DOCUMENTATION FOR NEWMAT09, A MATRIX LIBRARY IN C++

September, 1997

This is the *how to use* documentation for *newmat* plus the background information on its design.

This document is available as an ASCII file, newmat.txt, and in hypertext format for reading with an HTML browser such as Netscape or Internet Explorer. Cross-references in the ASCII version are given as section numbers. The HTML version can be downloaded from the same sites [1.5] as the source files.

Be sure to read the section on customising [2.3] before attempting to compile newmat.

1 Introduction

1.1 Conditions of use

Please report bugs to me at

robertd@netlink.co.nz

When reporting a bug please tell me which C++ compiler you are using (if known), and what version. Also give me details of your computer (if known). And tell me which version of *newmat* (e.g. newmat03 or newmat04) you are using. Note any changes you have made to my code. If at all possible give me a piece of code illustrating the bug. See the problem report form [7].

Please do report bugs to me.

1.2 General description

The package is intended for scientists and engineers who need to manipulate a variety of types of matrices using standard matrix operations. Emphasis is on the kind of operations needed in statistical calculations such as least squares, linear equation solve and eigenvalues.

It supports matrix types

Matrix	(rectangular matrix)
nricMatrix	(variant of rectangular matrix)
UpperTriangularMatrix	
LowerTriangularMatrix	
DiagonalMatrix	
SymmetricMatrix	
BandMatrix	
UpperBandMatrix	(upper triangular band matrix)
LowerBandMatrix	(lower triangular band matrix)
SymmetricBandMatrix	
RowVector	(derived from Matrix)
ColumnVector	(derived from Matrix).

Only one element type (float or double) is supported.

The package includes the operations `*`, `+`, `-`, concatenation, inverse, transpose, conversion between types, submatrix, determinant, Cholesky decomposition, QR triangularisation, singular value decomposition, eigenvalues of a symmetric matrix, sorting, fast Fourier transform, printing and an interface with *Numerical Recipes in C*.

It is intended for matrices in the range 15 x 15 to the maximum size your machine will accommodate in a single array. On a PC using the large model, this is 90 x 90 (125 x 125 for triangular matrices). For UNIX machines or PCs running under a 32 bit flat model the limit is set by the size of available memory. The number of elements in an array cannot exceed the maximum size of an *int*. The package will work for very small matrices but becomes rather inefficient. Some of the factorisation functions are not (yet) optimised for paged memory and so become inefficient when used with very large matrices.

A *lazy evaluation* approach to evaluating matrix expressions is used to improve efficiency and reduce the use of temporary storage.

I have tested the package on variety of compilers and platforms including AT&T, Borland, Gnu, HP, Microsoft, Sun and Watcom. For more details see the section on compiler performance [2.4].

1.3 Is this the library for you?

Do you

1: understand `*` to mean matrix multiply and not element by element multiply
2: need matrix operators such as `*` and `+` defined as operators so you can write things like
```
X  = A * (B + C);
```
3: need a variety of types of matrices (but not sparse);
4: need only one element type (float or double);
5: work with matrices in the range 10 x 10 up to what can be stored in memory;
6: tolerate a large package.

Then maybe this is the right package for you.

1.4 Other matrix libraries

For details of other C++ matrix libraries look at one of

- http://webnz.com/robert/cpp_site.html
- http://webnz.co.nz/robert/cpp_site.html
- ftp://webnz.com/robert/cpp_site.txt
- ftp://webnz.co.nz/robert/cpp_site.txt.

Look at the section *lists of libraries* which gives the locations of several very comprehensive lists of matrix and other C++ libraries.

1.5 Where to find this library

- http://webnz.com/robert/
- http://webnz.co.nz/robert/
- ftp://webnz.com/robert/newmat09.zip
- ftp://webnz.com/robert/newmat09.tar.gz
- ftp://webnz.co.nz/robert/newmat09.zip

- ftp://webnz.co.nz/robert/newmat09.tar.gz

For the HTML version of the documentation download either nm09html.zip or nm09html.tar.gz from the same sites.

1.6 How to contact the author

Robert Davies
16 Gloucester Street
Wilton
Wellington
New Zealand

email: robertd@netlink.co.nz

(don't forget the *d* in *robertd*).

1.7 Change history

Newmat09 - September, 1997: Operator ==, !=, +=, -=, *=, /=, |=, &=. Follow new rules for *for (int i; ...)* construct. Change Boolean, TRUE, FALSE to bool, true, false. Change ReDimension to ReSize. SubMatrix allows zero rows and columns. Scalar +, - or * matrix is OK. Simplify simulated exceptions. Fix non-linear programs for AT&T compilers. Dummy inequality operators. Improve internal row/column operations. Improve matrix LU decomposition. Improve sort. Reverse function. IsSingular function. Fast trig transforms. Namespace definitions.

Newmat08A - July, 1995: Fix error in SVD.

Newmat08 - January, 1995: Corrections to improve compatibility with Gnu, Watcom. Concatenation of matrices. Elementwise products. Option to use compilers supporting exceptions. Correction to exception module to allow global declarations of matrices. Fix problem with inverse of symmetric matrices. Fix divide-by-zero problem in SVD. Include new QR routines. Sum function. Non-linear optimisation. GenericMatrices.

Newmat07 - January, 1993: Minor corrections to improve compatibility with Zortech, Microsoft and Gnu. Correction to exception module. Additional FFT functions. Some minor increases in efficiency. Submatrices can now be used on RHS of =. Option for allowing C type subscripts. Method for loading short lists of numbers.

Newmat06 - December 1992: Added band matrices; *real* changed to *Real* (to avoid potential conflict in complex class); Inject doesn't check for no loss of information; fixes for AT&T C++ version 3.0; real(A) becomes A.AsScalar(); CopyToMatrix becomes AsMatrix, etc.; .c() is no longer required (to be deleted in next version); option for version 2.1 or later. Suffix for include files changed to .h; BOOL changed to Boolean (BOOL doesn't work in g++ v 2.0); modifications to allow for compilers that destroy temporaries very quickly; (Gnu users - see the section of compiler performance). Added CleanUp, LinearEquationSolver, primitive version of exceptions.

Newmat05 - June 1992: For private release only

Newmat04 - December 1991: Fix problem with G++1.40, some extra documentation

Newmat03 - November 1991: Col and Cols become Column and Columns. Added Sort, SVD, Jacobi, Eigenvalues, FFT, real conversion of 1x1 matrix, *Numerical Recipes in C* interface, output operations, various scalar functions. Improved return from functions. Reorganised setting options in *include.hxx*.

Newmat02 - July 1991: Version with matrix row/column operations and numerous additional functions.

Matrix - October 1990: Early version of package.

1.8 References

The matrix LU decomposition is from Golub, G.H. & Van Loan, C.F. (1996), *Matrix Computations*, published by John Hopkins University Press. Part of the matrix inverse/solve routine is adapted from Press, Flannery, Teukolsky, Vetterling (1988), *Numerical Recipes in C*, published by the Cambridge University Press.

Many of the advanced matrix routines are adapted from routines in Wilkinson and Reinsch (1971), *Handbook for Automatic Computation, Vol II, Linear Algebra* published by Springer Verlag.

The fast Fourier transform is adapted from Carl de Boor (1980), *Siam J Sci Stat Comput*, pp173-8 and the fast trigonometric transforms from Charles Van Loan (1992) in *Computational frameworks for the fast Fourier transform* published by SIAM.

The sort function is derived from Sedgewick, Robert (1992), *Algorithms in C++* published by Addison Wesley.

For references about *newmat* see

Davies, R.B. (1994) Writing a matrix package in C++. In OON-SKI'94: The second annual object-oriented numerics conference, pp 207-213. Rogue Wave Software, Corvallis.

Eddelbuttel, Dirk (1996) Object-oriented econometrics: matrix programming in C++ using GCC and Newmat. Journal of Applied Econometrics, Vol 11, No 2, pp 199-209.

2 Getting started

2.1 Overview

I use .h as the suffix of definition files and .cpp as the suffix of C++ source files.

You will need to compile all the *.cpp files listed as program files in the files section [6] to get the complete package. Ideally you should store the resulting object files as a library. The tmt*.cpp files are used for testing [2.7], example.cpp is an example [2.6] and sl_ex.cpp, nl_ex.cpp and garch.cpp are examples of the non-linear [3.27] solve and optimisation routines. A demonstration of the exception mechanism is in test_exc.cpp.

I include a number of *make* files for compiling the example and the test package. See the section on make files [2.2] for details. But with the PC compilers, its pretty quick just to load all the files in the interactive environments by pointing and clicking.

Use the large or win32 console model when you are using a PC. Do not *outline* inline functions. You may need to increase the stack size.

Your source files that access *newmat* will need to #include one or more of the following files.

