As for references on this subject, the one to turn to first is Knuth [1]. Then try [2]. Only a few of the standard books on numerical methods [3-4] treat topics relating to random numbers.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [2]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 Uniform Deviates

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think "random numbers" are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

System-Supplied Random Number Generators

Your computer very likely has lurking within it a library routine which is called a "random number generator." That routine typically has an unforgettable name like "`ran`," and a calling sequence like

```
x=ran(iseed)          sets x to the next random number and updates iseed
```

You initialize `iseed` to a (usually) arbitrary value before the first call to `ran`. Each initializing value will typically return a different subsequent random sequence, or at least a different subsequence of some one enormously long sequence. The *same* initializing value of `iseed` will always return the *same* random sequence, however.

Now our first, and perhaps most important, lesson in this chapter is: Be *very, very* suspicious of a system-supplied `ran` that resembles the one just described. If all scientific papers whose results are in doubt because of bad `ran`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied `ran`s are almost always *linear congruential generators*, which

generate a sequence of integers $I_1, I_2, I_3, \ldots$, each between 0 and $m-1$ (a large number) by the recurrence relation

$$I_{j+1} = aI_j + c \pmod{m} \tag{7.1.1}$$

Here m is called the *modulus*, and a and c are positive integers called the *multiplier* and the *increment*, respectively. The recurrence (7.1.1) will eventually repeat itself, with a period that is obviously no greater than m. If m, a, and c are properly chosen, then the period will be of maximal length, i.e., of length m. In that case, all possible integers between 0 and $m-1$ occur at some point, so any initial "seed" choice of I_0 is as good as any other: The sequence just takes off from that point. The real number between 0 and 1 which is returned is generally I_{j+1}/m, so that it is strictly less than 1, but occasionally (once in m calls) exactly equal to zero. `iseed` is set to I_{j+1} (or some encoding of it), so that it can be used on the next call to generate I_{j+2}, and so on.

The linear congruential method has the advantage of being very fast, requiring only a few operations per call, hence its almost universal use. It has the disadvantage that it is not free of sequential correlation on successive calls. If k random numbers at a time are used to plot points in k dimensional space (with each coordinate between 0 and 1), then the points will not tend to "fill up" the k-dimensional space, but rather will lie on $(k-1)$-dimensional "planes." There will be *at most* about $m^{1/k}$ such planes. If the constants m, a, and c are not very carefully chosen, there will be *many fewer than that*. The number m is usually close to the machine's largest representable integer, e.g., $\sim 2^{32}$. So, for example, the number of planes on which triples of points lie in three-dimensional space is usually no greater than about the cube root of 2^{32}, about 1600. You might well be focusing attention on a physical process that occurs in a small fraction of the total volume, so that the discreteness of the planes can be very pronounced.

Even worse, you might be using a `ran` whose choices of m, a, and c have been botched. One infamous such routine, `RANDU`, with $a = 65539$ and $m = 2^{31}$, was widespread on IBM mainframe computers for many years, and widely copied onto other systems [1]. One of us recalls producing a "random" plot with only 11 planes, and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." Figure that out.

Correlation in k-space is not the only weakness of linear congruential generators. Such generators often have their low-order (least significant) bits much less random than their high-order bits. If you want to generate a random integer between 1 and 10, you should always do it using high-order bits, as in

```
j=1+int(10.*ran(iseed))
```

and never by anything resembling

```
j=1+mod(int(1000000.*ran(iseed)),10)
```

(which uses lower-order bits). Similarly you should never try to take apart a "ran" number into several supposedly random pieces. Instead use separate calls for every piece.

Portable Random Number Generators

Park and Miller [1] have surveyed a large number of random number generators that have been used over the last 30 years or more. Along with a good theoretical review, they present an anecdotal sampling of a number of inadequate generators that have come into widespread use. The historical record is nothing if not appalling.

