Preliminaries

CHAPTER 1

1.0 Introduction

This book is supposed to teach you methods of numerical computing that are practical, efficient, and (insofar as possible) elegant. We presume throughout this book that you, the reader, have particular tasks that you want to get done. We view our job as educating you on how to proceed. Occasionally we may try to reroute you briefly onto a particularly beautiful side road; but by and large, we will guide you along main highways that lead to practical destinations.

Throughout this book, you will find us fearlessly editorializing, telling you what you should and shouldn't do. This prescriptive tone results from a conscious decision on our part, and we hope that you will not find it irritating. We do not claim that our advice is infallible! Rather, we are reacting against a tendency, in the textbook literature of computation, to discuss every possible method that has ever been invented, without ever offering a practical judgment on relative merit. We do, therefore, offer you our practical judgments whenever we can. As you gain experience, you will form your own opinion of how reliable our advice is. Be assured that it is not perfect!

We presume that you are able to read computer programs in C++. The question, "Why C++?", is a complicated one. For now, suffice it to say that we wanted a language with a C-like syntax in the small (because that is most universally readable by our audience), which had a rich set of facilities for object-oriented programming (because that is an emphasis of this third edition), and which was highly backward-compatible with some old, but established and well-tested, tricks in numerical programming. That pretty much led us to C++, although Java (and the closely related C#) were close contenders.

Honesty compels us to point out that in the 20-year history of *Numerical Recipes*, we have never been correct in our predictions about the future of programming languages for scientific programming, *not once*! At various times we convinced ourselves that the wave of the scientific future would be ... Fortran ... Pascal ... C ... Fortran 90 (or 95 or 2000) ... Mathematica ... Matlab ... C++ or Java .... Indeed, several of these enjoy continuing success and have significant followings (not including Pascal!). None, however, currently command a majority, or even a large plurality, of scientific users.

With this edition, we are no longer trying to predict the future of programming languages. Rather, we want a serviceable way of communicating ideas about scientific programming. We hope that these ideas transcend the language, C++, in which we are expressing them.

When we include programs in the text, they look like this:

calendar.h

```
void flmoon(const Int n, const Int nph, Int &jd, Doub &frac) {
```

Our routines begin with an introductory comment summarizing their purpose and explaining their calling sequence. This routine calculates the phases of the moon. Given an integer `n` and a code `nph` for the phase desired (`nph` = 0 for new moon, 1 for first quarter, 2 for full, 3 for last quarter), the routine returns the Julian Day Number `jd`, and the fractional part of a day `frac` to be added to it, of the n^{th} such phase since January, 1900. Greenwich Mean Time is assumed.

```
    const Doub RAD=3.14159265358979323 8/180.0;
    Int i;
    Doub am,as,c,t,t2,xtra;
    c=n+nph/4.0;                          This is how we comment an individual line.
    t=c/1236.85;
    t2=t*t;
    as=359.2242+29.105356*c;                    You aren't really intended to understand
    am=306.0253+385.816918*c+0.010730*t2;          this algorithm, but it does work!
    jd=2415020+28*n+7*nph;
    xtra=0.75933+1.53058868*c+((1.178e-4)-(1.55e-7)*t)*t2;
    if (nph == 0 || nph == 2)
        xtra += (0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am);
    else if (nph == 1 || nph == 3)
        xtra += (0.1721-4.0e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am);
    else throw("nph is unknown in flmoon");       This indicates an error condition.
    i=Int(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
    jd += i;
    frac=xtra-i;
}
```

Note our convention of handling all errors and exceptional cases with a statement like `throw("some error message");`. Since C++ has no built-in exception class for type `char*`, executing this statement results in a fairly rude program abort. However we will explain in §1.5.1 how to get a more elegant result without having to modify the source code.

1.0.1 *What* Numerical Recipes *Is Not*

We want to use the platform of this introductory section to emphasize what *Numerical Recipes* is *not*:

1. *Numerical Recipes* is not a textbook on programming, or on best programming practices, or on C++, or on software engineering. We are not opposed to good programming. We try to communicate good programming practices whenever we can — but only incidentally to our main purpose, which is to teach how practical numerical methods actually work. The unity of style and subordination of function to standardization that is necessary in a good programming (or software engineering) textbook is just not what we have in mind for this book. Each section in this book has as its focus a particular computational method. Our goal is to explain and illustrate *that* method as clearly as possible. No single programming style is best for all such methods, and, accordingly, our style varies from section to section.

