Chapter 8

Designing numerical libraries in C

The purpose of this chapter is to have a bit of a look "under the hood" to see how a library of routines in C can (and we believe, should) be built up. The philosophy here is to make use of the features of C to make programs more flexible and easier to write (and debug), while not sacrificing too much efficiency. There are other ways of designing numerical libraries, but this has been found to be a useful and flexible way of designing numerical libraries in C.

8.1 Numerical programming in C

Numerical and scientific programming has been traditionally associated with Fortran. Indeed, a great deal of software has been written in Fortran, in spite of its well known defects (lack of good data structures, lack of strong typing, reliance on "GOTO", poor lexical characteristics, clumsy input/output). This has led to the "historical" defense of Fortran: "There is so much already written in Fortran that we have to program in Fortran."

However, more sophisticated algorithms need more sophisticated data structures and more structured programs. Sparse matrix data structures and operations on them are one example of this. C is one of a number of languages that easily support such structuring. As well, C is a very flexible language, especially as regards memory management. While it is often argued that C is "merely a systems programming language", several aspects of C seem to indicate otherwise. For example, C has both single and double precision. Sometimes the argument is made that C is not suitable for numerical programming because single precision numbers are automatically converted to double precision whenever they are passed as arguments or used in expressions. This is no longer true in ANSI C. Even with the older C convention, the main drawbacks are the time spent converting between double and single precision numbers. Operations done entirely in double precision are immune to this inefficiency. It is, in any case, a better state of affairs than not having double or extended precision numbers as is the case with Pascal or the original version of Modula-2. Also, the standard Unix^{TM} mathematics library has not only the standard functions (exp, log, and the trigonometric

functions), but also Bessel functions, the Γ function and the error function. Admittedly, C does not have complex numbers, but this is a standard extension to C++.

8.1.1 On efficient compilers

The comment is sometimes made that Fortran *must* be more efficient than C. This is based on the fact that pre-Fortran 90 Fortrans are simpler languages, and that C has a rather more permissive structure. However, with modern compilers the difference in performance is usually fairly small, and is often non-existent. One of the reasons for this is that on many new machines compilers for different languages share common code-generation and optimisation parts. Indeed, the first NAG Fortran 90 compiler is actually a pre-processor that converts Fortran 90 into C — this is a sensible strategy because of the high quality and wide availability of many C compilers. The point that should be made is that efficiency is often a question of how much effort goes into developing the compilers. In the late 1970's the MACLISP compiler developed at MIT could produce machine code for compiled Lisp that rivalled Fortran in efficiency for numerical operations.

There are some inefficiencies that can be introduced in writing C code that would not appear in writing Fortran. But this is due to using a different style of programming. For example, overusing dynamic memory allocation can result in a great deal of overhead. (Beginners to programming in C can easily fall into a trap of writing code that spends most of its time allocating and deallocating temporary objects.) However, with a little care, this overhead can be kept to a negligible level while providing far more flexibility than is possible in Fortran 77.

8.1.2 Strategies for using C

The aspects of C that numerical programmers should make use of include

1. the ability to create self-contained data structures representing meaningful mathematical objects.
2. dynamic memory allocation and de-allocation of data structures and arrays, which often avoids the need for workspace arrays.
3. error and exception handling using `setjmp()` and `longjmp()`.
4. flexible input and output so that self-contained data structures can be read in and printed out.
5. use of pointers to represent user-defined objects whose characteristics are not known at compile time.

Self-contained data structures not only simplify argument lists, but can also be used for internal consistency checks to catch illegal operations. They should also make programs easier to understand in that they correspond closer to mathematical objects, and avoid the need to a plethora of additional length arguments and variables. By

using functions to perform most of the needed operations on these data structures, the chances of misusing the data structures can be greatly reduced.

Dynamic memory allocation and de-allocation not only avoids workspace arrays, but also avoids the need for the strategy of declaring the largest conceivable array sizes in local arrays. With this, memory can be used far more effectively.

A common error/exception handling mechanism means that the usual testing of "`IFLAG`" arguments can be avoided as well. A suitably structured mechanism can be used to provide a safe way of giving control back to the user if an error occurs. The users need to state what error they wish to "catch" and the code in which they wish to "catch" it; if an error occurs executing the code, control passes to the "catch" mechanism which can pass control back to the user's own code for handling the errors. Done properly, it can also provide a partial "backtrace" of the state of the active functions at the time of the error.

Input and output are, of course, very important. After all, a program without output is useless. More than this, by structuring input and output, output can be reused as input. Consider how often have you had to edit data just so that your program can use it as input?

Another aspect of structuring input is that comments can be incorporated into the input. Data, by itself, rarely means much. Including comments makes it much more intelligible to mere mortals. The flexibility of C's input and output has been used to do this.

User-defined objects (of any sort) can be handled by a combination of functions and pointers. Pointers to functions can be arguments to functions, and components of arrays or other data structures. This means that essentially arbitrary user-defined data structures can be used by code without knowing any of their characteristics at compile time. This style of programming has some of the flavour of object-oriented programming.

Meschach in various places makes use of all these aspects of C. We hope that you find this way of programming effective and efficient, not only in terms of CPU time, but your own (programming and debugging) time as well.

8.1.3 Non-C programmers start here!

Before going past this point, you really should read a book on C and programming in C. However, as there are undoubtedly non-C programmers who will want to follow the discussion in this chapter, here are some very brief notes which should help you understand the examples.

C programs consist of collections of functions, one of which is the main program (called "`main()`"). Routines consist of a header followed by a sequence of statements (the body of the routine) inside braces (`{...}`). Statements are either simple statements, which must end with a semi-colon (`;`), or compound statements, which is a collection of simple or compound statements bracketed by braces. The braces work very much like Algol, Pascal and Ada `begin...end` pairs. Comments in C have the form `/* ... */`.

Before a C program is compiled, it is passed through a *pre-processor*. Pre-processor directives must have a # as the first character on that line. The pre-processor can be used to define macros, to include files, and to delete code according to whether macros are defined. Standard header files are almost always included in C programs. Here is an example:

```
#include  <stdio.h>  /* standard input/output header file */
#include  "mydefs.h" /* uses file from current directory */
/* examples of macro definitions */
#define   max(a,b)   ((a) > (b) ? (a) : (b))
#define   DEBUG      TRUE
```

The basic data types in C include `int` ("integer"), `double` ("double precision floating point") and `char` ("character"). A declaration has the name of the data type before a list of variables, as in

```
int     i, j, idx;
double  alpha;
```

A *pointer* to a particular data type is declared by putting a * before the *variable* which is to be a pointer. For example, after the declarations

```
double  d, *pd, **ppd;
```

`d` is a `double`, `pd` is a pointer to `double`, and `ppd` is a pointer to a pointer to `double`.

Consistent with this, accessing the value pointed to by a pointer is simply a matter of putting a * before the variable. For example, the value pointed to by `pd` is `*pd`.

The reverse operation of finding a pointer that points to a variable is done by putting `&` before the variable; e.g. `pd = &d;` now makes `pd` point to the variable `d`.

