Documentation for newmat11, a matrix library in C++

next - skip - up - start
return to onlinedocumentation page

Copyright (C) 2008: R B Davies

20 November, 2008 .

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

There is additional support material on my web site.

Navigation: This page is arranged in sections, sub-sections and sub-sub-sections; four cross-references are given at the top of these. *Next* takes you throughthe sections, sub-sections and sub-sub-sections in order. *Skip* goes to the next section, sub-section or sub-sub-section at the same level in the hierarchy as the section, sub-section or sub-sub-section that you are currently reading. *Up* takes you up one level in the hierarchy and *start* gets you back here.

Please read the sections on customising and compilers before attempting to compile *newmat*.

1. Introduction

next - skip - up - start

1.1 Conditions of use

next - skip - up - start

I place no restrictions on the use of *newmat* except that I take no liability for any problems that may arise from its use, distribution or other dealings with it.

You can use it in your commercial projects (as well as your non-commercial projects).

You can make and distribute modified or merged versions. You can include parts of it in your own software.

If you distribute modified or merged versions, please make it clear which parts are mine and which parts are modified.

For a substantially modified version, simply note that it is, in part, derived from my software. A comment in the code will be sufficient.

The software is provided *as is*, without warranty of any kind.

Please understand that there may still be bugs and errors. Use at your own risk. I (Robert Davies) take no responsibility for any errors or omissions in this package or for any misfortune that may befall you or others as a result of your use, distribution or other dealings with it.

Please report bugs to me at **robert at statsresearch.co.nz** [replace **at** by you-know-what-character in the email address] .

When reporting a bug please tell me which C++ compiler you are using, and what version. Also give me details of your computer. And tell me which version of *newmat* (e.g. newmat03 or newmat04) you are using and its date. 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.

Please do report bugs to me.

1.2 General description

next - skip - up - start

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	rectangularmatrix
SquareMatrix	square matrix
nricMatrix	for use with *Numerical Recipes in C* programs
UpperTriangularMatrix	
LowerTriangularMatrix	
DiagonalMatrix	
SymmetricMatrix	
BandMatrix	
UpperBandMatrix	upper triangular band matrix
LowerBandMatrix	lower triangular band matrix
SymmetricBandMatrix	
RowVector	derived from Matrix
ColumnVector	derived from Matrix
IdentityMatrix	diagonalmatrix , elements have same value

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

The package includes the operations *, +, -, Kronecker product, Schur product, concatenation, inverse, transpose, conversion between types, submatrix, determinant, Cholesky decomposition, QR decomposition, 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 10 x 10 to the maximum size your machine will accommodate in a single array. 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 versions of the package on variety of compilers and platforms including Borland, Gnu, Microsoft and Sun. For more details see the section on compilers.

1.3 Is this the library for you?

next - skip - up - start

Do you

- understand * to mean matrix multiplyand not element by element multiply
- need matrix operators such as * and + defined as operators so you can write things like `X = A * (B + C);`
- need a variety of types of matrices (but not sparse)
- need only one element type (float or double)
- work with matrices in the range 10 x 10 up to what can be stored in memory
- tolerate a moderatelylarge but not huge package
- need high qualitybut not necessarily the latest numerical methods.

Then *newmat* may be the right matrix library for you.

1.4 Other matrix libraries

next - skip - up - start

For details of other C++ matrix libraries look at http://www.robertnz.net/cpp_site.html. Look at the section *lists of libraries* which gives the locations of several very comprehensive lists of matrix and other C++ libraries and at the section *source code*. Or just search on Google.

1.5 Where to find this library

next - skip - up - start

- http://www.robertnz.net

1.6 How to contact the author

next - skip - up - start


```
Robert Davies
16 Gloucester Street
Wilton
Wellington
New Zealand

Internet: robert at statsresearch.co.nz
```


[replace **at** by you-know-what.]

1.7 References

next - skip - up - start

- The matrix LU decomposition is from Golub, G.H. & VanLoan, C.F. (1996), *Matrix Computations*, publishedby Johns Hopkins University Press.

- Part of the matrix inverse/solve routine is adaptedfrom Press, Flannery, Teukolsky, Vetterling (1988), *Numerical Recipes in C*, publishedby the Cambridge University Press.
- Many of the advancedmatrix routines are adaptedfrom routines in Wilkinsonand Reinsch (1971), *Handbookfor AutomaticComputation , Vol II, Linear Algebra* publishedby Springer Verlag.
- The fast Fourier transform is adaptedfrom Carl de Boor (1980), *Siam J Sci Stat Comput*, pp173-8 and the fast trigonometrictransforms from Charles Van Loan (1992) in *Computational frameworks for the fast Fourier transform* publishedby SIAM.
- The sort functionis derived from Sedgewick, Robert (1992), *Algorithms in C++* publishedby 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

next - skip - up - start

2.1 Overview

next - skip - up - start

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 to get the complete package. Ideally you should store the resulting object files as a library. The tmt*.cpp files are used for testing, example.cpp is an example and sl_ex.cpp, nl_ex.cpp and garch.cpp are examples of the non-linear solve and optimisation routines. A demonstration and test of the exception mechanism is in test_exc.cpp. The files nm_ex1.cpp, nm_ex2.cpp and nm_ex3.cpp contain more simple examples.

I include a number of *make* files for compiling the example and the test package. See the section on make files for details. Alternatively, 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 on older operating systems or compilers.

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

include.h	to access just the compiler options
newmat.h	to access just the main matrix library (includesinclude .h)
newmatap.h	to access the advancedmatrix routines such as Choleskydecomposition , QR triangularisation etc (includesnewmat .h)
newmatio.h	to access the output routines (includesnewmat .h) You can use this only with compilers that support the standard input/outputroutines including manipulators(all recent compilers)
newmatnl.h	to access the non-linear optimisation routines (includesnewmat .h)

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

2.2 Make files

next - skip - up - start

I have included *make* files for compiling my test and example programs for various versions of CC, Borland, Microsoft, Open Watcom, Intel and Gnu compilers. You can generate make files for a number of other compilers with my genmake utility. *Make* files provide a way of compiling your programs without using the IDE that comes with PC compilers. See the files section for details.

PC

I include make files for various versions of Borland, Microsoft and Intel compilers. With the Borland compiler you will need to edit it to show where you have stored your Borland compiler. For make files for other compilers use my genmake utility. To compile my test and example programs use Borland 5.5 (Builder 5) use

```
make -f nm_b55.mak
```

or with Borland 5.6 (Builder 6) use

```
make -f nm_b56.mak
```

or with Microsoft VC++ 6 or 7 use

```
nmake -f nm_m6.mak
```

There are some more notes in the genmake documentation about using these make files.

Unix

The *make* file for the Unix CC 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* file for Gnu GCC on Unix systems is for use with `gmake` rather than `make`. I assume your compiler recognises the .cpp extension. Ordinary `make` works with it on the Sun but did not the Silicon Graphics or HP machines when I had access to them, many years ago. On Linux use `make`.

My make file for the CC compilers works with the ordinary make.

To compile everything with the CC compiler use

```
make -f nm_cc.mak
```

or for the gnu compiler use

```
make -f nm_gnu.mak
```

There is a line in the make file for CC `rm -f $*.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.

2.3 Customising

next - skip - up - start

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, 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. I now set *compiler supported exceptions* as the default. 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.

Activate the appropriate statement to make the element type *float* or *double*. I suggest you leave it at *double*.

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. Note that this does *not* change the starting point for indices when you are using round brackets for accessing elements or selecting submatrices. It does enable you to use C style square brackets. This also activates additional constructors for Matrix, ColumnVector and RowVector to simplify use with *Numerical Recipes in C++*.

Activate `#define use_namespace` if you want to use namespaces. Do this only if you are sure your compiler supports namespaces. If you do turn this option on, be prepared to turn it off again if the compiler reports inaccessible variables or the linker reports missing links.

Activate `#define _STANDARD_` to use the standard names for the included files and to find the floating point precision data using the floating point standard. This will work with most recent compilers and is done automatically for Borland, Gnu, Intel and Microsoft compilers.

If you haven't defined `_STANDARD_` and are using a compiler that *include.h* does not recognise and you want to pick up the floating point precision data from *float.h* then activate `#define use_float_h`. Otherwise the floating point precision data will be accessed from *values.h*. You may need to do this with computers from Digital, in particular.

There is a line

```
//#define set_unix_options
```

You can activate this if you are using a Linux or Unix system. It is not used by *Newmat* but is used by some of my other programs.

2.4 Compilers

next - skip - up - start

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 newest PC versions, I use a Pentium 4 computer running windows XP or Linux (Red Hat workstation version). The Unix versions are on a Sun Sparc station. Thanks to Victoria University for access to the Sparc.

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++ 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

next - skip - up - start

The AT&T compiler used to be available on a wide variety of Unix workstations. I don't know if anyone still uses it. However the AT&T options are the default if your compiler is not recognised.

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

next - skip - up - start

Newer compilers

Borland Builder version 8. My tests here have been compiling from a make file using *nm_b58.mak* and making a console program. This works fine except that my values of *LnMinimum* and *LnMaximum* in *precisio.h* are not correct.

Borland Builder version 6: My tests have been on the *personal* version. See the notes for version 5. If you are compiling with a make file you can use *nm_b56.mak* as a model. You can set the *newmat* options to use namespace and the standard library. If you are compiling a GUI program you may need to comment out the line defining *TypeDefException* in *include.h.* I don't believe exceptions work completely correctly in either version 5 or version 6. However, this does not seem to be a problem with my use of them in *newmat*.

Borland Builder version 5: This works fine in console mode and no special editing of the source codes is required. I haven't tested it in GUI mode. You can set the *newmat* options to use namespace and the standard library. **You should turn *off* the Borland option to use pre-compiled headers.** There are notes on compiling with the IDE on my website. Alternatively you can use the *nm_b55.mak* make file.

Borland Builder version 4: I have successfully used this on older versions of newmat using the console wizard (menu item file/new - select new tab). Use compiler exceptions. Suppose you are compiling my test program *tmt*. Rename my *main()* function in *tmt.cpp* to *my_main()*. Rename *tmt.cpp* to *tmt_main.cpp*. Borland will generate a new file *tmt.cpp* containing their *main()* function. Put the line `int my_main();` above this function and put `return my_main();` into the body of *main()*.

Borland compiler version 5.5: this is the free C++ compiler available from Borland's web site. I suggest you use the compiler supported exceptions and turn on *standard* in include.h. You can use the make file *nm_b55.mak* after editing to correct the file locations for your system.

Older compilers

Borland C++ 5.02:

I am not longer checking compatibility with this compiler.

Use the large or 32 bit flat model. If you are not debugging, turn off the options that collect debugging information. Use my simulated exceptions.

When running my test program 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.

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 will need to break the tmt set of test files into several parts to get the program to fit into your computer and run without stack overflow.

