REFERENCES AND FURTHER READING:

Bulirsch, Roland, 1965, *Numerische Mathematik*, vol. 7, p.78; 1965, *op. cit.*, vol. 7, p.353; 1969, *op. cit.*, vol. 13, p.305.

Abramowitz, Milton, and Stegun, Irene A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 17.

Mathews, Jon, and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, Mass.: W.A. Benjamin/Addison-Wesley), pp. 78-79.

Chapter 7. Random Numbers

7.0 Introduction

It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers. More than perverse, it may seem to be a conceptual impossibility. Any program, after all, will produce output that is entirely predictable, hence not truly "random."

Nevertheless, practical computer "random number generators" are in common use. We will leave it to philosophers of the computer age to resolve the paradox in a deep way (see, e.g., Knuth §3.5 for discussion and references). One sometimes hears computer-generated sequences termed *quasi-random*, while the word *random* is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. We will not try to make such fine distinctions.

A working, though imprecise, definition of randomness in the context of computer-generated sequences, is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don't, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a body of random number generators which mutually do satisfy the definition over a very, very broad class of applications programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. So, because such generators are known to exist, we can leave to the philosophers the problem of defining them.

A pragmatic point of view, then, is that randomness is in the eye of the beholder (or applications programmer). What is random enough for one application may not be random enough for another. Still, one is not entirely adrift in a sea of incommensurable applications programs: There is a certain list of statistical tests, some sensible and some merely enshrined by

history, which on the whole will do a very good job of ferreting out any correlations that are likely to be detected by an applications program (in this case, yours). Good random number generators ought to pass all of these tests; or else the user had better at least be aware of any that they fail, so that he or she is able to judge whether they are relevant to the case at hand.

As for references on this subject, there is really only one worth turning to first, and that is Knuth. Only a few of the standard books on numerical methods treat topics relating to random numbers.

REFERENCES AND FURTHER READING:

Knuth, Donald E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley), Chapter 3.

Dahlquist, Germund, and Bjorck, Ake. 1974, *Numerical Methods* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 11.

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 10.

7.1 Uniform Deviates

Uniform deviates are just random numbers which 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 which resembles the one just described. If all scientific papers whose results are in doubt because of bad RANs were to disappear from library shelves, there would be a gap on each shelf about as big as your fist. System-supplied RANs 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 returned as 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. *And if the constants m, a, and c are not very carefully chosen, there will be many fewer than that.* The number m is usually about the wordsize of the machine, e.g. 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 was widespread on IBM computers for many years, and widely copied onto other systems. 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."

Correlation in k-space is not the only weakness of linear congruential generators. Such generators often have their low-order (least significant) bits

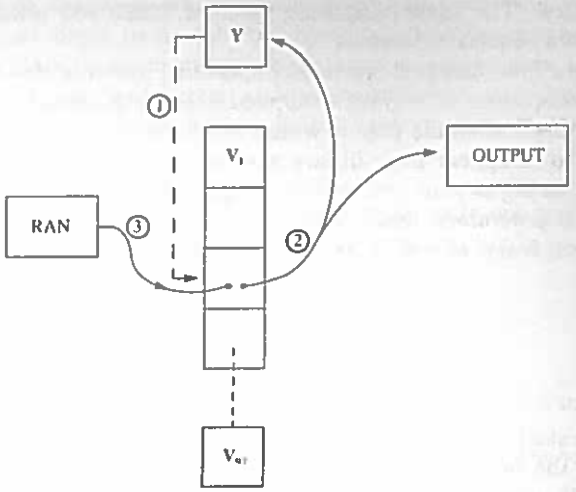

Figure 7.1.1. Shuffling procedure used in RANO to break up sequential correlations in a system-supplied random number generator. Circled numbers indicate the sequence of events: On each call, the random number in Y is used to choose a random element in the array V. That element becomes the output random number, and also is the next Y. Its spot in V is refilled from the system-supplied routine.

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 by

```
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.

How to Improve a System-Supplied Routine

So, if RAN is so dangerous, what is a body to do? A minimum is to do an additional randomizing shuffle on the numbers generated by RAN. Instead of calling RAN, you can call RANO, the following routine, based on the algorithm of Bays and Durham as described in Knuth and illustrated in Figure 7.1.1.


```
FUNCTION RANO(IDUM)
   Returns a uniform random deviate between 0.0 and 1.0 using a system-supplied routine
   RAN(ISEED). Set IDUM to any negative value to initialize or reinitialize the sequence.
DIMENSION V(97)                    The exact number 97 is unimportant.
DATA IFF /0/                       As a precaution against misuse, we will always initialize on the first
IF(IDUM.LT.0.OR.IFF.EQ.0)THEN          call, even if IDUM is not set negative.
   IFF=1
   ISEED=ABS(IDUM)
   IDUM=1
   DO 11 J=1,97                    Exercise the system routine, especially important if the system's mul-
      DUM=RAN(ISEED)                   tiplier is small.
   11 CONTINUE
   DO 12 J=1,97                    Then save 97 values
      V(J)=RAN(ISEED)
   12 CONTINUE
   Y=RAN(ISEED)                    and a 98th.
ENDIF
J=1+INT(97.*Y)                     This is where we start if not initializing. Use the previously saved
IF(J.GT.97.OR.J.LT.1)PAUSE             random number Y to get an index J between 1 and 97. Then
Y=V(J)                                 use the corresponding V(J) for both the next J and as the
RANO=Y                                 output number.
V(J)=RAN(ISEED)                    Finally, refill the table entry with the next random number from RAN.
RETURN
END
```

The sequence returned by RANO will be effectively free of sequential correlation. Unless your system-supplied RAN is truly a botch, the output of RANO ought to be good for all purposes, although we would tend to remain suspicious of its lowest-order (least significant) bits.

Portable Random Number Generators

You may well want to have a "portable" random number generator which can be programmed in a high-level language, and which will generate the same random sequence (from a given seed) on all machines. We can distinguish two classes of use for such portable routines: First, one might want a fully reliable generator, at least as good as RANO given above. For this, and portability too, you are going to have to pay a price: the routines that we are about to give you will run considerably more slowly than RANO. Second, one frequently 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.

Let us address the second class of use first. All we really need is a list of "good" choices for m, a, and c. Then we can 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.)

There is a reason that the above fragments of high-level language cannot be made as good as the system-supplied RAN: we must choose IM (which is a bound on JRAN) and IA small enough so that their product does not produce an integer overflow. The machine-language programmer has no such restriction: she can multiply two integers each as large as the machine's wordsize into a double-width register, then manipulate the high- and low-order words separately to take the MOD. So our "good" values of IM, IA, and IC are restricted by the largest product which does not overflow. (You can also use a machine's floating point arithmetic: in this case the restriction is to the largest product which represents an integer *exactly*, i.e. without losing the low-significance bits. Some computers have only 16-bit integers, but have 8-byte floating formats with up to 48 bits of mantissa; in this case a floating implementation is vastly preferred.)

Be sure to remember that when IM is small, the k^{th} root of it, which is the number of planes in k-space, is even smaller!

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 Knuth, p. 84).

We now turn to the first use of a portable random number generator, namely to generate good random numbers. We will give three suggested routines.

The first, RAN1, is based on three linear congruential generators from the above table. One generator is used for the most significant part of the output number, the second for the least significant part, and the third to control a shuffling routine (here absolutely essential because of the relatively small values of multiplier m available to us). The shuffling routine used is not the same as that illustrated in RANO above, but is another one which has been suggested by Knuth and widely used. RAN1 is, of course, very much better than any of its three generators individually: its period is (for all practical purposes) infinite, and it ought to have no sensible sequential correlations. The choice of which constants from the accompanying table to use is arbitrary, as is the length of the shuffling array.

