Developing a Matrix Library in C
An exercise in software design
(SML version 1.4)

Bahjat F. Qaqish
(bahjat_qaqish@unc.edu)
The University of North Carolina at Chapel Hill

Introduction

This document describes the design and implementation of **SML**, a C library for matrix computations. SML stands for *Small Matrix Library*. We'll discuss some of the implementation details, explain why certain decisions were made and how C language facilities were used. This document can be viewed as a hands-on tutorial on software design.

Even though SML is fairly small (compiles in a few seconds) it still provides a wide range of matrix operations including Cholesky, QR, singular value (SVD) and eigenvalue decompositions. SML is extremely flexible. It can be configured to choose: matrix element type to be `float`, `double`, or `long double` (or even other types if needed); memory layout to be by-row or by-column; matrix access to be by macros or by function calls with or without index range checking. SML offers a special storage mode that is compatible with the *Template Numerical Toolkit* (http://math.nist.gov/tnt) and the book *Numerical Recipes*.

Who is the target audience? This document will be most useful to those with a working knowledge of C but haven't tackled any fairly complex programming task, and specifically haven't developed any software libraries. This audience includes numerical analysts, mathematicians and statisticians who have learned C basics and were wondering about how to develop software for matrix computations.

Background: Matrices

Those familiar with matrices can skip this section. For those who are not, this is a minimal description; skimming through an elementary text is highly recommended.

A matrix can be described mathematically in many different ways. Here, we'll think of it in a practical way as a two-dimensional array of numbers. If the array has 5 rows and 3 columns we say that it represents a 5×3 matrix. If we call such a matrix A, then a_{23} denotes the entry in row 2, column 3. Generally, if A contains m rows and n columns, we say that it is an $m \times n$ matrix. and a_{ij} is the entry in row i, column j. Note that rows and columns are numbered starting at 1, not 0.

A special form of matrices are those with only one column, called column vectors, or only one row, called row vectors.

Example operations on matrices: Suppose A and B are 5×3 matrices, C a 3×8 matrix, x a 3×1 (column) vector and y a 1×3 (row) vector. The following are example operations and their results:
$A + B$ produces a 5×3 matrix,
$A * C$ produces a 5×8 matrix,
$A * x$ produces a 5×1 matrix (i.e. a column vector),

$y * C$ produces a 1×8 matrix (i.e. a row vector),
$x * y$ produces a 3×3 matrix,
$y * x$ produces a single number.

What we can see from the above is that software for matrix computations must be able to deal with matrices of different dimensions. Even the same matrix may change its dimensions during execution of a program; the operation $A \leftarrow A * C$ changes the dimensions of A from 5×3 to 5×8. Should the programmer be responsible for keeping track of matrix dimensions as they change from one statement to the next? Preferably not, since that is an inherently error-prone process. Matrix dimensions should be updated transparently; matrices should "know" their dimensions.

Another point is whether we should distinguish between vectors and matrices, or rather treat vectors as matrices that happen to have only one row (row vectors) or only one column (column vectors). It is certainly easier to treat both as matrices. This way we won't have to keep track of which variables are matrices and which are vectors. Another benefit is fewer functions; we won't need one function for matrix times matrix and another for matrix times vector multiplication. Looking at the matrix operations listed above, a single matrix multiply function will handle $A * C, A * x, y * C, x * y$, and $y * x$. There is no need for several variants of what is essentially the same function.

Background: Arrays in C

Lets briefly go over what the C language has to offer in terms of arrays.

The one-dimensional array defined as

```
double  x[20];
```

consists of elements `x[0], x[1], ..., x[19]`. C arrays are 0-based, so `x[]` starts at `x[0]`, not `x[1]`. In mathematics, vectors and matrices are traditionally indexed starting at 1, not 0. We would like our matrix library to follow that convention. There are the occasional cases when 0-based indexing is more convenient. Our approach is to provide both modes of indexing.

The C language offers 2-dimensional arrays as follows:

```
double  A[3][5];
double  B[][5];
```

Now `A` has elements `A[0][0], ..., A[0][4], A[1][0], ..., A[1][4], A[2][0], ..., A[2][4]`. Again, row and column indices are 0-based. In C, 2-dimensional arrays are stored in memory "by row", i.e. first row, second row, third row, and so on. This is also called "row-major" order. The array `B` declared above has an unspecified number of rows, but each row contains 5 columns. The number of rows is "flexible", but the number of columns is fixed at 5 and can't be changed. Essentially, `B` is an array of rows.

Array `A` above can be used to represent a 3×5 matrix. Such a matrix can also be represented by an array of 3 rows, 5 elements each, as follows:

```
typedef double VECTOR [5];    /* row vector */
typedef VECTOR MATRIX [3];    /* array of rows */
MATRIX A;
```

Now a_{ij} is in `A[i-1][j-1]`.

Alternatively, a 3×5 matrix can be represented by an array of 5 columns, 3 elements each:

```
typedef double VECTOR [3];    /* column vector */
typedef VECTOR MATRIX [5];    /* array of columns */
```

and now a_{ij} is in `A[j-1][i-1]`.

Most matrix code written in FORTRAN is developed with "by column" or "column-major" array access in mind (to benefit from memory caches). The last implementation above "matrix = array of columns" is particularly suited to direct FORTRAN-to-C ports if the by-column memory access advantage is to be maintained. However, keep in mind that array indexing must be reversed (and 0-based), a_{ij} is in `A[j-1][i-1]`, not `A[i][j]`. We'll revisit this issue later in the design of SML.

A few words about some existing software. The GNU Scientific Library (GSL) contains elegant data structures and a large suit of matrix routines. However, there are two things I don't like about GSL. First, row and column indexing is 0-based, which is completely unnatural in matrix computations. Second, vectors and matrices are treated as distinct entities. I prefer to handle vectors as matrices that happen to consist of only one column or one row.

The wonderful LAPACK is a well-organized and highly-polished library. However, it doesn't offer memory and matrix-dimension management routines. Using SML to do the matrix-management and LAPACK and GSL to do the computations would be a nice combination. Wrapper functions can be used to interface SML to LAPACK and GSL. However, we will not go that far in this short introduction.

