys.Print("detecting n option")
```


Note to self

```
1    PetscInitialize
2      (&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

Routine start/end, C

Debugging support:

```
1 PetscFunctionBeginUser;
2 // all statements
3 PetscFunctionReturn(0);
```

leads to informative tracebacks.

(Only in C, not in Fortran)

Example: function with error

```
1 // backtrace.c
2 PetscErrorCode this_function_bombs() {
3   PetscFunctionBegin;
4   SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
5   PetscFunctionReturn(0);
6 }
```


Example: error traceback

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.htm
[0]PETSC ERROR: Petsc Release Version 3.12.2, Nov, 22, 2019
[0]PETSC ERROR: backtrace on a [computer name]
[0]PETSC ERROR: Configure options [all options]
[0]PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
[0]PETSC ERROR: #2 main() line 30 in backtrace.c
```


Exercise 2 (root)

Start with `root.c`. Write a function that computes a square root, or displays an error on negative input: Look up the definition of `SETERRQ1`.

```
1    x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
2    PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
3    x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
4    PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
```

This should give as output:

```
Root of 1.500000 is 1.224745
[0]PETSC ERROR: ----- Error Message --------------------------------------------
[0]PETSC ERROR: Cannot compute the root of -2.600000
[...]
[0]PETSC ERROR: #1 square_root() line 23 in root.c
[0]PETSC ERROR: #2 main() line 39 in root.c
```


Commandline arguments, C

(I'm leaving out the `CHKERRQ(ierr)` in the examples,
but do use this in actual code)

```
1  ierr = PetscOptionsGetInt
2      (PETSC_NULL,PETSC_NULL,"-n",&n,&flag); CHKERRQ(ierr);
3  ierr = PetscPrintf
4      (comm,"Input parameter: %d\n",n); CHKERRQ(ierr);
```

Read commandline argument, print out from processor zero;
flag can be `PETSC_NULL` if not wanted

Commandline argument, F

```
1     call PetscOptionsGetInt(
2       PETSC_NULL_OPTIONS, PETSC_NULL_CHARACTER, &
3         "-n",n,PETSC_NULL_BOOL,ierr)
```

Note the *PETSC_NULL_XXX*: Fortran has strict type checking.

Program parameters, Python

```python
nlocal = PETSc.Options().getInt("n",10)
```

`Vec` datatype: vectors

Create calls

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

```
1  VecCreate(MPI_Comm comm,Vec *v);
2  VecDestroy(Vec *v);
3
4  Vec V;
5  VecCreate(MPI_COMM_WORLD,&V);
6  VecDestroy(&V);
```


Create calls, Fortran

```
1  Vec :: V
2  call VecCreate(MPI_COMM_WORLD,V,e)
3  call VecDestroy(V,e)
```

Note: in Fortran there are no 'star' arguments

More about vectors

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

The vector object is not completely created in one call:

```
VecSetType(V,VECMPI) // or VECSEQ
VecSetSizes(Vec v, int m, int M);
```

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

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

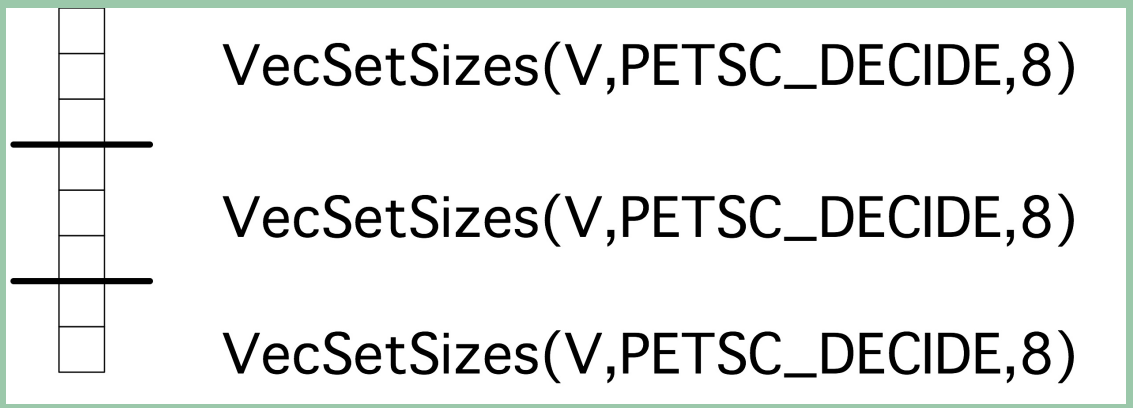

Create is a class method:

