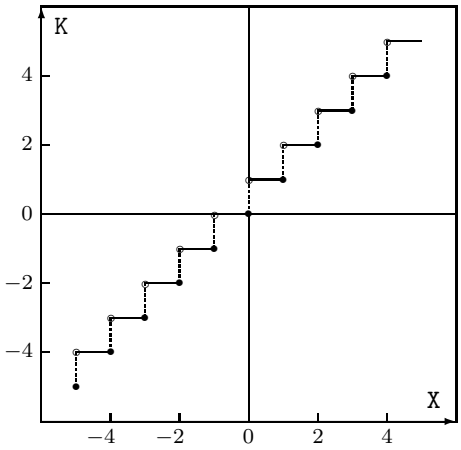
Additional Content for Use with *Classical Fortran*

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
9.6+	Undoing End-of-File	appending to a file; reattaching the keyboard
10.7+	Using `gnuplot` from FORTRAN	computing and graphing a table of coordinates
12.4.5	Typography	ordering subprogram parameters
18.4+	Extra-Precision Accumulation	computing $\mathbf{x}^\top\mathbf{y}$ more precisely
18.5+	Generating Pseudorandom Numbers	a generalized shift register algorithm

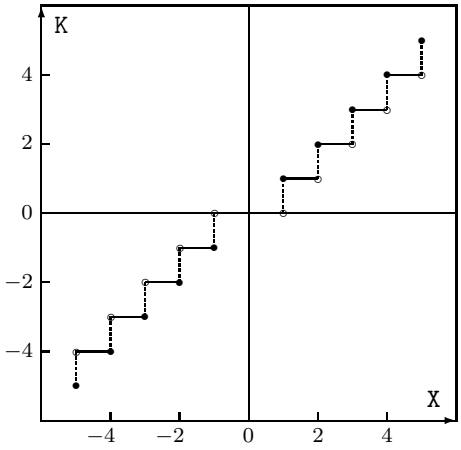
6.6.1+ Real-to-Integer Functions

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.

