11 Monte Carlo Method

11.1 Introduction

The classical way to solve a problem of numerical analysis relies on a rigorous algorithm, which, for a given input, provides a well-defined solution in a predetermined number of steps. Such an approach is essentially *deterministic*, in the sense that its implementation is expected to lead on repeated runs, upon completion of the same number of operations, to the same result. Still, for numerous topical problems in applied sciences and engineering, the complexity of deterministic algorithms renders the computations intractable within reasonable amounts of time.

Certain areas of modern sciences are concerned with systems composed of a huge number of coupled components, which are sometimes also prone to fluctuations. Such complex systems can be as different as collections of spins, biological populations, or galaxies. Their characterization is often accomplished by means of high-dimensional integrals. A typical example is the *classical canonical partition function* of a system of N interacting particles,

$$Z = \int \cdots \int d^3 r_1 \cdots d^3 r_N \; \exp\left[-\frac{1}{k_B T} E(\mathbf{r}_1 \cdots \mathbf{r}_N)\right],$$

where $E(\mathbf{r}_1 \cdots \mathbf{r}_N)$ is the total energy of the system, with $\mathbf{r}_i$, the position of particle i, T is the system temperature, and k_B, the Boltzmann constant. The evaluation of this $3N$-dimensional integral by any of the classical quadrature methods can be completely ruled out even for the lowest particle numbers of practical interest. In fact, considering the modest value $N = 20$ and only 10 integration points for each dimension, the number of required elementary operations would be on the order of 10^{60}. Even using the latest petascale supercomputers, which are capable of more than 10^{16} floating point operations per second, the computation would require approximately $3 \cdot 10^{36}$ years!

A rewarding alternative to the deterministic approaches for complex high-dimensional problems are the so-called *stochastic methods*, based on the *law of large numbers* from the probability theory. Here, the quantities of interest are defined as expectation values of random variables or, in other words, the average values of large sequences of random variables are considered under certain assumptions probabilistic estimates of the sought-for quantities. Such techniques, generically referred to as *Monte Carlo methods*, have an intrinsically *nondeterministic* character, since they use the outcome of stochastic experiments and, within statistical errors, they exhibit different behaviors on different runs. Essentially, instead of deterministically covering the domains of the involved functions, the Monte Carlo methods sample these randomly. In general, stochastic techniques do not require *genuine* random numbers, but rather *pseudorandom* sequences, nevertheless with a high degree of uniformity and low sequential correlations.

The seminal Monte Carlo method was developed in the late 1940s at the Los Alamos National Laboratory by Stanislaw Ulam, Nicholas Metropolis, and John von Neumann, and it was named, due to the gambling-like underlying principles, after the famous Monte Carlo casino. The Monte Carlo method started unveiling its virtues with the advent of the high-performance computers, because the attainment

of sufficiently accurate results generally implies a tremendous number of operations and conveying vast amounts of data. By comparison with the deterministic numerical methods, the success of the Monte Carlo method is mainly due to the more advantageous scaling of the computational effort with increasing problem size. In fact, while being the most inefficient quadrature method for one-dimensional integrals, with increasing dimensionality, the Monte Carlo method progressively exceeds the deterministic methods.

The main uses of the Monte Carlo method can be broadly categorized into optimization, numerical integration, and generation of samples from probability distributions. In applied mathematics, the Monte Carlo method reveals its efficiency most clearly in applications such as the evaluation of multidimensional integrals with complex boundaries, the solution of large linear systems, or the solution of Dirichlet problems for differential equations. In this chapter, we specifically focus on the integration of functions.

11.2 Integration of Functions

In the Monte Carlo method, the integral of a function f over a multidimensional domain $\mathcal{D}$ is simply estimated by the product of the arithmetic mean of the function and the domain volume V,

$$\int_{\mathcal{D}} f dV \approx V \langle f \rangle \pm \sigma, \tag{11.1}$$

where the average $\langle f \rangle$ is calculated for n random points, $\mathbf{x}_1, ..., \mathbf{x}_n$, sampling *randomly* and *uniformly* the domain $\mathcal{D}$. The statistical uncertainty associated with the result of the integration is specified by the *standard deviation* σ,

$$\sigma = V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{n}} = \frac{V}{\sqrt{n}} \sigma_f, \tag{11.2}$$

or its square, σ^2, called *variance*. The standard deviation,

$$\sigma_f = \sqrt{\left\langle \left(f - \langle f \rangle \right)^2 \right\rangle} = \sqrt{\langle f^2 \rangle - \langle f \rangle^2},$$

occurring in the above expression, measures the deviation of the integrand f from its mean $\langle f \rangle$ within the integration domain. The averages $\langle f \rangle$ and $\langle f^2 \rangle$ are given, respectively, by

$$\langle f \rangle \equiv \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}_i), \quad \langle f^2 \rangle \equiv \frac{1}{n} \sum_{i=1}^{n} f^2(\mathbf{x}_i). \tag{11.3}$$

Expression 11.2 of the standard deviation σ illustrates two important features of the Monte Carlo integration. The first refers to the $n^{-1/2}$ scaling of the uncertainty σ. As expected, the larger the number of sampling points, the more precise the result of the integration, even if the decrease of σ is rather slow. By comparison, the error of the basic trapezoidal formula, the most rudimentary of the Newton–Cotes quadrature family, features an n^{-2} dependence and the required numerical effort for achieving the same precision in a *one-dimensional* integration proves to be definitely lower than for the Monte Carlo method. However, the advantage of the more favorable scaling of the operation count for the classical quadrature formulas vanishes in multidimensional cases. It can be easily seen that the remainder of a d-dimensional integration scheme based on a conventional one-dimensional integrator, characterized by n function evaluations for each dimension, increases as $n^{-2/d}$. On the contrary, the $n^{-1/2}$ dependence of the uncertainty σ is conserved irrespective of the dimensionality, so that, for $d > 4$, the Monte Carlo method becomes more efficient.

The second defining aspect of the Monte Carlo integration concerns the fact that, since the uncertainty σ is proportional to the standard deviation σ_f of the integrand, it is smaller, the "flatter" the function f, that is, the less f departs from its average value. In the particular case of a constant function, its evaluation at a single sampling point is sufficient for the variance to vanish and the average to be precisely defined. On the contrary, assuming that f has significant values only in a confined region, while the integration points $\mathbf{x}_i$ are generated with *equal probability* within the entire domain $\mathcal{D}$, it is likely that their majority samples regions with insignificant contributions, resulting in an inaccurate estimate of the integral.

Even though the real virtues of the Monte Carlo method emerge in the evaluation of multidimensional integrals, for the sake of clarity, we confine the discussion for the moment to the one-dimensional case. The generic integral,

$$I = \int_a^b f(X)dX, \tag{11.4}$$

may be represented relative to the standard integration domain $[0, 1]$ by the change of variables $X = (b - a)x + a$, namely:

$$I = (b - a) \int_0^1 f((b - a)x + a)dx. \tag{11.5}$$

The general Monte Carlo formula 11.1 becomes in this particular case:

$$I \approx \big(b - a\big) \big\langle f \big\rangle = \frac{b - a}{n} \sum_{i=1}^{n} f((b - a)x_i + a). \tag{11.6}$$

The average $\big\langle f \big\rangle$ is expressed here by sampling the integrand at the points $X_i = (b - a)x_i + a$ from the interval $[a, b]$, corresponding to n random points x_i distributed uniformly (with equal probability) within the standard interval $[0, 1)$.

We defer the task of generating random variables with different distributions for Section 11.5. Since computers are by essence deterministic machines, by "random" we will rather understand "pseudo-random," as these numbers will result from deterministic recurrences producing bit-wise truncations. For the time being, we just briefly mention language built-in utility functions for creating uniform distributions of random numbers.

Python provides as part of the standard module `random.py` a wealth of functions related to random sequences. The function `seed()` called without arguments uses the current system time to initialize the random number generator. The call to `random()` returns the next random floating point number of a sequence in the range $[0, 1)$.

The C/C++ built-in random number generator can be initialized by invoking the function `srand`, which generates a "seed" for a subsequent random sequence depending on the received argument. A simple initialization method is to call `srand` passing the current Central Processing Unit (CPU) time as returned by the function `time` (defined in the standard header file `time.h`), like in the following macro:

```
// Initializes the random number generator using the current system time
#define seed() srand((unsigned)time(NULL))
```

In particular, `time(NULL)` returns the number of seconds elapsed since 00:00 hours, January 1, 1970.

The standard C/C++ function `rand()` (defined in the header file `stdlib.h`) generates on repeated calls sequences of integer pseudorandom numbers uniformly distributed in the interval `[0,RAND_MAX]`. The predefined constant `RAND_MAX` is library dependent, but is guaranteed to be

at least 32,767 in any standard implementation of the C/C++ compiler. To generate real random numbers in the standard interval $[0, 1)$, the values returned by `rand()` have to be simply divided by `(RAND_MAX+1)`:

```
// Generates random floating point numbers in the range [0,1)
#define random() ((double)rand()/(RAND_MAX+1))
```

The macros `seed()` and `random()` will be assumed in what follows to be included in the header file `random.h`.

As a first, elementary example of Monte Carlo quadrature, we consider the integral

$$\int_0^1 xe^{-x}dx = 1 - 2e^{-1} \approx 0.26424. \tag{11.7}$$

The corresponding program (Listing 11.1) does not actually implement the Monte Carlo methodology as a distinct procedure, since a suchlike routine lacks practical interest, as the whole idea of using the Monte Carlo method for performing one-dimensional integration. Instead, the main program is kept simple, with a view to emphasizing the general concept. Here, `n` represents the number of sampling points, `s` is the integral estimate, and `sigma` represents the associated standard deviation. The variables `f1` and `f2` are used to store, respectively, the average $\langle f \rangle$ and the average squared function $\langle f^2 \rangle$.