- include.h: if you want to access just the compiler options
- newmat.h: to access just the main matrix library (includes include.h)
- newmatap.h: to access the advanced matrix routines such as Cholesky decomposition, QR triangularisation etc (includes newmat.h)
- newmatio.h: to access the output routines (includes newmat.h) You can use this only with compilers that support the standard input/output routines including manipulators.
- newmatnl.h: to access the non-linear optimisation routines (includes newmat.h)

See the section on customising [2.3] to see how to edit include.h for your environment and the section on compilers [2.4] for any special problems with the compiler you are using.

2.2 Make files

I have included *make* files for a number of compilers. These provide an alternative way of compiling your programs than with the IDE that comes with PC compilers. See the files section [6] for details. See the example [2.6] for how to use them. Leave out the target name to compile and link all my examples and test files.

2.2.1 PC

I include make files for most of the various PC compilers I have access to. They all use the make utility that comes with the compiler. I don't know whether they work with versions of the compilers other than the ones I have. The make files for Borland need editing to show where you have stored your Borland compiler.

2.2.2 UNIX

The *make* files for the UNIX compilers link a .cxx file to each .cpp file since some of these compilers do not recognise .cpp as a legitimate extension for a C++ file. I suggest you delete this part of the *make* file and, if necessary, rename the .cpp files to something your compiler recognises.

My *make* files for UNIX systems are for use with `gmake` rather than `make`. Ordinary `make` works with them on the Sun but not the Silicon Graphics or HP machines. On Linux use `make`.

To compile everything with the CC compiler use

```
gmake -f cc.mak
```

or for the gnu compiler use

```
gmake -f gnu.mak
```

I have set O2 optimisation for CC but not for gnu, since I couldn't get it to compile with O2 set. You may need to change these options.

There is a line in the make file `rm -f -i $*.cxx`. Some systems won't accept this line and you will need to delete it. In this case, if you have a bad compile and you are using my scheme for linking .cxx files, you will need to delete the .cxx file link generated by that compile before you can do the next one. You may get away with just deleting the option `-i` in which case you won't be asked before a .cxx file is deleted.

2.3 Customising

The file include.h sets a variety of options including several compiler dependent options. You may need to edit include.h to get the options you require. If you are using a compiler different from one I have worked with you may have to set up a new section in include.h appropriate for your compiler.

Borland, Turbo, Gnu, Microsoft and Watcom are recognised automatically. If none of these are recognised a default set of options is used. These are fine for AT&T, HPUX and Sun C++. If you using a compiler I don't know about you may have to write a new set of options.

There is an option in include.h for selecting whether you use compiler supported exceptions, simulated exceptions, or disable exceptions. Use the option for compiler supported exceptions if and only if you have set the option on your compiler to recognise exceptions. Disabling exceptions sometimes helps with compilers that are incompatible with my exception simulation scheme.

If your compiler recognises bool as required by the standard activate the statement `#define bool_LIB`. This will deactivate my Boolean class.

Activate the appropriate statement to make the element type float or double.

I suggest you leave the option TEMPS_DESTROYED_QUICKLY activated, even though the Gnu compiler (<2.6) is the only one I know about that requires it (C-Set also requires it?). This stores the *trees* describing matrix expressions on the heap rather than the stack and, surprisingly, seems to give better performance. See the discussion on destruction of temporaries [8.8] for more explanation.

Leave the option TEMPS_DESTROYED_QUICKLY_R not activated unless you are using the Gnu G++ [2.4.3] compiler earlier than version 2.6. This option controls whether the ReturnMatrix [3.13] construct uses the stack or the heap. The heap version is rather kludgy and probably should be avoided where possible.

The option DO_FREE_CHECK is used for tracking memory leaks and normally should not be activated.

Activate SETUP_C_SUBSCRIPTS if you want to use traditional C style element access [3.2].

2.4 Compiler performance

I have tested this library on a number of compilers. Here are the levels of success and any special considerations. In most cases I have chosen code that works under all the compilers I have access to, but I have had to include some specific work-arounds for some compilers. For the PC versions, I use a 486dx computer running windows 95, a Pentium computer running windows NT or Linux (Red Hat 4.1) and a 386sx running MSDOS 5. The UNIX versions are on a Sun Sparc station or a HP UNIX workstation. Thanks to Victoria University and Industrial Research Ltd for access to the UNIX machines.

I have set up a block of code for each of the compilers in include.h. Turbo, Borland, Gnu, Microsoft and Watcom are recognised automatically. There is a default option that works for AT&T, Sun C++ 4.0.1 and HPUX. So you don't have to make any changes for these compilers. Otherwise you may have to build your own set of options in include.h.

2.4.1 AT&T

AT&T C++ 2.1; 3.0.1 on a Sun: Previous versions worked on these compilers, which I no longer have access to.

In AT&T 2.1 you may get an error when you use an expression for the single argument when constructing a Vector or DiagonalMatrix or one of the Triangular Matrices. You need to evaluate the expression separately.

2.4.2 Borland

Borland C++ 3.1, 4.5, 5.01A: Recently this has been my main development platform, so naturally everything works with this compiler. There was a problem with the library utility in version 2.0 which is now fixed. You will need to use the large or 32 bit flat model. If you are not debugging, turn off the options that collect debugging information. Make sure you don't run Borland's exceptions and my simulated exceptions at the same time.

If you are using version 5 remember to edit include.h to deactivate my Boolean class.

When running my test program with Borland 4.5 under ms-dos you may run out of memory. Either compile the test routine to run under *easywin* or use simulated exceptions rather than the built in exceptions. Under *easywin* the test program indicates a memory leak. I presume this is partly because of the way windows organises its heap rather than there being a real problem.

If you can, upgrade to windows 95 or window NT and use the 32 bit console model.

If you are using the 16 bit large model, don't forget to keep all matrices less than 64K bytes in length (90x90 for a rectangular matrix if you are using `double` as your element type). Otherwise your program will crash without warning or explanation. You may need to break the tmt [2.7] set of test files into two parts to get the program to fit into your computer.

In version 4.5, under *easywin* the automatic clean-up of objects by the exception mechanism does not seem to work correctly. Use my simulated exceptions if this is a problem.

One version of Borland had DBL_MIN incorrectly defined. If you are using an older version of Borland and are getting strange numerical errors in the test programs reinstate the commented out statements in precision.h.

I include make files that work under Borland 4.5 and 5. You will need to edit these to correctly locate the directories for the include and library files. Both assume you are using simulated exceptions. `BC.MAK` also works with version 3.1 if you delete the options that are not recognised and the reference to the library file that is not recognised.

2.4.3 Gnu G++

Gnu G++ 2.6.0, 2.7.2: These work OK.

If you are using 2.7.2 remember to edit include.h to deactivate my Boolean class.

For versions earlier than 2.6.0 you must enable the options TEMPS_DESTROYED_QUICKLY and TEMPS_DESTROYED_QUICKLY_R. You can't use expressions like `Matrix(X*Y)` in the middle of an expression and `(Matrix)(X*Y)` is unreliable. If you write a function returning a matrix, you MUST use the ReturnMatrix [3.13] method described in this documentation. This is because g++ destroys temporaries occurring in an expression too soon for the two stage way of evaluating expressions that newmat uses. You will have problems with versions of Gnu earlier than 2.3.1.

Gnu seems to leave some rubbish on the stack. Possibly this is a buffer or dynamically loaded subprogram so may not be a bug.

Linux: It works fine on my copy of G++ 2.7.2. In 2.6.?, `fabs(*X++)` causes a problem. You may need to write you own non-inlined version.

2.4.4 HP-UX

HP 9000 series HP-UX. The current version works without problems with the simulated exceptions; haven't tried the built-in exceptions.

Here are comments I made two years ago.

I have tried the library on two versions of HP-UX. (I don't know the version numbers, the older is a clone of AT&T 3, the newer is HP's version with exceptions). Both worked after the modifications described in this section.

With the older version of the compiler I needed to edit the math.h library file to remove a duplicate definition of abs.

With the newer version you can set the +eh option to enable exceptions and activate the UseExceptions option in include.h. If you are using my make file, you will need to replace CC with CC +eh where ever CC occurs. I recommend that you do not do this and either disable exceptions or use my simulated exceptions. I get core dumps when I use the built-in exceptions and suspect they are not sufficiently debugged as yet.

If you are using my simulated exceptions you may get a mass of error messages from the linker about __EH_JMPBUF_TEMP. In this case get file setjmp.h (in directory /usr/include/CC ?) and put extern in front of the line

```
  jmp_buf * __EH_JMPBUF_TEMP;
```

The file setjmp.h is accessed in my file myexcept.h. You may want to change the #include statement to access your edited copy of setjmp.h.

2.4.5 Microsoft

Microsoft Visual C++ 2.0: Seems to work OK. You can use the makefile `ms_nt.mak`.

