22nd Summer School on PARALLEL COMPUTING

Introduction to PETSc

Portable, Extensible Toolkit for Scientific Computation

Stefano Zampini – s.zampini@cineca.it
SuperComputing Applications and Innovation Department

CINECA

PETSc main features

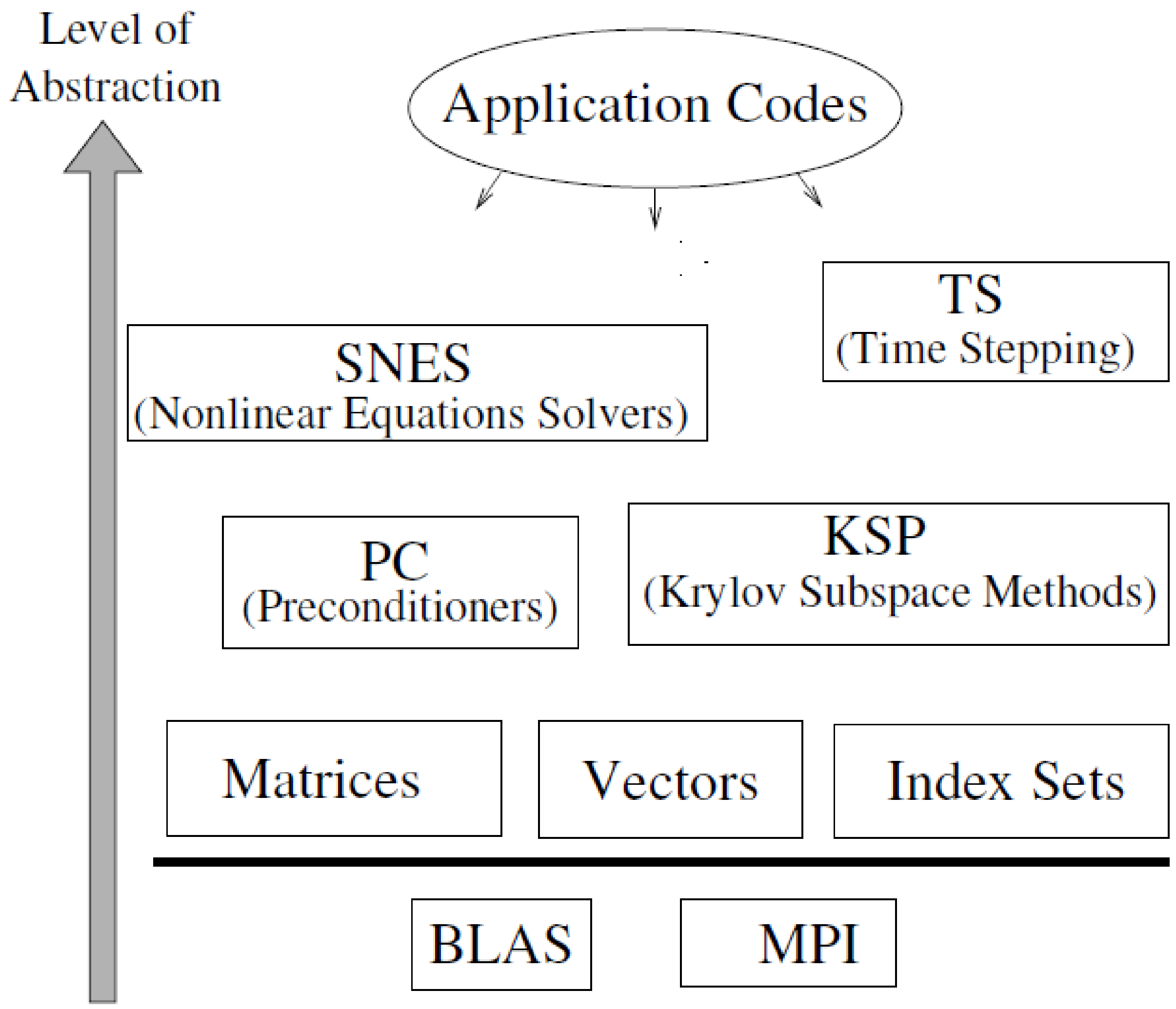
PETSc – Portable, Extensible Toolkit for Scientific Computation

