Computer Physics Communications 74 (1993) 327–334
North-Holland

Computer Physics Communications

Generation of Gaussian distributed random numbers by using a numerical inversion method

Raúl Toral [a] and Amitabha Chakrabarti [a,b]

[a] *Institut d'Estudis Avançats and Departament de Física, Consejo Superior de Investigaciones Científicas, and Universitat de les Illes Balears, 07071 Palma de Mallorca, Spain*
[b] *Physics Department, Kansas State University, Manhattan, KS 66506, USA* *

Received 7 November 1991; in revised form 7 September 1992

We describe a vectorizable implementation of a numerical inversion method to generate approximately Gaussian distributed random numbers. This method is, on the CRAY-YMP computer, several times faster than the standard Box–Muller–Wiener algorithm. The validity of the approximation is discussed.

1. Introduction

In many computer simulations one needs to generate a large number of Gaussian distributed random numbers. Actually, it is not uncommon to find computer simulations of Monte Carlo methods or numerical solutions of stochastic differential equations in which more time is spent in the generation of the Gaussian distributed random numbers than in the rest of the algorithm. (An example is given in the extensive numerical study of a partial differential equation describing spinodal decomposition by Gawlinski, Vinals and Gunton [1].)

The generation of Gaussian random number is usually done by the Box–Muller–Wiener (BMW) algorithm [2]: if ξ_1 and ξ_2 are two random numbers uniformly distributed in the interval [0,1], then the numbers x_1 and x_2 given by

$$x_1 = \sqrt{-2\log(\xi_1)}\ \sin(2\pi\xi_2),$$
$$x_2 = \sqrt{-2\log(\xi_1)}\ \cos(2\pi\xi_2), \qquad (1)$$

are random numbers distributed according to a Gaussian distribution of mean 0 and variance 1. Random numbers z of mean μ and variance σ^2 are obtained by the linear formation $z = \mu + \sigma x$.

The main advantage of the BMW algorithm is that it is exact producing an unbiased Gaussian distribution. Furthermore, it is easily implementable on a vector computer. A serious disadvantage is that it requires the calculation of a sine, a cosine, a square root and a logarithm function, resulting in a rather slow algorithm. One can avoid computing the sine and the cosine functions by using a rejection technique [3]. This produces an algorithm which is about 20 percent faster than the pure BMW on scalar computers, but it has the disadvantage that it is not efficiently vectorizable. Other algorithms have been proposed [4] but they all share the unwanted feature of not being efficiently vectorizable. Amongst the most efficient of all the proposed methods, we will mention the approximate method of adding 12 random numbers uniformly distributed in the interval $(-0.5, 0.5)$ (we will call this method the G12 generator), and the ratio method of Kinderman and Monahan (see ref. [3]), which generates a random vector (u, v) uniformly distributed in the region defined by: $0 < u \leq 1$, $-2u\sqrt{-\ln(u)} \leq v \leq 2u\sqrt{-\ln(u)}$, the

Correspondence to: R. Toral, Institut d'Estudis Avançats and Departament de Física, Consejo Superior de Investigaciones Científicas, and Universitat de les Illes Balears, 07071 Palma de Mallorca, Spain. E-mail: DFSRTG0@PS.UIB.ES.
* Present and permanent address.

Gaussian random number is then given as the ratio v/u.

In this paper we describe a fully vectorised implementation of the well known numerical inversion (NI) method [5] suitable to generate approximately Gaussian distributed random numbers. The main advantage of the method is that it is very fast. The effects of the approximation are clearly discussed. It turns out that for many problems the approximation is perfectly valid, resulting in a substantial saving of computer time.

