

Additional Content for Use with *Classical Fortran*

The section numbers in this table show approximately where each addition will go if it is ever included in a Third Edition of the book; + means “between this section and the next at its level, adjusting subsequent section numbers to make room.” The new article for §12.4.5 probably belongs before the one that is currently last.

new §	title	contents
6.6.1+	Real-to-Integer Functions	truncating and rounding real values to integer
7.2.3	Computing the Inverse of a Function	finding $x = f^{-1}(y)$ from $F(X)$
9.6+	Undoing End-of-File	appending to a file; reattaching the keyboard
10.7+	Using <code>gnuplot</code> from FORTRAN	computing and graphing a table of coordinates
12.4.5	Typography	add: ordering subprogram parameters
18.1.3	Natural Vector to Integer Vector	convert from characters to <code>INTEGER*4</code>
18.1.4	Decimal Vector to Real Vector	convert from characters to <code>REAL*8</code>
18.2.2	Inserting One String Into Another	add: opening space for an insertion
18.4+	Extra-Precision Accumulation	computing $\mathbf{x}^\top \mathbf{y}$ more precisely
18.5+	Generating Pseudorandom Numbers	a generalized shift register algorithm

Copyright © 2025 Michael Kupferschmid, all rights reserved.

This Additional Content is licensed under CC-BY 4.0.

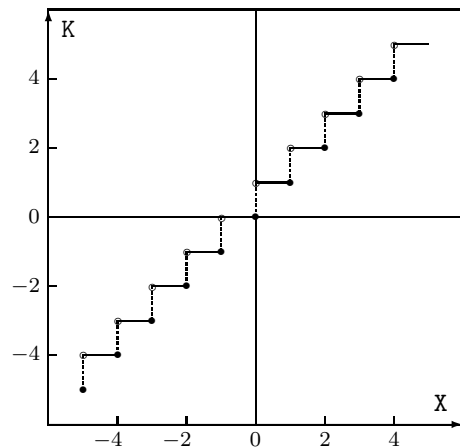
Anyone who complies with the terms specified in

<https://creativecommons.org/licenses/by/4.0/legalcode.txt>

may use the work in the ways therein permitted.

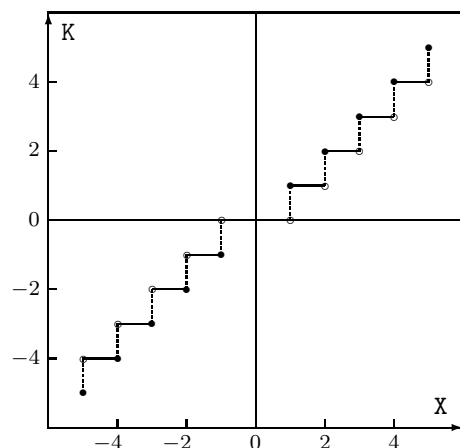
6.6.1+ Real-to-Integer Functions

I described in §4.4 how `IFIX` can be used to find the integer closest to a *positive* real value (top panel on next page). Often it is necessary to find the integer that is related to a real value in one of the slightly different ways illustrated below. Modern Fortran has built-in functions for ceiling and floor (see §17.1.3) and for the integer nearest a real value.

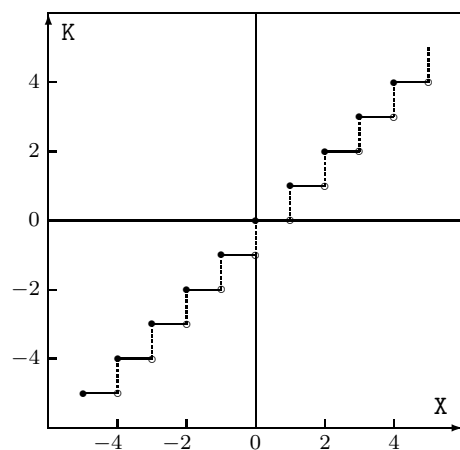


$K = \text{ICEIL}(X) = \text{lowest integer not less than } X$

```
FUNCTION ICEIL(X)
REAL*8 X
ICEIL=IFIX(SNGL(X))
IF(DFLOAT(ICEIL).LT.X) ICEIL=ICEIL+1
RETURN
END
```

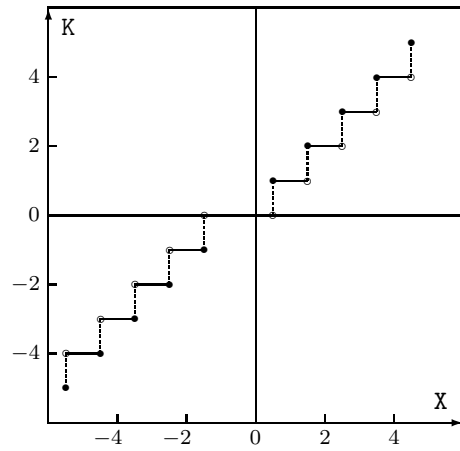


$K = \text{IFIX}(\text{SNGL}(X)) = \text{truncate fractional part}$

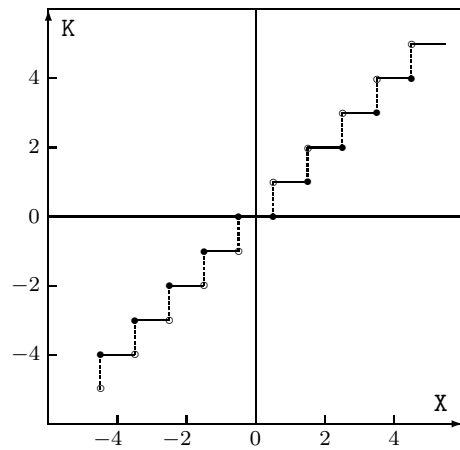


$K = \text{IFLOR}(X) = \text{highest integer not more than } X$

```
FUNCTION IFLOR(X)
REAL*8 X
IFLOR=IFIX(SNGL(X))
IF(DFLOAT(IFLOR).GT.X) IFLOR=IFLOR-1
RETURN
END
```



$K = \text{IFIX}(\text{SNGL}(X)+0.5) = \text{round positive values}$

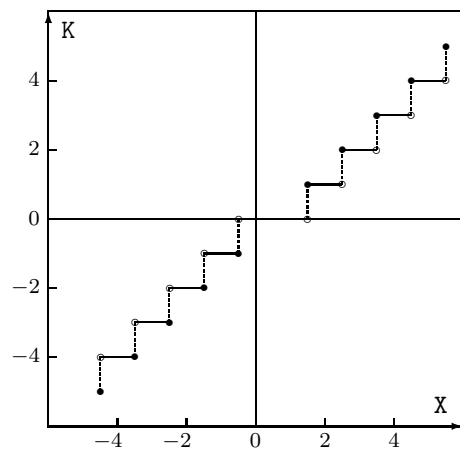


$K = \text{IROUND}(X) = \text{integer closest to } X$

```

FUNCTION IROUND(X)
REAL*8 X
IF(R.GT.0.DO) THEN
    IROUND=IFIX(SNGL(R)+0.5)
ELSE
    IROUND=IFIX(SNGL(R)-0.5)
ENDIF
RETURN
END

```



$K = \text{IFIX}(\text{SNGL}(X)-0.5) = \text{round negative values}$

7.2.3 Computing the Inverse of a Function

If $y = f(x)$ associates exactly one value of y with every value of x and exactly one value of x with every value of y , then $f(x)$ is a one-to-one function and it has an **inverse** $x = f^{-1}(y)$ [201, §1.6]. Here is an example in which the inverse function can be found analytically.

$$\begin{aligned} f(x) &= y = e^x - 2 \\ e^x &= y + 2 \\ x &= \ln(y + 2) = f^{-1}(y) \end{aligned}$$

At a point x where $y = 0$, $f(x) = 0$ and $f^{-1}(0) = x$. Here the x at which $f(x) = 0$ is $f^{-1}(0) = \ln(0 + 2) \approx 0.693$.

When the equation $y = 0$ cannot be solved analytically, the x at which $f(x) = 0$ can be approximated by a numerical method such as bisection, which we implemented in the `BISECT` subroutine of §7.2.2. Sometimes, however, it is more natural to think of the root as $x = f^{-1}(0)$ and approximate it by a numerical method that computes the inverse function. The `FINV` function subprogram listed below and on the next page evaluates the inverse of a given function at a particular point.

```

1 C
2 Code by Michael Kupferschmid
3 C
4     FUNCTION FINV(F,Y,TOL, XL,XR, RC)
5 C     This routine returns the value of X such that Y=F(X).
6 C
7 C     RC meaning
8 C     --
9 C     0 X was found
10 C     1 convergence was not attained in 100 bisections
11 C     2 there is no root in the interval [XL,XR]
12 C
13 C     variable meaning
14 C     -----
15 C     DABS      Fortran function returns |REAL*8|
16 C     F         function value at trial point
17 C     FL        error in function inverse at left end of interval
18 C     FR        error in function inverse at right end of interval
19 C     FX        error in function inverse at trial point
20 C     I         index on bisections
21 C     RC        return code; see table above
22 C     TOL       convergence tolerance on X
23 C     X         trial point
24 C     XL        left end of interval on X
25 C     XR        right end of interval on X
26 C     Y         value at which to find corresponding X
27 C
28 C     formal parameters
29     REAL*8 FINV,F,Y,TOL,XL,XR
30     EXTERNAL F
31     INTEGER*4 RC
32 C
33 C     local variables
34     REAL*8 X,FX,FL,FR
35 C
36 C -----
```

```

37 C
38 C      use bisection to solve F(X)-Y=0 for X
39      RC=0
40      FL=F(XL)-Y
41      FR=F(XR)-Y
42      DO 1 I=1,100
43          X=0.5DO*(XL+XR)
44          FX=F(X)-Y
45          FINV=X
46          IF(DABS(XR-XL).LE.TOL) RETURN
47          IF(FX*FL .LT. 0.DO) GO TO 2
48          IF(FX*FR .LT. 0.DO) GO TO 3
49 C
50 C      there is no root in the interval
51      RC=2
52      RETURN
53 C
54 C      the root is in the left half
55      2      XR=X
56          FR=FX
57          GO TO 1
58 C
59 C      the root is in the right half
60      3      XL=X
61          FL=FX
62      1 CONTINUE
63 C
64 C      convergence was not attained in 100 iterations
65      RC=1
66      RETURN
67      END

```

FINV is a slight rearrangement of BISECT. It begins with a preamble [1-36] (see §12.3.2) describing [5] what the routine does, what values the return code RC can have [7-11], and what the variables mean [13-26]. It also declares the variables that need to be explicitly typed [28-34]. The FUNCTION subprogram F computes $f(x)$, and because its name is passed as a parameter [4] it is declared EXTERNAL [30]. A comment line [36] drawn across the page ends the preamble.

