ARPACK++: A C++ Implementation of ARPACK Eigenvalue Package (Draft Version)

Francisco M. Gomes and Danny C. Sorensen

CRPC-TR97729
August 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997

ARPACK++

A c++ implementation of ARPACK eigenvalue package.

FRANCISCO M. GOMES
Departamento de Matemática Aplicada e Computacional
Universidade Estadual de Campinas - Brazil

DANNY C. SORENSEN
Department of Computational and Applied Mathematics
Rice University

August 7, 1997

Contents

Introduction

ARPACK++ is an object-oriented version of the ARPACK package. ARPACK [6] is a well-known collection of FORTRAN subroutines designed to compute a few eigenvalues and, upon request, eigenvectors of large scale sparse matrices and pencils. It is capable of solving a great variety of problems from single precision positive definite symmetric problems to double precision complex non-Hermitian generalized eigenvalue problems.

ARPACK implements a variant of the Arnoldi process for finding eigenvalues called *implicit restarted Arnoldi method (IRAM)* [10, 11]. IRAM combines Arnoldi factorizations with an implicitly shifted QR mechanism to create a new method that is appropriate for very large problems. In most cases only a compressed matrix or a matrix-vector product $y \leftarrow Ax$ must be supplied by the user.

ARPACK++ is a collection of classes that offers c++ programmers an interface to ARPACK. It preserves the full capability, performance, accuracy and memory requirements of the FORTRAN package, but takes advantage of the c++ object-oriented programming environment.

The main features of ARPACK preserved by the c++ version include:

- The ability to return a few eigenvalues that satisfy a user specified criterion, such as largest real part, largest absolute value, etc.

- A fixed pre-determined storage requirement. Usually, only $n \cdot O(k) + O(k^2)$ memory locations are used to find *k* eigenvalues of a *n*-dimensional problem.

- A user-specified numerical accuracy for the computed eigenvalues and eigenvectors. Residual tolerances may be set to the level of working precision.

- The ability to find multiple eigenvalues without any theoretical or computational difficulty other than some additional matrix-vector products required to expose the multiple instances. This is made possible through the implementation of deflation techniques similar to those employed to make the implicit shifted QR algorithm robust and practical. Since a block method is not required, the user does not need to "guess" the correct block size that would be needed to capture multiple eigenvalues.

- Several alternatives to solve the symmetric generalized problem $Ax = Mx\lambda$ for singular or ill-conditioned symmetric positive semi-definite *M*.

Other features that are exclusive to ARPACK++ are:

- **The use of templates.** Class templates, or containers, are the most noticeable way of defining generic data types. They combine run-time efficiency and massive code and design reutilization. ARPACK++ uses templates to reduce the work needed to establish and solve eigenvalue problems and to simplify the structure utilized to handle such problems. One class will handle single and double precision problems. Depending on the data structure used, a single class can also be used to define real and complex matrices.

- **A friendly interface.** ARPACK++ avoids the complication of the *reverse communication interface* that characterizes the FORTRAN version of ARPACK. It contains many class templates that are easy to use. Some of them require the user to supply only the nonzero elements of a matrix, while others demand the definition of a class that includes a matrix-vector function. Nevertheless, the *reverse communication interface* is also preserved in the c++ package, allowing the user to solve an eigenvalue problem iteratively, performing explicitly all the matrix-vector products required by the Arnoldi method.

- **A great number of auxiliary functions.** ARPACK++ gives the user various alternatives for handling an eigenvalue problem. There are many functions to set and modify problem parameters, and also several output functions. For instance, seven different functions can be used to determine the eigenvectors of a problem. There are also ten functions that return from a single element of an eigenvector to an STL vector that contains all the eigenvectors.

- **The ability to easily find interior eigenvalues and to solve generalized problems.** ARPACK++ includes several matrix classes that use state-of-the-art routines from SuperLU, UMFPACK and LAPACK to solve linear systems. When one of these classes is used, spectral transformations such as the shift and invert method can be employed to find internal eigenvalues of regular and generalized problems without requiring the user to explicitly solve linear systems.

- **A structure that simplify the linkage with other libraries.** The main aim of ARPACK++ is not only to allow the user to efficiently handle even the most intricate problems, but also to minimize the work needed to generate an interface between ARPACK and other libraries, such as the *Template Numerical Toolkit (TNT)* [9] and the *Hilbert Class Library (HCL)* [5].

In the first chapter, some instructions are given on how to install ARPACK++. Chapter 2 discusses briefly what is necessary to start solving eigenvalue problems with the library. Differences and similarities between ARPACK++ classes and its computational modes are described in chapter 3. Chapter 4 contains more detailed instructions on how to create an eigenvalue problem, while some examples that illustrate ARPACK++ usage were included in chapter 5. Finally, all classes, functions, constructor and template parameters are fully described in the appendix: *ARPACK++ reference guide*.

The authors would like to acknowledge Dr. Roldan Pozo and Dr. Kristyn Maschhoff for their insightful suggestions and support.

Financial support for this work was provided in part by FAPESP (Brazil), grant 96/2556-9, by the National Science Foundation cooperative agreement CCR-912008 and by the ARPA contract number DAAL03-91-C-0047 (administered by the U.S. Army Research Office).

Chapter

1

Installing and running ARPACK++

As a collection of class templates, ARPACK++ can be easily installed, provided that other libraries required by the software are available. This chapter describes how to obtain ARPACK++ and what is necessary to use it.

Obtaining the software.

ARPACK++ is distributed as a single file called `arpackpp.tar.gz`. This file contains all of the library files, some illustrative examples and also a copy of this manual. It can be obtained from the URL: `ftp://ftp.caam.rice.edu/pub/people/chico`.

Because ARPACK++ is an interface to the original ARPACK FORTRAN library, this library must be available when installing the c++ code. Although FORTRAN files are not distributed along with ARPACK++, they can be obtained from Netlib or directly from the URL: `ftp://ftp.caam.rice.edu/pub/software/ARPACK`.

ARPACK FORTRAN software comes with all necessary BLAS and LAPACK routines, but it is also possible to use local (optimized) installations of these libraries if they are available. ARPACK user's guide [6] provides further instructions on how to obtain and compile the FORTRAN code[1].

[1] The ARPACK user's guide also describes in detail how ARPACK works and contains a lot of useful examples. It is an excellent source of information for those interested in solving eigenvalue problems.

Other libraries required by some (but not all) ARPACK++ classes include SuperLU [3], UMFPACK [1] and the *Standard Template Library* (see [8]) packages.

The SuperLU package can be used to solve eigenvalue problems that require complex or real nonsymmetric matrix factorizations. It is called by `ARluNonSymStdEig`, `ARluNonSymGenEig`, `ARluCompStdEig` and `ARluCompGenEig` classes and must be installed if one of these classes is to be used. SuperLU is available at `http://www.netlib.org/scalapack/prototype`.

The above mentioned classes can also call UMFPACK library functions instead of using SuperLU to solve eigenvalue problems that require matrix decompositions. However, UMFPACK may be used solely for educational, research, and benchmarking purposes by non-profit organizations and the U.S. government. Commercial and other organizations may make use of UMFPACK only for benchmarking purposes. UMFPACK can be downloaded from `ftp://ftp.cis.ufl.edu/pub/umfpack`. The MA38 Package in the *Harwell Subroutine Library (HSL)* has equivalent functionality (and identical calling interface) as UMFPACK and is available for commercial use. Technical reports and information on HSL can be obtained from `http://www.cis.rl.ac.uk/struct/ARCD/NUM.html`. Unfortunately, MA38, as well as older versions of UMFPACK, cannot deal with complex matrices.

The vector class from the *Standard Template Library (STL)* can be used to retrieve eigenvalues and eigenvectors computed by all ARPACK++ classes. There are also plans to build an interface between ARPACK++ and *TNT* [9], an STL-based library, as soon as this package become available. Some compilers include their own version of STL. If it is not the case, the library can also be found at `ftp://butler.hpl.hp.com/stl`.

Installing ARPACK++.

To unbundle file `arpackpp.tar.gz` the user should use the following command:

```
gzip -d arpackpp.tar.gz | tar -xvf -
```

A main directory called `arpack++` will be automatically created. This directory should contain three other directories. One of them, `arpack++/include`, concentrates all ARPACK++ templates. Another, `arpack++/examples`, includes some selected examples, and the last, `arpack++/doc`, contains installation notes, a description of ARPACK++ structure and a list of known bugs.

ARPACK++ is a collection of templates. Templates are defined in header (`.h`) files, so they can be used directly by other programs without any previous compilation. No object (`.o`) or library (`.a`) files have to be generated when installing ARPACK++, except those corresponding to other libraries (ARPACK, LAPACK, SuperLU and

UMFPACK). Some hints on how to properly install these libraries can be found in the `doc/installation.txt` file.

ARPACK++ header files can be moved to any "include" directory, provided that an option in the form

```
-I<directory name>
```

is added to the command line when compiling programs that use the software, where *`<directory name>`* is the name of the include directory.

Compatibility.

At the present time, ARPACK++ has only been compiled with the CC and GNU g++ compilers and tested in a SUN SparcStation. Further work must be done in order to port the package to other environments.

To minimize this inconvenience, compiler-dependent functions and data types used by ARPACK++ were grouped in a file called `include/arch.h`. Thus, this file should be changed to reflect the characteristics of the user’s system.

Besides that, a list of known incompatibilities between ARPACK++ and some compilers can be found in `doc/bugs.txt`.

Declaring complex numbers with the `arcomplex` class.

One of the major drawbacks of building mathematical software in c++ is the lack of a standard complex class. Different c++ compilers tend to have different complex classes and most people agree that writing a portable code is almost impossible in such case.

Because of that, ARPACK++ includes its own complex class, called `arcomplex`, `arcomplex` is a class template created in an effort to emulate the g++ complex class when other compilers are being used. Both single and double precision complex numbers can be represented by `arcomplex`, as shown in the following example:

```
#include “arcomp.h”

arcomplex<float>  z;    // A single precision complex number.
arcomplex<double> w;    // A double precision complex.
```

`arcomplex` is the only complex type referenced by all the ARPACK++ files, so it is not necessary to change several files when using a compiler other than g++, but only `arcomp.h`, the file where `arcomplex` is defined.

Storage requirements.

The amount of memory required to run ARPACK++ depends on a great number of variables, including the type of the problem, its dimension (*n*) , the number desired eigenvalues (*nev*) and the number of Arnoldi vectors generated at each iteration (*ncv*).

The table below furnishes the amount of memory positions used to find eigenvalues and eigenvectors[2] of a standard problem as a function of *n*, *nev* and *ncv*. Since the user is not required to supply *ncv* (this is a optional parameter), the third column of the table indicates the memory required when *ncv* is set to its default value (*2nev+1*).

The table indicates the number of real positions required to solve the related problems. The number of bytes actually used in each case can be obtained by multiplying the value shown in the table by the size (in bytes) of the real element used[3]. These values correspond to a problem solved in regular mode. A (small) amount of memory that does not depend on *n*, *ncv* or *nev* is also required to store some other ARPACK++ variables and function parameters, but this memory is negligible for large problems.

type of problem	**memory positions required**	**memory usage with default ncv**
real symmetric	$4n + n.ncv + ncv^2 + 8ncv + nev$	$5n + 2n.nev + 4nev^2 + 21nev$
real nonsymmetric	$4n + n.ncv + 3ncv^2 + 9ncv + 2nev$	$5n + 2n.nev + 12nev^2 + 32nev$
complex	$8n + 2n.ncv + 6ncv^2 + 15ncv + 2nev$	$10n + 4n.nev + 24nev^2 + 56nev$

If the user wants to determine *nev* eigenvectors and *nev* Schur vectors at the same time, or if he wants to supply his own vector to store the eigenvectors, the storage requirements are increased by $nev \cdot n$ positions in the symmetric case, $nev \cdot n + n$ positions in the nonsymmetric case and $2nev \cdot n$ positions in the complex case.

The values mentioned above do not include the memory required to store the matrices of the problem, nor the LU factors that are generated when a spectral transformation is used. Since the exact number of elements of L and U are hard to determine, the user should also take in account at least an estimate of these additional memory positions required to store the problem data.

[2] The same amount of memory is required to find nev Schur vectors or an Arnoldi basis instead of the eigenvectors.
[3] Typical values are: 4 bytes for single precision variables and 8 bytes if double precision is used.

Comparing the performance of ARPACK and ARPACK++.

Comparing the performance of ARPACK and ARPACK++ is not so easy as it might appear, since the libraries are not exactly equivalent.

The first aspect that must be noted is that the FORTRAN version of ARPACK is not a package that finds eigenvalues at once, but rather a collection of functions that must be called iteratively when solving an eigenvalue problem. This structure is called the *reverse communication interface*.

Since ARPACK++ also includes this interface, the simplest comparison between both versions consists in determining the overhead produced by the c++ structure. This overhead comprises the time spent to declare an eigenvalue problem using one of the classes provided by ARPACK++, the time required to set the initialize all of the ARPACK++ internal variables and the overhead generated each time a FORTRAN subroutine is called.

Compared this way, both versions have shown the same performance. The difference between ARPACK and PARPACK++ is insignificant.

Another way to compare the c++ and FORTRAN codes consists in measuring the total time spent by each library to solve an eigenvalue problem. The disadvantage of this type of analysis is that the time consumed by the matrix-vector products (and occasionally some linear systems) required by the Arnoldi method is also considered, which means that not only the performance of ARPACK and ARPACK++ is compared, but also the ability of the FORTRAN and c++ compilers to optimize the matrix-vector product routine (and sometimes also the linear solver).

In a preliminary test, a very simple set of sample problems that are distributed with ARPACK was used to compare the performance of both packages. The computations were made on a Sun Workstation, with the f77 (version 3.0.1) and the g++ (version 2.7.2) compilers[4]. The compiler option `-O` was used in all tests. The dimension of the problem was set to values varying between 100 and 2025. All the tests were performed in double precision.

The results obtained suggest that for problems with real variables the performance of ARPACK and ARPACK++ is very similar. A closer look at the matrix-vector products reveals that they have taken a little more time in c++ than in FORTRAN, but this difference was usually attenuated when considering the total time spent by the Arnoldi method.

On the other hand, problems with complex variables have run much faster in FORTRAN than when compiled with g++. Generally, each matrix-vector product in c++ have taken about 750% more time than the same product in FORTRAN.

[4] The CC compiler was also tested but it has shown a worse performance when compared to g++.

This difference is so great that it suggests that for complex problems the time consumed by the matrix-vector products can dictate the overall performance of the program. This is particularly true for large scale eigenvalue problems.

Perhaps this poor behavior can be imputed to the fact that the c++ language does not contain a intrinsic complex data type as FORTRAN does. Although g++ includes a very attractive class template to define complex variables, the performance of this class is not so good.

The matrices of the complex problems tested were very sparse, so the total time spent by the c++ code was 32% greater than the time consumed by the FORTRAN version of ARPACK. However, worse results should be expected in general.

ARPACK++ also contains some classes that are capable of performing all the matrix-vector products (and occasionally solving the linear systems) required to solve an eigenvalue problem, but these classes were not used here, since they are not present in ARPACK. The comparison was made using the same matrix-vector routine translated to c++ and FORTRAN.

Naturally, some care must be taken before extending these results to other problems. One cannot analyze the behavior of both libraries based only on the results mentioned here, since the total time spent by the Arnoldi method is greatly affected by many factors such as the dimension of the system, the sparsity pattern of the matrices, the number of eigenvalues that are to be calculated, the desired portion of the spectrum and the number of Arnoldi vectors generated at each iteration. Without mentioning that the computer and the compiler used can also affect the measured results.

Chapter

2

Getting started with ARPACK++

The purpose of the chapter is to give a brief introduction to ARPACK++ while depicting the kind of information the user is required to provide to easily solve eigenvalue problems.

The example included here is very simple and is not intended to cover all ARPACK++ features. In this example, a real nonsymmetric matrix in *Compressed Sparse Column (CSC)* format is generated and its eigenvalues and eigenvectors determined by using the `AREig` function.

Other different ways of creating elaborate ARPACK++ objects and solving more difficult problems, such those that require the use of spectral transformations, will be presented in chapter four. A full description of all ARPACK++ classes and functions is the subject of the appendix.

A real nonsymmetric example.

Perhaps the easiest way of getting started with ARPACK++ is trying to run the `simple.cc` example contained in the `examples/areig/nonsym` directory.

A slightly modified version of `simple.cc` is reproduced below. It illustrates

1. How to declare a matrix in CSC format;
2. How to pass matrix data to the `AREig` function;

3. How to use `AREig` to obtain some eigenvalues and eigenvectors; and
4. How to store output data.

```
/*
  MODULE Simple.cc
  Simple example program that illustrates how to solve a real
  nonsymmetric standard eigenvalue problem in regular mode
  using the AREig function.
*/

#include "areig.h"
#include <math.h>
#include "lnmatrxc.h"

main()
{

  // Declaring variables needed to store
  // A in compressed sparse column (CSC) format.

  int     n;      // Dimension of matrix.
  int     nnz;    // Number of nonzero elements in A.
  int*    irow;   // Row index of all nonzero elements of A.
  int*    pcol;   // Pointer to the beginning of each column.
  double* A;      // Nonzero elements of A.

  // Generating a double precision nonsymmetric 100x100 matrix.

  n = 100;
  Matrix(n, nnz, A, irow, pcol);

  // Declaring AREig output variables.

  int     nconv;                 // Number of converged eigenvalues.
  double* EigValR = new double[100];    // Eigenvalues (real part).
  double* EigValI = new double[100];    // Eigenvalues (imag part).
  double* EigVec  = new double[1100];   // Eigenvectors.

  // Finding the five eigenvalues with largest magnitude
  // and the related eigenvectors.

  nconv = AREig(EigValR, EigValI, EigVec, n, nnz, A, irow, pcol, 5);

  // Printing eigenvalues.

  cout << "Eigenvalues:" << endl;
  for (int i=0; i<nconv; i++) {
    cout << "  lambda[" << (i+1) << "]: " << EigValR[i];
    if (EigValI[i]>=0.0) {
      cout << " + " << EigValI[i] << " I" << endl;
    }
    else {
      cout << " - " << fabs(EigValI[i]) << " I" << endl;
    }
  }

} // main.
```

In the example above, `Matrix` is a function that returns the variables `nnz`, `A`, `irow`, and `pcol`. These variables are used to pass matrix data to the `AREig` function, as described in the next section. The number of desired eigenvalues (five) must also be declared when calling `AREig`.

`AREig` is not a true ARPACK++ function. Although being a MATLAB-style function that can be used solve most eigenvalue problems in a very straightforward way, `AREig` does not explore most ARPACK++ capabilities. It was included here only as an example, merely to introduce the software. The user is urged to check out chapters 3 and 4 to see how to really take advantage of all ARPACK++ features.

`AREig` was defined in a file called `examples/areig/areig.h` and contains the some basic ARPACK++ commands needed to find eigenvalues using the SuperLU package. Therefore, to use this function, SuperLU must be previously installed (following the directions given in chapter one).

`EigVec`, `EigValR`, `EigValI` and `nconv` are output parameters of `AREig`. `EigVec` is a vector that stores all eigenvectors sequentially (see chapter 5 or the appendix). `EigValR` and `EigValI` are used to store, respectively, the real and imaginary parts of matrix eigenvalues. `nconv` indicates how many eigenvalues with the requested precision were actually obtained.

Many other ARPACK++ parameters can be passed as arguments to `AREig`. Because these other parameters were declared as *default arguments*, they should be declared only if the user does not want to use the default values provided by ARPACK++ .

Defining a matrix in Compress Sparse Column format.

The `Matrix` function below shows briefly how to generate a sparse real nonsymmetric matrix in CSC format. See the definition of the `ARluNonSymMatrix` and `ARumNonSymMatrix` classes in the appendix for further information on how to create a matrix using this format.

`n` is an input parameter that defines matrix dimension. All other parameters are returned by the function. `A` is a pointer to an array that contains the nonzero elements of the matrix. `irow` is a vector that gives the row index of each element stored in `A`. Elements of the same column are stored in an increasing order of rows. `pcol` gives integer pointers to the beginning of all matrix columns.

In this example, the matrix is tridiagonal[5], with `dd` on the main diagonal, `dl` on the subdiagonal and `du` on the superdiagonal.

[5] ARPACK++ also includes a band matrix class that could have been used here. It was not used since the purpose of the example is to show how to define a matrix using the CSC format.

```
template<class FLOAT, class INT>
void Matrix(INT n, INT& nnz, FLOAT* &A, INT* &irow, INT* &pcol)
{

  INT   i, j;

  // Defining constants.

  FLOAT dd = -1.5;
  FLOAT dl =  4.0;
  FLOAT du = -0.5;

  // Defining the number of nonzero matrix elements.

  nnz = 3*n-2;

  // Creating output vectors.

  A    = new FLOAT[nnz];
  irow = new INT[nnz];
  pcol = new INT[n+1];

  // Filling A, irow and pcol.

  pcol[0] = 0;
  j = 0;
  for (i=0; i!=n; i++) {

    // Superdiagonal.

    if (i != 0) {
      irow[j] = i-1;
      A[j++]  = du;
    }

    // Main diagonal.

    irow[j] = i;
    A[j++]  = dd;

    // Subdiagonal.

    if (i != (n-1)) {
      irow[j] = i+1;
      A[j++]  = dl;
    }

    // Defining where the next column will begin.

    pcol[i+1] = j;
  }

} // Matrix.
```

Building the example.

To compile `simple.cc` and link it to some libraries, ARPACK++ provides a `Makefile` file. The user just need to type the command

```
make symsimp
```

However, some other files, such as `Makefile.inc` and `include/arch.h`, should be modified prior to compiling the example, in order to correctly define search directories and some machine-dependent parameters (see chapter one).

The AREig function.

`AREig` is a function intended to illustrate how to solve all kinds of standard and generalized eigenvalue problems, inasmuch as the user can store matrices in CSC format. It is defined in the `examples/areig/areig.h` file.

Actually, `areig.h` contains one definition of the `AREig` function for each different problem supported by ARPACK. This is called function overloading and is a very used feature of the c++ language. One of the implementations of `AREig` (the one that is used by the example defined in this chapter) is shown below.

```
template <class FLOAT>
int AREig(FLOAT EigValR[], FLOAT EigValI[], FLOAT EigVec[], int n,
          int nnz, FLOAT A[], int irow[], int pcol[], int nev,
          char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
          int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
{

  // Creating a matrix in ARPACK++ format.

  ARluNonSymMatrix<FLOAT> matrix(n, nnz, A, irow, pcol);

  // Defining the eigenvalue problem.

  ARluNonSymStdEig<FLOAT> prob(nev, &matrix, which, ncv,
                               tol, maxit, resid, AutoShift);

  // Finding eigenvalues and eigenvectors.

  return prob.EigenValVectors(EigVec, EigValR, EigValI);

}
```

As stated earlier, this function is intended to be simple and easy to use. As a consequence of its simplicity, many features provided by ARPACK++ classes and their member functions are not covered by `AREig`. It cannot be used, for example, to obtain Schur vectors, but only eigenvalues and eigenvectors. The user also cannot choose among all the output formats supplied by ARPACK++. Only the simplest format can be used.

To overcome in part the limitations imposed by this very stringent structure, `AREig` uses default parameters. These parameters can be ignored if the default values provided by ARPACK++ are appropriate, but the user can also change their values at his convenience. `maxit`, the maximum number of Arnoldi iterations allowed, and `which`, the part of the spectrum that is to be computed, are just two of those parameters. For a complete description of all ARPACK++ parameters, the user should refer to the appendix.

The version of `AREig` depicted above contains only three commands. The first one declares a matrix using the `ARluNonSymMatrix` function. The second one defines the eigenvalue problem and sets the ARPACK++ parameters. The third command determines eigenvalues and eigenvectors.

All of these commands may be used directly (and in conjunction to other ARPACK++ functions and classes) so the user need not to call `AREig` to solve an eigenvalue problem. However, because this function is quite simple to use, it may be viewed as an interesting alternative to find eigenvalues of a matrix in CSC format.

Chapter

3

ARPACK++ classes and functions

Due to algorithmic considerations concerning program efficiency and simplicity, ARPACK divides eigenvalue problems into three main categories: real symmetric, real nonsymmetric and complex. ARPACK provides a separate FORTRAN subroutine for each one of these classes.

In the c++ library, these categories are subdivided further. ARPACK++ makes a distinction between regular and generalized problems, due to the different number of matrices that characterize them, so there are six types of classes that can be used to define eigenvalue problems.

By dividing problems this way, ARPACK++ assures that each type of problem belongs to a class with minimal template arguments, reducing the compilation time and the size of the programs. As a consequence, ARPACK++ has a large number of classes. On the other hand, the number of constructor parameters is small.

Generally, the user will be asked to define a dense matrix or a matrix in compressed sparse column (CSC) or band format, or to supply a matrix-vector product $y \leftarrow OPx$ in order to describe the eigenvalue problem. The desired part of the spectrum must also be specified.

ARPACK++ classes and their computational modes are briefly described below. Some of the problem characteristics that can be defined by parameters passed to constructors are also presented, along with a complete list of ARPACK++ functions.

Eigenvalue problem classes.

ARPACK is an extensive package that supplies a variety of alternatives to handle different regular and generalized eigenvalue problems. Whenever possible, advantage is taken of special structure or characteristics such as the symmetry of involved matrices, the type of its elements (if they are real or complex) and the spectral transformation strategy used.

In this section, ARPACK++ classes are divided into three levels according to the amount of information the user is required to supply. For example, if a sparse matrix is available, in the sense that its nonzero elements can be put in a vector, then one group of classes should be used. Another group of classes were made available for the case the user can supply only matrix-vector products. Finally, a third group should be used if the user wants to evaluate matrix-vector products by himself instead of passing them to ARPACK++ classes constructors.

ARPACK++ classes are summarized below. Some examples that clarify their use will be presented in the next chapter, but the user must refer to the *ARPACK++ reference guide* below for a complete description of them.

Classes that require matrices.

The first and most versatile group of ARPACK++ classes is shown in the table below. These classes can be used to solve all kinds of eigenvalue problems as long as the nonzero elements of matrices can be stored in compressed sparse column or band format, or sequentially in a $n \times n$ vector if the matrix is dense.

Type of matrix	**type of problem**	**class name**
real symmetric	standard	ARluSymStdEig
	generalized	ARluSymGenEig
real nonsymmetric	standard	ARluNonSymStdEig
	generalized	ARluNonSymGenEig
complex not Hermitian	standard	ARluCompStdEig
	generalized	ARluCompGenEig

Because classes of this group use ARPACK++ internal structure to perform matrix-vector products and solve linear systems (by direct methods), the kind of information that the user is supposed to supply is minimal: just an object that belongs to one of the matrix classes provided by ARPACK++, besides the number of desired eigenvalues. A list of available matrix classes is given later in this chapter.

Classes that require user-defined matrix-vector products.

This group includes classes that permit the user to define matrix-vector products when nonzero matrix elements cannot be passed directly to ARPACK++. For each combination of matrix type, problem type and computational mode, there is a different set of such products that must be supplied, as it will be described in the next section.

To allow these matrix-vector products to have the same number of parameters without preventing them from sharing information with other data structures, they must be declared as a member function of some specific matrix classes.

Type of matrix	**type of problem**	**class name**
real nonsymmetric	standard	ARNonSymStdEig
	generalized	ARNonSymGenEig
real symmetric	standard	ARSymStdEig
	generalized	ARSymGenEig
complex (Hermitian or not)	standard	ARCompStdEig
	generalized	ARCompGenEig

These classes are also useful if the user wants to build an interface between ARPACK++ and some other library that contains matrix classes. An example on how to create such interface will be presented in chapter 5.

Reverse communication classes.

These classes implement the so called *reverse communication interface* (the interface provided by the ARPACK FORTRAN code), and should be used only if the user wants to solve eigenvalue problems without passing any matrix information to ARPACK++. In this case, the Arnoldi process is interrupted each time a matrix-vector product is required, so the user's code can perform the product.

type of matrix	type of problem	class name
real nonsymmetric	standard	ARrcNonSymStdEig
	generalized	ARrcNonSymGenEig
real symmetric	standard	ARrcSymStdEig
	generalized	ARrcSymGenEig
complex (Hermitian or not)	standard	ARrcCompStdEig
	generalized	ARrcCompGenEig

Base classes.

All the above classes are derived from the lowest level base classes `ARStdEig`, `ARGenEig`, `ARrcStdEig` and `ARrcGenEig`. These classes contain the most fundamental ARPACK++ variables and functions and are not intended to be used directly. But they can be useful if someone wants to build his own classes related to some specific problems.

Constructor parameters.

Choosing one of the classes listed above is not the only requirement the user is supposed to meet when defining an eigenvalue problem. It is also necessary to provide information about the matrices that characterize the problem, to furnish the number of eigenvalues sought and to decide how to stop the Arnoldi method, for example.

This additional information is usually supplied to ARPACK++ by passing parameters to the class constructor when objects are being declared. Although some class constructors include more parameters than others, most of them usually require the user to specify

- the dimension of the eigenvalue problem, `n`;
- the number of eigenvalue to be computed, `nev`;
- one or two matrix objects, depending on whether a standard or a generalized problem is being solved.

These parameters are essential, which means that one cannot set up a problem without defining them. Various other parameters are usually defined internally by ARPACK++, but the user may also supply them when calling the constructor. In this case, the default values are ignored. Among these optional parameters, the most important are:

- `which`, the part of the spectrum that should be computed;
- `ncv`, the number of Arnoldi vectors generated at each iteration of ARPACK;
- `tol`, the relative accuracy to which eigenvalues are to be determined;
- `maxit`, the maximum number of iterations allowed;
- `resid`, an starting vector for the Arnoldi process.

Passing parameters through class constructors, as described above, is a very common procedure in c++. It is also a common practice to define more than one constructor for each class, so the number of parameters required to define slightly different problems can be reduced to a minimum.

In the ARPACK++ library, all classes[6] contain at least four different constructors: a default constructor (with no parameters), a copy constructor (to build an eigenvalue problem from another), a constructor that defines a problem in regular mode and a another one to solve problems using the shift and invert mode spectral transformation. However, several classes contain more than these four constructors.

The spectral transformations available for each class and the specific requirements related to them will described later in this chapter. A detailed description of all ARPACK++ class parameters and constructors can be found in the appendix.

Matrix classes.

Eigenvalue problems arising in real applications are frequently characterized by very large and sparse matrices. Usually, it is convenient to store such matrices in a dense vector, or using the compressed sparse column (CSC) or band format to efficiently perform the matrix-vector products or solve the linear systems required by the Arnoldi method. In such cases, the simplest way of defining a problem is to use some predefined matrix class provided by ARPACK++.

ARPACK++ contains six classes that can be used to store sparse matrices, as shown in the following table. They are divided according to two parameters: the library that is used to solve linear systems and the presence of symmetry. Other two classes are provided for dense matrices. Real and complex data are handled by the same classes, since ARPACK++ uses templates to define them.

[6] Except those classified as pure base classes.

This classification permits the user to supply only the minimum amount of information that is necessary to characterize a matrix. For example, only four parameters are required to create a real square nonsymmetric band matrix: its dimension, the upper and the lower bandwidth and a vector that contains the matrix elements that belong to the band. If the matrix is symmetric, only the upper or the lower triangular nonzero elements must be furnished.

Some instructions on how to declare matrices using these classes are given in the next chapter, which also describes how to define a new matrix class if none of those listed below can efficiently represent the problem being solved.

Library used	**matrix type**	**class name**
SuperLU (CSC format)	symmetric	ARluSymMatrix
	nonsymmetric	ARluNonSymMatrix
UMFPACK (CSC format)	symmetric	ARumSymMatrix
	nonsymmetric	ARumNonSymMatrix
LAPACK (band format)	symmetric	ARbdSymMatrix
	nonsymmetric	ARbdNonSymMatrix
LAPACK (dense)	symmetric	ARdsSymMatrix
	nonsymmetric	ARdsNonSymMatrix

Computational modes.

It is important to select the appropriate spectral transformation strategy for a given problem. Some spectral transformations are required to solve generalized problems, while others can be employed to enhance convergence to a particular portion of the spectrum. However, most of these transformations require the solution of linear systems, so the user must be aware of the memory requirements related to each one of them. Some care must also be taken to assure that the desired portion of the spectrum is computed.

In ARPACK++, these transformations are called *computational modes*. ARPACK++ classes contain a different constructor for each computational mode, so the user must select one of them when declaring an object of a specific class.

ARPACK++ computational modes are listed below. Some examples on how to define the right mode for a specific problem are given in the next chapter and in the appendix. An exhaustive description of all available ARPACK modes can be found in [6].

Real symmetric standard problems.

There are two drivers designed to solve a problem in the standard form $Ax = x\lambda$, with a real symmetric matrix *A*, depending on the portion of the spectrum to be computed. One of these drivers should be selected when declaring objects of `ARSymStdEig`, `ARluSymStdEig` or `ARrcSymStdEig` classes.

1. Regular mode.

This first driver is well suited to find eigenvalues with largest or smallest algebraic value, or eigenvalues with largest or smallest magnitude. Since this mode is straightforward and does not require any data transformation, it only requires the user to supply *A* stored as a dense matrix, or in CSC or band format, or a function that computes the matrix-vector product $y \leftarrow Ax$.

2. Shift-and-invert mode.

This driver may be used to compute eigenvalues near a shift σ and is often used when the desired eigenvalues are clustered or not extreme. With this spectral transformation, the eigenvalue problem is rewritten in the form

$$(A - \sigma I)^{-1} x = x\nu$$

It is easy to prove that ν, the eigenvalues of largest magnitude of $OP = (A - \sigma I)^{-1}$, can be used to calculate the eigenvalues λ that are nearest to σ in absolute value. The relation between them is $\nu = 1/(\lambda - \sigma)$ or $\lambda = \sigma + 1/\nu$. Eigenvectors of both the original and the transformed systems are the same, so no backward transformation is required in this case.

The major drawback of this mode is the necessity of evaluating the matrix-vector product $y \leftarrow OPx$, which means that a function that solves linear systems involving $(A - \sigma I)$ must be available. This function is provided internally by ARPACK++ if `ARluSymStdEig` is being used, but it must be defined by the user when one of the `ARSymStdEig` or `ARrcSymStdEig` classes is employed.

Real symmetric generalized problems.

ARPACK++ also provides three classes, named `ARSymGenEig`, `ARluSymGenEig` and `ARrcSymGenEig`, to solve symmetric real generalized problems in the form

$Ax = Bx\lambda$. These classes include four different *modes* that can be selected according to some problem characteristics

1. Regular mode.

As in the standard case, this mode is well suited to find eigenvalues with largest or smallest algebraic value or magnitude. Two matrix-vector products are performed (and must be supplied if `ARSymGenEig` or `ARrcSymGenEig` are being used) in this mode:

$$w \leftarrow OPx = B^{-1}Ax$$

and

$$z \leftarrow Bx.$$

The regular mode is effective when B is symmetric and positive definite but can not be efficiently decomposed using a sparse Cholesky factorization $B = LL^T$. If this decomposition is feasible and B is well conditioned, then it is better to rewrite the generalized problem in the form

$$(L^{-1}AL^{-T})y = y\lambda,$$

where $L^T x = y$, and use one of the classes designed to solve standard problems. Naturally, in this case, each matrix-vector product $(L^{-1}AL^{-T})z$ should be performed in three steps, including a product in the form $w \leftarrow Ax$ and the solution of two triangular systems.

2. Shift and invert mode.

To find eigenvalues near a shift σ in a generalized problem, it is necessary to transform the problem into

$$(A - \sigma B)^{-1}Bx = x\nu .$$

After finding the eigenvalues of largest magnitude for the above problem, the desired original eigenvalues are easily obtained using the relation $\lambda = \sigma + 1/\nu$, as in the standard case.