There is good evidence, both theoretical and empirical, that the simple multiplicative congruential algorithm

$$I_{j+1} = aI_j \pmod m \tag{7.1.2}$$

can be as good as any of the more general linear congruential generators that have $c \neq 0$ (equation 7.1.1) — *if* the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "Minimal Standard" generator based on the choices

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2147483647 \tag{7.1.3}$$

First proposed by Lewis, Goodman, and Miller in 1969, this generator has in subsequent years passed all new theoretical tests, and (perhaps more importantly) has accumulated a large amount of successful use. Park and Miller do not claim that the generator is "perfect" (we will see below that it is not), but only that it is a good minimal standard against which other generators should be judged.

It is not possible to implement equations (7.1.2) and (7.1.3) directly in a high-level language, since the product of a and $m - 1$ exceeds the maximum value for a 32-bit integer. Assembly language implementation using a 64-bit product register is straightforward, but not portable from machine to machine. A trick due to Schrage [2,3] for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit) is therefore extremely interesting: It allows the Minimal Standard generator to be implemented in essentially any programming language on essentially any machine.

Schrage's algorithm is based on an *approximate factorization* of m,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a],\ r = m \bmod a \tag{7.1.4}$$

with square brackets denoting integer part. If r is small, specifically $r < q$, and $0 < z < m - 1$, it can be shown that both $a(z \bmod q)$ and $r[z/q]$ lie in the range $0, \ldots, m - 1$, and that

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is} \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases} \tag{7.1.5}$$

The application of Schrage's algorithm to the constants (7.1.3) uses the values $q = 127773$ and $r = 2836$.

Here is an implementation of the Minimal Standard generator:

```
      FUNCTION ran0(idum)
      INTEGER idum,IA,IM,IQ,IR,MASK
      REAL ran0,AM
      PARAMETER (IA=16807,IM=2147483647,AM=1./IM,
     *     IQ=127773,IR=2836,MASK=123459876)
        "Minimal" random number generator of Park and Miller. Returns a uniform random deviate
        between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK)
        to initialize the sequence; idum must not be altered between calls for successive deviates
        in a sequence.
      INTEGER k
      idum=ieor(idum,MASK)            XORing with MASK allows use of zero and other simple
      k=idum/IQ                           bit patterns for idum.
      idum=IA*(idum-k*IQ)-IR*k        Compute idum=mod(IA*idum,IM) without overflows by
      if (idum.lt.0) idum=idum+IM         Schrage's method.
      ran0=AM*idum                    Convert idum to a floating result.
      idum=ieor(idum,MASK)            Unmask before return.
      return
      END
```

The period of ran0 is $2^{31} - 2 \approx 2.1 \times 10^9$. A peculiarity of generators of the form (7.1.2) is that the value 0 must never be allowed as the initial seed — it perpetuates itself — and it never occurs for any nonzero initial seed. Experience has shown that users always manage to call random number generators with the seed idum=0. That is why ran0 performs its exclusive-or with an arbitrary constant both on entry and exit. If you are the first user in history to be proof against human error, you can remove the two lines with the ieor function.

Park and Miller discuss two other multipliers a that can be used with the same $m = 2^{31} - 1$. These are $a = 48271$ (with $q = 44488$ and $r = 3399$) and $a = 69621$ (with $q = 30845$ and $r = 23902$). These can be substituted in the routine ran0 if desired; they may be slightly superior to Lewis *et al.*'s longer-tested values. No values other than these should be used.

The routine ran0 is a Minimal Standard, satisfactory for the majority of applications, but we do not recommend it as the final word on random number generators. Our reason is precisely the simplicity of the Minimal Standard. It is not hard to think of situations where successive random numbers might be used in a way that accidentally conflicts with the generation algorithm. For example, since successive numbers differ by a multiple of only 1.6×10^4 out of a modulus of more than 2×10^9, very small random numbers will tend to be followed by smaller than average values. One time in 10^6, for example, there will be a value $< 10^{-6}$ returned (as there should be), but this will *always* be followed by a value less than about 0.0168. One can easily think of applications involving rare events where this property would lead to wrong results.

