```
    if (julian >= IGREG) {          Cross-over to Gregorian Calendar produces this correc-
        jalpha=(long)(((float) (julian-1867216)-0.25)/36524.25);        tion.
        ja=julian+1+jalpha-(long) (0.25*jalpha);
    } else if (julian < 0) {     Make day number positive by adding integer number of
        ja=julian+36525*(1-julian/36525);       Julian centuries, then subtract them off
    } else                                      at the end.
        ja=julian;
    jb=ja+1524;
    jc=(long)(6680.0+((float) (jb-2439870)-122.1)/365.25);
    jd=(long)(365*jc+(0.25*jc));
    je=(long)((jb-jd)/30.6001);
    *id=jb-jd-(long) (30.6001*je);
    *mm=je-1;
    if (*mm > 12) *mm -= 12;
    *iyyy=jc-4715;
    if (*mm > 2) --(*iyyy);
    if (*iyyy <= 0) --(*iyyy);
    if (julian < 0) iyyy -= 100*(1-julian/36525);
}
```

(For additional calendrical algorithms, applicable to various historical calendars, see [8].)

CITED REFERENCES AND FURTHER READING:

Harbison, S.P., and Steele, G.L., Jr. 1991, *C: A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Kernighan, B.W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Jones, R., and Stewart, I. 1987, *The Art of C Programming* (New York: Springer-Verlag). [3]

Hoare, C.A.R. 1981, *Communications of the ACM*, vol. 24, pp. 75–83.

Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1986, *The C++ Programming Language* (Reading, MA: Addison-Wesley). [5]

Borland International, Inc. 1989, *Turbo Pascal 5.5 Object-Oriented Programming Guide* (Scotts Valley, CA: Borland International). [6]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [7]

Hatcher, D.A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507. [8]

1.2 Some C Conventions for Scientific Computing

The C language was devised originally for systems programming work, not for scientific computing. Relative to other high-level programming languages, C puts the programmer "very close to the machine" in several respects. It is operator-rich, giving direct access to most capabilities of a machine-language instruction set. It has a large variety of intrinsic data types (short and long, signed and unsigned integers; floating and double-precision reals; pointer types; etc.), and a concise syntax for effecting conversions and indirections. It defines an arithmetic on pointers (addresses) that relates gracefully to array addressing and is highly compatible with the index register structure of many computers.

Portability has always been another strong point of the C language. C is the underlying language of the UNIX operating system; both the language and the operating system have by now been implemented on literally hundreds of different computers. The language's universality, portability, and flexibility have attracted increasing numbers of scientists and engineers to it. It is commonly used for the real-time control of experimental hardware, often in spite of the fact that the standard UNIX kernel is less than ideal as an operating system for this purpose.

The use of C for higher level scientific calculations such as data analysis, modeling, and floating-point numerical work has generally been slower in developing. In part this is due to the entrenched position of FORTRAN as the mother-tongue of virtually all scientists and engineers born before 1960, and most born after. In part, also, the slowness of C's penetration into scientific computing has been due to deficiencies in the language that computer scientists have been (we think, stubbornly) slow to recognize. Examples are the lack of a good way to raise numbers to small integer powers, and the "implicit conversion of `float` to `double`" issue, discussed below. Many, though not all, of these deficiencies are overcome in the ANSI C Standard. Some remaining deficiencies will undoubtedly disappear over time.

Yet another inhibition to the mass conversion of scientists to the C cult has been, up to the time of writing, the decided lack of high-quality scientific or numerical libraries. That is the lacuna into which we thrust this edition of *Numerical Recipes*. We certainly do not claim to be a complete solution to the problem. We do hope to inspire further efforts, and to lay out by example a set of sensible, practical conventions for scientific C programming.

The need for programming conventions in C is very great. Far from the problem of overcoming constraints imposed by the language (our repeated experience with Pascal), the problem in C is to choose the best and most natural techniques from multiple opportunities — and then to use those techniques completely consistently from program to program. In the rest of this section, we set out some of the issues, and describe the adopted conventions that are used in all of the routines in this book.

