42. Monte Carlo Techniques

Revised September 2025 by G. Cowan (RHUL).

Monte Carlo techniques are often the only practical way to evaluate difficult integrals or to sample random variables governed by complicated probability density functions. Here we describe an assortment of methods for sampling some commonly occurring probability density functions.

42.1 Sampling the uniform distribution

Most Monte Carlo sampling or integration techniques assume a "random number generator," which generates uniform statistically independent values on the half open interval $[0, 1)$; for reviews see, *e.g.*, Refs. [1, 2].

Uniform random number generators are available in software libraries such as CLHEP [3], and ROOT [4], as well as from the standard C++ library [5]. The ROOT generators are available in Python using the PyROOT interface [6]. All of the algorithms produce a periodic sequence of numbers, and to obtain effectively random values, one must not use more than a small subset of a single period. The performance of the generators can be investigated with tests such as DIEHARD [7] or TestU01 [8]. Many commonly available congruential generators fail these tests and often have sequences (typically with periods less than 2^{32}), which can be easily exhausted on modern computers.

The generators in ROOT can be accessed through the interface class `TRandom`. These include the widely used `TRandom3` generator, which implements the Mersenne twister algorithm of Matsumoto and Nishimura [9]. This is very fast and has an extremely long period of $2^{19937} - 1$, although it fails some of the more stringent TestU01 tests. Two somewhat slower generators available in ROOT that do not fail any TestU01 tests are the `Ranlux++` generator [10] based on the method of Lüscher [11], and the `MIXMAX` generator of Savvidy [12]. A recent review of high-quality random number generators can be found in Ref. [13].

42.2 Inverse transform method

If the desired probability density function is $f(x)$ on the range $-\infty < x < \infty$, its cumulative distribution function (expressing the probability that $x \leq a$) is given by Eq. (39.6). If a is chosen with probability density $f(a)$, then the integrated probability up to point a, $F(a)$, is itself a random variable which will occur with uniform probability density on $[0, 1]$. Suppose u is generated according to a uniformly distributed in $(0, 1)$. If x can take on any value, and ignoring the endpoints, we can then find a unique x chosen from the p.d.f. $f(x)$ for a given u if we set

$$u = F(x) \ , \tag{42.1}$$

provided we can find an inverse of F, defined by

$$x = F^{-1}(u) \ . \tag{42.2}$$

This method is shown in Fig. 42.1a. It is most convenient when one can calculate by hand the inverse function of the indefinite integral of f. This is the case for some common functions $f(x)$ such as $\exp(x)$, $(1 - x)^n$, and $1/(1 + x^2)$ (Cauchy or Breit-Wigner), although it does not necessarily produce the fastest generator. Standard libraries contain software to implement this method numerically, working from functions or histograms in one or more dimensions, *e.g.*, the UNU.RAN package [14], available in ROOT. For a discrete distribution, $F(x)$ will have a discontinuous jump