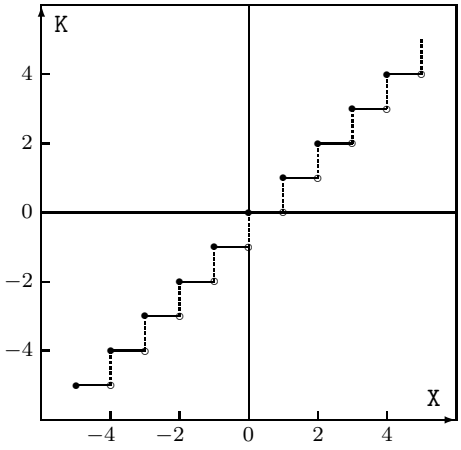


`K = ICEIL(X)` = lowest integer not less than `X`

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

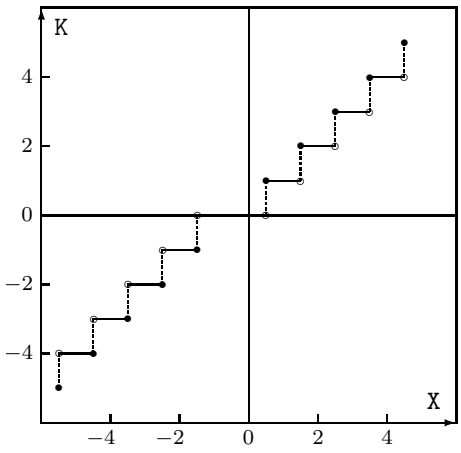


`K = IFIX(SNGL(X))` = truncate fractional part

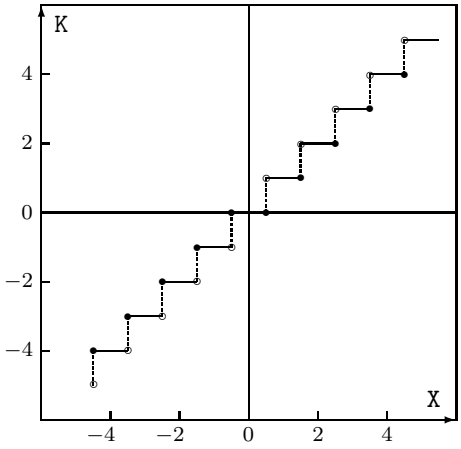


`K = IFLOR(X)` = highest integer not more than `X`

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

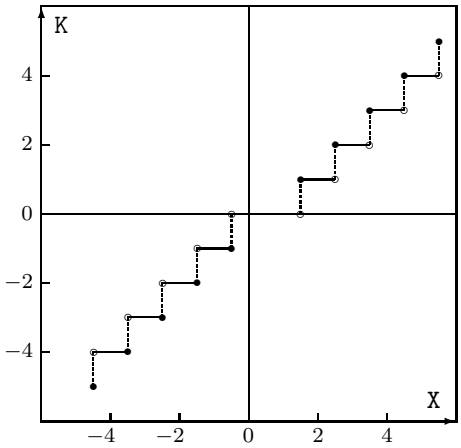


`K = IFIX(SNGL(X)+0.5)` = round positive values



`K = IROUND(X)` = integer closest to `X`

```
FUNCTION IROUND(X)
REAL*8 X
IF(R.GT.0.D0) THEN
   IROUND=IFIX(SNGL(R)+0.5)
ELSE
   IROUND=IFIX(SNGL(R)-0.5)
ENDIF
RETURN
END
```



`K = IFIX(SNGL(X)-0.5)` = round negative values

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.

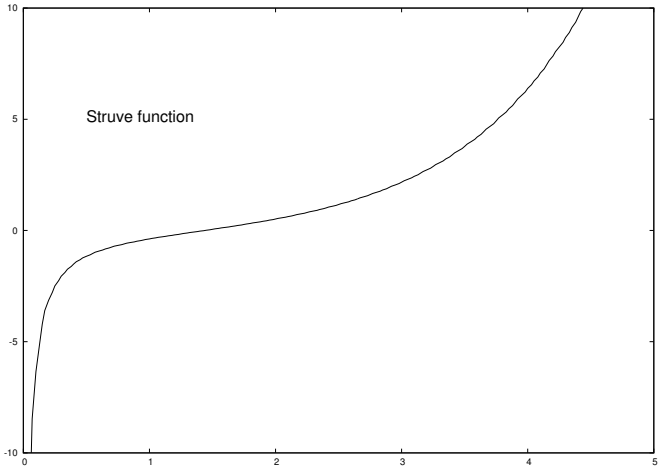
The Struve functions (see `https://en.wikipedia.org/wiki/Struve_function`), which are solutions to certain differential equations, are unknown to `gnuplot`. The program below begins by `6-16` computing points on the graph of the Struve function $\mathbf{L}_{-2}(x)$ and writing them to the file `struve.data`. 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.D0*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.

```
0.03   -25.4595
0.05   -12.7218
0.07    -8.4723
0.10    -6.3449
0.13    -5.0663
 :        :
```




```
set terminal postscript eps
set output "struve.eps"
set size 1.4,1.4
set xrange [ 0.00E+00 to  5.00E+00]
set yrange [-1.00E+01 to  1.00E+01]
set key off
set label "Struve function" font "Arial,24" at 5.00E-01, 5.00E+00
plot "struve.data" with lines
```