Arrays are declared using square brackets such as

```
double   x[10];
```

This declares `x` to be an array with 10 entries. However, the starting index is *zero*, not one. So the valid entries of `x` are `x[0]`, `x[1]`, ... , `x[9]`. This is called *zero–relative indexing*. This may appear unusual at first, but is no barrier in practice.

Arrays and pointers are very similar; when arrays are passed to subroutines, only a pointer is passed, and pointers can be used like arrays. For example, `pd[0]` is equivalent to `*pd`; `pd[1]` is the double precision number next to `*pd`. This is called *pointer arithmetic* and can be easily abused. There are two important differences between arrays and pointers: (1) pointers are not necessarily associated with any usable piece of memory, while arrays are, and (2) array names cannot be assigned, but pointers can. So `pd = x;` is legal, but `x = pd;` is not.

Data structures containing (possibly) different kinds of objects are declared using `struct`. For example, complex numbers can be declared as

```
typedef struct cmplx { double real, imag; } complex;
```

(Here we have used `typedef` in order not to use the longer name `struct cmplx`.) Complex numbers can then be declared by

```
complex z1, z2;
```

Structures can be imbedded in structures, and recursive structures (such as linked lists) can be declared using pointers to that structure. For example, here is a linked list structure:

```
struct list { int contents;  struct list *next; };
```

The components of a data structure can be obtained by using ".". The real part of `z1` is `z1.real`. If `pz` is a pointer to a complex number, then the real part of the complex number pointed to is `(*pz).real`, which has the equivalent shorthand form: `pz->real`.

The control structures in C are familiar to most programmers — if–then–else, while, do–while (*cf* Pascal's repeat–until) and for loops. These have a straightforward syntax except for the for loop construct. Before these constructs are described, it should be noted that C has no Boolean or logical data type. Instead, zero or NULL is regarded as "False", while non-zero and non-NULL values are regarded as "True". The results of logical and relational operations are always integers `int`, with 1 representing "True". The comparison operators are equality test (`==`), inequality test (`!=`), and the usual numerical comparison operators (`<`, `>`, `<=`, `>=`). Logical operators include "logical and" (`&&`), "logical or" (`||`), and "logical not" (`!`). (There are also bitwise and, or, not and exclusive or operators.) Expressions involving `&&` and `||` are evaluated left–to–right and evaluation is "short-circuited" so that latter expressions are not evaluated if not needed. This is very useful to avoid performing invalid operations. For example,

```
ok = ( i < array_length ) && item_ok[i];
```

does not evaluate `item_ok[i]` if `i >= array_length`.

If statements have an optional `else` part and can be strung together.

```
if ( condition1 )
    statement1;
else if ( condition2 )
{   statement2; statement3;   }
```

While loops have the form "`while ( condition ) statement;`" or "`while ( condition ) {... }`". The do–while variant has the form "`do statement; while ( condition );`" or "`do { ... } while ( condition );`". The for loop in C is the most flexible and has the form

```
for ( initialisation; test; update )
    statement;
```

where "`statement;`" can be replaced by a compound statment. This is equivalent to a while loop:

```
initialisation;
while ( test )
{   statement;  update;     }
```

The for loop is most commonly used in a standard idiom:

```
for ( i = 0; i < array_length; i++ )
    ..... array[i] .....
```

The expression `i++` returns the value of `i` and then increments the value of `i` by one. (Here, of course, the *value* of the expression is ignored.) This is the *post-increment* operation; `i--` is the *post-decrement* operation. Preceding the variable with `++` or `--` *pre-increments* and *pre-decrements* the value of that variable. Other updates commonly used include incrementing the index by a different *stride*: `i = i+stride`, or with the shorthand `i += stride`.

Inside all loop constructs in C you can put `break` and `continue` statements. The `break` statement causes the loop to exit immediately; the `continue` statement causes control to be passed to just before the end of the loop.

All routines in C are functions. They might have side–effects and they might return `void` (so that the returned value is unusable), but they are functions. It is not necessary to do anything with the returned value, whether or not it has type `void`. Also, all function arguments are passed *by value* rather than *by reference*. Thus if you wish a function to set the value of a variable, you need to pass a pointer to that variable. For example, an integer swap routine would be called like this:

```
int     i, j;
swap(&i,&j);
```

If the type of the returned value from a function is not `int` (i.e. the standard integer type) then it should be declared before use. For example, a routine to add complex numbers together might be declared before use as

```
complex cadd();    /* adds two complex numbers */
```

If this is preceded by `extern` it means that the function is defined in another file. In ANSI C argument types can also be checked if you declare your functions using *function prototypes* such as

```
complex cadd(complex, complex);     /* or */
complex cadd(complex z1, complex z2);
```

Ther are two styles for defining a function: the old way, and ANSI C. Here is the old way:

```
complex cadd(z1, z2)
complex z1, z2;
{   complex z;
    z.real = z1.real + z2.real;
```

```
    z.imag = z1.imag + z2.imag;
    return z;  /* z is the returned value of cadd() */
}
```

And here is the ANSI C way:

```
complex cadd(complex z1, complex z2)
{   complex z;
    z.real = z1.real + z2.real;
    z.imag = z1.imag + z2.imag;
    return z;  /* z is the returned value of cadd() */
}
```

Functions can be passed as parameters, but what is actually passed is a *pointer to a function.* A pointer to a function can be used as other pointers can: arrays of pointers to functions are legal, as are structures containing pointers to functions. Here is declaration of a pointer to a function returning a `double`:

```
double (*f)();
```

Or using ANSI C, the types of the argument(s) can be included:

```
double (*f)(double);
```

Then assigning `f = exp;` is perfectly valid.

8.2 The data structures

C allows for extensive use of data structures. The `struct` and `typedef` facilities provide means whereby heterogeneous structures and primitive types can be combined and used together. As such they provide a *static* way of describing the data structure; they define the way things are stored. Equally important to the way things are stored, is the question of how such information is *used.* This is the *dynamic* part of the data structure. While C is not really set up to deal with complete formal descriptions of both the static and dynamic aspects of a data structures in the way object-oriented languages (such as SmallTalk and C++) are, we can go part way by providing functions that do at least the basic operations on the data structures.

8.2.1 Pointers to struct's

One approach that we have taken throughout the library is to pass only *pointers* to the actual `struct`'s. Passing the actual `struct`'s is useful for relatively small objects, but we believe it is inappropriate to do this for large objects and for objects which contain pointers to allocated memory. For example, complex numbers

```
typedef struct { double real, imaginary; } complex;
```

should be passed as single entities, while vectors

```
typedef struct { int dim, ...; double *ve; } VEC;
```

should not.

Why should this distinction be made?

1. Passing large structures is less efficient.
2. Copying the `struct` itself will only copy the pointers in the `struct`, **not what those pointers are pointing to.**

The second item notes that only a *shallow copy* is made by an assignment of a `struct`. For example, the following code does not do a true copy (at least it is usually not what the writer intends). ***Do not do this!***

```
VEC  x, y;
  ......
y = x;  /* this is an error in pre-ANSI C */
y.ve[1] = 3.0;
/* now x.ve[1] is also 3.0 */
```

Pointers can be copied, but here it is clear that its effect is not a deep copy.

```
VEC  *x, *y;
  ......
y = x;            /* y and x now point to the same place */
y->ve[1] = 3.0;
/* now x->ve[1] is 3.0 */
```

It is only with C++ that assignment can be forced to result in a deep, rather than a shallow, copy.

