UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Documentation for the UCLA++ C++ Class Library

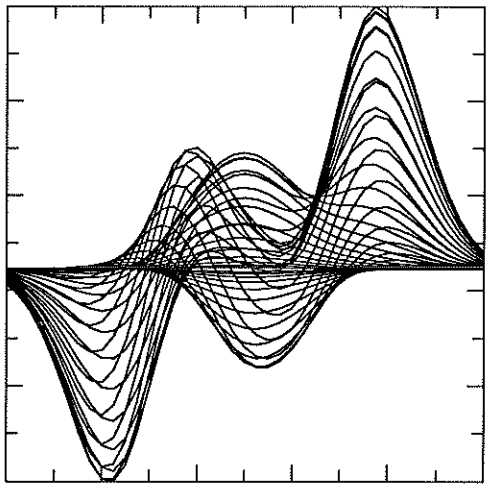
Christopher R. Anderson

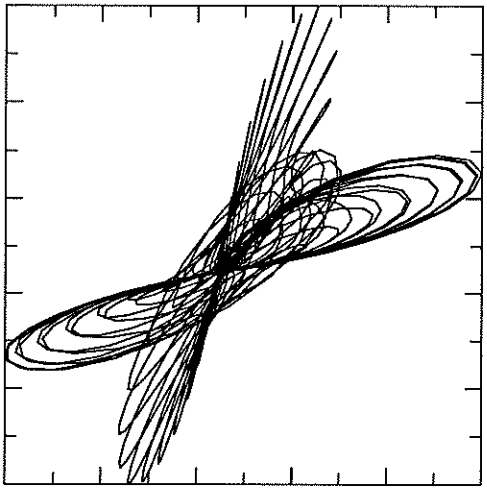
October 1993

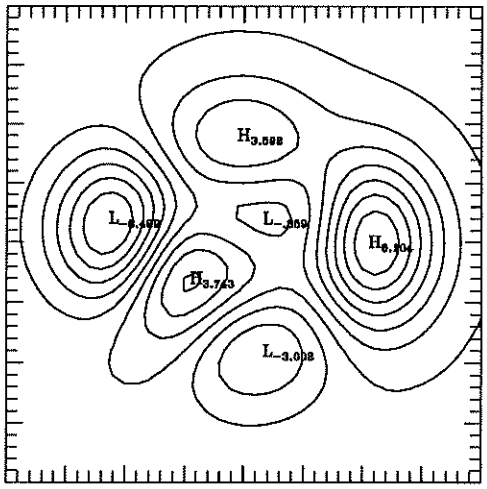
CAM Report 93-37

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

UCLA++ Class Libraries







CONTOUR FROM −5.366 TO 6.981 BY 1.122

DOCUMENTATION FOR THE UCLA++ C++ CLASS LIBRARY

CHRISTOPHER R. ANDERSON*

* Department of Mathematics, UCLA, Los Angeles, California, 90024
Research Supported by ONR Contract #N00014-92-J-1890

Preface

In order to use the UCLA++ class libraries you do not need to know C++. What you need to know is the basic syntax for C. I have found that it is better to look at C++ books than to look at books only concerned with C. The requisite C knowledge is usually summarized in the early chapters of C++ books. A presentation of this material which I like is that given in chapters 1-4 of "C++ Primer" (2nd edition) by Stanley B. Lippman. C++ books can be very intimidating - there are words and concepts which are unfamiliar to most programmers - derivation, inheritance, templates etc. Fortunately, you don't have to know these in order to use a C++ class library effectively. These concepts are needed by those who wish to create class libraries - not necessarily those that wish to use them.

The UCLA++ class library is incomplete. In fact, it is my hope that the library will never be completed. Like a true library in which books are continually added - the UCLA++ class library is meant to be a repository of software. At the present time the library contains routines of general utility. The library has incorporated existing subroutine libraries and packages - LAPACK for dense matrix computations and NCAR graphics for the UNIX graphics implementation. The library also includes the product of recent research results - in particular the sparse matrix routines developed by Barry Smith and William Gropp [5]. In the coming months more software will be added - so watch for announcements. We are also finishing up a version which will run on PC's.

This is the first general release of the class libraries, and we expect that there will be problems which our initial testing did not uncover. Please let us know if you encounter problems.

David Sansot provided the UNIX graphics implementation and the graphics documentation. Significant contributions to the library were also made by Matt Bookman and Robert MacDanial. I wish to express my thanks to these individuals, as well as all of the other individuals (and there are many) with whom conversations led to improvements in the design of this class library.

Documentation for the UCLA++ C++ Class Library

Contents

1. Introduction. The purpose of this report is to describe and document a C++ class library which has been developed to assist in the construction of programs for doing scientific computation.

First, for those who don't have experience with C++, a discussion of what C++ is, and what is meant by "class library" is needed. C++ is a language which is built upon C, it is C with extensions. The reason for wanting to use C++ instead of C is precisely because of these extensions. In C++ one can enhance the language - i.e. add data types and define operations such as * and + upon them. For example, we are able to create data types matrix and vector which are used to represent matrices and vectors of floating point numbers. Within C++, the C language can be extended, so that if `A` is a matrix and `x` and `b` are vectors then the assignment of `b` to the product of `A` and `x` is programmed as

```
b = A*x;
```

A *class* in C++ is a constructed data type with operations and functions associated with that data type. A *class library* is just a collection of classes. For instance, in the class library **UCLA++** is the class **matrix**. In this class the data structure for a matrix of values of type **double** is defined. Also associated with (or in) this class are functions and operators which work with the matrix data type. Besides an extension of the multiplication operator, there are also routines for solving linear systems, finding eigenvalues etc.

We envision two levels of use of C++ and the **UCLA++** class library. The first level of use is to program "as usual" (which is essentially to program in C) and just use the classes which are provided. In this mode, you just need to know the elementary aspects of the C++ language - essentially the C "core" consisting of the standard data types, control loops, calling functions, declaration of variables etc. You also need documentation about the classes so that they can be used effectively. It is our expectation that most users will be satisfied with programming at this level.

The second level of use is to program in a manner which utilizes the new features which C++ provides. This requires knowledge of the more advanced topics of the language; the construction of classes, the use of inheritance and derivation, operator and function overloading, templates etc. The class libraries which we provide should form a good basis for building more elaborate classes and make the task of constructing more elaborate classes easier.

For those of you that know C, the use of C++ with this class library shouldn't be a big change. You will still program in much the same way as C, but the task should be simpler because certain common data structures and operations on those data structures will be readily available. For those of you that program in Fortran, using C++ and the class library will be a substantial change. Most of the difference is due to the fact that the base language for C++ is C, and hence one has to learn new constructs for loops, flow control statements etc. However, your method and style of programming will be similar to Fortran — the form of the data structures in the **UCLA++** class library was motivated by the inherent data structures used in Fortran programs for scientific computation We also discuss how to call Fortran routines from C++ and so one does not have to completely abandon Fortran or necessarily rewrite old codes. (In fact, many of the operations for the matrix and vector classes are computed by calling Fortran subroutines.)

In this report we provide information about the basic classes contained within the **UCLA++** class library. At the present time these classes consist of the following :

matrix	:	A class for matrices of type double.
vector	:	A class for vectors of type double.
sparse_matrix	:	A class for matrices of type double which are stored in sparse form.
complex_matrix	:	A class for matrices of type complex.
complex_vector	:	A class for vectors of type complex.
array1D	:	A class for a one dimensional array of values of type double.
array2D	:	A class for a two dimensional array of values of type double.

Routines for graphical output are also contained within the class library.

The information provided here is intended to be sufficient for those who wish to program in C++ at the "first level". In a related report "On the Construction of C++ Classes for Scientific Computation" we provide a discussion and examples of the use of C++ and the **UCLA++** class library for programming at the "second level".

2. Accessing The UCLA++ C++ class library. In order to use the class library, you must include the UCLA++ header file `UCLA++.h`. This file should be included in all files which use the UCLA++ classes.

Using the library requires linking to several different object files and to make this task easier we have created a new command. For versions of the library which are compatible with the SUN CC compiler, the command is called `UCC`. For the first program in the next section one would use a command of the form

```
UCC firstexamp.c -o firstexamp
```

This compiles the program firstexamp.c and places the executable code in a file called firstexamp.

`Ug++` is the corresponding command for those who wish to use the GNU g++ compiler.

2.1. Getting Graphics Output. The graphics routines are based on an NCAR (National Center for Atmospheric Research) graphics package. If your program uses graphics calls, then when it runs it creates a graphics"metafile" in which the graphics output is placed. The default name for this file is `gmeta`. To view your graphics output you must execute a translator which translates the information in the metafile and displays the results on a viewing device — like your screen or a printer. The simplest way to view your graphics output is with the command `ctrans`. When executed the command

```
ctrans gmeta
```

causes a window to appear and you should mouse-click in the graph window to see the first frame. You then click to see each successive frame, and click to exit. There are fancier translators, namely, `idt` and `ictrans`. See the handout on NCARG for details.

To name the metafile other than the default name `gmeta` or to view the plots in "real" time (i.e. as the program runs) you must set the environment variable NCARG_GKS_OUTPUT. Here is how this is done:

```
setenv NCARG_GKS_OUTPUT filename      To send output to filename
setenv NCARG_GKS_OUTPUT "| ctrans"   To view in real time
```

You can also get a hard copy with ctrans by typing :

```
ctrans -d ps.mono gmeta | lpr -P<PostScript printer>
```

The variable <PostScript printer> should be replaced by your PostScript printer. If you have access to the Imagen printer then you can take advantage of the special Imagen encoding by typing:

```
ctrans -d imagen gmeta | lpr -Pimagen
```

This is useful with very complicated graphs that may show fragmentation using postscript.

2.2. Decreasing Compilation Time. If you include the file `UCLA++.h` you need not include any other file to access the classes in the **UCLA++** class library. However, the simplicity of this arrangement comes at a cost - the compile time can be long because the headers for all the classes are compiled. You can decrease your computation time by including header files which relate only to the specific classes which you use. The required files and the classes they pertain to are given in the following list:

Desired Class	Required Include File
vector	UCvector.h
matrix	UCmatrix.h
sparse_matrix	UCsparse.h
complex_vector	UCcomplex_vector.h
complex_matrix	UCcomplex_matrix.h
array1D and array2D	UCarray.h
graphics routines	UCgraphics.h

As an example — if you are writing a program which only uses matrices and vectors and graphics, then you would use the three include statements

```
#include "UCvector.h"
#include "UCmatrix.h"
#include "UCgraphics.h"
```

3. Beginning Samples. When one is working with a new language it is always useful to have a sample program which one can type in to see if things are working properly. Here is a complete C++ code which constructs a three by three Hilbert matrix and then solves a linear system of equations which involves this matrix.

```
#include "UCLA++.h"

void main(){

  matrix A(3,3);                    // declare a matrix
  vector b(3);                      // declare vectors
  vector x(3);
  vector residual(3);

  long i,j;                         // variables for loop index


  for(i=1; i<= 3; i++)              // nested do-loop for initializing
  for(j=1; j<= 3; j++)              // the matrix A.
  {
    A(i,j)=1.0/(double(i) + double(j));
  }

  for(i=1; i<=3; i++)               // initialize b
  {
    b(i)=double(i);
  }

  x=A/b;                            // solve the system A*x=b for x
  residual=A*x - b;                 // compute the residual

                                    // set scientific notation output
                                    // format

  cout.setf(ios::scientific, ios::floatfield)

  cout << " The matrix A " << endl;
  cout << A;
  cout << " The vector b " << endl;
  cout << b;
  cout << " The solution of A*x=b " << endl;
  cout << x;
  cout << " The residual " << endl;
  cout << residual;
}
```

The second sample is a slight modification of the first, and involves working with file input and output.

```
#include "UCLA++.h"

void main(){

  matrix A(3,3);                    // declare a matrix
  vector b(3);                      // declare vectors
  vector x(3);
  vector residual(3);

  ifstream InFile;                  // declare an input file stream

                                    // connect the file stream with
  InFile.open("tinput",ios::in);    // the external file tinput
  if (!InFile )
  {cerr <<  " Error in Opening input File "; exit(1); }

  InFile >> A;                      // read in the matrix A
  InFile >> b;                      // read in the vector b

  x=A/b;                            // solve the system A*x=b for x
  residual=A*x - b;                 // compute the residual

  ofstream OutFile;                 // declare an output file stream

                                    // connect the file stream with
  OutFile.open("toutput",ios::out); // the external file toutput.
  if (!OutFile )
  {cerr << " Error in Opening Output File "; exit(1); }

                                    // print our results to the file
                                    // first set scientific notation
                                    // output format

  OutFile.setf(ios::scientific, ios::floatfield)

  OutFile << " The matrix A " << endl;
  OutFile << A;
  OutFile << " The vector b " << endl;
  OutFile << b;
  OutFile << " The solution of A*x=b " << endl;
  OutFile << x;
  OutFile << " The residual " << endl;
  OutFile << residual;
}
```