The results of the program execution for different numbers of integration points n are compiled in the first three columns of Table 11.1. Along with the increase of n, the estimates of the integral converge toward the exact value, with obviously decreasing associated uncertainty estimates σ. As will be shown in the next section, the standard deviations σ scale indeed as $n^{-1/2}$ and a comparison with any

Listing 11.1 Monte Carlo Integration of a Function (Python Coding)

```
# One-dimensional Monte Carlo quadrature
from math import *
from random import *

def func(x): return x * exp(-x)                                  # integrand

# main

n = eval(input("n = "))                          # number of sampling points

seed()

f1 = f2 = 0e0                                 # quadrature with uniform sampling
for i in range(1,n+1):
   x = random()                                  # RNG with uniform distribution
   f = func(x)                                                   # integrand
   f1 += f; f2 += f * f                                              # sums

f1 /= n; f2 /= n                                                  # averages
s = f1                                                            # integral
sigma = sqrt((f2-f1*f1)/n)                                # standard deviation
print("s = ",s," +/- ",sigma)
```

TABLE 11.1 Monte Carlo Estimates of the Integral $I = \int_0^1 xe^{-x}dx$ and Associated Standard Deviations σ, Calculated Using Random Sequences with the Distributions $w(x) = 1$ and, Respectively, $w(x) = (3/2)x^{1/2}$

	$w(x) = 1$		$w(x) = (3/2)x^{1/2}$	
n	I	σ	I	σ
10	0.25108	0.03855	0.26860	0.00727
100	0.26252	0.01085	0.26397	0.00285
1000	0.26622	0.00336	0.26458	0.00089
10,000	0.26544	0.00105	0.26423	0.00027

classical one-dimensional quadrature formula proves the latter to be significantly more efficient. While the Monte Carlo integration can be yet improved, its full potential comes to light only in the evaluation of multidimensional integrals.

11.3 Importance Sampling

Having in view that the uncertainty σ (11.2) of the Monte Carlo quadrature is directly related to the variance of the integrand σ_f, which measures the deviation of the integrand from its average value, to increase the precision and efficiency of the quadrature, one can apply a general strategy, known as *variance reduction*. More precisely, we discuss a technique called *importance sampling*.

Essentially, one introduces a positive *weight function*, $w(x)$, normalized to unity over the interval $[0, 1]$,

$$\int_0^1 w(x)dx = 1, \tag{11.8}$$

by which integrand (11.5) is multiplied and divided:

$$I = (b-a)\int_0^1 \frac{f((b-a)x+a)}{w(x)}w(x)dx.$$

Performing the change of variable $d\xi = w(x)dx$ or, in integral form,

$$\xi(x) = \int_0^x w(x')dx', \tag{11.9}$$

with the boundary conditions $\xi(0) = 0$ and $\xi(1) = 1$ (to comply with normalization (11.8)), the integral becomes

$$I = (b-a)\int_0^1 \frac{f((b-a)x(\xi)+a)}{w(x(\xi))}d\xi. \tag{11.10}$$

The conservation of the integration domain upon the change of variable is a direct consequence of the normalization of $w(x)$. To evaluate this integral, one can directly apply the Monte Carlo method described in the preceding section, by averaging the new integrand, f/w, over a set of *uniformly distributed* sampling points ξ_i from the interval $[0, 1)$:

$$I \approx \frac{b-a}{n}\sum_{i=1}^{n}\frac{f((b-a)x(\xi_i)+a)}{w(x(\xi_i))}. \tag{11.11}$$

The performed change of variable may seem at first glance to turn the quadrature more complicated. The advantages of this technique become apparent only if one chooses the weight function $w(x)$, in as

much as possible, *similar* to the function $f((b-a)x+a)$. In such a case, the integrand f/w becomes *smooth and slowly varying* (in the ideal case $w \equiv f$, even constant). As a consequence of the reduced fluctuations of the integrand with respect to its average, there results a variance reduction for the integral itself. Naturally, this benefit is directly contingent upon choosing an adequate weight function $w(x)$ and on the possibility of expressing the dependence $x = x(\xi)$ by inverting the integral relation 11.9.

A deeper insight into the usefulness of the variable change (11.9) is gained by noting that the uniform distribution of the random points ξ_i corresponds to a, generally, *nonuniform distribution* of the arguments $x_i \equiv x(\xi_i)$, namely, according to the weight function $w(x)$. In other words, the probability of a transformed abscissa x_i to be located in the interval $(x, x+dx)$ is $w(x)dx$. This means that the arguments x_i are generated with maximum probability and are, therefore, concentrated in regions where w is large. For an adequate choice of w, the sampling points are, thus, predominantly concentrated in regions with significant values of the original integrand f. One achieves in such a manner an optimal sampling of the integration domain (importance sampling), diminishing the effort "wasted" on evaluating the integrand at points with reduced statistical significance in sum (11.11).

To illustrate the operation of the variance reduction by importance sampling, we consider again the integral

$$\int_0^1 xe^{-x}dx \approx 0.26424. \tag{11.12}$$

The integrand monotonically rises in the domain $[0,1]$ from 0 to e^{-1}, while maintaining the negative second derivative. A similar behavior is featured by the function $e^{-1}x^{1/2}$, for which, most importantly, the integral transformation (11.9) is invertible. The adequately normalized weight function is in this case

$$w(x) = (3/2)x^{1/2}. \tag{11.13}$$

By inverting the change of variable (11.9), one obtains

$$x = \xi^{2/3}. \tag{11.14}$$

Listing 11.2 shows the implementation of the Monte Carlo quadrature 11.12 with importance sampling based on Formula 11.11. The functions `seed()` and `random()` (defined in the modules `random.py` and `random.h`) are called to initialize and, respectively, generate real uniform random sequences. For variables with the distribution $w(x) = (3/2)x^{1/2}$, the program defines the routine `ranSqrt()`, which calls, in its turn, the routine `random()` for producing the underlying random numbers ξ with uniform distribution.

The results of the program execution are listed in the last two columns of Table 11.1 and evidence a clear-cut improvement over those not relying on variance reduction (for $w(x) = 1$). In fact, for the same number of sampling points, the standard deviation decreases typically more than 4 times.

The improvement brought about by the importance sampling may also be judged from the plots depicted in Figure 11.1. The left panel plots the estimates of the integral for numbers of sampling points between $n = 100$ and 30,000, and one can notice right away the significantly lower spread of the values obtained with importance sampling, even though both approaches converge to the same result. This behavior is also reflected by the standard deviations σ depicted in the right panel. While the scaling of σ (obtained by regression) is, for both approaches, the one predicted theoretically, namely $\sigma \sim n^{-1/2}$, the uncertainties resulting from the importance sampling can be seen to be shifted to lower values by a factor of approximately 3.7.

The variance reduction is particularly effective when the integrand is itself a product of two functions, $F(x) = w(x)f(x)$. In such a case, if the factor $w(x)$ can be integrated analytically according to Equation 11.9 to yield the dependence $\xi = \xi(x)$, and the latter may be furthermore inverted, then the factor $w(x)$ can be treated as a distribution function. On the basis of a sequence of uniformly distributed random

Listing 11.2 Monte Carlo integration of a Function with Importance Sampling (Python Coding)

```
# One-dimensional Monte Carlo quadrature with variance reduction
from math import *
from random import *

def ranSqrt():
#-------------------------------------------------------------------------------
#  Returns a random number x in the range [0,1) with the distribution
#  w(x) = 3/2 x^(1/2), and the corresponding value w(x)
#-------------------------------------------------------------------------------
   x = pow(random(),2e0/3e0)
   w = 1.5e0 * sqrt(x)
   return (x, w)

def func(x): return x * exp(-x)                                    # integrand

# main

n = eval(input("n = "))                              # number of sampling points

seed()

f1 = f2 = 0e0                                 # quadrature with importance sampling
for i in range(1,n+1):
   (x, w) = ranSqrt()                                  # RNG with distribution w(x)
   if (w):
      f = func(x) / w                                                # integrand
      f1 += f; f2 += f * f                                              # sums

f1 /= n; f2 /= n                                                    # averages
s = f1                                                              # integral
sigma = sqrt((f2-f1*f1)/n)                                 # standard deviation
print("s = ",s," +/- ",sigma)
```

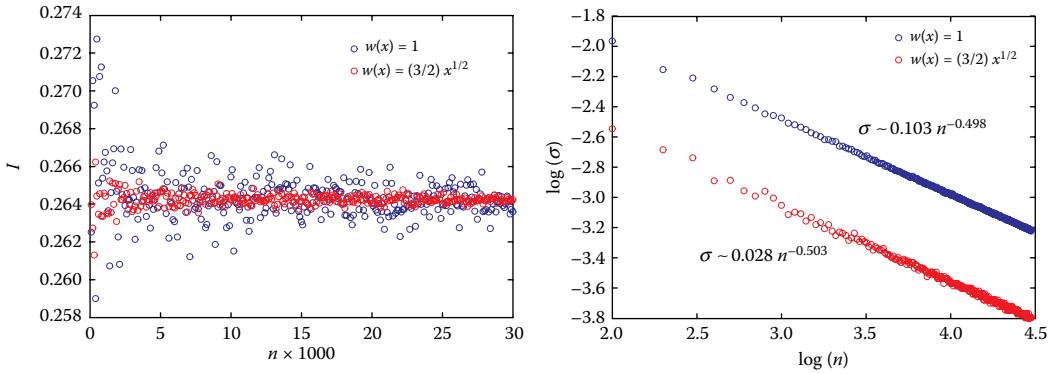

FIGURE 11.1 Monte Carlo estimates of the integral $\int_0^1 xe^{-x}dx$ (left panel) and associated standard deviations σ (right panel) using uniform sampling and, respectively, importance sampling with the distribution $w(x) = (3/2)x^{1/2}$, for numbers of integration points ranging between $n = 100$ and 30,000.

numbers ξ, the inverted relation $x = x(\xi)$ provides a sequence of random variables with the distribution $w(x)$, which is necessary for carrying out the Monte Carlo integration based on Equation 11.11.

