3 Monte Carlo

3.1 Pseudorandom Numbers

As most of the algorithms we introduce later include some randomization in their rules, we start with the question of how random events can be produced on a computer. After all, a computer is a machine working off a sequence of instructions in a deterministic way. A computer is therefore unable to produce any real random event. However, there has long been the need for simulating random events on a computer. (Think, e. g., of game simulations like roulette, blackjack, or poker.)

Therefore, many algorithms have been developed that produce a sequence of numbers that appears to be a sequence of random numbers. These so-called pseudorandom numbers are used instead of real random numbers in order to decide about the results of some sequence of events. For any unknowing spectator, the sequence of pseudorandom events should look like real random events.

As the city of Monte Carlo has been famous for decades for its casinos with roulette tables and many other games with random results, algorithms that make use of pseudorandom numbers are generally called Monte Carlo (MC) algorithms. The name inspired by the casino at Monte Carlo was first used by von Neumann and Ulam, when they used random numbers for simulating nuclear reactions in developing the atomic bomb. Usually, one speaks of random numbers instead of pseudorandom numbers.

It is worth mentioning here that the possible amount of generable pseudorandom numbers is always finite. After a certain number of calls the sequence of random numbers that has already been produced is repeated again. The larger this sequence length is, the better the random number generator should be. Random number generators always need at least one integer, an initial value x_0, called the seed, to get started. Different seeds do not usually lead to different sequences of random numbers, but the random number generator starts at different points in its finite sequence of random numbers.

The type of problem determines whether the knowledge of the seed, and therefore of the random numbers, is important or not. Sometimes one keeps a record of the seed in order to be able to reproduce the results of a MC program. The seed may be either encoded in the program or otherwise saved.

This is important, e. g., when trying to set new world records for some benchmark problems, because only if the seed is known can the result be reproduced, providing the proof that it was that program that generated the new record. However, in programming an application like a poker game, one would like to start every time with a new seed, one that cannot be controlled by the players. Here often the exact time of day, perhaps including even the milliseconds, is used as a seed.

Although "Monte Carlo" is commonly used to describe any randomized algorithm, some researchers distinguish between MC and Las Vegas (LV) algorithms. In contrast to MC algorithms, LV algorithms always lead to a correct result, whereas MC algorithms lead to the correct result only with a certain probability. A further difference is that MC algorithms have a deterministic running time whereas the running time of LV algorithms varies according to the random events. It it is also possible for a LV algorithm not to terminate. A subset of these LV algorithms are the Macao algorithms, which have a deterministic running time and which are guaranteed to terminate. In summary, different random seeds often lead to different results for a MC algorithm, such that its outcome is random, but a LV algorithm always produces the same result. The seed only influences the way to get there and therefore the running time [151].

3.2 Random Number Generation and Random Number Tests

Various algorithms leading to random numbers and tests checking the quality of such random numbers for their "randomness" have been developed. That is, the tests compare the results of using pseudorandom numbers instead of truly random numbers in the specific problems for which they are intended. The so-called random number generators are distinguished in different classes: "good" random numbers pass most of these elaborate tests. Some generators produce random numbers of a low quality but with low computational overhead. The resulting "quick and dirty" random numbers do not pass all tests checking their randomness. However, they should pass at least some basic tests to be useful for some MC techniques.

There are several ways to produce "quick and dirty" random numbers. A common way is to use the linear congruential method [128]: starting from the seed value x_0 the following iterative rule is applied to produce a sequence of random numbers:

$$x_{i+1} = (a \times x_i + b) \bmod c\,. \tag{3.1}$$

The parameters a, b, c, which are integers, define the random number generator. Each generated random number only depends on its predecessor. As the random numbers are calculated modulo the integer number c, they can only take the values $0, 1, \ldots, c-1$. The basic sequence of the random number

generator can therefore only have a length of up to c; after that the sequence repeats. Therefore, the chosen parameter c must be rather large. (You can also get longer sequences if you keep more previous states, see, e. g., the R250 random number generator [115].) One can also accelerate the speed of the random number generator: as the modulo operation takes more time than addition or multiplication, one wants to avoid performing this modulo operation. This can be done by making use of the fact that integers are stored in a finite amount of bits on a computer: if one is added to the largest positive integer that can be represented on a computer, then the smallest negative number that can be represented is returned as the result. This overflow is used for a natural type of the modulo operation. Working with signed integers of 32 bits, the largest positive number is $2^{31} - 1$ and the smallest negative number is -2^{31} such that c is effectively set to 2^{32} and only the rule $x_{i+1} = a \times x_i + b$ is applied. Then the following values for a and b provide some good quick and dirty random numbers:

