# ARPACK++ Users' Guide

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

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

# Contents

# 0 Introduction

ARPACK++ is an object-oriented version of the ARPACK package. ARPACK [1] 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) [2, 3]. 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 an $n$-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 of 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 of 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) [4].

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).

# 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.

## 1.1 Obtaining the software

ARPACK++ is distributed as a single file called arpack++.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: http://www.ime.unicamp.br/~chico/arpack++/.

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 easily downloaded from Netlib or directly from the URL: http://www.caam.rice.edu/software/ARPACK/.

The BLAS and LAPACK routines required by the ARPACK FORTRAN package are distributed along with the software. Most classes defined by ARPACK++ do not require other routines from these libraries than those distributed with ARPACK (classes for band and dense matrices are the only exception). However, some examples included in the ARPACK++ "examples" directory will not run without installing the full version of these two packages, so it would be better to assure that no LAPACK or BLAS routines are missing before compiling the examples.

LAPACK can be obtained from the URL: http://www.netlib.org/lapack/. The BLAS is available at: http://www.netlib.org/blas/. Since LAPACK includes a subset of the BLAS files, the user must take some care while installing these libraries to avoid code duplication. Moreover, before downloading these libraries, the user should verify if local (optimized) installations of LAPACK and the BLAS are available. Other libraries required by some (but not all) ARPACK++ classes include SuperLU [5], UMFPACK [6] and the Standard Template Library (see [7]) 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://crd-legacy.lbl.gov/~xiaoye/SuperLU/

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.

The vector class from the Standard Template Library (STL) can be used to retrieve eigenvalues and eigenvectors computed by all `ARPACK++` classes. 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.

## 1.2 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. Finally, some include files used to compile `ARPACK++` examples under different platforms can be found in the `arpack++/makefiles` directory.

`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.

The main `arpack++` directory also contains a file, called `README`, that gives full instructions on how to properly install the library and run the examples.

`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.

## 1.3 Compatibility

At the present time, ARPACK++ has only been compiled with the CC and GNU g++ compilers and tested under Linux and under 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 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.

## 1.4 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[1] of a standard problem as a function of $n$, $nev$ and $ncv$. Since the user is

[1]The same amount of memory is required to find $nev$ Schur vectors or an Arnoldi basis instead of the eigenvectors.

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 ($2nev + 1$).

| Type of problem | Memory positions required | Memory usage with default ncv |
|---|---|---|
| Real symmetric | $4n + n \cdot ncv + ncv^2$ $+\ 8ncv + nev$ | $5n + 2n \cdot nev + 4nev^2$ $+\ 21nev$ |
| Real nonsymmetric | $4n + n \cdot ncv + 3ncv^2$ $+\ 9ncv + 2nev$ | $5n + 2n \cdot nev + 12nev^2$ $+\ 32nev$ |
| Complex | $8n + 2n \cdot ncv + 6ncv^2$ $+\ 15ncv + 2nev$ | $10n + 4n \cdot nev + 24nev^2$ $+\ 56nev$ |

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[2]. 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.

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.

## 1.5 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.*

[2]Typical values are: 4 bytes for single precision variables and 8 bytes if double precision is used.

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[3]. The compiler option -O was used in all tests. The dimension of the problem was set to values varying between 100 and 2025. 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.

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. It has to be verified if better results can be obtained using other c++ compilers.

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 for matrices that are not so sparse.

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

`ARPACK++` also contains some classes that are capable of performing all of 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.

# 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.

## 2.1 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 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 defined only if the user does not want to use the default values provided by ARPACK++.

### 2.1.1 Defining a matrix in Compressed 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[1], with dd on the main diagonal, dl on the subdiagonal and du on the superdiagonal.

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

[1] ARPACK++ also includes a band matrix class that could have been used here. However a new class is defined since the purpose of the example is to show how to define a matrix using the CSC format.

```
{
    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.
```

### 2.1.2 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).

## 2.2 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 = NULL, bool AutoShift = true)
{
    // Creating a matrix in ARPACK++ format.
    ARluNonSymMatrix<FLOAT, 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 several 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 needs 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.

# 3 ARPACK++ classes and functions

Due to algorithmic considerations concerning program efficiency and simplicity, `ARPACK` FORTRAN code 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.

## 3.1 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.

### 3.1.1 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 $n \cdot n$ vector if the matrix is dense.

| Type of matrix | | Class name |
|---|---|---|
| Real symmetric | standard: | `ARluSymStdEig` |
| | generalized: | `ARluSymGenEig` |
| Real nonsymmetric | standard: | `ARluNonSymStdEig` |
| | generalized: | `ARluNonSymGenEig` |
| Complex | standard: | `ARluCompStdEig` |
| (not Hermitian) | 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++` and the number of desired eigenvalues. A list of available matrix classes is given later in this chapter.

