

```

FUNCTION irbit2(iseed)
INTEGER irbit2,iseed,IB1,IB2,IB5,IB18,MASK
PARAMETER (IB1=1,IB2=2,IB5=16,IB18=131072,MASK=IB1+IB2+IB5)
  Returns as an integer a random bit, based on the 18 low-significance bits in iseed (which
  is modified for the next call).
  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 [1]. Examples of acceptable uses of these random bits are: (i) multiplying a signal randomly by  $\pm 1$  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 [2] lists one primitive polynomial for each degree up to 100. (In fact there exist many such for each degree. For example, see §7.7 for a complete table up to degree 10.)

#### 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), pp. 29ff. [1]
- Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32–9.37.
- Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.
- Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

## 7.5 Random Sequences Based on Data Encryption

In *Numerical Recipes*' first edition, we described how to use the Data Encryption Standard (DES) [1-3] for the generation of random numbers. Unfortunately, when implemented in software in a high-level language like FORTRAN, DES is very slow, so excruciatingly slow, in fact, that our previous implementation can be viewed as more mischievous than useful. Here we give a much faster and simpler algorithm which, though it may not be secure in the cryptographic sense, generates about equally good random numbers.

DES, like its progenitor cryptographic system LUCIFER, is a so-called "block product cipher" [4]. It acts on 64 bits of input by iteratively applying (16 times, in fact) a kind of highly

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)  
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.  
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to [trade@cup.cam.ac.uk](mailto:trade@cup.cam.ac.uk) (outside North America).

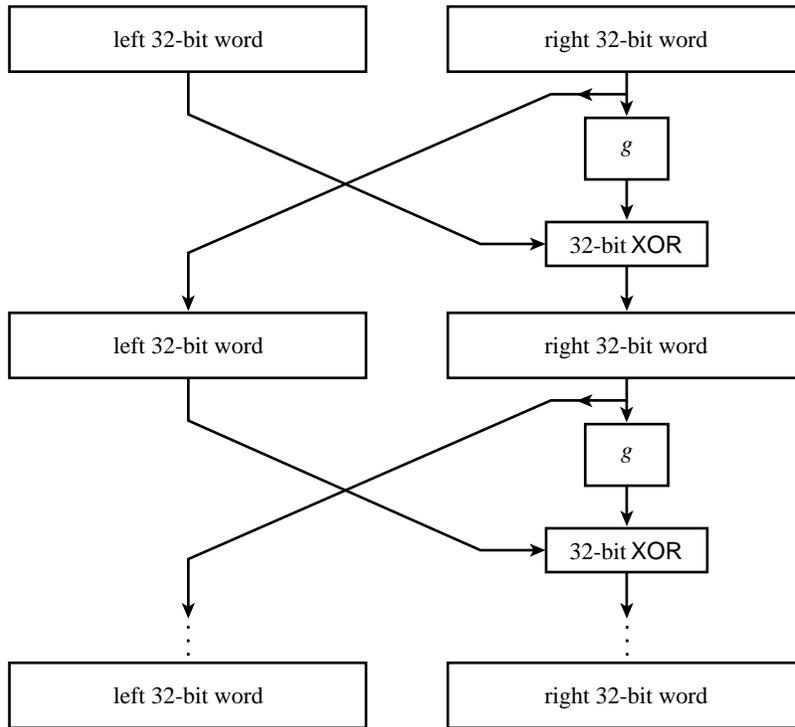


Figure 7.5.1. The Data Encryption Standard (DES) iterates a nonlinear function  $g$  on two 32-bit words, in the manner shown here (after Meyer and Matyas [4]).

nonlinear bit-mixing function. Figure 7.5.1 shows the flow of information in DES during this mixing. The function  $g$ , which takes 32-bits into 32-bits, is called the “cipher function.” Meyer and Matyas [4] discuss the importance of the cipher function being nonlinear, as well as other design criteria.

DES constructs its cipher function  $g$  from an intricate set of bit permutations and table lookups acting on short sequences of consecutive bits. Apparently, this function was chosen to be particularly strong cryptographically (or conceivably as some critics contend, to have an exquisitely subtle cryptographic flaw!). For our purposes, a different function  $g$  that can be rapidly computed in a high-level computer language is preferable. Such a function may weaken the algorithm cryptographically. Our purposes are not, however, cryptographic: We want to find the fastest  $g$ , and smallest number of iterations of the mixing procedure in Figure 7.5.1, such that our output random sequence passes the standard tests that are customarily applied to random number generators. The resulting algorithm will not be DES, but rather a kind of “pseudo-DES,” better suited to the purpose at hand.

Following the criterion, mentioned above, that  $g$  should be nonlinear, we must give the integer multiply operation a prominent place in  $g$ . Because 64-bit registers are not generally accessible in high-level languages, we must confine ourselves to multiplying 16-bit operands into a 32-bit result. So, the general idea of  $g$ , almost forced, is to calculate the three distinct 32-bit products of the high and low 16-bit input half-words, and then to combine these, and perhaps additional fixed constants, by fast operations (e.g., add or exclusive-or) into a single 32-bit result.

