Documentation for newmat10D - A matrix library in C++

(Auszug)

Gesamtdokumentation verfügbar unter http://www.robertnz.net/nm10.htm

---

3. Reference manual

3.1 Constructors

To construct an *m* x *n* matrix, `A`, (*m* and *n* are integers) use

```
Matrix A(m,n);
```

The UpperTriangularMatrix, LowerTriangularMatrix, SymmetricMatrix and DiagonalMatrix types are square. To construct an *n* x *n* matrix use, for example

```
UpperTriangularMatrix UT(n);
LowerTriangularMatrix LT(n);
SymmetricMatrix S(n);
DiagonalMatrix D(n);
```

Band matrices need to include bandwidth information in their constructors.

```
BandMatrix BM(n, lower, upper);
UpperBandMatrix UB(n, upper);
LowerBandMatrix LB(n, lower);
SymmetricBandMatrix SB(n, lower);
```

The integers *upper* and *lower* are the number of non-zero diagonals above and below the diagonal (*excluding* the diagonal) respectively.

The RowVector and ColumnVector types take just one argument in their constructors:

```
RowVector RV(n);
ColumnVector CV(n);
```

These constructors do *not* initialise the elements of the matrices. To set all the elements to zero use, for example,

```
Matrix A(m, n); A = 0.0;
```

The IdentityMatrix takes one argument in its constructor specifying its dimension.

```
IdentityMatrix I(n);
```

The value of the diagonal elements **is** set to 1 by default, but you can change this value as with other matrix types.

You can also construct vectors and matrices without specifying the dimension. For example

```
Matrix A;
```

In this case the dimension must be set by an assignment statement or a re-size statement.

You can also use a constructor to set a matrix equal to another matrix or matrix expression.

```
Matrix A = UT;
Matrix A = UT * LT;
```

Only conversions that don't lose information are supported - eg you cannot convert an upper triangular matrix into a diagonal matrix using =.

3.2 Accessing elements

Elements are accessed by expressions of the form `A(i,j)` where *i* and *j* run from 1 to the appropriate dimension. Access elements of vectors with just one argument. Diagonal matrices can accept one or two subscripts.

This is different from the earliest version of the package in which the subscripts ran from 0 to one less than the appropriate dimension. Use `A.element(i,j)` if you want this earlier convention.

`A(i,j)` and `A.element(i,j)` can appear on either side of an = sign.

If you activate the `#define SETUP_C_SUBSCRIPTS` in `include.h` you can also access elements using the traditional C style notation. That is `A[i][j]` for matrices (except diagonal) and `V[i]` for vectors and diagonal matrices. The subscripts start at zero (i.e. like element) and there is *no* range checking. Because of the possibility of confusing `V(i)` and `V[i]`, I suggest you do *not* activate this option unless you really want to use it.

Symmetric matrices are stored as lower triangular matrices. It is important to remember this if you are using the `A[i][j]` method of accessing elements. Make sure the first subscript is

greater than or equal to the second subscript. However, if you are using the `A(i,j)` method the program will swap `i` and `j` if necessary; so it doesn't matter if you think of the storage as being in the upper triangle (but it *does* matter in some other situations such as when entering data).

The IdentityMatrix type does not support element access.

3.3 Assignment and copying

The operator = is used for copying matrices, converting matrices, or evaluating expressions. For example

```
A = B;  A = L;  A = L * U;
```

Only conversions that don't lose information are supported. The dimensions of the matrix on the left hand side are adjusted to those of the matrix or expression on the right hand side. Elements on the right hand side which are not present on the left hand side are set to zero.

The operator << can be used in place of = where it is permissible for information to be lost.

For example

```
SymmetricMatrix S; Matrix A;
......
S << A.t() * A;
```

is acceptable whereas

```
S = A.t() * A;                          // error
```

will cause a runtime error since the package does not (yet?) recognise `A.t()*A` as symmetric.

Note that you can *not* use << with constructors. For example

```
SymmetricMatrix S << A.t() * A;         // error
```

does *not* work.

Also note that << cannot be used to load values from a full matrix into a band matrix, since it will be unable to determine the bandwidth of the band matrix.

A third copy routine is used in a similar role to =. Use

```
A.Inject(D);
```

to copy the elements of `D` to the corresponding elements of `A` but leave the elements of `A` unchanged if there is no corresponding element of `D` (the = operator would set them to 0). This is useful, for example, for setting the diagonal elements of a matrix without disturbing the rest of the matrix. Unlike = and <<, Inject does not reset the dimensions of `A`, which must match those of `D`. Inject does *not* test for no loss of information.