The executable code [38-62] uses bisection to find the x where $f(x) - y = 0$, which as we saw above is also $f^{-1}(y)$. It begins by [39] assuming the calculation will succeed and [40-41] finding $f(x) - y$ at the endpoints of the interval that is assumed to contain the root. Then [43] it finds the midpoint of the interval and [44] the value of $f(x) - y$ there. If [46] the interval of uncertainty in X is small enough the routine returns with [45] the current X as $f^{-1}(y)$. Otherwise it [47-48] determines which half of the remaining interval contains the root and [54-61] halves the interval. If neither half contains a root [50-52] the routine returns with RC=2. If 100 bisections fail to yield a root within TOL, the routine returns [64-66] with RC=1.

In §0.1 we used bisection to find an x where $f(x) = \sin(x) - \frac{1}{2}x = 0$, which amounts to finding $f^{-1}(0)$. That inverse function cannot be derived analytically, but the program on the next page uses FINV to compute its values numerically.

```

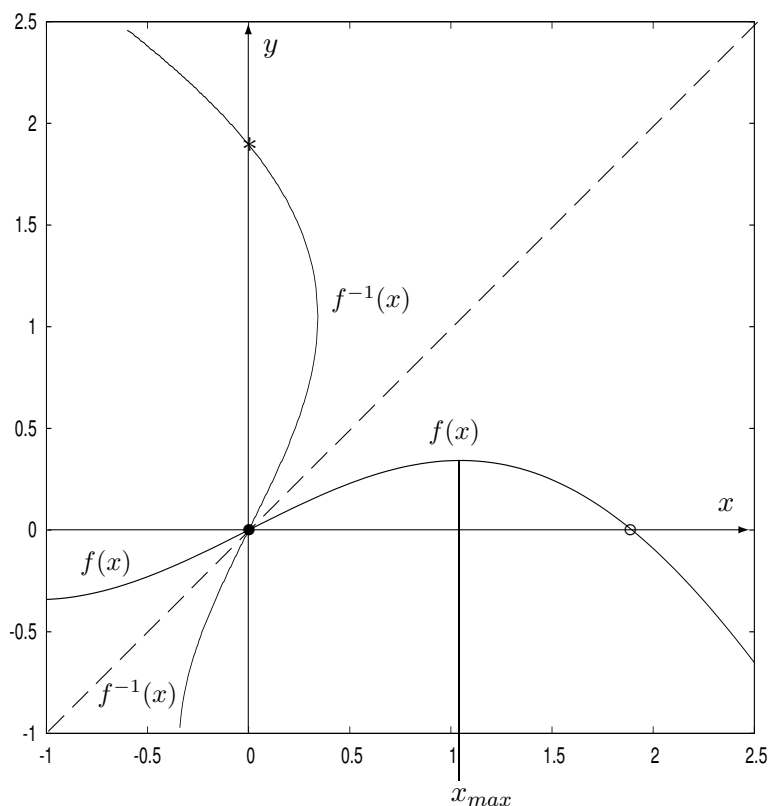
1      REAL*8 X,XMAX,XL,XR,Y,F,FINV,FL,FR
2      INTEGER*4 RC
3 C
4      DO 1 I=1,101
5          X=-1.D0+3.5*0.01D0*DFLOAT(I-1)
6          Y=F(X)
7          PRINT *,X,Y
8      1 CONTINUE
9 C
10     PRINT *
11     XMAX=DACOS(0.5D0)
12     FL= 0.3424D0
13     FR=-0.6D0
14     DO 2 I=1,101
15         Y=FR+(FL-FR)*0.01D0*DFLOAT(I-1)
16         XL=XMAX
17         XR=2.5D0
18         X=FINV(F,Y,0.01D0, XL,XR, RC)
19         PRINT *,Y,X
20     2 CONTINUE
21 C
22     FR= 0.3424D0
23     FL=-0.34D0
24     DO 3 I=1,101
25         Y=FR+(FL-FR)*0.01D0*DFLOAT(I-1)
26         XL=-1.D0
27         XR=XMAX
28         X=FINV(F,Y,0.01D0, XL,XR, RC)
29         PRINT *,Y,X
30     3 CONTINUE
31     STOP
32     END
33 C
34     FUNCTION F(X)
35     REAL*8 F,X
36     F=DSIN(X)-0.5D0*X
37     RETURN
38     END

```

This program begins [4-8] by computing points on the graph of $y = f(x)$ and printing their coordinates. When these data are plotted we get the curve shown on the next page, from which it is clear that $f(x)$ is one-to-one on the two intervals $[-1, x_{max}]$ and $[x_{max}, 2.5]$. The inverse function must be calculated separately over these two intervals. The function $f(x)$ has its maximum where $f'(x) = \cos(x) - \frac{1}{2} = 0$, at $x_{max} = \arccos(\frac{1}{2}) \approx 1.0472$.

To separate the coordinates of the inverse from those of the function, the program [10] prints a blank line. Then [11-20], for each of 101 values of y in the range of $f(x)$ over $x \in [x_{max}, 2.5]$, it [18] uses FINV to find the corresponding x and [19] prints the coordinates of the point. In the picture these points correspond to the top part of the inverse curve. The upper limit on Y [12] FL=0.3424, is slightly less than the actual height of the curve $f(x_{max}) \approx 0.34243$ so that the horizontal line grazes

the top of the curve rather than being precisely tangent to it. The final stanza of the program [22-32] repeats the calculation for $x \in [-1, x_{max}]$ to find the bottom part of the inverse curve. The FUNCTION subprogram F(X) is listed [34-38] last.



Because $f(x)$ is not one-to-one over its entire range neither is $f^{-1}(x)$, and because $f^{-1}(x)$ associates two values of y with some values of x it is a relation but not a function. The picture shows that $f(x) = 0$ at $x = 0$ • and $x \approx 1.89549$ ○ (see the solution to Exercise 0.10.5) and that $f^{-1}(0)$ is either $x = 0$ • or $x \approx 1.89549$ *. Notice that the graph of the inverse function is the mirror image of the graph of the function reflected across the line $y = x$ [201, p43].

In Chapter 9 you will learn how to capture the output of a program in a file, and the Additional Content for Chapter 10 explains how **gnuplot** can be used to read a file of data and generate graphs like the one above.

[201] **Weir, Maurice D.** and **Hass, Joel**, *Thomas' Calculus Early Transcendentals*, Addison-Wesley, 2011.

9.6+ Undoing End-of-File

When READ sees a ^D or encounters the end of a disk file, the I/O library marks that condition by setting a flag. This Section discusses two unusual situations in which persistence of that flag is inconvenient, and shows ways in which it can be reset.

Appending to a file. Some I/O library implementations provide one of these options

```
OPEN(UNIT=unit, FILE=file, POSITION='APPEND')
OPEN(UNIT=unit, FILE=file, ACCESS='APPEND')
```

for positioning the line pointer at the end of a file. If neither option is supported, I suggested in §9.6.0 reading the file until its end is reached and then writing more lines to it. That approach fails in implementations, including **gfortran**, that do not permit the file to be written to or read from after the end-of-file flag has been set. In that case we can resort to the strategy embodied in the **APPEND** routine listed below.

```

1      SUBROUTINE APPEND(NUNIT)
2  C      This routine positions the line pointer at the end of the file
3  C      attached to unit NUNIT so that it can be appended to, without
4  C      leaving the file closed so that it cannot be written.
5  C
6  C      variable  meaning
7  C      -----  -----
8  C      I          index on the lines in the file
9  C      LNUM       number of line just read
10 C      NUNIT      number of logical I/O unit attached to the file
11 C
12 C -----
13 C
14 C      count the lines in the file
15      REWIND(UNIT=NUNIT)
16      LNUM=0
17      2 READ(NUNIT,*,END=1)
18      LNUM=LNUM+1
19      GO TO 2
20 C
21 C      reread those lines without setting the EOF flag
22      1 REWIND(UNIT=NUNIT)
23      DO 3 I=1,LNUM
24          READ(NUNIT,*)
25      3 CONTINUE
26      RETURN
27      END
```

This routine 14-19 reads the file once to its end, counting its lines. Then 21-25 it reads that number of lines from the file again, stopping before the end-of-file flag is set. This positions the line pointer to the next line after those already present, so that more lines can be written into the file.

Reattaching unit 5 to the keyboard. Recall from §9.1 that logical unit number 5 is attached by default to the keyboard, which Unix refers to by a device name such as

`/dev/pts/0`. Sometimes it is desirable to take the `END=` exit from `READ` when an interactive user enters `^D`, perform some action, and eventually resume reading input from the keyboard. After the `^D` is received standard-in remains attached to the device, but if the end-of-file flag is persistent attempts to read from the keyboard elicit a Fortran runtime error message such as `Sequential READ not allowed after EOF`. To re-enable unit 5 for input from the keyboard after doing whatever is appropriate in response to the end-of-file, it is necessary to close and reopen the unit. The `UNCTLD` routine listed on the next page finds the device name of the keyboard, closes the unit, and reopens it on that device.

The code begins [29] by using `INQUIRE` to find the name of the file or device that is attached to unit 5.

If [30-34] the name begins with the string `stdin` then unit 5 is reading from a redirection (see §9.4) or a Unix pipe (see §14.1.0), in which case the end-of-file cannot be rescinded; the routine closes the unit and returns with `RC=3`.

If [36-40] the name is blank then the unit is already closed. We have no way of knowing what it had been attached to, so the routine returns with `RC=2`.

If [41-46] the name does *not* begin with the string `/dev/` then unit 5 is attached to a file rather than to the keyboard, and in this situation also the end-of-file cannot be rescinded; the routine closes the unit and returns with `RC=1`.