You must `#define TEMPS_DESTROYED_QUICKLY` owing to a bug in version 7 (at least) of MSC. There are some notes in the file `include.h` on changes to run under version 7. I haven't tried newmat09 on version 7.

Microsoft Visual C++ 1.51. Disable exceptions, comment out the line in include.h `#define TEMPS_DESTROYED_QUICKLY_R`. In `tmt.cpp`, comment out the `Try` and `CatchAll` lines at the beginning of `main()` and the line `trymati()`. You can use the makefile `ms.mak`. You will probably need to break the tmt [2.7] test files into two parts to get the program to link.

If you can, upgrade to windows 95 or window NT and use the 32 bit console model.

If you are using the 16 bit large model, don't forget to keep all matrices less than 64K bytes in length (90x90 for a rectangular matrix if you are using `double` as your element type). Otherwise your program will crash without warning or explanation. You may need to break the tmt [2.7] set of test files into two parts to get the program to fit into your computer.

Microsoft Visual C++ 4: I haven't tried this - a correspondent reports: I use Microsoft Visual C++ Version 4. there is only one minor problem. In all files you must include `#include "stdafx.h"` (presumably if you are using MFC). This file contains essential information for VC++. Leave it out and you get *Unexpected end of file*.

Microsoft Visual C++ 5: I have tried this in console mode and it seems to work satisfactorily. There may be a problem with namespace [3.29]. Turn optimisation off. Edit `include.h` to use the compiler supported exceptions.

2.4.6 Sun

Sun C++ (version 4.2): This seems to work fine.

2.4.7 Watcom & Optima++

Watcom C++ (version 10): this works fine. Don't try to run Watcom's exceptions and my simulated exceptions at the same time.

But best don't use Watcom's exceptions at all in this version as they do seem to cause problems; even when not used.

It seems fine on Optima++ 1.0 in console mode. I haven't tested it in gui mode. Optima++ provides some stuff to insert in the main program.

2.4.8 Zortech

I don't support Zortech any more and haven't tried the Symantec successors to Zortech.

2.5 Updating from previous versions

This is a minor upgrade on newmat08 to correct an error in SVD, improve compatibility with the standard, improve performance and add some new functions. You should upgrade.

- Boolean, TRUE, FALSE are now bool, true, false. See customising [2.3] if your compiler supports the bool class.
- ReDimension is now ReSize [3.10].
- The simulated exception [3.25] package has been updated.
- Operators `==`, `!=`, `+=`, `-=`, `*=`, `|=`, `&=` are now supported as binary [3.6] matrix operators.
- `A+=f`, `A-=f`, `A*=f`, `A/=f`, `f+A`, `f-A`, `f*A` are supported for `A` matrix, `f` scalar [3.7].
- Fast trigonometric transforms [3.23].
- Reverse [3.5] function for reversing order of elements in a vector or matrix.
- IsSingular [3.8] function.
- An option is included for defining namespaces [3.29].
- Dummy inequality operators are defined for compatibility with the STL.
- The row/column classes in newmat3.cpp have been modified to improve efficiency and correct an invalid use of pointer arithmetic. Most users won't be using these classes explicitly; if you are, please contact me for details of the changes.
- Matrix LU decomposition rewritten (faster for large arrays).
- The sort function rewritten (faster).
- The documentation files newmata.txt and newmatb.txt have been amalgamated and both are included in the hypertext version.
- Some of the make [2.2] files reorganised again.

If you are upgrading from newmat07 note the following:

- .cxx files are now .cpp files. Some versions of won't accept .cpp. The *make* files for Gnu and AT&T link the .cpp files to .cxx files before compilation and delete the links after compilation.
- An option [2.3] in include.h allows you to use compiler supported exceptions, simulated exceptions or disable exceptions. Edit the file include.h to select one of these three options. Don't simulate exceptions if you have set your compiler's option to implement exceptions.
- New QR decomposition [3.18] functions.
- A non-linear least squares [3.27] class.

- No need to explicitly set the AT&T option in include.h.
- Concatenation and elementwise multiplication [3.6].
- A new GenericMatrix [3.16] class.
- Sum [3.8] function.
- Some of the make [2.2] files reorganised.

If you are upgrading from newmat06 note the following:

If you are using << to load a Real into a submatrix change this to =.

If you are upgrading from newmat03 or newmat04 note the following

- .hxx files are now .h files
- real changed to Real
- BOOL changed to Boolean
- CopyToMatrix changed to AsMatrix, etc
- real(A) changed to A.AsScalar()

The current version is quite a bit longer that newmat04, so if you are almost out of space with newmat04, don't throw newmat04 away until you have checked your program will work under this version.

See the change history [1.7] for other changes.

2.6 Example

An example is given in `example.cpp`. This gives a simple linear regression example using five different algorithms. The correct output is given in `example.txt`. The program carries out a rough check that no memory is left allocated on the heap when it terminates. See the section on testing [2.7] for a comment on the reliability of this check and the use of the DO_FREE_CHECK option.

I include a variety of make files. To compile the example use a command like

```
gmake example -f gnu.mak                (Gnu G++)
gmake example -f cc.mak                 (AT&T, HPUX, Sun)
nmake example.exe -f ms_nt.mak          (Microsoft Visual C++ 2.0)
nmake example.exe -f ms.mak             (Microsoft Visual C++ 1.51)
make -f bc.mak example.exe              (Borland C++ 4.5, 5)
make -f bc32.mak example.exe            (Borland C++ 4.5, 5, 32 bit)
wmake example.exe -f watcom.mak         (Watcom C++ 10A)
wmake example.exe -f watco_nt.mak       (Watcom C++ 10A, for win NT)
```

To compile all the example and test files use a command like

```
gmake -f gnu.mak                        (Gnu G++)
```

The example uses io manipulators. It will not work with a compiler that does not support the standard io manipulators.

Other example files are `nl_ex.cpp` and `garch.cpp` for demonstrating the non-linear fitting routines, `sl_ex` for demonstrating the solve function and `test_exc` for demonstrating the exceptions.

2.7 Testing

The library package contains a comprehensive test program in the form of a series of files with names of the form tmt?.cxx. The files consist of a large number of matrix formulae all of which evaluate to zero (except the first one which is used to check that we are detecting non-zero matrices). The printout should state that it has found just one non-zero matrix.

The test program should be run with *Real* typedefed to *double* rather than *float* in include.h [2.3].

Make sure the C subscripts [3.2] are enabled if you want to test these.

Various versions of the make file (extension .mak) are included with the package. See the section on make files [2.2].

The program also allocates and deletes a large block and small block of memory before it starts the main testing and then at the end of the test. It then checks that the blocks of memory were allocated in the same place. If not then one suspects that there has been a memory leak. i.e. a piece of memory has been allocated and not deleted.

This is not completely foolproof. Programs may allocate extra print buffers while the program is running. I have tried to overcome this by doing a print before I allocate the first memory block. Programs may allocate memory for different sized items in different places, or might not allocate items consecutively. Or they might mix the items with memory blocks from other programs. Nevertheless, I seem to get consistent answers from many of the compilers I am working with, so I think this is a worthwhile test.

If the DO_FREE_CHECK [2.3] option in include.h is activated, the program checks that each *new* is balanced with exactly one *delete*. This provides a more definitive test of no memory leaks. There are additional statements in myexcept.cpp which can be activated to print out details of the memory being allocated and released.

I have included a facility for checking that each piece of code in the library is really exercised by the test routines. Each block of code in the main part of the library contains a word *REPORT*. *newmat.h* has a line defining *REPORT* that can be activated (deactivate the dummy version). This gives a printout of the number of times each of the *REPORT* statements in the *.cpp* files is accessed. Use a grep with line numbers to locate the lines on which *REPORT* occurs and compare these with the lines that the printout shows were actually accessed. One can then see which lines of code were not accessed.

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);
```

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 [3.3] or a re-dimension statement [3.10].

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 (ie 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 [3.4] data).

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 [3.9].

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

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

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 submatrices, or for long lists. Also try to restrict its use 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)
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 [3.12].
- 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 [3.9].
- See the section on submatrices [3.9] 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 [3.28]. If actually called, they will throw an exception. So don't try to sort a *list* of matrices.

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

```
int m = A.Nrows();                      // number of rows
int n = A.Ncols();                      // number of columns
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 mav = A.MaximumAbsoluteValue();    // maximum of absolute 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 t = A.Trace();                     // trace
LogAndSign ld = A.LogDeterminant();     // log of determinant
bool z = A.IsZero();                    // test all elements zero
MatrixType mt = A.Type();               // type of matrix
Real* s = Store();                      // pointer to array of elements
int l = Storage();                      // length of array of elements
bool s = A.IsSingular();                // A is a CroutMatrix or
                                           BandLUMatrix
MatrixBandWidth mbw = A.BandWidth();    // upper and lower bandwidths
```

`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.

`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.

`MatrixType mt = A.Type()` returns the type of a matrix. Use `(char*)mt` 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.

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

3.9 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. This 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 lr = fr-1, lc = fc-1 or 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 its type is 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.

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

If you are using the submatrix facility to build a matrix from a small number of components, consider instead using the concatenation operators [3.6].

3.10 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
```

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

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
```

3.11 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.

3.12 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. If you want to use the LinearEquationSolver `#include newmatap.h`.

3.13 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 copies 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.

3.14 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.

3.15 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 file 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 [3.6]:

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

3.16 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 [3.12].

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.

3.17 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);
```

3.18 QR decomposition

This is a variant on the usual QR transformation.