You can generate make files for versions 5 with my genmake utility.

Borland C++ 3 and 4.

The program used to compile in version 3.1 if you enable the *simulated booleans* - comment *out* the line `#define bool_LIB 0` in *include.h* and use the *simulated exceptions*. I haven't checked the latest versions of Newmat. The main test program is too large to run unless you break it up into several parts. I haven't tried it under version 4.

2.4.3 Gnu G++

next - skip - up - start

Gnu G++ 3, 4 (Linux), 3 (Sun): These work OK. If you are using a much earlier version see if you can upgrade. It used to work with 2.95 and 2.96 but I don't have access to these now. You can't use *standard* with the 2.9X versions. The namespace option worked with 2.96 on Linux but not with 2.95 on the Sun. Standard is automatically turned on with the 3.X.

This version of Newmat is not compatible with versions 2.6 or earlier.

2.4.4 HP-UX

next - skip - up - start

HP 9000 series HP-UX. I no longer have access to this compiler. Newmat09 worked without problems with the simulated exceptions; haven't tried the built-in exceptions.

With recent versions of the compiler you may get warning messages like `Unsafe cast between pointers/references to incomplete classes`. At present, I think these can be ignored.

2.4.5 Intel

next - skip - up - start

Newmat works correctly with the Intel 10 C++ compiler for Windows and Linux. (Not tested for the other versions). Standard is automatically turned on for both the Linux versions and Windows versions. If this causes a problem for the version you are using you can find the lines in *include.h* that control this and comment them out. Note that the Intel compiler for Linux is *free* for non-commercial use. (One of the versions of 8.1 gave a warning message every time I had something like `Real x; ... if (x==0.0) ...` , which was often. This is now seems to be fixed.)

2.4.6 Microsoft

next - skip - up - start

Newer versions

See my web site for instructions how to work Microsoft's IDE.

Microsoft Visual C++ 7, 7.1, 8: These work OK. All my tests have been in console mode. You can turn on my namespace option. Standard is turned on by default for these versions.

Microsoft Visual C++ 6: **Get the latest service pack**. I have tried this in console mode and it seems to work satisfactorily . Use thecompiler supported exceptions. You may be able to use the namespace and standard options. If you want to work under MFC you may need to `#include "stdafx.h"` at the beginning of each .cpp file (or turn off precompiled headers).

Microsoft Visual C++ 5: I have tried this in console mode on previous versions of Newmat. It seems to work satisfactorily . There maybe a problem with namespace (fixed by Service Pack 3?). **Turn optimisation off**. Use the compiler supported exceptions. If you want to work under MFC you may need to `#include "stdafx.h"` at the beginning of each .cpp file (or turn off precompiled headers).

Older versions

I doubt whether these will work.

2.4.7 Sun

next - skip - up - start

Sun C++ (version = ?): This seems to work fine with compiler supported exceptions. **Sun C++ (version 5):** There was a problem with exceptions. If you use my simulated exceptions the non-linear optimisation programs hang. If you use the compiler supported exceptions my tmt and test_exc programs crash. You should *disable* exceptions.

2.4.8 Watcom

next - skip - up - start

Open Watcom (version 1.7a): this works. You can set the standard option in `include.h`. The *scientific* and *fixed* manipulators don't work.

Watcom C++ (version 10a): this used to work, I don't know if it works now.

2.5 Updating from previous versions

next - skip - up - start

Newmat11 - if you are upgrading from earlier versions note the following:

- The class *Exception* in *myexcept.h* has been replaced by *BaseException* and a **typedef** statement includedso programs that use the *Exception* class will still work. If the *Exception* class was causing a problem comment out the line defining *TypeDefException* in *include.h*.
- Newmat11 does not support the TEMP_DESTROYED_QUICKLY optionsso won 't work with very old versions of Gnu G++
- Old AT&T work-arounds are removed
- The simulated Booleansclass is now stored at the end of include.h. In general, I am not testing with compilers that don't support **bool**.
- QRZ and QRZT no longer throw an exceptionif they generate a singular triangular matrix
- Converting functionsto lower case. Both the old and new names will work. See the list of functions for new and old names
- *nm_i5.mak* , *nm_il5.mak* replaced by *nm_i8.mak* , *nm_il8.mak*
- extra file *nm_misc.cpp* to be includedin compilation
- *scientific* and *fixed* manipulatorsnow recognisedin outputstatements . If you want *fixed* outputand have been using *scientific* in a previous outputstatement , you may need to includethe *fixed* manipulator. (Doesn't work in Visual C++, version 6 and open Watcom).

Newmat10 includes new maxima, minima, determinant, dot product and Frobenius norm functions, a faster FFT, revised make files for GCC and CC compilers, several corrections, new ReSize function, IdentityMatrix and Kronecker Product. Singular values from SVD are sorted. The program files include a new file, `newfft.cpp`, so you will need to include this in the list of files in your IDE and make files. There is also a new test file tmtm.cpp. Pointer arithmetic now mostly meets requirements of standard. You can use << to load data into rows of a matrix. The default options in include.h have been changed. If you are updating from a beta version of newmat09 look through the next section as there were some late changes to newmat09.

If you are upgrading from **newmat08** note the following:

- Boolean, TRUE, FALSE are now bool, true, false. See customising if your compiler supports the bool class.
- ReDimensionis now ReSize.
- The simulated exception package has been updated.
- Operators `==`, `!=`, `+=`, `-=`, `*=`, `|=`, `&=` are now supportedas binary matrix operators.
- `A+=f`, `A-=f`, `A*=f`, `A/=f`, `f+A`, `f-A`, `f*A` are supportedfor A matrix, f scalar.
- Fast trigonometrictransforms .
- Reverse functionfor reversing order of elements in a vector or matrix.
- IsSingular function.
- An optionis includedfor defining namespaces.
- 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 invaliduse 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 functionrewritten (faster).
- The documentation files newmata.txt and newmatb.txt have been amalgamatedand both are includedin the hypertextversion .
- Some of the make files reorganisedagain .

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 in include.h allows you to use compiler supportedexceptions , simulated exceptionsor disable exceptions. Edit the file include.h to select one of these three options. Don't simulate exceptionsif you have set your compiler's optionto implement exceptions.
- New QR decomposition functions.
- A non-linear least squares class.
- No need to explicitly set the AT&T optionin include.h.
- Concatenation and elementwise multiplication.
- A new GenericMatrix class.
- Sum function.
- Some of the make 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 changedto Real
- BOOL changedto Boolean
- CopyToMatrix changedto AsMatrix, etc
- real(A) changedto 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 for other changes.

2.6 Catching exceptions

next - skip - up - start

This section applies particularly to people using *compiler supported* exceptions rather than my *simulated* exceptions.

If newmat detects an error it will throw an exception. It is important that you catch this exception and print the error message. Otherwise you will get an unhelpful message like *abnormal termination*.

I suggest you set up your main program like

```
#define WANT_STREAM             // or #include <iostream>
#include "newmat.h"             // or #include "newmatap.h"
#include "newmatio.h"           // if you are using my matrix output functions

main()
{
   try
   {
      ... your program here
   }
   // catch exceptions thrown by my programs
   catch(BaseException) { cout << BaseException::what() << endl; }
   // catch exceptions thrown by other people's programs
   catch(...) { cout << "exception caught in main program" << endl; }
   return 0;
}
```

Or see my file *nm_ex1.cpp* for an easy way of organising this.

If you are using a GUI version rather a console version of the program you will need to catch the exception and display the error message in a pop-up window.

If you are using my simulated exceptions or have set the disable exceptions option in *include.h* then uncaught exceptions automatically print the error message generated by the exception so you can ignore this section. Alternatively use *Try*, *Catch* and *CatchAll* in place of *try*, *catch* and *catch(...)* in the preceding code.

See the section on exceptions for more information on the exception structure.

2.7 Examples

next - skip - up - start

I include a number of example files. See the sections on make files and on compilers for information about compiling them.

Invert matrix: `nm_ex1.cpp`. Load values into a 4x4 matrix; invert it and check the result. The output is in `nm_ex1.txt`.

Eigenvalues and eigenvectors of Hilbert matrix: `nm_ex2.cpp`. Calculate the eigenvalues and eigenvectors of a 7x7 Hilbert matrix. The output is in `nm_ex2.txt`.

Values in `precisio.h`: `nm_ex3.cpp`.

Linear regression example: `example.cpp`. This gives a 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 for a comment on the reliability of this check and the use of the DO_FREE_CHECK option.

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 and testing exception handling.

2.8 Testing

next - skip - up - start

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.

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

If you are carrying out some form of bounds checking, for example, with Borland's *CodeGuard*, then disable the testing of the Numerical Recipes in C interface. Activate the statement `#define DONT_DO_NRIC` in tmt.h.

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

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 foolproof. For example, 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 *some* of the compilers I work with, so I think this is a worthwhile test. The compilers that the test seems to work for include the Borland compilers, Microsoft VC++ 6 , Watcom, and Gnu 2.96 for Linux.

If the DO_FREE_CHECK 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.

2.9 Bugs

next - skip - up - start

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

2.10 Problem areas

next - skip - up - start

This section lists parts of *Newmat* which users (including me) have found difficult or unnatural. Also see the newmat FAQ list on my website.

Invert, element access, matrix multiply etc causes the program to crash

Newmat throws an exceptionwhen it detects an error. This can cause a program crash unless the exceptionis caught with a *catch* statement. See catchingexceptions .

`1x1` matrix not automatically converted to a Real

Use the as_scalar() member functionor the dotproduct() functionto take the dot product of two vectors.

Constructors do not initialise elements

For example, `Matrix A(4,5);` does not initialisethe elements of `A`. Use the statement `A=0.0` to set the values to zero.

resize does not initialise elements

For example, `A.resize(5,6);` does not set the elements of `A`. If you want to keep values use `resize_keep`. See resize.

Setting Matrix to a scalar sets all the values

`A(1,3) = 0.0;` sets one element of a Matrix to zero. `A = 0.0;` sets all the elements to zero. This is very convenient but also a source of error that is hard to see if you wanted `A(1,3) = 0.0;` but left out the element details.

Symmetry not detected automatically

For example, `SymmetricMatrix SM = A.t() * A;` will fail. Use `SymmetricMatrix SM; SM << A.t() * A;`

<< does not work with constructors

For example, `SymmetricMatrix SM << A.t() * A;` does not work.

Multiple multiplication may be inefficient

For example, `A * B * X` where `A` and `B` are matrices and `X` is a column vector is likely to be much slower than `A * (B * X)` .