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

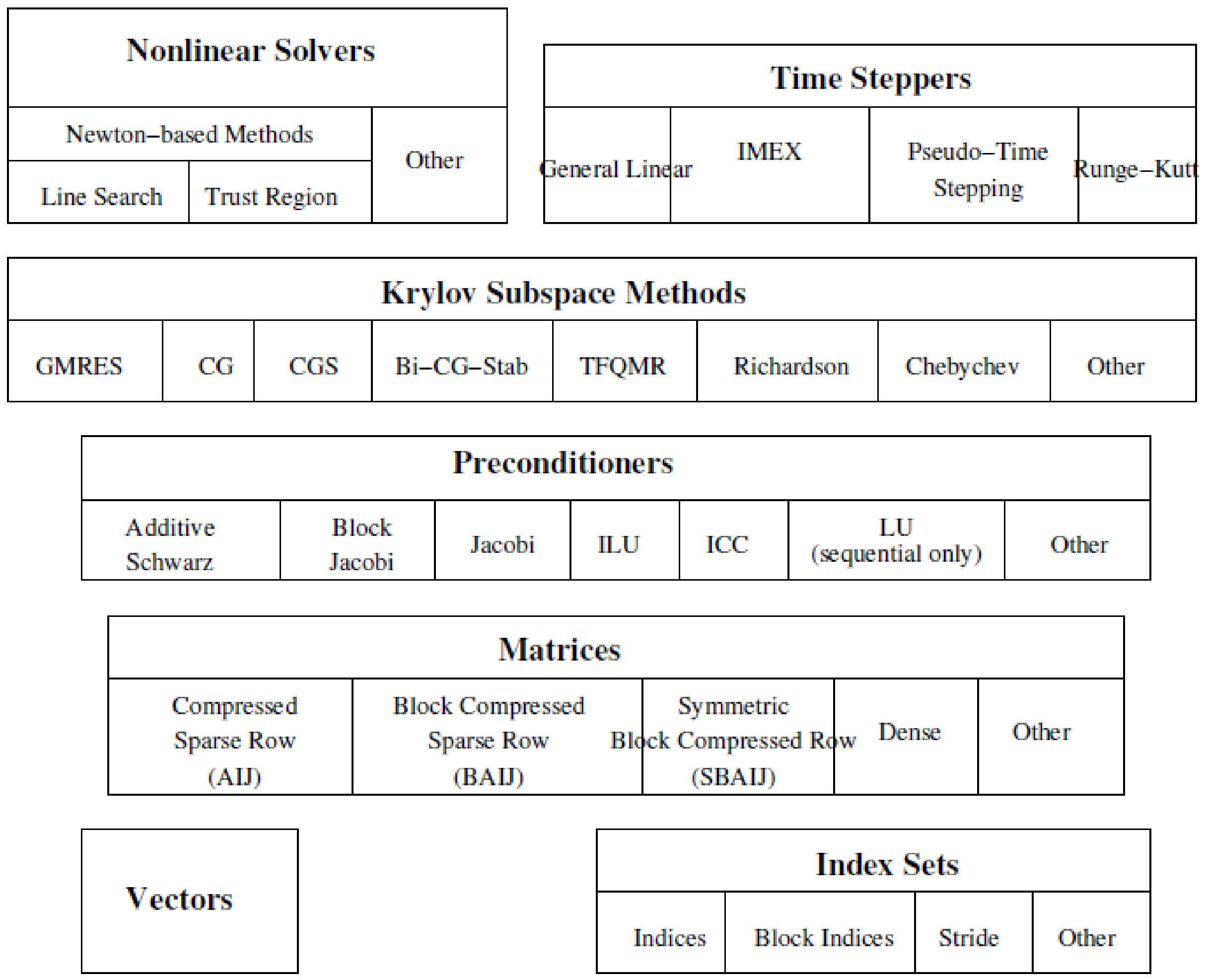
- Essentially Object Oriented code written in **C**
- Usable from **Fortran, C++** and **Python** (via **petsc4py**)
- Uses **MPI** for message-passing
- Uses **BLAS** and **LAPACK** for low-level data manipulation
- Can be configured for single or double precision, real or complex scalars
- Interfaces with many other numerical packages
- PETSc has been used for modeling in all of these **areas**:

  Acoustics, Aerodynamics, Air Pollution, Arterial Flow, Brain Surgery, Cancer Surgery and Treatment, Cardiology, Combustion, Corrosion, Earth Quakes, Economics, Fission, Fusion, Magnetic Films, Material Science, Medical Imaging, Ocean Dynamics, PageRank, Polymer Injection Molding, Seismology, Semiconductors, ...