Otherwise [49-52] the routine assumes that unit 5 is attached to the keyboard, so it can clear the end-of-file flag by closing the unit and reopening it on the device to which it was previously attached. To signal this normal outcome the routine returns `RC=0`.

```

1 C
2 SUBROUTINE UNCTLD(RC)
3 C This routine reestablishes unit 5 as the keyboard after EOF.
4 C
5 C RC meaning
6 C -- -----
7 C 0 all went well; unit 5 was closed and reopened
8 C 1 unit 5 was attached to a file; unit 5 was closed
9 C 2 unit 5 was already closed; nothing was done
10 C 3 unit 5 was a redirect or pipe target; unit 5 was closed
11 C
12 C variable meaning
13 C -----
14 C FYLE the file or device name to which unit 5 is attached
15 C NAM the first 5 characters of FYLE
16 C RC return code; see table above
17 C
18 C formal parameter
19 INTEGER*4 RC
20 C
21 C overlay to extract the first 5 characters of FYLE
22 CHARACTER*24 FYLE
23 CHARACTER*5 NAM
24 EQUIVALENCE(FYLE,NAM)
25 C
26 C -----
27 C
28 C where is unit 5 attached now?
29 INQUIRE(UNIT=5,NAME=FYLE)
30 IF(NAM.EQ.'stdin') THEN
31 C unit 5 is the target of a redirect or pipe
32 CLOSE(5)
33 RC=3
34 RETURN
35 ENDIF
36 IF(NAM.EQ.' ') THEN
37 C unit 5 is closed
38 RC=2
39 RETURN
40 ENDIF
41 IF(NAM.NE. '/dev/') THEN
42 C unit 5 is attached to a file
43 CLOSE(5)
44 RC=1
45 RETURN
46 ENDIF
47 C
48 C unit 5 is attached to some /dev, presumably the keyboard
49 CLOSE(5)
50 OPEN(UNIT=5,FILE=FYLE)
51 RC=0
52 RETURN
53 END

```

10.7+ Using gnuplot from FORTRAN

In §10.7, I mentioned that `gnuplot` can be used to draw graphs (see www.gnuplot.info and [108]). If a function whose graph is to be drawn is known to `gnuplot`, it can be specified simply by giving its formula. To plot the graph of a function whose formula is *not* recognized, you can write a FORTRAN program to generate coordinates on the curve and then use `gnuplot` to graph the data. That process can be simplified somewhat by also having your program construct an appropriate command file and invoke `gnuplot` to use it.

The Struve functions (see https://en.wikipedia.org/wiki/Struve_function), which are solutions to certain differential equations, are unknown to `gnuplot`. The program below begins by [6-16] computing points on the graph of the Struve function $L_{-2}(x)$ and writing them to the file `struve.data`. The first few lines of `struve.data` are shown at the top left on the next page. The FUNCTION subprogram `STRUVE` [11] uses formula 12.2.1 from [35, p498] to evaluate the function, and it returns `RC=0` if it is successful, but its inner workings need not concern us here.

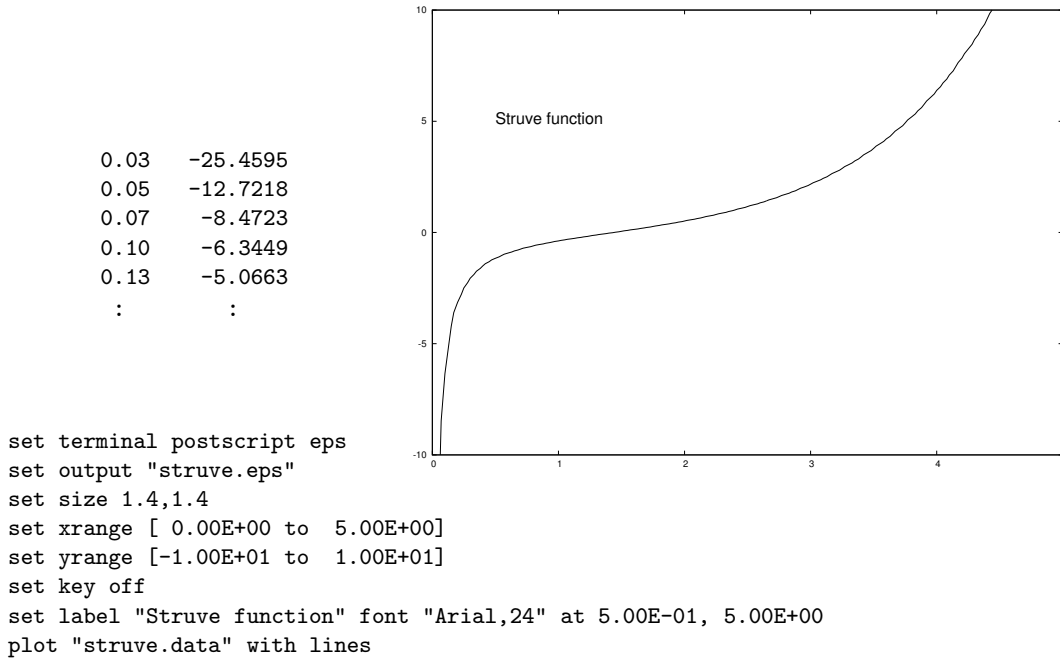
```

1 C      This program plots the graph of a Struve function.
2 C
3      REAL*8 X,L,STRUVE
4      INTEGER*4 RC,NPTS/200/
5 C
6 C      generate coordinates on the curve
7      OPEN(UNIT=1,FILE='struve.data')
8      NU=-2
9      DO 1 I=1,NPTS
10         X=5.DO*DFLOAT(I)/DFLOAT(NPTS)
11         L=STRUVE(X,NU, RC)
12         IF(RC.NE.0) STOP
13         WRITE(1,901) X,L
14 901     FORMAT(F5.2,1X,F10.4)
15 1 CONTINUE
16     CLOSE(UNIT=1)
17 C
18 C      generate gnuplot commands to graph the data
19     OPEN(UNIT=9,FILE='struve.cmds')
20     CALL WRTGNU('struve',6,0.,5.,-10.,10.,
21 ;             'Struve function',15,0.5,5.0)
22     CLOSE(UNIT=9)
23 C
24 C      invoke gnuplot to make the graph
25     CALL SYSTEM('cat struve.cmds | gnuplot')
26     STOP
27     END

```

Next [18-22] the program invokes subroutine `WRTGNU`, which is described on the next page, to construct a file of `gnuplot` commands. That file, `struve.cmds`, is listed below the input data at the top of the next page.

Finally it uses the `SYSTEM` subroutine first mentioned in §6.6.2 to issue a Unix command that invokes `gnuplot` with `struve.cmds` connected to its standard input. That causes `gnuplot` to read `struve.data` and generate its graph in `struve.eps`, which is displayed in the top right corner of the next page.



Subroutine WRTGNU is listed on the next two pages. It begins with a preamble [28-79] (see §12.3.2). This describes what the routine does, explains what its variables mean, declares the type and size of the parameters NAME and TITLE, and initializes the character string templates OUT, LBL, and PLT. A comment line [79] drawn across the page separates the preamble from the executable statements that follow it.

The stanzas of executable code on page 4 use the input parameters of WRTGNU to construct the `gnuplot` commands listed above. In some cases this can be accomplished with `WRITE` and `FORMAT` statements like those you learned about in §9.1, but in others it is necessary to insert characters into a template, edit the result, and write it as described in §10.2. To insert one character string into another this routine invokes the subroutine `STRINS`, which is described in §18.2.2. For example, to write the `gnuplot` command

```
set output "struve.eps"
```

`STRINS` is used [86] to insert the $LN \leq 24$ characters of NAME into the 41-character template OUT, starting at character position 13 of OUT. The parameter 2 that is passed to `STRINS` tells it to remove blanks that result from NAME not filling the space provided for it in OUT. The result string is returned in OUTPUT and the index of its last nonblank is returned in L, so the implied `DO` [87] (see §9.3) writes it omitting any trailing blanks.

This version of WRTGNU does not print axis labels, and it omits the `set label` command entirely if 0 is passed for the parameter LT, because often a Postscript plot will be included in a document where all such decorations are provided in another way (such as by using `overpic` in \LaTeX 2_ε). Many aspects of a graph's appearance that must be left to `gnuplot`, such as the font size it uses for tic-mark numbering, can be controlled by means of commands I have not used here. You might find it desirable to generalize WRTGNU so that it produces additional or more complicated `gnuplot` commands and can be used in every situation where you want to take the approach illustrated here for plotting a graph.

```

28 C
29 Code by Michael Kupferschmid
30 C
31 SUBROUTINE WRTGNU(NAME, LN, XMIN, XMAX, YMIN, YMAX, TITLE, LT, XL, YL)
32 C This routine constructs a file of commands that can be used
33 C to make gnuplot generate a graph of tabular data.
34 C
35 C variable meaning
36 C -----
37 C K index on the characters of a string
38 C L index of last nonblank in a character string
39 C LABEL LBL with TITLE and its coordinates filled in
40 C LBL 'set label ? at ?,?'
41 C LENGTH function returns index of last nonblank in a string
42 C LN number of characters in NAME
43 C LT number of characters in TITLE
44 C NAME identifier to use in filenames
45 C OUT 'set output ?.eps'
46 C OUTPUT OUT with NAME filled in
47 C PLOT PLT with NAME filled in
48 C PLT 'plot "?.data" with lines'
49 C SPACEL routine removes leading and extra blanks
50 C STRINS routine inserts one character string into another
51 C TITLE title for the graph
52 C X XL as characters
53 C XL x-coordinate of graph label
54 C XMAX high end of range of x-values to show
55 C XMIN low end of range of x values to show
56 C Y YL as characters
57 C YL y-coordinate of graph label
58 C YMAX high end of range of y-values to show
59 C YMIN low end of range of y-valeus to show
60 C
61 C formal parameters
62 CHARACTER*1 NAME(LN), TITLE(LT)
63 C
64 C character-string templates
65 CHARACTER*1 OUT(41)/'s','e','t',' ','o','u','t','p','u','t',
66 ; ' ','"', 24*' ', ' ','e','p','s','"/
67 CHARACTER*1 OUTPUT(41)
68 CHARACTER*1 LBL(75)/'s','e','t',' ','l','a','b','e','l',' ',
69 ; '"', 24*' ', '"', ' ','f','o','n','t',' ',
70 ; '"', 'A','r','i','a','l',' ','2','4','"',
71 ; ' ','a','t',' ', 9*' ', ' ', 9*' '/
72 CHARACTER*1 LABEL(75)
73 CHARACTER*1 PLT(47)/'p','l','o','t',' ',' ', 24*' ',
74 ; ' ','d','a','t','a','"', ' ',
75 ; 'w','i','t','h',' ', ' ','l','i','n','e','s'/'
76 CHARACTER*1 PLOT(47)
77 CHARACTER*9 X, Y
78 C
79 C -----