You cannot replace `D` by a matrix expression. The effect of `Inject(D)` depends on the type of `D`. If `D` is an expression it might not be obvious to the user what type it would have. So I thought it best to disallow expressions.

Inject can be used for loading values from a regular matrix into a band matrix. (Don't forget to zero any elements of the left hand side that will not be set by the loading operation).

Both << and Inject can be used with submatrix expressions on the left hand side. See the section on submatrices.

To set the elements of a matrix to a scalar use operator =

```
Real r; int m,n;
......
Matrix A(m,n); A = r;
```

Notes:

- When you do a matrix assignment to another matrix or matrix expression with either = or << the original data array associated with the matrix being assigned to is destroyed even if there is no change in length. See the section on storage. This means, in particular, that pointers to matrix elements - e.g. `Real* a; a = &(A(1,1));` become invalid. If you want avoid this you can use `Inject` rather than =. But remember that you may need to zero the matrix first.

3.4 Entering values

You can load the elements of a matrix from an array:

```
Matrix A(3,2);
Real a[] = { 11,12,21,22,31,33 };
A << a;
```

This construction does *not* check that the numbers of elements match correctly. This version of << can be used with submatrices on the left hand side. It is not defined for band matrices.

Alternatively you can enter short lists using a sequence of numbers separated by << .

```
Matrix A(3,2);
A << 11 << 12
  << 21 << 22
  << 31 << 32;
```

This does check for the correct total number of entries, although the message for there being insufficient numbers in the list may be delayed until the end of the block or the next use of this construction. This does *not* work for band matrices or for long lists. It does work for submatrices if the submatrix is a single complete row. For example

```
Matrix A(3,2);
A.Row(1) << 11 << 12;
A.Row(2) << 21 << 22;
A.Row(3) << 31 << 32;
```

Load only values that are actually stored in the matrix. For example

```
SymmetricMatrix S(2);
S.Row(1) << 11;
S.Row(2) << 21 << 22;
```

Try to restrict this way of loading data to numbers. You can include expressions, but these must not call a function which includes the same construction.

Remember that matrices are stored by rows and that symmetric matrices are stored as *lower* triangular matrices when using these methods to enter data.

3.5 Unary operators

The package supports unary operations

```
X = -A;           // change sign of elements
X = A.t();        // transpose
X = A.i();        // inverse (of square matrix A)
X = A.Reverse();  // reverse order of elements of vector
                  // or matrix (not band matrix)
```

3.6 Binary operators

The package supports binary operations

```
X = A + B;       // matrix addition
X = A - B;       // matrix subtraction
X = A * B;       // matrix multiplication
X = A.i() * B;   // equation solve (square matrix A)
X = A | B;       // concatenate horizontally (concatenate the rows)
X = A & B;       // concatenate vertically (concatenate the columns)
X = SP(A, B);    // elementwise product of A and B (Schur product)
X = KP(A, B);    // Kronecker product of A and B
bool b = A == B; // test whether A and B are equal
bool b = A != B; // ! (A == B)
A += B;          // A = A + B;
A -= B;          // A = A - B;
A *= B;          // A = A * B;
A |= B;          // A = A | B;
A &= B;          // A = A & B;
<, >, <=, >=     // included for compatibility with STL - see notes
```

Notes:

- If you are doing repeated multiplication. For example `A*B*C`, use brackets to force the order of evaluation to minimise the number of operations. If `C` is a column vector and `A` is not a vector, then it will usually reduce the number of operations to use `A*(B*C)`.
- In the equation solve example case the inverse is not explicitly calculated. An LU decomposition of `A` is performed and this is applied to `B`. This is more efficient than calculating the inverse and then multiplying. See also multiple matrix solving.
- The package does not (yet?) recognise `B*A.i()` as an equation solve and the inverse of `A` would be calculated. It is probably better to use `(A.t().i()*B.t()).t()`.

