Intro to PETSc & SLEPc

2024 NSF CyberTraining Workshop
Jan. 8, 2024 – Jan. 19, 2024
Clarkson University

Note: The lecture slides are adapted from the Tutorial Slides from Argonne National Laboratory and updated to match PETSc v3.15

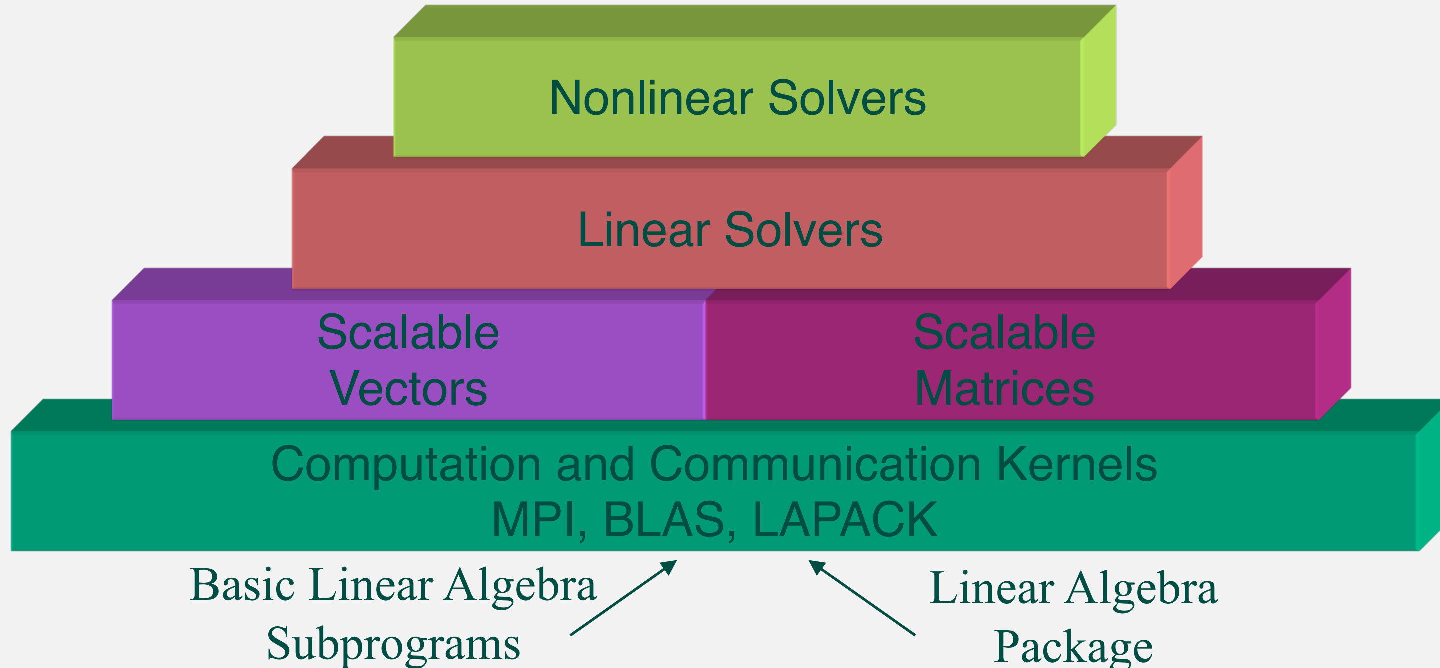
Topics

- Portable, Extensible Toolkit for Scientific Computing (PETSc)
- Scalable Library for Eigenvalue Problems (SLEPc)

What is PETSc

- A freely available and supported research code
  - Open Source, http://www.mcs.anl.gov/petsc
  - Usable from Fortran 77/90, C/C++
- Portable to any parallel system supporting MPI
- PETSc history: Begun in September 1991, and developed by Argonne National Laboratory
- PETSc funding and support: DOE & NSF
- PETSc is the world's most widely used parallel numerical software library for computation