There are other, more subtle, serial correlations present in ran0. For example, if successive points (I_i, I_{i+1}) are binned into a two-dimensional plane for $i = 1, 2, \ldots, N$, then the resulting distribution fails the χ^2 test when N is greater than a few $\times 10^7$, much less than the period $m - 2$. Since low-order serial correlations have historically been such a bugaboo, and since there is a very simple way to remove them, we think that it is prudent to do so.

The following routine, ran1, uses the Minimal Standard for its random value, but it shuffles the output to remove low-order serial correlations. A random deviate derived from the jth value in the sequence, I_j, is output not on the jth call, but rather on a randomized later call, $j + 32$ on average. The shuffling algorithm is due to Bays and Durham as described in Knuth [4], and is illustrated in Figure 7.1.1.

```
      FUNCTION ran1(idum)
      INTEGER idum,IA,IM,IQ,IR,NTAB,NDIV
      REAL ran1,AM,EPS,RNMX
      PARAMETER (IA=16807,IM=2147483647,AM=1./IM,IQ=127773,IR=2836,
     *     NTAB=32,NDIV=1+(IM-1)/NTAB,EPS=1.2e-7,RNMX=1.-EPS)
        "Minimal" random number generator of Park and Miller with Bays-Durham shuffle and
        added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of
        the endpoint values). Call with idum a negative integer to initialize; thereafter, do not
        alter idum between successive deviates in a sequence. RNMX should approximate the largest
        floating value that is less than 1.
      INTEGER j,k,iv(NTAB),iy
      SAVE iv,iy
      DATA iv /NTAB*0/, iy /0/
      if (idum.le.0.or.iy.eq.0) then  Initialize.
        idum=max(-idum,1)             Be sure to prevent idum = 0.
        do 11 j=NTAB+8,1,-1           Load the shuffle table (after 8 warm-ups).
          k=idum/IQ
          idum=IA*(idum-k*IQ)-IR*k
          if (idum.lt.0) idum=idum+IM
          if (j.le.NTAB) iv(j)=idum
        enddo 11
        iy=iv(1)
      endif
      k=idum/IQ                       Start here when not initializing.
      idum=IA*(idum-k*IQ)-IR*k        Compute idum=mod(IA*idum,IM) without overflows by
      if (idum.lt.0) idum=idum+IM       Schrage's method.
      j=1+iy/NDIV                     Will be in the range 1:NTAB.
      iy=iv(j)                        Output previously stored value and refill the shuffle ta-
      iv(j)=idum                        ble.
      ran1=min(AM*iy,RNMX)            Because users don't expect endpoint values.
      return
      END
```

The routine `ran1` passes those statistical tests that `ran0` is known to fail. In fact, we do not know of any statistical test that `ran1` fails to pass, except when the number of calls starts to become on the order of the period m, say $> 10^8 \approx m/20$.

For situations when even longer random sequences are needed, L'Ecuyer [6] has given a good way of combining two different sequences with different periods so as to obtain a new sequence whose period is the least common multiple of the two periods. The basic idea is simply to add the two sequences, modulo the modulus of *either* of them (call it m). A trick to avoid an intermediate value that overflows the integer wordsize is to subtract rather than add, and then add back the constant $m - 1$ if the result is ≤ 0, so as to wrap around into the desired interval $0, \ldots, m - 1$.

Notice that it is not necessary that this wrapped subtraction be able to reach all values $0, \ldots, m - 1$ from *every* value of the first sequence. Consider the absurd extreme case where the value subtracted was only between 1 and 10: The resulting sequence would still be no less random than the first sequence by itself. As a practical matter it is only necessary that the second sequence have a range covering *substantially* all of the range of the first. L'Ecuyer recommends the use of the two generators $m_1 = 2147483563$ (with $a_1 = 40014$, $q_1 = 53668$, $r_1 = 12211$) and $m_2 = 2147483399$ (with $a_2 = 40692$, $q_2 = 52774$, $r_2 = 3791$). Both moduli are slightly less than 2^{31}. The periods $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$ and $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ share only the factor 2, so the period of the combined generator is $\approx 2.3 \times 10^{18}$. For present computers, period exhaustion is a practical impossibility.