For the above example, the file `tinput` has the form

```
5.0000e-01  3.3333e-01  2.5000e-01
3.3333e-01  2.5000e-01  2.0000e-01
2.5000e-01  2.0000e-01  1.6667e-01

1.0000e+00
2.0000e+00
3.0000e+00
```

4. Important Aspects of C++. C++ has new features within it which distinguish it from other languages. While many of these features do not need to be understood or used to get quite a bit of work done, there are some aspects about C++ which one must be aware of. In this section we give those elements of the C++ language which we think are worth knowing about and a brief description of them.

class : Classes are natural extensions of the standard data types which are available in the C language. Just as one declares variables to be of a particular type, one declares variables to be of a particular class. It is common practice to call a variable which is of a given class to be an "object" of that class. There are predefined operations associated with the standard data types — for example +, -, *, /, on integers and floating point numbers. A class also has the capability of having predefined operators associated with it. This association allows one to have the compiler give meaning to expressions involving objects of a class and the symbols +,- ,*, =, etc.. Within C++ one can go further than just associating the common mathematical symbols with the objects of the class, one also has the capability of defining functions which are intimately linked with objects of that class. These functions are called class *member functions.*

Thus, a particular class can be considered as a data structure *and* a set of functions and operators which are associated with that data structure.

member function : A member function of a class is a function which is associated with that class. Member functions are referred to (or invoked) using the "." (dot) syntax. For example, if `P` is an object of a certain class, then one invokes the member function `myfun()` by the syntax `P.myfun();`.

operator overloading : This refers to the capability within C++ of defining the meanings of symbols such as +, -, =, etc. when the variables in the expressions involving these symbols are objects associated with a particular class.

function overloading : In C++ a function is determined by both it's name and it's calling sequence (or signature). Thus, one can have functions with the same name but which are distinguished by their input variables. The routine plot in the **UCLA++** library is a good example of this. One can invoke plot with a vector or a matrix object and the compiler distinguishes these calls, and uses the appropriate version of the command.

5. Class Library Summary. In this section we summarize the classes in the **UCLA++** class library and their associated operators and routines. The different classes which are contained within the library are

matrix	:	A class for matrices of type double.
vector	:	A class for vectors of type double.
sparse_matrix	:	A class for matrices of type double which are stored in sparse form.
complex_matrix	:	A class for matrices of type complex.
complex_vector	:	A class for vectors of type complex.
array1D	:	A class for a one dimensional array of values of type double.
array2D	:	A class for a two dimensional array of values of type double.

The use of the classes associated with matrices and vectors is similar for the different data types and storage format. For example, the syntax for accessing an element of a matrix which is declared with **sparse_matrix** is the same as that for accessing an element of a matrix which is declared with **matrix**. For this reason, we only give complete summaries of the commands for matrices and vectors of type double. For the matrices and vectors of different types we just summarize the functions and routines which are different from their double counterparts.

5.1. Classes for Matrix and Vector Computation. There are two components of the class libraries for matrix and vector computation. The first component is that which implements vector and matrix data types and provides basic functionality for these data types. The second component provides high level functionality associated with the vector and matrix data types. (While we consider these components separate for our discussion, both components are available when the UCLA++ library is linked to a users program.)

Since vectors and matrices are fundamental construct for a large number of scientific programming tasks, we expect the first component of the classes to be used extensively. Thus, in the design of the first component of the classes for matrix and vector computations it was our goal to introduce as little new syntax as possible - i.e. we followed the standard mathematical syntax associated with matrices and vectors. For language constructs which are not in the realm of standard mathematical notation but available for data types in C++ (such as the operators +=, or << and >>) we extended such constructs in the "natural way" to matrices and vectors. It is our hope that a user can write syntactically correct expressions involving vector and matrix data types without having to consult the documentation very often.

In creating the functions necessary for working with the matrix and vector data structures (functions and expressions associated with initialization, determination of size, etc.) we sought to keep the number of functions to a minimum. Again, a goal was to create matrix and vector classes so that a minimal amount of information needs to be remembered by a user in order to carry his or her work.

At this time, the second component consists primarily of C++ bindings to Fortran routines in the LAPACK, LINPACK and EISPACK libraries. As C++ becomes more widely used and new routines for matrix computations are developed, this component will be expanded.

The classes for matrix and vector computations associated with the **double** data type are :

class **vector** : A class for a vector of values of type **double** and whose member functions are operations on vectors. The vectors are *column* vectors (by default).

class **matrix** : A class for a matrix of values of type **double** and whose member functions are operations on matrices as well as matrix/vector operations.

The following table describes unary and binary operations which are defined on instances of these classes :

Operator	Description
=	assignment
+	addition
-	subtraction
*	multiplication
/	x=A/b $\equiv x = A^{-1} * b$
~	adjoint
-()	negation
+=	incremental addition
-=	incremental subtraction
*=	incremental multiplication
<<	vector and matrix output
>>	vector and matrix input

Table 5.1
Operators for Class Objects of **matrix** and **vector** type

The description of ~ as "adjoint" means that for real matrices and vectors the ~ operation is the transpose operator, and for complex vectors and matrices the ~ operation is the conjugate transpose operator. The operators in Table 5.1 are applicable whenever the expression in which they are contained makes mathematical sense. Some of the operators have been extended to have meaning when an expression does not make strict mathematical sense; for example if v and w are vectors and c is a scalar, then the expression $v = w + c;$ adds c to each component of w and stores the result in v.

In C++ one must declare variables and class objects before use. For objects this is accomplished by a statement consisting of the class name and a variable name to represent the particular object. In the declaration of a class object one can optionally specify parameters to be used for the initialization of that object. The following table describes the declaration statements for objects of the vector and matrix classes.

Statement	Result of Declaration
`vector v(m);`	A vector `v` of size m with index starting at 1.
`vector v(m1,m2);`	A vector `v` with index from m1 to m2.
`vector v;`	A vector `v` of 0 size (a null vector - see below).
`vector v = w;`	A vector `v` which is a copy of a vector `w`.
`vector v = M;`	A vector `v` which is a copy of a column or row matrix `M`.
`matrix M(m,n);`	A matrix `M` of size m by n with index starting at (1,1).
`matrix M(m1,m2,n1,n2);`	A matrix `M` with index from (m1,n1) to (m2,n2).
`matrix M;`	A matrix `M` of 0 size (a null matrix - see below).
`matrix M = T;`	A matrix `M` which is a copy of a matrix `T`.
`matrix M = v;`	A matrix `M` which is a copy of a vector `v`.

Table 5.2
Class **vector** and **matrix** Declaration Statements

As with the standard data types, one forms expressions which involve objects of classes. One uses the names of the variables to represent the objects - for example, if `A` and `B` are two square matrices of the same size, then to express their product as a matrix `C` we would simply write `C = A*B;`. However, to only allow access to the objects as complete entities would severely limit our capability for using them. So, in order to access the values of the data in a **matrix** or **vector** object we have a certain number of access functions. These are listed below.

Statement	Description
`v(i);`	The element of the vector `v` whose index is i.
`v(i1,i2);`	The sub-vector of the vector `v` with indices from i1 to i2.
`v.sub_vector(i1,i2);`	Another form of `v(i1,i2);`
`M(i);`	The element of the row or column matrix with index i
`M(i,j);`	The element of the matrix `M` whose index is (i,j).
`M(i,_);`	The ith row of the matrix `M`.
`M.row(i);`	The ith row of the matrix `M`.
`M(_,j);`	The jth column of the matrix `M`.
`M.column(j);`	The jth column of the matrix `M`.
`M(i1,i2,j1,j2);`	The sub-matrix of `M` whose indices run from (i1,j1) to (i2,j2)
`M.sub_matrix(i1,i2,j1,j2);`	Another form of `M(i1,i2,j1,j2)`

Table 5.3
Class **vector** and **matrix** Access Statements

Each of the expressions in the previous table can occur on the left as well as the right hand side of an assignment statement. So, for example, to add the second and the fourth row of a matrix A to its first row, one could use either of the expressions

```
A(1,_) = A(2,_) + A(4,_);
```

or

```
A.row(1) = A.row(2) + A.row(4);.
```

There is one syntax construction in C++ which does not work when dealing with sub-vectors and sub-matrices - "multiple assignment" statements of the form

```
A = B = C;.
```

For example `A = B(2,3,2,3) = C;` will cause the compiler to issue an error and stop. This is an artifact of our particular implementation of these classes which may or may not go away depending on how clever we are in the future. Another aspect of using sub-matrices and sub-vectors is that they are always passed by value — so it is not possible to modify a sub-matrix or sub-vector by passing it to another routine. (You need to copy the sub-matrix or sub-vector to a temporary variable, call the routine with the temporary variable, and then copy the result back to the appropriate place in the sub-vector or sub-matrix.)

Along with the values which are associated with the matrix or vector elements, there are also values associated with the data structure - the size of a vector or matrix, and its beginning and ending index. There are member functions which allow one to access to these values. These are given in the table below.

Statement v is of type **vector**	Result of the Statement
`n = v.get_num_rows();`	The integer n is the number of rows in `v`.
`n = v.get_row_begin();`	The integer n is the starting row index of `v`.
`n = v.get_row_end();`	The integer n is the ending row index of `v`.
`n = v.get_num_columns();`	The integer n is the number of columns in `v`.
`n = v.get_column_begin();`	The integer n is the starting column index of `v`.
`n = v.get_column_end();`	The integer n is the ending column index of `v`.
`d = v.get_data_ptr();`	The pointer `d` (of type `double*`) is set to the address of the first element of `v`
`v.set_row_begin(n);`	The starting index of `v` is set to the integer `n`
`v.resize(m);`	The size of `v` is changed by `m` elements. (`m` may be positive or negative)
`v.resize(m1,m2);`	The null vector `v` is resized to a vector whose index runs from `m1` to `m2`.

Table 5.4
Class **vector** Data Structure Access Functions

Statement M is of type **matrix**, n an integer	Result of the Statement
`n = M.get_num_rows();`	n is the number of rows in `M`.
`n = M.get_num_columns();`	n is the number of columns `M`.
`n = M.get_row_begin();`	n is the first index starting value for `M`.
`n = M.get_row_end();`	n is the first index ending value for `M`.
`n = M.get_column_begin();`	n is the second index starting value for `M`.
`n = M.get_column_end();`	n is the second index ending value for `M`.
`d = M.get_data_ptr();`	The pointer `d` (of type `double*`) is set to the address of the first element of `M`
`M.set_row_begin(n);`	The starting value of the first index of `M` is set to n
`M.set_row_begin(n);`	The starting value of the second index of `M` is set to n
`M.resize(m,n);`	The size of the `M` is changed by `m` row and `n` columns. (`m` and `n` may be positive or negative)
`M.resize(m1,m2,n1,n2);`	The null matrix `M` is resized to a matrix whose index runs from `(m1,n1)` to `(m2,n2)`.

Table 5.5
Class **matrix** Data Structure Access Functions

In addition to providing the basic functions and operations for matrix vector computations, there is also a collection of higher level functions associated with each class. The statements used to invoke the functions are given below.

Statement (`v`, `w` are vectors, `d` is a double)	Result of Statement
`d = v.norm(p);`	The value of `d` is assigned the discrete p-norm of `v`. `p` (an integer) with $1 \leq p < \infty$.
`d = v.norm("inf");`	The value of `d` is the infinity norm of `v`.
`d = v.dot(w);`	The value of `d` is the dot product of `v` and `w`.
`v.init(f);`	`v(i)` is assigned the value `f(i)`. `f` specified as `double f(long)`.
`w = v.apply(f);`	`w(i)` is assigned the value `f(v(i))`. `f` specified as `double f(double)`.
`w = v.elem_sin();`	`w(i)` is assigned the value `sin(v(i))`.
`w = v.elem_cos();`	`w(i)` is assigned the value `cos(v(i))`.
`w = v.elem_tan();`	`w(i)` is assigned the value `tan(v(i))`.
`w = v.elem_pow(p);`	`w(i)` is assigned the the pth power of `v(i)`.