```

```

80 C
81 C      put the graph in a .eps file
82      WRITE(9,901)
83 901 FORMAT('set terminal postscript eps')
84 C
85 C      specify the name of the output .eps file
86      CALL STRINS(OUT,41,NAME,LN,13,2, OUTPUT,L)
87      WRITE(9,902) (OUTPUT(K),K=1,L)
88 902 FORMAT(41A1)
89 C
90 C      specify the size of the graph
91      WRITE(9,903)
92 903 FORMAT('set size 1.4,1.4')
93 C
94 C      specify the ranges of x and y values to show
95      WRITE(9,904) XMIN,XMAX
96 904 FORMAT('set xrange [' ,1PE9.2,' to ' ,1PE9.2,']')
97      WRITE(9,905) YMIN,YMAX
98 905 FORMAT('set yrange [' ,1PE9.2,' to ' ,1PE9.2,']')
99 C
100 C      turn off the default legend
101      WRITE(9,906)
102 906 FORMAT('set key off')
103 C
104 C      specify the graph label and where it should appear
105      IF(LT.GT.0) THEN
106          WRITE(X,900) XL
107          WRITE(Y,900) YL
108 900      FORMAT(1PE9.2)
109          CALL STRINS(LBL,75,X,9,57,0, LABEL,L)
110          CALL STRINS(LABEL,75,Y,9,67,0, LABEL,L)
111          CALL STRINS(LABEL,75,TITLE,LT,12,2, LABEL,L)
112          CALL SPACEL(LABEL,L)
113          L=LENGTH(LABEL,L)
114          WRITE(9,907) (LABEL(K),K=1,L)
115 907      FORMAT(75A1)
116      ENDIF
117 C
118 C      specify the name of the input .data file
119      CALL STRINS(PLT,47,NAME,LN,7,2, PLOT,L)
120      WRITE(9,908) (PLOT(K),K=1,L)
121 908 FORMAT(47A1)
122      RETURN
123      END

```

12.4.5 Typography

Arrange subprogram parameters in a meaningful order. The SUBROUTINE below has the parameters listed in alphabetical order on the right.

	SUBROUTINE WATWAY(parameters) declarations for the parameters																			
C	I(J)=I(J)+1 A=1.23 PRINT *,B																			
C	RETURN END																			
		<table><tr><th>name</th><th>type</th><th>shape</th></tr><tr><td>A</td><td>REAL*4</td><td>scalar</td></tr><tr><td>B</td><td>REAL*8</td><td>scalar</td></tr><tr><td>I</td><td>INTEGER*4</td><td>array of N elements</td></tr><tr><td>J</td><td>INTEGER*4</td><td>scalar</td></tr><tr><td>N</td><td>INTEGER*4</td><td>adjustable dimension</td></tr></table>	name	type	shape	A	REAL*4	scalar	B	REAL*8	scalar	I	INTEGER*4	array of N elements	J	INTEGER*4	scalar	N	INTEGER*4	adjustable dimension
name	type	shape																		
A	REAL*4	scalar																		
B	REAL*8	scalar																		
I	INTEGER*4	array of N elements																		
J	INTEGER*4	scalar																		
N	INTEGER*4	adjustable dimension																		

In what order should they appear in the calling sequence of WATWAY? Several policies are evident, though perhaps not always deliberate, in the code that people write.

- List the parameters in arbitrary order, in alphabetical order, or in order of size. Doing this forfeits the opportunity to use the order for conveying useful information about the subprogram. Execution speed is not affected by the order of formal parameters (but see §15.2.10 for parameters passed in **COMMON**).
- List the parameters in the order they are encountered in the code of the subprogram, with each array followed immediately by its adjustable dimensions. The order of the parameters reflects the sequence of processes that constitute the calculation performed. For our example this yields

```
SUBROUTINE WATWAY(I,N,J,A,B)
```

- Put first the parameters that must be given a value before entering the subprogram and last the parameters that are given a value by the subprogram. If there are parameters that must have a value on entry but are changed by the subprogram, put them in the middle. For our example this yields

```
SUBROUTINE WATWAY(N,J,B, I, A)
```

or some variation in which N, J, and B appear in a different order. In Modern Fortran [98, §6.4] a dummy argument can be given an **INTENT** attribute that tells whether it is an input, an output, or both, which suggests a consensus for this approach. It is still desirable to make the distinction clear in the calling sequence.

Parameter declarations in the subprogram should be grouped together and separated from the declarations of local variables. Parameters having the same type should be included in the same type statement for conciseness, unless they won't all fit on one line or the goal of clarity is better served by separating them.

18.1.3 Numeral Vector to Integer Vector

A natural generalization of the DTB routine described in §18.1.2 is PRSI4V, which is listed below and on the next page. It uses DTB repeatedly to translate a row of numeral strings into a vector of INTEGER*4 values.

```
1 C
2 Code by Michael Kupferschmid
3 C
4     SUBROUTINE PRSI4V(TEXT,LTXT,LI, INTS,NI,RC)
5 C     This routine returns in INTS the integers in TEXT, or zeros.
6 C     TEXT is destroyed in the process.
7 C
8 C     RC  meaning
9 C     --  -----
10 C     0   all went well
11 C     1   TEXT contains more than LI integers
12 C     2   a string in TEXT contains more than 10 digits
13 C     3   TEXT contains tokens other than integers
14 C
15 C     variable  meaning
16 C     -----  -----
17 C     DTB       routine converts numerals to INTEGER*4
18 C     I         index on integers in INTS
19 C     INT       an integer found
20 C     INTS      the integers found
21 C     K         index on the characters of TEXT
22 C     L         length of what is left of TEXT
23 C     LENGTH    function returns index of last nonblank in a string
24 C     LI        number of integers expected
25 C     LTXT      number of characters in TEXT
26 C     NDIG      number of characters in a token
27 C     NI        number of integers found
28 C     RC        return code from this routine; see table above
29 C     RCD       return code from DTB; 0 => all went well
30 C     SHIFTL    routine removes leading blanks from a string
31 C     TEXT      character string to be parsed
32 C
33 C     formal parameters
34 C     CHARACTER*1 TEXT(LTXT)
35 C     INTEGER*4 INTS(LI),RC
36 C
37 C     local variables
38 C     INTEGER*4 RCD
39 C
40 C -----
41 C
42 C     assume that no integers will be found
43 C     RC=0
44 C     DO 1 I=1,LI
45 C         INTS(I)=0
46 C 1 CONTINUE
47 C     NI=0
48 C     IF(LTXT.LE.0) RETURN
```

If [48](#) TEXT is empty the routine returns with NI and INTS set to zeros and, because this case is a logical possibility, with RC=0 to indicate that nothing went wrong.

Here [49] L is initialized to the nonzero LTXT and control enters a free loop [52-95] to process the blank-separated tokens in TEXT one at a time. First [52] any leading blanks are removed; if [55-56] this results in an empty string, the routine returns normally with [43] RC=0. Otherwise [58-63] it counts the characters in the next token.

```

49      L=LTXT
50 C
51 C      shift the string left to put the next token first
52      5 CALL SHIFTL(TEXT,L)
53 C
54 C      when the string is empty we are done
55      L=LENGTH(TEXT,L)
56      IF(L.EQ.0) RETURN
57 C
58 C      not done yet; find the length of the first token
59      NDIG=0
60      DO 2 K=1,L
61          IF(TEXT(K).EQ.' ') GO TO 3
62          NDIG=NDIG+1
63      2 CONTINUE
64 C
65 C      might this token be an integer we can represent?
66      3 IF(TEXT(1).NE.'+' .AND. TEXT(1).NE.'-') THEN
67          IF(NDIG.GT.10) THEN
68              RC=2
69              RETURN
70          ENDIF
71      ELSE
72          IF(NDIG.GT.11) THEN
73              RC=2
74              RETURN
75          ENDIF
76      ENDIF
77 C
78 C      try to convert the first token to an integer
79      CALL DTB(TEXT,NDIG, INT,RCD)
80      IF(RCD.NE.0) THEN
81          RC=3
82          RETURN
83      ENDIF
84      IF(NI.EQ.LI) THEN
85          RC=1
86          RETURN
87      ENDIF
88      NI=NI+1
89      INTS(NI)=INT
90 C
91 C      blank out the token just converted
92      DO 4 K=1,NDIG
93          TEXT(K)=' '
94      4 CONTINUE
95      GO TO 5
96      END

```

An INTEGER*4 can represent a number having up to 10 digits, or 11 characters if it begins with a sign (see §4.1). If it has more [65-76] the routine resigns with RC=2.

Then [78-83] DTB is used to convert the first NDIG characters of TEXT to an INTEGER*4 in INT. If that token cannot be interpreted as an integer [80-83] the routine resigns with RC=3 leaving the offending token at the beginning of TEXT. If the conversion is successful but [84-87] LI values have already been found, the routine resigns with RC=1 to signal that fact. Otherwise the value is [88] counted and [89] stored in INTS. Finally [91-94] the token is blanked out and [95] control returns to statement 5 so the process can repeat.