Let us start by reviewing briefly the numerical inversion method. Let x be a random variable with a probability density function $f(x)$ and distribution function $F(x)$ related to $f(x)$ by

$$F(x) = \int_{-\infty}^{x} f(y)\, dy. \tag{2}$$

The straightforward method to generate random variables distributed according to $f(x)$ is the following. One notes that the random variable $\xi = F(x)$ is uniformly distributed between 0 and 1 (we will write, for short, that ξ is a U(0,1) number). One then generates a U(0,1) number ξ by any of the usual methods (which include using the machine built-in random number generator) and calculates $x = F^{-1}(\xi)$, $F^{-1}(\xi)$ being the inverse function of $F(x)$. The main problem arises when the function $F^{-1}(\xi)$ is not expressible in terms of elementary functions or when its analytical form is too complicated to be efficiently implemented on a computer. In the case of the Gaussian distribution, the inverse function $F^{-1}(\xi)$ is related to the inverse error function. A good approximate algorithm exists for the calculation of the inverse error function [6] and this approach has been used in ref. [7] to compute Gaussian distributed random numbers. However, due to the fact that the approximate algorithm also includes a logarithm and a square root functions (see later), one gets only a slight improvement in the necessary time to generate a Gaussian random number as compared to the BMW algorithm.

The standard trick to avoid the calculation of the inverse function $F^{-1}(\xi)$ every time one needs a random number is to divide the interval [0,1] in M subintervals $[i/M, (i+1)/M]$, $i = 0, 1, \ldots, M-1$, and tabulate the inverse probability distribution function in the points $\xi_i = i/M$, i.e. compute and store $x_i = F^{-1}(i/M)$ for $i = 0, \ldots, M$. One then substitutes the true probability distribution function $F(x)$ by its piecewise linear approximation between the points $[i/M, (i+1)/M]$. Then a random number ξ is generated from a uniform distribution in [0,1] and the following linear interpolation formula is used to compute the random number $x = F^{-1}(\xi)$:

$$x = (M\xi - i)x_{i+1} + (i + 1 - M\xi)x_i, \tag{3}$$

where i is such that the number ξ belongs to the interval $[i/M, (i+1)/M)$, or $i = [M\xi]$ (integer part of $M\xi$).

Whether this numerical inversion method will produce good quality random numbers will depend on the quality of the approximation of the true function $F(x)$ by its piecewise linear approximation. This is determined by the smoothness of the function $F(x)$ itself and also by the number of subdivisions M of the interval [0,1]. On the other hand, the NI method is usually fast, since it only involves additions and multiplications and can be easily vectorized in most of the modern vector compilers. Its only overhead is the calculation of the numbers x_i once.

The rest of the paper is organized as follows: in section 2 we describe a possible implementation of the NI method suitable for the Gaussian distribution; section 3 contains some programming details of the algorithm; section 4 compares the timings of the algorithm in different computers and discusses the validity of some of the approximations involved. Finally, a computer listing suitable for the CRAY family of computers is given in the appendix.

2. Implementation of the algorithm

We will now present an explicit implementation of the table inversion method to generate Gaussian random numbers of mean 0 and variance 1. The Gaussian probability density and

probability distribution functions are given respectively by:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2},$$

$$F(x) = \int_{-\infty}^{x} (y)\, dy = \tfrac{1}{2}\left(1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right). \qquad (4)$$

Here $\mathrm{erf}(x)$ is the error function [6].

The first problem appears when one notices that, due to the fact that the Gaussian distribution extends from $-\infty$ to $+\infty$, we have $x_0 = F^{-1}(0) = -\infty$, $x_M = F^{-1}(1) = \infty$ and those numbers cannot be used in the interpolation formula (3). A possible solution consists in cutting-off the Gaussian distribution to some value Γ, i.e. define the distribution:

$$\hat{f}(x) = \begin{cases} 0, & x < -\Gamma, \\ \alpha f(x), & -\Gamma \le x \le \Gamma, \\ 0, & x > \Gamma. \end{cases} \qquad (5)$$

The constant α, which is necessary for normalization, can be easily related to the value of the cut-off Γ,

$$1 = \int_{-\infty}^{\infty} \hat{f}(x)\, dx = \alpha\, \mathrm{erf}\left(\Gamma/\sqrt{2}\right), \qquad (6)$$

which yields

$$\alpha = \frac{1}{\mathrm{erf}\left(\Gamma/\sqrt{2}\right)}. \qquad (7)$$

The corresponding probability distribution function is given by:

$$\hat{F}(x) = \begin{cases} 0, & x \le -\Gamma, \\ \alpha\left(F(x) - F(-\Gamma)\right), & -\Gamma \le x \le \Gamma, \\ 1, & x \ge \Gamma. \end{cases} \qquad (8)$$