8.2.2 Really basic operations

Some operations are so basic that it is absolutely vital that they are implemented first. They are (in order):

1. Allocation and initialisation.
2. Output.
3. De-allocation.
4. Copying.

You might find it strange that output routines appear so soon. However, one thing is sure about developing data structures: you will want to debug them.

Writing allocation and initialisation routines is not difficult, but you should use the discipline that all returned values from `malloc()`, `calloc()` and `realloc()` are

checked. Also, check that the parameters passed make sense. If something goes wrong at this level it is unlikely that you can do much sensible. Passing control to an error handler, such as the `error()` macro does, is probably the most sensible thing to do here. Here is a hypothetical `struct` and the code to do (some) of the allocation and initialisation:

In the file `foo.h` we define the data structure and the new type `foo`:

```
typedef struct { int size; ... double *array; } foo;
```

In the file `foo.c` the basic operations are defined:

```
#include  "foo.h"
  ......
foo *get_foo(size)
int size;
{
    foo *my_foo;

    if ( size <= 0 )
        error(E_BOUNDS,"get_foo");
    /* get foo struct first */
    my_foo = (foo *)calloc(1,sizeof(foo));
    if ( my_foo = (foo *)NULL )
        error(E_MEM,"get_foo");
    /* now set up pointers */
    my_foo->array = (double *)calloc(size,sizeof(double));
    if ( my_foo->array = (double *)NULL )
        error(E_MEM,"get_foo");
    my_foo->size = size; /* now it is safe to set the size */
      ......
    return my_foo;
}
```

The function call `calloc(num_elts, size_elts)` allocates a block of memory for `num_elts` blocks of size `size_elts` characters. What is returned is a pointer to the allocated memory. If `calloc()` returns a NULL pointer, then this indicates that there is insufficient memory. The returned value of `calloc()`, `malloc()` and `realloc()` should always be checked before use. If an error occurs, then the `error()` macro is called, which raises an error at this point, and no further code in this function is executed.

The Meschach macros `NEW(type)` and `NEW_A(num,type)` in `matrix.h` simplify writing this sort of code:

```
if ( (my_foo = NEW(foo)) == (foo *)NULL )
    error(E_MEM,"get_foo");
  ......
```

```
if ( (my_foo->array = NEW_A(size,double)) == (double *)NULL )
    error(E_MEM,"get_foo");
  ......
```

De-allocation should be done using the function `free()` in the reverse order:

```
void free_foo(my_foo)
foo  *my_foo;
{
    if ( my_foo == (foo *)NULL )
        return;
      ......
    if ( my_foo->array != (double *)NULL )
        free(my_foo->array);
    free(my_foo);
}
```

There is not much more error checking that can be done at this stage. Checking that memory heaps are not corrupted can only be part of the design of the memory allocator, not the data structure or its routines.

Note that only pointers to memory that has been allocated by `calloc()`, `malloc()` or `realloc()` can be de-allocated using `free()`, and this can only be done once. Common errors are to try freeing memory more than once.

8.2.3 Output

Output should be structured but human readable. Usually we will want to be able to read the output back in later, so we should try to make the output reasonably machine-readable as well. (Writing input routines is usually much harder and more complex.) Hence the output should contain fore-warnings about what is coming, and how big it is before we get to it. It should also be possible to direct the output to any file or stream that we choose.

In the `foo` example,

```
void fout_foo(fp,my_foo)
FILE *fp;
foo  *my_foo;
{
    int i;

    fprintf(fp,"Foo: ");
    if ( my_foo == (foo *)NULL )
    {
        fprintf(fp,"NULL\n");
        return;
    }
```

```
    fprintf(fp,"size: %d\n",my_foo->size);
      ......
    fprintf(fp,"array: ");
    for ( i = 0; i < my_foo->size; i++ )
    {   /* no more than 6 items on a line */
        if ( (i % 6) == 5 || i == my_foo->size - 1 )
            fprintf(fp,"%g\n",my_foo->array[i]);
        else
            fprintf(fp,"%g ",my_foo->array[i]);
    }
}
```

(Actually, returning `my_foo` at the end would be useful behaviour, although we haven't done this in Meschach.)

Note that care is taken to treat the NULL case separately so that this will not result in failure; instead the message "`Foo: NULL`" is printed. For a proper allocated and initialised the output might look something like this:

```
Foo: size: 10
  ......
array: -3.7 2.5 3.141592 2.2 -1
1.5345 101 25.2321 -3.2 2.5
```

Writing an input routine to read this in is simplified because it can see how big to make the `array` before it has to read any of it in. Writing a routine to output every bit of the `foo` structure (even though most users won't want it) is often useful for debugging purposes. This can be done by writing an additional `foo_dump()` function.

8.2.4 Copying

The purpose of these routines is to provide a *deep copy* which copies all the component parts as well as the `struct` itself. There are two styles of doing this; one is to return a completely new `struct`, created and initialised, and the other is to copy the data structure into an already allocated and initialised one. One way to do both in one routine is to check the target structure pointer; if it is `NULL` then a new target structure should be created:

```
foo *cp_foo(from,to)
foo *from, *to;
{
    int   i;
      ......
    if ( from == (foo *)NULL )
        error(E_NULL,"cp_foo");    /* can't copy NULLs */
    if ( to   == (foo *)NULL )
```

```
        to = get_foo(from->size); /* create a new foo */
    else if ( to->size < from->size )
        /* make sure target is big enough */
        to = foo_resize(to,from->size);

    /* now do copying */
      ......
    for ( i = 0; i < from->size; i++ )
        to->array[i] = from->array[i];
      ......
}
```

The results of using `cp_foo()` can be used without checking as when a failure occurs, there is a call of the `error()` macro which invokes the error handling code. Once the checking is done, the actual copying can proceed as a straightforward loop. The efficiency of copying routines can be improved by using specialised copying routines such as `bcopy()` for BSD, or `memmove()` for ANSI C.

8.2.5 Input

Although this is not one of the "really basic" routines, they are useful and even important. Also, they are also trickier than output routines to write well.

It has been observed that in many software systems that the overall complexity of the code is usually dominated by the user interface. Writing a numerical library avoids a lot of that, and getting other programs/libraries to do your input/output is often a good idea. (Writing routines to output matrices in MATLAB save/load format means that you can use MATLAB to produce three-dimensional plots of "matrices".) However, writing input routines often cannot be avoided, and can also be useful for debugging purposes.

The input and output that is used by Meschach is all character-based. Fancy window-based input/output could also be done, but there the problem is more about standards and the many different ways of graphically displaying and inputting matrices and vectors.