This transformation is done automatically by ARPACK++, but the required matrix-vector products, $y \leftarrow OPz$, where $OP = (A - \sigma B)^{-1}$, and $w \leftarrow Bz$, must be performed by the user if the class being used is other than `ARluSymGenEig`. Matrix B is supposed to be symmetric positive semi-definite.

3. Buckling mode.

This mode can also be used to find eigenvalues near a shift σ. If $Ax = Bx\lambda$ is rewritten in the form

$$(A - \sigma B)^{-1} Ax = x\nu ,$$

the largest eigenvalues ν of this system and the eigenvalues of the original problem are related by $\lambda = \sigma\nu / (\nu - 1)$. The matrix-vector products involved in this mode are $y \leftarrow OPz$ and $w \leftarrow Az$, where $OP = (A - \sigma B)^{-1}$. They are required by the `ARSymGenEig` and `ARrcSymGenEig` classes. Moreover, matrix *A* must be symmetric positive semi-definite.

4. Cayley mode.

In this last mode, to find eigenvalues near a shift σ, the system $Ax = Bx\lambda$ is transformed into

$$(A - \sigma B)^{-1} (A + \sigma B) x = x\nu .$$

The relation between the largest eigenvalues of this system and the desired eigenvalues is given by $\lambda = \sigma(\nu + 1)/(\nu - 1)$. In this mode, matrix *B* is required to be symmetric positive semi-definite.

Only the shift must be defined by the user if `ARluSymGenEig` is being used. However, three different matrix-vector products must be supplied to both the `ARSymGenEig` and the `ARrcSymGenEig` classes. These products are $y \leftarrow OPz$, $w \leftarrow Az$ and $u \leftarrow Bz$, where $OP = (A - \sigma B)^{-1}$.

Real nonsymmetric standard problems.

There are also two drivers for nonsymmetric standard eigenvalue problems. They are handled by ARPACK++ classes `ARNonSymStdEig`, `ARluNonSymStdEig` and `ARrcNonSymStdEig`.

1. Regular mode.

This driver can be used to find eigenvalues with smallest or largest magnitude, real part or imaginary part. It only requires the user to supply the nonzero structure of matrix *A* or a function that computes the matrix-vector product $y \leftarrow Ax$. Naturally, when computing eigenvalues of smallest magnitude, the user must consider also the possibility of using A^{-1} and the shift and invert mode with zero shift, because ARPACK is more effective at finding extremal well separated eigenvalues.

2. Shift-and-invert mode.

This driver may be used to compute eigenvalues near a shift σ and is often used when the desired eigenvalues are not extreme (see the section on *real symmetric standard problems* for a brief description of this mode).

When class `ARluNonSymStdEig` is being used, only the shift must be furnished. However, to use this spectral transformation combined with one of the class `ARNonSymStdEig` or `ARrcNonSymStdEig`, the user must supply a function that evaluates the matrix-vector product

$$y \leftarrow OPx = (A - \sigma I)^{-1} x$$

where σ is supposed to be real. To define a complex shift, the user should use a generalized driver or redefine the problem as a complex one.

Real nonsymmetric generalized problems.

To find eigenvalues of nonsymmetric generalized problems, the user can use one of the three different modes supplied by ARPACK++ classes `ARNonSymGenEig`, `ARluNonSymGenEig` and `ARrcNonSymGenEig`. These modes differ on the part of the spectrum that is sought. All of them require matrix *B* to be symmetric positive semi-definite.

1. Regular mode.

As in the symmetric case, to solve problems in regular mode the user can supply only the nonzero structure of matrices *A* and *B* in CSC format. As an alternative, it is also possible to supply two functions: one that computes the matrix-vector product

$$w \leftarrow OPx = B^{-1} A x$$

and other that returns the product $z \leftarrow Bx$. Again, this mode is effective when *B* is ill-conditioned (nearly singular) or when *B* cannot be efficiently decomposed using a sparse Cholesky factorization $B = LL^T$. If *B* is well conditioned and can be factored then the conversion to a standard problem is recommended.

2. Real shift and invert mode.

This mode can be used to find eigenvalues near a real shift σ. Only matrices *A*, *B* and the shift σ are required to use class `ARluNonSymGenEig`. When using `ARNonSymGenEig` or `ARrcNonSymGenEig`, the user must supply two functions that evaluate the products:

$$y \leftarrow OPx = (A - \sigma B)^{-1} x \quad \text{and} \quad w \leftarrow Bx .$$

See the section about symmetric problems for more details.

3. Complex shift and invert mode.

To find eigenvalues near a complex shift σ in a nonsymmetric generalized problem, ARPACK++ needs to perform three different matrix-vector products:

$$y \leftarrow OPx\,, \quad v \leftarrow Bx \quad \text{and} \quad w \leftarrow Ax\,,$$

where OP can be set to the real or imaginary part of $(A-\sigma B)^{-1}$. The first two products are used to find the eigenvalues of largest magnitude of the problem

$$(A-\sigma B)^{-1}Bx = x\nu\,.$$

The last product is required to recover the eigenvalues of the original problem. Because the relation between ν and λ is not bijective in this case, the Rayleigh quotient

$$\lambda = x^H Ax \big/ x^H Bx$$

is used to obtain the correct eigenvalues.

These products are internally performed if class `ARluNonSymStdEig` is being used. Otherwise, they must be supplied by the user.

Complex standard problems.

If the eigenvalue problem $Ax = x\lambda$ has complex data, one of the two drivers of classes `ARCompStdEig`, `ARluCompStdEig` and `ARrcCompStdEig` should be used. These drivers are similar to those presented above, and are briefly described here.

1. Regular mode.

This driver can be used to find eigenvalues with smallest or largest magnitude, real part or imaginary part. $y \leftarrow Ax$ is the only matrix-vector product required to solve a problem in this mode. This product must be supplied if a class other than `ARluCompStdEig` is used.

2. Shift-and-invert mode.

This driver may be used to compute eigenvalues near a complex shift σ. When one of the `ARCompStdEig` or `ARrcCompStdEig` classes is being used, the user must supply a function that evaluates the matrix-vector product

$$y \leftarrow OPx = (A-\sigma I)^{-1}x$$

Complex generalized problems.

This is the last group of problems ARPACK++ is able to solve. The corresponding classes are called `ARCompGenEig`, `ARluCompGenEig` and `ARrcCompGenEig`. These classes also includes two drivers.

1. Regular mode.

When solving generalized problems in regular mode without using the `ARluCompStdEig` class, the user is required to supply two functions that compute the matrix-vector products

$$w \leftarrow OPx = B^{-1}Ax \quad \text{and} \quad z \leftarrow Bx\,.$$

If `ARluCompStdEig` is being used, *A* and *B* must be supplied as a dense matrix or in band or compressed sparse column format.

2. Shift and invert mode.

This mode can be used to find eigenvalues near a complex shift σ. When using one of the `ARCompStdEig` or `ARrcCompStdEig` classes, the user must supply two functions that evaluate the products

$$y \leftarrow OPx = (A - \sigma B)^{-1}x \quad \text{and} \quad w \leftarrow Bx\,.$$

See the section about symmetric problems for more details.

ARPACK++ functions.

ARPACK++ classes contain several member functions designed to supply information about problem data, to change some parameters, to solve problems in various computational modes and to return eigenvalues and eigenvectors. Most of them are defined as virtual members of only four classes — `ARrcStdEig`, `ARrcNonSymStdEig`, `ARrcSymStdEig` and `ARrcCompStdEig` — and inherited or changed by other derived classes. This procedure reduces the necessity of redefining functions and permits the user to add his own classes to the library by only defining a few constructors.

The functions can be divided according to their purposes into eight groups. These groups are summarized below. Only a brief explanation of each function is given, so the user should refer to the appendix for a more complete description of them.

Setting problem parameters.

Generally, an eigenvalue problem is set by passing all problem parameters to the class constructor. However, sometimes the parameters may not be available when the problem is declared, so the user may be forced to define them later. The functions listed below are intended to help the user in such cases.

`DefineParameters`	Sets the values of variables that are usually passed as parameters to class constructors.
`SetBucklingMode`	Turns a generalized symmetric real problem into buckling mode.
`SetCayleyMode`	Turns a generalized symmetric real problem into Cayley mode.
`SetComplexShiftMode`	Turns a generalized nonsymmetric real problem into complex shift and invert mode.
`SetRegularMode`	Turns any eigenvalue problem into regular mode.
`SetShiftInvertMode`	Turns any eigenvalue problem into shift and invert mode.

Changing problem parameters.

Although changes in problem data are not very common, they are allowed by ARPACK++ to permit the user to overcome some atypical situations were program fails to solve a problem with the mode or other parameter previously chosen. The functions that can be used with this purpose are:

`ChangeMaxit`	Changes the maximum number of iterations allowed.
`ChangeMultBx`	Changes the function that performs the product *Bx*.
`ChangeMultOPx`	Changes the function that performs the product *OPx*.
`ChangeNcv`	Changes the number of Arnoldi vectors generated at each iteration.
`ChangeNev`	Changes the number of eigenvalues to be computed.
`ChangeShift`	Turns the problem into shift and invert mode (or changes the shift if this mode is already being used).
`ChangeTol`	Changes the stopping criterion.
`ChangeWhich`	Changes the part of the spectrum that is sought.
`InvertAutoShift`	Changes the shift selection strategy used to restart the Arnoldi method.

`NoShift`	Turns the problem into regular mode.

Retrieving information about the problem.

Some ARPACK++ functions can be helpful if one wants to know which parameters were employed to solve an eigenvalue problem. They can be used, for example, to build other functions that require information about some details of the eigenvalue problem, such as the computational mode or the stopping criterion adopted, without explicitly passing each parameter in the function heading.

A list of all the ARPACK++ functions that return problem data is given below. Some of them are also used in various examples included in next two chapters.

`GetAutoShift`	Indicates if exact shifts are being used to restart the Arnoldi method.
`GetMaxit`	Returns the maximum number of iterations allowed.
`GetIter`	Returns the number of iterations actually taken to solve a problem.
`GetMode`	Returns the computation mode used.
`GetN`	Returns the dimension of the problem.
`GetNcv`	Returns the number of Arnoldi vectors generated at each iteration.
`GetNev`	Returns the number of required eigenvalues.
`GetShift`	Returns the shift used to define a spectral transformation.
`GetShiftImag`	Returns the imaginary part of the shift.
`GetTol`	Returns the tolerance used to declare convergence.
`GetWhich`	Returns the portion of the spectrum that was sought.
`ParametersDefined`	Indicates if all problem parameters were correctly defined.

Determining eigenvalues and eigenvectors.

The most important and most frequently used ARPACK++ functions are listed below. With them, one can determine eigenvalues, eigenvectors, Schur vectors or an Arnoldi basis.

Instead of containing one single function that solves the eigenvalue problem, ARPACK++ gives the user various alternatives to determine and store just the desired part of the solution. There are eleven different functions. Each one stores a particular group of vectors using a specific output format.

The three output formats available are used here to group the functions.

1. Functions that use ARPACK++ internal data structure.

This first group contains functions that solve the eigenvalue problem and store the output vectors into ARPACK++ internal data structure, so the user does not need to worry about how and where eigenvalues and eigenvectors are stored.

The output data generated by these functions can be retrieved later by using one of the several functions described in the *Retrieving eigenvalues and eigenvectors* section below.

`FindArnoldiBasis`	Determines an Arnoldi basis.
`FindEigenvalues`	Determines eigenvalues.
`FindEigenvectors`	Determine eigenvectors (and optionally Schur vectors).
`FindSchurVectors`	Determines Schur vectors.

2. Functions that store output data in user-supplied vectors.

Using functions of this second group, it is possible to solve the eigenvalue problem and store the output data in user-supplied c++ standard vectors.

`Eigenvalues`	Returns the eigenvalues of the problem being solved and optionally determines eigenvectors and Schur vectors.
`EigenValVectors`	Returns the eigenvalues and eigenvectors of the given problem (and optionally determines Schur vectors).
`Eigenvectors`	Return the eigenvectors of the given problem (and optionally determine Schur vectors).

3. Functions that generate objects of the STL vector class.

Functions of this last group are used to solve the eigenvalue problems and return output data into objects of the STL vector class.

`StlArnoldiBasisVectors`	Returns an Arnoldi basis for the problem being solved.
`StlEigenvalues`	Returns a vectors that contains the eigenvalues of the given problem. Optionally, Eigenvectors and Schur

	vectors can also be determined and stored into ARPACK++ internal data structure.
`StlEigenvectors`	Returns a vector that stores sequentially the eigenvectors of the problem being solved. Eigenvalues (and optionally Schur vectors) are also determined and stored internally by ARPACK++.
`StlSchurVectors`	Returns a vector that contains the Schur vectors of the problem being solved.

Tracking the progress of ARPACK.

The FORTRAN version of ARPACK provides a means to trace the progress of the computation of the eigenvalues and eigenvectors as it proceeds. Various levels of output are available, from no output to voluminous. This feature is also supported by ARPACK++ through the two functions listed below:

`Trace`	Turns trace mode on.
`NoTrace`	Turns trace mode off.

Executing ARPACK++ step by step.

The *reverse communication interface* classes requires the user to interact with ARPACK++ and perform matrix-vector products on request during the computation of the eigenvalue and eigenvectors.

However, to perform a product, say $y \leftarrow Mx$, one needs to know where x is stored, and also where to put y. The same occurs when the user decides to supply the shifts for the implicit restarting of the Arnoldi method: he must know where to store the shifts. This kind of information is provided by the functions listed below.

`GetIdo`	Indicates the operation that must be performed by the user between two successive calls to TakeStep.
`GetNp`	Returns the number of shifts that must be supplied for the implicit restarting of the Arnoldi method.
`GetProd`	Indicates where the product Bx is stored.
`GetVector`	Indicates where x is stored when a product in the form Mx must be performed.
`GetVectorImag`	Indicates where the imaginary part of the eigenvalues of the current Hessenberg matrix are stored.
`PutVector`	Indicates where to store the product OPx (or Bx).

`TakeStep`	Performs the calculations required between two successive matrix-vector products.

Detecting if the solution of the eigenvalue problem is available.

In various situations, notably when solving the eigenvalue problem step by step, the user needs to find out if the solution of the problem is already available, in order to proceed with his own computations. In such cases, one of the functions listed below should be used.

`ConvergedEigenvalues`	Returns the number of eigenvalues found so far.
`ArnoldiBasisFound`	Indicates if an the requested Arnoldi basis is available.
`EigenvaluesFound`	Indicates if the requested eigenvalues are available.
`EigenvectorsFound`	Indicates if the requested eigenvectors are available.
`SchurVectorsFound`	Indicates if the requested Schur vectors are available.

Retrieving eigenvalues and eigenvectors.

Various functions contained in the *Determining eigenvalues and eigenvectors* section above (`FindEigenvalues` and `Eigenvectors` are just two of them) use ARPACK++ internal data structure to store part of the solution, or even the whole solution of the eigenvalue problem.

This section contains several functions that permit the user to retrieve those output vectors internally stored by ARPACK++. The functions listed below can be used to obtain from a particular element of an Arnoldi basis vector to a vector that contains all eigenvectors stored sequentially.

1. Functions that return vector elements.

For those people that do not want to worry about how and where to store eigenvalues and eigenvectors, ARPACK++ includes some functions that permit direct access to every single element of the output vectors. These functions are listed below.

`ArnoldiBasisVector`	Returns one element of an Arnoldi basis vector.
`Eigenvalue`	Returns one of the “converged” eigenvalues.
`EigenvalueReal`	Returns the real part of an eigenvalue (when the problem is real and nonsymmetric).
`EigenvalueImag`	Returns the imaginary part of an eigenvalue (when the problem is real and nonsymmetric).

`Eigenvector`	Returns one element of a single eigenvector.
`EigenvectorReal`	Returns the real part of one element of an eigenvector (when the problem is real and nonsymmetric).
`EigenvectorImag`	Returns the imaginary part of one element of an eigenvector (when the problem is real and nonsymmetric).
`SchurVector`	Returns one element of a Schur vector.
`ResidualVector`	Returns one element of the residual vector.

2. Functions that return pointers to vectors.

ARPACK++ also includes functions that return vector addresses instead of vector components. Their purpose is to permit the user to supply eigenvalues and eigenvectors (or any other vector stored into ARPACK++ internal data structure) as input parameters to other functions.

`RawArnoldiBasisVector`	Returns a pointer to a vector that stores one of the Arnoldi basis vectors.
`RawArnoldiBasisVectors`	Returns a pointer to a vector that contains the Arnoldi basis.
`RawEigenvalues`	Returns a pointer to a vector that contains all the eigenvalues (or the real part of them, if the problem is real and nonsymmetric).
`RawEigenvaluesImag`	Returns a pointer to a vector that contains the imaginary part of all the eigenvalues, when the problem is real and nonsymmetric.
`RawEigenvectors`	Returns a pointer to a vector that stores all the eigenvalues consecutively.
`RawEigenvector`	Returns a pointer to a vector that stores one of the eigenvectors.
`RawSchurVectors`	Returns a pointer to a vector that stores the Schur vectors consecutively.
`RawSchurVector`	Returns a pointer to a vector that stores one of the Schur vectors.
`RawResidualVector`	Returns a pointer to a vector that contains the residual vector.

3. Functions that return STL vectors.

There are also functions that return output vectors using the STL `vector` class. Besides `StlEigenvalues`, `StlEigenvectors` and `StlSchurVectors`, listed earlier in this chapter, this group also includes:

`StlArnoldiBasisVector`	Returns one of the Arnoldi basis vectors.
`StlEigenvaluesReal`	Returns the real part of the eigenvalues, when the problem is real and nonsymmetric.
`StlEigenvaluesImag`	Returns the imaginary part of the eigenvalues, when the problem is real and nonsymmetric.
`StlEigenvector`	Returns one of the eigenvectors.
`StlEigenvectorReal`	Returns the real part of an eigenvector, when the problem is real and nonsymmetric.
`StlEigenvectorImag`	Returns the imaginary part of an eigenvector, when the problem is real and nonsymmetric.
`StlSchurVector`	Returns one of the Schur vectors.
`StlResidualVector`	Returns the residual vector.

Chapter

4

Solving eigenvalue problems

The purpose of this chapter is to show how easily one can define and solve eigenvalue problems using ARPACK++ classes. There is no intent to cover every single ARPACK++ detail here, but only to stress the most important characteristics of each kind of class and function, and give some hints that should be followed by the user when solving his own problems.

Solving problems in four steps.

As emphasized in chapter 3, ARPACK++ has a large number of classes and functions. This profusion of classes is easy to justify. It gives the user various alternatives to define and solve eigenvalue problems without having to pass extra parameters when calling constructors.

However, one can easily get confused when so many choices are available, especially when using the library for the first time. Therefore, the actions needed to define and solve an eigenvalue problem using a few simple steps shall be emphasized:

Step One. First of all, it is necessary to create one or more matrices using some user-defined class or one of the eight matrix classes provided by ARPACK++. If the user does not want to represent a matrix by means of a class, he still can use the *reverse communication interface*, but this option is not recommended and should be considered only after discarding the previous alternatives.

Step Two. Once available, these matrices must be used to declare the eigenvalue problem. Other relevant parameters, such as the number of desired eigenvalues, the spectral transformation and the shift, should also be defined.

Step Three. After that, the user must call one of the ARPACK++ functions specifically designed to solve the eigenvalue problem. `EigenValVectors` and `FindEigenvectors` are just two of these functions.

Step Four. Finally, some other ARPACK++ function can also be called to retrieve output data, such as eigenvalues, eigenvectors and Schur vectors, if this was not done by the function used in the third step above.

Notice that several functions mentioned in the last chapter were not included in these steps. Functions whose purpose is to change some of the problem parameters (such as the tolerance or the maximum number of iterations) or turn on the trace mode, for example, are seldom used and, because of their secondary role, will be described only in the appendix.

Defining matrices.

From the user's point of view, the hardest step in the list given above is the definition of the matrices that characterize the eigenvalue problem. This is particularly true when the problem is large.

The difficulty comes from the fact that, in order to define a matrix, it is necessary not only to store its elements, but also to create a function that performs a matrix-vector product and, in the case of a spectral transformation is being used, to define how a linear system should be solved.

Two different schemes are provided by ARPACK++ to mitigate this difficulty: one can use a predefined class, and let the library handle the matrices, or use his own class to define the required matrix-vector products.

In fact, the reverse communication interface can also be used, so it is even possible to avoid completely the use of a c++ class to store information about the matrix. In this case, the user is totally free to decide how matrix-vector products should be performed. However, because this liberty implies a much more complicated code, only the first two alternatives will be considered in this section.

Using ARPACK++ matrix classes.

The easiest way to create a matrix is to use one of the eight predefined classes provided by ARPACK++. These classes already contain member functions that perform matrix-vector products and solve linear systems, so the user needs only to supply matrix data in compressed sparse column or band format in order to use them to define a matrix. A single vector will suffice if the matrix is dense.

The first example below illustrates how a real nonsymmetric band matrix can be declared as an `ARbdNonSymMatrix` object.

```
int     n     = 10000;
int     nL    = 6;
int     nU    = 3;
double* nzval = MatrixData().

ARbdNonSymMatrix<double> A(n, nL, nU, nzval);
```

In this example, `n` is the dimension of the system, `nL` and `nU` are, respectively, the lower and the upper bandwidth (not considering the main diagonal), and `nzval` is a vector that contains all elements of the (`nL`+`nU`+1) nonzero diagonals of `A`. This last vector is generated by function `MatrixData` (not shown here).

Once declared, `A` can be passed as a parameter to all ARPACK++ classes that define a nonsymmetric eigenvalue problem. However, since class `ARbdNonSymMatrix` (like all other predefined matrix classes) uses a direct method to solve linear systems, one must take in account the memory that will be consumed if a spectral transformation is employed.

The next example contains the definition of a sparse complex matrix using class `ARluNonSymMatrix`.

```
int                n;     // Matrix dimension.
int                nnz;   // Number of nonzero elements in A.
int*               irow;  // pointer to an array that stores the row
                          // indices of the nonzeros in A.
int*               pcol;  // pointer to an array of pointers to the
                          // beginning of each column of A in nzval.
complex<double>*   nzval; // pointer to an array that stores the
                          // nonzero elements of A.

n = 10000;
CompMatrixA(n, nnz, nzval, irow, pcol);
ARluNonSymMatrix<complex<double> > A(n, nnz, nzval, irow, pcol);
```

Here, `CompMatrixA` is a function that generates `nnz`, `irow`, `pcol` and `nzval`. These four parameters, along with `n`, are used to define matrix `A`. In addition to them, the relative pivot tolerance and the column ordering that should be used to reduce the fill-ins that occur during the matrix factorization can also be passed to the `ARluNonSymMatrix` class constructor.

Letting the user define a matrix class.

If none of the matrix classes mentioned above meets the user's requirements, either because the data structure is not appropriate or due to the use of a direct method for solving linear systems, a new class can be defined from scratch. In this case, the class must contain some member functions that performs the matrix-vector products required by the Arnoldi method.

Different classes with particular member functions must be created for each combination of matrix (real symmetric, real nonsymmetric or complex) and computational mode (regular, shift and invert, etc) used to solve the eigenvalue problem. To solve, in regular mode, a standard eigenvalue problem that involves a real nonsymmetric matrix *A*, for example, one needs to define a matrix class with at least one member function that performs the product $w \leftarrow Av$, as shown below.

```
template<class T>
class MatrixWithProduct {

 private:

  int m, n;   // Number of rows and columns.

 public:

  int nrows() { return m; }

  int ncols() { return n; }

  void MultMv(T* v, T* w);   // Matrix-vector product: w = M*v.

  MatrixWithProduct(int nrows, int ncols = 0) // Simple constructor.
  {
    m = nrows;
    n = (ncols?ncols:nrows);
  }

}; // MatrixWithProduct.
```

The only condition imposed to this class by ARPACK++ is that `MultMv`, the function that performs the required matrix-vector product, contains only two parameters[7]. The first parameter must be a pointer to the vector that will be multiplied by *A*, while the second parameter must supply a pointer to the output vector. This is not a very strong restriction since any other information about the matrix, such as the number of rows or columns, can be passed indirectly to `MultMv` by using some class variables.

In the example above, parameters `v` and `w` are declared as pointers to a certain type `T`, allowing `MatrixWithProduct` to represent both single and double precision matrices.

[7] Naturally, default arguments are also allowed.

Other two variables used by `MultMv`, `m` and `n`, are defined when the constructor is called.

Creating eigenvalue problems.

There are two different ways of declaring an eigenvalue problem as an ARPACK++ object. The user can either define all problem parameters when creating the object or use a default constructor and define parameters later. Both alternatives are briefly described below.

Passing parameters to constructors.

All information that is necessary to set up the eigenvalue problem can be specified at once when declaring an object of the corresponding class. For example, to find the five eigenvalues closest to 5.7 + 2.3*i* of a complex generalized problem using the shift and invert mode, the user can declare an object of class `ARCompGenEig` writing

```
ARCompGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(10000, 5, &OP, &MatrixOP<double>::MultVet, &B,
          &MatrixB<double>::MultVet, complex<double>(5.7, 2.3));
```

Here, 10000 is the dimension of the system and `MatrixOP<double>::MultVet` and `MatrixB<double>::MultVet` are functions that evaluate the products $(A-\sigma B)^{-1}v$ and Bv, respectively.

The same complex problem mentioned above can be declared in a more straightforward way if the `ARluCompGenEig` class is used. In this case, after defining `A` and `B` as two `ARluNonSymMatrix` objects, one just needs to write

```
ARluCompGenEig<double> EigProb(5, A, B, complex<double>(5.7, 2.3));
```

Real symmetric and nonsymmetric standard and generalized problems can be created in an analogous manner.

Defining parameters after object declaration.

There are some cases where it is not necessary, and sometimes not even convenient, to supply all problem information when declaring an ARPACK++ object. If some parameter is not available when problem is being declared, for example, all data can be passed to ARPACK++ later, as in the following real nonsymmetric generalized problem:

```
ARNonSymGenEig<double, MatrixOP<double>, MatrixB<double> > EigProb;

// ...

EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
EigProb.SetShiftInvertMode(1.2, &OP, &MatrixOP<double>::MultVet);
```

In this example, the shift and invert mode will be used to find 4 eigenvalues near 1.2. The dimension of the problem is 100 and matrix-vector products are functions of classes `MatrixOP<double>` and `MatrixB<double>`. The first line only declares an object called EigProb. In the last two lines, all `ARNonSymGenEig` parameters are defined, including the spectral transformation mode.

Solving problems and getting output data.

Once declared an eigenvalue problem, ARPACK++ provides several alternatives to retrieve its solution. These alternatives are briefly described below.

Letting ARPACK++ handle data.

When solving an eigenvalue problem, ARPACK++ can hold the output vectors in its own data structure, so the user does not need to decide where they should be stored. In this case, each single element of the eigenvalues and eigenvectors can be recovered later using some functions provided by ARPACK++, as in the following example:

```
// Finding and printing a few eigenvectors of EigProb.

EigProb.FindEigenvectors();

for (int i=0; i<EigProb.ConvergedEigenvalues(); i++) {
  cout << “Eigenvalue[“ << (i+1) << “] = “;
  cout << EigProb.Eigenvalue(i) << endl;
  cout << “Eigenvector[“ << (i+1) << “] : “;
  for (j=0; j<EigProb.GetN(); j++) {
    cout << EigProb.Eigenvector(i, j) << endl;
  }
  cout << endl;
}
```

Here, `FindEigenvectors` is a function that determines eigenvalues and eigenvectors of a problem defined by `EigProb`, and store them into ARPACK++ internal structure. `ConvergedEigenvalues` returns the number of eigenvalues found by ARPACK++. To retrieve output data, functions `Eigenvalue` and `Eigenvector` were used[8].

[8] `SchurVector` and `ResidualVector` are other functions that could be used.

ARPACK++ also includes other functions that return vector addresses instead of vector elements. These functions provide direct access to output data without requiring the user to create a vector. They are well suited to those situations where eigenvalues and eigenvectors must be supplied as parameters to other functions. `RawEigenvector`, one of such functions[9], is used in the example below:

```
// ...
EigProb.FindEigenvectors();             // Finding eigenvectors.
double* w = new double[EigProb.GetN()]; // Defining a vector w.
A.MultMv(EigProb.RawEigenvector(0), w); // Setting w <- matrix*Eigenvector
// ...
```

In this example, `A`, a matrix declared elsewhere in the program, is multiplied by the first eigenvector of an eigenvalue problem defined by EigProb. `A.MultMv`, the function that performs the product, takes two pointers to double precision real vectors as parameters. ARPACK++ function `GetN` is used to determine the dimension of the problem.

Employing user-defined data structure.

ARPACK++ also permits the user to use his own vectors to store the solution of an eigenvalue problem. As an example, a function called `EigenValVectors` is used below to determine the `nconv` eigenvalues and eigenvectors of a real nonsymmetric standard problem (represented by `EigProb`). Similar functions can be used to find Arnoldi basis vectors, Schur vectors, etc.

```
double EigValR[10];
double EigValI[10];
double EigVec[1100];
int    nconv;

nconv = EigProb.EigenValVectors(EigVec, EigValR, EigValI);
for (int i=0; i<nconv; i++) {
  cout << "Eigenvalue[" << (i+1) << "] = ";
  cout << EigValR[i] << " + " << EigValI[i] << "I" << endl;
}
```

Since `EigProb` is a nonsymmetric problem and, in this case, some of the eigenvalues can be complex, two real vectors, `EigValR` and `EigValI`, are used to store, respectively, the real and imaginary part of the eigenvalues.

The eigenvectors are stored sequentially in `EigVec`. Real eigenvectors occupy `n` successive positions[10], while each complex eigenvector require 2*`n` positions (`n` for the real part and another `n` for the imaginary part of the vector). Since the last eigenvector

[9] Other functions with similar meaning are `RawEigenvalues` and `RawSchurVector`.
[10] Here, `n` is the dimension of the system.

found by `EigenValVectors` can be complex, `EigVec` must be dimensioned to store (`nconv`+1)*`n` elements.

Using the STL vector class.

Last but not least, ARPACK++ can store eigenvalues and eigenvectors using the `vector` class provided by the *Standard Template Library* (or *STL*).

STL is a library that provides an easy and powerful way to handle vectors, linked lists and other structures in c++. Among all its classes templates, only the `vector` class can be considered appropriate to store the dense vectors generated as output by ARPACK++. This class is used in the example below:

```
vector<double>* EigVec = prob.StlEigenvectors();
vector<double>* EigVal = prob.StlEigenvalues();

for (int i=0; i<prob.ConvergedEigenvalues(); i++) {
  cout << “Eigenvalue[“ << (i+1) << “] = “ << EigVal[i] << endl;
}
```

In this example, `StlEigenvectors` not only finds the eigenvectors of a problem called `prob`, but also creates a new object of class `vector` to store them sequentially, returning a pointer to this vector in `EigVec`. EigVal is used to store the pointer generated by `StlEigenvalues`. The number of eigenvalues found by ARPACK++ is supplied by function `ConvergedEigenvalues`.

Finding singular values.

ARPACK++ can also be used to find the *truncated singular value decomposition (truncated SVD)* of a generic real rectangular matrix. Supposing, for example, that *A* is a $m \times x$ matrix, the truncated SVD is obtained by decomposing *A* into the form

$$A = U \Sigma V^T$$

where *U* and *V* are matrices with orthonormal columns, $U^T U = V^T V = I_n$, and $\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_n)$ is a diagonal matrix that satisfies $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

Each element σ_i is called a *singular value* of *A*, while each column of *U* is a *left singular vector* and each column of *V* is a *right singular vector* of *A*.

To use ARPACK++ to obtain a few singular values (and the corresponding singular vectors) of *A*, one should notice that $\sigma_1, \sigma_2, \ldots, \sigma_n$ are precisely the square roots of the eigenvalues of the symmetric $n \times n$ matrix

$$A^T A = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T$$

and, in this case, the eigenvectors of $A^T A$ are the right singular vectors of A.

Naturally, this formulation is appropriate when m is greater or equal to n. To solve problems where $m < n$, it is sufficient to reverse the roles of A and A^T in the above equation.

When the singular values obtained by ARPACK++ are not multiple or tightly clustered, numerically orthogonal left singular vectors may also be computed from the right singular vectors using the relation:

$$U = AV\Sigma^{-1}.$$

As an alternative, one can use the relation

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}\begin{pmatrix} U \\ V \end{pmatrix} = \begin{pmatrix} U \\ V \end{pmatrix}\Sigma.$$

to determine the left and right leading singular vectors simultaneously. In this case, no transformation is required since the columns of U and V can be easily extracted from the converged eigenvectors of

$$\overline{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}.$$

In view of the fact that $\overline{A}$ has both σ_i and $-\sigma_i$ as eigenvalues, it is important to set the `which` variable to “`LA`” when calling ARPACK++, so only the positive eigenvalues (those with largest algebraic value) are computed.

The major drawback of this approach is related to the fact that $\overline{A}$ is an $(m+n)\times(m+n)$ matrix, while $A^T A$ contains only n^2 elements. Even considering that the sparse matrix-vector products $\overline{A}v$ and $A^T Av$ require the same amount of float point operations, the Arnoldi vectors generated at each iteration of ARPACK are greater when $\overline{A}$ is used. Moreover, setting `which` to “`LM`” is generally better than using “`LA`”.

As a result, in most cases it is better to use $A^T A$ than $\overline{A}$. Exceptions to this rule occur only when the leading eigenvalues of A are very tightly clustered.

Using the `ARSymStdEig` class.

ARPACK++ class `ARSymStdEig` can be easily adapted to solve SVD problems. This is particularly true if `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` is used to store matrix *A*, because these classes contain three member functions, `MultMtMv`, `MultMMtv` and `MultOMMtO`, that perform, respectively, the products $A^T Av$, $AA^T v$ and $\overline{A}.v$ for a given *v*.

Supposing, for example, that vectors `valA`, `irow` and `pcol` are used to store *A* in CSC format, so `ARluNonSymMatrix` can be used to define the matrix, the following commands are sufficient to find the four leading singular values of *A*.

```
// Using ARluNonSymMatrix to store matrix A and to perform the product
// A'Ax (LU decomposition is not used, so SuperLU is not required).

ARluNonSymMatrix<double> A(m, n, nnz, valA, irow, pcol);

// Defining the eigenvalue problem (MultMtM is used, so m >= n).

ARSymStdEig<double, ARluNonSymMatrix<double> >
  prob(n, 4, &A, &ARluNonSymMatrix<double>::MultMtMv);

// Finding eigenvalues.

double svalue[4];
dprob.Eigenvalues(svalue);

// Calculating the singular values.

for (i = 0; i < prob.ConvergedEigenvalues(); i++) {
  svalue[i] = sqrt(svalue[i]);
}
```

Other interesting examples where ARPACK++ is used to find singular values and vectors can be found in the `arpack++/examples` directory.

Chapter

5

ARPACK++ examples

This chapter contains some examples on how to use ARPACK++. The purpose of these examples is to illustrate the major characteristics of the software and to clarify the steps required to find eigenvalues and eigenvectors mentioned in the last chapter.

Several combinations of matrix classes, eigenvalue problems and output functions are considered here. Problems where ARPACK++ matrix classes were used are presented first, followed by some examples that involve user-defined matrix-vector products and the reverse communication interface. Some strategies to build an interface between ARPACK++ and other libraries are also briefly mentioned.