A word of caution is necessary concerning the value of the cut-off parameter Γ. In principle, one is tempted to take Γ as large as possible in order to mimic as closely as possible to tail of the true Gaussian distribution. On the other hand, we must not forget that we are substituting the Gaussian probability distribution function $F(x)$ by the piecewise linear approximation to $\hat{F}(x)$ computed in the points $x_i = \hat{F}^{-1}(i/M)$. In order to have the linear approximation as close as possible to the real Gaussian distribution we also need M as large as possible. It is not efficient to choose M too large because this would demand too much computer memory to store the table of the x_i's. A "reasonable" value for M that we have been using is $M = 2^{14} = 16384$ (although, of course, larger values could be used if computer memory is not a problem). With that particular value of M it turns out that $x_1 = \hat{F}^{-1}(1/M)$ is equal to $x_1 = -3.668$. If we choose now, say, $\Gamma = 10$, the linear approximation between $x_0 = -\Gamma = -10$ and x_1 is obviously very bad. In order to allow for a smooth linear approximation we have chosen $x_0 = -\Gamma = F^{-1}(1/(M+2))$ which is sufficiently close to x_1 to let the linear function be a reasonable approximation to the true Gaussian distribution (for $M = 16384$, $x_0 = -\Gamma = -3.842$). Greater values of Γ can be obtained by increasing M: $\Gamma = 4.170, 4.475, 4.763$, for $M = 2^{16}$, 2^{18}, 2^{20}, respectively (see table 1).

The distribution function $\hat{F}(x)$ is given by

$$\hat{F}(x) = \begin{cases} 0, & x \le F^{-1}\left(\dfrac{1}{M+2}\right) = -x_M, \\ \dfrac{M+2}{M} F(x) - \dfrac{1}{M}, & -x_M \le x \le x_M, \\ 1, & x \ge x_M. \end{cases} \qquad (9)$$

Using this choice it is easy to verify that the values of x_i are given by

$$x_i = \hat{F}^{-1}(i/M) = F^{-1}\left(\frac{i+1}{M+2}\right), \quad i = 0, \ldots, M. \qquad (10)$$

Due to the symmetry of the Gaussian distribution is obvious that

$$x_{M-i} = -x_i. \qquad (11)$$

It is clear also that if x is a random variable

distributed according to $\hat{f}(x)$, then its mean value is zero:

$$\langle x \rangle = 0. \tag{12}$$

The variance of x is given by

$$\sigma_M^2 = \langle x^2 \rangle = \int_{-\infty}^{\infty} y^2 \hat{f}(y)\, dy. \tag{13}$$

Integration by parts yields

$$\sigma_M^2 = 1 - \frac{M+2}{M}\sqrt{\frac{2}{\pi}}\, \Gamma\, e^{-\Gamma^2/2}. \tag{14}$$

This is to be compared with the value 1 for the real Gaussian distribution. When, for instance, $M = 16384$, then $\sigma_M^2 = 0.99814\ldots$. Although this value is already very close to 1 it might be desirable to have a distribution with an exact variance of 1 (see the discussion in section 4). This is easily achieved by defining a new random variable $\tilde{x} = x/\sigma_M$. This variable $\tilde{x}$ is our final proposal for a "pseudo-Gaussian" random variable. Its probability density function $\tilde{f}(x)$ is explicitly given by

$$\tilde{f}(\tilde{x}) = \sigma_M \hat{f}(\sigma_M \tilde{x}). \tag{15}$$

3. Programming details

For the explicit implementation of the method detailed above one needs to find in which interval $[i/M, (i+1)/M]$ belongs the U(0,1) number ξ. Since, in the most popular U(0,1) generators, ξ is obtained by a ratio of two integer numbers IR/L_0, where IR is a series of NBIT random bits and $L_0 = 2^{NBIT}$, it turns out that the interval index i can be found simply by the integer operation $i = [(M/L_0)IR]$. If we choose now M to be also a power of 2, $M = 2^{NP}$, then one needs to compute $i = [IR/2^{NBIT-NP}] \equiv [IR/2^{NP1}]$. Instead of this division one could simply shift right NP1 times the bit representation of the integer number IR. Although it is not Fortran 77 standard, the shift operation is implemented in many compilers (in the Cray compiler it is called SHIFTR(IR, NP1)) and it is usually faster to perform than an integer division. The difference $(M\xi - i)$ that appears in eq. (3) is equal to $(IR \bmod(2^{NP1}))/2^{NP1}$. The modulus operation can be done as the bit logical AND operation, namely:

$$I2 \equiv IR \bmod(2^{NP1}) = IR.AND.(2^{NP1} - 1). \tag{16}$$

Again, the bit logical AND operation is standard in many compilers (the above expression is given in the CRAY compiler format).

The linear interpolation formula (3) translates into:

$$x = \frac{I2}{2^{NP1}} x_{i+1} + \frac{2^{NP1} - I2}{2^{NP1}} x_i. \tag{17}$$

If we now introduce the vector $g_i = (x_i/2^{NP1}\sigma_M)$, the final formula producing the random numbers $\tilde{x}$ distributed according to $\tilde{f}(x)$ is

$$\tilde{x} = I2\, g_{i+1} + (2^{NP1} - I2)\, g_i \tag{18}$$

The algorithm described above has been implemented for the CRAY family of computers and a computer listing can be found in the appendix. The program is organized as two subroutines: INIG250(ISEED,IX,N) and G250(IX,U,N). A call to the subroutine G250(IX,U,N) will fill each element of the real vector U(N) with a Gaussian distributed random number. The subroutine INIG250 needs to be called once to initialize the algorithm with a suitable value for the random number generator seed ISEED. In this program we are using the R250 version of a shift register random number generator [8] to compute the integer random numbers required by the algorithm. This generator uses the recursion formula:

$$IX_i = IX_{i-p}.XOR.IX_{i-q}, \tag{19}$$

with $p = 250$, $q = 103$. IX is an integer vector (of which only the last 32 bits are used) of dimension $N + p$. This has to be initialized with p initial values for the vector IX. This is achieved by using the machine built-in random number generator (called RANF() on the CRAY) which is, in turn, initialized by the call to RANSET(ISEED).

The calculation of the numbers $x_i = F^{-1}((i+1)/(M+2))$ is done by the use of the following

approximation formula to the inverse function $x = F^{-1}(\xi)$ [6]:

$$x = t - \frac{c_0 + c_1 t + c_2 t^2}{1 + d_1 t + d_2 t^2 + d_3 t^3} + \epsilon, \qquad (20)$$

where

$$t = \sqrt{-2 \log(1 - \xi)}\,. \qquad (21)$$

c_0, c_1, c_2, d_1, d_2, d_3 are parameters. This approximation has an error ϵ less than 4.5×10^{-4} for the range $0.5 \leq \xi \leq 1.0$. The values for $0 \leq \xi \leq 0.5$ are found by using the symmetry relation equation (11).

4. Performance and discussion

The program given in the appendix has been timed on the CRAY-YMP of the Pittsburgh Supercomputer Center and NCSA, Urbana–Champaign (CFT77 compiler version 5.0). The time taken is 3.6×10^{-8} s per random number (that includes the overhead when calling the subroutine G250 for vector length $N = 10^6$). This is to be compared with 2.3×10^{-7} s required by the BMW algorithm, 3.2×10^{-6} s by the ratio method and 2.1×10^{-7} s by the G12 generator, i.e. the NI method is several times faster. On a VAX 9000 (with a vector processor), the NI method takes 3.5×10^{-7} s as compared with 1.3×10^{-6} s for the BMW algorithm, 3.6×10^{-6} s for the ratio method and 1.9×10^{-6} s for the G12 generator. Given that the ratio method is not the best for vector computers since it does not fully vertorise, we have also timed the different algorithms on a VAX 6000 which is a scalar computer, with the results: 6.4×10^{-7} s for the NI method, 4.1×10^{-6} s for the BMW, 3.6×10^{-6} s for the ratio method and 2.8×10^{-6} s for the G12 generator. In all the cases we have used the R250 routine for the generation of U(0,1) numbers. We conclude that, both for vector and scalar computers, the NI method is several times faster than the other methods tested here. The previous timings have been obtained using Fortran routines and it is conceivable that further improvement could be obtained using machine code programming. However, it is clear that the NI method can be programmed more efficiently than the other algorithms, since the only computations required are logical operations, additions and multiplications, whereas the other exact methods use more complicated functions.

