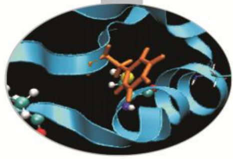
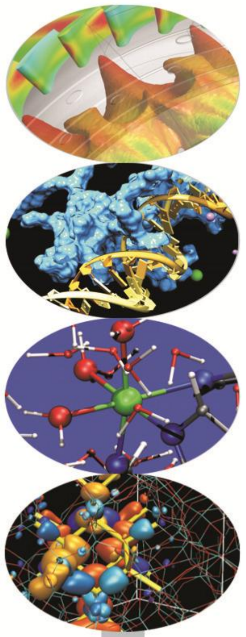
SuperComputing Applications and Innovation



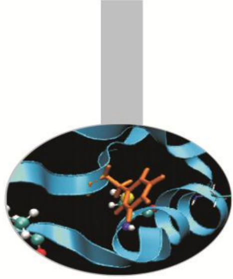
Introduction to scientific computing with PETSc

Portable, Extensible Toolkit for Scientific Computation

Simone Bnà – s.bn@cineca.it
SuperComputing Applications and Innovation Department

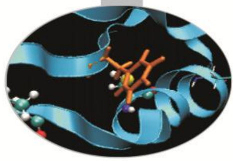
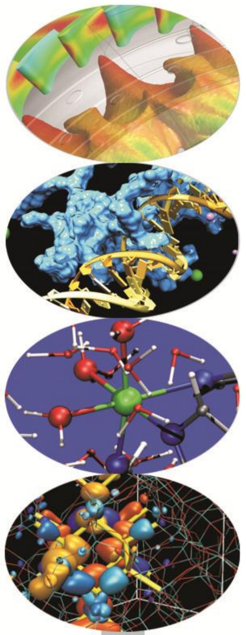


Outline

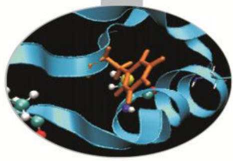


- Introduction
- Getting Started
- VEC and Mat
- KSP, SNES and TS
- Grid Manipulation and Discretization
- Debugging and Profiling

Introduction

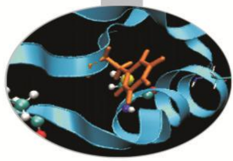
PETSc in a nutshell



PETSc – Portable, Extensible Toolkit for Scientific Computation

Is a suite of data structures and routines for the scalable (parallel) solution of scientific applications mainly modelled by partial differential equations.

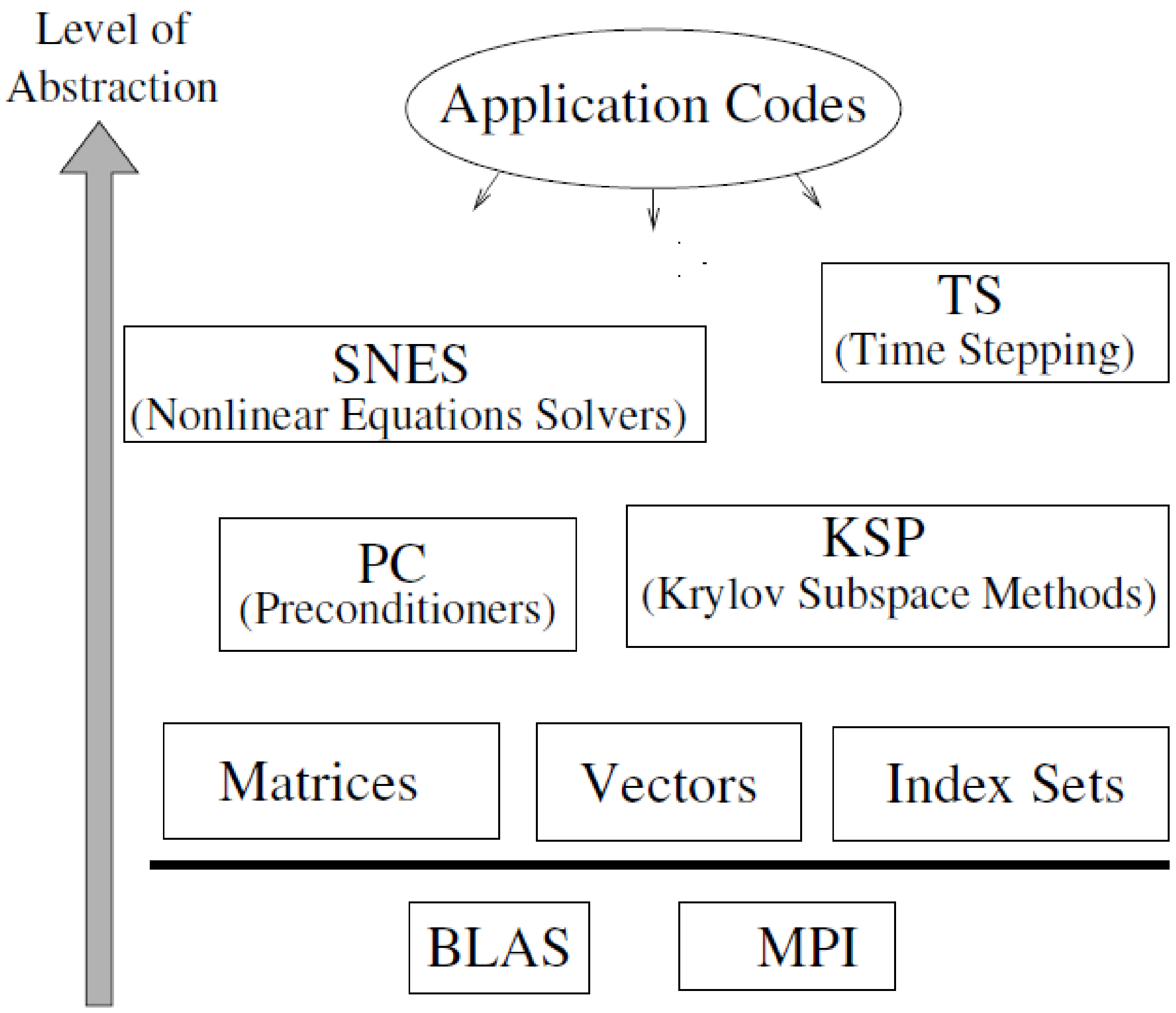
- Serial and parallel computation
- Linear and NonLinear solvers
- Support for Finite Difference and Finite Elements PDE discretizations
- Structured and Unstructured topologies

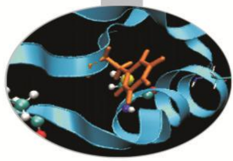


What is in PETSc?

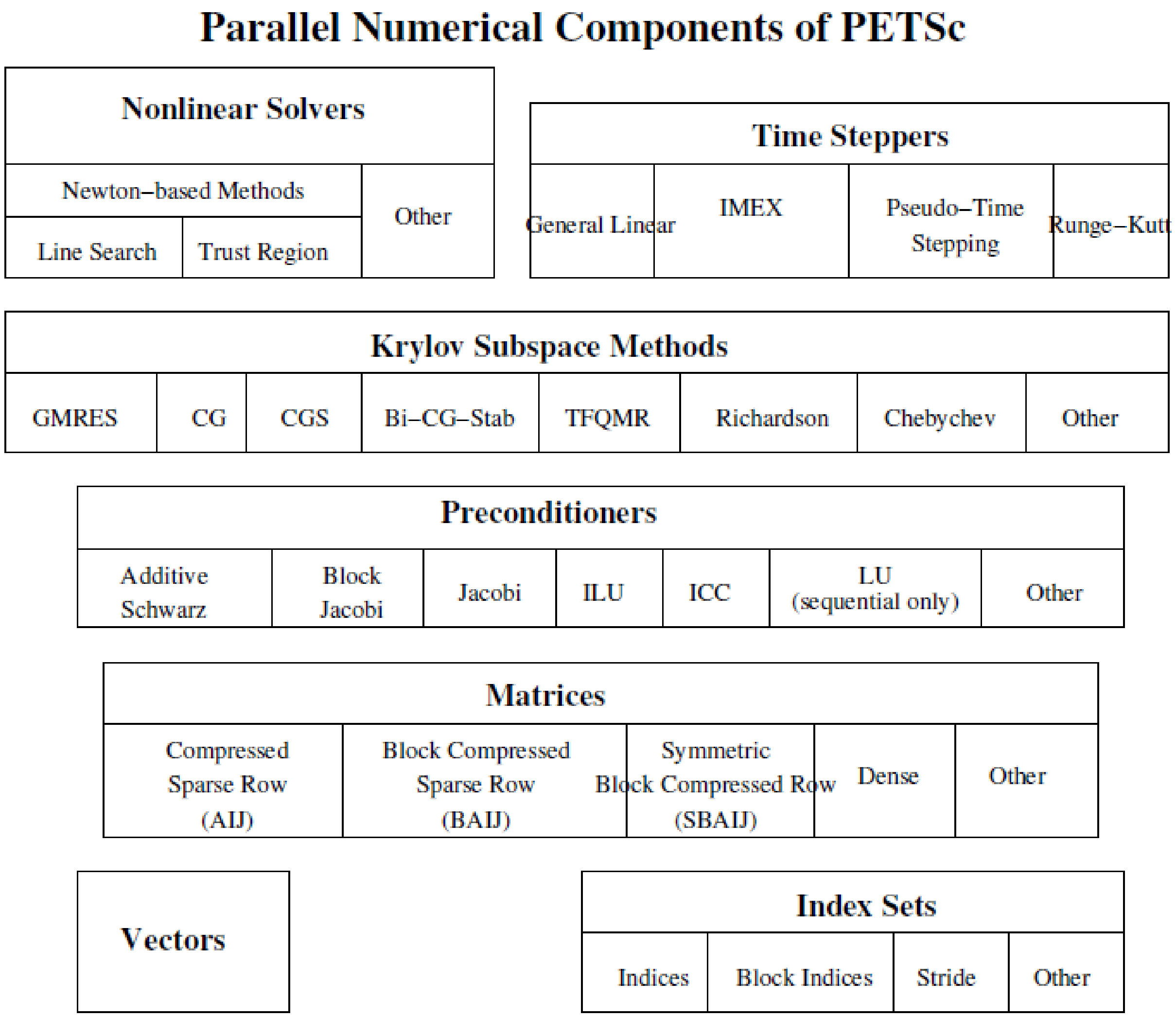
- Linear system solvers (sparse/dense, iterative/direct)
- Non linear system solvers
- Tools for distributed vectors and matrices
- Support for debugging, profiling and graphical output

