Random Numbers and Monte Carlo Methods

Methods which make use of random numbers are often called Monte Carlo Methods after the Monte Carlo Casino in Monaco which has long been famous for games of chance.

Monte Carlo methods are useful in:

Simulation: Random numbers are used to simulate natural phenomena. In nuclear physics, neutrons moving through a reactor are subject to random collisions. In operations research, people enter an airport at random intervals of time.

Sampling: When it is not possible to examine all possible cases, typical cases chosen “at random” can be useful in characterizing properties of the system.

Numerical analysis: Many very complicated numerical problems can be solved approximately using random numbers.

Computer programming: Random input data is often used to test computer programs.

Decision making: Random numbers are often used to construct optimal playing strategies in the theory of games.

Recreation: Many computer games make essential use of random numbers.
Random numbers

There is no such thing as one random number. A number has a definite value: there is no randomness associated with it.

When we talk about random numbers, we mean a sequence of numbers \( x_i \) chosen from a set according to some probability distribution \( P(x) \). In a truly random sequence, \( x_i \) does not depend on (i.e., cannot be predicted from) the previous values \( x_j, j < i \).

A simple example is provided by tosses of a coin. The set has two members, heads \( (H) \) and tails \( (T) \). The probability distribution is \( P(H) = P(T) = \frac{1}{2} \). Given a sequence of coin tosses \( H, H, T, H, T, H \), the next toss cannot be predicted: it is equally likely to be heads or tails.

Uniform deviates

A sequence of uniform random deviates is one chosen from the set of real numbers in the interval \([0, 1)\) with constant probability

\[
P(x)dx = \begin{cases} 
  dx & \text{if } 0 \leq x < 1 \\
  0 & \text{otherwise}
\end{cases}
\]

This is a very basic type of random sequence which can be used to generate more complicated sequences with non-uniform probability distributions. For example, if \( x_i \) are uniform deviates, then \( y_i = -\lambda \log(1-x_i) \)
are exponentially distributed in $[0, \infty)$ with probability

$$P(y)dy = \frac{1}{\lambda}e^{-y/\lambda}dy.$$

**Pseudo random number generators**

Suppose that we need a sequence of uniform deviates in the form of random double's in a numerical application. We can obtain such a sequence using coin tosses as follows: Toss a coin 32 times and construct an unsigned long int by filling the 32 bit positions with 0 for heads and 1 for tails. This provides a random integer $x_i$ in the range $[0, 2^{32} = 4294967296)$ chosen with uniform probability $P(x) = \frac{1}{4294967296}$. The sequence $x_i P(x_i)$ provides uniform random deviates.

The problem with this algorithm is that it is too slow for practical calculations!

A faster way of generating such a sequence is to use a deterministic computer program.

For example J. von Neumann proposed using the Middle square method. “His idea was to take the square of the previous random number and to extract the middle digits; for example, if we are generating 10-digit numbers and the previous value was 5772156649, we square it to get

$$33317792380594909201$$

and the next number is therefore 7923805949.” – D.E. Knuth, *Section 3.1 The Art of Computer Programming.*
The C++ Standard Library header cstdlib defines a generator \texttt{rand()} which can be used for simple applications, but is not adequate for serious numerical work.

The C++-11 Standard Library header \texttt{random} defines several better generators, including the very popular \textbf{Mersenne Twister}. Some C++ compilers have not fully implemented the C++-11 standard.

\textbf{Monte Carlo integration}

The simplest Monte Carlo algorithm to estimate a one-dimensional integral is:

\[ I = \int_a^b dx \, f(x) \simeq \frac{b - a}{N} \sum_{n=1}^{N} f(x_i) , \]

where \( x_n, n = 1 \ldots N \) is a sequence of \( N \) uniformly distributed random numbers in the interval \([a, b]\).

It is important to estimate the error in this result. One way of doing this is to repeat the “measurement” many times. If we use independent random number sequences for each measurement we will obtain a different result each time:

\[ I_m = \frac{b - a}{N} \sum_{n=1}^{N} f(x_{m,n}) , \quad m = 1 \ldots M , \]

where \( x_{m,n}, n = 1 \ldots N \) is the \( m \)-th sequence of random numbers. Just like is done in a real experiment,
the error in a measurement repeated many times is estimated as the standard deviation from the mean

\[ \sigma_M = \sqrt{\frac{1}{M} \sum_{m=1}^{M} I_m^2 - \left( \frac{1}{M} \sum_{m=1}^{M} I_m \right)^2} = \sqrt{\frac{1}{M} \sum_{m=1}^{M} \left( I_m - \frac{1}{M} \sum_{m'=1}^{M} I_{m'} \right)^2}. \]

