Writing a matrix package in C++

Robert B Davies
16 Gloucester Street, Wellington, New Zealand
internet: robert@statsresearch.co.nz

1 Introduction

Over the last several years I have been writing a library of **C++** classes and functions for manipulating matrices. Versions of this library, now known as *Newmat*, have been available over *internet* and *Compuserve* since 1990. In this article I describe some of the ideas behind the package, things I have learnt, the problems I have encountered, and the extent to which it meets my objectives.

1.1 What is a matrix?

Some people think of any two dimensional array of numbers as a matrix. Others, particularly scientists and engineers, think of a matrix as a two dimensional array on which it is meaningful to carry out certain operations; particularly matrix multiplication. I refer to a library for manipulating the first kind of two (or more) dimensional arrays of numbers as an *array* package and one for manipulating two dimensional arrays where matrix multiplication is a sensible operation as a *matrix* package. *Newmat* is most definitely a *matrix* package.

1.2 What is a matrix package?

By a *matrix package* or *matrix library* I mean an integrated set of classes and functions for manipulating matrices and vectors. (A *vector* is a matrix with either one row or one column). The package will include constructors for a variety of types of matrix, and the usual matrix operations for adding, multiplying, transposing, inverting, solving equations, finding eigenvalues, etc.

Different matrix packages may be built for different purposes. One is unlikely to be able to build a matrix package to meet everyone's requirement, except as an amalgamation of a number of packages. For example, if one is interested in very small matrices for manipulating the points in a three-dimensional image, one will want a package that spends little time on administration, but space will probably not be a problem. One the other hand, if one is dealing with large matrices, one can spend a modest amount of time on administration to conserve space or optimise the evaluation of expressions. With very large matrices one has to worry about the use of virtual memory.

1.3 Why a matrix package?

Matrices are a very natural object in a wide range of applications of numerical mathematics. Often, if you have the standard matrix operations defined as operators and functions, you hardly need look at the individual elements of the matrices. The operations are carried out on the matrices as complete entities. Thus the essential logic of a program using matrices as objects is more transparent and is not obscured by the details of handling the matrices. So there is less opportunity for error; both errors of logic and *silly* errors.

1.4 Why C++?

C++ is the only language that I am familiar with that has the facilities needed for a good matrix package. The memory management in Fortran 77 and Pascal is just too primitive. One of the simple but marvellous things one can do with

a **C**++ matrix package (anyone's **C**++ matrix package) is to write

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

and there is a matrix with its memory allocated. And in most cases the matrix will give up its memory and go away automatically when you are finished with it. You can write routines to set up a matrix in **C** (see Press et al, 1988) but it is rather awkward and you do have to tell the matrix to go away when you are finished with it.

The next great thing is to be able to define $*$, $+$, $-$ to operate directly on matrices. Finally you can define different types of matrix: UpperTriangularMatrix, DiagonalMatrix, etc.

I presume you can do at least some of this in Ada or Fortran 90. But Ada, at least is too big and expensive for my kind of computer, and Fortran 90 was *vapour-ware* when I started this project. I doubt whether either of these languages would give the flexibility of **C**++.

The other possibilities are the special purpose interactive languages such as *Genstat* and *Splus* (for statistics) and *Matlab* (for general numerical work). But my experience has been that while these are fine for many applications, sometimes the interactive languages don't have the necessary flexibility or efficiency and one needs a *real* programming language.

2 Writing the package

2.1 My objectives

I want the matrix package to have the following features:

- A variety of matrix types including rectangular, upper and lower triangular, diagonal, symmetric, row vector, column vector[1];
- Matrix expressions to be entered as closely as possible to the mathematical formulae;
- Efficiency to be similar to that of a program written in **C**;
- Suitable for medium sized matrices (for example, will fit into normal memory);
- Will run on a PC or small work-station;
- Can have complex element and sparse matrix facilities added later.

The variety of matrices reflects the types that arise in the kinds of calculations I need to carry out. This includes Cholesky decomposition, QR decomposition, and eigenvalues of symmetric matrices. I distinguish row vectors, column vectors and diagonal matrices because of the different way they behave in expressions.

2.2 The initial decisions

There are several decisions which need to be made at the outset.

