Introduction to SLEPc, the Scalable Library for Eigenvalue Problem Computations

Jose E. Roman

Universidad Politécnica de Valencia, Spain

(joint work with Andres Tomas)

Porto, June 27-29, 2007
Outline

1. Introduction
2. Overview of PETSc/SLEPc
3. Basic PETSc Usage
4. Basic SLEPc Usage
   - Eigenvalue Solvers
   - Spectral Transformation

Hands-on Exercises

Coffee Break

Hands-on Exercises
Eigenvalue Problems

Consider the following eigenvalue problems

**Standard Eigenproblem**
\[ Ax = \lambda x \]

**Generalized Eigenproblem**
\[ Ax = \lambda Bx \]

where

- \( \lambda \) is a (complex) scalar: *eigenvalue*
- \( x \) is a (complex) vector: *eigenvector*
- Matrices \( A \) and \( B \) can be real or complex
- Matrices \( A \) and \( B \) can be symmetric (Hermitian) or not
- Typically, \( B \) is symmetric positive (semi-) definite
Solution of the Eigenvalue Problem

There are $n$ eigenvalues (counted with their multiplicities)

Partial eigensolution: $nev$ solutions

\[
\lambda_0, \lambda_1, \ldots, \lambda_{nev-1} \in \mathbb{C} \\
x_0, x_1, \ldots, x_{nev-1} \in \mathbb{C}^n
\]

$nev = \text{number of eigenvalues} / \text{eigenvectors}$ (eigenpairs)

Different requirements:

- Compute a few of the dominant eigenvalues (largest magnitude)
- Compute a few $\lambda_i$’s with smallest or largest real parts
- Compute all $\lambda_i$’s in a certain region of the complex plane
Spectral Transformation

A general technique that can be used in many methods

\[ Ax = \lambda x \quad \Rightarrow \quad Tx = \theta x \]

In the transformed problem
- The eigenvectors are not altered
- The eigenvalues are modified by a simple relation
- Convergence is usually improved (better separation)

**Shift of Origin**
\[ T_S = A + \sigma I \]

**Shift-and-invert**
\[ T_{SI} = (A - \sigma I)^{-1} \]

**Cayley**
\[ T_C = (A - \sigma I)^{-1}(A + \tau I) \]

Drawback: \( T \) not computed explicitly, linear solves instead
Design Considerations

- Various problem characteristics: Problems can be real/complex, Hermitian/non-Hermitian
- Many ways of specifying which solutions must be sought
- Many formulations: not all eigenproblems are formulated as simply $Ax = \lambda x$ or $Ax = \lambda Bx$

Goal: provide a uniform, coherent way of addressing these problems

- Internally, solvers can be quite complex (deflation, restart, ...)
- Spectral transformations can be used irrespective of the solver
- Repeated linear solves may be required

Goal: hide eigensolver complexity and separate spectral transform
What Users Need

Provided by PETSc

- Abstraction of mathematical objects: vectors and matrices
- Efficient linear solvers (direct or iterative)
- Easy programming interface
- Run-time flexibility, full control over the solution process
- Parallel computing, mostly transparent to the user

Provided by SLEPc

- State-of-the-art eigensolvers
- Spectral transformations
Summary

**PETSc**: Portable, Extensible Toolkit for Scientific Computation

Software for the scalable (parallel) solution of algebraic systems arising from partial differential equation (PDE) simulations

- Developed at Argonne National Lab since 1991
- Usable from C, C++, Fortran77/90
- Focus on abstraction, portability, interoperability
- Extensive documentation and examples
- Freely available and supported through email

http://www.mcs.anl.gov/petsc

Current version: **2.3.3** (released May 2007)
PETSc Concepts

PETSc offers tools to facilitate development of parallel PDE solvers
However, it is not a black-box nor a silver bullet

PETSc concepts:
- Expressing the mathematics of the problem: matrices, vectors
- Problem solving: linear solvers, nonlinear, time stepping
- Parallelism: structured and unstructured grids
- Tools: index sets, scatter contexts
- Program development: debugging, profiling, and basic visualization support
Structure of PETSc

- Computation and Communication Kernels
  - MPI, MPI-IO, BLAS, LAPACK
- Profiling Interface
- PETSc Application Codes
  - ODE Integrators
  - Visualization Interface
  - Nonlinear Solvers
  - Interface
  - Linear Solvers
  - Preconditioners + Krylov Methods
  - Grid Management
- Matrices, Vectors, Indices
- Visualization Interface
- Grid Management
- PETSc Structure
- PETSc Application Codes
# PETSc/SLEPc Numerical Components

