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 `FORTRAN`, that being the language of this version of *Numerical Recipes* (Second Edition). The book *Numerical Recipes in C* (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:

```
SUBROUTINE flmoon(n,nph,jd,frac)
INTEGER jd,n,nph
REAL frac,RAD
PARAMETER (RAD=3.14159265/180.)
    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.
INTEGER i
REAL am,as,c,t,t2,xtra
c=n+nph/4.                          This is how we comment an individual line.
t=c/1236.85
t2=t**2
```

```
as=359.2242+29.105356*c                 You aren't really intended to understand this al-
am=306.0253+385.816918*c+0.010730*t2        gorithm, 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.eq.0.or.nph.eq.2)then
    xtra=xtra+(0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am)
else if(nph.eq.1.or.nph.eq.3)then
    xtra=xtra+(0.1721-4.e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am)
else
    pause 'nph is unknown in flmoon'    This is how we will indicate error conditions.
endif
if(xtra.ge.0.)then
    i=int(xtra)
else
    i=int(xtra-1.)
endif
jd=jd+i
frac=xtra-i
return
END
```

A few remarks about our typographical conventions and programming style are in order at this point:

- It is good programming practice to declare all variables and identifiers in explicit "type" statements (`REAL`, `INTEGER`, etc.), even though the implicit declaration rules of `FORTRAN` do not require this. We will always do so. (As an aside to non-`FORTRAN` programmers, the implicit declaration rules are that variables which begin with the letters `i,j,k,l,m,n` are implicitly declared to be type `INTEGER`, while all other variables are implicitly declared to be type `REAL`. Explicit declarations override these conventions.)
- In sympathy with modular and object-oriented programming practice, we separate, typographically, a routine's "public" or "interface" section from its "private" or "implementation" section. We do this even though `FORTRAN` is by no means a modular or object-oriented language: the separation makes sense simply as good programming style.
- The *public* section contains the calling interface and declarations of its variables. We find it useful to consider `PARAMETER` statements, and their associated declarations, as also being in the public section, since a user may want to modify parameter values to suit a particular purpose. `COMMON` blocks are likewise usually part of the public section, since they involve communication between routines.
- As the last entry in the public section, we will, where applicable, put a standardized comment line with the word `USES` (not a `FORTRAN` keyword), followed by a list of all external subroutines and functions that the routine references, excluding built-in `FORTRAN` functions. (For examples, see the routines in §6.1.)
- An introductory comment, set in type as an indented paragraph, separates the public section from the private or implementation section.
- Within the introductory comments, as well as in the text, we will frequently use the notation `a(1:m)` to mean "the array elements `a(1)`, `a(2)`, ..., `a(m)`." Likewise, notations like `b(2:7)` or `c(1:m,1:n)` are to be

interpreted as ranges of array indices. (This use of colon to denote ranges comes from FORTRAN-77's syntax for array declarators and character substrings.)

- The *implementation section* contains the declarations of variables that are used only internally in the routine, any necessary SAVE statements for static variables (variables that must be preserved between calls to the routine), and of course the routine's actual executable code.
- Case is not significant in FORTRAN, so it can be used to promote readability. Our convention is to use upper case for two different, nonconflicting, purposes. First, nonexecutable compiler keywords are in upper case (e.g., SUBROUTINE, REAL, COMMON); second, parameter identifiers are in upper case. The reason for capitalizing parameters is that, because their values are liable to be modified, the user often needs to scan the implementation section of code to see exactly how the parameters are used.
- For simplicity, we adopt the convention of handling all errors and exceptional cases by the pause statement. In general, we do not intend that you continue program execution after a pause occurs, but FORTRAN allows you to do so — if you want to see what kind of wrong answer or catastrophic error results. In many applications, you will want to modify our programs to do more sophisticated error handling, for example to return with an error flag set, or call an error-handling routine.
- In the printed form of this book, we take some special typographical liberties regarding statement labels, and do ... continue constructions. These are described in §1.1. Note that no such liberties are taken in the machine-readable *Numerical Recipes* diskettes, where all routines are in standard ANSI FORTRAN-77.

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 FORTRAN compilers. 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 FORTRAN-77 standard. At the time of writing, there are not enough installed implementations of the successor FORTRAN-90 standard to justify our using any of its more advanced features. Since FORTRAN-90 is backwards-compatible with FORTRAN-77, there should be no difficulty in using the programs in this book on FORTRAN-90 compilers, as they become available.