2. *Numerical Recipes* is not a program library. That may surprise you if you are one of the many scientists and engineers who use our source code regularly. What

makes our code *not* a program library is that it demands a greater intellectual commitment from the user than a program library ought to do. If you haven't read a routine's accompanying section and gone through the routine line by line to understand how it works, then you use it at great peril! We consider this a feature, not a bug, because our primary purpose is to teach methods, not provide packaged solutions. This book does not include formal exercises, in part because we consider each section's code to be the exercise: If you can understand each line of the code, then you have probably mastered the section.

There are some fine commercial program libraries [1,2] and integrated numerical environments [3-5] available. Comparable free resources are available, both program libraries [6,7] and integrated environments [8-10]. When you want a packaged solution, we recommend that you use one of these. *Numerical Recipes* is intended as a cookbook for cooks, not a restaurant menu for diners.

1.0.2 Frequently Asked Questions

This section is for people who want to jump right in.

1. *How do I use NR routines with my own program?*

The easiest way is to put a bunch of `#include`'s at the top of your program. Always start with `nr3.h`, since that defines some necessary utility classes and functions (see §1.4 for a lot more about this). For example, here's how you compute the mean and variance of the Julian Day numbers of the first 20 full moons after January 1900. (Now *there's* a useful pair of quantities!)

```
#include "nr3.h"
#include "calendar.h"
#include "moment.h"

Int main(void) {
    const Int NTOT=20;
    Int i,jd,nph=2;
    Doub frac,ave,vrnce;
    VecDoub data(NTOT);
    for (i=0;i<NTOT;i++) {
        flmoon(i,nph,jd,frac);
        data[i]=jd;
    }
    avevar(data,ave,vrnce);
    cout << "Average = " << setw(12) << ave;
    cout << " Variance = " << setw(13) << vrnce << endl;
    return 0;
}
```

Be sure that the NR source code files are in a place that your compiler can find them to `#include`. Compile and run the above file. (We can't tell you how to do this part.) Output should be something like this:

```
Average =  2.41532e+06 Variance =       30480.7
```

2. *Yes, but where do I actually get the NR source code as computer files?*

You can buy a code subscription, or a one-time code download, at the Web site http://www.nr.com, or you can get the code on media published by Cambridge

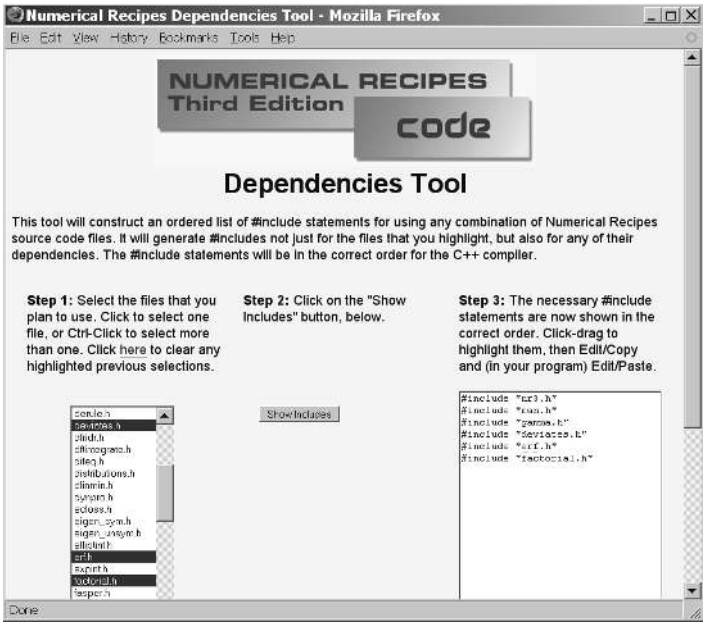

Figure 1.0.1. The interactive page located at http://www.nr.com/dependencies sorts out the dependencies for any combination of *Numerical Recipes* routines, giving an ordered list of the necessary #include files.

University Press (e.g., from Amazon.com or your favorite online or physical bookstore). The code comes with a personal, single-user license (see License and Legal Information on p. xix). The reason that the book (or its electronic version) and the code license are sold separately is to help keep down the price of each. Also, making these products separate meets the needs of more users: Your company or educational institution may have a site license — ask them.

3. *How do I know which files to* `#include`*? It's hard to sort out the dependencies among all the routines.*

In the margin next to each code listing is the name of the source code file that it is in. Make a list of the source code files that you are using. Then go to http://www.nr.com/dependencies and click on the name of each source code file. The interactive Web page will return a list of the necessary `#includes`, in the correct order, to satisfy all dependencies. Figure 1.0.1 will give you an idea of how this works.