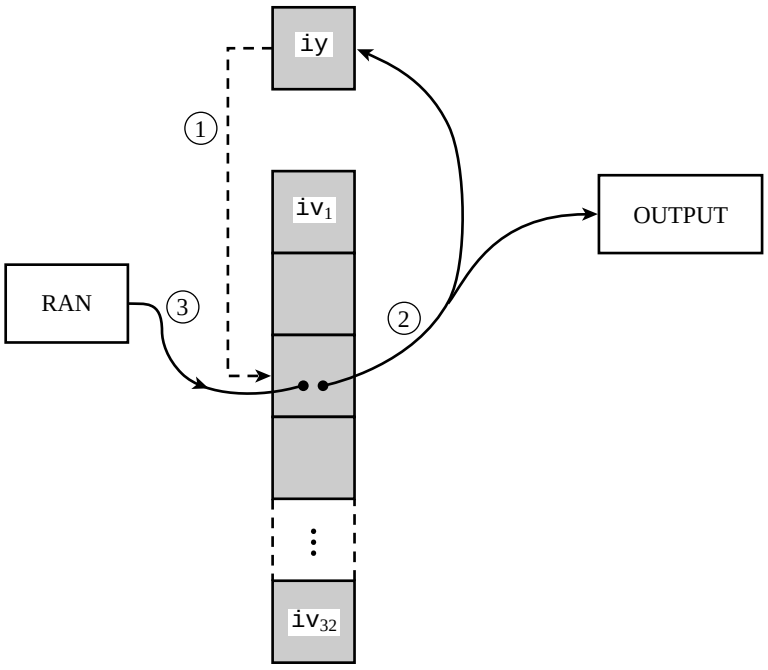

Figure 7.1.1. Shuffling procedure used in ran1 to break up sequential correlations in the Minimal Standard generator. Circled numbers indicate the sequence of events: On each call, the random number in iy is used to choose a random element in the array iv. That element becomes the output random number, and also is the next iy. Its spot in iv is refilled from the Minimal Standard routine.

Combining the two generators breaks up serial correlations to a considerable extent. We nevertheless recommend the additional shuffle that is implemented in the following routine, ran2. We think that, within the limits of its floating-point precision, ran2 provides perfect random numbers; a practical definition of "perfect" is that we will pay $1000 to the first reader who convinces us otherwise (by finding a statistical test that ran2 fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation).

```
      FUNCTION ran2(idum)
      INTEGER idum,IM1,IM2,IMM1,IA1,IA2,IQ1,IQ2,IR1,IR2,NTAB,NDIV
      REAL ran2,AM,EPS,RNMX
      PARAMETER (IM1=2147483563,IM2=2147483399,AM=1./IM1,IMM1=IM1-1,
     *     IA1=40014,IA2=40692,IQ1=53668,IQ2=52774,IR1=12211,
     *     IR2=3791,NTAB=32,NDIV=1+IMM1/NTAB,EPS=1.2e-7,RNMX=1.-EPS)
        Long period (> 2 × 10^18) random number generator of L'Ecuyer with Bays-Durham shuffle
        and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive
        of the endpoint values). Call with idum a negative integer to initialize; thereafter, do not
        alter idum between successive deviates in a sequence. RNMX should approximate the largest
        floating value that is less than 1.
      INTEGER idum2,j,k,iv(NTAB),iy
      SAVE iv,iy,idum2
      DATA idum2/123456789/, iv/NTAB*0/, iy/0/
      if (idum.le.0) then                    Initialize.
        idum=max(-idum,1)                    Be sure to prevent idum = 0.
        idum2=idum
        do 11 j=NTAB+8,1,-1                  Load the shuffle table (after 8 warm-ups).
          k=idum/IQ1
```

```
        idum=IA1*(idum-k*IQ1)-k*IR1
        if (idum.lt.0) idum=idum+IM1
        if (j.le.NTAB) iv(j)=idum
      enddo 11
      iy=iv(1)
    endif
    k=idum/IQ1                          Start here when not initializing.
    idum=IA1*(idum-k*IQ1)-k*IR1         Compute idum=mod(IA1*idum,IM1) without over-
    if (idum.lt.0) idum=idum+IM1            flows by Schrage's method.
    k=idum2/IQ2
    idum2=IA2*(idum2-k*IQ2)-k*IR2       Compute idum2=mod(IA2*idum2,IM2) likewise.
    if (idum2.lt.0) idum2=idum2+IM2
    j=1+iy/NDIV                         Will be in the range 1:NTAB.
    iy=iv(j)-idum2                      Here idum is shuffled, idum and idum2 are com-
    iv(j)=idum                              bined to generate output.
    if(iy.lt.1)iy=iy+IMM1
    ran2=min(AM*iy,RNMX)                Because users don't expect endpoint values.
    return
    END
```

