August 5, 1992

Rogue Wave Math Libraries

Eric Olson

National Radio Astronomy Observatory[1] Summer Student

In an Object-Oriented environment the most important portion of the software system is the set of class libraries. One of the most basic types of class library is one that encapsulates mathematical processing. Doing numerical computations in an O-O environment involves special challenges. Two articles by Keffer (C++ Journal, Vol. 1, No. 4, 1991 and Vol. 2 No. 1 1992) provide brief discussions of the approach towards numerics embodied in the Rogue Wave libraries. The latter constitute one of the first publicly available systems devoted primarily to numerics, and are oriented to use with C++ compilers. Both early public domain and commercially available versions are available. At NRAO Socorro and Green Bank the commercial versions have been purchased and installed. They can be used by anyone who wishes to do numerical computations in C++.

This document is an introduction to the Rogue Wave Math Libraries Math.h++, Matrix.h++, and Linpack.h++ for C++. It contains information on how to compile a program that implements these libraries, as well as several examples of C++ programs. Examples have been taken from the three manuals *Math.h++*, *Matrix.h++*, and *Linpack.h++*. Specifically, the first eight are taken verbatim from section 14 of *Math.h++*. Some familarity with C++ programming is assumed; the beginning programmer may have difficulty with some of the terminology or code in this document.

Compilation:

To compile any C++ file that includes the RW math libraries, type:

```
CC -D__ATT2__ -I/usr/local/include <filename>.cc -lrwlpak -lrwmatx -lrwmath -lm -lcomplex -o <filename>
```

The `CC` specifies the compiler, in this case, Sun C++; I briefly attempted to use the Gnu g++ compiler, but I could not make it the program compile correctly. The order of the "`-l`" libraries is very important. If some of these libraries are not used in the .cc file, the appropriate "`-l`" may be omitted. However, the "`-D__ATT2__`" is *absolutely mandatory*; without it, the compiler will return strange error messages, and the file may not compile correctly. The `-o <filename>` option instructs the compiler to name the executable output file `<filename>`. Without this option, the compiler automatically creates an executable file named `a.out`.

I have used some of the examples in the manual to test the math libraries. Following is the source code of each and my impressions thereof.

Example 1: Vectors (Math.h++, §14.1, pg.81)

```
/*
 * Example program using class DoubleVec
 * to do simple statistics with double precision vectors.
 */
```

[1]The National Radio Astronomy Observatory is operated by Associated Universities, Inc. under a cooperative agreement with the National Science Foundation

```
// Include the DoubleVec class header file.
#include <rw/dvec.h>

#include <rw/rstream.h>

main()
{
// Declare the double precision vector V
DoubleVec V;

// Read the vector from standard input:
cin >> V;

/*
 * Output the length of the vector.
 * This illustrates the use of a DoubleVec
 * member function:
 */

cout << "The length of the vector is:\t"
  << V.length() << "\n";

/*
 * Calculate the mean of the vector.
 * This illustrates the use
 * of the global function mean():
 */
double avg = mean(V);

cout << "The mean of the vector is:\t" << avg << "\n";

/*
 * Find the index of the maximum element of
 * the vector and output the value:
 */
int maxV = maxIndex(V);
cout << "The maximum value of the vector is:\t"
  << V(maxV) << "\n";

// Output the variance of the vector:

cout << "The variance of the vector is:\t"
 << variance(V) << "\n";

/*
 * Here is how to demean and normalize a
 * vector and print out the result:
```

```
 */
V -= mean(V); // Remove the mean
maxV = maxIndex(V); // Max element
if ( V(maxV) != 0) {
V /= fabs(V(maxV));
cout << "The normalized demeaned vector is:\n" << V;
}
return 0;
}
```

- RW overloads the characters "[" and "]". If one is inputting a vector directly from the keyboard, or redirecting the input from a file, the vector must be contained within these characters.
- "<<" has been overloaded to display a vector using `cout`. So, to display a vector, put the line

  `cout <<` *VectorName*;

  in the C++ program.
- ">>" has been overloaded to input a vector from the standard input (stdin) using `cin`. To read in a vector from the stdin, put:

  `cin >>` *someVector*

  in the C++ program.
- From UNIX, input can also be redirected to the program. To send the vector in the file `vec` to the executable program, type:

  ```
  a.out << vec
  ```

  at the UNIX prompt.
- 'Whitespace,' when inputting a vector, has absolutely no effect on the final result. Whitespace is any blank character; i.e. space, carriage return, etc.