Function Prototypes and Header Files

ANSI C allows functions to be defined with *function prototypes*, which specify the type of each function parameter. If a function declaration or definition with a prototype is visible, the compiler can check that a given function call invokes the function with the correct argument types. All the routines printed in this book are in ANSI C prototype form. For the benefit of readers with older "traditional K&R" C compilers, the *Numerical Recipes C Diskette* includes two complete sets of programs, one in ANSI, the other in K&R.

The easiest way to understand prototypes is by example. A function definition that would be written in traditional C as

```
int g(x,y,z)
int x,y;
float z;
```

becomes in ANSI C

```
int g(int x, int y, float z)
```

A function that has no parameters has the parameter type list `void`.

A function declaration (as contrasted to a function definition) is used to "introduce" a function to a routine that is going to *call* it. The calling routine needs to know the number and type of arguments and the type of the returned value. In a function declaration, you are allowed to omit the parameter names. Thus the declaration for the above function is allowed to be written

```
int g(int, int, float);
```

If a C program consists of multiple source files, the compiler cannot check the consistency of each function call without some additional assistance. The safest way to proceed is as follows:

- Every external function should have a single prototype declaration in a header (`.h`) file.
- The source file with the definition (body) of the function should also include the header file so that the compiler can check that the prototypes in the declaration and the definition match.
- Every source file that calls the function should include the appropriate header (`.h`) file.
- Optionally, a routine that calls a function can also include that function's prototype declaration internally. This is often useful when you are developing a program, since it gives you a visible reminder (checked by the compiler through the common `.h` file) of a function's argument types. Later, after your program is debugged, you can go back and delete the supernumary internal declarations.

For the routines in this book, the header file containing all the prototypes is `nr.h`, listed in Appendix A. You should put the statement `#include nr.h` at the top of every source file that contains *Numerical Recipes* routines. Since, more frequently than not, you will want to include more than one *Numerical Recipes* routine in a single source file, we have not printed this `#include` statement in front of this book's individual program listings, but you should make sure that it is present in your programs.

As backup, and in accordance with the last item on the indented list above, we declare the function prototype of all *Numerical Recipes* routines that are called by other *Numerical Recipes* routines *internally* to the calling routine. (That also makes our routines much more readable.) The only exception to this rule is that the small number of utility routines that we use repeatedly (described below) are declared in the additional header file `nrutil.h`, and the line `#include nrutil.h` is explicitly printed whenever it is needed.

A final important point about the header file `nr.h` is that, as furnished on the diskette, it contains both ANSI C and traditional K&R-style declarations. The ANSI forms are invoked if any of the following macros are defined: `__STDC__`, `ANSI`, or `NRANSI`. (The purpose of the last name is to give you an invocation that does not conflict with other possible uses of the first two names.) If you have an ANSI compiler, it is *essential* that you invoke it with one or more of these macros defined. The typical means for doing so is to include a switch like "-DANSI" on the compiler command line.

Some further details about the file nr.h are given in Appendix A.

Vectors and One-Dimensional Arrays

There is a close, and elegant, correspondence in C between pointers and arrays. The value referenced by an expression like a[j] is *defined* to be *((a)+(j)), that is, "the contents of the address obtained by incrementing the pointer a by j." A consequence of this definition is that if a points to a legal data location, the array element a[0] is always defined. Arrays in C are natively "zero-origin" or "zero-offset." An array declared by the statement float b[4]; has the valid references b[0], b[1], b[2], and b[3], but *not* b[4].

Right away we need a *notation* to indicate what is the valid range of an array index. (The issue comes up about a thousand times in this book!) For the above example, the index range of b will be henceforth denoted b[0..3], a notation borrowed from Pascal. In general, the range of an array declared by float a[M]; is a[0..$M-1$], and the same if float is replaced by any other data type.

One problem is that many algorithms naturally like to go from 1 to M, not from 0 to $M-1$. Sure, you can always convert them, but they then often acquire a baggage of additional arithmetic in array indices that is, at best, distracting. It is better to use the power of the C language, in a consistent way, to make the problem disappear. Consider

```
float b[4],*bb;
bb=b-1;
```

The pointer bb now points one location before b. An immediate consequence is that the array elements bb[1], bb[2], bb[3], and bb[4] all exist. In other words the range of bb is bb[1..4]. We will refer to bb as a *unit-offset* vector. (See Appendix B for some additional discussion of technical details.)