## PETSc

<table>
<thead>
<tr>
<th>Nonlinear Systems</th>
<th>Time Steppers</th>
<th>Preconditioners</th>
<th>Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Search</td>
<td>Euler</td>
<td>Additive Schwarz</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>Trust Region</td>
<td>Backward Euler</td>
<td>Block Jacobi</td>
<td>Block Compressed Sparse Row</td>
</tr>
<tr>
<td>Other</td>
<td>Pseudo Time Step</td>
<td>Jacobi</td>
<td>Block Diagonal</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>ILU</td>
<td>Dense</td>
</tr>
</tbody>
</table>

## SLEPc

<table>
<thead>
<tr>
<th>SVD Solvers</th>
<th>Eigensolvers</th>
<th>Spectral Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross Product</td>
<td>Krylov-Schur</td>
<td>Shift</td>
</tr>
<tr>
<td>Cyclic Matrix</td>
<td>Arnoldi</td>
<td>Shift-and-invert</td>
</tr>
<tr>
<td>Lanczos</td>
<td>Lanczos</td>
<td>Cayley</td>
</tr>
<tr>
<td>Thick Res. Lanczos</td>
<td>Other</td>
<td>Fold</td>
</tr>
</tbody>
</table>

## Krylov Subspace Methods

- GMRES
- CG
- CGS
- Bi-CGStab
- TFQMR
- Richardson
- Chebychev
- Other

## Preconditioners

- Additive Schwarz
- Block Jacobi
- Jacobi
- ILU
- ICC
- LU
- Other

## Matrices

- Compressed Sparse Row
- Block Compressed Sparse Row
- Block Diagonal
- Dense
- Other

## Index Sets

- Indices
- Block Indices
- Stride
- Other
What does PETSc not do?

Not provided by PETSc (yet):

- Discretizations
- Unstructured mesh generation and refinement tools
- Load balancing tools
- Sophisticated visualization

However, external packages:

- ODE integration: Sundials
- Mesh partitioning: Parmetis, Chaco, ...
- Mesh refinement: SAMRAI
- Optimization: TAO
- Linear solvers/preconditioners: SuperLU, MUMPS, Hypre, ...
Parallelism Model

Goals:

- Portable, runs everywhere
- Performance
- Scalable parallelism

Approach:

- Fully MPI-based
- ‘Nothing shared’, however OpenMP and such still possible
- All objects are created with respect to a communicator
- Hide within objects the details of the communication
- Some routines are collective (e.g. VecNorm), some not (e.g. VecGetLocalSize)
The Simplest Example

```c
#include "petsc.h"

int main( int argc, char *argv[] )
{
    PetscInitialize( &argc, &argv, NULL, NULL );
    PetscPrintf( PETSC_COMM_WORLD, "Hello World\n");
    PetscFinalize();
    return 0;
}
```
PETSc Objects

PETSc functionality is structured around objects

“Data structure-neutral” objects

- The programmer works with a “handle” to the object

Typical operations:

- Creation: VecCreate, VecDuplicate
- Operations: MatMult, PCApply
- Viewing objects: MatView
- Deletion: MatDestroy
Vectors

Vectors are fundamental objects for storing field solutions, right-hand sides, eigenvectors, etc.

- Construction: sequential or parallel
- Element specification: `VecSetValues`, either insert or add
- Elements specified by global index: need not be on the local processor
- Finish construction with `VecAssemblyBegin/End`: elements get moved to the appropriate processor
Vector Parallel Layout

Each process locally owns a subvector of contiguously numbered global indices

- simple block-row division of vectors (carries over to matrices)
- other numberings through permutation
Matrices: Creation

MatCreate(PETSC_COMM_WORLD,&A);
MatSetType(A,MATMPIAIJ);
MatSetSizes(A,m,n,M,N); // PETSC_DECIDE allowed
MatSetPreallocation(A,d,o,dd,oo);
for (i)
    for (j)
        MatSetValues(A,ni,i,nj,j,v,INSERT/ADD_VALUES);

- Sequential or parallel, various internal formats (AIJ, block, dense, etc. and external packages)
- Sizes can be specified globally or locally, simple block row partitioning
- Allocation: dynamic or user-specified
- Elements can be set anywhere (MatAssemblyBegin/End)
Matrices: Usage

- Polymorphism: MatMult(A, x, y);
- Matrix-free operations:
  MatShellSetOperation(A, MATOP_MUL, my_matmult);
- Viewers: screen viewing, binary dump, output to matlab, ...
Linear Solvers

KSPCreate(PETSC_COMM_WORLD,&solver);
KSPSetOperators(solver,A,...);
KSPSetType(solver,KSPBCGS);
KSPGetPC(solver,&prec);
PCSetType(prec,PCILU);
KSPSolve(solver,b,x);