- $a = 1{,}664{,}525$, $b = 1{,}013{,}904{,}223$
- $a = 16{,}807$, $b = 0$
- $a = 65{,}539$, $b = 0$
- $a = 65{,}549$, $b = 0$

In the multiplicative congruential cases where $b = 0$, only odd seed values shall be used; otherwise correlations are apparent even in the screen pixel test. Some such random number generators are famous; for example, $7^5 = 16{,}807$ is often called magic due to some of its properties. Throughout this book, we use only random numbers generated with $a = 1{,}664{,}525$ and $b = 1{,}013{,}904{,}223$.

From integer random numbers x_i, uniformly distributed in the interval $[-2^{31}; 2^{31} - 1]$, random numbers r_i are derived by dividing by the largest possible absolute random number (in this case 2^{31}) and taking the absolute value:

$$r_i = \mathrm{abs}\left(x_i \times 4.656612 \cdot 10^{-10}\right) . \tag{3.2}$$

These r_i are uniformly distributed in the interval $[0; 1]$.

Other methods also exist for producing random numbers very quickly, some of them leading to fairly good random numbers, for example the Kirkpatrick–Stoll random number generator [115].

More elaborate random number generators, which produce random numbers of a high quality, often use the basic generation methods, introduced above, not just once but several times and in a combined way. However, it takes more time to produce a sequence of random numbers using one of these generators [166]. One must be careful when trying this because there are ways of combining random number generators that lead to lower quality in the resulting random numbers. See [166] on this.

Therefore, one has to weigh the necessity of such good random numbers. For most purposes in optimization, the quick and dirty random numbers

are quite sufficient. Furthermore, one must even be careful when using such a mathematically proved good random number generator: it has been shown that sometimes (e. g., if they are correlated with the proposed problem) these lead to worse results than the quick and dirty generators [70].

To investigate the test routines of random numbers, let us consider here only random numbers that are uniformly distributed in some interval. The simplest test is the histogram test. In this test, the interval is divided into a certain number of subintervals. A counter is assigned to each subinterval and initialized with zero. Then a long sequence of random numbers is calculated. For each random number the counter for the subinterval in which it lies is incremented. For a large amount of random numbers, these counters should exhibit roughly the same value, i. e., they should be roughly equal to the overall number of random numbers divided by the number of subintervals, as seen in Fig. 3.1.

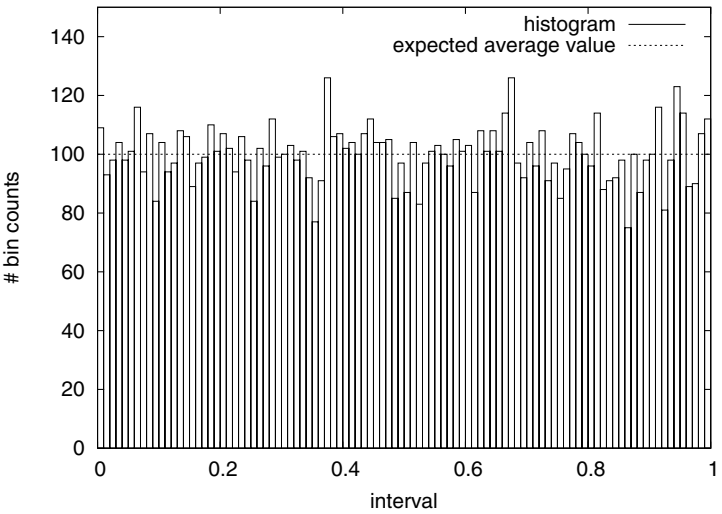

Fig. 3.1. Example of a histogram generated from 10,000 random numbers uniformly distributed in the unit interval. The unit interval is divided into 100 subintervals, also called bins, with a width of 0.01 each. As the random numbers are chosen uniformly, we expect to get on average 100 random numbers in each bin

However, even for very many random numbers, the heights of the histogram bars differ significantly. Actually they must differ with a certain variation if the random numbers are independent as well as random. Let us denote the overall number of random numbers as N, the number of bins as N_b, the probability that a random number gets sorted in bin No. i as p_i, and the expected number of random numbers in bin i as $N_\mathrm{exp}(i) = N \times p_i$. As our

random numbers are uniformly chosen, p_i is given as $p_i = 1/N_\mathrm{b}$ such that $N_\mathrm{exp} = N_\mathrm{exp}(i) \equiv N/N_\mathrm{b}$. The variations should be of order $\mathcal{O}(\sqrt{N})$.