### 3.1.2 Classes that require user-defined matrix-vector products

This group includes classes that allow 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 *Computational modes* section below.

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 | | Class name |
|---|---|---|
| Real symmetric | standard: | `ARSymStdEig` |
| | generalized: | `ARSymGenEig` |
| Real nonsymmetric | standard: | `ARNonSymStdEig` |
| | generalized: | `ARNonSymGenEig` |
| Complex | standard: | `ARCompStdEig` |
| (Hermitian or not) | 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.

### 3.1.3 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 | | Class name |
|---|---|---|
| Real symmetric | standard: | `ARrcSymStdEig` |
| | generalized: | `ARrcSymGenEig` |
| Real nonsymmetric | standard: | `ARrcNonSymStdEig` |
| | generalized: | `ARrcNonSymGenEig` |
| Complex | standard: | `ARrcCompStdEig` |
| (Hermitian or not) | generalized: | `ARrcCompGenEig` |

### 3.1.4 Base classes

All of 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. However, they can be useful if someone wants to build his own classes related to some specific problems.

## 3.2 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`, a 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[1] 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 be described later in this chapter. A detailed description of all `ARPACK++` class parameters and constructors can be found in the appendix.

[1]Except those classified as pure base classes.

## 3.3 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 one of the predefined matrix classes 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.

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 | | Class name |
|---|---|---|
| SuperLU (CSC format) | symmetric: | `ARluSymMatrix` |
| | non-symmetric: | `ARluNonSymMatrix` |
| UMFPACK (CSC format) | symmetric: | `ARumSymMatrix` |
| | non-symmetric: | `ARumNonSymMatrix` |
| LAPACK (band format) | symmetric: | `ARbdSymMatrix` |
| | non-symmetric: | `ARbdNonSymMatrix` |
| LAPACK (dense) | symmetric: | `ARdsSymMatrix` |
| | non-symmetric: | `ARdsNonSymMatrix` |

## 3.4 Computational modes

For some eigenvalue problems, it is also important to select the appropriate spectral transformation strategy to be used. 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 appropriate 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 [1].

### 3.4.1 Real symmetric standard problems

There are two computational modes 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 modes 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 $\sigma$ 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 $\nu$, the eigenvalues of largest magnitude of $OP = (A - \sigma I)^{-1}$, can be used to calculate the eigenvalues $\lambda$ that are nearest to $\sigma$ 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 a 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.

### 3.4.2 Real symmetric generalized problems

`ARPACK++` also provides three classes, named `ARSymGenEig`, `ARluSymGenEig` and `ARrc-SymGenEig`, 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:

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

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 $\sigma$ 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 $\sigma$. If $Ax = Bx\lambda$ is rewritten in the form

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

the largest eigenvalues $\nu$ 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 must be supplied by the user in case one of the `ARSymGenEig` or `ARrcSymGenEig` classes are being used. Moreover, matrix $A$ must be symmetric positive semi-definite.

**4. Cayley mode.** In this last mode, to find eigenvalues near a shift $\sigma$, 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 $OP = (A - \sigma B)^{-1}$.

### 3.4.3 Real nonsymmetric standard problems

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

**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 s 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 $\sigma$ 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.

### 3.4.4 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

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

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 $\sigma$. Only matrices $A$, $B$ and the shift $\sigma$ are required if class `ARluNonSymGenEig` is being used. 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 on symmetric problems for more details.

**3. Complex shift and invert mode.** To find eigenvalues near a complex shift $\sigma$ in a nonsymmetric generalized problem, `ARPACK++` needs to perform three different matrix-vector products:

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

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 $\nu$ and $\lambda$ is not bijective in this case, the Rayleigh quotient

$$\lambda = \frac{x^H Ax}{x^H Bx}$$

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.

### 3.4.5 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 $\sigma$ . When one of the `ARCompStdEig` or `ARrcCompStdEig` classes is being used, the user must supply a function that evaluates the matrix-vector product

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

### 3.4.6 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 include two drivers.

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

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

If `ARluCompGenEig` 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 $\sigma$. When using one of the `ARCompGenEig` or `ARrcCompGenEig` classes, the user must supply two functions that evaluate the products

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

See the section about symmetric problems for more details.

## 3.5 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 is given here, so the user should refer to the appendix for a complete description of `ARPACK++` functions and their parameters.