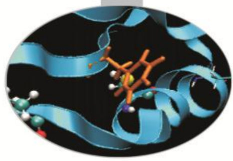
PETSc class hierarchy





PETSc numerical components

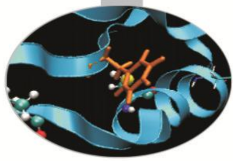




External Packages

- Dense linear algebra: Scalapack, Plapack
- Sparse direct linear solvers: Mumps, SuperLU, SuperLU_dist
- Grid partitioning software: Metis, ParMetis, Jostle, Chaco, Party
- ODE solvers: PVODE
- Eigenvalue solvers (including SVD): SLEPc
- Optimization: TAO

SuperComputing Applications and Innovation



PETSc design concepts

Goals

- Portability: available on many platforms, basically anything that has MPI
- Performance
- Scalable parallelism
- Flexibility: easy switch among different implementations

Approach

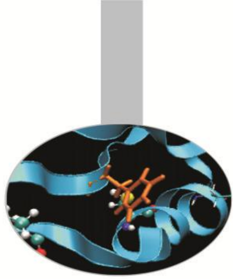
- Object Oriented Delegation Pattern : many specific implementations of the same object
- Shared interface (overloading):
  MATMult(A,x,y); // y <- A x
  same code for sequential, parallel, dense, sparse
- Command line customization

Drawback

- Nasty details of the implementation hidden

SuperComputing Applications and Innovation

PETSc object oriented model



- (Almost) all PETSc objects are essentially **delegator objects**
- From Wikipedia: "...*an object, instead of performing one of its stated tasks, delegates that task to an associated helper object*..."

  http://en.wikipedia.org/wiki/Delegation_pattern

- Example with a XXX object

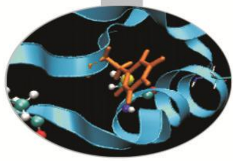
```
#include <petscxxx.h> //Includes the public interface for XXX and other stuff
PetscXXX xxx;
XXXCreate(....,&xxx); //Initializes the XXX object (no implementation yet)
XXXSetType(xxx,ANY_XXX_TYPE); //DELEGATION: Sets specific implementation
XXXSetOption(xxx,ANY_XXX_OPTION,XXX_OPTION_VALUE); //Sets options in DB
XXXAnyCustom(xxx,...); //Any XXX customization available through the interface
XXXSetFromOptions(xxx); //Allows options and command line customization
XXXSetUp(xxx); //Calls specific setup (not all objects need it)
```

- `XXXSetType` calls the specific creation routine `XXXCreate_ANYXXXTYPE(...)`.
- If `XXXSetType` is called at a later time, the old delegate is freed and `xxx` can be reused with a different low-level implementation.
- `XXXSetUp`, if needed, closes the setup procedure: `xxx` can then be used.
- Users can register their own delegates/classes using

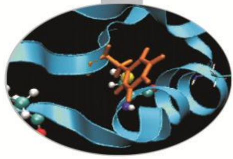
  `XXXRegister(...,XXXCreate_MYTYPE)`

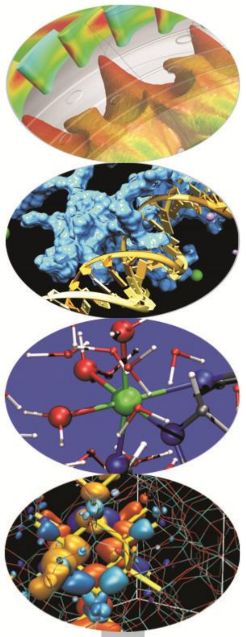
SuperComputing Applications and Innovation

PETSc and Parallelism



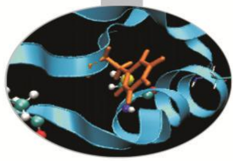
- All objects in PETSc are defined on a communicator; they can only interact if on the same communicator
- PETSc is layered on top of MPI: you do not need to know much MPI when you use PETSc
- Parallelism through MPI (Pure MPI). No support for threaded code that made Petsc calls (OpenMP, Pthreads) since PETSc is not thread-safe.
- PETSc supports to have individual threads (OpenMP or others) to each manage their own (sequential) PETSc objects (and each thread can interact only with its own objects).
- Transparent: same code works sequential and parallel.
- User can use shared memory programming





Getting Started

PETSc from a user perspective



- Home page

  http://www.mcs.anl.gov/petsc/index.html
- User manual

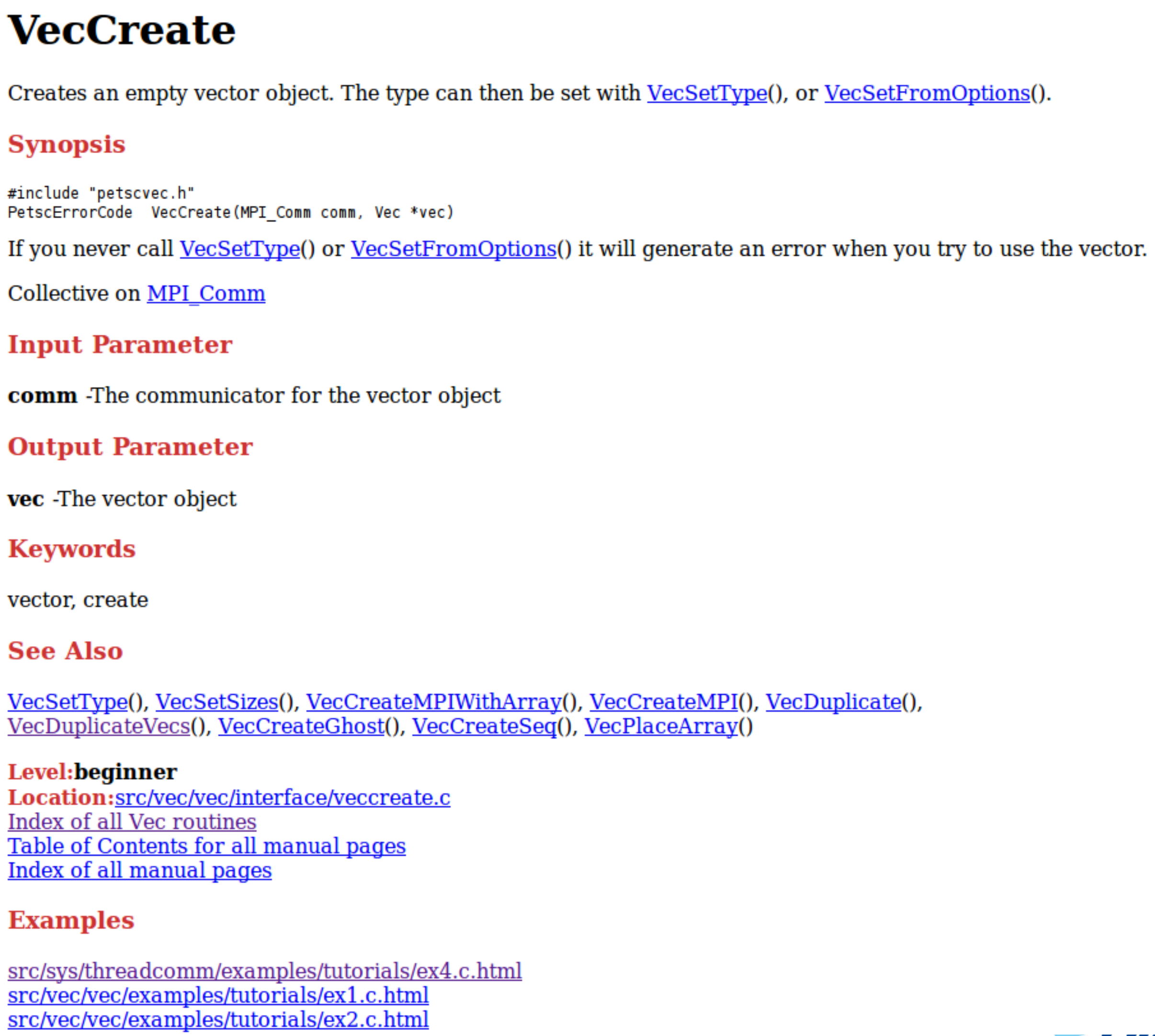
  http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf
- Public functions for XXX class (Vec, Mat, KSP, …) accessible at

  http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/XXX/index.html
- Each class has its own set of tutorials which can be compiled and run

  USE THEM TO LEARN HOW TO DEVELOP WITH PETSc!
- Always use a debug version of PETSc when developing.
- No need to download and install supported external packages separately: PETSc will do this for you if the needed packages are requested at configure time.
- An example:

```
$ ./configure --download-mpich=1 --download-mumps=1
```

PETSc from a user perspective



VecCreate

Creates an empty vector object. The type can then be set with VecSetType(), or VecSetFromOptions().

Synopsis

```
#include "petscvec.h"
PetscErrorCode  VecCreate(MPI_Comm comm, Vec *vec)
```

If you never call VecSetType() or VecSetFromOptions() it will generate an error when you try to use the vector.