Example 2: Matrices (Math.h++, §14.2, pg. 83)

```
/*
 * Example program showing the use of matrix
 * classes to multiply a double precision matrix by a vector.
 */

// Include the header files for double precision matrices
// and vectors.
#include <rw/dvec.h>
#include <rw/dgenmat.h>
#include <rw/rstream.h>
```

```
main()
{
// Define the vector and the matrix:
DoubleVec vector;
DoubleGenMat matrix;

// Read in the vector, then the matrix:
cin >> vector >> matrix;

// Matrix multiply (the inner product):
DoubleVec answer = product(matrix, vector);

// print out the results:
cout << "answer = \n" << answer << "\n";
return 0;
}
```

- Matrix syntax is almost identical to vector syntax. However, the size of the matrix must first be defined. This is in the form of **mxn** where m is the number of rows in the matrix, and n is the number of columns.
- Like a vector, whitespace also has no effect when entering a matrix, so:

  ```
  2 x 2
  [ 0 1
  2 3
  ]
  ```

  is the same as:

  ```
  2x2 [ 0 1 2 3 ]
  ```

  However, whitespace *is* used to separate one element from the other, so:

  ```
  2x2[0123]
  ```

  would result in a syntax error, since the program sees a 1x1 matrix with a single entry of 123.

Example 3: FFT's (Math.h++, §14.3, pg. 84)

```
/*
 * Using the Complex FFT server class.
 */

// Include the header file for complex FFT server class:
#include <rw/cfft.h>
// Header file for complex vectors:
#include <rw/cvec.h>
```

```
#include <math.h>
#include <rw/rstream.h>

main()
{
// Set series length:
unsigned npts = 12;

// Allocate a server:
DComplexFFTServer server;

/*
 * Create two series, one a cosine series, the
 * other a sine series.  This is done by first
 * using a constructor for a DoubleVec with
 * increasing elements, taking the cosine (or
 * sine) of this vector, then constructing a
 * complex vector from that.
 */

// one cycle:
DComplexVec a = cos( DoubleVec(npts,0,2.0*M_PI/npts) );

// two cycles:
DComplexVec a2 = sin( DoubleVec(npts,0,4.0*M_PI/npts) );

/*
 * Calculate the superposition of the two. Note that we
 * are adding two complex vectors here with one
 * simple statement:
 */

DComplexVec asum = a + a2;

// Output the vectors:
cout << "a:\n"    << a    << "\n";
cout << "a2:\n"   << a2   << "\n";
cout << "asum:\n" << asum << "\n";

/*
 * Print out the transforms, normalized by the
 * number of points.  The explicit DComplex constructors
 * are necessary for Zortech, which does not have
 * real to complex type conversion.
 */

cout << "\nTransform of a:\n";
cout << server.fourier(a)/DComplex(npts,0);
```

```
cout << "\nTransform of a2:\n";
cout << server.fourier(a2)/DComplex(npts,0);

DComplexVec asumFFT = server.fourier(asum);

cout << "\nTransform of asum:\n";
cout << asumFFT/DComplex(npts,0);

/*
 * Check Parseval's Theorem: First, calculate the variance of
 * the series asum, using the global function variance():
 */

cout << "\nOriginal Variance: "
  << variance(asum) << "\n";

/*
 * Next calculate the spectral variance of the fourier
 * transformed series, using the global function
 * spectralVariance():
 */

double var = spectralVariance(asumFFT/DComplex(npts,0));

cout << "Spectral Variance:  " << var << "\n\n";

/*
 * Print out the normalized back transform of
 * asumFFT; This should equal the original asum:
 */

cout << "\nBack Transform of asum:\n"
 <<server.ifourier(asumFFT)/DComplex(npts,0);
return 0;
}
```

- The member function `.fourier()` (and `.ifourier()`) must be called from an FFT server made from the data vector (i.e. DComplexFFTServer), not the vector of data itself.
- Be forewarned that it is an expensive calculation to reconfigure an FFT server for a series of different lengths. The manuals recommend constructing several severs (each for a different length) rather than reconfiguring an existing one.