An approach used in the Template Numerical Toolkit (TNT) (http://math.nist.gov/tnt) and in the book Numerical Recipes (NR) is to setup an $m \times n$ matrix as an array of m pointers, each pointing to a row that contains n values. An advantage is that we can access a_{ij} as `A[i][j]`, which is certainly attractive. A disadvantage is that, if one works with matrices that consist of one long column, a lot of memory will be used to hold row pointers, each of which pointing to a single element. I frequently work with such matrices (one long column). Another disadvantage is that the programmer has to keep track of matrix dimensions. SML allows the user to choose this storage scheme, and even goes further by allowing both row-major and column-major storage within this scheme. We'll revisit this approach later in this document and show how to call NR code from SML.

SML Design Goals

Who are the potential users of SML? My intention is that this library is for programmers who want to write their own matrix computation routines, but do not want to design and develop their own matrix data structures, memory management and input/output. The main motivation is flexibility. High-performance is a second, but important, motivation.

SML design goals:
Well, before we set out to write any code, we have to lay down our objectives or design goals. Lets start with these:

- Our library is aimed at dense (non-sparse) matrices. However, we'll see that it can be extended to allow other types of matrices.
- Simple access to matrix contents.
- Vectors will be handled as matrices, nothing special for vectors.
- Indexing will be 1-based; the first row (or column) is row (or column) 1. However, 0-based versions of the accessor functions will be provided.
- The potential for change in the details of matrix storage should be allowed (dense, diagonal, banded, triangular, sparse). Another aspect of storage is whether matrix data will live in memory, on disk or other devices. Although the library is written for memory storage, the MATRIX abstraction does not require that.
- Simplified memory management.
- Automatic management of matrix dimensions, so the programmer wouldn't need to keep track of them.
- Some degree of error checking, recovery and control.
- Bounds checking: This is useful for testing and debugging, but it degrades performance. So, we'd like it to be a switchable option.
- By-row or by-column? FORTRAN code optimized for by-column storage will benefit from by-column storage when ported to C. We'll make this a switchable option as well; the user decides on by-column or by-row. The decision will be library-wide as opposed to a per matrix basis.
- double or float? Elements of all matrices will be of only one type; float, double or long double. The choice will be left to the user. This decision also will be library-wide.

Abstraction

An abstract data type `MATRIX` will be defined along with some basic operations on objects of that type. At this point we don't worry about the details of implementing the `MATRIX` object.

Basic services are provided by the following functions:

```
MATRIX  MatDim   (size_t rows, size_t cols);
void    MatReDim (MATRIX A, size_t rows, size_t cols);
void    MatUnDim (MATRIX A);
REAL    MatGet   (MATRIX A, size_t i, size_t j);
REAL    MatSet   (MATRIX A, size_t i, size_t j, REAL x);
size_t  MatRows  (MATRIX A);
size_t  MatCols  (MATRIX A);
```

A matrix is created with `MatDim`. `MatDim` returns a matrix with the specified dimensions (`rows` and `cols`). It allocates all the needed storage. All other functions operate on matrices created by `MatDim`. Function `MatReDim` changes the dimensions of an existing matrix. Functions `MatRows` and `MatCols` return the number of rows and columns of matrix a, respectively. `MatGet` returns the value of an element of a matrix. `MatSet` assigns a value to an element of a matrix. `MatUnDim` frees the memory occupied by a matrix. The type `REAL` is the type of data that matrix objects will hold.

The type `size_t` is an unsigned integral type that is large enough to index any array supported by the implementation. A variable of type `size_t` can be zero or positive, but can't take negative values. This type is used throughout the library to number rows and columns.

Using a matrix in a function or a program involves three steps:

- Allocate the matrix (preferably) the moment it is defined with `MatDim(0, 0)`.
- Use the matrix, possibly resizing it with `MatReDim`.
- Free the matrix with `MatUnDim`.

Example:

```
void example (void)
{
  MATRIX A = MatDim(0, 0);
  MATRIX B = MatDim(0, 0);
  MATRIX C = MatDim(0, 0);


  MatReDim(A, 10, 5);
  MatReDim(B, 10, 5);

  /* ... assign values to A and B ... */

  MatAdd(C, A, B);      /* C <- A + B, this will resize C */

  ...

  MatUnDim (C);
  MatUnDim (B);
  MatUnDim (A);
}
```

Once the basic services listed above are available, matrix computations can be written without any knowledge of how `MATRIX` is actually implemented. Here is an example, matrix addition. For clarity, we omit the error checking code.

```
void MatAdd (MATRIX A, MATRIX B, MATRIX C)      /* A = B + C */
{
```

```
  size_t i, j;

  MatReDim (A, MatRows(B), MatCols(B));

  for (i = 1; i <= MatRows(A); ++i)
    for (j = 1; j <= MatCols(A); ++j)
      MatSet(A, i, j, MatGet(B, i, j) + MatGet(C, i, j) );

}
```

First we make sure that `A` has the right dimensions with `MatRedim`, then we run loops to compute $a_{ij} = b_{ij} + c_{ij}$. Astute programmers may wonder whether a call like `MatAdd(A, A, A)` would work because `MatReDim` may destroy or alter `A` which in this case is also `B` and `C`. The way `MatReDim` is written is the key - nothing is done if the matrix already has the requested dimensions.

Here is another example, `MatCopy`. This is a *deep* copy, it copies one matrix into another. The destination matrix is redimensioned (if necessary).

```
void MatCopy (MATRIX A, MATRIX B)      /* A <- B  */
{
  size_t i, j;

  if (A==B) return;
  MatReDim(A, MatRows(B), MatCols(B));
  for (i = 1; i <= MatRows(A); ++i)
    for (j = 1; j <= MatCols(A); ++j)
      MatSet(A, i, j, MatGet(B, i, j));
}
```

Why not simply write `A = B;`? That would be a *shallow* copy. A `MATRIX` is actually a pointer (details below) and shallow copying creates problematic aliases (for the programmer and for the compiler).

It should be clear now that with the abstract `MATRIX` given above we can proceed to write all sorts of matrix computation code. But, how is `MATRIX` actually implemented? That will be discussed in the next section.

