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

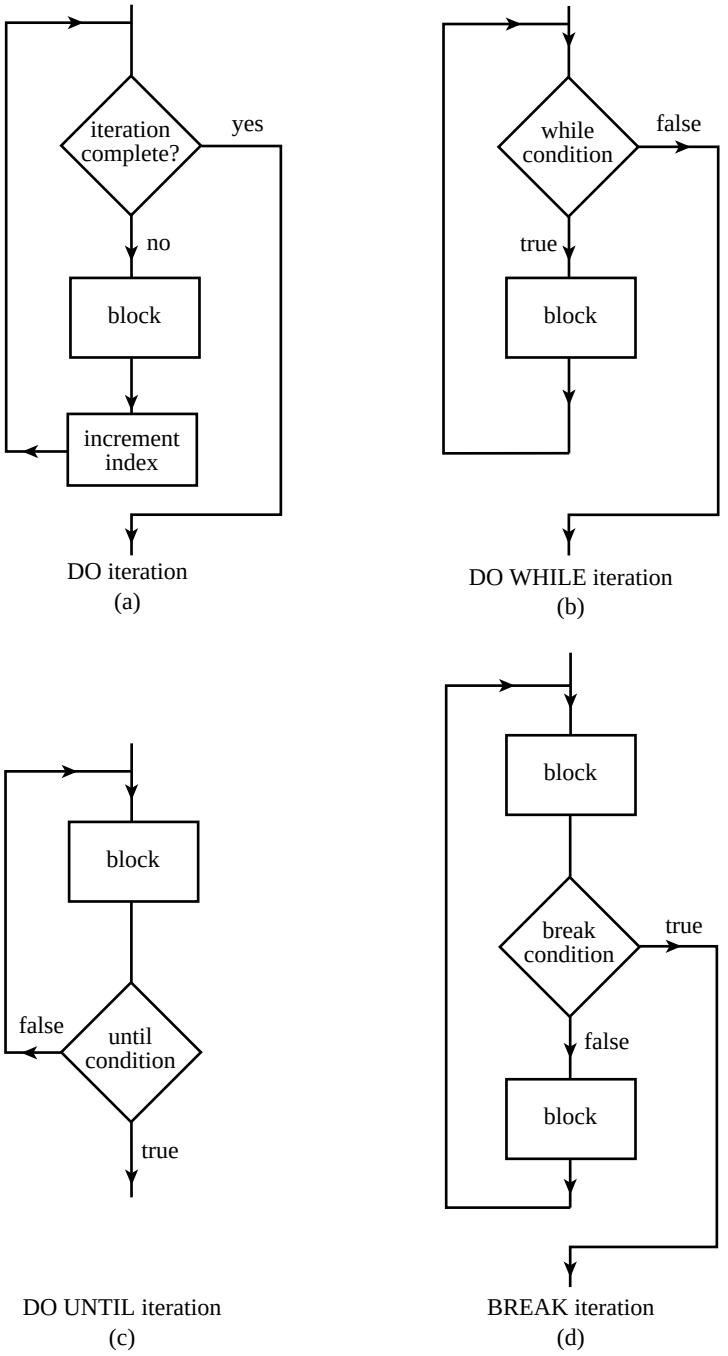
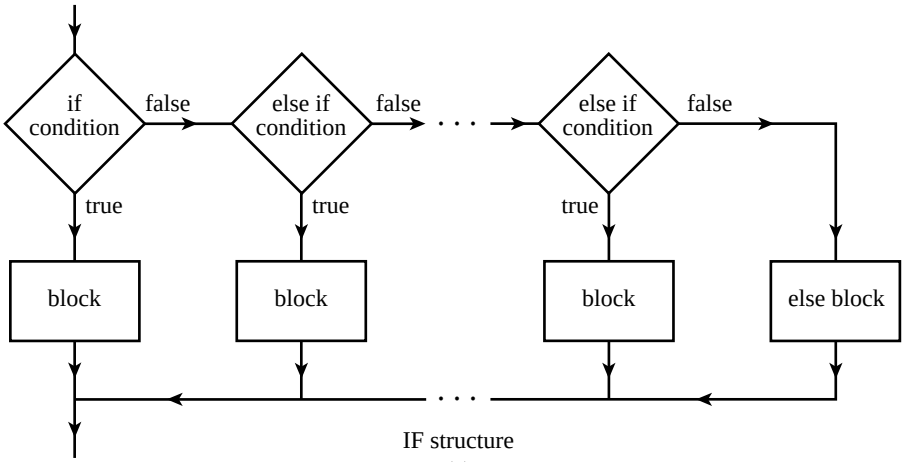
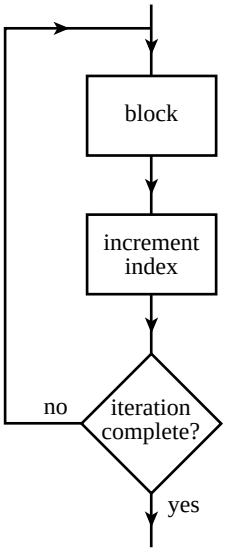