The most important is how the body of the matrix is to be stored—by rows, by columns, or some other way and in one block or several blocks. *Newmat* follows Algol and **C** and stores by row in a single block of memory. The memory, of course, is allocated with a *new* statement by the constructor. In retrospect, storing by row may have been a mistake as it makes it more difficult to access routines written in Fortran which stores by column. In addition, matrix algorithms have been optimised for systems that store by column because of the dominance of Fortran in numerical mathematics.

The next decision is index ranges—start at zero as in **C**; one as in Fortran; or be set by the user as in Algol. In *Newmat* they start at one. In the original version of *Newmat* they started at zero but I found it so confusing that I switched to one in the second version. Some users have suggested that the starting point should be set by the user, say in a global *define* statement. This wouldn't be hard, except that I would have to maintain two test suites, something which I am not enthusiastic about. In most instances where I want a starting value

[1]The current version of *Newmat* also includes band matrices

different from one, it is because I am using *Newmat* as an *array* package rather than a *matrix* package.

Finally there is the question of whether to follow **C**, "[a][b]", or Fortran, "(a,b)", in the way index numbers are expressed. I make no apology for following Fortran. **C**++ does not allow "[a,b]".

Now consider some of the areas of difficulty.

2.3 Expression evaluation

Suppose A, B and C are matrices or vectors of the same type and size and we want our package to calculate $A + B + C$ storing the answer in X. If we coded this directly in **C** our program would make one sweep through the matrices and we would require $3n$ memory reads, n memory writes and $2n$ additions where n is the number of elements in each of our matrices. If we had a simple matrix package and used the code

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

the package might add A and B storing the result in a temporary, add the temporary to C storing the result in another temporary and then copying the temporary into X. There might even be another copy. This process requires $5n$ memory reads, $3n$ memory writes and the $2n$ additions[2]. By using delayed copying with reference counting (see, for example, Hansen, 1990, p 235) one could eliminate the final copy(ies) giving $4n$ memory reads and $2n$ memory writes. *Newmat* does slightly better by including a status variable to show whether a matrix or vector is temporary or non-temporary. This enables it to reuse the temporary resulting from the first addition. This saves a pointer calculation although the number of reads and writes is unchanged. It also enables *Newmat* to destroy temporaries that are not to be recycled as soon as they are finished with.

Now suppose X is a matrix and Y is a vector and I want to calculate the least squares formula $b = (X'X)^{-1}X'Y$. In *Newmat* I would like to code this as

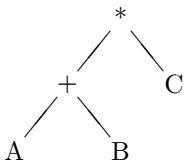
```
b = (X.t() * X).i() * X.t() * Y;
```

Here `.t()` denotes the transpose operation and `.i()` the inverse operation. Several things might go wrong. The transpose of X gets taken (twice), whereas, in fact one should be using an algorithm for multiplying directly by a transpose. A matrix is inverted, which is inefficient when one doesn't need the inverse itself. The middle multiply is carried out before the final one, leading to much extra work. Finally, in many situations this is the wrong way to do the calculation, which is more accurately carried out using a QR decomposition!

I think it is unreasonable to expect a matrix package to recognise the last problem. However, it can do something about the others. *Newmat* would not calculate the inverse explicitly. The transposes would be calculated but it would be a relatively minor extension to *Newmat* to avoid this. It might also be possible to get the multiplies to be evaluated in the most efficient order. The machinery for doing all this is described in the next section.

2.4 Two stage evaluation

Expressions are evaluated in two stages. In the first stage a tree representation of the expression is formed. For example $(A+B)*C$ is represented by the tree:



Rather than adding A and B, the + operator yields an object of class *AddedMatrix* which is just a pair of pointers to A and B. The $*$ operator yields a *MultipliedMatrix* which is a pair of pointers to the *AddedMatrix* object and C. The operator = then examines the tree for any simplifications and then initiates a recursive evaluation of the tree. At the moment, the only

[2]But a matrix package might be able improve the situation, for example, by calling a BLAS routine.

simplification it checks for is whether the object on the left hand side of an = also appears on the right hand side. In this case its memory can be recycled.