2.11 List of files

next - skip - up - start

Documentation	readme.txt	readme file
	nm11.htm	documentation file
	add_time.pgn	image used by nm11.htm
	rbd.css	style sheet for nm11.htm
	_newmat.dox	descriptionfile for Doxygen
	_rbd_com.dox	descriptionfile for Doxygen
Header files	controlw.h	control word definitionfile
	include.h	details of includefiles and options
	myexcept.h	general exceptionhandler definitions
	newmat.h	main matrix class definitionfile
	newmatap.h	applications definitionfile
	newmatio.h	input/outputdefinitionfile
	newmatnl.h	non-linear optimisation definitionfile
	newmatrc.h	row/column functionsdefinitionfiles
	newmatrm.h	rectangularmatrix access definitionfiles
	precisio.h	numerical precision constants
	solution.h	one dimensionalsolve definitionfile
Program files	bandmat.cpp	band matrix routines
	cholesky.cpp	Choleskydecomposition
	evalue.cpp	eigenvalues and eigenvectorcalculation
	fft.cpp	fast Fourier, trig. transforms
	hholder.cpp	QR routines
	jacobi.cpp	eigenvalues by the Jacobi method
	myexcept.cpp	general error and exceptionhandler
	newfft.cpp	new fast Fourier transform
	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	outputroutines
	newmatex.cpp	matrix exceptionhandler
	newmatnl.cpp	non-linear optimisation
	newmatrm.cpp	rectangularmatrix access functions
	nm_misc.cpp	miscellaneousclasses , functions
	sort.cpp	sorting functions
	solution.cpp	one dimensionalsolve
	submat.cpp	submatrix functions
	svd.cpp	singular value decomposition
Example files	nm_ex1.cpp	simple example - invert matrix
	nm_ex1.txt	outputfrom nm_ex 1.cpp
	nm_ex2.cpp	simple example - eigenvalues of Hilbert matrix
	nm_ex2.txt	outputfrom nm_ex 2.cpp
	nm_ex3.cpp	scientific format and constantsfrom precisio .h
	nm_ex3.txt	outputfrom nm_ex 3.cpp
	example.cpp	example of use of package
	example.txt	outputfrom example
	sl_ex.cpp	example of OneDimSolve routine
	sl_ex.txt	outputfrom example

	nl_ex.cpp	example of non-linear least squares
	nl_ex.txt	outputfrom example
	garch.cpp	example of maximum-likelihood fit
	garch.dat	data file for garch.cpp
	garch.txt	outputfrom example
	test_exc.cpp	demonstration exceptions
	test_exc.txt	outputfrom test _exc.cpp
Test files	tmt.h	header file for test files
	tmt*.cpp	test files (see the section on testing)
	tmt.txt	outputfrom test files
Make files	nm_gnu.mak	make file for Gnu G++
	nm_cc.mak	make file for AT&T, Sun and HPUX
	nm_b55.mak	make file for Borland C++ 5.5
	nm_b56.mak	make file for Borland Builder C++ 6
	nm_b58.mak	make file for Borland Builder C++ 8
	nm_m6.mak	make file for VC++ 6&7
	nm_m8.mak	make file for VC++ 8
	nm_i8.mak	make file for Intel C++ 8,9 under Windows
	nm_i10.mak	make file for Intel C++ 10 under Windows
	nm_il8.mak	make file for Intel C++ 8,9,10 under Linux
	nm_ow.mak	make file for Open Watcom
	newmat.lfl	library file list for use with genmake
	nm_targ.txt	target file list for use with genmake
	Makefile.in	used for compilingwith Opt++

These are the .cpp files you need to include in your make file or project:

Basic newmat

- bandmat.cpp
- myexcept.cpp
- newmat1.cpp
- newmat2.cpp
- newmat3.cpp
- newmat4.cpp
- newmat5.cpp
- newmat6.cpp
- newmat7.cpp
- newmat8.cpp
- newmat9.cpp
- newmatex.cpp
- newmatrm.cpp
- submat.cpp

Factorisations etc

- cholesky.cpp
- evalue.cpp
- fft.cpp
- hholder.cpp
- jacobi.cpp
- sort.cpp
- svd.cpp
- newfft.cpp
- nm_misc.cpp

Nonlinear routines

- newmatnl.cpp
- solution.cpp

3. Reference manual

next - skip - up - start

See also function summary.

3.1 Constructors

next - skip - up - start

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

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

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

```
SquareMatrix SQ(n);
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 UpperBandMatrix and LowerBandMatrix are upper and lower triangular band matrices. So an UpperBandMatrix is essentially a BandMatrix with *lower* = 0 and a LowerBandMatrix is a BandMatrix with *upper* = 0.

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

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

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

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

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

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

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

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

```
Matrix A;
```

In this case the dimension must be set by an assignment statement or a resize statement.

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

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

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

3.2 Accessing elements

next - skip - up - start

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

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

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

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

Symmetric matrices are stored as lower triangular matrices. It is important to remember this if you are using the `A[i][j]` method of accessing elements. Make sure the first subscript is greater than or equal to the second subscript. However, if you are using the `A(i,j)` method the program will swap `i` and `j` if necessary; so it doesn't matter if you think of the storage as being in the upper triangle (but it *does* matter in some other situations such as when entering data).

The IdentityMatrix type does not support element access.

For interfacing with traditional C functions that involve one and two dimensional arrays see accessing C functions.

3.3 Assignment and copying

next - skip - up - start

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. The name *Inject* can be used instead on *inject*.

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

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

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

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

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

To swap the values in two matrices `A` and `B` use one of the following expressions

```
A.swap(B);

swap(A,B);
```

The matrices A and B must be of the same type. This can be any of the matrix types listed in the section on constructors, CroutMatrix, BandLUMatrix or GenericMatrix. Swap works by switching pointers and does not do any actual copying of the main data arrays.

Notes:

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

3.4 Entering values

next - skip - up - start

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;
```

or

```
Matrix A(3,2);
int 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.

Note that you enter only the values stored in a matrix. For example

```
SymmetricMatrix A(2);
Real a[] = { 11,12,22 };
A << a;
```

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

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

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

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

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

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


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

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

3.5 Unary operators

next - skip - up - start

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)
ColumnVector X = A.sum_rows();         // sum of elements
                                       // of each row
RowVector X = A.sum_columns();         // sum of elements
                                       // of each column
ColumnVector X = A.sum_square_rows();  // sum of squares of
                                       // elements of each row
RowVector X = A.sum_square_columns();  // sum of squares of
                                       // elements of each column
```

See the functionsummary list for the older depreciatedfunctionname .

3.6 Binary operators

next - skip - up - start

The package supports binary operations

```
X = A + B;        // matrix addition
X = A - B;        // matrix subtraction
X = A * B;        // matrix multiplication
X = A.i() * B;    // equation solve (square matrix A)
X = A | B;        // concatenate horizontally (concatenate the rows)
X = A & B;        // concatenate vertically (concatenate the columns)
X = SP(A, B);     // elementwise product of A and B (Schur product)
X = KP(A, B);     // Kronecker product of A and B
X = crossproduct(A, B);          // vector cross product - see notes
X = crossproduct_rows(A, B);     // cross product of rows
X = crossproduct_columns(A, B);  // cross product of columns
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;
A.SP_eq(B);       // A = SP(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 operationsto use `A*(B*C)` .

- In the equationsolve example case the inverse is not explicitly calculated. An LU decomposition of `A` is performed and this is appliedto `B`. This is more efficient than calculating the inverse and then multiplying. See also multiplematrix_solving.
- The package does not (yet?) recognise `B*A.i()` as an equationsolve and the inverse of `A` would be calculated. It is probablybetter to use `(A.t().i()*B.t()).t()` .
- Horizontal or vertical concatenation returns a result of type Matrix, RowVectoror 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 complicatedconcatenations of matrices, consider instead using submatrices.
- See the section on submatrices on using a submatrix on the RHS of an expression.
- crossproduct- assumes A and B are both RowVectors of length3 or both ColumnVectors of length3. Result is a Matrix of same dimensionas A or B (will automatically convert to RowVectorif A and B are RowVectors and ColumnVector if they are ColumnVectors).
- crossproduct_rows - assumes A and B are of type Matrix with the same number of rows and 3 columns. Does a cross product on corresponding pairs of rows.
- crossproduct_columns - assumes A and B are of type Matrix with the same number of columns and 3 rows. Does a cross product on corresponding pairs of columns.
- Two matrices are equal if their difference is zero. They may be of different types. For the CroutMatrixor BandLUMatrixthey must be of the same type and have all their elements equal. This is not a very useful operator and is includedfor compatibility with some container templates.
- The inequality operators are includedfor compatibility with some versions of the standard template library. If actuallycalled , they will throw an exception. So don't try to sort a *list* of matrices.
- A row vector multiplied by a column vector yields a 1x1 matrix , *not* a Real. To get a Real result use either as_scalar or dotproduct.
- The result from Kronecker product, `KP(A, B)` , possesses an attributesuch as upper triangular, lower triangular, band, symmetric, diagonal if both of the matrices `A` and `B` have the attribute. If `A` is band and `B` is a square type (eg SquareMatrix, Diagonal, LowerTriangularMatrix etc) then the result is band.
- Elementwise product is also known as the Schur product or the Hadamard product.
- See the functionsummary_list for the older depreciatedfunctionnames .

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

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

Use instead

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

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

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

3.7 Matrix and scalar

next - skip - up - start

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

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

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

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

3.8 Scalar functions of a matrix - size & shape

next - skip - up - start

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

```
int m = A.nrows();                    // number of rows
int n = A.ncols();                    // number of columns
MatrixType mt = A.type();             // type of matrix
Real* s = A.data();                   // pointer to array of elements
const Real* s = A.data();             // pointer to array of elements
                                      //    where A is const
const Real* s = A.const_data();       // pointer to array of elements
int l = A.size();                     // length of array of elements
MatrixBandWidth mbw = A.bandwidth();  // upper and lower bandwidths
```

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

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

See the functionsummary_list for the older depreciatedfunctionnames .

3.9 Scalar functions of a matrix - maximum & minimum

next - skip - up - start

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

```
int i, j;
Real mv = A.maximum_absolute_value();    // maximum of absolute values
Real mv = A.minimum_absolute_value();    // minimum of absolute values
Real mv = A.maximum();                   // maximum value
Real mv = A.minimum();                   // minimum value
Real mv = A.maximum_absolute_value1(i);  // maximum of absolute values
Real mv = A.minimum_absolute_value1(i);  // minimum of absolute values
Real mv = A.maximum1(i);                 // maximum value
Real mv = A.minimum1(i);                 // minimum value
Real mv = A.maximum_absolute_value2(i,j);// maximum of absolute values
Real mv = A.minimum_absolute_value2(i,j);// minimum of absolute values
Real mv = A.maximum2(i,j);               // maximum value
Real mv = A.minimum2(i,j);               // minimum value
```

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

The versions `A.maximum_absolute_value1(i)`, etc return the location of the extreme element in a RowVector, ColumnVector or DiagonalMatrix. The versions `A.maximum_absolute_value2(i,j)`, etc return the row and column numbers of the extreme element. If the extreme value occurs more than once the location of the last one is given.

The versions maximum_absolute_value(A), minimum_absolute_value(A), maximum(A), minimum(A) can be used in place of A.maximum_absolute_value(), A.minimum_absolute_value(), A.maximum(), A.minimum().

3.10 Scalar functions of a matrix - numerical

next - skip - up - start