It is sometimes convenient to use zero-offset vectors, and sometimes convenient to use unit-offset vectors in algorithms. The choice should be whichever is most natural to the problem at hand. For example, the coefficients of a polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ clearly cry out for the zero-offset a[0..n], while a vector of N data points x_i, $i = 1 \ldots N$ calls for a unit-offset x[1..N]. When a routine in this book has an array as an argument, its header comment always gives the expected index range. For example,

```
void someroutine(float bb[], int nn)
This routine does something with the vector bb[1..nn].
...
```

Now, suppose you want someroutine() to do its thing on your own vector, of length 7, say. If your vector, call it aa, is already unit-offset (has the valid range aa[1..7]), then you can invoke someroutine(aa,7); in the obvious way. That is the recommended procedure, since someroutine() presumably has some logical, or at least aesthetic, reason for wanting a unit-offset vector.

But suppose that your vector of length 7, now call it a, is perversely a native C, zero-offset array (has range a[0..6]). Perhaps this is the case because you disagree with our aesthetic prejudices, Heaven help you! To use our recipe, do you have to copy a's contents element by element into another, unit-offset vector? No! Do you have to declare a new pointer aaa and set it equal to a-1? No! You simply invoke someroutine(a-1,7);. Then a[1], as seen from within our recipe, is actually a[0] as seen from your program. In other words, you can change conventions "on the fly" with just a couple of keystrokes.

Forgive us for belaboring these points. We want to free you from the zero-offset thinking that C encourages but (as we see) does not require. A final liberating point is that the utility file nrutil.c, listed in full in Appendix B, includes functions for allocating (using malloc()) *arbitrary-offset* vectors of arbitrary lengths. The synopses of these functions are as follows:

```
float *vector(long nl, long nh)
```
Allocates a float vector with range [nl..nh].

```
int *ivector(long nl, long nh)
```
Allocates an int vector with range [nl..nh].

```
unsigned char *cvector(long nl, long nh)
```
Allocates an unsigned char vector with range [nl..nh].

```
unsigned long *lvector(long nl, long nh)
```
Allocates an unsigned long vector with range [nl..nh].

```
double *dvector(long nl, long nh)
```
Allocates a double vector with range [nl..nh].

A typical use of the above utilities is the declaration float *b; followed by b=vector(1,7);, which makes the range b[1..7] come into existence and allows b to be passed to any function calling for a unit-offset vector.

The file nrutil.c also contains the corresponding deallocation routines,

```
void free_vector(float *v, long nl, long nh)

void free_ivector(int *v, long nl, long nh)

void free_cvector(unsigned char *v, long nl, long nh)

void free_lvector(unsigned long *v, long nl, long nh)

void free_dvector(double *v, long nl, long nh)
```

with the typical use being free_vector(b,1,7);.

Our recipes use the above utilities extensively for the allocation and deallocation of vector workspace. We also commend them to you for use in your main programs or other procedures. Note that if you want to allocate vectors of length longer than 64k on an IBM PC-compatible computer, you should replace all occurrences of malloc in nrutil.c by your compiler's special-purpose memory allocation function. This applies also to matrix allocation, to be discussed next.

Matrices and Two-Dimensional Arrays

The zero- versus unit-offset issue arises here, too. Let us, however, defer it for a moment in favor of an even more fundamental matter, that of *variable dimension arrays* (FORTRAN terminology) or *conformant arrays* (Pascal terminology). These are arrays that need to be passed to a function along with real-time information about their two-dimensional size. The systems programmer rarely deals with two-dimensional arrays, and almost never deals with two-dimensional arrays whose size is variable and known only at run time. Such arrays are, however, the bread and butter of scientific computing. Imagine trying to live with a matrix inversion routine that could work with only one size of matrix!