Table 5.6
Class Member Functions for **vector**

Statement (M, T are matrices, d is a double)	Result of Statement
`d = M.norm(p);`	The value of d is assigned the matrix p-norm of M. p (an integer) with p =1.
`d = M.norm("inf");`	The value of d is the infinity norm of M.
`d = M.max_abs();`	The value of d is $\max_{i,j} \lvert M(i,j) \rvert$.
`M.init(f);`	M(i,j) is assigned the value f(i,j). f specified as `double f(long, long)`.
`d = M.det();`	The value of d is the determinant of M.
`d = M.condition_number();`	The value of d is the condition number of M.
`T = M.pow(p);`	T is M raised to the pth power, p < 0 is allowed.
`T = M.inv();`	T is the inverse of M.
`T = M.apply(f);`	`T(i,j)` is assigned the value `f(M(i,j))`. f specified as `double f(double)`.
`T = M.elem_sin();`	`T(i,j)` is assigned the value `sin(M(i,j))`.
`T = M.elem_cos();`	`T(i,j)` is assigned the value `cos(M(i,j))`.
`T = M.elem_tan();`	`T(i,j)` is assigned the value `tan(M(i,j))`.
`T = M.elem_log();`	`T(i,j)` is assigned the value `log(M(i,j))`.
`T = M.elem_log10();`	`T(i,j)` is assigned the value `log10(M(i,j))`.
`T = M.elem_pow(r);`	`T(i,j)` is assigned the the rth power of `M(i,j)`.
`T = matrix::Id(p);`	`T(i,j)` is a p × p identity matrix.

Table 5.7
Class Member Functions for **matrix**

5.1.1. Complex Matrices and Vectors. The declaration and access syntax of matrices and vectors of complex data type is identical to that for those of double data type. The additional member functions associated with the complex data type are given in the following tables.

Statement v,w are of type **complex_vector** s is of type **vector**	Result of Statement
`s = v.real();`	s is the real part of v.
`s = v.imag();`	s is the imaginary part of v.
`w = v.conj();`	w is the complex conjugate of v.
`x = complex_vector::fortran_pack(v);`	pack v into fortran acceptable vector (x of type **vector**)
`v = complex_vector::fortran_unpack(x);`	unpack **vector** x into a **complex_vector** type.

Table 5.8
Additional Class Member Functions for **complex_vector**

Statement M, N are of type **complex_matrix** T is of type **matrix**	Result of Statement
`T = M.real();`	T is the real part of M.
`T = M.imag();`	T is the imaginary part of M.
`N = M.conj();`	N is the complex conjugate of M.
`X = complex_matrix::fortran_pack(M);`	pack M into fortran acceptable matrix (`X` of type **matrix**)
`M = complex_matrix::fortran_unpack(X);`	unpack **matrix** `X` into a **complex_matrix** type.

Table 5.9
Additional Class Member Functions for **complex_matrix**

The ability to combine complex scaler operations with matrices and vectors declared as **matrix** and **vector** is provided. Assigning a complex quantity to one of type double is an error. An expression which involves arguments of type double and complex must be assigned to an object of type complex. If one wants to assign the result of an expression to an object of type double, then one must use an explicit conversion (e.g. the member function `*.real();`). As mentioned previously the transpose operator ~ gives the conjugate transpose of a vector or matrix. If you want just the transpose, then you should form the conjugate of the the complex transpose.

The routines involving `fortran_pack(*)` and `fortran_unpack(*)` are used when one wants to pass a **complex_vector** or **complex_matrix** object to an external Fortran subroutine. The use of these routines is documented in the section "Class Library Explanations".

5.1.2. Sparse Matrices. The declaration and access syntax for a matrix of type **sparse_matrix** is identical to that for matrices declared as objects of class **matrix**. The designation of a matrix as belonging to the **sparse_matrix** class causes the matrix values to be stored in compact form which represents only non-zero values.

For matrices with many non-zero elements (such as banded matrices), one should declare them to be of type **sparse_matrix** as this will use up less storage than a matrix declared of type **matrix**. The operators and member functions for matrices of type **sparse_matrix** are identical to those for type **matrix**. While the syntax is the same, the operations themselves are different — those for the objects of type **sparse_matrix** are optimized for that type. One can mix matrices of **matrix** and **sparse_matrix** type freely in expressions.

In order to make best use of the class **sparse_matrix** one should avoid operations which results in the **sparse_matrix** becoming a dense matrix. In particular, assigning a **sparse_matrix** to a constant (except 0.0) will yield a dense matrix - a matrix more efficiently handled using objects of type **matrix**.

5.2. array1D and array2D Classes. Objects of type **array1D** and **array2D** are one and two dimensional arrays of numbers of type double. These classes are similar, but not identical, to the classes **vector** and **matrix**. (The $*$ operator and several others are not defined for objects of type **array2D**.) The following tables summarize the operators, declarations, access functions, and member functions associated with these classes.

Operator	Description
=	assignment
+	addition
-	subtraction
-()	negation
+=	incremental addition
-=	incremental subtraction
<<	array1D and array2D output
>>	array1D and array2D input

Table 5.10
Operators for Objects of **array1D** and **array2D** type

Statement	Result of Declaration
`array1D a(m);`	An array1D a of size m with index starting at 1.
`array1D a(m1,m2);`	An array1D a with index from m1 to m2.
`array1D a;`	An array1D a of 0 size (a null array1D - see below).
`array1D a = b;`	An array1D a which is a copy of an array1D b.
`array2D A(m,n);`	An array2D A of size m by n with index starting at (1,1).
`array2D A(m1,m2,n1,n2);`	An array2D A with index from (m1,n1) to (m2,n2).
`array2D A;`	An array2D A of 0 size (a null array2D - see below).
`array2D A = T;`	An array2D A which is a copy of an array2D T.
`array2D A = v;`	An array2D A which is a copy of an array1D v.

Table 5.11
Class **array1D** and **array2D** Declaration Statements

Statement	Description
`a(i);`	The element of an array1D a with index i
`a(i1,i2);`	The sub-array1D of a whose indices run from i1 to i2.
`a.sub_array1D(i1,i2);`	Another form of `a(i1,i2)`
`A(i);`	The element of a $1 \times *$ or $* \times 1$ array2D with index i
`A(i,j);`	The element of the array2D A whose index is (i,j).
`A(i,_);`	All of the elements of the array2D A with first index i.
`A(_,j);`	All of the elements of the array2D A with second index j.
`A(i1,i2,j1,j2);`	The sub-array2D of A whose indices run from (i1,j1) to (i2,j2)
`A.sub_array2D(i1,i2,j1,j2);`	Another form of `A(i1,i2,j1,j2)`

Table 5.12
Class **array1D** and **array2D** Access Statements

Statement (a is an array1D, m an integer)	Result of the Statement
`m = a.get_num_index1();`	m is the number of values for the index.
`m = a.get_index1_begin();`	m is the index starting value.
`m = a.get_index1_end();`	m is the index ending value.
`d = a.get_data_ptr();`	The pointer d (of type `double*`) is set to the address of the first element of `a`
`a.set_index1_begin(m);`	The starting value of the index is set to m.
`a.resize(m);`	The size of the `a` is changed by `m` values in the first index. (`m` may be positive or negative)
`a.resize(m1,m2);`	The null array1D `a` is resized to an array1D whose index runs from `m1` to `m2`.

Table 5.13a
Class **array1D** Data Access Statements

Statement (`A` is an array2D, m,n are integers)	Result of the Statement
`m = A.get_num_index1();`	m is the number of values for the first index.
`n = A.get_num_index2();`	n is the number of values for the second index.
`m = A.get_index1_begin();`	m is the first index starting value.
`m = A.get_index1_end();`	m is the first index ending value.
`n = A.get_index2_begin();`	n is the second index starting value..
`n = A.get_index2_end();`	n is the second index ending value..
`d = A.get_data_ptr();`	The pointer d (of type `double*`) is set to the address of the first element of `A`
`A.set_index1_begin(m);`	The starting value of the first index is set to m.
`A.set_index2_begin(n);`	The starting value of the second index is set to n.
`A.resize(m,n);`	The size of the `A` is changed by `m` values in the first index and `n` in the second index. (`m` and `n` may be positive or negative)
`A.resize(m1,m2,n1,n2);`	The null array2D `A` is resized to an array2D whose index runs from `(m1,n1)` to `(m2,n2)`.

Table 5.13b
Class **array2D** Data Access Statements

Statement (a, s are of type array1D, d is a double)	Result of Statement
`d = a.norm(p);`	The value of d is assigned the pth root of the sum of the pth powers of the values `a(i)`. (the discrete l^p norm).
`d = a.abs();`	The value of d is $\max_i \|a(i)\|$.
`d = a.norm("inf");`	The same as `a.abs();`
`a.init(f);`	`a(i)` is assigned the value `f(i)`. `f` specified as `double f(long)`.
`s = a.apply(f);`	`s(i)` is assigned the value `f(a(i))` . `f` specified as `double f(double)`.
`s = a.elem_sin();`	`s(i)` is assigned the value sin(`a(i)`).
`s = a.elem_cos();`	`s(i)` is assigned the value cos(`a(i)`).
`s = a.elem_tan();`	`s(i)` is assigned the value tan(`a(i)`).
`s = a.elem_log();`	`s(i)` is assigned the value log(`a(i)`).
`s = a.elem_log10();`	`s(i)` is assigned the value log10(`a(i)`).
`s = a.elem_pow(r);`	`s(i)` is assigned the the rth power of `a(i)`.

Table 5.14a
Class Member Functions for **array1D**

Statement (`A`, `T` are of type array2D, d is a double)	Result of Statement
`d = A.norm(p);`	The value of d is assigned the pth root of the sum of the pth powers of the values `A(i,j)` (the discrete l^p norm).
`d = A.abs();`	The value of d is $\max_{i,j} \|A(i,j)\|$.
`d = A.norm("inf");`	The same as `A.abs();`
`A.init(f);`	`A(i,j)` is assigned the value f(i,j). f specified as `double f(long, long)`.
`T = A.apply(f);`	`T(i,j)` is assigned the value f(`A(i,j)`). f specified as `double f(double)`.
`T = A.elem_sin();`	`T(i,j)` is assigned the value sin(`A(i,j)`).
`T = A.elem_cos();`	`T(i,j)` is assigned the value cos(`A(i,j)`).
`T = A.elem_tan();`	`T(i,j)` is assigned the value tan(`A(i,j)`).
`T = A.elem_log();`	`T(i,j)` is assigned the value log(`A(i,j)`).
`T = A.elem_log10();`	`T(i,j)` is assigned the value log10(`A(i,j)`).
`T = A.elem_pow(r);`	`T(i,j)` is assigned the the rth power of `A(i,j)`.

Table 5.14b
Class Member Functions for **array2D**

5.3. Graphics. There are three sets of routines associated with the UCLA++ graphics class. There are system control routines, plotting routines, and graphics style (or formatting) routines. The basic structure of a code for using the graphics routines has the form

```
    graphics::open();                          // initialize graphics

... style commands for first picture ...

... plotting commands for first picture ...

    graphics::frame();                         // close first picture

... style commands for second picture ...

... plotting commands for second picture ...

    graphics::frame();                         // close second picture

    graphics::close();                         // close graphics.
```

As the pseudo code illustrates, the plotting and style routines are sandwiched between graphic system control routines `graphics::open();`, `graphics::frame()` and `graphics::close();`. The style routines are called before the plotting routines. The following tables summarize the different sets of commands.

Statement	Description
`graphics::open();`	Open the graphics system.
`graphics::frame();`	Advance to new frame.
`graphics::close();`	Close the graphic system.
`graphics::set_frame(left,right,bottom,top);`	Sets the portion of the window the plot will occupy. Specified w.r.t [0,1] × [0,1].
`graphics::subplot_on(m,n);`	Enter subplot mode with an m by n grid of subplots.
`graphics::subplot(i,j);`	Plot to the i,j subplot.
`graphics::subplot_off();`	Leave subplot mode.
`graphics::axis(flag);`	Turn on or off autoscaling. Flag is either `AUTO` or `AUTO_OFF`.
`graphics::axis(xmin,xmax,ymin,ymax);`	Set range for future plots and turn off autoscaling.
`graphics::set_plot_style(`*style*`);`	*style* is CURVE, POINTS, or CURVE_AND_POINTS
`graphics::set_point_style(`*char*`);`	The character *char* to be plotted when plot_style is POINTS.

Table 5.15
Graphics System Control Routines

Statement x and y are of type **vector** or **array1D**.	Description
`y.plot(<style>);`	Plot the values of y vs. its index.
`y.plot(x,<style>);`	Plot the values of y vs. corresponding value of x.
`graphics::plot(f,x_min,x_max,<style>);`	Plots the function f between x_min and x_max.
`graphics::plot(f,<style>);`	Plots the function over range stored by the internal variables.