L'Ecuyer [6] lists additional short generators that can be combined into longer ones, including generators that can be implemented in 16-bit integer arithmetic.

Finally, we give you Knuth's suggestion [4] for a portable routine, which we have translated to the present conventions as `ran3`. This is not based on the linear congruential method at all, but rather on a *subtractive method* (see also [5]). One might hope that its weaknesses, if any, are therefore of a highly different character from the weaknesses, if any, of `ran1` above. If you ever suspect trouble with one routine, it is a good idea to try the other in the same application. `ran3` has one nice feature: if your machine is poor on integer arithmetic (i.e., is limited to 16-bit integers), substitution of the three "commented" lines for the ones directly preceding them will render the routine entirely floating-point.

```
    FUNCTION ran3(idum)
      Returns a uniform random deviate between 0.0 and 1.0. Set idum to any negative value
      to initialize or reinitialize the sequence.
    INTEGER idum
    INTEGER MBIG,MSEED,MZ
C   REAL MBIG,MSEED,MZ
    REAL ran3,FAC
    PARAMETER (MBIG=1000000000,MSEED=161803398,MZ=0,FAC=1./MBIG)
C   PARAMETER (MBIG=4000000.,MSEED=1618033.,MZ=0.,FAC=1./MBIG)
      According to Knuth, any large mbig, and any smaller (but still large) mseed can be sub-
      stituted for the above values.
    INTEGER i,iff,ii,inext,inextp,k
    INTEGER mj,mk,ma(55)                The value 55 is special and should not be modified; see
C   REAL mj,mk,ma(55)                       Knuth.
    SAVE iff,inext,inextp,ma
    DATA iff /0/
    if(idum.lt.0.or.iff.eq.0)then       Initialization.
      iff=1
      mj=abs(MSEED-abs(idum))           Initialize ma(55) using the seed idum and the large num-
      mj=mod(mj,MBIG)                       ber mseed.
      ma(55)=mj
      mk=1
      do 11 i=1,54                      Now initialize the rest of the table,
        ii=mod(21*i,55)                 in a slightly random order,
        ma(ii)=mk                       with numbers that are not especially random.
        mk=mj-mk
        if(mk.lt.MZ)mk=mk+MBIG
```

```
        mj=ma(ii)
    enddo 11
    do 13 k=1,4                    We randomize them by "warming up the generator."
        do 12 i=1,55
            ma(i)=ma(i)-ma(1+mod(i+30,55))
            if(ma(i).lt.MZ)ma(i)=ma(i)+MBIG
        enddo 12
    enddo 13
    inext=0                        Prepare indices for our first generated number.
    inextp=31                      The constant 31 is special; see Knuth.
    idum=1
endif
inext=inext+1                      Here is where we start, except on initialization. Increment
if(inext.eq.56)inext=1                 inext, wrapping around 56 to 1.
inextp=inextp+1                    Ditto for inextp.
if(inextp.eq.56)inextp=1
mj=ma(inext)-ma(inextp)            Now generate a new random number subtractively.
if(mj.lt.MZ)mj=mj+MBIG             Be sure that it is in range.
ma(inext)=mj                       Store it,
ran3=mj*FAC                        and output the derived uniform deviate.
return
END
```

