Computational Physics Lectures: Introduction to Monte Carlo methods

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

2017

Monte Carlo methods, plan for the lectures

1. Intro, MC integration and probability distribution functions (PDFs)
2. More on integration, PDFs, MC integration and random walks.
3. Random walks and statistical physics.
4. Statistical physics and the Ising and Potts models
5. Quantum Monte Carlo

Monte Carlo: Enhances algorithmic thinking!

- Be able to generate random variables following a given probability distribution function PDF
- Find a probability distribution function (PDF)
- Sampling rule for accepting a move
- Compute standard deviation and other expectation values
- Techniques for improving errors

© 1999-2017, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

Domains and probabilities

Consider the following simple example, namely the tossing of a dice, resulting in the following possible values

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$\{1/36, 2/36/3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36\}.$$

Monte Carlo methods, tossing a dice

The numbers in the domain are the outcomes of the physical process tossing the dice. We cannot tell beforehand whether the outcome is 3 or 5 or any other number in this domain. This defines the randomness of the outcome, or unexpectedness or any other synonimous word which encompasses the uncertitude of the final outcome.

The only thing we can tell beforehand is that say the outcome 2 has a certain probability. If our favorite hobby is to spend an hour every evening throwing dice and registering the sequence of outcomes, we will note that the numbers in the above domain

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\},$$

appear in a random order. After 11 throws the results may look like

$$\{10, 8, 6, 3, 6, 9, 11, 8, 12, 4, 5\}.$$

Stochastic variables

Random variables are characterized by a domain which contains all possible values that the random value may take. This domain has a corresponding PDF.

Stochastic variables and the main concepts, the discrete case

There are two main concepts associated with a stochastic variable. The *domain* is the set $\mathbb{D} = \{x\}$ of all accessible values the variable can assume, so that $X \in \mathbb{D}$. An example of a discrete domain is the set of six different numbers that we may get by throwing of a dice, $x \in \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$.

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur

$$p(x) = \text{Prob}(X = x).$$

Stochastic variables and the main concepts, the continuous case

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$ then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a,\, b]$ is then just the integral

$$\text{Prob}(a \leq X \leq b) = \int_a^b p(x)dx.$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

The cumulative probability

Of interest to us is the *cumulative probability distribution function* (**CDF**), $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x

$$P(x) = \text{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'.$$

The relation between a CDF and its corresponding PDF is then

$$p(x) = \frac{d}{dx}P(x).$$

Properties of PDFs

There are two properties that all PDFs must satisfy. The first one is positivity (assuming that the PDF is normalized)

$$0 \leq p(x) \leq 1.$$

Naturally, it would be nonsensical for any of the values of the domain to occur with a probability greater than 1 or less than 0. Also, the PDF must be normalized. That is, all the probabilities must add up to unity. The probability of "anything" to happen is always unity. For both discrete and continuous PDFs, this condition is

$$\sum_{x_i \in \mathbb{D}} p(x_i) = 1,$$

$$\int_{x \in \mathbb{D}} p(x)\, dx = 1.$$

Important distributions, the uniform distribution

The first one is the most basic PDF; namely the uniform distribution

$$p(x) = \frac{1}{b-a}\theta(x-a)\theta(b-x), \tag{1}$$

with

$$\begin{array}{ll} \theta(x) = 0 & x < 0 \\ \theta(x) = \frac{1}{b-a} & \in [a,b]. \end{array}$$

The normal distribution with $b = 1$ and $a = 0$ is used to generate random numbers.

Gaussian distribution

The second one is the Gaussian Distribution

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

with mean value μ and standard deviation σ. If $\mu = 0$ and $\sigma = 1$, it is normally called the **standard normal distribution**

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right),$$

The following simple Python code plots the above distribution for different values of μ and σ.

```
import numpy as np
from math import acos, exp, sqrt
from  matplotlib import pyplot as plt
from matplotlib import rc, rcParams
import matplotlib.units as units
import matplotlib.ticker as ticker
rc('text',usetex=True)
rc('font',**{'family':'serif','serif':['Gaussian distribution']})
font = {'family' : 'serif',
        'color'  : 'darkred',
        'weight' : 'normal',
        'size'   : 16,
        }
pi = acos(-1.0)
mu0 = 0.0
sigma0 = 1.0
mu1= 1.0
sigma1 = 2.0
mu2 = 2.0
sigma2 = 4.0

x = np.linspace(-20.0, 20.0)
v0 = np.exp(-(x*x-2*x*mu0+mu0*mu0)/(2*sigma0*sigma0))/sqrt(2*pi*sigma0*sigma0)
v1 = np.exp(-(x*x-2*x*mu1+mu1*mu1)/(2*sigma1*sigma1))/sqrt(2*pi*sigma1*sigma1)
v2 = np.exp(-(x*x-2*x*mu2+mu2*mu2)/(2*sigma2*sigma2))/sqrt(2*pi*sigma2*sigma2)
plt.plot(x, v0, 'b-', x, v1, 'r-', x, v2, 'g-')
```

```
plt.title(r'{\bf Gaussian distributions}', fontsize=20)
plt.text(-19, 0.3, r'Parameters: $\mu = 0$, $\sigma = 1$', fontdict=font)
plt.text(-19, 0.18, r'Parameters: $\mu = 1$, $\sigma = 2$', fontdict=font)
plt.text(-19, 0.08, r'Parameters: $\mu = 2$, $\sigma = 4$', fontdict=font)
plt.xlabel(r'$x$',fontsize=20)
plt.ylabel(r'$p(x)$ [MeV]',fontsize=20)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.savefig('gaussian.pdf', format='pdf')
plt.show()
```

Exponential distribution

Another important distribution in science is the exponential distribution

$$p(x) = \alpha \exp -(\alpha x).$$

Expectation values

Let $h(x)$ be an arbitrary continuous function on the domain of the stochastic variable X whose PDF is $p(x)$. We define the *expectation value* of h with respect to p as follows

$$\langle h \rangle_X \equiv \int h(x)p(x)\,dx \tag{2}$$

Whenever the PDF is known implicitly, like in this case, we will drop the index X for clarity. A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows

$$\langle x^n \rangle \equiv \int x^n p(x)\,dx$$

Stochastic variables and the main concepts, mean values

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ

$$\langle x \rangle = \mu \equiv \int xp(x)dx,$$

for a continuous distribution and

$$\langle x \rangle = \mu \equiv \frac{1}{N}\sum_{i=1}^{N} x_i p(x_i),$$

for a discrete distribution. Qualitatively it represents the centroid or the average value of the PDF and is therefore simply called the expectation value of $p(x)$.

Stochastic variables and the main concepts, central moments, the variance

A special version of the moments is the set of *central moments*, the n-th central moment defined as

$$\langle (x-\langle x\rangle)^n\rangle \equiv \int (x-\langle x\rangle)^n p(x)\,dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{Var}(X)$

$$\begin{aligned}
\sigma_X^2 = \mathrm{Var}(X) = \langle (x-\langle x\rangle)^2\rangle &= \int (x-\langle x\rangle)^2 p(x)dx \\
&= \int \left(x^2 - 2x\langle x\rangle^2 + \langle x\rangle^2\right) p(x)dx \\
&= \langle x^2\rangle\rangle - 2\langle x\rangle\langle x\rangle + \langle x\rangle^2 \\
&= \langle x^2\rangle - \langle x\rangle^2
\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x-\langle x\rangle)^2\rangle}$ is called the **standard deviation** of p. It is the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the "spread" of p around its mean.

First Illustration of the Use of Monte-Carlo Methods, integration

With this definition of a random variable and its associated PDF, we attempt now a clarification of the Monte-Carlo strategy by using the evaluation of an integral as our example.

In discussion on numerical integration we went through standard methods for evaluating an integral like

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i),$$

where ω_i are the weights determined by the specific integration method (like Simpson's method) with x_i the given mesh points. To give you a feeling of how we are to evaluate the above integral using Monte-Carlo, we employ here the crudest possible approach. Later on we will present slightly more refined approaches. This crude approach consists in setting all weights equal 1, $\omega_i = 1$. That corresponds to the rectangle method

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given value $x_{i-1/2}$.