The `MATRIX` data structure

To work with a matrix we need to know its dimensions and its contents. These are the components of the matrix descriptor `MATRIXDESC` structure, and `MATRIX` is a pointer to that structure:

```
typedef struct {
  size_t rows, cols;     /* dimensions */
  REAL* m;               /* pointer to actual contents */
}   MATRIXDESC;          /* matrix descriptor */
typedef MATRIXDESC* MATRIX;
```

The matrix elements will be stored in a 1-dimensional array allocated by `malloc`. Suppose that matrix `A` has 2 rows and 3 columns and `m` is the pointer returned by `malloc(2*3*sizeof(*m))`. Now, a_{11} is in `m[0]`, a_{12} in `m[1]`, a_{13} in `m[2]`, a_{21} in `m[3]`, a_{22} in `m[4]` and a_{23} in `m[5]`. We can calculate the index into `m[]` as follows: a_{ij} is in `m[3*(i-1)+j-1]`. Generally, if a matrix has `r` rows and `c` columns, a_{ij} will be in `m[c*(i-1)+j-1]`. Of course, this is for row-major storage with 1-based indexing. For column-major storage, with 1-based indexing, a_{ij} will be in `m[r*(j-1)+i-1]`.

We `#define SML_BY_ROW` (in file "sml.cfg") to select storage by row or `#define SML_BY_COL` to select storage by column. The effect is library-wide. The whole library must be recompiled if this option is changed.

We `#define SML_AC_MAC` to specify that `MatGet` and `MatSet` are macros, or `#define SML_AC_FUN` to specify that they are functions. These only affect how matrices are accessed, so there is no need to recompile the whole library if this option is changed. Indeed, it is possible to have a set of tested and debugged routines compiled with macro access, but new routines under development compiled with function access. When function access is active, range-checking can be activated by adding a `#define SML_BOUNDS_CHK` to "sml.cfg". This is extremely helpful while debugging.

Whether `MatSet` and `MatGet` are functions or macros, they take care of index calculations. They adapt to by-row or by-column access through `#ifdef` statements.

The `MatGet/MatSet` functions

In SML, the numbering of rows and columns is 1-based. For certain applications, 0-based indexing is more convenient. Further, for index computations, 0-based is actually easier. The approach taken in SML is to provide 1-based indexing as the default in functions (and macros). However, a full complement of accessor functions (and macros) ending with "0" offer 0-based indexing. For example `MatGet0` and `MatSet0` access matrices in a way such that the first row (or column) is row (or column) zero. Thus `MatGet0(A, 2, 5)` is equivalent to `MatGet(A, 3, 6)`. The 0-based scheme is more "natural" in the C language. In fact, many of the SML routines internally employ 0-based loops.

In C, one can write `x += y` using the compound operator "+=" instead of `x = x + y`. Similar functionality is provided for matrix elements by a set of functions (and macros). For example, `MatSetPE(A, 3, 6, 3.14)` adds 3.14 to a_{36}. The 0-based equivalent is `MatSetPE0(A, 2, 5, 3.14)`. Both of these are equivalent to `MatSet(A, 3, 6, MatGet(A, 3, 6) + 3.14)`, however they are written more efficiently. These functions are needed because we can't write `MatGet(A, 2, 5) += 3.14` since the function `MatGet` does not return an "lvalue".

The full set of `MatGet` and `MatSet` functions is:

```
/* 1-based */
REAL  MatGet(MATRIX a, size_t i, size_t j);
REAL  MatSet(MATRIX a, size_t i, size_t j, REAL x);
REAL  MatSetPE(MATRIX A, size_t i, size_t j, REAL x);  /* += */
REAL  MatSetME(MATRIX A, size_t i, size_t j, REAL x);  /* -= */
REAL  MatSetTE(MATRIX A, size_t i, size_t j, REAL x);  /* *= */
REAL  MatSetDE(MATRIX A, size_t i, size_t j, REAL x);  /* /= */
```

```
/* 0-based */
REAL  MatGet0(MATRIX a, size_t i, size_t j);
REAL  MatSet0(MATRIX a, size_t i, size_t j, REAL x);
REAL  MatSetPE0(MATRIX A, size_t i, size_t j, REAL x); /* += */
REAL  MatSetME0(MATRIX A, size_t i, size_t j, REAL x); /* -= */
REAL  MatSetTE0(MATRIX A, size_t i, size_t j, REAL x); /* *= */
REAL  MatSetDE0(MATRIX A, size_t i, size_t j, REAL x); /* /= */
```

Note that `MatSet` returns a `REAL`, rather than a `void`, so that it emulates an assignment expression:

```
MatSet(A, 3, 6, x);
z = x;
```

can be written as

```
z = MatSet(A, 3, 6, x);
```

and of course as `MatSet(A, 3, 6, z = x)`, assuming both `x` and `z` have type `REAL`.

Compound assignment expressions have values. The expression `x += y;` has a value (the new value of `x`) that can be assigned to another variable. For example, `z = (x += y)` is equivalent to

```
x += y;
z = x;
```

The SML functions that emulate compound assignment operators return the values that the compound assignment expressions would have returned. Thus, `z = MatSetPE(A, 3, 6, x)` is equivalent to

```
MatSet(A, 3, 6, MatGet(A, 3, 6) + x);
z = MatGet(A, 3, 6);
```

Modules

The core parts of SML are:

File	Description
sml.cfg	Configuration file
sml.h	Top-level header file
smlbase.h	Basic prototypes, typedefs and access
smlacm.h	Matrix access by macros
smlacf.*	Matrix access by functions
smlmem.*	Memory management
smlerr.*	Error-handling

Input/output and computational routines are:

File	Description
smlio.*	Input/output
smlcomp.*	Basic computations
smlchol.c	Cholesky root
smlqr.c	QR decomposition
smlsvd.c	Singular value decomposition (SVD)
smleig.c	Eigenvalue decomposition

It is worth emphasizing here that the only functions that access matrix contents directly are in "smlmem.c" and "smlacf.c". The macros in "smlacm.h" provide macro access through macros with the same names as the functions in "smlacf.c". All other modules go through the basic functions listed above under "Abstraction". Those basic functions are the *interface* to the `MATRIX` object. As long as the interface stays fixed, the actual implementation of `MATRIX` can be changed at will, and other routines don't even have to be recompiled. So, "smlmem.c" and "smlacf.c" and "smlbase.h" constitute the foundation or bottom layer of the matrix library. Everything else fits above that layer.