Simple Structure of PETSc



Levels of Abstraction

- Application-specific interface
  - Programmer manipulates objects associated with the application
- High-level mathematics interface (e.g., SLEPc)
  - Programmer manipulates mathematical objects, such as PDEs and boundary conditions
- Algorithmic and discrete mathematics interface (e.g., PETSc)
  - Programmer manipulates mathematical objects (e.g., sparse matrices, nonlinear equations), algorithmic objects (e.g., solvers) and discrete geometry (e.g., meshes)
- Low-level computational kernels
  - e.g., BLAS (Basic Linear Algebra Subprograms) operations

The PETSc Programming Model

- **Goals**
  - Portable, runs everywhere with MPI
  - Performance
  - Scalable parallelism

- **Approach**
  - Distributed memory: Access to data on remote machines through MPI
  - Hide within parallel objects the details of the communication
  - User orchestrates communication at a higher abstract level than message passing

Collectivity

- MPI communicators (MPI_Comm) specify collectivity (processes involved in a computation)
- All PETSc routines for creating solver and data objects are collective with respect to a communicator, e.g.,
  - VecCreate(MPI_Comm comm, int m, int M, Vec *x)
  - Use **PETSC_COMM_WORLD** for all processes (like MPI_COMM_WORLD, but allows the same code to work when PETSc is started with a smaller set of processes)
- Some operations are collective, while others are not, e.g.,
  - collective: VecNorm() – computers the vector norm
  - not collective: VecGetLocalSize()
- If a sequence of collective routines is used, they **must** be called in the same order by each process.

Install PETSc on the Server

- git clone -b release https://gitlab.com/petsc/petsc.git petsc
- cd petsc
- ./configure
- make PETSC_DIR=/home/<u>yourhomefoldername</u>/petsc PETSC_ARCH=arch-linux-c-debug all
- make PETSC_DIR=/home/<u> yourhomefoldername </u>/petsc PETSC_ARCH=arch-linux-c-debug check

Sequential Hello World

```
#include "petsc.h"

int main( int argc, char *argv[] )
{
  PetscInitialize(&argc,&argv,PETSC_NULL,PETSC_NULL);
  /* Prints to standard out, only from the first processor in the communicator.  Calls from
other processes are ignored.*/
  PetscPrintf(PETSC_COMM_WORLD,"Hello World\n");
  PetscFinalize();
  return 0;
}
```

Parellel Hello World

```
#include “petsc.h”

int main( int argc, char *argv[] )
{
  int rank;
  PetscInitialize(&argc,&argv,PETSC_NULL,PETSC_NULL);
  MPI_Comm_rank(PETSC_COMM_WORLD,&rank );
  /* Prints synchronized output from several processors. Output of the first processor is followed by
that of the second, etc. */
  PetscSynchronizedPrintf(PETSC_COMM_WORLD, “Hello World from %d\n”,rank);
  /* Flushes to the screen output from all processor */
  PetscSynchronizedFlush(PETSC_COMM_WORLD, NULL);
  PetscFinalize();
  return 0;
}
```

Data Objects

- Scalable Vectors (Vec): Object Creation (VecCreate), Object Assembly (VecAssemblyBegin, VecAssemblyEnd), Type Setting (VecSetType), etc.
- Scalable Matrices (Mat): Object Creation (MatCreate), Object Assembly (MatAssemblyBegin, MatAssemblyEnd), Type Setting (MatSetType), etc.

Vectors

- What are PETSc vectors?
  - Fundamental objects for storing solutions, right-hand sides (e.g., Ax=b, b is RHS), etc.
  - Each process locally owns a subvector of contiguously numbered global indices
- Create vectors via
  - VecCreate(MPI_Comm,Vec *)
    - MPI_Comm - processes that share the vector
  - VecSetSizes(Vec, int, int)
    - number of elements local to this process
    - or total number of elements
  - VecSetType(Vec,VecType)
    - Where VecType is
      - VEC_SEQ (the sequential vector), VEC_MPI (the parallel vector)
    - VecSetFromOptions(Vec) lets you set the type at *runtime*

Creating a vector

```
Vec x;
int n;
…
PetscInitialize(&argc,&argv,(char*)0,help);
PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
…
VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE,n);
VecSetType(x,VEC_MPI);
VecSetFromOptions(x);
```