- Polymorphism: calls independent of solver, preconditioner, matrix type
- Many popular solvers implemented, others available through external packages
- User specified monitors and convergence test
- Direct solvers implemented as preconditioner application
Introduction
Overview of PETSc/SLEPc
Basic PETSc Usage
Basic SLEPc Usage

Runtime Control of Settings

- Instruct PETSc to incorporate command-line options:
  KSPSetFromOptions

- Use command-line options:
  -ksp_type gmres -pc_type ilu -pc_ilu_levels 3

- Options can be used for many other purposes:
  -ksp_monitor -ksp_rtol 1e-8

- User can introduce new options
  PetscOptionsGetInt("-my_option", &myint,...);
Other PETSc Functionality

- Non-linear equation solvers (*SNES*), including finite difference Jacobian computation and support for automatic differentiation
- Time-stepping methods (built-in, Sundials)
- Structured and unstructured grid handling
- Multigrid (built-in, ML)
- Parallel direct linear solvers (*MUMPS*, *SuperLU*, ...)
- Debugging and profiling
SLEPc: Scalable Library for Eigenvalue Problem Computations

A general library for solving large-scale sparse eigenproblems on parallel computers

- For standard and generalized eigenproblems
- For real and complex arithmetic
- For Hermitian or non-Hermitian problems

Current version: 2.3.3 (released June 2007)

http://www.grycap.upv.es/slepc
Structure of SLEPc

SLEPc extends PETSc with new objects: **EPS, ST, SVD**

**EPS: Eigenvalue Problem Solver**
- The user specifies the problem via this object
- Provides a collection of eigensolvers
- Allows the user to specify a number of parameters (e.g. which portion of the spectrum)

**ST: Spectral Transformation**
- Used to transform the original problem into $T x = \theta x$
- Always associated to an EPS object, not used directly
Basic Usage

Usual steps for solving an eigenvalue problem with SLEPc:

1. Create an EPS object
2. Define the eigenvalue problem
3. (Optionally) Specify options for the solution
4. Run the eigensolver
5. Retrieve the computed solution
6. Destroy the EPS object

All these operations are done via a generic interface, common to all the eigensolvers
Simple Example

```c
EPS eps;    /* eigensolver context */
Mat A, B;    /* matrices of Ax=kBx */
Vec xr, xi;  /* eigenvector, x */
PetscScalar kr, ki; /* eigenvalue, k */

EPSCreate(PETSC_COMM_WORLD, &eps);
EPSSetOperators(eps, A, B);
EPSSetProblemType(eps, EPS_GNHEP);
EPSSetFromOptions(eps);
EPSSolve(eps);

EPSGetConverged(eps, &nconv);
for (i=0; i<nconv; i++) {
    EPSGetEigenpair(eps, i, &kr, &ki, xr, xi);
}
EPSDestroy(eps);
```
Details: Solving the Problem

EPSSolve(EPS eps)
Launches the eigensolver

Currently available eigensolvers:

- Power Iteration and RQI
- Subspace Iteration with Rayleigh-Ritz projection and locking
- Arnoldi method with explicit restart and deflation
- Lanczos method with explicit restart and deflation
  - Reorthogonalization: Local, Partial, Periodic, Selective, Full
- Krylov-Schur (default)

Also interfaces to external software: ARPACK, PRIMME, ...
Details: Problem Definition

EPSSetOperators(EPS eps, Mat A, Mat B)
Used for passing the matrices that constitute the problem

- A generalized problem \(Ax = \lambda Bx\) is specified by \(A\) and \(B\)
- For a standard problem \(Ax = \lambda x\) set \(B = \text{PETSC\_NULL}\)

EPSSetProblemType(EPS eps,EPSProblemType type)
Used to indicate the problem type

<table>
<thead>
<tr>
<th>Problem Type</th>
<th>EPSProblemType</th>
<th>Command line key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermitian</td>
<td>EPS_HEP</td>
<td>-eps_hermitian</td>
</tr>
<tr>
<td>Generalized Hermitian</td>
<td>EPS_GHEP</td>
<td>-eps_gen_hermitian</td>
</tr>
<tr>
<td>Non-Hermitian</td>
<td>EPS_NHEP</td>
<td>-eps_non_hermitian</td>
</tr>
<tr>
<td>Generalized Non-Herm.</td>
<td>EPS_GNHEP</td>
<td>-eps_gen_non_hermitian</td>
</tr>
</tbody>
</table>
Overview of PETSc/SLEPc