Table 1
Parameters of the random number generator as a function of the table size $M = 2^{NP}$.

NP	Γ	$\langle X^4 \rangle$	$\langle X^6 \rangle$	KS distance
6	2.166551	2.453355	9.327704	1.5134E-02
7	2.423623	2.586529	9.958819	7.6832E-03
8	2.663078	2.699921	10.859581	3.8714E-03
9	2.887209	2.790338	11.793347	1.9433E-03
10	3.098136	2.858432	12.631319	9.7370E-04
11	3.297699	2.907257	13.318054	4.8740E-04
12	3.487411	2.940824	13.844924	2.4386E-04
13	3.668452	2.963087	14.228882	2.0087E-04
14	3.842003	2.977428	14.497684	1.8332E-04
15	4.008797	2.986429	14.679537	1.7497E-04
16	4.169549	2.991958	14.799202	1.7089E-04
17	4.324857	2.995294	14.876134	1.6888E-04
18	4.475228	2.997277	14.924628	1.6787E-04
19	4.621095	2.998439	14.954684	1.6735E-04
20	4.762833	2.999113	14.973043	1.6710E-04

It is not only necessary that the program is fast, but it also has to deliver good quality random numbers. To show the effect of the tail cutoff we have computed analytically the values of the 4th and 6th moments of the distribution given by eq. (15) as a function of the cutoff value Γ (remember that the second-order moment is exactly 1 by construction). These values are shown in table 1. We can see from this table that for a table size of, e.g., $M = 2^{14}$ there is a relative error of 7.524×10^{-3} and 3.35×10^{-2} in the fourth- and sixth-order moments, respectively. Another measure of the inaccuracy of the generator is given by the Kolmogorov–Smirnov (KS) distance, which is defined as the maximum deviation between the true Gaussian distribution function, $F(x)$, and the piecewise linear approximation $\tilde{F}(x)$ used in the numerical inversion method:

$$D_{KS} \equiv \max_{-\infty < x < +\infty} |F(x) - \tilde{F}(x)|. \qquad (22)$$

We have included in table 1, the values of the KS distance, as a function of the size of the table

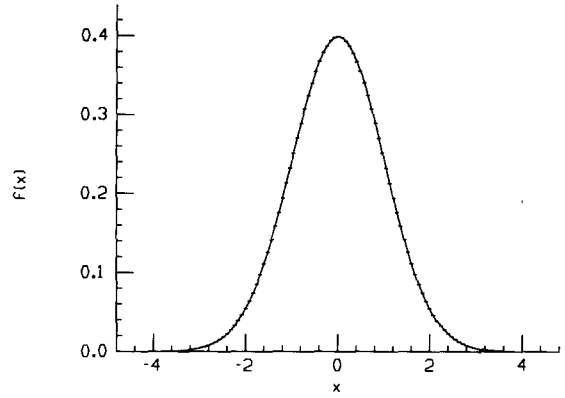

Fig. 1. Histogram of the distribution of Gaussian numbers created after generating 10^8 numbers according to the numerical inversion method explained in the text. The exact Gaussian distribution (solid line) is also plotted. The histogram width is $\Delta x = 0.08$. Note that at the scale of this figure, there is no difference between the exact Gaussian distribution and the numerical one.

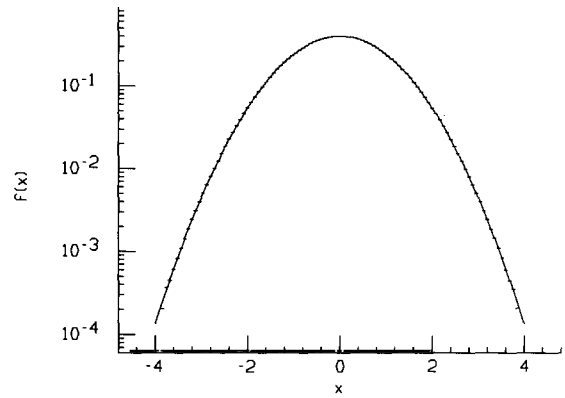

Fig. 2. Logarithmic plot of fig. 1 to outline the very small discrepancies in the tail of the distribution.

used in the NI method. For table size $M = 2^{14}$, for example, we obtain $D_{KS} = 1.833 \times 10^{-4}$.