Use PETSc to get value from command line

PETSc determines local size

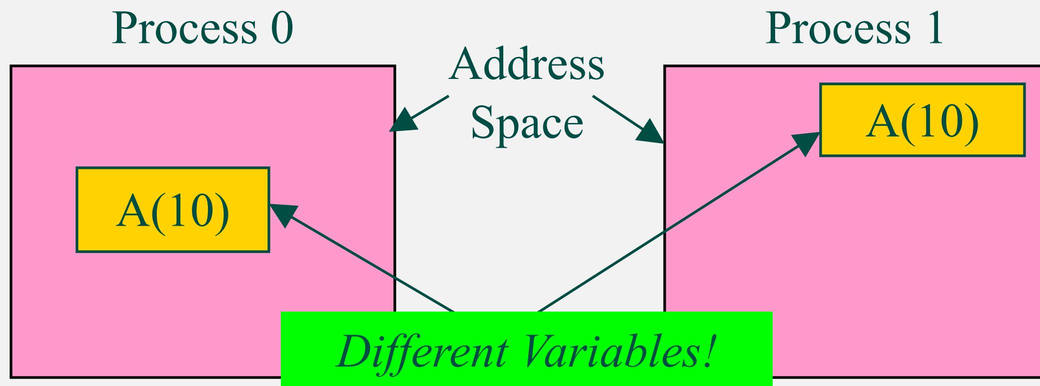
Global size

How Can We Use a PETSc Vector

- In order to support the distributed memory model, as well as single processors and shared memory systems, a PETSc vector is a “handle” to the real vector
  - Allows the vector to be distributed across many processes
  - To access the *elements* of the vector, we cannot simply do

    ```
    for (i=0; i<n; i++) v[i] = i;
    ```
  - We do not want to *require* that the programmer work only with the “local” part of the vector; we want to permit operations, such as setting an element of a vector, to be performed by any process.

Distributed Memory Model



- int A[10]

  printf(“%d\n”a[0]);
  …

- int A[10]
  for (i=0; i<10; i++){
    A[i] = i;
  }
  ...

This A is completely different from this one

Vector Assembly

- A three-step process
  - Each process tells PETSc what values to set or add to a vector component. Once *all* values provided,
  - Begin communication between processes to ensure that values end up where needed
  - (allow other operations, such as some computation, to proceed)
  - Complete the communication
- VecSetValues(Vec,…)
  - number of entries to insert/add
  - indices of entries
  - values to add
  - mode: [INSERT_VALUES,ADD_VALUES], where <u>ADD_VALUES</u> adds values to any existing entries, and <u>INSERT_VALUES</u> replaces existing entries with new values
- VecAssemblyBegin(Vec)
- VecAssemblyEnd(Vec)

Parallel Matrix and Vector Assembly

- Processes may generate any entries in vectors and matrices
- Entries need not be generated on the process on which they ultimately will be stored
- **PETSc automatically moves data during the assembly process if necessary**

One Way to Set the Elements of A Vector

```
VecGetSize(x,&N);  /* Global size */
MPI_Comm_rank( PETSC_COMM_WORLD, &rank );
if (rank == 0) {
    for (i=0; i<N; i++)
        VecSetValues(x,1,&i,&v,INSERT_VALUES);
}
/* These two routines ensure that the data is distributed to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

A Parallel Way to Set the Elements of A Distributed Vector

- VecGetOwnershipRange(x,&low,&high); /* return the range of indices owned by this process*/
  for (i=low; i<high; i++)
  VecSetValues(x,1,&i,&i,INSERT_VALUES);
  /* These two routines must be called (in case some other process contributed a value owned by another process) */
  VecAssemblyBegin(x);
  VecAssemblyEnd(x);

Selected Vector Operations

Function Name	Operation
VecAXPY(Scalar *a, Vec x, Vec y)	$y = y + a*x$
VecAYPX(Scalar *a, Vec x, Vec y)	$y = x + a*y$
VecWAXPY(Scalar *a, Vec x, Vec y, Vec w)	$w = a*x + y$
VecScale(Scalar *a, Vec x)	$x = a*x$
VecCopy(Vec x, Vec y)	$y = x$
VecPointwiseMult(Vec x, Vec y, Vec w)	$w_i = x_i *y_i$
VecMax(Vec x, int *idx, double *r)	$r = max\ x_i$
VecShift(Scalar *s, Vec x)	$x_i = s+x_i$
VecAbs(Vec x)	$x_i = \|x_i\|$
VecNorm(Vec x, NormType type , double *r)	$r = \|\|x\|\|$

A Complete PETSc Vec Program