In spite of the significant improvement achieved by using adequate weight functions, the Monte Carlo method still remains basically inefficient for one-dimensional integration, even when compared with the classical trapezoidal formula. While the accuracy of the Monte Carlo quadrature barely reaches 10^{-4} for $n = 5000$ sampling points, the trapezoidal formula performs by one order of magnitude better (10^{-5}). Nonetheless, the relative performance of the Monte Carlo integration as compared to the deterministic quadrature schemes qualitatively changes in the case of the multidimensional integration.

11.4 Multidimensional Integrals

As already mentioned, the real virtues of the Monte Carlo integration come plenary to light in multidimensional cases. Even though the description of the integration boundary remains one of the difficult tasks even for the Monte Carlo approach, the fact that the integration points are not generated as separate variables along each coordinate direction, but as tuples within the entire integration domain in a single cycle, renders possible the modeling of more complex frontiers than for deterministic quadratures.

Let us tackle the integration of a function f over a complex domain $\mathcal{D}$ and let us assume that, due to the complexity of the frontier, both expressing the volume V and its exclusive sampling by random points are difficult. To make the Monte Carlo quadrature formula (11.1) applicable, it is necessary to define an easy-to-sample *extended domain* $\widetilde{\mathcal{D}}$, that tightly encloses the original integration domain $\mathcal{D}$, and whose volume $\tilde{V}$ is easily expressible (usually, a hyperparallelepiped). By defining the new integrand

$$\tilde{f}(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{if } \mathbf{x} \in \mathcal{D}, \\ 0 & \text{if } \mathbf{x} \notin \mathcal{D}, \end{cases} \tag{11.15}$$

the quadrature formula becomes

$$\int_{\mathcal{D}} f dV \approx \tilde{V}\langle\tilde{f}\rangle \pm \tilde{\sigma}, \tag{11.16}$$

where the average $\left\langle\tilde{f}\right\rangle$ and the variance $\tilde{\sigma}$ refer this time to the extended volume.

The extended domain $\widetilde{\mathcal{D}}$ is desirable to be as similar as possible with the original domain $\mathcal{D}$ to reduce the relative contribution of the sampling points falling outside $\mathcal{D}$. Corresponding to vanishing integrand values, these points imply trivial information and indirectly deteriorate the standard deviation $\tilde{\sigma}$ by the entailed decrease of the number of effective points.

The Monte Carlo integration is not suitable for "boxed," general-purpose implementations operating with different user-supplied integrands, as in the case of classical quadratures (see Chapter 10). Even though the programs have, in principle, the same general structure, the code is susceptible to additional optimizations by taking advantage of the particular integrand features. To demonstrate the basic ideas, we discuss in what follows a couple of 2D integrals and a 3D integral.

First, we consider evaluating the area of the unit circle from the integral:

$$I = \iint_{x^2+y^2\le 1} dx\, dy = 4\int_0^1 dx \int_0^{\sqrt{1-x^2}} dy = \pi. \tag{11.17}$$

Taking advantage of the problem's symmetry, the integration domain $\mathcal{D}$ is reduced to the circular sector in the first quadrant. With a view to convenient sampling, we define as extended domain the unit square $\widetilde{\mathcal{D}} = [0,1) \times [0,1)$ and we rewrite the integral as

$$I = 4\int_0^1 dx \int_0^1 dy\, H[1-(x^2+y^2)], \tag{11.18}$$

where the points external to the domain $\mathcal{D}$ can be discriminated by using the Heaviside function

$$H(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x \geq 0. \end{cases} \tag{11.19}$$

The corresponding Monte Carlo quadrature formula is thus

$$I \approx \frac{4}{n} \sum_{i=1}^{n} H[1 - (x_i^2 + y_i^2)] = 4\frac{n_i}{n}, \tag{11.20}$$

and it implies n random sampling points (x_i, y_i), uniformly distributed within the square $\widetilde{\mathcal{D}}$. In this case, the sum has obviously the significance of the number n_i of points interior to the circular sector $\mathcal{D}$ and its area is, as can be seen, proportional to the ratio between n_i and the total number of sample points n. In other words, the area of the circular sector is proportional with the probability of generating interior sampling points. A simple calculation shows that the mean integrand and the mean squared integrand are equal in this case, that is, $\langle H \rangle = \langle H^2 \rangle = n_i/n$.

Program 11.3 precisely implements the discussed ideas. The estimates for the unit circle area are listed in Table 11.2 for several numbers of sampling points. For $n = 50{,}000$, the expected π value is reproduced with four exact decimal digits, even though the associated standard error σ suggests a poorer estimate.

Pursuing the idea that for a uniform sampling of the extended domain $\widetilde{\mathcal{D}}$, the ratio of the volumes V and $\tilde{V}$ is approximated by the ratio n_i/n of the corresponding numbers of sampling points; the above

Listing 11.3 Monte Carlo Evaluation of the Area of the Unit Circle (Python Coding)

```
# Monte Carlo calculation of the unit circle area
from math import *
from random import *

# main

n = eval(input("n = "))                              # number of sampling points

seed()

ni = 0                                               # number of interior points
for i in range(1,n+1):
   x = random(); y = random()
   if (x*x + y*y <= 1e0): ni += 1                              # add interior point

fi = ni/n                                          # fraction of interior points
s = 4e0 * fi                                                          # integral
sigma = 4e0 * sqrt((fi - fi*fi)/n)                          # standard deviation
print("Unit circle area = ",s," +/- ",sigma)
```

TABLE 11.2 Monte Carlo Estimates of the Integral $I = \int\int_{x^2+y^2 \leq 1} dx\, dy$ and Associated Standard Deviations σ

n	I	σ	n	I	σ
100	3.20000	0.16000	5,000	3.14400	0.02320
500	3.22400	0.07074	10,000	3.15520	0.01633
1000	3.12800	0.05223	50,000	3.14128	0.00734

technique can be generalized for the evaluation of complicated multidimensional volumes based on the relation:

$$V = \lim_{n\to\infty} \frac{n_i}{n}\,\tilde{V}.$$

Technically, the calculation amounts to register the points interior to the original integration domain along with the total number of points generated within the entire extended domain.

The second 2D example considers the integral:

$$\frac{1}{4\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left(x^2+y^2\right)e^{-(x^2+y^2)/2}dy\,dx = 1. \tag{11.21}$$

Certainly, it can be evaluated using the general Monte Carlo methodology with uniform sampling. However, one may take advantage of the Gaussian factor and perform a more meaningful sampling using integration points generated by the Gaussian distribution:

$$w(x,y) = \frac{1}{2\pi}e^{-(x^2+y^2)/2}. \tag{11.22}$$

Since this probability distribution is normalized to unity, $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} w(x,y)dy\,dx = 1$, the Monte Carlo quadrature amounts to calculating the average

$$\langle\tilde{f}\rangle = \frac{1}{n}\sum_{i=1}^{n}\frac{\tilde{f}(x_i,y_i)}{w(x_i,y_i)} = \frac{1}{2n}\sum_{i=1}^{n}\left(x_i^2+y_i^2\right), \tag{11.23}$$

for arguments sampled according to the normal probability distribution $w(x,y)$. Techniques for generating random numbers with such a distribution are presented in Section 11.5. One of the developed routines is `randNrm2`, which actually returns pairs of random variables `x` and `y`, along with the corresponding value `w` of the distribution function.

Program 11.4 comparatively uses uniform and Gaussian sampling. Results illustrating the variance reduction for the integral in question are depicted in Figure 11.2. The integral estimates obtained with importance sampling are seen in the left panel to feature a significantly lower spread than for uniform

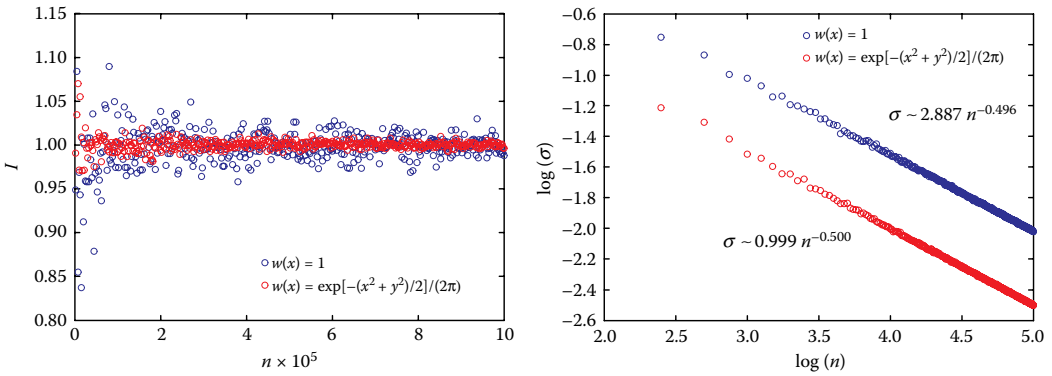

FIGURE 11.2 Monte Carlo estimates of the integral $(1/4\pi)\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(x^2+y^2)e^{-(x^2+y^2)/2}dy\,dx = 1$ (left panel) and associated standard deviations σ (right panel) using uniform and, respectively, Gaussian sampling, for numbers of integration points ranging between $n = 250$ and 100,000.