Quick and Dirty Generators

One sometimes would like a "quick and dirty" generator to embed in a program, perhaps taking only one or two lines of code, just to *somewhat* randomize things. One might wish to process data from an experiment not always in exactly the same order, for example, so that the first output is more "typical" than might otherwise be the case.

For this kind of application, all we really need is a list of "good" choices for m, a, and c in equation (7.1.1). If we don't need a period longer than 10^4 to 10^6, say, we can keep the value of $(m-1)a+c$ small enough to avoid overflows that would otherwise mandate the extra complexity of Schrage's method (above). We can thus easily embed in our programs

```
jran=mod(jran*ia+ic,im)
ran=float(jran)/float(im)
```

whenever we want a quick and dirty uniform deviate, or

```
jran=mod(jran*ia+ic,im)
j=jlo+((jhi-jlo+1)*jran)/im
```

whenever we want an integer between `jlo` and `jhi`, inclusive. (In both cases `jran` was once initialized to any seed value between 0 and `im-1`.)

Be sure to remember, however, that when `im` is small, the kth root of it, which is the number of planes in k-space, is even smaller! So a quick and dirty generator should never be used to select points in k-space with $k > 1$.

With these caveats, some "good" choices for the constants are given in the accompanying table. These constants (i) give a period of maximal length `im`, and, more important, (ii) pass Knuth's "spectral test" for dimensions 2, 3, 4, 5, and 6. The increment `ic` is a prime, close to the value $(\frac{1}{2} - \frac{1}{6}\sqrt{3})$`im`; actually almost any value of `ic` that is relatively prime to `im` will do just as well, but there is some "lore" favoring this choice (see [4], p. 84).

Constants for Quick and Dirty Random Number Generators

overflow at	im	ia	ic
	6075	106	1283
2^{20}			
	7875	211	1663
2^{21}			
	7875	421	1663
2^{22}			
	6075	1366	1283
	6655	936	1399
	11979	430	2531
2^{23}			
	14406	967	3041
	29282	419	6173
	53125	171	11213
2^{24}			
	12960	1741	2731
	14000	1541	2957
	21870	1291	4621
	31104	625	6571
	139968	205	29573
2^{25}			
	29282	1255	6173
	81000	421	17117
	134456	281	28411
2^{26}			

overflow at	im	ia	ic
	86436	1093	18257
	121500	1021	25673
	259200	421	54773
2^{27}			
	117128	1277	24749
	121500	2041	25673
	312500	741	66037
2^{28}			
	145800	3661	30809
	175000	2661	36979
	233280	1861	49297
	244944	1597	51749
2^{29}			
	139968	3877	29573
	214326	3613	45289
	714025	1366	150889
2^{30}			
	134456	8121	28411
	259200	7141	54773
2^{31}			
	233280	9301	49297
	714025	4096	150889
2^{32}			

An Even Quicker and Dirtier Generator

Many FORTRAN compilers can be abused in such a way that they will multiply two 32-bit integers *ignoring any resulting overflow*. In such cases, on many machines, the value returned is predictably the low-order 32 bits of the true 64-bit product. (C compilers, incidentally, can do this without the requirement of abuse — it is guaranteed behavior for so-called `unsigned long int` integers. On VMS VAXes, the necessary FORTRAN command is `FORTRAN/CHECK=NOOVERFLOW`.) If we now choose $m = 2^{32}$, the "mod" in equation (7.1.1) is free, and we have simply

$$I_{j+1} = aI_j + c \tag{7.1.6}$$

Knuth suggests $a = 1664525$ as a suitable multiplier for this value of m. H.W. Lewis has conducted extensive tests of this value of a with $c = 1013904223$, which is a prime close to $(\sqrt{5} - 2)m$. The resulting in-line generator (we will call it `ranqd1`) is simply

```
idum=1664525*idum+1013904223
```

This is about as good as any 32-bit linear congruential generator, entirely adequate for many uses. And, with only a single multiply and add, it is *very* fast.