In order to test the algorithm and to see the effect of the piecewise linear approximation, we have generated 10^8 Gaussian random numbers by using the NI method. The lower order moments obtained were $\langle x \rangle = -1.228 \times 10^{-6}$, $\langle x^2 \rangle = 0.9999923$, $\langle x^3 \rangle = 6.20 \times 10^{-5}$, $\langle x^4 \rangle = 2.979$, $\langle x^5 \rangle = 6.53 \times 10^{-4}$, $\langle x^6 \rangle = 14.56$ in good agreement with the analytical results of table 1. In fig. 1 we show a plot of the probability density function obtained numerically after generating 10^8 random numbers by the NI method compared with the exact Gaussian distribution. The main objective of this figure is to show that the discrepancies are too small to be seen in this sale. Figure 2 is a logarithmic plot of fig. 1 to show the small discrepancies in the tail of the distribution. These results show that linear approximation is sufficient for the table sizes used here. Of course, one could reduce the value of the table size required by using higher-order interpolation. However, we believe that since a table size of 16384 real elements cannot cause any memory problems in modern computers, there is no real need to use the more complex algorithms that involve higher-order interpolation.

Whether the NI method as implemented here would be suitable to a particular problem will depend on how much the tail of the Gaussian distribution can affect the results. Consider, for example, the problem of solving numerically a stochastic differential equation with a Gaussian random noise source [9]. The numerical algorithms use Gaussian random numbers of mean zero and variance 1. It has been suggested that, in order to speed up the algorithm, some of these Gaussian random numbers can be replaced by non-Gaussian random numbers with the same first lower moments, without modifying the order of convergence of the algorithm [10]. In particular, the random variable defined by the probability density function,

$$f(x) = \tfrac{1}{6}\delta(x + \sqrt{3}) + \tfrac{2}{3}\delta(x) + \tfrac{1}{6}\delta(x - \sqrt{3}), \tag{23}$$

($\delta(x)$ is the Dirac delta function) has the five lower moments, $\langle x \rangle$, $\langle x^2 \rangle$, $\langle x^3 \rangle$, $\langle x^4 \rangle$, $\langle x^5 \rangle$ equal to those of the Gaussian distribution. However, the fact that this distribution is clearly very different from a Gaussian distribution could introduce some unwanted bias in the errors. We believe that it is better to maintain the shape of the distribution function as close to that of a Gaussian as possible and, what is more important, it does not take longer with the NI method to generate a pseudo-Gaussian random number than it would take to generate a random number

with the distribution of eq. (23). In other problems, however, (like in the case of mean first passage time problems (see e.g. [11])) the tails of the distribution can be important and the NI method may not be suitable since it cuts-off the tail. The cut-off value Γ could be, of course, increased by increasing the value of the parameter M. We would like to stress that even the Box–Muller–Wiener algorithm, when implemented on a computer, gives a cutoff distribution because the maximum number one can obtain in a, say, 32 bits machine is $\sqrt{-2\log(2^{-31})} = 6.55$. On the other hand, one may also ask whether it is physically meaningful that the results of a particular problem depend so heavily on the tails of the Gaussian distribution, or whether the physical process is still well described by a Gaussian distribution even for events lying 3.842 times the standard deviation of the mean.