Example 4: Implicit Type Conversion (Math.h++, §14.4, pg. 86)

```
/*
 * Example program  illustrating implicit type conversion.
```

```
 * C++ provides for implicit conversion between data types,
 * performed by the compiler without programmer intervention.
 * If the types of the arguments to a function do not match
 * the function prototype, a match is sought using
 * class-defined conversions.
 */

// Include the DoubleVec and IntVec class header files.

#include <rw/dvec.h>
#include <rw/ivec.h>
#include <rw/rstream.h>

// Initial data for the vectors
double adata[] = {1., 3., -2., -6., 7.};
int    idata[] = {2, 6, -4, 2, 1};

main()
{
// Construct the vectors V and iV from the
// arrays defined above:

DoubleVec V(adata, 5);
IntVec iV(idata, 5);

/*
 * Output the dot product of the two vectors.
 * Note that the function dot() has no prototype
 *  dot(DoubleVec&, IntVec&).
 * However, the prototype
 *  dot(DoubleVec&, DoubleVec&)
 * does exist.  The conversion operator DoubleVec(), defined
 * for the class IntVec, provides for the implicit type
 * conversion:  IntVec to DoubleVec.
 */
cout << dot (V, iV);
return 0;
}
```

- If the compiler sees one kind of vector where it thinks another kind belongs, it will attempt to convert the vector to the correct kind. This process is called *implicit type conversion*. This prevents one from being forced to carry out explicit type conversions in order to, for example, take the dot product of an IntVec and a DoubleVec. This is because the compiler sees `dot(`*someIntVec, someDoubleVec*) but wants `dot(`*someDoubleVec, anotherDoubleVec*), and so calls `DoubleVec(`*someIntVec*) to get the correct matrix. If an appropriate constructor does not exist, however, an error will result at the time of compilation.

- This example will also work if both `V` and `iV` are IntVec's.

- This phenomenon can also be seen if one calls:

  `inverse(`*someDoubleGenMat*`)`.

  The compiler will automatically convert "someDoubleGenMat" to a DoubleGenFact and carry out the inversion procedure.

Example 5: Persistence (saveOn / restoreFrom) (Math.h++, §14.5, pg. 87)

```
/*
 * Example program showing the use of the member functions
 * saveOn() and restoreFrom() for data storage and retrieval.
 */

#include <rw/ivec.h>
#include <rw/pstream.h>
#include <fstream.h>

main()
{
/*
 * Construct an integer vector a, with 24 elements:
 * The first element has value 0; each succeeding element is
 * incremented by 1
 */

IntVec a(24, 0, 1);

// Store the vector to file "vec.dat" using class
// RWpostream which saves in a portable ASCII format:
{
ofstream fstr("vec.dat", ios::out);
RWpostream postr(fstr);
a.saveOn(postr);
}

// A vector that has been saved using function saveOn()
// may be restored using restoreFrom():

ifstream fstr("vec.dat", ios::in);
// Construct a RWpistream from fstr
RWpistream pistr(fstr);
IntVec b;
b.restoreFrom(pistr); // Restore from file "vec.dat"

cout << a << NL; // Print the original 'a'
cout << b << NL; // Print the restored 'b'
return 0;
}
```

- There are two types of RW streams: binary (bstreams) and portable ASCII (pstreams)
- In pstream format, the vector is stored in ASCII format, one number per line. The first number has to do with some internal formatting, the second number is the length of the vector, and the vector itself follows.
- In bstream format, the vector is stored in an unreadable format with lots of control characters.
- It is important not to get confused between the redirection of input (or output) and the use of RW persistence classes. These persistence classes are unique to RW, and typing:

  `a.out <` *somePstreamFile*

  would result in a syntax error. IO redirection is a UNIX function, and it is only incidental (although highly useful) that it works with C++ programs. One can think of redirecting input as what one would type at a prompt, while persistence classes can only be used from inside a C++ program.

Example 6: Class derivation (Math.h++, §14.6, pg. 88)

```
/*
 * This example shows how a CosVec class may be derived from
 * the class DoubleVec.  Class CosVec objects are double
 * precision vectors that have been initialized with cosines
 * with a specified number of integral cycles.
 */

// Include the DoubleVec class header file.
#include <rw/dvec.h>
#include <rw/rstream.h>

/*
 * Start derived class declarations:
 * DoubleVec is declared as a public base class for CosVec:
 */

class CosVec : public DoubleVec {

public:

  // unadorned constructor:
  CosVec();

/* This is the useful constructor.  We first use the
 * DoubleVec constructor
 *    DoubleVec(n, val, by);
 * to create a vector of phases.  Taking the cosine of this
 * vector will yield the desired initial values for the base
```

```
 * class of CosVec:
 */

  CosVec (unsigned N, int cycles = 1) :
    DoubleVec( cos(DoubleVec(N,0,2*M_PI*cycles/N)) )
    { }

  // Copy constructor:
  CosVec (const CosVec& v) :
    DoubleVec(v)
    { }
};

main()
{
/*
 * Initialize a CosVec object with 12 elements and one
 * cycle:
 */
CosVec c(12,1);

/*
 * Output the vector: note that class CosVec inherits
 * operator<< from the base class DoubleVec:
 */

cout << c;
return 0;
}
```

- The manual is incorrect in this example. It should read

  ```
  CosVec (unsigned N, int cycles = 1) :
  DoubleVec (cos (Double ...
  ```

  and

  ```
  CosVec (const CosVec& v) :
  DoubleVec (v) ...
  ```

- This example just shows how the RW classes can be used as base classes for even more specialized classes. The RW classes can be treated just like any other C++ class.