```
1  ## setvalues.py
2  comm = PETSc.COMM_WORLD
3  x = PETSc.Vec().create(comm=comm)
4  x.setType(PETSc.Vec.Type.MPI)
```

Parallel layout up to PETSc

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

Local size can be specified as `PETSC_DECIDE`.

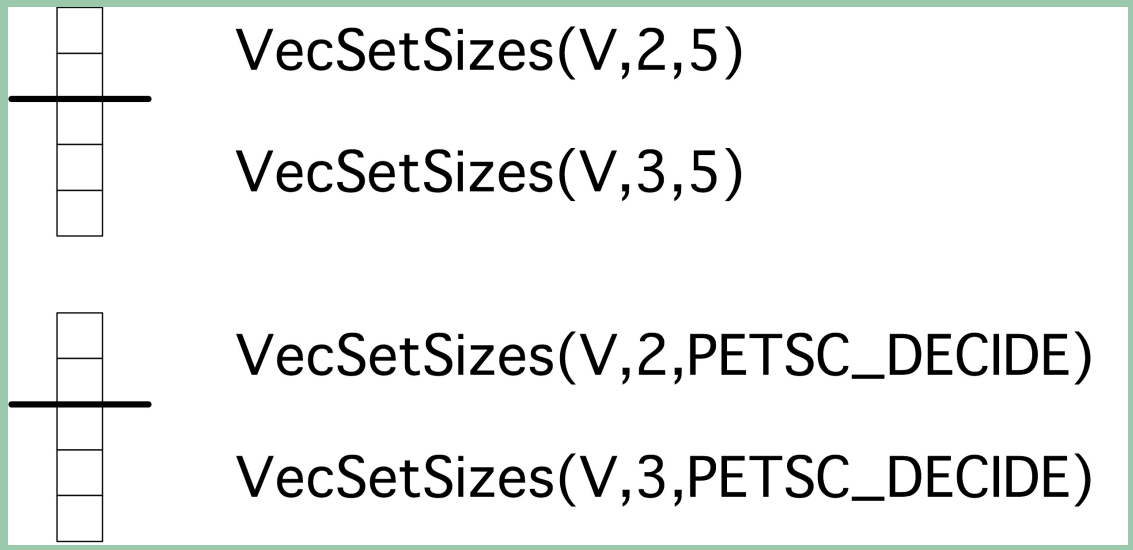


Parallel layout specified

Local or global size in

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

Global size can be specified as `PETSC_DECIDE`.



Vector layout in python

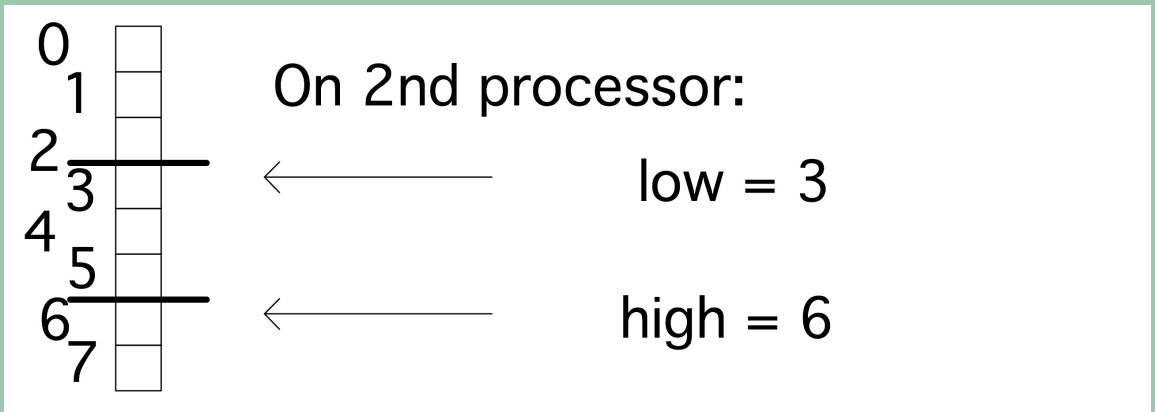
Local and global sizes in a tuple,
`PETSc.DECIDE` for parameter not specified.

```
1  x.setSizes([2,PETSc.DECIDE])
```


Query parallel layout

Query vector layout:

```
1 VecGetSize(Vec,PetscInt *globalsize)
2 VecGetLocalSize(Vec,PetscInt *localsize)
3 VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)
```



Layout, regardless object

Query general layout:

```
1  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!):

```
VecSetValue
   (Vec x,int row,PetscScalar value,
    InsertMode mode);

i = 1; v = 3.14;
VecSetValue(x,i,v,INSERT_VALUES);

call VecSetValue(x,i,v,INSERT_VALUES,e)
```

The other insertmode is `ADD_VALUES`.

Setting values by block

Set individual elements (global indexing!):

```
VecSetValues(Vec x,int n,int *rows,PetscScalar *values,
    InsertMode mode); // INSERT_VALUES or ADD_VALUES

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

ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1
call VecSetValues(x,2,ii,vv,INSERT_VALUES,ierr,e)
```

Setting values: Python

```
x.setValue(0,1.)
```

```
x.setValues( [2*procno,2*procno+1], [2.,3.] )
```


Setting values

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

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

'Latency hiding':
some of the implementation is visible here to the user

Basic operations

```
1  VecAXPY(Vec y,PetscScalar a,Vec x);   /* y <- y + a x */
2  VecAYPX(Vec y,PetscScalar a,Vec x);   /* y <- a y + x */
3  VecScale(Vec x, PetscScalar a);
4  VecDot(Vec x, Vec y, PetscScalar *r); /* several variants */
5  VecMDot(Vec x,int n,Vec y[],PetscScalar *r);
6  VecNorm(Vec x,NormType type, PetscReal *r);
7  VecSum(Vec x, PetscScalar *r);
8  VecCopy(Vec x, Vec y);
9  VecSwap(Vec x, Vec y);
10 VecPointwiseMult(Vec w,Vec x,Vec y);
11 VecPointwiseDivide(Vec w,Vec x,Vec y);
12 VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);
13 VecMax(Vec x, int *idx, double *r);
14 VecMin(Vec x, int *idx, double *r);
15 VecAbs(Vec x);
16 VecReciprocal(Vec x);
17 VecShift(Vec x,PetscScalar s);
```


Exercise 3 (vec)

Create a vector where the values are a single sine wave. using `VecGetSize`, `VecGetLocalSize`, `VecGetOwnershipRange`. Quick visual inspection:

```
ibrun vec -n 12 -vec_view
```


Exercise 4 (vec)

Use the routines `VecDot`, `VecScale` and `VecNorm` to compute the inner product of vectors x,y, scale the vector x, and check its norm:

$$p \leftarrow x^t y$$
$$x \leftarrow x/p$$
$$n \leftarrow \|x\|_2$$

Split dot products and norms

MPI is capable (in principle) of 'overlapping computation and communication'.

- Start inner product / norm with `VecDotBegin` / `VecNormBegin`;
- Conclude inner product / norm with `VecDotEnd` / `VecNormEnd`;

Also: start/end multiple norn/dotproduct operations.

Direct access to vector 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?

Solution 1. Create vector from user provided array:

```
1 VecCreateSeqWithArray(MPI_Comm comm,
2   PetscInt n,const PetscScalar array[],Vec *V)
3 VecCreateMPIWithArray(MPI_Comm comm,
4   PetscInt n,PetscInt N,const PetscScalar array[],Vec *vv)
```

Solution 2. Retrive the internal array:

```
1 VecGetArray(Vec x,PetscScalar *a[])
2 /* do something with the array */
3 VecRestoreArray(Vec x,PetscScalar *a[])
```

Note: local only; see `VecScatter` for more general mechanism)

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);
```

Fortran: `PETSC_NULL_INTEGER`

More array juggling

- `VecPlaceArray`: replace the internal array; the original can be restored with `VecRestoreArray`
- `VecReplaceArray`: replace and free the internal array.

Array handling in F90

```
1       PetscScalar, pointer :: xx_v(:)
2       ....
3       call VecGetArrayF90(x,xx_v,ierr)
4       a = xx_v(3)
5       call VecRestoreArrayF90(x,xx_v,ierr)
```

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

`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)

If you already have a CRS matrix