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 ≤ 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.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE WRTGNU(NAME,LN,XMIN,XMAX,YMIN,YMAX,TITLE,LT,XL,YL)
C     This routine constructs a file of commands that can be used
C     to make gnuplot generate a graph of tabular data.
C
C     variable  meaning
C     --------  -------
C     K         index on the characters of a string
C     L         index of last nonblank in a character string
C     LABEL     LBL with TITLE and its coordinates filled in
C     LBL       'set label ? at ?,?'
C     LENGTH    function returns index of last nonblank in a string
C     LN        number of characters in NAME
C     LT        number of characters in TITLE
C     NAME      identifier to use in filenames
C     OUT       'set output ?.eps'
C     OUTPUT    OUT with NAME filled in
C     PLOT      PLT with NAME filled in
C     PLT       'plot "?.data" with lines'
C     SPACEL    routine removes leading and extra blanks
C     STRINS    routine inserts one character string into another
C     TITLE     title for the graph
C     X         XL as characters
C     XL        x-coordinate of graph label
C     XMAX      high end of range of x-values to show
C     XMIN      low end of range of x values to show
C     Y         YL as characters
C     YL        y-coordinate of graph label
C     YMAX      high end of range of y-values to show
C     YMIN      low end of range of y-valeus to show
C
C     formal parameters
      CHARACTER*1 NAME(LN),TITLE(LT)
C
C     character-string templates
      CHARACTER*1 OUT(41)/'s','e','t',' ','o','u','t','p','u','t',
     ;                    ' ','"', 24*' ',    '.','e','p','s','"'/
      CHARACTER*1 OUTPUT(41)
      CHARACTER*1 LBL(75)/'s','e','t',' ','l','a','b','e','l',' ',
     ;                    '"', 24*' ','"',' ','f','o','n','t',' ',
     ;                    '"','A','r','i','a','l',',','2','4','"',
     ;                    ' ','a','t',' ',  9*' ', ',' ,9*' '/
      CHARACTER*1 LABEL(75)
      CHARACTER*1 PLT(47)/'p','l','o','t',' ','"', 24*' ',
     ;                    '.','d','a','t','a','"',' ',
     ;                    'w','i','t','h',' ','l','i','n','e','s'/
      CHARACTER*1 PLOT(47)
      CHARACTER*9 X,Y
C
C ---------------------------------------------------------------------
```

```
C
C     put the graph in a .eps file
      WRITE(9,901)
  901 FORMAT('set terminal postscript eps')
C
C     specify the name of the output .eps file
      CALL STRINS(OUT,41,NAME,LN,13,2, OUTPUT,L)
      WRITE(9,902) (OUTPUT(K),K=1,L)
  902 FORMAT(41A1)
C
C     specify the size of the graph
      WRITE(9,903)
  903 FORMAT('set size 1.4,1.4')
C
C     specify the ranges of x and y values to show
      WRITE(9,904) XMIN,XMAX
  904 FORMAT('set xrange [',1PE9.2,' to ',1PE9.2,']')
      WRITE(9,905) YMIN,YMAX
  905 FORMAT('set yrange [',1PE9.2,' to ',1PE9.2,']')
C
C     turn off the default legend
      WRITE(9,906)
  906 FORMAT('set key off')
C
C     specify the graph label and where it should appear
      IF(LT.GT.0) THEN
         WRITE(X,900) XL
         WRITE(Y,900) YL
  900    FORMAT(1PE9.2)
         CALL STRINS(LBL,75,X,9,57,0, LABEL,L)
         CALL STRINS(LABEL,75,Y,9,67,0, LABEL,L)
         CALL STRINS(LABEL,75,TITLE,LT,12,2, LABEL,L)
         CALL SPACEL(LABEL,L)
         L=LENGTH(LABEL,L)
         WRITE(9,907) (LABEL(K),K=1,L)
  907    FORMAT(75A1)
      ENDIF
C
C     specify the name of the input .data file
      CALL STRINS(PLT,47,NAME,LN,7,2, PLOT,L)
      WRITE(9,908) (PLOT(K),K=1,L)
  908 FORMAT(47A1)
      RETURN
      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
```

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.

18.4+ 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.D0
      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 [201], [202 §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 $n = 4$.

$$\begin{array}{rl}
 & \bullet\,1\,1\,1\,0 = a \\
\times & \bullet\,1\,1\,0\,1 = b \\
\hline
 & \quad 1\,1\,1\,0 \\
 & \quad 0\,0\,0\,0 \\
 & \quad 1\,1\,1\,0 \\
 & 1\,1\,1\,0 \\
\hline
 & \bullet\,1\,0\,1\,1\,0\,1\,1\,0 = ab
\end{array}$$

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.

$$\begin{aligned}
&\bullet 10010000 = a_h b_h \\
&\bullet 00001100 = a_h b_t \\
&\bullet 00011000 = a_t b_h \\
&\underline{\bullet 00000010 = a_t b_t} \\
&\bullet 10110110 = ab
\end{aligned}$$

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.