Start with 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).

This is good for solving least squares problems: choose b (matrix or row 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 attached 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:

```
    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.

3.19 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 decomposition is 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.

3.20 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`.

3.21 Sorting

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

```
    SortAscending(A);
```

or

```
    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?

3.22 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[1]

$$h_k = \sum_{j=0}^{n-1} z_j \exp(-2\pi ijk / n)$$

where z_j is 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 would take.

I use the 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.

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. The program sets the lengths of `F` and `G` to $n/2 + 1$. `RealFFTI` is the inverse of `RealFFT`.

[1] If you are viewing the PDF version of this file and this formula is not displaying correctly turn *off* the Acrobat Reader option *Use Greek Text Below* under File/Preferences/General.

See also the section on fast trigonometric transforms [3.23].

3.23 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 [3.22] and require an even transform length, denoted by m in these notes. 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(U,V);           // inverse of DCT
DST(U,V);                   // U, V are ColumnVectors, length m+1
DST_inverse(U,V);           // inverse of DST
DCT_II(U,V);                // U, V are ColumnVectors, length m
DCT_II_inverse(U,V);        // inverse of DCT_II
DST_II(U,V);                // U, V are ColumnVectors, length m
DST_II_inverse(U,V);        // inverse of DST_II
```

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

Here are the formulae[2]:

DCT

$$v_k = u_0/2 + \sum_{j=1}^{m-1} u_j \cos(\pi jk/m) + (-1)^k u_m/2$$

for $k = 0,\ldots,m$, 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 $k = 1,\ldots,m-1$, where u_j and v_k are stored in `U(j+1)` and `V(k+1)`. u_0 and u_m are ignored and v_0 and v_m are set to zero.

DCT_II

$$v_k = \sum_{j=0}^{m-1} u_j \cos\{\pi(j+\tfrac{1}{2})k/m\}$$

for $k = 0,\ldots,m-1$, where u_j and v_k are stored in `U(j+1)` and `V(k+1)`.

[2] If you are viewing the PDF version of this file and these formulae are not displaying correctly turn *off* the Acrobat Reader option *Use Greek Text Below* under File/Preferences/General.

DST_II

$$v_k = \sum_{j=1}^{m} u_j \sin\{\pi(j-\tfrac{1}{2})k/m\}$$

for $k = 1,\ldots,m$, 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).

3.24 Interface to Numerical Recipes in C

This package can be used with the vectors and matrices defined in *Numerical Recipes in C*. You need to edit the routines in Numerical Recipes so that the elements are of the same type as used in this package. Eg replace float by double, vector by dvector and matrix by dmatrix, etc. You may need to edit the function definitions to use the version acceptable to your compiler (if you are using the first edition of NRIC). You may need to enclose the code from Numerical Recipes in `extern "C" { ... }`. You will also need to include the matrix and vector utility routines.

Then any vector in Numerical Recipes with subscripts starting from 1 in a function call can be accessed by a RowVector, ColumnVector or DiagonalMatrix in the present package. Similarly any matrix with subscripts starting from 1 can be accessed by an nricMatrix in the present package. The class nricMatrix is derived from Matrix and can be used in place of Matrix. In each case, if you wish to refer to a RowVector, ColumnVector, DiagonalMatrix or nricMatrix X in an function from Numerical Recipes, use `X.nric()` in the function call.

Numerical Recipes cannot change the dimensions of a matrix or vector. So matrices or vectors must be correctly dimensioned before a Numerical Recipes routine is called.

For example

```
   SymmetricMatrix B(44);
   .....                              // load values into B
   nricMatrix BX = B;                 // copy values to an nricMatrix
   DiagonalMatrix D(44);              // Matrices for output
   nricMatrix V(44,44);               //    correctly dimensioned
   int nrot;
   jacobi(BX.nric(),44,D.nric(),V.nric(),&nrot);
                                      // jacobi from NRIC
   cout << D;                         // print eigenvalues
```

3.25 Exceptions

Here is the class structure for exceptions:

```
Exception
  Logic_error
    ProgramException                  miscellaneous matrix error
    IndexException                    index out of bounds
    VectorException                   unable to convert matrix to vector
    NotSquareException                matrix is not square (invert, solve)
    SubMatrixDimensionException       out of bounds index of submatrix
    IncompatibleDimensionsException  (multiply, add etc)
    NotDefinedException              operation not defined (eg <)
    CannotBuildException             copying a matrix where copy is undefined
    InternalException                probably an error in newmat
  Runtime_error
    NPDException                     matrix not positive definite (Cholesky)
    ConvergenceException             no convergence (e-values, non-linear, sort)
    SingularException                matrix is singular (invert, solve)
    SolutionException                no convergence in solution routine
    OverflowException                floating point overflow
  Bad_alloc                          out of space (new fails)
```

I have attempted to mimic the exception class structure in the C++ standard library, by defining the Logic_error and Runtime_error classes.

If there is no catch statement or exceptions are disabled then my `Terminate()` function in `myexcept.h` is called. This prints out an error message, the dimensions and types of the matrices involved, the name of the routine detecting the exception, and any other information set by the Tracer [4] class. Also see the section on error messages [4] for additional notes on the messages generated by the exceptions.

You can also print this information by printing `Exception::what()`.

See the file `test_exc.cpp` as an example of catching an exception and printing the error message.

The 08 version of newmat defined a member function `void SetAction(int)` to help customise the action when an exception is called. This has been deleted in the 09 version. Now include an instruction such as `cout << Exception::what() << endl;` in the `Catch` or `CatchAll` block to determine the action.

The library includes the alternatives of using the built-in exceptions provided by a compiler, simulating exceptions, or disabling exceptions. See customising [2.3] for selecting the correct exception option.

The rest of this section describes my partial simulation of exceptions for compilers which do not support C++ exceptions. I use Carlos Vidal's article in the September 1992 *C Users Journal* as a starting point.

Newmat does a partial clean up of memory following throwing an exception - see the next section. However, the present version will leave a little heap memory unrecovered under some circumstances. I would not expect this to be a major problem, but it is something that needs to be sorted out.

The functions/macros I define are Try, Throw, Catch, CatchAll and CatchAndThrow. Try, Throw, Catch and CatchAll correspond to try, throw, catch and catch(...) in the C++ standard. A list of Catch clauses must be terminated by either CatchAll or CatchAndThrow but not both. Throw takes an Exception as an argument or takes no argument (for passing on an exception). I do not have a version of Throw for specifying which exceptions a function might throw. Catch takes an exception class name as an argument; CatchAll and CatchAndThrow don't have any arguments. Try, Catch and CatchAll must be followed by blocks enclosed in curly brackets.

I have added another macro ReThrow to mean a rethrow, Throw(). This was necessary to enable the package to be compatible with both my exception package and C++ exceptions.

If you want to throw an exception, use a statement like

```
Throw(Exception("Error message\n"));
```

It is important to have the exception declaration in the Throw statement, rather than as a separate statement.

All exception classes must be derived from the class, Exception, defined in newmat and can contain only static variables. See the examples in newmat if you want to define additional exceptions.

3.26 Cleanup after an exception

This section is about the simulated exceptions used in newmat. It is irrelevant if you are using the exceptions built into a compiler or have set the disable-exceptions option.

The simulated exception mechanisms in newmat are based on the C functions setjmp and longjmp. These functions do not call destructors so can lead to garbage being left on the heap. (I refer to memory allocated by *new* as heap memory). For example, when you call

```
Matrix A(20,30);
```

a small amount of space is used on the stack containing the row and column dimensions of the matrix and 600 doubles are allocated on the heap for the actual values of the matrix. At the end of the block in which A is declared, the destructor for A is called and the 600 doubles are freed. The locations on the stack are freed as part of the normal operations of the stack. If you leave the block using a longjmp command those 600 doubles will not be freed and will occupy space until the program terminates.

To overcome this problem newmat keeps a list of all the currently declared matrices and its exception mechanism will return heap memory when you do a Throw and Catch.

However it will not return heap memory from objects from other packages.

If you want the mechanism to work with another class you will have to do four things:

1: derive your class from class Janitor defined in myexcept.h;

2: define a function `void CleanUp()` in that class to return all heap memory;

3: include the following lines in the class definition