4. *What is all this* `Doub`*,* `Int`*,* `VecDoub`*, etc., stuff?*

We always use defined types, not built-in types, so that they can be redefined if necessary. The definitions are in `nr3.h`. Generally, as you can guess, `Doub` means `double`, `Int` means `int`, and so forth. Our convention is to begin all defined types with an uppercase letter. `VecDoub` is a vector class type. Details on our types are in §1.4.

5. *What are* Numerical Recipes *Webnotes?*

Numerical Recipes Webnotes are documents, accessible on the Web, that include some code implementation listings, or other highly specialized topics, that are not included in the paper version of the book. A list of all Webnotes is at

Tested Operating Systems and Compilers	
O/S	Compiler
Microsoft Windows XP SP2	Visual C++ ver. 14.00 (Visual Studio 2005)
Microsoft Windows XP SP2	Visual C++ ver. 13.10 (Visual Studio 2003)
Microsoft Windows XP SP2	Intel C++ Compiler ver. 9.1
Novell SUSE Linux 10.1	GNU GCC (g++) ver. 4.1.0
Red Hat Enterprise Linux 4 (64-bit)	GNU GCC (g++) ver. 3.4.6 and ver. 4.1.0
Red Hat Linux 7.3	Intel C++ Compiler ver. 9.1
Apple Mac OS X 10.4 (Tiger) Intel Core	GNU GCC (g++) ver. 4.0.1

http://www.nr.com/webnotes. By moving some specialized material into Webnotes, we are able to keep down the size and price of the paper book. Webnotes are automatically included in the electronic version of the book; see next question.

6. *I am a post-paper person. I want* Numerical Recipes *on my laptop. Where do I get the complete, fully electronic version?*

A fully electronic version of *Numerical Recipes* is available by annual subscription. You can subscribe instead of, or in addition to, owning a paper copy of the book. A subscription is accessible via the Web, downloadable, printable, and, unlike any paper version, always up to date with the latest corrections. Since the electronic version does not share the page limits of the printed version, it will grow over time by the addition of completely new sections, available only electronically. This, we think, is the future of *Numerical Recipes* and perhaps of technical reference books generally. We anticipate various electronic formats, changing with time as technologies for display and rights management continuously improve: We place a big emphasis on user convenience and usability. See http://www.nr.com/electronic for further information.

7. *Are there bugs in NR?*

Of course! By now, most NR code has the benefit of long-time use by a large user community, but new bugs are sure to creep in. Look at http://www.nr.com for information about known bugs, or to report apparent new ones.

1.0.3 Computational Environment and Program Validation

The code in this book should run without modification on any compiler that implements the ANSI/ISO C++ standard, as described, for example, in Stroustrup's book [11].

As surrogates for the large number of hardware and software configurations, we have tested all the code in this book on the combinations of operating systems and compilers shown in the table above.

In validating the code, we have taken it directly from the machine-readable form of the book's manuscript, so that we have tested exactly what is printed. (This does not, of course, mean that the code is bug-free!)

1.0.4 About References

You will find references, and suggestions for further reading, listed at the end of most sections of this book. References are cited in the text by bracketed numbers like this [12].

We do not pretend to any degree of bibliographical completeness in this book. For topics where a substantial secondary literature exists (discussion in textbooks, reviews, etc.) we often limit our references to a few of the more useful secondary sources, especially those with good references to the primary literature. Where the existing secondary literature is insufficient, we give references to a few primary sources that are intended to serve as starting points for further reading, not as complete bibliographies for the field.

Since progress is ongoing, it is inevitable that our references for many topics are already, or will soon become, out of date. We have tried to include older references that are good for "forward" Web searching: A search for more recent papers that cite the references given should lead you to the most current work.

Web references and URLs present a problem, because there is no way for us to guarantee that they will still be there when you look for them. A date like 2007+ means "it was there in 2007." We try to give citations that are complete enough for you to find the document by Web search, even if it has moved from the location listed.

The order in which references are listed is not necessarily significant. It reflects a compromise between listing cited references in the order cited, and listing suggestions for further reading in a roughly prioritized order, with the most useful ones first.

1.0.5 About "Advanced Topics"

Material set in smaller type, like this, signals an "advanced topic," either one outside of the main argument of the chapter, or else one requiring of you more than the usual assumed mathematical background, or else (in a few cases) a discussion that is more speculative or an algorithm that is less well tested. Nothing important will be lost if you skip the advanced topics on a first reading of the book.