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 (FORTRAN)*, 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.

Tested Machines and Compilers		
Hardware	O/S Version	Compiler Version
IBM PC compatible 486/33	MS-DOS 5.0	Microsoft Fortran 5.1
IBM RS6000	AIX 3.0	IBM AIX XL FORTRAN Compiler/6000
IBM PC-RT	BSD UNIX 4.3	"UNIX Fortran 77"
DEC VAX 4000	VMS 5.4	VAX Fortran 5.4
DEC VAXstation 2000	BSD UNIX 4.3	Berkeley f77 2.0 (4.3 bsd, SCCS lev. 6)
DECstation 5000/200	ULTRIX 4.2	DEC Fortran for ULTRIX RISC 3.1
DECsystem 5400	ULTRIX 4.1	MIPS f77 2.10
Sun SPARCstation 2	SunOS 4.1	Sun Fortran 1.4 (SC 1.0)
Apple Macintosh	System 6.0.7 / MPW 3.2	Absoft Fortran 77 Compiler 3.1.2

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 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 [1].

Because computer algorithms often circulate informally for quite some time before appearing in a published form, the task of uncovering "primary literature"

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

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 two sections of this chapter review some basic concepts of programming (control structures, etc.) and of numerical analysis (roundoff error, etc.). Thereafter, we plunge into the substantive material of the book.

CITED REFERENCES AND FURTHER READING:

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [1]

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, operators. Then program statements, like `a(j+1)=b+c/3.0`. Here, the best programming advice is simply *be clear*, or (correspondingly) *don't be too tricky*. You might momentarily be proud of yourself at writing the single line

```
k=(2-j)*(1+3*j)/2
```

if you want to permute cyclically one of the values $\mathtt{j} = (0, 1, 2)$ into respectively $\mathtt{k} = (1, 2, 0)$. You will regret it later, however, when you try to understand that line. Better, and likely also faster, is

```
k=j+1
if (k.eq.3) k=0
```

Many programming stylists would even argue for the ploddingly literal

```
if (j.eq.0) then
    k=1
else if (j.eq.1) then
    k=2
```

```
else if (j.eq.2) then
    k=0
else
    pause 'never get here'
endif
```

on the grounds that it is both clear and additionally safeguarded from wrong assumptions about the possible values of `j`. Our preference among the implementations is for the middle one.

In this simple example, we have in fact traversed several levels of hierarchy: Statements frequently come in "groups" or "blocks" which make sense only taken as a whole. The middle fragment above is one example. Another is

```
swap=a(j)
a(j)=b(j)
b(j)=swap
```

which makes immediate sense to any programmer as the exchange of two variables, while

```
sum=0.0
ans=0.0
n=1
```

is very likely to be an initialization of variables prior to some iterative process. This level of hierarchy in a program is usually evident to the eye. It is good programming practice to put in comments at this level, e.g., "initialize" or "exchange variables."

The next level is that of *control structures*. These are things like the `if...then...else` clauses in the example above, `do` loops, and so on. This level is sufficiently important, and relevant to the hierarchical level of the routines in this book, that we will come back to it just below.

At still higher levels in the hierarchy, we have (in `FORTRAN`) subroutines, functions, and the whole "global" organization of the computational task to be done. In the musical analogy, we are now at the level of movements and complete works. At these levels, *modularization* and *encapsulation* become important programming concepts, the general idea being that program units should interact with one another only through clearly defined and narrowly circumscribed interfaces. Good modularization practice is an essential prerequisite to the success of large, complicated software projects, especially those employing the efforts of more than one programmer. It is also good practice (if not quite as essential) in the less massive programming tasks that an individual scientist, or reader of this book, encounters.

Some computer languages, such as Modula-2 and `C++`, promote good modularization with higher-level language constructs, absent in `FORTRAN`-77. In Modula-2, for example, subroutines, type definitions, and data structures can be encapsulated into "modules" that communicate through declared public interfaces and whose internal workings are hidden from the rest of the program [4]. In the `C++` language, the key concept is "class," a user-definable generalization of data type that provides for data hiding, automatic initialization of data, memory management, dynamic typing, and operator overloading (i.e., the user-definable extension of operators like `+` and `*` so as to be appropriate to operands in any particular class) [5]. Properly used in defining the data structures that are passed between program units, classes

can clarify and circumscribe these units' public interfaces, reducing the chances of programming error and also allowing a considerable degree of compile-time and run-time error checking.