Example 7: Linear Algebra (Math.h++, §15.7, pg. 90)

```
/*
 * This example shows how to use the class DoubleGenFact to do
 * linear algebra with double precision matrices.
```

```
 */

// Include the DoubleGenFact class header file.
#include <rw/dgenfct.h>
// Include the double precision matrix header file:
#include <rw/dgenmat.h>
#include <rw/rstream.h>

// Initial data for the vectors
const double adata[] = {-3.0, 2.0, 1.0, 8.0, -7.0,
9.0, 5.0, 4.0, -6.0};
const double arhs[]   = {6.0, 9.0, 1.0};

main()
{
// Construct a test matrix and print it out:
DoubleGenMat testmat(adata,3,3);
cout << "test matrix:\n" << testmat << "\n";

/*
 * Calculate and print the inverse.
 * Note that a type conversion occurs:
 * testmat is converted to type DoubleGenFact
 * before the inverse is computed:
 */
cout << "inverse:\n" << inverse(testmat);

// Now construct a DoubleGenFact from the matrix:
DoubleGenFact LUtest(testmat);

// Once constructed, LUtest may be reused as required:

// Find the determinant:
cout << "\ndeterminant\n" << determinant(LUtest) << NL;

// Solve the linear system testmat*x = rhs:
DoubleVec rhs(arhs, 3);
DoubleVec x = solve(LUtest, rhs);

cout << "solution:\n" << x << "\n";
return 0;
}
```

- This program will call the LU constructor as needed. (See Example 4.) Making the changes:

  ```
  cout << "determinant:" << NL << determinant(testmat) << NL;
  ```

  and

```
DoubleVec x2=solve(testmat, rhs);
```

would have no effect upon the output, since the compiler calls the constructor `DoubleGenFact(DoubleGenMat&)` as necessary.

- Conceivably, one would only rarely need to explicitly use the DoubleGenFact class.
- It is rather inconvenient that the RW Linear Algebra classes solve equations from vectors and matrices form, and not from explicit equations. It is up to the user/programmer to make use of the resultant matrices and/or vectors.

Example 8: Multiple inheritance (Math.h++, §14.8, pg. 91)

```
/*
 * Example to illustrate the use of multiple inheritance to do
 * run time binding.  Although the Rogue Wave Math.h++ Class
 * Library does not use any virtual functions, it is easy
 * enough to implement them in a derived class by using
 * Multiple Inheritance.
 */

#include <rw/cvec.h>
#include <rw/dvec.h>
#include <rw/rstream.h>

/*
 * This is the abstract base class where the virtual functions
 * that we plan to use are declared (but not defined).
 */

class Vector {
public:
  virtual void print() = 0;
};

/*
 * Here are two classes that inherit not only the abstract
 * base class above, but also the properties of a Math.h++
 * class.  The first class will inherit DComplexVec, the
 * second DoubleVec.
 */

class CVector : public DComplexVec, public Vector {
public:
  CVector(unsigned n, DComplex start, DComplex inc) :
      DComplexVec(n, start, inc) {}
  virtual void print();
```

```
};

class DVector : public DoubleVec, public Vector {
public:
  DVector(unsigned n, double start, double inc) :
      DoubleVec(n, start, inc) {}
  virtual void print();
};

/*
 *  A little test program to illustrate.
 */

main()
{
  int type;

  cout << "Demonstration of runtime binding.\n";
  cout << "Type 1 for a vector of complexes;\n";
  cout << "Type 2 for a vector of doubles: ";
  cin >> type;

  Vector* avec;

  switch (type) {

  case 1 :
   cout << "Constructing a CVector\n";
   avec = new CVector(10, DComplex(0,0), DComplex(1,0) );
break;
  default :
   cout << "Constructing a DVector\n";
   avec = new DVector(10, 0, 1);
  }

  /*
   * Note that "avec" points to the *base* class "Vector".
   * The actual type of what it points to is unknown at
   * compile time --- it depends on what the user typed above.
   * Runtime binding happens for the virtual function print():
   */

  avec->print();
  return 0;
}

void
CVector::print()
```

```
{
cout <<"Vector of complex:\n";
// Inherit the operator<<() function of DComplexVec:
cout << *this << "\n";
}

void
DVector::print()
{
cout <<"Vector of doubles:\n";
// Inherit the operator<<() function of DoubleVec:
cout << *this << "\n";
}
```

- The CC compiler requires an explicit:

  ```
  class (Vector:  public DComplexVec, public Vector ...
  ```

  and not

  ```
  class (Vector:  public DComplexVec, Vector ...
  ```

  as is contained in the manual.