There are two styles of input in Meschach. Interactive (from a "tty" in Unix jargon), or "batch" from a file or other input stream. Interactive input has fewer design rules than batch input, but still can be challenging to write well. (A fully featured input routine would really be an editor.) The basic design rules for batch input are:

1. The format produced by the output routine can be input.

2. Comments which begin with a "#" and continue to the end of the line are ignored.

Writing interactive input has a number of traps. For example, the following code looks fairly respectable:

```
int size = -1;
```

```
  ......
do {
    printf("Input size: ");
} while ( fscanf(fp,"%d", &size) != 1 || size <= 0 )
```

The idea here is that the loop is with the prompt `Input size:` is redisplayed until `size` is correctly scanned as input, and is positive. Note that the call to `scanf()` *must* take place before the test `size <= 0` is evaluated. The variable `fp` is the *file pointer* which indicates from which file `fscanf(fp,...)` reads data. The function `fscanf()` ignores leading and trailing blanks, so inserting leading or trailing blanks does not affect the code.

However, what happens if you input the letter "`x`"? The `fscanf()` routine would read the letter, realise that it cannot be part of a number, and put it back on the input stream. The result: the loop is an infinite loop giving the user no chance to take control as nothing beyond the "`x`" is read.

The way to avoid this is to use line–by–line input by means of `fgets()`. Also output to `stderr` instead of `stdout` means that output file re-direction does not prevent interactive input. Here is a better approach.

```
int size;
  ......
do {
    fprintf(stderr,"Input size: ");
    if ( fgets(line,MAXLINE,fp) == (char *)NULL )
        error(E_INPUT,"in_foo");
} while ( sscanf(line, "%d", &size) != 1 || size < 0 );
```

The idea here is to input a line into a character array, and then scan the character array. Since every failure results in a new line being read, it cannot get stuck. Failure to read a line from the file results in an error being raised so end–of–file situations are caught.

When interactively inputting arrays, it is a good idea to let the user (at the keyboard) know where you are in the array at all times. If the user makes a mistake, then re-display the prompt including the current position. Allowing the user to go back to correct mistakes, and then go forward again, helps to prevent the user from becoming too frustrated at the system. And what could be more frustrating than having hit the return key just after you realise that you made a mistake near the end of a large matrix with over a hundred entries? Here is how the code for inputting the entries of a vector allows for forward and backward motion, and printing out old values where necessary.

```
for ( i = 0; i < dim; i++ )
    do {
      redo:
        fprintf(stderr,"entry %u: ",i);
        if ( !dynamic )
            fprintf(stderr,"old %14.9g new: ",vec->ve[i]);
        if ( fgets(line,MAXLINE,fp) == NULL )
```

```
            error(E_INPUT,"ifin_vec");
        if ( (*line == 'b' || *line == 'B') && i > 0 )
        {    i--;    dynamic = FALSE;     goto redo;     }
        if ( (*line == 'f' || *line == 'F') && i < dim-1 )
        {    i++;    dynamic = FALSE;     goto redo;     }
    } while ( *line == '\0' ||
              sscanf(line, "%lf", &vec->ve[i]) < 1 );
```

By the way, there is only one other place (outside the input routines) where a `goto` is used. Note also that an end-of-file signal will result in an error being raised.

The batch input parts of input routines are relatively easy to write. Comments can be skipped over by using `skipjunk(fp);` and if an error in the input occurs, then an error should be raised. There is no need to try to re-read the input stream. The error handler may try to skip the input until some marker is reached, but this is up to the programmer. Apart from that, all that is necessary is to have enough `fscanf()` calls to skip over the markers that are printed by the output routine. For example, `fscanf(fp,"Foo:");` will skip over the header produced by the `fout_foo()` routine above. Ignoring the return value of `fscanf()` for *this purpose* is acceptable — the result is a less temperamental input routine.

8.2.6 Resizing

Resizing objects is an operation that cannot be done to all data structures, such as those involving hairy user-defined objects and functional arguments. However, allocated arrays can be resized by means of the standard library function `realloc()`. There is a macro `RENEW(var, num, type)` in `matrix.h` which calls `realloc()`, and also handles `NULL` values of `var`. For example, resizing a `foo` data structure could be done something like this:

```
foo *foo_resize(my_foo, new_size)
foo *my_foo;
int new_size;
{
    double *temp;
    if ( my_foo == (foo *)NULL )
        return get_foo(new_size);
    temp = my_foo->array;
    /* actual re-sizing operation: */
    temp = RENEW(temp, new_size, double);
    if ( temp == (double *)NULL )     /* check for failure */
        error(E_MEM,"foo_resize");
    my_foo->array = temp;
    my_foo->size  = new_size;
    return my_foo;
}
```

Note that the result of `RENEW()` is checked immediately. Also, resetting the size is the last thing that is done.

8.3 How to implement routines

The basic rule that should be used is that the more operations that a user wants to use that are provided by the designer of the library, the less the user has to do and the less likely it will be that the user will make mistakes. Finding a good set of kernel operations for a particular data structure is a crucial problem in good library design. Sometimes, not only the obvious operations should be supplied, but also "support" operations should be implemented. (An example of the need for this can be seen with sparse matrices where there are support routines for setting up the column access paths.) The more complex the data structure, the more support routines you will probably need to write to be able to effectively and efficiently use that data structure. Efficiency will often lead to additional routines. For example, even though there are routines for adding vectors `v_add()`, and for computing scalar multiples of vectors `sv_mlt()`, it is more efficient to use the "multiply and add" routine `v_mltadd()` than to use the add and scalar multiply routines separately.

8.3.1 Design for debugging

Arguments should be checked for consistency, except possibly at the lowest level(s) of the library. At the lowest levels it may not be worth doing the checking and losing efficiency. But at almost all other levels which deal with more time-consuming and complex operations, it is well worth checking the arguments. You probably should check at least that

1. none of the input arguments are `NULL`.

2. the sizes of the arguments are compatible.

For example, in a function `foo_bar()`, the following checking should be done:

```
foo *foo_bar(my_foo1, my_foo2, result_foo)
foo *my_foo1, *my_foo2, *result_foo;
{
    /* check that operands are not NULL */
    if ( my_foo1 == (foo *)NULL || my_foo2 == (foo *)NULL )
        error(E_NULL,"foo_bar");
    /* check that they have compatible sizes */
    if ( my_foo1->size != my_foo2->size )
        error(E_SIZES,"foo_bar");
      ......
}
```

Detailed checking for self-consistency of a data structure is not usually necessary; if the programmer using the library is using it properly, then they shouldn't have much opportunity to mess up the data structure. Of course, the library shouldn't mess up the data structure either. If debugging using a good and thorough output routine is not sufficient to debug the library, then maybe a function that checks internal consistency should be written. However, the checking function would probably be most effective when used to help to debug the library than as an automatic argument check.

An example of detailed argument checking that is not worthwhile is checking that a matrix is symmetric before a Cholesky factorisation. If detailed checking of this kind is wanted, then a checking routine would be written, such as a currently non-existent `chk_symm()` function.

There are a number of macros that have been written for error handling which work in conjunction with the function `ev_err()` (short for "evaluation error") in the file `err.c`. The first is clearly the `error()` macro, which calls `ev_err()` with the `__FILE__` and `__LINE__` macros so that the file and line number where the error was raised can be printed out. The file `err.c` and the error-handling macros in `matrix.h` are independent of the rest of the library, and can be used separately.