There are only a limited number of ways of effecting this general scheme, allowing systematic exploration of the alternatives. Experimentation, and tests of the randomness of the output, lead to the sequence of operations shown in Figure 7.5.2. The few new elements in the figure need explanation: The values  $C_1$  and  $C_2$  are fixed constants, chosen randomly with the constraint that they have exactly 16 1-bits and 16 0-bits; combining these constants

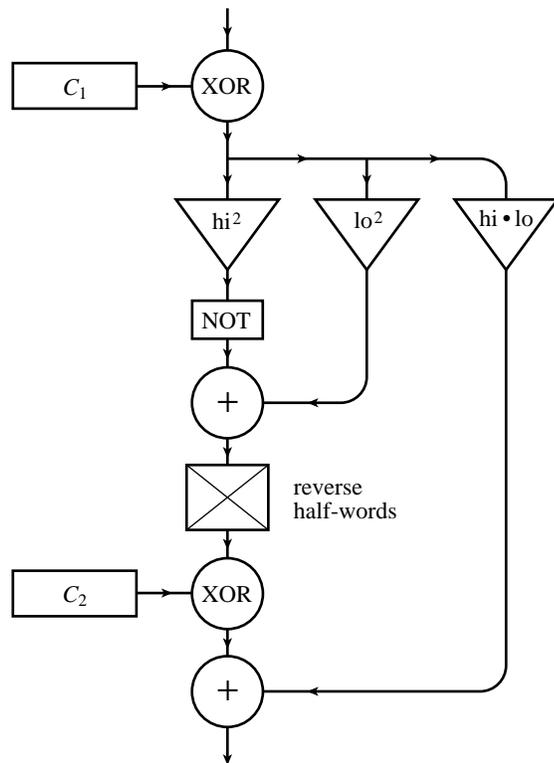


Figure 7.5.2. The nonlinear function  $g$  used by the routine `psdes`.

via exclusive-or ensures that the overall  $g$  has no bias towards 0 or 1 bits.

The “reverse half-words” operation in Figure 7.5.2 turns out to be essential; otherwise, the very lowest and very highest bits are not properly mixed by the three multiplications. The nonobvious choices in  $g$  are therefore: where along the vertical “pipeline” to do the reverse; in what order to combine the three products and  $C_2$ ; and with which operation (add or exclusive-or) should each combining be done? We tested these choices exhaustively before settling on the algorithm shown in the figure.

It remains to determine the smallest number of iterations  $N_{it}$  that we can get away with. The minimum meaningful  $N_{it}$  is evidently two, since a single iteration simply moves one 32-bit word without altering it. One can use the constants  $C_1$  and  $C_2$  to help determine an appropriate  $N_{it}$ : When  $N_{it} = 2$  and  $C_1 = C_2 = 0$  (an intentionally very poor choice), the generator fails several tests of randomness by easily measurable, though not overwhelming, amounts. When  $N_{it} = 4$ , on the other hand, or with  $N_{it} = 2$  but with the constants  $C_1, C_2$  nonsparse, we have been unable to find *any* statistical deviation from randomness in sequences of up to  $10^9$  floating numbers  $r_i$  derived from this scheme. The combined strength of  $N_{it} = 4$  and nonsparse  $C_1, C_2$  should therefore give sequences that are random to tests even far beyond those that we have actually tried. These are our recommended conservative parameter values, notwithstanding the fact that  $N_{it} = 2$  (which is, of course, twice as fast) has no nonrandomness discernible (by us).

We turn now to implementation. The nonlinear function shown in Figure 7.5.2 is not implementable in strictly portable FORTRAN, for at least three reasons: (1) The addition of two 32-bit integers may overflow, and the multiplication of two 16-bit integers may not produce the correct 32-bit product because of sign-bit conventions. We intend that the overflow be ignored, and that the 16-bit integers be multiplied as if they are positive. It is possible to force this behavior on most machines. (2) We assume 32-bit integers; however, there

is no reason to believe that *longer* integers would be in any way inferior (with suitable extensions of the constants  $C_1, C_2$ ). (3) Your compiler may require a different notation for hex constants (see below).

We have been able to run the following routine, `psdes`, successfully on machines ranging from PCs to VAXes and both “big-endian” and “little-endian” UNIX workstations. (Big- and little-endian refer to the order in which the bytes are stored in a word.) A strictly portable implementation is possible in C. If all else fails, you can make a FORTRAN-callable version of the C routine, found in *Numerical Recipes in C*.