```
Real r = A.as_scalar();                 // value of 1x1 matrix
Real ssq = A.sum_square();              // sum of squares of elements
Real sav = A.sum_absolute_value();      // sum of absolute values
Real s = A.sum();                       // sum of values
Real norm = A.norm1();                  // maximum of sum of absolute
                                           values of elements of a column
Real norm = A.norm_infinity();          // maximum of sum of absolute
                                           values of elements of a row
Real norm = A.norm_Frobenius();         // square root of sum of squares
                                        // of the elements
Real t = A.trace();                     // trace
Real d = A.determinant();               // determinant
LogAndSign ld = A.log_determinant();    // natural log of determinant
bool z = A.is_zero();                   // test all elements zero
bool s = A.is_singular();               // A is a CroutMatrix or
                                           BandLUMatrix
Real s = dotproduct(A, B);              // dot product of A and B
                                        // interpreted as vectors
```

See the functionsummary_list for the older depreciatedfunctionnames .

`A.log_determinant()` 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.log_value() to get the log of the absolute value.
```

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

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

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

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

The versions sum(A), sum_square(A), sum_absolute_value(A), trace(A), log_determinant(A), determinant(A), norm 1(A), norm_infinity(A), norm_Frobenius(A) can be used in place of A.sum(), A.sum_square(), A.sum_absolute_value(), A.trace(), A.log_determinant(), A.determinant(A), A.norm 1(), A.norm_infinity(), A.norm_Frobenius().

3.11 Submatrices

next - skip - up - start

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

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

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

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

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

to set a submatrix equal to a constant.

The following are variants of submatrix:

```
A.sym_submatrix(f,l)            //   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
```

See the functionsummary_list for the older depreciatedfunctionnames .

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

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

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

```
A.submatrix(fr,lr,fc,lc)          If A is RowVector or
                                  ColumnVector then same type
                                  otherwise type Matrix
A.sym_submatrix(f,l)              Same type as A
A.rows(f,l)                       Type Matrix
A.row(f)                          Type RowVector
A.columns(f,l)                    Type Matrix
A.column(f)                       Type ColumnVector
```

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

A submatrix can appear on the left-hand side of += or -= with a matrix expression on the right-hand side. It can also appear on the left-hand side of +=, -=, *= or /= with a Real on the right-hand side. In each case there must be no loss of information. The SP_eq function is also defined for a submatrix.

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

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

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

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

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

3.12 Change dimensions

next - skip - up - start

The following operations change the dimensions of a matrix. The values of the elements are lost, for `resize`. `resize_keep` keeps element values and zeros the elements in the matrix with the new size that are not in the matrix with the old size.

```
A.resize(nrows,ncols);        // for type Matrix or nricMatrix
A.resize(n);                  // for other types, except Band
A.resize(n,lower,upper);      // for BandMatrix
A.resize(n,lower);            // for LowerBandMatrix
A.resize(n,upper);            // for UpperBandMatrix
A.resize(n,lower);            // for SymmetricBandMatrix
A.resize(B);                  // set dims to those of B
A.cleanup();                  // resize to zero dimensions
A.resize_keep(nrows,ncols);   // for type Matrix or nricMatrix, keep values
A.resize_keep(n);             // for other types, except Band, keep values
```

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

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

Remember that `resize` destroys values. If you want to resize, but keep the values in the bit that is left use `resize_keep`.

See the functionsummary_list for the older depreciatedfunctionnames .

3.13 Change type

next - skip - up - start

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.as_row()
A.as_column()
A.as_diagonal()
A.as_matrix(nrows,ncols)
A.as_scalar()
```

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

See the functionsummary_list for the older depreciatedfunctionnames .

3.14 Multiple matrix solve

next - skip - up - start

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

```
ColumnVector p, q;
...
CroutMatrix X = A;                // carries out LU decomposition
ColumnVector Ap = X.i()*p; ColumnVector Aq = X.i()*q;
LogAndSign ld = X.log_determinant();
```

rather than

```
ColumnVector p, q;
...
ColumnVector Ap = A.i()*p; ColumnVector Aq = A.i()*q;
LogAndSign ld = A.log_determinant();
```

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 BandLUMatrix can be copied and you can have a constructor with no parameters (use `=` to give it values). They work with *release(), release_and_delete* and GenericMatrix and ReturnMatrix. You can't do any other manipulation apart from taking the inverse or solving with `i()`, or finding the determinant or log determinant. See the function summary list for accessing the internals of a CroutMatrix or BandLUMatrix.

You can 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. It doesn't know about positive definite matrices so won't use Cholesky.

3.15 Memory management

next - skip - up - start

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.

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

Make your function of type ReturnMatrix . Then precede the return statement with a *release* statement (or a *release_and_delete* 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->release_and_delete(); return *m;
}
```

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

```
return A.for_return();
```

or

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

Do not forget to make the function of type ReturnMatrix.

In particular, **don't** do

```
Matrix MakeAMatrix()
{
   Matrix A;                // or any other matrix type
   ......
   A.release(); return A;   // will compile but could give wrong answers.
}
```

since with some compilers `A` might retain its *released* status after being returned.

You can also use `.release()` or `->release_and_delete()` 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 `->release_and_delete` for matrices created by new if you want to completely delete the matrix after it is accessed.

See the functionsummary_list for the older depreciatedfunctionnames .

3.16 Efficiency

next - skip - up - start

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

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

3.17 Output

next - skip - up - start

To print a matrix use an expression like

```
Matrix A, B C;
......
cout << setw(10) << setprecision(5) << A << endl;
cout << setw(10) << setprecision(5) << scientific << B << endl;
cout << setw(10) << setprecision(5) << fixed << C << endl;
```

This will work only with systems that support the standard input/output routines including manipulators. The scientific and fixed manipulators won't work with Visual C++, version 6.

You need to #include the files iostream.h, iomanip.h, newmatio.h in your C++ source files that use this facility. The files iostream.h, iomanip.h will be included automatically if you include the statement `#define WANT_STREAM` at the beginning of your source file. So you can begin your file with either

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

or

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

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

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

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

3.18 Unspecified type

next - skip - up - start

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 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. `swap` does work with objects of type GenericMatrix. See also the paragraph on LinearEquationSolver.

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

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

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

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

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

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

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

Do not try to construct a GeneralMatrix or BaseMatrix.

3.19 Cholesky decomposition

next - skip - up - start

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);
```


See section 3.32 on updating a Cholesky decomposition.

3.20 QR decomposition

next - skip - up - start

This is a variant on the usual QR transformation.

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

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

  s    t
```

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

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

  s    t
```

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

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

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

```
Y - X*b
```

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

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

Two routines are provided in *newmat*:

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

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

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

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

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

which does both of these.

To extend `U` to a square orthogonal matrix see the function for extending an orthonormal set of columns.

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

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

See the section on updating a Cholesky decomposition for updating `U`.

Alternatively to update `U` or `L` with a new block of data, `X`, one can use

```
updateQRZ(X, U);
```

or

```
updateQRZT(X, L);
```

You can update the M matrix with

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

At present, there is no corresponding function for `updateQRZT`. Also note, at present, `updateQRZ(X, Y, M)` is not optimised for accessing contiguous locations.

If you have carried out QRZ transforms on two blocks of data, `X1`, `X2`, (same number of columns in each) with the data to be fitted in `Y1`, `Y2`, you can combine them as follows

```
// QRZ transforms of two blocks of data
UpperTriangularMatrix U1, U2; Matrix M1, M2;
QRZ(X1, U1); QRZ(X2, U2); QRZ(X1, Y1, M1); QRZ(X2, Y2, M2);

// Now combine the results
updateQRZ(U1, U2); updateQRZ(U1, M1, M2);
```

Results are in `U2` and `M2`. The values in `U1` and `M1` are modified so use copies if you still need the original values.

The functions `updateQRZ(U1, U2)` and `updateQRZ(U1, M1, M2)` are not optimised for accessing contiguous locations. This is probably not an issue, since usually all the data will fit into cache memory.

Notes on updateQRZ and updateQRZT:

- Use these routines if your data is arriving in blocks or there is too much to fit into memory.
- Use `QRZ` or `QRZT` on the first block of data or you can use `updateQRZ` or `updateQRZT` if `U` or `L` is correctly dimensioned and set to zero.
- The block of data, `X`, will be modified by these routines but the modified data is probablynot useful for further analysis.
- The signs of some of the rows of `U` or columns of `L` maybe the reverse of what you would expect (i.e. you get the negativeof what you get from QRZ or QRZT appliedto the whole dataset). Usually this will not be a problem.
- The second version of updateQRZ is sometimes a more useful way of combiningblocks , especiallyif you want to use the *reduce* functionin *Intel's ThreadedBuildingBlocks* to paralleliseQRZ. (Contactme for what you need to do to make Newmat compatiblewith *Threaded BuildingBlocks*).
- See the functionsummary list for the older depreciatedfunctionnames .

3.21 Singular value decomposition

next - skip - up - start

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

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

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

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

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

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

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

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

To extend `U` to a square orthogonal matrix see the function for extending an orthonormal set of columns.

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

If *m* < *n* apply the SVD transform to the transpose of A and swap U and V. If necessary, extend U to a square matrix as described above.

3.22 Eigenvalue decomposition

next - skip - up - start

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

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

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

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

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

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

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

eigenvalues(A,D);                 // A symmetric
eigenvalues(A,D,S);               // A, S symmetric; S is for back
```

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

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

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

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

See the functionsummary list for the older depreciatedfunctionnames .

3.23 Sorting

next - skip - up - start

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

```
sort_ascending(A);

sort_descending(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?

See the functionsummary list for the older depreciatedfunctionnames .

3.24 Fast Fourier transform

next - skip - up - start

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

where X, Y, F, G are ColumnVectors. 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

$$h[k] = \sum_{j=0}^{n-1} z[j] \exp(-2\pi i jk/n)$$

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

I use one of two methods:

- A program originallywritten by Sande and Gentleman. This requires that n can be expressed as a product of small numbers.

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

Related functions

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

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

See also the section on fast trigonometric transforms.

There are also two dimensional versions

```
FFT2(X, Y, F, G);                       // F=X and G=Y are OK
FFT2I(F, G, X, Y);                      // inverse, X=F and Y=G are OK
```

where X, Y, F, G are of type Matrix. X and Y are the real and imaginary input matrices; F and G are the real and imaginary output matrices. The dimensions of Y must be the same as those of X and should be the product of numbers less than about 10 for fast execution.

The formula is

$$h[p,q] = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} z[j,k] \exp(-2\pi i (jp/m + kq/n))$$

where z[j,k] is stored complex and stored in X(j+1,k+1) and Y(j+1,k+1) and X and Y have dimension m x n. Likewise h[p,q] is complex and stored in F(p+1,q+1) and G(p+1,q+1).

3.25 Fast trigonometric transforms

next - skip - up - start

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

The functions I define are

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

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

Here are the formulae:

DCT

```
                m-1                                 k
v[k] = u[0]/2 + SUM { u[j] cos (pi jk/m) } + (-) u[m]/2
                j=1
```

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

DST

```
        m-1
v[k] = SUM { u[j] sin (pi jk/m) }
        j=1
```

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

DCT_II

```
        m-1
v[k] = SUM { u[j] cos (pi (j+1/2)k/m) }
        j=0
```

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

DST_II

```
         m
v[k] = SUM { u[j] sin (pi (j-1/2)k/m) }
        j=1
```

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

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

3.26 Interface to Numerical Recipes in C

next - skip - up - start

This section refers to *Numerical Recipes in C*. This section is **not** relevant to *Numerical Recipes in C++*. I'll put a note on the website soon on how to interface with *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
```

3.27 Exceptions

next - skip - up - start

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.

