# Chapter 1. Preliminaries

## 1.0 Introduction

This book, like its predecessor edition, 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.

We presume that you are able to read computer programs in `C`, that being the language of this version of *Numerical Recipes* (Second Edition). The book *Numerical Recipes in FORTRAN* (Second Edition) is separately available, if you prefer to program in that language. Earlier editions of *Numerical Recipes in Pascal* and *Numerical Recipes Routines and Examples in BASIC* are also available; while not containing the additional material of the Second Edition versions in `C` and `FORTRAN`, these versions are perfectly serviceable if `Pascal` or `BASIC` is your language of choice.

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

```
#include <math.h>
#define RAD (3.14159265/180.0)

void flmoon(int n, int nph, long *jd, float *frac)
Our programs 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 nth such phase since January, 1900. Greenwich Mean Time is assumed.
{
    void nrerror(char error_text[]);
    int i;
    float 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+28L*n+7L*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 nrerror("nph is unknown in flmoon");        This is how we will indicate error
    i=(int)(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));    conditions.
    *jd += i;
    *frac=xtra-i;
}
```

If the syntax of the function definition above looks strange to you, then you are probably used to the older Kernighan and Ritchie ("K&R") syntax, rather than that of the newer ANSI C. In this edition, we adopt ANSI C as our standard. You might want to look ahead to §1.2 where ANSI C function prototypes are discussed in more detail.

Note our convention of handling all errors and exceptional cases with a statement like `nrerror("some error message");`. The function `nrerror()` is part of a small file of utility programs, `nrutil.c`, listed in Appendix B at the back of the book. This Appendix includes a number of other utilities that we will describe later in this chapter. Function `nrerror()` prints the indicated error message to your `stderr` device (usually your terminal screen), and then invokes the function `exit()`, which terminates execution. The function `exit()` is in every C library we know of; but if you find it missing, you can modify `nrerror()` so that it does anything else that will halt execution. For example, you can have it pause for input from the keyboard, and then manually interrupt execution. In some applications, you will want to modify `nrerror()` to do more sophisticated error handling, for example to transfer control somewhere else, with an error flag or error code set.

We will have more to say about the C programming language, its conventions and style, in §1.1 and §1.2.

## Computational Environment and Program Validation

Our goal is that the programs in this book be as portable as possible, across different platforms (models of computer), across different operating systems, and across different C compilers. C was designed with this type of portability in mind. Nevertheless, we have found that there is no substitute for actually checking all programs on a variety of compilers, in the process uncovering differences in library structure or contents, and even occasional differences in allowed syntax. As surrogates for the large number of possible combinations, we have tested all the programs in this book on the combinations of machines, operating systems, and compilers shown on the accompanying table. More generally, the programs should run without modification on any compiler that implements the ANSI C standard, as described for example in Harbison and Steele's excellent book [1]. With small modifications, our programs should run on any compiler that implements the older, *de facto* K&R standard [2]. An example of the kind of trivial incompatibility to watch out for is that ANSI C requires the memory allocation functions `malloc()`

| Tested Machines and Compilers | | |
|---|---|---|
| Hardware | O/S Version | Compiler Version |
| IBM PC compatible 486/33 | MS-DOS 5.0/Windows 3.1 | Microsoft C/C++ 7.0 |
| IBM PC compatible 486/33 | MS-DOS 5.0 | Borland C/C++ 2.0 |
| IBM RS/6000 | AIX 3.2 | IBM xlc 1.02 |
| DECstation 5000/25 | ULTRIX 4.2A | CodeCenter (Saber) C 3.1.1 |
| DECsystem 5400 | ULTRIX 4.1 | GNU C Compiler 2.1 |
| Sun SPARCstation 2 | SunOS 4.1 | GNU C Compiler 1.40 |
| DECstation 5000/200 | ULTRIX 4.2 | DEC RISC C 2.1* |
| Sun SPARCstation 2 | SunOS 4.1 | Sun cc 1.1* |
| *compiler version does not fully implement ANSI C; only K&R validated | | |

and `free()` to be declared via the header `stdlib.h`; some older compilers require them to be declared with the header file `malloc.h`, while others regard them as inherent in the language and require no header file at all.

In validating the programs, we have taken the program source code directly from the machine-readable form of the book's manuscript, to decrease the chance of propagating typographical errors. "Driver" or demonstration programs that we used as part of our validations are available separately as the *Numerical Recipes Example Book (C)*, as well as in machine-readable form. If you plan to use more than a few of the programs in this book, or if you plan to use programs in this book on more than one different computer, then you may find it useful to obtain a copy of these demonstration programs.

Of course we would be foolish to claim that there are no bugs in our programs, and we do not make such a claim. We have been very careful, and have benefitted from the experience of the many readers who have written to us. If you find a new bug, please document it and tell us!