Appendix. Program listing

```
C     G250
C     Gaussian random number generator
C
C     Fills vector U(N) with cutoff Gaussian
      distributed
C     random numbers by using a numerical in-
      version method.
C     The cutoff value depends on the parame-
      ter NP.
C
C     To use first initialize it by calling
            CALL INIG250 (ISEED,IX,N)
C
C     where in the main program one needs to
      put
C
C     DIMENSION IX(N+250)
C     DIMENSION U(N)
C
C       Then subsequent calls:
C     CALL G250(IX,U,N)
C
C     will fill up U with cutoff Gaussian
      random numbers.
      subroutine inig250(iseed,ix,n)
      parameter (np=14)
      parameter (m=2**np)
      parameter (ip=250, iq=250-103)
      parameter (nbit=32)
      parameter (np1=nbit-np)
      dimension ix(n+ip)
      dimension g(0":"m)
      common/c250/g
      common/d250/nn, nn1
      data c0,c1,c2/
        2.515517,0.802853,0.010328/
      data d1,d2,d3/
        1.432788,0.189269,0.001308/
      maxint=2**nbit-1
      call ranset(iseed)
      do 200 i=1, ip
      ix(i)=ranf()*maxint
200   continue
      nn1=2**np1
      nn=nn1-1
      pi=4.0d0*datan(1.0d0)
      do 1 i=m/2, m
      p=real(i+1)/(m+2)
      t=sqrt(-2.0*log(1.-p))
      x=t-(c0+t*(c1+c2*t))/(1.0+t
        *(d1+t*(d2+t*d3)))
      g(i)=x
      g(m-i)=-x
1     continue
      write(6, *) 'Cut-off value= ', g(m)
      u2th=1.0-real(m+2)/m*sqrt(2.0/
        pi)*g(m)*exp(-g(m)*g(m)/2)
      u2th=nn1*sqrt(u2th)
      do 856 i=0, m
856   g(i)=g(i)/u2th
      return
      end
      subroutine g250(ix, u,n)
      parameter (np=14)
      parameter (m=2**np)
      parameter (ip=250, iq=250-103)
      parameter (nbit=32)
      parameter (np1=nbit-np)
      dimension ix(n+ip)
      dimension u(n)
      dimension g(0:m)
      common/c250/g
      common/d250/nn,nn1
      do 150 k=1,n
      ir=ix(k+iq).xor.ix(k)
      i=shiftr(ir,np1)
      i2=ir.and.nn
      u(k)=i2*g(i+1)+(nn1-i2)*g(i)
      ix(k+ip)=ir
150   continue
      do 250 i=1, ip
      ix(i)=ix(i+n)
250   continue
      return
      end
```

Acknowledgements

We acknowledge financial support from the Dirección General de Investigación Científica y Técnica, contrast number PB 89-424, and by the Universitat de les Illes Balears (UIB). This work was also partially supported by NSF grant no. DMR-9100245 and grants of computer time at the Pittsburgh Supercomputing Center of Supercomputing Applications, Urbana–Champaign. Acknowledgement is also made to the Donors of the Petroleum Research Fund, administered by the American Chemical Society, for partial support for this research. A.C. also thanks the Universitat de les Illes Balears (UIB) for their hospitality while part of this work was carried out.

References

[1] E.T. Gawlinski, J. Vinals and J.D. Gunton, Phys. Rev. B 39 (1989) 7266.

[2] G.E.P. Box and M.E. Muller, Ann. Math. Statist. 29 (1958) 610.

[3] D.E. Knuth, The Art of Computer Programming, vol. 2, Seminumerical Algorithms (Addison–Wesley, Reading, MA, 1981).
W.H. Press, B.P. Flannery, S.A. Teulolsky and W. Vetterling, Numerical Recipes, Cambridge Univ. Press, Cambridge, 1986).

[4] J.H. Ahrens and U. Dieter, Commun. ACM 15 (1972) 873.

[5] M.H. Kalos and P.A. Whitlock, Monte Carlo Methods (Wiley, New York, 1986).

[6] M. Abramowitz and I.A. Stegun, Handbook of Mathematical Functions (Dover, New York, 1972).

[7] O.T. Valls and G.F. Mazenko, Phys. Rev. B 34 (1986) 7941.

[8] S. Kirpatrick and E.P. Stoll, J. Comput. Phys. 40 (1981) 517.

[9] T.C. Gard, Introduction to Stochastic Differential Equations, Monographs and Textbooks in Pure and Applied Mathematics, Vol. 114 (Marcel Dekker, New York, 1988).

[10] A. Greiner, W. Strittmatter and J. Honerkamp, J. Stat. Phys. 51 (1988) 95.

[11] C.W. Gardiner, Handbook of Stochastic Methods (Springer, Berlin, 1985).