Suppose you have edited `include.h` to use my *simulated* exceptions or to *disable* exceptions. If there is no catch statement or exceptions are disabled then my `Terminate()` function in `myexcept.h` is called when you throw an exception. 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 class. Also see the section on error messages for additional notes on the messages generated by the exceptions.

You can also print this information in a *catch* clause by printing `Exception::what()`.

If you are using *compiler supported* exceptions then see the section on catching exceptions.

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 and later versions. 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 inbuilt exceptions provided by a compiler, simulating exceptions, or disabling exceptions. See customising 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. Skip the rest of this section and the next section if you are using compiler supported 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.

Note that the simulation exception mechanism does not work if you define arrays of matrices.

3.28 Cleanup after an exception

next - skip - up - start

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 except.h;
2. define a function `void CleanUp()` in that class to return all heap memory;
3. includethe 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 includea copy constructorin you class definition, that is, somethinglike

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

Use `CleanUp()` rather than `cleanup()` since this is what is defined in class Janitor.

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()`.

3.29 Non-linear applications

next - skip - up - start

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 program and because it is a simpler version of the coding 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. These classes work on very well-behaved functions but need upgrading for less well-behaved functions. I haven't followed the usual practice of inflating the values of the diagonal elements of the matrix of second derivatives. I originally thought I could avoid this if my program had a good line search. But this was wrong and when I use this program on all but the most well-behaved problems I run the fit first with the diagonal elements inflated by a factor of 2 to 5 and the critical value for the stopping criterion set to something like 50. Then rerun with with no inflation factor and critical value 0.0001.

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

3.30 Standard template library

next - skip - up - start

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 operators which seem to be required by some versions of the STL.

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

- Don't use simulated exceptions.
- Make sure the option DO_FREE_CHECK is *not* turned on.
- 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 containertemplates like to copy their elements. For the vector containerthis 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.
- You won't be able to sort the containeror do anything that would call an inequality operator.

3.31 Namespace

next - skip - up - start

Namespace is used 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 C++ version 5 and Builder 5 and 6.

Microsoft Visual C++ version 5 works in my example and test files, but fails with apparently insignificant changes (it may be more reliable if you have applied service pack 3). If you #include "newmatap.h", but no other newmat include file, then also #include "newmatio.h". It seems to work with Microsoft Visual C++ version 6 *if* you have applied at least service pack 2.

My use of namespace does not work with Gnu g++ version 2.8.1 but does work with versions 3.x.

I have defined the following namespaces:

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

3.32 Updating the Cholesky matrix

next - skip - up - start

Suppose `X` is matrix and `U` has been formed with either

```
SymmetricMatrix A; A << X.t() * X;
UpperTriangularMatrix U = Cholesky(A).t();
```

or

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


See sections 3.19 and 3.20.

Now suppose we want to append an extra row to `X` or delete a row from `X` or rearrange the columns of `X`. The functions described here allow you to update `U` without recalculating it.

```
update_Cholesky(U, x);                    // x is a RowVector
downdate_Cholesky(U, x);                  // x is a RowVector
right_circular_update_Cholesky(U, j, k);  // j and k are integers
left_circular_update_Cholesky(U, j, k);   // j and k are integers
```

`update_Cholesky` carries out the modification of `U` corresponding to appending an extra row `x` to `X`.

`downdate_Cholesky` carries out the modification corresponding to removing a row `x` from `X`. A `ProgramException` exception is thrown if the modification fails.

`right_circular_update_Cholesky` supposes that columns `j,j+1,...k` of `X` are replaced by columns `k,j,j+1,...k-1`.

`left_circular_update_Cholesky` supposes that columns `j,j+1,...k` of `X` are replaced by columns `j+1,...k,j`.

These functions are based on a contribution from Nick Bennett of Schlumberger-Doll Research. See also the LINPACK User's Guide, Chapter 10, Dongarra et. al., SIAM, Philadelphia, 1979.

Where you want to append a number of new rows consider using the update routine in section 3.20.

See the functionsummary_list for the older depreciatedfunctionnames .

3.33 Accessing C functions

next - skip - up - start

You have a *C* function that uses one and two dimensional arrays as vectors and matrices. You want to access it from *Newmat*.

One dimensional arrays are easy. Set up a *ColumnVector*, *RowVector* or *DiagonalMatrix* with the correct dimension and where the function has a `double*` argument enter `X.data()` where `X` denotes the *ColumnVector*, *RowVector* or *DiagonalMatrix*. (I am assuming you have left *Real* being *typedef*ed as a *double*). If you have a `const double*` argument use `X.const_data()`.

You can't do this with two dimensional arrays where you have a `double**` argument. *Newmat* includes classes *RealStarStar* and *ConstRealStarStar* for this situation. To access a `Matrix A` from a function `c_function(double** a)` use either

```
c_function(RealStarStar(A));
```

or

```
RealStarStar a(A);
c_function(a);
```

If the argument is `const double**` use *ConstRealStarStar*.

The `Matrix A` must be correctly dimensioned and must not be resized or set equal to another matrix between setting up the *RealStarStar* object and its use in the function. Also the following construction will not work

```
double** a = RealStarStar(A);       // wrong
c_function(a);
```

since the *RealStarStar* structure will have been destroyed before you get to the second line.

3.34 Simple integer array class

next - skip - up - start

This is primarily for use within *Newmat*. You can set up a simple array of integers with the *SimpleIntArray* class. Here are the descriptions of the constructors and functions.

`SimpleIntArray A;`	Constructs int array of lengthzero
`SimpleIntArray A(n);`	Constructs int array of length *n* - individual elements are *not* initialised
`A = i;`	sets values to *i* where *i* is an int variable
`A = B;`	sets values to those of *B* where *B* is a SimpleIntArray, change size if necessary.
`n = A.size();`	return the lengthof *A*
`A.resize(n);`	change the lengthof *A*, don't keep values
`A.resize_keep(n);`	change the lengthof *A*, do keep values; if lengthis being increased set new elements to zero.
`int x = A[i];`	element access; *i* runs from 0 to *n*-1
`int* d = A.data();`	get beginning of data array
`const int* d = A.data();`	get beginning of data array as const int* if *A* is const
`const int* d = A.const_data();`	get beginning of data array as const int*
`A.cleanup()`	resize to zero length

3.35 Extend orthonormal set of columns

next - skip - up - start

Suppose a *Matrix* `A`'s first `n` columns are orthonormal so that `A.Columns(1,n).t() * A.Columns(1,n)` is the identity matrix. Suppose we want to fill out the remaining columns of `A` to make them orthonormal so that `A.t() * A` is the identity matrix. Then use the function

```
extend_orthonormal(A, n);
```

Matrix `A` is then *replaced* by the matrix with the additional columns.

Use this function to extend `U` from the ORZ or SVD decompositions to form a square (orthogonal) matrix.

Notes:

- the first `n` columns of `A` must be orthonormal
- `n` must be less or equal to the number of columns of `A`
- the number of columns of `A` must be less than or equal to the number of rows of `A`

3.36 Miscellaneous functions

next - skip - up - start

The section includes some miscellaneous functions I have needed for my work. So far there are only the Helmert transforms.

Helmert transforms

This section refers to the Helmert transform used in some statistical packages for extracting contrasts. It is different from the Helmert transform used in geodesy. The version of the transform I am going to use is based on the `n x n` matrix

```
| 1/sqrt(1*2)                          | * | -1  1              |
|     1/sqrt(2*3)                      |   | -1 -1  2           |
|          1/sqrt(3*4)                 |   | -1 -1 -1  3        |
|                ...                   |   |  ...               |
|                   1/(sqrt(n-1)*n)    |   | -1 -1  ...  -1 n-1 |
|                         1/sqrt(n)    |   |  1  1  ...       1 |
```

You can interpret multiplying a column vector by this matrix as follows. Form the *k*-th element in the resulting column vector by taking the *k+1*-th element and subtracting the average of the previous *k* elements. Then multiply by sqrt((*k+1*)/*k*) so we get an orthonormal transform. The last element is just the sum divided by sqrt(*n*). Usually one will omit the last element since we want just the contrasts.

Here are the functions. `X` and `Y` are ColumnVectors, `A` and `B` are matrices, `b` is a boolean, `j`, `n` are integers, `x` is a Real.

`A = Helmert(n, b);` `A = Helmert(n);`	Return the *n* x *n* Helmert matrix or the (*n*-1) x *n* version with the last row omitted.
`Y = Helmert(X,b);` `Y = Helmert(X);`	Multiplyby the Helmert matrix.
`Y = Helmert(n,j,b);` `Y = Helmert(n,j);`	Return *j*-th column of Helmert matrix as a ColumnVector.
`X = Helmert_transpose(Y,b);` `X = Helmert_transpose(Y);`	Multiplyby the transpose of the Helmert matrix.
`x = Helmert_transpose(Y,j,b);` `x = Helmert_transpose(Y,j);`	Multiplyby the transpose of the Helmert matrix, return just the *j*-th element.
`B = Helmert(A,b);` `B = Helmert(A);`	Multiplya Matrix by the Helmert matrix.
`A = Helmert_transpose(B,b);` `A = Helmert_transpose(B);`	Multiplya Matrix by the transpose of the Helmert matrix.

If *b* is true the full Helmert matrix is used, if it is false or omitted the *n*-th row (or *n*-column of the transpose) is omitted.

4. Error messages

next - skip - up - start

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 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. Notes on the design of the library

next - skip - up - start

5.1 Safety, usability, efficiency
5.2 Matrix vs array library
5.3 Design questions
5.4 Data storage
5.5 Memory management- 1
5.6 Memory management- 2
5.7 Evaluation of expressions
5.8 Explosionin the number of operations
5.9 Destruction of temporaries
5.10 A calculus of matrix types
5.11 Pointer arithmetic
5.12 Error handling
5.13 Sparse matrices
5.14 Complex matrices

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.