- Horizontal or vertical concatenation returns a result of type Matrix, RowVector or ColumnVector.
- If `A` is *m* x *p*, `B` is *m* x *q*, then `A | B` is *m* x (*p*+*q*) with the *k*-th row being the elements of the *k*-th row of `A` followed by the elements of the *k*-th row of `B`.
- If `A` is *p* x *n*, `B` is *q* x *n*, then `A & B` is (*p*+*q*) x *n* with the *k*-th column being the elements of the *k*-th column of `A` followed by the elements of the *k*-th column of `B`.
- For complicated concatenations of matrices, consider instead using submatrices.
- See the section on submatrices on using a submatrix on the RHS of an expression.
- Two matrices are equal if their difference is zero. They may be of different types. For the CroutMatrix or BandLUMatrix they must be of the same type and have all their elements equal. This is not a very useful operator and is included for compatibility with some container templates.
- The inequality operators are included for compatibility with the standard template library. If actually called, they will throw an exception. So don't try to sort a *list* of matrices.
- A row vector multiplied by a column vector yields a 1x1 matrix, *not* a Real. To get a Real result use either AsScalar or DotProduct.
- The result from Kronecker product, KP(A, B), possesses an attribute such as upper triangular, lower triangular, band, symmetric, diagonal if both of the matrices A and B have the attribute. (This differs slightly from the way the January 2002 version of newmat10 worked).

Remember that the product of symmetric matrices is not necessarily symmetric so the following code will not run:

```
SymmetricMatrix A, B;
.... put values in A, B ....
SymmetricMatrix C = A * B;   // run time error
```

Use instead

```
Matrix C = A * B;
```

or, if you *know* the product will be symmetric, use

```
SymmetricMatrix C; C << A * B;
```

3.7 Matrix and scalar

The following expressions multiply the elements of a matrix `A` by a scalar `f`: `A * f` or `f * A`. Likewise one can divide the elements of a matrix `A` by a scalar `f`: `A / f`.

The expressions `A + f` and `A - f` add or subtract a rectangular matrix of the same dimension as `A` with elements equal to `f` to or from the matrix `A`.

The expression `f + A` is an alternative to `A + f`. The expression `f - A` subtracts matrix `A` from a rectangular matrix of the same dimension as `A` and with elements equal to `f`.

The expression `A += f` replaces `A` by `A + f`. Operators `-=`, `*=`, `/=` are similarly defined.

3.8 Scalar functions of a matrix - size & shape

This page describes functions returning the values associated with the size and shape of matrices. The following pages describe other scalar matrix functions.

```
int m = A.Nrows();                      // number of rows
int n = A.Ncols();                      // number of columns
MatrixType mt = A.Type();               // type of matrix
Real* s = A.Store();                    // pointer to array of elements
int l = A.Storage();                    // length of array of elements
MatrixBandWidth mbw = A.BandWidth();    // upper and lower bandwidths
```

`MatrixType mt = A.Type()` returns the type of a matrix. Use `mt.Value()` to get a string (UT, LT, Rect, Sym, Diag, Band, UB, LB, Crout, BndLU) showing the type (Vector types are returned as Rect).

`MatrixBandWidth` has member functions `Upper()` and `Lower()` for finding the upper and lower bandwidths (number of diagonals above and below the diagonal, both zero for a diagonal matrix). For non-band matrices -1 is returned for both these values.

3.9 Scalar functions of a matrix - maximum & minimum

This page describes functions for finding the maximum and minimum elements of a matrix.

```
int i, j;
Real mv = A.MaximumAbsoluteValue();     // maximum of absolute values
Real mv = A.MinimumAbsoluteValue();     // minimum of absolute values
Real mv = A.Maximum();                  // maximum value
Real mv = A.Minimum();                  // minimum value
Real mv = A.MaximumAbsoluteValue1(i);   // maximum of absolute values
Real mv = A.MinimumAbsoluteValue1(i);   // minimum of absolute values
Real mv = A.Maximum1(i);                // maximum value
Real mv = A.Minimum1(i);                // minimum value
Real mv = A.MaximumAbsoluteValue2(i,j);// maximum of absolute values
Real mv = A.MinimumAbsoluteValue2(i,j);// minimum of absolute values
Real mv = A.Maximum2(i,j);              // maximum value
Real mv = A.Minimum2(i,j);              // minimum value
```

All these functions throw an exception if `A` has no rows or no columns.

The versions `A.MaximumAbsoluteValue1(i)`, etc return the location of the extreme element in a RowVector, ColumnVector or DiagonalMatrix. The versions `A.MaximumAbsoluteValue2(i,j)`, etc return the row and column numbers of the extreme element. If the extreme value occurs more than once the location of the last one is given.

The versions MaximumAbsoluteValue(A), MinimumAbsoluteValue(A), Maximum(A), Minimum(A) can be used in place of A.MaximumAbsoluteValue(), A.MinimumAbsoluteValue(), A.Maximum(), A.Minimum().