18.1.4 Decimal Vector to Real Vector

It is also sometimes convenient to read non-integer numbers as characters and convert them to REAL*8. PRSR8V, listed below and on the next page, resembles PRSI4V but rereads from a buffer rather than using DTB to perform the conversion.

```

1 C
2 Code by Michael Kupferschmid
3 C
4     SUBROUTINE PRSR8V(TEXT,LTXT,LR, RS,NR,RC)
5 C     This routine returns in RS the reals in TEXT, or zeros.
6 C     TEXT is destroyed in the process.
7 C
8 C     RC  meaning
9 C     --  -----
10 C     0   all went well
11 C     1   TEXT contains more than LR reals
12 C     3   TEXT contains tokens other than reals
13 C
14 C     variable  meaning
15 C     -----  -----
16 C     BUFFER    TOKEN as a long word
17 C     I         index on the values in RS
18 C     K         index on the characters of TEXT
19 C     L         length of what is left of TEXT
20 C     LENGTH    function returns index of last nonblank in a string
21 C     LTOK      length of the current token
22 C     LTXT      dimensioned size of TEXT
23 C     LR        number of reals expected
24 C     NR        the number of reals found
25 C     R         a value read from TOKEN
26 C     RS        the REAL*8 vector returned
27 C     RC        return code of this routine; see table above
28 C     SHIFTL    routine removes leading blanks from a string
29 C     TEXT      text line to be parsed
30 C     TOKEN     the next token to be read, as characters
31 C
32 C     formal parameters
33 CHARACTER*1 TEXT(LTXT)
34 REAL*8 RS(LR)
35 INTEGER*4 RC
36 C
37 C     prepare to read values from the text string
38 CHARACTER*1 TOKEN(24)
39 CHARACTER*24 BUFFER
40 EQUIVALENCE(TOKEN,BUFFER)
41 REAL*8 R

```

Here each substring of TEXT being processed gets [68-72] copied into TOKEN. That local variable is [38-40] EQUIVALENCED to BUFFER, so that puts the substring into BUFFER. Then [74-75,83-84] R is read from BUFFER. If the READ fails the ERR=5 clause transfers control to statement 5 where the routine sets RC=3 and resigns; if the READ is successful but LR reals have already been found [76-79] the routine resigns with RC=1. Otherwise [80-81] the value is [80] counted and [81] stored in RS.

```

42 C
43 C -----
44 C
45 C     assume that no reals will be found
46 C     RC=0
47 C     DO 1 I=1,LR
48 C         RS(I)=0.DO
49 C 1 CONTINUE
50 C     NR=0
51 C     IF(LTXT.LE.0) RETURN
52 C     L=LTXT
53 C
54 C     shift the string left to put the next token first
55 C 8 CALL SHIFTL(TEXT,L)
56 C
57 C     when the string is empty we are done
58 C     L=LENGTH(TEXT,L)
59 C     IF(L.EQ.0) RETURN
60 C
61 C     not done yet; find the length of the first token
62 C     LTOK=0
63 C     DO 2 K=1,L
64 C         IF(TEXT(K).EQ.' ') GO TO 3
65 C         LTOK=LTOK+1
66 C 2 CONTINUE
67 C
68 C     copy the token to the buffer
69 C 3 DO 4 K=1,LTOK
70 C         TOKEN(K)=TEXT(K)
71 C         IF(K.LE.LTOK) TOKEN(K)=TEXT(K)
72 C 4 CONTINUE
73 C
74 C     read the next value from the buffer
75 C     READ(BUFFER,*,ERR=5) R
76 C     IF(NR.EQ.LR) THEN
77 C         RC=1
78 C         RETURN
79 C     ENDIF
80 C     NR=NR+1
81 C     RS(NR)=R
82 C     GO TO 6
83 C 5 RC=3
84 C     RETURN
85 C
86 C     blank out the token just converted
87 C 6 DO 7 K=1,LTOK
88 C     TEXT(K)= ' '
89 C 7 CONTINUE
90 C     GO TO 8
91 C     END

```

18.2.2 Inserting One String Into Another

The **STRINS** routine inserts a string of characters into a fixed template, overwriting blanks that are already there. Sometimes it is desirable to instead wedge an insertion in between two characters of a string by opening up a space for it. The **STRWDG** routine listed on the next page does that, by using the **SHIFTL** routine of §10.3 and the **SHIFTR** routine listed below.

```

1 C
2 Code by Michael Kupferschmid
3 C
4     SUBROUTINE SHIFTR(String,L)
5 C     This routine right-justifies the nonblank characters in
6 C     String, moving blanks from after to before them.
7 C
8 C     variable    quantity
9 C     -----
10 C    K           index on the characters of String
11 C    KLNB        index of last nonblank in unjustified String
12 C    L           number of characters in String, including blanks
13 C    String      the character string to be right-justified
14 C
15     CHARACTER*1 String(L)
16 C
17 C -----
18 C
19 C     find the rightmost nonblank
20     IF(L.LT.2) RETURN
21     DO 1 K=1,L
22         KLNB=L-K+1
23         IF(String(KLNB).NE.' ') GO TO 2
24     1 CONTINUE
25 C     the string is all blanks
26     RETURN
27 C
28 C     is the last nonblank the last character?
29     2 IF(KLNB.EQ.L) RETURN
30 C
31 C     no; copy the string right to remove trailing blanks
32     DO 3 K=1,KLNB
33         String(L-K+1)=String(KLNB-K+1)
34     3 CONTINUE
35 C
36 C     blank out the front characters that were shifted
37     DO 4 K=1,L-KLNB
38         String(K)=' '
39     4 CONTINUE
40     RETURN
41     END
```

Just as **SHIFTL** removes leading blanks to shift a string left, **SHIFTR** shifts a string right within its dimensioned size. If the string is too short [20] or all blanks [25-26], or if it has no trailing blanks [28-29], the routine does nothing. Otherwise, it [21-24] finds the index **KLNB** of the last nonblank in the unshifted string, [31-34] copies the string right so that the last nonblank ends up in **String(L)**, and [36-39] blanks out the elements where the beginning of the string used to be.

```

1 C
2 Code by Michael Kupferschmid
3 C
4     SUBROUTINE STRWDG(LI,INSERT,LOC, LS,STRING, RC)
5 C     This routine wedges INSERT into STRING starting at LOC.
6 C
7 C     variable  meaning
8 C     -----  -----
9 C     INSERT    string to be inserted in STRING
10 C     K         index on the characters of INSERT
11 C     L         index of the last nonblank in STRING
12 C     LENGTH    function returns index of last nonblank in a string
13 C     LI        length of INSERT
14 C     LOC       where in STRING to begin wedged-in INSERT
15 C     LS        dimensioned size of STRING; then its length
16 C     RC        return code; 1 => insert will not fit
17 C     SHIFTL    routine removes leading blanks from a string
18 C     SHIFTR    routine removes trailing blanks from a string
19 C     STRING    the string being inserted into
20 C
21     CHARACTER*1 INSERT(LI),STRING(*)
22     INTEGER*4 RC
23 C
24 C -----
25 C
26 C     will the insertion fit?
27     L=LENGTH(STRING,LS)
28     IF(LI .GT. LS-L) THEN
29         RC=1
30         RETURN
31     ENDIF
32 C
33 C     shift the end of the string right as far as possible
34     CALL SHIFTR(STRING(LOC),LS-LOC+1)
35 C
36 C     copy the insert into the space just opened up
37     DO 1 K=1,LI
38         STRING(LOC+K-1)=INSERT(K)
39 1 CONTINUE
40 C
41 C     move the end of the string back to the end of the insert
42     CALL SHIFTL(STRING(LOC+LI),LS-(LOC+LI)+1)
43 C
44 C     return the index of the last nonblank in STRING
45     LS=LENGTH(STRING,LS)
46     RC=0
47     RETURN
48     END

```

STRWDG begins [27] by finding the index L of the last nonblank in STRING; then the number of blanks at the end of STRING is LS-L. If that is too few to fit the insert [28-31] the routine resigns with RC=1. Otherwise, it [33-34] shifts the characters from LOC on as far right as possible in STRING, [36-39] copies INSERT into STRING starting at element LOC, and [41-42] shifts the text that originally came after LOC back to meet the end of the insertion. Finally, [44-47] it updates LS to the index of the last nonblank in STRING. Because LS gets changed, STRING is dimensioned * [21].

18.4+ Extra-Precision Accumulation

A fundamental operation in numerical linear algebra, first mentioned in §6.3, is finding the **dot product** of two vectors \mathbf{x} and \mathbf{y} as the following sum.

$$\mathbf{x}^\top \mathbf{y} = \sum_{j=1}^n x_j y_j.$$

This calculation is likely to be imprecise because of rounding error in the multiplications and cancellation error when small terms are added into a large sum, as discussed in §4.3. I mentioned there that cancellation error can be reduced by adding up the terms in ascending order of absolute value, but that is seldom done in finding the dot product because precomputing and sorting the products $x_j y_j$ uses significant extra memory and CPU time. Nonetheless we often want a precise answer, so it is standard practice to instead accumulate the sum at **extra precision**. For example, if the basic calculation uses `REAL*4` numbers the dot product might be coded using `REAL*8` arithmetic like this.

```

REAL*4 X(100),Y(100),DOT
REAL*8 Z
:
Z=0.DO
DO 1 J=1,100
    Z=Z+DBLE(X(J))*DBLE(Y(J))
1 CONTINUE
DOT=SNGL(Z)
:
```

Here the `DBLE` function (see §4.4) is used to cast `X(J)` and `Y(J)` to `REAL*8` for the multiplication, and `SNGL` is used to convert the result `Z` back to `REAL*4`. If your compiler supports the `REAL*16` data type, you can modify this code to compute accurate `REAL*8` dot products. But what if your compiler does *not* recognize `REAL*16`, or it does but the basic calculation already uses `REAL*16` and you want more precision than that? There is in fact a clever way (see [202], [203 §4.4], and [9, §4.3.3]) to perform the dot product calculation at extra precision with variables of the same precision as those used to store the vectors, and with only a small penalty in memory and processor time.

Multiplying two n -bit binary fractions a and b yields a product ab that is $2n$ bits long, as in this example with $n = 4$.

$$\begin{array}{r}
 \bullet 1110 = a \\
 \times \bullet 1101 = b \\
 \hline
 1110 \\
 0000 \\
 1110 \\
 1110 \\
 \hline
 \bullet 10110110 = ab
 \end{array}$$

To store this result as a 4-bit binary fraction we must discard the least-significant 4 of its fraction bits, or *half* of the bits that make up the answer! These bits are of course much less important than the ones we keep, but neglecting them does introduce some error. The right answer is $.10110110_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} = 0.7109375_{10}$ but the result we keep is $.1011_2 = 0.6875_{10}$.

Instead suppose we **split** a into two parts so that $a = a_h + a_t$, where a_h is the value of the high or most-significant $n/2$ bit positions of a and a_t is the value of the trailing or least-significant $n/2$ bit positions. Then, if a_h and a_t are stored as floating-point binary fractions having n significand bits, the rightmost $n/2$ bits in each of them will be zero. Splitting b will yield parts b_h and b_t that similarly have zeros in their $n/2$ least-significant bit positions. Then we can find the product as

$$ab = (a_h + a_t)(b_h + b_t) = a_h b_h + a_h b_t + a_t b_h + a_t b_t$$

where each partial product is *exactly* represented by a floating-point binary fraction of n bits and can therefore be stored without any loss of precision. For our $n = 4$ example this is how the process works.

$$\begin{aligned} a &= .1110 = .1100 \times 2^0 + .1000 \times 2^{-2} = a_h + a_t \\ b &= .1101 = .1100 \times 2^0 + .0100 \times 2^{-2} = b_h + b_t \\ a_h b_h &= (.1100 \times 2^0) \times (.1100 \times 2^0) = .1001 \times 2^0 \\ a_h b_t &= (.1100 \times 2^0) \times (.0100 \times 2^{-2}) = .0011 \times 2^{-2} \\ a_t b_h &= (.1000 \times 2^{-2}) \times (.1100 \times 2^0) = .0110 \times 2^{-2} \\ a_t b_t &= (.1000 \times 2^{-2}) \times (.0100 \times 2^{-2}) = .0010 \times 2^{-4} \end{aligned}$$

Each of the parts has $n/2 = 2$ trailing zeros, and each partial product just fits in $n = 4$ bits. If we align binary points and add partial products we get the same answer as before.

$$\begin{array}{r} .10010000 = a_h b_h \\ .00001100 = a_h b_t \\ .00011000 = a_t b_h \\ .00000010 = a_t b_t \\ \hline .10110110 = ab \end{array}$$

To avoid losing the least-significant half of this result, we could accumulate the sum of the partial products into a two-element vector of 4-bit floating-point binary fractions, ending up with $ab = [(.1011 \times 2^0), (.0110 \times 2^{-4})]$. Once a whole dot product has been accumulated, the less-significant parts of all the partial products will have added up instead of being lost through cancellation, and we can obtain an accurate n -bit answer by adding the two n -bit vector elements that we used to store the $2n$ -bit sum.