First Illustration of the Use of Monte-Carlo Methods, integration

Setting $h = (b-a)/N$ where $b = 1$, $a = 0$, we can then rewrite the above integral as

$$I = \int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $x_{i-1/2}$ are the midpoint values of x. Introducing the concept of the average of the function f for a given PDF $p(x)$ as

$$\langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i)p(x_i),$$

and identify $p(x)$ with the uniform distribution, viz $p(x) = 1$ when $x \in [0,1]$ and zero for all other values of x. The integral is is then the average of f over the interval $x \in [0,1]$

$$I = \int_0^1 f(x)dx \approx \langle f \rangle.$$

First Illustration of the Use of Monte-Carlo Methods, variance in integration

In addition to the average value $\langle f \rangle$ the other important quantity in a Monte-Carlo calculation is the variance σ^2 and the standard deviation σ. We define first the variance of the integral with f for a uniform distribution in the interval $x \in [0,1]$ to be

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \langle f \rangle)^2 p(x_i),$$

and inserting the uniform distribution this yields

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2,$$

or

$$\sigma_f^2 = \left(\langle f^2 \rangle - \langle f \rangle^2\right).$$

Monte-Carlo integration, meaning of variance

The variance is nothing but a measure of the extent to which f deviates from its average over the region of integration. The standard deviation is defined as the square root of the variance. If we consider the above results for a fixed value of N as a measurement, we could recalculate the above average and variance for a series of different measurements. If each such measumerent produces a set of

averages for the integral I denoted $\langle f\rangle_l$, we have for M measurements that the integral is given by

$$\langle I\rangle_M = \frac{1}{M}\sum_{l=1}^{M}\langle f\rangle_l.$$

First Illustration of the Use of Monte-Carlo Methods, integration

If we can consider the probability of correlated events to be zero, we can rewrite the variance of these series of measurements as (equating $M = N$)

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2\rangle - \langle f\rangle^2\right) = \frac{\sigma_f^2}{N}. \tag{3}$$

We note that the standard deviation is proportional to the inverse square root of the number of measurements

$$\sigma_N \sim \frac{1}{\sqrt{N}}.$$

Important aspects of Monte-Carlo Methods

The aim of Monte Carlo calculations is to have σ_N as small as possible after N samples. The results from one sample represents, since we are using concepts from statistics, a 'measurement'.

Why Monte Carlo integration?

The scaling in the previous equation is clearly unfavorable compared even with the trapezoidal rule. We saw that the trapezoidal rule carries a truncation error

$$\text{error} \sim O(h^2),$$

with h the step length. In general, methods based on a Taylor expansion such as the trapezoidal rule or Simpson's rule have a truncation error which goes like $\sim O(h^k)$, with $k \geq 1$. Recalling that the step size is defined as $h = (b-a)/N$, we have an error which goes like

$$\text{error} \sim N^{-k}.$$

Why Monte Carlo integration?

Monte Carlo integration is more efficient in higher dimensions. To see this, let us assume that our integration volume is a hypercube with side L and dimension d. This cube contains hence $N = (L/h)^d$ points and therefore the error in the result scales as $N^{-k/d}$ for the traditional methods.

The error in the Monte carlo integration is however independent of d and scales as

$$\text{error} \sim 1/\sqrt{N}.$$

Always!

Comparing this error with that of the traditional methods, shows that Monte Carlo integration is more efficient than an algorithm with error in powers of k when

$$d > 2k.$$

Why Monte Carlo integration? Example

In order to expose this, consider the definition of the quantum mechanical energy of a system consisting of 10 particles in three dimensions. The energy is the expectation value of the Hamiltonian H and reads

$$E = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) H(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)},$$

where Ψ is the wave function of the system and $\mathbf{R}_i$ are the coordinates of each particle. If we want to compute the above integral using for example Gaussian quadrature and use for example ten mesh points for the ten particles, we need to compute a ten-dimensional integral with a total of 10^{30} mesh points. As an amusing exercise, assume that you have access to today's fastest computer with a theoretical peak capacity of more than one Petaflops, that is 10^{15} floating point operations per second. Assume also that every mesh point corresponds to one floating operation per second. Estimate then the time needed to compute this integral with a traditional method like Gaussian quadrature and compare this number with the estimated lifetime of the universe, $T \approx 4.7 \times 10^{17}$s. Do you have the patience to wait?

Monte Carlo integration, simple example

We end this first part with a discussion of a brute force Monte Carlo program which integrates

$$\int_0^1 dx \frac{4}{1+x^2} = \pi,$$

where the input is the desired number of Monte Carlo samples.

Monte Carlo integration, simple example

What we are doing is to employ a random number generator to obtain numbers x_i in the interval $[0,1]$ through a call to one of the library functions $ran0$, $ran1$, $ran2$ or $ran3$ which generate random numbers in the interval $x \in [0,1]$. These functions will be discussed in the next section. Here we simply employ these functions in order to generate a random variable. All random number generators

produce pseudo-random numbers in the interval $[0,1]$ using the so-called uniform probability distribution $p(x)$ defined as

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$

with $a = 0$ og $b = 1$ and where Θ is the standard Heaviside function or simply the step function.

Monte Carlo integration, simple example

If we have a general interval $[a,b]$, we can still use these random number generators through a change of variables

$$z = a + (b-a)x,$$

with x in the interval $x \in [0,1]$.

Monte Carlo integration, simple example

The present approach to the above integral is often called 'crude' or 'Brute-Force' Monte-Carlo. Later on in this chapter we will study refinements to this simple approach. The reason is that a random generator produces points that are distributed in a homogenous way in the interval $[0,1]$. If our function is peaked around certain values of x, we may end up sampling function values where $f(x)$ is small or near zero. Better schemes which reflect the properties of the function to be integrated are thence needed.

Monte Carlo integration, algorithm

The algorithm is as follows

- Choose the number of Monte Carlo samples N.
- Perform a loop over N and for each step generate a random number x_i in the interval $[0,1]$ through a call to a random number generator.
- Use this number to evaluate $f(x_i)$.
- Evaluate the contributions to the mean value and the standard deviation for each loop.
- After N samples calculate the final mean value and the standard deviation.

Monte Carlo integration, simple example, the program

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

//     Here we define various functions called by the main program
//     this function defines the function to integrate

double func(double x);

//     Main function begins here
int main()
{
     int n;
     double MCint, MCintsqr2, fx, Variance;
     cout << "Read in the number of Monte-Carlo samples" << endl;
     cin >> n;
     MCint = MCintsqr2=0.;
     double invers_period = 1./RAND_MAX; // initialise the random number generator
     srand(time(NULL));  // This produces the so-called seed in MC jargon
//   evaluate the integral with the a crude Monte-Carlo method
     for ( int i = 1;  i <= n; i++){
  // obtain a floating number x in [0,1]
           double x = double(rand())*invers_period;
           fx = func(x);
           MCint += fx;
           MCintsqr2 += fx*fx;
     }
     MCint = MCint/((double) n );
     MCintsqr2 = MCintsqr2/((double) n );
     double variance=MCintsqr2-MCint*MCint;
//   final output
     cout << " variance= " << variance << " Integral = " << MCint << " Exact= " << M_PI << endl;
}  // end of main program
// this function defines the function to integrate
double func(double x)
{
  double value;
  value = 4/(1.+x*x);
  return value;
} // end of function to evaluate
```

Monte Carlo integration, simple example and the results

N	I	σ_N
10	3.10263E+00	3.98802E-01
100	3.02933E+00	4.04822E-01
1000	3.13395E+00	4.22881E-01
10000	3.14195E+00	4.11195E-01
100000	3.14003E+00	4.14114E-01
1000000	3.14213E+00	4.13838E-01
10000000	3.14177E+00	4.13523E-01
10^9	3.14162E+00	4.13581E-01

We note that as N increases, the integral itself never reaches more than an agreement to the fourth or fifth digit. The variance also oscillates around its exact value $4.13581E-01$.

Testing against the trapezoidal rule for a one-dimensional integral

The following simple Python code, with pertaining plot shows the relative error for the above integral using a brute force Monte Carlo approach and the trapezoidal rule. Running the python code shows that the trapezoidal rule is clearly superior in this case. With importance sampling and multi-dimensional integrals, the Monte Carl method takes over again.