```
      SUBROUTINE ADDACC(P, XYSUM )
C     This routine adds P to the extra-precision accumulator XYSUM.
C     It must be compiled with optimization turned off.
C
C     variable  meaning
C     --------  -------
C     DABS      Fortran function returns |REAL*8|
C     P         quantity to be added to the accumulator
C     U         the larger in absolute value of P and XYSUM
C     V         the smaller in absolute value of P and XYSUM
C     XYSUM     the accumulator
C     Z         most significant part of sum
C     ZZ        least significant part of sum
C
C     formal parameters
      REAL*8 P,XYSUM(2)
C
C     local variables
      REAL*8 U,V,Z,ZZ
C
C ----------------------------------------------------------------
C
C     put the larger quantity in U and the smaller in V
      IF(DABS(XYSUM(1)) .LT. DABS(P)) THEN
         U=P
         V=XYSUM(1)
      ELSE
         U=XYSUM(1)
         V=P
      ENDIF
C
C     find the sum, imprecisely
      Z=U+V
C
C     compute the error that was made by rounding U+V to REAL*8
      ZZ=(U-Z)+V
C
C     add to it the least significant part of the accumulator
      ZZ=ZZ+XYSUM(2)
C
C     that might be enough to increase the most significant part
      XYSUM(1)=Z+ZZ
C
C     make the least significant part of accumulator what is left
      XYSUM(2)=(Z-XYSUM(1))+ZZ
C
      RETURN
      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) × 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.

$$\begin{aligned}\mathbf{x} &= [10^8, 1, 2, 3, \ldots, 100]\\ \mathbf{y} &= [10^8, 1, \tfrac{1}{2}, \tfrac{1}{3}, \ldots, \tfrac{1}{100}]\\ \mathbf{x}^\top\mathbf{y} &= 10^{16} + (1\times 1) + (2\times\tfrac{1}{2}) + (3\times\tfrac{1}{3}) + \cdots + (100\times\tfrac{1}{100}) = 10000000000000100\end{aligned}$$

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).

```
      FUNCTION DDOTQ(X,Y,N)
C     This routine computes the dot product of X with Y,
C     using extra-precision accumulation.
C
C     variable  meaning
C     --------  -------
C     J         index on the elements of X and Y
C     MPYACC    routine does extra-precision multiply and accumulate
C     N         number of elements in X and Y
C     X         one of the vectors in the dot product
C     XYSUM     extra-precision result
C     Y         the other vector in the dot product
C
C     formal parameters
      REAL*8 DDOTQ,X(N),Y(N)
C
C     local variable
      REAL*8 XYSUM(2)
C
C ---------------------------------------------------------------
C
C     check for a sensible value of N
      DDOTQ=0.D0
      IF(N.LE.0) RETURN
C
C     accumulate the product at extended precision
      XYSUM(1)=0.D0
      XYSUM(2)=0.D0
      DO 1 J=1,N
           CALL MPYACC(X(J),Y(J), XYSUM )
    1 CONTINUE
C
C     return a double-precision answer
      DDOTQ=XYSUM(1)+XYSUM(2)
      RETURN
      END
```