Figure 1.1.1. Standard control structures used in structured programming: (a) DO iteration; (b) DO WHILE iteration; (c) DO UNTIL iteration; (d) BREAK iteration; (e) IF structure; (f) obsolete form of DO iteration found in FORTRAN-66, where the block is executed once even if the iteration condition is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



IF structure
(e)



FORTRAN-66 DO (obsolete)
(f)

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

introduced in Digital Equipment Corporations's VAX-11 FORTRAN, the "enddo" statement is used as

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

In fact, it was a terrible mistake that the American National Standard for FORTRAN-77 (ANSI X3.9–1978) failed to provide an enddo or equivalent construction. This mistake by the people who write standards, whoever they are, presents us now, more than 15 years later, with a painful quandary: Do we stick to the standard, and clutter our programs with tame labels? Or do we adopt a nonstandard (albeit widely implemented) FORTRAN construction like enddo?

We have adopted a compromise position. Standards, even imperfect standards, are terribly important and highly necessary in a time of rapid evolution in computers and their applications. Therefore, all machine-readable forms of our programs (e.g., the diskettes that you can order from the publisher — see back of this book) are strictly FORTRAN-77 compliant. (Well, *almost* strictly: there is a minor anomaly regarding bit manipulation functions, see below.) In particular, do blocks always end with labeled continue statements, as in the first example above.

In the printed version of this book, however, we make use of typography to mitigate the standard's deficiencies. The statement label that follows the do is printed in small type — as a signal that it is a tame label that you can safely ignore. And, the word "continue" is printed as "enddo", which you may regard as a *very* peculiar change of font! The example above, in our adopted typographical format, is

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

(Notice that we also take the typographical liberty of writing the tame label after the "continue" statement, rather than before.)

A nested do loop looks like this:

```
do 12 j=1,20
    s(j)=0.
    do 11 k=5,10
        s(j)=s(j)+a(j,k)
    enddo 11
enddo 12
```

Generally, the numerical values of the tame labels are chosen to put the enddo's (labeled continue's on the diskette) into ascending numerical order, hence the do 12 before the do 11 in the above example.

IF structure. In this structure the FORTRAN-77 standard is exemplary. Here is a working program that consists dominantly of if control statements:

```
FUNCTION julday(mm,id,iyyy)
INTEGER julday,id,iyyy,mm,IGREG
PARAMETER (IGREG=15+31*(10+12*1582))        Gregorian Calendar adopted Oct. 15, 1582.
    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.
INTEGER ja,jm,jy
jy=iyyy
if (jy.eq.0) pause 'julday: there is no year zero'
if (jy.lt.0) jy=jy+1
if (mm.gt.2) then                           Here is an example of a block IF-structure.
    jm=mm+1
else
    jy=jy-1
    jm=mm+13
endif
julday=int(365.25*jy)+int(30.6001*jm)+id+1720995
if (id+31*(mm+12*iyyy).ge.IGREG) then       Test whether to change to Gregorian Calen-
    ja=int(0.01*jy)                             dar.
    julday=julday+2-ja+int(0.25*ja)
endif
return
END
```