3.10 Scalar functions of a matrix - numerical

```
Real r = A.AsScalar();                  // value of 1x1 matrix
Real ssq = A.SumSquare();               // sum of squares of elements
```

```
    Real sav = A.SumAbsoluteValue();       // sum of absolute values
    Real s = A.Sum();                      // sum of values
    Real norm = A.Norm1();                 // maximum of sum of absolute
                                              values of elements of a
column
    Real norm = A.NormInfinity();          // maximum of sum of absolute
                                              values of elements of a row
    Real norm = A.NormFrobenius();         // square root of sum of squares
                                           // of the elements
    Real t = A.Trace();                    // trace
    Real d = A.Determinant();              // determinant
    LogAndSign ld = A.LogDeterminant();    // log of determinant
    bool z = A.IsZero();                   // test all elements zero
    bool s = A.IsSingular();               // A is a CroutMatrix or
                                              BandLUMatrix
    Real s = DotProduct(A, B);             // dot product of A and B
                                           // interpreted as vectors
```

`A.LogDeterminant()` returns a value of type LogAndSign. If ld is of type LogAndSign use

```
    ld.Value()    to get the value of the determinant
    ld.Sign()     to get the sign of the determinant (values 1, 0, -1)
    ld.LogValue() to get the log of the absolute value.
```

Note that the direct use of the function `Determinant()` will often cause a floating point overflow exception.

`A.IsZero()` returns Boolean value `true` if the matrix `A` has all elements equal to 0.0.

`IsSingular` is defined only for CroutMatrix and BandLUMatrix. It returns `true` if one of the diagonal elements of the LU decomposition is exactly zero.

`DotProduct(const Matrix& A,const Matrix& B)` converts both of the arguments to rectangular matrices, checks that they have the same number of elements and then calculates the first element of `A *` first element of `B +` second element of `A *` second element of `B + ...` ignoring the row/column structure of `A` and `B`. It is primarily intended for the situation where `A` and `B` are row or column vectors.

The versions Sum(A), SumSquare(A), SumAbsoluteValue(A), Trace(A), LogDeterminant(A), Determinant(A), Norm1(A), NormInfinity(A), NormFrobenius(A) can be used in place of A.Sum(), A.SumSquare(), A.SumAbsoluteValue(), A.Trace(), A.LogDeterminant(), A.Norm1(), A.NormInfinity(), A.NormFrobenius().

3.11 Submatrices

```
    A.SubMatrix(fr,lr,fc,lc)
```

This selects a submatrix from `A`. The arguments `fr,lr,fc,lc` are the first row, last row, first column, last column of the submatrix with the numbering beginning at 1.

I allow `lr = fr-1` or `lc = fc-1` or to indicate that a matrix of zero rows or columns is to be returned.

A submatrix command may be used in any matrix expression or on the left hand side of =, << or Inject. Inject does *not* check no information loss. You can also use the construction

```
Real c; .... A.SubMatrix(fr,lr,fc,lc) = c;
```

to set a submatrix equal to a constant.

The following are variants of SubMatrix:

```
A.SymSubMatrix(f,l)             //   This assumes fr=fc and lr=lc.
A.Rows(f,l)                     //   select rows
A.Row(f)                        //   select single row
A.Columns(f,l)                  //   select columns
A.Column(f)                     //   select single column
```

In each case `f` and `l` mean the first and last row or column to be selected (starting at 1).

I allow `l = f-1` to indicate that a matrix of zero rows or columns is to be returned.

If SubMatrix or its variant occurs on the right hand side of an = or << or within an expression think of its type as follows

```
A.SubMatrix(fr,lr,fc,lc)          If A is RowVector or
                                  ColumnVector then same type
                                  otherwise type Matrix
A.SymSubMatrix(f,l)               Same type as A
A.Rows(f,l)                       Type Matrix
A.Row(f)                          Type RowVector
A.Columns(f,l)                    Type Matrix
A.Column(f)                       Type ColumnVector
```

If SubMatrix or its variant appears on the left hand side of = or << , think of its type being Matrix. Thus `L.Row(1)` where `L` is LowerTriangularMatrix expects `L.Ncols()` elements even though it will use only one of them. If you are using = the program will check for no loss of data.

A SubMatrix can appear on the left-hand side of `+=` or `-=` with a matrix expression on the right-hand side. It can also appear on the left-hand side of `+=`, `-=`, `*=` or `/=` with a Real on the right-hand side. In each case there must be no loss of information.

The `Row` version can appear on the left hand side of << for loading literal data into a row. Load only the number of elements that are actually going to be stored in memory.

Do not use the `+=` and `-=` operations with a submatrix of a SymmetricMatrix or BandSymmetricMatrix on the LHS and a Real on the RHS.