The χ^2-test is a good means for determining whether the random numbers are truly random: the normalized χ^2-measure is given as

$$\chi^2 = \frac{1}{N_\mathrm{b} - 1} \sum_{i=1}^{N_\mathrm{b}} \frac{(N_i - N \times p_i)^2}{N \times p_i}\,, \tag{3.3}$$

with N_i being the actual number of random numbers falling in bin i in the test. If N_i are identical to N_exp, the normalized χ^2-measure would give a value of 0. But with the usual fluctuations, the normalized χ^2-measure should have a value of roughly 1. For our example in Fig. 3.1, we get $\chi^2 = 0.909$.

We performed many more tests for various values of N_b and N_exp. The results are shown in Fig. 3.2. For example, we get $\chi^2 = 1.0019697 \pm 0.0045$ for $N_\mathrm{exp} = 100$ and $N_\mathrm{b} = 100$, averaged over 1000 sequences. The minimum value we find here is 0.687676768, and the maximum value is 1.7. Generally, we get that χ^2 is roughly 1, independent of the number of bins and the expected number of counts per bin. Moreover, the variation of our normalized χ^2-measure around 1 decreases with an increasing number of bins. For the precise values of the probability distribution of χ^2 without our normalization by $1/(N_\mathrm{b} - 1)$, see [1]. It can also be evaluated in popular math packages, such as Mathematica.

Now let us consider normalized χ^2-values of distributions tampered with by people thinking, e. g., that each bin should have the same number of

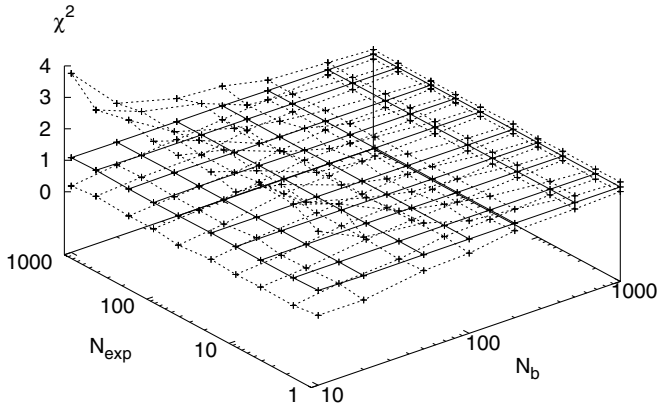

Fig. 3.2. χ^2-values for random numbers generated with a quick and dirty random number generator: for various numbers N_b of bins and for various expected values N_exp of counts per bin, we generated 1000 sequences containing $N_\mathrm{b} \times N_\mathrm{exp}$ random numbers each. The graphic shows the minimum, average, and maximum χ^2-values taken from the 1000 sequences each. The deviation of the minimum χ^2-value and the maximum χ^2-value from the average χ^2-value decreases with increasing N_b

counts. As already mentioned, the ideal normalized χ^2-value for large N_b is 1. If each bin had exactly the same number of counts, such that it was equal to N_exp, then the nominator in Eq. (3.3) would vanish, such that $\chi^2 = 0$. Thus, people showing you a "perfect" looking histogram with bins of the same size created by an "excellent" newly invented random number generator are simply trying to fool you. Then there are those who mess around with histograms trying to reduce the visible amount of randomness in order to impress their bosses, who do not have a clue as to what randomness is about.

There are approaches in which some people simply change N_i to $N_i + (N_\mathrm{exp}(i) - N_i)/2$, thus reducing the "distance" between the actual and the desired value by a factor of 2. Applying this approach to the instance in Fig. 3.1, χ^2 decreases to 0.245. If the distance were further reduced, e. g., even by a factor of 4, then χ^2 would be only 0.06 for this instance. But using the numbers from our experiment of Fig. 3.2, the values of χ^2 lie well outside the range we encountered in 1000 trials. Thus, their likelihood of coming from real independent random variables is less than one in 1000. In fact, much, much less.