```
#include "petsc.h"

int main(int argc,char **argv)
{
  Vec x;
  int n = 20,ierr;
  PetscScalar one = 1.0, dot;

  PetscInitialize(&argc,&argv,0,0);
  PetscOptionsGetInt(PETSC_NULL, PETSC_NULL, "-n",&n,PETSC_NULL);
  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,n);
  VecSetFromOptions(x);
  VecSet(x,one); /* sets all components of a vector to a single scalar value */
  VecDot(x,x,&dot);  /*computes the vector dot product*/
  PetscPrintf(PETSC_COMM_WORLD,"Vector length %d\n",(int)dot);
  VecDestroy(&x);
  PetscFinalize();
  return 0;
}
```

Working With Local Vectors

- It is sometimes needed to directly access the storage for the local part of a PETSc Vec.
- PETSc allows you to access the local storage with
  - VecGetArray(Vec, double *[]);
- You **must** return the array to PETSc when you are done with it
  - VecRestoreArray(Vec, double *[])
- This allows PETSc to handle and data structure conversions
  - For most common uses, these routines are inexpensive and do *not* involve a copy of the vector.

Example of VecGetArray

```
Vec      vec;
Double *avec;
…
VecCreate(PETSC_COMM_SELF,&vec);
VecSetSizes(vec,PETSC_DECIDE,n);
VecSetFromOptions(vec);
VecGetArray(vec,&avec);
/* compute with avec directly, e.g., */
PetscPrintf(PETSC_COMM_WORLD,
 “First element of local array of vec in each process is %f\n”, avec[0] );
VecRestoreArray(vec,&avec);
```

Matrices

- What are PETSc matrices?
  - Fundamental objects for storing data
- Create matrices via
  - MatCreate(…,Mat *)
    - MPI_Comm - processes that share the matrix
    - number of local/global rows and columns
  - MatSetType(Mat,MatType): *MPI* is for parallel matrices, and *SEQ* is for sequential ones
    - where MatType is one of
      - default sparse AIJ: MPIAIJ, SEQAIJ
      - block sparse AIJ: MPIAIJ, SEQAIJ
      - symmetric block sparse AIJ: MPISBAIJ, SAEQSBAIJ, only the upper triangular portion is stored
      - block diagonal: MPIBDIAG, SEQBDIAG
      - dense: MPIDENSE, SEQDENSE
    - MatSetFromOptions(Mat) lets you set the MatType at *runtime*.

Matrix Assembly

- Same form as for PETSc Vectors:
- MatSetValues(Mat,…)
  - number of rows to insert/add
  - indices of rows and columns
  - number of columns to insert/add
  - values to add
  - mode: [INSERT_VALUES,ADD_VALUES]
- MatAssemblyBegin(Mat)
- MatAssemblyEnd(Mat)

Matrix Assembly Example

```
Mat      A;
int        column[3], i;
double value[3];
…
MatCreate(PETSC_COMM_WORLD,
MatSetSizes(PETSC_DECIDE, PETSC_DECIDE, n, n, &A);

MatSetFromOptions(A);
MatSetUp(A);

/* mesh interior */
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
if (rank == 0) {  /* Only one process creates matrix */
    for (i=1; i<n-2; i++) {
        column[0] = i-1; column[1] = i; column[2] = i+1;
        MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
    }
}

/* also must set boundary points  (code for global row 0 and n-1 omitted) */
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

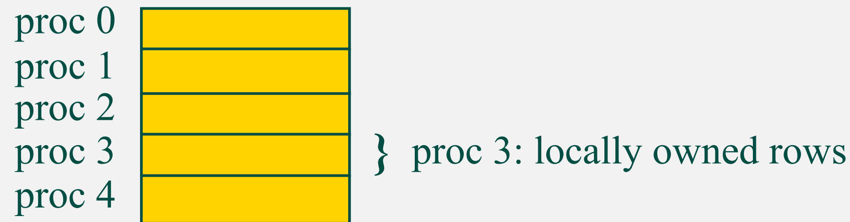
Choose the global
Size of the matrix

Let PETSc decide how
to allocate matrix
across processes

The matrix
# of rows & their global indices
# of columns & their global indices

Parallel Matrix Distribution

Each process locally owns a submatrix of contiguously numbered global rows.