The `examples` *directory.*

The problems mentioned in this chapter are also distributed as examples along with ARPACK++ code. The `arpack++/examples` directory contains some subdirectories — such as `superlu`, `product`, `umfpack`, `harwell`, `dense`, `band` and `reverse` — that include several sample programs covering all available spectral transformations for real symmetric, real nonsymmetric and complex problems. Although the purpose of these programs is only to illustrate ARPACK++ usage, they can also be employed to create new problems. The user just need to replace the matrix data or the matrix-vector product functions.

Some instructions on how to run these examples are given in `README` files included in all of the example directories. The required `Makefiles` are also supplied. However, prior to compiling the programs, some modifications should be made to the `Makefile.inc` file in order to correctly define the compiler and the path of the libraries referenced by ARPACK++.

Using ARPACK++ matrix structure.

Complex and real symmetric and nonsymmetric eigenvalue problems can be easily solved by ARPACK++ when matrix elements are stored in compressed sparse column (CSC) or band format (or sequentially in a vector, if the matrix is dense). In this case, only a few commands are required to obtain the desired eigenvalues and eigenvectors. To illustrate this, three different examples were included in this section. In the first, a real symmetric generalized problem is solved by using the Cayley mode. The second contains a complex standard problem that is solved in regular mode. Finally, ARPACK++ is also used to find some singular values of a real nonsymmetric matrix.

Solving a symmetric generalized eigenvalue problem.

In this first example, the Cayley mode[11] is used to find the four eigenvalues nearest to 150 of a generalized symmetric problem in the form $Ax = Bx\lambda$, where A is the one-dimensional discrete Laplacian on the interval [0, 1], with zero Dirichlet boundary conditions, and B is the mass matrix formed by using piecewise linear elements on the same interval. Both matrices are tridiagonal. This example is very similar to the one found in the `examples/band/sym/bsymgcay.cc` file.

1. Generating problem data.

Before generating A and B, it is worth noticing that, being symmetric, these matrices can be perfectly characterized by their upper or their lower triangular part. Therefore, some memory can be saved if not all their elements are stored.

Two functions, `MatrixA` and `MatrixB`, will be used here to create A and B, respectively. These functions have two input parameters:

- `n`, the dimension of the system; and
- `uplo`, a parameter that indicates which part of the matrix will be supplied;

and two output parameters,

- `nD`, the number of upper or lower nonzero diagonals (not including the main diagonal);
- `A`, a pointer to a vector that contains the nonzero matrix elements.

These output parameters are the minimum amount of information required by ARPACK++ to store a matrix as an `ARbdSymMatrix` object, so it can be used later to create an eigenvalue problem.

[11] See chapter 4 for a description of all computational modes available in ARPACK++.

Since the dimension of the vector pointed by `A` depends on `nD`, a parameter that is not known in advance by the user, `MatrixA` and `MatrixB` also allocate memory for this vector, as shown below.

```
template<class FLOAT, class INT>
void MatrixA(INT n, INT& nD, FLOAT* &A, char uplo = 'L')
{

  // Declaring internal variables.

  INT    i;
  FLOAT  h, df, dd;

  // Defining constants.

  h  = 1.0/FLOAT(n+1);   // mesh size.
  dd = 2.0/h;            // using 2/h instead of 2/h^2.
  df = -1.0/h;           // using 1/h instead of 1/h^2.

  // Defining the upper (or lower) bandwidth.

  nD  = 1;

  // Creating output vector A.

  A   = new FLOAT[2*n];

  if (uplo == 'L') {   // Storing the lower triangular part of A.

    for (i=0; i<n; i++) {
      A[2*i] = dd;                // Main diagonal element.
      if (n-i-1) A[2*i+1] = df;   // Lower diagonal element.
    }

  }
  else {               // Storing the upper triangular part of A.

    for (i=0; i<n; i++) {
      if (i) A[2*i]  = df;        // Upper diagonal element.
      A[2*i+1] = dd;              // Main diagonal element.
    }

  }

} // MatrixA.


template<class FLOAT, class INT>
void MatrixB(INT n, INT& nD, FLOAT* &A, char uplo = 'L')
{

  // Declaring internal variables.

  INT    i;
  FLOAT  h, df, dd;
  // Declaring constants.

  h  = 1.0/FLOAT(n+1);
  dd = (4.0/6.0)*h;
  df = (1.0/6.0)*h;

  // Defining the upper (or lower) bandwidth.

  nD  = 1;

  // Creating output vector A.

  A   = new FLOAT[2*n];

  if (uplo == 'L') {   // Storing the upper triangular part of B.


    for (i=0; i<n; i++) {
      A[2*i] = dd;                // Main diagonal element.
      if (n-i-1) A[2*i+1] = df;   // Lower diagonal element.
    }

  }
  else {               // Storing the upper triangular part of B.

    for (i=0; i<n; i++) {
      if (i) A[2*i] = df;         // Upper diagonal element.
      A[2*i+1] = dd;              // Main diagonal element.
    }

  }

} // MatrixB.
```

`MatrixA` and `MatrixB` were defined here as function templates. The first template parameter, `FLOAT`, permits the function to create both single and double precision matrices. The second parameter, `INT`, represents the integer type used and must be set to `int` or `long int`.

Since these functions do not make clear how a band symmetric matrix can be stored in a single vector, this will be illustrated by the example given below.

Consider the matrix

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}.$$

M is a generic 6×6 symmetric band matrix, with bandwidth 5, i.e. with 5 nonzero diagonals. Due to the symmetry, elements a_{ij} and a_{ji} are equal, which means that only the upper or lower nonzero *diagonals* of *M* are required to describe it.

Rewriting the 3 upper nonzero diagonals (including the main diagonal) of *M* as a rectangular 3×6 matrix, one obtains:

$$M_{upper} = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} & a_{46} \\ 0 & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \end{bmatrix}.$$

Notice that a few zeros were introduced in M_{upper}, due to the fact that some diagonals contain more elements than others.

Once M_{upper}. is available, it is easy to store this matrix, by columns, in a single vector, say $\overline{M}_{upper}$:

$$\overline{M}_{upper} = [\,0\ 0\ a_{11}\ 0\ a_{12}\ a_{22}\ a_{13}\ a_{23}\ a_{33}\ a_{24}\ a_{34}\ a_{44}\ a_{35}\ a_{45}\ a_{55}\ a_{46}\ a_{56}\ a_{66}\,].$$

A very similar procedure can be used to store the lower triangular part of *M*. In this case, a 3×6 rectangular matrix M_{lower} and a vector $\overline{M}_{upper}$ are generated, as shown below:

$$M_{lower} = \begin{bmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & 0 \\ a_{31} & a_{42} & a_{53} & a_{64} & 0 & 0 \end{bmatrix},$$

$$\overline{M}_{lower} = [\,a_{11}\ a_{21}\ a_{31}\ a_{22}\ a_{32}\ a_{42}\ a_{33}\ a_{43}\ a_{53}\ a_{44}\ a_{54}\ a_{64}\ a_{55}\ a_{65}\ 0\ a_{66}\ 0\ 0\,].$$

Both functions, `MatrixA` and `MatrixB`, permits the user to choose between storing the lower or the upper triangular part of the matrix. If this information is not supplied by the user, `uplo` is set to `'L'`.

2. Defining the main program.

Once `MatrixA` and `MatrixB` are available, it is now easy to write a program that solves an eigenvalue problem in Cayley mode.

A simple example is shown below. In this example, after calling both functions defined above, matrices *A* and *B* are declared as two `ARbdSymMatrix` o bjects. Then, class

`ARluSymGenEig` is used to create a generalized problem[12], `prob`. Finally, function `EigenValVectors` is called to determine eigenvalues and eigenvectors.

The parameters that are passed to the constructor of `ARluSymGenEig` are:

- The computational mode that should be used to solve the problem ('`C`' is passed, which means that the Cayley mode is to be used);
- The number of eigenvalues sought (`nev`);
- The matrices that define the problem (`A` and `B`); and
- The shift (150.0).

```
#include "arbsmat.h"   // ARbdSymMatrix definition.
#include "arbgsym.h"   // ARluSymGenEig definition.

main()
{

  // Declaring input variables;

  int     n;           // Dimension of the problem.
  int     nev;         // Number of eigenvalues sought.
  int     nsdiagA;     // Lower (and upper) bandwidth of A.
  int     nsdiagB;     // Lower (and upper) bandwidth of B.
  double* valA;        // pointer to an array that stores the nonzero
                       // elements of A.
  double* valB;        // pointer to an array that stores the nonzero
                       // elements of B.

  // Creating matrices A and B.

  n   = 100;
  MatrixA(n, nsdiagA, valA);
  ARbdSymMatrix<double> A(n, nsdiagA, valA);

  MatrixB(n, nsdiagB, valB);
  ARbdSymMatrix<double> B(n, nsdiagB, valB);

  // Defining the eigenvalue problem.

  nev = 4;
  ARluSymGenEig<double> prob('C', nev, A, B, 150.0);

  // Declaring output variables.

  int     nconv;                        // Number of converged eigenvalues.
  double* EigVal = new double[nev];     // Eigenvalues.
  double* EigVec = new double[nev*n]; // Eigenvectors.

  // Finding and storing eigenvalues and eigenvectors.

  nconv = prob.EigenValVectors(EigVec, EigVal);
```

[12] Since `ARluSymGenEig` calls the SuperLU library to solve the linear system $(A-\sigma B)w = v$ when the Cayley mode is being used, this library is supposed to be available.

```
  // ...

} // main.
```

In this example, the four eigenvalues nearest to 150 are determined and stored in `EigVal`. The corresponding eigenvectors are also stored sequentially in an array called `EigVec`.

`EigVec` was dimensioned here to store `n*nev` elements, where `nev` is the number of eigenvectors and `n` is the dimension of each one of them. For complex problems, a complex vector with `nev*n` elements is also sufficient. On the other hand, real nonsymmetric problems require a real vector with `(nev+1)*n` components, since some of the eigenvectors might be complex (see the description of `EigenValVectors` in the appendix).

Solving a complex standard eigenvalue problem.

To illustrate how to declare and solve a problem where the matrix is supplied using the compressed sparse column format, a standard complex eigenvalue problem will now be considered. In this example, the regular mode is used to find the four eigenvalues with largest magnitude of the block tridiagonal matrix A derived from the central-difference discretization of the two-dimensional convection-diffusion operator

$$-\Delta u + \rho \nabla u$$

on the unit square $[0,1]\times[0,1]$, with zero Dirichlet boundary conditions. Here, Δ represents the Laplacian operator, and ∇ the gradient. ρ is a complex parameter. A similar example can be found in the `examples/superlu/complex/lcompreg.cc` file.

1. Generating problem data.

A function, called `MatrixA`, will be used here to generate A in CSC format. This matrix has the form:

$$A = \frac{1}{h^2}\begin{bmatrix} T & -I & 0 & \cdots & 0 \\ -I & T & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & T & -I \\ 0 & \cdots & 0 & -I & T \end{bmatrix}$$

where h is the mesh size, I is the identity matrix and T is a tridiagonal matrix with 4 on the main diagonal, $(-1-\rho h/2)$ on the subdiagonal and $(-1+\rho h/2)$ on the superdiagonal.

`MatrixA` has only one input parameter:

- `nx`, the mesh size;

and five output parameters,

- `n`, the matrix dimension;
- `nnz`, the number of nonzero elements in `A`;
- `A`, a pointer to a vector that contains all nonzero matrix elements;
- `irow`, a pointer to a vector that contains the row indices of the nonzero elements stored in `A`; and
- `pcol`, a pointer to a vector that contain pointers to the first element in each column stored in `A` and `irow`.

These output parameters will be used later to store matrix *A* as an `ARluNonSymMatrix` object.

As in the first example of this chapter, a function template is used to define `MatrixA`:

```
template<class FLOAT, class INT>
void MatrixA(INT nx, INT& n, INT& nnz, complex<FLOAT>* &A,
             INT* &irow, INT* &pcol)
{

// Declaring internal variables.

  INT              i, j, k, id;
  complex<FLOAT> h, h2, dd, dl, du, f;

  // Defining constants.

  const complex<FLOAT> half(0.5, 0.0);
  const complex<FLOAT> one(1.0, 0.0);
  const complex<FLOAT> four(4.0, 0.0);
  const complex<FLOAT> rho(1.0e2, 0.0);

  h   = one/complex<FLOAT>(nx+1, 0);  // mesh size.
  h2  = h*h;
  f   = -one/h2;
  dd  = four/h2;
  dl  = f - half*rho/h;
  du  = f + half*rho/h;

  // Defining the number of columns and nonzero elements in A.

  n   = nx*nx;
  nnz = (5*nx-4)*nx;

  // Creating output vectors.

  A    = new complex<FLOAT>[nnz];
  irow = new INT[nnz];
  pcol = new INT[nx*nx+1];
  // Defining matrix A.

  pcol[0] = 0;
  j       = 0;
  id      = 0;

  for (k=0; k!=nx; k++) {
    for (i=0; i!=nx; i++) {

      if (k) {
        irow[j] = id-nx;
        A[j++]  = f;        // A(i-nx,i) = f.
      }

      if (i) {
        irow[j] = id-1;
        A[j++]  = du;       // A(i-1,i) = du.
      }

      irow[j]   = id;
      A[j++]    = dd;       // A(i,i) = dd.

      if (i!=(nx-1)) {
        irow[j] = id+1;
        A[j++]  = dl;       // A(i+1,i) = dl.
      }

      if (k!=(nx-1)) {
        irow[j] = id+nx;
        A[j++]  = f;        // A(i+nx,i) = f.
      }

      pcol[++id]= j;
    }
  }

} // MatrixA.
```

2. Defining the main program.

Now that the matrix data is available, it is time to write the main program. To create a complex standard eigenvalue problem, two ARPACK++ classes will be required. One, `ARluNonSymMatrix`, to define `A` as the matrix represented by {`n`, `nnz`, `valA`, `irow`, `pcol`}, and the other, `ARluCompStdEig`, to declare `prob` as the problem to be solved and to set some parameters.

As shown below, only two parameters are passed to the constructor of `ARluCompStdEig` in this case. The first is the number of desired eigenvalues. The second is the matrix. No other information is required, since the default values supplied by ARPACK++ for the other parameters are adequate.

```
#include "arlnsmat.h" // ARluNonSymMatrix definition.
#include "arlscomp.h" // ARluCompStdEig definition.
```

```
main()
{

  // Declaring problem data.

  int                nx;
  int                n;     // Dimension of the problem.
  int                nnz;   // Number of nonzero elements in A.
  int*               irow;  // pointer to an array that stores the row
                            // indices of the nonzeros in A.
  int*               pcol;  // pointer to an array of pointers to the
                            // beginning of each column of A in valA.
  complex<double>*   valA;  // pointer to an array that stores the
                            // nonzero elements of A.

  // Creating a complex matrix.

  nx = 10;
  MatrixA(nx, n, nnz, valA, irow, pcol);
  ARluNonSymMatrix<complex<double> > A(n, nnz, valA, irow, pcol);

  // Defining the eigenvalue problem.

  ARluCompStdEig<double> prob(4, A);

  // Declaring output variables.

  vector<double>* EigVal;   // Eigenvalues.
  vector<double>* EigVec;   // Eigenvectors.

  // Finding eigenvalues and eigenvectors.

  EigVec = prob.StlEigenvectors();
  EigVal = prob.StlEigenvalues();

  // ...

} // main.
```

In this example, the four eigenvalues and the corresponding eigenvectors with largest magnitude of `A` were found by using function `StlEigenvectors`. The eigenvectors were stored sequentially in an STL vector called `EigVec`, which was internally dimensioned by ARPACK++ to store `4*n` elements. `StlEigenvalues` was used to store the eigenvalues in `EigVal`.

As it will become clear in the *Working with user-defined matrix-vector products* section below, it is not necessary for the user to supply arrays such as `EigVec` and `EigVal` when solving eigenvalue problems. ARPACK++ can handle eigenvalues and eigenvectors using its own data structure. In this case, `FindEigenvectors` should replace `StlEigenvectors`, and one of the several output functions provided by the software (`Eigenvalue` and `RawEigenvector` are just two examples) used to recover the solution.

Solving truncated SVD problems.

In the last example of this section, ARPACK++ will be used to obtain the some of the singular values of a real nonsymmetric matrix. As described in chapter four, the truncated singular value decomposition of a generic real rectangular matrix A can be obtained by finding the eigenvalues and eigenvectors of the symmetric $n \times n$ matrix A^TA [13]. In this case, the eigenvalues of this matrix are precisely the singular values of A squared, while the eigenvectors are the right singular vectors of A.

1. Generating problem data.

A function template, `RectangularMatrix`, is used below to generate a very simple $2n \times n$ matrix in the form

$$A = \begin{bmatrix} T \\ T \end{bmatrix},$$

where T is a tridiagonal matrix with 4 on the main diagonal, 1 on the subdiagonal and 2 on the superdiagonal.

The function takes one input parameter:

- `n`, the number of columns of A,

and return five parameters:

- m, the number of rows of A;
- `nnz`, the number of nonzero elements in A;
- `A`, a pointer to a vector that contains all nonzero matrix elements;
- `irow`, a pointer to a vector that contains the row indices of the nonzero elements stored in `A`; and
- `pcol`, a pointer to a vector that contain pointers to the first element in each column stored in `A` and `irow`.

```
template<class FLOAT, class INT>
void RectangularMatrix(INT n, INT& m, INT& nnz, FLOAT* &A,
                       INT* &irow, INT* &pcol)
{

// Declaring internal variables.

  INT   i, j;
  FLOAT dd, dl, du;
```

[13] Supposing that m is greater or equal to n. If $m < n$, AA^T must be formed instead of A^TA. For a complete description of all schemes provided by ARPACK++ to find singular values and vectors, the user should refer to chapter four.

```
    // Defining constants.

    dl = 1.0;
    dd = 4.0;
    du = 2.0;

    // Defining the number of rows and nonzero elements in A.

    nnz =  n*6-2;
    m   =  n*2;

    // Creating output vectors.

    A    = new FLOAT[nnz];
    irow = new INT[nnz];
    pcol = new INT[n+1];

    // Defining A.

    pcol[0] = 0;
    j = 0;

    for (i=0; i!=n; i++) {

      if (i != 0) {
        irow[j] = i-1;
        A[j++]  = du;
      }

      irow[j] = i;
      A[j++]  = dd;

      irow[j] = i+1;
      A[j++]  = dl;

      irow[j] = i+n-1;
      A[j++]  = dl;

      irow[j] = i+n;
      A[j++]  = dd;

      if (i != (n-1)) {
        irow[j] = i+n+1;
        A[j++]  = du;
      }

      pcol[i+1] = j;

    }

  } // Rectangular matrix.
```

2. Defining the main program.

The main program listed below shows how to find some of the largest and smallest singular values of *A*, and how the two-norm condition number of the matrix can be calculated.

```
#include "arssym.h"    // ARSymStdEig class definition.
#include "arlnsmat.h"  // ARluNonSymMatrix class definition.
#include <math.h>      // sqrt function declaration.

main()
{

  // Declaring variables;

  int     m;          // Number of rows in A.
  int     n;          // Number of columns in A.
  int     nnz;        // Number of nonzero elements in A.
  int     nconv;      // Number of “converged eigenvalues”.
  int*    irow;       // pointer to an array that stores the row
                      // indices of the nonzeros in A.
  int*    pcol;       // pointer to an array of pointers to the
                      // beginning of each column of A in valA.
  double* valA;       // pointer to an array that stores the
                      // nonzero elements of A.
  double  cond;       // Condition number of A.
  double  svalue[6]   // Singular values.

  // Creating a rectangular matrix with m = 200 and n = 100.

  n = 100;
  RectangularMatrix(n, m, nnz, valA, irow, pcol);

  // Using ARluNonSymMatrix to store matrix information
  // and to perform the product A'Ax.

  ARluNonSymMatrix<double> A(m, n, nnz, valA, irow, pcol);

  // Defining the eigenvalue problem.

  ARSymStdEig<double, ARluNonSymMatrix<double> >
    prob(n, 6, &A, &ARluNonSymMatrix<double>::MultMtMv, "BE");

  // Finding eigenvalues.

  nconv = prob.Eigenvalues(svalue);

  // Calculating singular values and the condition number.

  for (int i=0; i<nconv; i++) svalue[i] = sqrt(svalue[i]);
  cond = svalue[5]/svalue[0];

  // ...

} // main.
```

In this program, the output parameters generated by function `RectangularMatrix` were used to store matrix A as an object of class `ARluNonSymMatrix`[14]. This class was chosen because it contains a function, called `MultMtMv`, that performs the matrix-vector product $w \leftarrow A^T Av$, required to solve the eigenvalue problem.

After storing matrix data, class ARSymStdEig was used to declare a variable, prob, that represents the real symmetric eigenvalue problem defined by A^TA. Five parameters were passed to the constructor of this class:

- The dimension of the system (n);
- The number of eigenvalues sought (6);
- The matrix (A);
- The function that performs the matrix-vector product $w \leftarrow A^T Av$ (MultMtMv);
- The desired part of the spectrum ("BE" is passed here, which means that eigenvalues from both ends of the spectrum are sought).

The eigenvalues of A^TA were determined by function Eigenvalues and stored in a vector called svalue. After computing the square roots of the elements of svalue, the largest and the smallest singular values — svalue[0] and svalue[6], respectively — were used to calculate the condition number of A.

Working with user-defined matrix-vector products.

This section contains a very simple nonsymmetric standard eigenvalue that illustrates how to define a class that includes a matrix-vector product as required by ARPACK++ and also how this class can be used to obtain eigenvalues and eigenvectors.

Creating a matrix class.

The objective of this simple example is to obtain the eigenvalues and eigenvectors of the matrix A derived from the standard central difference discretization of the one-dimensional convection-diffusion operator $-u'' + \rho u'$ on the interval [0,1], with zero Dirichlet boundary conditions. This matrix is nonsymmetric and has a tridiagonal form, with $2/h^2$ as the main diagonal elements, $-1-\rho/2h$ in the subdiagonal and $-1+\rho/2h$ on the superdiagonal, where h is the mesh size.

Before defining an eigenvalue problem using ARPACK++, it is necessary to build at least one class that includes the required matrix-vector product as a member function. This class could be called NonSymMatrix, for example, and the name of the function could be MultMv.

It is better to declare NonSymMatrix as a class template, in order to permit the eigenvalue problem to be solved in single or double precision. So, hereafter, parameter

[14] Class ARumNonSymMatrix can also be used. Or even ARbdNonSymMatrix, if the matrix is stored in band format.

T will designate one of the c++ predefined types `float` or `double`. `NonSymMatrix` can contain variables and functions other than `MultMv`. There only requirements made by ARPACK++ are that `MultMv` must have two pointers to vectors of type `T` as parameters and the input vector must precede the output vector. The class definition is shown below.

```
template<class T>
class NonSymMatrix {
/*
  This simple class exemplifies how to create a matrix class that
  can be used by ARPACK++. Basically, NonSymMatrix is required to
  have a member function that calculates the matrix-vector product
  NonSymMatrix*v, where v is a vector with elements of type T.
*/

 private:

  int m, n;

 public:

  int ncols() { return n; }
  // Function that returns the dimension of the matrix.

  void MultMv(T* v, T* w)
  /*
    Function that performs the product w <- A*v for the matrix A
    derived from the standard central difference discretization of
    the 1-dimensional convection diffusion operator u" + rho*u' on
    the interval [0, 1], with zero Dirichlet boundary conditions.
    A is scaled by h^2 in this example.
  */
  {

    int  j;
    T    dd, dl, du, s, h;

    h  = 1.0/T(ncols()+1);
    s  = 0.5*rho*h;
    dd = 2.0;
    dl = -1.0 - s;
    du = -1.0 + s;

    w[0] = dd*v[0] + du*v[1];
    for (j=1; j<ncols()-1; j++) {
      w[j] = dl*v[j-1] + dd*v[j] + du*v[j+1];
    }
    w[ncols()-1] = dl*v[ncols()-2] + dd*v[ncols()-1];

    return;

  } // MultMv

  NonSymMatrix(int nval) { n = nval; }
  // Constructor.

}; // NonSymMatrix.
```

Solving the eigenvalue problem.

Once defined the matrix-vector product, it is necessary to create a matrix that belongs to class `NonSymMatrix`, and also an object of class `ARNonSymStdEig`. After that, the desired number of eigenvalues can be obtained by calling function `FindEigenvectors`.

Because `ARNonSymStdEig` was declared as a template by ARPACK++, some parameters must be used to create a specific class when the program is compiled. In this example, those parameters are set to `double`, the type of the elements of matrix A, and to `NonSymMatrix<double>`, the name of the class that handles the matrix-vector product.

Besides that, the constructor of class `ARNonSymStdEig` also accepts some parameters, such as the dimension of the eigenvalue system (`A.ncols`), the number of desired eigenvalues (4), an object of class `NonSymMatrix<double>` (`A`), the address of the function that evaluates the matrix-vector product (`&NonSymMatrix<double>::MultMv`) and the portion of the spectrum that is sought (”`SM`”, which means the eigenvalues with smallest magnitude). Other options and parameters (not used here) are described in the appendix.

```
#include "arsnsym.h"

main()
{

  int nconv;

  // Creating a double precision 100x100 matrix.

  NonSymMatrix<double> A(100);

  // Creating an eigenvalue problem and defining what we need:
  // the four eigenvectors of A with smallest magnitude.

  ARNonSymStdEig<double, NonSymMatrix<double> >
    dprob(A.ncols(), 4, &A, &NonSymMatrix<double>::MultMv, "SM");

  /*
    It is possible to pass other parameters directly to the
    constructor of class ARNonSymStdEig in order to define a
    problem. The list of parameters includes, among other values,
    the maximum number of iterations allowed and the relative
    accuracy used to define the stopping criterion. Alternatively,
    it is also possible to use function DefineParameters to set
    ARPACK++ variables after declaring dprob as an object of
    class ARNonSymStdEig using the default constructor.
  */

  // Finding eigenvectors.

  nconv = dprob.FindEigenvectors();
```

```
   // Printing the solution.

  Solution(A, dprob);

} // main.
```

Printing some information about eigenvalues and eigenvectors.

The function `Solution` was included in this example to illustrate how to extract information about eigenvalues and eigenvectors from class `ARNonSymStdEig`. Only a few suggestions are shown here. A complete list of ARPACK++ functions can be found in the appendix: *ARPACK++ reference guide*.

```
#include “blas1c.h”   // ARPACK++ version of blas1 routines.
#include “lapackc.h”  // ARPACK++ version of lapack routines.

template<class FLOAT, class EIGPROB>
void Solution(SymMatrix<FLOAT> &A, EIGPROB &Prob)
/*
  This function prints eigenvalues and eigenvectors on
  standard "cout" stream and exemplifies how to retrieve
  information from ARPACK++ classes.
*/

{

  int   i, n, nconv, mode;
  FLOAT *Ax;
  FLOAT *ResNorm;

  /*
     ARPACK++ includes some functions that provide information
     about the problem. For example, GetN furnishes the dimension
     of the problem and ConvergedEigenvalues the number of
     eigenvalues that attained the required accuracy. GetMode
     indicates if the problem was solved in regular,
     shift-and-invert or other mode.
  */

  n     = Prob.GetN();
  nconv = Prob.ConvergedEigenvalues();
  mode  = Prob.GetMode();

  cout << "Testing ARPACK++ class ARNonSymStdEig" << endl;
  cout << "Real nonsymmetric eigenvalue problem: A*x-lambda*x"<< endl;
  switch (mode) {
  case 1:
    cout << "Regular mode" << endl << endl;
    break;
  case 3:
    cout << "Shift and invert mode" << endl << endl;
  }

  cout << "Dimension of the system  : " << n             << endl;
  cout << "'requested' eigenvalues  : " << Prob.GetNev() << endl;
  cout << "'converged' eigenvalues  : " << nconv         << endl;
cout << "Arnoldi vectors generated: " << Prob.GetNcv() << endl;
cout << endl;

/*
  EigenvaluesFound is a boolean function that indicates
  if the eigenvalues were found or not. Eigenvalue can be
  used to obtain one of the "converged" eigenvalues. There
  are other functions that return eigenvectors elements,
  Schur vectors elements, residual vector elements, etc.
*/

if (Prob.EigenvaluesFound()) {

  // Printing eigenvalues.

  cout << "Eigenvalues:" << endl;
  for (i=0; i<nconv; i++) {
    cout << "  lambda[" << (i+1) << "]: " << Prob.EigenvalueReal(i);
    if (Prob.EigenvalueImag(i)>=0.0) {
      cout << " + " << Prob.EigenvalueImag(i) << " I" << endl;
    }
    else {
      cout << " - " << fabs(Prob.EigenvalueImag(i)) << " I" << endl;
    }
  }
  cout << endl;
}

/*
  EigenvectorsFound indicates if the eigenvectors are
  available. RawEigenvector is one of the functions that
  provide raw access to ARPACK++ output data. Other functions
  of this type include RawEigenvalues, RawEigenvectors,
  RawSchurVector, RawResidualVector, etc.
*/

if (Prob.EigenvectorsFound()) {

  // Printing the residual norm || A*x - lambda*x || for the
  // nconv accurately computed eigenvectors.
  // axpy and nrm2 are blas 1 fortran subroutines. The first
  // calculates y <- y + a*x, and the second determines the
  // two-norm of a vector. lapy2 is the lapack function that
  // computes sqrt(x*x+y*y) carefully.

  Ax      = new FLOAT[n];
  ResNorm = new FLOAT[nconv+1];

  for (i=0; i<nconv; i++) {

    if (Prob.EigenvalueImag(i)==0.0) { // Eigenvalue is real.
      A.MultMv(Prob.RawEigenvector(i), Ax);
      axpy(n,-Prob.EigenvalueReal(i),Prob.RawEigenvector(i),1,Ax,1);
      ResNorm[i] = nrm2(n, Ax, 1)/fabs(Prob.EigenvalueReal(i));

    }
    else {                             // Eigenvalue is complex.
      A.MultMv(Prob.RawEigenvector(i), Ax);
      axpy(n,-Prob.EigenvalueReal(i),Prob.RawEigenvector(i),1,Ax,1);
      axpy(n,Prob.EigenvalueImag(i),Prob.RawEigenvector(i+1),1,Ax,1);
```

```
        ResNorm[i] = nrm2(n, Ax, 1);
        A.MultMv(Prob.RawEigenvector(i+1), Ax);
        axpy(n,-Prob.EigenvalueImag(i),Prob.RawEigenvector(i),1,Ax,1);
        axpy(n,-Prob.EigenvalueReal(i),Prob.RawEigenvector(i+1),1,Ax,1);
        ResNorm[i] = lapy2(ResNorm[i],nrm2(n, Ax, 1))/
                     lapy2(Prob.EigenvalueReal(i),Prob.EigenvalueImag(i));
        ResNorm[i+1] = ResNorm[i];
        i++;
      }

    }

    for (i=0; i<nconv; i++) {
      cout << "||A*x(" << (i+1) << ") - lambda(" << (i+1);
      cout << ")*x(" << (i+1) << ")||: " << ResNorm[i] << endl;
    }
    cout << endl;

    delete[] Ax;
    delete[] ResNorm;

  }

} // Solution
```

Using the reverse communication interface.

ARPACK++ provides a somewhat simple structure for handling eigenvalue problems. However, sometimes it is inconvenient to explicitly define a function that evaluates a matrix-vector product using the format required by the above mentioned classes.

To deal with such cases, ARPACK++ also includes a set of classes and functions that allow the user to perform matrix-vector products on his own. This structure is called the *reverse communication interface* and is derived from the FORTRAN version of the software.

Although this interface gives the user some freedom, it requires a step-by-step execution of ARPACK++. Therefore, to find an Arnoldi basis it is necessary to define a sequence of calls to a function called `TakeStep` combined with matrix-vector products until convergence is attained.

One example that illustrate the use of these classes is given below The matrix used in this example, say A, is real and symmetric. It is not defined by a class, but only by the function `MultMv` that performs the product $y \leftarrow Ax$. A slightly different version of this program can be found in directory `examples/reverse/sym`.

```
#include "arrssym.h"
```

```
template<class T>
void MultMv(int n, T* v, T* w)
/*
  Function that evaluates the matrix-vector product w <- A*v,
  where A is the one dimensional discrete Laplacian on
  the interval [0,1] with zero Dirichlet boundary conditions.
*/
{

  int  j;
  T    h2;

  w[0] =  2.0*v[0] - v[1];
  for (j=1; j<n-1; j++) {
    w[j] = - v[j-1] + 2.0*v[j] - v[j+1];
  }
  w[n-1] = - v[n-2] + 2.0*v[n-1];

  // Scaling vector w by (1/h^2) using blas routine scal.

  h2 = T((n+1)*(n+1));
  scal(n, h2, w, 1L);

} // MultMv


main()
{

  // Declaring matrix A.

  SymMatrixA<double> A(100); // n = 100.

  // Creating a symmetric eigenvalue problem and defining what
  // we need: the four eigenvectors of A with largest magnitude.

  ARrcSymStdEig<double> prob(A.ncols(), 4L);

  // Finding an Arnoldi basis.

  while (!prob.ArnoldiBasisFound()) {

    // Calling ARPACK fortran code. Almost all work needed to
    // find an Arnoldi basis is performed by TakeStep.

    prob.TakeStep();

    if ((prob.GetIdo() == 1)||(prob.GetIdo() == -1)) {

      // Performing the matrix-vector product.
      // GetIdo indicates which product must be performed
      // (in this case, only y <- Ax).
      // GetVector supplies a pointer to the input vector
      // and Put vector a pointer to the output vector.

      A.MultMv(prob.GetVector(), prob.PutVector());

    }
  }
  // Finding eigenvalues and eigenvectors.

  prob.FindEigenvectors();

  // ...

} // main.
```

In the above example, the definition of the eigenvalue problem was made without any mention to the matrix class. Because of that, ARPACK++ was not able to handle the matrix-vector products needed by the Arnoldi process and it was necessary to include a `while` statement in the main program in order to iteratively find an Arnoldi basis. Only after that, `FindEigenvectors` was called to find eigenvalues and eigenvectors.

In this iterative search for an Arnoldi basis, `TakeStep` was used to perform almost all work needed by the algorithm and only the product $y \leftarrow Ax$ was left to the user. When solving a generalized eigenvalue problem, however, at least two different matrix-vector products must be performed, and the `GetIdo` function should be used to determine which product must be taken after each call to `TakeStep`.

Some other useful ARPACK++ functions included in the example are `GetVector`, `PutVector` and `ArnoldiBasisFound`. `GetVector` and `PutVector` are two functions that return pointers to the exact position where, respectively, `x`, the input vector, and `y`, the output vector of the matrix-vector product, are stored. `ArnoldiBasisFound` is used to detect if the desired eigenvalues have attained the desired accuracy.

Finally, it is worth mentioning that, although no output command was included in the above program, functions such as `EigenValVectors`, `RawEigenvectors`, `StlEigenvalues` and `Eigenvalue` are also available when using the *reverse communication interface*.

Building an interface with another library.

More than a c++ version of the ARPACK FORTRAN package, ARPACK++ is intended to be an interface between ARPACK and other mathematical libraries. Virtually all numerical libraries that represent matrices and their operations by means of c++ classes can be linked to ARPACK++. This is the main reason why class templates were used to define eigenvalue problems.

The simplest way to connect ARPACK++ with another library is to pass a matrix generated by this library as a parameter to one of the classes `ARNonSymStdEig`, `ARSymStdEig`, `ARCompStdEig`, `ARNonSymGenEig`, `ARSymGenEig` or `ARCompGenEig`. In this case, the user can also pass the matrix class as template parameter, so the problem can be solved almost immediately, as shown in the *Working with user-defined matrix-vector products* section above.