```
1  PetscErrorCode  MatCreateSeqAIJWithArrays
2    (MPI_Comm comm,PetscInt m,PetscInt n,
3     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
  $\Rightarrow$ potential for lots of malloc calls
- tell PETSc the matrix' sparsity structure
  (do construction loop twice: once counting, once making)
- Re-allocating is expensive:

```
1  MatSetOption(A,MAT_NEW_NONZERO_LOCATIONS,PETSC_FALSE);
```

(is default) Otherwise:

```
[1]PETSC ERROR: Argument out of range
[1]PETSC ERROR: New nonzero at (0,1) caused a malloc
```

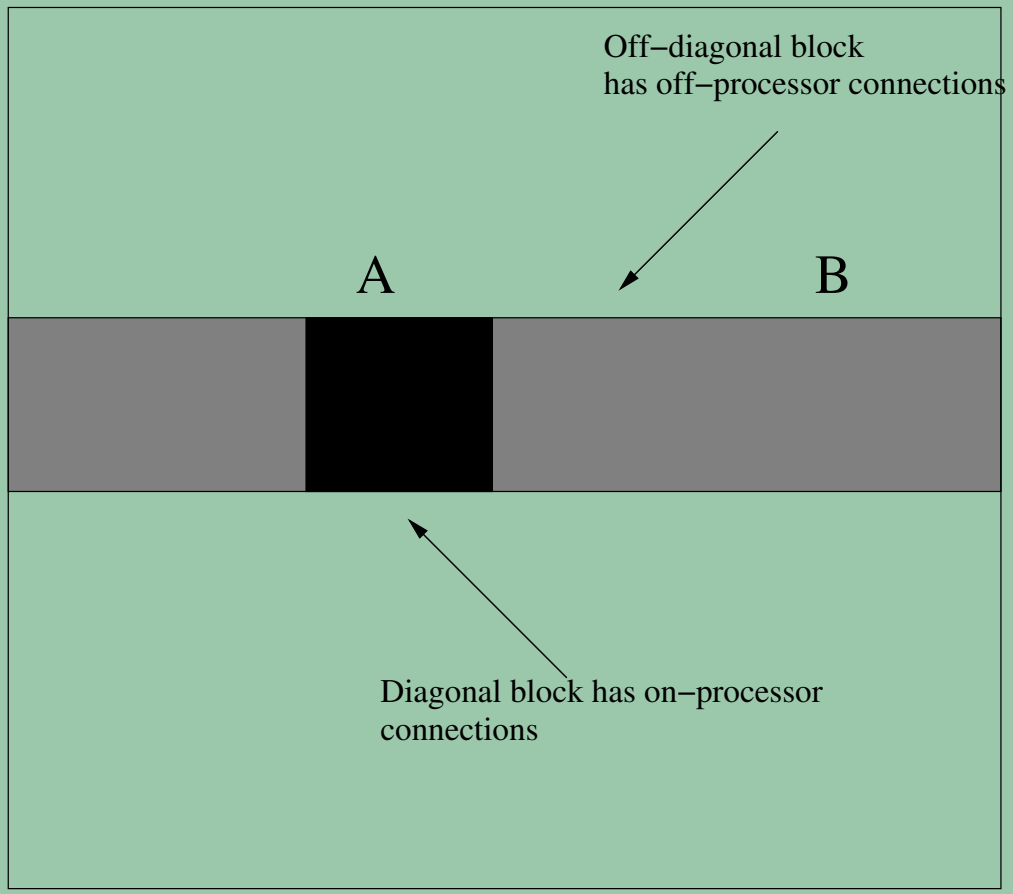

Sequential matrix structure

```
1  MatSeqAIJSetPreallocation
2      (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!

In Fortran use `PETSC_NULL_INTEGER` if not specifying *nnz* array

Parallel matrix structure



(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

TACC

- `m`,`n` local size; `M`,`N` global. Note: If the matrix is square, specify m, n equal, even though distribution by block rows
- `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

```
1 MatMPIAIJSetPreallocation
2   (Mat B,
3    PetscInt d_nz,const PetscInt d_nnz[],
4    PetscInt o_nz,const PetscInt o_nnz[])
```

In Fortran use `PETSC_NULL_INTEGER` if not specifying arrays

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)
```


Querying parallel structure

Matrix partitioned by block rows:

```
1 MatGetSize(Mat mat,PetscInt *M,PetscInt* N);
2 MatGetLocalSize(Mat mat,PetscInt *m,PetscInt* n);
3 MatGetOwnershipRange(Mat A,int *first_row,int *last_row);
```

In query functions, unneeded components can be specified as `PETSC_NULL`.

Fortran: `PETSC_NULL_INTEGER`

Setting values

Set one value:

```
MatSetValue(Mat A,
  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)

Set only one element

```
MatSetValue(A,i,j,&v,INSERT_VALUES);
```

Special case of the general case:

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


Assembling the matrix

Setting elements is independent of parallelism; move elements to proper processor:

```
1  MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
2  MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

Cannot mix inserting/adding values: need to do assembly in between with `MAT_FLUSH_ASSEMBLY`

Exercise 5 (`matvec`)

Pretend that you do not know how the matrix is created. Use `MatGetOwnershipRange` or `MatGetLocalSize` to create a vector with the same distribution, and then compute $y \leftarrow Ax$.

(Part of the code has been disabled with `#if 0`. We will get to that next.)

Getting values (C)

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

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

Note: for inspection only; possibly expensive.

Getting values (F)

```
1    MatGetRow(A,row,ncols,cols,vals,ierr)
2    MatRestoreRow(A,row,ncols,cols,vals,ierr)
```

where *cols*(*maxcols*), *vals*(*maxcols*) are long enough arrays (allocated by the user)

Exercise 6 (`matvec`)

Advanced exercise: create a sequential (uni-processor) vector.
Question: how does the code achieve this? Give it the data of the distributed vector. Use that to compute the vector norm on each process separately.

(Start by removing the `#if 0` and `#endif`.)