```
FUNCTION RAN1(IDUM)
   Returns a uniform random deviate between 0.0 and 1.0. Set IDUM to any negative value
   to initialize or reinitialize the sequence.
DIMENSION R(97)
PARAMETER (M1=259200,IA1=7141,IC1=54773,RM1=1./M1)
PARAMETER (M2=134456,IA2=8121,IC2=28411,RM2=1./M2)
PARAMETER (M3=243000,IA3=4561,IC3=51349)
DATA IFF /0/                    As above. Initialize on first call even if IDUM is not negative.
IF (IDUM.LT.0.OR.IFF.EQ.0) THEN
```



```
  IFF=1
  IX1=MOD(IC1-IDUM,M1)          Seed the first routine,
  IX1=MOD(IA1*IX1+IC1,M1)
  IX2=MOD(IX1,M2)               and use it to seed the second
  IX1=MOD(IA1*IX1+IC1,M1)
  IX3=MOD(IX1,M3)               and third routines.
  DO 11 J=1,97                  Fill the table with sequential uniform deviates generated by the first
    IX1=MOD(IA1*IX1+IC1,M1)       two routines.
    IX2=MOD(IA2*IX2+IC2,M2)
    R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1     Low- and high-order pieces combined here.
  11 CONTINUE
  IDUM=1
ENDIF
IX1=MOD(IA1*IX1+IC1,M1)         Except when initializing, this is where we start. Generate the next
IX2=MOD(IA2*IX2+IC2,M2)           number for each sequence.
IX3=MOD(IA3*IX3+IC3,M3)
J=1+(97*IX3)/M3                 Use the third sequence to get an integer between 1 and 97.
IF(J.GT.97.OR.J.LT.1)PAUSE
RAN1=R(J)                       Return that table entry,
R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1     and refill it.
RETURN
END
```

For many purposes we *do* care that there be no sequential correlations, but we *don't* care that the discreteness of the random values returned be as fine as is allowed by all significant bits of the wordsize. In this case, one might desire the gain in speed that comes from using only one linear congruential generator, shuffling as was done in RANO above. Then the following routine is perfectly adequate:

```
FUNCTION RAN2(IDUM)
   Returns a uniform random deviate between 0.0 and 1.0. Set IDUM to any negative value
   to initialize or reinitialize the sequence.
PARAMETER (M=714025,IA=1366,IC=150889,RM=1./M)
DIMENSION IR(97)
DATA IFF /0/                    As above.
IF(IDUM.LT.0.OR.IFF.EQ.0)THEN
  IFF=1
  IDUM=MOD(IC-IDUM,M)
  DO 11 J=1,97                  Initialize the shuffle table.
    IDUM=MOD(IA*IDUM+IC,M)
    IR(J)=IDUM
  11 CONTINUE
  IDUM=MOD(IA*IDUM+IC,M)
  IY=IDUM                       Compare to RANO, above.
ENDIF
J=1+(97*IY)/M                   Here is where we start except on initialization.
IF(J.GT.97.OR.J.LT.1)PAUSE
IY=IR(J)
RAN2=IY*RM
IDUM=MOD(IA*IDUM+IC,M)
IR(J)=IDUM
RETURN
END
```

Constants for Portable Random Number Generators

overflow at	IM	IA	IC	overflow at	IM	IA	IC
	6075	106	1283		117128	1277	24749
2^{20}					312500	741	66037
	7875	211	1663		121500	2041	25673
2^{21}				2^{28}			
	7875	421	1663		120050	2311	25367
2^{22}					214326	1807	45289
	11979	430	2531		244944	1597	51749
	6655	936	1399		233280	1861	49297
	6075	1366	1283		175000	2661	36979
2^{23}					121500	4081	25673
	53125	171	11213		145800	3661	30809
	11979	859	2531	2^{29}			
	29282	419	6173		139968	3877	29573
	14406	967	3041		214326	3613	45289
2^{24}					714025	1366	150889
	134456	141	28411	2^{30}			
	31104	625	6571		134456	8121	28411
	14000	1541	2957		243000	4561	51349
	12960	1741	2731		259200	7141	54773
	21870	1291	4621	2^{31}			
	139968	205	29573		233280	9301	49297
2^{25}					714025	4096	150889
	81000	421	17117	2^{32}			
	29282	1255	6173		1771875	2416	374441
	134456	281	28411	2^{33}			
2^{26}					510300	17221	107839
	86436	1093	18257		312500	36261	66037
	259200	421	54773	2^{34}			
	116640	1021	24631		217728	84589	45989
	121500	1021	25673	2^{35}			
2^{27}							

The period of RAN2 is again effectively infinite. Its principal limitation is that it returns one of only 714025 possible values, equally spaced as a "comb" in the interval $[0, 1)$.

Finally, we give you Knuth's suggestion 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*. 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 two "commented" lines for the one directly following 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.
      IMPLICIT REAL*4(M)
      PARAMETER (MBIG=4000000.,MSEED=1618033.,MZ=0.,FAC=1./MBIG)
PARAMETER (MBIG=1000000000,MSEED=161803398,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.
DIMENSION MA(55)          This value is special and should not be modified; see Knuth.
DATA IFF /0/
IF(IDUM.LT.0.OR.IFF.EQ.0)THEN     Initialization.
  IFF=1
  MJ=MSEED-IABS(IDUM)     Initialize MA(55) using the seed IDUM and the large number MSEED.
  MJ=MOD(MJ,MBIG)
  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)
  11 CONTINUE
  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
    12 CONTINUE
  13 CONTINUE
  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 INEXT,
IF(INEXT.EQ.56)INEXT=1          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
```

REFERENCES AND FURTHER READING:

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

Forsythe, George E., Malcolm, Michael A., and Moler, Cleve B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, N.J.: 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

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

Exponential Deviates

As an example, suppose that $y(x) \equiv -\ln(x)$, and that $p(x)$ is as given by equation (7.2.1) for a uniform deviate. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.2.5}$$

which is distributed exponentially. This exponential distribution occurs frequently in real problems, usually as the distribution of waiting times between independent Poisson-random events, for example the radioactive decay of nuclei. You can also easily see (from 7.2.4) that the quantity y/λ has the probability distribution $\lambda e^{-\lambda y}$.

So we have

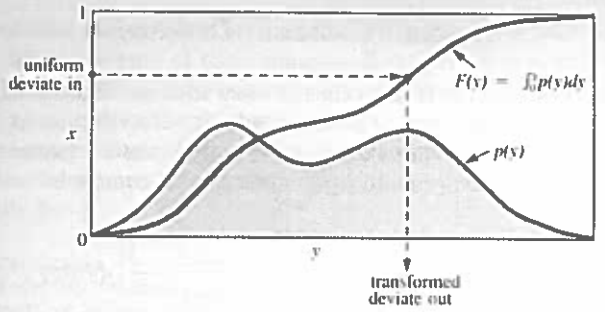

Figure 7.2.1. Transformation method for generating a random deviate y from a known probability distribution $p(y)$. The indefinite integral of $p(y)$ must be known and invertible. A uniform deviate x is chosen between 0 and 1. Its corresponding y on the definite-integral curve is the desired deviate.

```
FUNCTION EXPDEV(IDUM)
   Returns an exponentially distributed, positive, random deviate of unit mean, using
   RAN1(IDUM) as the source of uniform deviates.
EXPDEV=-LOG(RAN1(IDUM))
RETURN
END
```

Let's see what is involved in using the above *transformation method* to generate some arbitrary desired distribution of y's, say one with $p(y) = f(y)$ for some positive function f whose integral is 1. (See Figure 7.2.1.) According to (7.2.4), we need to solve the differential equation

$$\frac{dx}{dy} = f(y) \tag{7.2.6}$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$. The desired transformation which takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y(x) = F^{-1}(x) \tag{7.2.7}$$

where F^{-1} is the inverse function to F. Whether (7.2.7) is feasible to implement depends on whether the *inverse function of the integral of $f(y)$* is itself feasible to compute, either analytically or numerically. Sometimes it is, and sometimes it isn't.

Incidentally, (7.2.7) has an immediate geometric interpretation: Since $F(y)$ is the area under the probability curve to the left of y, (7.2.7) is just the prescription: choose a uniform random x, then find the value y that has that fraction x of probability area to its left, and return the value y.

Normal (Gaussian) Deviates

Transformation methods generalize to more than one dimension. If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)\, dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots \tag{7.2.8}$$

where $|\partial(\ )/\partial(\ )|$ is the Jacobian determinant of the x's with respect to the y's (or reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2 \ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2 \ln x_1} \sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi} \arctan \frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right] \left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.2.9).

One further trick is useful in applying (7.2.10). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R \equiv v_1^2 + v_2^2$ is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.2.10) can now be written as $v_1/\sqrt{R}$ and $v_2/\sqrt{R}$, obviating the trigonometric function calls!

We thus have

```
FUNCTION GASDEV(IDUM)
    Returns a normally distributed deviate with zero mean and unit variance, using RAN1(IDUM)
    as the source of uniform deviates.
DATA ISET/0/
IF (ISET.EQ.0) THEN                 We don't have an extra deviate handy, so
    V1=2.*RAN1(IDUM)-1.             pick two uniform numbers in the square extending from -1 to +1 in
    V2=2.*RAN1(IDUM)-1.                 each direction,
    R=V1**2+V2**2                   see if they are in the unit circle,
    IF(R.GE.1.)GO TO 1              and if they are not, try again.
    FAC=SQRT(-2.*LOG(R)/R)          Now make the Box-Muller transformation
    GSET=V1*FAC                     to get two normal deviates. Return one and save the other for next
    GASDEV=V2*FAC                       time.
    ISET=1                          Set flag.
ELSE                                We have an extra deviate handy,
    GASDEV=GSET                     so return it,
    ISET=0                          and unset the flag.
ENDIF
RETURN
END
```

REFERENCES AND FURTHER READING:

Knuth, Donald E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley), pp. 116ff.

7.3 Rejection Method: Gamma, Poisson, Binomial Deviates

The *rejection method* is a powerful, general technique for generating random deviates whose distribution function $p(x)dx$ (probability of a value occurring between x and $x + dx$) is known and computable. The rejection method does *not* require that the cumulative distribution function [indefinite integral of $p(x)$] be readily computible, much less the inverse of that function — which was required for the transformation method in the previous section.