Statement	Description
M is of type **matrix**, **sparse_matrix** or **array2D**.	
`M.contour();`	Contour plot of the values of M.
`M.contour(n);`	n evenly spaced contour lines.
`M.contour(inc);`	Contours at values $k * inc$, where k is an integer.
`M.contour(low,hi);`	Only contours between low and hi drawn.
`M.contour(n,low,hi);`	n contours between low and hi.
`M.contour(inc,low,hi);`	Contours at values $k * inc + low$, where k is an integer.
`M.contour(v);`	Contours at values stored in a vector v.
`M.surface();`	Surface (3-D perspective) plot of the M.
`M.surface(h_ang,v_ang);`	Surface plot of M from a view point rotated h_ang degrees counterclockwise in the X-Y plane and v_ang degrees up from the X-Y plane.
`graphics::turn_on_contour_scaling();`	The aspect ratio of the contour is that of that of the data (default).
`graphics::turn_off_contour_scaling();`	Contour plots fill the frame.
`graphics::set_horizontal_angle(h_ang);`	Set the horizontal viewing angle of surface plots.
`graphics::set_vertical_angle(v_ang);`	Set the vertical viewing angle.
`M.plot(<style>);`	Plot the columns of M vs. the indices.
`M.plot(v,<style>);`	Plot the columns of M vs. the vector v.
`M.plot(M2,<style>);`	Plot the columns of M vs. the columns of M2

Table 5.16
Plotting Routines for Class Objects

<style> is an optional parameter list. If <style> is nothing then the default values of plot_style and point_style are used for the plot. If <style> is a single argument, it can be one of the constants CURVE, POINTS, or CURVE_AND_POINTS, resulting in the plot being drawn in the style described by the constant. If it is just a character, e.g. `v.plot('*')`), the the points are plotted and marked by the character specified. If the <style> represents two arguments, then the first should be one of the constants CURVE, POINTS, or CURVE_AND_POINTS and the second should be a character, e.g. `v.plot(CURVE_AND_POINTS, '+')` .

Statement	Description
`graphics::title(string);`	Draws string at top of window.
`graphics::label_x(string);`	Draws string under graph.
`graphics::label_y(string,<place>);`	Draws string to side of graph.
`graphics::draw_string(x,y,string,size, angle,cntr);`	General purpose string drawing routine

Table 5.17
Character Drawing and Labelling Routines

`<place>` is an optional argument. If equal to 1, then the label will be on the right side of the graph. Any other value (or absence) causes the label to be drawn on the left.

Statement	Description
`graphics::set_ticks(majorx,minorx, majory,minory);`	Set the major and minor subdivisions of both axes.
`graphics::set_x_ticks(major,minor);` `graphics::set_y_ticks(major,minor);`	Set the major and minor subdivisions of respective axis.
`graphics::set_scale(scale_type);`	Choose between LIN_LIN, LIN_LOG, LOG_LIN, and LOG_LOG scales.
`graphics::set_label_format(format);`	Set format to either `SCIENTIFIC` of `FLOATING_POINT`
`graphics::set_label_format(width,precision);`	Set the format of the labels.
`graphics::set_char_size(size);`	Set the size of the labels.
`graphics::turn_off_labels();` `graphics::turn_off_x_labels();` `graphics::turn_off_y_labels();`	Suppresses the drawing of labels on the appropriate axis.
`graphics::turn_on_labels();` `graphics::turn_on_x_labels();` `graphics::turn_on_y_labels();`	Enables the drawing of labels on the appropriate axis
`graphics::turn_off_axis();` `graphics::turn_off_x_axis();` `graphics::turn_off_y_axis();`	Suppresses the drawing of axis and labels.
`graphics::turn_on_axis();` `graphics::turn_on_x_axis();` `graphics::turn_on_y_axis();`	Enables the drawing of axis.
`graphics::set_axis_type(type);`	Choose between a GRID, PERIMETER or FLOATING.
`graphics::set_intercepts(x,y);`	Set where axes intercept each other when of type FLOATING.

Table 5.18
Formatting Routines

6. Class Library Explanations. In this section we go over in more detail the procedures and functions which are associated with UCLA++ classes. The following topics are covered

- Declaration
- Null Vectors
- Standard Input and Output
- File Input and Output
- Formatting
- Graphics
- Calling External Fortran Routines

6.1. Declaration. Classes are a natural extension of the standard data types which are available in the C language. The standard data types must be declared before use and the same is true of variables or "objects" of a particular class type. With the standard data types, one can initialize a variable in a declaration statement, e.g. `double x = 1.0;` . With class objects one can also initialize within a declaration. However, the syntax with which one initializes objects of a class type is particular to the implementation of that class. The purpose of this section is to present the syntax and meaning of declaration and initialization for the classes **vector** and **matrix**. The initialization for the other classes in the **UCLA++** class library is similar.

The standard declaration for a vector `v` of dimension `n` or a matrix `A` of dimensions `m` by `n` is

```
vector v(n);        // v = a vector of dimension n

matrix A(m,n);     //  A = a matrix of dimension m by n
```

In this declaration `n` or `m` and `n` are specific integers or variables of type `int` or `long`. The entries of the vector `v` and the matrix `A` are initialized to zero. The starting index of the vector `v` is 1 and the starting index of the matrix `A` is $(1,1)$ (1 is the default starting index). Elements of `v` and `A` may be accessed using the notation `( )` and `( , )`. For example, the following expression adds 6 to the element with index `(3,4)` of `A` and stores the result in the element with index 2 of `v`,

```
v(2) = A(3,4) + 6.0;
```

An alternate method of declaring vectors and matrices allows one to have indexing which does not start at 1. If one desires a vector whose indices run from `n1` to `n2` or a matrix with indices which run from `m1` to `m2` and `n1` to `n2` one uses expressions of the form

```
vector v(n1 n2);            // v = a vector whose index runs
                            // from  n1 to n2. v has dimension
                            // (n2 - n1) +1)

matrix A(m1, m2, n1 ,n2); //  A = a matrix whose index runs
                            //  from (m1,n1) to (m2,n2).
                            //  A has dimension (m2 - m1)+1
                            //  by (n2 - n1) + 1
```

As with the previous method of declaration, the elements of the vector `v` and the matrix `A` are initialized to zero. As discussed in the section on indexing - one can mix matrices and vectors with different indexing in expressions as long as the dimensions are correct.

One can declare a vector or a matrix and initialize it within the same statement. For example if `A` is a previously declared matrix of size `m` by `n` then the expression

```
matrix B = A;              // B is given the size and indexing of A.
                           // The elements of B are initialized to
                           // those of A.
```

creates a matrix `B` with the same dimensions and index as `A`. The elements of `B` are initialized to the values of the elements of `A`.

6.2. Null Vectors. One can also declare a null vector or matrix. This is a vector or matrix with zero dimensions i.e. it is an "empty" vector or "empty" matrix. The following is an example of such a declarations;

```
vector v;                    // v is a null vector.

matrix A;                    // A is a null matrix.
```

Null vectors and matrices can be initialized (and hence acquire dimensions) by assignment to a non-null vector or matrix. A null vector or matrix can also be given dimension by using the command `resize` - a command which adds rows to a vector or rows and columns to a matrix. Some examples -

```
vector w(3);          // w is a vector with 3 elements - each
                      // initialized to zero

vector v;             // v and z  are a null vectors
vector z;

v=w;                  // v acquires the dimension of w and it's
                      // entries are initialized to those of w

z.resize(20);         // z is now resized with the addition of 20
                      // rows. The new elements are initialized with
                      // zero. Indexing begins with 1.

matrix A;             // A is a null matrix.

A.resize(3,4)         // A is resized to a matrix of dimensions
                      // 3 by 4. Its  elements are set to zero and
                      // the indexing begins with (1,1).

matrix B;             // B is a null matrix.

B.resize(1,3,2,4)     // B is resized to a matrix of dimensions
                      // 3 by 2. Its  elements are set to zero and
                      // the indexing begins with (1,2).
```

6.3. Indexing. Vectors and matrices have indices and dimensions. When one is carrying out linear algebra operations, what is important are the dimensions of the matrices and the vectors - not the indices used to label the individual elements. This fact is reflected in our implementation of the **matrix** and **vector** classes. In an algebraic expression with matrices and vectors the index of the objects is not used - so one can freely mix matrices and vectors with different indexing as long as the dimensions are correct.

Through the initialization process and various member functions one can set (or reset) the starting value of the indices used for vectors and matrices. The default starting index is 1 for vectors and (1,1) for matrices. The indexing of a vector or matrix is not propagated across assignment statements - so the indexing of the result of a particular expression is the indexing associated with the left hand side of the assignment statement. (There are two exceptions to this rule about propagation of indexing across assignments - the initialization of one matrix by another and the assignment of a non-null matrix to a null-matrix.)

When one declares a vector one can specify the indexing by using notation of the form

```
vector w( m1, m2);
```

to obtain a vector `w` whose index starts at `m1` and ends at `m2`. Similarly for a matrix, one can specify the indexing with syntax of the form

```
matrix A( m1, m2, n1, n2);
```

to obtain a matrix `A` with indices starting at `(m1, n1)` and ending at `(m2, n2)`. Here are some annotated examples associated with indexing.

```
vector w(3)             // vector with 3 elements and index starting at 1
vector x(1,2);          // vector with index from 1 to 2
vector y(2,3);          // vector with index from 2 to 3
vector z(0,1);          // vector with index from 0 to 1

matrix A(3,3);          // The matrix A has 3 rows and 3 columns
                        // it's index starts at (1,1)
matrix B(0,2,-1,1);     // The matrix B has 3 rows and 3 columns
                        //  it's index starts at (0,-1)

z=x + y;    // The data in z is the sum of the data in x and y.
            // The indexing of z is preserved.
z=B*x + y; // The data in z is the sum of B*x and y. The indexing
            // of z is preserved.

vector p=x; // The data in x initializes that of p. The indexing
             //  of p is the same as that of x.

vector w;    // w is initialized as a null vector.
w=x;         // Since w is a null vector, w is extended to be a
             // copy of x. w is given the indexing of x.
```

6.4. Standard Input and Output. To perform input and output in C++ one can either use C procedures or those procedures specifically associated with C++. In this section we discuss the use of the C++ specific routines. To use the functions and constructs described here one must include the file `iostream.h`. Our discussion will include the input and output of the standard data types as well as objects of the **matrix** and **vector** classes.

The input and output of data within C++ seems peculiar at first glance. However, this appearance is due to the syntax - the procedures and actions accomplished by the input/output functions are standard. For example, to input or output a standard data type one uses "streams". A stream is another name for an input or output device – a file, the screen, the keyboard etc. C++ comes with some pre-defined streams for dealing with the standard input and standard output devices (usually the keyboard and your screen.) These streams have the names - `cin` for input and `cout` for output. The C++ syntax to output the value of a variable `d` to the standard output takes the form

```
cout << d;
```

This notation is suggestive of the operation of "putting" or "sending" the value of `d` to the standard output (represented here by the name `cout`). An alternate interpretation is that a function called `<<` is being invoked with the arguments `cout` and `d` - we might think of the same expression as being `<<(cout,d)`. The latter interpretation is similar to other languages in which one calls an output routine and specifies the output device and variables as arguments. (Statements of the form `<<(cout,d);` don't compile, but when one defines the meaning of the symbol `<<` then one does use such syntax.)

6.4.1. Standard Output. To output a standard data type to the standard output one uses the operator `<<` and the output stream `cout`. Examples of its use are

```
cout << d;                                          // d is output
cout << d << endl;                                  // d followed by endl
cout << "This is the value of d : " << d << endl; // message, d, then endl
```

The use of multiple instances of `<<` is equivalent to several instances of `<<` individually, i.e. the last command given above is equivalent to

```
cout <<  "This is the value of d : " ;
cout <<   d;
cout <<   endl;
```

The symbol `endl` generates a new line and flushes the output buffer. The values of the variables are printed with a default format. The method for changing this format is described in the section on formatting.

To output objects of the vector and matrix class we have overloaded (defined) the operator `<<` so that it accepts vector and matrix arguments. If `v` is the vector

$$\begin{pmatrix} .1 \\ .2 \\ .3 \end{pmatrix}$$

then the statement

```
cout << v << endl;
```

results in the the output

```
1.0000e-01
2.0000e-01
3.0000e-01
```

If `M` is the matrix

$$\begin{pmatrix} .1 & .2 \\ .3 & .4 \\ .5 & .6 \end{pmatrix}$$

then the statement

```
cout << M << endl;
```

results in the the output

```
1.0000e-01  2.0000e-01
3.0000e-01  4.0000e-01
5.0000e-01  6.0000e-01
```

As with the standard data types one can control the output formatting when using the operator << with objects of the vector and matrix class. (This is also discussed in the formatting section.) The operator << provides a quick and convenient way for the output of matrices and vectors. Of course one need not (or may not want to) use this operator. In such cases one writes and uses output routines in the same manner as other languages.