Here is a function for getting the Julian Day Number from a calendar date.

calendar.h

```
Int julday(const Int mm, const Int id, const Int iyyy) {
```

In this routine julday returns the Julian Day Number that begins at noon of the calendar date specified by month mm, day id, and year iyyy, all integer variables. Positive year signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D.

```
    const Int IGREG=15+31*(10+12*1582);          Gregorian Calendar adopted Oct. 15, 1582.
    Int ja,jul,jy=iyyy,jm;
    if (jy == 0) throw("julday: there is no year zero.");
    if (jy < 0) ++jy;
    if (mm > 2) {
        jm=mm+1;
    } else {
        --jy;
        jm=mm+13;
    }
    jul = Int(floor(365.25*jy)+floor(30.6001*jm)+id+1720995);
    if (id+31*(mm+12*iyyy) >= IGREG) {          Test whether to change to Gregorian Cal-
        ja=Int(0.01*jy);                            endar.
        jul += 2-ja+Int(0.25*ja);
    }
    return jul;
}
```

And here is its inverse.

calendar.h

```
void caldat(const Int julian, Int &mm, Int &id, Int &iyyy) {
```

Inverse of the function julday given above. Here julian is input as a Julian Day Number, and the routine outputs mm,id, and iyyy as the month, day, and year on which the specified Julian Day started at noon.

```
    const Int IGREG=2299161;
    Int ja,jalpha,jb,jc,jd,je;

    if (julian >= IGREG) {       Cross-over to Gregorian Calendar produces this correc-
        jalpha=Int((Doub(julian-1867216)-0.25)/36524.25);          tion.
        ja=julian+1+jalpha-Int(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=Int(6680.0+(Doub(jb-2439870)-122.1)/365.25);
    jd=Int(365*jc+(0.25*jc));
    je=Int((jb-jd)/30.6001);
    id=jb-jd-Int(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);
}
```

As an exercise, you might try using these functions, along with `flmoon` in §1.0, to search for future occurrences of a full moon on Friday the 13th. (Answers, in time zone GMT minus 5: 9/13/2019 and 8/13/2049.) For additional calendrical algorithms, applicable to various historical calendars, see [13].

CITED REFERENCES AND FURTHER READING:

Visual Numerics, 2007+, *IMSL Numerical Libraries*, at `http://www.vni.com`.[1]

Numerical Algorithms Group, 2007+, *NAG Numerical Library*, at `http://www.nag.co.uk`.[2]

Wolfram Research, Inc., 2007+, *Mathematica*, at `http://www.wolfram.com`.[3]

The MathWorks, Inc., 2007+, *MATLAB*, at `http://www.mathworks.com`.[4]

Maplesoft, Inc., 2007+, *Maple*, at `http://www.maplesoft.com`.[5]

GNU Scientific Library, 2007+, at `http://www.gnu.org/software/gsl`.[6]

Netlib Repository, 2007+, at `http://www.netlib.org`.[7]

Scilab Scientific Software Package, 2007+, at `http://www.scilab.org`.[8]

GNU Octave, 2007+, at `http://www.gnu.org/software/octave`.[9]

R Software Environment for Statistical Computing and Graphics, 2007+, at `http://www.r-project.org`.[10]

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).[11]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell).[12]

Hatcher, D.A. 1984, "Simple Formulae for Julian Day Numbers and Calendar Dates," *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.[13]

1.1 Error, Accuracy, and Stability

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* (like `int`) or *floating-point* (like `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.

1.1.1 Floating-Point Representation

In a 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 exactly represented binary mantissa M. Taken together these represent the number

$$S \times M \times b^{E-e} \tag{1.1.1}$$

where b is the base of the representation ($b = 2$ almost always), and e is the *bias* of the exponent, a fixed integer constant for any given machine and representation.

	S	E	F	Value
`float`	any	1–254	any	$(-1)^S \times 2^{E-127} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-126} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	255	0	$+\infty$
	1	255	0	$-\infty$
	any	255	nonzero	`NaN`
`double`	any	1–2046	any	$(-1)^S \times 2^{E-1023} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-1022} \times 0.F*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	2047	0	$+\infty$
	1	2047	0	$-\infty$
	any	2047	nonzero	`NaN`
*unnormalized values				

Several floating-point bit patterns can in principle 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*.

Virtually all modern processors share the same floating-point data representations, namely those specified in IEEE Standard 754-1985 [1]. (For some discussion of nonstandard processors, see §22.2.) For 32-bit `float` values, the exponent is represented in 8 bits (with $e = 127$), the mantissa in 23; for 64-bit `double` values, the exponent is 11 bits (with $e = 1023$), the mantissa, 52. An additional trick is used for the mantissa for most nonzero floating values: Since the high-order bit of a properly normalized mantissa is *always* one, the stored mantissa bits are viewed as being preceded by a "phantom" bit with the value 1. In other words, the mantissa M has the numerical value $1.F$, where F (called the *fraction*) consists of the bits (23 or 52 in number) that are actually stored. This trick gains a little "bit" of precision.