Other matrix types

TACC

MATBAIJ : blocked matrices (dof per node)

(see PETSC_DIR/include/petscmat.h)

Dense:

```
1  MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,
2    PetscScalar *data,Mat *A);
3  MatCreateDense(MPI_Comm comm,
4    PetscInt m,PetscInt n,PetscInt M,PetscInt N,
5    PetscScalar *data,Mat *A)
6  fg
```

Data argument optional: PETSC_NULL or PETSC_NULL_SCALAR causes allocation

GPU support

- Create as GPU matrix,
- Otherwise transparent through overloading

```
1 // cudainit.c
2 PetscDeviceType cuda = PETSC_DEVICE_CUDA;
3 ierr = PetscDeviceInitialize(cuda);
4 PetscBool has_cuda;
5 has_cuda = PetscDeviceInitialized(cuda);
```

`VECCUDA`, `MatCreateDenseCUDA`, `MATAIJCUSPARSE`

```
ierr = MatCreate(comm,&A);
#ifdef PETSC_HAVE_CUDA
ierr = MatSetType(A,MATMPIAIJCUSPARSE);
#else
ierr = MatSetType(A,MATMPIAIJ);
#endif
```


Matrix operations

Main operations are matrix-vector:

```
1 MatMult(Mat A,Vec in,Vec out);
2 MatMultAdd
3 MatMultTranspose
4 MatMultTransposeAdd
```

Simple operations on matrices:

```
1 MatNorm
2
3 MatScale
4 MatDiagonalScale
```


Some matrix-matrix operations

```
1  MatMatMult(Mat,Mat,MatReuse,PetscReal,Mat*);
2
3  MatPtAP(Mat,Mat,MatReuse,PetscReal,Mat*);
4
5  MatMatMultTranspose(Mat,Mat,MatReuse,PetscReal,Mat*);
6
7  MatAXPY(Mat,PetscScalar,Mat,MatStructure);
```


Matrix viewers

```
1  MatView(A,PETSC_VIEWER_STDOUT_WORLD);
2
3  row 0: (0, 1)  (2, 0.333333)  (3, 0.25)  (4, 0.2)
4  row 1: (0, 0.5)  (1, 0.333333)  (2, 0.25)  (3, 0.2)
5  ....
```

(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'

- Terminal output: useful for vectors and matrices but also for solver objects.
- Binary output: great for vectors and matrices.
- Viewing can go both ways: load a matrix from file or URL into an object.
- Viewing through a socket, to Matlab or Mathematica, HDF5, VTK.

```
1 PetscViewer fd;
2 PetscViewerCreate( comm, &fd );
3 PetscViewerSetType( fd,PETSCVIEWERVTK );
4 MatView( A,fd );
5 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

```
1  MatShellSetContext(Mat mat,void *ctx);
2  MatShellGetContext(Mat mat,void **ctx);
```

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

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:

```
1  MatGetSubMatrix(Mat mat,
2    IS isrow,IS iscol,PetscInt csize,MatReuse cll,
3    Mat *newmat)
```

Extract multiple single-processor matrices:

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

Collective call, but different index sets per processor

Load balancing

```
1  MatPartitioningCreate
2       (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 $B \approx A^{-1}$

Topics

- All linear solvers in PETSc are iterative, even the direct ones
- Preconditioners
- Fargoing control through commandline options
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests

Iterative solver basics

- KSP object: solver
- set linear system operator
- solve with rhs/sol vector
- this is a default setup

```
KSPCreate(comm,&solver); KSPDestroy(solver);
// set Amat and Pmat
KSPSetOperators(solver,A,B); // usually: A,A
// solve
KSPSolve(solver,rhs,sol);
```

Optional: `KSPSetUp(solver)`

Solver settings

Change default settings by program calls
example: solver type

```
1  KSPSetType(solver,KSPGMRES);
```

Settings can be controlled from the commandline:

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

then options `-ksp....` are parsed.

- type: `-ksp_type gmres -ksp_gmres_restart 20`
- `-ksp_view` for seeing all settings

Convergence

Iterative solvers can fail

- Solve call itself gives no feedback: solution may be completely wrong
- `KSPGetConvergedReason(solver,&reason)` :
  positive is convergence, negative divergence
  `KSPConvergedReasons[reason]` is string
- `KSPGetIterationNumber(solver,&nits)` : after how many iterations did the method stop?

Reason for convergence

Query the solver object:

```
PetscInt its; KSPConvergedReason reason;
PetscCall( KSPGetConvergedReason(solver,&reason) );
PetscCall( KSPGetIterationNumber(solver,&its) );
if (reason<0) {
  PetscPrintf
        (comm,"Failure to converge after %d iterations;
    ↪reason %s\n",
         its,KSPConvergedReasons[reason]);
} else {
  PetscPrintf
        (comm,"Number of iterations to convergence: %d\n",
         its);
}
```

Preconditioners

System $Ax = b$ is transformed:

$$M^{-1}A = M^{-1}b$$

- M is constructed once, applied in every iteration
- If $M = A$: convergence in one iteration
- Tradeoff: M expensive to construct $\Rightarrow$ low number of iterations; construction can sometimes be amortized.
- Other tradeoff: M more expensive to apply and only modest decrease in number of iterations
- Symmetry: A, M symmetric $\not\Rightarrow$ $M^{-1}A$ symmetric, however can be symmetrized by change of inner product
- Can be tricky to make both parallel and efficient

PC basics

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

```
1  // kspcg.c
2  PetscCall( KSPCreate(comm,&solver);
3                PetscCall( KSPSetOperators(solver,A,A) );
4                PetscCall( KSPSetType(solver,KSPCG) ) );
5  {
6    PC prec;
7    PetscCall( KSPGetPC(solver,&prec) );
8    PetscCall( PCSetType(prec,PCNONE) );
9  }
```