This alternative is recommended when only a few eigenvalue problems are to be solved. However, if the user intends to solve many eigenvalue problems, it is better to define a new class to interface ARPACK++ with the other library.

The creation of a new class is very simple, since most of its member functions can be inherited from other parent classes. As an example, one of the declarations of the `ARluNonSymStdEig` class is transcribed below. Actually, this is the UMFPACK version of this class, exactly as it is declared in the `arpack++/include/arusnsym.h` file.

```
/*
  MODULE ARUSNSym.h.
  Arpack++ class ARluNonSymStdEig definition (umfpack version).
*/

#ifndef ARUSNSYM_H
#define ARUSNSYM_H

#include "arch.h"       // Machine dependent functions and variable types.
#include "arsnsym.h"    // ARNonSymStdEig class definition.
#include "arunsmat.h"   // ARumNonSymMatrix class definition.

template<class FLOAT>
class ARluNonSymStdEig:
  public virtual ARNonSymStdEig<FLOAT, ARumNonSymMatrix<FLOAT> > {

 public:

 // a) Public functions:

 // a.1) Function that allows changes in problem parameters.

  virtual void ChangeShift(FLOAT sigmaRp);

 // a.2) Constructors and destructor.

  ARluNonSymStdEig() { }
  // Short constructor.

  ARluNonSymStdEig(int nevp, ARumNonSymMatrix<FLOAT>& A,
                   char* whichp = "LM", int ncvp = 0,
                   FLOAT tolp = 0.0, int maxitp = 0,
                   FLOAT* residp = 0, bool ishiftp = true);
  // Long constructor (regular mode).

  ARluNonSymStdEig(int nevp, ARumNonSymMatrix<FLOAT>& A,
                   FLOAT sigma, char* whichp = "LM", int ncvp = 0,
                   FLOAT tolp = 0.0, int maxitp = 0,
                   FLOAT* residp = 0, bool ishiftp = true);
  // Long constructor (shift and invert mode).

  ARluNonSymStdEig(const ARluNonSymStdEig& other) { Copy(other); }
  // Copy constructor.
  virtual ~ARluNonSymStdEig() { }
  // Destructor.

 // b) Operators.

  ARluNonSymStdEig& operator=(const ARluNonSymStdEig& other);
  // Assignment operator.

}; // class ARluNonSymStdEig.

#endif // ARUSNSYM_H
```

`ARluNonSymStdEig` is derived from `ARNonSymStdEig`. All functions and variables of this base class are inherited by the new class. The only function redefined here is `ChangeShift`. Naturally, the class constructors, the destructor and the assignment operator are not inherited as well.

The main reason for function `ChangeShift` to be redefined is to include the command

```
objOP->FactorAsI(sigmaR);
```

This command tells ARPACK++ to factorize matrix $A-\sigma I$ each time a new shift σ is defined. This factorization is necessary since `ARluNonSymStdEig` cannot solve an eigenvalue problem in shift and invert mode without solving several linear systems involving $A-\sigma I$.

In ARPACK++, every time the copy constructor or the assignment operator is called, a function named `copy` is called to make a copy of the class. Fortunately, `ARluNonSymStdEig` does not contain variable declarations, but if the user intends to create a class that contains new variables, a new `Copy` function should also be defined. Doing this way, the user assures that neither the copy constructor nor the assignment operator need to be changed.

The standard constructor and the destructor of `ARluNonSymStdEig` contains no commands. These functions do nothing but calling the constructors and destructors of the base classes. The other three constructors contain exactly the same commands defined in the constructors of the `ARluNonSymStdEig` class. The same happens to the assignment operator. Actually, these functions were redefined just because the language does not allow them to be inherited.

Appendix

A

ARPACK++ reference guide

This appendix contains a detailed description of all the ARPACK++ classes, variables and functions. Problem and template parameters are presented first. After that, each class is described with examples that illustrate how to use all available constructors. Finally, ARPACK++ functions are classified and grouped accordingly their use.

Through this chapter, complex numbers will be presented using g++ notation. Thus, `complex<double>` represents a double precision complex type, i.e. a complex number with double precision real and imaginary parts.

Template parameters.

ARPACK++ is a collection of templates. Because of that, its classes are not unique but depend on some parameters that permit specific classes to be built only at compilation time. These parameters give the user some freedom to define matrix classes that describe the eigenvalue problem and also to use different floating point precision. The four available parameters are described below.

FLOAT

Description

This is the predefined c++ type used to represent real numbers. It must be set to `double` or `float`.

TYPE

Description

This is the type used to represent elements of vectors and matrices. If the problem being solved is complex, it must be set to `complex<double>` or `complex<float>`, otherwise it must be set to `double` or `float`, depending on the value of the `FLOAT` parameter.

Because ARPACK++ has specialized classes that handle complex and real symmetric and nonsymmetric problems, this parameter is seldom used to define an eigenvalue problem. Only the matrix classes and ARPACK++ base classes require the definition of `TYPE`.

FOP

Description

This is the c++ class that handles matrix information in standard eigenvalue problems. It also the class used to define one of the matrices in generalized problems. `FOP` must contain a member function which matches the definition of `MultOPx` given below.

FB

Description

This is the c++ class that contains information about the second matrix in generalized problems (the first matrix is handled by `FOP`). It must contain a member function with exactly the same type of `MultBx` (and also another function that matches the definition of `MultAx`, in certain cases). `MultBx` and `MultAx` are described below.

Types of matrix-vector product functions.

ARPACK++ classes that require matrix-vector product functions impose the user only one restriction: these functions must follow a very stringent pattern. This limitation is related to the format used by the ARPACK FORTRAN code to store vectors. Actually, functions can be created using other types, but there must be an explicit conversion between them and one of the types described below.

TypeOPx

Declaration

```
typedef void (FOP::* TypeOPx)(TYPE[], TYPE[])
```

Description

TypeOPx is a pointer to a function that has two vectors of type TYPE as parameters and returns nothing. This function must be a member of class FOP.

Some ARPACK++ classes require the user to create a class which includes a function that evaluates the matrix-vector product $y \leftarrow OPx$ (see the description of MultOPx below). TypeOPx is used to define the name of this function, while FOP is used to represent the class name. The first parameter is the input vector, x, and the second is the output vector, y.

TypeBx

Declaration

```
typedef void (FB::* TypeBx)(TYPE[], TYPE[])
```

Description

Analogously to TypeOPx, TypeBx represents the name of a function that is member of class FB and evaluates a matrix-vector product in the form $y \leftarrow Bx$ (see MultBx below). The first parameter of the function is the input vector, x, and the second is the output vector, y.

TypeBx is also used to represent another member function of class FB. This function evaluates the matrix-vector product $y \leftarrow Ax$ and is required only by two classes: ARSymGenEig (when using the Cayley constructor) and ARNonSymGenEig (when in complex shift-and-invert mode).

Problem parameters.

Various ARPACK++ class constructors and functions that will be described later in this section include one or more parameters. To avoid redefining these parameters each time a function is mentioned, a complete list is given below. The list include parameters from all ARPACK++ classes, each one followed by its type (displayed on the right) and a brief description.

Some parameters are compulsory, i.e. must be supplied by the user when solving an eigenvalue problem. Other are internally set by ARPACK++, but the user may change them if the default value is not appropriate.

Compulsory parameters.

n

int

Description

Dimension of the eigenvalue problem. `n` > 1.

nev

int

Description

Number of eigenvalues to be computed. 0 < `nev` < `n`-1.

objOP

FOP*

Description

Pointer to an object of class `FOP`[15]. Class `FOP` must have `MultOPx` as a member function. This parameter is required only if the user intends to use his own matrix class.

MultOPx

TypeOPx

Description

Member function of class `FOP` that evaluates the product $y \leftarrow OPx$. The specification of OP depends on the problem type and the computational mode being used. The alternatives are summarized in the following table:

[15] See the description of `FOP` in the *Template parameters* section above.

Problem type	**mode**	$y \leftarrow OPx$
Standard	regular	$y \leftarrow Ax$
	shift and invert	$y \leftarrow (A - \sigma I)^{-1} x$
Generalized	regular	$y \leftarrow B^{-1} Ax$
	all other	$y \leftarrow (A - \sigma B)^{-1} x$

Warning: When solving real nonsymmetric problems in complex shift-and-invert mode, *OP* elements are complex, but *y* must be a real vector. In this case, *y* should be set to the real or the imaginary part of the complex vector $z = OPx$ **.**

objB

FB*

Description

Pointer to an object of class `FB`. The class of this object must have `MultBx` as a member function. `objB` is required only if the user wants to supply his own matrix classes when solving a generalized eigenvalue problem.

MultBx

TypeBx

Description

Member function of class `FB` that evaluates a product in the form $y \leftarrow Bx$ or $y \leftarrow Ax$, when defining a generalized eigenvalue problem $Ax = Bx\lambda$.

ARPACK++ assumes that this class will return $y \leftarrow Ax$ only if the user is solving a real symmetric generalized problem in buckling mode. In all other cases, `MultBx` is supposed to evaluate the product $y \leftarrow Bx$.

objA

FB*

Description

Pointer to an object of class `FB`. The class of this object must have `MultAx` as a member function. `objA` is used with some particular generalized real problems only (see `MultAx`).

MultAx

TypeBx

Description

Member function of class `FB` that evaluates the product $y \leftarrow Ax$. This parameter is required only by two classes: `ARSymGenEig`, when using the Cayley constructor, and `ARNonSymGenEig`, when using the complex shift-and-invert mode.

sigma (or sigmaR)

TYPE

Description

Shift. This parameter is required if a spectral transformation is employed. It represents the real part of a complex shift if the problem is real and nonsymmetric.

sigmaI

FLOAT

Description

Imaginary part of the shift. This parameter must be supplied when solving nonsymmetric problems in complex shift and invert mode.

InvertMode

char

Description

Spectral transformation used to find eigenvalues of symmetric generalized problems. If the shift and invert mode is being used, this parameter must be set to "`S`". Buckling and Cayley modes are represented by "`B`" and "`C`", respectively.

part

char

Description

This parameter is required only if the problem to be solved is a real nonsymmetric generalized one and a complex shift is used to characterize the desired portion of the spectrum. In this case, the user needs to supply a matrix-vector routine in the form $y \leftarrow OP.x$, where OP is one of the $real\{(A-\sigma B)^{-1}\}$ or $imag\{(A-\sigma B)^{-1}\}$ and the variable `part` must be set to one of "`R`" or "`I`" in order to reflect the choice made.

Optional parameters.

ncv

int

Description

Number of Arnoldi vectors generated at each iteration. ncv must be set to a value between nev + 1 and n - 1.

ncv is strongly related to the computational time and also to the storage space required by ARPACK++. The computational work needed to find eigenvalues is proportional to $n.ncv^2$ flops, while memory consumption is n.O(ncv) + O(ncv^2). Unfortunately, ncv it is very problem dependent and there is no *a-priori* analysis to guide the selection of this parameter. Generally, if matrix-vector products are cheap, a smaller value of ncv may lead to a decrease in the overall computational time, in spite of the larger amount of products required.

Default value.

min { 2 nev + 1, n - 1 }

maxit

int

Description

Maximum number of Arnoldi update iterations allowed. If the user supplies a positive maxit, this value is maintained, otherwise the default value is employed.

Default value.

100 nev.

which

char*

Description

This parameter specifies which of the Ritz values of OP to compute and depends on the class being used. The options available are depicted in the table below:

option	**desired part of spectrum**
LA	eigenvalues with largest algebraic value
SA	eigenvalues with smallest algebraic value
LM	eigenvalues with largest magnitude
SM	eigenvalues with smallest magnitude
LR	eigenvalues with largest real part
SR	eigenvalues with smallest real part
LI	eigenvalues with largest imaginary part
SI	eigenvalues with smallest imaginary part
BE	eigenvalues from both ends of spectrum (if `nev` is odd, one more eigenvalue is computed from the high end than from the low end).

For symmetric problems, `which` must set to be one of `LA`, `SA`, `LM`, `SM` or `BE`. For real nonsymmetric and complex problems, the alternatives are `LM`, `SM`, `LR`, `SR`, `LI` and `SI`.

Default value.

`LM`.

tol

FLOAT

Description

Stopping criterion (relative accuracy of Ritz values). The user should expect a computed eigenvalue, λ, to satisfy the relation $|\lambda - \lambda^*| \le \texttt{tol}|\lambda|$, where λ^* is the eigenvalue of A closest to λ.

The Arnoldi process is somewhat sensitive to this parameter, so it must be set with some care. Though large values of `tol` can reduce the number of iterations required to attain convergence, some eigenvalues can be missed if they are multiple or tightly clustered. On the other hand, very small values can prevent the convergence of the method.

Default value.

The machine precision is used if `tol` is not supplied or set to zero.

resid

TYPE*

Description

Initial vector. Although generally the default starting vector is a good choice, `resid` can be supplied, for example, when a sequence of related problems is being solved. In such cases, ARPACK can converge faster if a starting vector based on previous eigenvalue calculations is used.

Default value.

When this parameter is not provided by the user, a random vector is adopted.

AutoShift

bool

Description

This parameter indicates if exact shifts for the implicit restarting of the Arnoldi method are being generated internally by ARPACK++ or shifts are being supplied by the user.

Default value.

`true` (exact shifts are being used).

Eigenvalue problem classes.

There are twenty two predefined template classes in ARPACK++. These classes are intended to cover all types of problem handled by ARPACK FORTRAN code and also to provide an easy way of creating eigenvalue problems. The first eighteen classes described below may be used to define objects directly. The main purpose of the last four is to serve as a basis for the former classes, but they also may be used to create new user defined classes.

The filename shown under each class name (on the right) corresponds to the header file that contains the class definition.

Classes that require matrices.

ARluSymStdEig

arlssym.h (SuperLU version)
arussym.h (UMFPACK version)
ardssym.h (LAPACK dense version)
arbssym.h (LAPACK band version)

Declaration

```
template <class FLOAT> class ARluSymStdEig
```

Description

This class defines a real symmetric standard eigenvalue problem using `ARluSymMatrix`, `ARumSymMatrix`, `ARdsSymMatrix` or `ARbdSymMatrix` as the class that stores matrix data.

Warning: `ARluSymStdEig` does a sparse LU factorization of matrix $(A - \sigma I)$ when shift and invert mode is used, so the user must be aware of the memory requirements associated to this spectral transformation[16].

Such factorization is performed by the SuperLU package if `ARluSymMatrix` is the matrix class being used, while `ARluSymMatrix` calls UMFPACK routines and `ARdsSymMatrix` and `ARbdSymMatrix` use LAPACK matrix factorizations. All these libraries can be obtained as described in chapter one.

Parent class (SuperLU version)

```
public virtual ARSymStdEig<FLOAT, ARluSymMatrix<FLOAT> >
```

Parent class (UMFPACK version)

```
public virtual ARSymStdEig<FLOAT, ARumSymMatrix<FLOAT> >
```

Parent class (LAPACK band version)

```
public virtual ARSymStdEig<FLOAT, ARbdSymMatrix<FLOAT> >
```

Parent class (LAPACK dense version)

```
public virtual ARSymStdEig<FLOAT, ARdsSymMatrix<FLOAT> >
```

Default constructor

```
ARluSymStdEig()
```

Regular mode constructor (SuperLU version)

```
ARluSymStdEig( int nev, ARluSymMatrix<FLOAT>& A, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

[16] Errors such as a memory overflow can be caught by the user. See the *Handling Errors* section below.

Regular mode constructor (UMFPACK version)

```
ARluSymStdEig( int nev, ARumSymMatrix<FLOAT>& A, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluSymStdEig( int nev, ARbdSymMatrix<FLOAT>& A, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluSymStdEig( int nev, ARdsSymMatrix<FLOAT>& A, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (SuperLU version)

```
ARluSymStdEig( int nev, ARluSymMatrix<FLOAT>& A, FLOAT sigma,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (UMFPACK version)

```
ARluSymStdEig( int nev, ARumSymMatrix<FLOAT>& A, FLOAT sigma,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK band version)

```
ARluSymStdEig( int nev, ARbdSymMatrix<FLOAT>& A, FLOAT sigma,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK dense version)

```
ARluSymStdEig( int nev, ARdsSymMatrix<FLOAT>& A, FLOAT sigma,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Examples

This class requires the user to declare a matrix using one of the `ARluSymMatrix`, `ARumSymMatrix`, `ARdsSymMatrix` or `ARbdSymMatrix` classes[17], as in the following examples:

```
ARluSymMatrix<double> A(n, nnz, nzval, irow, pcol);

ARumSymMatrix<double> A(n, nnz, nzval, irow, pcol);

ARdsSymMatrix<double> A(n, nzval);

ARbdSymMatrix<double> A(n, nsdiag, nzval);
```

Once *A* has been defined, one of the constructors mentioned above should be used to declare the problem. As the default constructor would be harder to use than the others, because it does not permit the user to simultaneously define an object and pass all the

[17] For a full description of these classes, see the *Available matrix classes* section below.

required information, only the regular and the shift and invert mode constructors were included here.

1. Using the regular mode constructor

```
ARluSymStdEig<double> prob(4, A);
```

2. Using the real shift and invert mode constructor

```
ARluSymStdEig<double> prob(4, A, 13.2);
```

ARluNonSymStdEig

arlsnsym.h (SuperLU version)
arusnsym.h (UMFPACK version)
ardsnsym.h (LAPACK dense version)
arbsnsym.h (LAPACK band version)

Declaration

```
template <class FLOAT> class ARluNonSymStdEig
```

Description

This class defines a real nonsymmetric standard eigenvalue problem using one of the `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes to store matrix data.

Warning: A sparse LU factorization of matrix $(A-\sigma I)$ is performed when the shift and invert mode is used, so the user must be aware of the memory requirements associated to this spectral transformation (see how to catch a memory overflow in the *Handling errors* section below).

Such factorization is done by the SuperLU package if `ARluNonSymMatrix` is the matrix class being used, while `ARluNonSymMatrix` calls UMFPACK routines and `ARdsNonSymMatrix` and `ARbdNonSymMatrix` use LAPACK matrix factorizations. All these libraries can be obtained as described in chapter one.

Parent class (SuperLU version)

```
public virtual ARNonSymStdEig<FLOAT, ARluNonSymMatrix<FLOAT> >
```

Parent class (UMFPACK version)

```
public virtual ARNonSymStdEig<FLOAT, ARumNonSymMatrix<FLOAT> >
```

Parent class (LAPACK dense version)

```
public virtual ARNonSymStdEig<FLOAT, ARdsNonSymMatrix<FLOAT> >
```

Parent class (LAPACK band version)

```
public virtual ARNonSymStdEig<FLOAT, ARbdNonSymMatrix<FLOAT> >
```

Default constructor

```
ARluNonSymStdEig()
```

Regular mode constructor (SuperLU version)

```
ARluNonSymStdEig( int nev, ARluNonSymMatrix<FLOAT>& A,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (UMFPACK version)

```
ARluNonSymStdEig( int nev, ARumNonSymMatrix<FLOAT>& A,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluNonSymStdEig( int nev, ARdsNonSymMatrix<FLOAT>& A,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluNonSymStdEig( int nev, ARbdNonSymMatrix<FLOAT>& A,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (SuperLU version)

```
ARluNonSymStdEig( int nev, ARluNonSymMatrix<FLOAT>& A, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (UMFPACK version)

```
ARluNonSymStdEig( int nev, ARumNonSymMatrix<FLOAT>& A, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (LAPACK dense version)

```
ARluNonSymStdEig( int nev, ARdsNonSymMatrix<FLOAT>& A, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (LAPACK band version)

```
ARluNonSymStdEig( int nev, ARbdNonSymMatrix<FLOAT>& A, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Examples

This class requires the user to declare a matrix using the `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes, as in the following examples:

```
ARluNonSymMatrix<double> A(n, nnz, nzval, irow, pcol);
```

```
ARumNonSymMatrix<double> A(n, nnz, nzval, irow, pcol);
```

```
ARdsNonSymMatrix<double> A(n, nzval);
ARbdNonSymMatrix<double> A(n, ndiagL, ndiagU, nzval);
```

Once *A* has been defined, one of the constructors mentioned above should be used to declare the problem. As in the symmetric class, the default constructor is harder to use than the others, since it does not permit the user to pass all required information at once. Thus, only the regular and shift and invert constructors are shown below.

1. Using the regular mode constructor

```
ARluNonSymStdEig<double> prob(4, A);
```

2. Using the real shift and invert mode constructor

```
ARluNonSymStdEig<double> prob(4, A, 13.2);
```

ARluCompStdEig

arlscomp.h (SuperLU version)
aruscomp.h (UMFPACK version)
ardscomp.h (LAPACK dense version)
arbscomp.h (LAPACK band version)

Declaration

```
template <class FLOAT> class ARluCompStdEig
```

Description

This class defines a complex standard eigenvalue problem using `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` as the class that stores matrix data.

Warning: `ARluCompStdEig` calls one of the SuperLU, UMFPACK or LAPACK packages to perform a sparse LU factorization of matrix $(A-\sigma I)$ when the eigenvalue problem is being solved in shift and invert mode, so the user must be aware of the memory requirements associated to this spectral transformation.

Parent class (SuperLU version)

```
public virtual
  ARCompStdEig<FLOAT, ARluNonSymMatrix<complex<FLOAT> > >
```

Parent class (UMFPACK version)

```
public virtual
  ARCompStdEig<FLOAT, ARumNonSymMatrix<complex<FLOAT> > >
```

Parent class (LAPACK dense version)

```
public virtual
  ARCompStdEig<FLOAT, ARdsNonSymMatrix<complex<FLOAT> > >
```

Parent class (LAPACK band version)

```
public virtual
  ARCompStdEig<FLOAT, ARbdNonSymMatrix<complex<FLOAT> > >
```

Default constructor

```
ARluCompStdEig()
```

Regular mode constructor (SuperLU version)

```
ARluCompStdEig( int nev, ARluNonSymMatrix<complex<FLOAT> >& A,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Regular mode constructor (UMFPACK version)

```
ARluCompStdEig( int nev, ARumNonSymMatrix<complex<FLOAT> >& A,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluCompStdEig( int nev, ARdsNonSymMatrix<complex<FLOAT> >& A,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluCompStdEig( int nev, ARbdNonSymMatrix<complex<FLOAT> >& A,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Shift and invert mode constructor (SuperLU version)

```
ARluCompStdEig( int nev, ARluNonSymMatrix<complex<FLOAT> >& A,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (UMFPACK version)

```
ARluCompStdEig( int nev, ARumNonSymMatrix<complex<FLOAT> >& A,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK dense version)

```
ARluCompStdEig( int nev, ARdsNonSymMatrix<complex<FLOAT> >& A,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK band version)

```
ARluCompStdEig( int nev, ARbdNonSymMatrix<complex<FLOAT> >& A,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Examples

To use this class, one must declare a matrix using `ARumNonSymMatrix`, `ARluNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix`[18], as in the following examples:

```
ARluNonSymMatrix<complex<double> > A(n, nnz, nzval, irow, pcol);

ARumNonSymMatrix<complex<double> > A(n, nnz, nzval, irow, pcol);

ARdsNonSymMatrix<complex<double> > A(n, nzval);

ARbdNonSymMatrix<complex<double> > A(n, ndiagL, ndiagU, nzval);
```

Once created the matrix, one of the above constructors should be used to create the problem. As the default constructor does not permit the user to declare all the parameters required by ARPACK++ while defining an object of this class, only the last two are shown below.

1. Using the regular mode constructor

```
ARluCompStdEig<double> prob(4, A);
```

2. Using the real shift and invert mode constructor

```
ARluCompStdEig<double> prob(4, A, complex<double>(0.8, 0.4));
```

ARluSymGenEig

arlgsym.h (SuperLU version)
arugsym.h (UMFPACK version)
ardgsym.h (LAPACK dense version)
arbgsym.h (LAPACK band version)

Declaration

```
template <class FLOAT > class ARluSymGenEig
```

Description

This class defines a real symmetric generalized eigenvalue problem with matrices stored using one of the `ARluSymMatrix`, `ARumSymMatrix`, `ARdsSymMatrix` or `ARbdSymMatrix` classes, depending on which package is being used to perform the factorization of matrix B (if regular mode is chosen) or $(A - \sigma B)$ (when in shift and invert mode). `ARluSymMatrix` class requires the SuperLU library, while UMFPACK

[18] The functionality of these classes is described in the *Available matrix classes* section below.

is used by `ARumSymMatrix` and LAPACK is called by `ARbdSymMatrix` and `ARdsSymMatrix`.

Because some fill-in can be generated by a sparse LU factorization, the user must be aware of the memory requirements associated to each spectral transformation.

Parent class (SuperLU version)

```
public virtual ARSymGenEig<FLOAT, ARluSymPencil<FLOAT>,
                           ARluSymPencil<FLOAT> >
```

Parent class (UMFPACK version)

```
public virtual ARSymGenEig<FLOAT, ARumSymPencil<FLOAT>,
                           ARumSymPencil<FLOAT> >
```

Parent class (LAPACK dense version)

```
public virtual ARSymGenEig<FLOAT, ARdsSymPencil<FLOAT>,
                           ARdsSymPencil<FLOAT> >
```

Parent class (LAPACK band version)

```
public virtual ARSymGenEig<FLOAT, ARbdSymPencil<FLOAT>,
                           ARbdSymPencil<FLOAT> >
```

Default constructor

```
ARluSymGenEig()
```

Regular mode constructor (SuperLU version)

```
ARluSymGenEig( int nev, ARluSymMatrix<FLOAT>& A, ARluSymMatrix<FLOAT>& B,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (UMFPACK version)

```
ARluSymGenEig( int nev, ARumSymMatrix<FLOAT>& A, ARumSymMatrix<FLOAT>& B,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluSymGenEig( int nev, ARdsSymMatrix<FLOAT>& A, ARdsSymMatrix<FLOAT>& B,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluSymGenEig( int nev, ARbdSymMatrix<FLOAT>& A, ARbdSymMatrix<FLOAT>& B,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert, buckling and Cayley modes constructor (SuperLU version)[19]

```
ARluSymGenEig( char InvertMode, int nev, ARluSymMatrix<FLOAT>& A,
               ARluSymMatrix<FLOAT>& B, FLOAT sigma, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert, buckling and Cayley modes constructor (UMFPACK version)[20]

```
ARluSymGenEig( char InvertMode, int nev, ARumSymMatrix<FLOAT>& A,
               ARumSymMatrix<FLOAT>& B, FLOAT sigma, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert, buckling and Cayley modes constructor (LAPACK dense version)

```
ARluSymGenEig( char InvertMode, int nev, ARdsSymMatrix<FLOAT>& A,
               ARdsSymMatrix<FLOAT>& B, FLOAT sigma, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert, buckling and Cayley modes constructor (LAPACK band version)

```
ARluSymGenEig( char InvertMode, int nev, ARbdSymMatrix<FLOAT>& A,
               ARbdSymMatrix<FLOAT>& B, FLOAT sigma, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Examples

This class requires the user to declare two matrices, say *A* and *B*, using `ARluSymMatrix`, `ARumSymMatrix`, `ARdsSymMatrix` or `ARbdSymMatrix` classes[21], as in the example below:

```
ARumSymMatrix<double> A(nA, nnzA, nzvalA, irowA, pcolA);
ARumSymMatrix<double> B(nB, nnzB, nzvalB, irowB, pcolB);
```

Henceforth, there are two different ways of creating a real symmetric eigenvalue problem (excluding the default constructor, as in the description of all other classes of this section):

1. Using the regular mode constructor

```
ARluSymGenEig<double> prob(4, A, B);
```

2. Using the shift and invert, buckling and Cayley modes constructor

```
ARluSymGenEig<double> prob('B', 4, A, B, 13.2); // Buckling mode.
```

[19] This constructor requires `A.uplo` and `B.uplo` to be equal (see the description of the `ARluSymMatrix` class for a description of `uplo`).
[20] This constructor also requires `A.uplo` and `B.uplo` to be equal (in this case, `uplo` is a parameter of the `ARumSymMatrix` class).
[21] For a description of all ARPACK++ matrix classes, see the *Available matrix classes* section below.

ARluNonSymGenEig

arlgnsym.h (SuperLU version)
arugnsym.h (UMFPACK version)
ardgnsym.h (LAPACK dense version)
arbgnsym.h (LAPACK band version)

Declaration

```
template <class FLOAT > class ARluNonSymGenEig
```

Description

This class defines a real nonsymmetric generalized eigenvalue problem with matrices stored using `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes, depending on which package is being used to perform the factorization of matrix B (if regular mode is chosen) or $(A - \sigma B)$ (when in shift and invert mode). `ARluNonSymMatrix` class requires the SuperLU library, while UMFPACK is used by `ARumNonSymMatrix` and LAPACK is called by `ARbdNonSymMatrix` and `ARdsNonSymMatrix`.

Because some fill-in can be generated by a sparse LU factorization, the user must be aware of the memory requirements associated to each spectral transformation.

Parent class (SuperLU version)

```
public virtual ARNonSymGenEig<FLOAT, ARluNonSymPencil<FLOAT, FLOAT>,
                                     ARluNonSymPencil<FLOAT, FLOAT> >
```

Parent class (UMFPACK version)

```
public virtual ARNonSymGenEig<FLOAT, ARumNonSymPencil<FLOAT, FLOAT>,
                                     ARumNonSymPencil<FLOAT, FLOAT> >
```

Parent class (LAPACK dense version)

```
public virtual ARNonSymGenEig<FLOAT, ARdsNonSymPencil<FLOAT, FLOAT>,
                                     ARdsNonSymPencil<FLOAT, FLOAT> >
```

Parent class (LAPACK band version)

```
public virtual ARNonSymGenEig<FLOAT, ARbdNonSymPencil<FLOAT, FLOAT>,
                                     ARbdNonSymPencil<FLOAT, FLOAT> >
```

Default constructor

```
ARluNonSymGenEig()
```

Regular mode constructor (SuperLU version)

```
ARluNonSymGenEig( int nev, ARluNonSymMatrix<FLOAT>& A,
                  ARluNonSymMatrix<FLOAT>& B, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (UMFPACK version)

```
ARluNonSymGenEig( int nev, ARumNonSymMatrix<FLOAT>& A,
                  ARumNonSymMatrix<FLOAT>& B, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluNonSymGenEig( int nev, ARdsNonSymMatrix<FLOAT>& A,
                  ARdsNonSymMatrix<FLOAT>& B, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluNonSymGenEig( int nev, ARbdNonSymMatrix<FLOAT>& A,
                  ARbdNonSymMatrix<FLOAT>& B, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (SuperLU version)

```
ARluNonSymGenEig( int nev, ARluNonSymMatrix<FLOAT>& A,
                  ARluNonSymMatrix<FLOAT>& B, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (UMFPACK version)

```
ARluNonSymGenEig( int nev, ARumNonSymMatrix<FLOAT>& A,
                  ARumNonSymMatrix<FLOAT>& B, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (LAPACK dense version)

```
ARluNonSymGenEig( int nev, ARdsNonSymMatrix<FLOAT>& A,
                  ARdsNonSymMatrix<FLOAT>& B, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor (LAPACK band version)

```
ARluNonSymGenEig( int nev, ARbdNonSymMatrix<FLOAT>& A,
                  ARbdNonSymMatrix<FLOAT>& B, FLOAT sigma,
                  char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                  int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor (SuperLU version)

```
ARluNonSymGenEig( int nev, ARluNonSymMatrix<FLOAT>& A,
                  ARluNonSymMatrix<FLOAT>& B, char part,
                  FLOAT sigmaR, FLOAT sigmaI, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor (UMFPACK version)

```
ARluNonSymGenEig( int nev, ARumNonSymMatrix<FLOAT>& A,
                  ARumNonSymMatrix<FLOAT>& B, char part,
                  FLOAT sigmaR, FLOAT sigmaI, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor (LAPACK dense version)

```
ARluNonSymGenEig( int nev, ARdsNonSymMatrix<FLOAT>& A,
                  ARdsNonSymMatrix<FLOAT>& B, char part,
                  FLOAT sigmaR, FLOAT sigmaI, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor (LAPACK band version)

```
ARluNonSymGenEig( int nev, ARbdNonSymMatrix<FLOAT>& A,
                  ARbdNonSymMatrix<FLOAT>& B, char part,
                  FLOAT sigmaR, FLOAT sigmaI, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Examples

This class requires the user to declare two matrices, say *A* and *B*, using one of the `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes, as in the example below:

```
ARluNonSymMatrix<double> A(nA, nnzA, nzvalA, irowA, pcolA);
ARluNonSymMatrix<double> B(nB, nnzB, nzvalB, irowB, pcolB);
```

Henceforth, there are three different ways of creating a nonsymmetric problem (excluding the default constructor, as in the above classes):

1. Using the regular mode constructor

```
ARluNonSymGenEig<double> prob(4, A, B);
```

2. Using the real shift and invert mode constructor

```
ARluNonSymGenEig<double> prob(4, A, B, 13.2);
```

3. Using the complex shift and invert mode constructor

```
ARluNonSymGenEig<double> prob(4, A, B, 'R', 1.4, 2.2);
```

ARluCompGenEig

arlgcomp.h (SuperLU version)
arugcomp.h (UMFPACK version)
ardgcomp.h (LAPACK dense version)
arbgcomp.h (LAPACK band version)

Declaration

```
template <class FLOAT > class ARluCompGenEig
```

Description

This class defines a complex generalized eigenvalue problem in the form $Ax = Bx\lambda$ with matrices stored using one of the `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes.

Both computational modes available in this class call SuperLU, UMFPACK or LAPACK routines to perform a sparse LU decomposition. In the regular mode, matrix B is factored. $(A - \sigma B)$ is decomposed when the shift and invert mode is used.

Because some fill-in can be generated during the factorization, the user must be aware of the memory requirements associated to each spectral transformation.

Parent class (SuperLU version)

```
public virtual
  ARCompGenEig<FLOAT, ARluNonSymPencil<complex<FLOAT>, FLOAT >,
               ARluNonSymPencil<complex<FLOAT>, FLOAT > >
```

Parent class (UMFPACK version)

```
public virtual
  ARCompGenEig<FLOAT, ARumNonSymPencil<complex<FLOAT>, FLOAT >,
               ARumNonSymPencil<complex<FLOAT>, FLOAT > >
```

Parent class (LAPACK dense version)

```
public virtual
  ARCompGenEig<FLOAT, ARdsNonSymPencil<complex<FLOAT>, FLOAT >,
               ARdsNonSymPencil<complex<FLOAT>, FLOAT > >
```

Parent class (LAPACK band version)

```
public virtual
  ARCompGenEig<FLOAT, ARbdNonSymPencil<complex<FLOAT>, FLOAT >,
               ARbdNonSymPencil<complex<FLOAT>, FLOAT > >
```

Default constructor

```
ARluCompGenEig()
```

Regular mode constructor (SuperLU version)

```
ARluCompGenEig( int nev, ARluNonSymMatrix<complex<FLOAT> >& A,
                ARluNonSymMatrix<complex<FLOAT> >& B, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Regular mode constructor (UMFPACK version)

```
ARluCompGenEig( int nev, ARumNonSymMatrix<complex<FLOAT> >& A,
                ARumNonSymMatrix<complex<FLOAT> >& B, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK dense version)

```
ARluCompGenEig( int nev, ARdsNonSymMatrix<complex<FLOAT> >& A,
                ARdsNonSymMatrix<complex<FLOAT> >& B, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Regular mode constructor (LAPACK band version)

```
ARluCompGenEig( int nev, ARbdNonSymMatrix<complex<FLOAT> >& A,
                ARbdNonSymMatrix<complex<FLOAT> >& B, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (SuperLU version)

```
ARluCompGenEig( int nev, ARluNonSymMatrix<complex<FLOAT> >& A,
                ARluNonSymMatrix<complex<FLOAT> >& B,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (UMFPACK version)

```
ARluCompGenEig( int nev, ARumNonSymMatrix<complex<FLOAT> >& A,
                ARumNonSymMatrix<complex<FLOAT> >& B,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK dense version)

```
ARluCompGenEig( int nev, ARdsNonSymMatrix<complex<FLOAT> >& A,
                ARdsNonSymMatrix<complex<FLOAT> >& B,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor (LAPACK band version)

```
ARluCompGenEig( int nev, ARbdNonSymMatrix<complex<FLOAT> >& A,
                ARbdNonSymMatrix<complex<FLOAT> >& B,
                complex<FLOAT> sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Examples

To use this class the user must declare two matrices, using the `ARluNonSymMatrix`, `ARumNonSymMatrix`, `ARdsNonSymMatrix` or `ARbdNonSymMatrix` classes.