The rejection method is based on a simple geometrical argument:

Draw a graph of the probability distribution $p(x)$ that you wish to generate, so that the area under the curve in any range of x corresponds to the desired probability of generating an x in that range. If we had some way of choosing a random point *in two dimensions*, with uniform probability in the

area under your curve, then the x value of that random point would have the desired distribution.

Now, on the same graph, draw any other curve $f(x)$ which has finite (not infinite) area and lies everywhere *above* your original probability distribution. (This is always possible, because your original curve encloses only unit area, by definition of probability.) We will call this $f(x)$ the *comparison function*. Imagine now that you have some way of choosing a random point in two dimensions that is uniform in the area under the comparison function. Whenever that point lies outside the area under the original probability distribution, we will *reject* it and choose another random point. Whenever it lies inside the area under the original probability distribution, we will *accept* it. It should be obvious that the accepted points are uniform in the accepted area, so that their x values have the desired distribution. It should also be obvious that the fraction of points rejected just depends on the ratio of the area of the comparison function to the area of the probability distribution function, not on the details of shape of either function. For example, a comparison function whose area is less than 2 will reject fewer than half the points, even if it approximates the probability function very badly at some values of x, e.g. remains finite in some region where x is zero.

It remains only to suggest how to choose a uniform random point in two dimensions under the comparison function $f(x)$. A variant of the transformation method (§7.2) does nicely: Be sure to have chosen a comparison function whose indefinite integral is known analytically, and is also analytically invertible to give x as a function of "area under the comparison function to the left of x." Now pick a uniform deviate between 0 and A, where A is the total area under $f(x)$, and use it to get a corresponding x. Then pick a uniform deviate between 0 and $f(x)$ as the y value for the two-dimensional point. You should be able to convince yourself that the point (x, y) is uniformly distributed in the area under the comparison function $f(x)$.

An equivalent procedure is to pick the second uniform deviate between zero and one, and accept or reject according to whether it is respectively less than or greater than the ratio $p(x)/f(x)$.

So, to summarize, the rejection method for some given $p(x)$ requires that one find, once and for all, some reasonably good comparison function $f(x)$. Thereafter, each deviate generated requires two uniform random deviates, one evaluation of f (to get the coordinate y), and one evaluation of p (to decide whether to accept or reject the point x, y). Figure 7.3.1 illustrates the procedure. Then, of course, this procedure must be repeated, on the average, A times before the final deviate is obtained.

Gamma Distribution

The gamma distribution of integer order $a > 0$ is the waiting time to the a^{th} event in a Poisson random process of unit mean. For example, when $a = 1$, it is just the exponential distribution of §7.2, the waiting time to the first event.

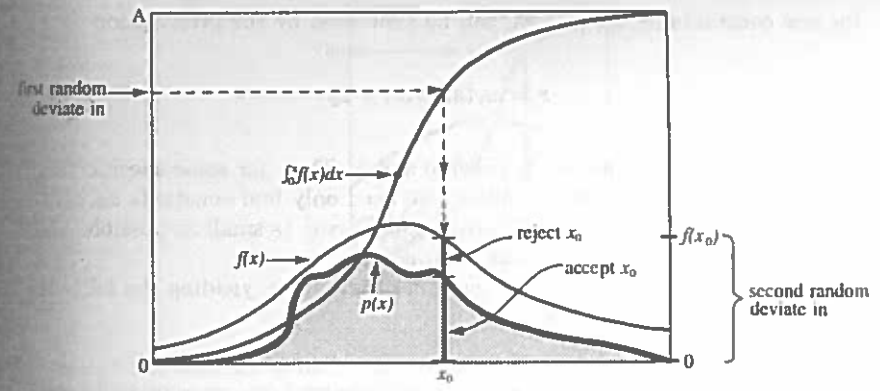

Figure 7.3.1. Rejection method for generating a random deviate x from a known probability distribution $p(x)$ that is everywhere less than some other function $f(x)$. The transformation method is first used to generate a random deviate x of the distribution f (compare Figure 7.2.1). A second uniform deviate is used to decide whether to accept or reject that x. If it is rejected, a new deviate of f is found; and so on. The ratio of accepted to rejected points is the ratio of the area under p to the area between p and f.

A gamma deviate has probability $p_a(x)dx$ of occurring with a value between x and $x + dx$, where

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x > 0 \tag{7.3.1}$$

To generate deviates of (7.3.1) for small values of a, it is best to add up a exponentially distributed waiting times, i.e. logarithms of uniform deviates. Since the sum of logarithms is the logarithm of the product, one really has only to generate the product of a uniform deviates, then take the log.

For larger values of a, the distribution (7.3.1) has a typically "bell-shaped" form, with a peak at $x = a$ and a half-width of about $\sqrt{a}$.

We will be interested in several probability distributions with this same qualitative form. A useful comparison function in such cases is derived from the *Lorentzian distribution*

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

whose inverse indefinite integral is just the tangent function. It follows that the x-coordinate of an area-uniform random point under the comparison function

$$f(x) = \frac{c_0}{1 + (x - x_0)^2/a_0^2} \tag{7.3.3}$$

for any constants a_0, c_0, and x_0, can be generated by the prescription

$$x = a_0 \tan(\pi U) + x_0 \tag{7.3.4}$$

where U is a uniform deviate between 0 and 1. Thus, for some specific "bell-shaped" $p(x)$ probability distribution, we need only find constants a_0, c_0, x_0, with the product $a_0 c_0$ (which determines the area) as small as possible, such that (7.3.3) is everywhere greater than $p(x)$.

Ahrens has done this for the gamma distribution, yielding the following algorithm (as described in Knuth):

```
      FUNCTION GAMDEV(IA,IDUM)
        Returns a deviate distributed as a gamma distribution of integer order IA, i.e. a waiting
        time to the IAth event in a Poisson process of unit mean, using RAN1(IDUM) as the source
        of uniform deviates.
      IF(IA.LT.1)PAUSE
      IF(IA.LT.6)THEN                  Use direct method, adding waiting times.
        X=1.
        DO 11 J=1,IA
          X=X*RAN1(IDUM)
        11 CONTINUE
        X=-LOG(X)
      ELSE                             Use rejection method.
1       V1=2.*RAN1(IDUM)-1.            These four lines generate the tangent of a random angle, i.e. are
        V2=2.*RAN1(IDUM)-1.              equivalent to Y = TAN(3.14159265 * RAN1(IDUM)).
        IF(V1**2+V2**2.GT.1.)GO TO 1
        Y=V2/V1
        AM=IA-1
        S=SQRT(2.*AM+1.)
        X=S*Y+AM                       We decide whether to reject X:
        IF(X.LE.0.)GO TO 1             Reject in region of zero probability.
        E=(1.+Y**2)*EXP(AM*LOG(X/AM)-S*Y)   Ratio of probability fn. to comparison fn.
        IF(RAN1(IDUM).GT.E)GO TO 1     Reject on basis of a second uniform deviate.
      ENDIF
      GAMDEV=X
      RETURN
      END
```

Poisson Deviates

The Poisson distribution is conceptually related to the gamma distribution. It gives the probability of a certain integer number m of unit rate Poisson random events occurring in a given interval of time x, while the gamma distribution was the probability of waiting time between x and $x + dx$ to the m^{th} event. Note that m takes on only integer values ≥ 0, so that the Poisson distribution, viewed as a continuous distribution function $p_x(m)dm$, is zero everywhere except where m is an integer ≥ 0. At such places, it is infinite, such that the integrated probability over a region containing the integer is some finite number. The total probability at an integer j is

$$\text{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

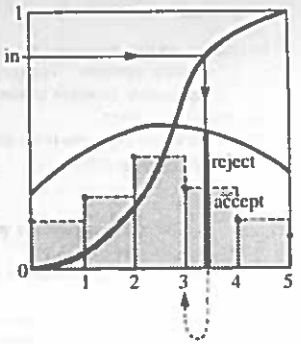

Figure 7.3.2. Rejection method as applied to an integer-valued distribution. The method is performed on the step function shown as a dashed line, yielding a real-valued deviate. This deviate is rounded down to the next lower integer, which is output.

At first sight this might seem an unlikely candidate distribution for the rejection method, since no continuous comparison function can be larger than the infinitely tall, but infinitely narrow, *Dirac delta functions* in $p_x(m)$. However there is a trick that we can do: Spread the finite area in the spike at j uniformly into the interval between j and $j + 1$. This defines a continuous distribution $q_x(m)dm$ given by

$$q_x(m)dm = \frac{x^{[m]} e^{-x}}{[m]!} \tag{7.3.6}$$

where $[m]$ represents the largest integer less than m. If we now use the rejection method to generate a (non-integer) deviate from (7.3.6), and then take the integer part of that deviate, it will be as if drawn from the desired distribution (7.3.5). (See Figure 7.3.2.) This trick is general for any integer-valued probability distribution.