(Astronomers number each 24-hour period, starting and ending at *noon*, with a unique integer, the Julian Day Number [7]. Julian Day Zero was a very long time ago; a convenient reference point is that Julian Day 2440000 began at noon of May 23, 1968. If you know the Julian Day Number that begins at noon of a given calendar date, then the day of the week of that date is obtained by adding 1 and taking the result modulo base 7; a zero answer corresponds to Sunday, 1 to Monday, ..., 6 to Saturday.)

Do-While iteration. Most good languages, except FORTRAN, provide for structures like the following C example:

```
while (n<1000) {
    n=2*n;
    j++;                    In C this has the meaning j=j+1.
}
```

In fact, many FORTRAN implementations have the nonstandard extension

```
do while (n.lt.1000)
    n=2*n
    j=j+1
enddo
```

Within the FORTRAN-77 standard, however, the structure requires a tame label:

```
17 if (n.lt.1000) then
    n=2*n
    j=j+1
  goto 17
  endif
```

There are other ways of constructing a Do-While in FORTRAN, but we try to use the above format consistently. You will quickly get used to a statement like 17 if as signaling this structure. Notice that the two final statements are not indented, since they are part of the control structure, not of the inside block.

Do-Until iteration. In Pascal, for example, this is rendered as

```
REPEAT
    n:=n DIV 2;          Pascal's integer divide is DIV.
    k:=k+1;
UNTIL (n=1);
```

In FORTRAN we write

```
19 continue
      n=n/2
      k=k+1
   if (n.ne.1) goto 19
```

Break. In this case, you have a loop that is repeated indefinitely until some condition *tested somewhere in the middle of the loop* (and possibly tested in more than one place) becomes true. At that point you wish to exit the loop and proceed with what comes after it. Standard FORTRAN does not make this structure accessible without labels. We will try to avoid using the structure when we can. Sometimes, however, it is plainly necessary. We do not have the patience to argue with the designers of computer languages over this point. In FORTRAN we write

```
13 continue
      [statements before the test]
      if (···) goto 14
      [statements after the test]
   goto 13
14 continue
```

Here is a program that uses several different iteration structures. One of us was once asked, for a scavenger hunt, to find the date of a Friday the 13th on which the moon was full. This is a program which accomplishes that task, giving incidentally all other Fridays the 13th as a by-product.

```
      PROGRAM badluk
      INTEGER ic,icon,idwk,ifrac,im,iybeg,iyend,iyyy,jd,jday,n,
     *     julday
      REAL TIMZON,frac
      PARAMETER (TIMZON=-5./24.)          Time zone −5 is Eastern Standard Time.
      DATA iybeg,iyend /1900,2000/        The range of dates to be searched.
C     USES flmoon,julday
      write (*,'(1x,a,i5,a,i5)') 'Full moons on Friday the 13th from',
     *     iybeg,' to',iyend
      do 12 iyyy=iybeg,iyend                        Loop over each year,
        do 11 im=1,12                               and each month.
          jday=julday(im,13,iyyy)                   Is the 13th a Friday?
          idwk=mod(jday+1,7)
          if(idwk.eq.5) then
            n=12.37*(iyyy-1900+(im-0.5)/12.)
              This value n is a first approximation to how many full moons have occurred
              since 1900. We will feed it into the phase routine and adjust it up or down until
```

```
                we determine that our desired 13th was or was not a full moon. The variable
                icon signals the direction of adjustment.
              icon=0
1             call flmoon(n,2,jd,frac)        Get date of full moon n.
              ifrac=nint(24.*(frac+TIMZON))   Convert to hours in correct time zone.
              if(ifrac.lt.0)then              Convert from Julian Days beginning at noon
                  jd=jd-1                         to civil days beginning at midnight.
                  ifrac=ifrac+24
              endif
              if(ifrac.gt.12)then
                  jd=jd+1
                  ifrac=ifrac-12
              else
                  ifrac=ifrac+12
              endif
              if(jd.eq.jday)then          Did we hit our target day?
                  write (*,'(/1x,i2,a,i2,a,i4)') im,'/',13,'/',iyyy
                  write (*,'(1x,a,i2,a)') 'Full moon ',ifrac,
*                      ' hrs after midnight (EST).'
                    Don't worry if you are unfamiliar with FORTRAN's esoteric input/output
                    statements; very few programs in this book do any input/output.
                  goto 2                      Part of the break-structure, case of a match.
              else                            Didn't hit it.
                  ic=isign(1,jday-jd)
                  if(ic.eq.-icon) goto 2      Another break, case of no match.
                  icon=ic
                  n=n+ic
              endif
              goto 1
2             continue
            endif
          enddo 11
        enddo 12
        END
```