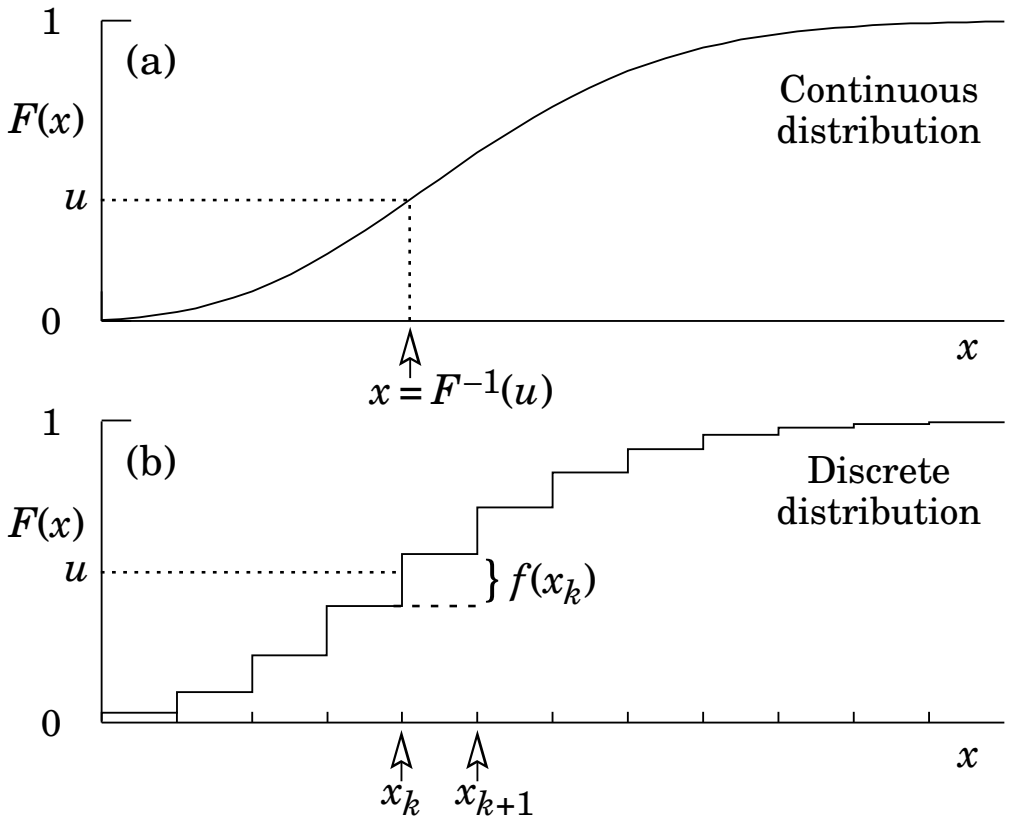

Figure 42.1: Use of a random number u chosen from a uniform distribution (0,1) to find a random number x from a distribution with cumulative distribution function $F(x)$.

of size $f(x_k)$ at each allowed $x_k, k = 1, 2, \cdots$. Choose u from a uniform distribution on (0,1) as before. Find x_k such that

$$F(x_{k-1}) < u \leq F(x_k) \equiv \text{Prob}\ (x \leq x_k) = \sum_{i=1}^{k} f(x_i)\ ; \tag{42.3}$$

then x_k is the value we seek (note: $F(x_0) \equiv 0$). This algorithm is illustrated in Fig. 42.1b.

42.3 Acceptance-rejection method (Von Neumann)

Very commonly an analytic form for $F(x)$ is unknown or too complex to work with, so that obtaining an inverse as in Eq. (42.2) is impractical. We suppose that for any given value of x, the probability density function $f(x)$ can be computed, and further that enough is known about $f(x)$ that we can enclose it entirely inside a shape which is C times an easily generated distribution $h(x)$, as illustrated in Fig. 42.2. That is, $Ch(x) \geq f(x)$ must hold for all x. Frequently $h(x)$ is uniform or is a normalized sum of uniform distributions. Note that both $f(x)$ and $h(x)$ must be normalized to unit area, and therefore, the proportionality constant $C > 1$. To generate $f(x)$, first generate a candidate x according to $h(x)$. Calculate $f(x)$ and the height of the envelope $C\,h(x)$; generate u and test if $uC\,h(x) \leq f(x)$. If so, accept x; if not reject x and try again. If we regard x and $uC\,h(x)$ as the abscissa and ordinate of a point in a two-dimensional plot, these points will populate the entire area $C\,h(x)$ in a smooth manner; then we accept those which fall under $f(x)$. The efficiency is the ratio of areas, which must equal $1/C$; therefore we must keep C as close as