Expressions such as `A.i() * X` are detected at compile time. If the $*$ operator has an *InvertedMatrix* as its first argument then it returns a *SolvedMatrix* which solves $A^{-1}X$ directly when evaluated. Similar tricks could be used to recognise multiplication by transposes and possibly to evaluate $A + B + C$ with a single sweep.

The representation of expressions by trees as is used here is well-known to users of Lisp. A similar scheme is used in **C**++ by Van Wyk (1991) for representing simultaneous equations.

2.5 The explosion in the number of operations

Newmat has 7 distinct classes of matrix and I would like to add a few more. This means I would need 49 different versions of each binary operation. In practice, the number could be reduced a little, particularly for +. Nevertheless, I consider this situation impossible. Doug Lea told me it was possible to avoid this problem. I don't know what his method is. Here is mine.

For each type of matrix I provide functions for extracting individual rows or columns. I assume a row or column consists of a sequence of zeros, then a sequence of stored values, then a sequence of zeros. Now only a single algorithm is required for each binary operation. The rows can be located very quickly since most of the matrices are stored by row. Columns must be copied and so access is somewhat slower. As far as possible my algorithms access the matrices by row. For adding and subtracting matrices of the same type I provide a separate algorithm since there is no need to access individual rows.

One of the unexpected complications with the method is that it isn't obvious what the type of the matrix resulting from an operation is. So there has to be a block of code that, in effect, defines an *algebra* of matrix types.

Generally the method works well[3]. I switched to this method of handling the binary operations when I had 5 distinct types. The code file got just a little shorter when I made the switch. Apparently 5 matrix types is about the break-even point. However, it must also be admitted that there is a substantial overhead in this approach if one is working with small matrices. The test program developed for the original version of the package takes 30 to 50% longer to run with the current version of the package. This is for matrices in the range 6×6 to 10×10. To improve the situation a little, I provide a separate multiplication routine for the case where all the matrices are rectangular.

2.6 The internals

Newmat provides a subset of the standard analyses one would expect to find in a matrix package[4]. Most of the routines are translations of the Algol routines in Wilkinson and Reinsch (1971), the starting point for many of the matrix routines in common use today. A few of the routines are adapted from Press et al (1988).

Of course, the interface to these routines is much cleaner, than with the corresponding routines in Fortran. There is no argument for workspace arrays, no arguments for array sizes, and fewer and more intuitive names for the routines.

But I am not happy with the actual implementation of these functions. In most of the functions, access to the bodies of the matrices is via pointers which are incremented as the analysis proceeds. The code tends to be *write only*. In a few of the routines I have tried to make the access more transparent by defining appropriate classes, but I don't think the attempt has been very successful as yet. I think this is a problem area for **C**++. I find it hard to see how the array indexing we are used to in Fortran can be made efficient in **C**++ as one cannot expect the **C**++ compiler to carry out the optimisations the Fortran compiler can. So some alternative way has to be found to write code that is both understandable and efficient.

[3] Symmetric matrices are still a problem since complete rows are not stored explicitly.

[4] In particular eigenvalue analysis of a general matrix is still missing.

Newmat does not use the BLAS, the highly optimised routines for carrying out the basic operations on linear algebra. I don't think there would be any great difficulty in incorporating these into *Newmat* and they may solve some of the problems of the code not being very understandable as well as improving efficiency. The main problem is the non-availability of the BLAS on the machines I have access to and the lack of a standard **C**++ interface. Hopefully the emergence of *Lapack*++ will ease this last problem.

2.7 Testing

I have an extensive test suite of programs which are intended to test every nook and cranny of *Newmat*. I have a series of counters which are activated by means of a *define* statement that test that all the nooks and crannies have been accessed (there are 350 of them). I also have a series of tests for memory leaks and unbalanced *new*s and *delete*s. Of course, all this doesn't test everything but I have had remarkably few error reports. And I know that there are users who do report errors.

2.8 Vital statistics

Newmat has 80 classes, 1920 lines of code (excluding comments) in the *.h* files and 4830 lines in the *.cpp* files. The class diagram of the main class tree is shown in figure 1.