If you are merely curious, there were (or will be) occurrences of a full moon on Friday the 13th (time zone GMT-5) on: 3/13/1903, 10/13/1905, 6/13/1919, 1/13/1922, 11/13/1970, 2/13/1987, 10/13/2000, 9/13/2019, and 8/13/2049.

Other "standard" structures. Our advice is to avoid them. Every programming language has some number of "goodies" that the designer just couldn't resist throwing in. They seemed like a good idea at the time. Unfortunately they don't stand the *test* of time! Your program becomes difficult to translate into other languages, and difficult to read (because rarely used structures are unfamiliar to the reader). You can almost always accomplish the supposed conveniences of these structures in other ways. Try to do so with the above standard structures, which really *are* standard. If you can't, then use straightforward, unstructured, tests and `goto`'s. This will introduce real (not tame) statement labels, whose very existence will warn the reader to give special thought to the program's control flow.

In `FORTRAN` we consider the ill-advised control structures to be

- assigned `goto` and `assign` statements
- computed `goto` statement
- arithmetic `if` statement

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.

You may have noticed that, by its looping over the months and years, the program `badluk` avoids using any algorithm for converting a Julian Day Number back into a calendar date. A routine for doing just this is not very interesting structurally, but it is occasionally useful:

```
SUBROUTINE caldat(julian,mm,id,iyyy)
INTEGER id,iyyy,julian,mm,IGREG
PARAMETER (IGREG=2299161)
    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.
INTEGER ja,jalpha,jb,jc,jd,je
if(julian.ge.IGREG)then                                  Cross-over to Gregorian Calen-
    jalpha=int(((julian-1867216)-0.25)/36524.25)           dar produces this correction,
    ja=julian+1+jalpha-int(0.25*jalpha)
else                                                     or else no correction.
    ja=julian
endif
jb=ja+1524
jc=int(6680.+((jb-2439870)-122.1)/365.25)
jd=365*jc+int(0.25*jc)
je=int((jb-jd)/30.6001)
id=jb-jd-int(30.6001*je)
mm=je-1
if(mm.gt.12)mm=mm-12
iyyy=jc-4715
if(mm.gt.2)iyyy=iyyy-1
if(iyyy.le.0)iyyy=iyyy-1
return
END
```

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

Some Habits and Assumed ANSI Extensions

Mentioning a few of our programming habits here will make it easier for you to read the programs in this book.

- We habitually use `m` and `n` to refer to the logical dimensions of a matrix, `mp` and `np` to refer to the physical dimensions. (These important concepts are detailed in §2.0 and Figure 2.0.1.)
- Often, when a subroutine or procedure is to be passed some integer `n`, it needs an internally preset value for the largest possible value that will be passed. We habitually call this `NMAX`, and set it in a `PARAMETER` statement. When we say in a comment, "largest value of `n`," we do not mean to imply that the program will fail algorithmically for larger values, but only that `NMAX` must be altered.
- A number represented by `TINY`, usually a parameter, is supposed to be much smaller than any number of interest to you, but not so small that it underflows. Its use is usually prosaic, to prevent divide checks in some circumstances.