You can't pass a submatrix (or any of its variants) as a reference non-constant matrix in a function argument. For example, the following will not work:

```
void YourFunction(Matrix& A);
...
Matrix B(10,10);
YourFunction(B.SubMatrix(1,5,1,5))    // won't compile
```

If you are are using the submatrix facility to build a matrix from a small number of components, consider instead using the concatenation operators.

3.12 Change dimensions

The following operations change the dimensions of a matrix. The values of the elements are lost.

```
A.ReSize(nrows,ncols);        // for type Matrix or nricMatrix
A.ReSize(n);                  // for all other types, except Band
A.ReSize(n,lower,upper);      // for BandMatrix
A.ReSize(n,lower);            // for LowerBandMatrix
A.ReSize(n,upper);            // for UpperBandMatrix
A.ReSize(n,lower);            // for SymmetricBandMatrix
A.ReSize(B);                  // set dims to those of B
```

Use `A.CleanUp()` to set the dimensions of `A` to zero and release all the heap memory.

`A.ReSize(B)` sets the dimensions of `A` to those of a matrix `B`. This includes the band-width in the case of a band matrix. It is an error for `A` to be a band matrix and `B` not a band matrix (or diagonal matrix).

Remember that `ReSize` destroys values. If you want to `ReSize`, but keep the values in the bit that is left use something like

```
ColumnVector V(100);
...                           // load values
V = V.Rows(1,50);             // to get first 50 values.
```

If you want to extend a matrix or vector use something like

```
ColumnVector V(50);
...                           // load values
{ V.Release(); ColumnVector X=V; V.ReSize(100); V.Rows(1,50)=X; }
                              // V now length 100
```

3.13 Change type

The following functions interpret the elements of a matrix (stored row by row) to be a vector or matrix of a different type. Actual copying is usually avoided where these occur as part of a more complicated expression.

```
A.AsRow()
A.AsColumn()
A.AsDiagonal()
A.AsMatrix(nrows,ncols)
A.AsScalar()
```

The expression `A.AsScalar()` is used to convert a 1 x 1 matrix to a scalar.

3.14 Multiple matrix solve

To solve the matrix equation `Ay = b` where `A` is a square matrix of equation coefficients, `y` is a column vector of values to be solved for, and `b` is a column vector, use the code

```
int n = something
Matrix A(n,n); ColumnVector b(n);
... put values in A and b
ColumnVector y = A.i() * b;       // solves matrix equation
```

The following notes are for the case where you want to solve more than one matrix equation with different values of `b` but the same `A`. Or where you want to solve a matrix equation and also find the determinant of `A`. In these cases you probably want to avoid repeating the LU decomposition of `A` for each solve or determinant calculation.

If `A` is a square or symmetric matrix use

```
CroutMatrix X = A;                // carries out LU decomposition
Matrix AP = X.i()*P; Matrix AQ = X.i()*Q;
LogAndSign ld = X.LogDeterminant();
```

rather than

```
Matrix AP = A.i()*P; Matrix AQ = A.i()*Q;
LogAndSign ld = A.LogDeterminant();
```

since each operation will repeat the LU decomposition.

If `A` is a BandMatrix or a SymmetricBandMatrix begin with

```
BandLUMatrix X = A;               // carries out LU decomposition
```

A CroutMatrix or a BandLUMatrix can't be manipulated or copied. Use references as an alternative to copying.

Alternatively use

```
LinearEquationSolver X = A;
```

This will choose the most appropriate decomposition of `A`. That is, the band form if `A` is banded; the Crout decomposition if `A` is square or symmetric and no decomposition if `A` is triangular or diagonal.

3.15 Memory management

The package does not support delayed copy. Several strategies are required to prevent unnecessary matrix copies.

Where a matrix is called as a function argument use a constant reference. For example

```
YourFunction(const Matrix& A)
```

rather than

```
YourFunction(Matrix A)
```

Skip the rest of this section on your first reading.

Gnu g++ (< 2.6) users please read on: if you are returning matrix values from a function, then you must use the ReturnMatrix construct.

A second place where it is desirable to avoid unnecessary copies is when a function is returning a matrix. Matrices can be returned from a function with the return command as you would expect. However these may incur one and possibly two copyings of the matrix. To avoid this use the following instructions.

Make your function of type ReturnMatrix . Then precede the return statement with a Release statement (or a ReleaseAndDelete statement if the matrix was created with new). For example

```
ReturnMatrix MakeAMatrix()
{
   Matrix A;                // or any other matrix type
   ......
   A.Release(); return A;
}
```

or

```
ReturnMatrix MakeAMatrix()
{
   Matrix* m = new Matrix;
   ......
   m->ReleaseAndDelete(); return *m;
}
```

If your compiler objects to this code, replace the return statements with

```
return A.ForReturn();
```

or

```
return m->ForReturn();
```

If you are using AT&T C++ you may wish to replace `return A;` by return `(ReturnMatrix)A;` to avoid a warning message; but this will give a runtime error with Gnu. (You can't please everyone.)