Another widely used way of tampering with histograms is setting some of the histogram counts to their expected values. But if we, e. g., select randomly half of the bins and set their counters N_i to the expected number $N_\mathrm{exp} = 100$ for our example instance above, we get $\chi^2 = 0.531$; if we do this for even three quarters of the bins, then we get $\chi^2 = 0.276$.

One could also think of other strategies to reduce the randomness in a histogram, but most of this removing of true randomness can be detected with the χ^2-test, as shown above, as the value of χ^2 is not roughly 1 then, as it should be, as our results for thousands of runs in Fig. 3.2 show.

One might also wonder whether to add additional randomness to such a histogram. This can be done, e. g., by changing N_i into $N_i + 2(N_\mathrm{exp} - N_i)$. Applying this change to our histogram instance, we get $\chi^2 = 2.026$. Thus, values much larger than 1 are achieved when adding randomness. It is basically impossible to get such a large value of χ^2 if the number of bins is sufficiently large.

A more elaborate test, one that checks for correlations between the random numbers, is the screen pixel test: the successive random numbers x_i are considered to be the coordinates of the points $p_j = (x_{2j-1}, x_{2j})$. These points are printed as pixels on the screen. If the pixels fill the screen completely and smoothly, then also the requirement of the uniform distribution in the whole interval is fulfilled. Furthermore, the pixels should not form any patterns when printed on the screen, such that the correlation between successive random numbers is not too large. One can proceed further and look in three and even higher dimensions at pixels with the coordinates $(x_{dj-d+1}, \ldots, x_{dj})$ [133], with d denoting the number of dimensions. Looking from the right angle, one finds that the random points lie in only a small amount of (hyper)planes instead of being distributed randomly over the whole space. This Marsaglia effect reveals bad instances of the multiplicative random number genera-

tor. These three eye checks only show whether these basic requirements are fulfilled, but such checks are, of course, no replacement for a real test of "randomness".

3.3 Transformation of Random Numbers

Nonuniform distributions of random numbers may be necessary. The simplest case is a uniform distribution over a different random interval, say, $[a;b]$. In this case, the following linear transformation is applied to the random numbers:

$$\tilde{r}_i = a + (b-a) \times r_i \,. \tag{3.4}$$

The new random numbers $\tilde{r}_i$ are uniformly distributed in the interval $[a;b]$.

Sometimes one wants to work with uniformly distributed integer random numbers $1,\ldots,N$. For this purpose, the following rule is used: let r_i again be random numbers that are uniformly distributed in the interval $[0;1]$; then the integer random numbers n_i are created according to

$$n_i = [1 + N \times r_i] \,, \tag{3.5}$$

with $[x]$ denoting the Gaussian brackets, which simply take the integer part of x. One must be careful when applying this rule, as r_i could be exactly 1, so that $n_i = N + 1$. In this marginal case, either a new random number is calculated or n_i is simply set to one of the integer numbers in its range. Of course, if the real-valued random numbers r_i are derived from integer-value random numbers x_i, as in Sect. 3.2, it is faster to turn the x_i directly into the n_i by

$$n_i = |x_i \bmod N| + 1 \,. \tag{3.6}$$

However, if N is rather large, then an error occurs: simply assume that N is $\frac{2}{3}$ of the maximum possible random number. Then the first half of the numbers are generated twice as often as the second half of the numbers.

So far, we have only generated uniformly distributed random numbers. But often also other distributions are needed. For generating random numbers that are distributed in a different way, one usually starts out with a random number generator as discussed above that creates random numbers r_i that are uniformly distributed in the unit interval $[0;1]$. Now if one needs a distribution of the random numbers according to some distribution function P, which for simplicity we assume to lie in the range between 0 and 1, there is always the possibility of using the von Neumann rejection principle: first, one determines the interval in which the desired random numbers will be found; let this interval be $[a;b]$. Then the random numbers r_i are transformed in a linear way, as discussed above, such that they are uniformly distributed in the interval $[a;b]$; one sets $\tilde{r}_i = a + (b-a) \times r_i$. Then one chooses a second random number s_i for each $\tilde{r}_i$; these s_i are also uniformly distributed in the interval $[0;1]$. Now there are two cases, either $s_i \leq P(\tilde{r}_i)$ or $s_i > P(\tilde{r}_i)$. As