- This example just shows that one can create multiple inheritance and run-time binding even for the RW libraries, once again showing that RW classes can be treated like any other C++ class.

LU-Decomposition: (Linpack.h++, pg. 50)
(Linpack.h++, §4.2, pg. 19)
(Linpack.h++, §5.3, pg. 25)

```
\* LU Decomposition: lsinc.cc *\
  \*  Used to test LU Decomposition classes *\

#include <rw/rstream.h>
#include <rw/dlsinc.h>

main(){
  DoubleGenMat A;
  DoubleVec b;
  cin >> A >> b;
  DoubleLeastSqInc decomp(A.cols());
  for (int i =0; i <A.rows(); i++)
    decomp.addEquation(A.row(i), b[i]);
  if (decomp.fail())
    cout << "Can't find a solution" << NL;
  else {
```

```
    DoubleVec x = decomp.solve();
    cout << "the solution is: " << NL << x << NL;
  }
  return 0;
}


\* LU-Decomposition - inc.cc *\
  \*  Used to test the LU incremental decomposition classes. *\

#include <rw/rstream.h>
#include <rw/dlsinc.h>

main(){
  unsigned numUnknowns;
  cin >> numUnknowns;
  DoubleLeastSqInc S(numUnknowns);
  DoubleVec a;
  double b;
  while (cin >> a >> b){
    S.addEquation(a,b);
    if (S.good()){
      cout << "Solution:" << NL;
      cout << S.solve() << NL;
      cout << "Error: " << S.residualNorm() << NL;
    }
  }
}


\* LU Decomposition: several.cc *\
   \*  Used to test Lu incremental decomposion classes *\

#include <rw/rstream.h>
#include <rw/dgenfct.h>

main(){
  DoubleGenMat A;
  cin >> A;
  DoubleGenFact fact(A);
  if (fact.good()){
    DoubleVec b;
    while (cin >> b){
      DoubleVec x = solve(fact,b);
      cout << x << NL;
    }
  } else {
    cout << "Not 'good'\n";
```

```
  }
}
```

- The files `lsinc.cc` and `inc.cc` were used to test the RW LU decomposition classes and LU incremental decomposition classes. There were no surprises. The program `several.cc` was used to compare these to another method of equation solving.
- The LU decomposition classes can be used as a precursor to fast matrix inversion or finding the matrix determinant, or in order to solve a group of algebraic equations. The incremental class is used to solve an "over-determined" set of equations (i.e. one to which there is no unique solution).

Least Square Fit: (Linpack.h++, §5.1, pg. 24)

```
\* Least Square Fit: lsch.cc *\
  \*  Used to test linear least square fit class *\

#include <rw/rstream.h>
#include <rw/flsch.h>

main(){
  FloatGenMat A;
  FloatVec b;
  cin >> A >> b;
  FloatLeastSqCh ls(A);
  FloatVec x = solve (ls,b);
  cout << x << NL;
}
```

- The file `lsch.cc` was used to test the LeastSqFit class. Once again, there were no surprises.
- Note that this class performs only a *linear* fit. There is no RW matrix class for a *non-linear* (polynomial) least-squares fit. (See below.)
- There are two methods of performing a Least Square Fit: Cholesky and QR. The program `lsch.cc` uses Cholesky method, but changing two lines to read:

  ```
  #include <rw/flsqr.h>
  FloatLeastSqQR ls(A);
  ```

  would implement the QR method.
- The Cholesky method provides superior speed, while the QR method provides better numeric properties. The most appropriate method will depend upon the problem; not to mention, even experts will sometimes disagree as to which method should be used!