Do not forget to make the function of type ReturnMatrix; otherwise you may get incomprehensible run-time errors.

You can also use `.Release()` or `->ReleaseAndDelete()` to allow a matrix expression to recycle space. Suppose you call

```
A.Release();
```

just before `A` is used just once in an expression. Then the memory used by `A` is either returned to the system or reused in the expression. In either case, `A`'s memory is destroyed. This procedure can be used to improve efficiency and reduce the use of memory.

Use `->ReleaseAndDelete` for matrices created by new if you want to completely delete the matrix after it is accessed.

3.16 Efficiency

The package tends to be not very efficient for dealing with matrices with short rows. This is because some administration is required for accessing rows for a variety of types of matrices. To reduce the administration a special multiply routine is used for rectangular matrices in place of the generic one. Where operations can be done without reference to the individual rows (such as adding matrices of the same type) appropriate routines are used.

When you are using small matrices (say smaller than 10 x 10) you may find it faster to use rectangular matrices rather than the triangular or symmetric ones.

3.17 Output

To print a matrix use an expression like

```
Matrix A;
......
cout << setw(10) << setprecision(5) << A;
```

This will work only with systems that support the standard input/output routines including manipulators. You need to #include the files iostream.h, iomanip.h, newmatio.h in your C++ source files that use this facility. The files iostream.h, iomanip.h will be included automatically if you include the statement `#define WANT_STREAM` at the beginning of your source file. So you can begin your file with either

```
#define WANT_STREAM
#include "newmatio.h"
```

or

```
#include <iostream.h>
#include <iomanip.h>
#include "newmatio.h"
```

The present version of this routine is useful only for matrices small enough to fit within a page or screen width.

To print several vectors or matrices in columns use a concatenation operator:

```
ColumnVector A, B;
.....
cout << setw(10) << setprecision(5) << (A | B);
```

3.18 Unspecified type

Skip this section on your first reading.

If you want to work with a matrix of unknown type, say in a function. You can construct a matrix of type `GenericMatrix`. Eg

```
Matrix A;
.....                                   // put some values in A
GenericMatrix GM = A;
```

A GenericMatrix matrix can be used anywhere where a matrix expression can be used and also on the left hand side of an =. You can pass any type of matrix (excluding the Crout and BandLUMatrix types) to a `const GenericMatrix&` argument in a function. However most scalar functions including Nrows(), Ncols(), Type() and element access do not work with it. Nor does the ReturnMatrix construct. See also the paragraph on LinearEquationSolver.

An alternative and less flexible approach is to use BaseMatrix or GeneralMatrix.

Suppose you wish to write a function which accesses a matrix of unknown type including expressions (eg `A*B`). Then use a layout similar to the following:

```
void YourFunction(BaseMatrix& X)
{
   GeneralMatrix* gm = X.Evaluate();    // evaluate an expression
                                        // if necessary
   ........                             // operations on *gm
   gm->tDelete();                       // delete *gm if a temporary
}
```

See, as an example, the definitions of `operator<<` in newmat9.cpp.

Under certain circumstances; particularly where `X` is to be used just once in an expression you can leave out the `Evaluate()` statement and the corresponding `tDelete()`. Just use `X` in the expression.

If you know YourFunction will never have to handle a formula as its argument you could also use

```
void YourFunction(const GeneralMatrix& X)
{
   ........                             // operations on X
}
```

Do not try to construct a GeneralMatrix or BaseMatrix.

3.19 Cholesky decomposition

Suppose `S` is symmetric and positive definite. Then there exists a unique lower triangular matrix `L` such that `L * L.t() = S`. To calculate this use

```
SymmetricMatrix S;
......
LowerTriangularMatrix L = Cholesky(S);
```

If `S` is a symmetric band matrix then `L` is a band matrix and an alternative procedure is provided for carrying out the decomposition:

```
SymmetricBandMatrix S;
......
LowerBandMatrix L = Cholesky(S);
```

3.20 QR decomposition

This is a variant on the usual QR transformation.

