

Hands-On Exercises for SLEPc

February 2009
Intended for use with version 3.0.0 of SLEPc

These exercises introduce application development in SLEPc, the Scalable Library for Eigenvalue Problem Computations. A basic knowledge of PETSc is assumed. The first exercise (exercise 0) is just a guided tour on how to get started compiling and running programs. Exercises 1, 2, 3, and 4 are intended to cover most of the basic SLEPc functionality. The rest of the exercises illustrate more advanced features. All the example programs used in the exercises are included in the SLEPc distribution and are also available at its [web site](#).

- Exercise 0:** [Hello World](#)
- Exercise 1:** [Standard Symmetric Eigenvalue Problem](#)
- Exercise 2:** [Standard Non-Symmetric Eigenvalue Problem](#)
- Exercise 3:** [Generalized Eigenvalue Problem Stored in a File](#)
- Exercise 4:** [Singular Value Decomposition](#)
- Exercise 5:** [Problem without Explicit Matrix Storage](#)
- Exercise 6:** [Parallel Execution](#)
- Exercise 7:** [Use of Deflation Subspaces](#)

For reference, detailed information on usage of SLEPc and PETSc may be found at the following links:

- [SLEPc on-line documentation](#)
- [PETSc on-line documentation](#)

Exercise 0: Hello World

This exercise shows how to build and run a simple example program with SLEPc.

Note: The description below related to directories and the use of the `PETSC_ARCH` variable will be different in the case of a prefix-based installation.

Compiling

SLEPc needs the following environment variables to be set:

`SLEPC_DIR` - the location of SLEPc
`PETSC_DIR` - the location of PETSc
`PETSC_ARCH` - the architecture being used

Make sure that you have them correctly set.

Like in PETSc, a makefile is necessary to compile a SLEPc program. Paste this simple example into a file named `makefile` in your working directory:

```
hello: hello.o chkopts
    -${CLINKER} -o hello hello.o ${SLEPC_LIB}
    ${RM} hello.o

include ${SLEPC_DIR}/conf/slepcc_common
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Also place the following source code into a file named "hello.c" in the same directory:

```
static char help[] = "Simple Hello World example program in SLEPc\n";

#include "slepceps.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main( int argc, char **argv )
{
    int ierr;

    SlepcInitialize(&argc,&argv,(char*)0,help);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Hello world\n");CHKERRQ(ierr);
    ierr = SlepcFinalize();CHKERRQ(ierr);
    return 0;
}
```

Compile the program with the following command:

```
% make hello
```

Source Code Details

Examine the source code of the sample program. The following comments will help you understand the code thoroughly.

Header File: All SLEPc programs must include a header file with all the necessary definitions. This file is typically `slepceps.h` (the include file for the EPS component), although in this simple example the file `slepc.h` (base SLEPc header) would be enough.

Note: SLEPc header files automatically include some PETSc header files.

Library Initialization: All SLEPc programs must begin with a call to `SlepcInitialize`, which in turn initializes PETSc and MPI. Similarly, at the end of the program `SlepcFinalize` must be called for library cleanup.

Input/Output: In this example, we do input/output via a call to a PETSc function, `PetscPrintf`. Remember that in parallel programs input/output cannot be done simply via C standard library functions. Note that in SLEPc programs we can freely use any PETSc function.

Error Checking: All SLEPc routines return an integer indicating whether an error has occurred during the call. The PETSc macro `CHKERRQ` checks the value of `ierr` and calls the PETSc error handler upon error detection. `CHKERRQ(ierr)` should be placed after all function calls to enable a complete error traceback. Also, the `__FUNCT__` macro should be redefined before each user function so that the error handler can help us locate the error.

Running the Program

SLEPc programs are executed as any other MPI program. Note that this typically differs from one system to another. To run the program with only one processor, in some systems you can launch it as a normal program:

```
% hello
```

but in other systems this would not work. Check the documentation of your system. In IBM SP systems, you should use the `poe` command as in

```
% poe hello
```

or

```
% poe hello -procs 4
```

for executing with more than one processor. Other MPI implementations require the `mpirun` command to launch the applications

```
% mpirun -np 4 hello
```

In SLEPc (and PETSc) there are a lot of options (run-time parameters) to control program behavior. These options are usually equivalent to function calls, so the user can test its effect without changing the source code (this will be illustrated in the next exercises). To show which options are available in a program use:

```
% hello -help
```

Support for Debugging and Complex Numbers

The support for debugging capabilities, complex scalar arithmetic, and other features is managed by SLEPc and PETSc by means of different *architectures*, represented by different values of the `PETSC_ARCH` variable. In a given system, you can typically find several versions of SLEPc and PETSc, each of them built with different configuration options. For instance, suppose the following values are available:

- `rs6000_sp_o`: built with compiler optimization
- `rs6000_sp_g`: built with debugging support
- `rs6000_sp_o_complex`: optimized with complex scalars
- `rs6000_sp_g_complex`: debug with complex scalars

Note: In order to learn about the particular architectures available in your system, type `ls $SLEPC_DIR`. There should be a subdirectory for each allowed value of `PETSC_ARCH`.

When using an architecture with support for complex scalars, all scalar values are complex instead of real. Try compiling the example program for complex numbers:

```
% make PETSC_ARCH=rs6000_sp_o_complex hello
```

When using the debug versions some options are available to support debugging. For example

```
% hello -start_in_debugger
```

opens the program in a debugger stopped at the `SlepcInitialize` function.

Other useful options are: `-info` to get informative messages about progress of the calculations, `-malloc_info` to print memory usage at end of run, `-log_trace [filename]` to get a full trace of the execution (in a file), `-malloc_dump` to list memory blocks not freed at the end of the program, and `-log_summary` to get a summary including performance results.

Exercise 1: Standard Symmetric Eigenvalue Problem

This example solves a standard symmetric eigenproblem $Ax=\lambda x$, where A is the matrix resulting from the discretization of the Laplacian operator in 1 dimension by centered finite differences.

$$\begin{vmatrix} 2 & -1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

$$A = \begin{vmatrix} -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{vmatrix}$$

Compiling

Copy the file [ex1.c \[plain text\]](#) to the working directory and add these lines to the makefile

```
ex1: ex1.o chkopts
    -${CLINKER} -o ex1 ex1.o ${SLEPC_LIB}
    ${RM} ex1.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex1
```

Note for Fortran users: Example ex1 is also available in Fortran [ex1f.F \[plain text\]](#).

Running the Program

In order to run the program for a problem of order 20, type the following

```
% ex1 -n 20
```

You should get an output similar to this

```
1-D Laplacian Eigenproblem, n=20

Number of iterations of the method: 2
Solution method: krylovschur

Number of requested eigenvalues: 1
Stopping condition: tol=1e-07, maxit=100
Number of converged eigenpairs: 1

-----
      k          ||Ax-kx|| / ||kx||
-----
    3.977662    1.69774e-08
```

Source Code Details

Examine the source code of the sample program and locate the function calls mentioned in the following comments.

The Options Database: All the PETSc functionality related to the options database is available in SLEPc. This allows the user to input control data at run time very easily. In this example, the function [PetscOptionsGetInt](#) is used to check whether the user has provided a command line option to set the value of n, the problem dimension. If so, the variable n is set accordingly; otherwise, n remains unchanged.

Vectors and Matrices: Usage of matrices and vectors in SLEPc is exactly the same as in PETSc. The user can create a new parallel or sequential matrix, A, with subroutine [MatCreate](#), where the matrix format can be specified at runtime. The example creates a matrix, sets the nonzero

values with `MatSetValues` and then assembles it.

Solving the Eigenvalue Problem: Usage of eigensolvers is very similar to other kinds of solvers provided by PETSc. After creating the matrix, the problem is solved by means of an EPS object (Eigenvalue Problem Solver) via the following sequence of function calls:

```
EPSCreate(MPI_Comm comm, EPS *eps);
EPSSetOperators(EPS eps, Mat A, Mat B);
EPSSetProblemType(EPS eps, EPSProblemType type);
EPSSetFromOptions(EPS eps);
EPSSolve(EPS eps);
EPSGetConverged(EPS eps, int *nconv);
EPSGetEigenpair(EPS eps, int i, PetscScalar *kr, PetscScalar *ki, Vec
xr, Vec xi);
EPSDestroy(EPS eps);
```

First, the eigenproblem solver (EPS) context is created and the operator(s) associated with the eigensystem are set, as well as the problem type. Then various options are set for customized solution. After that, the program solves the problem, retrieves the solution, and finally destroys the EPS context.

The above function calls are very important and will be present in most SLEPc programs. In the example source code `ex1.c` you will find other functions apart from these. What do they do?

Playing with EPS Options

Now we are going to experiment with different options of the EPS object. A full list of command-line options can be obtained by running the example with the option `-help`.

To show information about the solver object:

```
% ex1 -eps_view
```

Note: This option internally calls the function `EPSView`. Alternatively, we could include a direct call to this function in the source code. Almost all command-line options have a related function call.

Note: All the command-line options related to the EPS object have the `-eps_` prefix.

This time, your output will include something like this