```
from  matplotlib import pyplot as plt
from math import exp, acos, log10
import numpy as np
import random

# function for the trapezoidal rule
def TrapezoidalRule(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b))+s
   return h*s
# function to perform the Monte Carlo calculations
def MonteCarloIntegration(f,n):
    sum = 0
# Define the seed for the rng
    random.seed()
    for i in range (1, n, 1):
        x = random.random()
        sum = sum +f(x)
    return sum/n

#  function to compute
def function(x):
    return 4/(1+x*x)

# Integration limits for the Trapezoidal rule
a = 0.0; b = 1.0
exact = acos(-1.0)
# set up the arrays for plotting the relative error
log10n = np.zeros(6); Trapez = np.zeros(6); MCint = np.zeros(6);
# find the relative error as function of integration points
for i in range(1, 6):
    npts = 10**(i+1)
    log10n[i] = log10(npts)
    Trapez[i] = log10(abs((TrapezoidalRule(a,b,function,npts)-exact)/exact))
    MCint[i] = log10(abs((MonteCarloIntegration(function,npts)-exact)/exact))
plt.plot(log10n, Trapez ,'b-',log10n, MCint,'g-')
plt.axis([1,6,-14.0, 0.0])
plt.xlabel('$\log_{10}(n)$')
plt.ylabel('Relative error')
```

```
plt.title('Relative errors for Monte Carlo integration and Trapezoidal rule')
plt.legend(['Trapezoidal rule', 'Brute force Monte Carlo integration'], loc='best')
plt.savefig('mcintegration.pdf')
plt.show()
```

Second example, particles in a box

We give here an example of how a system evolves towards a well defined equilibrium state.

Consider a box divided into two equal halves separated by a wall. At the beginning, time $t = 0$, there are N particles on the left side. A small hole in the wall is then opened and one particle can pass through the hole per unit time.

After some time the system reaches its equilibrium state with equally many particles in both halves, $N/2$. Instead of determining complicated initial conditions for a system of N particles, we model the system by a simple statistical model. In order to simulate this system, which may consist of $N \gg 1$ particles, we assume that all particles in the left half have equal probabilities of going to the right half. We introduce the label n_l to denote the number of particles at every time on the left side, and $n_r = N - n_l$ for those on the right side.

Second example, particles in a box

The probability for a move to the right during a time step Δt is n_l/N. The algorithm for simulating this problem may then look like this

- Choose the number of particles N.

b* Make a loop over time, where the maximum time (or maximum number of steps) should be larger than the number of particles N.

- For every time step Δt there is a probability n_l/N for a move to the right. Compare this probability with a random number x.
- If $x \leq n_l/N$, decrease the number of particles in the left half by one, i.e., $n_l = n_l - 1$. Else, move a particle from the right half to the left, i.e., $n_l = n_l + 1$.
- Increase the time by one unit (the external loop).

Second example, particles in a box

In this case, a Monte Carlo sample corresponds to one time unit Δt.

The following simple C/C++-program illustrates this model.

```
// Particles in a box
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
```

```
using namespace  std;

ofstream ofile;
int main(int argc, char* argv[])
{
  char *outfilename;
  int initial_n_particles, max_time, time, random_n, nleft;
  long idum;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  // Read in data
  cout << "Initial number of particles = " << endl ;
  cin >> initial_n_particles;
  // setup of initial conditions
  nleft = initial_n_particles;
  max_time = 10*initial_n_particles;
  idum = -1;
  // sampling over number of particles
  for( time=0; time <= max_time; time++){
    random_n = ((int) initial_n_particles*ran0(&idum));
    if ( random_n <= nleft){
      nleft -= 1;
    }
    else{
      nleft += 1;
    }
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << time;
    ofile << setw(15) << nleft << endl;
  }
  return 0;
} // end main function
```

Second example, particles in a box, discussion

If we denote $\langle n_l \rangle$ as the number of particles in the left half as a time average after *equilibrium is reached*, we can define the standard deviation as

$$\sigma = \sqrt{\langle n_l^2 \rangle - \langle n_l \rangle^2}. \tag{4}$$

This problem has also an analytic solution to which we can compare our numerical simulation.

Second example, particles in a box, discussion

If $n_l(t)$ is the number of particles in the left half after t moves, the change in $n_l(t)$ in the time interval Δt is

$$\Delta n = \left(\frac{N - n_l(t)}{N} - \frac{n_l(t)}{N} \right) \Delta t,$$

and assuming that n_l and t are continuous variables we arrive at

$$\frac{dn_l(t)}{dt} = 1 - \frac{2n_l(t)}{N},$$

whose solution is

$$n_l(t) = \frac{N}{2} \left(1 + e^{-2t/N} \right),$$

with the initial condition $n_l(t = 0) = N$. Note that we have assumed n to be a continuous variable. Obviously, particles are discrete objects.

Simple demonstration using python

The following simple Python code implements the above algorithm for particles in a box and plots the final number of particles in each part of the box.

```
#!/usr/bin/env python
from  matplotlib import pyplot as plt
from math import exp
import numpy as np
import random

# initial number of particles
N0 = 1000
MaxTime = 10*N0
values = np.zeros(MaxTime)
time = np.zeros(MaxTime)
random.seed()
# initial number of particles in left half
nleft = N0
for t in range (0, MaxTime, 1):
    if N0*random.random() <= nleft:
       nleft -= 1
    else:
       nleft += 1
    time[t] = t
    values[t] = nleft