### 3.5.1 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. |

### 3.5.2 Changing problem parameters

Although changes in problem data are not very common, they are allowed by `ARPACK++`, so the user can 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. |

### 3.5.3 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 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. |

### 3.5.4 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 supplying 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, retrieving 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` | Determines 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` | Returns 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 STLStandard Template Library 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. |

### 3.5.5 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. |

### 3.5.6 Executing ARPACK++ step by step

The *reverse communication interface* classes requires the user to interact with `ARPACK++` and perform matrix-vector products, user-supplied 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: it is necessary to determine 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. |

### 3.5.7 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 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. |

### 3.5.8 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 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 the eigenvalues, when the problem is real and nonsymmetric. |
| `RawEigenvectors` | Returns a pointer to a vector that stores all of 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. |

# 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.

## 4.1 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, since 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.

## 4.2 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.

## 4.3 Using ARPACK++ matrix classes

The easiest way to define 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, 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 `ARluNon-SymMatrix`.

```
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.
arcomplex<double>* nzval; // pointer to an array that stores the
                          // nonzero elements of A.

n = 10000;
CompMatrixA(n, nnz, nzval, irow, pcol);
ARluNonSymMatrix<arcomplex<double>, 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, some other parameters not shown here, such as 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.

### 4.3.1 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[1]. 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 stringent restriction since any other information about the matrix, such as the number of rows or columns, can be passed indirectly to `MultMv` through 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. Other two variables used by `MultMv`, m and n, are defined when the constructor is called.

[1]Naturally, default arguments are also allowed.

## 4.4 Creating eigenvalue problems

There are two different ways to declare 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

### 4.4.1 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 $5.7 + 2.3i$ 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, arcomplex<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, arcomplex<double>(5.7, 2.3));
```

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

### 4.4.2 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.

## 4.5 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.

### 4.5.1 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[2].

`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 are to be supplied as parameters to other functions. `RawEigenvector`, one of such functions[3], 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`. The function that performs the product, `A.MultMv`, takes two pointers to double precision real vectors as parameters. `ARPACK++` function `GetN` is used to determine the dimension of the problem.

### 4.5.2 Employing user-defined data structure

`ARPACK++` also allows 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) << "] = ";
```

[2]`SchurVector` and `ResidualVector` are other functions that could be used here.
[3]Other functions with similar meaning are `RawEigenvalues` and `RawSchurVector`.

```
    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`. For real nonsymmetric problems, real eigenvectors occupy n successive positions[4], 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 found by `EigenValVectors` can be complex, `EigVec` must be dimensioned to store (`nconv`+1)·`n` real elements.

### 4.5.3 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 its class 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`.

[4]Here, n is the dimension of the eigensystem.

## 4.6 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 $\sigma_i$ 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

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

In view of the fact that $\bar{A}$ has both $\sigma_i$ and $-\sigma_i$ as eigenvalues, it is important to set variable `which` 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 $\bar{A}$ is a $(m+n) \times (m+n)$ matrix, while $A^TA$ contains only $n^2$ elements. Even considering that the sparse matrix-vector products $\bar{A}v$ and $A^TAv$ require the same amount of float point operations, the Arnoldi vectors generated at each iteration of `ARPACK` are greater when $\bar{A}$ is used. Moreover, setting `which` to "LM" is generally better than using "LA".

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

### 4.6.1 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 $A^TAv$, $AA^Tv$ and $\bar{A}v$ 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, double> A(m, n, nnz, valA, irow, pcol);

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

ARSymStdEig<double, ARluNonSymMatrix<double, double> >
prob(n, 4, &A, &ARluNonSymMatrix<double, 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.

# 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 are 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.

## 5.1 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 needs 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++` (see chapter one).

## 5.2 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.

### 5.2.1 Solving a symmetric generalized eigenvalue problem

In this first example, the Cayley mode[1] 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 only one of the upper or lower triangular parts of the matrix is 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.

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;
```

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

```
    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 matrixband format 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 $6 \times 6$ 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 $3 \times 6$ 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 $\bar{M}_{upper}$:

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

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

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

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

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` objects. Then, class `ARluSymGenEig` is used to create a generalized problem[2], 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

[2]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.

- 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);

    // ...

} // 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 size 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).

### 5.2.2 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, $\Delta$ represents the Laplacian operator, and $\nabla$ the gradient. $\rho$ is a complex parameter. A similar example can be found in the `examples/superlu/complex/lcompreg.cc` file.