- Many choices, some with options: `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)
```

- *B* is basis for preconditioner, need not be *A*
- if *A* or *B* is to be reused, use *NULL*

Types of preconditioners

- Simple preconditioners: Jacobi, SOR, ILU
- Compose simple preconditioners:
  - composing in space: Block Jacobi, Schwarz
  - composing in physics: Fieldsplit
- Global parallel preconditioners: multigrid, approximate inverses

Simple preconditioners

$A = D_A + L_A + U_A$, $M = \ldots$

- None: $M = I$
- Jacobi: $M = D_A$
  - very simple, better than nothing
  - Watch out for zero diagonal elements
- Gauss-Seidel: $M = D_A + L_A$
  - Non-symmetric
  - popular as multigrid smoother
- SOR: $M = \omega^{-1} D_A + L_A$
  - estimating ω often infeasible
- SSOR: $M = (I + (\omega^{-1} D_A)^{-1} + L_A)(\omega^{-1} D_A + U_A)$

Mostly of textbook value.

See next for more state-of-the-art.

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 $y \leftarrow Ax$

ILU

PCICC: symmetric, PCILU: nonsymmetric
many options:

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

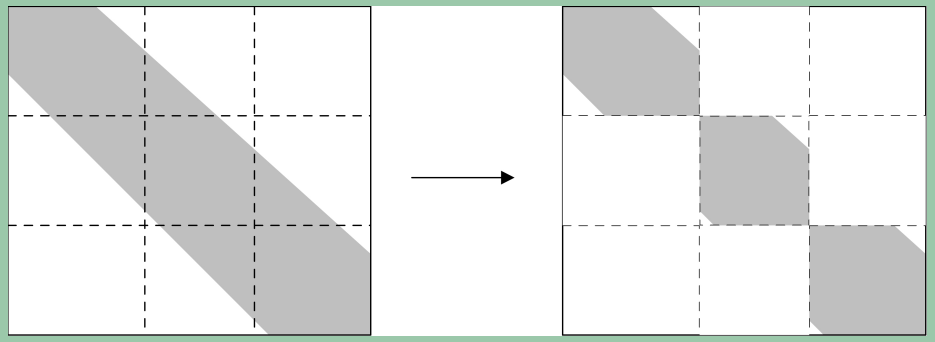
Prevent indefinite preconditioners:

```
1 PCFactorSetShiftType(PC pc,MatFactorShiftType type);
```

value MAT_SHIFT_POSITIVE_DEFINITE et cetera

Factorization preconditioners are sequential
but still useful; see later

Block Jacobi and Additive Schwarz, theory



- Both methods parallel
- Jacobi fully parallel
  Schwarz local communication between neighbours
- Both require sequential local solver: composition with simple precondtitioners
- 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);
```


Exercise 7 (`ksp`)

File `ksp.c` / `ksp.F90` contains the solution of a (possibly nonsymmetric) linear system.

Compile the code and run it. Now experiment with commandline options. Make notes on your choices and their outcomes.

- The code has two custom commandline switch:
  - `-n 123` set the domain size to 123 and therefore the matrix size to 123^2.
  - `-unsymmetry 456` adds a convection-like term to the matrix, making it unsymmetric. The numerical value is the actual element size that is set in the matrix.
- What is the default solver in the code? Run with `-ksp_view`
- Print out the matrix for a small size with `-mat_view`.
- Now out different solvers for different matrix sizes and amounts of unsymmetry. See the instructions in the code.

Exercise 8 (shell)

After the main program, a routine *mymatmult* is declared, which is attached by MatShellSetOperation to the matrix *A* as the means of computing the product MatMult(*A*,*in*,*out*), for instance inside an iterative method.

In addition to the shell matrix *A*, the code also creates a traditional matrix *AA*. Your assignment is to make it so that *mymatmult* computes the product $y \leftarrow A^t A x$.

In C, use MatShellSetContext to attach *AA* to *A* and MatShellGetContext to retrieve it again for use; in Fortran use a common block (or a module) to store *AA*.

The code uses a preconditioner PCNONE. What happens if you run it with option `-pc_type jacobi`?

Monitors and convergence tests

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

Monitors can be set in code, but simple cases:

- `-ksp_monitor`
- `-ksp_monitor_true_residual`

Custom monitors and convergence tests

```
1  KSPMonitorSet(KSP ksp,
2   PetscErrorCode (*monitor)
3         (KSP,PetscInt,PetscReal,void*),
4   void *mctx,
5   PetscErrorCode (*monitordestroy)(void*));
6  KSPSetConvergenceTest(KSP ksp,
7   PetscErrorCode (*converge)
8         (KSP,PetscInt,PetscReal,KSPConvergedReason*,void*),
9   void *cctx,
10  PetscErrorCode (*destroy)(void*))
```

Example of convergence tests

```
1  PetscErrorCode resconverge
2  (KSP solver,PetscInt it,PetscReal res,
3   KSPConvergedReason *reason,void *ctx)
4  {
5    MPI_Comm comm; Mat A; Vec X,R; PetscErrorCode ierr;
6    PetscFunctionBegin;
7    KSPGetOperators(solver,&A,PETSC_NULL,PETSC_NULL);
8    PetscObjectGetComm((PetscObject)A,&comm);
9    KSPBuildResidual(solver,PETSC_NULL,PETSC_NULL,&R);
10   KSPBuildSolution(solver,PETSC_NULL,&X);
11   /* stuff */
12   if (sometest) *reason = 15;
13   else *reason = KSP_CONVERGED_ITERATING;
14   PetscFunctionReturn(0);
```


Advanced options

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

```
1 KSPSetInitialGuessNonzero
2 KSPGMRESSetRestart
3 KSPSetPreconditionerSide
4 KSPSetNormType
```

Many options easier through commandline.

Null spaces

Iterating orthogonal to the null space of the operator:

```
1 MatNullSpace sp;
2 MatNullSpaceCreate /* constant vector */
3     (PETSC_COMM_WORLD,PETSC_TRUE,0,PETSC_NULL,&sp);
4 MatNullSpaceCreate /* general vectors */
5     (PETSC_COMM_WORLD,PETSC_FALSE,5,vecs,&sp);
6 MatSetNullSpace(mat,sp);
```

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

Matrix-free solvers

Shell matrix requires shell preconditioner in `KSPSetOperators`):

```
1  PCSetType(pc,PCSHELL);
2  PCShellSetContext(PC pc,void *ctx);
3  PCShellGetContext(PC pc,void **ctx);
4  PCShellSetApply(PC pc,
5      PetscErrorCode (*apply)(void*,Vec,Vec));
6  PCShellSetSetUp(PC pc,
7      PetscErrorCode (*setup)(void*))
```

similar idea to shell matrices

Alternative: use different operator for preconditioner

Fieldsplit preconditioners

If a problem contains multiple physics, seperate preconditioning can make sense

Matrix block storage: `MatCreateNest`

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

However, it makes more sense to interleave these fields

Fieldsplit use

Easy case: all fields are the same size

```
1 PCSetType(prec,PCFIELDSPLIT);
2 PCFieldSplitSetBlockSize(prec,3);
3 PCFieldSplitSetType(prec,PC_COMPOSITE_ADDITIVE);
```

Subpreconditioners can be specified in code, but easier with options:

```
1 PetscOptionsSetValue
2  ("-fieldsplit_0_pc_type","lu");
3 PetscOptionsSetValue
4  ("-fieldsplit_0_pc_factor_mat_solver_package","mumps");
```

Fields can be named instead of numbered.

