

Random Numbers and Monte Carlo Methods

Methods which make use of *random numbers* are often called Monte Carlo Methods after the *Casino Monte Carlo* 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 $P(x) = \frac{1}{4294967296.0}$. 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. The following algorithm from *Numerical Recipes* is coded in a header file called `rng.h` which can be included in application programs:

```
// Random Number Generators from "Numerical Recipes"

#ifdef RNG_H_DEFINED
#define RNG_H_DEFINED
```

```
// Quick and dirty random number generator
// returns a uniform deviate in the interval [0,1)
// To seed the generator, reset qadseed
unsigned long qadseed = 123456789;
inline double qadran ( ) {
    qadseed = qadseed*1664525L + 1013904223L;
    return qadseed/4294967296.0;
}
```

This code uses a *linear congruential generator*. Given a *seed* x_0 and fixed integers m (modulus), a (multiplier), and c (increment), a uniform random sequence of integers in the range $[0, m)$ is constructed using the recurrence relation

$$x_i = (ax_{i-1} + c) \bmod m.$$

Note that this deterministic algorithm violates the essential property of true random numbers, namely x_i cannot be predicated in advance from the value of x_{i-1} . Such a sequence is called *pseudo-random*.

Another property of a deterministic pseudo-random sequence is that it is *periodic* with maximum period equal to the cardinality (number of distinct members) of the set. For the linear congruential algorithm, the set comprises the integers $0, 1, 2, \dots, m-1$ and the cardinality is the modulus m . By contrast, a true random sequence does not have a finite period: the cardinality of the set of coin toss outcomes is 2, but a sequence of coin tosses has period ∞ .

A *good pseudo-random sequence* is one which gives approximately the same results as a true random sequence when used in a particular application. Much research has been done, and continues to be done, on finding good pseudo-random algorithms for various applications.

A cleaner random number generator from *Numerical Recipes*

Chapter 7 of *Numerical Recipes* recommends several random number generators with good properties. The

code for `ran2` is included in the header file `rng.h`:

```
// Long period >2x1018 random number generator of L'Ecuyer with
// Bays-Durham shuffle.  Return a uniform deviate in the interval (0,1).
// Call with int variable as argument set to a negative value, and
// don't this variable variable unless you want to reseed the generator

double ran2 (int& idum) {
    const int IM1 = 2147483563, IM2 = 2147483399;
    const double AM=(1.0/IM1);
    const int IMM1 = IM1-1;
    const int IA1 = 40014, IA2 = 40692, IQ1 = 53668, IQ2 = 52774;
    const int IR1 = 12211, IR2 = 3791, NTAB = 32;
    const int NDIV = 1+IMM1/NTAB;
    const double EPS = 3.0e-16, RNMX = 1.0-EPS;

    int j, k;
    static int idum2=123456789, iy = 0;
    static int iv[NTAB];
    double temp;

    if (idum <= 0) {
        idum = (idum == 0 ? 1 : -idum);
        idum2=idum;
        for (j=NTAB+7;j>=0;j--) {
            k=idum/IQ1;
            idum=IA1*(idum-k*IQ1)-k*IR1;
            if (idum < 0) idum += IM1;
        }
    }
    iy = iy + 1;
    if (iy >= NTAB) iy = 0;
    temp = idum2 + iy*IM2;
    idum2 = idum2 - iy*IM1;
    return temp*AM;
}
```

```

        if (j < NTAB) iv[j] = idum;
    }
    iy=iv[0];
}
k=idum/IQ1;
idum=IA1*(idum-k*IQ1)-k*IR1;
if (idum < 0) idum += IM1;
k=idum2/IQ2;
idum2=IA2*(idum2-k*IQ2)-k*IR2;
if (idum2 < 0) idum2 += IM2;
j=iy/NDIV;
iy=iv[j]-idum2;
iv[j] = idum;
if (iy < 1) iy += IMM1;
if ((temp=AM*iy) > RNMX) return RNMX;
else return temp;
}

```

Gaussian deviates using the Box-Muller algorithm

Gaussian deviates are random numbers on the interval $(-\infty, \infty)$ distributed according to the probability

$$P(x)dx = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx .$$

This is a Gaussian with mean $\langle x \rangle = 0$ and unit variance $\langle (x - \langle x \rangle)^2 \rangle = 1$.

The *Box-Muller* algorithm is based on a product of two such Gaussians

$$P(x)dx P(y)dy = \frac{1}{2\pi} e^{-(x^2+y^2)/2} dx dy = r e^{-r^2/2} dr \frac{d\theta}{2\pi} ,$$

where

$$x = r \cos \theta, \quad y = r \sin \theta.$$

Make a change of variable

$$u = \frac{1}{2}r^2, \quad re^{-r^2/2}dr = e^{-u}du.$$

Thus we have a product of an exponential distribution in u and a uniform distribution in θ .