**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, arcomplex<FLOAT>* &A,
INT* &irow, INT* &pcol)
{
    // Declaring internal variables.
    INT                i, j, k, id;
    arcomplex<FLOAT> h, h2, dd, dl, du, f;

    // Defining constants.
    const arcomplex<FLOAT> half(0.5, 0.0);
    const arcomplex<FLOAT> one(1.0, 0.0);
    const arcomplex<FLOAT> four(4.0, 0.0);
    const arcomplex<FLOAT> rho(1.0e2, 0.0);

    h   = one/arcomplex<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 arcomplex<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.
    arcomplex<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<arcomplex<double>, 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.

### 5.2.3 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 $A^T A$[3]. 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.

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

```
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;

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

    // Defining what we need: eigenvalues from both ends of the spectrum.
    ARSymStdEig<double, ARluNonSymMatrix<double, double>>
        prob(n, 6, &A, &ARluNonSymMatrix<double, double>::MultMtMv, "BE", 20);

    // 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`[4]. 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 $A^T A$. Six 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 number of Arnoldi vectors generated at each iteration (see the description of `ncv` on the appendix).

The eigenvalues of $A^T A$ 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.

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

## 5.3 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.

### 5.3.1 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 $4/h^2$ as the main diagonal elements, $-1/h^2 - \rho/(2h)$ in the subdiagonal and $-1/h^2 + \rho/(2h)$ 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 `T` will designate one of the c++ predefined types float or double. `NonSymMatrix` can contain variables and functions other than `MultMv`. The 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, h;

        h  = 1.0/T(ncols()+1);
        dd = -4.0/h*h;
        dl = -1.0/h*h - 0.5*rho/h;
        du = -1.0/h*h + 0.5*rho/h;

        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.
```

### 5.3.2 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.
```

## 5.4 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 illustrates 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. 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`, `StlEigenval-ues` and Eigenvalue are also available when using the *reverse communication interface*.

## 5.5 Printing some information about eigenvalues and eigenvectors

The function `Solution` below illustrates how to use class `ARNonSymStdEig` to extract information about eigenvalues and eigenvectors from a real nonsymmetric problem. Only a few suggestions are shown here. A complete list of `ARPACK++` output 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.
#include "arsnsym.h"

template<class MATRIX, class FLOAT>
void Solution(MATRIX &A, ARNonSymStdEig<FLOAT, MATRIX> &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 redefined
    // as function templates by ARPACK++. 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
```

## 5.6 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 is the best alternative when only a few eigenvalue problems are to be solved. However, if the user intends to solve many eigenvalue problems, it can be worth defining 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.
#include <stddef.h>

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

    public:

    // a) Public functions:

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

    virtual void ChangeShift(FLOAT sigmaRp);

    virtual void SetRegularMode();

    virtual void SetShiftInvertMode(FLOAT sigmap);

    // a.2) Constructors and destructor.

    ARluNonSymStdEig() { }
    // Short constructor.

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

    ARluNonSymStdEig(int nevp, ARumNonSymMatrix<FLOAT, FLOAT>& A,
        FLOAT sigma, char* whichp = "LM", int ncvp = 0,
        FLOAT tolp = 0.0, int maxitp = 0,
        FLOAT* residp = NULL, 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. The new class inherits all functions and variables of this base 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 $\sigma$ 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, so this function can be inherited from ARNonSymStdEig. However, 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.

## References

[1] R. B. Lehoucq, D. C. Sorensen, and C. Yan, "Arpack users guide: Solution of large scale eigenvalue problems by implicitly restarted arnoldi methods.," 1997.

[2] D. C. Sorensen, "Implicit application of polynomial filters in a k-step arnoldi method," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 357–385, 1992.

[3] D. C. Sorensen, "Implicitly restarted arnoldi/lanczos methods for large scale eigenvalue calculations," tech. rep., Institute for Computer Applications in Science and Engineering (ICASE), 1996.

[4] R. Pozo, "Template numerical toolkit for linear algebra: High performance programming with c++ and the standard template library," *Int. J. High Perform. Comput. Appl.*, vol. 11, no. 3, pp. 251–263, 1997.

[5] X. S. Li, J. W. Demmel, J. R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki, "Superlu users' guide," *Tech. Rep. LBNL-44289*, 1999. Last update: August 2011.

[6] T. A. Davis and I. S. Duff, "A combined unifrontal/multifrontal method for unsymmetric sparse matrices," *ACM Trans. Math. Softw.*, vol. 25, no. 1, pp. 1–20, 1999.

[7] M. Nelson, *C++ Program Guide to Standard Template Library.* Foster City, CA, USA: IDG Books Worldwide, Inc., 1995.

# Index