5.1 Safety, usability, efficiency

next - skip - up - start

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.

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.

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 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 with less errors.

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

Then there is the data() 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.

Some performance data

This section looks at the performance on *newmat* for simple sums, comparing it with C code and with a simple array program. (This is now very old data).

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;`, `X=A+B+C+D+E;` where `X,A,B,C,D,E` are of type ColumnVector, with a variety of programs. I am using Microsoft VC++, version 6 in console mode under windows 2000 on a PC with a 1 ghz Pentium III and 512 mbytes of memory.

length	iters.	newmat	C	C-res.	subs.	array
X = A + B						
2	5000000	27.8	0.3	8.8	1.9	9.5
20	500000	3.0	0.3	1.1	1.9	1.2
200	50000	0.5	0.3	0.4	1.9	0.3
2000	5000	0.4	0.3	0.4	2.0	1.0
20000	500	4.5	4.5	4.5	6.7	4.4
200000	50	5.2	4.7	5.5	5.8	5.2
X = A + B + C						
2	5000000	36.6	0.4	8.9	2.5	12.2
20	500000	4.0	0.4	1.2	2.5	1.6
200	50000	0.8	0.3	0.5	2.5	0.5
2000	5000	3.6	4.4	4.6	9.0	4.4
20000	500	6.8	5.4	5.4	9.6	6.8
200000	50	8.6	6.0	6.7	7.1	8.6
X = A + B + C + D						
2	5000000	44.0	0.7	9.3	3.1	14.6
20	500000	4.9	0.6	1.5	3.1	1.9
200	50000	1.0	0.6	0.8	3.2	0.8
2000	5000	5.6	6.6	6.8	11.5	5.9
20000	500	9.0	6.7	6.8	11.0	8.5
200000	50	11.9	7.1	7.9	9.5	12.0
X = A + B + C + D + E						
2	5000000	50.6	1.0	9.5	3.8	17.1
20	500000	5.7	0.8	1.7	3.9	2.4
200	50000	1.3	0.9	1.0	3.9	1.0
2000	5000	7.0	8.3	8.2	13.8	7.1
20000	500	11.5	8.1	8.4	13.2	11.0
200000	50	15.2	8.7	9.5	12.4	15.4

I have graphed the results and included rather more array lengths.

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 summation is repeated 5 times within each loop, for example:

```
for (i=1; i<=m; ++i)
{
   X1 = A1+B1+C1; X2 = A2+B2+C2; X3 = A3+B3+C3;
   X4 = A4+B4+C4; X5 = A5+B5+C5;
}
```

The column labelled *newmat* is using the standard *newmat* add. The column labelled *C* uses the usual C method: `while (j1--) *xl++ = *al++ + *bl++;` . The following column also includes an `X.resize()` in the outer loop to correspond to the reassignment of memory that *newmat* would do. 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 final column is the time taken by a simple array package. This uses an alternative method for avoiding temporaries and unnecessary copies that does not involve runtime tests. It does 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 length2 and doesn't do well for length20 . There is a lot of code in *newmat* for determiningwhich sum algorithm to use and it is not surprising that this impacts on performance for small lengths. However the *array* program is also havingdifficultywith length2 so it is unlikelythat the problem could be completelyeliminated .
- For arrays of length2000 or longer *newmat* is doing about as well as C and slightlybetter than C with resize in the `X=A+B` table. For the other two tables it tends to be slower, but not dramaticallyso.
- It is really important for fast processing with the Pentium III to stay within the Pentium cache.
- Additionusing the *newmat* subscripts, while considerably slower than the others, is still surprisinglygood for the longer arrays.
- The *array* program and *newmat* are similar for lengths 2000 or higher (the longer times for the array program for the longest arrays shown on the graph are probablya quirk of the timing program).

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.

5.2 Matrix vs array library

next - skip - up - start

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. Here are some contrasting properties.

Feature	**Matrix library**	**Array library**
Expressions	Matrix expressions; * means matrix multiply; inverse function	Arithmetic operations, if supported, mean elementwise combination of arrays
Element access	Access to the elements of a matrix	High speed access to elements directly and perhaps with iterators
Elementary functions	For example: determinant, trace	Matrix multiplication as a function
Advanced functions	For example: eigenvalue analysis	
Element types	Real and possiblycomplex	Wide range - real, integer, string etc
Types	Rectangular, symmetric, diagonal, etc	One, two and three dimensionalarrays, at least

Both types of library need to support access to sub-matrices or sub-arrays, have good efficiency and storage management, and graceful exit for errors. In both cases, we probably need two versions, one optimised for large matrices or arrays and one for small matrices or arrays.

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

5.3 Design questions

next - skip - up - start

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.

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.

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.

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 multi-stage QR decompositions.

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.

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()`.

I am now switching to using lowercase for functions with individual words separated by "_". This is following the convention in the standard library and I think it looks neater. Class names will remain with individual words being capitalised.

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.

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.

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.

5.4 Data storage

next - skip - up - start

The stack and heap

To understand how *newmat* stores matrices you need to know a little bit about the *heap* and *stack*.

The data values of variables or objects in a C++ program are stored in either of two sections of memory called the *stack* and the *heap*. Sometimes there is more than one *heap* to cater for different sized variables.

If you declare an *automatic* variable

```
int x;
```

then the value of *x* is stored on the *stack*. As you declare more variables the stack gets bigger. When you exit a block (i.e a section of code delimited by curly brackets *{...}*) the memory used by the automatic variables declared in the block is released and the *stack* shrinks.

When you declare a variable with *new*, for example,

```
int* y = new int;
```

the pointer *y* is stored on the *stack* but the value it is pointing to is stored on the *heap*. Memory on the *heap* is not released until the program explicitly does this with a *delete* statement

```
delete y;
```

or the program exits.

On the *stack*, variables and objects are is always added to the end of the *stack* and are removed in the reverse order to that in which they are added - that is the last on will be the first off. This is not the case with the *heap*, where the variables and objects can be removed in any order. So one can get alternating pieces of used and unused memory. When a new variable or object is declared on the *heap* the system needs to search for piece of unused memory large enough to hold it. This means that storing on the *heap* will usually be a slower process than storing on the *stack*. There is also likely to be waste space on the *heap* because of gaps between the used blocks of memory that are too small for the next object you want to store on the *heap*. There is also the possibility of wasting space if you forget to remove a variable or object on the *heap* even though you have finished using it. However, the *stack* is usually limited to holding small objects with size known at compile time. Large objects, objects whose size you don't know at compile time, and objects that you want to persist after the end of the block need to be stored on the *heap*.

In C++, the *constructor*/*destructor* system enables one to build complicated objects such as matrices that behave as automatic variables stored on the *stack*, so the programmer doesn't have to worry about deleting them at the end of the block, but which really utilise the *heap* for storing their data.

Structure of matrix objects

Each matrix object contains the basic information such as the number of rows and columns, the amount of memory used, a status variable and a pointer to the data array which is on the heap. So if you declare a matrix

```
Matrix A(1000,1000);
```

there is an small amount of memory used on the stack for storing the numbers of rows and columns, the amount of memory used, the status variable and the pointer together with 1,000,000 *Real* locations stored on the heap. When you exit the block in which *A* is declared, the heap memory used by *A* is automatically returned to the system, as well as the memory used on the stack.

Of course, if you use new to declare a matrix

```
Matrix* B = new Matrix(1000,1000);
```

both the information about the size and the actual data are stored on heap and not deleted until the program exits or you do an explicit delete:

```
delete B;
```

If you carry out an assignment with `=` or `<<` or do a `resize()` the data array currently associated with a matrix is destroyed and a new array generated. For example

```
Matrix A(1000,1000);
Matrix B(50, 50);
... put some values in B
A = B;
```

At the last step the heap memory associated with *A* is returned to the system and a new block of heap memory is assigned to contain the new values. This happens even if there is no change in the amount of memory required.

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. [The 16 bit PC problem has largely gone away but it was a problem when much of *newmat* was written. Now, occasionally I run into the 32 bit PC problem.]

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.

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.

5.5 Memory management - reference counting or status variable?

next - skip - up - start

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:

```
Matrix
  |
  +------> Array Object
  |          |
  |          +------> Data array
  |          |
  |          +------- Counter
  |
  +------ Dimension information
```

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 a 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.

5.6 Memory management - accessing contiguous locations

next - skip - up - start

Modern computers work faster if one accesses memory by running through contiguous locations rather than by jumping around all over the place. Newmat stores matrices by rows so that algorithms that access memory by running along rows will tend to work faster than one that runs down columns. A number of the algorithms used in *Newmat* were developed before this was an issue and so are not as efficient as possible.

I have gradually upgrading the algorithms to access memory by rows. The following table shows the current status of this process.

Function	Contiguous memory access	Comment
Add, subtract	Yes	
Multiply	Yes	
Concatenate	Yes	
Transpose	No	
Invert and solve	Yes	Mostly
Cholesky	Yes	
QRZ, QRZT	Yes	
updateQRZ, updateQRZT	Partially	
SVD	No	
Jacobi	No	Not an issue; used only for smaller matrices
Eigenvalues	No	
Sort	Yes	Quick-sort is naturallygood
FFT	?	Could be improved?

This is now all rather out of date. With Pentiums, at least, the important requirement for speed seems to be to minimise transfers between the RAM memory and the on-chip memory. There isn't much you can do about add and subtract, but there lots of possibilities for some of the other operations.