## Compatibility with the First Edition

If you are accustomed to the *Numerical Recipes* routines of the First Edition, rest assured: almost all of them are still here, with the same names and functionalities, often with major improvements in the code itself. In addition, we hope that you will soon become equally familiar with the added capabilities of the more than 100 routines that are new to this edition.

We have retired a small number of First Edition routines, those that we believe to be clearly dominated by better methods implemented in this edition. A table, following, lists the retired routines and suggests replacements.

First Edition users should also be aware that some routines common to both editions have alterations in their calling interfaces, so are not directly "plug compatible." A fairly complete list is: `chsone`, `chstwo`, `covsrt`, `dfpmin`, `laguer`, `lfit`, `memcof`, `mrqcof`, `mrqmin`, `pzextr`, `ran4`, `realft`, `rzextr`, `shoot`, `shootf`. There may be others (depending in part on which printing of the First Edition is taken for the comparison). If you have written software of any appreciable complexity

| Previous Routines Omitted from This Edition | | |
|---|---|---|
| Name(s) | Replacement(s) | Comment |
| `adi` | `mglin` or `mgfas` | better method |
| `cosft` | `cosft1` or `cosft2` | choice of boundary conditions |
| `cel`, `el2` | `rf`, `rd`, `rj`, `rc` | better algorithms |
| `des`, `desks` | `ran4` now uses `psdes` | was too slow |
| `mdian1`, `mdian2` | `select`, `selip` | more general |
| `qcksrt` | `sort` | name change (`sort` is now `hpsort`) |
| `rkqc` | `rkqs` | better method |
| `smooft` | use `convlv` with coefficients from `savgol` | |
| `sparse` | `linbcg` | more general |

that is dependent on First Edition routines, we do *not* recommend blindly replacing them by the corresponding routines in this book. We do recommend that any new programming efforts use the new routines.

## 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 [3].

Because computer algorithms often circulate informally for quite some time before appearing in a published form, the task of uncovering "primary literature" is sometimes quite difficult. We have not attempted this, and 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 have consciously limited 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.

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.

The remaining three sections of this chapter review some basic concepts of programming (control structures, etc.), discuss a set of conventions specific to C that we have adopted in this book, and introduce some fundamental concepts in numerical analysis (roundoff error, etc.). Thereafter, we plunge into the substantive material of the book.

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]

Kernighan, B., and Ritchie, D. 1978, *The C Programming Language* (Englewood Cliffs, NJ: Prentice-Hall). [2] [Reference for K&R "traditional" C. Later editions of this book conform to the ANSI C standard.]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [3]



## 1.1 Program Organization and Control Structures

We sometimes like to point out the close analogies between computer programs, on the one hand, and written poetry or written musical scores, on the other. All three present themselves as visual media, symbols on a two-dimensional page or computer screen. Yet, in all three cases, the visual, two-dimensional, *frozen-in-time* representation communicates (or is supposed to communicate) something rather different, namely a process that *unfolds in time*. A poem is meant to be read; music, played; a program, executed as a sequential series of computer instructions.

In all three cases, the target of the communication, in its visual form, is a human being. The goal is to transfer to him/her, as efficiently as can be accomplished, the greatest degree of understanding, in advance, of how the process *will* unfold in time. In poetry, this human target is the reader. In music, it is the performer. In programming, it is the program user.

Now, you may object that the target of communication of a program is not a human but a computer, that the program user is only an irrelevant intermediary, a lackey who feeds the machine. This is perhaps the case in the situation where the business executive pops a diskette into a desktop computer and feeds that computer a black-box program in binary executable form. The computer, in this case, doesn't much care whether that program was written with "good programming practice" or not.

We envision, however, that you, the readers of this book, are in quite a different situation. You need, or want, to know not just *what* a program does, but also *how* it does it, so that you can tinker with it and modify it to your particular application. You need others to be able to see what you have done, so that they can criticize or admire. In such cases, where the desired goal is *maintainable* or *reusable* code, the targets of a program's communication are surely human, not machine.

One key to achieving good programming practice is to recognize that programming, music, and poetry — all three being symbolic constructs of the human brain — are naturally structured into hierarchies that have many different nested levels. Sounds (phonemes) form small meaningful units (morphemes) which in turn form words; words group into phrases, which group into sentences; sentences make paragraphs, and these are organized into higher levels of meaning. Notes form musical phrases, which form themes, counterpoints, harmonies, etc.; which form movements, which form concertos, symphonies, and so on.

The structure in programs is equally hierarchical. Appropriately, good programming practice brings different techniques to bear on the different levels [1-3]. At a low level is the `ascii` character set. Then, constants, identifiers, operands,