Collective on MPI_Comm

Input Parameter

comm -The communicator for the vector object

Output Parameter

vec -The vector object

Keywords

vector, create

See Also

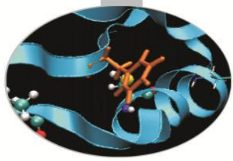
VecSetType(), VecSetSizes(), VecCreateMPIWithArray(), VecCreateMPI(), VecDuplicate(), VecDuplicateVecs(), VecCreateGhost(), VecCreateSeq(), VecPlaceArray()

Level:beginner
Location:src/vec/vec/interface/veccreate.c
Index of all Vec routines
Table of Contents for all manual pages
Index of all manual pages

Examples

src/sys/threadcomm/examples/tutorials/ex4.c.html
src/vec/vec/examples/tutorials/ex1.c.html
src/vec/vec/examples/tutorials/ex2.c.html

Writing PETSc programs: initialization and finalization



```
PetscInitialize(int *argc, char ***args, const char
options_file[], const char help_string[])
```

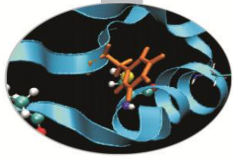
- Setup of static data
- Registers all PETSc specific implementations (of all classes)
- Setup of services (logging, error-handling, profiling)
- Setup of MPI (if it is not already been initialized)

```
PetscFinalize()
```

- Calculates logging summary
- Checks for memory leaks (already allocated mem, if required)
- Finalizes MPI (if `PetscInitialize()` began MPI)
- Shutdowns all PETSc services

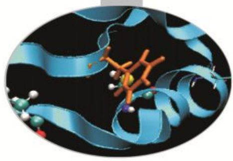
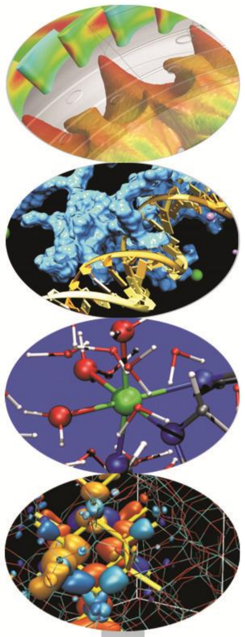
A simple hello_world.c program (C language)

```
#include "petscsys.h"

int main(int argc,char **args) {
  PetscErrorCode ierr;
  PetscMPIInt    rank;

  ierr = PetscInitialize(&argc, &args,(char *)0, NULL);CHKERRQ(ierr);
  ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_SELF,
                     "Hello by process %d!\n",rank);CHKERRQ(ierr);
  ierr = PetscFinalize();
  return 0;
}
```

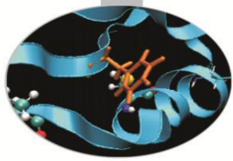


A simple hello_world.f90 program (Fortran language)

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

program main

PetscErrorCode :: ierr
PetscMPIInt :: rank
character(len=6)  :: num
character(len=30) :: hello

call PetscInitialize( PETSC_NULL_CHARACTER,ierr );CHKERRQ(ierr)
call MPI_Comm_rank( PETSC_COMM_WORLD, rank, ierr );CHKERRQ(ierr)
write(num,*) rank
hello = 'Hello from process '//num
call PetscPrintf( PETSC_COMM_SELF, hello//achar(10), ierr );CHKERRQ(ierr)
call PetscFinalize(ierr)

end program
```

PETSc Vectors



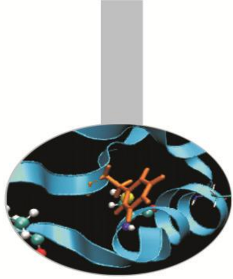
Vectors

What are PETSc vectors?

- Represent elements of a vector space over a field (e.g. R^n)
- Usually they store field solutions and right-hand sides of PDE
- Vector elements are **PetscScalars** (there are no vectors of integers)
- Each process locally owns a subvector of contiguously numbered global indices

Features

- Vector types: **STANDARD** (**SEQ** on one process and **MPI** on several), **PTHREAD**, **VIENNACL** and **CUSP**
- Supports all vector space operations
  - `VecDot(),VecNorm(),VecScale(), …`
- Also unusual ops, like e.g. `VecSqrt(),VecReciprocal()`
- Hidden communication of vector values during assembly
- Communications between different parallel vectors



Crate and Destroy a Vector

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

`VecCreate(MPI_Comm comm, Vec *v)`

– Automatically generates the appropriate vector type (sequential or parallel) over all processes in `comm`

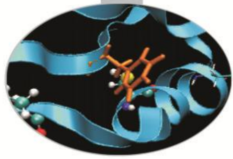
`VecDestroy(Vec *x)`

– Destroys the Vec object

```
/* C */

Vec V;

ierr = VecCreate(MPI_COMM_SELF,&V); CHKERRQ(ierr);

ierr = VecDestroy(&V); CHKERRQ(ierr);
```

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

More about vectors



A vector object is not completely created in one call:

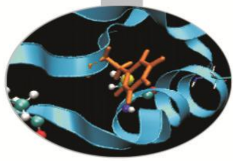
```
VecSetSizes(Vec v, PetscInt m, PetscInt M)
```

- Sets local and global sizes

Other ways of creating a vector (by duplication):

```
VecDuplicate(Vec old, Vec *new)
```

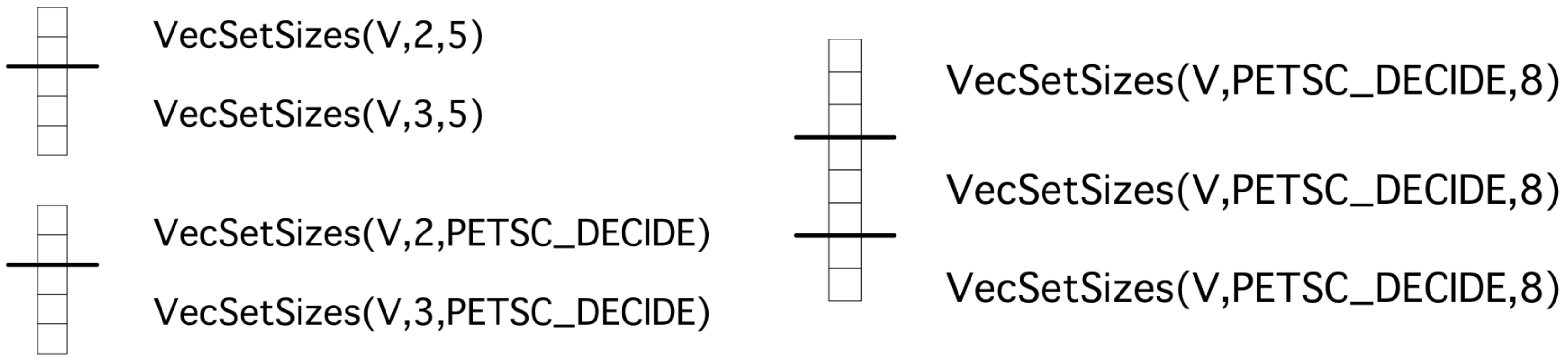
- Duplicates the vector (doesn't copy values)



Vector Parallel layout

```
VecSetSizes(Vec v, PetscInt m, PetscInt M)
```

Global size M or local size m can be specified as **PETSC_DECIDE** but cannot be both. If one processor calls this function with M equal to **PETSC_DECIDE** then all processors must, otherwise the program will hang.


VecSetSizes(V,2,5)
VecSetSizes(V,3,5)
VecSetSizes(V,2,PETSC_DECIDE)
VecSetSizes(V,3,PETSC_DECIDE)
VecSetSizes(V,PETSC_DECIDE,8)
VecSetSizes(V,PETSC_DECIDE,8)
VecSetSizes(V,PETSC_DECIDE,8)

Setting values

VecSet(Vec x, PetscScalar v)

- set vector to constant value

VecSetValue(Vec x, PetscInt idx, PetscScalar v,
InsertMode mode)

- Set a single entry into a vector

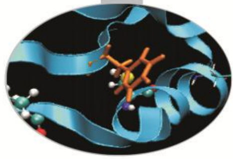
VecSetValues(Vec x, PetscInt n, PetscInt *idx,
PetscScalar *v, InsertMode mode)

- Insert or add values in certain locations of a vector (global indexing!)

```
\* C *\
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);
```

```
!Fortran
i = 1; v = 3.14;
call VecSetValues(x,1,i,v,INSERT_VALUES,ierr,e)

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

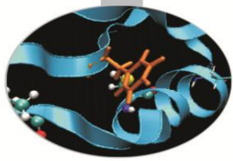
Vector assembly



```
VecAssemblyBegin(Vec x)
…
VecAssemblyEnd(Vec x)
```

A **three step process**

- `VecSetValues` can be called as many times as the user wants to tell PETSc what values are to be inserted (or added to existing ones) and where
- `VecAssseblyBegin` starts communications to ensure that values end up where needed (allow other operations, such as some independent computation, to proceed).
- `VecAssemblyEnd` completes the communication



Working with local vectors

Setting values is done without user access to the stored data.
Getting values is often not necessary since many operations provided.
However, sometimes is more efficient to directly access the local internal data array of a PETSc Vec(local only).

`VecGetArray(Vec x, PetscScalar *[])`

- Access the internal array

`VecRestoreArray(Vec x, PetscScalar *[])`

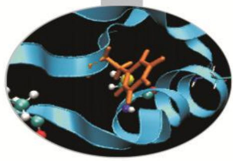
- You must return the array to PETSc when you have done computing with local data

For most common uses (e.g. standard PETSc vectors), these routines are inexpensive and **do not involve** a copy of local data. They simply returns a pointer to a contiguous array that contains this processor's portion of the vector data. If the underlying vector data is not stored in a contiguous array this routine will copy the data to a contiguous array and return a pointer to that.