[201] **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."*

[202] **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**

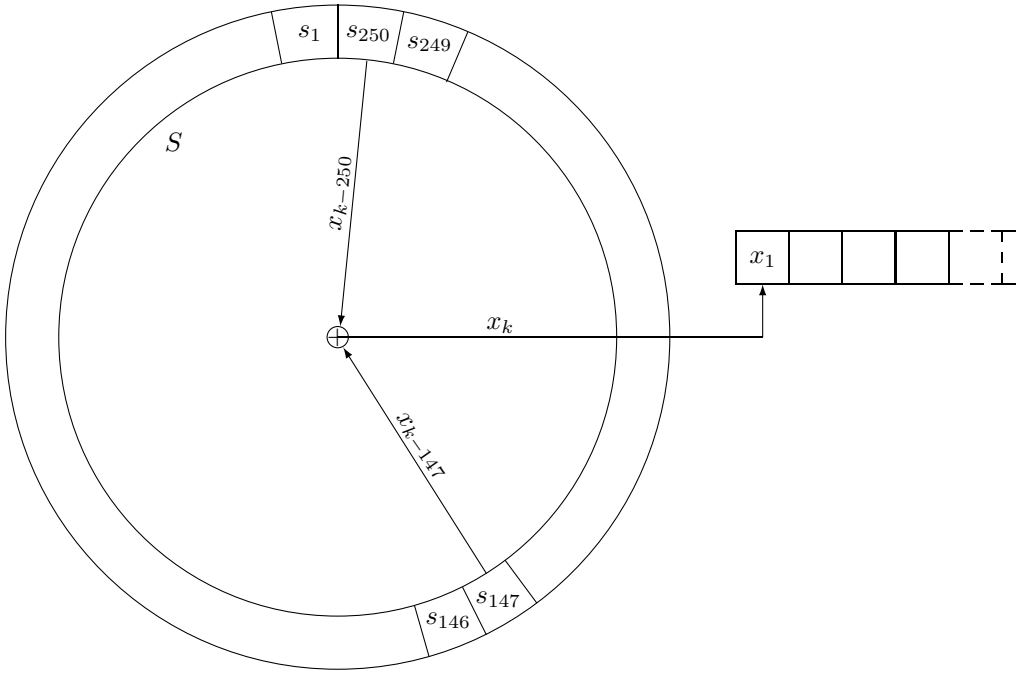
$$x_k = (ax_{k-1} + b) \bmod 2^{31}$$

[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.

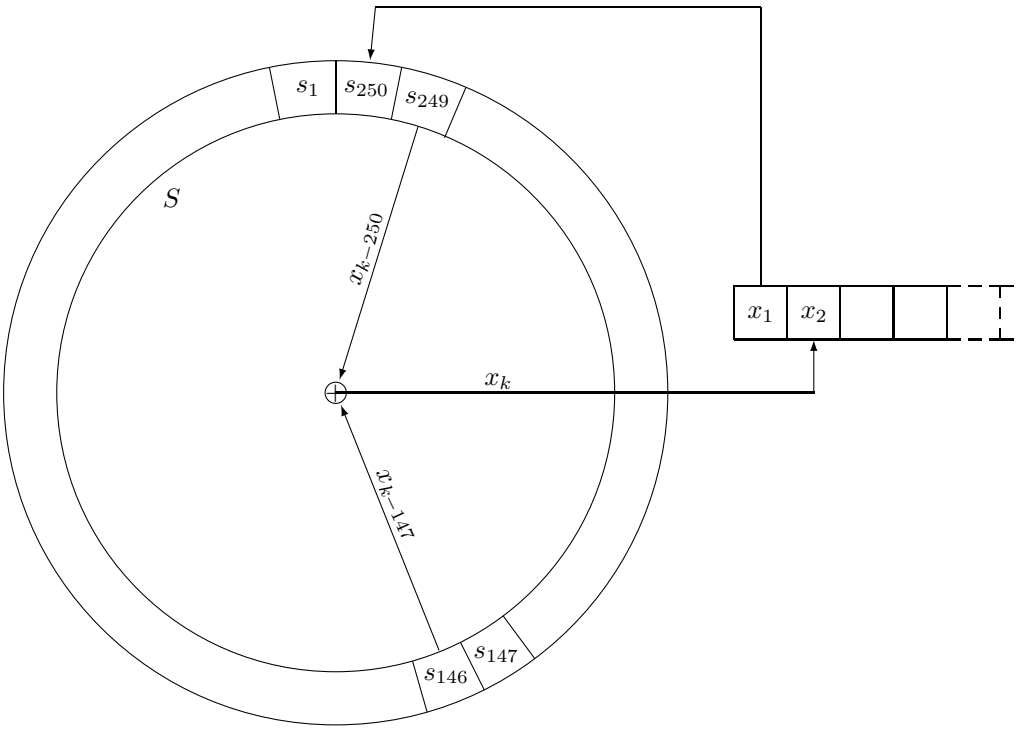
Depending on the numbers chosen for a and b the mixed congruential algorithm can generate up to 2^{31} numbers before the sequence begins to repeat, but some simulations use more. The **generalized shift register algorithm**