Let’s denote the average of all \( M \times N \) function evaluations as

\[ \bar{f} = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} f(x_{m,n}), \]

and let the deviation from this average be denoted

\[ \delta f_{m,n} = f(x_{m,n}) - \bar{f}. \]

Then we can write

\[ \sigma_M^2 = \frac{(b - a)^2}{MN^2} \sum_{m=1}^{M} \left( \sum_{n=1}^{N} \delta f_{m,n} \right)^2 \]

\[ = \frac{(b - a)^2}{MN^2} \sum_{m=1}^{M} \left( \sum_{n=1}^{N} \delta f_{m,n} \right) \left( \sum_{n'=1}^{N} \delta f_{m,n'} \right). \]

We divide the double sum over \( n, n' \) into two sets of terms, the first in which \( n = n' \) and the second in which \( n \neq n' \). Consider first the terms with \( n \neq n' \) which involves:

\[ \sum_{n=1}^{N} \sum_{n' \neq n}^{N} \sum_{m=1}^{M} \delta f_{m,n} \delta f_{m,n'}. \]
Since the deviations $\delta f_{m,n}$ and $\delta f_{m,n}'$ are independent and randomly distributed around zero, the sum will vanish, or strictly speaking be of $\mathcal{O}(1/\sqrt{MN(N-1)/2})$. The terms with $n = n'$ however are all positive:

$$\sigma^2_M = \frac{(b - a)^2}{MN^2} \sum_{m=1}^{M} \sum_{n=1}^{N} \delta f^2_{m,n} = \frac{(b - a)^2}{N} \left[ \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} \delta f^2_{m,n} \right] = \frac{(b - a)^2}{N} \sigma_f^2,$$

where

$$\sigma_f^2 = \overline{f^2} - (\overline{f})^2 = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} f(x_{m,n})^2 - \left( \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} f(x_{m,n}) \right)^2.$$

Thus, the simple Monte Carlo result with error estimate is

$$I = \int_{a}^{b} dx \ f(x) \simeq (b - a) \left[ \frac{1}{N} \sum_{n=1}^{N} f(x_i) \pm \frac{\sigma_f}{\sqrt{N}} \right].$$

**Example: computing $\pi$ in 1 dimension**

Consider the integral

$$\int_{0}^{1} \frac{dx}{1 + x^2} = \tan^{-1}(1) - \tan^{-1}(0) = \frac{\pi}{4}.$$

The following program `pi.cpp` evaluates this integral using the simple Monte Carlo algorithm and estimates the error in two different ways:
using the Monte Carlo error estimate

\[
\frac{(b - a)\sigma_f}{\sqrt{N}},
\]

for a single trial with \( N \) integration points, and

repeating the trial \( M \) times and computing the mean and standard deviation from the mean.

---

```python
# Monte Carlo integration in 1 dimension

from math import pi, sqrt
from random import random
from sys import stdout

a = 0.0
b = 1.0

def f(x):
    return 4.0 / (1.0 + x**2)

N = int(input(" Enter number of integration points N: "))
M = int(input(" Enter number of trials M: "))
```
I_M = 0.0
sigma_M = 0.0
sigmaAverage = 0.0

stdout.write(" Trial Integral MC Error Actual Error\n" +
   " --------------------------------------------------\n")

for m in range(1, M + 1):
    I = 0.0
    sigma = 0.0
    for n in range(1, N + 1):
        x = a + (b - a) * random()
        fx = f(x)
        I += fx
        sigma += fx**2
    I /= N
    sigma /= N
    sigma -= I**2
    I *= b - a
    sigma = (b - a) * sqrt(sigma / N)
    stdout.write(" " + str(m).ljust(8) + str(I)[:12].ljust(15) + str(sigma)[:12].ljust(15) + str(I - pi)[:12] + "\n")
I_M += I
sigma_M += I**2
sigmaAverage += sigma

stdout.write(" --------------------------------------------------\n")
I_M /= M
sigma_M /= M
sigma_M -= I_M**2
sigma_M = sqrt(sigma_M)
sigmaAverage /= M
stdout.write(" Average " + str(I_M)[:12].ljust(15) + str(sigmaAverage)[:12].ljust(15) + str(I_M - pi)[:12] + "\n" + " Standard Deviation " + str(sigma_M)[:12] + "\n" + " (Std. Dev.)/sqrt(M) " + str(sigma_M / sqrt(float(M))[:12] + "\n")
stdout.write(" --------------------------------------------------\n")