PETSc class hierarchy



PETSc numerical components


Parallel Numerical Components of PETSc

PETSc model

Goals

- Portability
- Performance
- Scalable parallelism

Approach

- Object Oriented Delegation Pattern
- Many specific implementations of the same object
- Shared interface
- Command line customization

Benefit

- Most of linear and nonlinear algebra techniques implemented
- Flexibility: easy switch among different implementations
- Nasty details of implementation hidden

Ongoing (development version only)

- **GPU**, **MIC** (via **OpenCL**) and **pthread** low-level implementations

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)`

PETSc from a user perspective

- Home page

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

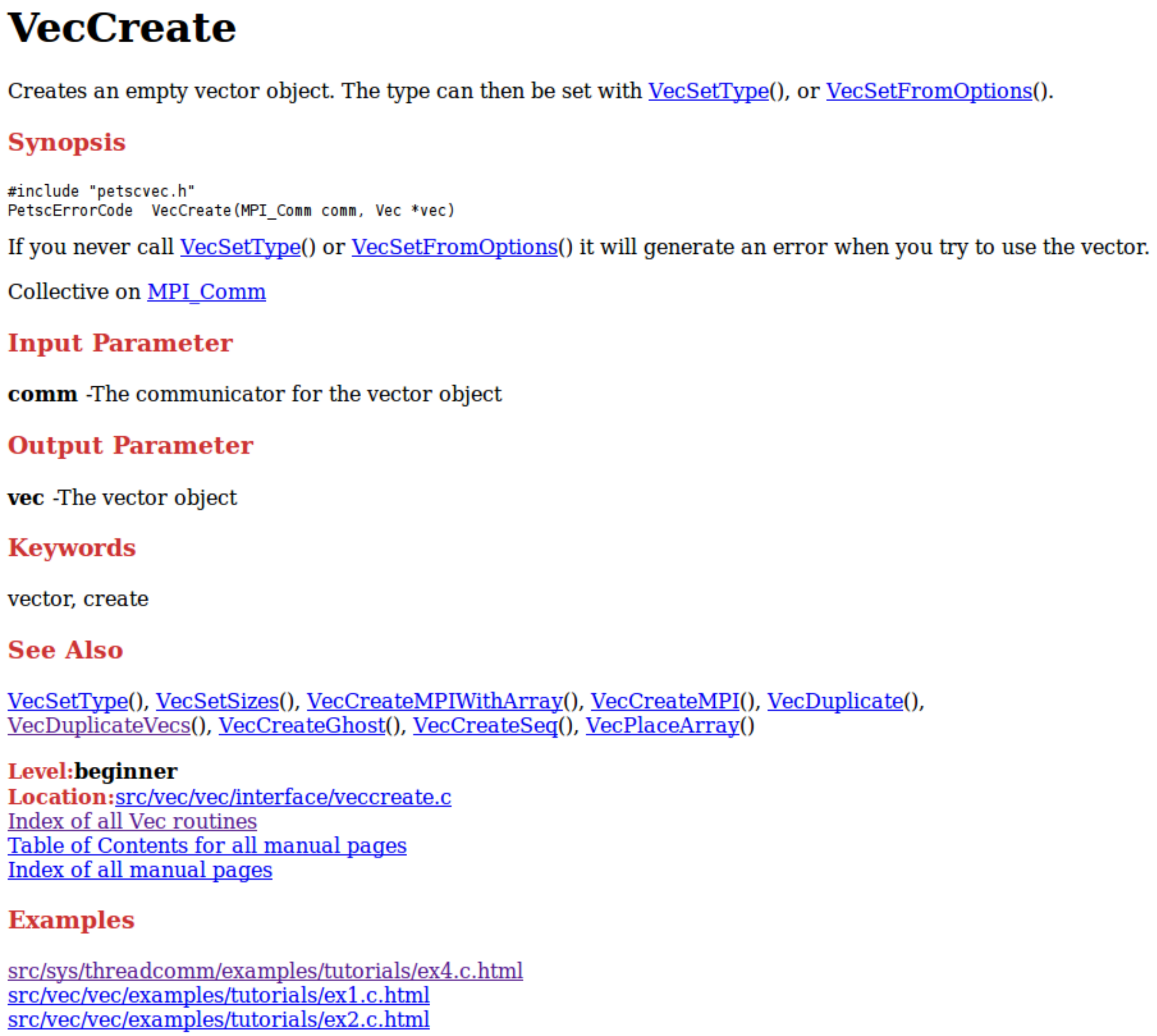
  http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf
- Public functions for XXX class (Vec, Mat, KSP, …) accessible at

  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 ran

  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 any if the packages is 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 req'ed)
- Finalizes MPI (if `PetscInitialize()` began MPI)
- Shutdowns all PETSc services

A simple hello world program

```
#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 program