MatGetOwnershipRange(Mat A, int *rstart, int *rend)

- rstart: first locally owned row of global matrix
- rend -1: last locally owned row of global matrix

Matrix Assembly Example With Parallel Assembly

```
Mat      A;
int       column[3], i, start, end,istart,iend;
double value[3];
…
MatCreate(PETSC_COMM_WORLD, &A);
MatSetSizes(A, PETSC_DECIDE,PETSC_DECIDE,n,n);

MatSetFromOptions(A);
MatSetUp(A);
MatGetOwnershipRange(A,&start,&end);

/* mesh interior */
istart = start; if (start == 0) istart = 1;
iend = end; if (iend == n) iend = n-1;
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=istart; i<iend; i++) {
   column[0] = i-1; column[1] = i; column[2] = i+1;
   MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
}
/* also must set boundary points  (code for global row 0 and n-1 omitted) */
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Choose the global Size of the matrix

One more than the global index of the last local row

Let PETSc decide how to allocate matrix across processes

The global index of the first low row

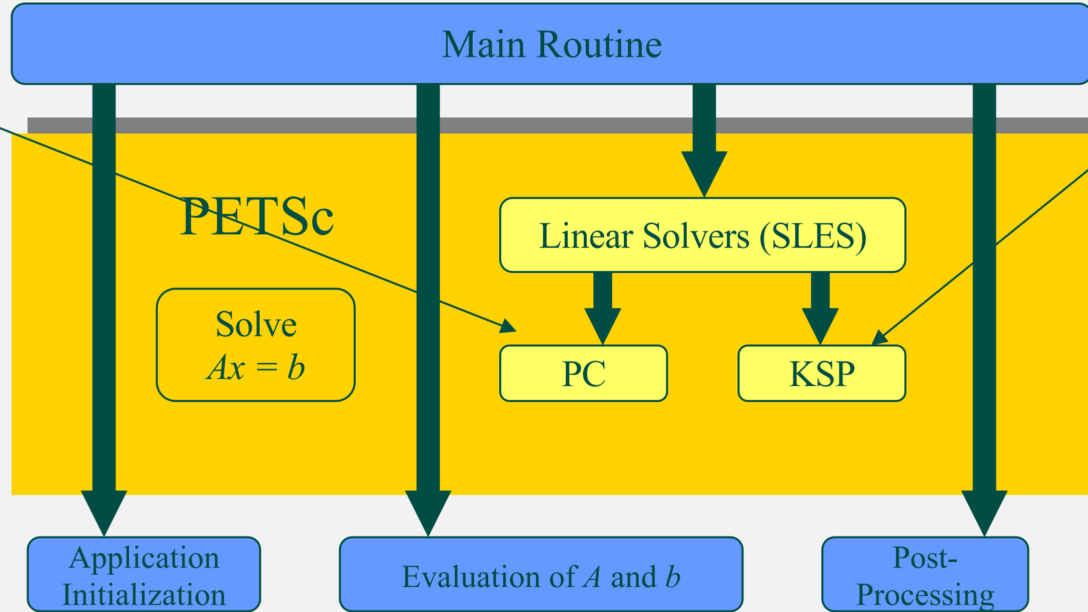
Viewing Vector/Matrix Fields

- VecView(Vec x,PetscViewer v); MatView(Mat A, PetscViewer v);

- Default viewers
  - ASCII (sequential): PETSC_VIEWER_STDOUT_SELF
  - ASCII (parallel): PETSC_VIEWER_STDOUT_WORLD
  - X-windows: PETSC_VIEWER_DRAW_WORLD
- Default ASCII formats
  - PETSC_VIEWER_ASCII_DEFAULT
  - PETSC_VIEWER_ASCII_MATLAB
  - PETSC_VIEWER_ASCII_COMMON
  - PETSC_VIEWER_ASCII_INFO

Linear Equation Solution

Preconditioning is the application of a transformation, called the preconditioner, that conditions a given problem into a form that is more suitable for numerical solving methods



Krylov Subspace Method: a family of iterative based algorithms for solving Ax=b that search for an approximate solution from a Krylov subspace

Linear Solvers

Goal: Support the solution of linear systems,

$$Ax=b,$$

particularly for sparse, parallel problems arising within PDE-based models

User provides:

- Code to evaluate A, b

Objects In PETSc

- How should a matrix be described in a program?
    - Mat M
- Hides the choice of data structure
    - Of course, the library still needs to represent the matrix with some choice of data structure, but this is an *implementation detail*
- Benefit