For x large enough, the distribution (7.3.6) is qualitatively bell-shaped (albeit with a bell made out of small, square steps), and we can use the same kind of Lorentzian comparison function as was already used above. For small x, we can generate independent exponential deviates (waiting times between events); when the sum of these first exceeds x, then the number of events which would have occurred in waiting time x becomes known and is one less than the number of terms in the sum.

These ideas produce the following routine:

```
      FUNCTION POIDEV(XM,IDUM)
        Returns as a floating-point number an integer value that is a random deviate drawn from a
        Poisson distribution of mean XM, using RAN1(IDUM) as a source of uniform random deviates.
      PARAMETER (PI=3.141592654)
      DATA OLDM /-1./                  Flag for whether XM has changed since last call.
      IF (XM.LT.12.)THEN               Use direct method.
        IF (XM.NE.OLDM) THEN
          OLDM=XM
          G=EXP(-XM)                   If XM is new, compute the exponential.
        ENDIF
        EM=-1
```

```
      T=1.
2     EM=EM+1.                    Instead of adding exponential deviates it is equivalent to multiply
      T=T*RAN1(IDUM)                uniform deviates.  Then we never actually have to take the
      IF (T.GT.G) GO TO 2           log, merely compare to the pre-computed exponential.
    ELSE                          Use rejection method.
      IF (XM.NE.OLDM) THEN        If XM has changed since the last call, then precompute some functions
        OLDM=XM                       which occur below.
        SQ=SQRT(2.*XM)
        ALXM=ALOG(XM)
        G=XM*ALXM-GAMMLN(XM+1.)       The function GAMMLN is the natural log of the gamma func-
      ENDIF                             tion, as given in §6.2.
1     Y=TAN(PI*RAN1(IDUM))        Y is a deviate from a Lorentzian comparison function.
      EM=SQ*Y+XM                  EM is Y, shifted and scaled.
      IF (EM.LT.0.)  GO TO 1      Reject if in regime of zero probability.
      EM=INT(EM)                  The trick for integer-valued distributions.
      T=0.9*(1.+Y**2)*EXP(EM*ALXM-GAMMLN(EM+1.)-G)     The ratio of the desired distribution to
      IF (RAN1(IDUM).GT.T) GO TO 1    the comparison function; we accept or reject by comparing it
    ENDIF                             to another uniform deviate.  The factor 0.9 is chosen so that
    POIDEV=EM                         T never exceeds 1.
    RETURN
    END
```

Binomial Deviates

If an event occurs with probability q, and we make n trials, then the number of times m that it occurs has the binomial distribution,

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j (1-q)^{n-j} \qquad (7.3.7)$$

The binomial distribution is integer valued, with m taking on possible values from 0 to n. It depends on *two* parameters, n and q, so is correspondingly a bit harder to implement than our previous examples. Nevertheless, the techniques already illustrated are sufficiently powerful to do the job:

```
FUNCTION BNLDEV(PP,N,IDUM)
    Returns as a floating-point number an integer value that is a random deviate drawn from
    a binomial distribution of N trials each of probability PP, using RAN1(IDUM) as a source
    of uniform random deviates.
PARAMETER (PI=3.141592654)
DATA NOLD /-1./, POLD /-1./   Arguments from previous calls.
IF(PP.LE.0.5)THEN             The binomial distribution is invariant under changing PP to 1.-PP, if
    P=PP                          we also change the answer to N minus itself; we'll remember
ELSE                              to do this below.
    P=1.-PP
ENDIF
AM=N*P                        This is the mean of the deviate to be produced.
IF (N.LT.25)THEN              Use the direct method while N is not too large.  This can require up
    BNLDEV=0.                     to 25 calls to RAN1.
    DO 11 J=1,N
      IF(RAN1(IDUM).LT.P)BNLDEV=BNLDEV+1.
    11 CONTINUE
ELSE IF (AM.LT.1.)  THEN      If fewer than one event is expected out of 25 or more trials, then the
    G=EXP(-AM)                    distribution is quite accurately Poisson.  Use direct Poisson
    T=1.                          method.
```



```
    DO 12 J=0,N
      T=T*RAN1(IDUM)
      IF (T.LT.G) GO TO 1
    12 CONTINUE
    J=N
1   BNLDEV=J
  ELSE                            Use the rejection method.
    IF (N.NE.NOLD) THEN           If N has changed, then compute useful quantities.
      EN=N
      OLDG=GAMMLN(EN+1.)
      NOLD=N
    ENDIF
    IF (P.NE.POLD) THEN           If P has changed, then compute useful quantities.
      PC=1.-P
      PLOG=LOG(P)
      PCLOG=LOG(PC)
      POLD=P
    ENDIF
    SQ=SQRT(2.*AM*PC)             The following code should by now seem familiar: rejection method
2   Y=TAN(PI*RAN1(IDUM))              with a Lorentzian comparison function.
    EM=SQ*Y+AM
    IF (EM.LT.0..OR.EM.GE.EN+1.)  GO TO 2      Reject.
    EM=INT(EM)                    Trick for integer-valued distribution.
    T=1.2*SQ*(1.+Y**2)*EXP(OLDG-GAMMLN(EM+1.)
     -GAMMLN(EN-EM+1.)+EM*PLOG+(EN-EM)*PCLOG)
    IF (RAN1(IDUM).GT.T) GO TO 2      Reject.  This happens about 1.5 times per deviate, on av-
    BNLDEV=EM                         erage.
  ENDIF
  IF (P.NE.PP) BNLDEV=N-BNLDEV    Remember to undo the symmetry transformation.
  RETURN
  END
```

REFERENCES AND FURTHER READING:

Knuth, Donald E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley), pp. 120ff.

7.4 Generation of Random Bits

This topic is not very useful for programming in high-level languages, but it can be quite useful when you have access to the machine-language level of a machine or when you are in a position to build special-purpose hardware out of readily available chips.

The problem is how to generate single random bits, with 0 and 1 equally probable. Of course you can just generate uniform random deviates between zero and one and use their first bit (i.e. test if they are greater than or less than 0.5). However this takes a lot of arithmetic; there are special purpose applications, such as real-time signal processing, where you want to generate bits very much faster than that.

One method for generating random bits, with two variant implementations, is based on the theory of "primitive polynomials modulo 2." It is

beyond our scope to discuss this theory. Suffice it to say that there are such things, special polynomials among those whose coefficients are zero or one. An example is

$$x^{18} + x^5 + x^2 + x^1 + x^0 \tag{7.4.1}$$

which we can abbreviate by just writing the nonzero powers of x, e.g.,

$$(18, 5, 2, 1, 0)$$

Every primitive polynomial modulo 2 of order n (=18 above) defines a recurrence relation for obtaining a new random bit from the n preceding ones. The recurrence relation is guaranteed to produce a sequence of maximal length, i.e. cycle through all possible sequences of n bits (except all zeros) before it repeats. Therefore one can seed the sequence with any initial bit pattern (except all zeros), and get $2^n - 1$ random bits before the sequence repeats.

Let the bits be numbered from 1 (most recently generated) through n (generated n steps ago), and denoted $a_1, a_2, \ldots, a_n$. We want to give a formula for a new bit a_0. After generating a_0 we will shift all the bits by one, so that the old a_n is finally lost, and the new a_0 becomes a_1. We then apply the formula again, and so on.

"Method I" is the easiest to implement in hardware, requiring only a single shift register n bits long and a few XOR ("exclusive or" or bit addition mod 2) gates. For the primitive polynomial given above, the recurrence formula is

$$a_0 = a_{18} \,.\mathtt{XOR}.\, a_5 \,.\mathtt{XOR}.\, a_2 \,.\mathtt{XOR}.\, a_1 \tag{7.4.2}$$

The terms that are XOR'd together can be thought of as "taps" on the shift register, XOR'd into the register's input. More generally, there is precisely one term for each nonzero coefficient in the primitive polynomial except the constant (zero bit) term. So the first term will always be a_n for a primitive polynomial of degree n, while the last term might or might not be a_1, depending on whether the primitive polynomial has a term in x^1.