Listing 11.4 2D Monte Carlo Integration with Importance Sampling (Python Coding)

```
# Two-dimensional Monte Carlo quadrature with variance reduction
from math import *
from random import *
from random1 import *

def func(x,y):                                                  # integrand
   return (x*x + y*y)*exp(-0.5e0*(x*x + y*y))/(4e0*pi)

# main

L = 8e0                                  # integration domain [-L,L] x [-L,L]
L2 = L * L                                          # area of sampling domain

n = 100000                                        # number of sampling points

seed()

f1 = f2 = 0e0                              # quadrature with uniform sampling
for i in range(1,n+1):
   x = L * random(); y = L * random()
   f = func(x,y)                                                # integrand
   f1 += f; f2 += f * f                                             # sums

f1 /= n; f2 /= n                                                 # averages
s = 4e0 * L2 * f1                                                # integral
sigma = 4e0 * L2 * sqrt((f2-f1*f1)/n)                     # standard deviation
print("Uniform sampling : s = ",s," +/- ",sigma)

f1 = f2 = 0e0                             # quadrature with importance sampling
for i in range(1,n+1):
   (w, x, y) = randNrm2()          # random numbers with normal distribution
   f = func(x,y) / w                                            # integrand
   f1 += f; f2 += f * f                                             # sums

f1 /= n; f2 /= n                                                 # averages
s = f1                                                           # integral
sigma = sqrt((f2-f1*f1)/n)                                # standard deviation
print("Gaussian sampling: s = ",s," +/- ",sigma)
```

sampling. A further confirmation of the better performance of the Gaussian sampling is provided by the linear fits of the log–log plots of the standard deviations as functions of the number of sampling points, which indicate a reduction of the uncertainties roughly by a factor of 2.9 when using normally instead of uniformly distributed random numbers.

Let us consider as an application of the Monte Carlo methodology to 3D quadratures the evaluation of the mass and mass center position of a *torus* of major radius $R = 3$ and minor radius $r = 1$ (see Figure 11.3). The torus is centered at the origin and is symmetric about the z-axis. A similar problem was treated in Chapter 10 by using deterministic quadratures. The equation of the toroidal surface is

$$\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2. \tag{11.24}$$

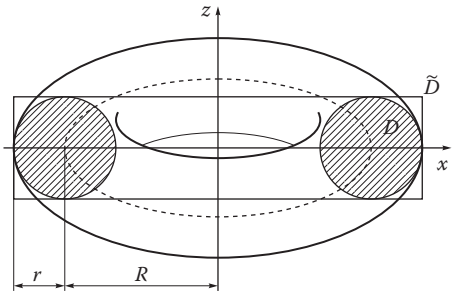

FIGURE 11.3 Integration domain and coordinate system for the Monte Carlo calculation of the mass of a torus and its mass center position.

The radial distances from a given point (x, y, z) to the z-axis and, respectively, to the tube axis are given by

$$R' = \sqrt{x^2 + y^2}, \quad r' = \sqrt{(R - R')^2 + z^2}. \tag{11.25}$$

The condition for a point to be *inside the torus* can be simply expressed as

$$r' \leq r. \tag{11.26}$$

As a density function, we consider

$$f(x, y, z) = \begin{cases} (1 - r'/r)^2 & r' \leq r, \\ 0 & r' > r, \end{cases} \tag{11.27}$$

which depends only on the minor radial distance r' and decreases parabolically from 1 at the tube axis to 0 at the torus' surface. The integral of the density yields the mass and amounts to $m = \frac{1}{3}\pi^2 R r^2 \approx 9.8696044$. As for the mass center, due to the symmetry, it is obviously located at the origin: $x_c = y_c = z_c = 0$.

As an extended integration domain, we consider the parallelepiped tightly enclosing the torus, $\widetilde{\mathcal{D}} = [-R - r, R + r] \times [-R - r, R + r] \times [-r, r]$. The mass and the coordinates of the mass center are given, respectively, by

$$m = \int_{\widetilde{\mathcal{D}}} \rho \, dx \, dy \, dz,$$

$$x_c = \frac{1}{m} \int_{\widetilde{\mathcal{D}}} \rho x \, dx \, dy \, dz, \quad y_c = \frac{1}{m} \int_{\widetilde{\mathcal{D}}} \rho y \, dx \, dy \, dz, \quad z_c = \frac{1}{m} \int_{\widetilde{\mathcal{D}}} \rho z \, dx \, dy \, dz.$$

The Monte Carlo evaluation of these integrals can be accomplished according to the relations:

$$m \approx \tilde{V}\langle\rho\rangle \pm \tilde{V}\sqrt{\left[\langle\rho^2\rangle - \langle\rho\rangle^2\right]/n}, \tag{11.28}$$

$$x_c \approx (\tilde{V}/m)\langle\rho x\rangle \pm (\tilde{V}/m)\sqrt{\left[\langle(\rho x)^2\rangle - \langle\rho x\rangle^2\right]/n}, \tag{11.29}$$

$$y_c \approx (\tilde{V}/m)\langle\rho y\rangle \pm (\tilde{V}/m)\sqrt{\left[\langle(\rho y)^2\rangle - \langle\rho y\rangle^2\right]/n}, \tag{11.30}$$

$$z_c \approx (\tilde{V}/m)\langle\rho z\rangle \pm (\tilde{V}/m)\sqrt{\left[\langle(\rho z)^2\rangle - \langle\rho z\rangle^2\right]/n}. \tag{11.31}$$

On the basis of the random variables ξ_i, η_i, and ζ_i, distributed uniformly in the interval $[0, 1)$, one can generate random points (x_i, y_i, z_i) uniformly distributed within the domain $\widetilde{\mathcal{D}}$ as

$$x_i = (2\xi_i - 1)\,(R + r), \quad y_i = (2\eta_i - 1)\,(R + r), \quad z_i = (2\zeta_i - 1)\,r. \tag{11.32}$$

The program in Listing 11.5 implements relations 11.28 through 11.32, and the particular functional form of the density is coded in function Func.

Listing 11.5 Monte Carlo Evaluation of the Mass Center of a Torus (Python Coding)

```
# Monte Carlo calculation of the mass center of a torus of radii R and r,
# centered at the origin and with Oz as symmetry axis
from math import *
from random import *

def Func(x, y, z):
   global R, r
   Rp = sqrt(x*x + y*y)                              # major radial distance
   dR = R - Rp
   rp = sqrt(dR*dR + z*z)                            # minor radial distance
   dr = 1e0 - rp/r
   return (dr*dr if rp <= r else 0e0)                          # zero-padding

# main

R = 3e0; r = 1e0                                  # major & minor torus radii
Lx = Ly = R + r; Lz = r                         # extended domain: 1st octant
V = 8e0 * Lx * Ly * Lz                       # volume of total extended domain

n = 10000000                                      # number of sampling points

seed()

sm  = sx  = sy  = sz  = 0e0
sm2 = sx2 = sy2 = sz2 = 0e0
for i in range(1,n+1):
   x = Lx * (2e0*random() - 1e0)                                # -Lx <= x <= Lx
   y = Ly * (2e0*random() - 1e0)                                # -Ly <= y <= Ly
   z = Lz * (2e0*random() - 1e0)                                # -Lz <= x <= Lz
   dens = Func(x,y,z)                                                 # density
   if (dens):
      f = dens    ; sm += f; sm2 += f * f                                 # sums
      f = dens * x; sx += f; sx2 += f * f
      f = dens * y; sy += f; sy2 += f * f
      f = dens * z; sz += f; sz2 += f * f

sm /= n; sx /= n; sy /= n; sz /= n                                  # averages
m  = V * sm; sigm = V * sqrt((sm2/n - sm*sm)/n); f = V/m           # integrals
xc = f * sx; sigx = f * sqrt((sx2/n - sx*sx)/n)
yc = f * sy; sigy = f * sqrt((sy2/n - sy*sy)/n)
zc = f * sz; sigz = f * sqrt((sz2/n - sz*sz)/n)

print("m  = {0:8.5f} +/- {1:8.5f}".format(m ,sigm))
print("xc = {0:8.5f} +/- {1:8.5f}".format(xc,sigx))
print("yc = {0:8.5f} +/- {1:8.5f}".format(yc,sigy))
print("zc = {0:8.5f} +/- {1:8.5f}".format(zc,sigz))
```

TABLE 11.3 Monte Carlo Estimations of the Mass and Mass Center Position of a Torus of Major Radius $R = 3$ and Minor Radius $r = 1$

n	m	x_c	y_c	z_c
100,000	9.96246	0.02386	0.02089	0.00139
1,000,000	9.87478	0.00374	0.00026	0.00024
10,000,000	9.86233	0.00114	0.00236	0.00006
100,000,000	9.86754	−0.00027	−0.00005	0.00011

As can be seen from Table 11.3, the Monte Carlo estimate of the mass features only three exact significant digits for $n = 100{,}000{,}000$ integration points, which reflects the rather slow convergence of the method. Anyway, the simplicity of the methodology and the invariant scaling law of the associated uncertainties, independent of dimensionality, makes Monte Carlo the method of choice for high-dimension integrals.

11.5 Generation of Random Numbers

The preceding sections have shown that the Monte Carlo integration of a function $f(x)$ basically implies

- Generation of sampling points with a certain distribution $w(x)$ within the integration domain.
- Evaluation of the modified integrand $f(x)/w(x)$ for the generated sampling points.

While the evaluation of the integrand does not require, in general, any techniques in addition to those presented in Chapter 5, the generation of random variables with a given distribution may require special treatment.

Randomness is not an attribute of an individual number, but rather of a *sequence* of (ideally) uncorrelated values characterized by a given distribution function.