6.4.2. Standard Input. The use of the standard input is similar to that of the standard output. One uses the operator >> and the stream `cin`. The statement

```
cin >> d;
```

reads (or "extracts") from the standard input a value for the variable d.

The operator >> is a "smart" operator; it extracts the data in a way which is appropriate for the data type which is occurring in the input statement. For example, if d is an integer, the command above reads digits until a non-digit is encountered. If d is a floating point value then the command reads digits, decimal points, and exponents until a non-appropriate character is encountered.

One can input vectors and matrices from the standard input. If a vector `v` has been declared with specific dimensions and has size `n` then the command

```
cin >> v;
```

will extract values from the input until `n` values have been input. The extraction will skip spaces and line feeds. If one doesn't provide enough values, then an error message is generated which reports that an insufficient number of values was input.

If `v` is declared as a null vector (i.e. with a statement such as `vector v;`), then the input command will extract values until an inappropriate character is reached. (An isolated decimal point works well to terminate the input.) The vector `v` is then returned with the size determined by the number of elements input.

If `M` is a matrix with finite dimensions, then the command

```
cin >> M;
```

will read values into `M` by rows. The elements of the same row need not be on the same line, nor must the individual rows be separated by a newline. For example, if the statements

```
matrix M(2,2);
cin >> M;
```

are executed and we type at the keyboard the values

```
.1   .2   .3   .4
```

Then `M` is the matrix

$$\begin{pmatrix} .1 & .2 \\ .3 & .4 \end{pmatrix}$$

A matrix object must be dimensioned before input with the >> operator - so there is not instance of statements of the form `cin >> M` in which `M` is a null matrix.

A problem with input can occur if one combines output and input statements together. One often writes programs which query the user to input numerical data. In such programs there is an output statement which writes a prompt to the user followed by an input statement to read in the users response. With some C++ compilers, it is important that one outputs a line feed before the execution of the input statement. For example, one should use pairs of statements such as

```
cout  << " Enter the value of n : " << endl;
cin   >> n;
```

and not

```
cout  << " Enter the value of n : ";
cin   >> n;
```

6.5. File Input and Output. The method by which one reads or writes to files consists of declaring a stream and connecting it to an input or output file. To read or write to a file one just reads or writes to the stream in the same way as one uses the standard input and output streams `cin` and `cout`. In order to use the routines described in this section one must include the file `fstream.h`.

6.5.1. File Input. The declaration of an input file stream is a statement of the form

```
ifstream InFile;
```

Here we have given the stream a name `InFile` – it may be given any valid variable name. To connect the stream to a particular file one invokes the `open` member function;

```
InFile.open("finput", ios::in);
```

The first argument of the open function is a string, or a string pointer, containing the file name. The remaining arguments are flags which specify attributes to be associated with the stream. In the instance above `ios::in` specifies that the stream is open for input. (This is the default, but we include it here so that one can see how flags are specified.)

When dealing with external files it is useful to know when an error has occurred in the process of connecting (or opening) a file. This can be checked by seeing if the error flags of the stream have been set after the open invocation. For example, a construct we often use is

```
InFile.open("finput", ios::in);
if(!InFile)
{ cerr << " Error In Opening File For Input " << endl; exit(1); }
```

When applied to a stream the operator `!` returns a non-zero value if an error flag associated with that stream has been set. We use the "error output" stream, `cerr`, to send a message to the user and the command `exit(1)` to terminate the program.

Declarations and open invocations can be combined in a single statement, e.g. the statements

```
ifstream InFile("finput", ios::in);
if(!InFile)
{ cerr << " Error In Opening File For Input " << endl; exit(1); }
```

are equivalent to the declaration and open statements given previously.

To break the connection between a stream and a file one invokes the member function `close`. The statement

```
InFile.close();
```

closes the file `finput` which has been associated with the stream `InFile`.

Once a file stream has been declared, it may be connected to several different files within a program (one at a time of course). So, for example, after `InFile` is closed with respect to `finput` it may be connected to another file by invoking the `open` function once again. (It may be re-connected to `finput` if one so desires.)

6.5.2. File Output. The declaration of an output file stream is a statement of the form

```
ofstream OutFile;
```

Here we have given the name `OutFile` to the output file stream. To connect this stream to a file we use the `open` member function;

```
OutFile.open("foutput", (ios::out | ios::ate));
```

The statement connects `OutFile` to the file `foutput`. The flags `ios::out` and `ios::app` which have been "or"d together dictate that the file is opened for output and that the output will be appended to the end of the file. Any number of flags may be "or"d together and used as the second argument to the open command. Useful flags are given in the following table

Flag	Description
`ios::app`	Append data - write at end of file.
`ios::ate`	Seek end of file upon open.
`ios::out`	Open for output - default for ofstreams.
`ios::trunc`	Discard contents if file exits. Default with `ios::out` unless an append mode is set.
`ios::nocreate`	Does not create a file if open fails.
`ios::noreplace`	If files exits, open for output fails unless an append mode is set.

6.6. Formatting. There are two ways to format the output of variables. One way is to directly set the format state of the output stream, and the other way is through input/output manipulators. When one chooses to work with the format state, one sets flags and variables associated with the output stream, then, when any data is output to this stream, the resulting output will be formatted in the style which determined by these flags and variables. The use of input/output manipulators allows one to embed formatting commands within an input/output statement. (This is similar to other languages in which the formatting of output and the output statement are combined.) We favor the former because in numerical work one doesn't usually require a large number of different output formats - usually we choose one format for all of our output. Setting the output format state is a convenient way to have a single output format used.

The format operations we describe here apply to the pre-defined streams `cin` and `cout` as well as to any streams which have been declared within a program. In our examples we use the stream `cout` — for other streams one just substitutes their names in place of `cout`.

There are many variables and flags associated with the setting of the format state of an output stream. We will discuss those which we found most useful and not give a complete description. (For a complete description one should consult a C++ compiler manual.) The variables which we find most useful in numerical work are `precision` and `width`. The variable precision dictates the number of digits past the decimal point in the output format and the variable `width` is the number of spaces allowed in the output. To set these variables one uses statements of the form

```
cout.precision(4);       // sets the number of digits output to 4
cout.width(10);          // sets the width of the output field to 10
```

To determine their value at any time one can use statements of the form

```
n = cout.precision();   // n is set to the current number of digits output.
m = cout.width();       // m is set to the current output width.
```

Once the variable `precision` is set, its value remains constant until set again. The variable `width` is the minimum number of spaces which are allocated to output a variable. If more spaces are needed, then more spaces are allocated. (Values will never be truncated if the width is insufficient.) The variable `width` only applies to the very next output operation. Once the output operation is completed `width` is reset to it's default value.

Also useful are format flags associated with output streams. There are flags which allow one to specify either fixed or scientific notation, the justification of the output values (either right, left or internal) and whether or not trailing zeros should be shown. These flags are set using the member function `setf`.

As an example, to specify that one wants scientific notation, left justified values, and that trailing zeros be shown with the standard output one uses statements of the form

```
cout.setf(ios::left,       ios::adjustfield);
cout.setf(ios::showpoint,  ios::showpoint);
cout.setf(ios::scientific, ios::floatfield);
```

In the above statements the first argument to `setf` is the value of the flag (i.e. `ios::left` refers to a static enumerated constant) and the second argument is the flag to be set. The specification of scientific or fixed notation does not apply when an integer is output.

To set the format state of another stream, one just uses statements of similar to those above, but with a stream other than `cout`. For example

```
OutFile.setf(ios::fixed, ios::floatfield);
```

sets the output format associated with the stream `OutFile` to have fixed decimal point notation for floating point values. A format state is tied to a particular stream - so the setting of flags for one stream does not effect the flags for another stream.

An example which demonstrates these calls as well as indicating some of the other aspects of output is the following:

```
void main(){

  long  IntValA   =  12345;
  long  IntValB   =  54321;
 double RealValA  = 123.45;
 double RealValB  = 543.21;

 cout.precision(4);                              //  4 digits output
 cout.setf(ios::internal,  ios::adjustfield); // internal justification
 cout.setf(ios::showpoint,  ios::showpoint);  // show trailing zeros
 cout.setf(ios::scientific, ios::floatfield); // scientific notation

 cout << IntValA << "   " << IntValB << endl; // Both integers on one line with
                                              // character space between them.
 cout.width(15);                              // next output field will have 15 spaces
 cout << RealValA << endl;
 cout << RealValB << endl;
}
```

When the above program is run, the following output is obtained.

```
12345   54321
     1.2345e+02
5.4321e+02
```

Note that in the statement which outputs IntValA and IntValB that there is also the output of the string of spaces " ". This string is included so that spaces are printed between the two integers output. If it were not included the digits for the two values would run together.

Another way to set the format state of a stream is through the use of io manipulators. One can embed formatting commands within an output statement and thus avoid the use of the function `setf`. The manipulators which set the output precision and width are `setprecision(n)` and `setw(n)`. In `setprecision`, `n` is an integer specifying the number of digits past the decimal point and in `setw`, `n` is the number of spaces in the output field. Manipulators are "sent" to the output stream before the

output of a variable. As an example, in the output of a variable v, to specify 8 digits to be printed past the decimal point and a minimum of 10 spaces for the output field, one would use the command

```
cout << setprecsion(10) << setw(10) << v;
```

6.6.1. Format of UCLA++ Class Objects. When using the operator << on objects associated with the **UCLA++** classes, the format of the output is determined by the output state of the stream. Thus, the format of the output of these objects is the same as for the standard data types.

When one sends a vector to the output, then the values come out in a single column. (The transpose of the vector will come out in a single row). A matrix (declared as **matrix**, **complex_matrix** or **sparse_matrix**) is printed row by row, with the first row being at the top of a page. The element with lowest first and second index is in the upper left hand corner.

An object of type **array2D** is output so that the element with the lowest first and second index is in the *lower* left hand corner. The values associated with an increasing first index are printed across the page. The values associated with an increasing second index are printed up the page. (This is a natural form of output if the values of the array2D are associated with the set of points in the first quadrant of the Cartesian plane.)

The following code and it's output illustrates the difference between the output of a matrix and an object of type **array2D**.

```
void main()
{
    matrix M(3,3);
    M(1,1) = 1;
    M(2,2) = 2;
    M(3,3) = 3;
    cout << " matrix M Output " << endl << endl;
    cout << M << endl;

    array2D A(3,3);
    A(1,1) = 1;
    A(2,2) = 2;
    A(3,3) = 3;

    cout << " array2D A Output " << endl << endl;
    cout << A << endl;

}
```

The output of the code is

```
matrix M Output

 1  0  0
 0  2  0
 0  0  3

array2D A Output

 0  0  3
 0  2  0
 1  0  0
```

6.7. Graphics. The graphics commands fall into three categories - system control routines, plotting routines, and graphics style (or formatting) routines. The routines for system control and graphics style initialize the graphics system and set parameters used by the general plotting functions. These system and style routines are distinguished by the specifier `graphics::`. The general plotting routines are member functions of each of the classes, so the normal member function selection procedures are used. For example, `M.contour()`, where `M` is of type **array2D**, is a call which will contour the values associated with `M`.

6.8. System Control Routines.

6.8.1. Open, Close and Frame Routines. There are a few routines that control the plotting process which must be included in all programs. These are the routines which open and close the graphics system and advance the frame. A call to `open()` is necessary before any call to a plotting function. There is also a corresponding required call to `close()`. This routine is called after all the plot calls are completed but before the end of the program. All plotting output is drawn on the same frame (or window) until `frame()` is called. Thus, a call to `frame()` is necessary to separate different plots.

6.8.2. Plot Placement. The user can set the part of the window in which the plots are to appear with `set_frame`. By default, graphics leaves a tenth of the window width between the edge of the window and the edge of the plot on the left and right, and a tenth of the window height on the top and bottom. The arguments to `set_frame` are required to be between 0 and 1, and represent the position of the edge of the plot from the left or bottom edge of the window. For example, the default is obtained by the call:

```
graphics::set_frame(0.1,0.9,0.1,0.9)
```

6.8.3. Subplots. The subplot commands allow one to draw several graphs next to each other in the same frame. In order to produce subplots, the graphics system has to be in the subplot state. This state is entered with a call to `subplot_on(m,n)` and exited with a call to `subplot_off()`. Besides turning on subplot mode, `subplot_on(m,n)` also defines the size of the grid of subplots - i.e. `m` rows and `n` columns of plots. When in subplot mode, the user tells the system where to put the output of the plotting routines with a call to `subplot(i,j)`. All plot calls after the statement `subplot(i,j)` will be plotted in the subplot corresponding to ith row and the jth column of subplots. Plotting will continue in this subplot until another call to `subplot` or a call to `subplot_off()` is encountered. The statement `subplot_off();` returns the system to plotting to the full frame. Here is an example session in which the vector `v1` is plotted in the (1,1) subplot, `v2` in the (1,2) subplot, etc.

```
vector v1(40), v2(40), v3(40), v4(40);
     .
     .
v1,v2,v3 and v4 are given values
     .
     .
graphics::open();
graphics::subplot_on(2,2);
graphics::subplot(1,1);
v1.plot();
graphics::subplot(1,2);
v2.plot();
graphics::subplot(2,1);
v3.plot();
graphics::subplot(2,2);
v4.plot();
graphics::subplot_off();
graphics::frame();
graphics::close();
```

6.8.4. Auto-Scaling. The UCLA++ plotting routines perform autoscaling by default. One can change this with a call to `axis`. Passing the constant `AUTO_OFF` to `axis` will set an internal flag so that following plots (contour and surface plots excepted) will be plotted over the range held by the internal variables xmin, xmax, ymin, and ymax until the flag is reset with another call to `axis` with the argument `AUTO`. The call `axis(x1,x2,y1,y2)` is eqivalent to the call `axis(AUTO_OFF)` and a setting of the internal values `xmin = x1`, `xmax = x2`, `ymin = y1` and `ymin = y2`.