There is no technical reason that a C compiler could not allow a syntax like

```
void someroutine(a,m,n)
float a[m][n];              /* ILLEGAL DECLARATION */
```

and emit code to evaluate the variable dimensions m and n (or any variable-dimension expression) each time someroutine() is entered. Alas! the above fragment is forbidden by the C language definition. The implementation of variable dimensions in C instead requires some additional finesse; however, we will see that one is rewarded for the effort.

There is a subtle near-ambiguity in the C syntax for two-dimensional array references. Let us elucidate it, and then turn it to our advantage. Consider the array reference to a (say) float value a[i][j], where i and j are expressions that evaluate to type int. A C compiler will emit quite different machine code for this reference, depending on how the identifier a has been declared. If a has been declared as a fixed-size array, e.g., float a[5][9];, then the machine code is: "to the address a add 9 times i, then add j, return the value thus addressed." Notice that the constant 9 needs to be known in order to effect the calculation, and an integer multiplication is required (see Figure 1.2.1).

Suppose, on the other hand, that a has been declared by float **a;. Then the machine code for a[i][j] is: "to the address of a add i, take the value thus addressed as a new address, add j to it, return the value addressed by this new address." Notice that the underlying size of a[][] does not enter this calculation at all, and that there is no multiplication; an additional indirection replaces it. We thus have, in general, a faster and more versatile scheme than the previous one. The price that we pay is the storage requirement for one array of pointers (to the rows of a[][]), and the slight inconvenience of remembering to initialize those pointers when we declare an array.

Here is our bottom line: We avoid the fixed-size two-dimensional arrays of C as being unsuitable data structures for representing matrices in scientific computing. We adopt instead the convention "pointer to array of pointers," with the array elements pointing to the first element in the rows of each matrix. Figure 1.2.1 contrasts the rejected and adopted schemes.

The following fragment shows how a fixed-size array a of size 13 by 9 is converted to a "pointer to array of pointers" reference aa:

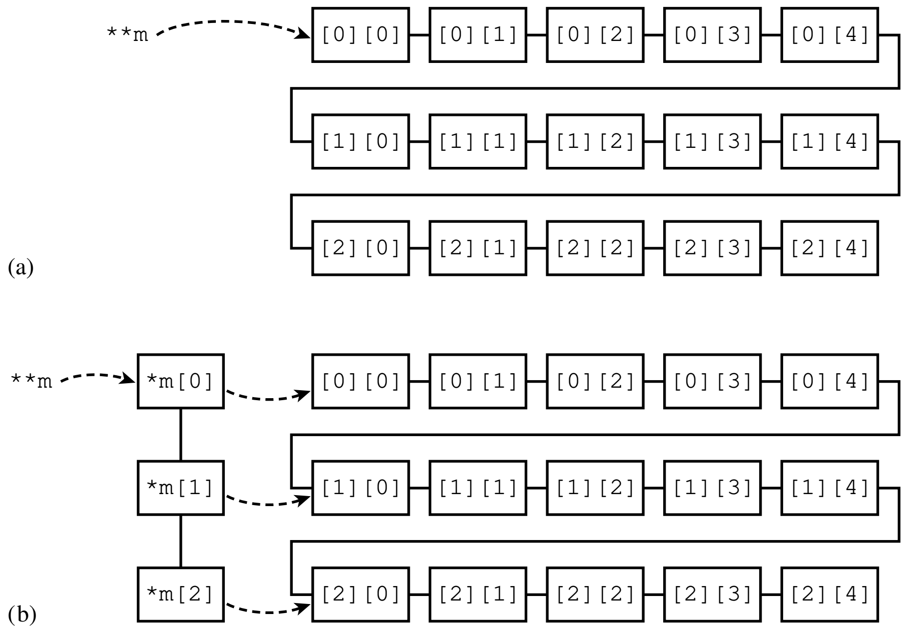

Figure 1.2.1. Two storage schemes for a matrix `m`. Dotted lines denote address reference, while solid lines connect sequential memory locations. (a) Pointer to a fixed size two-dimensional array. (b) Pointer to an array of pointers to rows; this is the scheme adopted in this book.

```
float a[13][9],**aa;
int i;
aa=(float **) malloc((unsigned) 13*sizeof(float*));
for(i=0;i<=12;i++) aa[i]=a[i];        a[i] is a pointer to a[i][0]
```