```
EPS Object:
  problem type: symmetric eigenvalue problem
  method: krylovschur
  extraction type: Rayleigh-Ritz
  selected portion of the spectrum: largest eigenvalues in magnitude
  number of eigenvalues (nev): 1
  number of column vectors (ncv): 16
  maximum dimension of projected problem (mpd): 16
  maximum number of iterations: 100
  tolerance: 1e-07
  dimension of user-provided deflation space: 0
IP Object:
  orthogonalization method: classical Gram-Schmidt
  orthogonalization refinement: if needed (eta: 0.707100)
ST Object:
  type: shift
  shift: 0
```

This option is very useful to see which solver and options the program is using.

Try solving a much larger problem, for instance with $n=400$. Note that in that case the program does not return a solution. This means that the solver has reached the maximum number of allowed iterations and the convergence criterion was not satisfied. What we can do is either

increase the number of iterations or relax the convergence criterion.

```
% ex1 -n 400 -eps_max_it 400
% ex1 -n 400 -eps_tol 1e-3
```

Note that in the latter case the relative error displayed by the program is significantly larger, meaning that the solution has only 3 correct decimal digits, as expected.

It is possible to change the number of requested eigenvalues. Try the following execution

```
% ex1 -n 400 -eps_nev 2
```

In this case, the program did not succeed to compute the two requested eigenpairs. This is again due to the convergence criterion, which is satisfied by some eigenpairs but not for all. As in the previous case, we could increase further the number of iterations or relax the convergence criterion. Another alternative is to increase the number of column vectors (i.e. the dimension of the subspace with which the eigensolver works). This usually improves the convergence behavior at the expense of larger memory requirements.

```
% ex1 -n 400 -eps_nev 2 -eps_ncv 24
```

Note that the default value of `ncv` depends on the value of `nev`.

Try to set some of the above options directly in the source code by calling the related functions [EPSSetTolerances](#) and [EPSSetDimensions](#). Modify and recompile the program. Use `-eps_view` to check that the values are correctly set. Is it now possible to change these options from the command-line? Does this change whether you place the calls before or after the call to [EPSSetFromOptions](#)?

Convergence is usually bad when eigenvalues are close to each other, which is the case in this example. In order to see what is happening while the eigensolver iterates, we can use a monitor to display information associated to the convergence of eigenpairs at each iteration:

```
% ex1 -eps_monitor
```

Also, in some SLEPc installations, it is possible to monitor convergence graphically with `-eps_monitor_draw`. For example, try this:

```
% ex1 -n 700 -eps_nev 5 -eps_ncv 35 -eps_monitor_draw
```

Note: The plot is drawn in an X11 pop-up window. So this requires that the display is correctly exported.

Changing the Eigensolver

The convergence behavior for a particular problem also depends on the properties of the eigensolver being used. SLEPc provides several eigensolvers which can be selected in the source code with the function [EPSSetType](#), or at run time:

```
% ex1 -eps_nev 4 -eps_type lanczos
```

The following table shows some of the solvers available in SLEPc.

Solver	Command-line Name	Parameter
Krylov-Schur	krylovschur	EPSKRYLOVSCHUR
Lanczos with Explicit Restart	lanczos	EPSLANCZOS

Arnoldi with Explicit Restart	arnoldi	EPSARNOLDI
Subspace Iteration	subspace	EPSSUBSPACE
Power / RQI	power	EPSPOWER
Lapack	lapack	EPSLAPACK
ARPACK	arpack	EPSARPACK

Note: The Lapack solver is not really a full-featured eigensolver but simply an interface to some LAPACK routines. These routines operate in dense mode with only one processor and therefore are suitable only for moderate size problems. This solver should be used only for debugging purposes.

Note: The last one (ARPACK) may or may not be available in your system, depending on whether it was enabled during installation of SLEPc. It consists in an interface to the external [ARPACK library](#). Interfaces to other external libraries may be available as well. These can be used as any other SLEPc native eigensolver.

Note: The default solver is `krylovschur` for both symmetric and non-symmetric problems.

Exercise 2: Standard Non-Symmetric Eigenvalue Problem

In this exercise we are going to work with a non-symmetric problem. The example solves the eigenvalue problem associated with a Markov model of a random walk on a triangular grid. Although the matrix is non-symmetric, all eigenvalues are real. Eigenvalues come in pairs with the same magnitude and different signs. The values 1 and -1 are eigenvalues for any matrix size. More details about this problem can be found at [Matrix Market](#).

Compiling

Copy the file [ex5.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex5: ex5.o chkopts
    -${CLINKER} -o ex5 ex5.o ${SLEPC_LIB}
    ${RM} ex5.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex5
```

Source Code Details

The example program is very similar to that in Exercise 1. The main difference is that the problem is set to be non-symmetric with `EPSSetProblemType`:

```
ierr = EPSSetProblemType(eps, EPS_NHEP); CHKERRQ(ierr);
```

In this example we also illustrate the use of `EPSSetInitialVector`.