If `ARumNonSymMatrix` is being used, these matrices can be declared as in the following example:

```
ARumNonSymMatrix<complex<double> > A(nA, nnzA, nzvalA, irowA, pcolA);
ARumNonSymMatrix<complex<double> > B(nB, nnzB, nzvalB, irowB, pcolB);
```

After that, the spectral transformation dictates which constructor should be used (the default constructor is not considered here):

1. Using the regular mode constructor

```
ARluCompGenEig<double> prob(4, A, B);
```

2. Using the shift and invert mode constructor

```
ARluCompGenEig<double> prob(4, A, B, complex<double>(0.8, 0.4));
```

Classes that require user-defined matrix-vector products.

ARSymStdEig

arssym.h

Declaration

```
template<class FLOAT, class FOP> class ARSymStdEig.
```

Description

This class defines a real symmetric standard eigenvalue problem.

Parent classes

```
public virtual ARStdEig<FLOAT, FLOAT, FOP>
public virtual ARrcSymStdEig<FLOAT>
```

Default constructor

```
ARSymStdEig()
```

Regular mode constructor

```
ARSymStdEig(int n, int nev, FOP* objOP, TypeOPx MultOPx,
            char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
            int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor

```
ARSymStdEig(int n, int nev, FOP* objOP, TypeOPx MultOPx, FLOAT sigma,
            char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
            int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Examples

Supposing that `Matrix` is a class that contains information concerning a specific symmetric matrix and a also contains a function called `MultVet`, which performs the matrix-vector product required by the computational mode being used[22], the three different ways of declaring a problem with class `ARSymStdEig` are exemplified below:

1. Using the default constructor

```
ARSymStdEig<double, Matrix<double> > EigProb;
```

If this constructor is used, all information about the problem must be passed elsewhere in the program using the `DefineParameters` function:

```
Matrix<double> A;
EigProb.DefineParameters(100, 4, &A, &Matrix<double>::MultVet);
```

Because a shift cannot be defined using `DefineParameters`, the following command is also necessary when solving the problem in shift and invert mode:

```
EigProb.ChangeShift(13.2);
```

2. Using the regular mode constructor

```
Matrix<double> A;
ARSymStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet);
```

3. Using the shift and invert mode constructor

```
Matrix<double> A;
ARSymStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet, 13.2);
```

ARNonSymStdEig

arsnsym.h

Declaration

```
template <class FLOAT, class FOP> class ARNonSymStdEig
```

Description

This class defines a real nonsymmetric standard eigenvalue problem.

Parent classes

```
public virtual ARStdEig<FLOAT, FLOAT, FOP>
public virtual ARrcNonSymStdEig<FLOAT>
```

Default constructor

```
ARNonSymStdEig()
```

[22] In regular mode, function `MultVet` should evaluate the matrix-vector product Av. In shift and invert mode, `MultVet` must return the product $(A\text{-}\sigma I)^{-1}v$.

Regular mode constructor

```
ARNonSymStdEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor

```
ARNonSymStdEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
                FLOAT sigma, char* which = "LM", int ncv = 0,
                FLOAT tol = 0.0, int maxit = 0,
                FLOAT* resid = 0, bool AutoShift = true)
```

Examples

Supposing that `Matrix` is a class that contains information concerning a specific matrix and a also contains a function called `MultVet`, which performs the matrix-vector product required by the computational mode being used, as described in chapter 3, it is possible to declare a nonsymmetric problem using three different constructors:

1. Using the default constructor

```
ARNonSymStdEig<double, Matrix<double> > EigProb;
```

In this case, after declaring the problem, the user must pass some information about the problem elsewhere in the program, using the `DefineParameters` function:

```
Matrix<double> A;
EigProb.DefineParameters(100, 4, &A, &Matrix<double>::MultVet);
```

Because a shift cannot be defined using `DefineParameters`, the following command is also necessary when solving the problem in shift and invert mode:

```
EigProb.ChangeShift(13.2);
```

2. Using the regular mode constructor

```
Matrix<double> A;
ARNonSymStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet);
```

3. Using the shift and invert mode constructor

```
Matrix<double> A;
ARNonSymStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet, 13.2);
```

ARCompStdEig

arscomp.h

Declaration

```
template<class FLOAT, class FOP> class ARCompStdEig
```

Description

This class defines a complex (Hermitian or non-Hermitian) standard eigenvalue problem.

Parent classes

```
public virtual ARStdEig<FLOAT, complex<FLOAT>, FOP>
public virtual ARrcCompStdEig<FLOAT>
```

Default constructor

```
ARCompStdEig()
```

Regular mode constructor

```
ARCompStdEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
              char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
              int maxit = 0, complex<FLOAT>* resid = 0,
              bool AutoShift = true)
```

Shift and invert mode constructor

```
ARCompStdEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
              complex<FLOAT> sigma, char* which = "LM",
              int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
              complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Examples

If `Matrix` is a class that contains information regarding a complex matrix and also contains a function called `MultVet`, which performs the matrix-vector product required by the computational mode selected, it is possible to declare a complex problem:

1. Using the default constructor

```
ARCompStdEig<double, Matrix<double> > EigProb;
```

If the default constructor is being used, all the parameters required by ARPACK++ must be passed elsewhere in the program using the `DefineParameters` function:

```
Matrix<double> A;
EigProb.DefineParameters(100, 4, &A, &Matrix<double>::MultVet);
```

Because a shift cannot be defined using `DefineParameters`, the following command is also necessary when solving the problem in shift and invert mode:

```
EigProb.ChangeShift(complex<double>(13.2, 10.4));
```

2. Using the regular mode constructor

```
Matrix<double> A;
ARCompStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet);
```

3. Using the shift and invert mode constructor

```
Matrix<double> A;
ARCompStdEig<double, Matrix<double> >
  EigProb(100, 4, &A, &Matrix<double>::MultVet,
          complex<double>(13.2, 10.4));
```

ARSymGenEig

argsym.h

Declaration

```
template <class FLOAT, class FOP, class FB> class ARSymGenEig
```

Description

Defines a real symmetric generalized eigenvalue problem.

Parent classes

```
public virtual ARGenEig<FLOAT, FLOAT, FOP, FB>
public virtual ARSymStdEig<FLOAT, FOP>
public virtual ARrcSymGenEig<FLOAT>
```

Default constructor

```
ARSymGenEig()
```

Regular mode constructor

```
ARSymGenEig(int n, int nev, FOP* objOP, TypeOPx MultOPx, FB* objB,
            TypeBx MultBx, char* which = "LM", int ncv = 0,
            FLOAT tol = 0.0, int maxit = 0, FLOAT* resid = 0,
            bool AutoShift = true)
```

Shift and invert and buckling modes constructor

```
ARSymGenEig(char InvertMode, int n, int nev, FOP* objOP,
            TypeOPx MultOPx, FB* objB, TypeBx MultBx, FLOAT sigma,
            char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
            int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Cayley mode constructor

```
ARSymGenEig(int n, int nev, FOP* objOP, TypeOPx MultOPx, FB* objA,
            TypeBx MultAx, FB* objB, TypeBx MultBx, FLOAT sigma,
            char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
            int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Examples

Because generalized problems include more than one matrix, before solving a problem using `ARSymGenEig` it is generally necessary to create two different classes, each one containing at least one function, say `MultVet`, which performs a matrix-vector product[23]. Supposing that all these classes are available, there are several ways to create a symmetric generalized problem:

1. Using the default constructor

In this case, after declaring an object of the `ARSymGenEig` class, the user should supply all the information about the problem required by ARPACK++ elsewhere in the program using the `DefineParameters` function:

```
ARSymGenEig<double, MatrixOP<double>, MatrixB<double> > EigProb;

// . . .

MatrixOP<double> OP;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
```

When solving the problem in shift and invert mode, another command is also necessary, because a shift cannot be defined using `DefineParameters`:

```
MatrixOP<double> OP;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
EigProb.SetShiftInvertMode(1.2, &OP, &MatrixOP<double>::MultVet);
```

The same occurs when using buckling mode:

```
MatrixOP<double> OP;
MatrixB<double> A;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &A, &MatrixB<double>::MultVet);
EigProb.SetBucklingMode(1.2, &OP, &MatrixOP<double>::MultVet);
```

In Cayley mode, three matrices – *OP*, *A* and *B* – are required. Although matrices *A* and *OP* must share the same class, they use different matrix-vector product functions:

```
MatrixOP<double> OP;
MatrixB<double> A;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
EigProb.SetCayleyMode(1.2, &OP, &MatrixOP<double>::MultVet,
                      &A, &MatrixB<double>::MultAVet);
```

2. Using the regular mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
```

[23] The required matrix-vector products are: a) in regular mode: $B^{-1}Ax$ and Bx; b) in shift and invert mode: $(A-\sigma B)^{-1}x$ and Bx; c) in buckling mode: $(A-\sigma B)^{-1}x$ and Ax; d) in Cayley mode: $(A-\sigma B)^{-1}x$, Ax and Bx. See chapter 4 for a detailed description of all modes.

```
ARSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet,
          &B, &MatrixB<double>::MultVet);
```

3. Using the shift and invert mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
ARSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb('S', 100, 4, &OP, &MatrixOP<double>::MultVet,
          &B, &MatrixB<double>::MultVet, 1.2);
```

4. Using the buckling mode constructor

```
MatrixOP<double> OP;
MatrixA<double> A;
ARSymGenEig<double, MatrixOP<double>, MatrixA<double> >
  EigProb('B', 100, 4, &OP, &MatrixOP<double>::MultVet,
          &A, &MatrixA<double>::MultVet, 1.2);
```

5. Using the Cayley mode constructor

```
MatrixOP<double> OP;
MatrixB<double> A;
MatrixB<double> B;
ARSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet, &A,
          &MatrixB<double>::MultAVet,
          &B, &MatrixB<double>::MultBVet, 1.2);
```

ARNonSymGenEig

argnsym.h

Declaration

```
template<class FLOAT, class FOP, class FB> class ARNonSymGenEig
```

Description

Defines a real nonsymmetric generalized eigenvalue problem.

Parent classes

```
public virtual ARGenEig<FLOAT, FLOAT, FOP, FB>
public virtual ARNonSymStdEig<FLOAT, FOP>
public virtual ARrcNonSymGenEig<FLOAT>
```

Default constructor

```
ARNonSymGenEig()
```

Regular mode constructor

```
ARNonSymGenEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
                FB* objB, TypeBx MultBx, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor

```
ARNonSymGenEig( int n, int nev, FOP* objOP, TypeOPx MultOPx, FB* objB,
                TypeBx MultBx, FLOAT sigma, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor

```
ARNonSymGenEig( int n, int nev, FOP* objOP, TypeOPx MultOPx, FB* objA,
                TypeBx MultAx, FB* objB, TypeBx MultBx, char part,
                FLOAT sigmaR, FLOAT sigmaI, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                FLOAT* resid = 0, bool AutoShift = true)
```

Examples

`ARNonSymGenEig` requires the user to supply two (or sometimes three) different matrix-vector products[24]. Supposing that at least two matrix classes are available, each one containing at least one function, say `MultVet`, which performs a matrix-vector product as required by the computational mode being used, some ways to create a real nonsymmetric generalized problem include:

1. Using the default constructor

In this case, after declaring the problem, the user must pass all the information required by ARPACK++ elsewhere in the program using the `DefineParameters` function:

```
ARNonSymGenEig<double, MatrixOP<double>, MatrixB<double> > EigProb;

// ...

MatrixOP<double> OP;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
```

Another command is also necessary when solving the problem in shift and invert mode, since a shift cannot be defined using `DefineParameters`:

```
MatrixOP<double> OP;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
EigProb.SetShiftInvertMode(1.2, &OP, &MatrixOP<double>::MultVet);
```

When shift is complex, three matrices – *OP*, *A* and *B* – are required. Although matrices *A* and *OP* must share the same class, they use different matrix-vector product functions:

[24] The required products are: a) in regular mode: $B^{-1}Ax$ and Bx; b) in real shift and invert mode: $(A-\sigma B)^{-1}x$ and Bx; c) in complex shift and invert mode: $real\{(A-\sigma B)^{-1}\}x$ *or* $imag\{(A-\sigma B)^{-1}\}x$, Ax and Bx. See chapter 4.

```
MatrixOP<double> OP;
MatrixB<double> A;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
EigProb.SetComplexShiftMode('R', 1.2, 0.8, &OP,
                            &MatrixOP<double>::MultVet,
                            &A, &MatrixB<double>::MultAVet);
```

2. Using the regular mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
ARNonSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet,
          &B, &MatrixB<double>::MultVet);
```

3. Using the real shift and invert mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
ARNonSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet,
          &B, &MatrixB<double>::MultVet, 1.2);
```

4. Using the complex shift and invert mode constructor

```
MatrixOP<double> OP;
MatrixOP<double> A;
MatrixB<double> B;
ARNonSymGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet,
          &A, &MatrixB<double>::MultAVet,
          &B, &MatrixB<double>::MultVet, 'R', 1.2, 0.8);
```

ARCompGenEig

argcomp.h

Declaration

```
template <class FLOAT, class FOP, class FB> class ARCompGenEig
```

Description

Defines a complex (Hermitian or non-Hermitian) generalized eigenvalue problem.

Parent classes

```
public virtual ARGenEig<FLOAT, complex<FLOAT>, FOP, FB>
public virtual ARCompStdEig<FLOAT, FOP>
```

Default constructor

```
ARCompGenEig()
```

Regular mode constructor

```
ARCompGenEig( int n, int nev, FOP* objOP, TypeOPx MultOPx,
              FB* objB, TypeBx MultBx, char* which = "LM",
              int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
              complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor

```
ARCompGenEig( int n, int nev, FOP* objOP, TypeOPx MultOPx, FB* objB,
              TypeBx MultBx, complex<FLOAT> sigma, char* which = "LM",
              int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
              complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Examples

To solve complex generalized problems it is also necessary to create two matrix-vector product functions[25]. In the examples given below, both these functions are called `MultVet`, but each one belongs to a different class.

1. Using the default constructor

```
ARCompGenEig<double, MatrixOP<double>, MatrixB<double> > EigProb;
```

In this case, after declaring an object that belongs to `ARCompGenEig`, it is necessary to use the `DefineParameters` function to pass all the remaining information required by ARPACK++:

```
MatrixOP<double> OP;
MatrixB<double> B;
EigProb.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet,
                         &B, &MatrixB<double>::MultVet);
```

Because a shift cannot be defined using `DefineParameters`, the following command is also necessary when using shift and invert mode:

```
EigProb.SetShiftInvertMode(complex<double>(1.2, 0.8),
                           &OP, &MatrixOP<double>::MultVet);
```

2. Using the regular mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
ARCompGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet,
          &B, &MatrixB<double>::MultVet);
```

3. Using the shift and invert mode constructor

```
MatrixOP<double> OP;
MatrixB<double> B;
```

[25] These functions are: a) in regular mode: $B^{-1}Ax$ and Bx; b) in shift and invert mode: $(A-\sigma B)^{-1}x$ and Bx. See chapter 4.

```
ARCompGenEig<double, MatrixOP<double>, MatrixB<double> >
  EigProb(100, 4, &OP, &MatrixOP<double>::MultVet, &B,
          &MatrixB<double>::MultVet, complex<double>(1.2, 0.8));
```

Classes that do not handle matrix information.

These classes were created only to maintain full compatibility between c++ and FORTRAN versions of ARPACK. They implement the so called *reverse communication interface* and should be used only if matrix neither matrix data nor matrix-vector products can be passed to class constructors, as described in the above sections.

Although it is easy to declare objects using these classes, obtaining eigenvalues and eigenvectors require the user to explicitly define an awkward sequence of calls to a function called `TakeStep` combined with matrix-vector products until convergence is attained.

ARrcSymStdEig

arrssym.h

Declaration

```
template<class FLOAT> class ARrcSymStdEig.
```

Description

Defines a real symmetric standard eigenvalue problem without requiring a matrix-vector product function.

Parent class

```
public virtual ARrcStdEig<FLOAT, FLOAT>
```

Default constructor

```
ARrcSymStdEig()
```

Regular mode constructor

```
ARrcSymStdEig( int n, int nev, char* which = "LM", int ncv = 0,
               FLOAT tol = 0.0, int maxit = 0, FLOAT* resid = 0,
               bool AutoShift = true)
```

Shift and invert mode constructor

```
ARrcSymStdEig( int n, int nev, FLOAT sigma, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Examples

Defining objects of this class is straightforward:

1. Using the default constructor

```
ARrcSymStdEig<double> EigProb;
```

When using the default constructor, problem parameters must be passed elsewhere in the program using functions `DefineParameters` and `ChangeShift`[26]:

```
EigProb.DefineParameters(100, 4);
EigProb.ChangeShift(13.2);
```

2. Using the regular mode constructor

```
ARrcSymStdEig<double> EigProb(100, 4);
```

3. Using the shift and invert mode constructor

```
ARrcSymStdEig<double> EigProb(100, 4, 13.2);
```

ARrcNonSymStdEig

arrsnsym.h

Declaration

```
template <class FLOAT> class ARrcNonSymStdEig
```

Description

Defines a real nonsymmetric standard eigenvalue problem.

Parent class

```
public virtual ARrcStdEig<FLOAT, FLOAT>
```

Default constructor

```
ARrcNonSymStdEig()
```

Regular mode constructor

```
ARrcNonSymStdEig( int n, int nev, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor

```
ARrcNonSymStdEig( int n, int nev, FLOAT sigma, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Examples

The user can easily declare objects of this class:

[26] Only when solving a problem in shift and invert mode.

1. **Using the default constructor**

```
ARrcNonSymStdEig<double> EigProb;
```

When using this constructor, all other information about the problem must be passed elsewhere in the program using `DefineParameters` and `ChangeShift26`:

```
EigProb.DefineParameters(100, 4);
EigProb.ChangeShift(13.2);
```

2. **Using the regular mode constructor**

```
ARrcNonSymStdEig<double> EigProb(100, 4);
```

3. **Using the shift and invert mode constructor**

```
ARrcNonSymStdEig<double> EigProb(100, 4, 13.2);
```

ARrcCompStdEig

arrscomp.h

Declaration

```
template<class FLOAT> class ARrcCompStdEig
```

Description

Defines a complex (Hermitian or non-Hermitian) standard eigenvalue problem.

Parent class

```
public virtual ARrcStdEig<FLOAT, complex<FLOAT> >
```

Default constructor

```
ARrcCompStdEig()
```

Regular mode constructor

```
ARrcCompStdEig( int n, int nev, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0,  bool AutoShift = true)
```

Shift and invert mode constructor

```
ARrcCompStdEig( int n, int nev, complex<FLOAT> sigma,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Examples

1. **Using the default constructor**

```
ARrcCompStdEig<double> EigProb;
```

```
// . . .

EigProb.DefineParameters(100, 4);
EigProb.ChangeShift(complex<double>(13.2, 10.4));
```

As illustrated in the above example, to use the default constructor, it is also necessary to call other functions, such as `DefineParameters` and `ChangeShift`, to pass some problem parameters to ARPACK++.

In this case, `DefineParameters` was used to define the size of the problem and the number of desired eigenvalues, while `ChangeShift` was called to define the shift (supposing that the shift and invert mode should be employed).

2. Using the regular mode constructor

```
ARrcCompStdEig<double> EigProb(100, 4);
```

3. Using the shift and invert mode constructor

```
ARrcCompStdEig<double> EigProb(100, 4, complex<double>(13.2, 10.4));
```

ARrcSymGenEig

arrgsym.h

Declaration

```
template <class FLOAT> class ARrcSymGenEig
```

Description

Defines a real symmetric generalized eigenvalue problem.

Parent classes

```
public virtual ARrcGenEig<FLOAT, FLOAT>
public virtual ARrcSymStdEig<FLOAT>
```

Default constructor

```
ARrcSymGenEig()
```

Regular mode constructor

```
ARrcSymGenEig( int n, int nev, char* which = "LM",
               int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
               FLOAT* resid = 0, bool AutoShift = true)
```

Shift and invert, buckling and Cayley modes constructor

```
ARrcSymGenEig( char InvertMode, int n, int nev, FLOAT sigma,
               char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
               int maxit = 0, FLOAT* resid = 0, bool AutoShift = true)
```

Examples

1. Using the default constructor

```
ARrcSymGenEig<double> EigProb;
```

When using this constructor, all problem parameters must be passed elsewhere in the program using the `DefineParameters` function:

```
EigProb.DefineParameters(100, 4);
```

One of the three commands below is also required when a spectral transformation is being used:

```
EigProb.SetShiftInvertMode(1.2);
EigProb.SetBucklingMode(1.2);
EigProb.SetCayleyMode(1.2);
```

2. Using the regular mode constructor

```
ARrcSymGenEig<double> EigProb(100, 4);
```

3. Using the shift and invert, buckling and Cayley modes constructor

To define a spectral transformation while defining an object of this class, one of the following commands must be used:

```
ARrcSymGenEig<double> EigProb('S', 100, 4, 1.2);
ARrcSymGenEig<double> EigProb('B', 100, 4, 1.2);
ARrcSymGenEig<double> EigProb('C', 100, 4, 1.2);
```

ARrcNonSymGenEig

arrgnsym.h

Declaration

```
template<class FLOAT> class ARrcNonSymGenEig
```

Description

Defines a real nonsymmetric generalized eigenvalue problem.

Parent classes

```
public virtual ARrcGenEig<FLOAT, FLOAT>
public virtual ARrcNonSymStdEig<FLOAT>
```

Default constructor

```
ARrcNonSymGenEig()
```

Regular mode constructor

```
ARrcNonSymGenEig( int n, int nev, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Real shift and invert mode constructor

```
ARrcNonSymGenEig( int n, int nev, FLOAT sigma, char* which = "LM",
                  int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                  FLOAT* resid = 0, bool AutoShift = true)
```

Complex shift and invert mode constructor

```
ARrcNonSymGenEig( int n, int nev, char part, FLOAT sigmaR,
                  FLOAT sigmaI, char* which = "LM",int ncv = 0,
                  FLOAT tol = 0.0, int maxit = 0, FLOAT* resid = 0,
                  bool AutoShift = true)
```

Examples

1. Using the default constructor

```
ARrcNonSymGenEig<double> EigProb;
```

When using the default constructor, all other relevant information about the problem must be passed using the `DefineParameters` function:

```
EigProb.DefineParameters(100, 4);
```

One of the following commands is also necessary when solving the problem in real or complex shift and invert mode:

```
EigProb.SetShiftInvertMode(1.2);
EigProb.SetComplexShiftMode('R', 1.2, 0.8);
```

2. Using the regular mode constructor

```
ARrcNonSymGenEig<double> EigProb(100, 4);
```

3. Using the real shift and invert mode constructor

```
ARrcNonSymGenEig<double> EigProb(100, 4, 1.2);
```

4. Using the complex shift and invert mode constructor

```
ARCompGenEig<double> EigProb(100, 4, 'R', 1.2, 0.8);
```

ARrcCompGenEig

arrgcomp.h

Declaration

```
template <class FLOAT> class ARrcCompGenEig
```

Description

Defines a complex (Hermitian or non-Hermitian) generalized eigenvalue problem.

Parent classes

```
public virtual ARrcGenEig<FLOAT, complex<FLOAT> >
public virtual ARrcCompStdEig<FLOAT>
```

Default constructor

```
ARrcCompGenEig()
```

Regular mode constructor

```
ARrcCompGenEig( int n, int nev, char* which = "LM",
                int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                complex<FLOAT>* resid = 0, bool AutoShift = true)
```

Shift and invert mode constructor

```
ARrcCompGenEig( int n, int nev, complex<FLOAT> sigma,
                char* which = "LM", int ncv = 0, FLOAT tol = 0.0,
                int maxit = 0, complex<FLOAT>* resid = 0,
                bool AutoShift = true)
```

Examples

1. Using the default constructor

```
ARrcCompGenEig<double> EigProb;
```

When using this constructor, the user must employ functions `DefineParameters` and `SetShiftInvertMode` to supply all problem parameters:

```
EigProb.DefineParameters(100, 4);
EigProb.SetShiftInvertMode(complex<double>(1.2, 0.8));
```

2. Using the regular mode constructor

```
ARrcCompGenEig<double> EigProb(100, 4);
```

3. Using the shift and invert mode constructor

```
ARrcCompGenEig<double> EigProb(100, 4, complex<double>(1.2, 0.8));
```

Base classes.

The classes described below are used as a basis for the definition of all other ARPACK++ classes. They are not intended to be used directly and were included here only for the sake of completeness.

ARrcStdEig

arrseig.h

Declaration

`template<class FLOAT> class ARrcStdEig`

Description

Defines a generic standard eigenvalue problem.

Default constructor

`ARrcStdEig()`

ARrcGenEig

arrgeig.h

Declaration

`template<class FLOAT> class ARrcGenEig`

Description

Defines a generic generalized eigenvalue problem.

Parent class

`public virtual ARrcStdEig<FLOAT, TYPE>`

Default constructor

`ARrcGenEig()`

ARStdEig

arseig.h

Declaration

`template<class FLOAT, class TYPE, class FOP > class ARStdEig`

Description

Defines a generic standard eigenvalue problem, supposing that class `FOP` contains a member function that performs a matrix-vector product.

Parent class

`public virtual ARrcStdEig<FLOAT, TYPE>`

Default constructor

`ARStdEig()`

ARGenEig

argeig.h

Declaration

```
template<class FLOAT, class TYPE, class FOP, class FB> class ARGenEig
```

Description

Defines a generic generalized eigenvalue problem $Ax = Bx\lambda$, supposing that information about matrices A and B is provided by classes `FOP` and `FB`.

Parent classes

```
public virtual ARrcGenEig<FLOAT, TYPE>
public virtual ARStdEig<FLOAT, TYPE, OP>
```

Default constructor

```
ARGenEig()
```

Matrix classes.

Although the user is encouraged to use his own matrix classes, ARPACK++ includes some predefined classes that can be used to create dense matrices and sparse matrices in compressed sparse column (CSC) or band format.

The main purpose of these classes is to help the user to define eigenvalue problems, but some of them can also be used to solve linear systems or to perform matrix-vector products.

There are eight matrix classes and another eight classes that represent pencils. They are divided according to the presence of symmetry and also according to the library that is used to solve linear systems.

Matrices.

ARbdSymMatrix

arbsmat.h

Declaration

```
template<class TYPE> class ARbdSymMatrix
```

Description

This class defines a real symmetric band matrix.

Warning: Two member functions of this class, `FactorA` and `FactorAsI`, call LAPACK to perform a matrix decomposition. These two functions are used by ARPACK++ to solve generalized eigenvalue problems and also standard problems in shift and invert mode. Although `ARbdSymMatrix` should only be used to define symmetric matrices, a LU factorization with partial pivoting is used. Since many nonzero elements are generated during the matrix decomposition, memory availability must be taken in account when using these member functions.

Default constructor

```
ARbdSymMatrix()
```

Long constructor

```
ARbdSymMatrix(int n, int nsdiag, TYPE* nzval, char uplo = 'L')
```

Constructor parameters

`n`	number of columns.
`nsdiag`	number subdiagonals (or superdiagonals), not including the main diagonal.
`nzval`	pointer to an array that contains the nonzero elements of the upper or lower triangular part of the matrix. The nonzero elements must be supplied by columns and, within each column, they must also be in ascending order of row indices. `nzval` can be viewed as a matrix with `n` columns and `nsdiag+1` rows, where the columns are stored sequentially in the same vector. In this representation, each row of the $(nsdiag+1)\times n$ matrix contains a diagonal from the original matrix. If the upper triangular part of the matrix is supplied, the main diagonal is stored in the last row. On the other hand, the main diagonal is the first row if `uplo = 'L'`.
`uplo`	character variable that indicates if the user intends to define the matrix by supplying its lower (`uplo = 'L'`) or upper triangular (`uplo = 'U'`) nonzero elements.

Public member functions

`int nrows()`

returns n.

`int ncols()`

returns `n`.

`void FactorA()`

Performs the LU factorization of A, a matrix that belongs to this class.

`void FactorAsI(TYPE sigma)`

Performs the LU decomposition of $(A-\sigma I)$, where I is the identity matrix.

`bool IsFactored()`

indicates if the LU decomposition of matrix A or $(A-\sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultInvv(TYPE* v, TYPE* w)`

Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

`void DefineMatrix(int n, int nsdiag, TYPE* nzval, char uplo = 'L')`

Stores matrix data when the default constructor is being used.

Example

To store the symmetric matrix

$$A = \begin{bmatrix} 4 & 1 & 3 & 0 & 0 & 0 \\ 1 & 6 & 2 & 0 & 0 & 0 \\ 3 & 2 & 1 & 4 & 0 & 0 \\ 0 & 0 & 4 & 2 & 1 & 4 \\ 0 & 0 & 0 & 1 & 3 & 2 \\ 0 & 0 & 0 & 4 & 2 & 5 \end{bmatrix}.$$

in band format it is necessary first to rewrite the lower triangular part of A as a $(nsdiag+1)\times n$ matrix:

$$A_L^{band} = \begin{bmatrix} 4 & 6 & 1 & 2 & 3 & 5 \\ 1 & 2 & 4 & 1 & 2 & 0 \\ 3 & 0 & 0 & 4 & 0 & 0 \end{bmatrix}$$

After that, the user must declare a vector, say `nzval`, that contains the columns of A_L^{band}:

```
double nzval[] = { 4.0, 1.0, 3.0, 6.0, 2.0, 0.0, 1.0, 4.0, 0.0,
                   2.0, 1.0, 4.0, 3.0, 2.0, 0.0, 5.0, 0.0, 0.0 };
```

As an alternative, the upper triangular part of matrix A can also be used to declare `nzval`:

$$A_U^{band} = \begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 4 \\ 0 & 1 & 2 & 4 & 1 & 2 \\ 4 & 6 & 1 & 2 & 3 & 5 \end{bmatrix}$$

```
double nzval[] = { 0.0, 0.0, 4.0, 0.0, 1.0, 6.0, 3.0, 2.0, 1.0,
                   0.0, 4.0, 2.0, 0.0, 1.0, 3.0, 4.0, 2.0, 5.0 };
```

Notice that some entries in A_L^{band} and A_U^{band} were artificially set to zero because the diagonals of A do not contain the same number of elements.

Once `nzval` is available, one of the constructors described above should be used to declare *A* as an `ARbdSymMatrix` object:

1. Using the long constructor

```
ARbdSymMatrix<double> A(6, 2, nzval);      // uplo = ’L’.
ARbdSymMatrix<double> B(6, 2, nzval, ’U’); // uplo = ’U’.
```

2. Using the default constructor

```
ARbdSymMatrix<double> A;
A.DefineMatrix(6, 2, nzval); // uplo = ’L’.
```

After declaring *A*, to solve a linear system, say $Ax = y$, where x and y are dense vectors with compatible dimensions, the matrix must be factored and then `MultInvv` called:

```
A.FactorA();
A.MultInvv(y, x);
```

ARdsSymMatrix

ardsmat.h

Declaration

`template<class TYPE> class ARdsSymMatrix`

Description

This class defines a real symmetric dense matrix.

Warning: Two member functions of this class, `FactorA` and `FactorAsI`, call LAPACK to perform a matrix decomposition. These two functions are used by ARPACK++ to solve generalized eigenvalue problems and also standard problems in shift and invert mode. Although `ARdsSymMatrix` should only be used to define symmetric matrices, a LU factorization with partial pivoting is used. Since many nonzero elements are generated during the matrix decomposition, memory availability must be taken in account when using these member functions.

Default constructor

`ARdsSymMatrix()`

Long constructor

`ARdsSymMatrix(int n, TYPE* nzval, char uplo = 'L')`

Constructor parameters

`n`	number of columns.
`nzval`	pointer to an array that contains the nonzero elements of the upper or lower triangular part of the matrix. The nonzero elements must be

supplied by columns and, within each column, they must also be in ascending order of row indices.

`uplo` character variable that indicates if the user intends to define the matrix by supplying its lower (`uplo = 'L'`) or upper triangular (`uplo = 'U'`) nonzero elements.

Public member functions

`int nrows()`
returns n.

`int ncols()`
returns `n`.

`void FactorA()`
Performs the LU factorization of A, a matrix that belongs to this class.

`void FactorAsI(TYPE sigma)`
Performs the LU decomposition of $(A-\sigma I)$, where I is the identity matrix.

`bool IsFactored()`
indicates if the LU decomposition of matrix A or $(A-\sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`
Calculates $w \leftarrow Av$.

`void MultInvv(TYPE* v, TYPE* w)`
Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

`void DefineMatrix(int n, TYPE* nzval, char uplo = 'L')`
Stores matrix data when the default constructor is being used.

Example

To store the symmetric matrix

$$A = \begin{bmatrix} 4 & 1 & 3 & -1 & 2 \\ 1 & 5 & 6 & 2 & 0 \\ 3 & 6 & -3 & -1 & 4 \\ -1 & 2 & -1 & 3 & 2 \\ 2 & 0 & 4 & 2 & 5 \end{bmatrix}.$$

in a compact format it is necessary declare a vector, say `nzval`, that contains the lower triangular part of the columns of A:

```
double nzval[] = { 4.0, 1.0, 3.0, -1.0, 2.0, 5.0, 6.0, 2.0,
                   0.0, -3.0, -1.0, 4.0, 3.0, 2.0, 5.0 };
```

As an alternative, the upper triangular part of matrix A can also be used to declare `nzval`:

```
double nzval[] = { 4.0, 1.0, 5.0, 3.0, 6.0, -3.0, -1.0, 2.0,
                   -1.0, 3.0, 2.0, 0.0, 4.0, 2.0, 5.0 };
```

Once `nzval` is available, one of the constructors described above should be used to declare *A* as an `ARdsSymMatrix` object:

1. Using the long constructor

```
ARdsSymMatrix<double> A(5, nzval);      // uplo = 'L'.
ARdsSymMatrix<double> B(5, nzval, 'U'); // uplo = 'U'.
```

2. Using the default constructor

```
ARdsSymMatrix<double> A;
A.DefineMatrix(6, nzval); // uplo = 'L'.
```

After declaring *A*, to solve a linear system, say $Ax = y$, where *x* and *y* are dense vectors with compatible dimensions, the matrix must be factored and then `MultInvv` called:

```
A.FactorA();
A.MultInvv(y, x);
```

ARluSymMatrix

arlsmat.h

Declaration

`template<class TYPE> class ARluSymMatrix`

Description

This class defines a real symmetric matrix in compressed sparse column (CSC) format.

Warning: Two member functions of this class, `FactorA` and `FactorAsI`, call the SuperLU package to perform a matrix decomposition. These functions are used by ARPACK++ to solve generalized eigenvalue problems and standard problems in shift and invert mode.