The identifier `aa` is now a matrix with index range `aa[0..12][0..8]`. You can use or modify its elements *ad lib*, and more importantly you can pass it as an argument to any function by its name `aa`. That function, which declares the corresponding dummy argument as `float **aa;`, can address its elements as `aa[i][j]` *without knowing its physical size.*

You may rightly not wish to clutter your programs with code like the above fragment. Also, there is still the outstanding problem of how to treat unit-offset indices, so that (for example) the above matrix `aa` could be addressed with the range `a[1..13][1..9]`. Both of these problems are solved by additional utility routines in `nrutil.c` (Appendix B) which allocate and deallocate matrices of arbitrary range. The synopses are

```
float **matrix(long nrl, long nrh, long ncl, long nch)
Allocates a float matrix with range [nrl..nrh][ncl..nch].

double **dmatrix(long nrl, long nrh, long ncl, long nch)
Allocates a double matrix with range [nrl..nrh][ncl..nch].

int **imatrix(long nrl, long nrh, long ncl, long nch)
Allocates an int matrix with range [nrl..nrh][ncl..nch].

void free_matrix(float **m, long nrl, long nrh, long ncl, long nch)
Frees a matrix allocated with matrix.
```

```
void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
Frees a matrix allocated with dmatrix.

void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch)
Frees a matrix allocated with imatrix.
```

A typical use is

```
float **a;
a=matrix(1,13,1,9);
...
a[3][5]=...
...+a[2][9]/3.0...
someroutine(a,...);
...
free_matrix(a,1,13,1,9);
```

All matrices in *Numerical Recipes* are handled with the above paradigm, and we commend it to you.

Some further utilities for handling matrices are also included in `nrutil.c`. The first is a function `submatrix()` that sets up a new pointer reference to an already-existing matrix (or sub-block thereof), along with new offsets if desired. Its synopsis is

```
float **submatrix(float **a, long oldrl, long oldrh, long oldcl,
    long oldch, long newrl, long newcl)
Point a submatrix [newrl..newrl+(oldrh-oldrl)][newcl..newcl+(oldch-oldcl)] to
the existing matrix range a[oldrl..oldrh][oldcl..oldch].
```

Here `oldrl` and `oldrh` are respectively the lower and upper row indices of the original matrix that are to be represented by the new matrix, `oldcl` and `oldch` are the corresponding column indices, and `newrl` and `newcl` are the lower row and column indices for the new matrix. (We don't need upper row and column indices, since they are implied by the quantities already given.)

Two sample uses might be, first, to select as a 2×2 submatrix `b[1..2][1..2]` some interior range of an existing matrix, say `a[4..5][2..3]`,

```
float **a,**b;
a=matrix(1,13,1,9);
...
b=submatrix(a,4,5,2,3,1,1);
```

and second, to map an existing matrix `a[1..13][1..9]` into a new matrix `b[0..12][0..8]`,

```
float **a,**b;
a=matrix(1,13,1,9);
...
b=submatrix(a,1,13,1,9,0,0);
```

Incidentally, you can use `submatrix()` for matrices of any type whose `sizeof()` is the same as `sizeof(float)` (often true for `int`, e.g.); just cast the first argument to type `float **` and cast the result to the desired type, e.g., `int **`.

The function

```
void free_submatrix(float **b, long nrl, long nrh, long ncl, long nch)
```

frees the array of row-pointers allocated by `submatrix()`. Note that it does *not* free the memory allocated to the *data* in the submatrix, since that space still lies within the memory allocation of some original matrix.

Finally, if you have a standard C matrix declared as `a[nrow][ncol]`, and you want to convert it into a matrix declared in our pointer-to-row-of-pointers manner, the following function does the trick:

```
float **convert_matrix(float *a, long nrl, long nrh, long ncl, long nch)
```
Allocate a `float` matrix `m[nrl..nrh][ncl..nch]` that points to the matrix declared in the standard C manner as `a[nrow][ncol]`, where `nrow=nrh-nrl+1` and `ncol=nch-ncl+1`. The routine should be called with the address `&a[0][0]` as the first argument.