```
#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
```

Vec and Mat

Vectors

What are PETSc vectors?

- Roughly represent elements of a Banach space
- Usually they store solutions and right-hand sides.
- Vector elements are **PetscScalars**
- Each process locally owns a subvector of contiguously numbered global indices

Features

- Vector types: STANDARD, PTHREAD and CUSP (dev only)
- 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

Vector basic interface 1/2

`VecCreate(MPI_Comm comm, Vec *v)`

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

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

- Sets local and global sizes

`VecSetType(Vec v, VecType type)`

- Sets vector type (defines the delegated object)

`VecSetFromOptions(Vec v)`

- Configures the vector from the options database

`VecDuplicate(Vec old, Vec *new)`

- Duplicates the vector (doesn't copy values)

Vector basic interface 2/2

`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

`VecSet(Vec x, PetscScalar value)`

- Sets all values of the vector to a specific value

`VecDestroy(Vec *x)`

- Destroys the Vec object

Vector assembly

```
VecSetValue(Vec x, PetscInt idx, PetscScalar v,
            InsertMode mode)
VecSetValues(Vec x, PetscInt n, PetscInt *idx,
             PetscScalar *v, InsertMode mode)
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
- `VecAsseblyBegin` starts communications to ensure that values end up where needed (allow other operations, such as some independent computation, to proceed).
- `VecAssemblyEnd` completes the communication

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 */
```

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);	$x_i = \alpha$

Working with local vectors

Sometimes is more efficient to directly access local storage of a PETSc `Vec` (e.g. in finite difference computations involving vector elements)

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

- Access the local storage

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

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

PETSc handles data structure conversions (e.g. if data resides on GPU)

- For most common uses, these routines are inexpensive and **do not involve** a copy of local data.

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);
...
```

Matrices

What are PETSc matrices?

- Roughly represent linear operators in Banach spaces
- 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

Parallel sparse matrices

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.

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

Matrix operations 1/2

`MatCreate(MPI_Comm comm, Mat *A)`

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

```
MatSetSizes(Mat A, PetscInt m, PetscInt n,
                   PetscInt M, PetscInt N)
```

- Sets the local and global sizes

`MatSetType(Mat A, MatType type)`

- Sets matrix type (defines the delegated object)

`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 operations 2/2

`MatView(Mat A, PetscViewer v)`

– Prints matrix content using the viewer object

`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

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

– Gets global number of rows and columns

`MatDestroy(Mat *A)`

– Destroys the Mat object

Matrix assembly

Like PETSc vectors, Mat assembling process involves calls to

```
MatSetValue(Mat A, PetscInt idxm, PetscInt idxn,
            PetscScalar value,InsertMode mode)
MatSetValues(Mat A, PetscInt m, PetscInt idxm[],
                    PetscInt n, PetscInt idxn[],
                    PetscScalar values[],
                    InsertMode mode)
MatAssemblyBegin(Mat A, MatAssemblyType type)
MatAssemblyEnd(Mat A, MatAssemblyType type)
```