```
public:
   void* operator new(size_t size)
   { do_not_link=true; void* t = ::operator new(size); return t; }
   void operator delete(void* t) { ::operator delete(t); }
```

4: be sure to include a copy constructor in you class definition, that is, something like

```
X(const X&);
```

Note that the function `CleanUp()` does somewhat the same duties as the destructor. However `CleanUp()` has to do the *cleaning* for the class you are working with and also the classes it is derived from. So it will often be wrong to use exactly the same code for both `CleanUp()` and the destructor or to define your destructor as a call to `CleanUp()`.

3.27 Non-linear applications

Files solution.h, solution.cpp contain a class for solving for x in y = f(x) where x is a one-dimensional continuous monotonic function. This is not a matrix thing at all but is included because it is a useful thing and because it is a simpler version of the technique used in the non-linear least squares.

Files newmatnl.h, newmatnl.cpp contain a series of classes for non-linear least squares and maximum likelihood.

Documentation for both of these is in the definition files. Simple examples are in sl_ex.cpp, nl_ex.cpp and garch.cpp.

3.28 Standard template library

The standard template library (STL) is the set of *container templates* (vector, deque, list etc) defined by the C++ standards committee. Newmat is intended to be compatible with the STL in the sense that you can store matrices in the standard containers. I have defined == and inequality [3.6] operators to help make this possible. Probably there will have to be some other changes. My experiments with the Rogue Wave STL that comes with Borland C++ 5.0 showed that some things worked and some things unexpectedly didn't work. In particular, I couldn't get the list container to work. I don't know whose fault this is.

If you want to use the container classes with Newmat please note

- Don't use simulated exceptions.
- You can store only one type of matrix in a container. If you want to use a variety of types use the GenericMatrix type or store pointers to the matrices.
- The vector and deque container templates like to copy their elements. For the vector container this happens when you insert an element anywhere except at the end or when you append an element and the current vector storage overflows. Since Newmat does not have *copy-on-write* this could get very inefficient. (Later versions may have *copy-on-write* for the GenericMatrix type).
- You won't be able to sort the container or do anything that would call an inequality operator.

I doubt whether the STL container will be used often for matrices. So I don't think these limitations are very critical. If you think otherwise, please tell me.

3.29 Namespace

Namespace is a new facility in C++. Its purpose is to avoid name clashes between different libraries. I have included the namespace capability. Activate the line `#define use_namespace` in include.h. Then include either the statement

```
   using namespace NEWMAT;
```

at the beginning of any file that needs to access the newmat library or

```
   using namespace RBD_LIBRARIES;
```

at the beginning of any file that needs to access all my libraries.

This works correctly with Borland [2.4.2] C++ version 5.

Microsoft [2.4.5] Visual C++ version 5 works in my example and test files, but fails with apparently insignificant changes. If you `#include "newmatap.h"`, but no other newmat include file, then also `#include "newmatio.h"`.

My use of namespace does not work with Gnu g++ [2.4.3] version 2.7.2.

I have defined the following namespaces:

- RBD_COMMON for functions and classes used by most of my libraries
- NEWMAT for the newmat library
- RBD_LIBRARIES for all my libraries

4 Error messages

Most error messages are self-explanatory. The message gives the size of the matrices involved. Matrix types are referred to by the following codes:

`Matrix or vector`	`Rect`
`Symmetric matrix`	`Sym`
`Band matrix`	`Band`
`Symmetric band matrix`	`SmBnd`
`Lower triangular matrix`	`LT`
`Lower triangular band matrix`	`LwBnd`
`Upper triangular matrix`	`UT`
`Upper triangular band matrix`	`UpBnd`
`Diagonal matrix`	`Diag`
`Crout matrix (LU matrix)`	`Crout`
`Band LU matrix`	`BndLU`

Other codes should not occur.

See the section on exceptions [3.25] for more details on the structure of the exception classes.

I have defined a class Tracer that is intended to help locate the place where an error has occurred. At the beginning of a function I suggest you include a statement like

```
Tracer tr("name");
```

where name is the name of the function. This name will be printed as part of the error message, if an exception occurs in that function, or in a function called from that function. You can change the name as you proceed through a function with the ReName function

```
tr.ReName("new name");
```

if, for example, you want to track progress through the function.

5 Bugs

- Small memory leaks may occur when an exception is thrown and caught.
- My exception scheme may not be not properly linked in with the standard library exceptions. In particular, my scheme may fail to catch out-of-memory exceptions.

6 List of files

Documentation

README		readme file
NEWMAT	TXT	documentation file

Definition files

BOOLEAN	H	boolean class definition
CONTROLW	H	control word definition file
INCLUDE	H	details of include files and options
MYEXCEPT	H	general exception handler definitions
NEWMAT	H	main matrix class definition file
NEWMATAP	H	applications definition file
NEWMATIO	H	input/output definition file
NEWMATNL	H	non-linear optimisation definition file
NEWMATRC	H	row/column functions definition files
NEWMATRM	H	rectangular matrix access definition files
PRECISIO	H	numerical precision constants
SOLUTION	H	one dimensional solve definition file

Program files

BANDMAT	CPP	band matrix routines
CHOLESKY	CPP	Cholesky decomposition
EVALUE	CPP	eigenvalues and eigenvector calculation
FFT	CPP	fast Fourier, trig. transforms
HHOLDER	CPP	QR routines
JACOBI	CPP	eigenvalues by the Jacobi method
MYEXCEPT	CPP	general error and exception handler
NEWMAT1	CPP	type manipulation routines
NEWMAT2	CPP	row and column manipulation functions
NEWMAT3	CPP	row and column access functions
NEWMAT4	CPP	constructors, resize, utilities
NEWMAT5	CPP	transpose, evaluate, matrix functions
NEWMAT6	CPP	operators, element access
NEWMAT7	CPP	invert, solve, binary operations
NEWMAT8	CPP	LU decomposition, scalar functions
NEWMAT9	CPP	output routines
NEWMATEX	CPP	matrix exception handler
NEWMATNL	CPP	non-linear optimisation
NEWMATRM	CPP	rectangular matrix access functions
SORT	CPP	sorting functions
SOLUTION	CPP	one dimensional solve
SUBMAT	CPP	submatrix functions
SVD	CPP	singular value decomposition

Example files

EXAMPLE	CPP	example of use of package
EXAMPLE	TXT	output from example
SL_EX	CPP	example of OneDimSolve routine
SL_EX	TXT	output from example
NL_EX	CPP	example of non-linear least squares
NL_EX	TXT	output from example
GARCH	CPP	example of maximum-likelihood fit
GARCH	DAT	data file for garch.cpp
GARCH	TXT	output from example
TEST_EXC	CPP	demonstration exceptions
TEST_EXC	TXT	output from TEST_EXC.CPP

Test files

TMT*	CPP	test files
TMT	TXT	output from the test files

See the section on testing [2.7] for details of the test files.

Make files

GNU	MAK	make file for Gnu G++
CC	MAK	make file for AT&T, Sun and HPUX
MS_NT	MAK	make file for Microsoft Visual C++ 2 (windows NT)
MS	MAK	make file for Microsoft Visual C++ 1.51 (DOS)
BC	MAK	make file for Borland C++ 4.5,5 (large model)
BC32	MAK	make file for Borland C++ 4.5,5 (32 bit console model)
WATCOM	MAK	make file for Watcom 10
WATCO_NT	MAK	make file for Watcom 10 (windows NT)

7 Problem report form

Copy and paste this to your editor; fill it out and email to

robertd@netlink.co.nz (don't forget the *d* in robertd).

But first look in my web page http://webnz.com/robert/ to see if the bug has already been reported.