6.8.5. Setting Default Plotting Styles. The functions `set_plot_style` and `set_point_style` set the default plotting style. The changes are permanent (until reset) allowing the user to make one call which affect all the following plots. The three constants `CURVE`, `POINTS`, and `CURVE_AND_POINTS` are the possible arguments to `set_plot_style`. The routine `set_point_style` accepts any character. For example

```
graphics::set_point_style('*');
```

will set the point style to be an asterisk. The default plot style is `CURVE` and the default point style is a dot ".".

6.9. General Plotting Routines.

6.9.1. Curve Plotting. The routine for plotting a one dimensional set of values such as those in an object of type **vector** or **array1D**, as well as plotting a user specified function, is `plot`. Different actions are taken depending on the form of invocation and the type and number of arguments. The two simplest calls are

```
y.plot();
y.plot(x);
```

If `x` and `y` are of type **vector**, then the first call plots the values of `y` verses the index of `y` and the second plots the values if `y` verses the values of `x` - i.e. it plots the

values (x(i),y(i)) where i runs over the index of the vectors. The vectors must be the same length.

The default plotting style is used to create the plot unless the a plot style variable is specified as an argument to the plot command. The plotting style variable is one one of the constants CURVE, POINTS, or CURVE_AND_POINTS, resulting in the plot being drawn in the style described by the constant. For example to plot a vector y verses x with just the the data points being marked is accomplished with

```
y.plot(x,POINTS);
```

The marker used in the plot is the default marker - a value specified using the command graphics::set_point_style. One can also have a character as an input argument - for example, y.plot('*'), results in just the points being plotted and marked by the character *. The use of a plot or line style specification within a given plotting call does not set the default state of the system. To set the default state, see the previous section "Setting Default Plotting Styles".

In order to allow users to plot functions of a single variable there is the command graphics::plot(f, *) where f is a user defined function. Table 5.16 "Plotting Routines for Class Objects" describes the input options and syntax for this routine.

6.9.2. Contour Plotting. The routines for contouring the two-dimensional sets of values in an **array2D**, **matrix** or **sparse_matrix** object are provided by a call to the routine contour. By specifying different input arguments, different contouring styles can be selected. These styles and the required input arguments are specified in Table 5.16 in the Class Summaries section.

Since the different contouring styles are selected by the arguments to the contour call some problems can arise. In particular, this is the case when a contouring style is determined by its value as well as its type. For example consider the statement

```
M.contour(2);
```

There is a possible ambiguity — will the contour plot be drawn with two contours or will the contour plot be drawn with contours associated with values which differ by 2 (i.e. a contour increment of 2)? In this case the contour plot would be drawn with two contours. If the argument is a integer (int or long) then the argument is assumed to be the number of contours, while if the argument is a floating point (float or double) then the argument is assumed to be the desired spacing between contours. For contour plot with contours associated with values which are 2.0 units apart, the call would be

```
M.contour(2.0);
```

If the contour routines do not do what you expect or if you receive a compiler error, you should try explicitly casting the arguments sent to contour. For example,

```
M.contour(int(num_lines), low, hi);
```

If one is contouring an array of values in which the number of data points associated with one index is not the same as the second index (e.g. a non-square **matrix** or **array2D**), then the contour plot which results will have the same aspect ratio as the set of values which are being contoured. If one desires to have the contour plot fill out the frame independently of the number of data values associated with each index, one executes the call graphics::turn_off_contour_scaling();. One can return to the default state with the command graphics::turn_on_contour_scaling();.

6.9.3. Surface Plotting. A surface, or perspective, plot of a two dimensional set of values is created with the `surface` command. The user can choose the angle from which to view a surface plot by setting the horizontal and vertical viewing angles. The horizontal angle is measured in degrees counterclockwise from the positive x axis to the projection of the line of sight in the x–y plane. The vertical angle is the number of degrees up from the x–y plane to the line of sight. These angles can be set permanently with `set_horizontal_angle` and `set_verticle_angle` or the can be passed to the routine `surface` as parameters. The default for both angles is 30 degrees.

6.9.4. Character and Line Drawing Routines. The routine `draw_string` is the general string plotting routine. This routine allows the user to put any string of any size anywhere in the window at any orientation. (Specific routines for drawing labels and titles is described below). The first two arguments to `draw_string` are the x and y values of the point in the viewing window which is used as a reference point to draw the string. The coordinates of the reference point are expected to be between 0 and 1, with 0 representing the left or bottom edge of the window and 1 representing the right or top edge. For example the call

```
graphics::draw_string(0.5,0.5,"My string",0.015,0,0);
```

would draw `My string` centered in the window.

The fourth argument is the size in which to draw the string. If size is between 0 and 1, then it specifies the size of the characters as a fraction of the plotting frame. If size is greater than 1 then it specifies the size of the characters in plotter units. Size can also be negative, in which case the absolute value is the size as a multiple of the digitized size. Using values between 0 and 1 is usually easiest and gives the user the most predictable results.

The fifth and sixth arguments define the orientation of the string around the point specified by the first two arguments. The fifth argument is an angle in degrees from the positive x axis specifying the direction the string is to be drawn. The last argument specifies the justification of the string about the reference point. For example, the call

```
graphics::draw_string(0.5,0.5,"My string",0.015,90,0);
```

will produce a string which is rotated 90 degrees about the reference point, the middle of the window.

A value of -1 for the justification parameter (the sixth argument) causes the left edge of the string to be placed at the reference point, a value of 1 causes the right edge to be placed at the reference point, and a value of 0 causes the string to be centered about the reference point. When any other value is passed, the position of the reference point related to the string is obtained by interpolating along the line defined by the points associated with the values -1 and 1. For example, passing -0.5 causes the string to be printed such that the reference point is half way between the left edge of the string and the midpoint of the string.

6.9.5. Lables and Titles. The routines `title`, `label_x`, and `label_y` are provided to simpify the construction of titles and labels. These routines are designed to be easy to use - if they do not do what you desire, then you should use the routine `draw_string` described above. The parameters for these routines are given in Table 5.17, "Character Drawing and Labelling Routines". There is an optional second argument for the routine `label_y`. If this argument has the value 1 then the string will be

drawn on the right side of the graph. Passing any other value or not passing anything causes the string to be drawn on the left side of the graph. This is useful since the left side of the graph is often filled with numerical labels.

6.9.6. Formatting. A variety of routines are provided for the formatting of plots. These are described in Table 5.18 "Formatting Routines".

The user can choose between linear and logarithmic scales with a call to `set_scale(scale_type)`. One of the constants `LIN_LIN`, `LIN_LOG`, `LOG_LIN`, or `LOG_LOG` is passed to `set_scale(scale_type)` as an argument. The first part of the constant is the scale to use for the x axis, and the second part is the scale for the y axis. For example, the call

```
graphics::set_scale(LIN_LIN);
```

results in plots with a linear scale on both axes, the default, while the call

```
graphics::set_scale(LOG_LIN);
```

results in a logarithmic scale on the x axis and a linear scale on the y axis.

When using a linear scale, the axes are divided into intervals for labeling. These divisions are controlled with the `set_ticks` functions (the functions don't actually set the number of ticks like the names imply, but the number of intervals). The major divisions are the number of divisions of the entire axis and are separated by labeled tick marks, while the minor divisions are the number of subdivisions of the major divisions and are separated by unlabeled tick marks. The default for both axes is 5 major and 2 minor divisions.

By default, both the axes and labels are drawn when a plot function is called, but the user can tell the system to do otherwise. Unfortunately, labels can't be drawn without the axes being drawn, so a call to `turn_off_axis()` also turns off the labels and a call to `turn_on_labels()` also turns on the axes.

When the axes are drawn, each axis is drawn in one of three styles: perimeter, grid, or floating. Perimeter is the default on both axes and causes the axis to appear on the edges of the plot. For example, if the x axis is set to perimeter, a horizontal line with the appropriate tick marks will be drawn on the top and bottom of the graph. Grid causes lines to be drawn across the picture perpendicular to the axis which is set to grid. The last style, floating, causes the axis to be plotted within the graph (if possible) at the position specified by the appropriate intercept stored in internal variables. For example, if the x axis is set to floating and the y intercept is equal to two, then the x axis will be the line $y = 2$.

The intercepts (default = 0,0) are set with the function `set_intercepts(x,y)` and the axis type is set with the function `set_axis_type(type)`. The argument to `set_axis_type` is a constant constructed out of the words `PERIMETER`, `GRID`, and `FLOATING` in the same way as the constants passed to `set_scale`. For example, `PERIMETER_PERIMETER`, `PERIMETER_GRID`, and `GRID_FLOATING` are all acceptable arguments.

6.10. Calling Fortran Routines from C++. The purpose of this discussion is to provide information which makes the procedure of calling external Fortran routines intelligable, and hopefully, assist one in sorting out problems which occur. We also describe the procedure for passing the data associated with objects of the matrix and vector classes to Fortran routines. Several samples of calling Fortran subroutines and functions are given in the samples section. We recommend that one takes a look at these samples before reading this section in order to familiarize oneself with the overall form of the syntax and structure of the code.

In general the use and syntax of inter-language calls is compiler specific. What we present here works with the SUN CC, GNU g++, and Macintosh MPW C++ compilers. We expect that with small modifications the procedures discussed here will be applicable to other compilers - one should consult compiler documentation for details. We do not discuss issues related to linking - i.e. one must often link with the appropriate Fortran libraries. If you are using the UCC or Ug++ commands, then this is automatically done. If you are using your own commands, then you will likely have to consult the compiler manuals to determine which libraries need to be linked.

In order to call a Fortran function or a subroutine from C++ there are four basic facts about Fortran and C++ which one should be aware of;

1. Fortran passes arguments by reference.
2. Fortran passes an array of numbers by passing a reference to the first element of the array.
3. There are equivalences between the standard C++ data types and the standard Fortran data types.
4. The syntax for declaring and calling a Fortran routine from C++ is non-standard.

An implication of the first fact is that in the declaration of a Fortran routine within a C++ program *every* argument must be a pointer. For example if the subroutine `fortsub_` which has as arguments two double precision numbers then it will be declared as

```
extern "C" void fortsub_( double* x, double* y);
```

(The meaning of `extern "C"` and the underscore in the name will be discussed later.) To call this subroutine, one can use either pointer arguments or reference arguments as the following code demonstrates

```
double x = 1.0;
double y = 2.0;

fortsub_(&x, &y);
```

or

```
double x = 1.0;
double y = 2.0;
double* xptr = &x;
double* yptr = &y;

fortsub_(xptr, yptr);
```

The second basic fact dictates how one passes the array of numbers associated with an object of matrix or vector class. An external Fortran routine expects a pointer (an address) to the first element of the array of numbers associated with a vector or a matrix. To obtain this pointer we use the member function `get_data_ptr` associated with the matrix and vector classes. In the following example we illustrate the procedure. (We also provide the "equivalent" Fortran code to emphasize the similarities of the calling procedures.)

C++

```
long n;
long m;
matrix a(m,n);
double* aptr = a.get_data_ptr();
fortsub_(aptr, &m, &n);
```

Fortran

```
integer*4 n
integer*4 m
dimension a(m,n)

call fortsub(a,m,n)
```

In the C++ code, one needs to declare the routine `fortsub_` before use. In this instance the declaration would take the form

```
extern "C" void fortsub_( double* aptr, long* m, long* n);
```

The implication of the third fact is that one must know the equivalence between the data types in C++ and Fortran and pass equivalent data types in subroutine and function calls. The equivalence of data types does not appear to be standardized, however, we have found the information contained in the following table to be accurate on the compilers which we have encountered.

