



Introduction to Monte Carlo Methods

Professor Richard J. Sadus

Centre for Molecular Simulation

Swinburne University of Technology

PO Box 218, Hawthorn Victoria 3122, Australia

Email: RSadus@swin.edu.au



Overview

This module comprises material for two lectures. The aim is to introduce the fundamental concepts behind Monte Carlo methods.

The specific learning objectives are:

- (a) To become familiar with the general scope of Monte Carlo methods;
- (b) To understand the key components of the Monte Carlo method;
- (c) To understand the role of “randomness” in Monte Carlo;
- (d) To understand the importance of careful choice of random number generators;
- (e) To be able to write simple Monte Carlo programs.



What is Meant by Monte Carlo Method?

- The term “Monte Carlo method” is used to embrace a wide range of problem solving techniques which use random numbers and the statistics of probability.
- Not surprisingly, the term was coined after the casino in the principality of Monte Carlo. Every game in a casino is a game of chance relying on random events such as ball falling into a particular slot on a roulette wheel, being dealt useful cards from a randomly shuffled deck, or the dice falling the right way!
- In principle, any method that uses random numbers to examine some problem is a Monte Carlo method.



Examples of MC in Science

- Classical Monte Carlo (CMC) – draw samples from a probability distribution to determine such things as energy minimum structures.
- Quantum Monte Carlo (QMC) – random walks are used to determine such things as quantum-mechanical energies.
- Path-Integral Monte Carlo (PMC) – quantum statistical mechanical integrals are evaluated to obtain thermodynamic properties.
- Simulation Monte Carlo (SMC) – algorithms that evolve configurations depending on acceptance rules.

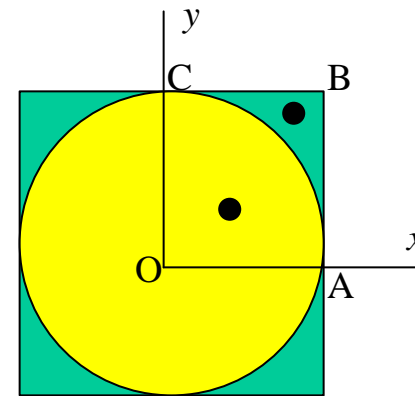


A Classic Example – The Calculation of π

- The value of π can be obtained by finding the area of a circle of unit radius. The diagram opposite represents such a circle centred at the origin (O) inside of a unit square.
- Trial “shots” (●) are generated in the OABC square. At each trial, two independent random numbers are generated on a uniform distribution between 0 and 1. These random numbers represent the x and y coordinates of the shot. If N is the number of shots fired and H is the number of hits (within the

circle):

$$p \approx \frac{4 \times \text{Area under CA}}{\text{Area of OABC}} = \frac{4H}{N}$$





The Calculation of π (contd)

A simple C function to calculate π

```
double calculate(int nTrials)
{
    int hit = 0;
    double x, y distanceFromOrigin;
    for (int i = 1; i <= nTrials; i++)
    {
        x = myRand();
        y = myRand();
        distanceFromOrigin = sqrt(x*x + y*y);
        if (distanceFromOrigin <= 1.0) hit++;
    }
    return 4*(double) hit/nTrials;
}
```

Random number between 0 and 1

Pythagoras theorem

Inside the quadrant



Another Example – Radioactive Decay

The kinetics of radioactive decay are first-order, which means that the decay constant (λ) is independent of the amount of material (N). In general:

$$\frac{\Delta N(t)}{N(t)\Delta t} = -\lambda$$

The decay of radioactive material, although constant with time is purely a random event. As such it can be simulated using random numbers.



Radioactive Decay (contd)

C function to simulate radioactive decay

```
void RadioactiveDecay(int maxTime, int maxAtom, double lambda)
{
    int numLeft = maxAtom, time, counter;
    counter = maxAtom;
    for (time = 0; time <= maxTime; time++)
    {
        for (atom = 1; atom <= numLeft; atom++)
            if (myRandom() < lambda) counter--;
        numLeft = counter;
        printf("(%d\t%f\n", time, (double) numLeft/maxAtom);
    }
}
```

Time loop

Atom loop

Atom decays



Monte Carlo Integration

The calculation of π is actually an example of “hit and miss” integration. It relies on its success of generating random numbers on an appropriate interval. Sample mean integration is a more general and accurate alternative to simple hit and miss integration. The general integral

$$I = \int_{x_l}^{x_u} dx f(x)$$

is rewritten:

$$I = \int_{x_l}^{x_u} dx \left(\frac{f(x)}{\mathbf{r}(x)} \right) \mathbf{r}(x)$$

where $\rho(x)$ is an arbitrary probability density function. If a number of trials n are performed by choosing a random number R_t from the distribution $\rho(x)$ in the range (x_l, x_u) then