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

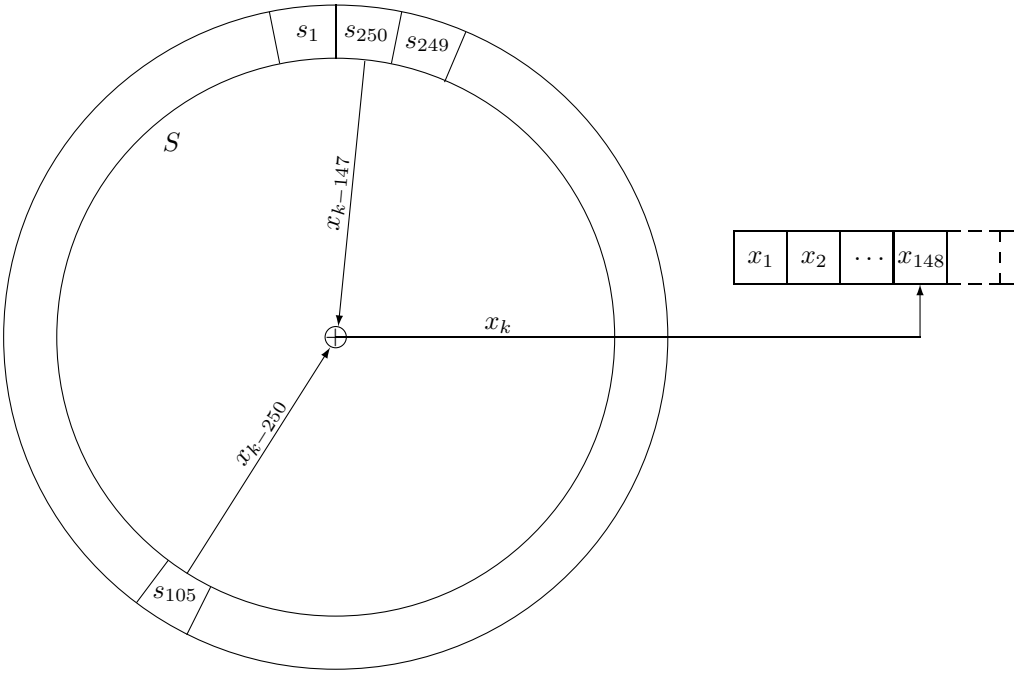
[53, §13.3] [203], in which $\oplus$ denotes the bitwise XOR operation (see §4.6.3), generates $2^{250}-1$ values before it repeats. It requires that we remember the previous 250 numbers that have been generated, which makes it also interesting but not so simple as the mixed congruential algorithm. To 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** $s_1 \ldots s_{250}$, 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) ⊕ 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(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 [203, 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 ≤ 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.