The MPYACC subroutine listed on the next page uses the splitting idea to perform a single multiplication of the scalar X times the scalar Y , calling **ADDACC** to add each partial product to the two-element accumulator **XYSUM**. Unlike a and b in the discussion above, X and Y are [21] **REAL*8** variables. According to §4.2 they have a sign bit and 11 exponent bits preceding an implied “1.” and 52 bits of binary fraction, so in splitting them it is necessary to preserve the sign and exponent bits. To split X we begin [35] by copying it into XH , which is [25-26] overlaid by the two-element **INTEGER*4** vector IXH . On a little-endian processor the least-significant word of X comes first in memory (see §4.8) so another name for it is $IXH(1)$. This fullword we bitwise-AND (see §4.6.3) with **HMASK** [37] which is initialized [27] at compile time to the bit pattern 11111100000000000000000000000000. The resulting value of XH is thus X with its least-significant 26 ($= n/2$ in the discussion above) bits set to zero. We want XH and XT to add up to X , so [38] XT is just X minus the XH we found. The same process is used [39-42] to split Y into YH and YT . The parts XH , XT , YH , and YT , are **REAL*8** so they have 52 fraction bits, but of these the trailing 26 are zero. Finally the code [45-52] computes the four 52-bit partial products (in order from smallest to largest) and adds each to the extra-precision accumulator.

```

1      SUBROUTINE MPYACC(X,Y, XYSUM )
2 C      This routine accumulates XYSUM=XYSUM+X*Y at extra precision.
3 C
4 C      variable  meaning
5 C      -----  -----
6 C      ADDACC    routine adds to an extra-precision accumulator
7 C      HMASK     deletes the 26 least-significant fraction bits
8 C      IAND      Fortran function for bitwise logical AND
9 C      IXH       XH as 2 fullwords
10 C     IYH       YH as 2 fullwords
11 C     P         a partial product
12 C     X         first number in product
13 C     XH        split of X containing its high 26 fraction bits
14 C     XT        split of X containing value of trailing 26 bits
15 C     XYSUM     extra-precision accumulator
16 C     Y         second number in product
17 C     YH        split of Y containing its high 26 fraction bits
18 C     YT        split of Y containing value of trailing 26 bits
19 C
20 C     formal parameters
21     REAL*8 X,Y,XYSUM(2)
22 C
23 C     prepare to split X and Y
24     REAL*8 XH,XT,YH,YT
25     INTEGER*4 IXH(2),IYH(2)
26     EQUIVALENCE(XH,IXH),(YH,IYH)
27     INTEGER*4 HMASK/'Z'FC000000'/'
28 C
29 C     prepare to compute the partial products
30     REAL*8 P
31 C
32 C -----
33 C
34 C     split X and Y into parts having 26 trailing fraction bits zero
35     XH=X
36 C     this assumes the processor is little-endian
37     IXH(1)=IAND(IXH(1),HMASK)
38     XT=X-XH
39     YH=Y
40 C     this assumes the processor is little-endian
41     IYH(1)=IAND(IYH(1),HMASK)
42     YT=Y-YH
43 C
44 C     add the 52-fraction-bit exact partial products to accumulator
45     P=XT*YT
46     CALL ADDACC(P,XYSUM)
47     P=XT*YH
48     CALL ADDACC(P,XYSUM)
49     P=XH*YT
50     CALL ADDACC(P,XYSUM)
51     P=XH*YH
52     CALL ADDACC(P,XYSUM)
53     RETURN
54     END

```


The additions are accomplished by the `ADDACC` subroutine, which is listed on the next page. `ADDACC` begins [24-30] by putting the larger of `P` and `XYSUM(1)` in `U` and the smaller in `V`. This is to minimize cancellation error in the calculation [36] of $U-Z$ (if `U` is close to $Z=U+V$ then little or no shifting will be needed to align the binary points in finding $U-Z$). Then [33] we find $Z=U+V$. Here some of the less-significant fraction bits of `V` are probably lost because `V` must be shifted to align its binary point with that of `U`. How much error does that introduce? The difference $U-Z$ should be exactly $-V$, but because of cancellation it will differ from $-V$ by the error we seek. This is calculated [36] as `ZZ`. To that we add [39] the current contents of the least-significant doubleword of the accumulator. If the least-significant doubleword has grown big enough to be noticed if we added it to the most-significant doubleword, we want to move that much of it there. So the most-significant doubleword of the accumulator then becomes [42] the imprecise sum plus the correction to the sum plus the least significant doubleword of the accumulator. Finally [45] we replace the least-significant doubleword of the accumulator with the (small) amount that is necessary to make $XYSUM(1)+XYSUM(2)$ equal to the corrected sum $Z+ZZ$. The complicated process just described has the effect of adding `P` to `XYSUM` at $2 \times 52 = 104$ bits of precision, which is almost the 112 bits of precision we would get if we were able to use `REAL*16` arithmetic.

```

1      SUBROUTINE ADDACC(P, XYSUM )
2 C      This routine adds P to the extra-precision accumulator XYSUM.
3 C      It must be compiled with optimization turned off.
4 C
5 C      variable  meaning
6 C      -----
7 C      DABS      Fortran function returns |REAL*8|
8 C      P          quantity to be added to the accumulator
9 C      U          the larger in absolute value of P and XYSUM
10 C     V          the smaller in absolute value of P and XYSUM
11 C     XYSUM      the accumulator
12 C     Z          most significant part of sum
13 C     ZZ         least significant part of sum
14 C
15 C     formal parameters
16 C     REAL*8 P,XYSUM(2)
17 C
18 C     local variables
19 C     REAL*8 U,V,Z,ZZ
20 C
21 C -----
22 C
23 C     put the larger quantity in U and the smaller in V
24 C     IF(DABS(XYSUM(1)) .LT. DABS(P)) THEN
25 C         U=P
26 C         V=XYSUM(1)
27 C     ELSE
28 C         U=XYSUM(1)
29 C         V=P
30 C     ENDIF
31 C
32 C     find the sum, imprecisely
33 C     Z=U+V
34 C
35 C     compute the error that was made by rounding U+V to REAL*8
36 C     ZZ=(U-Z)+V
37 C
38 C     add to it the least significant part of the accumulator
39 C     ZZ=ZZ+XYSUM(2)
40 C
41 C     that might be enough to increase the most significant part
42 C     XYSUM(1)=Z+ZZ
43 C
44 C     make the least significant part of accumulator what is left
45 C     XYSUM(2)=(Z-XYSUM(1))+ZZ
46 C
47 C     RETURN
48 C     END

```

The DDOTQ function listed on the next page uses MPYACC to compute a dot product using extra-precision accumulation. After doing some sanity-checking [23-24] it initializes the accumulator XYSUM to zeros [27-28]. Instead of the multiply-and-add loop we had before we now have [29-31] a loop of calls to MPYACC. On each invocation that routine computes $X(J) \times Y(J)$ and adds it to the accumulator as described above. When the loop is finished we find the dot product [34] by adding together the most- and least-significant doublewords of the accumulator.

The program below compares DDOTQ to DDOT for finding a troublesome dot product.

```

      REAL*8 X(101),Y(101),DDOT,ANS,DDOTQ,ANSQ
      X(1)=1.D+08
      Y(1)=1.D+08
      DO 1 J=2,101
          X(J)=DFLOAT(J-1)
          Y(J)=1.D0/DFLOAT(J-1)
1  CONTINUE
      ANS=DDOT(X,Y,101)
      ANSQ=DDOTQ(X,Y,101)
      WRITE(6,901) ANS,ANSQ
901  FORMAT('DDOT finds ',1PD23.16/
;      'DDOTQ finds ',1PD23.16)
      STOP
      END

```

The program manufactures the following problem.

$$\begin{aligned}
 \mathbf{x} &= [10^8, 1, 2, 3, \dots, 100] \\
 \mathbf{y} &= [10^8, 1, \frac{1}{2}, \frac{1}{3}, \dots, \frac{1}{100}] \\
 \mathbf{x}^\top \mathbf{y} &= 10^{16} + (1 \times 1) + (2 \times \frac{1}{2}) + (3 \times \frac{1}{3}) + \dots + (100 \times \frac{1}{100}) = 100000000000000100
 \end{aligned}$$

When the program is compiled with `gfortran` and run, it produces the following output. The product of the first two terms, 10^{16} , is big enough so that the subsequent terms contribute nothing to the sum when DDOT does the calculation using `REAL*8` arithmetic. However, when DDOTQ does the calculation using extra-precision accumulation the correct result is obtained.

```

unix[1] a.out
DDOT finds  1.0000000000000000D+16
DDOTQ finds 1.0000000000000100D+16
unix[2]

```

This chapter has introduced two-part values, which can be used to perform fixed-point arithmetic with numbers too big to store in an `INTEGER*4`, and extra-precision accumulation for computing floating-point dot products more precisely than we can by simply doing `REAL*8` arithmetic. It is also possible to use Classical FORTRAN for integer calculations of *arbitrary* precision, as described for example in [12, §20.6], and for floating-point calculations of *arbitrary* precision by invoking Brent's multiple precision package (see <https://maths-people.anu.edu.au/~brent/pub/pub043.html>).

```

1      FUNCTION DDOTQ(X,Y,N)
2 C      This routine computes the dot product of X with Y,
3 C      using extra-precision accumulation.
4 C
5 C      variable  meaning
6 C      -----  -----
7 C      J          index on the elements of X and Y
8 C      MPYACC      routine does extra-precision multiply and accumulate
9 C      N          number of elements in X and Y
10 C      X          one of the vectors in the dot product
11 C      XYSUM       extra-precision result
12 C      Y          the other vector in the dot product
13 C
14 C      formal parameters
15      REAL*8 DDOTQ,X(N),Y(N)
16 C
17 C      local variable
18      REAL*8 XYSUM(2)
19 C
20 C -----
21 C
22 C      check for a sensible value of N
23      DDOTQ=0.DO
24      IF(N.LE.0) RETURN
25 C
26 C      accumulate the product at extended precision
27      XYSUM(1)=0.DO
28      XYSUM(2)=0.DO
29      DO 1 J=1,N
30          CALL MPYACC(X(J),Y(J), XYSUM )
31 1 CONTINUE
32 C
33 C      return a double-precision answer
34      DDOTQ=XYSUM(1)+XYSUM(2)
35      RETURN
36      END

```

[202] Stokes, H. H., “The sensitivity of econometric results to alternative implementations of least squares,” *Journal of Economic and Social Measurement* 30 (2005) 9-38. In the source code of Stokes’ B34S program, this approach to implementing the extra precision accumulation idea is attributed to “1980 IMSL code that is no longer supported.”