Non-strided, arbitrary fields: `PCFieldSplitSetIS()`

Stokes equation can be detected:
`-pc_fieldsplit_detect_saddle_point`

Combining fields multiplicatively: solve

$$\begin{pmatrix} I & \\ A_{10}A_{00}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{00} & A_{01} \\ & A_{11} \end{pmatrix}$$

If there are just two fields, they can be combined by Schur complement

$$\begin{pmatrix} I & \\ A_{10}A_{00}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{00} & A_{01} \\ & A_{11} - A_{10}A_{00}^{-1}A_{01} \end{pmatrix}$$

Fieldsplit example

```
1   KSPGetPC(solver,&prec);
2   PCSetType(prec,PCFIELDSPLIT);
3   PCFieldSplitSetBlockSize(prec,2);
4   PCFieldSplitSetType(prec,PC_COMPOSITE_ADDITIVE);
5   PetscOptionsSetValue
6      ("-fieldsplit_0_pc_type","lu");
7   PetscOptionsSetValue
8      ("-fieldsplit_0_pc_factor_mat_solver_package","mumps");
9   PetscOptionsSetValue
10     ("-fieldsplit_1_pc_type","lu");
11  PetscOptionsSetValue
12     ("-fieldsplit_1_pc_factor_mat_solver_package","mumps");
```

Global preconditioners: MG

```
PCSetType(PC pc,PCMG);
PCMGSetLevels(pc,int levels,MPI Comm *comms);
PCMGSetType(PC pc,PCMGType mode);
PCMGSetCycleType(PC pc,PCMGCycleType ctype);
PCMGSetNumberSmoothUp(PC pc,int m);
PCMGSetNumberSmoothDown(PC pc,int n);
PCMGGetCoarseSolve(PC pc,KSP *ksp);
PCMGSetInterpolation(PC pc,int level,Mat P); and
PCMGSetRestriction(PC pc,int level,Mat R);
PCMGSetResidual(PC pc,int level,PetscErrorCode
    (*residual)(Mat,Vec,Vec,Vec),Mat mat);
```

Global preconditioners: Hypre

- Hypre is a package like PETSc
- selling point: fancy preconditioners
- Install with `--with-hypre=yes --download-hypre=yes`
- then use `-pc_type hypre -pc_hypre_type parasails/boomeramg/euclid/pilut`

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`:

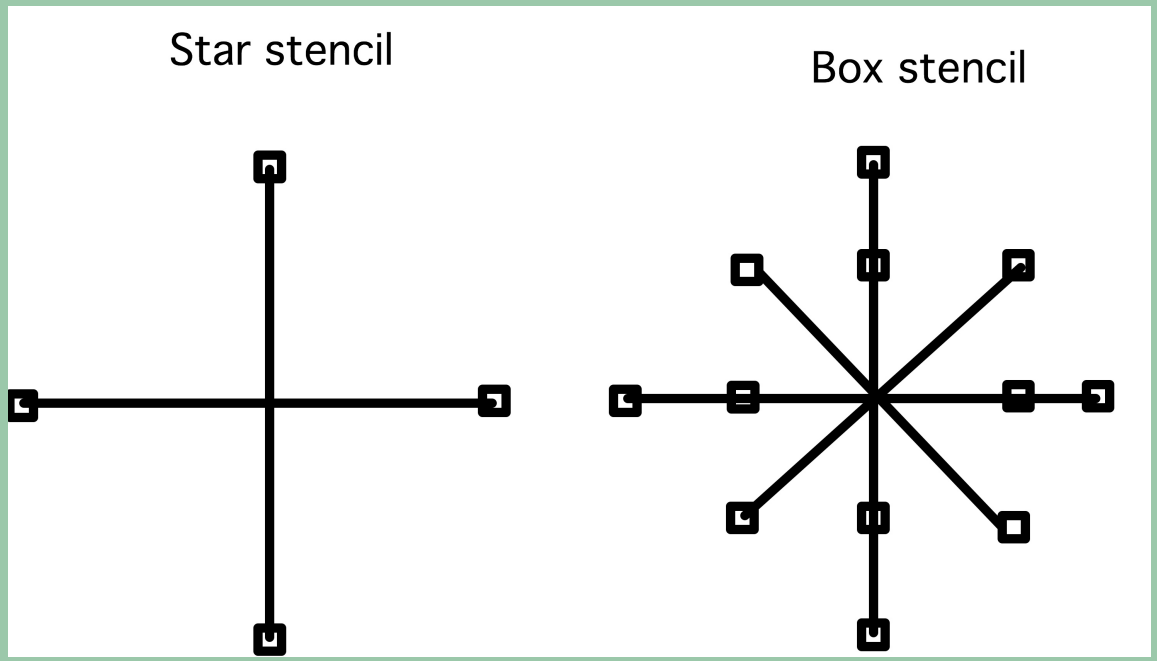
```
myprog -pc_type lu -ksp_type preonly \
    -pc_factor_mat_solver_package mumps
```

Grid manipulation

Regular grid: DMDA

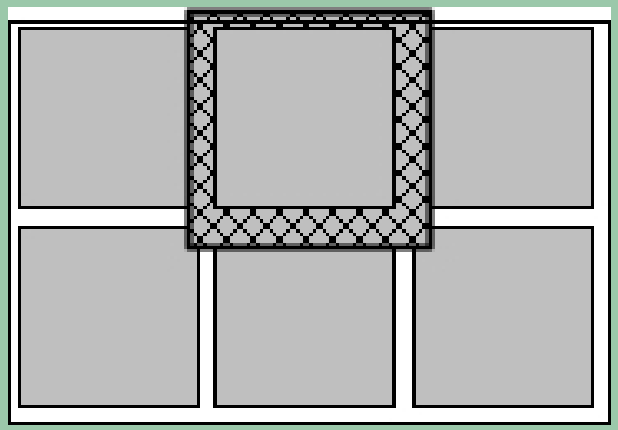
DMDAs are for storing vector field, not matrix.

