Chapter 1. Preliminaries

1.0 Introduction

This book is supposed to teach you methods of numerical computing which 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 travel with 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 `Pascal`. In this edition of *Numerical Recipes* that is the language in which all the "recipes" are implemented. If you are more comfortable with `FORTRAN` or `C`, you will find other editions of the book in those languages. The various editions are quite similar with respect to mathematical and algorithmic content. Moreover, we have chosen our programming conventions to accentuate the similarities between languages. If you are multilingual, any single edition should suffice.

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

```
PROCEDURE flmoon(n,nph: integer;
                 VAR jd: longint;
               VAR frac: real);
```

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. Note that this routine will not run on systems with 2-byte integers.
   The type longint (4-byte integer) is discussed further in §1.2.
VAR
   i: integer;
   nl: longint;
   rad,xtra,t2,t,c,as,am: real;
BEGIN
   rad := 3.14159265/180.0;
   c := n+nph/4.0;                                  This is how we comment an individual
   t := c/1236.85;                                      line.
   t2 := sqr(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!
   nl := n;                                         Cast integer to longint.
   jd := 2415020+28*nl+7*nph;
   xtra := 0.75933+1.53058868*c+(1.178e-4-1.55e-7*t)*t2;
   IF (nph = 0) OR (nph = 2) THEN
      xtra := xtra+(0.1734-3.93e-4*t)*sin(rad*as)-0.4068*sin(rad*am)
   ELSE IF (nph = 1) OR (nph = 3) THEN
      xtra := xtra+(0.1721-4.0e-4*t)*sin(rad*as)-0.6280*sin(rad*am)
   ELSE BEGIN
      writeln('pause in FLMOON - nph is unknown.');     This is how we will indicate error
      readln                                                conditions.
   END;
   IF xtra >= 0.0 THEN
      i := trunc(xtra)
   ELSE
      i := trunc(xtra-1.0);
   jd := jd+i;
   frac := xtra-i
END;
```

Note our convention of handling all errors and exceptional cases with statements like

```
   writeln('pause in RoutineName - some error message');
   readln;
```

The purpose of the `readln` statement is simply to pause your program, and to invite you to contemplate whatever unfortunate situation has occured. In general, we do not intend that you continue program execution after such a pause occurs. Incidentally, we do *not* put forth this pause convention as a good example of software engineering! In serious applications, you will certainly want to modify our programs to do more sophisticated error handling, for example to transfer control somewhere else, with an error flag or error code set.

We have attempted to make the procedures in this book as generally useful as possible, not just in terms of the subjects covered, but also in terms of their degree of portability among very different computers. Specifically, we intend that all the procedures should work on personal computers, single and multi-user workstations, and mainframes. We have not, of course, been able to test the procedures on all combinations of computer, operating system, and `Pascal` compiler. We have, however, sampled a considerable range of machines, including the IBM AT and PS/2 with MS-DOS and Borland Turbo Pascal, the Apple Macintosh and Macintosh II with Think's Lightspeed Pascal and Borland Turbo Pascal, the DEC VAX with VMS and VAX PASCAL, and

IBM mainframes with IBM Pascal/VS. The procedures should in principle run without modification (except for implementation-dependent bindings of files to external file names) on any compiler that implements the ISO or ANSI/IEEE standard. Some further details on customizing the programs to specific operating systems and compilers are given in §1.2.

Diskettes or magtapes containing all the programs in this book are available from (depending on format) Cambridge University Press or Numerical Recipes Software. Ordering information can be found on the back of the title page, and at the end of the book. In validating the programs, we have taken the program source code directly from the machine-readable form of the book's manuscript, so as 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 (Pascal)*, 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. This `Pascal` edition has the benefit of about three years of reader comments and corrections to the `FORTRAN` and `C` editions. We have been very careful. But if you find a bug, please document it and tell us!

In the remaining sections of this chapter we first review some basic concepts of structured programming that are common to all modern programming languages. We then discuss, more specifically, the issues that arise in using `Pascal` as a language for scientific computing, illustrating and motivating some of the conventions that we have adopted in this book. Finally, we introduce some fundamental concepts in numerical analysis (roundoff error, truncation error, and so forth). Thereafter, we plunge into the substantive material of the book.

REFERENCES AND FURTHER READING:

Meeus, Jean 1982, *Astronomical Formulae for Calculators*, Second Edition, revised and enlarged (Richmond, Virginia: Willmann-Bell).

You will find references listed like this at the end of most sections of this book.

Because computer algorithms often circulate informally for quite some time before appearing in a published form, the task of uncovering "primary literature" is 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 references at the end of each section are arranged in order, roughly, according to their usefulness or relevance to that section. References relevant to more than one section are also repeated in the separate list at the back of the book.

1.1 Structured Programming

Numerical Recipes is not a textbook on software engineering. We are not going to teach you techniques for managing very large software construction projects. Our procedures are not "objects" in the sense of "object-oriented programming"; they are supposed to be, at most, algorithmically efficient *pieces* of objects, ready for you to knit into a more rigorous interface, as befits your need. In such an undertaking, you will almost certainly want to modify some (non-algorithmic) aspects of our procedures, introducing more elaborate handling of errors, for example, and more rigorous range checking of input arguments.

The fact that our procedures are not "software engineered" on the interface scale does not mean that we condone an undisciplined programming style. Far from it. At the scale of program size relevant in this book, we are strong proponents of *structured programming*, a discipline that we urge you to take seriously.

Within the general field of software engineering, the term *structured programming* has become a catch phrase that means different things to different people. Among the uninformed, some people think that structured programming means "break everything up into procedures and include a lot of comments." Others opine that structured programming is possible in some computer languages (e.g. `Pascal`) and not in others (e.g. `FORTRAN`). This, too, is wrong, although some languages certainly do make structured programming relatively easier or more difficult.

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 disk into his personal computer

and feeds it an elaborate, commercial program in binary executable form. The computer doesn't care whether that program is "structured" 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, the targets of a program's communication are surely human, not machine.

We are not yet done drawing the analogies. Music, poetry, and programming, all three as symbolic constructs of the human brain, are found to be 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, if not so universally recognized. 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;`.

At the next level, the terminology becomes vague, there being no standard vocabulary in use. Statements frequently come in "groups" or "blocks" which make sense only taken as a whole. For example,