```
Version: ............... newmat09 (September 1997)
Your email address: ....
Today's date: ..........
Your machine: ..........
Operating system: ......
Compiler & version: ....
Compiler options
  (eg GUI or console)...
Describe the problem - attach examples if possible:
```

8 Notes on the design of the library

I describe some of the ideas behind this package, some of the decisions that I needed to make and give some details about the way it works. You don't need to read this part of the documentation in order to use the package.

It isn't obvious what is the best way of going about structuring a matrix package. I don't think you can figure this out with *thought* experiments. Different people have to try out different approaches. And someone else may have to figure out which is best. Or, more likely, the ultimate packages will lift some ideas from each of a variety of trial packages. So, I don't claim my package is an *ultimate* package, but simply a trial of a number of ideas. The following pages give some background on these ideas.

8.1 Safety, usability, efficiency

8.1.1 Some general comments

A library like *newmat* needs to balance *safety*, *usability* and *efficiency*.

By safety, I mean getting the right answer, and not causing crashes or damage to the computer system.

By usability, I mean being easy to learn and use, including not being too complicated, being intuitive, saving the users' time, being nice to use.

Efficiency means minimising the use of computer memory and time.

In the early days of computers the emphasis was on efficiency. But computer power gets cheaper and cheaper, halving in price every 18 months. On the other hand the unaided human brain is probably not a lot better than it was 100,000 years ago! So we should expect the balance to shift to put more emphasis on safety and usability and a little less on efficiency. So I don't mind if my programs are a little less efficient than programs written in pure C (or Fortran) if I gain substantially in safety and usability. But I would mind if they were a lot less efficient.

8.1.2 Type of use

Second reason for putting extra emphasis on safety and usability is the way I and, I suspect, most other users actually use *newmat*. Most completed programs are used only a few times. Some result is required for a client, paper or thesis. The program is developed and tested, the result is obtained, and the program archived. Of course bits of the program will be recycled for the next project. But it may be less usual for the same program to be run over and over again. So the cost, computer time + people time, is in the development time and often, much less in the actual time to run the final program. So good use of people time, especially during development is really important. This means you need highly usable libraries.

So if you are dealing with matrices, you want the good interface that I have tried to provide in *newmat*, and, of course, reliable methods underneath it.

Of course, efficiency is still important. We often want to run the biggest problem our computer will handle and often a little bigger. The C++ language almost lets us have both worlds. We can define a reasonably good interface, and get good efficiency in the use of the computer.

8.1.3 *Levels of access*

We can imagine the *black box* model of a *newmat*. Suppose the inside is hidden but can be accessed by the methods described in the reference [3] section. Then the interface is reasonably consistent and intuitive. Matrices can be accessed and manipulated in much the same way as doubles or ints in regular C. All accesses are checked. It is most unlikely that an incorrect index will crash the system. In general, users do not need to use pointers, so one shouldn't get pointers pointing into space. And, hopefully, you will get simpler code and so less errors.

There are some exceptions to this. In particular, the C-like subscripts [3.2] are not checked for validity. They give faster access but with a lower level of safety.

Then there is the Store() [3.8] function which takes you to the data array within a matrix. This takes you right inside the *black box*. But this is what you have to use if you are writing, for example, a new matrix factorisation, and require fast access to the data array. I have tried to write code to simplify access to the interior of a rectangular matrix, see file newmatrm.cpp, but I don't regard this as very successful, as yet, and have not included it in the documentation. Ideally we should have improved versions of this code for each of the major types of matrix. But, in reality, most of my matrix factorisations are written in what is basically the C language with very little C++.

So our *box* is not very *black*. You have a choice of how far you penetrate. On the outside you have a good level of safety, but in some cases efficiency is compromised a little. If you penetrate inside the *box* safety is reduced but you can get better efficiency.

8.1.4 *Some performance data*

This section looks at the performance on *newmat* for simple sums, comparing it with pure C code and with a somewhat unintelligent array program.

The following table lists the time (in seconds) for carrying out the operations `X=A+B;`, `X=A+B+C;`, `X=A+B+C+D;`, where `X,A,B,C,D` are of type ColumnVector, with a variety of programs. I am using Borland C++, version 5 in 32 bit console mode under windows NT 4.0 on a PC with a 150 mhz Pentium and 128 Mbytes of memory.

	length	**iters.**	**newmat**	**subs.**	**C**	**C-resz**	**array**
X=A+B							
	2000000	2	1.2	3.7	1.2	1.4	3.3
	200000	20	1.2	3.7	1.2	1.5	3.1
	20000	200	1.0	3.6	1.0	1.2	2.9
	2000	2000	1.0	3.6	0.9	0.9	2.2
	200	20000	0.8	3.0	0.4	0.5	1.9
	20	200000	5.5	2.9	0.4	0.9	2.8
	2	2000000	43.9	3.2	1.0	4.2	12.3

X=A+B+C						
2000000	2	2.5	4.6	1.6	2.1	32.5
200000	20	2.1	4.6	1.6	1.8	6.2
20000	200	1.8	4.5	1.5	1.8	5.6
2000	2000	1.8	4.4	1.3	1.3	3.9
200	20000	2.2	4.3	1.0	0.9	2.3
20	200000	8.5	4.3	1.0	1.2	3.0
2	2000000	62.5	3.9	1.0	4.4	17.7
X=A+B+C+D						
2000000	2	3.7	6.7	2.4	2.8	260.7
200000	20	3.7	6.7	2.4	2.5	9.2
20000	200	3.4	6.6	2.3	2.9	8.2
2000	2000	2.9	6.4	2.0	2.0	5.9
200	20000	2.5	5.5	1.0	1.3	4.3
20	200000	9.9	5.5	1.0	1.6	4.5
2	2000000	76.5	5.8	1.7	5.0	24.9

The first column gives the lengths of the arrays, the second the number of iterations and the remaining columns the total time required in seconds. If the only thing that consumed time was the double precision addition then the numbers within each block of the table would be the same.

The column labelled *newmat* is using the standard *newmat* add. In the next column the calculation is using the usual C style *for* loop and accessing the elements using *newmat* subscripts such as `A(i)`. The column labelled *C* uses the usual C method: `while (j--) *x++ = *a++ + *b++;`. The following column also includes an `X.ReSize()` in the outer loop to correspond to the reassignment of memory that *newmat* would do. The final column is the time taken by a simple array package that makes no attempt to eliminate unnecessary copying or to recycle temporary memory but does have array definitions of the basic operators. It does, however, do its sums in blocks of 4 and copies in blocks of 8 in the same way that *newmat* does.

Here are my conclusions.

- *newmat* does very badly for length 2 and doesn't do very well for length 20. There is some particularly tortuous code in *newmat* for determining which sum algorithm to use and I am sure this could be improved. However the *array* program is also having difficulty with length 2 so it is unlikely that the problem could be completely eliminated.
- The *array* program is running into problems for length 2,000,000. This is because RAM memory is being exhausted.
- For arrays of length 2000 or longer *newmat* is doing about as well as C and slightly better than C with resize in the `X=A+B` table. For the other two tables it is slower, but not dramatically so.
- Addition using the *newmat* subscripts, while considerably slower than the others, is still surprisingly good.
- The *array* program is more than 2 times slower than *newmat* for lengths 2000 or higher.

In summary: for the situation considered here, *newmat* is doing very well for large ColumnVectors, even for sums with several terms, but not so well for shorter ColumnVectors.

8.2 Matrix vs array library

The *newmat* library is for the manipulation of matrices, including the standard operations such as multiplication as understood by numerical analysts, engineers and mathematicians.

A matrix is a two dimensional array of numbers. However, very special operations such as matrix multiplication are defined specifically for matrices. This means that a *matrix* library, as I understand the term, is different from a general *array* library.

I see a matrix package as providing the following

1. Evaluation of matrix expressions in a form familiar to scientists and engineers. For example `A = B * (C + D.t());`. In particular `*` means matrix multiply.
2. Access to the elements of a matrix;
3. Access to submatrices;
4. Common elementary matrix functions such as determinant and trace;
5. A platform for developing advanced matrix functions such as eigenvalue analysis;
6. Good efficiency and storage management;
7. Graceful exit from errors.

It may also provide

8. A variety of types of elements (eg real and complex);
9. A variety of types of matrices (eg rectangular and symmetric).

In contrast an array package should provide

1'. Arrays can be copied with a single instruction; may have some arithmetic operations for elementwise combination of arrays, say +, -, *, /, it may provide matrix multiplication as a function;
2'. High speed access to elements directly and perhaps with iterators;
3'. Access to subarrays and rows (and columns?);
6'. Good efficiency and storage management;
7'. Graceful exit from errors;
8'. A wide variety of types of elements (eg int, float, double, string etc);
9'. One, two and three dimensional arrays, at least, with starting points of the indices set by user.

It may be possible to amalgamate these two sets of requirements to some extent. However *newmat* is definitely oriented towards the first set.

8.3 Design questions

Even within the bounds set by the requirements of a matrix library there is a substantial opportunity for variation between what different matrix packages might provide. It is not possible to build a matrix package that will meet everyone's requirements. In many cases if you put in one facility, you impose overheads on everyone using the package. This both in storage required for the program and in efficiency. Likewise a package that is optimised towards handling large matrices is likely to become less efficient for very small matrices where the administration time for the matrix may become significant compared with the time to carry out the operations. It is better to provide a variety of packages (hopefully compatible) so that most users can find one that meets their requirements. This package is intended to be one of these packages; but not all of them.

Since my background is in statistical methods, this package is oriented towards the kinds things you need for statistical analyses.

Now looking at some specific questions.

8.3.1 What size of matrices?

A matrix library may target small matrices (say 3 x 3), or medium sized matrices, or very large matrices.

A library targeting very small matrices will seek to minimise administration. A library for medium sized or very large matrices can spend more time on administration in order to conserve space or optimise the evaluation of expressions. A library for very large matrices will need to pay special attention to storage and numerical properties. This library is designed for medium sized matrices. This means it is worth introducing some optimisations, but I don't have to worry about setting up some form of virtual memory.

8.3.2 Which matrix types?

As well as the usual rectangular matrices, matrices occurring repeatedly in numerical calculations are upper and lower triangular matrices, symmetric matrices and diagonal matrices. This is particularly the case in calculations involving least squares and eigenvalue calculations. So as a first stage these were the types I decided to include.

It is also necessary to have types row vector and column vector. In a *matrix* package, in contrast to an *array* package, it is necessary to have both these types since they behave differently in matrix expressions. The vector types, can be derived for the rectangular matrix type, so having them does not greatly increase the complexity of the package.

The problem with having several matrix types is the number of versions of the binary operators one needs. If one has 5 distinct matrix types then a simple library will need 25 versions of each of the binary operators. In fact, we can evade this problem, but at the cost of some complexity.

8.3.3 What element types?

Ideally we would allow element types double, float, complex and int, at least. It might be reasonably easy, using templates or equivalent, to provide a library which could handle a variety of element types. However, as soon as one starts implementing the binary operators between matrices with different element types, again one gets an explosion in the number of operations one needs to consider. At the present time the compilers I deal with are not up to handling this problem with templates. (Of course, when I started writing *newmat* there were no templates). But even when the compilers do meet the specifications of the draft standard, writing a matrix package that allows for a variety of element types using the template mechanism is going to be very difficult. I am inclined to use templates in an *array* library but not in a *matrix* library.

Hence I decided to implement only one element type. But the user can decide whether this is float or double. The package assumes elements are of type Real. The user typedefs Real to float or double.

It might also be worth including symmetric and triangular matrices with extra precision elements (double or long double) to be used for storage only and with a minimum of operations defined. These would be used for accumulating the results of sums of squares and product matrices or multistage QR triangularisations.

8.3.4 Allow matrix expressions

I want to be able to write matrix expressions the way I would on paper. So if I want to multiply two matrices and then add the transpose of a third one I can write something like `X = A * B + C.t();`. I want this expression to be evaluated with close to the same efficiency as a hand-coded version. This is not so much of a problem with expressions including a multiply since the multiply will dominate the time. However, it is not so easy to achieve with expressions with just + and `-`.

A second requirement is that temporary matrices generated during the evaluation of an expression are destroyed as quickly as possible.

A desirable feature is that a certain amount of *intelligence* be displayed in the evaluation of an expression. For example, in the expression `X = A.i() * B;` where `i()` denotes inverse, it would be desirable if the inverse wasn't explicitly calculated.

8.3.5 Naming convention

How are classes and public member functions to be named? As a general rule I have spelt identifiers out in full with individual words being capitalised. For example UpperTriangularMatrix. If you don't like this you can #define or typedef shorter names. This convention means you can select an abbreviation scheme that makes sense to you.

Exceptions to the general rule are the functions for transpose and inverse. To make matrix expressions more like the corresponding mathematical formulae, I have used the single letter abbreviations, `t()` and `i()`.

8.3.6 Row and column index ranges

In mathematical work matrix subscripts usually start at one. In C, array subscripts start at zero. In Fortran, they start at one. Possibilities for this package were to make them start at 0 or 1 or be arbitrary.

Alternatively one could specify an *index set* for indexing the rows and columns of a matrix. One would be able to add or multiply matrices only if the appropriate row and column index sets were identical.

In fact, I adopted the simpler convention of making the rows and columns of a matrix be indexed by an integer starting at one, following the traditional convention. In an earlier version of the package I had them starting at zero, but even I was getting mixed up when trying to use this earlier package. So I reverted to the more usual notation and started at 1.

8.3.7 Element access - method and checking

We want to be able to use the notation `A(i,j)` to specify the `(i,j)`-th element of a matrix. This is the way mathematicians expect to address the elements of matrices. I consider the notation `A[i][j]` totally alien. However I include this as an option to help people converting from C.

There are two ways of working out the address of `A(i,j)`. One is using a *dope* vector which contains the first address of each row. Alternatively you can calculate the address using the formula appropriate for the structure of `A`. I use this second approach. It is probably slower, but saves worrying about an extra bit of storage.

The other question is whether to check for `i` and `j` being in range. I do carry out this check following years of experience with both systems that do and systems that don't do this check. I would hope that the routines I supply with this package will reduce your need to access elements of matrices so speed of access is not a high priority.

8.3.8 Use iterators

Iterators are an alternative way of providing fast access to the elements of an array or matrix when they are to be accessed sequentially. They need to be customised for each type of matrix. I have not implemented iterators in this package, although some iterator like functions are used internally for some row and column functions.

8.4 Data storage

8.4.1 Structure

Each matrix object contains the basic information such as the number of rows and columns and a status variable plus a pointer to the data array which is on the heap.

8.4.2 One block or several

The elements of the matrix are stored as a single array. Alternatives would have been to store each row as a separate array or a set of adjacent rows as a separate array. The present solution simplifies the program but limits the size of matrices in 16 bit PCs that have a 64k byte limit on the size of arrays (I don't use the `huge` keyword). The large arrays may also cause problems for memory management in smaller machines.

8.4.3 By row or by column or other

In Fortran two dimensional arrays are stored by column. In most other systems they are stored by row. I have followed this later convention. This makes it easier to interface with other packages written in C but harder to interface with those written in Fortran. This may have been a wrong decision. Most work on the efficient manipulation of large matrices is being done in Fortran. It would have been easier to use this work if I had adopted the Fortran convention.

An alternative would be to store the elements by mid-sized rectangular blocks. This might impose less strain on memory management when one needs to access both rows and columns.

8.4.4 Storage of symmetric matrices

Symmetric matrices are stored as lower triangular matrices. The decision was pretty arbitrary, but it does slightly simplify the Cholesky decomposition program.

8.5 Memory management - reference counting or status variable?

Consider the instruction