In spite of the fact that this class should be used to define symmetric matrices, a LU factorization is used instead of a symmetric decomposition. However, a column reordering based on the minimum degree algorithm can be done before the factorization, as described below. Since some fill-ins can occur during the elimination phase, memory availability must be taken in account when using the two functions mentioned above

Default constructor

`ARluSymMatrix()`

Long constructor

```
ARluSymMatrix(int n, int nnz, TYPE* nzval, int* irow,
              int* pcol, char uplo = 'L', double thresh = 0.1,
              int order = 2, bool check = true)
```

Long constructor (Harwell-Boeing file)

```
ARluSymMatrix(char* name, double thresh = 0.1,
              int order = 2, bool check = true)
```

Constructor parameters

`n`	number of columns.
`nnz`	number of nonzero elements in `nzval`.
`nzval`	pointer to an array that contains the nonzero elements of the upper or lower triangular part of the matrix. Within each column, `nzval` components must be supplied in ascending order of row indices.
`irow`	pointer to an array of row indices of the nonzero elements contained in `nzval`.
`pcol`	pointer to an array of pointers to the beginning of columns in `nzval` and `irow`. Such array must have `n`+1 elements and the last element must be equal to `nnz`.
`uplo`	character variable that indicates if the user intends to define the matrix by supplying its lower (`uplo = ’L’`) or upper triangular (`uplo = ’U’`) nonzero elements.
`name`	name of the file that stores the matrix in Harwell-Boeing format.
`thresh`	relative pivot tolerance used during the matrix factorization. At step i of the Gaussian elimination process, a_{ii} is used as pivot if $\lvert a_{ii}\rvert \geq$ `thresh` $\cdot \max_{j \geq i} \lvert a_{ji}\rvert$. No pivoting will be used if `thresh = 0`, while `thresh = 1` corresponds to partial pivoting.
`order`	integer variable that indicates which column permutation should be performed prior to the decomposition of A or $(A - \sigma I)$. If `order = 0`, the original column order will be preserved. If `order = 1`, the minimum degree ordering on the structure of $A^T A$ is used (this option is intended to be used with nonsymmetric matrices, and should be avoided when declaring matrices with this class). Finally, `order = 2` means that the minimum degree ordering should be applied on the structure of $A^T + A$ (this is the default option for symmetric matrices).
`check`	boolean variable that indicates if matrix data is to be checked for inconsistencies. When `check = true`, ARPACK++ checks if `pcol` is in ascending order and if `irow` components are in order and within bounds.

Public member functions

`int  nrows()`

returns n.

`int  ncols()`

returns `n`.

`int  nzeros()`

returns `nnz`.

`void FactorA()`

Performs the LU factorization of A, a matrix that belongs to this class. This function can only be used if the SuperLU library was previously installed.

`void FactorAsI(TYPE sigma)`

Performs the LU decomposition of $(A-\sigma I)$, where I is the identity matrix. This function also calls SuperLU routines.

`bool IsFactored()`

indicates if the LU decomposition of matrix A or $(A-\sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultInvv(TYPE* v, TYPE* w)`

Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

```
void DefineMatrix( int n, int nnz, TYPE* nzval, int* irow,
                   int* pcol, char uplo = 'L', double thresh = 0.1,
                   int order = 2, bool check = true)
```

Stores matrix data when the default constructor is being used.

Example

To store the lower triangular part of the symmetric matrix

$$A = \begin{bmatrix} -1 & 0 & 0 & 0 & -2 \\ 0 & 3 & 2 & 0 & 0 \\ 0 & 2 & 1 & 4 & 0 \\ 0 & 0 & 4 & 3 & 0 \\ -2 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

in CSC format it is necessary to declare the following vectors

```
double nzval[] = { -1.0, -2.0, 3.0, 2.0, 1.0, 4.0, 3.0, -1.0 };
int irow[]     = { 1, 5, 2, 3, 3, 4, 4, 5 };
int pcol[]     = { 0, 2, 4, 6, 7, 8 };
```

Alternatively, `nzval`, `irow` and `pcol` can also contain the upper triangular part of matrix A, as in the following example:

```
double nzval[] = { -1.0, 3.0, 2.0, 1.0, 4.0, 3.0, -2.0, -1.0 };
```

```
int irow[]     = { 1, 2, 2, 3, 3, 4, 1, 5 };
int pcol[]     = { 0, 1, 2, 4, 6, 8 };
```

Once these vectors are available, one of the constructors described above should be used to declare *A* as an `ARluSymMatrix` object:

1. Using the long constructor

```
ARluSymMatrix<double> A(5, 8, nzval, irow, pcol);      // uplo = 'L'.
ARluSymMatrix<double> B(5, 8, nzval, irow, pcol, 'U'); // uplo = 'U'.
```

2. Using the default constructor

```
ARluSymMatrix<double> A;
A.DefineMatrix(5, 8, nzval, irow, pcol); // uplo = 'L'.
```

After declaring the matrix, to solve a linear system, say $Ax = y$, where x and y are dense vectors with compatible dimensions, *A* must be factored and then `MultInvv` called:

```
A.FactorA();
A.MultInvv(y, x);
```

ARumSymMatrix

arusmat.h

Declaration

`template<class TYPE> class ARumSymMatrix`

Description

This class defines a real symmetric matrix in compressed sparse column (CSC) format.

Warning: The UMFPACK library is called by `FactorA` and `FactorAsI` member functions to perform a matrix decomposition. These functions are used by ARPACK++ to solve generalized eigenvalue problems and standard problems in shift and invert mode. In spite of the fact that this class should be used to define symmetric matrices, a LU factorization is used[27]. Due to fill-ins that occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

`ARumSymMatrix()`

[27] The main purpose of the UMFPACK package is to solve nonsymmetric systems, but its factorization routines contain a few parameters that can be adjusted to take in account the symmetric structure of the matrix.

Long constructor

```
ARumSymMatrix( int n, int nnz, TYPE* nzval, int* irow, int* pcol,
               char uplo = 'L', double thresh = 0.1, int fillin = 9,
               bool redcbl = true, bool check = true)
```

Long constructor (Harwell-Boeing file)

```
ARumSymMatrix( char* name, double thresh = 0.1, int fillin = 9,
               bool redcbl = true, bool check = true)
```

Constructor parameters

`n`	number of columns.
`nnz`	number of nonzero elements in `nzval`.
`nzval`	pointer to an array that contains the nonzero elements of the upper or the lower triangular part of the matrix. Within each column, `nzval` components must be supplied in ascending order of row indices.
`irow`	pointer to an array of row indices of the elements contained in `nzval`.
`pcol`	pointer to an array of pointers to the beginning of columns in `nzval` and `irow`. This array must have `n`+1 elements and the last element must be equal to `nnz`.
`uplo`	a character that indicates if the user intends to define the matrix by supplying its lower (`uplo = 'L'`) or upper triangular (`uplo = 'U'`) nonzero elements.
`name`	name of the file that stores the matrix in Harwell-Boeing format.
`thresh`	relative pivot tolerance used during the matrix factorization. At step *i* of the Gaussian elimination process, a_{ii} is used as pivot if $\|a_{ii}\| \geq \texttt{thresh} \cdot \max_{j \geq i} \|a_{ji}\|$. No pivoting will be used if `thresh = 0`, while `thresh = 1` corresponds to partial pivoting.
`fillin`	expected growth in matrix elements due to factorization. `FactorA` and `FactorAsI` functions will reserve 2*`fillin`*`nnz` memory positions for fill-ins occurred during LU decomposition.
`redcbl`	boolean variable that indicates whether or not to attempt a permutation to block triangular form. If `redcbl` is set to `true`, the permutation is attempted.
`check`	boolean variable that indicates if matrix data is to be checked for inconsistencies. When `check = true`, ARPACK++ checks if `pcol` is in ascending order and if `irow` components are in order and within bounds.

Member functions

```
int nrows()
```

returns `n`.

`int ncols()`
: returns `n`.

`int nzeros()`
: returns `nnz`.

`int FillFact()`
: returns `fillin`.

`void FactorA()`
: Performs the LU factorization of A, a matrix that belongs to this class. This function can only be used if the UMFPACK library was previously installed.

`void FactorAsI(TYPE sigma)`
: Performs the LU decomposition of $(A - \sigma I)$, where I is the identity matrix. This function also calls UMFPACK routines.

`bool IsFactored()`
: indicates if the LU decomposition of matrix A or $(A - \sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow Av$.

`void MultInvv(TYPE* v, TYPE* w)`
: Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

```
void DefineMatrix( int n, int nnz, TYPE* nzval, int* irow,
                   int* pcol, char uplo = 'L', double thresh = 0.1,
                   int fillin = 9, bool redcbl = true, bool check = true)
```
: Stores matrix data when the default constructor is being used.

Example

Supposing that the same three vectors `nzval`, `irow` and `pcol` defined in the example of the `ARluSymMatrix` class above are available, it is easy to declare a symmetric matrix using one of the constructors of the `ARumSymMatrix` class:

1. Using the long constructor

```
ARumSymMatrix<double> A(5, 8, nzval, irow, pcol);      // uplo = ’L’.
ARumSymMatrix<double> B(5, 8, nzval, irow, pcol, ’U’); // uplo = ’U’.
```

2. Using the default constructor

```
ARumSymMatrix<double> A;
A.DefineMatrix(5, 8, nzval, irow, pcol); // uplo = ’L’.
```

After declaring *A*, to solve a linear system, say $Ax = y$, where x and y are dense vectors with compatible dimensions, the matrix must be factored and then `MultInvv` called:

```
A.FactorA();
A.MultInvv(y, x);
```

ARbdNonSymMatrix

arbnsmat.h

Declaration

`template<class TYPE> class ARbdNonSymMatrix`

Description

This class defines a real or complex nonsymmetric square matrix in band format.

Warning: The LAPACK library is called by two member functions of this class, `FactorA` and `FactorAsI`, to perform a matrix decomposition. These functions are used by ARPACK++ to solve generalized eigenvalue problems and also standard problems in shift and invert mode. Since many nonzero elements are generated during the LU factorization, memory availability must be taken in account when using these two member functions.

Default constructor

`ARbdNonSymMatrix()`

Long constructor

`ARbdNonSymMatrix(int n, int ndiagL, int ndiagU, TYPE* nzval)`

Constructor parameters

`n`	number of columns.
`ndiagL`	number of subdiagonals.
`ndiagU`	number of superdiagonals.
`nzval`	pointer to an array that contains all the nonzero matrix elements, ordered by columns. Within each column, `nzval` components must be supplied in ascending order of row indices. `nzval` can be viewed as a matrix with `n` columns and `ndiagL+ndiagU+1` rows, where the columns are stored sequentially in the same vector. In this representation, each row of the compact matrix contains a diagonal from the original matrix.

Member functions

`int  nrows()`

returns `n`.

`int  ncols()`

returns `n`.

`void FactorA()`

Performs the LU factorization of A, a matrix that belongs to this class.

`void FactorAsI(TYPE sigma)`

Performs the LU decomposition of $(A - \sigma I)$, where I is the identity matrix.

`bool IsFactored()`

indicates if the LU decomposition of matrix A or $(A-\sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultMtv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow A^T v$.

`void MultMtMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow A^T Av$.

`void MultMMtv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow AA^T v$.

`void MultOMMtOv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow \overline{A}v$, where

$$\overline{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

`void MultInvv(TYPE* v, TYPE* w)`

Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

`void DefineMatrix(int n, int ndiagL, int ndiagU, TYPE* nzval)`

Stores matrix data when the default constructor is being used.

Example

To store matrix

$$A = \begin{bmatrix} 4 & 3 & 0 & 0 & 0 & 0 \\ 1 & 6 & 2 & 0 & 0 & 0 \\ -1 & 2 & 1 & 4 & 0 & 0 \\ 0 & 0 & 3 & -2 & -1 & 0 \\ 0 & 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 4 & 2 & 5 \end{bmatrix}.$$

in band format it is necessary first to rewrite A as a $(ndiagL + ndiagU + 1) \times n$ matrix[28]:

$$A_{band} = \begin{bmatrix} 0 & 3 & 2 & 4 & -1 & 1 \\ 4 & 6 & 1 & -2 & 3 & 5 \\ 1 & 2 & 3 & 1 & 2 & 0 \\ -1 & 0 & 0 & 4 & 0 & 0 \end{bmatrix}$$

[28] Notice that some entries in A_{band} are artificially set to zero because the diagonals of A do not contain the same number of elements.

After that, it is possible to define a vector, say `nzval`, that stores the columns of this matrix sequentially:

```
double nzval[] = { 0.0, 4.0, 1.0, -1.0, 3.0, 6.0, 2.0, 0.0,
                   2.0, 1.0, 3.0, 0.0, 4.0, -2.0, 1.0, 4.0,
                   -1.0, 3.0, 2.0, 0.0, 1.0, 5.0, 0.0, 0.0 };
```

Finally, one of the constructors described above should be used to declare *A* as an `ARbdNonSymMatrix` object:

1. Using the long constructor

```
ARbdNonSymMatrix<double> A(6, 2, 1, nzval);
```

2. Using the default constructor

```
ARbdNonSymMatrix<double> A;
A.DefineMatrix(6, 2, 1, nzval);
```

ARdsNonSymMatrix

ardnsmat.h

Declaration

```
template<class TYPE> class ARdsNonSymMatrix
```

Description

This class defines a real or complex nonsymmetric dense matrix.

Warning: The LAPACK library is called by two member functions of this class, `FactorA` and `FactorAsI`, to perform a matrix decomposition. These functions are used by ARPACK++ to solve generalized eigenvalue problems and also standard problems in shift and invert mode. Since many nonzero elements are generated during the LU factorization, memory availability must be taken in account when using these two member functions.

Default constructor

```
ARdsNonSymMatrix()
```

Long constructor (square matrix)

```
ARdsNonSymMatrix(int n, TYPE* nzval)
```

Long constructor (rectangular matrix)

```
ARdsNonSymMatrix(int m, int n, TYPE* nzval)
```

Long constructor (square or rectangular matrix stored in a file)

```
ARdsNonSymMatrix(char* name, int blksize = 0)
```

Constructor parameters

m	number of rows.
`n`	number of columns.
`nzval`	pointer to an array that contains all the matrix elements ordered by columns. Within each column, `nzval` components must be supplied in ascending order of row indices.
`name`	name of the file that stores the matrix.
`blksize`	size of the block that is to be read at once when the matrix does not fit into memory (see the *Out-of-core matrices* section below). If `blksize` is set to zero (the default option) all of the matrix elements are read.

Member functions

`int nrows()`
: returns `n`.

`int ncols()`
: returns `n`.

`void FactorA()`
: Performs the LU factorization of A, a matrix that belongs to this class. This function cannot be used with out-of-core matrices.

`void FactorAsI(TYPE sigma)`
: Performs the LU decomposition of $(A - \sigma I)$, where I is the identity matrix. This function cannot be used with out-of-core matrices.

`bool IsFactored()`
: indicates if the LU decomposition of matrix A or $(A - \sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow Av$.

`void MultMtv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow A^T v$.

`void MultMtMv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow A^T Av$.

`void MultMMtv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow AA^T v$.

`void MultOMMtOv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow \overline{A}v$, where

$$\overline{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

`void MultInvv(TYPE* v, TYPE* w)`
: Solves $LUw = v$. `FactorA` or `FactorAsI` must be called prior to using this function.

`void DefineMatrix(int n, TYPE* nzval)`
: Stores matrix data when matrix is square and the default constructor is being used.

`void DefineMatrix(int m, int n, TYPE* nzval)`
: Stores matrix data when matrix is rectangular and the default constructor is being used.

`void DefineMatrix(char* name, int blksize = 0)`
: Stores matrix data when matrix is stored in a file.

Out-of-core matrices

The `ARdsNonSymMatrix` class can also be used to handle dense out-of-core matrices, i.e. matrices that are too big to fit into memory. In this case, only `blksize` rows or columns are read at once, so the matrix is entirely reread from disk every time a matrix-vector product function is called and the product is performed one block after another.

Unfortunately, `FactorA` and `FactorAsI` cannot be used with out-of-core matrices.

Reading matrices from files

This class permits the user to declare a matrix from a file. However, this file must follow a very strict format. A brief description of this format is given below.

1. The user can include comments in the file, but these comments must precede everything else. Comment lines must begin with a % sign.
2. Immediately after the comments, there must be a line containing the number of rows and columns of the matrix.
3. Finally, all of the matrix elements must be stored one per line (if the matrix is complex the real part comes first in the line).

In most cases, the matrix elements are assumed to be stored following a column-major order (one column at a time). However, there is one exception. If the matrix is out-of-core and $m > n$, the matrix elements must be stored row-wise. This is required because only `blksize` rows are read at once and the elements of these rows must be stored contiguously in the file.

Example

To store matrix

$$A = \begin{bmatrix} 6 & 1 & -1 & 0 \\ -2 & 4 & 2 & 4 \\ 1 & 3 & 5 & 2 \\ 3 & 0 & 1 & 4 \end{bmatrix}.$$

One can define a vector, say `nzval`, that stores the columns of this matrix sequentially:

```
double nzval[] = { 6.0, -2.0, 1.0, 3.0, 1.0, 4.0, 3.0, 0.0,
                  -1.0, 2.0, 5.0, 1.0, 0.0, 4.0, 2.0, 4.0 };
```

As an alternative, the matrix can also be stored in a file. One example of such a file is given below:

```
% Comment lines (lines that start with an % sign) are allowed,
% but they must precede everything else in the file.
% The line below contains the number of rows and columns of the matrix.
4 4
6.0
-2.0
1.0
3.0
1.0
4.0
3.0
0.0
-1.0
2.0
5.0
1.0
0.0
4.0
2.0
4.0
```

Once the matrix data is generated (and perhaps stored in a file), one of the constructors described above can be used to declare *A* as an `ARdsNonSymMatrix` object:

1. Using the long constructor

```
ARdsNonSymMatrix<double> A(4, nzval);

ARdsNonSymMatrix<double> B("matrix.dat");
```

2. Using the default constructor

```
ARdsNonSymMatrix<double> A;
A.DefineMatrix(4, nzval);

ARdsNonSymMatrix<double> B;
B.DefineMatrix("matrix.dat");
```

ARluNonSymMatrix

arlnsmat.h

Declaration

```
template<class TYPE> class ARluNonSymMatrix
```

Description

This class defines a real or complex nonsymmetric matrix in compressed sparse column (CSC) format.

Warning: The SuperLU library is called by `FactorA` and `FactorAsI` member functions to perform a matrix decomposition. These functions are used by ARPACK++ to solve generalized eigenvalue problems and standard problems in shift and invert mode. A column reordering can be done before the factorization, as described below. Since some fill-in can occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

```
ARluNonSymMatrix()
```

Long constructor (square matrix)

```
ARluNonSymMatrix( int n, int nnz, TYPE* nzval, int* irow, int* pcol,
                  double thresh = 0.1, int order = 1, bool check = true)
```

Long constructor (rectangular matrix)

```
ARluNonSymMatrix(int m, int n, int nnz, TYPE* nzval, int* irow, int* pcol)
```

Long constructor (Harwell-Boeing file)

```
ARluNonSymMatrix( char* name, double thresh = 0.1,
                  int order = 1, bool check = true)
```

Constructor parameters

`m`	number of rows.
`n`	number of columns.
`nnz`	number of nonzero elements.
`nzval`	pointer to an array of nonzero elements in matrix. Within each column, `nzval` components must be supplied in ascending order of row indices.
`irow`	pointer to an array of row indices of the nonzeros.
`pcol`	pointer to an array of pointers to the beginning of columns in `nzval` and `irow`. This array must have `n`+1 elements and the last element must be `nnz`.
`name`	name of the file that stores the matrix in Harwell-Boeing format.
`thresh`	relative pivot tolerance used during the matrix factorization. At step i of the Gaussian elimination process, a_{ii} is used as pivot if $\lvert a_{ii}\rvert \geq \texttt{thresh} \cdot \max_{j \geq i} \lvert a_{ji}\rvert$. No pivoting will be done if `thresh = 0`, while `thresh = 1` corresponds to partial pivoting.
`order`	integer variable that indicates which column permutation should be performed prior to the decomposition of A or $(A - \sigma I)$. If `order = 0`, the original column order will be preserved. If `order = 1`, the minimum degree ordering on the structure of $A^T A$ is used. Finally, `order = 2` means that the minimum degree ordering should be applied on the structure of $A^T + A$.

`check` boolean variable that indicates if matrix data is to be checked for inconsistencies. When `check = true`, ARPACK++ checks if `pcol` is in ascending order and if `irow` components are in order and within bounds.

Member functions

`int nrows()`

returns `m`.

`int ncols()`

returns `n`.

`int nzeros()`

returns `nnz`.

`void FactorA()`

Performs the LU factorization of A, a matrix that belongs to this class. A must be square. This function can only be used if the SuperLU library was previously installed.

`void FactorAsI(TYPE sigma)`

Performs the LU decomposition of $(A-\sigma I)$, where I is the identity matrix. A must be square. This function also calls SuperLU routines.

`bool IsFactored()`

indicates if the LU decomposition of matrix A or $(A-\sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultMtv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow A^T v$.

`void MultMtMv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow A^T Av$.

`void MultMMtv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow AA^T v$.

`void MultOMMtOv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow \overline{A}v$, where

$$\overline{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

`void MultInvv(TYPE* v, TYPE* w)`

Solves $LUw = v$. Matrix A must be square. `FactorA` or `FactorAsI` must be called prior to using this function.

```
void DefineMatrix( int n, int nnz, TYPE* nzval, int* irow, int* pcol,
                   double thresh = 0.1, int order = 1, bool check = true)
```

Stores matrix data when matrix is square and the default constructor is being used.

```
void DefineMatrix( int m, int n, int nnz, TYPE* nzval,
                   int* irow, int* pcol)
```

Stores matrix data when matrix is rectangular and the default constructor is being used.

Example

To store matrix

$$A = \begin{bmatrix} -1 & 0 & 2 & 5 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & -4 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix}.$$

in CSC format it is necessary to define the following vectors

```
double nzval[] = { -1.0, 1.0, 3.0, 2.0, -4.0, 5.0, 2.0 };
int irow[]     = { 1, 4, 2, 1, 3, 1, 4 };
int pcol[]     = { 0, 2, 3, 5, 7 };
```

After that, one of the constructors described above should be used to declare *A* as an `ARluNonSymMatrix` object:

1. Using the long constructor

```
ARluNonSymMatrix<double> A(4, 7, nzval, irow, pcol);
```

2. Using the default constructor

```
ARluNonSymMatrix<double> A;
A.DefineMatrix(4, 7, nzval, irow, pcol);
```

After declaring *A*, to solve a linear system, say $Ax = y$, where x and y are dense vectors with compatible dimensions, the matrix must be factored and then `MultInvv` called:

```
A.FactorA();
A.MultInvv(y, x);
```

ARumNonSymMatrix

arunsmat.h

Declaration

```
template<class TYPE> class ARumNonSymMatrix
```

Description

This class defines a real or complex nonsymmetric matrix in compressed sparse column (CSC) format.

Warning: UMFPACK library must be available if `FactorA` and `FactorAsI` member functions are to be used, because both functions perform a sparse LU decomposition. These functions are called by ARPACK++ to solve generalized eigenvalue problems and standard problems in shift and invert mode. Due to fill-ins that occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

```
ARumNonSymMatrix()
```

Long constructor (square matrix)

```
ARumNonSymMatrix( int n, int nnz, TYPE* nzval, int* irow, int* pcol,
                  double thresh = 0.1, int fillin = 9, bool simest = false,
                  bool redcbl = true, bool check = true)
```

Long constructor (rectangular matrix)

```
ARumNonSymMatrix( int m, int n, int nnz, TYPE* nzval,
                  int* irow, int* pcol)
```

Long constructor (Harwell-Boeing file)

```
ARumNonSymMatrix( char* name, double thresh = 0.1, int fillin = 9,
                  bool simest = false, bool redcbl = true,
                  bool check = true)
```

Constructor parameters

`m`	number of rows (if matrix is rectangular).
`n`	number of columns.
`nnz`	number of nonzero elements.
`nzval`	pointer to an array of nonzero elements in matrix. Within each column, `nzval` components must be supplied in ascending order of row indices.
`irow`	pointer to an array of row indices of the nonzeros.
`pcol`	pointer to an array of pointers to the beginning of columns in `nzval` and `irow`. Such array must have `n`+1 elements and the last element must be `nnz`.
`name`	name of the file that stores the matrix in Harwell-Boeing format.
`thresh`	relative pivot tolerance used during the matrix factorization. At step i of the Gaussian elimination process, a_{ii} is used as pivot if $\lvert a_{ii}\rvert \geq \texttt{thresh} \cdot \max_{j \geq i} \lvert a_{ji}\rvert$. No pivoting will be done if `thresh = 0`, while `thresh = 1` corresponds to partial pivoting.
`fillin`	expected growth in matrix elements due to factorization. `FactorA` and `FactorAsI` functions will reserve `fillin*nnz` memory positions for fill-ins occurred during LU decomposition.

`simest` boolean variable that indicates if the matrix has nearly symmetric nonzero pattern.

`redcbl` boolean variable that indicates whether or not to attempt a permutation to block triangular form. When `redcbl` is set to `true`, the permutation is attempted.

`check` boolean variable that indicates if matrix data is to be checked for inconsistencies. When `check = true`, ARPACK++ checks if `pcol` is in ascending order and if `irow` components are in order and within bounds.

Member functions

`int nrows()`
returns `m`.

`int ncols()`
returns `n`.

`int nzeros()`
returns `nnz`.

`int FillFact()`
returns `fillin`.

`bool IsSymmetric()`
returns `simest`.

`void FactorA()`
Performs the LU factorization of A, a matrix that belongs to this class. A must be square. This function can only be used if the UMFPACK library was previously installed.

`void FactorAsI(TYPE sigma)`
Performs the LU decomposition of $(A - \sigma I)$, where I is the identity matrix. A must be square. This function also calls some UMFPACK routines.

`bool IsFactored()`
indicates if the LU decomposition of matrix A or $(A - \sigma I)$ is available.

`void MultMv(TYPE* v, TYPE* w)`
Calculates $w \leftarrow Av$.

`void MultMtv(TYPE* v, TYPE* w);`
Calculates $w \leftarrow A^T v$.

`void MultMtMv(TYPE* v, TYPE* w);`
Calculates $w \leftarrow A^T Av$.

`void MultMMtv(TYPE* v, TYPE* w);`
Calculates $w \leftarrow AA^T v$.

```
void MultOMMtOv(TYPE* v, TYPE* w)
```

Calculates $w \leftarrow \overline{A}v$, where

$$\overline{A} = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

```
void MultInvv(TYPE* v, TYPE* w)
```

Solves $LUw = v$. Matrix A must be square. `FactorA` or `FactorAsI` must be called prior to using this function.

```
void DefineMatrix( int n, int nnz, TYPE* nzval, int* irow,
                   int* pcol, double thresh = 0.1, int fillin = 9,
                   bool simest = false, bool redcbl = true,
                   bool check = true)
```

Stores matrix data when matrix is square and the default constructor is being used.

```
void DefineMatrix( int m, int n, int nnz, TYPE* nzval,
                   int* irow, int* pcol)
```

Stores matrix data when matrix is rectangular and the default constructor is being used.

Example

To store matrix

$$A = \begin{bmatrix} 3 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & 4 & 0 \\ 0 & 2 & 5 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

in CSC format it is necessary to define the following vectors

```
double nzval[] = { 3.0, 1.0, 1.0, 2.0, 5.0, -2.0, 4.0, 1.0 };
int  irow[]    = { 1, 4, 2, 3, 3, 1, 2, 4 };
int  pcol[]    = { 0, 2, 4, 5, 7, 8 };
```

After that, one of the constructors described above should be used to declare A as an `ARumNonSymMatrix` object:

1. Using the long constructor

```
ARumNonSymMatrix<double> A(4, 5, 8, nzval, irow, pcol);
```

2. Using the default constructor

```
ARumNonSymMatrix<double> A;
A.DefineMatrix(4, 5, 8, nzval, irow, pcol);
```

Pencils.

ARbdSymPencil

arbspen.h

Declaration

`template<class TYPE> class ARbdSymPencil`

Description

This class defines a real symmetric pencil $Ax = Bx\lambda$, where both matrices are stored in band format. Actually, this class is used internally by `ARluSymGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: LAPACK is called by `FactorAsB` and `MultInvBAv` functions to perform a matrix decomposition. Although this class should only be used to define symmetric pencils, a LU factorization with partial pivoting is used. Since the decomposition usually generates some fill-in, memory availability must be taken in account when using these two functions.

Default constructor

`ARbdSymPencil()`

Long constructor

`ARbdSymPencil(ARbdSymMatrix<TYPE>& A, ARbdSymMatrix<TYPE>& B)`

Constructor parameters

`A, B`	matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`

Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

`bool IsFactored()`

indicates if the LU decomposition of $(A - \sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow B^{-1}Av$. B is automatically factored when this function is called for the first time. This function also overwrites *v* with *Av*, so the user must make a copy of *v* before calling `MultInvBAv` if it contains data that cannot be lost.

`void MultInvAsBv(TYPE* v, TYPE* w)`

Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

```
void DefineMatrices(ARbdSymMatrix<TYPE>& Ap, ARbdSymMatrix<TYPE>& Bp)
```
Stores matrices A and B when default constructor is being used.

Example

Matrices

$$A = \begin{bmatrix} -1 & 2 & 0 & 0 & 0 \\ 2 & 3 & -1 & 0 & 0 \\ 0 & -1 & 4 & 1 & 0 \\ 0 & 0 & 1 & 2 & -3 \\ 0 & 0 & 0 & -3 & 5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 3 & -1 & 0 & 0 & 0 \\ -1 & 4 & 1 & 0 & 0 \\ 0 & 1 & 5 & 2 & 0 \\ 0 & 0 & 2 & -3 & -1 \\ 0 & 0 & 0 & -1 & 4 \end{bmatrix}$$

can be stored in band format using the `ARbdSymMatrix` class, as described below

```
double Anzval[] = { -1.0, 2.0, 3.0, -1.0,
                    4.0, 1.0, 2.0, -3.0, 5.0, 0.0 };
ARbdSymMatrix<double> A(5, 1, Anzval);

double Bnzval[] = { 3.0, -1.0, 4.0, 1.0,
                    5.0, 2.0, -3.0, -1.0, 4.0, 0.0 };
ARbdSymMatrix<double> B(5, 1, Bnzval);
```

After that, to declare the pencil $Ax = Bx\lambda$ as an `ARbdSymPencil` object, the user should use one of the constructors mentioned above:

1. Using the long constructor

```
ARbdSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARbdSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARdsSymPencil

ardspen.h

Declaration

```
template<class TYPE> class ARdsSymPencil
```

Description

This class defines a real symmetric pencil $Ax = Bx\lambda$, where both matrices are dense. Actually, this class is used internally by `ARluSymGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: LAPACK is called by `FactorAsB` and `MultInvBAv` functions to perform a matrix decomposition. Although this class should only be used to define symmetric pencils, a LU factorization with partial pivoting is used. Since the decomposition usually generates some fill-in, memory availability must be taken in account when using these two functions.

Default constructor

```
ARdsSymPencil()
```

Long constructor

```
ARdsSymPencil(ARdsSymMatrix<TYPE>& A, ARdsSymMatrix<TYPE>& B)
```

Constructor parameters

`A, B`	matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`
: Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

`bool IsFactored()`
: indicates if the LU decomposition of $(A - \sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`
: Calculates $w \leftarrow B^{-1}Av$. B is automatically factored when this function is called for the first time. This function also overwrites *v* with *Av*, so the user must make a copy of *v* before calling `MultInvBAv` if it contains data that cannot be lost.

`void MultInvAsBv(TYPE* v, TYPE* w)`
: Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

`void DefineMatrices(ARdsSymMatrix<TYPE>& Ap, ARdsSymMatrix<TYPE>& Bp)`
: Stores matrices *A* and *B* when default constructor is being used.

Example

Matrices

$$A = \begin{bmatrix} 4 & 1 & 3 & -1 & 2 \\ 1 & 5 & 6 & 2 & 0 \\ 3 & 6 & -3 & -1 & 4 \\ -1 & 2 & -1 & 3 & 2 \\ 2 & 0 & 4 & 2 & 5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 6 & -1 & 1 & 0 & 1 \\ -1 & 4 & 1 & 0 & 0 \\ 1 & 1 & 5 & 2 & 0 \\ 0 & 0 & 2 & -3 & -1 \\ 1 & 0 & 0 & -1 & 4 \end{bmatrix}$$

can be defined using the `ARdsSymMatrix` class, as described below

```
double Anzval[] = { 4.0, 1.0, 3.0, -1.0, 2.0, 5.0, 6.0, 2.0,
                    0.0, -3.0, -1.0, 4.0, 3.0, 2.0, 5.0 };
```

```
ARdsSymMatrix<double> A(5, Anzval);

double Bnzval[] = { 6.0, -1.0, 1.0, 0.0, 1.0, 4.0, 1.0, 0.0,
                    0.0, 5.0, 2.0, 0.0, -3.0, -1.0, 4.0 };
ARdsSymMatrix<double> B(5, Bnzval);
```

After that, to declare the pencil $Ax = Bx\lambda$ as an `ARdsSymPencil` object, the user should use one of the constructors mentioned above:

1. Using the long constructor

```
ARdsSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARdsSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARluSymPencil

arlspen.h

Declaration

`template<class TYPE> class ARluSymPencil`

Description

This class defines a real symmetric pencil $Ax = Bx\lambda$, where both matrices are stored in CSC format. Actually, this class is used internally by `ARluSymGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: The SuperLU library is called by two member functions, `FactorAsB` and `MultInvBAv`, to perform a matrix decomposition. Although this class should only be used to define symmetric pencils, a LU factorization is used. A column reordering is usually done before the factorization. Since some fill-in can occur during the elimination phase, memory availability must be taken in account when using these two functions.

Default constructor

`ARluSymPencil()`

Long constructor

`ARluSymPencil(ARluSymMatrix<TYPE>& A, ARluSymMatrix<TYPE>& B)`

Constructor parameters

`A, B`	matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`

Performs the LU decomposition of $(A-\sigma B)$ for a given σ. To use this function, A and B must be declared using the same `uplo` parameter (see the description of `ARluSymMatrix`).

`bool IsFactored()`

indicates if the LU decomposition of $(A-\sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow B^{-1}Av$. B is automatically factored when this function is called for the first time. This function also overwrites *v* with *Av,* so the user must make a copy of *v* before calling `MultInvBAv` if it contains data that cannot be lost.

`void MultInvAsBv(TYPE* v, TYPE* w)`

Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

`void DefineMatrices(ARluSymMatrix<TYPE>& Ap, ARluSymMatrix<TYPE>& Bp)`

Stores matrices *A* and *B* when default constructor is being used.