Files

SML consists of a set of header files ending with `.h` and C source files ending with `.c`. Further there is a configuration file `sml.cfg`, which is actually a header file.

Some header files include other header files. The following table shows the include hierarchy among the header files.

File	#includes these files
smlbase.h	float.h + sml.cfg + (smlacf.h | smlacm.h) + smlmem.h + smlerr.h
sml.cfg	
smlacf.h	
smlacm.h	
smlmem.h	
smlerr.h	
smlio.h	smlbase.h
smlcomp.h	smlbase.h
sml.h	smlbase.h + smlio.h + smlcomp.h

Note that `smlbase.h` includes either `smlacf.h` or `smlacm.h`, but not both. File `smlacf.h` is included for function access of matrix data, while `smlacm.h` is included for macro access.

The user who doesn't need input/output or computations can use the basic services provided by `smlbase.h`. All basic, input/output and computation services can be accessed by simply including `sml.h`.

The following table shows which header files are included by different C source files.

File	#includes these files
smlacf.c	smlbase.h
smlerr.c	smlbase.h
smlmem.c	smlbase.h
smlio.c	smlio.h
smlcomp.c	smlcomp.h
smlchol.c	smlcomp.h
smlqr.c	smlcomp.h
smlsvd.c	smlcomp.h
smleig.c	smlcomp.h

Pitfalls of the `unsigned`

An important decision was to use the unsigned type `size_t`, rather than the signed `int`, throughout the library to number rows and columns. The advantage is that SML routines don't need to worry about a negative value being passed as the number of rows or columns or a row or a column number (but they have to deal with the value zero). Note also that the functions `MatRows` and `MatCols` return values of type `size_t`.

It is worth warning about comparisons between signed and unsigned types in C. There is a complicated set of promotion rules, and the results can be system-dependent. To avoid any confusion, if you have to compare signed and unsigned types, check first whether the signed variable is negative or not. Example, this function returns 1 if `s > u`, -1 if `s < u` and 0 if `s == u`.

```
int su_compare (int s, unsigned u)
{
  if (s < 0) return -1;
  if (s < u) return -1;
  if (s > u) return  1;
  return  0;
}
```

A second pitfall concerns loops that run backwards to 0 using an unsigned counter (as in a loop that runs backwards and uses 0-based indexing). Suppose we want to print the integers from 5 down to 0 in that order. The code

```
  const unsigned int n = 5;
  unsigned int i;
  for (i = n;  i >= 0; i--) printf("%d\n", i);
```

does not work - the `for` loop above is an infinite loop. The reason is that since `i` is unsigned, the comparison `i >= 0` will always evaluate to true. One solution is to write the *whole loop* backwards,

```
  const unsigned int n = 5;
```

```
unsigned int i;
for (i = n+1;  i != 0; ) {     /* start 1 higher */
    --i;                       /* decrement first */
    printf("%d\n", i);         /* body of the loop */
}
```

The counter `i` starts at the value 6. The first time through the loop `--i` decrements its value to 5, and that will be the first value passed to `printf`. When `i` reaches 0, the test `i != 0` yileds false and the loop ends. So the last number passed to `printf` will be 0 as desired.

Note that simply declaring `i` as a (signed) `int` can introduce a bug. The largest positive value that can be represented by an `int` is smaller than the largest value that can be represented by an `unsigned int` that has the same number of bits. For example, on many implementations using 4-byte `int`'s, the largest `int` is about 2 billion while the largest `unsigned int` is about 4 billion.

Performance issues (1)

Well, lets establish some principles before delving into this topic. First, the best optimization is good software design. Code that is well-structured and documented is easier to fine-tune, and improve when needed, than messy code. Premature optimization is evil (Knuth). Hasty optimization hacks are often too risky and can place severe restrictions on the flexibility and robustness of the code. That is, if they don't break the code and render it unusable.

Second, any attempt at performance tuning must be guided by actual performance data collected using a profiler or similar tools. It is nearly impossible to predict in advance where the bottleneck is going to be.

Third, bottlenecks tend to be highly localized, and often lie within a single line of code. Trying to blindly "optimize" the whole library one line at a time is a futile exercise.

Fourth, the library avoids pointer aliasing almost entirely. Smart optimizing compilers can do magic on such code. Use compiler optimizations, and profile the code with different optimizations to see which one works best for a particular program. The results may vary by system and compiler.

Fifth, changing matrix storage from by-row to by-column, or vice versa, can make a bigger difference in runtime than most other changes. This is an easy switch in the library, try it before anything else. However, this is a library-wide option, all library and other code must be recompiled. It is not possible to "mix and match". (Contrast this to function versus macro access options which allow mix-and-match). As an example of this, an SVD function applied to a 400×400 matrix took 9.8 seconds (timed using `clock()`) with by-row storage. With by-column storage, the same function took 6.3 seconds. That is a 36% saving in runtime. That particular SVD function, originally in FORTRAN, was optimized for by-column storage.

The last point is that SML was designed for matrices that are not too small. For small matrices, the overhead of memory management and loops becomes large realtive to the time needed to do the actual computations. SML would not be efficient in an application in which all matrices are, for example 2×2. However, for matrices that contain hunderds, thousands or more elements, the overhead becomes negligible relative to the cost of the actual computations.

It often happens that optimizations that seem good or promising in theory do not work in practice. This is not becuase the "theory" is wrong, but rather because some important factors are ignored. Any claims of

optimization must be backed up by actual performance data. Also keep in mind that system and compiler specifics can have a huge influence on the relative performance of one code to another.

Performance issues (2)

Out of the basic functions, the following will be the ones used most frequently:

```
size_t  MatRows (MATRIX a);
size_t  MatCols (MATRIX a);
REAL    MatGet  (MATRIX a, size_t i, size_t j);
void    MatSet  (MATRIX a, size_t i, size_t j, double x);
```

Functions `MatGet` and `MatSet` access matrix elements. Macros can be more efficient, but they can introduce other problems. Macros (as implemented in SML) don't do range checking, while the access functions can.