QR decomposition: (Linpack.h++, pg. 59)

```
\* QR decomposition:   *\
  \*  Used to test QR decomposition class *\


#include <rw/rstream.h>
#include <rw/dqr.h>
#include <rw/dutrimat.h>

main(){
  DoubleGenMat A;
  cin >> A;
  DoubleQRDecomp QR;
  QR.doPivoting(FALSE);
  QR.factor(A);
  DoubleGenMat Rsq = QR;
  if (QR.rows() > QR. cols())
    Rsq.resize(QR.cols(), QR.cols());
  else
    Rsq.resize(QR.rows(), QR.rows());
  DoubleUpperTriMat R = toUpperTriMat(Rsq);
  DoubleGenMat Q(QR.rows(),QR.rows());
  DoubleVec e(QR.rows(),0);
  for (int i = 0; i < QR.rows();i++){
    e[i]=1;
    Q.col(i) = QR.computeQy(e);
    e[i] =0;
  }
  cout << "Q is " << NL << Q << NL;
  cout << "R is " << NL << R << NL;
}
```

- The file `qrdec.cc` was used to test the QR decomposition class.
- Note that calling `QR.rows` or `QR.cols` (instead of `QR.rows()` and `QR.cols()`) will result in "...(anachronism)" errors from the compiler. If you are seeing many of these errors, you may have left out a "()."

Cholesky decomposition: (There is no code in the manuals for this example.)

```
/* Cholesky Decompositions: chdec.cc */
   /*  This file tests the RW Cholesky decomposition class.  */

#include <rw/dch.h>
#include <rw/dgenmat.h>
#include <rw/rstream.h>
#include <rw/dutrimat.h>

main(){
```

```
  DoubleGenMat A;
  cin >> A;
  cout << "Original:\n" << A << NL;
  DoubleChDecomp d;
  d.doPivoting(FALSE);
  d.factor(A);
  DoubleGenMat D(d);
  for (int i = -1; i > -(D.rows()); i--)
    D.diagonal(i) =0;
  if (d.isPD()==TRUE)
    cout << ".isPD sucessful\n";
  if (d.fail())
    cout << "Decomposition returned \"fail\"!\n";
  DoubleGenMat Dt = transpose(D);
  cout << "Result:\n" << product(Dt,D);
}
```

- The file `chdec.cc` was used to test the Cholesky decomposition class.
- Since the manuals contain no explicit example for using the Cholesky decomposition classes, I have prepared a simple one. Also, since I have no way of independently verifying the results of this decomposition, the program outputs transpose(decomp) * decomp, which should return the original matrix.
- This program has been tested for matrix sizes of 2x2, 3x3, 4x4, and 5x5. The only problem occurred when testing 3x3 matrices. The final result when using a 3x3 (transpose(decomp) * decomp) did not return the original matrix. Generally speaking, only the bottom right corner was not correctly "reconstructed". Also note, `d.fail()` returned a fail for virtually every 3x3 matrix tried. However, a very "simple" matrix like

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

  does correctly decompose and reconstruct.
- The Cholesky decomposition class assumes that the matrix being decomposed is symmetric, and only examines the upper right triangle. The result of the `chdec.cc` program is a symmetric matrix, regardless of the original matrix.
- This class also uses "pivoting" when decomposing. This example has disabled pivoting by `d.doPivoting(FALSE)`
- Since a Cholesky decomposition is an upper triangular matrix (having zeros in the lower triangle) a small loop has been added which removes any non-zero entries in the lower triangle.

Singular Value Decomposition: (Linpack.h++, pg. 64)

```
/*  Singular Value Decomposition:  svd.cc */
  /*  Used to test singular value decomposion class  */

#include <rw/rstream.h>
#include <rw/dsv.h>

main(){
  DoubleGenMat A;
  cin >> A;
  DoubleSVDecomp SVD;
  SVD.computeLeftVecs(FALSE);
  SVD.computeRightVecs(FALSE);
  SVD.factor(A);
  if (SVD.fail())
    cout << "Couldn't calculate singular values";
  else {
    cout << "Singular values are: " <<NL;
    cout << SVD.singularValues() << NL;
  }

  return 0;
}
```

- The file `svd.cc` was used to test the singular value decomposition class.
- The singular values are always returned largest to smallest. It is also possible to select which singular vectors are to be computed.

Matrix Arithmetic: (Matrix.h++, §4.2, pg. 16)

```
\*  Matrix Arithmatic:  matarith.cc *\
  \*  used to test simple matrix arithmatic  *\

#include <rw/ftrdgmat.h>
#include <rw/rstream.h>

main(){
  FloatTriDiagMat T(6,6);
  T.diagonal(-1)=1;
  T.diagonal(0)=2;
  T.diagonal(1)=3;
  FloatVec x(6,1,1);
  FloatVec Tx = product(T,x);
  cout << "T = " << T << NL;
  cout << "x = " << x << NL;
  cout << "Tx = " << Tx << NL;
}
```

- As a simple test of matrix Arithmetic, the file `matarith.cc` was used.
- Line 4 should read:

  `FloatTriDiagMat T(6,6)`
- and not

  `FloatTriDiagMat T(6,6,0)`
- as is in the manual.