# Finally we plot the results
plt.plot(time, values,'b-')
plt.axis([0,MaxTime, N0/4, N0])
plt.xlabel('$t$')
plt.ylabel('$N$')
plt.title('Number of particles in left half')
plt.savefig('box.pdf')
plt.show()
```

The produced figure shows the development of this system as function of time steps. We note that for $N = 1000$ after roughly 2000 time steps, the system has reached the equilibrium state. There are however noteworthy fluctuations around equilibrium.

Brief Summary

In essence the Monte Carlo method contains the following ingredients

- A PDF which characterizes the system
- Random numbers which are generated so as to cover in an as uniform as possible way on the unity interval [0,1].
- A sampling rule
- An error estimation
- Techniques for improving the errors

Probability Distribution Functions

The following table collects properties of probability distribution functions. In our notation we reserve the label $p(x)$ for the probability of a certain event, while $P(x)$ is the cumulative probability.

	Discrete PDF	Continuous PDF
Domain	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
Probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \leq p(x_i) \leq 1$	$p(x) \geq 0$
Positivity	$0 \leq P_i \leq 1$	$0 \leq P(x) \leq 1$
Monotonic	$P_i \geq P_j$ if $x_i \geq x_j$	$P(x_i) \geq P(x_j)$ if $x_i \geq x_j$
Normalization	$P_N = 1$	$P(b) = 1$

Probability Distribution Functions

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i),$$

if we have a discrete PDF or

$$\langle x^k \rangle = \int_a^b x^k p(x) dx,$$

in the case of a continuous PDF. We have already defined the mean value μ and the variance σ^2.

The three famous Probability Distribution Functions

There are at least three PDFs which one may encounter. These are the

Uniform distribution

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$

yielding probabilities different from zero in the interval $[a, b]$.

The exponential distribution

$$p(x) = \alpha \exp(-\alpha x),$$

yielding probabilities different from zero in the interval $[0, \infty)$ and with mean value

$$\mu = \int_0^\infty xp(x)dx = \int_0^\infty x\alpha \exp(-\alpha x)dx = \frac{1}{\alpha},$$

with variance

$$\sigma^2 = \int_0^\infty x^2 p(x)dx - \mu^2 = \frac{1}{\alpha^2}.$$

Probability Distribution Functions, the normal distribution

Finally, we have the so-called univariate normal distribution, or just the **normal distribution**

$$p(x) = \frac{1}{b\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2b^2}\right)$$

with probabilities different from zero in the interval $(-\infty, \infty)$. The integral $\int_{-\infty}^{\infty} \exp\left(-(x^2)dx\right.$ appears in many calculations, its value is $\sqrt{\pi}$, a result we will need when we compute the mean value and the variance. The mean value is

$$\mu = \int_0^\infty xp(x)dx = \frac{1}{b\sqrt{2\pi}} \int_{-\infty}^{\infty} x \exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

which becomes with a suitable change of variables

$$\mu = \frac{1}{b\sqrt{2\pi}} \int_{-\infty}^{\infty} b\sqrt{2}(a + b\sqrt{2}y) \exp -y^2 dy = a.$$

Probability Distribution Functions, the normal distribution

Similarly, the variance becomes

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}} \int_{-\infty}^{\infty} (x-\mu)^2 \exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

and inserting the mean value and performing a variable change we obtain

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(b\sqrt{2}y)^2 \exp\left(-y^2\right)dy = \frac{2b^2}{\sqrt{\pi}}\int_{-\infty}^{\infty} y^2 \exp\left(-y^2\right)dy,$$

and performing a final integration by parts we obtain the well-known result $\sigma^2 = b^2$. It is useful to introduce the standard normal distribution as well, defined by $\mu = a = 0$, viz. a distribution centered around zero and with a variance $\sigma^2 = 1$, leading to

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right). \tag{5}$$

Probability Distribution Functions, the cumulative distribution

The exponential and uniform distributions have simple cumulative functions, whereas the normal distribution does not, being proportional to the so-called error function $erf(x)$, given by

$$P(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right)dt,$$

which is difficult to evaluate in a quick way.

Probability Distribution Functions, other important distribution

Some other PDFs which one encounters often in the natural sciences are the binomial distribution

$$p(x) = \left(\begin{array}{c} n \\ x \end{array}\right) y^x (1-y)^{n-x} \qquad x = 0, 1, \ldots, n,$$

where y is the probability for a specific event, such as the tossing of a coin or moving left or right in case of a random walker. Note that x is a discrete stochastic variable.

The sequence of binomial trials is characterized by the following definitions

- Every experiment is thought to consist of N independent trials.
- In every independent trial one registers if a specific situation happens or not, such as the jump to the left or right of a random walker.
- The probability for every outcome in a single trial has the same value, for example the outcome of tossing (either heads or tails) a coin is always 1/2.

Probability Distribution Functions, the binomial distribution

In order to compute the mean and variance we need to recall Newton's binomial formula

$$(a+b)^m = \sum_{n=0}^{m} \left(\begin{array}{c} m \\ n \end{array} \right) a^n b^{m-n},$$

which can be used to show that

$$\sum_{x=0}^{n} \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = (y+1-y)^n = 1,$$

the PDF is normalized to one. The mean value is

$$\mu = \sum_{x=0}^{n} x \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = \sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} y^x (1-y)^{n-x},$$

resulting in

$$\mu = \sum_{x=0}^{n} x \frac{(n-1)!}{(x-1)!(n-1-(x-1))!} y^{x-1} (1-y)^{n-1-(x-1)},$$

which we rewrite as

$$\mu = ny \sum_{\nu=0}^{n} \left(\begin{array}{c} n-1 \\ \nu \end{array} \right) y^{\nu} (1-y)^{n-1-\nu} = ny(y+1-y)^{n-1} = ny.$$

The variance is slightly trickier to get. It reads $\sigma^2 = ny(1-y)$.

Probability Distribution Functions, Poisson's distribution

Another important distribution with discrete stochastic variables x is the Poisson model, which resembles the exponential distribution and reads

$$p(x) = \frac{\lambda^x}{x!} e^{-\lambda} \qquad x = 0, 1, \dots, ; \lambda > 0.$$

In this case both the mean value and the variance are easier to calculate,

$$\mu = \sum_{x=0}^{\infty} x \frac{\lambda^x}{x!} e^{-\lambda} = \lambda e^{-\lambda} \sum_{x=1}^{\infty} \frac{\lambda^{x-1}}{(x-1)!} = \lambda,$$

and the variance is $\sigma^2 = \lambda$.

Probability Distribution Functions, Poisson's distribution

An example of applications of the Poisson distribution could be the counting of the number of α-particles emitted from a radioactive source in a given time interval. In the limit of $n \to \infty$ and for small probabilities y, the binomial distribution approaches the Poisson distribution. Setting $\lambda = ny$, with y the probability for an event in the binomial distribution we can show that

$$\lim_{n\to\infty} \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} e^{-\lambda} = \sum_{x=1}^{\infty} \frac{\lambda^x}{x!} e^{-\lambda}.$$

Meet the covariance!

An important quantity in a statistical analysis is the so-called covariance.

Consider the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \ldots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows

$$\begin{aligned} \mathrm{Cov}(X_i,\, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle & (6) \\ &= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n, & (7) \end{aligned}$$

with

$$\langle x_i \rangle = \int \cdots \int x_i P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n.$$

Meet the covariance in matrix disguise

If we consider the above covariance as a matrix

$$C_{ij} = \mathrm{Cov}(X_i,\, X_j),$$

then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{Cov}(X_i,\, X_i) = \mathrm{Var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated.

Meet the covariance, uncorrelated events

This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$

$$\begin{aligned} \mathrm{Cov}(X_i,\, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \end{aligned}$$

If X_i and X_j are independent, we get

$$\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle = \mathrm{Cov}(X_i, X_j) = 0 \quad (i \neq j).$$

Numerical experiments and the covariance

Now that we have constructed an idealized mathematical framework, let us try to apply it to empirical observations. Examples of relevant physical phenomena may be spontaneous decays of nuclei, or a purely mathematical set of numbers produced by some deterministic mechanism. It is the latter we will deal with, using so-called pseudo-random number generators. In general our observations will contain only a limited set of observables. We remind the reader that a *stochastic process* is a process that produces sequentially a chain of values

$$\{x_1, x_2, \dots x_k, \dots\}.$$

Numerical experiments and the covariance

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* (since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment). We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

Numerical experiments and the covariance, actual situations

In practical situations however, a sample is always of finite size. Let that size be n. The expectation value of a sample α, the **sample mean**, is then defined as follows

$$\langle x_\alpha \rangle \equiv \frac{1}{n} \sum_{k=1}^{n} x_{\alpha,k}.$$

The *sample variance* is:

$$\mathrm{Var}(x) \equiv \frac{1}{n} \sum_{k=1}^{n} (x_{\alpha,k} - \langle x_\alpha \rangle)^2,$$

with its square root being the *standard deviation of the sample*.

Numerical experiments and the covariance, our observables

You can think of the above observables as a set of quantities which define a given experiment. This experiment is then repeated several times, say m times. The total average is then

$$\langle X_m \rangle = \frac{1}{m}\sum_{\alpha=1}^{m} x_\alpha = \frac{1}{mn}\sum_{\alpha,k} x_{\alpha,k}, \tag{8}$$

where the last sums end at m and n. The total variance is

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}(\langle x_\alpha \rangle - \langle X_m \rangle)^2,$$

which we rewrite as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle). \tag{9}$$

Numerical experiments and the covariance, the sample variance

We define also the sample variance σ^2 of all mn individual experiments as

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)^2. \tag{10}$$

These quantities, being known experimental values or the results from our calculations, may differ, in some cases significantly, from the similarly named exact values for the mean value μ_X, the variance $\mathrm{Var}(X)$ and the covariance $\mathrm{Cov}(X,Y)$.

Numerical experiments and the covariance, central limit theorem

The central limit theorem states that the PDF $\tilde{p}(z)$ of the average of m random values corresponding to a PDF $p(x)$ is a normal distribution whose mean is the mean value of the PDF $p(x)$ and whose variance is the variance of the PDF $p(x)$ divided by m, the number of values used to compute z.

The central limit theorem leads then to the well-known expression for the standard deviation, given by

$$\sigma_m = \frac{\sigma}{\sqrt{m}}.$$

In many cases the above estimate for the standard deviation, in particular if correlations are strong, may be too simplistic. We need therefore a more precise defintion of the error and the variance in our results.

Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

$$\mu_X \approx X_m = \frac{1}{mn} \sum_{\alpha=1}^{m} \sum_{k=1}^{n} x_{\alpha,k}.$$

We can then use Eq. (9)

$$\sigma_m^2 = \frac{1}{mn^2} \sum_{\alpha=1}^{m} \sum_{kl=1}^{n} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

and rewrite it as

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{mn^2} \sum_{\alpha=1}^{m} \sum_{k<l}^{n} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

where the first term is the sample variance of all mn experiments divided by n and the last term is nothing but the covariance which arises when $k \neq l$.

Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

If the observables are uncorrelated, then the covariance is zero and we obtain a total variance which agrees with the central limit theorem. Correlations may often be present in our data set, resulting in a non-zero covariance. The first term is normally called the uncorrelated contribution. Computationally the uncorrelated first term is much easier to treat efficiently than the second. We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Definition of Correlation Functions and Standard Deviation

Let us analyze the problem by splitting up the correlation term into partial sums of the form

$$f_d = \frac{1}{nm} \sum_{\alpha=1}^{m} \sum_{k=1}^{n-d} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,k+d} - \langle X_m \rangle),$$

The correlation term of the total variance can now be rewritten in terms of f_d

$$\frac{2}{mn^2} \sum_{\alpha=1}^{m} \sum_{k<l}^{n} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle) = \frac{2}{n} \sum_{d=1}^{n-1} f_d$$

Definition of Correlation Functions and Standard Deviation

The value of f_d reflects the correlation between measurements separated by the distance d in the samples. Notice that for $d = 0$, f is just the sample variance, σ^2. If we divide f_d by σ^2, we arrive at the so called **autocorrelation function**

$$\kappa_d = \frac{f_d}{\sigma^2} \tag{11}$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d = 0$.

Definition of Correlation Functions and Standard Deviation, sample variance

The sample variance of the mn experiments can now be written in terms of the autocorrelation function

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{n} \cdot \sigma^2 \sum_{d=1}^{n-1} \frac{f_d}{\sigma^2} = \left(1 + 2\sum_{d=1}^{n-1} \kappa_d\right) \frac{1}{n}\sigma^2 = \frac{\tau}{n} \cdot \sigma^2 \tag{12}$$

and we see that σ_m can be expressed in terms of the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*

$$\tau = 1 + 2\sum_{d=1}^{n-1} \kappa_d \tag{13}$$

For a correlation free experiment, τ equals 1.

Definition of Correlation Functions and Standard Deviation

From the point of view of Eq. (12) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes

$$n_{\text{eff}} = \frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\sigma_m^2 \approx \sigma^2/n$ to be less than the true sample error. The estimate of the error will be too "good". On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large. The solution to this problem is given by more practically oriented methods like the blocking technique.

Random Numbers

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think random numbers are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

Random Numbers, better name: pseudo random numbers

A disclaimer is however appropriate. It should be fairly obvious that something as deterministic as a computer cannot generate purely random numbers.

Numbers generated by any of the standard algorithms are in reality pseudo random numbers, hopefully abiding to the following criteria:

- they produce a uniform distribution in the interval [0,1].
- correlations between random numbers are negligible
- the period before the same sequence of random numbers is repeated is as large as possible and finally
- the algorithm should be fast.

Random number generator RNG

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\mathrm{MOD}(M),$$

which yield a number in the interval [0,1] through

$$x_i = N_i/M$$

The number M is called the period and it should be as large as possible and N_0 is the starting value, or seed. The function MOD means the remainder, that is if we were to evaluate (13)MOD(9), the outcome is the remainder of the division 13/9, namely 4.

Random number generator RNG and periodic outputs

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.

Consider the following example

$$N_i = (6N_{i-1} + 7)\text{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, \ldots\ldots,$ i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\text{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \ldots,$ a period of just 2.

Random number generator RNG and its period

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose lth number is the sum of the $l-i$th and $l-j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\text{MOD}(M).$$

Random number generator RNG, other examples

Such a generator again produces a sequence of pseudorandom numbers but this time with a period much larger than M. It is also possible to construct more elaborate algorithms by including more than two past terms in the sum of each iteration. One example is the generator of Marsaglia and Zaman which consists of two congruential relations

$$N_l = (N_{l-3} - N_{l-1})\text{MOD}(2^{31} - 69), \tag{14}$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\text{MOD}(2^{32}), \tag{15}$$

which according to the authors has a period larger than 2^{94}.

Random number generator RNG, other examples

Instead of using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j})$$

where the bitwise action of $\oplus$ means that if $N_{l-i} = N_{l-j}$ the result is 0 whereas if $N_{l-i} \neq N_{l-j}$ the result is 1. As an example, consider the case where $N_{l-i} = 6$ and $N_{l-j} = 11$. The first one has a bit representation (using 4 bits only) which reads 0110 whereas the second number is 1011. Employing the $\oplus$ operator yields 1101, or $2^3 + 2^2 + 2^0 = 13$.

In Fortran90, the bitwise $\oplus$ operation is coded through the intrinsic function IEOR(m, n) where m and n are the input numbers, while in C it is given by $m \wedge n$.

Random number generator RNG, RAN0

We show here how the linear congruential algorithm can be implemented, namely

$$N_i = (aN_{i-1})\text{MOD}(M).$$

However, since a and N_{i-1} are integers and their multiplication could become greater than the standard 32 bit integer, there is a trick via Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

where we have defined

$$q = [M/a],$$

and

$$r = M \text{ MOD } a.$$

where the brackets denote integer division. In the code below the numbers q and r are chosen so that $r < q$.

Random number generator RNG, RAN0

To see how this works we note first that

$$(aN_{i-1})\text{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\text{MOD}(M), \tag{16}$$

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\text{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M.

Random number generator RNG, RAN0

We can now rewrite Eq. (16) as

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\mathrm{MOD}(M), \tag{17}$$

which results in

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\,\mathrm{MOD}(M), \tag{18}$$

yielding

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1}\mathrm{MOD}(q)) - [N_{i-1}/q]r))\,\mathrm{MOD}(M). \tag{19}$$

Random number generator RNG, RAN0

The term $[N_{i-1}/q]r$ is always smaller or equal $N_{i-1}(r/q)$ and with $r < q$ we obtain always a number smaller than N_{i-1}, which is smaller than M. And since the number $N_{i-1}\mathrm{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\mathrm{MOD}(q)) < aq$. Combined with our definition of $q = [M/a]$ ensures that this term is also smaller than M meaning that both terms fit into a 32-bit signed integer. None of these two terms can be negative, but their difference could. The algorithm below adds M if their difference is negative. Note that the program uses the bitwise $\oplus$ operator to generate the starting point for each generation of a random number. The period of $ran0$ is $\sim 2.1 \times 10^9$. A special feature of this algorithm is that is should never be called with the initial seed set to 0.

Random number generator RNG, RAN0 code