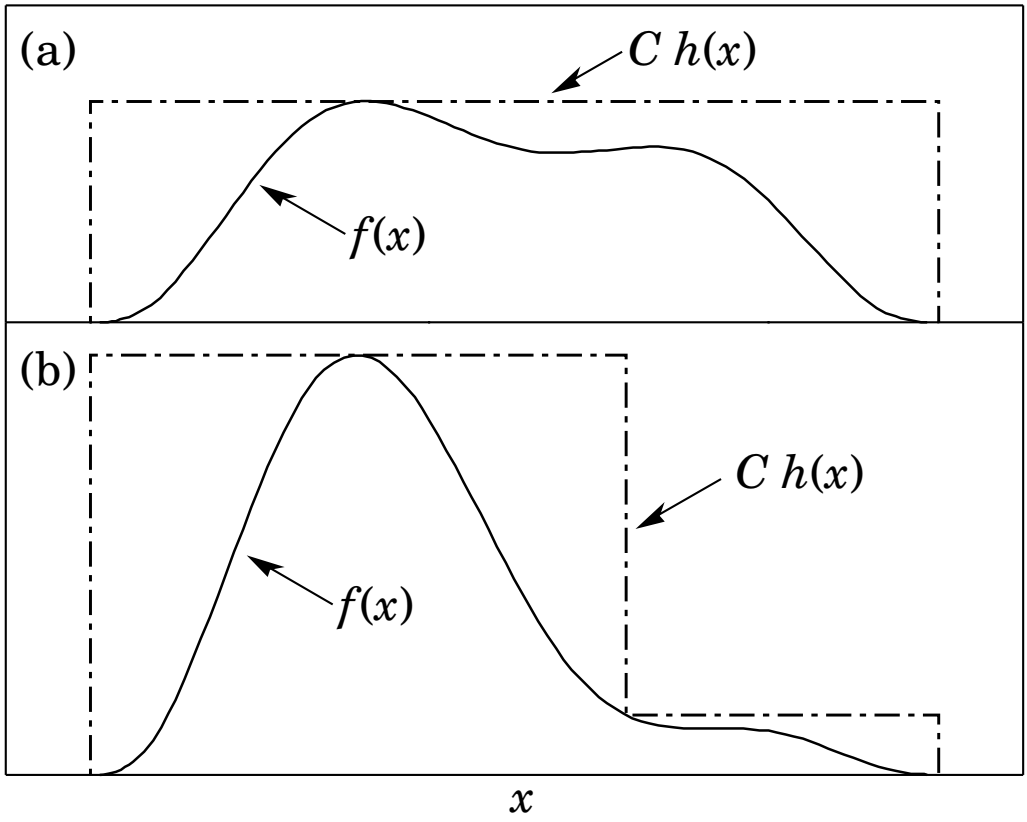

Figure 42.2: Illustration of the acceptance-rejection method. Random points are chosen inside the upper bounding figure, and rejected if the ordinate exceeds $f(x)$. The lower figure illustrates a method to increase the efficiency (see text).

possible to 1.0. Therefore, we try to choose $C\,h(x)$ to be as close to $f(x)$ as convenience dictates, as in the lower part of Fig. 42.2.

42.4 Algorithms

Algorithms for generating random numbers belonging to many different distributions are given for example by Press [15], Ahrens and Dieter [16], Rubinstein [17], Devroye [18], Walck [19] and Gentle [20]. For many distributions, alternative algorithms exist, varying in complexity, speed, and accuracy. For time-critical applications, these algorithms may be coded in-line to remove the significant overhead often encountered in making function calls.

In the examples given below, we use the notation for the variables and parameters given in Table 39.1. Variables named "u" are assumed to be independent and uniform on [0,1). Denominators must be verified to be non-zero where relevant.

42.4.1 *Exponential decay*

This is a common application of the inverse transform method, and uses the fact that if u is uniformly distributed in $[0, 1]$, then $(1 - u)$ is as well. Consider an exponential p.d.f. $f(t) = (1/\tau)\exp(-t/\tau)$ that is truncated so as to lie between two values, a and b, and renormalized to unit area. To generate decay times t according to this p.d.f., first let $\alpha = \exp(-a/\tau)$ and $\beta = \exp(-b/\tau)$; then generate u and let

$$t = -\tau \ln(\beta + u(\alpha - \beta)). \tag{42.4}$$

For $(a, b) = (0, \infty)$, we have simply $t = -\tau \ln u$. (See also Sec. 42.4.6.)

42.4.2 *Isotropic direction in 3D*

Isotropy means the density is proportional to solid angle, the differential element of which is $d\Omega = d(\cos\theta)d\phi$. Hence $\cos\theta$ is uniform $(2u_1 - 1)$ and ϕ is uniform $(2\pi u_2)$. For alternative generation of $\sin\phi$ and $\cos\phi$, see the next subsection.