// Monte Carlo integration in 1 dimension

#include <cmath>
#include <cstdlib>
#include <iostream>

pi.cpp
```cpp
#include <iomanip>
#include <tools/random.hpp>  // for Mersenne Twister generator

const double pi = 4*atan(1.0);
double a = 0, b = 1;
double f (double x) {
    return 4/(1+x*x);
}

int main () {
    cout << "Enter number of integration points N: ";
    int N;
    cin >> N;
    cout << "Enter number of trials M: ";
    int M;
    cin >> M;
    double I_M = 0, sigma_M = 0, sigmaAverage = 0;
    cout << " Trial " << "Integral " << "MC Error " << "Actual Error " << "\n ";
    for (int i = 0; i < 50; i++) cout << '-'; cout << endl;
```
for (int m = 1; m <= M; m++) {
    double I = 0, sigma = 0;
    for (int n = 1; n <= N; n++) {
        double x = a + (b-a)*uniform_dist();
        double fx = f(x);
        I  += fx;
        sigma  += fx*fx;
    }
    I  /= N;
    sigma  /= N;
    sigma  -= I*I;
    I  *= b-a;
    sigma  = (b-a)*sqrt(sigma/N);
    cout.setf(ios::left);
    cout << ' ' << setw(8) << m << setw(15) << I
         << setw(15) << sigma << I-pi << endl;
    I_M  += I;
    sigma_M  += I*I;
    sigmaAverage  += sigma;
}
cout << " ";
for (int i = 0; i < 50; i++) cout << '-'; cout << endl;
The two error estimates should agree if the integration points are genuinely random. The output of the program is shown on the next page. Note that

- The Monte Carlo error estimates for each trial are in the ballpark of 0.0020 with an average of 0.00203415. The standard deviation error estimate of 0.0025356 is consistent with the average Monte Carlo error estimate.

- The average of the actual errors over the 10 trials has magnitude 0.0000931473, which is somewhat smaller than the standard deviation estimate divided by $\sqrt{10} = 0.000801829$. This is an anomaly. With a larger number of trials, these two estimates are also generally consistent.
Enter number of integration points N: 100000
Enter number of trials M: 10

<table>
<thead>
<tr>
<th>Trial</th>
<th>Integral</th>
<th>MC Error</th>
<th>Actual Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14295</td>
<td>0.00203401</td>
<td>0.00136188</td>
</tr>
<tr>
<td>2</td>
<td>3.14646</td>
<td>0.0020283</td>
<td>0.00487098</td>
</tr>
<tr>
<td>3</td>
<td>3.14371</td>
<td>0.00202878</td>
<td>0.00212204</td>
</tr>
<tr>
<td>4</td>
<td>3.13896</td>
<td>0.00204009</td>
<td>-0.00263496</td>
</tr>
<tr>
<td>5</td>
<td>3.14179</td>
<td>0.00203735</td>
<td>0.000193409</td>
</tr>
<tr>
<td>6</td>
<td>3.14032</td>
<td>0.00203623</td>
<td>-0.00126935</td>
</tr>
<tr>
<td>7</td>
<td>3.13847</td>
<td>0.0020362</td>
<td>-0.00312668</td>
</tr>
<tr>
<td>8</td>
<td>3.14382</td>
<td>0.00202985</td>
<td>0.00222704</td>
</tr>
<tr>
<td>9</td>
<td>3.13912</td>
<td>0.00203721</td>
<td>-0.00247249</td>
</tr>
<tr>
<td>10</td>
<td>3.13939</td>
<td>0.00203346</td>
<td>-0.00220335</td>
</tr>
</tbody>
</table>

Average 3.1415 0.00203415 -9.31473e-05

Standard Deviation 0.0025356
(Std. Dev.)/sqrt(M) 0.000801829
Importance Sampling

We have seen that uniform sampling can be used to estimate an integral

\[
\int_a^b f(x) \, dx \simeq \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \pm \frac{(b-a)\sigma_f}{\sqrt{N}},
\]

where the random numbers \(x_i\) are uniformly distributed in the interval \([a, b]\), and the variance of the function \(f\) is

\[
\sigma_f^2 = \left( \frac{1}{b-a} \int_a^b f^2(x) \, dx \right) - \left( \frac{1}{b-a} \int_a^b f(x) \, dx \right)^2 \simeq \left( \frac{1}{N} \sum_{i=1}^{N} f^2(x_i) \right) - \left( \frac{1}{N} \sum_{i=1}^{N} f(x_i) \right)^2.
\]