A tool that is useful for debugging is to use

```
tracecatch(code_to_execute,"function");
```

The effect of this macro is that if `code_to_execute` raises an error, then once the error is processed (which usually means printing out an error message) the error is re-raised at the place of the `tracecatch()`. If the body of each function (excluding the usual initial argument checks) is enclosed in a `tracecatch()`, then what is effectively a stack backtrace would be printed when an error occurs, indicating what functions were active when the error occurred.

A related macro is `catchall(code_to_execute,error_code)`. This macro executes `code_to_execute` normally, but if this raises an error, then `error_code` is executed. This can be used to print out particular information that might be the cause (or result) of the error. You can put a line containing

```
error(_err_num,"catchall");
```

at the end of `error_code` to re-raise the error, and continue the stack backtrace if desired.

For more information about designing for debugging, see §8.6 on debugging.

8.3.2 Workspace

In most Fortran libraries, routines using extra memory require workspace arguments to be passed to the routine. The programmer using the library has to pass a workspace array of a particular size (which the user has to work out before-hand). With C's memory allocation/de-allocation facilities this is not necessary in C, though sometimes it might be useful.

Passing workspace arrays adds to the complexity of using a function, and is usually a headache for the user. Getting the workspace size right is also a way in which errors can occur.

To avoid having to pass workspace arrays, there are two main approaches to making the necessary workspace available. The first is to allocate the workspace on entry (as soon as its size can be worked out) and deallocated on exiting the function. The second is to have a `static` local array which is first allocated and then reallocated.

The first approach keeps the memory available only for as long as is necessary. This is more efficient in memory, but less efficient in time as the workspace has to be reallocated every time the routine is called. The second approach keeps the workspace memory, and so is less memory efficient, but is more time efficient. In one sense, the two methods are two extremes of a range of "compromises" between memory efficiency and time efficiency.

Here's one way of setting up the second sort of internal workspace:

```
foo *foo_bar(...)
{
      ......
    static double *wkspace = NULL;
    static int    wksize = 0;
      ......
    new_wksize = .... ;
    if ( wkspace == (double *)NULL )
      wkspace = (double *)calloc(new_wksize,sizeof(double));
    else if ( wksize < new_wksize )
      wkspace = (double *)realloc(wkspace,
                                  new_wksize,sizeof(double));
    /* check results of calloc() or realloc() before use! */
    if ( wkspace == (double *)NULL )
      error(E_MEM,"foo_bar");
    wksize = new_wksize;
      ......
}
```

(Note that the initialisation of `wkspace` and `wksize` are unnecessary as un-initialised `static` variables are initialised to zero or `NULL`.) This sort of approach is even more convenient with self-contained data structures which can be resized as needed, such as the vectors in the Meschach library:

```
foo *foo_bar(...)
{
      ......
    static VEC *wkspace = VNULL;
      ......
    new_wksize = .... ;
```

```
    wkspace = v_resize(wkspace,new_wksize);
      ......
}
```

Both of these approaches for workspace have their limits.

However, in Meschach, the "compromise" between memory and time efficiency is put in the hands of the user. This involves "registering" workspace arrays so that they can be freed on request by a call outside of the function where the static workspace variable is defined. Registering a static variable is easy:

```
foo *foo_bar(...)
{
      ......
    static VEC *wkspace = VNULL;
      ......
    new_wksize = .... ;
    wkspace = v_resize(wkspace,new_wksize);
    MEM_STAT_REG(wkspace,TYPE_VEC);
      ......
}
```

Note that you can only register *static* variables. If you try to register an automatic variable, the program will most likely crash. There is no way that the variable can be checked for whether it is static or not.

There is a "workspace group number" or "mark" that must be set before (in the dynamic sense, not necessarily in the code sequence) a workspace variable is registered. When a static workspace variable is registered, it is "marked" as belonging to the current workspace group or "mark". This "mark" can be set by, for example,

```
mem_stat_mark(1);
```

This call is usually made in the main calling routine before any routines using static workspace variables are called. The "mark" can be changed by calling `mem_stat_mark()` with a new "mark" or "group number". All of the static workspace variables registered with a particular "mark" can be deallocated and their memory freed with a call `mem_stat_free(mark)`. Note that this unsets the "mark".

Examples of how the `mem_stat_..()` routines work are in chapter 2.

8.3.3 Incorporating user-defined types into Meschach

Meschach 1.2 provides a number of facilities to track memory usage and to control the allocation and deallocation of static workspace arrays. User-defined data structures can be incorporated into these mechanisms so that it can track memory usage and free up workspace variables for your own data structures.

Since related data structures are often defined together, the information about the data structures is passed to the `mem_info_...()` and `mem_stat_...()` routines

by a collection of arrays containing the names of the types, the `.._free()` functions for these data structures, and an array of `long`'s for storing information about the amount of memory used by the various data structures. This collection of arrays is called a *list*, and it describes a family of types. Each family of types known to Meschach has its own list number; the family of standard Meschach types has zero as its list number.

Here is an example taken from `memtort.c`. First there are the definitions:

```
/* the number of a new list */
#define FOO_LIST 2

/* type numbers */
#define TYPE_FOO_1    1
#define TYPE_FOO_2    2

/* new types */
typedef struct {
   int dim;
   int fix_dim;
   Real (*a)[10];
} FOO_1;

typedef struct {
  int dim;
  int fix_dim;
  Real (*a)[2];
} FOO_2;
```

The arrays which contain the information are:

```
char *foo_type_name[] = {
   "nothing",
   "FOO_1",
   "FOO_2"  };

#define FOO_NUM_TYPES \
   (sizeof(foo_type_name)/sizeof(*foo_type_name))

int (*foo_free_func[FOO_NUM_TYPES])() = {
   NULL,
   foo_1_free,
   foo_2_free  };

static MEM_ARRAY foo_info_sum[FOO_NUM_TYPES];
```

Note that the type number `TYPE_FOO_1` and `TYPE_FOO_2` correspond to the position their type names and `.._free()` functions have in the arrays. This list of types is made known to the Meschach routines by the call

```
mem_attach_list(FOO_LIST, FOO_NUM_TYPES, foo_type_name,
                foo_free_func, foo_info_sum);
if ( ! mem_is_list_attached(FOO_LIST) )
    printf("Error: list FOO_LIST is not attached\n");
```

which should be at the beginning of the `main(...)` routine.

Knowing that certain types exists is a start, but to track memory usage, the routines that perform memory allocation, deallocation and resizing need to keep the Meschach system informed about changing memory usage. For example, in `foo_1_get()`:

```
FOO_1 *foo_1_get(dim)
int dim;
{
   FOO_1 *f;

   if ((f = (FOO_1 *)malloc(sizeof(FOO_1))) == NULL)
     error(E_MEM,"foo_1_get");
   else if (mem_info_is_on())
   {
     mem_bytes_list(TYPE_FOO_1,0,sizeof(FOO_1),FOO_LIST);
     mem_numvar_list(TYPE_FOO_1,1,FOO_LIST);  /* 1 more */
   }
   f->dim = dim;
   f->fix_dim = 10;
   if ((f->a = (Real (*)[10])
              malloc(dim*sizeof(Real [10]))) == NULL)
      error(E_MEM,"foo_1_get");
   else if (mem_info_is_on())
      mem_bytes_list(TYPE_FOO_1,0,
                     dim*sizeof(Real [10]),FOO_LIST);

   return f;
}
```

The routine that actually notifies the Meschach system about the change in the *amount* of memory usage is `mem_bytes_list()`, and the routine that notifies Meschach about the *number* of allocated structures is `mem_numvar_list()`. For `mem_bytes_list()` the first argument is the type number, the second is the old size in bytes, the third is the new size in bytes, and the last parameter is the list number of the family of types. It is not important that the absolute values of old and new sizes are correct, other than being non-negative; rather it is the difference between them

that is important. For `mem_numvar_list()` the *change* in the number of allocated structures is passed.