```

SUBROUTINE psdes(lword,irword)
INTEGER irword,lword,NITER
PARAMETER (NITER=4)
    "Pseudo-DES" hashing of the 64-bit word (lword,irword). Both 32-bit arguments are
    returned hashed on all bits. NOTE: This routine assumes that arbitrary 32-bit integers can
    be added without overflow. To accomplish this, you may need to compile with a special
    directive (e.g., /check=nooverflow for VMS). In other languages, such as C, one can
    instead type the integers as "unsigned."
INTEGER i,ia,ib,iswap,itmph,itmpl,c1(4),c2(4)
SAVE c1,c2
DATA c1 /Z'BAA96887',Z'1E17D32C',Z'03BCDC3C',      Your compiler may use a differ-
      Z'0F33D1B2'/, c2 /Z'4B0F3B58',Z'E874F0C3',    ent notation for hex constants!
*      Z'6955C5A6', Z'55A7CA46'/
*
do 11 i=1,NITER      Perform niter iterations of DES logic, using a simpler (non-
iswap=irword        cryptographic) nonlinear function instead of DES's.
ia=ieor(irword,c1(i)) The bit-rich constants c1 and (below) c2 guarantee lots of
itmpl=iand(ia,65535)  nonlinear mixing.
itmph=iand(ishft(ia,-16),65535)
ib=itmpl**2+not(itmph**2)
ia=ior(ishft(ib,16),iand(ishft(ib,-16),65535))
irword=ieor(lword,ieor(c2(i),ia)+itmpl*itmph)
lword=iswap
enddo 11
return
END

```

The routine `ran4`, listed below, uses `psdes` to generate uniform random deviates. We adopt the convention that a negative value of the argument `idum` sets the left 32-bit word, while a positive value  $i$  sets the right 32-bit word, returns the  $i$ th random deviate, and increments `idum` to  $i + 1$ . This is no more than a convenient way of defining many different sequences (negative values of `idum`), but still with random access to each sequence (positive values of `idum`). For getting a floating-point number from the 32-bit integer, we like to do it by the masking trick described at the end of §7.1, above. 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. Note that your compiler may require a different notation for hex constants, e.g., `x'3f800000'`, `'3F800000'X`, or even `16#3F800000`. For greater portability, you can instead construct a floating number by making the (signed) 32-bit integer nonnegative (typically, you add exactly  $2^{31}$  if it is negative) and then multiplying it by a floating constant (typically  $2^{-31}$ ).

An interesting, and sometimes useful, feature of the routine `ran4`, below, is that it allows random access to the  $n$ th random value in a sequence, without the necessity of first generating values  $1 \cdots n - 1$ . This property is shared by any random number generator based on *hashing* (the technique of mapping data keys, which may be highly clustered in value, approximately uniformly into a storage address space) [5,6]. One might have a simulation problem in which some certain rare situation becomes recognizable by its consequences only considerably after it has occurred. One may wish to restart the simulation back at that occurrence, using identical random values but, say, varying some other control parameters. The relevant question might then be something like “what random numbers were used in cycle number 337098901?” It might already be cycle number 395100273 before the question comes up. Random generators based on recursion, rather than hashing, cannot easily answer such a question.

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)  
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.  
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-  
 readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs  
 visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to [trade@cup.cam.ac.uk](mailto:trade@cup.cam.ac.uk) (outside North America).

Values for Verifying the Implementation of <code>psdes</code>						
idum	before <code>psdes</code> call		after <code>psdes</code> call (hex)		<code>ran4(idum)</code>	
	lword	irword	lword	irword	VAX	PC
-1	1	1	604D1DCE	509C0C23	0.275898	0.219120
99	1	99	D97F8571	A66CB41A	0.208204	0.849246
-99	99	1	7822309D	64300984	0.034307	0.375290
99	99	99	D7F376F0	59BA89EB	0.838676	0.457334

Successive calls to `psdes` with arguments `-1`, `99`, `-99`, and `1`, should produce exactly the `lword` and `irword` values shown. Masking conversion to a returned floating random value is allowed to be machine dependent; values for VAX and PC are shown.