Monte Carlo Integration (contd)

$$I = \left\langle \frac{f(R_t)}{r(R_t)} \right\rangle_{trials}$$

Where the brackets denote the average over all trails (this is standard notation for simulated quantities). If we choose $\rho(x)$ to be uniform,

$$r(x) = \frac{1}{(x_u - x_l)} \quad x_l \leq x \leq x_u$$

The integral I can be obtained from:

$$I \approx \frac{(x_u - x_l)}{t_{\max}} \sum_{t=1}^{t_{\max}} f(R_t)$$



Monte Carlo Integration (contd)

The general MC procedure for integrating a function $f(x)$ is to randomly obtain values of x within the upper and lower bounds of the integrand, determine the value of the function, accumulate a sum of these values and finally obtain an average by dividing the sum by the number of trials.

In general, to calculate an integral of two independent values $f(x,y)$ over an area in the (x,y) -values:

- (a) Choose random points in the area and determine the value of the function at these points.
- (b) Determine the average value of the function.
- (c) Multiply the average value of the function by the area over which the integration was performed.

This procedure can be easily extended to multiple integrals



Simple Monte Carlo Integration (contd)

MC integration is not usually a computationally viable alternative to other methods of integration such as Simpson's rule. However, it can be of some value in solving multiple integrals. Consider, the following multiple integral

$$\int_{x_l}^{x_u} \int_{y_l}^{y_u} \int_{z_l}^{z_u} (x^2 + y^2 + z^2) dx dy dz = \frac{xyz(x^2 + y^2 + z^2)}{3} \Bigg|_{x_l}^{x_u} \Bigg|_{y_l}^{y_u} \Bigg|_{z_l}^{z_u}$$

This integral has an analytical solution (RHS), but many multiple integrals are not so easily evaluated in which case MC integration is a useful alternative.



Simple Monte Carlo Integration (contd)

The skeletal outline of simple C program to solve this integral is:

```
main()
{
  int i;
  double xL, xU, yU,yL, zU, zL x, y, z, xRange, yRange, zRange function, diff, sum = 0.0, accur = 0.001, oldVal = 0.0;
  printf("Enter xL ");
  scanf("%f", &xL);
  /* add similar statements for xU, yL, yU, zL and zU here */
  xRange = xU - xL;
  /* add similar statements for yRange, and zRange here */
  do
  {
    i = i + 1;
    x = myRand()*xRange + xL;
    /* add similar statements fo y and z here */
    function = x*x + y*y + z*z;
    sum = sum + function;
    integrand = (sum/i)*xRange*yRange*zRange;
    diff = fabs(integrand - oldVal);
    oldVal = integrand;
  } while (diff > accur);
  printf("The integrand is %f", integrand);
}
```

Returns random number between 0 - 1

Scale random number to suitable interval

Multiply by volume since it is a triple integral



Key Components of a Monte Carlo Algorithm

Having examined some simple Monte Carlo applications, it is timely to identify the key components that typically compose a Monte Carlo algorithm. They are:

- Random number generator
 - Sampling rule
 - Probability Distribution Functions (PDF)
 - Error estimation
-
- In the remainder of this module will briefly examine the first three of these features. Both sampling and PDF are discussed in greater detail in Module 2. Error estimation is handled subsequently in the context of ensemble averaging (Module 4).





Random Number Generators

- A reliable random number generator is critical for the success of a Monte Carlo program. This is particularly important for Monte Carlo simulations which typically involve the use of literally millions of random numbers. If the numbers used are poorly chosen, i.e. if they show non-random behaviour over a relatively short interval, the integrity of the Monte Carlo method is severely compromised.
- In real life, it is very easy to generate truly random numbers. The lottery machine does this every Saturday night! Indeed, something as simple as drawing numbers out of a hat is an excellent way of obtaining numbers that are truly random.
- In contrast, it is impossible to conceive of an algorithm that results in purely random numbers because by definition an algorithm, and the computer that executes it, is deterministic. That is it is based on well-defined, reproducible concepts.



Pseudo Random Number Generators

- The closest that can be obtained to a random number generator is a pseudo random number generator.
- Most pseudo random number generators typically have two things in common: (a) the use of very large prime numbers and (b) the use of modulo arithmetic.
- Most language compilers typically supply a so-called random number generator. Treat this with the utmost suspicion because they typically generate a random number via a recurrence relationship such as:

$$R_{j+1} = aR_j + c \pmod{m}$$



Pseudo Random Number Generators (contd)

The above relationship will generate a sequence of random numbers R_1, R_2 etc between 0 and $m - 1$. The a and c terms are positive constants.