```
   X = A + B + C;
```

To evaluate this a simple program will add `A` to `B` putting the total in a temporary `T1`. Then it will add `T1` to `C` creating another temporary `T2` which will be copied into `X`. `T1` and `T2` will sit around till the end of the execution of the statement and perhaps of the block. It would be faster if the program recognised that `T1` was temporary and stored the sum of `T1` and `C` back into `T1` instead of creating `T2` and then avoided the final copy by just assigning the contents of `T1` to `X` rather than copying. In this case there will be no temporaries requiring deletion. (More precisely there will be a header to be deleted but no contents).

For an instruction like

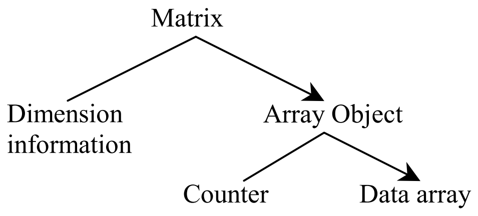
```
   X = (A * B) + (C * D);
```

we can't easily avoid one temporary being left over, so we would like this temporary deleted as quickly as possible.

I provide the functionality for doing all this by attaching a status variable to each matrix. This indicates if the matrix is temporary so that its memory is available for recycling or deleting. Any matrix operation checks the status variables of the matrices it is working with and recycles or deletes any temporary memory.

An alternative or additional approach would be to use *reference counting and delayed copying* - also known as *copy on write*. If a program requests a matrix to be copied, the copy is delayed until an instruction is executed which modifies the memory of either the original matrix or the copy. If the original matrix is deleted before either matrix is modified, in effect, the values of the original matrix are transferred to the copy without any actual copying taking place. This solves the difficult problem of returning an object from a function without copying and saves the unnecessary copying in the previous examples.

There are downsides to the delayed copying approach. Typically, for delayed copying one uses a structure like the following:



where the arrows denote a pointer to a data structure. If one wants to access the *Data array* one will need to track through two pointers. If one is going to write, one will have to check whether one needs to copy first. This is not important when one is going to access the whole array, say, for an add operation. But if one wants to access just a single element, then it imposes a significant additional overhead on that operation. Any subscript operation would need to check whether an update was required - even read since it is hard for the compiler to tell whether a subscript access is a read or write.

Some matrix libraries don't bother to do this. So if you write `A = B;` and then modify an element of one of `A` or `B`, then the same element of the other is also modified. I don't think this is acceptable behaviour.

Delayed copy does not provide the additional functionality of my approach but I suppose it would be possible to have both delayed copy and tagging temporaries.

My approach does not automatically avoid all copying. In particular, you need use a special technique to return a matrix from a function without copying.

8.6 Evaluation of expressions - lazy evaluation

Consider the instruction

```
X = B - X;
```

A simple program will subtract `X` from `B`, store the result in a temporary `T1` and copy `T1` into `X`. It would be faster if the program recognised that the result could be stored directly into `X`. This would happen automatically if the program could look at the instruction first and mark `X` as temporary.

C programmers would expect to avoid the same problem with