```
     /*
      ** The function
      **           ran0()
      ** is an "Minimal" random number generator of Park and Miller
      ** Set or reset the input value
      ** idum to any integer value (except the unlikely value MASK)
      ** to initialize the sequence; idum must not be altered between
      ** calls for sucessive deviates in a sequence.
      ** The function returns a uniform deviate between 0.0 and 1.0.
      */
double ran0(long &idum)
{
   const int a = 16807, m = 2147483647, q = 127773;
   const int r = 2836, MASK = 123459876;
   const double am = 1./m;
   long     k;
   double   ans;
   idum ^= MASK;
   k = (*idum)/q;
```

```
    idum = a*(idum - k*q) - r*k;
    // add m if negative difference
    if(idum < 0) idum += m;
    ans=am*(idum);
    idum ^= MASK;
    return ans;
} // End: function ran0()
```

Properties of Selected Random Number Generators

As mentioned previously, the underlying PDF for the generation of random numbers is the uniform distribution, meaning that the probability for finding a number x in the interval [0,1] is $p(x) = 1$.

A random number generator should produce numbers which are uniformly distributed in this interval. The table shows the distribution of $N = 10000$ random numbers generated by the functions in the program library. We note in this table that the number of points in the various intervals $0.0 - 0.1$, $0.1 - 0.2$ etc are fairly close to 1000, with some minor deviations.

Two additional measures are the standard deviation σ and the mean $\mu = \langle x \rangle$.

Properties of Selected Random Number Generators

For the uniform distribution, the mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886.$$

Properties of Selected Random Number Generators

The various random number generators produce results which agree rather well with these limiting values.

x-bin	ran0	ran1	ran2	ran3
0.0-0.1	1013	991	938	1047
0.1-0.2	1002	1009	1040	1030
0.2-0.3	989	999	1030	993
0.3-0.4	939	960	1023	937
0.4-0.5	1038	1001	1002	992
0.5-0.6	1037	1047	1009	1009
0.6-0.7	1005	989	1003	989
0.7-0.8	986	962	985	954
0.8-0.9	1000	1027	1009	1023
0.9-1.0	991	1015	961	1026
μ	0.4997	0.5018	0.4992	0.4990
σ	0.2882	0.2892	0.2861	0.2915

Simple demonstration of RNGs using python

The following simple Python code plots the distribution of the produced random numbers using the linear congruential RNG employed by Python. The trend displayed in the previous table is seen rather clearly.

```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random

# initialize the rng with a seed
random.seed()
counts = 10000
values = np.zeros(counts)
for i in range (1, counts, 1):
    values[i] = random.random()

# the histogram of the data
n, bins, patches = plt.hist(values, 10, facecolor='green')

plt.xlabel('$x$')
plt.ylabel('Number of counts')
plt.title(r'Test of uniform distribution')
plt.axis([0, 1, 0, 1100])
plt.grid(True)
plt.show()
```

Properties of Selected Random Number Generators

Since our random numbers, which are typically generated via a linear congruential algorithm, are never fully independent, we can then define an important test which measures the degree of correlation, namely the so-called auto-correlation function defined previously, see again Eq. (11). We rewrite it here as

$$C_k = \frac{f_d}{\sigma^2},$$

with $C_0 = 1$. Recall that $\sigma^2 = \langle x_i^2 \rangle - \langle x_i \rangle^2$ and that

$$f_d = \frac{1}{nm} \sum_{\alpha=1}^{m} \sum_{k=1}^{n-d} (x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,k+d} - \langle X_m \rangle),$$

The non-vanishing of C_k for $k \neq 0$ means that the random numbers are not independent. The independence of the random numbers is crucial in the evaluation of other expectation values. If they are not independent, our assumption for approximating σ_N in Eq. (3) is no longer valid.

Correlation function and which random number generators should I use

The program here computes the correlation function for one of the standard functions included with the c++ compiler.