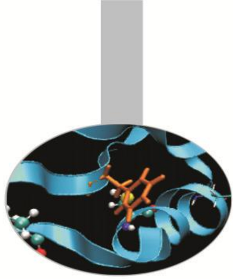
Working with local vectors

```
/* C */
...
PetscScalar * xx_v;
...
ierr = VecGetArray(x,& xx_v);
a = xx_v[3];
ierr = VecRestoreArray(x,xx_v);
...
```

```
!Fortran
...
PetscScalar, pointer :: xx_v(:)
...
call VecGetArrayF90(x,xx_v,ierr)
a = xx_v(3)
call VecRestoreArrayF90(x,xx_v,ierr)
...
```



Other basic Vector interface APIs



`VecSetType(Vec v, VecType type)`

– Sets vector type (defines the delegated object)

`VecSetFromOptions(Vec v)`

– Configures the vector from the options database

`VecGetSize(Vec v, PetscInt *size)`

– Gets global size of v

`VecGetLocalSize(Vec v, PetscInt *size)`

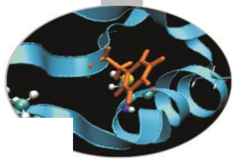
– Gets local size of v

`VecView(Vec x, PetscViewer v)`

– Prints the content of the vector using the viewer object

`VecCopy(Vec x, Vec y)`

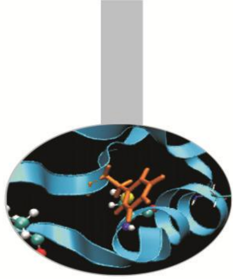
– Copies vector values



Numerical vector operations

Function Name	Operation
VecAXPY(Vec y,PetscScalar a,Vec x);	$y = y + a * x$
VecAYPX(Vec y,PetscScalar a,Vec x);	$y = x + a * y$
VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a);	$x = a * x$
VecDot(Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot(Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm(Vec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
VecSum(Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
VecPointwiseDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
VecMDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax(Vec x, int *idx, PetscReal *r);	$r = \max x_i$
VecMin(Vec x, int *idx, PetscReal *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Vec x,PetscScalar s);	$x_i = s + x_i$
VecSet(Vec x,PetscScalar alpha);	$x_i = \alpha$

Vector - Example 1



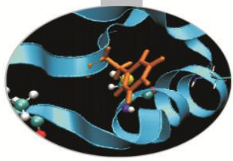
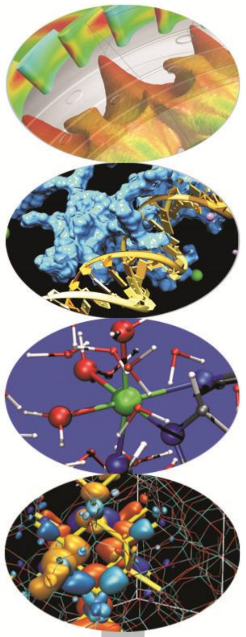
```
#include "petscvec.h"
...
Vec x;
PetscInt i,N;
PetscMPIInt rank;
PetscScalar value=1.0;
PetscErrorCode ierr;
...
ierr = VecGetSize(x, &N);CHKERRQ(ierr);  /* Global size */
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr)
if (rank == 0) { /* Only rank 0 sets all values into the vector */
  for (i=0; i<N; i++) {
    ierr = VecSetValue(x,i,value,INSERT_VALUES);CHKERRQ(ierr);
  }
}
/* data is distributed to the other processes */
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
/* the vector can then be used */
```

Vector - Example 2

```
#include "petscvec.h"
...
Vec x;
PetscInt i,low,high;
PetscScalar value=1.0;
PetscErrorCode ierr;
...
ierr = VecGetOwnershipRange(x, &low, &high);CHKERRQ(ierr);
for (i=low; i<high; i++) { /* each process fill its own part */
  ierr = VecSetValue(x, i, value, INSERT_VALUES);CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(x);CHKERRQ(ierr);
ierr = VecAssemblyEnd(x);CHKERRQ(ierr);
/* the vector can then be used */
```

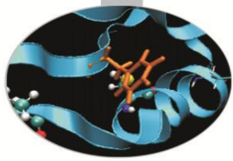
Vector - Example 3

```
#include “petscvec.h”
...
Vec vec;
PetscMPIInt rank;
PetscScalar *avec;
...
ierr = VecCreate(PETSC_COMM_WORLD,&vec);CHKERRQ(ierr);
ierr = VecSetSizes(vec,PETSC_DECIDE,100);CHKERRQ(ierr);
ierr = VecSetType(vec,VECSTANDARD);CHKERRQ(ierr);
...
ierr = VecGetArray(vec, &avec);CHKERRQ(ierr);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_SELF,“First element of local array for rank
%d is %f\n”,rank,avec[0]);CHKERRQ(ierr);
ierr = VecRestoreArray(vec, &avec);CHKERRQ(ierr);
...
```

PETSc Matrices

Matrices

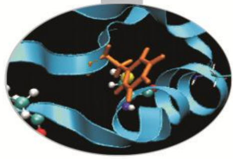


What are PETSc matrices?

- Roughly represent linear operators that belong to the dual of a vector space over a field (e.g. R^n)
- In most of the PETSc low-level implementations, each process logically owns a submatrix of contiguous rows

Features

- Supports many storage formats
  - AIJ, BAIJ, SBAIJ, DENSE, CUSP (GPU, dev-only) ...
- Data structures for many external packages
  - MUMPS (parallel), SuperLU_dist (parallel), SuperLU, UMFPack
- Hidden communications in parallel matrix assembly
- Matrix operations are defined from a common interface
- Shell matrices via user defined MatMult and other ops



Matrix Creation I

Matrices are PETSc objects like vectors and can be consequently created and destroyed

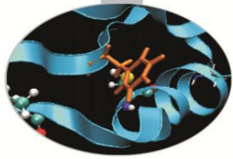
```
MatCreate(MPI_Comm comm, Mat *A)
```

- Automatically generates the appropriate matrix type (sequential or parallel) over all processes in `comm`.

```
MatDestroy(Mat *A)
```

- Destroys the Mat object

Matrix Creation II



Matrices are PETSc objects like vectors and can be consequently created and destroyed

MatCreate(MPI_Comm comm, Mat *A)

– Automatically generates the appropriate matrix type (sequential or parallel) over all processes in `comm`.

Several more aspects to creation:

MatSetType(Mat A, Mat_Type=MATSEQAIJ or MATMPIAIJ)
MatSetSizes(Mat A, int m, int n, int M, int N)
MatSeqAIJSetPreallocation(Mat A, PetscInt nz, PetscInt nnz[])

Local or global size can be **PETSC_DECIDE** (as in the vector case)

MatSetFromOptions(Mat A)

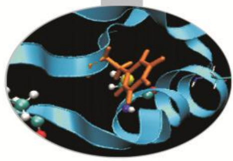
– Configures the matrix from the options database.

MatDuplicate(Mat B, MatDuplicateOption op, Mat *A)

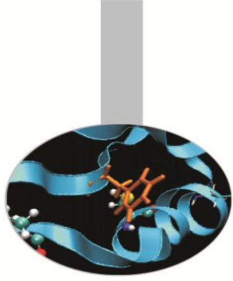
– Duplicates a matrix (including or not its nonzeros).

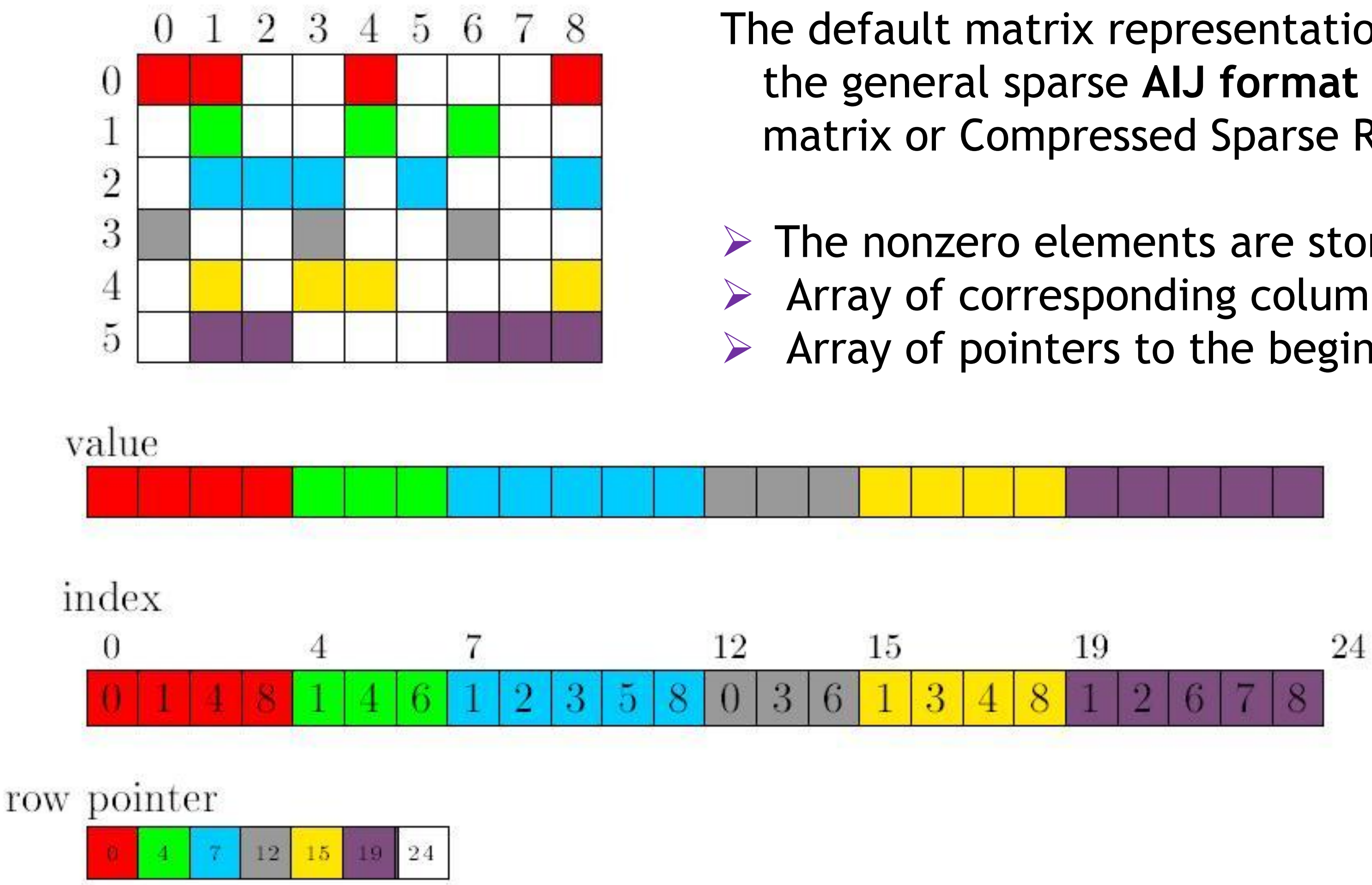
Matrix Creation (all in one)

```
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 AIJ format





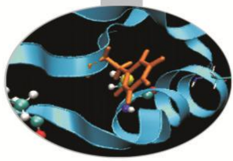
The default matrix representation within PETSc is the general sparse **AIJ format** (Yale sparse matrix or Compressed Sparse Row, CSR)

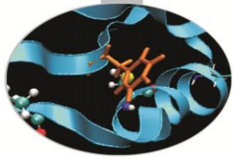
- The nonzero elements are stored by rows
- Array of corresponding column numbers
- Array of pointers to the beginning of each row

Matrix memory preallocation

- PETSc matrix creation is very flexible: No preset sparsity pattern
- Memory **preallocation** is critical for achieving **good performance** during matrix assembly, as this reduces the number of allocations and copies required during the assembling process. Remember: malloc is very expensive (run your code with –memory_info, -malloc_log)
- Private representations of PETSc sparse matrices are dynamic data structures: **additional nonzeros can be freely added** (if no preallocation has been explicitly provided).
- No preset sparsity pattern, any processor can set any element: potential for lots of malloc calls
- Dynamically adding many nonzeros
  - requires additional memory allocations
  - requires copies

  → **kills performances!**





Preallocation of a sequential sparse matrix (1/2)