The corresponding `.._free()` routine also needs to call `mem_byte_list()`:

```
int foo_1_free(f)
FOO_1 *f;
{
   if ( f != NULL) {
      if (mem_info_is_on())
      {
          mem_bytes_list(TYPE_FOO_1,
              sizeof(FOO_1)+f->dim*sizeof(Real [10]),0L,2);
          mem_numvar_list(TYPE_FOO,-1,2);  /* 1 less */
      }
      free(f->a);
      free(f);
   }
   return 0;
}
```

Similarly, `.._resize()` routines need to call `mem_bytes_list()` if there is any actual memory allocation, deallocation or resizing. If the argument is NULL, then the main `.._get()` routine should be called; otherwise there is no change in the number of `FOO_1` structures, and so there is no need to call `mem_numvar_list()`. Merely re-arranging the internal structure doesn't have to be reported via `mem_bytes_list()`.

User-defined data structures can be used as static workspace arrays, just like the standard Meschach data structures. They can be registered as workspace variables just like the standard Meschach data structures, except that the list number of the family of types needs to be given, and is positive. For example,

```
hairy1(...)
{
    static FOO_1 *f;   /* initially NULL */
      ......
    if ( ! f )  f = foo_1_get();  /* allocate if f NULL */
    /* ...or could use a .._resize() routine */
    mem_stat_reg_list(&f, TYPE_FOO_1, FOO_LIST);
      ......
}
```

These static workspace variables will be deallocated using a call to `mem_stat_free_list()`. Note that unlike the `MEM_STAT_REG()` macro, you have to explicitly take the address of `f`; `MEM_STAT_REG()` is a macro.

This is an example of how to use this to free `f`:

```
main(...)
```

```
{
      ......
    mem_stat_mark(1);
      ......
    for ( i = 0; i < 1000; i++ )
        hairy1(...);
      ......
    /* now free up FOO_1 and FOO_2 workspace structures */
    mem_stat_free_list(1,FOO_LIST);
    /* now free up standard Meschach workspace structures */
    mem_stat_free(1);
    /* which is equivalent to: mem_stat_free_list(1,0); */
      ......
}
```

If you have a family of types, where creating one type involves creating another in the same family, care should be taken to avoid double counting. In this case a "main-type" contains a pointer to a "sub-type", say. There are two ways around this: one is to call `mem_bytes_list()` and `mem_numvar_list()` only for those parts of the data structure not in the "sub-type". The other, more complex approach, is to inform the routines that create the "sub-type" that it is created as part of the "main-type", and to account for all of the memory and structure allocation as part of the "main-type". This second approach is only really of use if the "sub-type" is understand as being only of use as part of the larger "main-type". This approach *is* used in Meschach for sparse rows in sparse matrices. Stand alone sparse rows can be created, destroyed, etc., but are almost never used in this way.

8.3.4 Output and object resizing

While it is quite possible to create a new data structure and allocate new memory for every new result, this reduces the efficiency of the algorithms and rapidly loses memory. As there is no garbage collection in C, the memory that is "lost" is unrecoverable. Also, numerical analysts and applications people are often working with large problems on the limits of the machine(s) that they use. So it is rather important that the programmer using a library will want control over memory allocation, or at least over the allocation of the large objects.

The standard used in Meschach is that whenever a large or composite object results from a computation, there is an extra parameter in which the result is to be put. As before, this parameter is a pointer to a data structure. If this pointer is `NULL`, then the output data structure is allocated and initialised. This allows for the creation of the output when the user desires, but still gives control over memory allocation.

If the output object is not `NULL`, but is not of the correct size, then a resizing function should be used. An example of this might be:

```
foo *foo_bar(my_foo1, my_foo2, out_foo)
```

```
foo *my_foo1, *my_foo2, *out_foo;
{
      ......
    if ( out_foo == NULL  || out_foo->size != my_foo1->size )
        out_foo = foo_resize(out_foo, my_foo1->size);
      ......
}
```

The call to `get_foo()` is not necessary if the resizing function (here `foo_resize()`) allocates and initialises a new `foo` data structure if it is passed a `NULL`.

If you cannot write a resizing function, then raise an error if the sizes are incompatible. In such a case, it is better to get the user to create the thing with the right size to start with. The alternative approach to that of creating a new object when the output data structure has the wrong size will result in "memory leaks" with code such as

```
foo *my_foo1, *my_foo2, *out_foo;
  ......
out_foo = foo_bar(my_foo1,my_foo2,out_foo);
```

If `out_foo` is the wrong size, then creating a new data structure will result in the original `out_foo` data structure being lost, and being replaced by a newly created data structure. This memory would be lost until the program terminates.

To repeat: the output parameter should be resized if it is the wrong size, or raise an error.

8.4 User-defined functions

When data structures of a conventional sort cannot explicitly and easily cope with the complexities of a problem, it is usual for programmers to use functional parameters — especially numerical and scientific programmers. In C these are not difficult to use: just remember that you are actually passing pointers to functions, rather than the code itself!

A standard example used is working out the definite integral

$$\int_a^b f(x)\,dx$$

using a quadrature (integration) rule of some kind. The function that computed the integral might look like this:

```
double integrate(f, a, b, n)
double (*f)();      /* function to integrate */
double a, b;        /* lower and upper limits */
int    n;           /* number of sub-intervals to use */
{
    int i;
```

```
    double sum;
      ......
    sum += (*f)(a+i*(b-a)/n);
      ......
    return sum/n;
}
```

Then `integrate(sin, 0.0, PI, 100)` would give an approximation to $\int_0^\pi \sin(x)\,dx$. If you want to integrate a particular function, then you have to write it yourself. So far, so good. However, the function `f` in `integrate()` must be a function of only one variable — the variable that is integrated. Usually functions have parameters, and usually those parameters are changed from run to run, or call to call. These parameters are outside this model of how `f` works as a function.

The standard way of dealing with this in C is to set up some global variables containing the parameters and then modifying them as necessary from run to run, or call to call, of `integrate()`. This is not a very good way of dealing with parameters: as a general rule, the more global variables, and "pathological" (i.e. hidden) connections between routines, the more unpredictable a piece of code becomes.