C++	Fortran
`short`	`INTEGER*2`
`int`	`INTEGER*4`
`long`	`INTEGER*4`
`float`	`REAL*4`
`double`	`REAL*8`
`char c[n]`	`CHARACTER*n`

Unfortunately, there is no simple equivalence between the C++ data type `complex` and the corresponding `complex` data type in Fortran. One can still pass complex numbers, vectors, and matrices, but this requires a few more steps. The procedure for passing objects based on the complex data type is discussed at the end of this section.

The fourth fact, that the syntax for declaration and calling an external Fortran routine, is non-standard is very annoying. However, there appears to be a general similarity between compilers. Consider again the declaration used in the example above -

```
extern "C" void fortsub_( double* aptr, long* m, long* n);
```

Most compilers use this syntatic form for declaration — the statement `extern "C"` is used even though we are using an external Fortran routine. (The extern statement

tells the compiler that the particular routine will be available at link time.) Also note the underscore in the name of the external routine. This is another artifact of inter-language calls. For a Fortran routine named `fortsub`, it is declared and used in the C++ code as `fortsub_`. (This last aspect is definitely non-standard, as some Fortran compilers allow one the option of specifying that an underscore is not appended to a routines name in the object code - thus one can use such routines without adding underscores to their names.) Since the syntax is compiler specific one should consult the manuals for a particular compiler for information. We have found that manuals for Fortran compilers often contain more information about inter-language calls than manuals for C++ compilers.

The process of calling an external Fortran routine thus consists of several steps. First the routine must be declared as `extern "C"` and an underscore should be appended to its name. In the C++ program the routine name with the underscore is used. All arguments passed to the external routine must be pointers. If one is dealing with objects of type **matrix**, **vector**, **array1D** or **array2D**, then the pointer to the data associated with these objects is obtained with the member function `get_data_ptr()`.

6.10.1. Use of the Complex Data Type. To pass a variable of complex data type to a Fortran routine is a bit more complicated than that associated with the other data types. The procedure we employ consists of packing the complex values into a **matrix** or **vector** and then passing this to the Fortran routine using the procedures described above. This works because a complex value in Fortran is stored as two consecutive real values - the real part followed by the imaginary part. Thus, we pass our complex values by constructing an array of real values in which have placed the real and imaginary parts of our complex values.

The routines to carry out this "packing" of the complex data are `fortran_pack` and `fortran_unpack`. The following example illustrates their use

```
extern "C" void matvec_(long* m, long* n, double* A, double* x);

void main()
{
long m = 4; long n = 4;
complex_matrix A(m,n);
complex_vector x(n); complex_vector y(n);


matrix Atmp = complex_matrix::fortran_pack(A);  // pack A into a double matrix
matrix xtmp = complex_vector::fortran_pack(x);  // pack x into a double vector

double *Aptr = Atmp.get_data_ptr();             // get the data pointers for the
double *xptr = xtmp.get_data_ptr();             // packed matrices

matvec_(&m, &n, Aptr, xptr);                    // call the routine

y  = complex_vector::fortran_unpack(xtmp);      // unpack the result xtmp
}
```

Note that one uses different packing routines for an object of type **complex_matrix** and an object of type **complex_vector**.

If one has a complex value to pass to a routine, then one creates a vector of length 2 with the first component of the vector being the real part of complex value and the second component being the imaginary part of the complex value. This vector is then passed to the routine using the procedures described above. Examples of this are illustrated in the program samples provided in the section "Samples".

7. Samples. In this section we provide samples of code which demonstrate how to carry out tasks which are associated with scientific programming - i.e. opening and closing files, using external routines, using the graphics capabilities etc. The following table lists the samples which are provided.

Sample Name	Description
FILE_OUPUT_1	Demonstrates output to a file.
FILE_OUPUT_2	Demonstrates internal creation of a filename with subsequent output to that file.
FILE_INPUT_1	Demonstrates input from a file.
GRAPHICS_1	Demonstrates the plotting of a user defined function.
GRAPHICS_2	Demonstrates the plotting of one vector vs. another.
GRAPHICS_3	Demonstrates contour and surface plotting.
GRAPHICS_4	Demonstrates the use of subplot and contour plotting.
SPARSE_1	Demonstrates the use of **sparse_matrix** for solving a tri-diagonal system of equations.
FORTRAN_CALL_1	Demonstrates a call to an external Fortran subroutine which involves vectors.
FORTRAN_CALL_2	Demonstrates a call to an external Fortran function which returns a value.
FORTRAN_CALL_3	Demonstrates a call to an external Fortran subroutine which involves matrices.
FORTRAN_CALL_4	Demonstrates a call to an external Fortran subroutine which involves complex matrices.
FORTRAN_CALL_5	Demonstrates a call to an external Fortran subroutine which involves complex vectors and values.
FORTRAN_CALL_6	Demonstrates the creation of a C++ routine which calls a Fortran LAPACK routine.

```
/*                      FILE_OUPUT_1

        THIS SAMPLE DEMONSTRATES OUTPUT TO A FILE

    The output file stream Fout is declared.

    The external file named sample.out is associated with
    this stream (with error checking).

    The format state of the output stream is set.

    Values are output to the file using the operator <<.

    The output file stream is closed
    (i.e. dissassociated from the file sample.out).
*/
#include<fstream.h>
#include<stdlib.h>

void main()
{
    char str[20] = "Sample_Values";
    double x = 3.0;   double y = 6.0;   long k   = 30;
//
//  Open Output File
//
    ofstream Fout;
    Fout.open("sample.out",ios::out);
    if (!Fout )
    { cerr << " Error in Opening Ouput File "; exit(1); }
//
// Set Ouput File Formatting - scientific notation with 8 digits past
//                             the decimal point
//
    Fout.setf(ios::scientific, ios::floatfield);
    Fout.precision(8);
//
//   Output Values
//
    Fout << str << endl;
    Fout << x   << "   " << y << endl;
    Fout << k << endl;
//
    Fout.close();
}
```

```
/*                      FILE_OUTPUT_2

    A FILENAME IS INTERNALLY CREATED AND DATA IS OUTPUT THAT FILE

    The user is requested to input a step number

    A character string (for a file name) of the form sample.***
    is created where *** is the step number.

    An output stream is associated with a file of the constructed name.

    A message is output to the file.
*/
#include<fstream.h>
#include<strstream.h>
#include<stdlib.h>

void main()
{
    char fname[30];
    long step_number;
//
//  Obtain Step Number
//
    cout << " Enter Step Number " << endl;
    cin  >> step_number;
//
//  Create Output File Name of the form sample.***
//  where *** is the step number.
//
    ostrstream(fname,sizeof(fname)) << "sample." << step_number << ends;
//
//  Declare and open the output file stream Fout
//
    ofstream Fout;
    Fout.open(fname,ios::out);
    if (!Fout )
    { cerr << " Error in Opening Output File "; exit(1); }
//
//  Write a message to the file and close
//
    Fout << " The name of this file should be sample." << step_number << endl;
    Fout.close();
}
```

```
/*                      FILE_INPUT_1

          THIS SAMPLE DEMONSTRATES INPUT FROM A FILE

    The input file stream Fin is declared.

    The external file named sample.out created by SAMPLE F.1
    is associated with this stream (with error checking).

    Values are input from the file using the operator >>.

    Values are written to the standard output using  <<.

    The input file stream is closed.
*/
#include<fstream.h>
#include<stdlib.h>

void main()
{
    char  str[20];
    double x;   double y;   long   k;
//
//  Open Input File
//
    ifstream Fin;
    Fin.open("sample.out");
    if (!Fin )
    { cerr << " Error in Opening Input File "; exit(1); }
//
//   Input information
//
    Fin >> str;
    Fin >> x   >>  y;
    Fin >> k;
//
//  Output values to standard output
//
    cout.setf(ios::scientific, ios::floatfield);
    cout.precision(8);

    cout <<  str << endl;
    cout << x << "   " <<  y << endl;
    cout << k << endl;
//
    Fin.close();
}
```

```
/*                        GRAPHICS_1

   THIS SAMPLE DEMONSTRATES THE PLOTTING OF A USER SPECIFIED FUNCTION

*/
#include "UCLA++.h"

double f(double x){ return sin(x*x);};
void main()
{
    graphics::open();

    graphics::axis(-3.0,3.0,-2.0,2.0);
    graphics::plot(f);

    graphics::frame();
    graphics::close();
}
```

```
/*                        GRAPHICS_2

   THIS SAMPLE DEMONSTRATES THE PLOTTING OF ONE VECTOR VS. ANOTHER
*/
#include "UCLA++.h"

double f(double x){ return sin(x*x);};
void main()
{
    long n   =  50;
    double a = -3.0;
    double b =  3.0;
    double h = (b - a)/double(n);

    vector v(0,n);
    vector x(0,n);
    for(long i = 0; i <= n  ; i++)
    {
      x(i) = a + i*h;
      v(i) = f(a + i*h)
    }

    graphics::open();
    graphics::axis(-3.0,3.0,-2.0,2.0);

    v.plot(x);
    graphics::frame();

    graphics::set_point_style('+');
    v.plot(x,CURVE_AND_POINTS);
    graphics::frame();

    v.plot(x,'*');
    graphics::frame();
    graphics::close();
}
```

```
/*                        GRAPHICS_3

   THIS SAMPLE DEMONSTRATES CONTOUR AND SURFACE PLOTTING
*/
#include "UCLA++.h"

double f(double x, double y)
{return cos(4.0*y)*sin(5.0*sqrt(x*x + y*y));};

void main()
{
    long i; long j;
    long m = 10;
    long n = 10;
    double hx = .1; double hy = .1;
    double x_i; double y_j;

    array2D A(-m,m,-n,n);

    for(i = -m; i<= m; i++)
    for(j = -n; j<= n; j++)
    { x_i = i*hx;
      y_j = j*hy;
      A(i,j) = f(x_i, y_j);
    }

    graphics::open();
    graphics::turn_off_labels();  // Otherwise contour labels lie on top of
                                  // axis labels.
    A.contour();
    graphics::frame();

    A.contour(.2);
    graphics::frame();

    A.surface();
    graphics::frame();

    graphics::close();
}
```

```
/*                      GRAPHICS_4

   THIS SAMPLE DEMONSTRATES SUBPLOTS AND CONTOUR PLOTTING
*/
#include "UCLA++.h"

double f(double x, double y)
{return cos(4.0*y)*sin(5.0*sqrt(x*x + y*y));};

void main()
{
    long i; long j;
    long m = 10;
    long n = 10;
    double hx = .1; double hy = .1;
    double x_i; double y_j;

    array2D A(-m,m,-n,n);

    for(i = -m; i<= m; i++)
    for(j = -n; j<= n; j++)
    { x_i = i*hx;
      y_j = j*hy;
      A(i,j) = f(x_i, y_j);
    }

    graphics::open();
    graphics::turn_off_labels();

    graphics::subplot_on(2,2);  // Use a 2 by 2 grid of subplots

    graphics::subplot(1,1);
    A.contour();                // Default contour call

    graphics::subplot(1,2);
    A.contour(.2);              // Use contour spacing of .2

    graphics::subplot(2,1);
    A.contour(10);              // Use 10 contours between max and min

    graphics::subplot(2,2);
    A.contour(10,-2.0,2.0);     // Use 10 contours between -2 and 2

    graphics::subplot_off();
    graphics::frame();

    graphics::close();
}
```

```
/*                          SPARSE_1

   THIS SAMPLE DEMONSTRATES THE USE OF THE SPARSE MATRIX CLASS

    The size of the matrix, n, is input by a user.

    A tri-diagonal system of equations of size n is constructed.

    A system of equations involving this tridiagonal matrix is
    solved.

*/
#include "UCLA++.h"
void main()
{
    long i;
    long npanel;
    double h;

    cout << "Enter Number of Panels " << endl;
    cin  >> npanel;

    sparse_matrix A(npanel+1,npanel+1);
    vector b(npanel+1);
    vector x(npanel+1);

    h = 1.0/double(npanel);

    A(1,1) = -2.0/(h*h);
    A(1,2) =  1.0/(h*h);
    for(i = 2; i <= npanel; i++)
    {
       A(i,i-1) =  1.0/(h*h);
       A(i,i)   = -2.0/(h*h);
       A(i,i+1) =  1.0/(h*h);
    }
    A(npanel+1,npanel)     =  1.0/(h*h);
    A(npanel+1,npanel+1)   = -2.0/(h*h);

    b(npanel/2) = 1.0;
    x = A/b;
    cout << (A*x - b).norm(2) << endl;
}
```

```
/*                    FORTRAN_CALL.1

THIS SAMPLE DEMONSTRATES AN EXTERNAL FORTRAN CALL INVOLVING VECTORS

   The external fortran subroutine addvec is declared.

   Two vectors v and w are declared and initialized.

   Pointers to the data associated with v, w and r are obtained.

   The external fortran subroutine addvec is called.
*/
#include "UCLA++.h"

extern "C" void addvec_(long*n, double* v, double* w, double* r);

void main()
{
    long i;
    long n = 5;
    vector v(n); vector w(n); vector r(n);
    for(i = 1; i<=n; i++)
    {v(i) = i;  w(i) = i;}

    double *vptr = v.get_data_ptr();
    double *wptr = w.get_data_ptr();
    double *rptr = r.get_data_ptr();
    addvec_(&n, vptr, wptr, rptr);

    cout << " v "     << endl << v << endl;
    cout << " w "     << endl << w << endl;
    cout << " v + w " << endl << r << endl;
}
```