```
MatSeqAIJSetPreallocation(Mat A, Petscint nz,
                          PetscInt *nnz)
```

- Dynamic preallocation if (`nz == 0 && nnz == PETSC_NULL`)

- Quick and dirty preallocation if nz is set to the maximum number of nonzeros in any row .
  - Fine if the number of nonzeros per row is roughly the same throughout the matrix

Preallocation of sequential sparse matrix (2/2)

- A finer preallocation

  `nnz[0]` = *<nonzeros in row 0>*

  …

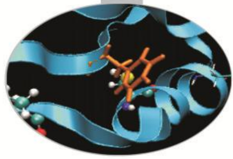
  `nnz[m]` = *<nonzeros in row m>*

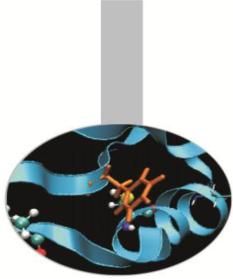
- If one **underestimates** the actual number of nonzeros in a given row, then during the assembly process PETSc will complain unless otherwise told.

Preallocation of a parallel sparse matrix (1/2)

```
MatMPIAIJSetPreallocation(Mat A,
                          Petscint dnz,
                          PetscInt *dnnz,
                          Petscint onz,
                          PetscInt *onnz)
```

- Same logic as before for dynamic allocation
- `dnz` and `dnnz` specify preallocation for the diagonal block
- `onz` and `onnz` specify preallocation for the off-diagonal block





Preallocation of a parallel sparse matrix (2/2)

Each process **logically owns** a matrix subset of contiguously numbered global rows. Each subset consists of two sequential matrices corresponding to diagonal and off-diagonal parts.

$$
\begin{array}{c}
\text{P0} \\ \\ \\ \text{P1} \\ \\ \\ \text{P2} \\ \\
\end{array}
\left(\begin{array}{ccc|ccc|cc}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
\hline
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
\hline
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34
\end{array}\right)
$$

Process 0

```
dnz=2, onz=2
dnnz[0]=2, onnz[0]=2
dnnz[1]=2, onnz[1]=2
dnnz[2]=2, onnz[2]=2
```

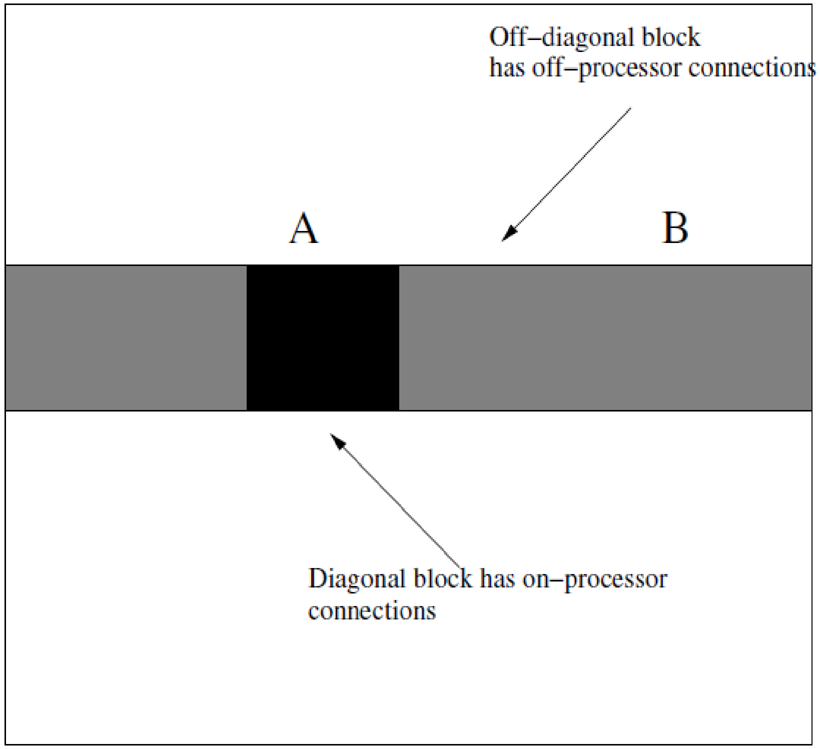
Process 1

```
dnz=3, onz=2
dnnz[0]=3, onnz[0]=2
dnnz[1]=3, onnz[1]=1
dnnz[2]=2, onnz[2]=1
```

Process 2

```
dnz=1, onz=4
dnnz[0]=1, onnz[0]=4
dnnz[1]=1, onnz[1]=4
```

Preallocation of a parallel sparse matrix 3


Off–diagonal block
has off–processor connections
A
B
Diagonal block has on–processor
connections

$y \leftarrow A\,x_A + B\,x_B$

- x_B needs to be communicated
- $A\,x_A$ can be computed in the meantime

Algorithm

- Initiate asynchronous sends/receives for x_B
- compute $A\,x_A$
- make sure x_B is in
- compute $B\,x_B$

The splitting of the matrix storage into A (diag) and B (off-diag) part, code for the
sequential case can be reused.

Querying parallel structure

PETSc Matrices are partitioned by block rows:

`MatGetSize(Mat A, PetscInt *M, PetscInt* N)`

– Gets global number of rows and columns

`MatGetLocalSize(Mat A, PetscInt *m, PetscInt* n)`

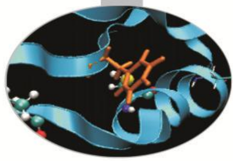
– Gets local number of rows and columns (output can be implementation dependent)

`MatGetOwnershipRange(Mat A, PetscInt *m, PetscInt* n)`

– Gets the first and last (+1) of locally owned rows

`MatGetOwnershipRanges(Mat A, const PetscInt **ranges)`

– Gets start and end rows of each process sharing the matrix



Setting values

Set one value:

```
MatSetValue(Mat A, PetscInt idxm, PetscInt idxn,
            PetscScalar value,InsertMode mode)
```

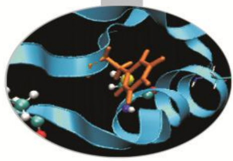
Insert mode can be **INSERT_VALUES**, **ADD_VALUES**

Set block of values:

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

`Mat A` is row oriented.

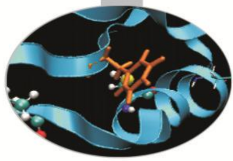
Matrix assembly



Like PETSc vectors, PETSc Mat assembling process involves calls to (independent of parallelism)

```
MatAssemblyBegin(Mat A, MatAssemblyType type)
…
MatAssemblyEnd(Mat A, MatAssemblyType type)
```

Getting Values



- Values are often not needed: many matrix operations supported
- Matrix elements can only be obtained locally.
- They are normally used for inspection only, they are expensive.

```
MatGetRow(Mat A, PetscInt row, PetscInt* ncols,
          const PetscInt* cols[], const PetscScalar*
          vals[])
MatRestoreRow(…)

MatGetValues(Mat A, PetscInt m, PetscInt* idxm[],
             PetscInt n, const PetscInt* idxn[],
             Petscscalar* vals[])
```

Other Matrix Type

- Block Matrix

```
MatCreateBAIJ(MPI_Comm comm, PetscInt bs, PetscInt
m, PetscInt n, PetscInt M, PetscInt N, PetscInt
d_nz, const PetscInt d_nnz[], PetscInt o_nz, const
PetscInt o_nnz, Mat *A)
```

  - Creates a sparse parallel matrix in block AIJ format (more than one dof per node)

- Dense Matrix:

```
MatCreateSeqDense(MPI_Comm comm, PetscInt m,
PetscInt n, PetscScalar* data[], Mat *A)
```

  - Creates a sequential dense matrix that is stored in column major order (the usual Fortran 77 manner). Many of the matrix operations use the BLAS and LAPACK routines.