5.7 Evaluation of expressions - lazy evaluation

next - skip - up - start

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;
```

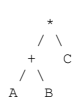
where `i()` denotes inverse. Numerical analysts know it is inefficient to evaluate this expression by carrying out the inverse operation and then the multiply. Yet it is a convenient way of writing the instruction. It would be helpful if the program recognised this expression and carried out the more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

I regard this interpretation of `A.i() * B` as just providing a convenient notation. The objective is not primarily 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.

The C++ standard (section 12.8/15) allows the compiler to optimise away the copy when returning an object from a function (but there will still be one copy is an assignment (=) is involved). This means special handling of returns from a function is less important when a modern optimising compiler is being used.

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



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.

5.8 How to overcome an explosion in number of operations

next - skip - up - start

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.

5.9 Destruction of temporaries

next - skip - up - start

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. Early versions of Gnu C++ destroyed temporary structures as soon as the function that accesses them finished. To overcome this problem, there was an option to store the temporaries forming the tree structure on the heap (created with new) and to delete them explicitly. Now that the C++ standards committee has said that temporary structures should not be destroyed before a statement finishes, I have deleted this option.

5.10 A calculus of matrix types

next - skip - up - start

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 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 program.

5.11 Pointer arithmetic

next - skip - up - start

Suppose you do something like

```
int* y = new int[100];
y += 200;          // y points to something outside the array
// y is never accessed
```

Then the standard says that the behaviour of the program is *undefined* even if `y` is never accessed. (You are allowed to calculate a pointer value one location beyond the end of the array). In practice, a program like this does not cause any problems with any compiler I have come across and no-one has reported any such problems to me.

However, this *error* is detected by Borland's *Code Guard* bound's checker and this makes it very difficult to use this to use *Code Guard* to detect other problems since the output is swamped by reports of this *error*.

Now consider

```
int* y = new int[100];
y += 200;          // y points to something outside the array
y -= 150;          // y points to something inside the array
// y is accessed
```

Again this is not strictly correct but does not seem to cause a problem. But it is much more doubtful than the previous example.

I removed most instances of the second version of the problem from Newmat09. Hopefully the remainder of these instances were removed from Newmat10. In addition, most instances of the first version of the problem have also been fixed.

There is one exception. The interface to the Numerical Recipes in C does still contain the second version of the problem. This is inevitable because of the way Numerical Recipes in C stores vectors and matrices. If you are running the test program with a bounds checking program, edit `tmt.h` to disable the testing of the NRIC interface.

The rule does does cause a problem for authors of matrix and multidimensional array packages. If we want to run down a column of a matrix we would like to do something like

```
// set values of column 1
Matrix A;
... set dimensions and put values in A
Real* a = A.data();               // points to first element
int nr = A.nrows();                // number of rows
int nc = A.ncols();                // number of columns
while (nr--)
{
   *a = something to put in first element of row
   a += nc;                        // jump to next element of column
}
```

If the matrix has more than one column the last execution of `a += nc;` will run off the end of the space allocated to the matrix and we'll get a bounds error report.

Instead we have to use a program like