Support for different stencil types:



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

```
1  DMDACreate2d(comm, bndx,bndy, type, M, N, m, n,
2    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)

Grid info

Divide 100×100 grid over 4 processes, stencil width$= 1$:

Code:

```
1  // dmrhs.c
2  DM grid;
3  PetscCall( DMDACreate2d
4               ( comm,
5                 DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
6                 DMDA_STENCIL_STAR,
7                 100,100,
8                 PETSC_DECIDE,PETSC_DECIDE,
9                 1,
10                1,
11                NULL,NULL,
12                &grid
13                ) );
14 PetscCall( DMSetFromOptions(grid) );
15 PetscCall( DMSetUp(grid) );
16 PetscCall( DMViewFromOptions(grid,NULL,"-dm_view") );
```

Output:

```
1  ld: warning: dylib
       ↪(/Users/eijkhout/Inst
       ↪was built for
       ↪newer macOS
       ↪version
       ↪(11.5) than
       ↪being linked
       ↪(11.0)
2  [0] Local = 0-50 x
       ↪0-50, halo =
       ↪0-51 x 0-51
3  [1] Local = 50-100
       ↪x 0-50, halo
       ↪= 49-100 x
       ↪0-51
```

Associated vectors

- Global vector: based on grid partitioning.
- Local vector: including halo regions

```
1  Vec ghostvector;
2  PetscCall( DMGetLocalVector(grid,&ghostvector) );
3  PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector)
       ↪);
4  PetscReal **xyarray,**gh;
5  PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
6  PetscCall( DMDAVecGetArray(grid,ghostvector,&gh) );
7  // computation on the arrays
8  PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
9  PetscCall( DMDAVecRestoreArray(grid,ghostvector,&gh) );
10 PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy)
       ↪);
11 PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```

Grid info

```
1  typedef struct {
2    PetscInt      dim,dof,sw;
3    PetscInt      mx,my,mz;     /* grid points in x,y,z */
4    PetscInt      xs,ys,zs;     /* starting point, excluding
       ↪ghosts */
5    PetscInt      xm,ym,zm;     /* grid points, excluding ghosts
       ↪*/
6    PetscInt      gxs,gys,gzs; /* starting point, including
       ↪ghosts */
7    PetscInt      gxm,gym,gzm; /* grid points, including ghosts
       ↪*/
8    DMBoundaryType   bx,by,bz;     /* type of ghost nodes */
9    DMDAStencilType  st;
10   DM               da;
11 } DMDALocalInfo;
```

Range over local subdomain

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    // actions on point i,j
  }
}
```

Arrays of vectors

```
1  Vec ghostvector;
2  PetscCall( DMGetLocalVector(grid,&ghostvector) );
3  PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector)
       ↪);
4  PetscReal **xyarray,**gh;
5  PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
6  PetscCall( DMDAVecGetArray(grid,ghostvector,&gh) );
7  // computation on the arrays
8  PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
9  PetscCall( DMDAVecRestoreArray(grid,ghostvector,&gh) );
10 PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy)
       ↪);
11 PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```


Operating on arrays

```
1  for (int j=info.ys; j<info.ys+info.ym; j++) {
2    for (int i=info.xs; i<info.xs+info.xm; i++) {
3          if (info.gxs<info.xs && info.gys<info.ys)
4            if (i-1>=info.gxs && i+1<=info.gxs+info.gxm &&
5                j-1>=info.gys && j+1<=info.gys+info.gym )
6              xyarray[j][i] =
7                ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] +
     ↪gh[j+1][i] )
8                /4.;
```

Associated matrix

Matrix that has knowledge of the grid:

```
DMSetUp(DM grid);
DMCreateMatrix(DM grid,Mat *J)
```

Set matrix values based on stencil:

```
MatSetValuesStencil(Mat mat,
  PetscInt m,const MatStencil idxm[],
  PetscInt n,const MatStencil idxn[],
  const PetscScalar v[],InsertMode addv)
```

(ordering of row/col variables too complicated for `MatSetValues`)

Set values by stencil

```
1  // grid2d.c
2  for (int j=info.ys; j<info.ys+info.ym; j++) {
3    for (int i=info.xs; i<info.xs+info.xm; i++) {
4      MatStencil  row,col[5];
5      PetscScalar v[5];
6      PetscInt    ncols = 0;
7      row.j        = j; row.i = i;
8      /**** local connection: diagonal element ****/
9      col[ncols].j = j; col[ncols].i = i; v[ncols++] = 4.;
10     /* boundaries: top and bottom row */
11     if (i>0)          {col[ncols].j = j;   col[ncols].i = i-1;
       ↪v[ncols++] = -1.;}
12     if (i<info.mx-1) {col[ncols].j = j;   col[ncols].i = i+1;
       ↪v[ncols++] = -1.;}
13     /* boundary left and right */
14     if (j>0)          {col[ncols].j = j-1; col[ncols].i = i;
       ↪v[ncols++] = -1.;}
15     if (j<info.my-1) {col[ncols].j = j+1; col[ncols].i = i;
       ↪v[ncols++] = -1.;}
16
```


DMPlex

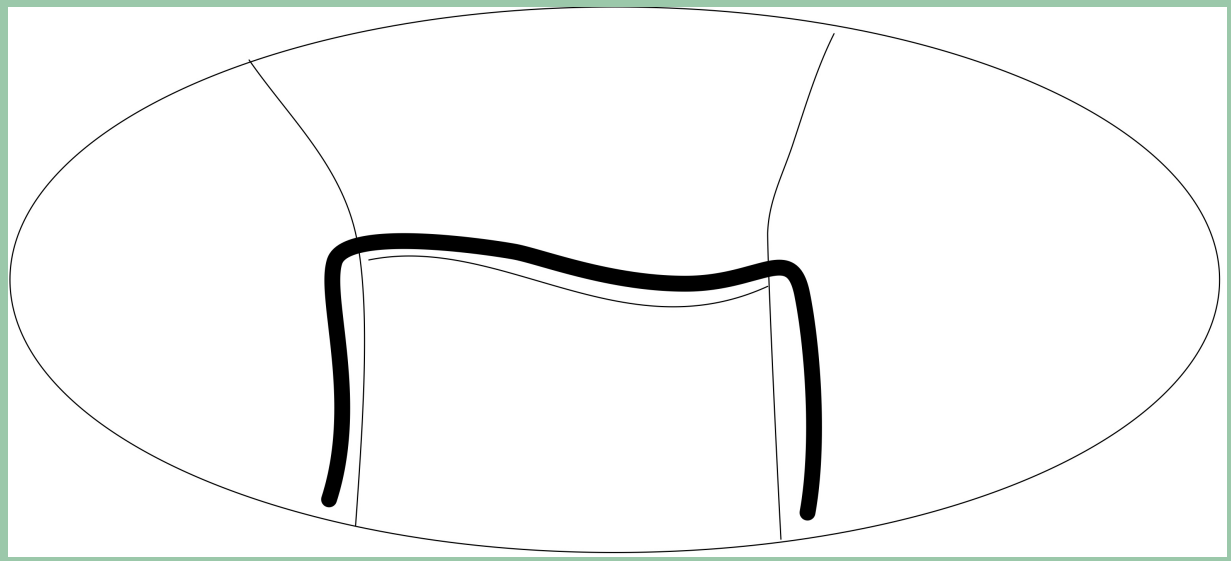
Support for unstructured grids and node/edge/cell relations.