[203] Muller, Jean-Michel; Brisebarre, Nicolas; de Dinechin, Florent; Jeannerod, Claude-Pierre; Lefèvre, Vincent; Melquiond, Guillaume; Revol, Nathalie; Stehlé, Damien; and Torres, Serge, *Handbook of Floating-Point Arithmetic*, Birkhäuser, 2010.

18.5+ Generating Pseudorandom Numbers

Some scientific and engineering calculations are best performed by means of **simulation** [53, §13] in which random trials are used to approximate the solution of a deterministic problem or the random behavior of a real system is modeled by a computer program. Either sort of simulation requires a sequence of numbers that appear to be random. Among the many algorithms that have been proposed for generating such a **pseudorandom sequence**, the **mixed congruential algorithm**

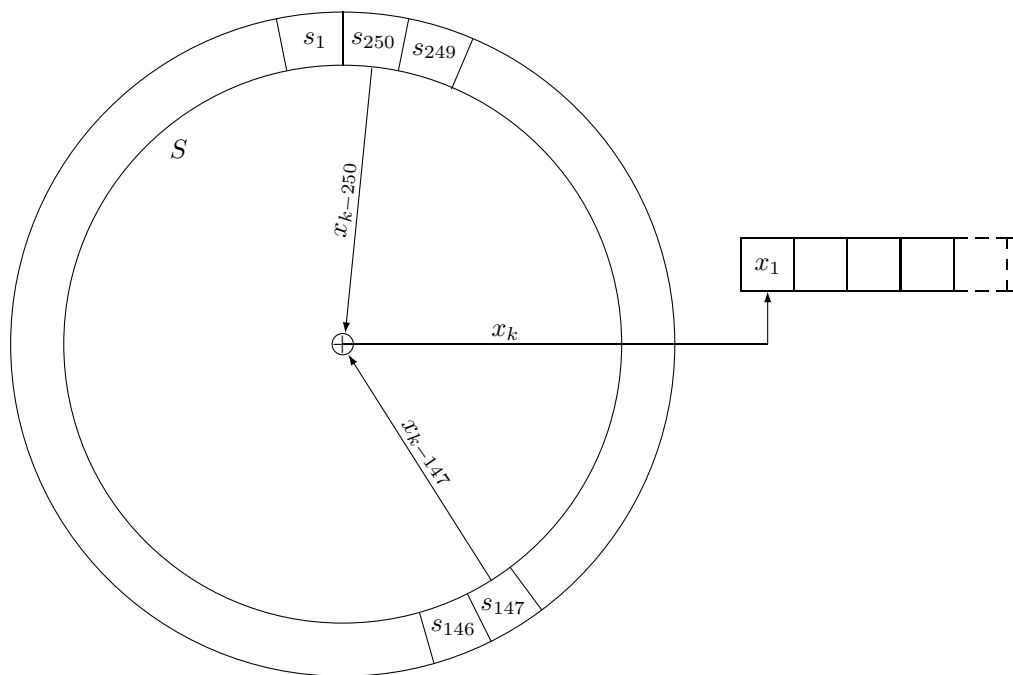
$$x_k = (ax_{k-1} + b) \bmod 2^{31}$$

[9, §3] is the most widely used. It is simple enough to provide a nice example of FORTRAN programming and it is interesting because of the trivial way in which its modulus operation can be performed, so it makes an appearance in Exercises 4.10.48, 6.8.20, and 8.8.20 of this book. You might wish to reread the first of those now.

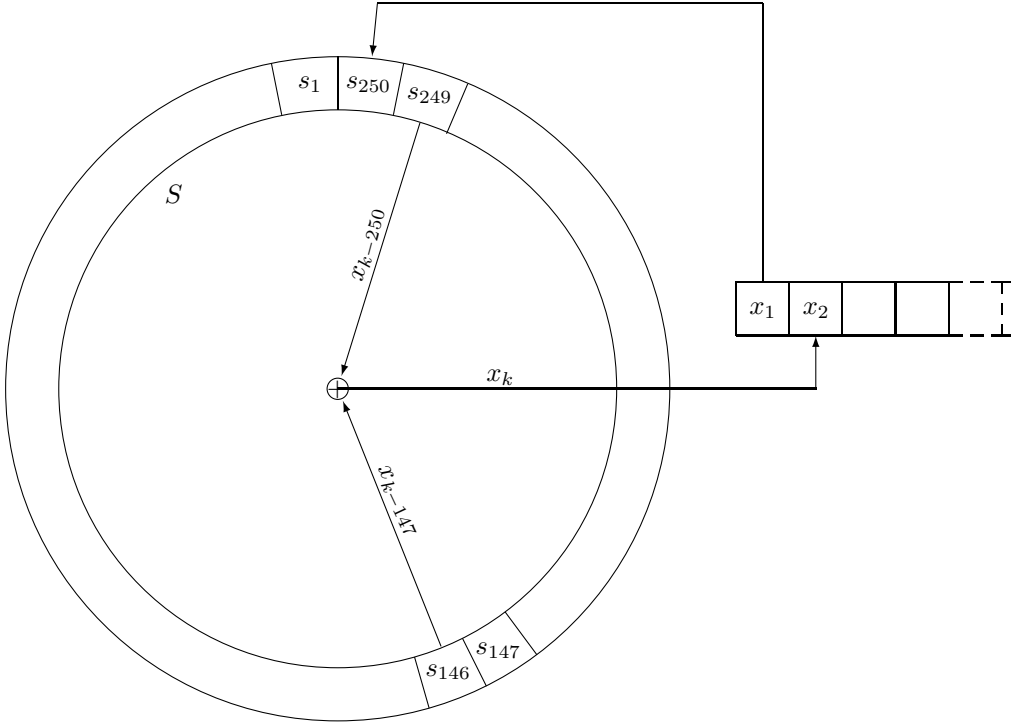
Depending on the numbers chosen for a and b the mixed congruential algorithm can generate up to 2^{31} numbers before the sequence begins to repeat, but some simulations use more. The **generalized shift register algorithm**

$$x_k = x_{k-147} \oplus x_{k-250}$$

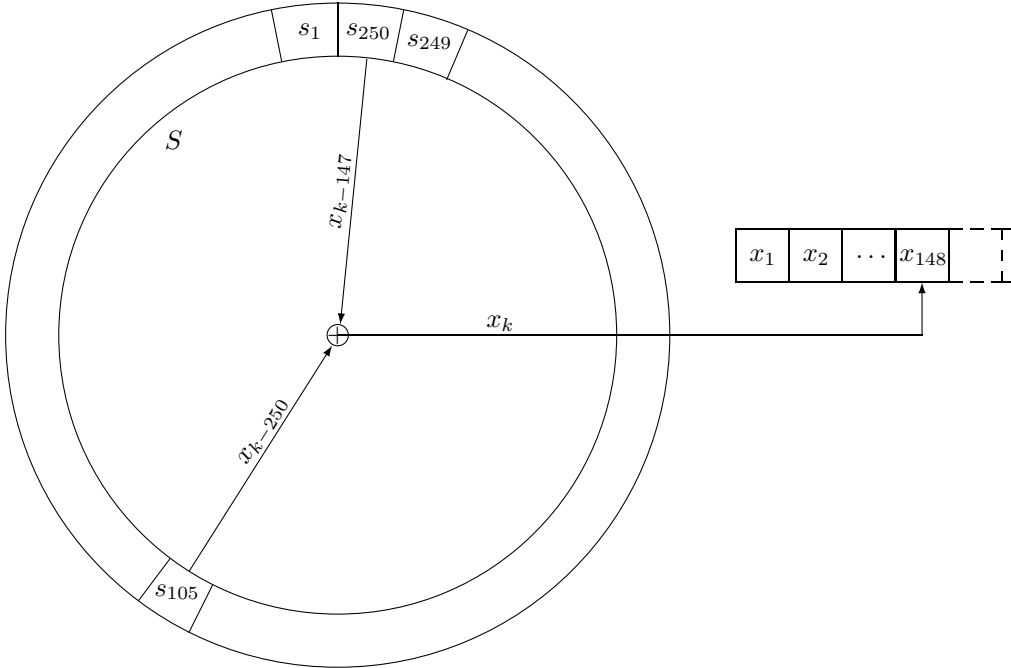
[53, §13.3] [204], in which \oplus denotes the bitwise XOR operation (see §4.6.3), generates $2^{250} - 1$ values before it repeats. It requires that we remember the previous 250 numbers that have been generated, which makes it also interesting but not so simple as the mixed congruential algorithm. To see how this algorithm can be implemented, imagine that we have already filled the circular shift register S pictured below with 250 randomly-generated **seeds** $s_1 \dots s_{250}$, five of which are shown. If we let $x_{k-147} = s_{148-k}$ and $x_{k-250} = s_{251-k}$ then we can compute $x_1 = s_{147} \oplus s_{250}$ as shown.



Then we can remember x_1 by using it to replace s_{250} , and compute $x_2 = s_{146} \oplus s_{249}$ as illustrated below.