```
// set values of column 1
Matrix A;
... set dimensions and put values in A
Real* a = A.data();               // points to first element
int nr = A.nrows();                // number of rows
int nc = A.ncols();                // number of columns
if (nr != 0)
{
   for(;;)
   {
      *a = something to put in first element of row
      if (!(--nr)) break;
      a += nc;                     // jump to next element of column
   }
}
```

which is more complicated and consequently introduces more chance of error.

5.12 Error handling

next - skip - up - start

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. In 1997 compiler supported exceptions seemed to work on a variety of compilers - but not all compilers. This was still true in 2001. One compiler company was still having problems in 2003 (not sure about 2004). Try using the compiler supported exceptions if you have a recent compiler, but if you are getting strange crashes or errors try going back to my simulated exceptions.

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.

5.13 Sparse matrices

next - skip - up - start

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.

5.14 Complex matrices

next - skip - up - start

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 essentially twice as big as you would need with a genuine complex matrix library.

6. Function summary

next - skip - up - start

This section lists member and global functions for matrices defined in *newmat.h*. Where there are alternative names the lower-case non-capitalised versions are the preferred ones.

6.1 Member functions for matrices and matrix expressions

next - skip - up - start

Member functions for matrices and matrix expressions. These do not apply to *CroutMatrix* and *BandLUMatrix* unless explicitly noted.

Function group	function name	description
Unary operators (see also operators)	.**t**()	matrix transpose
	.**reverse**() .Reverse()	reverse order of elements (not band matrices)
	.**i**()	invert matrix or solve (also works with CroutMatrixand BandLUMatrix)
	.**sum_rows**()	sum elements in each row
	.**sum_columns**()	sum elements in each column
	.**sum_square_rows**()	sum squares of elements in each row
	.**sum_square_columns**()	sum squares of elements in each column
Change type	.**as_row**() .AsRow()	interpret matrix body as a single row
	.**as_column**() .AsColumn()	interpret matrix body as a single column
	.**as_diagonal**() .AsDiagonal()	interpret matrix body as a diagonalmatrix
	.**as_matrix**() .AsMatrix()	interpret matrix body as a rectangularmatrix
	.**as_scalar**() .AsScalar()	convert 1x1 to Real
Submatrices	.**submatrix**(int,int,int,int) .SubMatrix(int,int,int,int)	submatrix
	.**sym_submatrix**(int,int) .SymSubMatrix(int,int)	submatrix with same row and column range
	.**row**(int) .Row(int)	select a row of a matrix
	.**rows**(int,int) .Rows(int,int)	select a range of rows of a matrix
	.**column**(int) .Column(int,int)	select a column of a matrix
	.**columns**(int) .Columns(int,int)	select a range of columns of a matrix

Scalar functions- maxima & minima (also globalversions of maximum(), minimum(), maximum_absolute_value(), mimimum_absolute_value())	**.maximum_absolute_value**() .MaximumAbsoluteValue()	maximum absolutevalue of elements
	.maximum_absolute_value1(int&) .MaximumAbsoluteValue1(int&)	maximum absolutevalue , return location
	.maximum_absolute_value2(int&,int&) .MaximumAbsoluteValue2(int&,int&)	maximum absolutevalue , return location
	.minimum_absolute_value() .MinimumAbsoluteValue()	minimum absolutevalue of elements
	.minimum_absolute_value1(int&) .MinimumAbsoluteValue1(int&)	minimum absolutevalue , return location
	.minimum_absolute_value2(int&,int&) .MinimumAbsoluteValue2(int&,int&)	minimum absolutevalue , return location
	.maximum() .Maximum()	maximum value of elements
	.maximum1(int&) .Maximum1(int&)	maximum value, return location
	.maximum2(int&,int&) .Maximum2(int&,int&)	maximum value, return location
	.minimum() .Minimum()	minimum value of elements
	.minimum1(int&) .Minimum1(int&)	minimum value, return location
	.minimum2(int&,int&) .Minimum2(int&,int&)	minimum value, return location
Scalar functions- numerical (also globalversions of these functions)	**.log_determinant**() .LogDeterminant()	natural logarithm of the determinant(also works with CroutMatrixand BandLUMatrix)
	.determinant() .Determinant()	determinant(also works with CroutMatrixand BandLUMatrix)
	.sum_square() .SumSquare()	sum of squares of elements
	.norm_Frobenius() .norm_frobenius() .NormFrobenius()	square root of sum of squares of elements
	.sum_absolute_value() .SumAbsoluteValue()	sum of absolutevalues of elements
	.sum() .Sum()	sum of elements
	.trace() .Trace()	trace of a matrix
	.norm1() .Norm1()	maximum of sum of absolutevalues of elements of a column
	.norm_infinity() .NormInfinity()	maximum of sum of absolutevalues of elements of a row
Scalar functions- size and shape	**.bandwidth**() .BandWidth()	bandwidthof matrix (also works with CroutMatrix and BandLUMatrix)

6.2 Member functions for matrices

next - skip - up - start

Member functions for matrices but not matrix expressions. These do not apply to CroutMatrix and BandLUMatrix unless explicitly noted.

Function group	function name	description
Element access (see also operators)	**.element**(int,int)	access element - subscripts start at 0
	.element(int)	access element - subscripts start at 0

Copying	**.inject**(const GeneralMatrix&) .Inject(const GeneralMatrix&)	copy elements into a matrix
	.swap(Matrix&)	swap bodies of two matrices of same type (also globalversion , also works with CroutMatrixand BandLUMatrix)
Scalar functions- size and shape (also work with CroutMatrixand BandLUMatrix)	**.type**() .Type()	type of a matrix
	.nrows() .Nrows()	number of rows
	.ncols() .Ncols()	number of columns
	.size() .Storage()	number of stored elements (including unused elements in band matrices)
	.size2()	size of second array (BandLUMatrixonly)
	.data() .Store()	pointer to stored elements
	.const_data()	constantpointer to stored elements
	.const_data2()	constantpointer to second array (BandLUMatrix only)
	.const_data_indx()	constantpointer to row swap array (CroutMatrix and BandLUMatrixonly)
Scalar functions- numerical	**.is_zero**() .IsZero()	test all elements are exactly zero (also global version)
	.is_singular() .IsSingular()	test for exact singularity(CroutMatrixand BandLUMatrixonly)
	.even_exchanges()	true if there have been an even number of row exchanges(CroutMatrixand BandLUMatrixonly)
Memory management	**.release**() .Release()	release memory after next operation
	.release(int) .Release(int)	release memory after specified number of operations
	.release_and_delete() .ReleaseAndDelete()	delete after next operation
	.for_return() .ForReturn()	*place in an envelope* for efficient return from a function
Change dimensions	**.resize**(int) .ReSize(int)	change the dimensions(vectors and square matrices)
	.resize(int,int) .ReSize(int,int)	change the dimensions(non -square matrices, triangular band matrices and symmetric band matrices)
	.resize(int,int,int) .ReSize(int,int,int)	change the dimensions(band matrices)
	.resize(const GeneralMatrix&) .ReSize(const GeneralMatrix&)	change dimensionsto match those of another matrix
	.cleanup() .CleanUp()	resize to 0x0
	.resize_keep(int)	change the dimensions, keep values (vectors and square matrices, not band)
	.resize_keep(int,int)	change the dimensions, keep values (non-square matrices)

6.3 Operators

next - skip - up - start

Operators for matrices and matrix expressions. These do not apply to CroutMatrix and BandLUMatrix unless explicitly noted.

Function group	function name	description
Element access (matrices only, not functionsof a matrix)	()	access element - subscripts start at 1
	()	access element - subscripts start at 1
	[]	access element C style - subscripts start at zero; if *SETUP_C_SUBSCRIPTS* is defined.
Unary operators	-	change sign of elements
Binary operators	+, +=	add matrices
	-, -=	subtract matrices
	*, *=	matrix multiplication
	|, |=	horizontalconcatenation
	&, &=	vertical stacking
	==	test for *exact equality* (also works with CroutMatrixand BandLUMatrix)
	!=	test for inequality (i.e. not *exact equality*, also works with CroutMatrixand BandLUMatrix)
Matrix and scalar	+, +=	add *Real* to matrix
	-, -=	subtract *Real* from matrix; subtract matrix from *Real*
	*, *=	multiplymatrix by *Real*
	/, /=	dividematrix by *Real*
Copying	=	copy matrix (error if there is loss of data, also works with CroutMatrixand BandLUMatrix)
	=	copy *Real* to all elements
	<<	copy matrix (no error if there is loss of data)
Enter values	<<	enter list of values into matrix
Output (header in *newmatio.h*)	<<	print matrix to file

6.4 Global functions - newmat.h

next - skip - up - start

Operators for matrices and matrix expressions. These do not apply to CroutMatrix and BandLUMatrix.

Function group	function name	description
Binary operators (see also operators)	**SP**(const BaseMatrix&, const BaseMatrix&)	element-wise product of two matrices
	KP(const BaseMatrix&, const BaseMatrix&)	Kronecker product of two matrices
	crossproduct(const Matrix&, const Matrix&) CrossProduct(const Matrix&, const Matrix&)	cross product of two 3x1 or 1x3 matrices or vectors.
	crossproduct_rows(const Matrix&, const Matrix&) CrossProductRows(const Matrix&, const Matrix&)	row-wise cross product
	crossproduct_columns(const Matrix&, const Matrix&) CrossProductColumns(const Matrix&, const Matrix&)	column-wise cross product
Scalar functions- numerical	**dotproduct**(const Matrix&, const Matrix&) DotProduct(const Matrix&, const Matrix&)	dot product of two vectors

6.5 Global functions - newmatap.h

next - skip - up - start

Advanced operators for matrices and matrix expressions. These do not apply to CroutMatrix and BandLUMatrix.

Function group	function name	description
QR transform	**QRZT**(Matrix&, LowerTriangularMatrix&)	transposed version of QRZ transform
	QRZT(const Matrix&, Matrix&, Matrix&)	transposed version of QRZ transform - solve part
	QRZT(Matrix&, Matrix&, LowerTriangularMatrix&, Matrix&)	both of previous two lines
	QRZ(Matrix&, UpperTriangularMatrix&)	QRZ transform
	QRZ(const Matrix&, Matrix&, Matrix&)	QRZ transform - solve part
	QRZ(Matrix&, Matrix&, UpperTriangularMatrix&, Matrix&)	both of previous two lines
	updateQRZT(Matrix&, LowerTriangularMatrix&) UpdateQRZT(Matrix&, LowerTriangularMatrix&)	add extra columns to transposed version of QRZ transform
	updateQRZ(Matrix&, UpperTriangularMatrix&) UpdateQRZ(Matrix&, UpperTriangularMatrix&)	add extra rows to QRZ transform
	updateQRZ(const Matrix&, Matrix&, Matrix&) UpdateQRZ(const Matrix&, Matrix&, Matrix&)	combine the results of the solve parts of QRZ transforms on two blocks of data
	updateQRZ(UpperTriangularMatrix&, UpperTriangularMatrix&) UpdateQRZ(UpperTriangularMatrix&, UpperTriangularMatrix&)	combine the results of QRZ transforms on two blocks of data
	updateQRZ(const UpperTriangularMatrix&, Matrix&, Matrix&) UpdateQRZ(const UpperTriangularMatrix&, Matrix&, Matrix&)	combine the results of the solve parts of QRZ transforms on two blocks of data
Extend orthonormalset	**extend_orthonormal**(Matrix&, int)	extend a set of orthonormalcolumns
Choleskydecomposition	**Cholesky**(const SymmetricMatrix&)	Choleskydecomposition of symmetric matrix
	Cholesky(const SymmetricBandMatrix&)	Choleskydecomposition of symmetric band matrix
Update Choleskydecomposition	**update_Cholesky** (UpperTriangularMatrix&, RowVector) UpdateCholesky (UpperTriangularMatrix&, RowVector)	add extra row to Cholesky/QR decomposition
	downdate_Cholesky (UpperTriangularMatrix&, RowVector) DowndateCholesky (UpperTriangularMatrix&, RowVector)	remove row from Cholesky/QR decomposition
	right_circular_update_Cholesky (UpperTriangularMatrix&, int, int) RightCircularUpdateCholesky (UpperTriangularMatrix&, int, int)	rearrange columns in Cholesky/QR decomposition
	left_circular_update_Cholesky (UpperTriangularMatrix&, int, int) LeftCircularUpdateCholesky (UpperTriangularMatrix&, int, int)	rearrange columns in Cholesky/QR decomposition

Singular value decomposition	**SVD**(const Matrix&, DiagonalMatrix&, Matrix&, Matrix&, bool, bool)	singular value decomposition - get U and V
	SVD(const Matrix&, DiagonalMatrix&)	SVD decomposition - get just singular values
	SVD(const Matrix& A, DiagonalMatrix& D, Matrix&, bool)	SVD decomposition - get U
Eigenvalue decomposition of a symmetric matrix	**Jacobi**(const SymmetricMatrix&, DiagonalMatrix&)	Jacobi eigenvalue decomposition - get only eigenvalues
	Jacobi(const SymmetricMatrix&, DiagonalMatrix&, SymmetricMatrix&)	Jacobi eigenvalue decomposition - get only eigenvalues
	Jacobi(const SymmetricMatrix&, DiagonalMatrix&, Matrix&)	Jacobi eigenvalue decomposition - get eigenvalues and eigenvectors
	Jacobi(const SymmetricMatrix&, DiagonalMatrix&, SymmetricMatrix&, Matrix&, bool)	Jacobi eigenvalue decomposition - get eigenvalues and eigenvectors
	eigenvalues(const SymmetricMatrix&, DiagonalMatrix&) EigenValues(const SymmetricMatrix&, DiagonalMatrix&)	Householdereigenvalue decomposition - get only eigenvalues
	eigenvalues(const SymmetricMatrix&, DiagonalMatrix&, SymmetricMatrix&) EigenValues(const SymmetricMatrix&, DiagonalMatrix&, SymmetricMatrix&)	Householdereigenvalue decomposition with back transform - get only eigenvalues
	eigenvalues(const SymmetricMatrix&, DiagonalMatrix&, Matrix&) EigenValues(const SymmetricMatrix&, DiagonalMatrix&, Matrix&)	Householdereigenvalue decomposition - get eigenvalues and eigenvectors
Sorting	**sort_ascending**(GeneralMatrix&) SortAscending(GeneralMatrix&)	ascendingsort
	sort_descending(GeneralMatrix&) SortDescending(GeneralMatrix&)	descending sort
Fast Fourier transform	**FFT**(const ColumnVector&, const ColumnVector&, ColumnVector&, ColumnVector&)	fast Fourier transform
	FFTI(const ColumnVector&, const ColumnVector&, ColumnVector&, ColumnVector&)	fast Fourier transform - inverse
	RealFFT(const ColumnVector&, ColumnVector&, ColumnVector&)	FFT of real vector
	RealFFTI(const ColumnVector&, const ColumnVector&, ColumnVector&)	FFT of real vector - inverse
	FFT2(const Matrix& U, const Matrix& V, Matrix& X, Matrix& Y)	two dimensionalFFT
	FFT2I(const Matrix& U, const Matrix& V, Matrix& X, Matrix& Y)	two dimensionalFFT - inverse

Fast trigonometrictransform	**DCT_II**(const ColumnVector&, ColumnVector&)	type II discrete cosine transform
	DCT_II_inverse(const ColumnVector&, ColumnVector&)	type II discrete cosine transform - inverse
	DST_II(const ColumnVector&, ColumnVector&)	type II discrete sine transform
	DST_II_inverse(const ColumnVector&, ColumnVector&)	type II discrete sine transform - inverse
	DCT(const ColumnVector&, ColumnVector&)	discrete cosine transform
	DCT_inverse(const ColumnVector&, ColumnVector&)	discrete cosine transform - inverse
	DST(const ColumnVector&, ColumnVector&)	discrete sine transform
	DST_inverse(const ColumnVector&, ColumnVector&)	discrete sine transform - inverse
Helmert transform	**Helmert**(int, bool=false)	return Helmert transform matrix
	Helmert(const ColumnVector&, bool=false) **Helmert**(const Matrix&, bool=false)	multiplyby Helmert transform matrix
	Helmert(int, int, bool=false)	return column of Helmert transform matrix
	Helmert_transpose(const ColumnVector&, bool=false) **Helmert_transpose**(const Matrix&, bool=false)	multiplyby transpose of Helmert transform matrix
	Helmert_transpose(const ColumnVector&, int, bool=false)	multiplyby transpose of Helmert transform matrix, return one element of result

6.6 Other classes - member functions

next - skip - up - start

Class	function name	description
LogAndSign	.**pow_eq**(int) .PowEq(int)	raise to a power
	.**change_sign**() .ChangeSign()	change sign
	.**log_value**() .LogValue()	return the natural logarithm of the value
	.**sign**() .Sign()	return the sign
	.**value**() .Value()	return the value (no log transform)
MatrixType	.**value**() .Value()	return the value (as character string)
	.**is_diagonal**()	has diagonalattribute
	.**is_symmetric**()	has symmetric attribute
	.**is_band**()	has band attribute
MatrixBandWidth	.**upper**() .Upper()	return upper band width
	.**lower**() .Lower()	return lower bandwidth

SimpleIntArray	**.size**() .Size()	return size of array
	.data() .Data()	return a pointer to the data
	.const_data()	return a constantpointer to the data
	.resize() .Resize()	change the size of an array
	.resize_keep() .resize(true) .Resize(true)	change the size of an array - keep the values
	.cleanup() .CleanUp()	resize to zero

7. Change history

next - skip - up - start

Newmat11 - November, 2008:

Remove work-arounds for older compilers, Borland Builder 6 and Open Watcom compatibility, SquareMatrix, load from array of ints, crossproducts, Choleskyand QRZ update functions, swap functions, FFT2, access to arrays in traditional C functions, SimpleIntArray class, compatibility with Numerical Recipes in C++, sum_rows(), sum_columns(), sum_squares_rows() and sum_squares_columns() functions, extend_orthogonal function, resize_keep function, speed-ups and bug-fixes, change to lower case for functions, can copy CroutMatrix, BandLUMatrix, Helmert transform, compatibility fix for G++ 4.1, start insertingcomments for Doxygen, SP_eq, scientific format for output, fix for 64 bits.

Newmat10A - October, 2002, Newmat10B - January 2005:

Fix error in Kronecker product; fixes for Intel and GCC3 compilers.

Newmat10 - January, 2002:

Improve compatibility with GCC, fix errors in FFT and GenericMatrix, update simulated exceptions, maxima, minima, determinant, dot product and Frobenius norm functions, update make files for CC and GCC, faster FFT, `A.ReSize(B)`, fix pointer arithmetic, << for loadingdata into rows, IdentityMatrix, Kronecker product, sort singular values.

Newmat09 - September, 1997:

Operator ==, !=, +=, -=, *=, /=, |=, &=. Follow new rules for *for (int i; ...)* construct. Change Boolean, TRUE, FALSE to bool, true, false. Change ReDimensionto 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 exceptionmodule to allow globaldeclarationsof 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 exceptionmodule . AdditionalFFT functions . Some minor increases in efficiency. Submatrices can now be used on RHS of =. Option for allowingC type subscripts. Methodfor loadingshort lists of numbers.

Newmat06 - December 1992:

Added band matrices; 'real' changedto '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); optionfor version 2.1 or later. Suffix for includefiles changedto .h; BOOL changedto 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 compilers). 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 conversionof 1x1 matrix , *Numerical Recipes in C* interface, outputoperations , various scalar functions. Improved return from functions. Reorganisedsetting optionsin "include.hxx".

Newmat02 - July 1991:

Version with matrix row/column operationsand numerous additional functions.

Matrix - October 1990:

Early version of package.

8. Problem report form

next - skip - up - start

Copy and paste this to your editor, fill it out and email to robert **at** statsresearch.co.nz

But first look in my web page http://www.robertnz.net to see if the bug has already been reported.

```
Version: ............... newmat 11 (20 November 2008)
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:



------------------------------------------------------------
```