To check whether your compiler and machine have the desired overflow properties, see if you can generate the following sequence of 32-bit values (given here in hex): 00000000, 3C6EF35F, 47502932, D1CCF6E9, AAF95334, 6252E503, 9F2EC686, 57FE6C2D, A3D95FA8, 81FDBEE7, 94F0AF1A, CBF633B1.

If you need floating-point values instead of 32-bit integers, and want to avoid a divide by floating-point 2^{32}, a dirty trick is to mask in an exponent that makes the value lie between 1 and 2, then subtract 1.0. The resulting in-line generator (call it `ranqd2`) will look something like

```
      INTEGER idum,itemp,jflone,jflmsk
      REAL ftemp
      EQUIVALENCE (itemp,ftemp)
      DATA jflone /Z'3F800000'/, jflmsk /Z'007FFFFF'/
C     ...
      idum=1664525*idum+1013904223
      itemp=ior(jflone,iand(jflmsk,idum))
      ran=ftemp-1.0
```

The hex constants 3F800000 and 007FFFFF are the appropriate ones for computers using the IEEE representation for 32-bit floating-point numbers (e.g., IBM PCs and most UNIX workstations). For DEC VAXes, the correct hex constants are, respectively, 00004080 and FFFF007F. Notice that the IEEE mask results in the floating-point number being constructed out of the 23 low-order bits of the integer, which is not ideal. Also notice that your compiler may require a different notation for hex constants, e.g., `x'3f800000'`, `'3F800000'X`, or even `16#3F800000`. (Your authors have tried very hard to make *almost all* of the material in this book machine and compiler independent — indeed, even programming language independent. This subsection is a rare aberration. Forgive us. Once in a great while the temptation to be *really dirty* is just irresistible.)

Relative Timings and Recommendations

Timings are inevitably machine dependent. Nevertheless the following table is indicative of the *relative* timings, for typical machines, of the various uniform generators discussed in this section, plus `ran4` from §7.5. Smaller values in the table indicate faster generators. The generators `ranqd1` and `ranqd2` refer to the "quick and dirty" generators immediately above.

Generator	Relative Execution Time
ran0	$\equiv 1.0$
ran1	≈ 1.3
ran2	≈ 2.0
ran3	≈ 0.6
ranqd1	≈ 0.10
ranqd2	≈ 0.25
ran4	≈ 4.0

On balance, we recommend `ran1` for general use. It is portable, based on Park and Miller's Minimal Standard generator with an additional shuffle, and has no known (to us) flaws other than period exhaustion.

If you are generating more than 100,000,000 random numbers in a single calculation (that is, more than about 5% of `ran1`'s period), we recommend the use of `ran2`, with its much longer period.

Knuth's subtractive routine `ran3` seems to be the timing winner among portable routines. Unfortunately the subtractive method is not so well studied, and not a standard. We like to keep `ran3` in reserve for a "second opinion," substituting it when we suspect another generator of introducing unwanted correlations into a calculation.

The routine `ran4` generates *extremely* good random deviates, and has some other nice properties, but it is slow. See §7.5 for discussion.

Finally, the quick and dirty in-line generators `ranqd1` and `ranqd2` are very fast, but they are machine dependent, nonportable, and at best only as good as a 32-bit linear congruential generator ever is — in our view not good enough in many situations. We would use these only in very special cases, where speed is critical.

CITED REFERENCES AND FURTHER READING:

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation* (New York: Springer-Verlag). [3]

Knuth, D.E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 Transformation Method: Exponential and Normal Deviates

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.2.1}$$

The probability distribution $p(x)$ is of course normalized, so that

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

or

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$