The use of computers, which are intrinsically deterministic machines, for generating random numbers may appear deprived of sense. In fact, a sequence of numbers produced by a computer can be just *pseudorandom*, since it is completely determined by the first number, called *seed*, and the underlying algorithm. Nevertheless, the reduced sequential correlation enables one to use pseudorandom sequences successfully in most applications instead of the genuinely random sequences. Therefore, we commonly refer to computer-generated pseudorandom numbers simply by "random numbers" and, within the limits of the associated statistical errors, the applications using them should produce compatible results for different random number generators.

Truly random quantities can only result from physical processes, such as, for instance, the decay of the nuclei from a radioactive sample. Since one can neither predict which of the nuclei will decay, nor the moment of time when this will happen, a genuinely random quantity is the time interval between two consecutive radioactive decays.

The random number generator for sequences with *uniform distribution* is the basic building block for any algorithm based on random variables, even in the case of nonuniform distributions. The uniformity of the distribution implies the generation of the numbers with *equal probability* in any equal-sized subintervals of the definition domain. The generators most commonly implemented by high-level programming languages are *linear congruential generators*, which, starting from a *seed*, x_0, "grow" by repeated application of an entire sequence of random numbers x_i, using a recurrence relation of the form:

$$x_{i+1} = (ax_i + c) \mod m. \tag{11.33}$$

a, c, and m are integers, named *multiplier*, *increment*, and *modulus*, respectively, with the latter being typically large, since it determines the period length of the sequence, that is, the number of iterations

after which the values repeat themselves. Obviously, the same "magic numbers" a, c, and m and the same seed always lead to the same sequence.

The random sequence generated using the recurrence relation 11.33 is confined between 0 and $m-1$ and may repeat itself with a period inferior to m. For an optimal choice of the parameters a, c, and m, the period may be maximized and become equal to m. In such a case, *all* the integers between 0 and $m-1$ appear in a certain order, no matter which of them was used as a seed.

In practical implementations, the value employed for the modulus m has to reconcile two contradictory requirements: on the one hand, m needs to be large enough to lead to the longest possible sequence of distinct values, and, on the other, the product of m, as the upper limit of the generated numbers, and the multiplier a should not exceed the machine representation of integers. To generate real random numbers in the unit interval $[0, 1)$, one may use the fractions x_{j+1}/m.

The function `randLCG1` given in Listing 11.6 generates real random deviates in the range $[0, 1)$ using a linear congruential generator with the coefficients a, c, and m defined in Press et al. (2007). With a view to extending the length of the generated sequence, the routine `randLCG1` employs long (32-bit) integers. The sequence may be reinitialized at any time by calling the routine with the argument `iopt` set to 0, which causes the local control variable of the sequence, `irnd`, to be reinitialized by a call to the built-in integer random number generator.

Figure 11.4 represents as histogram the distribution of 100,000 random numbers generated with the help of routine `randLCG1`. The roughly constant distribution envelope suggests that, despite its simplicity, the featured generator produces deviates with a reduced degree of sequential correlation and remarkable uniformity.

Listing 11.6 Linear Congruential Generators (Python Coding)

```
#==========================================================================
def randLCG1(iopt=1):
#--------------------------------------------------------------------------
#  Linear Congruential Generator for real random numbers in the range [0,1)
#  Press, Teukolsky, Vetterling, Flannery, Numerical Recipes 3rd Ed., 2007
#  iopt = 0 initializes the sequence
#--------------------------------------------------------------------------
   global irnd                                        # conserved between calls
   a = 8121; c = 28411; m = 134456

   if (iopt == 0): irnd = randrange(0xFFFFFFFF)             # initialization

   irnd = (a * irnd + c) % m
   return irnd/m

#==========================================================================
def randLCG2(iopt=1):
#--------------------------------------------------------------------------
#  Linear Congruential Generator for real random numbers in the range [0,1)
#  D. Rapaport, The Art of Molecular Dynamics Simulation, Cambridge, 2004
#  iopt = 0 initializes the sequence
#--------------------------------------------------------------------------
   global irnd                                        # conserved between calls
   a = 314159269; c = 453806245; m = 2147483647

   if (iopt == 0): irnd = randrange(0xFFFFFFFF)             # initialization

   irnd = (a * irnd + c) & m
   return irnd/m
```

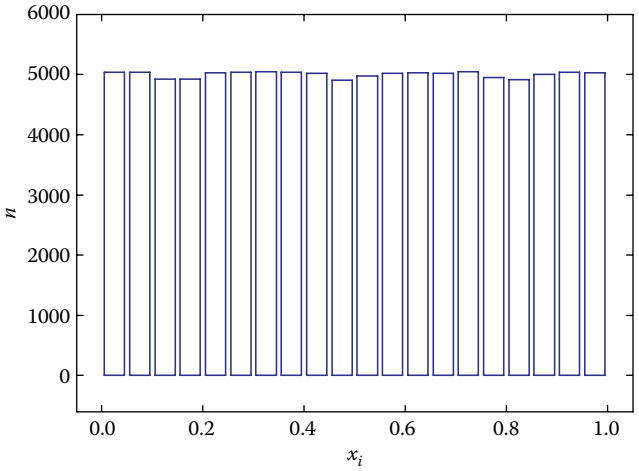

FIGURE 11.4 Distribution of 100,000 random numbers generated with the function `randLCG1`.

The binning of the successively generated random values can be carried out by calling the routine `HistoBin` presented in Chapter 3. Performing the binning in parallel with the generation of the random numbers is preferable to the prior generation and storage of the entire sequence, followed by binning, since it requires significantly less storage. To effectively plot the histogram, in the end, one can call the routine `Plot` with the style parameter `sty` set to 4.

Another, equally simple and efficient linear congruential approach uses the bit-wise AND operator & instead of the integer modulo operation (Rapaport 2004):

$$x_{i+1} = (ax_i + c) \,\&\, m, \tag{11.34}$$

and is coded as function `LCG2` in Listing 11.6.

George Marsaglia, a mathematician and computer scientist who earned his fame for outstanding contributions to the development of random number generators, described in 1999 in a post on the *Sci. Stat.*

Listing 11.7 Multiply-with-Carry Generator of George Marsaglia (Python Coding)

```
#==========================================================================
def randMCG(iopt=1):
#--------------------------------------------------------------------------
#  Multiply-with-Carry Generator for real random numbers in the range [0,1)
#  George Marsaglia, post to Sci. Stat. Math, 1999
#  iopt = 0 initializes the sequence
#--------------------------------------------------------------------------
   global irnd1, irnd2                                # conserved between calls

   if (iopt == 0):                                                # initialization
      irnd1 = randrange(0xFFFFFFFF); irnd2 = randrange(0xFFFFFFFF)

   irnd1 = 36969 * (irnd1 & 0xFFFF) + (irnd1 >> 0xF)
   irnd2 = 18000 * (irnd2 & 0xFFFF) + (irnd2 >> 0xF)
   return (((irnd1 << 0xF) + irnd2) & 0xFFFFFFFF)/0xFFFFFFFF
```

Math forum a series of eight high-quality 32-bit random number generators. One of these uses the so-called *multiply-with-carry* method to generate a sequence of uniformly distributed random integers based on two seed values. Essentially, this fast generator, implemented as function `randMCG` in Listing 11.7, concatenates two 16-bit multiply-with-carry generators and has an extremely long period of about 2^{60}.

The hexadecimal value `0xFFFF` (decimal 65,535) is used in bit-wise AND operations to produce from the integers `irand1` and `irand2` two 16-bit integer values, which are then combined. 0xFFFFFFFF is the largest representable 32-bit integer value. An entry value of 0 of the parameter `iopt` causes the routine to reinitialize the random sequence by calling the built-in integer random number generator.

Quite frequently in practice, random numbers with nonuniform distributions are required. Such an example was considered in Section 11.3, where the method of importance sampling was exposed. Let us now deal in this context with the integral

$$I = \int_0^\infty f(x)dx \equiv \int_0^\infty g(x)e^{-x}dx. \tag{11.35}$$

Importance sampling can be carried out in this case by considering the normalized weight function

$$w(x) = e^{-x}, \tag{11.36}$$

and performing the change of variable

$$\xi(x) = \int_0^x e^{-x'}dx' = 1 - e^{-x}.$$

Inversion of the above relation leads to

$$x(\xi) = -ln(1 - \xi), \tag{11.37}$$

and enables one to express the integral as

$$I = \int_0^\infty g(x(\xi))d\xi. \tag{11.38}$$

Applying the Monte Carlo quadrature to the function $g(x) \equiv f(x)/w(x)$ leads to a significant variance reduction as compared to function $f(x)$, at the expense, however, of having to evaluate $g(x)$ for random arguments $x(\xi_i)$ with *exponential distribution*. The `randExp` function listed below generates numbers with such a distribution based on Relation 11.37, while producing the intermediate numbers with uniform distribution ξ_i by means of the function `random()` (presented in Section 11.2). Figure 11.5 illustrates the distribution of 100,000 random numbers generated by function `randExp`, which are binned in 20 subintervals in the range $[0, 4]$.

```
# Generates random numbers with exponential distribution
def randExp(): -log(1e0 - random())
```

One of the frequently used distribution functions in applications based on Monte Carlo techniques, also playing a primordial role in the probability theory, is the *normal* or *Gaussian distribution*:

$$w(x) = (2\pi)^{-1/2}e^{-x^2/2}. \tag{11.39}$$

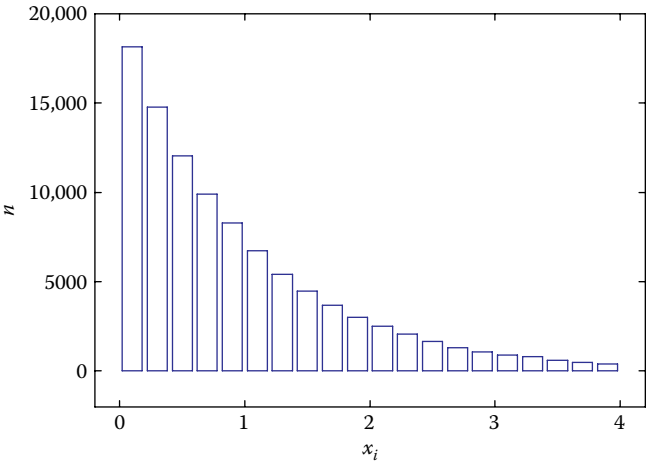

FIGURE 11.5 Distribution of 100,000 random numbers generated with the function `randExp`.