(You can use this function when you want to make use of C's initializer syntax to set values for a matrix, but then be able to pass the matrix to programs in this book.) The function

```
void free_convert_matrix(float **b, long nrl, long nrh, long ncl, long nch)
```
Free a matrix allocated by `convert_matrix()`.

frees the allocation, without affecting the original matrix a.

The only examples of allocating a *three*-dimensional array as a pointer-to-pointer-to-pointer structure in this book are found in the routines `rlft3` in §12.5 and `sfroid` in §17.4. The necessary allocation and deallocation functions are

```
float ***f3tensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
```
Allocate a `float` 3-dimensional array with subscript range `[nrl..nrh][ncl..nch][ndl..ndh]`.

```
void free_f3tensor(float ***t, long nrl, long nrh, long ncl, long nch,
    long ndl, long ndh)
```
Free a `float` 3-dimensional array allocated by `f3tensor()`.

Complex Arithmetic

C does not have complex data types, or predefined arithmetic operations on complex numbers. That omission is easily remedied with the set of functions in the file `complex.c` which is printed in full in Appendix C at the back of the book. A synopsis is as follows:

```
typedef struct FCOMPLEX {float r,i;} fcomplex;

fcomplex Cadd(fcomplex a, fcomplex b)
Returns the complex sum of two complex numbers.

fcomplex Csub(fcomplex a, fcomplex b)
Returns the complex difference of two complex numbers.

fcomplex Cmul(fcomplex a, fcomplex b)
Returns the complex product of two complex numbers.

fcomplex Cdiv(fcomplex a, fcomplex b)
Returns the complex quotient of two complex numbers.

fcomplex Csqrt(fcomplex z)
Returns the complex square root of a complex number.

fcomplex Conjg(fcomplex z)
Returns the complex conjugate of a complex number.

float Cabs(fcomplex z)
Returns the absolute value (modulus) of a complex number.

fcomplex Complex(float re, float im)
Returns a complex number with specified real and imaginary parts.

fcomplex RCmul(float x, fcomplex a)
Returns the complex product of a real number and a complex number.
```

The implementation of several of these complex operations in floating-point arithmetic is not entirely trivial; see §5.4.

Only about half a dozen routines in this book make explicit use of these complex arithmetic functions. The resulting code is not as readable as one would like, because the familiar operations `+-*/` are replaced by function calls. The `C++` extension to the `C` language allows operators to be redefined. That would allow more readable code. However, in this book we are committed to standard `C`.

We should mention that the above functions assume the ability to pass, return, and assign structures like `FCOMPLEX` (or types such as `fcomplex` that are defined to be structures) *by value*. All recent `C` compilers have this ability, but it is not in the original K&R `C` definition. If you are missing it, you will have to rewrite the functions in `complex.c`, making them pass and return *pointers* to variables of type `fcomplex` instead of the variables themselves. Likewise, you will need to modify the recipes that use the functions.

Several other routines (e.g., the Fourier transforms `four1` and `fourn`) do complex arithmetic "by hand," that is, they carry around real and imaginary parts as `float` variables. This results in more efficient code than would be obtained by using the functions in `complex.c`. But the code is *even less* readable. There is simply no ideal solution to the complex arithmetic problem in `C`.

Implicit Conversion of Float to Double

In traditional (K&R) `C`, `float` variables are automatically converted to `double` before *any* operation is attempted, including both arithmetic operations and passing as arguments to functions. All arithmetic is then done in double precision. If a `float` variable receives the result of such an arithmetic operation, the high precision is immediately thrown away. A corollary of these rules is that all the real-number standard `C` library functions are of type `double` and compute to double precision.

The justification for these conversion rules is, "well, there's nothing wrong with a little extra precision," and "this way the libraries need only one version of each function." One does not need much experience in scientific computing to recognize that the implicit conversion rules are, in fact, sheer madness! In effect, they make it impossible to write efficient numerical programs. One of the cultural barriers that separates computer scientists from "regular" scientists and engineers is a differing point of view on whether a 30% or 50% loss of speed is worth worrying about. In many real-time or state-of-the-art scientific applications, such a loss is catastrophic. The practical scientist is trying to solve tomorrow's problem with yesterday's computer; the computer scientist, we think, often has it the other way around.