Fortran Code (Compiled Separately)

```
      subroutine addvec(n,v,w,r)
      real*8 v(n),w(n),r(n)
c
c     This routine adds the two vectors v and w and returns the
c     result in r.
c
      do 100 i=1,n
       r(i) = v(i) + w(i)
100   continue
      return
      end
```

```
/*                    FORTRAN_CALL.2

  AN EXTERNAL FORTRAN CALL RETURNING A VALUE IS DEMONSTRATED

   The external fortran function dotvec is declared.

   Two vectors v and w are declared and initialized.

   Pointers to the data associated with v, w are obtained.

   The external fortran function dotvec is called and the result output.s
*/
#include "UCLA++.h"
extern "C" double dotvec_(long*n, double* v, double* w);

void main()
{
    long i;
    double dotval;
    long n = 5;
    vector v(n); vector w(n);
    for(i = 1; i<=n; i++)
    {v(i) = i;  w(i) = i;}

    double *vptr = v.get_data_ptr();
    double *wptr = w.get_data_ptr();
    dotval = dotvec_(&n, vptr, wptr);

    cout << " v "     << endl << v << endl;
    cout << " w "     << endl << w << endl;
    cout << " <v, w> = "  << dotval << endl;
}
```

Fortran Code (Compiled Separately)

```
        real*8 function dotvec(n,v,w)
        real*8 v(n),w(n)
        real*8 sum
c
c  This routine computes the dot product of the vectors v and w
c
        sum = 0.0d00
        do 100 i=1,n
          sum = sum +(v(i) * w(i))
100     continue
        dotvec = sum
        return
        end
```

```
/*                    FORTRAN_CALL.3

  AN EXTERNAL FORTRAN CALL INVOLVING MATRICES IS DEMONSTRATED

   The external fortran function addmat is declared.

   Two matrices A and B are declared and initialized.

   Pointers to the data associated with A, B are obtained.

   The external fortran function addmat is called.
*/
#include "UCLA++.h"

extern "C" void addmat_(long* m, long* n, double* A, double* B, double* C);

void main()
{
    long m = 4; long n = 4;
    matrix A(m,n); matrix B(m,n); matrix C(m,n);

    A =     matrix::Id(m);
    B = 3.0*matrix::Id(m);

    double *Aptr = A.get_data_ptr();
    double *Bptr = B.get_data_ptr();
    double *Cptr = C.get_data_ptr();
    addmat_(&m, &n, Aptr, Bptr, Cptr);

    cout << " A "     << endl << A << endl;
    cout << " B "     << endl << B << endl;
    cout << " A + B " << endl << C << endl;
}
```

Fortran Code (Compiled Separately)

```
        subroutine addmat(m,n,v,w,r)
        real*8 v(m,n),w(m,n),r(m,n)
c
c   This subroutine adds the matrices v and w and returns the result in r.
c
        do 100 j=1,n
        do 100 i=1,m
        r(i,j) = v(i,j) + w(i,j)
100     continue
        return
        end
```

```
/*                    FORTRAN_CALL.4

  AN EXTERNAL FORTRAN CALL INVOLVING COMPLEX MATRICES IS DEMONSTRATED

   The external fortran subroutine addcmat is declared.

   Complex matrices A, B and C are declared and initialized.

   Temporary matrices of type double are associated with matrices A,B, and C.

   Pointers to the data associated with the temporary matrices are obtained

   The external fortran subroutine addcmat is called.

   The temporary matrix Ctmp is "unpacked" to obtain the result of the routine.
*/
#include "UCLA++.h"

extern "C" void addcmat_(long* m, long* n, double* A, double* B, double* C);

void main()
{
    long m = 4; long n = 4;
    complex_matrix A(m,n); complex_matrix B(m,n);
    complex_matrix C(m,n);

    A = matrix::Id(m);
    B = complex(1.0,1.0)*matrix::Id(m);

    matrix Atmp = complex_matrix::fortran_pack(A);
    matrix Btmp = complex_matrix::fortran_pack(B);
    matrix Ctmp = complex_matrix::fortran_pack(C);

    double *Aptr = Atmp.get_data_ptr();
    double *Bptr = Btmp.get_data_ptr();
    double *Cptr = Ctmp.get_data_ptr();
    addcmat_(&m, &n, Aptr, Bptr, Cptr);

    C  = complex_matrix::fortran_unpack(Ctmp);

    cout << " A "     << endl << A << endl;
    cout << " B "     << endl << B << endl;
    cout << " A + B " << endl << C << endl;
}
```

Fortran Code (Compiled Separately)

```
      subroutine addcmat(m,n,v,w,r)
      complex*16 v(m,n),w(m,n),r(m,n)
c
c  This subroutine adds the two complex matrices v and w and
c  returns the result in r.
c
      do 100 j=1,n
      do 100 i=1,m
        r(i,j) = v(i,j) + w(i,j)
100   continue
      return
      end
```

```
/*                    FORTRAN_CALL.5

  AN EXTERNAL FORTRAN CALL INVOLVING COMPLEX VECTORS AND COMPLEX VALUES
                        IS DEMONSTRATED

   The external fortran function alphaxy is declared.

   Temporary vectors of type double are associated with the complex
   vectors x and y and the complex values alpha and beta.

   Pointers to the data associated with the temporary vectors is obtained.

   The external fortran subroutine alphaxy is called.

   The values in the vector btmp are converted to a complex value.
*/
#include "UCLA++.h"

extern "C" void alphaxy_(double* alpha, double* beta, long* n, double* x, double* y);

void main(){

    long n = 3;
    complex_vector x(n); complex_vector y(n);
    complex alpha;       complex beta;

    for(long i = 1; i<= n; i++)                      // initialize x,y and alpha
    {x(i) = complex(i,i); y(i) = complex(i,i);}
    alpha = complex(0.0,2.0);

    vector xtmp = complex_vector::fortran_pack(x);
    vector ytmp = complex_vector::fortran_pack(y);

    vector atmp(2); vector btmp(2);                   // create temporary vectors
    atmp(1) = real(alpha);  atmp(2) = imag(alpha);    // put complex value into atmp

    double* xtmp_ptr = xtmp.get_data_ptr(); double* ytmp_ptr = ytmp.get_data_ptr();
    double* atmp_ptr = atmp.get_data_ptr(); double* btmp_ptr = btmp.get_data_ptr();

    alphaxy_(atmp_ptr, btmp_ptr, &n, xtmp_ptr, ytmp_ptr);
    beta = complex(btmp(1),btmp(2));                 // convert btmp to a complex
                                                     // value.
    cout << " x  "   << endl << x      << endl;
    cout << " y  "   << endl << y      << endl;
    cout << "alpha " << endl << alpha  << endl;
    cout << "alpha < x , y > " << endl << beta ;
}
```

Fortran Code (Compiled Separately)

```
      subroutine alphaxy(alpha,beta,n,x,y)
      complex*16 alpha,beta
      complex*16 x(n),y(n)
      complex*16 sum
c
c     This routine computes the quantity
c     alpha times the complex inner product of x and y
c
      sum = cmplx(0.0,0.0)
      do 100 i=1,n
       sum = sum + x(i)*conjg(y(i))
100   continue
      beta = alpha*sum
      return
      end
```

```
/*                    FORTRAN_CALL.6

  AN EXTERNAL FORTRAN CALL TO A LAPACK ROUTINE IS DEMONSTRATED

*/
#include "UCLA++.h"
//
// External declaration of the routine dgesvx.
//
extern "C" void dgesvx_(char* fact, char* trans,long* n, long* nrhs,double* a,
long* lda, double* af, long* lda,long*ipiv,char* equed, double* r, double* c,
double*b, long* ldb, double* x, long* ldx, double* rcond, double* ferr,
double* berr, double* work, long* iwork, long* info);
//
//  Routine solvesys is a C++ binding routine to the LAPACK routine
//  dgesvx.
//
matrix solvesys(matrix A, matrix B)
{
//
//  Get information about input matrices
//
    double*  Aelements = A.get_data_ptr();
    long     Acolumns  = A.get_num_columns();
    long     Arows     =  A.get_num_rows();

    double*  Belements = B.get_data_ptr();
    long     Bcolumns  = B.get_num_columns();
    long     Brows     =  B.get_num_rows();
//
//  Create work temporaries
//
    long* ipiv    = new long[Arows];
    double* r     = new double[Arows];
    double* c     = new double[Arows];
    double* ferr  = new double[Bcolumns];
    double* berr  = new double[Bcolumns];
    double* work  = new double[4*Arows];
    long* iwork   = new long[Arows];
//
//  Create matrices for the factored form of A and the solution X
//
    matrix AF(Arows,Acolumns);
    matrix X(Brows, Bcolumns);

    double*  Xelements=X.get_data_ptr();
    long Xcolumns =X.get_num_columns();
    long Xrows    =X.get_num_rows();
```

```
    double*  AFelements=AF.get_data_ptr();
    long AFcolumns =AF.get_num_columns();
    long AFrows    =AF.get_num_rows();
//
//  Set parameters for the LAPACK solver
//
    char fact='E';
    char trans='N';
    char equed=' ';
//
    long n       = Arows;
    long nrhs    = Bcolumns;
    long lda     = Arows;
    long ldaf    = Arows;
    long ldb     = Arows;
    long ldx     = Arows;
    double rcond = 0;
    long info    = 0;

    dgesvx_(&fact,&trans,&n,&nrhs,Aelements,&lda,AFelements,&ldaf,ipiv,
           &equed,r,c,Belements,&ldb,Xelements,&ldx,&rcond,ferr,berr,
           work,iwork,&info);
//
//  Error Checking
//
   if(info != 0)
   {    if(info <= A.get_num_rows())
        {cerr << "          Matrix Singular " << endl;
         cerr << " LU Factorization Stopped at Step " << info << endl;
         cerr << "        No Solution Returned " << endl;}

        if(info = (A.get_num_rows() +1))
        {cerr << " Matrix Singular or Badly Conditioned " << endl;
        cerr << " Computed Solution May Be Inaccurate "  << endl;
        cerr << " Condition Number = " << 1.0/rcond << endl;}
    }
//
// clean-up !!! REMEMBER TO DO THIS !!!!
//
    delete [] iwork; delete []  work;
    delete []  berr; delete []  ferr;
    delete []     c; delete []     r;
    delete []   ipiv;
//
    return(X);
}
```

```
//
//   Test program for the routine solvesys
//
void main()
{
//
// This program computes the inverse of a 5 by 5 Hilbert matrix
// by solveing A*X = B where B is the 5 by 5 identity matrix.
// The routine used to compute the solution is solvesys - a routine
// which is bound to the LAPACK routine dgesvx.
//
    long n = 5;
    matrix A(n,n); matrix B(n,n); matrix X(n,n);

    for(long i = 1; i <= n; i++)
    for(long j = 1; j <= n; j++)
    {A(i,j) = 1.0/double(i+j);}

    B = matrix::Id(n);

    X=solvesys(A,B);

    cout << (B - A*X).norm(1) << endl;
}
```

References

The primary C++ references for this work are

[1] Lippman, Stanley B., *C++ Primer* 2nd ed., Addison-Wesley, 1991.

[2] Stroustrup, Bjarne, *The C++ Programming Language* 2nd ed., Addison-Wesley, 1992.

We found the C++ language guides which come with C++ compilers for PC's very useful (especially their discussion of input/output). Two that were used most often are

[3] *Borland C++ 3.1 Programmers Guide*, Borland International, 1992.

[4] *Macintosh Programmers Workshop C++ Reference*, ver. 3.2, 030-1953-A, Developer Technical Publications, Apple Computer, Inc. 1992.

The routines for the sparse matrix computations are currently provided by binding the class library routines to those in the PET C library developed by Willian Gropp and Barry Smith. A reference to this C library (and others) is

[5] Gropp, William and Smith, Barry, ''Simplified Linear Equation Solvers Users Manual'', ANL-9318, Argonne National Laboratory, Argonne, Illinois.