```
//  This function computes the autocorrelation function for
//  the standard c++ random number generator

#include <fstream>
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;
// output file as global variable
ofstream ofile;

//     Main function begins here
int main(int argc, char* argv[])
{
     int n;
     char *outfilename;

     cin >> n;
     double MCint = 0.;      double MCintsqr2=0.;
     double invers_period = 1./RAND_MAX; // initialise the random number generator
     srand(time(NULL));  // This produces the so-called seed in MC jargon
     // Compute the variance and the mean value of the uniform distribution
     // Compute also the specific values x for each cycle in order to be able to
     // the covariance and the correlation function
     // Read in output file, abort if there are too few command-line arguments
     if( argc <= 2 ){
       cout << "Bad Usage: " << argv[0] <<
         " read also output file and number of cycles on same line" << endl;
       exit(1);
     }
     else{
       outfilename=argv[1];
     }
     ofile.open(outfilename);
     // Get  the number of Monte-Carlo samples
     n = atoi(argv[2]);
     double *X;
     X = new double[n];
     for (int i = 0;  i < n; i++){
           double x = double(rand())*invers_period;
           X[i] = x;
           MCint += x;
           MCintsqr2 += x*x;
     }
     double Mean = MCint/((double) n );
     MCintsqr2 = MCintsqr2/((double) n );
     double STDev = sqrt(MCintsqr2-Mean*Mean);
     double Variance = MCintsqr2-Mean*Mean;
//   Write mean value and standard deviation
     cout << " Standard deviation= " << STDev << " Integral = " << Mean << endl;

     // Now we compute the autocorrelation function
     double *autocor;  autocor = new double[n];
     for (int j = 0; j < n; j++){
       double sum = 0.0;
       for (int k = 0; k < (n-j); k++){
         sum  += (X[k]-Mean)*(X[k+j]-Mean);
       }
       autocor[j] = sum/Variance/((double) n );
       ofile << setiosflags(ios::showpoint | ios::uppercase);
       ofile << setw(15) << setprecision(8) << j;
```

```
        ofile << setw(15) << setprecision(8) << autocor[j] << endl;
      }
      ofile.close();  // close output file
      return 0;
}  // end of main program
```

Correlation function and which random number generators should I use

The following Python code plots the results for the correlation function from the above program.

```
import numpy as np
from  matplotlib import pyplot as plt
# Load in data file
data = np.loadtxt("datafiles/autocor.dat")
# Make arrays containing x-axis and binding energies as function of A
x = data[:,0]
corr = data[:,1]
plt.plot(x, corr ,'ro')
plt.axis([0,1000,-0.2, 1.1])
plt.xlabel(r'$d$')
plt.ylabel(r'$C_d$')
plt.title(r'autocorrelation function for RNG')
plt.savefig('autocorr.pdf')
plt.show()
```

Which RNG should I use?

- In the library files lib.cpp and lib.h we have included four popular RNGs taken from the widely used textbook Numerical Recipes. These are called ran0, ran1, ran2 and ran3.
- C++ has a class called **random**. The random class contains a large selection of RNGs and is highly recommended. Some of these RNGs have very large periods making it thereby very safe to use these RNGs in case one is performing large calculations. In particular, the Mersenne twister random number engine has a period of 2^{19937}.

How to use the Mersenne generator

The following part of a c++ code (from project 4) sets up the uniform distribution for $x \in [0, 1]$.

```
/*

//  You need this
#include <random>

// Initialize the seed and call the Mersienne algo
```

```
std::random_device rd;
std::mt19937_64 gen(rd());
// Set up the uniform distribution for x \in [[0, 1]
std::uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);

// Now use the RNG
int ix = (int) (RandomNumberGenerator(gen)*NSpins);
```

Improved Monte Carlo Integration

We have presented a simple brute force approach to integration with the Monte Carlo method. There we sampled over a given number of points distributed uniformly in the interval $[0, 1]$

$$I = \int_0^1 f(x)dx = \langle f \rangle.$$

Here we introduce two important steps which in most cases improve upon the above simple brute force approach with the uniform distribution, namely

- change of variables and
- importance sampling

Change of Variables

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \leq x \leq 1 \\ 0 & else \end{cases}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

All random number generators use the uniform distribution to generate numbers $x \in [0, 1]$.

Change of Variables

When we attempt a transformation to a new variable $x \to y$ we have to conserve the probability

$$p(y)dy = p(x)dx,$$

which for the uniform distribution implies

$$p(y)dy = dx.$$

Let us assume that $p(y)$ is a PDF different from the uniform PDF $p(x) = 1$ with $x \in [0, 1]$. If we integrate the last expression we arrive at

$$x(y) = \int_0^y p(y')dy',$$

which is nothing but the cumulative distribution of $p(y)$, i.e.,

$$x(y) = P(y) = \int_0^y p(y')dy'.$$

Transformed Uniform Distribution

Suppose we have the general uniform distribution

$$p(y)dy = \begin{cases} \frac{dy}{b-a} & a \leq y \leq b \\ 0 & else \end{cases}$$

If we wish to relate this distribution to the one in the interval $x \in [0, 1]$ we have

$$p(y)dy = \frac{dy}{b-a} = dx,$$

and integrating we obtain the cumulative function

$$x(y) = \int_a^y \frac{dy'}{b-a},$$

yielding

$$y = a + (b-a)x,$$

a well-known result!

Exponential Distribution

Assume that

$$p(y) = \exp(-y),$$

which is the exponential distribution, important for the analysis of e.g., radioactive decay. Again, $p(x)$ is given by the uniform distribution with $x \in [0, 1]$, and with the assumption that the probability is conserved we have

$$p(y)dy = \exp(-y)dy = dx,$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp(-y')dy' = 1 - \exp(-y),$$

or

$$y(x) = -\ln(1-x).$$

Exponential Distribution

This gives us the new random variable y in the domain $y \in [0, \infty)$ determined through the random variable $x \in [0, 1]$ generated by functions like $ran0$.

This means that if we can factor out $\exp(-y)$ from an integrand we may have

$$I = \int_0^\infty F(y)dy = \int_0^\infty \exp(-y)G(y)dy$$

which we rewrite as

$$\int_0^\infty \exp(-y)G(y)dy = \int_0^1 G(y(x))dx \approx \frac{1}{N}\sum_{i=1}^{N} G(y(x_i)),$$

where x_i is a random number in the interval $[0, 1]$.

Exponential Distribution

We have changed the integration limits in the second integral, since we have performed a change of variables. Since we have used the uniform distribution defined for $x \in [0, 1]$, the integration limits change to 0 and 1. The variable y is now a function of x. Note also that in practical implementations, our random number generators for the uniform distribution never return exactly 0 or 1, but we may come very close.

The algorithm for the last example is rather simple. In the function which sets up the integral, we simply need to call one of the random number generators like $ran0$, $ran1$, $ran2$ or $ran3$ in order to obtain numbers in the interval [0,1]. We obtain y by the taking the logarithm of $(1 - x)$. Our calling function which sets up the new random variable y may then include statements like