The ANSI C standard happily does *not* allow implicit conversion for arithmetic operations, but it *does* require it for function arguments, *unless* the function is fully prototyped by an ANSI declaration as described earlier in this section. That is another reason for our being rigorous about using the ANSI prototype mechanism, and a good reason for you to use an ANSI-compatible compiler.

Some older `C` compilers do provide an optional compilation mode in which the implicit conversion of `float` to `double` is suppressed. Use this if you can. In this book, when we write `float`, we mean `float`; when we write `double`, we mean `double`, i.e., there is a good algorithmic reason for having higher precision. Our routines all can tolerate the traditional implicit conversion rules, but they are more efficient without them. Of course, if your application actually requires double precision, you can change our declarations from `float` to `double` without difficulty. (The brute force approach is to add a preprocessor statement `#define float double` !)

A Few Wrinkles

We like to keep code compact, avoiding unnecessary spaces unless they add immediate clarity. We usually don't put space around the assignment operator "`=`". Through a quirk of history, however, some `C` compilers recognize the (nonexistent) operator "`=-`" as being equivalent to the subtractive assignment operator "`-=`", and "`=*`" as being the same as the multiplicative assignment operator "`*=`". That is why you will see us write `y= -10.0;` or `y=(-10.0);`, and `y= *a;` or `y=(*a);`.

We have the same viewpoint regarding unnecessary parentheses. You can't write (or read) `C` effectively unless you memorize its operator precedence and associativity rules. Please study the accompanying table while you brush your teeth every night.

We never use the `register` storage class specifier. Good optimizing compilers are quite sophisticated in making their own decisions about what to keep in registers, and the best choices are sometimes rather counter-intuitive.

Different compilers use different methods of distinguishing between *defining* and *referencing* declarations of the same external name in several files. We follow the most common scheme, which is also the ANSI standard. The storage class `extern` is explicitly included on all referencing top-level declarations. The storage class is omitted from the single defining declaration for each external variable. We have commented these declarations, so that if your compiler uses a different scheme you can change the code. The various schemes are discussed in §4.8 of [1].

Operator Precedence and Associativity Rules in C		
()	function call	left-to-right
[]	array element	
.	structure or union member	
->	pointer reference to structure	
!	logical not	**right-to-left**
~	bitwise complement	
-	unary minus	
++	increment	
--	decrement	
&	address of	
*	contents of	
(type)	cast to type	
sizeof	size in bytes	
*	multiply	left-to-right
/	divide	
%	remainder	
+	add	left-to-right
-	subtract	
<<	bitwise left shift	left-to-right
>>	bitwise right shift	
<	arithmetic less than	left-to-right
>	arithmetic greater than	
<=	arithmetic less than or equal to	
>=	arithmetic greater than or equal to	
==	arithmetic equal	left-to-right
!=	arithmetic not equal	
&	bitwise and	left-to-right
^	bitwise exclusive or	left-to-right
|	bitwise or	left-to-right
&&	logical and	left-to-right
||	logical or	left-to-right
? :	conditional expression	**right-to-left**
=	assignment operator	**right-to-left**
also += -= *= /= %=		
<<= >>= &= ^= |=		
,	sequential expression	left-to-right

We have already alluded to the problem of computing small integer powers of numbers, most notably the square and cube. The omission of this operation from C is perhaps the language's most galling insult to the scientific programmer. All good FORTRAN compilers recognize expressions like (A+B)**4 and produce in-line code, in this case with only *one* add and *two* multiplies. It is typical for constant integer powers up to 12 to be thus recognized.