    - Programs become independent of particular choices of data structure, making it easier to modify and adapt

Krylov Method

- Unaffordable efforts for solving linear equation for large A matrix using the regular method: $x = A^{-1}b$
- Krylov Method: $$A^{-1}b \approx \sum_{i=0}^{m-1} \beta_i A^i b = \beta_0 b + \beta_1 Ab + \beta_2 A^2 b \ldots + \beta_{m-1} A^{m-1} b$$
- Get the coefficients: $$0 = b - Ax^{(m)} = b - A\sum_{i=0}^{m-1} \beta_i A^i b$$
- r(m) is the error $r^{(m)} = b - Ax^{(m)}$
- LSM: $$\frac{\partial r}{\partial \beta_0} = 0, \frac{\partial r}{\partial \beta_1} = 0, \frac{\partial r}{\partial \beta_2} = 0, \ldots \frac{\partial r}{\partial \beta_{m-1}} = 0$$

Operations In PETSc

- How should operations like “solve linear system” be described in a program?
    - KSPSolve(ksp, b, x, &its)
- Hides the choice of algorithm
    - Algorithms are to operations as data structures are to objects
- Benefit
    - Programs become independent of particular choices of algorithm, making it easier to explore algorithmic choices and to adapt to new methods.
- In PETSc, operations have their own handle, called a *“context variable”*

Context Variables

- Are the key to solver organization
- Contain the complete state of an algorithm, including
  - parameters (e.g., convergence tolerance)
  - functions that run the algorithm
  - information about the current state (e.g., iteration number)

Creating the SLES Context

- KSP ksp; KSPCreate(PETSC_COMM_WORLD,&ksp);
- Provides an identical user interface for all linear solvers
- Each KSP object actually contains two other objects:
  - KSP — Krylov Space Method
    - The iterative method
    - The context contains information on method parameters
  - PC — Preconditioners
    - Knows how to apply a preconditioner
    - The context contains information on the preconditioner, such as what routine to call to apply it

Basic Functions for KSP

- KSPCreate( ) - Create KSP context
- KSPSetOperators( ) - Set linear operators
- KSPSetFromOptions( ) - Set runtime solver options for [SLES, KSP,PC]
- KSPSolve( ) - Run linear solver
- KSPView( ) - View solver options actually used at runtime (alternative: -sles_view)
- KSPDestroy( ) - Destroy solver
- KSPGetPC() - Get preconditioner
- PCSetType() - Set preconditioner
- KSPSetTolerance() - Set convergence tolerance

Basic Linear Solver Code