Running the Program

Run the program requesting four eigenvalues with different solvers. In particular, try with the Power Method

```
% ex5 -eps_nev 4 -eps_type power
```

The Power Iteration is a very simple and usually not competitive method, but it serves us here to introduce the concept of spectral transformation.

In this particular problem, the Power Method fails to converge irrespective of the convergence criterion, while the rest of the solvers are successful. Why? This convergence anomaly can be solved by simply shifting the matrix or using another spectral transformation.

Getting Started with Spectral Transformations

The general idea of the spectral transformation is to substitute the original problem, $Ax=\lambda x$, by another one, $Tx=\theta x$, in which the eigenvalues are mapped to a different position but eigenvectors remain unchanged.

Each EPS object uses internally an ST object, which manages the spectral transformation. Currently, there are four spectral transformations available, which can be selected with the function `STSetType` or at run time.

Spectral Transform	Operator	Command-line Name	Parameter
Shift of Origin	$A+\sigma I$	shift	STSHIFT
Spectrum Folding	$(A-\sigma I)^2$	fold	STFOLD
Shift-and-invert	$(A-\sigma I)^{-1}$	sinvert	STSINVERT
Cayley	$(A-\sigma I)^{-1}(A+\tau I)$	cayley	STCAYLEY

Note: The default is to do shift of origin with a value $\sigma=0$. This was reported by `-eps_view` in the previous example.

Try repeating the execution of the Power method with a positive value of the shift.

```
% ex5 -eps_nev 4 -eps_type power -st_type shift -st_shift 1
```

Note: All the command-line options related to the ST object have the `-st_` prefix.

Note: In the above command line, the option `-st_type shift` is not really necessary since it is the default.

With the above execution, all the spectrum is shifted to the right by a unity and then 1 becomes the only dominant eigenvalue. Try now with $\sigma=-1$:

```
% ex5 -eps_nev 4 -eps_type power -st_shift -1
```

Note that in this case, -1 becomes dominant, so in this case this allows us to compute eigenvalues from the left side of the spectrum. Try with other values of the shift as well as with other eigensolvers.

The value of σ (the shift) can be specified directly in the source code with the function `STSetShift`. Modify the source code to change the default value. Recompile and run the program with `-eps_view` to check that it was changed correctly.

Note: Since the ST object is created internally by the EPS object (and not by the user), in order to be able to change one of its properties it is necessary to get it first with `EPSGetST`.

We have seen that the shift of origin can be used in some cases to compute solutions from any end of the spectrum, but in general this is not easy. SLEPc allows to specify which part of the spectrum we are interested in, without having to use any spectral transformations. See `EPSSetWhichEigenpairs` for details.

```
% ex5 -eps_nev 4 -eps_smallest_real
% ex5 -eps_nev 4 -eps_largest_real
```

In order to compute eigenvalues located in the interior of the spectrum, the shift-and-invert or Cayley transforms must be used. These allow computing the eigenvalues that are closest to σ .

```
% ex5 -eps_nev 2 -st_type sinvert -st_shift 0.8
```

Note: In exercise 5 we will see an alternative way of computing interior eigenvalues without a spectral transformation.

Handling the Inverses

In the table of spectral transformations shown above, there are some operators that include the inverse of a certain matrix. These operators are not computed explicitly in order to preserve sparsity. Instead, in the ST object the multiplication by these inverses is replaced by a linear equation solve via a KSP object from PETSc.

SLEPc allows us to pass options to this KSP linear solver object. For instance,

```
% ex5 -st_type sinvert -st_shift 0.8 -st_ksp_type preonly -st_pc_type lu
```

Note: In order to specify a command-line option related to the linear solver contained in ST, simply add the `-st_` prefix in front.

The options of the above example specify a direct linear solver (LU factorization). This is what SLEPc does by default. This strategy is usually called *exact shift-and-invert*. Its main drawback is that direct solvers are more costly in terms of flops and storage and are less parallelizable.