Basic PETSc Usage

Basic SLEPc Usage

Details: Specification of Options

**EPSSetFromOptions(EPS eps)**

Looks in the command line for options related to EPS

For example, the following command line

% program -eps_hermitian

is equivalent to a call `EPSSetProblemType(eps,EPS_HEP)`

Other options have an associated function call

% program -eps_nev 6 -eps_tol 1e-8

**EPSView(EPS eps, PetscViewer viewer)**

Prints information about the object (equivalent to `-eps_view`
Details: Viewing Current Options

Sample output of `-eps_view`

EPS Object:
- problem type: symmetric eigenvalue problem
- method: lanczos
- reorthogonalization: selective
- selected portion of spectrum: largest eigenvalues in magnitude
- number of eigenvalues (nev): 1
- number of column vectors (ncv): 16
- maximum number of iterations: 100
- tolerance: 1e-07
- orthogonalization method: classical Gram-Schmidt
- orthogonalization refinement: if needed (eta: 0.500000)
- dimension of user-provided deflation space: 0

ST Object:
- type: shift
- shift: 0
Run-Time Examples

% program -eps_view -eps_monitor

% program -eps_type power -eps_nev 6 -eps_ncv 24

% program -eps_type arnoldi -eps_tol 1e-8 -eps_max_it 2000

% program -eps_type subspace -eps_hermitian -log_summary

% program -eps_type lapack

% program -eps_type arpack -eps_plot_eigs -draw_pause -1

% program -eps_type blzpack -eps_smallest_real
Built-in Support Tools

- Plotting computed eigenvalues
  
  % program -eps_plot_eigs

- Printing profiling information
  
  % program -log_summary

- Debugging
  
  % program -start_in_debugger
  % program -malloc_dump
Built-in Support Tools

- Monitoring convergence (textually)
  % program -eps_monitor

- Monitoring convergence (graphically)
  % program -draw_pause 1
  -eps_monitor_draw
Spectral Transformation in SLEPc

An ST object is always associated to any EPS object

\[ Ax = \lambda x \quad \Rightarrow \quad Tx = \theta x \]

- The user need not manage the ST object directly
- Internally, the eigensolver works with the operator \( T \)
- At the end, eigenvalues are transformed back automatically

<table>
<thead>
<tr>
<th>ST</th>
<th>Standard problem</th>
<th>Generalized problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>( A + \sigma I )</td>
<td>( B^{-1}A + \sigma I )</td>
</tr>
<tr>
<td>fold</td>
<td>( (A + \sigma I)^2 )</td>
<td>( (B^{-1}A + \sigma I)^2 )</td>
</tr>
<tr>
<td>sinvert</td>
<td>( (A - \sigma I)^{-1} )</td>
<td>( (A - \sigma B)^{-1}B )</td>
</tr>
<tr>
<td>cayley</td>
<td>( (A - \sigma I)^{-1}(A + \tau I) )</td>
<td>( (A - \sigma B)^{-1}(A + \tau B) )</td>
</tr>
</tbody>
</table>
Illustration of Spectral Transformation

**Spectrum folding**

\[ \theta = (\lambda - \sigma)^2 \]

**Shift-and-invert**

\[ \theta = \frac{1}{\lambda - \sigma} \]
Accessing the ST Object

The user does not create the ST object

**EPSGetST(EPS eps, ST *st)**

Gets the ST object associated to an EPS

Necessary for setting options in the source code

**Linear Solves.** All operators contain an inverse (except $B^{-1}A + \sigma I$ in the case of a standard problem)

- Linear solves are handled internally via a KSP object

**STGetKSP(ST st, KSP *ksp)**

Gets the KSP object associated to an ST

All KSP options are available, by prepending the `-st_` prefix
More Run-Time Examples

```plaintext
% program -eps_type power -st_type shift -st_shift 1.5

% program -eps_type power -st_type sinvert -st_shift 1.5

% program -eps_type power -st_type sinvert
eps_power_shift_type rayleigh

% program -eps_type arpack -eps_tol 1e-6
  -st_type sinvert -st_shift 1
  -st_ksp_type cgs -st_ksp_rtol 1e-8
  -st_pc_type sor -st_pc_sor_omega 1.3
```
SLEPc Highlights

- Growing number of eigensolvers
- Seamlessly integrated spectral transformation
- Easy programming with PETSc’s object-oriented style
- Data-structure neutral implementation
- Run-time flexibility, giving full control over the solution process
- Portability to a wide range of parallel platforms
- Usable from code written in C, C++ and Fortran
- Extensive documentation
Thanks!

http://www.grycap.upv.es/slepc
slepc-maint@grycap.upv.es