```
SLES   sles;        /*  linear solver context  */
Mat    A;           /*  matrix  */
Vec    x, b;        /*  solution, RHS vectors  */
int    n, its;      /*  problem dimension, number of iterations  */

MatCreate(PETSC_COMM_WORLD, &A)
MatSetSizes(A, PETSC_DECIDE,PETSC_DECIDE,n,n);
MatSetFromOptions(A);
/* (code to assemble matrix not shown) */
VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE, n);
VecSetFromOptions(x);
VecDuplicate(x,&b);
/* (code to assemble RHS vector not shown)*/

KSPCreate(PETSC_COMM_WORLD,&ksp);
KSPSetOperators(ksp,A,A);
KSPSetFromOptions(ksp);
KSPSolve(ksp,b,x);
KSPDestroy(ksp);
```

The matrix that defines the linear system, and the matrix to be used in constructing the preconditioner, usually the same as the first one

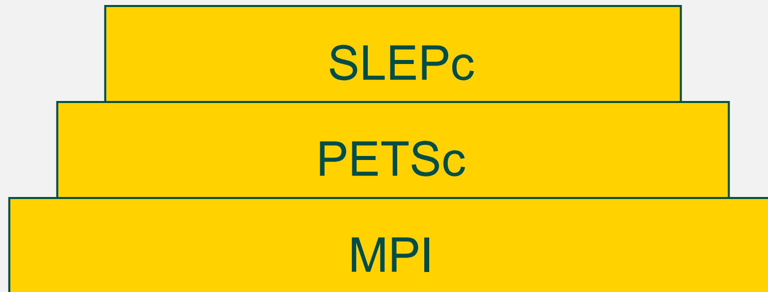
Note: you will work on the complete codes in the PM lab session

Topics

- Portable, Extensible Toolkit for Scientific Computing (PETSc)
- Scalable Library for Eigenvalue Problems (SLEPc)

SLEPc Overview

SLEPc: Scalable Library for Eigenvalue Problem Comptations. A general library for solving large-scale sparse eigenproblems on parallel computers.



High-level mathematics interface

Algorithmic and discrete mathematics interface

Low-level computational kernels

SLEPc Installation

- Download SLEPc v3.15: https://slepc.upv.es/download/, and untar by tar *xzf slepc-3.15.0.tar.gz*
- *export SLEPC_DIR=/home/username/slepc-3.15.0*
- *export PETSC_DIR=/home/username/petsc*
- *export PETSC_ARCH=arch-linux-c-debug*
- *./configure*
- *make SLEPC_DIR=/home/username/slepc-3.15.0 PETSC_DIR=/home/username/petsc PETSC_ARCH=arch-linux-c-debug*
- *make check*

Basic Functions for SLEPc

SlepcInitialize is to initializes SLEPc, PETSc and MPI.

```
SlepcInitialize(&argc,&argv,(char*)0,help);
```

SLEPC is based on PETSc. Thus, SLEPc uses the same functions for matrix and vector operation.

```
Mat           A;          /* problem matrix */
PetscReal     error, tol, re, im;
PetscScalar   kr, ki;
Vec           xr, xi;
PetscInt      n=30,i,Istart,Iend,nev,maxit,its,nconv;
MatCreate(PETSC_COMM_WORLD,&A);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);
MatSetFromOptions(A);
MatSetUp(A);
```

Solving Linear Eigenvalue Problem (EPS)

Compute eigenvalue pair (x, λ) for a standard eigenvalue problem.

$$Ax = \lambda x$$

EPS: Eigenvalue Problem Solver

```
EPS        eps;     /* eigenproblem solver context */
EPSType    type;
```

Create eigen solver context

```
EPSCreate(PETSC_COMM_WORLD,&eps);
```

Set operators

```
EPSSetOperators(eps,A,NULL);
EPSSetProblemType(eps,EPS_HEP);
```

Solve eigensystems

```
EPSSolve(eps);
```

Cont. Solving Linear Eigenvalue Problem

Get some information from the solver.

- The number of iteration EPSGetIterationNumber(eps,&its);
- The type of problem EPSGetType(eps,&type);
- The number of requested eigenvalue EPSGetDimensions(eps, &nev, NULL, NULL);
- Get converged eigenpairs: i^{th} eigenvalue is stored in kr (real part) and ki (imaginary part)
  i^{th} eigen vector is stored in Vr and Vi
  EPSGetEigenpair(eps,i,&kr,&ki,xr,xi);

A complete example

- eps.c: Solve eigenvalue pair (x, λ) for the 9x9 matrix A
- Building script: mpicc -g -I/home/yuliu/petsc/include -I/home/yuliu/petsc/arch-linux-c-debug/include -I/home/yuliu/slepc-3.15.0/include -I/home/yuliu/slepc-3.15.0/arch-linux-c-debug/include eps.c -Wl,-rpath,/home/yuliu/petsc/arch-linux-c-debug/lib -rpath,/home/yuliu/slepc-3.15.0/arch-linux-c-debug/lib -L/home/yuliu/petsc/arch-linux-c-debug/lib -L/home/yuliu/slepc-3.15.0/arch-linux-c-debug/lib -lpetsc -lmpi -lslepc -o eps