```
FUNCTION ran4(idum)
  INTEGER idum
  REAL ran4
```

C *USES psdes*

Returns a uniform random deviate in the range 0.0 to 1.0, generated by pseudo-DES (DES-like) hashing of the 64-bit word (`idums`, `idum`), where `idums` was set by a previous call with negative `idum`. Also increments `idum`. Routine can be used to generate a random sequence by successive calls, leaving `idum` unaltered between calls; or it can randomly access the  $n$ th deviate in a sequence by calling with `idum = n`. Different sequences are initialized by calls with differing negative values of `idum`.

```
INTEGER idums,irword,itemp,jflmsk,jflone,lword
```

```
REAL ftemp
```

```
EQUIVALENCE (itemp,ftemp)
```

```
SAVE idums,jflone,jflmsk
```

```
DATA idums /0/, jflone /Z'3F800000'/, jflmsk /Z'007FFFFFFF'/
```

The hexadecimal constants `jflone` and `jflmsk` are used to produce a floating number between 1. and 2. by bitwise masking. They are machine-dependent. See text.

```
if(idum.lt.0)then                                Reset idums and prepare to return the first deviate in its sequence.
  idums=-idum
  idum=1
```

```
endif
```

```
irword=idum
```

```
lword=idums
```

```
call psdes(lword,irword)                        "Pseudo-DES" encode the words.
```

```
itemp=iior(jflone,iand(jflmsk,irword))        Mask to a floating number between 1 and 2.
```

```
ran4=ftemp-1.0                                  Subtraction moves range to 0. to 1.
```

```
idum=idum+1
```

```
return
```

```
END
```

The accompanying table gives data for verifying that `ran4` and `psdes` work correctly on your machine. We do not advise the use of `ran4` unless you are able to reproduce the hex values shown. Typically, `ran4` is about 4 times slower than `ran0` (§7.1), or about 3 times slower than `ran1`.

#### CITED 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). [1]

*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). [2]

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)  
 Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.  
 Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one), to any server computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs visit website <http://www.nr.com> or call 1-800-872-7423 (North America only), or send email to [trade@cup.cam.ac.uk](mailto:trade@cup.cam.ac.uk) (outside North America).

- 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). [3]
- Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [4]
- Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6. [5]
- Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press). [6]

## 7.6 Simple Monte Carlo Integration

Inspirations for numerical methods can spring from unlikely sources. “Splines” first were flexible strips of wood used by draftsmen. “Simulated annealing” (we shall see in §10.9) is rooted in a thermodynamic analogy. And who does not feel at least a faint echo of glamor in the name “Monte Carlo method”?

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

$$\int f dV \approx V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \quad (7.6.1)$$

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

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

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  $W$  that is not easy to sample randomly. For example,  $W$  might have a very complicated shape. No problem. Just find a region  $V$  that *includes*  $W$  and that *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.

General purpose routines for Monte Carlo integration are quite complicated (see §7.8), but a worked example will show the underlying simplicity of the method. Suppose that we want to find the weight and the position of the center of mass of an