```
MatCreateDense(MPI_Comm comm, PetscInt m, PetscInt
n, PetscInt M, PetscInt N, PetscScalar* data[], Mat
*A)
```

Matrix Viewers

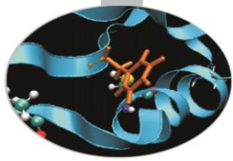
MatView(Mat A, PetscViewer v)

- Prints matrix content using the viewer object

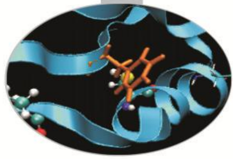
MatView(A,0); (Fortran: 0 -> PETSC_NULL_INTEGER)

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

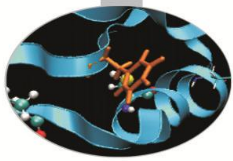


General Viewers



Any PETSc object (`Vec, Mat, Ksp,` …) can be viewed using a `PetscViewer.`
Let’s see an example with a PETSc Matrix.

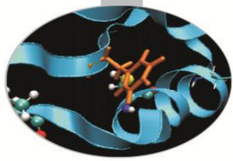
```
..
PetscViewer viewer;
PetscViewerBinaryOpen(PETSC_COMM_WORLD, “matdata”,
FILE_MODE_WRITE, &viewer):
PetscViewerSetFormat(viewer, PETSC_VIEWER_BINARY_MATLAB)
MatView(A, viewer);
PetscViewerDestroy(viewer);
...
```



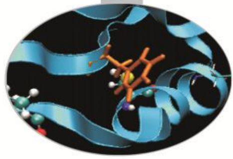
Numerical Matrix Operations

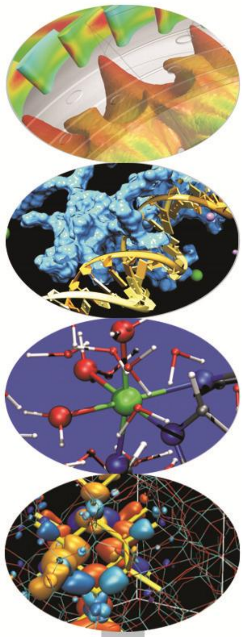
Function Name	Operation
MatAXPY(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
MatMult(Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose(Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale(Mat A,PetscScalar a);	$A = a * A$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose(Mat A,MatReuse,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Mat Y,PetscScalar a);	$Y = Y + a * I$

Matrix - Example



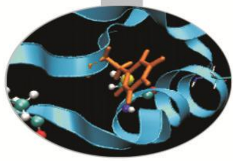
```
#include "petscmat.h"
...
Mat A;
PetscInt cols[3], i, istart, iend;
PetscScalar vals[3];
PetscErrorCode ierr;
...
/* suppose A has been already created and have its type set */
ierr = MatGetOwnershipRange(A,&istart,&iend);CHKERRQ(ierr);
...
vals[0] = -1.0; vals[1] = 2.0; vals[2] = -1.0; /* 1D laplacian stencil */
for (i=istart; i<iend; i++) {
  cols[0] = i-1; cols[1] = i; cols[2] = i+1; /* 1D laplacian stencil */
  ierr = MatSetValues(A,1,&i,3,cols,value,INSERT_VALUES);CHKERRQ(ierr);
}
ierr = MatAssemblyBegin(A,MAT_FLUSH_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FLUSH_ASSEMBLY);CHKERRQ(ierr);
/* all processes contribute to 0,0 entry */
ierr = MatSetValue(A,0,0,vals[0],ADD_VALUES);CHKERRQ(ierr);
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
```





Petsc linear solvers: KSP & PC

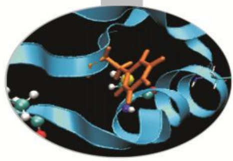
Why using iterative solvers?



- Solve a linear system A x = b using the Gauss Elimination method can be very time-resource consuming

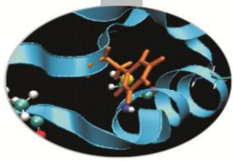
- Alternatives to direct solvers are iterative solvers
- Convergence of the succession is not always guarateed
- Possibly much faster and less memory consuming
- Basic iteration: y <- A x executed once x iteration
- Also needed a good preconditioner: $B \approx A^{-1}$

Iterative solver basics



- **KSP** (K stands for *Krylov*) objects are used for solving linear systems by means of iterative methods.
- Convergence can be improved by using a suitable **PC** object (preconditoner).
- Almost all iterative methods are implemented.
- Classical iterative methods (not belonging to KSP solvers) are classified as preconditioners
- Direct solution for parallel square matrices available through external solvers (MUMPS, SuperLU_dist). Petsc provides a built-in LU serial solver.
- Many KSP options can be controlled by command line
- Tolerances, convergence and divergence reason
- Custom monitors and convergence tests

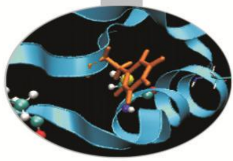
PETSc Iterative Solver Basics



```
KspCreate(MPI_Comm comm, KSP* inksp)
KspDestroy(KSP* inksp)

KSPSetOperators(KSP ksp, Mat amat, Mat pmat)

KspSolve(KSP ksp, Vec b, Vec x)
/* optional */ KSPSetUp(KSP ksp)
```

SuperComputing Applications and Innovation

Solver Type



`KspSetType(KSP ksp, KSPType ksptype)`

The default KSP type is GMRES with a restart of 30, using modified Gram-Schmidt orthogonalization.

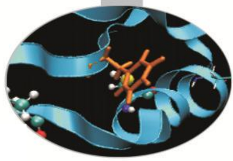
KSP can be controlled from the command line:

`KspSetFromOptions(KSP ksp)`
/* right before KSPSolve or KSPSetUp */

then options -ksp.... are parsed.

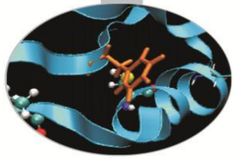
- type: `-ksp_type gmres -ksp_gmres_restart 20`
- `-ksp_view`

Solver Type II



Method	KSPType	Options Database Name
Richardson	KSPRICHARDSON	richardson
Chebyshev	KSPCHEBYSHEV	chebyshev
Conjugate Gradient [12]	KSPCG	cg
BiConjugate Gradient	KSPBICG	bicg
Generalized Minimal Residual [16]	KSPGMRES	gmres
Flexible Generalized Minimal Residual	KSPFGMRES	fgmres
Deflated Generalized Minimal Residual	KSPDGMRES	dgmres
Generalized Conjugate Residual	KSPGCR	gcr
BiCGSTAB [19]	KSPBCGS	bcgs
Conjugate Gradient Squared [18]	KSPCGS	cgs
Transpose-Free Quasi-Minimal Residual (1) [8]	KSPTFQMR	tfqmr
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr
Conjugate Residual	KSPCR	cr
Least Squares Method	KSPLSQR	lsqr
Shell for no KSP method	KSPPREONLY	preonly

Convergence



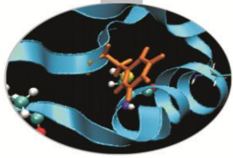
Iterative solvers can fail to converge

- The KSP Solve call itself gives no feedback: user don't know if the solution is wrong or not

- `KspGetConvergedReason(KSP ksp, KSPConvergedReason* reason)`
  - positive is convergence, negative divergence
  - (`$fPETSC_DIRg/include/petscksp.h` for list)

- `KspGetIterationNumber(KSP ksp, PetscInt* its)`
  - Gets the current iteration number (During the ith iteration this returns i-1); if the KSPSolve function is complete, returns the number of iterations used.

Monitors and convergence tests

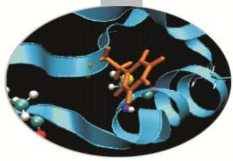


- **KspSetTolerances(KSP ksp, PetscReal* rtol, PetscReal* atol, PetscReal* dtol, PetscInt maxits)**
  - `rtol`: the relative convergence tolerance, relative decrease in the (possibly preconditioned) residual norm
  - `atol`: the absolute convergence tolerance absolute size of the (possibly preconditioned) residual norm
  - `dtol`: the divergence tolerance
  - `maxits`: Maximum number of iterations to use

Monitors can also be set in code, but easier:

- `-ksp_monitor`
- `-ksp_monitor_true_residual`

PC basics



- PCs are usually created as part of a KSP object. PCs can be created and destroyed independently but it's almost never done in practice

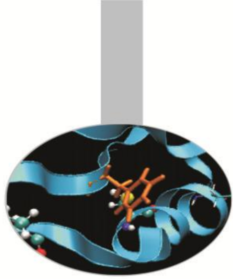
```
...
KSP solver; PC precond;
KSPCreate(comm, &solver);
KSPGetPC(solver, &precond);
PCSetType(precond, PCJACOBI);
...
```

- Controllable through command line options:

```
-pc_type ilu –pc_factor_levels 3
```

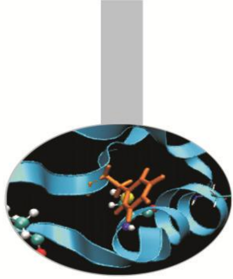
SuperComputing Applications and Innovation

PETSc PC methods



Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Algebraic Multigrid	PCGAMG	gamg
Linear solver	PCKSP	ksp
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCHOLESKY	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

Preconditioner Reuse

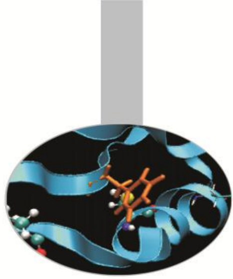


- In the context of non-linear solvers, time-dependent (linear and not) solvers, the preconditioner can sometimes be reused:
  - If the jacobian doesn't change much, reuse the preconditioner completely
  - If the preconditioner is recomputed, the sparsity pattern will probably not change

KSPSetOperators(solver, A, B, structureflag)

- B is the basis for the preconditioner, need not be A
- structureflag can be **SAME_PRECONDITIONER**, **SAME_NON_ZERO_PATTERN**, **DIFFERENT_NON_ZERO_PATTERN**:
- Avoid the recomputation of the preconditioner (sparsity pattern) if possible

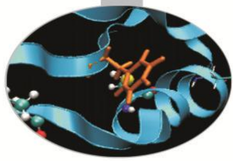
Direct solver



- Iterative method with direct solver as preconditioner would converge in one step
- Direct methods in PETSc implemented as special iterative method: KSPPREONLY -> only apply the preconditioner
- All direct methods are preconditioner type PCLU