In principle, random numbers with such a distribution can be generated by inverting its incomplete integral, namely, the *error function*,

$$\xi(x) = (2\pi)^{-1/2} \int_0^x e^{-x'^2/2} dx',$$

by using, for example, a polynomial approximation thereof. However, this method lacks efficiency and we present, therefore, two alternative algorithms.

An efficient method for generating variables with normal distribution is based on the *central limit theorem*, according to which, the mean of a large number of uniformly distributed random variables approximates a normal distribution. Having in view that the mean of, for example, 12 random numbers uniformly distributed in the interval $[0, 1)$ is 6 and the corresponding standard deviation equals 1, the function `randNrm` (Listing 11.8) returns a random variable with normal distribution, featuring a vanishing mean and unitary standard deviation. As an illustration, Figure 11.6 shows the distribution of 100,000 values generated by function `randNrm`.

Another efficient technique of generating random deviates with normal distribution is known as the *Box–Muller method* (Box and Muller 1958) and is based on the 2D Gaussian distribution:

$$w(x, y) = \frac{1}{2\pi} e^{-(x^2+y^2)/2} \equiv \frac{1}{2\pi} e^{-r^2/2},$$

with $x = r\cos\theta$ and $y = r\sin\theta$ in polar coordinates. Performing the change of variable $u = r^2/2$, one may rewrite this distribution as

$$w(u, \theta) = \frac{1}{2\pi} e^{-u}.$$

Accordingly, by generating for u exponentially distributed deviates in the range $[0, \infty)$ and for θ uniformly distributed values in the interval $[0, 2\pi]$, the corresponding Cartesian projections,

$$x = (2u)^{1/2} \cos\theta, \quad y = (2u)^{1/2} \sin\theta, \tag{11.40}$$

are *normally distributed*. Function `randNrm2` (Listing 11.8) uses these relations to generate two random deviates, `x` and `y`, with normal distribution. The intermediate variable u with exponential distribution,

Listing 11.8 Random Number Generators with Normal Distribution (Python Coding)

```
#=============================================================================
def randNrm():
#-----------------------------------------------------------------------------
#  Returns random numbers with normal distribution
#  w = exp(-0.5e0*x*x) / sqrt(2*pi)
#  using the central limit theorem and sums of 12 uniform random numbers
#-----------------------------------------------------------------------------
   sum = 0e0
   for i in range(1,13): sum += random()
   x = sum - 6e0
   w = 0.398942280401433 * exp(-0.5e0*x*x)         # 1/sqrt(2*pi) = 0.3989...
   return (w, x)

#=============================================================================
def randNrm2():
#-----------------------------------------------------------------------------
#  Returns 2 random numbers (x,y) with normal distribution
#  w = exp(-(x*x+y*y)/2) / (2*pi)
#  and the corresponding distribution value
#-----------------------------------------------------------------------------
   r2 = -log(1e0 - random())            # exponential distribution for r**2/2
   w = exp(-r2) / (2e0*pi)                        # distribution function value
   r = sqrt(2e0*r2); theta = 2e0 * pi * random()          # polar coordinates
   x = r * cos(theta); y = r * sin(theta)             # Cartesian projections
   return (w, x, y)
```

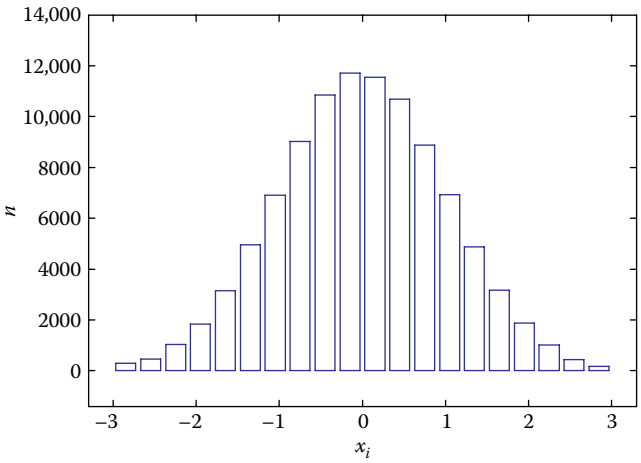

FIGURE 11.6 Distribution of 100,000 random numbers generated with the function `randNrm`.

though not coded as an explicit program variable, is generated by the technique also employed in the case of the routine `randExp()`.

The methodology of variance reduction discussed in Section 11.3 may be directly applied for generating random numbers with particular distributions only provided that the corresponding integral relations prescribing the change of variable are invertible. Unfortunately, in many applications, the inversion is difficult, if at all possible. Furthermore, if the distribution is multidimensional, the difficulties may become

insurmountable. There is an obvious need for algorithms avoiding the inversion of the integral change of variable and solely implying the evaluation of the distribution function itself. One of the most representative algorithms of this category, used successfully in a wide range of topical problems of applied statistics, is the so-called *Metropolis algorithm* (Metropolis et al. 1953; Beichl and Sullivan 2000).

Let us assume that it is required to generate a sequence of random points $\mathbf{x_i}$, distributed according to the probability density $w(\mathbf{x})$ in a certain (possibly multidimensional) space. The Metropolis algorithm generates these random deviates as angular points of a so-called *random walk*.

The rules according to which the random walk is performed reflect its character of Markow chain. This means that, having reached a given point $\mathbf{x}_i$ in the sequence, the probability density $w(\mathbf{x})$ completely determines the next point, $\mathbf{x}_{i+1}$, and no memory of the previous states is involved. Specifically, to determine $\mathbf{x}_{i+1}$, one first performs a *trial step* to a point $\mathbf{x}_t = \mathbf{x}_i + \delta$, randomly chosen within a hypercube of predefined size δ, centered about the starting point $\mathbf{x}_i$.

Defining the ratio between the trial and the old value of the distribution function,

$$r = \frac{w(\mathbf{x}_t)}{w(\mathbf{x}_i)}, \tag{11.41}$$

if $r \geq 1$, the trial step is accepted and one sets $\mathbf{x}_{i+1} = \mathbf{x}_t$, favoring moves toward regions with higher probability density $w(\mathbf{x})$. Otherwise, if $r < 1$, the trial step is accepted only conditionally, namely, with the probability r. Technically, this amounts to generating a random variable ρ with uniform distribution in $[0, 1)$, comparing it with r, and accepting the trial step only if

$$\rho \leq r.$$

Actually, this can be used as the sole acceptance condition of the trial step, since it combines both cases $r \geq 1$ and $r < 1$. If the trial step is rejected, one maintains the starting point, setting $\mathbf{x}_{i+1} = \mathbf{x}_i$. Once $\mathbf{x}_{i+1}$ is determined, one can pass to generating the following point of the sequence, according to the same rules. One can choose as origin of the entire random walk any point of the respective space.

Finally, we present in Listing 11.9 the routine `randMet`, which illustrates the described algorithm. The routine receives as input parameters the actual name of the distribution function `w`, the size `del` of the sampling hypercube, and the option parameter `iopt`, which enables one to (re)initialize the sequence if it is set to 0. Every call to the routine executes a single step of the random walk, returning based on

Listing 11.9 Random Number Generator Based on the Metropolis Method (Python Coding)

```
#=============================================================================
def randMet(w, delta, iopt=1):
#-----------------------------------------------------------------------------
#  Generates random numbers x with the distribution w(x) by the Metropolis
#  method, using a maximal trial step size delta
#  iopt = 0 initializes the sequence
#-----------------------------------------------------------------------------
   global xrnd                                        # conserved between calls

   if (iopt == 0): xrnd = 2e0*random() - 1e0                # initialization

   dx = delta * (2e0*random() - 1e0)                       # trial step size
   if (random() <= w(xrnd+dx)/w(xrnd)): xrnd += dx   # accept with prob. w(x)

   return xrnd
```

the starting value `xrnd` (stored locally between the calls) a new value corresponding to the distribution function $w(x)$. The acceptance condition of the trial step is checked by comparing the ratio between the trial and the old values of the distribution function with a subunitary random number generated by calling the routine `random()`.

The distribution of 100,000 random deviates generated by the routine `randMet` for the Gaussian distribution $w(x) = exp(-x^2/2)$ is similar with the one depicted in Figure 11.6.

11.6 Implementations in C/C++

Listing 11.10 shows the content of the file `random.h`, which contains equivalent C/C++ implementations of the Python functions developed in the main text and included in the module `random1.py`. The corresponding routines have identical names, parameters, and functionalities.