An alternative is to do an *inexact shift-and-invert*, that is, to use an iterative linear solver. The following line illustrates how to use an iterative solver

```
% ex5 -st_type sinvert -st_shift 0.8 -st_ksp_type gmres
    -st_pc_type bjacobi -st_ksp_rtol 1e-12
```

Iterative linear solvers may fail to converge if the coefficient matrix is ill-conditioned or close to singular. Also, the accuracy of the eigensolver may be compromised if the iterative linear solver provides a solution far from full working precision.

Note that in SLEPc it is extremely easy to switch between exact and inexact schemes.

Exercise 3: Generalized Eigenvalue Problem Stored in a File

In this exercise we are going to work with a generalized eigenvalue problem, $Ax=\lambda Bx$. The example program loads two matrices A and B from a file and then solves the associated eigensystem.

The matrices we are going to work with are BFW62A and BFW62B, which are available at [Matrix Market](#). This particular problem is non-symmetric. Some of the eigenvalues (those of largest magnitude) come in complex conjugate pairs while the rest are real.

Compiling

Copy the file [ex7.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex7: ex7.o chkopts
    -${CLINKER} -o ex7 ex7.o ${SLEPC_LIB}
    ${RM} ex7.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex7
```

Source Code Details

This example uses the PETSc function `MatLoad` to load a matrix from a file. The two matrix files are specified in the command line. Note that these files have been converted from Matrix Market format to PETSc binary format.

Compare the source code of the example program with the previous ones. Note that, in this case, two matrix objects are passed in the `EPSSetOperators` function call:

```
ierr = EPSSetOperators(eps,A,B);CHKERRQ(ierr);
```

Running the Program

Run the program with the following command line

```
% ex7 -f1 ${SLEPC_DIR}/src/mat/examples/bfw62a.petsc  
-f2 ${SLEPC_DIR}/src/mat/examples/bfw62b.petsc
```

Note that two eigenvalues are returned although only one was requested, since complex conjugate pairs always converge simultaneously. Try to solve the problem with other eigensolvers. In particular, try the Power Method and observe that it fails to converge. Try to use a shift to obtain the eigenvalues located at the other end of the spectrum.

Run the program to compute more than one eigenpair. Use the following option to plot the computed eigenvalues:

```
% ex7 -f1 ${SLEPC_DIR}/src/mat/examples/bfw62a.petsc  
-f2 ${SLEPC_DIR}/src/mat/examples/bfw62b.petsc  
-eps_type subspace -eps_nev 6 -eps_plot_eigs -draw_pause -1
```

Note: The plot is drawn in an X11 pop-up window. So this requires that the display is correctly exported.

Spectral Transformations in Generalized Problems

The following table shows the expressions of the operator in each of the available spectral transformations in the case of generalized problems. Note that both matrices A and B are involved.

Spectral Transform	Operator
Shift of Origin	$B^{-1}A + \sigma I$
Spectrum Folding	$(B^{-1}A - \sigma I)^2$
Shift-and-invert	$(A - \sigma B)^{-1}B$
Cayley	$(A - \sigma B)^{-1}(A + \tau B)$

In the case of generalized problems, the shift-and-invert transformation does not represent a cost

penalty with respect to the simpler shift of origin, since in both cases the inverse of a matrix is required.

```
% ex7 -f1 ${SLEPC_DIR}/src/mat/examples/bfw62a.petsc
       -f2 ${SLEPC_DIR}/src/mat/examples/bfw62b.petsc
       -st_type sinvert
```

Note that the above execution uses the default value of the shift $\sigma=0$ and therefore it is computing the eigenvalues closest to the origin. Use a shift near the left end of the spectrum to compute the largest magnitude eigenvalues

```
% ex7 -f1 ${SLEPC_DIR}/src/mat/examples/bfw62a.petsc
       -f2 ${SLEPC_DIR}/src/mat/examples/bfw62b.petsc
       -st_type sinvert -st_shift -250000
```

Exercise 4: Singular Value Decomposition

In this exercise we turn our attention to the Singular Value Decomposition (SVD). Remember that in real symmetric (or complex Hermitian) matrices, singular values coincide with eigenvalues, but in general this is not the case. The SVD is defined for any matrix, even rectangular. Singular values are always non-negative real values.

This example works also by reading a matrix from a file. In particular, the matrix to be used is related to a 2D reaction-diffusion model. More details about this problem can be found at [Matrix Market](#).

Compiling

Copy the file [ex14.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex14: ex14.o chkopts
      -${CLINKER} -o ex14 ex14.o ${SLEPC_LIB}
      ${RM} ex14.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex14
```