The alternative that we would recommend here is to allow for an extra parameter in `f` of the type `void *`. This could be a pointer to a `struct` containing the relevant parameters, or even much larger, more complex, data structures. The code for the integration function would then look like:

```
double integrate2(f, fparams, a, b, n)
double (*f)();      /* function to integrate */
void   *fparams;    /* extra parameters for f */
double a, b;        /* lower and upper limits */
int    n;           /* number of sub-intervals to use */
{
      ......
    sum += (*f)(fparams,a+i*(b-a)/n);
      ......
}
```

Then, for example, for a general quadratic $f(x) = ax^2 + bx + c$, the following code could be used:

```
struct PQ { double a, b, c; };

double quadratic(params, x)
struct PQ *params;
double x;
{   /* using Horner's nested multiplication scheme */
    return x*(params->a*x + params->b) + params->c;
}
```

This could be used in something like the following:

```
{
    struct PQ par_quad;
      ......
    par_quad.a = 5.0;
    par_quad.b = -3.7;
    par_quad.c = 101.433445;
    printf("Integral = %g\n",
      integrate2(quadratic,(void *)&par_quad,0.0,1.0,100));
      ......
}
```

What if you want to integrate a function that really is just of one variable, with no additional parameters? At the cost of an extra layer of function calls it can be done using

```
double apply(f, x)
double (*f)(), x;
{       return (*f)(x);       }
```

so that $\int_0^\pi \sin(x)\,dx$ can be computed (approximately) by the call

```
int_val = integrate2(apply, sin, 0.0, PI, 100);
```

Ideally, both styles should probably be implemented, but the additional flexibility in having a `void *` parameter for functional parameters is well worth the effort of writing them into a library.

This approach is an alternative to the "*reverse communication*" path that is taken in most Fortran libraries. The disadvantage of reverse communication is the complexity needed to handle a routine that uses reverse communication. There are possibly some particularly complex things for which reverse communication is still the best technique. However, implementing a number of separate routines which act on the same data structure might still be a more convenient way of doing things than reverse communication.

8.5 Building the library

Building up a library of routines to be generally useful, or even to solve a single problem, usually takes a few steps. The best advice here is summed up in the term "*incremental testing*". As routines are added to the collection that forms your library or problem solver, they should be tested. There is very little more disheartening than to spend a week trying to find an unexpected bug buried somewhere deep in the code. Keep the argument checking and debugging tools (e.g. print-out routines) around — they are still useful.

Build new data structures as you need them, and test them and their routines before going on to the next level. Even if you decide later that you would prefer to use a different way of doing the sub-problems, the interface to a modified data structure should probably stay pretty much the same as for the original data structure used. Use previous (debugged) data structures and their routines. This prevents a lot of errors and simplifies programming; they start to work more like building blocks than isolated bits of code. For example, if you are a control systems designer, you might want to have a "rational function" data structure representing ratios of polynomials:

$$R(x) = \frac{P(x)}{Q(x)}.$$

Each of the polynomials $P(x)$ and $Q(x)$ can be represented by vectors of coefficients. The data structure for $R(x)$ might be

```
typdef struct { int deg_P, deg_Q;     VEC *P, *Q; } rational;
```

There is some redundancy in this data structure since `deg_P` should be one more than the dimension of the vector `P`. Whether or not this degree of redundancy is acceptable will depend on whether users of the library will want to have direct access to `deg_P` and `deg_Q`, and whether routines are written to rely on `deg_P` and `deg_Q` or `P->dim` and `Q->dim`.

Before defining the operations to be performed on objects of type `rational`, the basic operations on polynomials should be defined: adding, subtracting, multiplying and normalising polynomials; synthetic division of polynomials, and evaluating a polynomial at a real or complex value of x. Some of these can be defined in terms of operations on `VEC`'s. Then the operations on rational functions can be defined in terms of the polynomial operations.

8.5.1 Numerical aspects

An important issue in numerical computations is that of the accumulation and magnification of roundoff error. That is, the computations should be *numerically stable*, and avoid accumulating or magnifying roundoff errors. While it can, in general, be very difficult to predict the effects of roundoff error, some situations are more likely to lead to bad results than others. For example, polynomials can be rather badly behaved in this regard. An example can be found in K. Atkinson's *Introduction to Numerical Analysis*, 1st Edition, pp. 80–84 (1979). The designers of MATLAB's polynomial root finding algorithm in fact avoid polynomials altogether in their approach: they find instead the eigenvalues of the companion matrix of the polynomial $p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$,

$$C = \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & 0 \\ 0 & 0 & 1 & \dots & \dots & 0 \\ 0 & 0 & 0 & \ddots & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ -a_0 & -a_1 & -a_2 & \dots & \dots & -a_{n-1} \end{bmatrix}.$$

Since rootfinding of polynomials can be badly conditioned, setting up the companion matrix would lead to an equally ill conditioned eigenproblem. However, generally eigenproblems are apparently less likely to suffer such ill conditioning as extreme as for polynomial rootfinding. A control system designer might take this as a hint and deal with control systems in $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$ matrix form, using companion matrices to represent polynomial systems.

A rule of thumb that seems to work for a great many applications for keeping good numerical stability, is to keep intermediate computations in a form close to the form of the original data. Elaborate transformations might give exactly equivalent problems, but the introduction of noise and rounding errors can make some methods far better or far worse than others.

There are a number of hard-won rules which numerical analysts have discovered over the years (and re-discovered far too many times!). In relation to matrix computations the oldest and most important one is:

Don't compute the inverse of a matrix if all you want is to solve some equations.

Computing the inverse of a matrix does not make any of the subsequent calculations for solving a system of equations faster than using its LU factors, the accuracy is slightly worse usually, and it takes longer to compute the inverse in the first place. For sparse matrices it is even more important. The LU factors of a sparse matrix are usually fairly sparse, but the inverse is almost never sparse for practical problems. Forming the inverse of a large sparse matrix may be an impossible undertaking on a machine, even though solving the system of equations can be accomplished quite quickly on that same machine.

Another problem that one would do well to avoid is "finding all eigenvectors of a large matrix". Finding all the eigen*values* of a large symmetric matrix is not an unreasonable task (use the Lanczos routines). Generating the eigenvectors can then often be done using inverse iteration (see K. Atkinson's *An Introduction to Numerical Analysis*, 1st Edition, pp. 548–553 (1979)) on demand for large sparse matrices. Remember: just storing all the eigenvectors of a 10 000 × 10 000 matrix will take up 800Mbyte — not a small amount on any current computer!

8.6 Debugging

While the `error()` macro will save many types of errors, it cannot save you from all of them. If your program is crashing, then put

```
setbuf(stdout,(char *)NULL);
```

at the start of your `main()` program (at least on Unix systems) to ensure that you are seeing all your output. Use liberal `printf()` and `..._output()` calls to check the values of your data types, and to "checkpoint" your program. This also means you should *write* `..._output()` routines for any new data structures that you define.

Potential bugs can sometimes be spotted by automatic tools, such as `lint` on Unix machines, which can detect things like unreachable code, unportable pointer conversions, and function argument incompatibilities for non-ANSI C code. The GNU compiler `gcc` can detect potential portability and related problems in a similar way to `lint` if you use the `-Wall` option (which reports *all* warnings).