The vectors pointing from x_{k-250} and x_{k-147} to the exclusive-or symbol at the center of the circle rotate clockwise together as successive x_k are produced, through $k = 147$ when $148 - k = 1$, $251 - k = 104$, and $x_{147} = s_1 \oplus s_{104}$. To find x_{148} the vectors rotate one more element clockwise as shown below, so $x_{148} = s_0 \oplus s_{105} = s_{250} \oplus s_{105}$.



Here is a pseudocode description of the process pictured above, in which I have assumed that n numbers are to be generated.

```

initialize pointers
km250=251
km147=148

do k=1,n
  rotate the vectors one element clockwise
  km250=km250-1
  if(km250=0) km250=250
  km147=km147-1
  if(km147=0) km147=250

  compute the result
  x(k)=s(km250)  $\oplus$  s(km147)

  and save it in the shift register for use later
  s(km250)=x(k)
enddo

```

The variable `km250` represents $k - 250$ and `km147` represents $k - 147$. The first pass of the loop decrements these pointers to have the values 250 and 147 respectively and computes $x_1 = s_{250} \oplus s_{147}$, the second pass of the loop makes `km250=249` and `km147=146` and computes $x_2 = s_{249} \oplus s_{146}$, and so on. Each time a seed $s(\text{km250})$ is used to compute a new x_k it is replaced by that new value. Eventually decrementing `km147` gives it a value of zero, but as shown in the third diagram above the element of S that should enter the calculation is not s_0 but s_{250} (in which we stored the previous value we calculated for x_1) so `km147` is reset to 250. When the vector pointing from x_{k-250} rotates past s_1 to s_0 the variable `km250` is similarly reset from 0 to 250. You should convince yourself that the loop in this pseudocode faithfully describes the rotation of the vectors in the pictures shown above and thus the calculation of the successive x_k .

To be useful in a simulation the pseudorandom values we generate must be floating-point numbers uniformly distributed on the closed interval from 0 to 1. Sequences generated using the generalized shift register algorithm have been shown [204, pp519-523] to be statistically indistinguishable from those drawn from a uniform distribution, but making our FORTRAN implementation of the algorithm produce `REAL*8` numbers in the right range entails several complications.

First, the built-in FORTRAN function `IEOR` discussed in §4.6.3 operates on bitstrings that are stored in `INTEGER*4` variables. To generate one `REAL*8` number we must XOR two *pairs* of `INTEGER*4` numbers and store the two-word result in the same memory occupied by the `REAL*8` we want.

Second, the magnitude of the `REAL*8` is determined by its most significant 12 bits, which represent its sign and exponent. Recall from §4.2 that the value of such a number is

$$r = (-1)^s \times 2^{p-1023} \times (1 + f)$$

where s is the sign bit, p is the exponent represented by the next 11 bits, and f is the binary fraction. The bits of f should be randomly generated, but for r not to exceed 1 we must set $s = 0$ and $p = 1023$ in each number we generate and subtract 1 from the result.

Third, because we will be manipulating the bits of the floating-point number representation, we must pay attention to the order in which the bytes are stored. Recall from §4.8 that on a little-endian processor it is the *most*-significant byte of a number that is stored at the *lowest* address.

The subroutine DR250 listed on the next two pages takes account of all these considerations. The internal shift register ISR is initialized at compile time with seeds [49-298] that are suitable for many applications.

If the routine is invoked with $N > 0$ the contents of the shift register are used [305-325] to generate N random numbers in X . The DO 2 loop is a practical version of the idealized pseudocode given above. Each iteration of the loop generates [321-322] two INTEGER*4 bitstrings IX(1) and IX(2), which [45] occupy the same memory as the REAL*8 quantity XK and are thus the two halves of that doubleword; subtracting 1.D0 from it [323] yields the normalized result X(K). This code is intended for a little-endian processor (such as those in the Intel Pentium family [320]) so IX(1) and IX(2) are respectively JX(2) and JX(1); to revise it for a big-endian processor it is necessary only to make these assignment statements *not* reverse the order of the words. The fullwords JX(1) and JX(2) are made [318-319] from the adjacent shift register elements ISR(2*KM250-1) and ISR(2*KM250), which together make up the doubleword that we called $s(km250)$ in the pseudocode. JX(2) is just the second half of the doubleword [319] but the first half of the doubleword needs to begin with the bit pattern for a REAL*8 number that is in the interval $[0, 1]$; this is achieved [318] by OR-ing the fullword with the bit pattern in MASKC [42]. The updating of $s(km250)$ indicated in the pseudocode is accomplished here by (very fast) IEOR operations [314-315] on the ISR elements. When DR250 is used in this way it returns with N unchanged.

If the routine is invoked with $N \leq 0$ the shift register is reloaded [339-354] with values obtained using the particular mixed congruential algorithm described in [5, §10.1], and X is left unchanged. Each iteration of the DO 4 loop performs the calculations of the mixed congruential algorithm for the first [342-344] and then the second [347-349] fullword of that element in the shift register. Each result is AND-ed with MASKZ to zero its high-order bit, making it a positive integer; the high-order word is first AND-ed with MASKX to set its characteristic to zero so that the REAL*8 value of which it is a part will be in the range $[0, 1]$. After DR250 is invoked in this way to reload the internal shift register, it can then be called a second time with $N > 0$ to generate a random vector X (see its **man** page).

If $N < 0$ on input the starting value used for the reloading process is [329] $|N|$ and on return N contains [351] the negative of the final value resulting from the reloading process.

If $N = 0$ on input, the starting value for the reloading process is obtained [331-336] from the time-of-day clock and on return N contains the negative of that value. First [331] GETIMEOFDAY (see §18.5.3) obtains the two-part value TOD containing the seconds and microseconds elapsed since midnight. Each of these is used as the starting value for one iteration [332-333] of the mixed congruential algorithm. Then N is found [334] as their exclusive-or. This sequence of operations reliably produces a large integer having an irregular bit pattern; to ensure that it is negative and nonzero, the result is OR-ed [335] with NODD [36].

The long repetition period and high execution speed of DR250 make it suitable for many large simulations.


```

1      SUBROUTINE DR250(N,X)
2 C      This routine generates a vector X of N normalized
3 C      double-precision pseudorandom numbers in the interval [0,1]
4 C
5 C      variable  meaning
6 C      -----  -----
7 C      A          multiplier for mixed congruential algorithm
8 C      C          increment for mixed congruential algorithm
9 C      GETIMEOFDAY  unix routine returns time of day and time zone
10 C      IABS       Fortran function gives |INTEGER*4|
11 C      IAND       Fortran function gives bitwise AND of fullwords
12 C      IEOR       Fortran function gives bitwise XOR of fullwords
13 C      IOR        Fortran function gives bitwise OR of fullwords
14 C      ISR        internal shift register
15 C      IX         a doubleword of X as 2 singlewords
16 C      JX         IX with the words switched
17 C      K          index on random numbers generated
18 C      KM147      index in ISR of doubleword K-147
19 C      KM250      index in ISR of doubleword K-250
20 C      MASKC      to set characteristic of ISR values generated
21 C      MASKX      to zero characteristic of ISR values generated
22 C      MASKZ      to zero the high-order bit of a word
23 C      N          number of random values needed, or seed (see above)
24 C      NODD       to make SEED end-bits ones
25 C      TOD        time-of-day [seconds,microseconds]
26 C      SEED       seed used for mixed congruential algorithm
27 C      SET        T => N has been set to -SEED for return
28 C      X          vector of random values returned
29 C      XK         a doubleword of X
30 C      ZONE       unused; for GETIMEOFDAY
31 C
32 C      formal parameter
33 C      REAL*8 X(N)
34 C
35 C      prepare to seed the generator
36 C      INTEGER*4 SEED,TOD(2),ZONE(2),NODD/Z'80000001'/
37 C      INTEGER*4 A/843314861/,C/453816693/
38 C      INTEGER*4 MASKZ/Z'7FFFFFFF'/,MASKX/Z'00FFFFFF'/
39 C      LOGICAL*4 SET
40 C
41 C      prepare to run the generator
42 C      INTEGER*4 KM250/251/,KM147/148/,MASKC/Z'3FF00000'/
43 C      REAL*8 XK
44 C      INTEGER*4 IX(2),JX(2)
45 C      EQUIVALENCE(XK,IX(1))
46 C
47 C      internal shift register with values from SEED=123457
48 C      INTEGER*4 ISR(500)/
49 C      ;          Z'0007E8AF',Z'D4C00D62',
                    248 lines of data
298 C      ;          Z'0003731D',Z'8AD80548'/
299 C
300 C -----
301 C

```

```

302 C      is this an initialization call?
303      IF(N.LE.0) GO TO 1
304 C
305 C      generate N new random numbers while reseeding the generator
306      DO 2 K=1,N
307 C          find the indices in ISR of X(K-250) and X(K-147)
308          KM250=KM250-1
309          IF(KM250.LE.0) KM250=250
310          KM147=KM147-1
311          IF(KM147.LE.0) KM147=250
312 C
313 C          exclusive-or the singlewords
314          ISR(2*KM250)=IEOR(ISR(2*KM250),ISR(2*KM147))
315          ISR(2*KM250-1)=IEOR(ISR(2*KM250-1),ISR(2*KM147-1))
316 C
317 C          extract the resulting doubleword
318          JX(1)=IOR(ISR(2*KM250-1),MASKC)
319          JX(2)=ISR(2*KM250)
320 C          the word order is reversed in the Pentium
321          IX(1)=JX(2)
322          IX(2)=JX(1)
323          X(K)=XK-1.DO
324      2 CONTINUE
325      RETURN
326 C
327 C      get a seed to use in the reloading process
328      1 SET=.FALSE.
329      SEED=IABS(N)
330      IF(SEED.GT.0) GO TO 3
331      CALL GETIMEOFDAY(TOD,ZONE)
332      TOD(1)=A*TOD(1)+C
333      TOD(2)=A*TOD(2)+C
334      N=IEOR(TOD(1),TOD(2))
335      N=IOR(N,NODD)
336      SEED=-N
337      SET=.TRUE.
338 C
339 C      reload the ISR using a mixed congruential algorithm
340      3 DO 4 K=1,250
341 C          low-order word of the 8-byte value
342          SEED=A*SEED+C
343          ISR(2*K)=SEED
344          SEED=IAND(SEED,MASKZ)
345 C
346 C          high-order word of the 8-byte value
347          SEED=A*SEED+C
348          ISR(2*K-1)=IAND(SEED,MASKX)
349          SEED=IAND(SEED,MASKZ)
350      4 CONTINUE
351      IF(.NOT.SET) N=-SEED
352      KM250=251
353      KM147=148
354      RETURN
355      END

```