As a matter of typography, the printed FORTRAN programs in this book, if typed into a computer exactly as written, would violate the FORTRAN-77 standard in a few trivial ways. The anomalies, which are *not* present in the machine-readable program distributions, are as follows:

- As already discussed, we use `enddo` followed by the statement label instead of `continue` preceded by the label.
- Standard FORTRAN reads no more than 72 characters on a line and ignores input from column 73 onward. Longer statements are broken up onto "continuation lines." In the printed programs in this book, some lines contain more than 72 characters. When the break to a continuation line is not shown explicitly, it should be inserted when you type the program into a computer.
- In standard FORTRAN, columns 1 through 6 on each line are used variously for (i) statement labels, (ii) signaling a comment line, and (iii) signaling a continuation line. We simplify the format slightly: To the left of the "program left margin," an integer is a statement label (not a "tame label" as described above), an asterisk (*) indicates a continuation line, and a "C" indicates a comment line. Comment lines shown in this way are generally either USES statements (see §1.0), or else "commented-out program lines" that are separately explained in each instance.

A small number of routines in this book require the use of functions that act bitwise on integers, e.g., bitwise "and" or "exclusive or". Unfortunately, although these functions are available in virtually all modern FORTRAN implementations, they are not a part of the FORTRAN-77 standard. Even more unfortunate is the fact that there are two different naming conventions in widespread use. We use the names `iand(i,j)`, `ior(i,j)`, `not(i)`, `ieor(i,j)`, and `ishft(i,j)`, for *and*, *or*, *not*, *exclusive-or*, and *left-shift*, respectively, as well as the subroutines `ibset(i,j)`, `ibclr(i,j)`, and the logical function `btest(i,j)` for *bit-set*, *bit-clear*, and *bit-test*. Some (mainly UNIX) FORTRAN compilers use a different set of names, with the following correspondences:

Us...	Them...	
`iand(i,j)`	= `and(i,j)`	
`ior(i,j)`	= `or(i,j)`	
`not(i)`	= `not(i)`	
`ieor(i,j)`	= `xor(i,j)`	
`ishft(i,j)`	= `lshft(i,j)`	
`ibset(i,j)`	= `bis(j,i)`	Note reversed arguments!
`ibclr(i,j)`	= `bic(j,i)`	Ditto!
`btest(i,j)`	= `bit(j,i)`	Ditto!

If you are one of "Them," you can either modify the small number of programs affected (e.g., by inserting FORTRAN statement function definitions at the beginning of the routines), or else link to an object file into which you have compiled the trivial functions that define "our" names in terms of "yours," as in the above table. Standards really are important!

Hexadecimal constants, for which there is no standard notation in FORTRAN compilers, occur at three places in Chapter 7: a program fragment at the end of §7.1,

and routines psdes and ran4 in §7.5. We use a notation like Z'3F800000', which is consistent with the new FORTRAN-90 standard, but you may need to change this to, e.g., x'3f800000', '3F800000'X, or even 16#3F800000. In extremis, you can convert the hex values to decimal integers; but note that most compilers will require a *negative* decimal integer as the value of a hex constant with its high-order bit set.

As already mentioned in §1.0, the notation a(1:m), in program comments and in the text, denotes the array element range a(1), a(2), ..., a(m). Likewise, notations like b(2:7) or c(1:m,1:n) are to be interpreted as denoting ranges of array indices.

CITED REFERENCES AND FURTHER READING:

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]

Meissner, L.P. and Organick, E.I. 1980, *Fortran 77 Featuring Structured Programming* (Reading, MA: Addison-Wesley). [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 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* (also called *integer*) or *floating-point* (also called *real*) 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.