Try using open-ended test programs so that you can input any object of a particular data structure, and checking the result. Avoid tests which only give you a "yes/no" answer. If it got the answer by chance, then it has a 50% chance of fooling you. Compute residuals. For systems of equations this means printing out $\|Ax - b\|$; for eigenvalues/eigenvectors this means $\|Ax - \lambda x\|/\|x\|$; for solving $f(x) = 0$ this means printing $\|f(x)\|$; for least squares problems this means printing $\|A^T(Ax - b)\|$. Whatever your problem is, try to compute sufficient information that it is easy to *verify* the complete computed results. For optimisation problems, this would mean checking the first order necessary conditions at least. Use the routines that you have available, not just for doing the computations, but also for helping you to do the verification as well (such as `v_norm2()`).

If a program has a problem, try to find out *where* the problem is. If the program crashes at an unknown point for some reason, put in checkpoints in you main program. Once you've narrowed down the range in which the error occurs there to a single statement, the chances are that it will be a function call. "Open up" that function, putting in checkpoint statements, and printing any relevant quantities until the problem can be located in that function, continuing until the problem is localised.

8.6.1 Memory allocation bugs

These bugs occur when the memory allocation heap has been corrupted. This can occur when an allocated array is written to at an invalid location, or `free()` is called with an invalid address (that is, an address that wasn't returned by `malloc()`, `calloc()` or `realloc()`). Either way the memory heap's headers are corrupted. The results of memory heap corruption can be unpredictable, sometimes resulting in the program crashing, sometimes resulting in apparently "intermittent" bugs. The rules given above for localising bugs don't work for these sorts of bugs, since the corruption is not evident until a call to `malloc()` or `free()` etc. Most programmers could use some help with these sorts of memory heap corruption bugs.

As of version 1.2 of Meschach, there are some built-in routines for keeping a watch on memory usage which are `mem_info_on()`, `mem_info_file()` and `mem_info_type()`. These routines respectively turn the "`mem_info_...`" system on or off, printout a summary of the memory used in Meschach data structures to a file or stream, and return the amount of memory used for a particular Meschach data type. They can be used as follows to check for memory leaks, here in a function `hairy()`:

```
main()
{
        .....
    mem_info_on(TRUE);
```

```
    hairy(....);
    mem_info_f(stdout);  /* print out summary */
    printf("Memory used for vectors by hairy(): %d\n",
           (int)mem_info_type(TYPE_VEC));
}
```

If you get negative amounts of memory in use then something has gone wrong. If static workspace arrays are used you may need to use the `MEM_STAT_REG()` and `mem_stat_...()` routines. The routine `mem_stat_dump()` can also be useful in determining the status of workspace variables.

If you suspect that there is a subtle memory over-writing error, then you should use a package that replaces the standard (fast) memory allocation package `malloc()` and `free()` etc, with something like the public domain package by Conor P. Cahill (uunet address: `uunet!virtech!cpcahil`). This provides a drop-in replacement for the standard library routines: compile your program as in

```
cc  -o my_prog my_prog.c ..... meschach.a libmalloc.a -lm
```

and use his `malloc_chain_check(0)` to check for corruption of the `malloc()` heap. There may be other "debugging" memory allocation/deallocation packages that you have access to.

There are also tools that come with the GNU C compiler for tracking bugs that affect the memory heap.

These are also useful tools to determine if your program has a "memory leak" that results in memory being allocated and then thrown away, although `mem_info()` should be enough to track down memory leaks.

8.6.2 If all else fails

Beyond these things, there are two ways of dealing with these problems.

1. Look at the source code. No-one's code is perfectly readable but we believe that it is not too difficult to follow, especially for experienced C programmers.

2. Contact us. This is best done by e-mail; a current e-mail address is

   ```
   david.stewart@anu.edu.au
   zbigniew.leyk@anu.edu.au
   ```

We cannot guarantee to even look at your problem as we are not employed as programmers, but as academic mathematicians. Our e-mail addresses are also subject to change without notice.

8.7 Suggestions for enthusiasts

There are a number of areas which seem to be particularly ripe for additions. Porting to C++ and making use of classes and operator overloading in itself would be a useful project.

Sets can be implemented a number of ways using permutations and/or integer vectors.

Some extensions that have been considered (and maybe something will be released eventually) include linear programming extensions, ODE solvers, and maybe some nonlinear equation solvers. But there is much more that can be done. One item conspicuously absent are sparse matrix re-ordering routines. A good minimum degree algorithm should be implemented for Meschach.

8.8 Pride and Prejudice

This section is about our own personal beliefs and prejudices. These opinions are nobody's but our own. If you find them obnoxious or frivolous, remember, you have been warned!

8.8.1 What about Fortran 90?

We might have started thinking about it if it had been around when we started on this project six years ago. As it is, we still haven't seen a Fortran 90 compiler, although we have seen a very near miss in the Connection Machine Fortran.

Learning Fortran 90, especially the parts of interest to us, would involve learning a whole new language. When it comes to pointers, dynamic memory allocation and de-allocation, structures/records etc, it *is* a completely new language. We doubt that many future users of Fortran 90 will use the full power of the language for a good many years yet. And then, the people who do make full use of it will be people who have programmed before in C, C++, Ada, Modula-2 (or perhaps Modula-3) and the like. They will know the benefit of using these advanced features.

Porting it to Fortran 90 might be a possibility someday. We don't want to do that job. Porting to C++ would be a much more useful task in the near future. (Meschach has already been used within a C++ program.)

8.8.2 Why should people writing numerical code care about good software?

Numerical analysts and scientists often write unreadable programs.

One of us remembers trying to translate Bill Gear's DIFSUB program from Fortran 77 to C. And failed. He got lost in the spaghetti. So he looked at his description of what it was supposed to do, and implemented *that*. And the result worked.

Quite a few older programmers find this situation normal or even desirable, almost as a sort of job security, or a sense of machismo: "Real programmers don't document

their code; if it was hard to write, it should be hard to read." It wasn't academic politics that made this attitude unacceptable in any modern computer science department, but practical experience combined with the urgency of the "software crisis" of the late sixties and seventies. This "software crisis" still hasn't gone away; big, complex systems (such as commercial and military aircraft) rely more than ever on good, bug-proof software.

On a more personal level, not being a masochist, we much prefer being able to write programs and modify them without having to remember to juggle a dozen flags, set and reset global variables, and so on. Modifying programs is the nature of research. You need to be able to modify the code to do things in different, but meaningful, ways. Trying to do this without helpful software underneath is painful; usually we find that the same underlying operation needs to be re-implemented for the nth time.

Routines which are general purpose, and are designed with flexibility in mind, make an enormous difference when it comes to programming and designing new algorithms. This is why people use numerical libraries. And that is why we wrote this library. The state of the art moves on, and instead of waiting for one's favourite numerical library to be updated with something you would like to see, this library enables you to implement new algorithms. The code is there for inspection, use and modification. (But, please, don't modify old routines unless they have bugs in them — real bugs — but modify the code to create *new* routines.) In doing so, you can provide a platform for further development by yourself or others. Thus the computer can be used not just to crunch numbers, but also to improve your "personal productivity" as the advertisements say. After all, if computers can't make life easier, or more productive, what good are they?