This is complicated and under-documented.

IS and VecScatter: irregular grids

Irregular data movement

Example: collect distributed boundary onto a single processor (this happens in the matrix-vector product)



Problem: figuring out communication is hard, actual communication is cheap

VecScatter

Preprocessing: determine mapping between input vector and output:

```
VecScatterCreate(Vec,IS,Vec,IS,VecScatter*)
// also Destroy
```

Application to specific vectors:

```
VecScatterBegin(VecScatter,
  Vec,Vec, InsertMode mode, ScatterMode direction)
VecScatterEnd  (VecScatter,
  Vec,Vec, InsertMode mode, ScatterMode direction)
```


IS: index set

Index Set is a set of indices

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

ISDestroy(is);
```

Use `MPI_COMM_SELF` most of the time.

Various manipulations: *ISSum*, *ISDifference*, *ISInvertPermutations* et cetera.
Also `ISGetIndices` / `ISRestoreIndices` / *ISGetSize*

Example: split odd and even

Input:

```
Process [0]
0.
1.
2.
3.
4.
5.
```

```
Process [1]
6.
7.
8.
9.
10.
11.
```

Output:

```
Process [0]
0.
2.
4.
6.
8.
10.
```

```
Process [1]
1.
3.
5.
7.
9.
11.
```

index sets for this example

```
1  // oddeven.c
2  IS oddeven;
3  if (procid==0) {
4    PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
5  } else {
6    PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
7  }
```


scatter for this example

```
1 VecScatter separate;
2 PetscCall( VecScatterCreate
3   (in,oddeven,out,NULL,&separate) );
4 PetscCall( VecScatterBegin
5   (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
6 PetscCall( VecScatterEnd
7   (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```


Exercise 9 (oddeven)

Now alter the `IS` objects so that the output becomes:

```
Process [0]
10.
8.
6.
4.
2.
0.
```

```
Process [1]
11.
9.
7.
5.
3.
1.
```


Example: simulate allgather

```
/* create the distributed vector with one element per processor
ierr = VecCreate(MPI_COMM_WORLD,&global);
ierr = VecSetType(global,VECMPI);
ierr = VecSetSizes(global,1,PETSC_DECIDE);

/* create the local copy */
ierr = VecCreate(MPI_COMM_SELF,&local);
ierr = VecSetType(local,VECSEQ);
ierr = VecSetSizes(local,ntids,ntids);
```


```
IS indices;
ierr = ISCreateStride(MPI_COMM_SELF,ntids,0,1, &indices);
/* create a scatter from the source indices to target */
ierr = VecScatterCreate
  (global,indices,local,indices,&scatter);
/* index set is no longer needed */
ierr = ISDestroy(&indices);
```


Example: even and odd indices

```
1  // oddeven.c
2  IS oddeven;
3  if (procid==0) {
4    PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
5  } else {
6    PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
7  }
```


scattering odd and even

```
1 VecScatter separate;
2 PetscCall( VecScatterCreate
3   (in,oddeven,out,NULL,&separate) );
4 PetscCall( VecScatterBegin
5   (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
6 PetscCall( VecScatterEnd
7   (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```


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:

```
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: function evaluation

```
1 PetscErrorCode (*FunctionEvaluation)(SNES,Vec,Vec,void*);
2 VecCreate(PETSC_COMM_WORLD,&r);
3 SNESSetFunction(snes,r,FunctionEvaluation,*ctx);
```

SNES specification: jacobian evaluation

```
1 PetscErrorCode (*FormJacobian)(SNES,Vec,Mat,Mat,void*);
2 MatCreate(PETSC_COMM_WORLD,&J);
3 SNESSetJacobian(snes,J,J,FormJacobian,*ctx);
```


SNES solution

```
1 SNESSolve(snes,/* rhs= */ PETSC_NULL,x)
2 SNESGetConvergedReason(snes,&reason)
3 SNESGetIterationNumber(snes,&its)
```

Example: two-variable problem

Define a context

```
typedef struct {
  Vec xloc,rloc; VecScatter scatter; } AppCtx;

/* User context */
AppCtx          user;

/* Work vectors in the user context */
VecCreateSeq(PETSC_COMM_SELF,2,&user.xloc);
VecDuplicate(user.xloc,&user.rloc);

/* Create the scatter between the global and local x */
ISCreateStride(MPI_COMM_SELF,2,0,1,&idx);
VecScatterCreate(x,idx,user.xloc,idx,&user.scatter);
```


n the user function:

```
PetscErrorCode FormFunction
    (SNES snes,Vec x,Vec f,void *ctx)
{
  VecScatterBegin(user->scatter,
    x,user->xloc,INSERT_VALUES,SCATTER_FORWARD); // & End
  VecGetArray(xloc,&xx);CHKERRQ(ierr);
  VecSetValue
    (f,0,/* something with xx[0]) & xx[1] */,
     INSERT_VALUES);
  VecRestoreArray(x,&xx);
  PetscFunctionReturn(0);
}
```

Jacobian calculation through finite differences

Jacobian calculation is difficult. It can be approximated through finite differences:

$$J(u)v \approx \frac{f(u+hv) - f(u)}{h}$$

```
MatCreateSNESMF(snes,&J);
SNESSetJacobian
 (snes,J,J,MatMFFDComputeJacobian,(void*)&user);
```


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 converngence: `-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)
```

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

```
1 PetscLogStagePush(int stage); /* 0 <= stage <= 9 */
2 PetscLogStagePop();
3 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)
- Better than `PetscMalloc`: `PetscMalloc1` aligned to `PETSC_MEMALIGN`