Running the Program

In order to run the program, type the following

```
% ex14 -file ${SLEPC_DIR}/src/mat/examples/rdb200.petsc
```

You should get an output similar to this

```
Singular value problem stored in file.

Reading REAL matrix from a binary file...
Number of iterations of the method: 3
Solution method: cross

Number of requested singular values: 1
Stopping condition: tol=1e-07, maxit=100
```

Number of converged approximate singular triplets: 2

sigma	residual norm
35.007519	1.09161e-10
34.104187	2.5202e-09

Source Code Details

The way in which the SVD object works is very similar to that of EPS. However, some important differences exist. Examine the source code of the example program and pay attention to the differences with respect to EPS. After loading the matrix, the problem is solved by the following sequence of function calls:

```
SVDCreate(MPI_Comm comm,SVD *svd);
SVDSetOperator(SVD svd,Mat A);
SVDSetFromOptions(SVD svd);
SVDSolve(SVD svd);
SVDGetConverged(SVD svd, int *nconv);
SVDGetSingularTriplet(SVD svd,int i,PetscReal *sigma,Vec u,Vec
v);
SVDDestroy(SVD svd);
```

First, the singular value solver (SVD) context is created and the matrix associated with the problem is specified. Then various options are set for customized solution. After that, the program solves the problem, retrieves the solution, and finally destroys the SVD context.

Note that the singular value, `sigma`, is defined as a `PetscReal`, and that the singular vectors are simple `Vec`'s.

SVD Options

Most of the options available in the EPS object have their equivalent in SVD. A full list of command-line options can be obtained by running the example with the option `-help`.

To show information about the SVD solver, add the `-svd_view` option:

```
% ex14 -file $SLEPC_DIR/src/mat/examples/rdb200.petsc -svd_view
```

Note: All the command-line options related to the SVD object have the `-svd_` prefix.

Your output will include something like this

```
SVD Object:
method: cross
transpose mode: explicit
selected portion of the spectrum: largest
number of singular values (nsv): 1
number of column vectors (ncv): 16
maximum number of iterations: 100
tolerance: 1e-07
EPS Object:
problem type: symmetric eigenvalue problem
method: krylovschur
extraction type: Rayleigh-Ritz
selected portion of the spectrum: largest real parts
number of eigenvalues (nev): 1
number of column vectors (ncv): 16
maximum dimension of projected problem (mpd): 16
maximum number of iterations: 100
```

```

tolerance: 1e-07
dimension of user-provided deflation space: 0
IP Object:
  orthogonalization method:    classical Gram-Schmidt
  orthogonalization refinement: if needed (eta: 0.707100)
ST Object:
  type: shift
  shift: 0
Using a shell matrix
IP Object:
  orthogonalization method:    classical Gram-Schmidt
  orthogonalization refinement: if needed (eta: 0.707100)

```

The output shows all the options that are susceptible of being changed, either from the command line or from the source code of the program: the method, the portion of the spectrum (largest or smallest singular values), the number of singular values (`nsv`), etc.

Try to change some of the values, for instance:

```

% ex14 -file $SLEPC_DIR/src/mat/examples/rdb200.petsc -svd_nsv 10
        -svd_ncv 40 -svd_smallest

```

The "transpose mode" refers to whether the transpose of matrix A is being built explicitly or not (see [SVDSetTransposeMode](#) for an explanation).

Note that in the sample output above, the SVD object contains an EPS object. This only happens in some SVD solver types, as detailed below.

Changing the Singular Value Solver

SLEPc provides several solvers for computing the SVD, which can be selected in the source code with the function [SVDSetType](#), or at run time:

```

% ex14 -file $SLEPC_DIR/src/mat/examples/rdb200.petsc -svd_type trlanczos

```

The following table shows the list of SVD solvers available in SLEPc.

Solver	Command-line Name	Parameter
Cross Product	cross	SVDCROSS
Cyclic Matrix	cyclic	SVDCYCLIC
Lanczos with Explicit Restart	lanczos	SVDLANCZOS
Lanczos with Thick Restart	trlanczos	SVDTRLANCZOS
Lapack	lapack	SVDLAPACK

Note: The Lapack solver is not really a full-featured singular value solver but simply an interface to some LAPACK routines. These routines operate in dense mode with only one processor and therefore are suitable only for moderate size problems. This solver should be used only for debugging purposes.

Note: The default solver is `cross`.