3 The problem areas

3.1 Complex elements

At present *Newmat* allows only one element type. The user can choose *float* or *double*. It doesn't use templates and I don't think templates would fit in well with the two stage method of expression evaluation. In any case, changing from *float* or *double* to *complex* is not a trivial exercise and couldn't be accomplished within the template mechanism. Not having *complex* partly reflects my interests and partly reflects the difficulty in including *complex*. Again one has the explosion in the number of operations. As well as *real* and *complex* types I am probably going to want a partial implementation of an *imaginary* type. This gets us back to the problem of an explosion in the number of versions of binary operations. It won't be as bad as nine versions of each binary operation but it might be worse than four. This might be acceptable, but I would like to have a *sparse* matrix class as well and this tends to interact multiplicatively with the *real–complex* problem. Even just implementing *complex* might make *Newmat* too large for PCs running DOS without using overlays or extenders.

3.2 C++ problems

There are a few areas where **C**++ causes problems. I have already mentioned the problems with the efficient access of elements in multidimensional arrays.

Another problem area is when wants to return a matrix from a function via the *return* statement. **C**++ insists on copying objects when they are returned and it has been quite difficult to defeat this and my solution is awkward. This is one situation where delayed copying with reference counting would do a better job than the status variable system used by *Newmat*. It is a particular problem with the *Gnu* compiler which likes to destroy temporaries before you have had time to use them.

Another area of annoyance is how to initialise a matrix with a set of values. Something one would do with a *data* statement in Fortran. *Newmat* has a couple of facilities for doing this, but neither are really satisfactory.

A problem that is rapidly going away is what to do about errors. The **C**++ *exception* mechanism does seem to provide an adequate way of handling errors. *Newmat* includes a mechanism for simulating the exception mechanism and the next version gives you the option of using the simulated ones or the ones provided by the compiler. I want *Newmat* to be useable in interactive programs and this required a way of handling errors that does not lead to immediate termination of the program. An earlier attempt to do this without using the exception mecha-