```
swap := a[j];
a[j] := b[j];
b[j] := swap;
```

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. The more compulsive of programmers put in comments at this level, e.g., "Initialize" or "Exchange variables."

The next level is that of *control structures*. These are things like FOR-loops, IF's, and so on. They are so important that we will come back to them just below. Then we have the level of functions and procedures, and their interfaces. It is here that software engineering, beyond the scope of this book, has quite a lot to say. Finally, there is 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. Organization at this level is so closely linked to *what* the composer or programmer hopes to accomplish, rather than *how* it is accomplished, that we will offer few, if any, general principles. At this level, *you* are supposed to be already an expert.

Control Structures

An executing program unfolds in time, but not strictly in the linear order in which the statements are written. Program statements which affect the order in which statements are executed, or which affect whether statements are executed, are called *control statements.* Control statements never make useful sense by themselves. They only make sense 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 which occur over and over again in programming, and which 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 labeled statements 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 examples are now in order to make these considerations more concrete (see Figure 1.1.1).

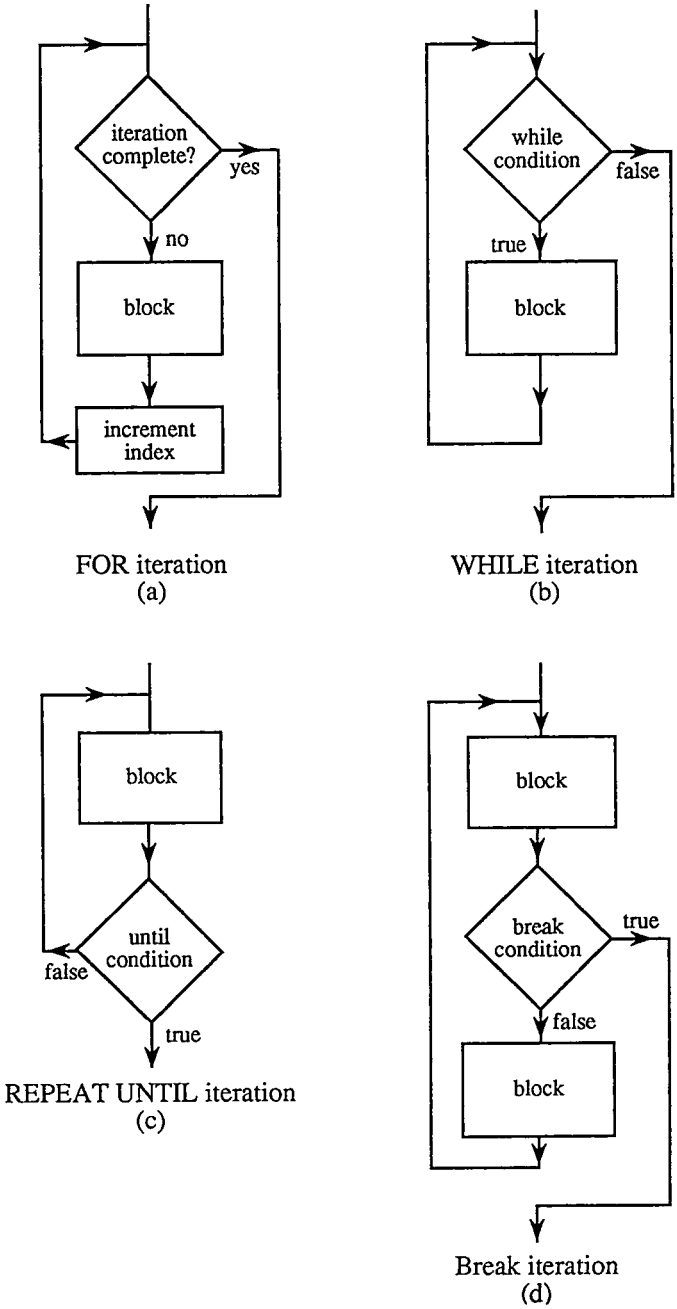
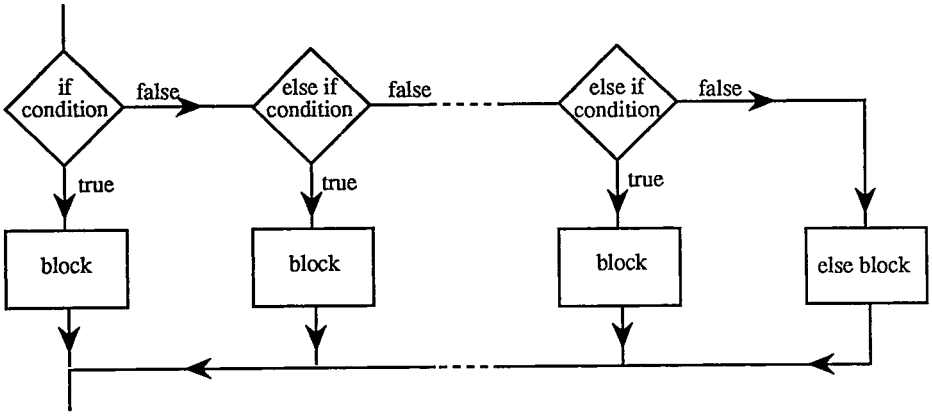
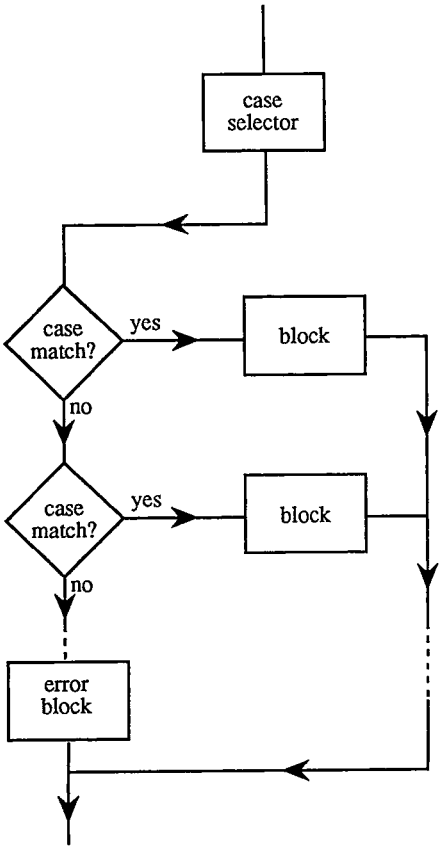


Figure 1.1.1. Standard control structures used in structured programming: (a) FOR iteration; (b) WHILE iteration; (c) REPEAT UNTIL iteration; (d) break iteration; (e) IF structure; (f) CASE structure (not recommended!)



IF structure
(e)



CASE structure (standard Pascal)
(f)

Figure 1.1.1. Standard control structures used in structured programming (see caption on previous page).

Catalog of Standard Structures

"FOR" iteration. In Pascal, simple iteration is performed with a FOR-loop, for example