Beyond modularization, though depending on it, lie the concepts of *object-oriented programming*. Here a programming language, such as C++ or Turbo Pascal 5.5 [6], allows a module's public interface to accept redefinitions of types or actions, and these redefinitions become shared all the way down through the module's hierarchy (so-called *polymorphism*). For example, a routine written to invert a matrix of real numbers could — dynamically, at run time — be made able to handle complex numbers by overloading complex data types and corresponding definitions of the arithmetic operations. Additional concepts of *inheritance* (the ability to define a data type that "inherits" all the structure of another type, plus additional structure of its own), and *object extensibility* (the ability to add functionality to a module without access to its source code, e.g., at run time), also come into play.

We have not attempted to modularize, or make objects out of, the routines in this book, for at least two reasons. First, the chosen language, FORTRAN-77, does not really make this possible. Second, we envision that you, the reader, might want to incorporate the algorithms in this book, a few at a time, into modules or objects with a structure of your own choosing. There does not exist, at present, a standard or accepted set of "classes" for scientific object-oriented computing. While we might have tried to invent such a set, doing so would have inevitably tied the algorithmic content of the book (which is its *raison d'être*) to some rather specific, and perhaps haphazard, set of choices regarding class definitions.

On the other hand, we are not unfriendly to the goals of modular and object-oriented programming. Within the limits of FORTRAN, we have therefore tried to structure our programs to be "object friendly," principally via the clear delineation of interface vs. implementation (§1.0) and the explicit declaration of variables. Within our implementation sections, we have paid particular attention to the practices of *structured programming*, as we now discuss.

Control Structures

An executing program unfolds in time, but not strictly in the linear order in which the statements are written. Program statements that affect the order in which statements are executed, or that affect whether statements are executed, are called *control statements*. Control statements never make useful sense by themselves. They make sense only in the context of the groups or blocks of statements that they in turn control. If you think of those blocks as paragraphs containing sentences, then the control statements are perhaps best thought of as the indentation of the paragraph and the punctuation between the sentences, not the words within the sentences.

We can now say what the goal of structured programming is. It is *to make program control manifestly apparent in the visual presentation of the program*. You see that this goal has nothing at all to do with how the computer sees the program. As already remarked, computers don't care whether you use structured programming or not. Human readers, however, *do* care. You yourself will also care, once you discover how much easier it is to perfect and debug a well-structured program than one whose control structure is obscure.

You accomplish the goals of structured programming in two complementary ways. First, you acquaint yourself with the small number of essential control structures that occur over and over again in programming, and that are therefore given convenient representations in most programming languages. You should learn to think about your programming tasks, insofar as possible, exclusively in terms of these standard control structures. In writing programs, you should get into the habit of representing these standard control structures in consistent, conventional ways.

"Doesn't this inhibit *creativity*?" our students sometimes ask. Yes, just as Mozart's creativity was inhibited by the sonata form, or Shakespeare's by the metrical requirements of the sonnet. The point is that creativity, when it is meant to communicate, does *well* under the inhibitions of appropriate restrictions on format.

Second, you *avoid*, insofar as possible, control statements whose controlled blocks or objects are difficult to discern at a glance. This means, in practice, that *you must try to avoid statement labels and* `goto`*'s*. It is not the `goto`'s that are dangerous (although they do interrupt one's reading of a program); the statement labels are the hazard. In fact, whenever you encounter a statement label while reading a program, you will soon become conditioned to get a sinking feeling in the pit of your stomach. Why? Because the following questions will, by habit, immediately spring to mind: Where did control come *from* in a branch to this label? It could be anywhere in the routine! What circumstances resulted in a branch to this label? They could be anything! Certainty becomes uncertainty, understanding dissolves into a morass of possibilities.

Some older languages, notably 1966 `FORTRAN` and to a lesser extent `FORTRAN`-77, *require* statement labels in the construction of certain standard control structures. We will see this in more detail below. This is a demerit for these languages. In such cases, you must use labels as required. But you should never branch to them independently of the standard control structure. If you must branch, let it be to an additional label, one that is not masquerading as part of a standard control structure.

We call labels that are part of a standard construction and never otherwise branched to *tame labels*. They do not interfere with structured programming in any way, except possibly typographically as distractions to the eye.

Some examples are now in order to make these considerations more concrete (see Figure 1.1.1).