Our approach will be as follows. We'll make `MatGet` and `MatSet` available as either functions or macros. The function version will optionally perform bounds checking, the macros will not. The user makes a decision by `#define SML_AC_MAC` for macros or `#define SML_AC_FUN` for functions. Thus, when developing and debugging we can use function access with bounds checking. When the code has been fully debugged and tested, we can switch to either functions with no bounds checking, or to macros, and recoup the performance. Remember to profile the code to get actual performance data.

Also, it is always a good idea to have functions check that their input `MATRIX` arguments have compatible and correct dimensions.

---

smlcomp.c

File "smlcomp.c" is not part of the basic SML, but an example of how to code common matrix operations using SML.

We have coded matrix addition above. How about a trickier operation, matrix multiplication? Again, for clarity, we omit the error checking code.

```
void MatMul (MATRIX A, MATRIX B, MATRIX C)      /* A <- B * C */

{
  size_t i, j;
  MATRIX temp = MatDim(MatRows(B), MatCols(C));

  for (i = 1; i <= MatRows(B); ++i)
    for (j = 1; j <= MatCols(C); ++j) {
      REAL   s = 0;
      size_t  k;
      for (k = 1; k <= MatCols(B); ++k)
```

```
            s += MatGet(B, i, k) * MatGet(C, k, j);
        MatSet(temp, i, j, s);
      }

  MatSwap (A, temp);
  MatUnDim(temp);
}
```

What is the reason for using a local temporary matrix (`temp`)? We didn't use one in `MatAdd`. The reason has to do with the fact that matrix multiplication is a more complicated operation than matrix addition. A user may issue calls such as `MatMul(A, A, A)`, `MatMul(A, B, A)` or `MatMul(A, A, C)`. Updating `A` immediately would overwrite values that will be needed later to compute other elements. So we store the product in the temporary matrix `temp`. Next, we call `MatSwap` to swap the contents of `temp` and `A`. Note that we could have called `MatCopy(A, temp)` instead of `MatSwap(A, temp)`, but `MatSwap` is more efficient. Finally, we call `MatUnDim` to free `temp` (which at this point contains the "old" `A`). Many matrix functions can run into this situation, and they have to use temporary variables before assigning results to their final destination.

Memory management (1)

The memory management in `smlmem.*` is self-explanatory. Here we'll just explain why certain decisions were made.

The first question that comes to mind is why wasn't SML designed so that we can write `A = MatProduct(B, C)` instead of `MatMul(A, B, C)`. It can be claimed that the first form is more appealing. Certainly, since `MATRIX` is a pointer, `MatProduct` can allocate a matrix and return it as its return value. The problem with this design is that if `A` pointed to previously allocated memory, then after the call `A = MatProduct(B, C)` that memory would be left reserved but with no pointer pointing to it (assuming a copy of `A` hasn't been saved); that memory would be "leaked". The function `MatProduct(B, C)` has no access to `A` and thus can't manage `A`'s memory. If we were to insist on the form `A = MatMul(B, C)` then the user must free `A` before assigning to it:

```
MatFree (A);
A = MatProduct(B, C);
```

However, this doesn't solve the problem if `A` happens to be one of the arguments as in

```
A = MatProduct(A, C);
A = MatProduct(A, A);
```

It also doesn't solve the problem when function calls are cascaded:

```
A = MatProduct(B, MatProduct(C, D));
```

`MatProduct(C, D)` returns a temporary unnamed object that is never freed, and is thus a resource leak.

The design of SML allows writing

```
  MatMul(A, B, C);
  MatMul(A, A, C);
  MatMul(A, B, A);
  MatMul(A, A, A);
```

and these calls will work as expected, with no memory leakage.

Avoiding memory leakage while using SML requires only a simple strategy:

- Initialize all matrices with `MatDim(0,0)` and later use `MatReDim` or simply let the computations resize the matrices as necessary. This will eliminate the potential probelm of unintentional aliasing of `MATRIX` objects (explained below).
- Before any function returns, it must call `MatUnDim` for each local `MATRIX`. Functions that have several `return` points must arrange to call `MatUnDim` before returning.
- Never write a function that returns a `MATRIX`.

Example:

```
void my_function (MATRIX A, MATRIX B, MATRIX C, MATRIX D, MATRIX E)
/*   A <- B * C  + D * E      */
{
  MATRIX t1 = MatDim(0, 0);
  MATRIX t2 = MatDim(0, 0);

  MatMul(t1, B, C);
  MatMul(t2, D, E);
  MatAdd(A, t1, t2);

  MatUnDim(t2);
  MatUnDim(t1);
}
```

The only function in SML that returns a `MATRIX` is `MatDim`. Functions that compute new matrices place the results in one or more of their arguments. Further, the convention used throughout SML is that destination matrices are listed in the argument list before source matrices. For example, to compute `B+C` and place the result in `A`, we write `MatAdd(A, B, C)`. Remembering this is easy; arguments appear in `MatAdd` in the same order as they do in `A = B + C`.

When extending SML, it is strongly recommended that you don't write any functions that return a `MATRIX`. The memory management problems such functions create are impossible to eliminate.

The functions `MatDim` and `MatReDim` are guaranteed to not leak resources. `MatDim(nrow, ncol)` first allocates a matrix descriptor. If that fails it returns a NULL pointer. Otherwise, it allocates space for the actual data array. If that allocation fails, the matrix descriptor is freed before retruning a NULL. Similarly, if `MatReDim(A, nrow, ncol)` fails to reallocate the array, it leaves the original `A` unmodifed.

Memory management (2)

An important design decision was to make SML alias-free. That means no two distinct objects share the same underlying data. For example, suppose that `A` is a 5×2 `MATRIX`. It is quite easy, through pointer manipulation, to create, say, a 1×2 `MATRIX`, `B`, that actually refers the fourth row of `A`. However, that would open the room for a large number of bugs that are hard to track down. If `A` is resized or freed, the pointers in `B` will be invalidated. Any attempt to access `B` will result in undefined behavior. That means a crash (if we are lucky!) or wrong results that are extremely difficult to track down.

Of course, one can easily write `B = A` which causes `A` and `B` to refer to same object. However, that violates the alias-free design of SML, and is to be avoided.