Start with matrix (dimensions shown to left and below the matrix)

```
/ 0    0 \      s
\ X    Y /      n

  s    t
```

Our version of the QR decomposition multiplies this matrix by an orthogonal matrix Q to get

```
/ U    M \      s
\ 0    Z /      n

  s    t
```

where `U` is upper triangular (the R of the QR transform). That is

```
Q  / 0   0 \  =  / U   M \
   \ X   Y /     \ 0   Z /
```

This is good for solving least squares problems: choose b (matrix or column vector) to minimise the sum of the squares of the elements of

```
Y - X*b
```

Then choose `b = U.i()*M;` The residuals `Y - X*b` are in `Z`.

This is the usual QR transformation applied to the matrix `X` with the square zero matrix concatenated on top of it. It gives the same triangular matrix as the QR transform applied directly to `X` and generally seems to work in the same way as the usual QR transform. However it fits into the matrix package better and also gives us the residuals directly. It turns out to be essentially a modified Gram-Schmidt decomposition.

Two routines are provided in *newmat*:

```
QRZ(X, U);
```

replaces `X` by orthogonal columns and forms `U`.

```
QRZ(X, Y, M);
```

uses `X` from the first routine, replaces `Y` by `Z` and forms `M`.

The are also two routines `QRZT(X, L)` and `QRZT(X, Y, M)` which do the same decomposition on the transposes of all these matrices. QRZT replaces the routines HHDecompose in earlier versions of newmat. HHDecompose is still defined but just calls QRZT.

For an example of the use of this decomposition see the file example.cpp.

3.21 Singular value decomposition

The singular value decomposition of an *m* x *n* `Matrix` `A` (where *m* >= *n*) is a decomposition

```
A  = U * D * V.t()
```

where `U` is *m* x *n* with `U.t() * U` equalling the identity, `D` is an *n* x *n* `DiagonalMatrix` and `V` is an *n* x *n* orthogonal matrix (type `Matrix` in *Newmat*).

Singular value decompositions are useful for understanding the structure of ill-conditioned matrices, solving least squares problems, and for finding the eigenvalues of `A.t() * A`.

To calculate the singular value decomposition of `A` (with *m* >= *n*) use one of

```
SVD(A, D, U, V);                  // U = A is OK
SVD(A, D);
SVD(A, D, U);                     // U = A is OK
SVD(A, D, U, false);              // U (can = A) for workspace only
SVD(A, D, U, V, false);           // U (can = A) for workspace only
```

where `A`, `U` and `V` are of type `Matrix` and `D` is a `DiagonalMatrix`. The values of `A` are not changed unless `A` is also inserted as the third argument.

The elements of `D` are sorted in *descending* order.

Remember that the SVD decomposition is not completely unique. The signs of the elements in a column of `U` may be reversed if the signs in the corresponding column in `V` are reversed. If a number of the singular values are identical one can apply an orthogonal transformation to the corresponding columns of `U` and the corresponding columns of `V`.

3.22 Eigenvalue decomposition

An eigenvalue decomposition of a SymmetricMatrix `A` is a decomposition

```
A  = V * D * V.t()
```

where `V` is an orthogonal matrix (type `Matrix` in *Newmat*) and `D` is a DiagonalMatrix.

Eigenvalue analyses are used in a wide variety of engineering, statistical and other mathematical analyses.

The package includes two algorithms: Jacobi and Householder. The first is extremely reliable but much slower than the second.

The code is adapted from routines in *Handbook for Automatic Computation, Vol II, Linear Algebra* by Wilkinson and Reinsch, published by Springer Verlag.

```
Jacobi(A,D,S,V);                  // A, S symmetric; S is workspace,
                                  //    S = A is OK; V is a matrix
Jacobi(A,D);                      // A symmetric
Jacobi(A,D,S);                    // A, S symmetric; S is workspace,
                                  //    S = A is OK
Jacobi(A,D,V);                    // A symmetric; V is a matrix

EigenValues(A,D);                 // A symmetric
EigenValues(A,D,S);               // A, S symmetric; S is for back
                                  //    transforming, S = A is OK
EigenValues(A,D,V);               // A symmetric; V is a matrix
```

where `A`, `S` are of type `SymmetricMatrix`, `D` is of type `DiagonalMatrix` and `V` is of type `Matrix`. The values of `A` are not changed unless `A` is also inserted as the third argument. If you need eigenvectors use one of the forms with matrix `V`. The eigenvectors are returned as the columns of `V`.

The elements of `D` are sorted in *ascending* order.

Remember that an eigenvalue decomposition is not completely unique - see the comments about the SVD decomposition.

3.23 Sorting

To sort the values in a matrix or vector, `A`, (in general this operation makes sense only for vectors and diagonal matrices) use one of

```
SortAscending(A);

SortDescending(A);
```

I use the quicksort algorithm. The algorithm is similar to that in Sedgewick's algorithms in C++. If the sort seems to be failing (as quicksort can do) an exception is thrown.