It is rather cumbersome to illustrate the method in FORTRAN. Assume that IAND is a bitwise AND function, NOT is bitwise complement, ISHFT(,1) is leftshift by one bit, IOR is bitwise OR. (These are available, e.g., in VAX-11 FORTRAN.) Then we have:

```
FUNCTION IRBIT1(ISEED)
    Returns as an integer a random bit, based on the 18 low-significance bits in ISEED (which
    is modified for the next call).
LOGICAL NEWBIT                  The accumulated XOR's.
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072)      Powers of 2.
NEWBIT=IAND(ISEED,IB18).NE.0 Get bit 18.
IF(IAND(ISEED,IB5).NE.0)NEWBIT=.NOT.NEWBIT  XOR with bit 5.
IF(IAND(ISEED,IB2).NE.0)NEWBIT=.NOT.NEWBIT  XOR with bit 2.
```


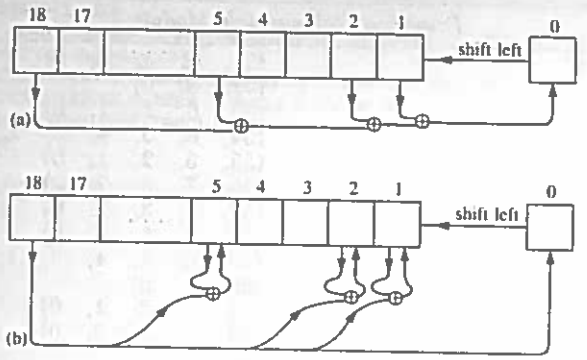

Figure 7.4.1. Two related methods for obtaining random bits from a shift register and a primitive polynomial modulo 2. (a) The contents of selected taps are combined by exclusive-or (addition modulo 2), and the result is shifted in from the right. This method is easiest to implement in hardware. (b) Selected bits are modified by exclusive-or with the leftmost bit, which is then shifted in from the right. This method is easiest to implement in software.

```
IF(IAND(ISEED,IB1).NE.0)NEWBIT=.NOT.NEWBIT  XOR with bit 1.
IRBIT1=0
ISEED=IAND(ISHFT(ISEED,1),NOT(IB1))  Leftshift the seed and put a zero in its bit 1.
IF(NEWBIT)THEN                  But if the XOR calculation gave a 1,
    IRBIT1=1
    ISEED=IOR(ISEED,IB1)        then put that in bit 1 instead.
ENDIF
RETURN
END
```

"Method II" is less suited to direct hardware implementation (though still possible), but is more suited to machine-language implementation. It modifies more than one bit among the saved n bits as each new bit is generated (Figure 7.4.1). It generates the maximal length sequence, but not in the same order as Method I. The prescription for the primitive polynomial (7.4.1) is:

$$\begin{aligned} a_0 &= a_{18} \\ a_5 &= a_5 \,.\mathtt{XOR}.\, a_0 \\ a_2 &= a_2 \,.\mathtt{XOR}.\, a_0 \\ a_1 &= a_1 \,.\mathtt{XOR}.\, a_0 \end{aligned} \tag{7.4.3}$$

In general there will be an XOR to be done for each nonzero term in the primitive polynomial except 0 and n. The nice feature about Method II is that all the XOR's can usually be done as a single masked word XOR (here assumed to be the FORTRAN function IEOR):

Primitive Polynomials Modulo 2

(1, 0)
(2, 1, 0)
(3, 1, 0)
(4, 1, 0)
(5, 2, 0)
(6, 1, 0)
(7, 1, 0)
(8, 4, 3, 2, 0)
(9, 4, 0)
(10, 3, 0)
(11, 2, 0)
(12, 6, 4, 1, 0)
(13, 4, 3, 1, 0)
(14, 5, 3, 1, 0)
(15, 1, 0)
(16, 5, 3, 2, 0)
(17, 3, 0)
(18, 5, 2, 1, 0)
(19, 5, 2, 1, 0)
(20, 3, 0)
(21, 2, 0)
(22, 1, 0)
(23, 5, 0)
(24, 4, 3, 1, 0)
(25, 3, 0)
(26, 6, 2, 1, 0)
(27, 5, 2, 1, 0)
(28, 3, 0)
(29, 2, 0)
(30, 6, 4, 1, 0)
(31, 3, 0)
(32, 7, 5, 3, 2, 1, 0)
(33, 6, 4, 1, 0)
(34, 7, 6, 5, 2, 1, 0)
(35, 2, 0)
(36, 6, 5, 4, 2, 1, 0)
(37, 5, 4, 3, 2, 1, 0)
(38, 6, 5, 1, 0)
(39, 4, 0)
(40, 5, 4 3, 0)
(41, 3, 0)
(42, 5, 4, 3, 2, 1, 0)
(43, 6, 4, 3, 0)
(44, 6, 5, 2, 0)
(45, 4, 3, 1, 0)
(46, 8, 5, 3, 2, 1, 0)
(47, 5, 0)
(48, 7, 5, 4, 2, 1, 0)
(49, 6, 5, 4, 0)
(50, 4, 3, 2, 0)
(51, 6, 3, 1, 0)
(52, 3, 0)
(53, 6, 2, 1, 0)
(54, 6, 5, 4, 3, 2, 0)
(55, 6, 2, 1, 0)
(56, 7, 4, 2, 0)
(57, 5, 3, 2, 0)
(58, 6, 5, 1, 0)
(59, 6, 5, 4, 3, 1, 0)
(60, 1, 0)
(61, 5, 2, 1, 0)
(62, 6, 5, 3, 0)
(63, 1, 0)
(64, 4, 3, 1, 0)
(65, 4, 3, 1, 0)
(66, 8, 6, 5, 3, 2, 0)
(67, 5, 2, 1, 0)
(68, 7, 5, 1, 0)
(69, 6, 5, 2, 0)
(70, 5, 3, 1, 0)
(71, 5, 3, 1, 0)
(72, 6, 4, 3, 2, 1, 0)
(73, 4, 3, 2, 0)
(74, 7, 4, 3, 0)
(75, 6, 3, 1, 0)
(76, 5, 4, 2, 0)
(77, 6, 5, 2, 0)
(78, 7, 2, 1, 0)
(79, 4, 3, 2, 0)
(80, 7, 5, 3, 2, 1, 0)
(81, 4 0)
(82, 8, 7, 6, 4, 1, 0)
(83, 7, 4, 2, 0)
(84, 8, 7, 5, 3, 1, 0)
(85, 8, 2, 1, 0)
(86, 6, 5, 2, 0)
(87, 7, 5, 1, 0)
(88, 8, 5, 4, 3, 1, 0)
(89, 6, 5, 3, 0)
(90, 5, 3, 2, 0)
(91, 7, 6, 5, 3, 2, 0)
(92, 6, 5, 2, 0)
(93, 2, 0)
(94, 6, 5, 1, 0)
(95, 6, 5, 4, 2, 1, 0)
(96, 7, 6, 4, 3, 2, 0)
(97, 6, 0)
(98, 7, 4, 3, 2, 1, 0)
(99, 7, 5, 4, 0)
(100, 8, 7, 2, 0)


```
FUNCTION IRBIT2(ISEED)
    Returns as an integer a random bit, based on the 18 low-significance bits in ISEED (which
    is modified for the next call).
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072,MASK=IB1+IB2+IB5)
IF(IAND(ISEED,IB18).NE.0)THEN     Change all masked bits, shift, and put 1 into bit 1.
    ISEED=IOR(ISHFT(IEOR(ISEED,MASK),1),IB1)
    IRBIT2=1
ELSE                              Shift and put 0 into bit 1.
    ISEED=IAND(ISHFT(ISEED,1),NOT(IB1))
    IRBIT2=0
ENDIF
RETURN
END
```

A word of caution is: Don't use sequential bits from these routines as the bits of a large, supposedly random, integer, or as the bits in the mantissa of a supposedly random floating point number. They are not very random for that purpose; see Knuth. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by ± 1 at a rapid "chip rate," so as to spread its spectrum uniformly (but recoverably) across some desired bandpass, or (ii) Monte Carlo exploration of a binary tree, where decisions as to whether to branch left or right are to be made randomly.

Now we do not want you to go through life thinking that there is something special about the primitive polynomial of degree 18 used in the above examples. (We chose 18 because 2^{18} is small enough for you to verify our claims directly by numerical experiment.) The accompanying table lists some (random) primitive polynomials mod 2, as tabulated by Watson.

REFERENCES AND FURTHER READING:

Knuth, Donald E. 1981, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley), pp. 29ff.

Horowitz, Paul, and Hill, Winfield 1980, *The Art of Electronics* (Cambridge: Cambridge University Press), §§9.34–9.39.

Tausworthe, Robert C. 1965, "Random Numbers Generated by Linear Recurrence Modulo Two," *Mathematics of Computation*, vol. 19, p. 201.

Watson, E.J. 1962, "Primitive Polynomials (Mod 2)," *Mathematics of Computation*, vol. 16, p. 368.

7.5 The Data Encryption Standard

At the time of writing, this section is intended for your cultural edification, not for your day-to-day use, *unless* you know how to build special purpose hardware out of commercially available chips. We expect, however, that the hardware that we describe will be available on many computer systems quite soon.

After reading all the caveats and conditions which accompany the uniform random number generators which were described in §7.1, you might very well ask: "Why doesn't someone just do the job *right*, design a random number generator so good that there are no ifs-ands-or-buts about it?" The answer is that the U.S. Government has done so, but for purposes rather different from those of interest to us in this book.

There is a close connection between random number generation and encryption, the latter being a subject of great interest in the "real world" of governmental and industrial secrecy. Suppose that you have a stream of data ("plaintext") to encrypt. One way to do this, which can be shown to be as mathematically perfect as any other way, is to add the data stream bit-by-bit modulo 2 to a stream of perfectly random numbers. Here, the notion of perfect randomness is a strong one. It requires not only that the random bit stream have no detectable correlations of any sort, but also that it be generated from a sequence so long (or from a universe of sequences so large) that even a very long stretch of its past behavior does not provide enough information, by itself, to predict the future bit stream. This latter condition is required so that the enciphering scheme is resistant to "plaintext attack," where the enemy cryptanalyst has by other means obtained the unciphered data stream corresponding to a part of your encoded message.