[203] **Kirkpatrick, Scott.** and **Stoll, Erich P.**, "A Very Fast Shift-Register Sequence Random Number Generator," *Journal of Computational Physics* 40 (1981) 517-526.

```
      SUBROUTINE DR250(N,X)
C     This routine generates a vector X of N normalized
C     double-precision pseudorandom numbers in the interval [0,1]
C
C     variable  meaning
C     --------  -------
C     A         multiplier for mixed congruential algorithm
C     C         increment for mixed congruential algorithm
C     GETIMEOFDAY  unix routine returns time of day and time zone
C     IABS      Fortran function gives |INTEGER*4|
C     IAND      Fortran function gives bitwise AND of fullwords
C     IEOR      Fortran function gives bitwise XOR of fullwords
C     IOR       Fortran function gives bitwise OR of fullwords
C     ISR       internal shift register
C     IX        a doubleword of X as 2 singlewords
C     JX        IX with the words switched
C     K         index on random numbers generated
C     KM147     index in ISR of doubleword K-147
C     KM250     index in ISR of doubleword K-250
C     MASKC     to set characteristic of ISR values generated
C     MASKX     to zero characteristic of ISR values generated
C     MASKZ     to zero the high-order bit of a word
C     N         number of random values needed, or seed (see above)
C     NODD      to make SEED end-bits ones
C     TOD       time-of-day [seconds,microseconds]
C     SEED      seed used for mixed congruential algorithm
C     SET       T => N has been set to -SEED for return
C     X         vector of random values returned
C     XK        a doubleword of X
C     ZONE      unused; for GETIMEOFDAY
C
C     formal parameter
      REAL*8 X(N)
C
C     prepare to seed the generator
      INTEGER*4 SEED,TOD(2),ZONE(2),NODD/Z'80000001'/
      INTEGER*4 A/843314861/,C/453816693/
      INTEGER*4 MASKZ/Z'7FFFFFFF'/,MASKX/Z'000FFFFF'/
      LOGICAL*4 SET
C
C     prepare to run the generator
      INTEGER*4 KM250/251/,KM147/148/,MASKC/Z'3FF00000'/
      REAL*8 XK
      INTEGER*4 IX(2),JX(2)
      EQUIVALENCE(XK,IX(1))
C
C     internal shift register with values from SEED=123457
      INTEGER*4 ISR(500)/
     ;          Z'0007E8AF',Z'D4C00D62',
                248 lines of data
     ;          Z'0003731D',Z'8AD80548'/
C
C ---------------------------------------------------------------------
C
```

```
C     is this an initialization call?
      IF(N.LE.0) GO TO 1
C
C     generate N new random numbers while reseeding the generator
      DO 2 K=1,N
C          find the indices in ISR of X(K-250) and X(K-147)
           KM250=KM250-1
           IF(KM250.LE.0) KM250=250
           KM147=KM147-1
           IF(KM147.LE.0) KM147=250
C
C          exclusive-or the singlewords
           ISR(2*KM250  )=IEOR(ISR(2*KM250  ),ISR(2*KM147  ))
           ISR(2*KM250-1)=IEOR(ISR(2*KM250-1),ISR(2*KM147-1))
C
C          extract the resulting doubleword
           JX(1)=IOR(ISR(2*KM250-1),MASKC)
           JX(2)=ISR(2*KM250)
C          the word order is reversed in the Pentium
           IX(1)=JX(2)
           IX(2)=JX(1)
           X(K)=XK-1.D0
    2 CONTINUE
      RETURN
C
C     get a seed to use in the reloading process
    1 SET=.FALSE.
      SEED=IABS(N)
      IF(SEED.GT.0) GO TO 3
      CALL GETIMEOFDAY(TOD,ZONE)
      TOD(1)=A*TOD(1)+C
      TOD(2)=A*TOD(2)+C
      N=IEOR(TOD(1),TOD(2))
      N=IOR(N,NODD)
      SEED=-N
      SET=.TRUE.
C
C     reload the ISR using a mixed congruential algorithm
    3 DO 4 K=1,250
C          low-order word of the 8-byte value
           SEED=A*SEED+C
           ISR(2*K)=SEED
           SEED=IAND(SEED,MASKZ)
C
C          high-order word of the 8-byte value
           SEED=A*SEED+C
           ISR(2*K-1)=IAND(SEED,MASKX)
           SEED=IAND(SEED,MASKZ)
    4 CONTINUE
      IF(.NOT.SET) N=-SEED
      KM250=251
      KM147=148
      RETURN
      END
```