Sometimes, copying can be avoided by using the function `MatSwap`. To destroy `A` after saving a copy in `B`, the code

```
MatReDim(B, 0, 0);
MatSwap(A, B);
MatUnDim(A);
```

is more efficient, in memory and time, than

```
MatCopy(B, A);        /* now two copies of A exist */
MatUnDim(A);
```

`MatSwap` swaps only the matrix descriptors, not the actual matrix data, and is thus very efficient (for large matrices). Functions that use temporary local matrices to hold intermediate results can benefit from using `MatSwap`. For example, to replace `A` with the product `A*B`, the code

```
void example(MATRIX A, MATRIX B)  /*   A <- A * B   */
{
  MATRIX T = MatDim(0,0);

  MatMul(T, A, B);

  MatSwap(A, T);
  MatUnDim(T);
}
```

is better than

```
void example(MATRIX A, MATRIX B)  /*   A <- A * B   */
{
  MATRIX T = MatDim(0,0);

  MatMul(T, A, B);
```

```
  MatCopy(A, T);  /* ???  MatSwap(A, T) is better here */
  MatUnDim(T);
}
```

Memory management (3)
Unintentional Aliasing

Here we discuss *aliasing* in a little more detail. We do that because the strategy in SML is to *avoid* aliasing.

The term *aliasing* refers to the situation where two pointers refer to the same object. If `A` and `B` are of type `MATRIX`, then after the assignment `A = B`, whatever is done to the object denoted by `A` will apply to the object denoted by `B`. So, even though the *pointers* `A` and `B` are distinct, they refer to exactly the same object. This can be an advantage, but can also be a problem. If we call `MatFill(A, 0)`, then we have to remember that matrix `B` has been filled with zeros. It becomes the programmer's responsibility to remember that `A` and `B` denote the same matrix. Aliasing can arise unintentionally, and key SML functions employ strategies that help prevent this from happening, provided some simple rules are followed. This is discussed below.

The call `A = MatDim(0, 0)` sets `A->rows=0`, `A->cols=0` and `A->m = NULL` (recall that `A->m` is a pointer to array data). Thus `A->m` is never left undefined.

Similarly, if `A` has been successfully allocated by `A = MatDim(5, 5)`, the call `MatReDim(A, 0, 0)` frees `A->m` then assigns `NULL` to it, so that the old pointer value will not be accessed by mistake.

Consider this code

```
MATRIX A  = MatDim(10, 10);
MATRIX B;             /* ??  B = MatDim(0,0) is better */

...
MatUnDim(A);
B = MatDim(10, 10);   /* now A == B is possible */
MatAdd(A, A, A);
...
```

The matrix descriptor that is freed by `MatUnDim(A)` may be reused (by `malloc`) and its address may be assigned to `B`. That is, right after

```
MatUnDim(A);
B = MatDim(10, 10);
```

the equality `A == B` may hold. This is "unintentional aliasing", because it was not the result of an explicit assignment such as `A = B` or `B = A`. It would cause the call `MatAdd(A, A, A)` to modify `B`, hardly what a programmer may have intended. A potential solution is to follow `MatUnDim(A)` by the assignment `A = NULL` so that `MatAdd(A, A, A)` will generate a memory violation. A cleaner solution is to initialize all matrices by `MatDim(0, 0)` the moment they are defined, and free them by `MatUnDim` just before they

go out of scope. This prevents any two matrices from sharing the same matrix descriptor in memory, and eliminates the problem of unintentional aliasing of `MATRIX` objects.

If matrix `A` is no longer needed, and we wish to reclaim the memory it uses long before it goes out of scope, the following strategy

```
void example ( /* ... arguments here ... */ )
{
  MATRIX A  = MatDim(0, 0);

  /* ... code that uses A ... */
  MatReDim(A, 0, 0);    /* A is no longer needed */
  /* ... code that does not use A ... */
  MatUnDim(A);  /* just before A goes out of scope */
}
```

is better than

```
void example ( /* ... arguments here ... */ )
{
  MATRIX A  = MatDim(0, 0);

  ...
  MatUnDim(A);          /* ??? too early */
  /* ... code that does not use A ... */
}
```

To summarize, the safest way to use `MATRIX` objects in a function is to call `MatDim(0, 0)` as early as possible, and `MatUnDim` as late as possible, for each such local object,

```
void example ( /* ... arguments here ... */ )
{
  MATRIX t1 = MatDim(0, 0);
  MATRIX t2 = MatDim(0, 0);
  MATRIX t3 = MatDim(0, 0);

  ...
  MatUnDim(t3);
  MatUnDim(t2);
  MatUnDim(t1);
}
```

This strategy guarantees that all local `MATRIX` objects will be distinct for the entire life of the function, so there will be no risk that operations on one of them will modify any of the others.

Performance issues (3)

Due to the critical role that memory caches play in modern computer systems, it is to be expected that the performance of matrix routines will depend highly on the order in which they access matrix elements. The best performance is obtained by accessing matrix elements in the same order in which they are laid out in memory. A simple example is offered by the function `MatSum`, which returns the sum of all elements of a matrix.

We can implement `MatSum` as follows:

```
REAL  MatSum (MATRIX A)             /*  sum of elements of A */
{
  size_t i, j, m = MatRows(A), n = MatCols(A);
  REAL  s=0;

  for (i = 1; i <= m; ++i)
    for (j = 1; j <= n; ++j)
      s += MatGet(A, i, j);

  return s;
}
```

This function processes the matrix by row. If the matrix happens to be stored by column, many cache misses will occur and performance will degrade. The extent of performance loss depends on many factors including matrix dimensions and system specifics such as CPU type, size of memory caches, compiler, etc. On large matrices, slowdowns by a factor of 5 to 10 are not unusual. [As an exercise, obtain run times of this function first with SML configured "by row" and another time with SML configured "by row". For the size of `A`, try 10×10, 100×100, 1000×1000 and 2000×2000.]