```
.....
idum=-1;
x=ran0(&idum);
y=-log(1.-x);
.....
```

Normal Distribution

For the normal distribution, expressed here as

$$g(x, y) = \exp(-(x^2 + y^2)/2)dxdy.$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt.$$

Both c++ and Fortran have this function as intrinsic ones.

Normal Distribution

We obviously would like to avoid computing an integral everytime we need a random variable. If we however switch to polar coordinates, we have for x and y

$$r = \left(x^2 + y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y},$$

resulting in

$$g(r, \theta) = r \exp\left(-r^2/2\right) dr d\theta,$$

where the angle θ could be given by a uniform distribution in the region $[0, 2\pi]$. Following example 1 above, this implies simply multiplying random numbers $x \in [0, 1]$ by 2π.

Normal Distribution

The variable r, defined for $r \in [0, \infty)$ needs to be related to to random numbers $x' \in [0, 1]$. To achieve that, we introduce a new variable

$$u = \frac{1}{2} r^2,$$

and define a PDF

$$\exp\left(-u\right) du,$$

with $u \in [0, \infty)$. Using the results from example 2 for the exponential distribution, we have

$$u = -\ln\left(1 - x'\right),$$

where x' is a random number generated for $x' \in [0, 1]$.

Normal Distribution

With

$$x = r\cos\left(\theta\right) = \sqrt{2u}\cos\left(\theta\right),$$

and

$$y = r\sin\left(\theta\right) = \sqrt{2u}\sin\left(\theta\right),$$

we can obtain new random numbers x, y through

$$x = \sqrt{-2\ln\left(1 - x'\right)}\cos\left(\theta\right),$$

and

$$y = \sqrt{-2\ln\left(1 - x'\right)}\sin\left(\theta\right),$$

with $x' \in [0, 1]$ and $\theta \in 2\pi[0, 1]$.

Normal Distribution

A function which yields such random numbers for the normal distribution would include statements like

```
.....
idum=-1;
radius=sqrt(-2*ln(1.-ran0(idum)));
theta=2*pi*ran0(idum);
x=radius*cos(theta);
y=radius*sin(theta);
.....
```

Importance Sampling

With the aid of the above variable transformations we address now one of the most widely used approaches to Monte Carlo integration, namely importance sampling.

Let us assume that $p(y)$ is a PDF whose behavior resembles that of a function F defined in a certain interval $[a, b]$. The normalization condition is

$$\int_a^b p(y)dy = 1.$$

We can rewrite our integral as

$$I = \int_a^b F(y)dy = \int_a^b p(y)\frac{F(y)}{p(y)}dy.$$

Importance Sampling

Since random numbers are generated for the uniform distribution $p(x)$ with $x \in [0, 1]$, we need to perform a change of variables $x \to y$ through

$$x(y) = \int_a^y p(y')dy',$$

where we used

$$p(x)dx = dx = p(y)dy.$$

If we can invert $x(y)$, we find $y(x)$ as well.

Importance Sampling

With this change of variables we can express the integral of Eq. () as

$$I = \int_a^b p(y)\frac{F(y)}{p(y)}dy = \int_{\tilde{a}}^{\tilde{b}} \frac{F(y(x))}{p(y(x))}dx,$$

meaning that a Monte Carlo evaluation of the above integral gives

$$\int_{\tilde{a}}^{\tilde{b}} \frac{F(y(x))}{p(y(x))}dx = \frac{1}{N}\sum_{i=1}^{N} \frac{F(y(x_i))}{p(y(x_i))}.$$

Importance Sampling

Note the change in integration limits from a and b to $\tilde{a}$ and $\tilde{b}$. The advantage of such a change of variables in case $p(y)$ follows closely F is that the integrand becomes smooth and we can sample over relevant values for the integrand. It is however not trivial to find such a function p. The conditions on p which allow us to perform these transformations are

- p is normalizable and positive definite,
- it is analytically integrable and
- the integral is invertible, allowing us thereby to express a new variable in terms of the old one.

Importance Sampling

The variance is now with the definition

$$\tilde{F} = \frac{F(y(x))}{p(y(x))},$$

given by

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{F}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\tilde{F}\right)^2.$$

Importance Sampling

The algorithm for this procedure is

- Use the uniform distribution to find the random variable y in the interval [0,1]. The function $p(x)$ is a user provided PDF.
- Evaluate thereafter

$$I = \int_a^b F(x)dx = \int_a^b p(x)\frac{F(x)}{p(x)}dx,$$

by rewriting

$$\int_a^b p(x)\frac{F(x)}{p(x)}dx = \int_{\tilde{a}}^{\tilde{b}} \frac{F(x(y))}{p(x(y))}dy,$$

since

$$\frac{dy}{dx} = p(x).$$

- Perform then a Monte Carlo sampling for

$$\int_{\tilde{a}}^{\tilde{b}} \frac{F(x(y))}{p(x(y))} dy \approx \frac{1}{N} \sum_{i=1}^{N} \frac{F(x(y_i))}{p(x(y_i))},$$

with $y_i \in [0,1]$,

- and evaluate the variance as well.

Importance Sampling, a simple example

Let us look again at the integral

$$I = \int_0^1 F(x)dx = \int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4}.$$

We choose the following PDF (which follows closely the function to integrate)

$$p(x) = \frac{1}{3}(4-2x) \qquad \int_0^1 p(x)dx = 1,$$

resulting in

$$\frac{F(0)}{p(0)} = \frac{F(1)}{p(1)} = \frac{3}{4}.$$

Check that it fullfils the requirements of a PDF! We perform then the change of variables (via the Cumulative function)

$$y(x) = \int_0^x p(x')dx' = \frac{1}{3}x(4-x),$$

or

$$x = 2 - (4-3y)^{1/2}$$

We have that when $y = 0$ then $x = 0$ and when $y = 1$ we have $x = 1$.

Importance Sampling, a simple example, a simple plot

```
from  matplotlib import pyplot as plt
from math import exp, acos, log10
import numpy as np

#  function to integrate
def function(x):
    return 1.0/(1+x*x)

#  new probability
def newfunction(x):
    return 0.3333333*(4.0-2*x)

Dim = 100
x = np.linspace(0.0,1.0,Dim)
f = np.zeros(Dim)
```

```
g = np.zeros(Dim)
for i in xrange(Dim):
    f[i] = function(x[i])
    g[i] = newfunction(x[i])

plt.plot(x, f ,'b-',x, g,'g-')
plt.axis([0,1,0.5, 1.5])
plt.xlabel('$x$')
plt.ylabel('Functions')
plt.title('Similarities between functions')
plt.legend(['Integrand', 'New integrand'], loc='best')
plt.savefig('newprobability.pdf')
plt.show()
```

Importance Sampling, a simple example, the code part

```
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
       x = ran0(&idum);  // random numbers in [0,1]
       y = 2 - sqrt(4-3*x);  // new random numbers
       fy=3*func(y)/(4-2*y); // weighted function
       int_mc += fy;
       sum_sigma += fy*fy;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=(sum_sigma-int_mc*int_mc);
```

Importance Sampling, a simple example, and the results

The suffix *cr* stands for the brute force approach while *is* stands for the use of importance sampling. All calculations use ran0 as function to generate the uniform distribution.

N	I_{cr}	σ_{cr}	I_{is}	σ_{is}
10000	3.13395E+00	4.22881E-01	3.14163E+00	6.49921E-03
100000	3.14195E+00	4.11195E-01	3.14163E+00	6.36837E-03
1000000	3.14003E+00	4.14114E-01	3.14128E+00	6.39217E-03
10000000	3.14213E+00	4.13838E-01	3.14160E+00	6.40784E-03

However, it is unfair to study one-dimensional integrals with MC methods!

Acceptance-Rejection Method

This is a rather simple and appealing method after von Neumann. Assume that we are looking at an interval $x \in [a, b]$, this being the domain of the PDF $p(x)$. Suppose also that the largest value our distribution function takes in this interval is M, that is

$$p(x) \leq M \qquad x \in [a, b].$$

Then we generate a random number x from the uniform distribution for $x \in [a, b]$ and a corresponding number s for the uniform distribution between $[0, M]$. If

$$p(x) \geq s,$$

we accept the new value of x, else we generate again two new random numbers x and s and perform the test in the latter equation again.

Acceptance-Rejection Method

As an example, consider the evaluation of the integral

$$I = \int_0^3 \exp(x) dx.$$

Obviously to derive a closed-form expression is much easier, however the integrand could pose some more difficult challenges. The aim here is simply to show how to implent the acceptance-rejection algorithm. The integral is the area below the curve $f(x) = \exp(x)$. If we uniformly fill the rectangle spanned by $x \in [0, 3]$ and $y \in [0, \exp(3)]$, the fraction below the curve obtained from a uniform distribution, and multiplied by the area of the rectangle, should approximate the chosen integral.

Acceptance-Rejection Method

It is rather easy to implement this numerically, as shown in the following code.