Listing 11.10 Routines for Generating Random Numbers with Various Distributions (`random.h`)

```
//------------------------------- random.h --------------------------------
// Contains routines for generating pseudo-random numbers.
// Part of the numxlib numerics library. Author: Titus Beu, 2013
//-------------------------------------------------------------------------
#ifndef _RANDOM_
#define _RANDOM_

#include <stdlib.h>
#include <time.h>
#include <math.h>

// Initializes the random number generator using the current system time
#define seed() srand((unsigned)time(NULL))

// Generates random floating point numbers in the range [0,1)
#define random() ((double)rand()/(RAND_MAX+1))

// Generates random numbers with exponential distribution
#define randExp() -log(fabs(1e0 - random()))

//=========================================================================
double randLCG1(int iopt)
//-------------------------------------------------------------------------
// Linear Congruential Generator for real random numbers in the range [0,1)
// Press, Teukolsky, Vetterling, Flannery, Numerical Recipes 3rd Ed., 2007
// iopt = 0 initializes the sequence
//-------------------------------------------------------------------------
{
   const unsigned int a = 8121, c = 28411, m = 134456;
   static unsigned int irnd;                      // conserved between calls

   if (iopt == 0) irnd = rand();                            // initialization

   irnd = (a * irnd + c) % m;
   return (double)irnd/m;
}

//=========================================================================
double randLCG2(int iopt)
//-------------------------------------------------------------------------
// Linear Congruential Generator for real random numbers in the range [0,1)
// D. Rapaport, The Art of Molecular Dynamics Simulation, Cambridge, 2004
```

```
// iopt = 0 initializes the sequence
//---------------------------------------------------------------------------
{
   const unsigned int a = 314159269, c = 453806245, m = 2147483647;
   static unsigned int irnd;                        // conserved between calls

   if (iopt == 0) irnd = rand();                              // initialization

   irnd = (a * irnd + c) & m;
   return (double)irnd/m;
}

//===========================================================================
double randMCG(int iopt)
//---------------------------------------------------------------------------
// Multiply-with-Carry Generator for real random numbers in the range [0,1)
// George Marsaglia, post to Sci. Stat. Math, 1999
// iopt = 0 initializes the sequence
//---------------------------------------------------------------------------
{
   static unsigned int irand1, irand2;             // conserved between calls

   if (iopt == 0) { irand1 = rand(); irand2 = rand(); }     // initialization

   irand1 = 36969 * (irand1 & 0xFFFF) + (irand1 >> 16);
   irand2 = 18000 * (irand2 & 0xFFFF) + (irand2 >> 16);
   return (double)((irand1 << 16) + irand2)/0xFFFFFFFF;
}

//===========================================================================
double randNrm(double &w)
//---------------------------------------------------------------------------
// Returns random numbers with normal distribution
// w = exp(-0.5e0*x*x) / sqrt(2*pi)
// using the central limit theorem and sums of 12 uniform random numbers
//---------------------------------------------------------------------------
{
   double sum, x;
   int i;

   sum = 0e0;
   for (i=1; i<=12; i++) sum += random();
   x = sum - 6e0;
   w = 0.398942280401433 * exp(-0.5e0*x*x);       // 1/sqrt(2*pi) = 0.3989...
   return x;
}

//===========================================================================
void randNrm2(double &w, double &x, double &y)
//---------------------------------------------------------------------------
// Returns 2 random numbers (x,y) with normal distribution
// w = exp(-(x*x+y*y)/2) / (2*pi)
// and the corresponding distribution value
//---------------------------------------------------------------------------
{
#define pi2 6.283185307179586
   double r, r2, theta;

   r2 = -log(1e0 - random());          // exponential distribution for r**2/2
```

```
   w = exp(-r2) / pi2;                          // distribution function value
   r = sqrt(2e0*r2); theta = pi2 * random();              // polar coordinates
   x = r * cos(theta); y = r * sin(theta);           // Cartesian projections
}

//============================================================================
double randMet(double w(double), double delta, int iopt)
//----------------------------------------------------------------------------
// Generates random numbers x with the distribution w(x) by the Metropolis
// method, using a maximal trial step size delta
// iopt = 0 initializes the sequence
//----------------------------------------------------------------------------
{
   static double dx, xrnd;                          // conserved between calls

   if (iopt == 0) xrnd = random();                           // initialization

   dx = delta * (2e0*random() - 1e0);                      // trial step size
   if (random() <= w(xrnd+dx)/w(xrnd)) xrnd += dx;  // accept with prob. w(x)

   return xrnd;
}

#endif
```

11.7 Problems

The Python and C/C++ programs for the following problems may import the functions developed in this chapter from the modules `random1.py` and, respectively, `random.h`, which are available as supplementary material. For creating runtime plots, the graphical routines contained in the libraries `graphlib.py` and `graphlib.h` may be employed.

PROBLEM 11.1

Consider the one-dimensional integrals:

$$\frac{1}{\sqrt{2\pi}}\int_0^\infty e^{-x^2/2}dx = \frac{1}{2}, \tag{11.42}$$

$$\frac{1}{\sqrt{2\pi}}\int_0^\infty x^4 e^{-x^2/2}dx = \frac{3}{2}. \tag{11.43}$$

a. Evaluate the integrals by Monte Carlo techniques, applying comparatively uniform and exponential sampling (using function `randExp`). Consider numbers of sampling points ranging from $n = 250$ to 100,000, and plot the integral estimates and the associated standard deviations.
b. Explain the significant variance reduction when using exponential sampling for the first integral and the lack of effect in the case of the second integral.
c. Use the routine `LinFit` (from `modfunc.py`) to perform linear regression and extract the scaling laws of the standard deviations for the two probability distributions from their log–log plots.

Solution

The Python implementation is provided in Listing 11.11 and the C/C++ version is available as supplementary material (`P11-MC-1Dx.cpp`). The plots of the Gaussian integral 11.42 and of the corresponding standard deviations are shown in Figure 11.7.

Listing 11.11 Monte Carlo Integration of a Function with Importance Sampling (Python Coding)

```
# One-dimensional Monte Carlo quadrature with variance reduction
from math import *
from random import *
from random1 import *
from modfunc import *
from graphlib import *

wnorm = 1e0/sqrt(2e0*pi)
def func(x): return wnorm * exp(-0.5e0*x*x)                          # integrand

# main

L = 8e0                                                  # integration domain [0,L)

nn  = [0]*3                                               # ending indexes of plots
col = [""]*3                                                     # colors of plots
sty = [0]*3                                                      # styles of plots

np = 400                                                # number of plotting points
x1 = [0]*(2*np+1); y1 = [0]*(2*np+1)                             # plotting points
x2 = [0]*(2*np+1); y2 = [0]*(2*np+1); sig = [0]*(2*np+1)

seed()

out = open("mcarlo.txt","w")                                  # open output file
out.write("     n       Int       sig      Int_w     sig_w\n")

for ip in range(1,np+1):
   n = 250 * ip                                       # number of sampling points

   f1 = f2 = 0e0                                 # quadrature with uniform sampling
   for i in range(1,n+1):
      x = L * random()                 # RNG with uniform distribution in [0,L)
      f = func(x)                                                    # integrand
      f1 += f; f2 += f * f                                                # sums

   f1 /= n; f2 /= n                                                   # averages
   s = L * f1                                                         # integral
   sigma = L * sqrt((f2-f1*f1)/n)                             # standard deviation
   out.write(("{0:8d}{1:10.5f}{2:10.5f}").format(n,s,sigma))
   x1[ip] =       n ; y1[ip] = s
   x2[ip] = log10(n); y2[ip] = log10(sigma)

   f1 = f2 = 0e0                           # quadrature with importance sampling
   for i in range(1,n+1):
      x = randExp()                          # RNG with exponential distribution
      f = func(x) / exp(-x)                                          # integrand
      f1 += f; f2 += f * f                                                # sums

   f1 /= n; f2 /= n                                                   # averages
   s = f1                                                             # integral
   sigma = sqrt((f2-f1*f1)/n)                                 # standard deviation
```

```
    out.write(("{0:10.5f}{1:10.5f}\n").format(s,sigma))
    x1[np+ip] =       n ; y1[np+ip] = s
    x2[np+ip] = log10(n); y2[np+ip] = log10(sigma)

out.close()
                                                        # linear regression
(a, b, sigma, sigmb, chi2) = LinFit(x2[0:],y2[0:],sig,np,0)
print("sigma = {0:6.3f} n**({1:6.3f})  w(x) = 1".format(pow(10e0,b),a))

(a, b, sigma, sigmb, chi2) = LinFit(x2[np:],y2[np:],sig,np,0)
print("sigma = {0:6.3f} n**({1:6.3f})  w(x) = exp(-x)". \
      format(pow(10e0,b),a))

GraphInit(1200,600)

nn[1] =   np; col[1] = "blue"; sty[1] = 0                  # uniform sampling
nn[2] = 2*np; col[2] = "red" ; sty[2] = 0                # importance sampling
MultiPlot(x1,y1,y1,nn,col,sty,2,10,0e0,0e0,0,0e0,0e0,0,
          0.10,0.45,0.15,0.85,"n","Int","Monte Carlo - integral")

MultiPlot(x2,y2,y2,nn,col,sty,2,10,0e0,0e0,0,0e0,0e0,0,
          0.60,0.95,0.15,0.85,"log(n)","log(sig)",
          "Monte Carlo - standard deviation")

MainLoop()
```

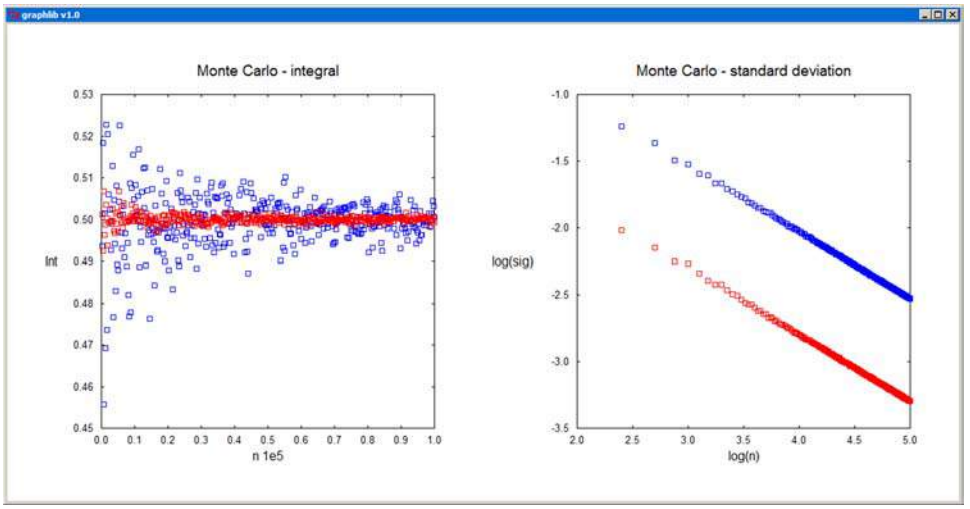

FIGURE 11.7 Monte Carlo estimates of integral 11.42 (left panel) and associated standard deviations (right panel) using uniform and, respectively, exponential sampling.