The following code from *Numerical Recipes* implements this algorithm: as follows:

- Generate a point (v_1, v_2) chosen at random inside a circle of unit radius with the origin excluded. This is done in a loop:
 - Generate a point uniformly and randomly inside a square $-1 < v_1 < 1, -1 < v_2 < 1$. This is done by stretching and shifting a uniform deviate with the algorithm `2.0*ran2(idum)-1.0`.
 - Repeat the above step until $0 < v^2 < 1$.

Such points are obviously distributed uniformly in the angle $\theta = \tan^{-1}(v_2/v_1)$.

- These points are distributed with probability

$$\frac{2\pi v \, dv}{\pi \times 1^2} = 2v \, dv = dv^2$$

i.e., uniformly in $0 < v^2 < 1$. Then $u = -\log v^2$ will be distributed exponentially in $(0, \infty)$. The desired Gaussian points are recovered by scaling

$$(x, y) = r \times \left(\frac{v_1}{\sqrt{v^2}}, \frac{v_2}{\sqrt{v^2}} \right) = \sqrt{\frac{2u}{v^2}} \times (v_1, v_2) = \sqrt{\frac{-2 \log v^2}{v^2}} \times (v_1, v_2).$$

```
#include <cmath>
```

```
// Returns a normally distributed deviate with zero mean and unit variance
```

```
double gasdev (int& idum) {  
    static int iset = 0;  
    static double gset;  
    double fac, rsq, v1, v2;  
    if (idum < 0) iset = 0;  
    if (iset == 0) {  
        do {  
            v1 = 2.0*ran2(idum)-1.0;  
            v2 = 2.0*ran2(idum)-1.0;  
            rsq = v1*v1 + v2*v2;  
        } while (rsq >= 1.0 || rsq == 0.0);  
        fac = std::sqrt(-2.0*std::log(rsq)/rsq);  
        gset = v1*fac;  
        iset = 1;  
        return v2*fac;  
    } else {  
        iset = 0;  
        return gset;  
    }  
}  
  
#endif /* RNG_H_DEFINED */
```

Note that each call to `gasdev` generates two independent gaussian deviates: one of these is returned to the caller, and the other is saved for the next call to the function.

Monte Carlo integration

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

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

where $x_n, n = 1 \dots N$ 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:

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

where $x_{m,n}, n = 1 \dots N$ is the m -th sequence of random numbers. Just like is done in a real experiment, the error in a measurment 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

$$\begin{aligned}\sigma_M^2 &= \frac{(b-a)^2}{MN^2} \sum_{m=1}^M \left(\sum_{n=1}^N \delta f_{m,n} \right)^2 \\ &= \frac{(b-a)^2}{MN^2} \sum_{m=1}^M \left(\sum_{n=1}^N \delta f_{m,n} \right) \left(\sum_{n'=1}^N \delta f_{m,n'} \right)\end{aligned}$$

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:

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

where

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

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 π 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.

```
// Monte Carlo integration in 1 dimension
```

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

```
#include "rng.h"
```

```
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;
    cout << "Enter negative integer seed to initialize ran2: ";
    int seed;
    cin >> seed;
    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)*ran2(seed);
            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;
I_M /= M;
sigma_M /= M;
sigma_M -= I_M*I_M;
sigma_M = sqrt(sigma_M);
sigmaAverage /= M;
cout << " Average " << setw(15) << I_M << setw(15) << sigmaAverage
     << setw(15) << I_M-pi << '\n'
     << " Standard Deviation      " << sigma_M << '\n'
     << " (Std. Dev.)/sqrt(M)      " << sigma_M/sqrt(double(M)) << "\n ";
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
 Enter negative integer seed to initialize ran2: -78903

Trial	Integral	MC Error	Actual Error

1	3.14295	0.00203401	0.00136188
2	3.14646	0.0020283	0.00487098
3	3.14371	0.00202878	0.00212204
4	3.13896	0.00204009	-0.00263496
5	3.14179	0.00203735	0.000193409
6	3.14032	0.00203623	-0.00126935
7	3.13847	0.0020362	-0.00312668
8	3.14382	0.00202985	0.00222704
9	3.13912	0.00203721	-0.00247249
10	3.13939	0.00203346	-0.00220335

Average	3.1415	0.00203415	-9.31473e-05
Standard Deviation		0.0025356	
(Std. Dev.)/sqrt(M)		0.000801829	