In C, the mere problem of squaring is hard enough! Some people "macro-ize" the operation as

```
#define SQR(a) ((a)*(a))
```

However, this is likely to produce code where SQR(sin(x)) results in *two* calls to the sine routine! You might be tempted to avoid this by storing the argument of the squaring function in a temporary variable:

```
static float sqrarg;
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)
```

The global variable sqrarg now has (and needs to keep) scope over the whole module, which is a little dangerous. Also, one needs a completely different macro to square expressions of type int. More seriously, this macro can fail if there are *two* SQR operations in a single expression. Since in C the order of evaluation of pieces of the expression is at the compiler's discretion, the value of sqrarg in one evaluation of SQR can be that from the other evaluation in the same expression, producing nonsensical results. When we need a guaranteed-correct SQR macro, we use the following, which exploits the guaranteed complete evaluation of subexpressions in a conditional expression:

```
static float sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)
```

A collection of macros for other simple operations is included in the file nrutil.h (see Appendix B) and used by many of our programs. Here are the synopses:

SQR(a)	Square a float value.
DSQR(a)	Square a double value.
FMAX(a,b)	Maximum of two float values.
FMIN(a,b)	Minimum of two float values.
DMAX(a,b)	Maximum of two double values.
DMIN(a,b)	Minimum of two double values.
IMAX(a,b)	Maximum of two int values.
IMIN(a,b)	Minimum of two int values.
LMAX(a,b)	Maximum of two long values.
LMIN(a,b)	Minimum of two long values.
SIGN(a,b)	Magnitude of a times sign of b.

Scientific programming in C may someday become a bed of roses; for now, watch out for the thorns!

CITED REFERENCES AND FURTHER READING:

Harbison, S.P., and Steele, G.L., Jr. 1991, *C: A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall). [1]

AT&T Bell Laboratories 1985, *The C Programmer's Handbook* (Englewood Cliffs, NJ: Prentice-Hall).

Kernighan, B., and Ritchie, D. 1978, *The C Programming Language* (Englewood Cliffs, NJ: Prentice-Hall). [Reference for K&R "traditional" C. Later editions of this book conform to the ANSI C standard.]

Hogan, T. 1984, *The C Programmer's Handbook* (Bowie, MD: Brady Communications).

1.3 Error, Accuracy, and Stability

Although we assume no prior training of the reader in formal numerical analysis, we will need to presume a common understanding of a few key concepts. We will define these briefly in this section.

Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of *bits* (binary digits) or *bytes* (groups of 8 bits). Almost all computers allow the programmer a choice among several different such *representations* or *data types*. Data types can differ in the number of bits utilized (the *wordlength*), but also in the more fundamental respect of whether the stored number is represented in *fixed-point* (`int` or `long`) or *floating-point* (`float` or `double`) format.

A number in integer representation is exact. Arithmetic between numbers in integer representation is also exact, with the provisos that (i) the answer is not outside the range of (usually, signed) integers that can be represented, and (ii) that division is interpreted as producing an integer result, throwing away any integer remainder.

In floating-point representation, a number is represented internally by a sign bit s (interpreted as plus or minus), an exact integer exponent e, and an exact positive integer mantissa M. Taken together these represent the number

$$s \times M \times B^{e-E} \tag{1.3.1}$$

where B is the base of the representation (usually $B = 2$, but sometimes $B = 16$), and E is the *bias* of the exponent, a fixed integer constant for any given machine and representation. An example is shown in Figure 1.3.1.

Several floating-point bit patterns can represent the same number. If $B = 2$, for example, a mantissa with leading (high-order) zero bits can be left-shifted, i.e., multiplied by a power of 2, if the exponent is decreased by a compensating amount. Bit patterns that are "as left-shifted as they can be" are termed *normalized*. Most computers always produce normalized results, since these don't waste any bits of the mantissa and thus allow a greater accuracy of the representation. Since the high-order bit of a properly normalized mantissa (when $B = 2$) is *always* one, some computers don't store this bit at all, giving one extra bit of significance.

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented (i.e., have exact values in the form of equation 1.3.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one, simultaneously increasing its exponent, until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too greatly in magnitude, then the smaller operand is effectively replaced by zero, since it is right-shifted to oblivion.

The smallest (in magnitude) floating-point number which, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the *machine accuracy* ϵ_m. A typical computer with $B = 2$ and a 32-bit wordlength has ϵ_m around 3×10^{-8}. (A more detailed discussion of machine characteristics, and a program to determine them, is given in §20.1.) Roughly