Most encryption schemes that are any good are, of course, highly secret. An exception is the *Data Encryption Standard* (DES), issued by the U.S. National Bureau of Standards (NBS). The DES has a somewhat controversial history: Strictly speaking, it derives from IBM's response to a public solicitation put forth by NBS. It has been widely reported, however, that the supersecret National Security Agency (NSA) was intimately involved in the development and acceptance testing of the final algorithm. (The DES document itself uses the guarded language, "NBS, supported by the technical assistance of appropriate Government agencies....") A key controversial question is whether NSA purposely weakened the algorithm, so that it had vulnerabilities significant enough to be exploited by NSA's own multi-billion dollar resources, but not so significant as to be exploitable by anyone else. For our purposes we hardly need know the answer to this: A random number generator whose deviations from randomness can be discerned only by concerted attack with resources comparable to NSA — that random number generator should surely be a contender for the "World's Best" title!

Unfortunately for those of us who work with high-level computer languages, the DES consists almost entirely of bit-level permutations, substitutions, shifts and shufflings. It is designed for hardware implementation on a single large-scale integrated chip; such chips are available as we write, and should soon be cheap.

The basic DES is a substitution cipher which takes a block of 64 bits of input into a unique block of 64 bits of output, under the control of a 64 bit key, which is known only to the persons intended to read the message. (One point of controversy is that only 56 bits in the key are actually used, the remaining 8 being predictable parity checks.) As long as the key is held fixed, the same 64 bits of input will always produce the same 64 bits of output. The mapping of input to output is, of course, one-to-one and invertible, so that the message can be decrypted. Thus, if we imagine looping through all 2^{64} possible input bit configurations, all possible output configurations will also be produced, but in a highly scrambled order. We can therefore generate random numbers by generating a random bit stream by the methods of §7.4 and running that stream through DES.

The routines which follow are illustrative (and work), but are terribly inefficient: They store one bit per FORTRAN integer word, and do all bit manipulations as explicit operations on full words.

Random Number Generator Which Calls DES

Here is a routine which can be substituted for any of the RANs of §7.1. It presumes the existence of a subroutine called DES, whose arguments are 64 bits of input, 64 bits of key, 64 bits of output, an encrypt/decrypt flag (input to the routine), and a flag telling that the key has been changed since the previous call (input to the routine). If you have the DES available on your system as a system call, you should be able to rewrite this routine to take advantage of it (presumably using bits instead of words!).

```
FUNCTION RAN4(IDUM)
  Returns a uniform random deviate between 0.0 and 1.0 using DES. Set IDUM negative to
  initialize. There are IM possible initializations. This routine is extremely slow and should
  be used for demonstration purposes only.
PARAMETER (IM=11979,IA=430,IC=2531,NACC=24)
  The first three parameters are used in initializing, cf. §7.1. NACC is the number of bits
  floating point precision desired on the random deviate.
DIMENSION INP(64),JOT(64),KEY(64),POW(65)
DATA IFF/0/
IF(IDUM.LT.0.OR.IFF.EQ.0)THEN          Initialize:
  IFF=1
  IDUM=MOD(IDUM,IM)
  IF(IDUM.LT.0)IDUM=IDUM+IM
  POW(1)=0.5
  DO 11 J=1,64                 Set both the 64 bits of key and also the starting configuration of the
    IDUM=MOD(IDUM*IA+IC,IM)      64-bit input array.
    KEY(J)=(2*IDUM)/IM           Highest order bit.
    INP(J)=MOD((4*IDUM)/IM,2)    Next highest order bit.
    POW(J+1)=0.5*POW(J)  Inverse powers of 2 in floating point.
  11 CONTINUE
  NEWKEY=1                     Set this flag.
ENDIF
ISAV=INP(64)                   Start here except on initialization.
IF(ISAV.NE.0)THEN              Generate the next input bit configuration, cf. §7.4.
  INP(4)=1-INP(4)              I.e., change 1 to 0 and 0 to 1.
  INP(3)=1-INP(3)
```

```
    INP(1)=1-INP(1)
ENDIF
DO 12 J=64,2,-1
    INP(J)=INP(J-1)
 12 CONTINUE
INP(1)=ISAV                    Input bit configuration now ready.
CALL DES(INP,KEY,NEWKEY,0,JOT)   Here is the real business.
RAN4=0.0                       It remains only to make a floating number out of random bits.
DO 13 J=1,NACC
    IF(JOT(J).NE.0)RAN4=RAN4+POW(J)
 13 CONTINUE
RETURN
END
```

Details of the Data Encryption Standard

Here we give a FORTRAN implementation of the Data Encryption Standard for illustrative purposes, terribly inefficient to use in this high-level form. (Technically this is *not* an implementation of DES since the standard itself states that *no* implementation in software is acceptably secure! We, however, are not concerned with security but only with algorithmics.)

The DES has three distinguishable components. First there is the *key schedule*. This is simply a certain prescription for taking the 56 active bits in the key and shuffling them into 16 different configurations, each 48 bits long. The shuffling procedure is complicated, but straightforward: bits are never combined with other bits, or modified, but only moved around from place to place. In other words, the key schedule makes 16 "sub-master keys" out of a "master key." This need be done only whenever the key is changed, not every time an input block is enciphered.

Second, as the heart and soul of the DES, there is the *cipher function*. This is a fixed, highly non-linear function which combines 32 bits of input (a half-word of the total block being encrypted) with 48 bits of sub-master key to produce 32 bits of highly random output. The procedure in detail is to expand the 32 input bits to 48 bits by a permutation that repeats some bits twice, then to add the expanded input to the 48-bit submaster key (bit by bit modulo 2). Then, most important, the 48 bits are reduced down to 32 bits of output by a table look-up which maps every 6 sequential bits into 4 bits. This table, which is called the *S-box*, is the soul of the algorithm. It is a point of controversy that the theory behind the design of the S-box has never been fully revealed. Finally in the cipher function, a bitwise permutation of the 32 output bits is performed.

It is important to understand that the cipher function is not particularly invertible: to decode a message, it is *not* necessary to recover the input to the cipher function from its output and a knowledge of its key. In fact, to be resistant to plaintext attack, the cipher function should be highly non-invertible. The DES gets its invertibility (which, by the way, we don't need for the purpose of generating random numbers) from its overall encoding strategy, the third component of DES that we now describe:

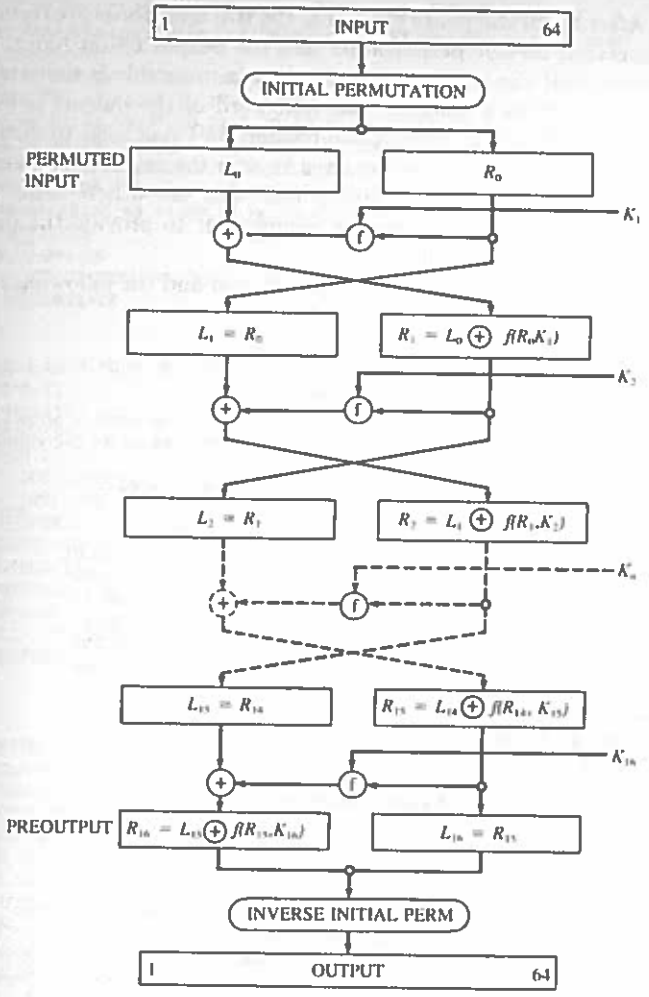

Figure 7.5.1. Block diagram of the Data Encryption Standard showing how the 16 sub-master keys $K_1 \ldots K_{16}$ are combined with 32-bit half-words by means of a highly nonlinear cipher function f.