Non-Linear Least Squares Fit: (There is no code in the manual for this example.)

```
/* RWFIT.CC -- performs leas - squares - fit on data from file or */
 /* entered by keyboard.  Also does some error analysis on the */
 /* calculated values.  This version of the program uses the Rogue */
 /* Wave matrix libraries. */

#include <fstream.h>
#include <rw/dgenfct.h>
#include <rw/dgenmat.h>
#include <rw/rstream.h>
#include <rw/pstream.h>

double power(double base, double exp){
  if (exp == 0)
    return 1;
  else if (base == 0)
    return 0;
  else
    return pow(base,exp);
}


class LeastSquareFit{ // class to do a polynomial least-squares-fit on
 public:              //   supplied data matrices
  LeastSquareFit();
  void inputxy();     // for keyboard input of data
  void calcFit();     // calculates actual LU fit from existing arrays
  void errorAnal();   // performs error analysis from calculated values
  void fileLoad();    // loads data into matrices from given file
 private:
  int n;              // number of data points
  DoubleGenMat *y;    // pointer to array of y data
  DoubleGenMat *a;    // pointer to x matrix, size depends on order of fit
  DoubleGenMat *calcValues; // pointer to calculated values
  int order;          // order of polynomial to fit
  DoubleGenMat *X;    // matrix of polynomial coefficients
  double errorSqd();  // finds sum of error squared
```

```
  double rms();       // finds rms of errors
  double std();       // finds standard deviation of errors
  double corrCoeff(); // finds correlation coeff. ("linearness") of data
};

 LeastSquareFit::LeastSquareFit(){
   cout << "What order should the fit be?";
   cin >> order;
   X = new DoubleGenMat(order+1, 1);
   char answer;
   cout << "Load from file? (y/n)";
   cin >> answer;
   if ((answer == 'Y') || (answer == 'y')){
     fileLoad();
   } else
     inputxy();
 }

void LeastSquareFit::inputxy(){
  cout << "How many sets of data are there? ";
  cin >> n;
  a = new DoubleGenMat(n, order+1);
  y = new DoubleGenMat(n,1);
  double xtemp, ytemp;
  for (int i = 0;i < n; i++){
    cout << "#" << i+1;
    cout << "\tX:";
    cin >> xtemp;
    cout << "\tY:";
    cin >> ytemp;
    (*y)(i,0) = ytemp;
    for (int j = 0; j <= order; j++)
      (*a)(i,j) = power(xtemp,j);
  }
}

void LeastSquareFit::calcFit(){
  DoubleGenMat At = transpose(*a);
  DoubleGenMat G(At.rows(), a->cols());
  G = product(At,*a);
  if(DoubleGenFact(G).fail()){
    cout << "Matrix constructed was singular,\n";
    cout << "perhaps you need more data points.\n";
    exit (1);
  }
  DoubleGenMat ginv = inverse(G);
  *X = product(ginv, product(At,*y));
  printf ("\n\nThe coeff's. are, in increasing order:\n");
```

```
  for (int i = 0; i <= order; i++)
    printf("x^%i = %.2f\n", i, (*X)(i,0));
  printf("\n\n");
}

void LeastSquareFit::errorAnal(){
  DoubleGenMat yCalc(n,1);
  for (int i = 0; i < n; i++){
    double temp = 0;
    for (int j = order; j >= 0; j--)
      temp += (*X)(j,0) * power ((*a)(i,1),j);
    yCalc(i,0) = temp;
  }
  calcValues = &yCalc;
  printf("Sum of Errors Squared = %.3f\n", errorSqd());
  printf("Rms of Error          = %.3f\n", rms());
  printf("Standard Deviation    = %.3f\n", std());
  printf("Correlation Coeff.    = %f\n", corrCoeff());
}

double LeastSquareFit::errorSqd(){
  double sum = 0;
  for (int i = 0; i < n; i++)
    sum += power ((*calcValues)(i,0) - (*y)(i,0), 2);
  return sum;
}

double LeastSquareFit::rms(){
  return sqrt( errorSqd() / n );
}

double LeastSquareFit::std(){
  //  find standard dev. of errors;
  double std = 0;
  //  find mean of errors;
  double mean = 0;
  for (int i = 0;i < n; i++)
    mean += (*calcValues)(i,0) - (*y)(i,0);
  mean /= n;
  // now calculate variance;
  for (i = 0; i < n; i++)
    std += power ( ((*calcValues)(i,0) - (*y)(i,0)) - mean, 2);
  return sqrt ( std / n);
}

double LeastSquareFit::corrCoeff(){
  double ymean = 0;
  double xmean = 0;
```

```
  double calcMean = 0;
  for (int i = 0; i < n; i++){
    ymean += (*y)(i,0);
    xmean += (*a)(i,1);
    calcMean += (*calcValues)(i,0);
  }
  xmean /= n;
  ymean /=n;
  calcMean /=n;
  double xy = 0, xx = 0, yy = 0;
  for (i = 0; i < n; i++){
    xy += ((*a)(i,1) - xmean) * ((*y)(i,0)-ymean);
    xx += power ((*a)(i,1) - xmean,2);
    yy += power ((*y)(i,0) - ymean,2);
  }
  return (xy/ sqrt (xx*yy));
}

void LeastSquareFit::fileLoad(){
  char *filename;
  filename = new char[100];
  cout << "What is the filename? ";
  cin >> filename;
  FILE *input;
  input = fopen(filename, "r");
  if (!input)
    while (!input){
      cout << "File \"" << filename << "\" does not exist\n";
      cout << "Load from what file? ";
      cin >> filename;
      input = fopen(filename, "r");
    }
  delete filename;
  float tempx, tempy;
  n = 0;
  while (1){
    (void) fscanf(input, "%f%f", &tempx, &tempy);
    if feof(input)
      break;
    n++;
  }
  double xtemp, ytemp;
  a = new DoubleGenMat (n, order+1);
  y = new DoubleGenMat (n, 1);
  rewind (input);
  for (int i = 0; i < n; i++){
    (void)fscanf(input, "%lf%lf", &xtemp, &ytemp);
    (*y)(i,0) = ytemp;
```

```
    for (int j = 0; j <= order; j++)
      (*a)(i,j) = power(xtemp,j);
  }
  fclose (input);
}

int main(){
  LeastSquareFit x;
  x.calcFit();
  x.errorAnal();
}
```