You will get incorrect results if you try to sort a band matrix - but why would you want to sort a band matrix?

3.24 Fast Fourier transform

```
FFT(X, Y, F, G);                          // F=X and G=Y are OK
```

where `X`, `Y`, `F`, `G` are column vectors. `X` and `Y` are the real and imaginary input vectors; `F` and `G` are the real and imaginary output vectors. The lengths of `X` and `Y` must be equal and should be the product of numbers less than about 10 for fast execution.

The formula is

```
       n-1
h[k] = SUM  z[j] exp (-2 pi i jk/n)
       j=0
```

where `z[j]` is stored complex and stored in `X(j+1)` and `Y(j+1)`. Likewise `h[k]` is complex and stored in `F(k+1)` and `G(k+1)`. The fast Fourier algorithm takes order *n* log(*n*) operations (for *good* values of *n*) rather than *n***2 that straight evaluation (see the file `tmtf.cpp`) takes.

I use one of two methods:

- A program originally written by Sande and Gentleman. This requires that *n* can be expressed as a product of small numbers.
- A method of Carl de Boor (1980), *Siam J Sci Stat Comput*, pp 173-8. The sines and cosines are calculated explicitly. This gives better accuracy, at an expense of being a little slower than is otherwise possible. This is slower than the Sande-Gentleman program but will work for all *n* --- although it will be very slow for *bad* values of *n*.

Related functions

```
FFTI(F, G, X, Y);                         // X=F and Y=G are OK
RealFFT(X, F, G);
RealFFTI(F, G, X);
```

`FFTI` is the inverse transform for `FFT`. `RealFFT` is for the case when the input vector is real, that is `Y = 0`. I assume the length of `X`, denoted by *n*, is *even*. That is, *n* must be divisible by 2. The program sets the lengths of `F` and `G` to *n*/2 + 1. `RealFFTI` is the inverse of `RealFFT`.

See also the section on fast trigonometric transforms.

3.25 Fast trigonometric transforms

These are the sin and cosine transforms as defined by Charles Van Loan (1992) in *Computational frameworks for the fast Fourier transform* published by SIAM. See page 229. Some other authors use slightly different conventions. All the functions call the fast Fourier transforms and require an *even* transform length, denoted by *m* in these notes. That is, *m* must be divisible by 2. As with the FFT *m* should be the product of numbers less than about 10 for fast execution.

The functions I define are

```
DCT(U,V);                // U, V are ColumnVectors, length m+1
DCT_inverse(V,U);        // inverse of DCT
DST(U,V);                // U, V are ColumnVectors, length m+1
DST_inverse(V,U);        // inverse of DST
DCT_II(U,V);             // U, V are ColumnVectors, length m
DCT_II_inverse(V,U);     // inverse of DCT_II
DST_II(U,V);             // U, V are ColumnVectors, length m
DST_II_inverse(V,U);     // inverse of DST_II
```

where the first argument is the input and the second argument is the output. `V = U` is OK. The length of the output ColumnVector is set by the functions.

Here are the formulae:

DCT

$$v[k] = u[0]/2 + \sum_{j=1}^{m-1} \{ u[j] \cos (\pi jk/m) \} + (-)^k u[m]/2$$

for `k = 0...m`, where `u[j]` and `v[k]` are stored in `U(j+1)` and `V(k+1)`.

DST

$$v[k] = \sum_{j=1}^{m-1} \{ u[j] \sin (\pi jk/m) \}$$

for `k = 1...(m-1)`, where `u[j]` and `v[k]` are stored in `U(j+1)` and `V(k+1)`and where `u[0]` and `u[m]` are ignored and `v[0]` and `v[m]` are set to zero. For the inverse function `v[0]` and `v[m]` are ignored and `u[0]` and `u[m]` are set to zero.

DCT_II

$$v[k] = \sum_{j=0}^{m-1} \{ u[j] \cos (\pi (j+1/2)k/m) \}$$

for `k = 0...(m-1)`, where `u[j]` and `v[k]` are stored in `U(j+1)` and `V(k+1)`.

DST_II

$$v[k] = \sum_{j=1}^{m} \{ u[j] \sin (\pi (j-1/2)k/m) \}$$

for `k = 1...m`, where `u[j]` and `v[k]` are stored in `U(j)` and `V(k)`.

Note that the relationship between the subscripts in the formulae and those used in *newmat* is different for DST_II (and DST_II_inverse).