Example

Matrices

$$A=\begin{bmatrix}-1&0&0&0&2\\0&3&-1&0&0\\0&-1&4&1&2\\0&0&1&2&0\\2&0&2&0&5\end{bmatrix} \text{ and } B=\begin{bmatrix}3&0&0&-1&0\\0&4&1&0&0\\0&1&5&2&0\\-1&0&2&3&-1\\0&0&0&-1&4\end{bmatrix}$$

can be stored in CSC format using the `ARluSymMatrix` class, as described below

```
double Anzval[] = { -1.0, 2.0, 3.0, -1.0, 4.0, 1.0, 2.0, 2.0, 5.0 };
int    Airow[]  = { 1, 5, 2, 3, 3, 4, 5, 4, 5 };
int    Apcol[]  = { 0, 2, 4, 7, 8, 9 };
ARluSymMatrix<double> A(5, 9, Anzval, Airow, Apcol);

double Bnzval[] = { 3.0, -1.0, 4.0, 1.0, 5.0, 2.0, 3.0, -1.0, 4.0 };
int    Birow[]  = { 1, 4, 2, 3, 3, 4, 4, 5, 5 };
int    Bpcol[]  = { 0, 2, 4, 6, 8, 9 };
ARluSymMatrix<double> B(5, 9, Bnzval, Birow, Bpcol);
```

After that, to declare the pencil $Ax = Bx\lambda$ as an `ARluSymPencil` object, the user should use one of the constructors mentioned above:

1. Using the long constructor

```
ARluSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARluSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARumSymPencil

aruspen.h

Declaration

```
template<class TYPE> class ARumSymPencil
```

Description

This class defines a real symmetric pencil $Ax = Bx\lambda$, where both matrices are stored in CSC format. Actually, this class is used internally by `ARluSymGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: The UMFPACK library must be available if `FactorAsB` and `MultInvBAv` member functions are to be used, because both functions perform a sparse matrix decomposition. Although this class should only be used to define symmetric pencils, a LU factorization is used. Moreover, due to fill-ins that occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

```
ARumSymPencil()
```

Long constructor

```
ARumSymPencil(ARumSymMatrix<TYPE>& A, ARumSymMatrix<TYPE>& B)
```

Constructor parameters

`A, B` matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`

Performs the LU decomposition of $(A - \sigma B)$ for a given σ. To use this function, A and B must be declared using the same `uplo` parameter (see the description of `ARumSymMatrix`).

`bool IsFactored()`

indicates if the LU decomposition of $(A - \sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow B^{-1}Av$. B is automatically factored when this function is called for the first time. This function also overwrites v with Av, so the user must make a copy of v before calling `MultInvBAv` if it contains data that cannot be lost.

`void MultInvAsBv(TYPE* v, TYPE* w)`

Solves $LUw = v$, where L and U were generated by `FactorAsB`.

`void DefineMatrices(ARumSymMatrix<TYPE>& Ap, ARumSymMatrix<TYPE>& Bp)`

Stores matrices A and B when default constructor is being used.

Example

Using the same vectors `Anzval`, `Airow`, `Apcol`, `Bnval`, `Birow` and `Bpcol` defined in the example given for the `ARluSymPencil` above, matrices A and B can be easily stored in CRC format using the `ARumSymMatrix` class:

```
ARumSymMatrix<double> A(5, 9, Anzval, Airow, Apcol);
ARumSymMatrix<double> B(5, 9, Bnzval, Birow, Bpcol);
```

After that, to declare the pencil $Ax = Bx\lambda$ as an `ARumSymPencil` object, the user should use one of the constructors mentioned above:

1. Using the long constructor

```
ARumSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARumSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARbdNonSymPencil

arbnspen.h

Declaration

```
template<class TYPE, class FLOAT> class ARbdNonSymPencil
```

Description

This class defines a complex or real nonsymmetric pencil $Ax = Bx\lambda$, where both matrices are stored in band format. Actually, this class is used internally by `ARluNonSymGenEig` and `ARluCompGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: The LAPACK library is called by `FactorAsB` and `MultInvBAv` functions to perform a matrix decomposition. Since the LU decomposition usually generates some fill-in, memory availability must be taken in account when using these two functions.

Default constructor

```
ARbdNonSymPencil()
```

Long constructor

```
ARbdNonSymPencil(ARbdNonSymMatrix<TYPE>& A, ARbdNonSymMatrix<TYPE>& B)
```

Constructor parameters

`A, B` matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

```
void FactorAsB(TYPE sigma)
```

Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

```
void FactorAsB(FLOAT sigmaR, FLOAT sigmaI, char part = 'R')
```

Performs the LU decomposition of $(A - \sigma B)$ for a given $\sigma = \sigma_R + i\sigma_I$. This function should only be used if *A* and *B* are real matrices and the shift is complex. `Part` is a parameter that will be used later by the `MultInvAsBv` function to decide which part of the vector *w* will be discarded when solving the complex linear system $(A - \sigma B)w = v$. If `part = 'R'`, *w* is set to $real\{(A - \sigma B)^{-1}v\}$, while $imag\{(A - \sigma B)^{-1}v\}$ is used if `part = 'I'` (see chapter 3 for the description of the complex shift and invert mode for real nonsymmetric eigenvalue problems).

```
bool IsFactored()
```

indicates if the LU decomposition of $(A - \sigma B)$ is available.

```
void MultAv(TYPE* v, TYPE* w)
```

Calculates $w \leftarrow Av$.

```
void MultBv(TYPE* v, TYPE* w)
```

Calculates $w \leftarrow Bv$.

```
void MultInvBAv(TYPE* v, TYPE* w)
```

Calculates $w \leftarrow B^{-1}Av$. *B* is automatically factored when this function is called for the first time.

```
void MultInvAsBv(TYPE* v, TYPE* w)
```

Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

```
void DefineMatrices(ARbdNonSymMatrix<TYPE>& Ap,
                    ARbdNonSymMatrix<TYPE>& Bp)
```

Stores matrices *A* and *B* when default constructor is being used.

Example

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 2 & 3 & -1 & 0 & 0 \\ 0 & 2 & 4 & 1 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & -1 & -3 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & -1 & 0 & 0 & 0 \\ 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & 5 & 1 & 0 \\ 0 & 0 & 2 & -3 & -1 \\ 0 & 0 & 0 & 3 & 4 \end{bmatrix}$$

Matrices *A* and *B* above can be stored in band format using the `ARbdNonSymMatrix` class as described below

```
double Anzval[] = { 0.0, -1.0, 2.0, 1.0, 3.0, 2.0, -1.0,
                    4.0, 1.0, 1.0, 2.0, -1.0, 0.0, -3.0, 0.0 };
ARbdNonSymMatrix<double> A(5, 1, 1, Anzval);

double Bnzval[] = { 0.0, 3.0, 2.0, -1.0, 4.0, 1.0, 1.0,
                    5.0, 2.0, 1.0, -3.0, 3.0, -1.0, 4.0, 0.0 };
ARbdNonSymMatrix<double> B(5, 1, 1, Bnzval);
```

After that, the user can declare the pencil $Ax = Bx\lambda$ as an `ARbdNonSymPencil` object using one of the constructors mentioned above:

1. Using the long constructor

```
ARbdNonSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARbdNonSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARdsNonSymPencil

ardnspen.h

Declaration

```
template<class TYPE, class FLOAT> class ARdsNonSymPencil
```

Description

This class defines a complex or real nonsymmetric pencil $Ax = Bx\lambda$, where both matrices are dense. Actually, this class is used internally by `ARluNonSymGenEig` and `ARluCompGenEig`, so the user does not need to use it to declare any eigenvalue problem.

Warning: The LAPACK library is called by `FactorAsB` and `MultInvBAv` functions to perform a matrix decomposition. Since the LU decomposition usually generates some fill-in, memory availability must be taken in account when using these two functions.

Default constructor

```
ARdsNonSymPencil()
```

Long constructor

```
ARdsNonSymPencil(ARdsNonSymMatrix<TYPE>& A, ARdsNonSymMatrix<TYPE>& B)
```

Constructor parameters

`A, B` matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`

Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

`void FactorAsB(FLOAT sigmaR, FLOAT sigmaI, char part = 'R')`

Performs the LU decomposition of $(A - \sigma B)$ for a given $\sigma = \sigma_R + i\sigma_I$. This function should only be used if *A* and *B* are real matrices and the shift is complex. `Part` is a parameter that will be used later by the `MultInvAsBv` function to decide which part of the vector *w* will be discarded when solving the complex linear system $(A - \sigma B)w = v$. If `part = 'R'`, *w* is set to $real\{(A - \sigma B)^{-1} v\}$, while $imag\{(A - \sigma B)^{-1} v\}$ is used if `part = 'I'` (see chapter 3 for the description of the complex shift and invert mode for real nonsymmetric eigenvalue problems).

`bool IsFactored()`

indicates if the LU decomposition of $(A - \sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow B^{-1}Av$. *B* is automatically factored when this function is called for the first time.

`void MultInvAsBv(TYPE* v, TYPE* w)`

Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

```
void DefineMatrices(ARdsNonSymMatrix<TYPE>& Ap,
                    ARdsNonSymMatrix<TYPE>& Bp)
```

Stores matrices *A* and *B* when default constructor is being used.

Example

Matrices

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 2 \\ 1 & 5 & 2 & 4 & -1 \\ 3 & 6 & -3 & 1 & 4 \\ -1 & 2 & -1 & 3 & 2 \\ 2 & 0 & 4 & 2 & 5 \end{bmatrix} \text{ and } B = \begin{bmatrix} 6 & -1 & 1 & 0 & 1 \\ -1 & 4 & 1 & 0 & 0 \\ 1 & 1 & 5 & 2 & 0 \\ 0 & 0 & 2 & -3 & -1 \\ 1 & 0 & 0 & -1 & 4 \end{bmatrix}$$

can be defined using the `ARdsNonSymMatrix` class as described below

```
double Anzval[] = { 4.0, 1.0, 3.0, -1.0, 2.0, -1.0, 5.0, 6.0,
                    2.0, 0.0, 0.0, 2.0, -3.0, -1.0, 4.0, -1.0,
                    4.0, 1.0, 3.0, 2.0, 2.0, -1.0, 4.0, 2.0, 5.0 };
ARdsSymMatrix<double> A(5, Anzval);
```

```
double Bnzval[] = { 6.0, -1.0, 1.0, 0.0, 1.0, -1.0, 4.0, 1.0,
                    0.0, 0.0, 1.0, 1.0, 5.0, 2.0, 0.0, 0.0,
                    0.0, 2.0, -3.0, -1.0, 1.0, 0.0, 0.0, -1.0, 4.0 };
ARdsSymMatrix<double> B(5, Bnzval);
```

After that, the user can declare the pencil $Ax = Bx\lambda$ as an `ARdsNonSymPencil` object using one of the constructors mentioned above:

1. Using the long constructor

```
ARdsNonSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARdsNonSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARluNonSymPencil

arlnspen.h

Declaration

```
template<class TYPE, class FLOAT> class ARluNonSymPencil
```

Description

This class defines a real or complex nonsymmetric pencil $Ax = Bx\lambda$, where both matrices are stored in compressed sparse column (CSC) format. Actually, although this function is available as an ARPACK++ class, the user does not need to use it to declare any eigenvalue problem[29].

Warning: The SuperLU library is called by `FactorAsB` and `MultInvBAv` functions to perform a matrix decomposition. A column reordering can be done before the factorization. Due to fill-ins that occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

```
ARluNonSymPencil()
```

Long constructor

```
ARluNonSymPencil(ARluNonSymMatrix<TYPE>& A, ARluNonSymMatrix<TYPE>& B)
```

Constructor parameters

`A, B` matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

```
void FactorAsB(TYPE sigma)
```

Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

[29] This class is used internally by `ARluNonSymGenEig` and `ARluCompGenEig`.

`void FactorAsB(FLOAT sigmaR, FLOAT sigmaI, char part = 'R')`

Performs the LU decomposition of $(A-\sigma B)$ for a given $\sigma = \sigma_R + i\sigma_I$. This function should only be used if *A* and *B* are real matrices and the shift is complex. `Part` is a parameter that will be used later by the `MultInvAsBv` function to decide which part of the vector *w* will be discarded when solving the complex linear system $(A-\sigma B)w = v$. If `part = 'R'`, *w* is set to $real\{(A-\sigma B)^{-1}v\}$, while $imag\{(A-\sigma B)^{-1}v\}$ is used if `part = 'I'` (see chapter 3 for the description of the complex shift and invert mode for real nonsymmetric eigenvalue problems).

`bool IsFactored()`

indicates if the LU decomposition of $(A-\sigma B)$ is available.

`void MultAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`

Calculates $w \leftarrow B^{-1}Av$.

`void MultInvAsBv(TYPE* v, TYPE* w)`

Solves $LUw = v$, where *L* and *U* were generated by `FactorAsB`.

```
void DefineMatrices(ARluNonSymMatrix<TYPE>& Ap,
                    ARluNonSymMatrix<TYPE>& Bp)
```

Stores matrices *A* and *B* when default constructor is being used.

Example

Matrices

$$A = \begin{bmatrix} -1 & 0 & 2 & 5 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & -4 & 0 \\ 1 & 0 & 0 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 4 & 0 & 6 & 0 \\ 0 & 1 & 0 & 0 \\ 6 & 0 & 9 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

can be stored in CRC format using the `ARluNonSymMatrix` class as described below

```
double Anzval[] = { -1.0, 1.0, 3.0, 2.0, -4.0, 5.0, 2.0 };
int Airow[]     = { 1, 4, 2, 1, 3, 1, 4 };
int Apcol[]     = { 0, 2, 3, 5, 7 };
ARluNonSymMatrix<double> A(4, 7, Anzval, Airow, Apcol);

double Bnzval[] = { 4.0, 6.0, 1.0, 6.0, 9.0, 5.0 };
int Birow[]     = { 1, 3, 2, 1, 3, 4 };
int Bpcol[]     = { 0, 2, 3, 5, 6 };
ARluNonSymMatrix<double> B(4, 6, Bnzval, Birow, Bpcol);
```

After that, it is easy to declare the pencil $Ax = Bx\lambda$ as an `ARluNonSymPencil` object using one of the constructors mentioned above:

1. Using the long constructor

```
ARluNonSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARluNonSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

ARumNonSymPencil

arunspen.h

Declaration

```
template<class TYPE> class ARumNonSymPencil
```

Description

This class defines a real or complex nonsymmetric pencil $Ax = Bx\lambda$, where both matrices are stored in compressed sparse column (CSC) format. Actually, although this function is available as an ARPACK++ class, the user does not need to use it to declare any eigenvalue problem[30].

Warning: The UMFPACK library must be available if `FactorAsB` and `MultInvBAv` member functions are to be used, because both functions perform a sparse LU decomposition. Moreover, due to fill-ins that occur during the elimination phase, memory availability must be taken in account when using these functions.

Default constructor

```
ARumNonSymPencil()
```

Long constructor

```
ARumNonSymPencil(ARumNonSymMatrix<TYPE>& A, ARumNonSymMatrix<TYPE>& B);
```

Constructor parameters

`A, B`	matrices that characterize the pencil $Ax = Bx\lambda$.

Member functions

`void FactorAsB(TYPE sigma)`

Performs the LU decomposition of $(A - \sigma B)$ for a given σ.

`void FactorAsB(FLOAT sigmaR, FLOAT sigmaI, char part = 'R')`

Performs the LU decomposition of $(A - \sigma B)$ for a given $\sigma = \sigma_R + i\sigma_I$. This function should only be used if A and B are real matrices and the shift is complex. `part` indicates whether the real (`part = 'R'`) or the imaginary part (`part = 'I'`) of $(A - \sigma B)v$ is to be stored in w when `MultInvAsBv` is used to solve the complex linear system $(A - \sigma B)w = v$ (see chapter 3 for

[30] This class is used internally by `ARluNonSymGenEig` and `ARluCompGenEig`.

the description of the complex shift and invert mode for real nonsymmetric eigenvalue problems).

`bool IsFactored()`
indicates if the LU decomposition of $(A - \sigma B)$ is available.

`bool IsSymmetric()`
indicates if $(A - \sigma B)$ has nearly symmetric nonzero pattern.

`void MultAv(TYPE* v, TYPE* w)`
Calculates $w \leftarrow Av$.

`void MultBv(TYPE* v, TYPE* w)`
Calculates $w \leftarrow Bv$.

`void MultInvBAv(TYPE* v, TYPE* w)`
Calculates $w \leftarrow B^{-1}Av$.

`void MultInvAsBv(TYPE* v, TYPE* w)`
Solves $LUw = v$, where L and U were generated by `FactorAsB`.

```
void DefineMatrices(ARumNonSymMatrix<TYPE>& Ap,
                    ARumNonSymMatrix<TYPE>& Bp)
```
Stores matrices A and B when default constructor is being used.

Example

Using the same vectors `Anzval`, `Airow`, `Apcol`, `Bnval`, `Birow` and `Bpcol`, defined in the example given for the `ARluNonSymPencil` above, matrices A and B can be stored in CRC format using `ARumNonSymMatrix` class:

```
ARumNonSymMatrix<double> A(4, 7, Anzval, Airow, Apcol);
ARumNonSymMatrix<double> B(4, 6, Bnzval, Birow, Bpcol);
```

After that, there are two different ways to declare the pencil $Ax = Bx\lambda$ as an `ARumNonSymPencil` object:

1. Using the long constructor

```
ARumNonSymPencil<double> Pen(A, B);
```

2. Using the default constructor

```
ARumNonSymPencil<double> Pen;
Pen.DefineMatrices(A, B);
```

Available functions

In this section, all available ARPACK++ functions are described. To make the reading easier, functions are grouped according to their use. Most functions are generic, but because there are some functions that can only be used with specific classes, each function name is followed by the classes it applies.

Some examples are included after function definitions to illustrate their use. In almost all these examples, a variable called `prob` is used to represent an ARPACK++ object, i.e., an object that belongs to one of the classes mentioned above.

Functions that store user defined parameters.

When declaring an ARPACK++ class using the default constructor, the user must supply all required problem parameters through function `DefineParameters` as described below. If an spectral transformation is being used, one of the other functions included in this section should also be called to set the computational mode.

DefineParameters

All classes

Declaration (Classes that do not handle matrix information)

```
void DefineParameters( int n, int nev, char* which = "LM",
                       int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                       TYPE* resid = 0, bool ishift = true);
```

Declaration (classes that handle matrix information, standard problems)

```
void DefineParameters( int n, int nev, FOP* const objOP,
                       TypeOPx MultOPx, char* which = "LM",
                       int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                       TYPE* resid = 0, bool ishift = true)
```

Declaration (classes that handle matrix information, generalized problems)

```
void DefineParameters( int n, int nev, FOP* const objOP,
                       TypeOPx MultOPx, FB* const objB,
                       TypeBx MultBx, char* which = "LM",
                       int ncv = 0, FLOAT tol = 0.0, int maxit = 0,
                       TYPE* resid = 0, bool ishift = true)
```

Description

This function set values of some ARPACK++ variables when the default constructor is being used to declare the eigenvalue problem.

Example

To find the four eigenvalues with smallest magnitude of a 100×100 matrix defined by object `OB`, which belongs to class `MatrixOP<double>`, the user can write

```
ARSymStdEig<double, MatrixOP<double> > prob;
prob.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet, ”SM”, 30);
```

SetBucklingMode

Real symmetric generalized classes

Declaration (ARSymGenEig)

```
void SetBucklingMode(FLOAT sigma, FOP* const objOP, TypeOPx MultOPx)
```

Declaration (all other real symmetric generalized classes)

```
void SetBucklingMode(FLOAT sigma)
```

Description

This function turns a real symmetric generalized problem into buckling mode with shift defined by `sigmaR`. If `ARSymGenEig` is the ARPACK++ class being used, `SetBucklingMode` also declares `objOP` as the object and `MultOPx` as the function that performs the matrix-vector product *OPx*.

Example

```
MatrixOP<double> OP; // Defining a symmetric generalized problem.
prob.SetBucklingMode(2.3, &OP, &MatrixOP<double>::MultMv);
```

SetCayleyMode

Real symmetric generalized classes

Declaration (ARSymGenEig)

```
void SetCayleyMode( FLOAT sigmaR, FOP* objOP, TypeOPx MultOPx,
                    FB* objA, TypeOPx MultAx)
```

Declaration (all other real symmetric generalized classes)

```
void SetCayleyMode(FLOAT sigma)
```

Description

This function turns a real symmetric generalized problem into Cayley mode with shift defined by `sigmaR`. `SetCayleyMode` also declares `objOP` as the object and `MultOPx` as the function that performs the matrix-vector product *OPx* when `ARSymGenEig` is the class being used.

Example

```
MatrixOP<double> OP; // Defining a symmetric generalized problem.
MatrixA<double>  A;
prob.SetCayleyMode(2.3, &OP, &MatrixOP<double>::MultOPv,
                   &A, &MatrixA<double>::MultAv);
```

SetComplexShiftMode

Real nonsymmetric generalized classes

Declaration (ARNonSymGenEig)

```
void SetComplexShiftMode( char part, FLOAT sigmaR, FLOAT sigmaI,
                          FOP* objOP, TypeOPx MultOPx,
                          FB* objA, TypeOPx MultAx)
```

Declaration (all other real nonsymmetric generalized classes)

```
void SetComplexShiftMode(char part, FLOAT sigmaR, FLOAT sigmaI)
```

Description

This function turns a real nonsymmetric generalized eigenvalue problem into complex shift-and-invert mode with shift defined by `sigmaR` and `sigmaI`. `objOP` and `MultOPx` are, respectively, the object and the function that handle the matrix-vector product *OPx*. `MultAx` is required to retrieve eigenvalues.

Example

```
MatrixOP<double> OP; // Defining a nonsymmetric generalized problem.
MatrixOP<double> A;  // Also defining the product Ax.
prob.SetComplexShiftMode(2.3, -1.1, &OP, &MatrixOP<double>::MultOPv,
                         &A, &MatrixOP<double>::MultAv);
```

SetRegularMode

All classes

Declaration (ARStdEig, ARSymStdEig, ARNonSymStdEig, ARCompStdEig, ARGenEig, ARSymGenEig, ARNonSymGenEig, ARCompGenEig)

```
void SetRegularMode(FOP* const objOP, TypeOPx MultOPx)
```

Declaration (all other classes)

```
void SetRegularMode()
```

Description

This function turns an eigenvalue problem into regular mode. In some cases, it also declares `objOP` as the object and `MultOPx` as the function that performs the matrix-vector product *OPx*.

It is also possible to use function `NoShift()` to turn a problem into regular mode. In this case, however, no changes can be made to `objOP` and `MultOPx`.

Example

```
MatrixOP<double> OP;
prob.SetRegularMode(&OP, &MatrixOP<double>::MultMv);
```

SetShiftInvertMode

All classes

Declaration (ARStdEig, ARSymStdEig, ARNonSymStdEig, ARCompStdEig, ARGenEig, ARSymGenEig, ARNonSymGenEig, ARCompGenEig)

```
void SetShiftInvertMode(TYPE sigma, FOP* const objOP, TypeOPx MultOPx)
```

Declaration (all other classes)

```
void SetShiftInvertMode(TYPE sigma)
```

Description

This function turns an eigenvalue problem into shift and invert mode. In soma cases, `SetShiftInvertMode` also declares `objOP` as the object and `MultOPx` as the function that performs the matrix-vector product *OPx*.

If the eigenvalue problem is a standard one, function `ChangeShift` may be used instead of `SetShiftInvertMode`, In this case, no changes can be made to `objOP` and `MultOPx`.

Example

```
MatrixOP<double> OP; // Here, MatrixOP defines a real matrix.
prob.SetShiftInvertMode(2.3, &OP, &MatrixOP<double>::MultMv);

CompMatr<double> CP; // In this case, eigenvalue problem is complex.
prob.SetShiftInvertMode(complex<double>(2.3, -1.1), &CP,
                        &CompMatr<double>::MultMv);
```

Functions that detect if output data is ready.

ARPACK++ has some functions that indicate if the desired output data is available. These functions can be used, for example, to write loop structures or to detect errors in parameters.

ArnoldiBasisFound

All classes

Declaration

```
bool ArnoldiBasisFound()
```

Description

This function returns `true` if an Arnoldi basis is available and `false` otherwise.

Example

```
bool basis_found = prob.ArnoldiBasisFound();
if (!basis_found) prob.FindArnoldiBasis;
```

ConvergedEigenvalues

All classes

Declaration

```
int ConvergedEigenvalues()
```

Description

This function returns the number of eigenvalues found so far.

Example

```
int n_eig = prob.ConvergedEigenvalues();
for (int i = 0; i < n_eig; i++) cout << prob.Eigenvalue(i) << endl;
```

EigenvaluesFound

All classes

Declaration

```
bool EigenvaluesFound()
```

Description

This function returns `true` if the requested eigenvalues are available and `false` otherwise.

Example

```
if (prob.EigenvaluesFound()) {
  val_0 = prob.Eigenvalue(0);
}
```

EigenvectorsFound

All classes

Declaration

```
bool EigenvectorsFound()
```

Description

This function returns `true` if the requested eigenvectors are available and `false` otherwise.

Example

```
if (prob.EigenvectorsFound()) {
  elem_00 = prob.Eigenvector(0, 0);
}
```

SchurVectorsFound

Real nonsymmetric and complex classes

Declaration

```
bool SchurVectorsFound()
```

Description

This function returns `true` if the requested Schur vectors are available and `false` otherwise.

Example

```
if (prob.SchurVectorsFound()) {
  elem_00 = prob.SchurVector(0, 0);
}
```

Functions that provide access to internal variables values.

These functions provide access to all the ARPACK++ parameters, allowing the user to know which options are in effect.

ParametersDefined

All classes

Declaration

```
bool ParametersDefined()
```

Description

This function returns `true` if all internal variables and arrays were defined and `false` otherwise.

Example

```
if (!prob.ParametersDefined()) {
  prob.DefineParameters(100, 4, &OP, &MatrixOP<double>::MultVet);
}
```

GetAutoShift

All classes

Declaration

```
bool GetAutoShift()
```

Description

This function returns `true` if exact shifts are being used by ARPACK++ to restart the Arnoldi method and `false` if the shifts are being supplied by the user.

Example

```
bool exact_shifts = prob.GetAutoShift();
```

GetMaxit

All classes

Declaration

```
int GetMaxit()
```

Description

This function returns the maximum number of Arnoldi update iterations allowed.

Example

```
int maxit = prob.GetMaxit();
```

GetMode

All classes

Declaration

```
int GetMode()
```

Description

This function returns the computational mode used as described in the following table.

value	mode
1	regular mode (standard problems)
2	regular inverse mode (generalized problems)
3	shift and invert mode. For real nonsymmetric generalized problems, this option can also mean that a complex shift is being used but, in this case, $OP = real\{(A - \sigma B)^{-1})\}$.
4	buckling mode (real symmetric generalized problems) or shift and invert mode with $OP = imag\{(A - \sigma B)^{-1})\}$ (real nonsymmetric problems)
5	Cayley mode (real symmetric generalized problems).

Example

```
if (prob.GetMode()==3) {
  cout << ”shift and invert mode is being used” << endl;
}
```

GetIter

All classes

Declaration

```
int GetIter()
```

Description

This function returns the number of Arnoldi update iterations actually taken by arpack++ to solve the eigenvalue problem.

Example

```
int iter = prob.GetIter();
```

GetN

All classes

Declaration

```
int GetN()
```

Description

This function returns the dimension of the problem.

Example

```
cout << ”First eigenvector” << endl;
for (i=0; i<prob.GetN(); i++) cout << prob.Eigenvector(0, i) << endl;
```

GetNcv

All classes

Declaration

```
int GetNcv()
```

Description

This function returns the number of Arnoldi vectors generated at each iteration (see the description of `ncv`).

Example

```
int ncv = prob.GetNcv();
```

GetNev

All classes

Declaration

```
int GetNev()
```

Description

This function returns the number of required eigenvalues. The number of eigenvalues actually found, however, is given by function `ConvergedEigenvalues`.

Example

```
if (prob.GetNev() != prob.ConvergedEigenvalues()) {
  cout << ”only ” << prob.ConvergedEigenvalues() << ” of ”;
  cout << prob.GetNev() << ” eigenvalues were actually found \n”;
}
```

GetShift

All classes

Declaration

```
TYPE GetShift()
```

Description

This function returns the shift `sigma` used to define a spectral transformation. If the problem is real and nonsymmetric, this function returns only the real part of the shift (the imaginary par is given by `GetShiftImag()`). If the problem is being solved in standard mode, `GetShift()` will return 0.0. To avoid any confusion in this case, the user should call function `GetMode()` before calling `GetShift()`.

Examples

```
if (prob.GetMode() > 2)
  double sigma = prob.GetShift();    // real symmetric problems and
                                     // nonsymmetric standard problems.
if (prob.GetMode() > 2)
  complex<double> sigma = prob.GetShift(); // complex problems.
```

GetShiftImag

Real nonsymmetric generalized classes

Declaration

```
FLOAT GetShiftImag()
```

Description

This function returns the imaginary part of the shift when the shift and invert mode is being used to solve real nonsymmetric problems.

Example

```
if (prob.GetMode() > 2) {
  double sigmaR = prob.GetShift();
  double sigmaI = prob.GetShiftImag();
}
```

GetTol

All classes

Declaration

```
FLOAT GetTol()
```

Description

This function returns the stopping criterion used to find eigenvalues. In other words, it returns the relative accuracy of Ritz values.

Example

```
double tol = prob.GetTol();
```

GetWhich

All classes

Declaration

```
char* GetWhich()
```

Description

This function returns which of the Ritz values were required (see the description of `which`).

Example

```
char* which = prob.GetWhich();
```

Functions that allow changes in problem parameters.

The user can change almost all ARPACK++ parameters after their definition by using the following functions.

ChangeMaxit

All classes

Declaration

```
void ChangeMaxit(int maxit)
```

Description

This function changes the maximum number of Arnoldi update iterations allowed to the value given by `maxit`.

Example

```
prob.ChangeMaxit(1000);
```

ChangeMultBx

All generalized classes that handle matrix information

Declaration

```
void ChangeMultBx(FB* const objB, TypeBx MultBx)
```

Description

This function changes the matrix-vector function that performs the product *Bx* (see the description of `MultBx`).

Example

Supposing that function `MultMv` is a public member of class `MatrixB<double>`, it is possible to declare `MultMv` as the function that performs the product *Bx* by using the following commands:

```
MatrixB<double> B;
prob.ChangeMultBx(&B, &MatrixB<double>::MultMv);
```

ChangeMultOPx

All classes that handle matrix information

Declaration

```
void ChangeMultOPx(FOP* const objOP, TypeOPx MultOPx)
```

Description

This function changes the matrix-vector function that performs the product *OPx* (see the description of `MultOPx`).

Example

Supposing that `MatrixOP<double>` is a class that contains the function `MultMv` as a public member, the following commands show how to declare `MultMv` as the function that performs the product *OPx*.

```
MatrixOP<double> OP;
prob.ChangeMultOPx(&OP, &MatrixOP<double>::MultMv);
```

ChangeNcv

All classes

Declaration

```
void ChangeNcv(int ncv)
```

Description

This function changes the number of Arnoldi vectors generated at each iteration to the value given by `ncv`.

Example

```
prob.ChangeNcv(2*prob.GetNev()+1);
```

ChangeNev

All classes

Declaration

```
void ChangeNev(int nev)
```

Description

This function changes the number of eigenvalues to be computed to the value given by `nev`.

Example

```
prob.ChangeNev(12);
```

ChangeShift

All classes

Declaration (all problems)

```
void ChangeShift(TYPE sigmaR)
```

Declaration (real nonsymmetric generalized problems, complex shift)

```
void ChangeShift(FLOAT sigmaR, FLOAT sigmaI = 0)
```

Description

This function turns the problem to shift-and-invert mode with shift defined by `sigmaR`. Real nonsymmetric generalized problems may have a complex shift defined by `sigmaR` (real part) and `sigmaI` (imaginary part).

Examples

```
prob.ChangeShift(1.0);        // real symmetric and nonsymmetric problems.
prob.ChangeShift(1.0, -2.0);                // real nonsymmetric problems.
prob.ChangeShift(complex<double>(1.0, -2.0));         // complex problems.
```

ChangeTol

All classes

Declaration

```
void ChangeTol(FLOAT tol)
```

Description

This function changes the stopping criterion to the value given by `tol`. If `tol` is zero, the machine precision is used as the stopping criterion.

Example

```
prob.ChangeTol(1.0E-12);
```

ChangeWhich

All classes

Declaration

```
void ChangeWhich(char* which)
```

Description

This function changes the part of spectrum that is sought according to the value given by `which`.(see the description of variable `which`).

Example

```
prob.ChangeWhich("SM"); // Seeking eigenvalues with smallest magnitude.
```

NoShift

All classes

Declaration

```
void NoShift()
```

Description

This function turns the problem into regular mode.

Example

```
prob.NoShift();
```

InvertAutoShift

All classes

Declaration

```
void InvertAutoShift()
```

Description

This function changes the shift selection strategy used by ARPACK++ to implicitly restart the Arnoldi method (see the description of `AutoShift`).

Example

```
if (!prob.GetAutoShift()) // if shifts are to be defined by the user
  prob.InvertAutoShift(); // change to the exact shifts strategy.
```

Trace functions.

As in the FORTRAN code, ARPACK++ allows the user to trace the progress of the computation as it proceeds. Various levels of output may be specified from no output to voluminous printing. The following two functions reproduce the functionality of all ARPACK debugging statements.

NoTrace

All classes

Declaration

```
void NoTrace()
```

Description

This function turns off the trace mode.

Example

```
prob.NoTrace();
```

Trace

All classes

Declaration

```
void Trace(int digit = -5, int getv0 = 0, int aupd = 1,
           int aup2 = 0, int aitr = 0, int eigh = 0,
           int apps = 0, int gets = 0, int eupd = 0)
```

Description

This function turns on the trace mode, so some output is generated by ARPACK functions to reflect the progress of the Arnoldi process.

`digit` specifies the number of decimal digits and the width of the output lines. A positive value indicates that 132 columns will be used during output and a negative value specifies that eighty columns are to be used.

Each remaining parameter represents the volume of output generated by the ARPACK function with a similar name. For example, `aitr` indicates the level of output requested for one of the subroutines `ssaitr`, `dsaitr`, `snaitr`, `dnaitr`, `cnaitr` or `znaitr`[31]. The volume of output increases with the value of each parameter. A zero value means that no output will be generated.

If no parameter is passed to `Trace`, only `aupd` will be set to a value greater than zero. In this case, ARPACK will print the total number of iterations taken, the number of converged eigenvalues, the final Ritz values and corresponding Ritz estimates, and various timing statistics in the standard `cout` stream.

A complete description of all the above parameters can be found in the `debug.doc` file that is distributed along with the ARPACK FORTRAN package (in the `documents` directory).

Example

Adding the command

```
prob.Trace();
```

to the beginning of a program (after the problem declaration but before trying to find eigenvalues and eigenvectors) causes ARPACK to display the following output in the `cout` stream[32]:

```
=========================================
= Symmetric implicit Arnoldi update code =
= Version Number: 2.1                    =
= Version Date:   11/15/95               =
=========================================
= Summary of timing statistics           =
=========================================
Total number update iterations             =     8
Total number of OP*x operations            =   125
Total number of B*x operations             =     0
Total number of reorthogonalization steps  =   125
Total number of iterative refinement steps =     0
Total number of restart steps              =     0
Total time in user OP*x operation          =     0.02000
Total time in user B*x operation           =     0.00000
Total time in Arnoldi update routine       =     0.21002
Total time in ssaup2 routine               =     0.19002
Total time in basic Arnoldi iteration loop =     0.11001
Total time in reorthogonalization phase    =     0.07001
Total time in (re)start vector generation  =     0.00000
Total time in trid eigenvalue subproblem   =     0.04000
Total time in getting the shifts           =     0.00000
Total time in applying the shifts          =     0.04000
Total time in convergence testing          =     0.00000
```

[31] In the naming convention adopted by ARPACK FORTRAN subroutines, the first two letters denote the combination of problem type (symmetric or nonsymmetric), data type (real or complex) and precision (single or double) used. As all the names have six letters, only the last four letters are really employed to describe what is really done by each routine. See the ARPACK manual for further information about the FORTRAN functions.

[32] Actually, the output shown here corresponds to a real symmetric problem. Real nonsymmetric an complex problems generate a slightly different output.

Functions that permit step-by-step execution of ARPACK.

This set of functions implements the so called *reverse communication interface,* a major feature of the original ARPACK FORTRAN code that was preserved in the c++ version. With this interface, the user can solve eigenvalue problems without passing information about matrix-vector products to ARPACK++ classes constructors. And even with ARPACK++ classes that handle matrix information these functions allow the user to interfere in the process of finding an Arnoldi basis.

GetIdo

All classes

Declaration

`int GetIdo()`

Description

This function indicates the type of operation the user must perform between two successive calls to function `TakeStep()`.

Return value	**action the user must perform**
-1	compute $y \leftarrow OPx$, where `GetVector()` gives a pointer to vector *x*, and `PutVector()` indicates where to store *y*.
1	compute one of $y \leftarrow OPx$ (standard problems) or $y \leftarrow OP.B.x$ (generalized problems). `GetVector()` gives a pointer to vector *x* and `PutVector()` indicates where to store *y*. When solving generalized problems, a pointer to the product *Bx* is also available by using `GetProd()`.
2	compute $y \leftarrow Bx$, where `GetVector()` gives a pointer to vector *x*, and `PutVector()` indicates where to store *y*.
3	compute shifts for implicit restarting of the Arnoldi method. `PutVector()` indicates where to store them.

Examples

The following example shows how to find eigenvectors for a standard eigenvalue problem, supposing that `prob` is an object of any ARPACK++ standard class and `OP` belongs to a class that includes `MultMv` as a member function. `MultMv` must perform matrix-vector product *OPx* (see the description of `MultOPx`).