Consider however, the following integral:

\[
\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x^2 e^{-x^2/2} \, dx = 1.
\]

There are two problems with using uniform sampling to estimate this integral:

\begin{itemize}
  \item \((b - a) = \infty\), so a cut-off such as \(b = -a = L \gg 1\) must be used, and
  \item \(f(x)\) is exponentially small almost everywhere except near \(x = 0\), so the variance of \(f\) will be large.
\end{itemize}
These problems can be avoided by using importance sampling. Let $w(x)$ be a weight function which is positive definite everywhere in the integration interval and normalized to unity:

$$w(x) > 0 \quad \text{for} \quad a < x < b \,, \quad \int_a^b w(x) \, dx = 1 \,.$$  

The weight function is chosen to reduce the variance of the integrand by means of a change of variable

$$y = \int_a^x w(x') \, dx' \,, \quad \int_a^b f(x) \, dx = \int_0^1 \frac{f(x(y))}{w(x(y))} \, dy \,.$$  

If $w(x)$ is chosen to be large where $|f(x)|$ is large, and small where $|f(x)|$ is small, then $f(x)/w(x)$ can have a much smaller variance than $f(x)$.

The integral can now be estimated by uniform sampling in $y$, or equivalently, by sampling in $x$ with probability $w(x)$. For example, to estimate the Gaussian integral example on the previous page, let

$$w(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}} \,, \quad \text{so} \quad \frac{f(x)}{w(x)} = x^2 \,.$$  

The function `gasdev` in the header file `rng.h` can be used to generate random numbers with probability $w(x)$, as shown in the following program:

```cpp
// estimate Gaussian integral using uniform and importance sampling

#include <cmath>
```
#include <cstdlib>
#include <iostream>
using namespace std;

cost double pi = 4 * atan(1.0);

#include ".../tools/random.hpp"

double f(double x)
{
    return x * x * exp(- x * x / 2) / sqrt(2 * pi);
}

double f_over_w(double x)
{
    return x * x;
}

int main() {

    // get input parameters from user
    cout << "Enter number of points N: ";
int N;
cin >> N;
cout << "Enter cut-off L for uniform sampling: ";
double L;
cin >> L;

// uniform sampling
double avg = 0;
double var = 0;
for (int i = 0; i < N; i++) {
    double x = (2 * uniform_dist() - 1) * L;
    double fx = f(x);
    avg += fx;
    var += fx * fx;
}
avg /= N;
var /= N;
var = var - avg * avg;
cout << "\n  Uniform sampling: " << 2 * L * avg << " +- " << 2 * L * sqrt(var / N) << endl;

// importance sampling
avg = var = 0;
for (int i = 0; i < N; i++) {
    double x = normal_dist();
    double fx = f_over_w(x);
    avg += fx;
    var += fx * fx;
}
avg /= N;
var /= N;
var = var - avg * avg;
cout << "Importance sampling: " << avg << " +- "
    << sqrt(var / N) << endl;

cout << "Exact answer: " << 1.0 << endl;
}

# estimate Gaussian integral using uniform and importance sampling

from math import pi, sqrt
from random import random, normalvariate
double f_over_w(x):
    return x**2

# get input parameters from user
N = int(input(" Enter number of points N: "))
L = float(input(" Enter cut-off L for uniform sampling: "))

# uniform sampling
avg = 0.0
var = 0.0
for i in range(N):
    x = (2 * random() - 1) * L;
    fx = f(x)
    avg += fx
    var += fx**2
    print()
print(" Uniform sampling: " + str(2 * L * avg) + " +- ")
```python
# importance sampling
avg = var = 0.0
for i in range(N):
    x = normalvariate();
    fx = f_over_w(x)
    avg += fx
    var += fx**2
avg /= N
var /= N
var = var - avg**2
print("Importance sampling: " + str(avg) + " +- " + str(sqrt(var / N)))
print(" Exact answer: " + str(1.0))
```

Run this program and note the following:

- The error estimate for importance sampling agrees with the actual error, and decreases like $1/\sqrt{N}$.
- If the cut-off $L$ for uniform sampling is chosen too small, i.e., $\sim 1$, then the error estimate is small but does not agree with the actual error which is much larger.
If the cut-off $L$ for uniform sampling is chosen $\gg 1$, then the error estimate agrees with the actual error, but they are both much bigger than the error in importance sampling.

The error in both sampling methods is $O(1/\sqrt{N})$ and both methods converge at the same rate.

It is possible to get good results using uniform sampling by choosing the cut-off carefully so that it is not too big or too small. By contrast, importance sampling does not require a cut-off.