The first two solvers, `cross` and `cyclic`, are not real methods implemented in the SVD module, but are two convenient ways of solving the SVD problem by making use of the eigensolvers available in the EPS module. In those two cases, the SVD object manages an EPS object internally, whose parameters can be set as desired (typically only the method). For example:

```
% ex14 -file $$SLEPC_DIR/src/mat/examples/rdb200.petsc -svd_type cyclic
        -svd_eps_type lanczos
```

Exercise 5: Problem without Explicit Matrix Storage

In many applications, it may be better to keep the matrix (or matrices) that define the eigenvalue problem implicit, that is, without storing its nonzero entries explicitly. An example is when we have a matrix-vector routine available. SLEPc allows easy management of this case. This exercise tries to illustrate it by solving a standard symmetric eigenproblem corresponding to the Laplacian operator in 2 dimensions in which the matrix is not built explicitly.

Compiling

Copy the file [ex3.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex3: ex3.o chkopts
    -${CLINKER} -o ex3 ex3.o ${SLEPC_LIB}
    ${RM} ex3.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex3
```

Source Code Details

PETSc provides support for matrix-free problems via the *shell* matrix type. This kind of matrices is created with a call to [MatCreateShell](#), and their operations are specified with [MatShellSetOperation](#). For basic use of these matrices with EPS solvers only the matrix-vector product operation is required. In the example, this operation is performed by a separate function `MatLaplacian2D_Mult`.

Running the Program

Run the program without any spectral transformation options. For instance:

```
% ex3 -eps_type subspace -eps_tol 1e-9 -eps_nev 8
```

Now try running the program with shift-and-invert to get the eigenvalues closest to the origin

```
% ex3 -st_type sinvert
```

Note that the above command yields a run-time error. Observe the information printed in the screen and try to deduce the reason of the error. In this case, the error is due to the fact that SLEPc tries to use a direct linear solver withing the ST object, and this is not possible unless the matrix has been created explicitly as in previous examples.

There are more chances to have success if an inexact shift-and-invert scheme is used. Try using an iterative linear solver without preconditioning:

```
% ex3 -st_type sinvert -st_ksp_rtol 1e-10 -st_ksp_type gmres
        -st_pc_type none
```

The above example works. However, try with a nonzero shift:

```
% ex3 -st_type sinvert -st_ksp_rtol 1e-10 -st_ksp_type gmres
      -st_pc_type none -st_shift 2
```

As you may see, at some point of the execution inside SLEPc, the execution fails because the ST object tries to make a copy of the problem matrix. However, the copy operation has not been defined in our shell matrix.

This matrix copy can be avoided by changing the default ST *matmode* (see [STSetMatMode](#) for details). For example, in order to force ST to work with a shell matrix itself (thus avoiding copying the matrix) one can execute

```
% ex3 -st_type sinvert -st_ksp_rtol 1e-10 -st_ksp_type gmres
      -st_pc_type none -st_shift 2 -st_matmode shell
```

The last example is much slower. This is because the iterative linear solver takes a lot of iterations to reach the required precision (add `-st_ksp_monitor` to monitor the convergence of the linear solver). In order to alleviate this problem, a preconditioner should be used. However, a powerful preconditioner such as ILU cannot be used in this case, for the same reason a direct solver is not available. The only possibility is to use a simple preconditioner such as Jacobi. Try running the last example again with `-st_pc_type jacobi`. You will get an error. Which is the source of the problem? How could this be avoided (by modifying the source code)?

Avoiding the Linear Solvers with Harmonic Extraction

There is a completely different alternative to spectral transformations, that consists in changing the way in which the method extracts the spectral information from the built subspace; see [EPSSetExtraction](#) for details. One such technique is called harmonic extraction. Try it with the following options:

```
% ex3 -eps_harmonic -eps_target 2
```

Here, the target is a scalar value that indicates the region of the spectrum where eigenvalues must be sought (it is equivalent to the shift in ST). Although harmonic extraction is generally less effective than spectral transformations, it completely avoids the use of any linear solver, thus being ideal for problems with an implicit matrix.

Exercise 6: Parallel Execution

The objective of this exercise is to run an example program with different number of processors to see how execution time is reduced. This time, we are going to solve a standard eigensystem $Ax=\lambda x$ with the matrix loaded from a file. In particular, the matrix we are going to use is QC2534. It is a complex matrix of order 2534 arising from a quantum chemistry application (more details can be found at [Matrix Market](#)).

Compiling

Copy the file [ex4.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex4: ex4.o chkopts
     -${CLINKER} -o ex4 ex4.o ${SLEPC_LIB}
     ${RM} ex4.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command (optimized complex version)

```
% make PETSC_ARCH=rs6000_sp_0_complex ex4
```

Source Code Details

This example program is very similar to that of exercise 3. It uses the PETSc function `MatLoad` to load a matrix from a file. The matrix file is specified in the command line.

Running the Program

In order to run this example, you will need the file `qc2534.petsc`. Locate it in the file system and then run the program with the command

```
% ex4 -file qc2534.petsc
```

For execution with more than one processor:

```
% poe ex4 -procs 2 -file qc2534.petsc
```

Check the output of the program. It should be the same as with one processor.

Try using the `-log_summary` option to have a look at the profiling information collected by PETSc and SLEPc. For instance, check the size and number of MPI messages.