42.4.3 *Sine and cosine of random angle in 2D*

Generate u_1 and u_2. Then $v_1 = 2u_1 - 1$ is uniform on (−1,1), and $v_2 = u_2$ is uniform on (0,1). Calculate $r^2 = v_1^2 + v_2^2$. If $r^2 > 1$, start over. Otherwise, the sine (S) and cosine (C) of a random angle (*i.e.*, uniformly distributed between zero and 2π) are given by

$$S = 2v_1v_2/r^2 \quad \text{and} \quad C = (v_1^2 - v_2^2)/r^2 \ . \tag{42.5}$$

42.4.4 *Gaussian distribution*

If u_1 and u_2 are uniform on (0,1), then

$$z_1 = \sin(2\pi u_1)\sqrt{-2\ln u_2} \quad \text{and} \quad z_2 = \cos(2\pi u_1)\sqrt{-2\ln u_2} \tag{42.6}$$

are independent and Gaussian distributed with mean 0 and $\sigma = 1$.

There are many variants of this basic algorithm, which may be faster. For example, construct $v_1 = 2u_1 - 1$ and $v_2 = 2u_2 - 1$, which are uniform on (−1,1). Calculate $r^2 = v_1^2 + v_2^2$, and if $r^2 > 1$ start over. If $r^2 < 1$, it is uniform on (0,1). Then

$$z_1 = v_1\sqrt{\frac{-2\ln r^2}{r^2}} \quad \text{and} \quad z_2 = v_2\sqrt{\frac{-2\ln r^2}{r^2}} \tag{42.7}$$

are independent numbers chosen from a normal distribution with mean 0 and variance 1. $z_i' = \mu + \sigma z_i$ distributes with mean μ and variance σ^2.

For a multivariate Gaussian with an $n \times n$ covariance matrix V, one can start by generating n independent Gaussian variables, $\{\eta_j\}$, with mean 0 and variance 1 as above. Then the new set $\{x_i\}$ is obtained as $x_i = \mu_i + \sum_j L_{ij}\eta_j$, where μ_i is the mean of x_i, and L_{ij} are the components of L, the unique lower triangular matrix that fulfils $V = LL^T$. The matrix L can be easily computed by the following recursive relation (Cholesky's method):

$$L_{jj} = \left(V_{jj} - \sum_{k=1}^{j-1} L_{jk}^2 \right)^{1/2} \ , \tag{42.8a}$$

$$L_{ij} = \frac{V_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}}{L_{jj}} \ , \ j = 1, ..., n \, ; \ i = j+1, ..., n, \tag{42.8b}$$

where $V_{ij} = \rho_{ij}\sigma_i\sigma_j$ are the components of V. For $n = 2$ one has

$$L = \begin{pmatrix} \sigma_1 & 0 \\ \rho\sigma_2 & \sqrt{1-\rho^2}\,\sigma_2 \end{pmatrix} \ , \tag{42.9}$$

and therefore the correlated Gaussian variables are generated as $x_1 = \mu_1 + \sigma_1\eta_1$, $x_2 = \mu_2 + \rho\sigma_2\eta_1 + \sqrt{1-\rho^2}\,\sigma_2\eta_2$.

42.4.5 $\chi^2(n)$ distribution

To generate a variable following the χ^2 distribution for n degrees of freedom, use the Gamma distribution with $k = n/2$ and $\lambda = 1/2$ using the method of Sec. 42.4.6.

42.4.6 *Gamma distribution*

All of the following algorithms are given for $\lambda = 1$. For $\lambda \neq 1$, divide the resulting random number x by λ.

- If $k = 1$ (the *exponential* distribution), accept $x = -\ln u$. (See also Sec. 42.4.1.)

- If $0 < k < 1$, initialize with $v_1 = (e + k)/e$ (with $e = 2.71828...$ being the natural log base). Generate u_1, u_2. Define $v_2 = v_1 u_1$.

  Case 1: $v_2 \leq 1$. Define $x = v_2^{1/k}$. If $u_2 \leq e^{-x}$, accept x and stop, else restart by generating new u_1, u_2.