```
//   Loop over Monte Carlo trials n
     integral =0.;
     for ( int i = 1;  i <= n; i++){
//   Finds a random value for x in the interval [0,3]
          x = 3*ran0(&idum);
//   Finds y-value between [0,exp(3)]
          y = exp(3.0)*ran0(&idum);
//   if the value of y at exp(x) is below the curve, we accept
          if ( y  < exp(x)) s = s+ 1.0;
//   The integral is area enclosed below the line f(x)=exp(x)
    }
//  Then we multiply with the area of the rectangle and divide by the number of cycles
    Integral = 3.*exp(3.)*s/n
```

Monte Carlo Integration of Multidimensional Integrals

When we deal with multidimensional integrals of the form

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \cdots \int_{a_d}^{b_d} dx_d g(x_1, \ldots, x_d),$$

with x_i defined in the interval $[a_i, b_i]$ we would typically need a transformation of variables of the form

$$x_i = a_i + (b_i - a_i) t_i,$$

if we were to use the uniform distribution on the interval $[0, 1]$.

Monte Carlo Integration of Multidimensional Integrals

In this case, we need a Jacobi determinant

$$\prod_{i=1}^{d}(b_i - a_i),$$

and to convert the function $g(x_1, \dots, x_d)$ to

$$g(x_1, \dots, x_d) \to g(a_1 + (b_1 - a_1)t_1, \dots, a_d + (b_d - a_d)t_d).$$

Monte Carlo Integration of Multidimensional Integrals

As an example, consider the following six-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dxdy} g(\mathbf{x}, \mathbf{y}),$$

where

$$g(\mathbf{x}, \mathbf{y}) = \exp(-\mathbf{x}^2 - \mathbf{y}^2)(\mathbf{x} - \mathbf{y})^2$$

with $d = 6$.

Monte Carlo Integration of Multidimensional Integrals

We can solve this integral by employing our brute force scheme, or using importance sampling and random variables distributed according to a gaussian PDF. For the latter, if we set the mean value $\mu = 0$ and the standard deviation $\sigma = 1/\sqrt{2}$, we have

$$\frac{1}{\sqrt{\pi}} \exp(-x^2),$$

and using this normal distribution we rewrite our integral as

$$\pi^3 \int \prod_{i=1}^{6} \left(\frac{1}{\sqrt{\pi}} \exp(-x_i^2) \right) (\mathbf{x} - \mathbf{y})^2 dx_1 \dots dx_6.$$

Monte Carlo Integration of Multidimensional Integrals

We rewrite it in a more compact form as

$$\int f(x_1, \dots, x_d) F(x_1, \dots, x_d) \prod_{i=1}^{6} dx_i,$$

where f is the above normal distribution and

$$F(x_1, \dots, x_6) = F(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^2,$$

Brute Force Integration

Below we list two codes, one for the brute force integration and the other employing importance sampling with a gaussian distribution.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double brute_force_MC(double *);
//     Main function begins here
int main()
{
     int n;
     double x[6], y, fx;
     double int_mc = 0.;  double variance = 0.;
     double sum_sigma= 0. ; long idum=-1 ;
     double length = 5.; // we fix the max size of the box to L=5
     double jacobidet = pow((2*length),6);
     cout << "Read in the number of Monte-Carlo samples" << endl;
     cin >> n;
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
//   x[] contains the random numbers for all dimensions
       for (int j = 0; j< 6; j++) {
           x[j]=-length+2*length*ran0(&idum);
       }
       fx=brute_force_MC(x);
       int_mc += fx;
       sum_sigma += fx*fx;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=sum_sigma-int_mc*int_mc;
//   final output
      cout << setiosflags(ios::showpoint | ios::uppercase);
      cout << " Monte carlo result= " << setw(10) << setprecision(8) << jacobidet*int_mc;
      cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt(variance/((double) n )) <<
     return 0;
}  // end of main program

// this function defines the integrand to integrate

double  brute_force_MC(double *x)
{
// evaluate the different terms of the exponential
   double xx=x[0]*x[0]+x[1]*x[1]+x[2]*x[2];
   double yy=x[3]*x[3]+x[4]*x[4]+x[5]*x[5];
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-xx-yy)*xy;
} // end function for the integrand
```

Importance Sampling

This code includes a call to the function $normal_random$, which produces random numbers from a gaussian distribution.

```
// importance sampling with gaussian deviates
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double gaussian_MC(double *);
double gaussian_deviate(long *);
//     Main function begins here
int main()
{
     int n;
     double x[6], y, fx;
     cout << "Read in the number of Monte-Carlo samples" << endl;
     cin >> n;
     double int_mc = 0.;  double variance = 0.;
     double sum_sigma= 0. ; long idum=-1 ;
     double jacobidet = pow(acos(-1.),3.);
     double sqrt2 = 1./sqrt(2.);
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
//   x[] contains the random numbers for all dimensions
       for (int j = 0; j < 6; j++) {
         x[j] = gaussian_deviate(&idum)*sqrt2;
       }
       fx=gaussian_MC(x);
       int_mc += fx;
       sum_sigma += fx*fx;
     }
     int_mc = int_mc/((double) n );
     sum_sigma = sum_sigma/((double) n );
     variance=sum_sigma-int_mc*int_mc;
//   final output
      cout << setiosflags(ios::showpoint | ios::uppercase);
      cout << " Monte carlo result= " << setw(10) << setprecision(8) << jacobidet*int_mc;
      cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt(variance/((double) n )) <<
     return 0;
}  // end of main program

// this function defines the integrand to integrate

double  gaussian_MC(double *x)
{
// evaluate the different terms of the exponential
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return xy;
} // end function for the integrand

// random numbers with gaussian distribution
double gaussian_deviate(long * idum)
{
  static int iset = 0;
  static double gset;
  double fac, rsq, v1, v2;

  if ( idum < 0) iset =0;
  if (iset == 0) {
    do {
      v1 = 2.*ran0(idum) -1.0;
      v2 = 2.*ran0(idum) -1.0;
```

```
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  } else {
    iset =0;
    return gset;
  }
} // end function for gaussian deviates
```

Python codes

The first code here is an example of a python which computes the above integral using the brute force approach

```
#Monte Carlo integration in 6 dimensions

import numpy,math
import sys

def integrand(x):
    """Calculates the integrand
    exp(-a*(x1^2+x2^2+...+x6^2)-b*[(x1-x4)^2+...+(x3-x6)^2])
    from the values in the 6-dimensional array x."""
    a = 1.0
    b = 0.5

    x2 = numpy.sum(x**2)
    xy = (x[0]-x[3])**2 + (x[1]-x[4])**2 + (x[2]-x[5])**2

    return numpy.exp(-a*x2-b*xy)

#Main program

#Integration limits: x[i] in (-5, 5)
L      = 5.0
jacobi = (2*L)**6

N = 100000

#Evaluate the integral
sum  = 0.0
sum2 = 0.0
for i in xrange(N):
    #Generate random coordinates to sample at
    x = numpy.array([-L+2*L*numpy.random.random() for j in xrange(6)])

    fx         = integrand(x)
    sum       += fx
    sum2      += fx**2
#Calculate expt. values for fx and fx^2
sum /=float(N)
sum2/=float(N)

#Result
int_mc  = jacobi*sum;
#Gaussian standard deviation
```

```
sigma   = jacobi*math.sqrt((sum2-sum**2)/float(N))

#Output
print "Montecarlo result = %10.8E" % int_mc
print "Sigma             = %10.8E" % sigma
```

Python codes, importance sampling

The second code, displayed here, uses importance sampling and random numbers that follow the normal distribution the brute force approach

```
#Monte Carlo integration with importance sampling

import numpy,math
import sys

def integrand(x):
    """Calculates the integrand
    exp(-b*[(x1-x4)^2+...+(x3-x6)^2])
    from the values in the 6-dimensional array x."""
    b = 0.5

    xy = (x[0]-x[3])**2 + (x[1]-x[4])**2 + (x[2]-x[5])**2
    return numpy.exp(-b*xy)

#Main program

#Jacobi determinant
jacobi = math.acos(-1.0)**3
sqrt2  = 1.0/math.sqrt(2)

N = 100000

#Evaluate the integral
sum  = 0.0
sum2 = 0.0
for i in xrange(N):
    #Generate random, gaussian distributed coordinates to sample at
    x = numpy.array([numpy.random.normal()*sqrt2 for j in xrange(6)])

    fx         = integrand(x)
    sum       += fx
    sum2      += fx**2
#Calculate expt. values for fx and fx^2
sum /=float(N)
sum2/=float(N)

#Result
int_mc  = jacobi*sum;
#Gaussian standard deviation
sigma   = jacobi*math.sqrt((sum2-sum**2)/float(N))

#Output
print "Montecarlo result = %10.8E" % int_mc
print "Sigma             = %10.8E" % sigma
```