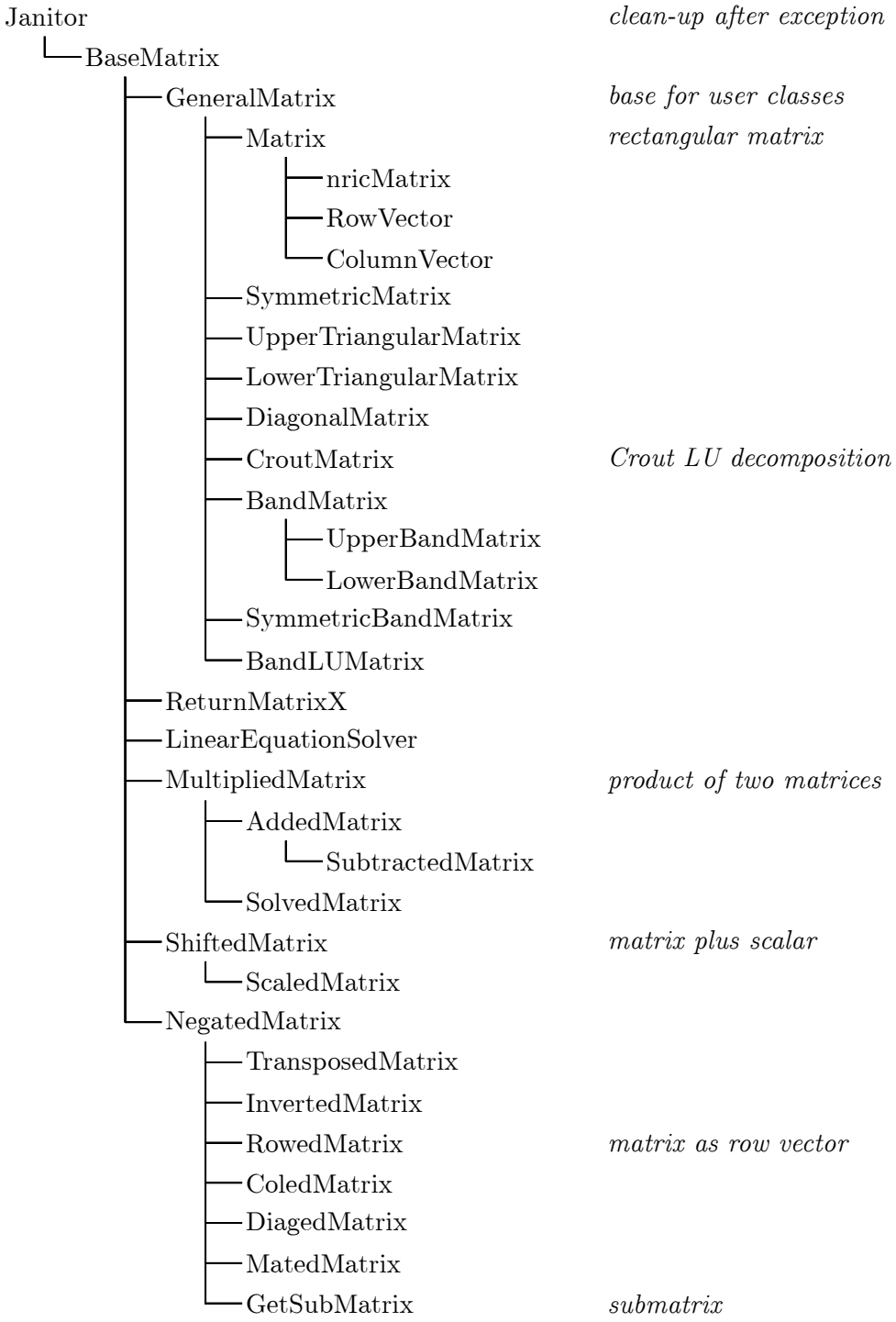


Figure 1: Main class tree

nism turned out to be hopelessly complicated and unworkable. For the record, there are 114 calls to *throw* in *Newmat* excluding out-of-space exceptions and rethrows. So people who think library classes should avoid exceptions are going to be disappointed.

One thing that would help error reporting is if **C**++ could identify a matrix's identifier. If, for example, you try to use an out-of-range index, *Newmat* will tell you the size and type of the matrix concerned, but it can't tell you its name.

4 Uses, users and evaluation

4.1 Uses and users

My normal work is research and consulting in Statistics. I use *Newmat* for research work into the properties of statistical tests and estimators for unusual situations. An example of where I would have used *Newmat*, had it been available, is Davies and Harte (1987) which compares tests for long range dependence in time-series. However, *Newmat* would have to be vastly extended before it was useful for doing real statistical analyses, that I currently use *Splus* for.

I don't have much information on other users of *Newmat* as I hear from them mainly when things go wrong. Many are relatively new users of **C**++ and are PhD students. Topics are solutions of differential equations using finite differences or finite elements and statistics. The most usual compilers are *Gnu* and *Borland*.

4.2 Evaluation

Newmat certainly works, I believe with reasonable efficiency in most cases, although I have not carried out a lot of tests. In particular, it is much much nicer to use than matrix libraries in Fortran.

It is vastly more complicated than I had anticipated. I still haven't decided whether the two-stage evaluation scheme is really worthwhile. It is so much built into *Newmat* that it would be very difficult to remove to see what difference it made to the performance or flexibility. Possibly I have taken the two-stage evaluation further than is really appropriate. Otherwise I am inclined to think the complexity is due to the nature of matrices rather than problems with *Newmat* or **C**++. I am interested to see that *Lapack*++ is also filling up a lot of lines of code.

Persistent storage is not implemented, but few people have complained about that.

4.3 What's next

I would still like to get complex and sparse matrices into the package. I would also like to add a moderately simple array class. The eigenvalue routines need to be brought up-to-date and possibly I will be able to use the routines from *Lapack*++. Otherwise I am looking at applications of the package rather than extensions, particularly ones relevant to my own area of work.

References

Davies, R.B. and Harte, D.S. (1987). Tests for Hurst effect. *Biometrika* 74, 95-101.

Hansen, T.L. (1990). *The* **C**++ *Answer Book.* Addison-Wesley, New York.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. (1988). *Numerical Recipes in C.* Cambridge University Press, Cambridge.

Van Wyk, C. J. (1991). A Class Library for Solving Simultaneous Equations. *Usenix* **C**++ *Conference Proceedings,* 229-234. Usenix Association, Berkeley.

Wilkinson, J.H. and Reinsch, C. (1971). *Handbook for Automatic Computation, Volume II: Linear Algebra.* Springer-Verlag, Berlin.