  Case 2: $v_2 > 1$. Define $x = -\ln([v_1 - v_2]/k)$. If $u_2 \leq x^{k-1}$, accept x and stop, else restart by generating new u_1, u_2. Note that, for $k < 1$, the probability density has a pole at $x = 0$, so that return values of zero due to underflow must be accepted or otherwise dealt with.

- Otherwise, if $k > 1$, initialize with $c = 3k - 0.75$. Generate u_1 and compute $v_1 = u_1(1-u_1)$ and $v_2 = (u_1 - 0.5)\sqrt{c/v_1}$. If $x = k + v_2 - 1 \leq 0$, go back and generate new u_1; otherwise generate u_2 and compute $v_3 = 64 v_1^3 u_2^2$. If $v_3 \leq 1 - 2v_2^2/x$ **or** if $\ln v_3 \leq 2\{[k-1]\ln[x/(k-1)] - v_2\}$, accept x and stop; otherwise go back and generate new u_1.

42.4.7 *Binomial distribution*

Begin with $k = 0$ and generate u uniform in $[0, 1)$. Compute $P_k = (1-p)^n$ and store P_k into B. If $u \leq B$ accept $r_k = k$ and stop. Otherwise, increment k by one; compute the next P_k as $P_k \cdot (p/(1-p)) \cdot (n-k)/(k+1)$; add this to B. Again, if $u \leq B$, accept $r_k = k$ and stop, otherwise iterate until a value is accepted. If $p > 1/2$, it will be more efficient to generate r from $f(r; n, q)$, *i.e.*, with p and q interchanged, and then set $r_k = n - r$.

42.4.8 *Poisson distribution*

Iterate until a successful choice is made: Begin with $k = 1$ and set $A = 1$ to start. Generate u. Replace A with uA; if now $A < \exp(-\mu)$, where μ is the Poisson parameter, accept $n_k = k - 1$ and stop. Otherwise increment k by 1, generate a new u and repeat, always starting with the value of A left from the previous try.

Note that the Poisson generator used in ROOT's `TRandom` classes before version 5.12 (including the derived classes `TRandom1, TRandom2, TRandom3`) uses a Gaussian approximation when μ exceeds a given threshold. This may be satisfactory (and much faster) for some applications. To do this, generate z from a Gaussian with zero mean and unit standard deviation; then use $x = \max(0, [\mu + z\sqrt{\mu} + 0.5])$ where $[\,]$ signifies the greatest integer $\leq$ the expression. The routines from Numerical Recipes [15] and CLHEP's routine `RandPoisson` do not make this approximation.

42.4.9 *Student's t distribution*

Generate u_1 and u_2 uniform in $(0, 1)$; then $t = \sin(2\pi u_1)[n(u_2^{-2/n} - 1)]^{1/2}$ follows the Student's t distribution for $n > 0$ degrees of freedom (n not necessarily an integer).

Alternatively, generate x from a Gaussian with mean 0 and $\sigma^2 = 1$ according to the method of 42.4.4. Next generate y, an independent gamma random variate, according to 42.4.6 with $\lambda = 1/2$ and $k = n/2$. Then $z = x/\sqrt{y/n}$ is distributed as a t with n degrees of freedom.

For the special case $n = 1$, the Breit-Wigner distribution, generate u_1 and u_2; set $v_1 = 2u_1 - 1$ and $v_2 = 2u_2 - 1$. If $v_1^2 + v_2^2 \leq 1$ accept $z = v_1/v_2$ as a Breit-Wigner distribution with unit area, center at 0.0, and FWHM 2.0. Otherwise start over. For center M_0 and FWHM Γ, use $W = z\Gamma/2 + M_0$.

42.4.10 *Beta distribution*

The choice of an appropriate algorithm for generation of beta distributed random numbers depends on the values of the parameters α and β. For, *e.g.*, $\alpha = 1$, one can use the transformation method to find $x = 1 - u^{1/\beta}$, and similarly if $\beta = 1$ one has $x = u^{1/\alpha}$. For more general cases see, *e.g.*, Refs. [19, 20] and references therein.