```
./myprog -pc_type lu -ksp_type preonly -pc_factor_mat_solver_package mumps
```

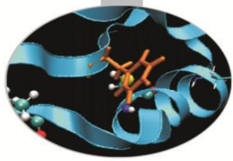
Factorization preconditioner



- Exact factorization: A = LU
- Inexact factorization: A ≈ M = $\underline{L}$ $\underline{U}$ where $\underline{L}$, $\underline{U}$ obtained by throwing away the 'fill-in' during the factorization process (sparsity pattern of M is the same as A)
- Application of the preconditioner (that is, solve Mx = y) approx same cost as matrix-vector product y <- A x
- Factorization preconditioners are sequential

- PCICC: symmetric matrix, PCILU: nonsymmetric matrix

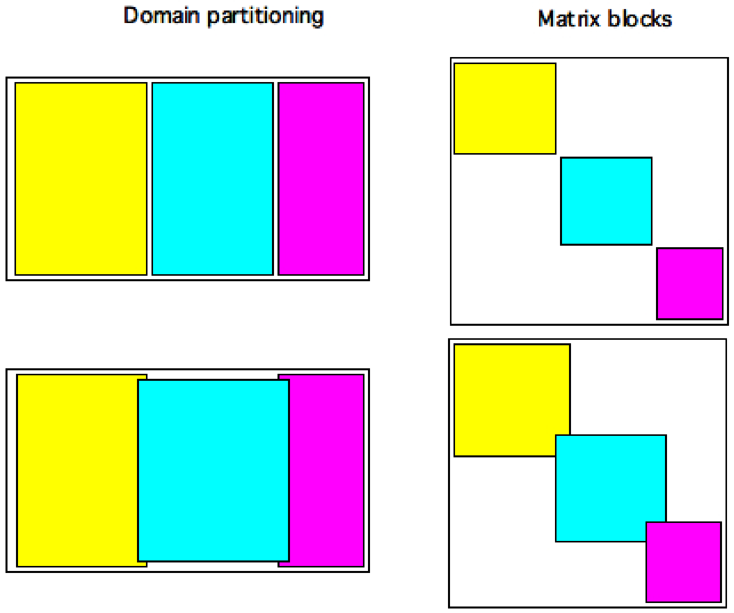
Parallel preconditioners

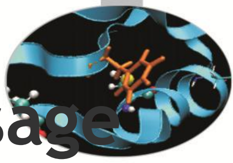


- Factorization preconditioners are sequential
- We can use them in parallel as a subpreconditioner of a parallel preconditioner as Block Jacobi or Additive Schwarz methods
- Each processor has its own block(s) to work with

Block Jacobi and Additive Schwarz preconditioners

- Both methods are parallel
- BlockJacobi is fully parallel, Schwarz requires communications between neighbours
- Both require sequential local solver
- Schwarz can be more robust than BlockJacobi and have better convergence properties

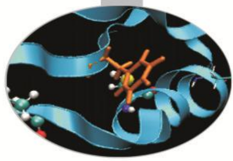


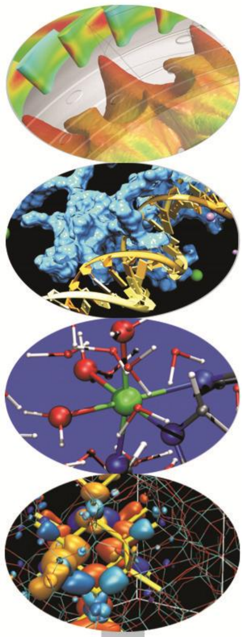


Block Jacobi and Additive Schwarz example usage

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

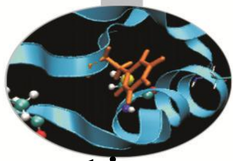
It can also be set by command line: `–sub_ksp_type` and `–sub_pc_type` (subksp is PREONLY by default)





Petsc nonlinear solvers: SNES

SNES: nonlinear solvers



The SNES class includes methods for solving systems of nonlinear equations coming from the discretization of a PDE in the form

$$F(x) = 0,\ F : \Re^n \rightarrow \Re^n.$$

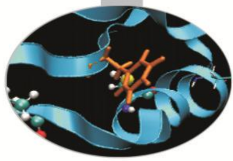
Typical solution method:

$$\mathbf{x}^{n+1} = \mathbf{x}^n - \mathbf{J}^{-1}(\mathbf{x}^n)\mathbf{F}(\mathbf{x}^n)$$

User has to provide:

- Function F
- Jacobian J

```
SNESSetFunction(SNES snes, Vec v,
  PetscErrorCode (*SNESFunction)(SNES, Vec, Vec, void*),
void *ctx)
SNESSetJacobian(SNES snes, Mat amat, Mat pmat,
  PetscErrorCode (*SNESJacobianFunction)
  (SNES, Vec, Mat*, Mat*, MatStructure*,void *),
  void *ctx)
```



Basis SNES usage

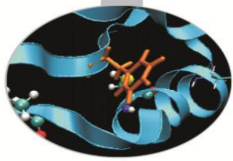
```
SNESCreate(MPI_Comm comm, SNES snes)
SNESSetType(SNES snes, SNESType type)
```

Method	SNESType	Options Name	Default Convergence Test
Line search	SNESNEWTONLS	newtonls	SNESConverged_NEWTONLS()
Trust region	SNESNEWTONTR	newtontr	SNESConverged_NEWTONTR()
Test Jacobian	SNESTEST	test	

```
SNESSetFromOptions(SNES snes)
SNESDestroy(SNES snes)
```

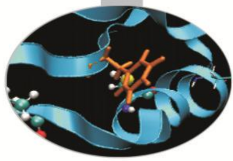
SuperComputing Applications and Innovation

SNES Specification



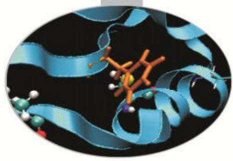
```
SNESSetFunction(SNES snes, Vec v,
  PetscErrorCode (*SNESFunction)(SNES, Vec, Vec, void*),
void *ctx)

SNESSetJacobian(SNES snes, Mat amat, Mat pmat,
  PetscErrorCode (*SNESJacobianFunction)
  (SNES, Vec, Mat*, Mat*, MatStructure*,void *),
  void *ctx)

SNESSetTolerances(SNES snes, PetscReal abstol, PetscReal
rtol, PetscReal stol, PetscInt maxit, PetscInt maxf)

SNESSolve(SNES snes, Vec rhs, Vec x)

SNESGetIterationNumber(SNES snes, PetscInt* iter)
```

Solve customization



```
...
SNES snes;
SNESSetType(snes, SNESTR); /*Newton with trust region*/
SNESGetKSP(snes, &ksp);
KSPGetPC(ksp, &pc);
PCSetType(pc, PCJACOBI);
KSPSetTolerances(ksp, 1.e-04, PETSC_DEFAULT,
PETSC_DEFAULT, 20);
...
```

Example of the Target Function

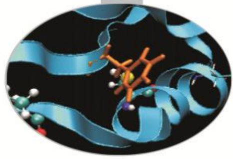


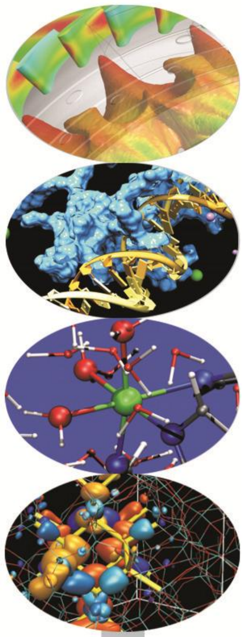
```
PetscErrorCode (*SNESFunction)(SNES snes, Vec x, Vec f,
void* ctx)