```
X = X - B;
```

by using an operator `-=`

```
X -= B;
```

However this is an unnatural notation for non C users and it may be nicer to write

```
X = X - B;
```

and know that the program will carry out the simplification.

Another example where this intelligent analysis of an instruction is helpful is in

```
X = A.i() * B;
```

where `i()` denotes inverse. Numerical analysts know it is inefficient to evaluate this expression by carrying out the inverse operation and then the multiply. Yet it is a convenient way of writing the instruction. It would be helpful if the program recognised this expression and carried out the more appropriate approach.

I regard this interpretation of `A.i() * B` as just providing a convenient notation. The objective is not to correct the errors of people who are unaware of the inefficiency of `A.i() * B` if interpreted literally.

There is a third reason for the two-stage evaluation of expressions and this is probably the most important one. In C++ it is quite hard to return an expression from a function such as (`*`, `+` etc) without a copy. This is particularly the case when an assignment (`=`) is involved. The mechanism described here provides one way for avoiding this in matrix expressions.

To carry out this *intelligent* analysis of an instruction matrix expressions are evaluated in two stages. In the first stage a tree representation of the expression is formed. For example `(A+B)*C` is represented by a tree

```
    *
   / \
  +   C
 / \
A   B
```

Rather than adding `A` and `B` the `+` operator yields an object of a class *AddedMatrix* which is just a pair of pointers to `A` and `B`. Then the `*` operator yields a *MultipliedMatrix* which is a pair of pointers to the *AddedMatrix* and `C`. The tree is examined for any simplifications and then evaluated recursively.

Further possibilities not yet included are to recognise `A.t()*A` and `A.t()+A` as symmetric or to improve the efficiency of evaluation of expressions like `A+B+C`, `A*B*C`, `A*B.t()` (`t()` denotes transpose).

One of the disadvantages of the two-stage approach is that the types of matrix expressions are determined at run-time. So the compiler will not detect errors of the type

```
Matrix M;
DiagonalMatrix D;
....;
D = M;
```

We don't allow conversions using `=` when information would be lost. Such errors will be detected when the statement is executed.

8.7 How to overcome an explosion in number of operations

The package attempts to solve the problem of the large number of versions of the binary operations required when one has a variety of types.

With *n* types of matrices the binary operations will each require *n*-squared separate algorithms. Some reduction in the number may be possible by carrying out conversions. However, the situation rapidly becomes impossible with more than 4 or 5 types. Doug Lea told me that it was possible to avoid this problem. I don't know what his solution is. Here's mine.

Each matrix type includes routines for extracting individual rows or columns. I assume a row or column consists of a sequence of zeros, a sequence of stored values and then another sequence of zeros. Only a single algorithm is then required for each binary operation. The rows can be located very quickly since most of the matrices are stored row by row. Columns must be copied and so the access is somewhat slower. As far as possible my algorithms access the matrices by row.

There is another approach. Each of the matrix types defined in this package can be set up so both rows and columns have their elements at equal intervals provided we are prepared to store the rows and columns in up to three chunks. With such an approach one could write a single *generic* algorithm for each of multiply and add. This would be a reasonable alternative to my approach.

I provide several algorithms for operations like `+` . If one is adding two matrices of the same type then there is no need to access the individual rows or columns and a faster general algorithm is appropriate.

Generally the method works well. However symmetric matrices are not always handled very efficiently (yet) since complete rows are not stored explicitly.

The original version of the package did not use this access by row or column method and provided the multitude of algorithms for the combination of different matrix types. The code file length turned out to be just a little longer than the present one when providing the same facilities with 5 distinct types of matrices. It would have been very difficult to increase the number of matrix types in the original version. Apparently 4 to 5 types is about the break-even point for switching to the approach adopted in the present package.

However it must also be admitted that there is a substantial overhead in the approach adopted in the present package for small matrices. The test program developed for the original version of the package takes 30 to 50% longer to run with the current version (though there may be some other reasons for this). This is for matrices in the range 6x6 to 10x10.

To try to improve the situation a little I do provide an ordinary matrix multiplication routine for the case when all the matrices involved are rectangular.

8.8 Destruction of temporaries

Versions before version 5 of newmat did not work correctly with Gnu C++ (version 5 or earlier). This was because the tree structure used to represent a matrix expression was set up on the stack. This was fine for AT&T, Borland and Zortech C++.

However early version Gnu C++ destroys temporary structures as soon as the function that accesses them finishes. The other compilers wait until the end of the current expression or current block. To overcome this problem, there is now an option to store the temporaries forming the tree structure on the heap (created with new) and to delete them explicitly. Activate the definition of TEMPS_DESTROYED_QUICKLY to set this option.

In fact, I suggest this be the default option as, with it, the package uses less storage and runs faster. There still exist situations Gnu C++ will go wrong. These include statements like

```
A = X * Matrix(P * Q);

Real r = (A*B)(3,4);
```

Neither of these kinds of statements will occur often in practice.

Now that the C++ standards committee has said that temporary structures should not be destroyed before a statement finishes, my policy needs to be re-evaluated. Probably, I'll return to using the stack, because of the difficulty of managing exceptions with the heap version.

8.9 A calculus of matrix types

The program needs to be able to work out the class of the result of a matrix expression. This is to check that a conversion is legal or to determine the class of an intermediate result. To assist with this, a class MatrixType is defined. Operators +, -, *, >= are defined to calculate the types of the results of expressions or to check that conversions are legal.

Early versions of *newmat* stored the types of the results of operations in a table. So, for example, if you multiplied an UpperTriangularMatrix by a LowerTriangularMatrix, *newmat* would look up the table and see that the result was of type Matrix. With this approach the exploding [8.7] number of operations problem recurred although not as seriously as when code had to be written for each pair of types. But there was always the suspicion that somewhere, there was an error in one of those 9x9 tables, that would be very hard to find. And the problem would get worse as additional matrix types or operators were included.

The present version of *newmat* solves the problem by assigning *attributes* such as *diagonal* or *band* or *upper triangular* to each matrix type. Which attributes a matrix type has, is stored as bits in an integer. As an example, the DiagonalMatrix type has the bits corresponding to *diagonal*, *symmetric* and *band* equal to 1. By looking at the attributes of each of the operands of a binary operator, the program can work out the attributes of the result of the operation with simple bitwise operations. Hence it can deduce an appropriate type. The *symmetric* attribute is a minor problem because *symmetric* * *symmetric* does not yield *symmetric* unless both operands are *diagonal*. But otherwise very simple code can be used to deduce the attributes of the result of a binary operation.

Tables of the types resulting from the binary operators are output at the beginning of the test [2.7] program.

8.10 Error handling

The library now does have a moderately graceful exit from errors. One can use either the simulated exceptions or the compiler supported exceptions. When newmat08 was released (in 1995), compiler exception handling in the compilers I had access to was unreliable. I recommended you used my simulated exceptions. Now (in 1997) compiler supported exceptions do seem to work on a variety of compilers - but not all compilers.

The approach in the present library, attempting to simulate C++ exceptions, is not completely satisfactory, but seems a good interim solution for those who cannot use compiler supported exceptions. People who don't want exceptions in any shape or form, can set the option to exit the program if an exception is thrown.

The exception mechanism cannot clean-up objects explicitly created by new. This must be explicitly carried out by the package writer or the package user. I have not yet done this completely with the present package so occasionally a little garbage may be left behind after an exception. I don't think this is a big problem, but it is one that needs fixing.

8.11 Sparse matrices

The library does not support sparse matrices.

For sparse matrices there is going to be some kind of structure vector. It is going to have to be calculated for the results of expressions in much the same way that types are calculated. In addition, a whole new set of row and column operations would have to be written.

Sparse matrices are important for people solving large sets of differential equations as well as being important for statistical and operational research applications.

But there are packages being developed specifically for sparse matrices and these might present the best approach, at least where sparse matrices are the main interest.

8.12 Complex matrices

The package does not yet support matrices with complex elements. There are at least two approaches to including these. One is to have matrices with complex elements.

This probably means making new versions of the basic row and column operations for Real`*`Complex, Complex`*`Complex, Complex`*`Real and similarly for + and `-`. This would be OK, except that if I also want to do this for sparse matrices, then when you put these together, the whole thing will get out of hand.

The alternative is to represent a complex matrix by a pair of real matrices. One probably needs another level of decoding expressions but I think it might still be simpler than the first approach. But there is going to be a problem with accessing elements and it does not seem possible to solve this in an entirely satisfactory way.

Complex matrices are used extensively by electrical engineers and physicists and really should be fully supported in a comprehensive package.

You can simulate most complex operations by representing `Z = X + iY` by

```
/  X   Y \
\ -Y'  X /
```

Most matrix operations will simulate the corresponding complex operation, when applied to this matrix. But, of course, this matrix is twice as big as you would need with a genuine complex matrix library.

9 Index