42.5 Importance sampling and weighted Monte Carlo

Often the goal of a Monte Carlo calculation is to determine an expectation value of a function $h(x)$, where x is a (single or vector) random variable that follows a pdf $f(x)$,

$$E_f[h(x)] = \int h(x) f(x)\, dx \equiv \mu \; . \tag{42.10}$$

A Monte-Carlo estimator $\hat{\mu}_{\rm MC}$ for μ is the average of N values of $h(x)$ where x is sampled (generated) from $f(x)$, i.e., $\hat{\mu}_{\rm MC} = \frac{1}{N}\sum_{i=1}^{N} h(x_i)$. This has a variance

$$V[\hat{\mu}_{\rm MC}] = \frac{1}{N} V_f[h(x)] = \frac{1}{N}\left(E_f[h^2(x)] - \mu^2\right) \; . \tag{42.11}$$

By using the method of *importance sampling*, one can achieve a reduction in this variance and thus a more accurate determination of the expectation value for a given number of random values generated. The key idea is to rewrite the expectation value in Eq. (42.10) as

$$\mu = \int h(x) f(x)\, dx = \int \frac{h(x) f(x)}{g(x)}\, g(x)\, dx = E_g\left[\frac{h(x) f(x)}{g(x)}\right] \; , \tag{42.12}$$

where $g(x)$ is any other pdf of x with the same support as $f(x)$ (i.e., nonzero for the same region of x). Thus the desired quantity μ is the expectation value with respect to g of $h(x)f(x)/g(x)$. It can be estimated by generating N values of x sampled from g and computing

$$\hat{\mu}_{\rm IS} = \frac{1}{N}\sum_{i=1}^{N} \frac{h(x_i) f(x_i)}{g(x_i)} \; . \tag{42.13}$$

The variance of $\hat{\mu}_{\rm IS}$ is given by

$$V[\hat{\mu}_{\rm IS}] = \frac{1}{N} V_g\left[\frac{h(x) f(x)}{g(x)}\right] = \frac{1}{N}\left(E_g\left[\frac{h^2(x) f^2(x)}{g^2(x)}\right] - \mu^2\right) \; , \tag{42.14}$$

By choosing $g(x)$ such that $h(x)f(x)/g(x)$ is as constant as possible, the variance of $\hat{\mu}_{\rm IS}$ can be substantially reduced. One can show (see, e.g., Refs. [21, 22]) that the variance is minimized when $g(x) \propto |h(x)| f(x)$.

An alternative importance sampling estimator can be constructed by replacing the number of generated values N in Eq. (42.13) by the sum $\sum_{i=1}^{N} f(x_i)/g(x_i)$. This can given an even smaller variance at the price of a small bias. It can be of further advantage in problems where the pdf $f(x)$ is known only up to a normalization constant, which then cancels (see Refs. [21, 22]).

A closely related application of importance sampling is the use of weighted Monte Carlo to compute the probability $P_f(x \in A)$ for x to be in a specified region A:

$$P_f(x \in A) = \int_A f(x)\, dx \ . \tag{42.15}$$

It may be, however, that one does not have an MC model capable of generating $x \sim f(x)$, but rather one can generate x according to a different density $g(x)$. The probability $P_f(x \in A)$ can be written

$$P_f(x \in A) = \frac{\int_A \frac{f(x)}{g(x)}\, g(x)\, dx}{\int_A g(x)\, dx} \int_A g(x)\, dx = E_g[w(x)|x \in A]\, P_g(x \in A) \ , \tag{42.16}$$

where $w(x) = f(x)/g(x)$ is the *weight function* and $P_g(x \in A) = \int_A g(x)\, dx$ is the probability to find $x \in A$ assuming $x \sim g(x)$. That is, $P_f(x \in A)$ is the conditional expectation value of $w(x)$ with respect to $g(x)$ given $x \in A$ multiplied by the probability to find $x \in A$ under assumption of $g(x)$.