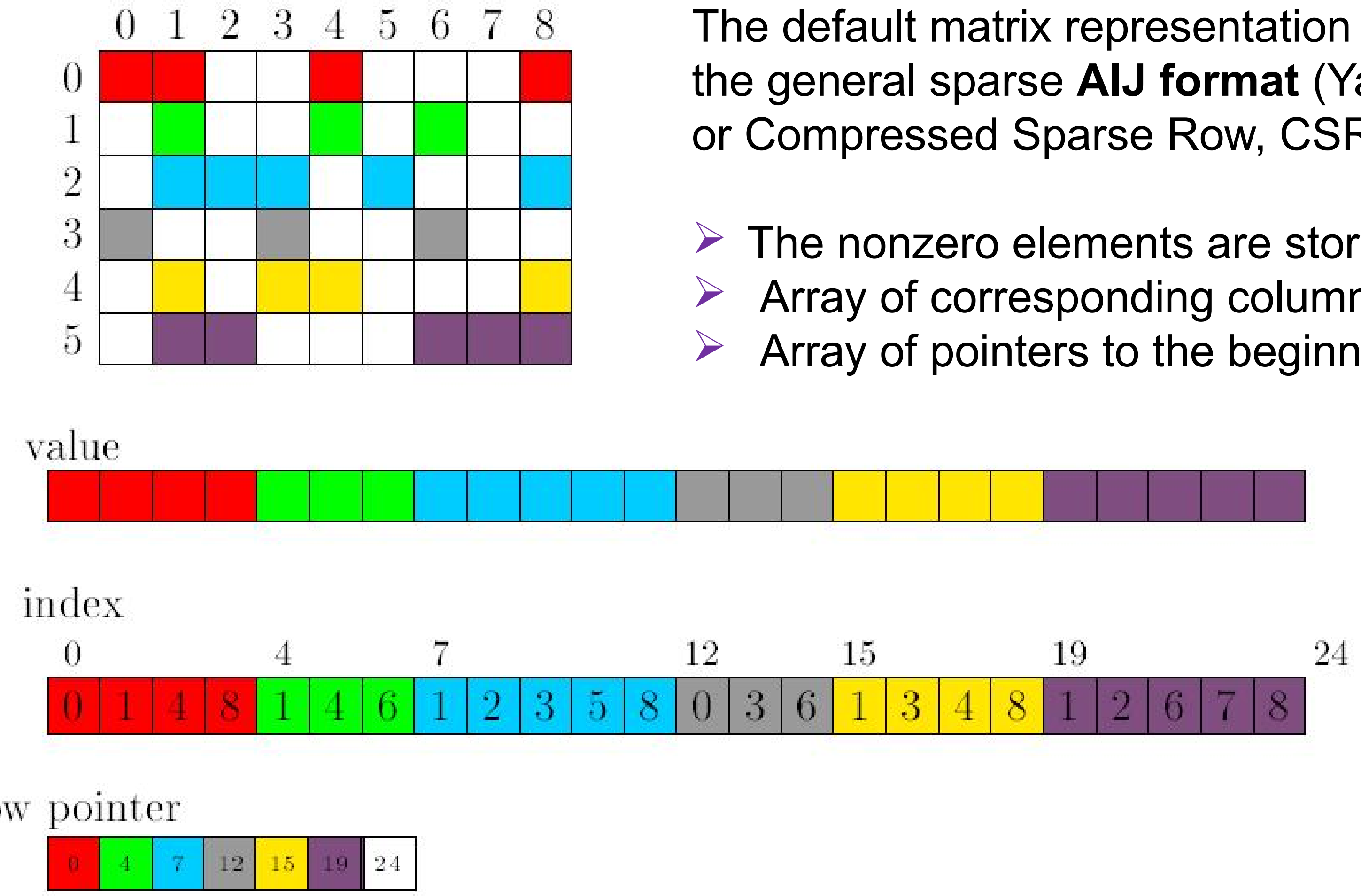
Matrix - Example

```
#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);
```

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 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

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.

Private representations of PETSc sparse matrices are dynamic data structures: **additional nonzeros can be freely added** (if no preallocation has been explicitly provided).

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 parallel sparse matrix (2/2)

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

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
```

PETSc solvers: KSP, SNES and TS

KSP: linear equations solvers

- **KSP** (K stands for *Krylov*) objects are used for solving linear systems by means of direct or iterative methods.
- In the iterative case, convergence can be improved by using a suitable **PC** object (preconditoner).
- Almost all iterative methods are implemented.
- Direct solution for parallel square matrices available through external solvers (MUMPS, SuperLU_dist)
- Linear operators set in KSP by using

```
KSPSetOperators(KSP ksp, Mat amat, Mat pmat,
                MatStructure matflag)
```

PETSc KSP methods

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

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

SNES: nonlinear solvers

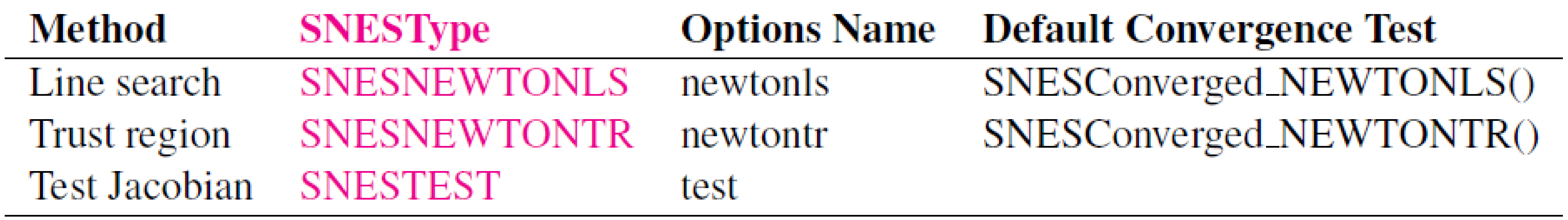
The SNES class includes methods for solving systems of nonlinear equations of the form

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

Newton-like methods provide the core of the package, including both line search and trust region techniques.

```
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)
```

PETSc SNES methods



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

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)

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.**

Profiling and performance tuning

Profiling:

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

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.