- Since the RW libraries only contain a linear LS fit, I have written a non-linear least squares fit class and adapted it to use the RW classes. The code for this fit is located in `/zia/u/eolson/cpp/rw/rwfit.cc`.
- This code also does some minor error analysis.
- I have not used the RW persistence classes (bstream, pstream, etc.) in the member function that retrieves matrices from disk. This is only for maximum portability between the RW version of this program and previous versions. When writing other programs, it may be considerably easier to use the RW classes, rather than create another method of disk access.
- Note that the calculation of the fit is a matrix procedure, with the inversion doing most of the work.

Summary:

- Each vector and matrix must be declared with a certain precision. Therefore a DoubleGenMat and a FloatGenMat and an IntGenMat are all the same basic matrix shape (GenMat), but the precisions differ.
- The RW libraries define a new data type, RWBoolean. This type can be either `TRUE` or `FALSE`.
  Certain member functions (for example, FloatCHDecomp.doPivoting(RWBoolean)) require a RWBoolean. This should be either `TRUE` or `FALSE` (caps required).
- The Matrix.h++ library has many, many matrix shapes. (Probably more than will ever be used.) It is quite easy to become confused with similar types (for example, DComplexHermMat and DComplexHermBandMat).
- It is also quite easy to become confused when using other RW classes. For example, DoubleGenMat and DoubleGenFact are quite similar, yet one is a matrix class, the other a decomposition.
- The libraries will perform bounds checking on vectors (to make sure that a vector of length 10 is not accessed at length 11). Referencing the vector by '[' and ']' (e.g. Vector[2]) will always perform bounds checking. Referencing the vector by '(' and ')' (e.g. Vector(2))

will perform bounds checking only if the directive **BOUNDS_CHECK** has been defined. Note that **BOUNDS_CHECK** must be defined *before* the **#include** files are defined.

- The vector classes have a function called **slice**. This returns a vector that is a subset of the vector for which it is called, referenced at fixed intervals. For example, if vector v is `[ 0 1 2 3 4 5 ]`, `v.slice(0,3,2)` is equal to `[0 2 4]`. Slicing is much faster than referencing each individual element. However, note that a slice is a *reference* to certain elements in the vector, change the original vector, and the slice is changed as well.

- When one calls a vector or matrix constructor by `copyOfObject(originalObject)`, the data in `originalObject` is *referenced* by `copyOfObject`, not copied. The member function `.deepenShallowCopy` generates a discrete set of data for the object after which changing the original will not affect the copy.