Suppose N values of x are generated according to $g(x)$ and m of them are found in the region A. Then the probability to be in A for $x \sim g(x)$ can be estimated by m/N, and the expectation value above can be obtained from the average of the weights in A. Therefore the desired probability $P_f(x \in A)$ can be estimated using

$$\hat{P}_f(x \in A) = \frac{1}{m} \sum_{i=1}^{m} w_i \ \times \ \frac{m}{N} = \frac{1}{N} \sum_{i=1}^{m} w_i \ , \tag{42.17}$$

where $w_i = w(x_i)$ and the sum includes only the m values of x found in A. That is, when generating the x values according to $g(x)$ instead of $f(x)$, the number of events m found in A is replaced by the corresponding sum of weights. The variance of $\hat{P}_f(x \in A)$ can be found from

$$\widehat{V}[\hat{P}_f(x \in A)] = \frac{1}{N^2} \sum_{i=1}^{m} w_i^2 \ . \tag{42.18}$$

By choosing $g(x)$ so that a larger fraction of x values are sampled in the "important" region A, one can reduce the variance of the estimated probability for a given total number of generated values.

42.6 Markov Chain Monte Carlo

In applications involving generation of random numbers following a multivariate distribution with a high number of dimensions, the transformation method may not be possible and the acceptance-rejection technique may have too low of an efficiency to be practical. If it is not required to have independent random values, but only that they follow a certain distribution, then Markov Chain Monte Carlo (MCMC) methods can be used. In depth treatments of MCMC can be found, *e.g.*, in the texts by Robert and Casella [21], Liu [22], and the review by Neal [23]. HEP-oriented software for MCMC is available from the Bayesian Analysis Toolkit (BAT) [24, 25].

MCMC is particularly useful in connection with Bayesian statistics, where a p.d.f. $p(\boldsymbol{\theta})$ for an n-dimensional vector of parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$ is obtained, and one needs the marginal distribution of a subset of the components. Here one samples $\boldsymbol{\theta}$ from $p(\boldsymbol{\theta})$ and simply records the marginal distribution for the components of interest.

A simple and broadly applicable MCMC method is the Metropolis-Hastings algorithm, which allows one to generate multidimensional points $\boldsymbol{\theta}$ distributed according to a target p.d.f. that is proportional to a given function $p(\boldsymbol{\theta})$. It is not necessary to have $p(\boldsymbol{\theta})$ normalized to unit area, which is useful in Bayesian statistics, as posterior probability densities are often determined only up to an unknown normalization constant.

To generate points that follow $p(\boldsymbol{\theta})$, one first needs a proposal p.d.f. $q(\boldsymbol{\theta}; \boldsymbol{\theta}_0)$, which can be (almost) any p.d.f. from which independent random values $\boldsymbol{\theta}$ can be generated, and which contains as a parameter another point in the same space $\boldsymbol{\theta}_0$. For example, a multivariate Gaussian centered about $\boldsymbol{\theta}_0$ can be used. Beginning at an arbitrary starting point $\boldsymbol{\theta}_0$, the Hastings algorithm iterates the following steps:

1. Generate a value $\boldsymbol{\theta}$ using the proposal density $q(\boldsymbol{\theta}; \boldsymbol{\theta}_0)$;
2. Form the Hastings test ratio, $\alpha = \min\left[1, \frac{p(\boldsymbol{\theta})q(\boldsymbol{\theta}_0;\boldsymbol{\theta})}{p(\boldsymbol{\theta}_0)q(\boldsymbol{\theta};\boldsymbol{\theta}_0)}\right]$;
3. Generate a value u uniformly distributed in $[0, 1]$;
4. If $u \leq \alpha$, take $\boldsymbol{\theta}_1 = \boldsymbol{\theta}$. Otherwise, repeat the old point, *i.e.*, $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0$.
5. Set $\boldsymbol{\theta}_0 = \boldsymbol{\theta}_1$ and return to step 1.

If one takes the proposal density to be symmetric in $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_0$, then this is the *Metropolis*-Hastings algorithm, and the test ratio becomes $\alpha = \min[1, p(\boldsymbol{\theta})/p(\boldsymbol{\theta}_0)]$. That is, if the proposed $\boldsymbol{\theta}$ is at a value of probability higher than $\boldsymbol{\theta}_0$, the step is taken. If the proposed step is rejected, the old point is repeated.