PROBLEM 11.2

Consider the 2D integral:

$$\frac{1}{4\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left(x^2+y^2\right)e^{-(x^2+y^2)/2}dy\,dx. \tag{11.44}$$

a. Evaluate the integral by Monte Carlo quadratures, applying for the x and y coordinates comparatively uniform sampling, Gaussian sampling (routine `randNrm2`), and exponential sampling (routine `randExp`). Taking into account the symmetry of the problem, restrict for exponential sampling the integration domain to the first quadrant and multiply the results accordingly. Consider numbers of sampling points ranging between $n = 250$ and 100,000, and plot the integral values and the associated standard deviations.
b. Use the routine `LinFit` (from `modfunc.py`) to fit the log–log plots of the standard deviations, and verify that these scale as $n^{-1/2}$ independent of the employed sampling technique. Judging by the lower leading factor in the corresponding scaling law, explain the apparently better performance of exponential sampling.

Solution

The Python implementation is given in Listing 11.12 and the C/C++ version is available as supplementary material (`P11-MC-2Dx.cpp`). The plots of integral (11.44) and of the corresponding standard deviations for uniform, Gaussian, and exponential sampling are shown in Figure 11.8.

Listing 11.12 2D Monte Carlo Integration with Importance Sampling (Python Coding)

```
# Two-dimensional Monte Carlo quadrature with variance reduction
from math import *
from random import *
from random1 import *
from modfunc import *
from graphlib import *

pi4 = 4 * pi

def func(x,y):                                                  # integrand
   r2 = x*x + y*y
   return r2*exp(-0.5e0*r2)/pi4

# main

L = 8e0                                    # integration domain [-L,L] x [-L,L]
L2 = L * L                                              # area of sampling domain

nn  = [0]*4                                             # ending indexes of plots
col = [""]*4                                                    # colors of plots
sty = [0]*4                                                     # styles of plots

np = 400                                               # number of plotting points
x1 = [0]*(3*np+1); y1 = [0]*(3*np+1)                            # plotting points
x2 = [0]*(3*np+1); y2 = [0]*(3*np+1); sig = [0]*(3*np+1)

seed()

out = open("mcarlo.txt","w")                                  # open output file
out.write("     n       Int       sig      Int_w     sig_w\n")

for ip in range(1,np+1):
   n = 250 * ip                                         # number of sampling points

   f1 = f2 = 0.0          # quadrature with uniform sampling in [0,L] x [0,L]
   for i in range(1,n+1):
      x = L * random(); y = L * random()
      f = func(x,y)                                                 # integrand
```

```
      f1 += f; f2 += f * f                                          # sums

   f1 /= n; f2 /= n                                                 # averages
   s = 4e0 * L2 * f1                                                # integral
   sigma = 4e0 * L2 * sqrt(fabs(f2-f1*f1)/n)              # standard deviation
   out.write(("{0:8d}{1:10.5f}{2:10.5f}").format(n,s,sigma))
   x1[ip] =       n ; y1[ip] = s
   x2[ip] = log10(n); y2[ip] = log10(sigma)

   f1 = f2 = 0.0                           # quadrature with Gaussian sampling
   for i in range(1,n+1):
      (w, x, y) = randNrm2()        # random numbers with normal distribution
      f = func(x,y) / w                                             # integrand
      f1 += f; f2 += f * f                                          # sums

   f1 /= n; f2 /= n                                                 # averages
   s = f1                                                           # integral
   sigma = sqrt((f2-f1*f1)/n)                             # standard deviation
   out.write(("{0:10.5f}{1:10.5f}").format(s,sigma))
   x1[np+ip] =       n ; y1[np+ip] = s
   x2[np+ip] = log10(n); y2[np+ip] = log10(sigma)

   f1 = f2 = 0.0                        # quadrature with exponential sampling
   for i in range(1,n+1):
      x = randExp()          # random variables with exponential distribution
      y = randExp()
      w = exp(-(x+y))
      f = func(x,y) / w                                             # integrand
      f1 += f; f2 += f * f                                          # sums

   f1 /= n; f2 /= n                                                 # averages
   s = 4e0 * f1                                                     # integral
   sigma = 4e0 * sqrt((f2-f1*f1)/n)                       # standard deviation
   out.write(("{0:10.5f}{1:10.5f}\n").format(s,sigma))
   x1[2*np+ip] =       n ; y1[2*np+ip] = s
   x2[2*np+ip] = log10(n); y2[2*np+ip] = log10(sigma)

out.close()
                                                                    # linear regression
(a, b, sigma, sigmb, chi2) = LinFit(x2[0:],y2[0:],sig,np,0)
print("sigma = {0:6.3f} n**({1:6.3f})  uniform sampling". \
      format(pow(10e0,b),a))

(a, b, sigma, sigmb, chi2) = LinFit(x2[np:],y2[np:],sig,np,0)
print("sigma = {0:6.3f} n**({1:6.3f})  Gaussian sampling". \
      format(pow(10e0,b),a))

(a, b, sigma, sigmb, chi2) = LinFit(x2[2*np:],y2[2*np:],sig,np,0)
print("sigma = {0:6.3f} n**({1:6.3f})  exponential sampling". \
      format(pow(10e0,b),a))

GraphInit(1200,600)

nn[1] =   np; col[1] = "blue" ; sty[1] = 0                 # uniform sampling
nn[2] = 2*np; col[2] = "red"  ; sty[2] = 0                # Gaussian sampling
nn[3] = 3*np; col[3] = "green"; sty[3] = 0             # exponential sampling
MultiPlot(x1,y1,y1,nn,col,sty,3,10,0e0,0e0,0,0e0,0e0,0,
          0.10,0.45,0.15,0.85,"n","Int","Monte Carlo - integral")
```

```
MultiPlot(x2,y2,y2,nn,col,sty,3,10,0e0,0e0,0,0e0,0e0,0,
          0.60,0.95,0.15,0.85,"log(n)","log(sig)",
          "Monte Carlo - standard deviation")

MainLoop()
```

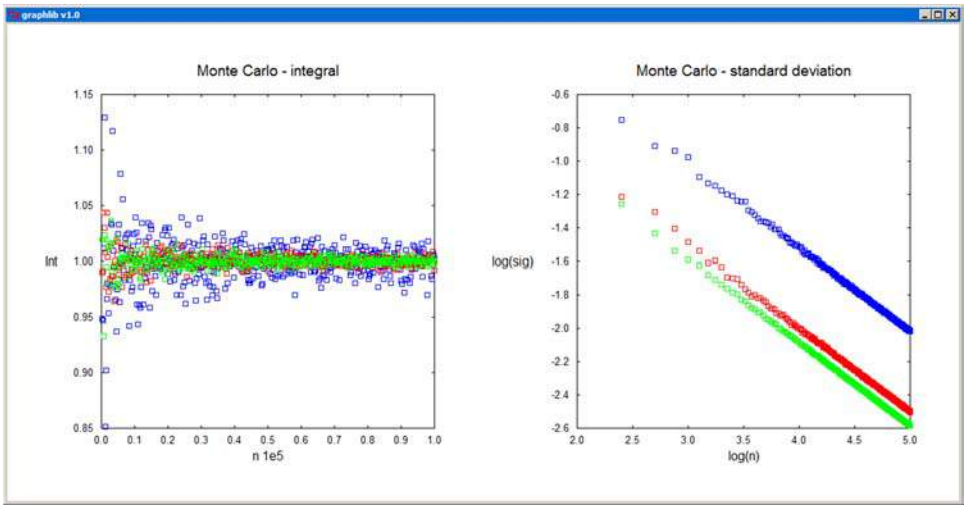

FIGURE 11.8 Monte Carlo estimates of integral (11.44) (left panel) and associated standard deviations (right panel) using uniform, Gaussian, and exponential sampling (lowest log–log dependence).

PROBLEM 11.3

Build the histograms for 10^6 random deviates, generated uniformly in the range $[0, 1)$ by the routines `random` (based on the built-in random number generator), `randLCG1` (linear congruential generator using specifications from Press et al. 2007), `randLCG2` (linear congruential generator based on specifications from Rapaport 2004), and, respectively, `randMCG` (George Marsaglia's multiply-with-carry generator), which are included in the libraries `random1.py` and, respectively, `random.h`.

For initializing, cumulating, and normalizing the data for the histograms, use the routine `HistoBin`, and for plotting the histograms, the function `Plot` with the style parameter `sty` set to 4. Both routines are part of the graphic modules `graphlib.py` and, respectively, `graphlib.h`.

Assess the quality of the random number generators by the uniformity of the corresponding histograms.

Solution

The Python implementation is given in Listing 11.13 and the C/C++ version is available as supplementary material (`P11-RandUni.cpp`). The histograms of the generated random distributions are shown in Figure 11.9.

PROBLEM 11.4

Build the histograms for 10^6 random deviates with normal (Gaussian) distribution, generated by the routines `randNrm` (based on the central limit theorem), `randNrm2` (based on polar change of variables), and `randMet` (based on the Metropolis algorithm). For `randMet` consider sampling hypercubes of sizes $\delta = 0.1$ and $\delta = 0.5$.