```
% poe ex4 -procs 2 -file qc2534.petsc -log_summary
```

Try to find out how much time was spent is solving the eigenvalue problem. Is there a significant reduction when we increase the number of processors?

Instrumenting the Source Code

If we are just interested in knowing the time used by the eigensolver, then it may be better to let our example program inform us. With the function `PetscGetTime`, it is possible to obtain the current time of day (wall-clock time) in seconds. Edit the source code and add two calls to this function just before and after the `EPSSolve` call, as in the following fragment of code

```
ierr = PetscGetTime(&t1);CHKERRQ(ierr);  
ierr = EPSSolve(eps);CHKERRQ(ierr);  
ierr = PetscGetTime(&t2);CHKERRQ(ierr);  
ierr = PetscPrintf(PETSC_COMM_WORLD," Elapsed Time: %f\n",t2-t1);
```

Also you must add the definition of the two new variables

```
PetscLogDouble t1,t2;
```

Recompile the program with

```
% make PETSC_ARCH=rs6000_sp_0_complex ex4
```

Run it with one, two and four processors checking the time spent by the solver

```
% poe ex4 -procs 1 -file qc2534.petsc  
% poe ex4 -procs 2 -file qc2534.petsc  
% poe ex4 -procs 4 -file qc2534.petsc
```

Exercise 7: Use of Deflation Subspaces

The term deflation refers to the use of the knowledge of one or more eigenpairs to find other eigenpairs. For instance, most eigensolvers try to approximate a number of eigenpairs and, as soon as one of them has converged, they deflate it for better approximating the other ones. Another case is when one eigenpair is known a priori and one wants to use this knowledge to compute other eigenpairs. SLEPc supports this by means of *deflation subspaces*.

This example illustrates the use of deflation subspaces to compute the smallest nonzero eigenvalue of the Laplacian of a graph corresponding to a 2-D regular mesh. The problem is a standard symmetric eigenproblem $Ax=\lambda x$, where $A = L(G)$ is the Laplacian of graph G , defined as follows: $A_{ij} = \text{degree of node } i$, $A_{ij} = -1$ if edge (i,j) exists in G , zero otherwise. This matrix is symmetric positive semidefinite and singular, and $[1 \ 1 \ \dots \ 1]^T$ is the eigenvector associated with the zero eigenvalue. In graph theory, one is usually interested in computing the eigenvector associated with the next eigenvalue (the so-called Fiedler vector).

Compiling

Copy the file [ex11.c \[plain text\]](#) to your directory and add these lines to the makefile

```
ex11: ex11.o chkopts
      -${CLINKER} -o ex11 ex11.o ${SLEPC_LIB}
      ${RM} ex11.o
```

Note: In the above text, the blank space in the 2nd and 3rd lines represents a tab.

Build the executable with the command

```
% make ex11
```

Source Code Details

This example computes the smallest eigenvalue by setting `EPS_SMALLEST_REAL` in [EPSSetWhichEigenpairs](#). An alternative would be to use a shift-and-invert spectral transformation with a zero shift to compute the eigenvalues closest to the origin, or to use harmonic extraction with a zero target.

By specifying a deflation subspace (the one associated to the eigenvector $[1 \ 1 \ \dots \ 1]^T$) with the function [EPSAttachDeflationSpace](#), the convergence to the zero eigenvalue is avoided. Thus, the program should compute the smallest nonzero eigenvalues.

Running the Program

Run the program simply with

```
% ex11
```

For the case of using a spectral transformation, the command line would be:

```
% ex11 -eps_largest_real -st_type sinvert -st_shift 0
      -st_ksp_rtol 1e-10 -st_ksp_type gmres -st_pc_type jacobi
```

Note that a shift-and-invert spectral transformation should always be used in combination with `EPS_LARGEST_MAGNITUDE` or `EPS_LARGEST_REAL`.

And for the case of harmonic extraction:

```
% ex11 -eps_largest_real -eps_harmonic -eps_target 0
```