the s_i are uniformly distributed in the range $[0;1]$, the case $s_i \leq P(\tilde{r}_i)$ occurs with the exact probability $P(\tilde{r}_i)$. Thus, the von Neumann rejection principle works as follows. Create pairs (r_i, s_i) of random numbers. Then, if required, transform the r_i into $\tilde{r}_i$. For each i, determine whether $s_i \leq P(\tilde{r}_i)$. If this is the case, then keep $\tilde{r}_i$; otherwise delete it from the sequence of random numbers. The remaining $\tilde{r}_i$ are then distributed according to the distribution function P.

The von Neumann method can be used for creating any desired probability distribution. However, it often consumes a lot of computing time for generating random numbers that are thrown away later on, especially if the desired distribution peaks in some areas of the interval and nearly vanishes in other areas. Furthermore, it converges rather slowly toward the desired distribution function. Therefore, other techniques for creating a certain distribution must be given preference.

In another widely used approach, uniformly distributed random numbers are transformed directly into random numbers that are distributed according to the desired distribution function P. Let (r_i) be a set of uniformly distributed random numbers. Then the task is to transform this set into a set (s_i) of random numbers distributed according to the probability distribution P. Here one makes use of the fundamental transformation law of probabilities [166], according to which one gets

$$|\,\mathrm{d}r| = |P(s)\,\mathrm{d}s|\,. \tag{3.7}$$

This equation can be transformed into

$$P(s) = \left|\frac{\mathrm{d}r}{\mathrm{d}s}\right|\,. \tag{3.8}$$

Now we want to derive the transformation for the widely used exponential distribution

$$P(s) = \exp(-s)\,. \tag{3.9}$$

Thus, one gets $r(s) = \exp(-s)$, i. e., $s(r) = -\ln(r)$. Therefore, one can transform the random numbers r_i, which are uniformly distributed in the unit interval, into the random numbers $s_i = -\ln(r_i)$, which are exponentially distributed over the interval $[0;+\infty[$. One has only to address the fact that the random number generator for the r_i might create $r_i = 0$. This random number is dropped, and another random number r_i is then generated, from which the s_i can be derived.

Another widely used distribution of random numbers is the Gaussian or normal distribution. The Gaussian probability distribution P is given by the formula

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)\,. \tag{3.10}$$

A simple way of generating such a distribution would be to generate first a set of random numbers $(r_i)_{i=1,\ldots,N}$ that are uniformly distributed in the

interval $[\mu - \alpha\sigma; \mu + \alpha\sigma]$. α has to be chosen large enough such that $P(\mu + \alpha\sigma)$ is sufficiently small. Usually, α is chosen to be ≥ 3. Then a second set of random numbers $(s_i)_{i=1,\ldots,N}$ is generated; these random numbers are uniformly distributed in the interval $[0; 1/\sqrt{2\pi}\sigma]$. Now the following rule is applied: if $s_i \leq P(r_i)$, then the random number r_i is accepted, otherwise it is rejected. The remaining accepted random numbers are roughly Gaussian distributed. However, the tails of a real Gaussian distribution are missing in this distribution. Therefore, one has to find some compromise: if α is chosen very large, then most of the random numbers are rejected. However, the tails are at least partially represented. If α is small, then the tails are completely missing but little calculation time is wasted for calculating random numbers that are rejected afterwards. As was already mentioned, other techniques for creating a certain distribution must be given preference.