```
FOR j := 2 TO 1000 DO BEGIN
   b[j] := a[j-1];
   a[j-1] := j
END;
```

Notice how we always indent the block of code that is acted upon by the control structure, leaving unindented the structure. Notice also our habit of putting the initial BEGIN on the same line as the FOR statement, instead of on the next line. This saves a full line of white space, and our publisher loves us for it.

"IF" structure. This structure in Pascal is similar to that found in C, Algol, FORTRAN and other languages, and typically looks like

```
IF (...) THEN BEGIN
   ...
END
ELSE IF (...) THEN BEGIN
   ...
END
ELSE BEGIN
   ...
END;
```

Since a compound-statement BEGIN...END is required only when there is more than one statement in a block, however, Pascal's IF construction can be somewhat less explicit than the corresponding structure in FORTRAN. Some care must be exercised in constructing nested IF clauses. For example, consider the following:

```
IF b > 3 THEN
   IF a > 3 THEN
      b := b+1
ELSE
   b := b-1;
```

As judged by the indentation used on successive lines, the intent of the writer of this code is the following: 'If b is greater than 3 and a is greater than 3, then increment b. If b is not greater than 3, then decrement b.' According to the rules of Pascal, however, the actual meaning is 'If b is greater than 3, then evaluate a. If a is greater than 3, then increment b, and if a is less than or equal to 3, decrement b.' The point is that an ELSE clause is associated with the most recent open IF statement, no matter how you lay it out on the page. Such confusions in meaning are easily resolved by the inclusion of a BEGIN...END. The above fragment should be written as

```
IF b > 3 THEN BEGIN
   IF a > 3 THEN
      b := b+1
END
ELSE
   b := b-1;
```

Alternatively, you can insert an ELSE followed by a null statement:

```
IF b > 3 THEN
   IF a > 3 THEN
      b := b+1
   ELSE
ELSE
   b := b-1;
```

Whenever you have a long set of instructions following a DO or IF, we recommend that you insert BEGIN...END's, even when they may in some instances be technically superfluous: they clarify your intent and improve the program.

Here is a working program which consists dominantly of IF control statements:

```
FUNCTION julday(mm,id,iyyy: integer): longint;
   In this routine julday returns the Julian Day Number which 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. Note
   that this routine will not run on systems with 2-byte integers. The type longint (4-byte
   integer) is discussed further in §1.2.
CONST
   igreg = 588829;                    Gregorian Calendar was adopted on Oct. 15, 1582.
VAR
   ja,jm,jy: integer;
   jul,jyyy: longint;
BEGIN
   IF iyyy = 0 THEN BEGIN
      writeln('there is no year zero.');
      readln;
   END;
   IF iyyy < 0 THEN iyyy := iyyy+1;
   IF mm > 2 THEN BEGIN               Here is an example of a block IF-structure.
      jy := iyyy;
      jm := mm+1
   END
   ELSE BEGIN
      jy := iyyy-1;
      jm := mm+13
   END;
   jul := trunc(365.25*jy)+trunc(30.6001*jm)+id+1720995;
   jyyy := iyyy;                                  Cast integer to longint.
   IF id+31*(mm+12*jyyy) >= igreg THEN BEGIN      Test whether to change to Gregorian Cal-
      ja := trunc(0.01*jy);                          endar.
      jul := jul+2-ja+trunc(0.25*ja)
   END;
   julday := jul
END;
```