The original 64 bits of input are put through a fixed initial permutation and then divided into two half-words of 32 bits each. Now 16 stages of the following procedure are performed: One half-word is passed on to the next stage, untouched. That same half-word is also sent, with one of the 16 sub-master keys, to the cipher function, producing a 32 bit "random" configuration. The half-word which has not yet been used is now encrypted by adding it (bitwise modulo 2) to this random configuration and then passed on to the next stage. Figure 7.5.1 should make the process clear.

Between each stage, the half-words are exchanged, so that the one passed on unchanged is not the one that was passed on unchanged in the previous

transition. After all 16 stages are complete, the two half-words are recombined through a specified bitwise permutation into the output 64-bit block.

The reason that the whole DES procedure is invertible is that (after the final, fixed permutation is undone) one half-word of the output is precisely the bit configuration which (with a sub-master key) was used to encode the other half. Thus, by using the sub-master keys in the reverse order, one can "peel away" the 16 stages of encryption one after the other. That is why one half-word is always passed through unchanged: to provide the means of decrypting the other half-word!

Referring to the above description, you should find the following routines comprehensible:

```
      SUBROUTINE DES(INPUT,KEY,NEWKEY,ISW,JOTPUT)
          Data Encryption Standard. Encrypts 64 bits, stored one bit per word, in array INPUT into
          JOTPUT using KEY. Set NEWKEY=1 when the key is new. Set ISW=0 for encryption, =1 for
          decryption. Timing: about 24 ms per call on a VAX11/780.
      DIMENSION INPUT(64),KEY(64),JOTPUT(64),ITMP(64),IP(64),IPM(64)
     *  ,ICF(32),KNS(48,16)
      DATA IP/58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,62,54,46
     *  ,38,30,22,14,6,64,56,48,40,32,24,16,8,57,49,41,33,25,17,9,1,59,51
     *  ,43,35,27,19,11,3,61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7/
      DATA IPM/40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,38,6,46,14
     *  ,54,22,62,30,37,5,45,13,53,21,61,29,36,4,44,12,52,20,60,28,35,3
     *  ,43,11,51,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25/
      IF(NEWKEY.NE.0)THEN              Get the 16 sub-master keys from the master key.
        NEWKEY=0
        DO 11 I=1,16
          CALL KS(KEY,I,KNS(1,I))
   11   CONTINUE
      ENDIF
      DO 12 J=1,64                     The initial permutation.
        ITMP(J)=INPUT(IP(J))
   12 CONTINUE
      DO 14 I=1,16                     The 16 stages of encryption.
        II=I
        IF(ISW.EQ.1)II=17-I            Use the sub-master keys in reverse order for decryption.
        CALL CYFUN(ITMP(33),KNS(1,II),ICF)   Get cipher function.
        DO 13 J=1,32                   Pass one half-word through unchanged, while encrypting the other
          IC=ICF(J)+ITMP(J)                  half-word and exchanging the two half-words output.
          ITMP(J)=ITMP(J+32)
          ITMP(J+32)=IAND(IC,1)
        If you don't have the IAND function, instead use:
C         ITMP(J+32)=MOD(MOD(IC,2)+2,2)
   13   CONTINUE
   14 CONTINUE                         Done with the 16 stages.
      DO 15 J=1,32                     A final exchange of the two half-words is required.
        IC=ITMP(J)
        ITMP(J)=ITMP(J+32)
        ITMP(J+32)=IC
   15 CONTINUE
      DO 16 J=1,64                     Final output permutation.
        JOTPUT(J)=ITMP(IPM(J))
   16 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE KS(KEY,N,KN)
          Key schedule calculation, returns KN given KEY and N=1,2,...,16, must be called with N
          in that order.
      DIMENSION KEY(64),KN(48),ICD(56),IPC1(56),IPC2(48)
      DATA IPC1/57,49,41,33,25,17,9,1,58,50,42,34,26,18
     *  ,10,2,59,51,43,35,27,19,11,3,60,52,44,36,63,55,47,39,31,23,15
     *  ,7,62,54,46,38,30,22,14,6,61,53,45,37,29,21,13,5,28,20,12,4/
      DATA IPC2/14,17,11,24,1,5,3,28,15,6,21,10
     *  ,23,19,12,4,26,8,16,7,27,20,13,2,41,52,31,37,47,55
     *  ,30,40,51,45,33,48,44,49,39,56,34,53,46,42,50,36,29,32/
      IF(N.EQ.1)THEN                   Initial selection and permutation.
        DO 11 J=1,56
          ICD(J)=KEY(IPC1(J))
   11   CONTINUE
      ENDIF
      IT=2                             For most values of N perform two shifts,
      IF(N.EQ.1.OR.N.EQ.2.OR.N.EQ.9.OR.N.EQ.16)IT=1    but for these perform only one.
      DO 13 I=1,IT                     Circular left-shifts of the two halves of the array ICD.
        IC=ICD(1)
        ID=ICD(29)
        DO 12 J=1,27
          ICD(J)=ICD(J+1)
          ICD(J+28)=ICD(J+29)
   12   CONTINUE
        ICD(28)=IC
        ICD(56)=ID
   13 CONTINUE                         Done with the shifts.
      DO 14 J=1,48                     The sub-master key is a selection of bits from the shifted ICD.
        KN(J)=ICD(IPC2(J))
   14 CONTINUE
      RETURN
      END

      SUBROUTINE CYFUN(IR,K,IOUT)
          Returns the cipher function of IR and K in IOUT.
      DIMENSION IR(32),K(48),IOUT(32),IE(48),IET(48),IP(32)
     *  ,ITMP(32),IS(16,4,8),IBIN(4,16)
      DATA IET/32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13
     *  ,14,15,16,17,16,17,18,19,20,21,20,21,22,23,24
     *  ,25,24,25,26,27,28,29,28,29,30,31,32,1/
      DATA IP/16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10
     *  ,2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25/
          Here follows the S-Box, in full glory. Alternatively, you might read IS from a data file.
      DATA IS/14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7
     *  ,0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8
     *  ,4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0
     *  ,15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13
     *  ,15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10
     *  ,3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5
     *  ,0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15
     *  ,13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9
     *  ,10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8
     *  ,13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1
     *  ,13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7
     *  ,1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12
     *  ,7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15
     *  ,13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9
     *  ,10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4
     *  ,3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14
     *  ,2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9
     *  ,14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6
     *  ,4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14
     *  ,11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3
     *  ,12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11
```

```
     *    ,10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8
     *    ,9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6
     *    ,4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13
     *    ,4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1
     *    ,13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6
     *    ,1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2
     *    ,6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12
     *    ,13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7
     *    ,1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2
     *    ,7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8
     *    ,2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11/
          Next follows the table of bits in the integers 0 to 15:
      DATA IBIN/0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1
     *    ,0,1,0,0,0,1,0,1,0,1,1,0,0,0,1,1,1,1,0,0,0,1,0,0,1
     *    ,1,0,1,0,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,0,1,1,1,1/
      DO 15 J=1,48                    Expand IR to 48 bits and combine it with K.
        IE(J)=IAND(IR(IET(J))+K(J),1)
        If you don't have the IAND function, instead use:
C       IE(J)=MOD(MOD(IR(IET(J))+K(J),2)+2,2)
   15 CONTINUE
      DO 17 JJ=1,8                    Loop over 8 groups of 6 bits.
        J=6*JJ-5
        IROW=IOR(IE(J+5),ISHFT(IE(J),1))      Find place in the S-box table.
        ICOL=IOR(IE(J+4),ISHFT(IOR(IE(J+3),ISHFT(IOR(IE(J+2),
     *       ISHFT(IE(J+1),1)),1)),1))
        If you don't have the above bit functions, instead use:
C         IROW=2*IE(J)+IE(J+5)
C         ICOL=8*IE(J+1)+4*IE(J+2)+2*IE(J+3)+IE(J+4)
        ISS=IS(ICOL+1,IROW+1,JJ)      Look up the number in the S-box table
        KK=4*(JJ-1)
        DO 10 KI=1,4                  and plug its bits into the output.
          ITMP(KK+KI)=IBIN(KI,ISS+1)
   10   CONTINUE
   17 CONTINUE
      DO 18 J=1,32                    Final permutation.
        IOUT(J)=ITMP(IP(J))
   18 CONTINUE
      RETURN
      END
```


REFERENCES AND FURTHER READING:

Data Encryption Standard, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards).

Guidelines for Implementing and Using the NBS Data Encryption Standard, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards).

Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard, 1980, NBS Special Publication 500-20 (Washington: U.S. Department of Commerce, National Bureau of Standards).

7.6 Monte Carlo Integration

Suppose that we pick N points, uniformly randomly in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f \, dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{7.6.1}$$

Here the angle brackets denote taking the arithmetic mean over the N sample points,

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \tag{7.6.2}$$

The "plus-or-minus" term in (7.6.1) is a one standard deviation error estimate for the integral, not a rigorous bound; further, there is no guarantee that the error is distributed as a Gaussian, so the error term should be taken only as a rough indication of probable error.