For the Gaussian distribution, a very simple approach based on the central limit theorem can be used for getting quick and dirty Gaussian random numbers: according to the central limit theorem, the sum over several probability distributions (which are either bounded or have a finite variance) always tends toward the Gaussian distribution. Therefore, summing up several uniformly distributed random numbers leads to Gaussian random numbers:

$$g_i = -\frac{N}{2} + \sum_{j=1}^{N} r_{i,j} \,. \tag{3.11}$$

$N/2$ has to be subtracted as $\frac{1}{2}$ is the expectation value of a $[0; 1]$ uniformly distributed random number, such that the resulting g_is are centered at 0. The larger N is, the better is the approximation to the Gaussian distribution. A natural choice is $N = 12$, as then the standard deviation of the resulting Gaussian distribution is 1: the expectation value of a uniformly distributed random number r_i in the interval $[0; 1]$ is $\langle r_i \rangle = \frac{1}{2}$, $\langle r_i^2 \rangle = \frac{1}{3}$, $\langle r_i r_j \rangle = \frac{1}{4}$ for $i \neq j$. Therefore,

$$\begin{aligned}
\sigma^2(g_i) &= \left\langle \left(-6 + \textstyle\sum_{j=1}^{12} r_{i,j}\right)^2 \right\rangle - \left\langle -6 + \textstyle\sum_{j=1}^{12} r_{i,j} \right\rangle^2 \\
&= \left\langle \left(\textstyle\sum_{j=1}^{12} r_{i,j}\right)^2 \right\rangle - 12 \left\langle \textstyle\sum_{j=1}^{12} r_{i,j} \right\rangle + 36 - 0 \\
&= \left\langle \left(r_{i,1} + r_{i,2} + \ldots + r_{i,12}\right)^2 \right\rangle - 12 \times 6 + 36 \\
&= 12 \left\langle {r_{i,j}}^2 \right\rangle + 2 \times (11 + 10 + \ldots + 1) \times \langle r_{i,j} \times r_{i,k\neq j} \rangle - 36 \\
&= 12 \times \tfrac{1}{3} + 2 \times \tfrac{11 \times 12}{2} \times \tfrac{1}{4} - 36 \\
&= 4 + 33 - 36 = 1 \,.
\end{aligned} \tag{3.12}$$

Note that the disadvantage of summing up only 12 uniformly distributed random numbers is that only Gaussian random numbers in the interval $[-6;6]$ can be obtained. Beyond that, the tails of the Gaussian distribution are completely missing.

There is another method leading to very good Gaussian random numbers with zero mean and unit variance, the Box–Muller method: the trick of this method consists of creating two Gaussian random numbers a and b from two uniformly distributed random numbers u and v in the interval $[0;1]$ at the same time. Let

$$f(x,y) = \frac{1}{2\pi} \exp\left(-\frac{x^2+y^2}{2}\right) \tag{3.13}$$

be the density distribution and

$$\varrho^2 = x^2 + y^2\,, \quad \tan(\varphi) = \frac{y}{x} \tag{3.14}$$

be the transformation to polar coordinates. Then the relation between the area elements is given by

$$\mathrm{d}x\,\mathrm{d}y = \varrho\,\mathrm{d}\varrho\,\mathrm{d}\varphi\,. \tag{3.15}$$

The local distribution has to be equal for both coordinate systems:

$$f(x,y)\,\mathrm{d}x\,\mathrm{d}y = f(\varrho,\varphi)\,\mathrm{d}\varrho\,\mathrm{d}\varphi \tag{3.16}$$

such that

$$f(\varrho,\varphi) = \frac{1}{2\pi}\varrho \exp\left(-\frac{\varrho^2}{2}\right)\,. \tag{3.17}$$

The goal is to generate the radius ϱ according to the distribution $\varrho\exp(-\varrho^2/2)$ by using the first random number u and φ being homogenously distributed in $[0;2\pi]$ by using the second random number v. Therefore,

$$F(\varrho) = \frac{1}{2\pi}\int_0^{2\pi} \mathrm{d}\varphi \int_0^{\varrho} \varrho' \exp\left(-\frac{\varrho'^2}{2}\right) \mathrm{d}\varrho' = 1 - \exp\left(-\frac{\varrho^2}{2}\right) \tag{3.18}$$

can be equalized with some uniformly distributed random number z in the interval $[0;1]$. The radius ϱ is therefore given by

$$\varrho = \sqrt{-2\ln(1-z)}\,. \tag{3.19}$$

$u = 1 - z$ is also a uniformly distributed random number from the unit interval. Let v be the second uniformly distributed random number and set $\varphi = 2\pi v$. With the relations $x = \varrho\cos(\varphi)$ and $y = \varrho\sin(\varphi)$, one obtains the two Gaussian distributed random numbers

$$a = \cos(2\pi v)\sqrt{-2\ln(u)} \tag{3.20}$$

and

$$b = \sin(2\pi v)\sqrt{-2\ln(u)}\,. \tag{3.21}$$

However, this method of calculating Gaussian distributed random numbers is very time consuming, as the functions natural logarithm, sine, cosine, and the square root must be calculated. Furthermore, it is recommended that v should not depend strongly on u, as then a and b are not truly independently normally distributed.