```
while (!prob.ArnoldiBasisFound()) {
  prob.TakeStep();
  if ((prob.GetIdo() == 1)||(prob.GetIdo() == -1)) {
    // Performing matrix-vector multiplication.
    OP.MultMv(prob.GetVector(), prob.PutVector());
  }
}
prob.FindEigenvectors();
```

Next example shows how to find eigenvectors of a generalized eigenvalue problem[33]. Generally, `B.MultMv` is supposed to be a function that performs the matrix-vector product *Bx* (see the description of `MultBx`).

```
while (!prob.ArnoldiBasisFound()) {
  prob.TakeStep();
  switch (prob.GetIdo()) {
    case -1: // Performing y <- OP*B*x for the first time.
      B.MultMx(prob.GetVector(), temp);
      OP.MultMv(temp, prob.PutVector());
      break;
    case  1: // Performing y <- OP*B*x. B*x is already available.
      OP.MultMv(prob.GetProd(), prob.PutVector());
      break;
    case  2: // Performing y <- B*x.
      B.MultMv(prob.GetVector(), prob.PutVector());
  }
}
prob.FindEigenvectors();
```

GetNp

All classes

Declaration

```
int GetNp()
```

Description

This function returns the number of shifts that must be supplied after a call to `TakeStep()` when shifts for implicit restarting of the Arnoldi method are being provided by the user (See the description of `AutoShift`).

Example

```
prob.TakeStep();
if (prob.GetIdo() == 3) {
  np = prob.GetNp();
  ... // Define np shifts.
}
```

[33] Except for real symmetric problems in Cayley mode.

GetProd

All classes

Declaration

```
TYPE* GetProd()
```

Description

When `GetIdo` returns 1 and the user must perform a product in the form $y \leftarrow OP.B.x$, this function indicates where Bx is stored.

Example

See the generalized example that follows the description of `GetIdo`.

GetVector

All classes

Declaration

```
TYPE* GetVector()
```

Description

When `GetIdo` returns -1, 1 or 2 and the user must perform a product in the form $y \leftarrow Mx$, this function indicates where x is stored. When `GetIdo` returns 3, this function indicates where the eigenvalues of the current Hessenberg matrix are located.

Example

See the examples that follow the description of `GetIdo`.

GetVectorImag

Real nonsymmetric classes

Declaration

```
FLOAT* GetVectorImag()
```

Description

When `GetIdo` returns 3, this function indicates where the imaginary part of the eigenvalues of the current Hessenberg matrix are located.

PutVector

All classes

Declaration

```
TYPE* PutVector()
```

Description

When `GetIdo` returns -1, 1 or 2 and the user must perform a product in the form $y \leftarrow Mx$, this function indicates where to store y. When `GetIdo` returns 3, this function indicates where to store the shifts.

Example

See the examples that follow the description of `GetIdo`.

`TakeStep`

All classes

Declaration

```
int TakeStep()
```

Description

This function performs all calculations required by the Arnoldi method between two successive matrix-vector products. It corresponds to subroutines `__aupd` of the original FORTRAN code. `TakeStep` returns the same value obtained by calling function `GetIdo`.

Example

See the examples that follow the description of `GetIdo`.

Functions that perform all calculations in one step.

Functions included in this group should be used to compel ARPACK++ to find eigenvalues, eigenvectors, Schur vectors or an Arnoldi basis for the current problem. Output data is stored into ARPACK++ internal data structure and can be retrieved by some functions that will be described later in this section.

If one of the *reverse communication* classes is being used, these functions should only be called after convergence of the Arnoldi process is attained.

`FindArnoldiBasis`

All classes

Declaration

```
int FindArnoldiBasis()
```

Description

This function determines an Arnoldi basis related to the given problem. It returns the number of “converged” eigenvalues obtained for the transformed problem (as described in chapter 3).

Example

```
prob.FindArnoldiBasis();
cout << “first Arnoldi basis vector:” << endl;
for (int i = 0; i < prob.GetN(); i++) {
  cout << prob.ArnoldiBasisVector(0, i) << endl;
}
```

FindEigenvalues

All classes

Declaration

```
int FindEigenvalues()
```

Description

This function determines `nev` approximated eigenvalues of the given problem. It returns the number of “converged” eigenvalues obtained.

Example

```
int nconv = prob.FindEigenvalues();
cout << “Eigenvalues:” << endl;
for (int i = 0; i < nconv; i++) {
  cout << prob.Eigenvalue(i) << endl;
}
```

FindEigenvectors

All classes

Declaration

```
int FindEigenvectors(bool ischur = false)
```

Description

This function determines `nev` approximated eigenvectors of the given eigenvalue problem. Optionally, it also determines `nev` Schur vectors that span the desired invariant subspace. The returned value corresponds to the number of “converged” eigenvectors obtained.

Example

```
int nconv = prob.FindEigenvectors();
for (int k = 0; k < nconv; k++) {
  cout << “Eigenvector “ << k << “ :” << endl;
```

```
    for (int i = 0; i < prob.GetN(); i++) {
      cout << prob.Eigenvector(k, i) << endl;
    }
  }
```

FindSchurVectors

Real nonsymmetric and complex classes

Declaration

```
int FindSchurVectors()
```

Description

This function determines `nev` Schur vectors that span the desired invariant subspace. It returns the number of “converged” Schur vectors obtained.

Example

```
int nconv = prob.FindSchurVectors();
for (k = 0; k < nconv; k++) {
  cout << “Schur vector “ << k << “ :” << endl;
  for (i = 0; i < prob.GetN(); i++) {
    cout << prob.Eigenvector(k, i) << endl;
  }
}
```

Functions that perform calculations on user-supplied data structure.

Functions described below should be used to find eigenvalues, eigenvectors, Schur vectors or an Arnoldi basis for the current problem and to store them in a data structure supplied by the user.

To avoid excessive memory consumption, ARPACK++ does not retain a copy of data stored in user-supplied vectors, so no other function can be used to retrieve them.

If one of the *reverse communication* classes is being used, these functions should only be called after convergence of the Arnoldi process is attained.

Eigenvalues

Real symmetric, nonsymmetric and complex classes

Declaration (Real symmetric and complex problems)

```
int Eigenvalues(TYPE EigVal[], bool ivec = false,  bool ischur = false)
```

Declaration (Real nonsymmetric problems)

```
int Eigenvalues( FLOAT EigValR[], FLOAT EigValI[],
                 bool ivec = false, bool ischur = false)
```

Description

This function overrides array `EigVal` with eigenvalues of the problem. It can also determine eigenvectors (if `ivec = true`) and Schur vectors (if `ischur = true`). For real nonsymmetric problems, arrays `EigValR` and `EigValI` are overwritten with the real and imaginary part of eigenvalues, respectively. `Eigenvalues` returns the number of "converged" eigenvalues obtained. Functions `SchurVector`, `RawSchurVectors` and `RawSchurVector` should be used to retrieve Schur vectors.

Arrays `EigVal`, `EigValR` and `EigValI` must be dimensioned to store at least `nev` elements.

Example

```
float EigValR[nev], EigValI[nev];
float EigVec[n*(nev+1)];
int   conv_eig;
conv_eig = prob.Eigenvalues(EigValR, EigValI, true);
prob.Eigenvectors(EigVec);
```

EigenValVectors

Real symmetric, nonsymmetric and complex classes

Declaration (Real symmetric and complex problems)

```
int EigenValVectors(TYPE EigVec[], TYPE EigVal[], bool ischur = false)
```

Declaration (Real nonsymmetric problems)

```
int EigenValVectors( FLOAT EigVec[], FLOAT EigValR[],
                     FLOAT EigValI[], bool ischur = false)
```

Description

This function overrides array `EigVec` sequentially with eigenvectors of the given problem and stores the eigenvalues in `EigVal` (or in the pair {`EigValR`, `EigValI`} if the problem is real and nonsymmetric). If `ischur` is set to true, Schur vectors are also determined. In this case, the user should use functions `SchurVector`, `RawSchurVectors` and `RawSchurVector` to retrieve them.

Arrays `EigVal`, `EigValR` and `EigValI` must be dimensioned to store at least `nev` elements. If the problem is real and nonsymmetric, `EigVec` must have at least `(nev+1)*n` positions, otherwise a vector with `nev*n` elements will be sufficient.

Example

```
double EigValR[nev], EigValI[nev];
double EigVec[(nev+1)*n];
int    conv_eig;
conv_eig = prob.EigenValVectors(EigVec, EigValR, EigValI);
```

Eigenvectors

Real symmetric, nonsymmetric and complex classes

Declaration

```
int Eigenvectors(TYPE EigVec[], bool ischur = false)
```

Description

This function overrides array `EigVec` with eigenvectors of the given problem. It can also calculate Schur vectors if `ischur` is set to `true`. In this case, one of the `SchurVector`, `RawSchurVectors` or `RawSchurVector` functions should be used to retrieve Schur vectors. `Eigenvectors` returns the number of "converged" eigenvectors obtained.

Elements of eigenvectors are stored sequentially, one eigenvector at a time. For real nonsymmetric problems, real and imaginary parts of complex eigenvectors are given as two consecutive vectors, so the user can use them to build the related conjugate pair.

If the problem is real and nonsymmetric, array `EigVec` must be dimensioned to store `(nev+1)*n` elements. For real symmetric and complex problems, a vector with `nev*n` elements will be sufficient.

Example

```
double EigVec[(nev+1)*n];
int    conv_eig;
conv_eig = prob.Eigenvectors(EigVec);
```

Functions that return vector and matrix elements.

After determining eigenvalues, eigenvectors, Schur vectors or an Arnoldi basis, it is possible to retrieve any of their elements using one of the functions listed below.

ArnoldiBasisVector

All classes

Declaration

```
TYPE ArnoldiBasisVector(int i, int j)
```

Description

This function furnishes element j of the i-th Arnoldi basis vector.

Examples

```
float elem_ij = prob.ArnoldiBasisVector(i, j);

complex<float> elem_ij = prob.ArnoldiBasisVector(i, j);
```

See also the example that follows the definition of `FindArnoldiBasis`.

Eigenvalue

Real symmetric, nonsymmetric and complex classes

Declaration (Real symmetric problems)

```
FLOAT Eigenvalue(int i)
```

Declaration (Real nonsymmetric and complex problems)

```
complex<FLOAT> Eigenvalue(int i)
```

Description

This function provides the i-th "converged" eigenvalue.

Examples

```
float eigval = prob.Eigenvalue(i);          // real symmetric problems.

complex<float> eigval = prob.Eigenvalue(i); // real nonsymmetric and
                                            // complex problems.
```

See also the example that follows the definition of `FindEigenvalues`.

EigenvalueImag

Real nonsymmetric classes

Declaration

```
FLOAT EigenvalueImag(int i)
```

Description

This function provides the imaginary part of the i-th "converged" eigenvalue.

Example

```
float eigvalR = prob.EigenvalueReal(i);
float eigvalI = prob.EigenvalueImag(i);
```

EigenvalueReal

Real nonsymmetric classes

Declaration

```
FLOAT EigenvalueReal(int i)
```

Description

This function provides the real part of the i-th "converged" eigenvalue.

Example

```
float eigvalR = prob.EigenvalueReal(i);
float eigvalI = prob.EigenvalueImag(i);
```

Eigenvector

Real symmetric, nonsymmetric and complex classes

Declaration (Real symmetric classes)

```
FLOAT Eigenvector(int i, int j)
```

Declaration (Real nonsymmetric and complex classes)

```
complex<FLOAT> Eigenvector(int i, int j)
```

Description

This function provides element j of the i-th “converged” eigenvector.

Examples

```
float eig_ij = prob.Eigenvector(i,j);        // real symmetric problems.

complex<float> eig_ij=prob.Eigenvector(i,j); // real nonsymmetric
                                             // and complex problems
```

See also the example that follows the definition of `FindEigenvectors`.

EigenvectorImag

Real nonsymmetric classes

Declaration

```
FLOAT EigenvectorImag(int i, int j)
```

Description

This function provides the imaginary part of element j of the i-th “converged” eigenvector.

Example

```
float eig_re, eig_im;
complex<float> eig1, eig2;
eig_re = prob.EigenvectorReal(i, j);
eig_im = prob.EigenvectorImag(i, j);
eig1   = complex<float>(eig_re, eig_im);
eig2   = complex<float>(eig_re, -eig_im);
```

EigenvectorReal

Real nonsymmetric classes

Declaration

```
FLOAT EigenvectorReal(int i, int j)
```

Description

This function provides the real part of element j of the i-th “converged” eigenvector.

Example

See the example that follows the description of `EigenvectorImag`.

SchurVector

Real nonsymmetric and complex classes

Declaration

```
TYPE SchurVector(int i, int j)
```

Description

This function furnishes element j of the i-th Schur vector.

Examples

```
float schur_ij = prob.SchurVector(i, j);
```

```
complex<float> schur_ij = prob.SchurVector(i, j);
```

See also the example that follows the definition of `FindSchurVectors`.

ResidualVector

All classes

Declaration

```
TYPE ResidualVector(int i)
```

Description

This function furnishes element i of the residual vector.

Example

```
prob.FindEigenvalues();
cout << “Residual vector:” << endl;
for (i = 0; i < prob.GetN(); i++) {
  cout << prob.ResidualVector(i) << endl;
}
```

Functions that provide raw access to output data.

If eigenvalues, eigenvectors, Schur vectors or Arnoldi basis vectors are already available in ARPACK++ internal data structure, they can also be referenced by using functions that provide pointers to their first elements.

These functions, which are described below, can be used to efficiently pass ARPACK++ output data to other functions that use them as input parameters.

RawArnoldiBasisVectors

All classes

Declaration

```
TYPE* RawArnoldiBasisVectors()
```

Description

This function provides raw access to Arnoldi basis vectors. The return value is a pointer to a vector that stores all the Arnoldi basis vectors consecutively.

Example

```
// Writing y as a linear combination of some Arnoldi basis vectors.
// Function gemv calculates y = alfa*X*r + beta*y.
// The parameters of gemv are trans, m, n, alpha, X, ldx,
// r, incr, beta, y and incy, in that order.

double* y = new double[prob.GetN()];
gemv('N', prob.GetN(), prob.ConvergedEigenvalues(), 1.0,
     prob.RawArnoldiBasisVectors(), prob.GetN(), r, 1, 0.0, y, 1);
```

RawArnoldiBasisVector

All classes

Declaration

```
TYPE* RawArnoldiBasisVector(int i)
```

Description

This function provides raw access to the i-th Arnoldi basis vector by returning a pointer to its first element.

Example

```
// Using two Arnoldi basis vectors in a saxpy ...
Copy(prob.GetN(), prob.RawArnoldiBasisVector(j), 1, x, 1);
axpy(prob.GetN(), 0.52, prob.RawArnoldiBasisVector(i), 1, x, 1);
```

RawEigenvalues

All classes

Declaration

```
TYPE* RawEigenvalues()
```

Description

This function returns a pointer to a vector that contains all the “converged” eigenvalues.

For real nonsymmetric problems, only the real part of eigenvalues is given by this vector. In this case, the imaginary part should be referenced by using function `RawEigenvaluesImag()`.

Example

```
// Taking the absolute value of some eigenvalues
// of a real nonsymmetric matrix.

int     conv = prob.ConvergedEigenvalues();
double* posr = prob.RawEigenvalues();
double* posi = prob.RawEigenvaluesImag();
double* Sol  = new double[prob.ConvergedEigenvalues()];
double* poss = Sol;

for (int i = 0; i < conv; i++, posr++, posi++, poss++) {
  poss = sqrt(posr*posr+posi*posi);
}
```

RawEigenvaluesImag

Real nonsymmetric classes

Declaration

```
FLOAT* RawEigenvaluesImag()
```

Description

This function returns a pointer to a vector that contains the imaginary part of all the “converged” eigenvalues.

Example

See the example that follows the description of `RawEigenvalues`.

RawEigenvectors

All classes

Declaration

```
TYPE* RawEigenvectors()
```

Description

This function provides raw access to eigenvectors. The return value is a pointer to a vector that stores all the eigenvectors consecutively.

For real nonsymmetric problems, complex eigenvectors are given as two consecutive vectors. The first contains the real part of the eigenvector, while the imaginary part is stored in the second vector.

Example

```
// Taking y as a linear combination of some eigenvectors
// of a symmetric matrix A, i.e., evaluating y = X*r,
// where X is a matrix whose columns are eigenvectors of A.
// Function gemv calculates y = alfa*X*r + beta*y.
// The parameters of gemv are trans, m, n, alpha, X, ldx,
// r, incr, beta, y and incy, in that order.

double* y = new double[prob.GetN()];
gemv('N', prob.GetN(), prob.ConvergedEigenvalues(), 1.0,
     prob.RawEigenvectors(), prob.GetN(), r, 1, 0.0, y, 1);
```

RawEigenvector

All classes

Declaration

`TYPE* RawEigenvector(int i)`

Description

This function provides raw access to the i-th "converged" eigenvector. It returns a pointer to the first element of such vector.

For real nonsymmetric problems, complex eigenvectors are given as two consecutive vectors, so if `Eigenvalue(i)` and `Eigenvalue(i+1)` are complex conjugate eigenvalues, `RawEigenvector(I)` will contain the real part and `RawEigenvector(I+1)` the imaginary part of the corresponding complex conjugate eigenvectors.

Example

```
double         ResNorm;
Matrix<double> A;
int            nconv   = prob.FindEigenvectors();
double*        Ax      = new double[prob.GetN()];

// Evaluating the two-norm of A*x - x*lambda, where
// x is an eigenvector and lambda an eigenvalue of A.

A.MultMv(prob.RawEigenvector(1), Ax);
axpy(n, -prob.Eigenvalue(1), prob.RawEigenvector(1), 1, Ax, 1);
ResNorm = nrm2(n, Ax, 1) / fabs(Prob.Eigenvalue(1));
```

RawSchurVectors

Real nonsymmetric and complex classes

Declaration

```
TYPE* RawSchurVectors()
```

Description

This function returns a pointer to a vector that contains all the “converged” Schur vectors stored sequentially.

Example

```
// Writing y as a linear combination of some Schur vectors.
// Vector r contains some constants.
int     n      = prob.GetN();
double* y      = new double[n];
double* eigvec = prob.RawSchurVectors();
for (int i = 0; i < prob.ConvergedEigenvalues(); i++, eigvec+=n) {
  axpy(n, r[i], eigvec, 1, y, 1);
}
```

RawSchurVector

Real nonsymmetric and complex classes

Declaration

```
TYPE* RawSchurVector(int i)
```

Description

This function provides raw access to the i-th “converged” Schur vector by returning a pointer to its first element.

Example

```
// Writing x as a linear combination of two Schur vectors.
Int     n = prob.GetN();
double* x = new double[n];
Copy(n, prob.RawSchurVector(j), 1, x, 1);
axpy(n, 0.52, prob.RawSchurVector(i), 1, x, 1);
```

RawResidualVector

All classes

Declaration

```
TYPE* RawResidualVector()
```

Description

This function returns a pointer to the first element of the residual vector.

Example

```
// Taking the two-norm of the residual vector.
double norm = nrm2(prob.GetN(), prob.RawResidualVector(), 1);
```

Output functions that use the STL vector class.

ARPACK++ also includes twelve functions that return eigenvalues, eigenvectors, Schur vectors, Arnoldi basis vectors and the residual vector using the STL `vector` class.

In view of the fact that ARPACK++ is intended to solve large-scale eigenvalue problems, all the functions included in this section return a pointer to an STL vector, rather than the vector itself. This is done to avoid generating temporary vectors that can be very memory consuming.

StlArnoldiBasisVectors

All classes

Declaration

```
vector<TYPE>* StlArnoldiBasisVectors()
```

Description

This function returns a pointer to a STL vector that stores all `ncv` Arnoldi basis vectors consecutively. If an Arnoldi basis is available, it is just copied to the output vector. Otherwise, the basis is determined by ARPACK++ and then all its `n*ncv` elements copied.

As the Arnoldi basis is used to generate eigenvectors and Schur vectors, this function should be called before `StlEigenvectors` or `StlSchurVectors` to prevent ARPACK++ from recalculating the basis.

Example

```
// Obtaining the Arnoldi basis and the eigenvalues of a problem.
vector<double>* Basis     = prob.StlArnoldiBasisVectors();
vector<double>* EigValues = prob.StlEigenvalues();
```

StlArnoldiBasisVector

All classes

Declaration

```
vector<TYPE>* StlArnoldiBasisVector(int i)
```

Description

This function returns a pointer to a STL vector that contains the i-th Arnoldi basis vector. The Arnoldi basis must be available, otherwise an error message is printed.

Example

```
// Extracting the second Arnoldi basis vector.
prob.FindArnoldiBasis();
vector<double>* Second = prob.StlArnoldiBasisVector(1);
```

StlEigenvalues

All classes

Declaration (Real symmetric problems)

```
vector<FLOAT>* StlEigenvalues(bool ivec = false, bool ischur = false)
```

Declaration (Real nonsymmetric and complex problems)

```
vector<complex<FLOAT> >* StlEigenvalues(bool ivec = false,
                                        bool ischur = false)
```

Description

This function returns a pointer to a STL vector that contains all the “converged” eigenvalues.

If the eigenvalues are already available, they are just copied into the STL vector. Otherwise, the eigenvalues are determined and stored in the output vector. In this last case, eigenvectors and Schur vectors can also be determined by setting `ivec` and `ischur` to `true`, respectively.

For real nonsymmetric problems, the output vector is complex. To obtain only the real or imaginary part of the eigenvalues, the user should use `StlEigenvaluesReal` and `StlEigenvaluesImag`. Functions `StlSchurVectors`, `StlSchurVector`, `StlEigenvectors` and `StlEigenvector` can be used to retrieve Schur vectors and eigenvectors.

Examples

There are several ways of retrieving eigenvalues by using `StlEigenvalues`. In the first example below, the `ivec` function parameter was set to `true` because the eigenvalues were also sought. When the commands are called in this order, ARPACK++ stores the eigenvalues only in `EigVal`, while the eigenvectors are stored internally and then copied onto `EigVec`.

```
// Obtaining the eigenvalues and eigenvectors of a
// real nonsymmetric double precision matrix.

vector<complex<double> >* EigVal = prob.StlEigenvalues(true);
vector<complex<double> >* EigVec = prob.StlEigenvectors();
```

The user can avoid storing the eigenvectors twice by just inverting the two commands above. In this case, a copy of the eigenvalues is kept in the ARPACK++ internal storage, while the eigenvectors are stored only in `EigVec`. Moreover, in this case `StlEigenvalues` can be called without parameters, since `StlEigenvectors` always determine eigenvalues as a by-product.

```
// Another way to obtain the same eigenvalues and eigenvectors.

vector<complex<double> >* EigVec = prob.StlEigenvectors();
vector<complex<double> >* EigVal = prob.StlEigenvalues();
```

When `FindEigenvectors` (or perhaps `FindEigenvalues`) is called before `StlEigenvalues` and `StlEigenvectors`, the ARPACK++ internal data structure is used to store the eigenvalues and eigenvectors, while a copy of them is also supplied by `EigVal` and `EigVec`.

```
// Maintaining two copies of the eigenvalues and eigenvectors.

prob.FindEigenvectors();
vector<complex<double> >* EigVec = prob.StlEigenvalues();
vector<complex<double> >* EigVal = prob.StlEigenvectors();
```

StlEigenvaluesReal

Real nonsymmetric classes

Declaration

```
vector<FLOAT>* StlEigenvaluesReal()
```

Description

This function returns a pointer to a STL vector that contains the real part of all the “converged” eigenvalues. The eigenvalues must be determined (by calling `FindEigenvalues` or `FindEigenvectors`, for example) before this function is used, otherwise an error message is printed.

Example

```
// Storing the real and imaginary parts of the eigenvalues
// of a real nonsymmetric matrix in two different vectors.

prob.FindEigenvalues();
vector<double>* EigValR = prob.StlEigenvalueReal();
vector<double>* EigValI = prob.StlEigenvalueImag();
```

StlEigenvaluesImag

Real nonsymmetric classes

Declaration

```
vector<FLOAT>* StlEigenvaluesImag()
```

Description

This function returns a pointer to a STL vector that contains the imaginary part of all the “converged” eigenvalues. The eigenvalues must be determined (by calling `FindEigenvalues` or `FindEigenvectors`, for example) before this function is used, otherwise an error message is printed.

Example

See the example that follows the description of `StlEigenvaluesReal`.

StlEigenvectors

All classes

Declaration

```
vector<TYPE>* StlEigenvectors(bool ischur = false)
```

Description

This function returns all the “converged” eigenvectors in a single STL vector. If the eigenvectors are already available, they are just copied into the STL vector. Otherwise, they are determined and stored in the output vector. In this last case, Schur vectors can also be determined by setting `ischur` to `true`. One of the `StlSchurVectors` or the `StlSchurVector` functions should be used to retrieve Schur vectors. `StlEigenvectors` always determine eigenvalues as a by-product.

Elements of the eigenvectors are stored sequentially, one eigenvector at a time. For real nonsymmetric problems, real and imaginary parts of complex eigenvectors are given as two consecutive vectors, so the user can use them to build the related conjugate pair.

Example

See the examples that follow the description of `StlEigenvalues`.

StlEigenvector

All classes

Declaration (Real symmetric problems)

```
vector<FLOAT>* StlEigenvector(int i)
```

Declaration (Real nonsymmetric and complex problems)

```
vector<complex<FLOAT> >* StlEigenvector(int i)
```

Description

This function returns a pointer to a STL vector that contains the i-th “converged” eigenvector. For real nonsymmetric problems, the output vector is complex. To obtain only the real or the imaginary part of the desired eigenvector, one should use

`StlEigenvectorReal` or `StlEigenvectorImag`. The eigenvectors must be determined before this function is called, otherwise an error message is printed.

Example

```
// Extracting the third eigenvector of a real nonsymmetric matrix.

prob.FindEigenvectors();
vector<complex<double> >* EigVec = prob.StlEigenvector(2);
```

StlEigenvectorReal

Real nonsymmetric classes

Declaration

```
vector<FLOAT>* StlEigenvectorReal(int i)
```

Description

This function returns a pointer to a STL vector that contains the real part of the i-th eigenvector. The eigenvectors must be determined before this function is called, otherwise an error message is printed.

Example

```
// Storing the real and imaginary parts of the third eigenvector
// of a real nonsymmetric matrix in two different vectors.

prob.FindEigenvectors();
vector<double>* EigVecR = prob.StlEigenvectorReal(2);
vector<double>* EigVecI = prob.StlEigenvectorImag(2);
```

StlEigenvectorImag

Real nonsymmetric classes

Declaration

```
vector<FLOAT>* StlEigenvectorImag(int i)
```

Description

This function returns a pointer to a STL vector that contains the imaginary part of the i-th eigenvector. The eigenvectors must be determined before this function is called, otherwise an error message is printed.

Example

See the examples that follow the description of `StlEigenvectorReal`.

StlSchurVectors

Real nonsymmetric and complex classes

Declaration

```
vector<TYPE>* StlSchurVectors()
```

Description

This function returns a pointer to a STL vector that contains all the `nev` “converged” Schur vectors stored sequentially. They are determined, stored in the ARPACK++ internal data structure and then copied to the output vector. `StlSchurVectors` also determines eigenvalues as a by-product.

Example

```
// Determining the Schur vectors and the eigenvalues of a
// real nonsymmetric problem. StlEigenvalues can be called
// before StlSchurVectors but, in this case, the parameter
// ischur must be set to true (see the definition of
// StlEigenvalues).

vector<float>* Schur  = prob.StlSchurVectors();
vector<float>* EigVal = prob.StlEigenvalues();
```

StlSchurVector

Real nonsymmetric and complex classes

Declaration

```
vector<TYPE>* StlSchurVector(int i)
```

Description

This function returns a pointer to a STL vector that contains the i-th Schur vector. The Schur vectors must be determined before this function is called, otherwise an error message is printed.

Example

```
// Extracting the first Schur vector of a real nonsymmetric problem.

prob.FindSchurVectors();
vector<double>* Schur = prob.StlSchurVector(0);
```

StlResidualVector

All classes

Declaration

```
vector<TYPE>* StlResidualVector()
```

Description

This function returns a pointer to a STL vector that contains the residual vector.

Example

```
// Determining the eigenvalues and the residual vector.

vector<float>* EigVal = prob.StlEigenvalues();
vector<float>* Resid  = prob.StlResidualvector();
```

Handling Errors.

ARPACK++ uses c++ exception handling mechanisms, such as the `throw` command, to report run-time errors. Thus, the user can catch an error using `try` and `catch` statements.

Besides throwing errors, ARPACK++ also prints messages on the `cerr` predefined c++ error stream unless the user defines a variable called `ARPACK_SILENT_MODE` using the command

```
#define ARPACK_SILENT_MODE
```

There is only one exception class. It is called `ArpackError` and can be used by an exception handler defined by a `catch` command. As described below, `ArpackError` contains some variables and member functions to help the user cope errors.

A "new" handler is also defined. It throws an error using `ArpackError` class, so the user can catch a "memory overflow" and decide which action must be taken thereafter.

Error class.

ArpackError

Declaration

```
class ArpackError.
```

Description

Exception class intended to help the user detecting and handling run-time errors.

Member functions

```
int Status()
```

returns an integer value representing the most recently encountered error. For a complete list of error codes see section *ARPACK++ errors and error messages* below.

Example

```
// Catching an error with ArpackError class.

ARluNonSymMatrix<double> A(n, nnz, nzval, irow, pcol);
ARluNonSymStdEig<double> prob(4L, &A);
try {
  prob.FindEigenvectors();
}
catch (ArpackError Err) {
  switch (Err.Status()) {
    case -501:
      // …
    case -303:
      // …
    default:
      // …
  }
}
```

ARPACK++ errors and error messages.

A brief description of all ARPACK++ error codes and messages is given below. The integer numbers shown on left are equivalent to the values returned by `ArpackError::Status()` function. Messages on the right side are sent to the `cerr` predefined c++ error stream unless a variable called `ARPACK_SILENT_MODE` is defined.

Negative numbers represent fatal errors, while positive numbers are related to occurrences that the user should be aware and errors that ARPACK++ was able to fix.

1. Errors in parameter definitions:

-101	Some parameters were not correctly defined.
-102	'n' must be greater than one.
-103	'nev' is out of bounds".
-104	'which' was not correctly defined.
-105	'part' must be one of 'R' or 'I'.
-106	'InvertMode' must be one of 'S' or 'B'.
-107	Range error.

2. Errors that may occur during the Arnoldi process:

-201	Could not perform LAPACK eigenvalue calculation.
-202	Starting vector is zero.
-203	Could not find any eigenvalue to sufficient accuracy.
-204	Reordering of Schur form was not possible.
-205	Could not build an Arnoldi factorization.
-291	Error in ARPACK Aupd fortran code.
-292	Error in ARPACK Eupd fortran code.

3. Errors that may occur when calling main functions:

-301	Could not correctly define internal variables.
-302	Could not find an Arnoldi basis.
-303	Could not find any eigenvalue.
-304	Could not find any eigenvector.
-305	Could not find any Schur vector.
-306	FindEigenvectors must be used instead of FindSchurVectors.

4. Errors due to incorrect function calling sequence:

-401	Vector is not already available.
-402	Matrix-vector product is not already available.
-403	Could not store vector.
-404	DefineParameters must be called prior to this function.
-405	An Arnoldi basis is not available.
-406	Eigenvalues are not available.
-407	Eigenvectors are not available.
-408	Schur vectors are not available.
-409	Residual vector is not available.

5. Errors in classes that perform LU decompositions:

-501	Matrix is singular and could not be factored.
-502	Matrix data was not defined.
-503	Fill-in factor must be increased (ARumNonSymMatrix).
-504	Matrix must be square to be factored.
-505	Matrix must be factored before solving a system.
-506,	Matrix dimensions must agree.
-507,	A.uplo and B.uplo must be equal.
-508,	Matrix data contain inconsistencies.
-509,	Data file could not be read.

6. Errors in matrix files:

-551,	Invalid path or filename.
-552,	Wrong matrix type.
-553,	Wrong data type.
-554,	RHS vectors will be ignored.
-555,	Unexpected end of file.

7. Other severe errors:

-901	This function was not implemented yet.
-902	Memory overflow.

8. Warnings:

101	'ncv' is out of bounds.
102	'maxit' must be greater than zero.
201	Maximum number of iterations taken.
202	No shifts could be applied during a cycle of IRAM iteration.
301	Turning to automatic selection of implicit shifts.
401	Factors L and U were not copied. Matrix must be factored.

References

1. T. Davis & I. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical report TR-95-020, Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL, 1995.

2. T. Davis & I. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. To appear in: *SIAM Journal on Matrix Analysis and Applications*.

3. J. Demmel, J. Gilbert, X. Li. SuperLU user's guide. Available at http://www.netlib. org/scalapack/prototype. 1997.

4. J. Demmel, S. Eisenstat, J. Gilbert, X. Li, J. Liu. A supernodal approach to sparse partial pivoting. Technical report UCB-95-??, Computer Science Division, University of California at Berkeley, Berkeley, CA, 1995.\

5. Mark Gockenbach. The Hilbert class library. Available at http://www.math.lsa. umich.edu/~gock/HCL.html.

6. R. B. Lehoucq, D. C. Sorensen, C. Yang. ARPACK user's guide: solution of large scale eigenvalue problems by implicitly restarted Arnoldi methods. Available at http://www.netlib.org/scalapack. 1996.

7. K. J. Maschhoff & D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In: Proceedings PARA96 conference, Lingby, Denmark, *Applied parallel computing in industrial problems and optimization*, J. Wasniewski; J. Dongarra; K. Madsen; D. Olesen eds. Springer Lecture Notes in Computer Science 1184: 478-86. Berlin, Springer-Verlag, 1996.

8. M. Nelson. *C++ programmer's guide to the Standard Template Library*. Foster City, IDG, 1995.

9. R. Pozo. Template numerical toolkit for linear algebra: high performance programming with c++ and the Standard Template Library. Available from http://math.nist.gov/tnt. 1996.

10. D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM Journal on Matrix Analysis and Applications,* **13**(1):357-85, 1992.

11. D. C. Sorensen. Implicitly-restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations. In: D. E. Keyes, A. Sameh, V. Venkatakrishnan eds., *Parallel numerical algorithms.* Dordrecht, Kluwer, 1995.

Index