Suppose that you want to integrate a function g over a region W which is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V which *includes* W and which *can* easily be sampled (Figure 7.6.1), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

It is not feasible to give a general purpose routine for Monte Carlo integration, but a worked example ought to show you how it is done. Suppose that we want to find the weight and the position of the center of mass of an object of complicated shape, namely the intersection of a torus with the edge of a large box. In particular let the object be defined by the three simultaneous conditions

$$z^2 + \left(\sqrt{x^2 + y^2} - 3\right)^2 \le 1 \tag{7.6.3}$$

(torus centered on the origin with major radius =4, minor radius =2,)

$$x \ge 1 \qquad y \ge -3 \tag{7.6.4}$$

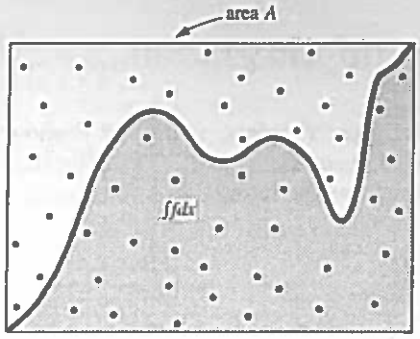

Figure 7.6.1. Monte Carlo integration. Random points are chosen within the area A. The integral of the function f is estimated as the area of A multiplied by the fraction of random points that fall below the curve f. Refinements on this procedure can improve the accuracy of the method; see text.

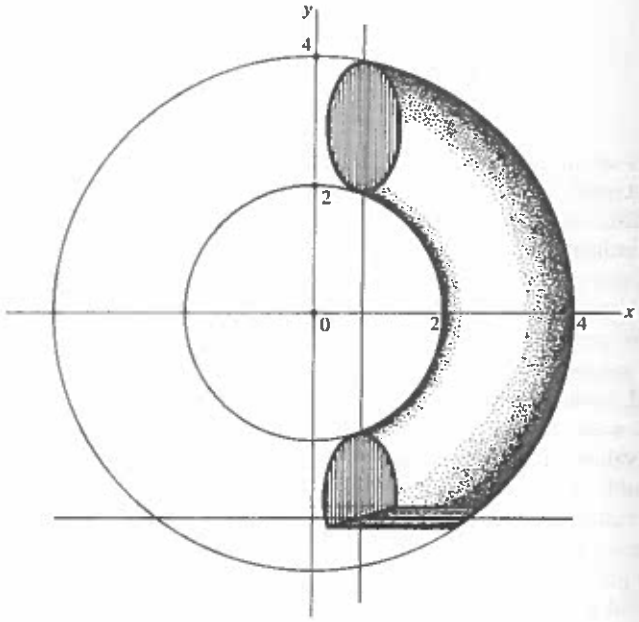

Figure 7.6.2. Example of Monte Carlo integration (see text). The region of interest is a piece of a torus, bounded by the intersection of two planes. The limits of integration of the region cannot easily be written in analytically closed form, so Monte Carlo is a useful technique.

(two faces of the box, see Figure 7.6.2). Suppose for the moment that the object has a constant density ρ.

We want to estimate the following integrals over the interior of the complicated object:

$$\int \rho\, dxdydz \qquad \int x\rho\, dxdydz \qquad \int y\rho\, dxdydz \qquad \int z\rho\, dxdydz \tag{7.6.5}$$

The coordinates of the center of mass will be the ratio of the latter three integrals (linear moments) to the first one (the weight).

In the following fragment, the region V, enclosing the piece-of-torus W, is the rectangular box extending from 1 to 4 in x, -3 to 4 in y, and -1 to 1 in z.

```
N=                                  Set to the number of sample points desired.
DEN=                                Set to the constant value of the density
SW=0.                               Zero the various sums to be accumulated.
SWX=0.
SWY=0.
SWZ=0.
VARW=0.
VARX=0.
VARY=0.
VARZ=0.
VOL=3.*7.*2.                        Volume of the sampled region.
DO 11 J=1,N
    X=1.+3.*RAN2(IDUM)              Pick a point randomly in the sampled region.
    Y=-3.+7.*RAN2(IDUM)
    Z=-1.+2.*RAN2(IDUM)
    IF (Z**2+(SQRT(X**2+Y**2)-3.)**2.LE.1.)THEN Is it in the torus?
        SW=SW+DEN                   If so, add to the various cumulants.
        SWX=SWX+X*DEN
        SWY=SWY+Y*DEN
        SWZ=SWZ+Z*DEN
        VARW=VARW+DEN**2
        VARX=VARX+(X*DEN)**2
        VARY=VARY+(Y*DEN)**2
        VARZ=VARZ+(Z*DEN)**2
    ENDIF
11  CONTINUE
W=VOL*SW/N                          The values of the integrals (7.6.5),
X=VOL*SWX/N
Y=VOL*SWY/N
Z=VOL*SWZ/N
DW=VOL*SQRT((VARW/N-(SW/N)**2)/N)   and their corresponding error estimates.
DX=VOL*SQRT((VARX/N-(SWX/N)**2)/N)
DY=VOL*SQRT((VARY/N-(SWY/N)**2)/N)
DZ=VOL*SQRT((VARZ/N-(SWZ/N)**2)/N)
```

Next, suppose that we want to evaluate the same integrals, but for a piece-of-torus whose density is a strong function of z, in fact varying according to

$$\rho(x, y, z) = e^{5z} \tag{7.6.6}$$

One way to do this is to put the statement

```
DEN=EXP(5.*Z)
```

inside the IF...THEN block, just before DEN is first used. This will work, but it is not the optimal way to proceed. Since (7.6.6) falls so rapidly to zero as z decreases (down to its lower limit -1), most sampled points contribute almost nothing to the sum of the weight or moments. These points are effectively wasted, almost as badly as those that fall outside of the region W. The right

way to do the problem is to make use of a change of variable, exactly as was done in developing the transformation methods of §7.2. Let

$$ds = e^{5z}dz \qquad \text{so that} \qquad s = \frac{1}{5}e^{5z}, \quad z = \frac{1}{5}\ln(5s) \tag{7.6.7}$$

Then $\rho dz = ds$, and the limits $-1 < z < 1$ become $.00135 < s < 29.682$. The program fragment now looks like this

```
N=                              Set to the number of sample points desired.
SW=0.
SWX=0.
SWY=0.
SWZ=0.
VARW=0.
VARX=0.
VARY=0.
VARZ=0.
SS=(0.2*(EXP(5.)-EXP(-5.)))    Interval of S to be random sampled.
VOL=3.*7.*SS                    Volume in X,Y,S-space.
DO 11 J=1,N
    X=1.+3.*RAN2(IDUM)
    Y=-3.+7.*RAN2(IDUM)
    S=.00136+SS*RAN2(IDUM)      Pick a point in S.
    Z=0.2*LOG(5.*S)             Equation (7.6.7).
    IF (Z**2+(SQRT(X**2+Y**2)-3.)**2.LT.1.)THEN
        SW=SW+1.                Density is 1, since absorbed into definition of S.
        SWX=SWX+X
        SWY=SWY+Y
        SWZ=SWZ+Z
        VARW=VARW+1.
        VARX=VARX+X**2
        VARY=VARY+Y**2
        VARZ=VARZ+Z**2
    ENDIF
 11 CONTINUE
W=VOL*SW/N                      The values of the integrals (7.6.5),
X=VOL*SWX/N
Y=VOL*SWY/N
Z=VOL*SWZ/N
DW=VOL*SQRT((VARW/N-(SW/N)**2)/N)       and their corresponding error estimates.
DX=VOL*SQRT((VARX/N-(SWX/N)**2)/N)
DY=VOL*SQRT((VARY/N-(SWY/N)**2)/N)
DZ=VOL*SQRT((VARZ/N-(SWZ/N)**2)/N)
```

If you think for a minute, you will realize that equation (7.6.7) was useful only because the part of the integrand that we wanted to eliminate (e^{5z}) was both integrable analytically, and had an integral that could be analytically inverted. (Compare §7.2!) In general these properties will not hold. Question: What then? Answer: Pull out of the integrand the "best" factor that *can* be integrated and inverted. The criterion for "best" is to try to reduce the remaining integrand to a function that is as close as possible to constant.

The limiting case is instructive: If you manage to make the integrand f *exactly* constant, and if the region V, of known volume, *exactly* encloses the desired region W, then the average of f that you compute will be exactly its constant value, and the error estimate in equation (7.6.1) will exactly vanish.

You will, in fact, have done the integral exactly, and the Monte Carlo numerical evaluations are superfluous. So, backing off from the extreme limiting case, *to the extent* that you are able to make f approximately constant by change of variable, and *to the extent* that you can sample a region only slightly larger than W, you will increase the accuracy of the Monte Carlo integral. This technique is generically called *reduction of variance* in the literature.

The fundamental disadvantage of Monte Carlo integration is just that its accuracy increases only as the square root of N, the number of sampled points. If your accuracy requirements are modest, or if your computer budget is large, then the technique is highly recommended as one of great generality.

REFERENCES AND FURTHER READING:

Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Shreider, Yu. A., ed. 1966, *The Monte Carlo Method* (Oxford: Pergamon).