The performance of `MatSum` as written above is fragile; it is highly dependent on the storage mode. This problem can be fixed in several ways. One solution is to write two versions of the function (specifically, the loops), one that processes the matrix by row, the other by column, and choose between them at compile time by preprocessor directives

```
REAL  MatSum (MATRIX A)             /*  sum of elements of A */
{
  size_t i, j, m = MatRows(A), n = MatCols(A);
  REAL  s=0;

#if defined(SML_BY_ROW)
  for (i = 1; i <= m; ++i)
    for (j = 1; j <= n; ++j)
#else
  for (j = 1; j <= n; ++j)
    for (i = 1; i <= m; ++i)
#endif
      s += MatGet(A, i, j);

  return s;
}
```

This new version solves the problem, but is a little complex. It is also tedious to test and debug - we have to test and debug twice; once with SML_BY_ROW and again with SML_BY_COL.

There is another solution that not only eliminates the performance fragility, but simplifies the code considerably as well. The technique is general enough that it is worth discussing in detail.

Some of the functions in SML are primarily "one-dimensional" in nature, i.e. their operation requires knowing only where the matrix data are and how many elements there are. They don't need knowledge of the number of rows or columns, or the orientation (by column or by row) of the matrix. An example is the function `MatSum`. Whether a matrix has dimensions 10×1, 5×2, 2×5 or 1×10, `MatSum` returns the sum of the 10 numbers residing in a contiguous block of memory. The two-dimensional layout of the matrix has no bearing on the result.

By diverting the work to a special "one-dimensional" function we can improve both the performance and the readability of `MatSum`,

```
REAL  MatSum (MATRIX A)             /*  sum of elements of A */
{
  return vec_sum(MatData(A), MatRows(A) *  MatCols(A));
}
```

The SML function `MatData` returns a pointer to the first element in the matrix, i.e. the element in the first row and first column. The function `vec_sum` takes a pointer to the first element of an array and a count of the number of elements, and returns the sum of those elements. It can possibly be implemented as follows:

```
REAL  vec_sum (REAL *p, size_t n)  /*  sum of p[0:n-1] */
{
  REAL s=0;
  size_t i;

  for (i=0; i != n; s += p[i++]);

  return s;
}
```

Now each function is essentially one line, and is easier to read and test/debug. The function `vec_sum` marches through memory linearly regardless of whether the original matrix is stored by row or by column. The new version of `MatSum` accesses memory in an optimal fashion, regardless of the storage mode (by row or by column). Further, the simple loop in `vec_sum` can be unrolled (explained below) to improve performance a little more.

Many other SML functions are inherently "one-dimensional" and are amenable to the above technique. Examples include `MatApply`, `MatAdd`, `MatSS`, `MatSumAbs`, `MatMin`, `MatMax`, `MatSqrt`, `MatMulScalar`, etc.

The disadvantage of the above approach is that it bypasses the accessor functions `MatSet`, `MatGet` and their relatives. Using this option is not advisable unless the code has been thoroughly tested using

function access with bounds checking. Another disadvantage is that if the details of matrix storage change, or if we decide to add support for special storage modes (e.g. triangular, Hessenberg, banded, diagonal, symmetric) then every matrix function that diverts work to one-dimensional functions will have to be rewritten to handle the new types of storage properly.

Delegation to one-dimensional functions is available in SML by defining `SML_1DIM` in `sml.cfg`. This affects only a subset of the functions in `smlcomp.c`.

Loop unrolling was mentioned above. It refers to the expansion of inner loops so that they process more elements in each pass through the loop. This takes advantage of the "instruction pipeline" by keeping it full for longer periods. Here are two unrolled versions of `vec_sum`:

```
REAL    vec_sum2 (REAL *p, size_t n)  /*  sum of p[0:n-1] */

{
  REAL s0=0, s1=0;
  size_t i, m = 2 * (n / 2);

  for (i=0; i != m; i += 2) {
    s0 += p[i];
    s1 += p[i+1];
  }

  for (; i != n; s0 += p[i++]);

  return s0 + s1;
}

/*-------------------------------------------------------*/

REAL    vec_sum4 (REAL *p, size_t n)  /*  sum of p[0:n-1] */

{
  REAL s0=0, s1=0, s2=0, s3=0;
  size_t i, m = 4 * (n / 4);

  for (i=0; i != m; i += 4) {
    s0 += p[i];
    s1 += p[i+1];
    s2 += p[i+2];
    s3 += p[i+3];
  }

  for (; i != n; s0 += p[i++]);

  return s0 + s1 + s2 + s3;
```

```
}
```

vec_sum2 processes two elements per iteration while vec_sum4 processes four. Modern processors have long instruction pipelines and loop unrolling tends to be helpful. Some compilers can apply loop unrolling (thus the source code doesn't need to be modified) as a compliation option. The actual results are compiler and system-dependent and are hard to predict in advance. The table below shows timing data obtained on an Athlon 1700 PC using the Watcom compiler. The time shown is the number of seconds it took for each function to compute the sum of an array of 1024^2 (a little over one million) elements 100 times. Timing was obtained with `clock()`. The program was compiled once with no switches, and another time with the `-ol+` switch which applies loop optimization and unrolling.

Function	No Switches	`-ol+`
vec_sum	1.67	1.41
vec_sum2	1.19	1.36
vec_sum4	0.92	1.31

The results may be surprising. Looking across rows of the above table, the original vec_sum actually runs 16% faster with the `-ol+` switch, vec_sum2 takes 14% longer, while vec_sum4 takes 42% longer. Looking down the columns shows that the manually-unrolled versions run faster than the original. The benefit is only marginal (3% for vec_sum2 and 7% for vec_sum4) when the code is compiled with `-ol+`. For code compiled without `-ol+`, vec_sum4 takes 45% less time than the original version. The overall winner is vec_sum4 compiled *without* `-ol+`. If we were to seek the best performance, we may consider unrolling the loop even further.

It is often the case that hand-written optimizations tend to interfere with, or even counteract, compiler-supplied optimizations. Note that the results may be completely different with a different compiler, or on a different system.