Of course, this distribution can simply be transferred to a Gaussian distribution with another mean value μ or another standard deviation σ.

The Box–Muller method was improved by Marsaglia and Bray in 1964 [134]. Their improved method, which eliminates the calculation of the sine and the cosine, has become known as the polar method. As in the Box–Muller method, first two uniformly distributed random numbers u and v are chosen. Then they are linearly transformed to the interval $[-1; 1]$:

$$\tilde{u} = 2u - 1 \text{ and } \tilde{v} = 2v - 1\,. \tag{3.22}$$

Then

$$w = \tilde{u}^2 + \tilde{v}^2 \tag{3.23}$$

is calculated. If $w > 1$, then the algorithm jumps back to its start. Otherwise, let

$$t = \sqrt{\frac{-2\ln w}{w}}\,. \tag{3.24}$$

Then the Gaussian random numbers a and b are given by

$$a = t \times \tilde{u} \quad \text{and} \quad b = t \times \tilde{v}\,. \tag{3.25}$$

The savings from eliminating the calculation of one sine and one cosine is balanced by having to apply a rejection rule such that two new random numbers might have to be chosen. Note that in contrast to the Box–Muller method, in which u was used to calculate the radius and v was used to determine the phase, here u and v are used together. The rejection principle is applied in order to accept only those pairs of (u, v) for which this trick can be used, namely, those inside the unit circle.

Finally, we compare the four methods for creating Gaussian distributed random numbers. Table 3.1 shows the calculation times of the various methods. It turns out that the rejection method is the slowest method, although only $\alpha = 3$ was used for creating the random numbers, whereas the polar method is the fastest method.

Table 3.1. Comparison of calculation times of four methods creating Gaussian random numbers: in each case, an array containing 1 million Gaussian distributed random numbers was created 100 times. The times are given for a 400-MHz Pentium II

Method	Calculation time
Rejection method	200 s
Summing up 12	81 s
Box–Muller method	74 s
Polar method	55 s

3.4 Example: Calculation of π with MC

A famous example for the usage of the MC technique is the calculation of the number π by means of MC. Figure 3.3 shows the geometry used for this calculation: one shoots randomly into a square of edge length r and counts the hits in the quarter circle of a radius r. The position of the random point, i. e., both its x- and y-coordinates, is determined by two successive calls of a random number generator. It is simply checked with $x^2 + y^2 \leq r^2$ to see whether the point is inside the quarter circle or not. The area of the square is given by r^2, and the area of the quarter circle is given by $\frac{1}{4}\pi r^2$. Therefore, the ratio of the number of hits divided by the number of shots approximates $\frac{1}{4}\pi$, if the number of shots is chosen large enough.

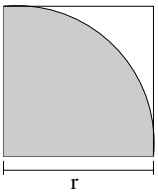

Fig. 3.3. A quarter circle in a square

Table 3.2. For each number of shots, 100 runs were performed. The minimum and maximum approximations for π for the corresponding number of shots are given, as are the mean value and the error bar of these approximations

Shots	$\pi_{\min}$	$\pi_{\max}$	$\pi_{\text{mean}} \pm \Delta\pi$
10	1.6	4.0	3.148 ± 0.06
100	2.64	3.40	3.1408 ± 0.015
1000	3.032	3.284	3.14484 ± 0.0052
10,000	3.1044	3.1852	3.143668 ± 0.0016
100,000	3.12500	3.15344	3.1415816 ± 0.00057
1,000,000	3.138464	3.14534	3.1415648 ± 0.00015
10,000,000	3.1408864	3.142138	3.14159355 ± 0.000023

Table 3.2 shows results for various numbers of shots. As can be easily seen, it is much better to calculate π with the formula

$$\pi = 4 \times \arctan(1) \tag{3.26}$$

or to memorize the first 15 digits of π. These can be remembered as the lengths of the words in the sentence "Now I want a drink: alcoholic, of course, after the heavy lectures involving quantum mechanics!"