Methods for assessing and optimizing the performance of the algorithm are discussed in, *e.g.*, Refs. [21–23]. One can, for example, examine the autocorrelation as a function of the lag k, *i.e.*, the correlation of a sampled point with that k steps removed. This should decrease as quickly as possible for increasing k.

Generally one chooses the proposal density so as to optimize some quality measure such as the autocorrelation. For certain problems it has been shown that one achieves optimal performance when the acceptance fraction, that is, the fraction of points with $u \leq \alpha$, is around 40%. This can be adjusted by varying the width of the proposal density. For example, one can use for the proposal p.d.f. a multivariate Gaussian with the same covariance matrix as that of the target p.d.f., but scaled by a constant.

References

[1] F. James and L. Moneta, Computing and Software for Big Science **4**, 1, 2 (2020).

[2] P. L'Ecuyer, *Proc. 1997 Winter Simulation Conference*, IEEE Press, Dec. 1997, 127–134.

[3] L. Lonnblad, Comput. Phys. Commun. **84**, 307 (1994).

[4] R. Brun and F. Rademakers, Nucl. Instrum. Meth. **A389**, 81 (1997); See also `root.cern.ch`.

[5] International Organization for Standardization/International Electrotechnical Commission, *Information technology – Programming languages – C++*, ISO/IEC 14882:2020, International Organization for Standardization, Geneva, Switzerland, sixth edition (2020).

[6] M. Galli, E. Tejedor and S. Wunsch, EPJ Web Conf. **245**, 06004 (2020).

[7] Much of DIEHARD is described in: G. Marsaglia, *A Current View of Random Number Generators*, keynote address, *Computer Science and Statistics: 16th Symposium on the Interface*, Elsevier (1985).

[8] P. L'Ecuyer and R. Simard, *ACM Transactions on Mathematical Software* 33, 4, Article 1, December 2007.

[9] M. Matsumoto and T. Nishimura, *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998, 3–30.

[10] Hahnfeld, Jonas and Moneta, Lorenzo, EPJ Web Conf. **251**, 03008 (2021).

[11] M. Luscher, Comput. Phys. Commun. **79**, 100 (1994), [hep-lat/9309020].

[12] K. G. Savvidy, Computer Physics Communications **196**, 161 (2015), ISSN 0010-4655.

[13] F. James and L. Moneta, Comput. Softw. Big Sci. **4**, 2. 12 p (2019), 4 figures, [arXiv:1903.01247], URL `http://cds.cern.ch/record/2678858`.

[14] UNU.RAN is described at `statmath.wu.ac.at/software/ unuran`; See also W. Hörmann, J. Leydold, and G. Derflinger, *Automatic Nonuniform Random Variate Generation*, (Springer, New York, 2004).

[15] W.H. Press *et al.*, *Numerical Recipes*, 3rd edition, (Cambridge University Press, New York, 2007).

[16] J.H. Ahrens and U. Dieter, Computing **12**, 223 (1974).

[17] R.Y. Rubinstein, *Simulation and the Monte Carlo Method*, (John Wiley and Sons, Inc., New York, 1981).

[18] L. Devroye, *Non-Uniform Random Variate Generation*, (Springer-Verlag, New York, 1986); Available online at `luc.devroye.org/rnbookindex.html`.

[19] C. Walck, *Handbook on Statistical Distributions for Experimentalists*, University of Stockholm Report SUF-PFY/96-01, available from `www.fysik.su.se/~walck`.

[20] J.E. Gentle, *Random Number Generation and Monte Carlo Methods*, 2nd ed., (Springer, New York, 2003).

[21] C.P. Robert and G. Casella, *Monte Carlo Statistical Methods*, 2nd ed., (Springer, New York, 2004).

[22] J.S. Liu, *Monte Carlo Strategies in Scientific Computing*, (Springer, New York, 2001).

[23] R.M. Neal, *Probabilistic Inference Using Markov Chain Monte Carlo Methods*, Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto, available from `www.cs.toronto.edu/~radford/res-mcmc.html` .

[24] A. Caldwell, D. Kollar, K. Kröninger, Comput. Phys. Commun. 180 (2009) pages 2197-2209.

[25] Schulz, O., Beaujean, F., Caldwell, A. et al., SN Computer Science 2, 210 (2021).