Here are some examples of IEEE 754 representations of `double` values:

$$\begin{aligned}
&\texttt{0 01111111111 0000} \ (+ \text{ 48 more zeros}) = +1 \times 2^{1023-1023} \times 1.0_2 = 1. \\
&\texttt{1 01111111111 0000} \ (+ \text{ 48 more zeros}) = -1 \times 2^{1023-1023} \times 1.0_2 = -1. \\
&\texttt{0 01111111111 1000} \ (+ \text{ 48 more zeros}) = +1 \times 2^{1023-1023} \times 1.1_2 = 1.5 \\
&\texttt{0 10000000000 0000} \ (+ \text{ 48 more zeros}) = +1 \times 2^{1024-1023} \times 1.0_2 = 2. \\
&\texttt{0 10000000001 1010} \ (+ \text{ 48 more zeros}) = +1 \times 2^{1025-1023} \times 1.1010_2 = 6.5
\end{aligned} \tag{1.1.2}$$

You can examine the representation of any value by code like this:

```
union Udoub {
    double d;
    unsigned char c[8];
};

void main() {
    Udoub u;
    u.d = 6.5;
    for (int i=7;i>=0;i--) printf("%02x",u.c[i]);
    printf("\n");
}
```

This is C, and deprecated style, but it will work. On most processors, including Intel Pentium and successors, you'll get the printed result `401a000000000000`, which (writing out each hex digit as four binary digits) is the last line in equation (1.1.2). If you get the bytes (groups of two hex digits) in reverse order, then your processor is *big-endian* instead of *little-endian*: The IEEE 754 standard does not specify (or care) in which order the bytes in a floating-point value are stored.

The IEEE 754 standard includes representations of positive and negative infinity, positive and negative zero (treated as computationally equivalent, of course), and also `NaN` ("not a number"). The table on the previous page gives details of how these are represented.

The reason for representing some *unnormalized* values, as shown in the table, is to make "underflow to zero" more graceful. For a sequence of smaller and smaller values, after you pass the smallest normalizable value (with magnitude 2^{-127} or 2^{-1023}; see table), you start right-shifting the leading bit of the mantissa. Although

you gradually lose precision, you don't actually underflow to zero until 23 or 52 bits later.

When a routine needs to know properties of the floating-point representation, it can reference the `numeric_limits` class, which is part of the C++ Standard Library. For example, `numeric_limits<double>::min()` returns the smallest normalized `double` value, usually $2^{-1022} \approx 2.23 \times 10^{-308}$. For more on this, see §22.2.

1.1.2 Roundoff Error

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 1.1.1). For example, two floating numbers are added by first right-shifting (dividing by two) the mantissa of the smaller (in magnitude) one and 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 that, 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. IEEE 754 standard `float` has ϵ_m about 1.19×10^{-7}, while `double` has about 2.22×10^{-16}. Values like this are accessible as, e.g., `numeric _limits <double>::epsilon()`. (A more detailed discussion of machine characteristics is in §22.2.) Roughly speaking, the machine accuracy ϵ_m is the fractional accuracy to which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Pretty much any arithmetic operation among floating numbers should be thought of as introducing an additional fractional error of at least ϵ_m. This type of error is called *roundoff error*.

It is important to understand that ϵ_m is not the smallest floating-point number that can be represented on a machine. *That* number depends on how many bits there are in the exponent, while ϵ_m depends on how many bits there are in the mantissa.

Roundoff errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, you perform N such arithmetic operations, you *might* be so lucky as to have a total roundoff error on the order of $\sqrt{N}\epsilon_m$, if the roundoff errors come in randomly up or down. (The square root comes from a random-walk.) However, this estimate can be very badly off the mark for two reasons:

(1) It very frequently happens that the regularities of your calculation, or the peculiarities of your computer, cause the roundoff errors to accumulate preferentially in one direction. In this case the total will be of order $N\epsilon_m$.

(2) Some especially unfavorable occurrences can vastly increase the roundoff error of single operations. Generally these can be traced to the subtraction of two very nearly equal numbers, giving a result whose only significant bits are those (few) low-order ones in which the operands differed. You might think that such a "coincidental" subtraction is unlikely to occur. Not always so. Some mathematical expressions magnify its probability of occurrence tremendously. For example, in the familiar formula for the solution of a quadratic equation,

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{1.1.3}$$