Catalog of Standard Structures

Iteration. In `FORTRAN`, simple iteration is performed with a `do` loop, for example

```
      do 10 j=2,1000
          b(j)=a(j-1)
          a(j-1)=j
10    continue
```

Notice how we always indent the block of code that is acted upon by the control structure, leaving the structure itself unindented. The statement label `10` in this example is a tame label. The majority of modern implementations of `FORTRAN`-77 provide a nonstandard language extension that obviates the tame label. Originally

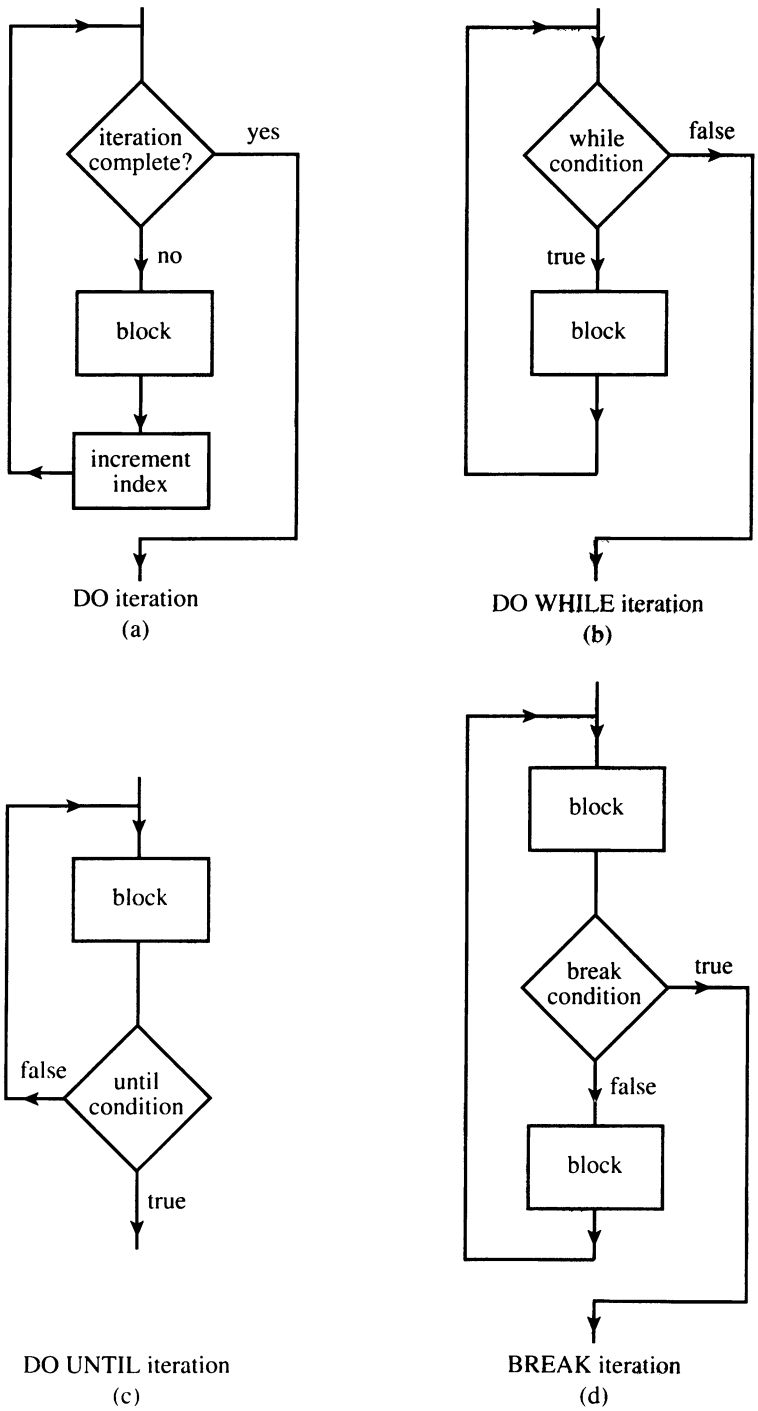

Figure 1.1.1. Standard control structures used in structured programming: (a) DO iteration; (b) DO WHILE iteration; (c) DO UNTIL iteration; (d) BREAK iteration; (e) IF structure; (f) obsolete form of DO iteration found in FORTRAN-66, where the block is executed once even if the iteration condition is initially not satisfied.