...
FormFunction(snes, x, f, ctx);
{
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;
}
...
```

Example of the Jacobian Function

PetscErrorCode (*SNESJacobianFunction)(SNES snes, Vec x, Mat amat, Mat pmat, void* ctx)

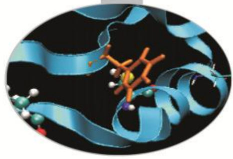
```
...
FormJacobian(snes, x, jac, precond, ctx);
{
PetscScalar a[]; PetscScalar p[];
VecGetArray(x, &xx);
a[0] = …; /* code here the linear operator discretization */
p[0] = …; /* code here the preconditioner operator discretization */
MatSetValues(*jac, m, idxm, n, idxn, a, INSERT_VALUES);
MatSetValues(*precond, m, idxm, n, idxn, p, INSERT_VALUES);
VecRestoreArray(x, &xx);
MatAssemblyBegin(*precond, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(*precond, MAT_FINAL_ASSEMBLY);
MatAssemblyBegin(*jac, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(*jac, MAT_FINAL_ASSEMBLY);
return 0;
}
...
```





PETSc time-stepping solvers: TS

TS: time steppers

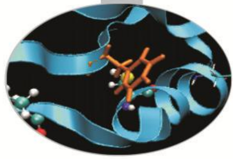
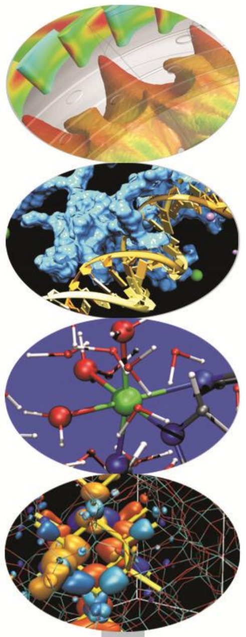


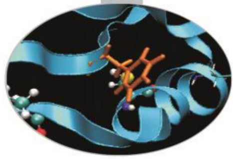
TS class includes methods for solving systems of linear or nonlinear Ordinary Differential Equations (ODEs) or Differential Algebraic Equations (DAEs), i.e. problems which can be written down as

$$F(t, u, \dot{u}) = G(t, u), \quad u(t_0) = u_0.$$

The class provides explicits, implicits or semi-implicit methods and the user has to provide functions on how to compute the fundamental pieces of equation (F, G and a Jacobian

Grid Manipulation and Discretization



DM object

DM is an abstract PETSc object that manages an abstract grid object and its interactions with the algebraic solvers.

- **DMDA**

An object that is used to manage data for a structured grid in 1, 2, or 3 dimensions. In the global representation of the vector each process stores a non-overlapping rectangular (or slab in 3d) portion of the grid points. In the local representation these rectangular regions (slabs) are extended in all directions by a stencil width. The vectors can be thought of as either cell centered or vertex centered on the mesh.

- **DMPLEX**

An object that encapsulates an unstructured mesh, or CW Complex, which can be expressed using a Hasse Diagram. In the local representation, Vecs contain all unknowns in the interior and shared boundary. This is specified by a PetscSection object.

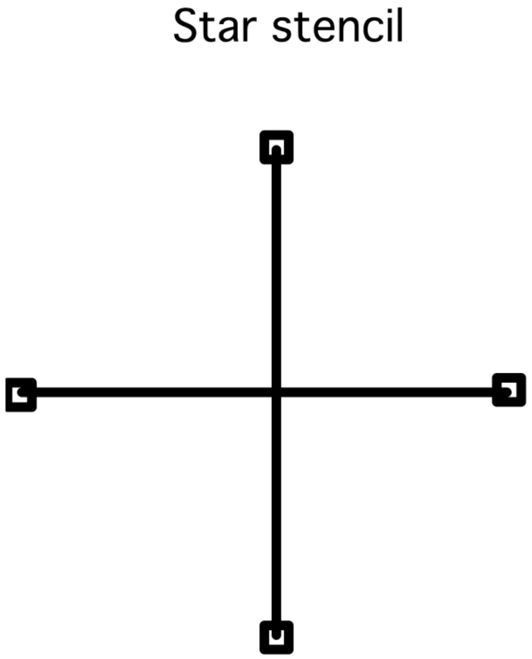
Regular GRID: DMDA

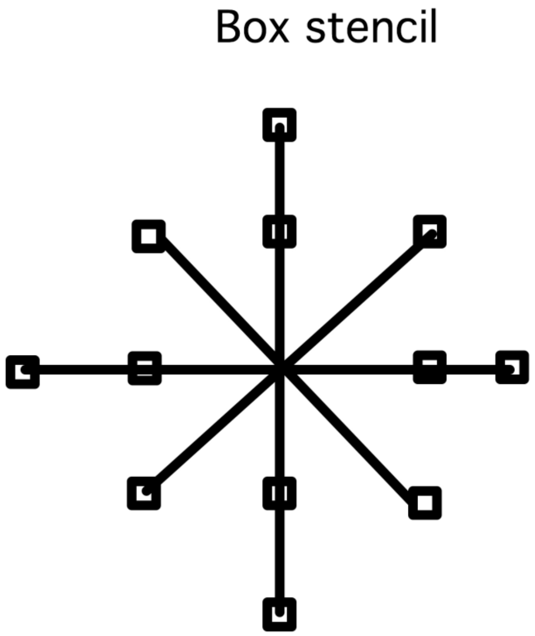
DMDAs are for storing vector field, not matrix.

To set the DMDA coordinates to be a uniform grid:

```
DMDASetUniformCoordinates(DM da, PetscReal xmin,
PetscReal xmax, PetscReal ymin, PetscReal ymax,
PetscReal zmin, PetscReal zmax)
```

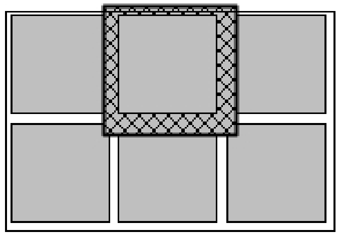
Support for different stencil types:



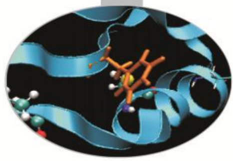


Ghost regions around processors

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".

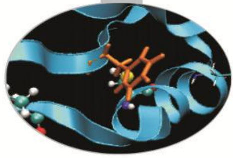


DMDA construction



```
DMDACreate2D(MPI_Comm comm, DMBoundaryType bndx,
DMBoundaryType bndy, DMDAStencilType stencil_type, PetscInt
M, PetscInt N, PetscInt m, PetscInt n, PetscInt dof,
PetscInt stencil_width, const PetscInt lx[], const PetscInt
ly[], DM* da)
```

- bndx, bndy -> boundary (physical domain) behaviour: DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, M_BOUNDARY_MIRROR, DM_BOUNDARY_PERIODIC, DM_BOUNDARY_TWIST
- DMDAStencilType: DMDA_STENCIL_STAR,DMDA_STENCIL_BOX
- M/N: Number of grid points in x/y-direction
- m/n: Number of processes in x/y-direction (or PETSC_DECIDE)
- dof: Degrees of freedom per node
- s: The stencil width (for instance, 1 for 2D five-point stencil)
- lm/n: arrays containing the number of nodes in each cell along the x and y coordinates, or NULL. If non-null, these must be of length as m and n, and the corresponding m and n cannot be PETSC_DECIDE. The sum of the lx[] entries must be M, and the sum of the ly[] entries must be N.



Associated vectors

```
DMCreateGlobalVector(DMDA da, Vec* g)

DMCreateLocalVector(DMDA da, Vec* l)
```

- global -> local

```
DMGlobalToLocalBegin/End(DMDA da, Vec g, InsertMode
mode, Vec l)
InsertMode mode: INSERT_VALUES, ADD_VALUES
```

- local -> global

```
DMLocalToGlobalBegin/End(DMDA da, Vec l, InsertMode
mode, Vec g)
```

- local -> global -> local

```
DMLocalToLocalBegin/End(DMDA da, Vec l1, InsertMode
mode, Vec l2)
```

Associated Matrix, Ksp and SNES

- `DMCreateMatrix(DM dm, Mat* mat)`
- `MatSetDM(DMDA da, Vec* g)`
- `KSPSetDM(DMDA da, Vec* g)`
- `SNESSetDM(DMDA da, Vec* g)`

Operator Discretization on a Regular Grid

- Which function can be used to discretize a linear operator with Finite Difference?

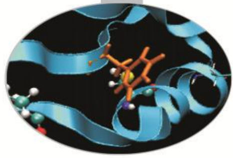
```
MatSetValuesStencil(Mat mat, PetscInt m, const
MatStencil idxm[], PetscInt n, const MatStencil
idxn[], const PetscScalar v[], InsertMode addv)
```

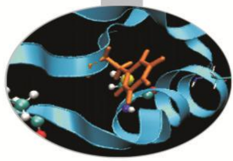
   – Inserts or adds a block of values into a matrix using structured grid indexing.

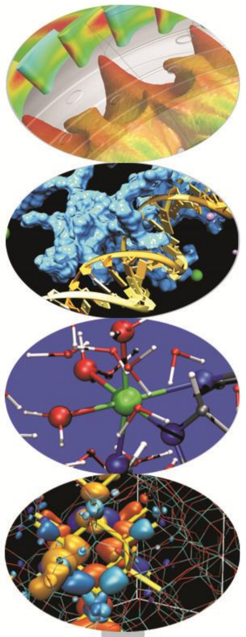
- Which function can be used to impose Dirichlet BDCs?

```
MatZeroRowsColumnsStencil(Mat mat, PetscInt numRow,
const MatStencil rows[], PetscScalar diag, Vec x,
Vec b)
```

   – Zeros all row and column entries (except possibly the main diagonal) of a set of rows and columns of a matrix.

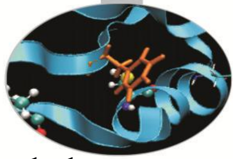






Debugging and Profiling

Debugging



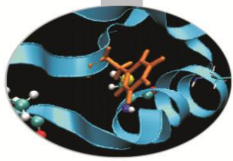
If configured in debug mode (default), PETSc provides large support to error handling, backtracing and memory leak detection for C/C++ codes by simply adhering to very basic guidelines for code developing

PETSc programs may be debugged using one of the two options:
`-start_in_debugger` - start all processes in debugger
`-on_error_attach_debugger` - start debugger only on error

Also, if configured with MPICH for the message passing interface and with GNU compilers, PETSc code is completely **valgrind-free** (e.g. not true with OpenMPI).

- 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)

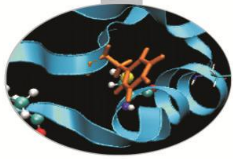
Profiling and performance tuning



Profiling:

- Integrated profiling of:
  - time
  - floating-point performance
  - memory usage
  - communication
- User-defined events
- Profiling by stages of an application

SuperComputing Applications and Innovation



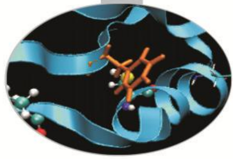
Log summary: overall

```
                        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 26100 96 17 92  26100 96 17 92   194
```

PETSc profiling options

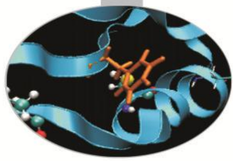


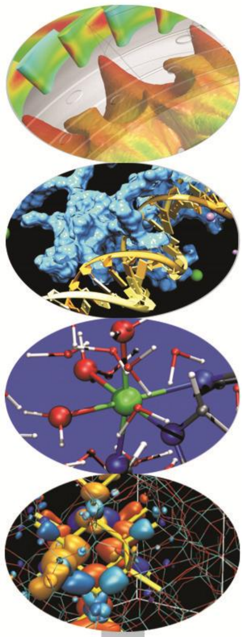
The profiling options include the following:

`-log_summary` - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, `-log_summary` is intended as the primary means of monitoring the performance of PETSc codes.

`-info [infofile]` - Prints verbose information about code to `stdout` or an optional file. This option provides details about algorithms, data structures, etc.

`-log_trace [logfile]` - Traces the beginning and ending of all PETSc events. If used in conjunction with `-info`, this option is useful to see where a program is hanging without running in the debugger.





Thank you for the attention