Input/Output

Arrays of Matrices

Error Handling

Error handling in SML is quite simple. If any SML functions encounters an error it calls MatErrThrow, which prints a message to stderr and sets an internal *error flag*. The internal error flag is modeled after errno, but thru functions rather than as a global variable. (Also, unlike errno, currently the flag is either set or clear; it doesn't hold any error codes.) SML functions set the error flag in case of errors, but they never clear it. The user can clear the error flag before calling one or more SML functions, then check its status

```
...
MatClrAbort();        /* this is explained below */
```

```
MatClrErr();
MatAdd (A, B, C);
if (MatErr()) printf("MatAdd() reported an error\n");
else printf("MatAdd() was successful\n");
```

The reason for calling `MatClrAbort` in the code above is to prevent the program from exiting in case `MatAdd` detects an error. An internal *abort flag* controls whether SML calls `exit` in case of errors (if the abort flag is set) or it does not (if the abort flag is clear). The default is that the abort flag is set, so any SML errors will cause the program to `exit`. This is acceptable in many programs. However that behaviour can be changed by calling `MatClrAbort`, or restored by calling `MatSetAbort`. These functions can be called more than once so that certain sections of a program can run with the abort flag set, and other sections with the abort flag cleared.

The functions `MatClrAbort` and `MatSetAbort` return the value of the flag that existed before the call. Thus the old status of the abort flag can be restored later if necessary

```
int old_abort_flag;

/* example 1 */
old_abort_flag = MatClrAbort ();
/* some code here ... */
if (old_abort_flag) MatSetAbort ();

/* example 2 */
old_abort_flag = MatSetAbort ();
/* some code here ... */
if (!old_abort_flag) MatClrAbort ();
```

SML keeps an internal count of the number of errors it encounters. The count starts at 0 and is reset to 0 every time `MatClrErr` is called. The number of errors can be obtained by calling `MatErrCount`

```
MatClrAbort();
MatClrErr();

MatI(A, 2, 1);  /* 2x2 identity matrix */
MatI(B, 3, 1);  /* 3x3 identity matrix */
MatI(C, 4, 1);  /* 4x4 identity matrix */
MatSub(A, B, C);      /* error */
MatAdd(A, B, C);      /* error */
MatMul(A, B, C);      /* error */
printf("SML has encountered %d errors\n", MatErrCount()); /* 3 */
MatClrErr();
printf("SML has encountered %d errors\n", MatErrCount()); /* 0 */
```

Calling Template Numerical Toolkit and Numerical Recipes code from SML

An approach used in the Template Numerical Toolkit (TNT) (http://math.nist.gov/tnt) and in the book Numerical Recipes (NR) is to setup an $m \times n$ matrix as an array of m pointers, each pointing to a row that contains n values. A compatible storage mode is available in SML. It is supported via an alternative set of the files `smlacf.c, smlacm.h` and `smlmem.c`. The alternative files are simply named `smlacf2.c, smlacm2.h` and `smlmem2.c`. If these files are used to replace the originals (without the suffix "2"), we get an implementation of SML that is compatible with the TNT approach and the NR book, provided `SML_BY_ROW` is `#defined`.

As an example, there is a NR function that computes all eigenvalues and eigenvectors of a real symmetric matrix, declared as follows (we'll assume here that REAL is double):

```
void jacobi(double **a, int n, double d[], double **v, int *nrot);
```

This function can be called from SML as follows:

```
jacobi(MatDataNR2(A), MatRows(A), MatDataNR1(D), MatDataNR2(V), &nrot);
```

assuming matrices `A` and `V` have been allocated as $n \times n$ and `D` has been allocated as $1 \times n$ ($n \times 1$ would also work). Matrix `A` serves as the input to `jacobi`, while `V` and `D`, and the integer `*nrot` are the output (computed results).

The rule for passing SML matrices as arguments to NR functions is very simple.

- Use `MatDataNR1` if NR expects a 1-dimensional array. If NR expects `double* d` (or `double d[]`), use `MatDataNR1(D)`.
- Use `MatDataNR2` if NR expects a 2-dimensional array. If NR expects `double** a`, use `MatDataNR2(A)`.

Fortunately, violating this rule means that the wrong data type is being passed, and it should generate a compile-time error, or at least a warning.

The function `MatDataNR1` returns a pointer to the element in the first row and first column of the matrix. The function `MatDataNR2` returns a pointer to the first element in the array of pointers to the rows of the matrix.

A possible sequence that calls `jacobi` might look as follows

```
MATRIX A = MatDim(0,0);
MATRIX V = MatDim(0,0);
MATRIX D = MatDim(0,0);
int n;
int nrot;

n = 5;
MatReDim(A, n, n);
```

```
/* ... assign values to elements of A here ... */

MatReDim(V, MatRows(A), MatRows(A)); /* allocate the required space */
MatReDim(D, 1, MatRows(A));          /* allocate the required space */

jacobi(MatDataNR2(A), MatRows(A), MatDataNR1(D), MatDataNR2(V), &nrot);

/* ... process V, D and nrot ... */

MatUnDim(D);
MatUnDim(V);
MatUnDim(A);
```

In summary, to be able to interface SML with TNT or NR:

- Use `smlacf2.c, smlacm2.h` and `smlmem2.c.` to replace `smlacf.c, smlacm.h` and `smlmem.c` (by copying or renaming)
- `#define SML_BY_ROW` in `sml.cfg`.
- Make sure the element types are identical; `float`, `double` or `long double` in both SML (by editing `sml.cfg`) and the code to which it is being interfaced.
- Make sure NR or TNT are using 0-based indexing.

Note: In this mode, SML can be configured by-column (`#define SML_BY_COL`), so that a matrix is represented by an array of pointers to columns. The SML code itself will run properly, but it won't be compatible with TNT or NR.

To Do List

Volunteers are needed to:

- Develop test routines
- Develop Timing routines
- Organize the error messages
- Standardize the error codes

Conclusion

SML started life many years ago and has gone through iterations that spanned several programming languages, last of which was Pascal. The Pascal (and Modula-2) influence on function naming conventions is obvious (`MatDim`, not `mat_dim`). A hint of C++ influence is also detectable (although SML never existed in C++). SML was written over an extended period of time. The coding style is highly consistent, but it is not 100% consistent.

This is nearly the end of the road. It is unlikely that SML will change in a significant way in the future. That is, as far as the C language version is concerned.

If you have any questions, comments or bug reports, or if you'd like to contribute to SML, please contact me by email (my email address is on the first page of this document).

Thanks for reading this far!