This is an example of a *linear congruential generator*. The advantage of such a random number generator is that it is very fast because it only requires a few operations. However, the major disadvantage is that the recurrence relationship will repeat itself with a period that is no greater than m .

If a , c and m are chosen properly, then the repetition period can be stretched to its maximum length of m . However, m is typically not very large. For example, in many C applications m (called `RAND_MAX`) is only 32767.



Pseudo Random Number Generators (contd)

A number like 32767 seems like a large number but a moment thought will quickly dispel this illusion. For example, a typical Monte Carlo integration involves evaluating one million different points but if the random numbers used to obtain these points repeat every 32767, the result is that the same 32767 points are evaluated 30 times!

Another disadvantage is that the successive random numbers obtained from congruential generators are highly correlated with the previous random number.

The generation of random numbers is a specialist topic. However, a practical solution is to use more than one congruential generators and “shuffle” the results.



Pseudo Random Number Generators (contd)

The following C++ function (from Press et al.) illustrates the process of shuffling. I use this code in all my Monte Carlo programs and I have found it to be very reliable.

```
double myRandom(int *idum)
{
    /* constants for random number generator */

    const long int M1 = 259200;
    const int IA1 = 7141;
    const long int IC1 = 54773;
    const long int M2 = 134456;
    const int IA2 = 8121;
    const int IC2 = 28411;
    const long int M3 = 243000;
    const int IA3 = 4561;
    const long int IC3 = 51349;

    int j;
    static int iff = 0;
    static long int ix1, ix2, ix3;
    double RM1, RM2, temp;
    static double r[97];
```



Pseudo Random Number Generators (contd)

```
RM1 = 1.0/M1;
RM2 = 1.0/M2;

if(*idum < 0 || iff == 0)      /*initialise on first call */
{
    iff = 1;
    ix1 = (IC1 - (*idum)) % M1;  /* seeding routines */
    ix1 = (IA1 *ix1 + IC1) % M1;
    ix2 = ix1 % M2;
    ix1 = (IA1 * ix1 + IC1) % M1;
    ix3 = ix1 % M3;

    for (j = 0; j < 97; j++)
    {
        ix1 = (IA1 * ix1 + IC1) % M1;
        ix2 = (IA2 * ix2 + IC2) % M2;
        r[j] = (ix1 + ix2 * RM2) * RM1;
    }
    *idum = 1;
}
```



Pseudo Random Number Generators (contd)

```
/*generate next number for each sequence */  
  
ix1 = (IA1 * ix1 + IC1) % M1;  
ix2 = (IA2 * ix2 + IC2) % M2;  
ix3 = (IA3 * ix3 + IC3) % M3;  
  
/* randomly sample r vector */  
  
j = 0 + ((96 * ix3) / M3);  
if (j > 96 || j < 0)  
    cout <<"Error in random number generator\n";  
  
temp = r[j];  
  
/* replace r[j] with next value in the sequence */  
  
r[j] = (ix1 + ix2 * RM2) * RM1;  
  
return temp;  
}
```



Sampling

- Monte Carlo works by using random numbers to sample the “solution space” of the problem to be solved. In our examples of the calculation of π , simulation of radioactive decay and Monte Carlo integration we employed a “simple” sampling technique in which all points were accepted with equal probability.
- Such simple sampling is inefficient because we must obtain many points to obtain an accurate solution. Indeed the accuracy of our solution is directly proportional to the number of points sampled.
- However, not all points in the solution-space contribute equally to the solution. Some are of major importance, whereas others could be safely ignored without adversely affecting the accuracy of our calculation.



Sampling (contd)

- In view of this, rather than sampling the entire region randomly, it would be computationally advantageous to sample those regions which make the largest contribution to the solution while avoiding low-contributing regions. This type of sampling is referred to as “importance sampling.”

To illustrate the difference between simple sampling and importance sampling, consider the difference between estimating an integral via simple sampling:

$$I_{est} = \frac{1}{N} \sum_i^N f(x_i)$$

and using importance sampling

$$I_{est} = \sum_i^N f(x_i) / p(x_i)$$



Sampling (contd)

In the later case, each point is chosen in accordance with the anticipated importance of the value to the function and the contribution it makes weighted by the inverse of the probability (p) of choice. Unlike the simple sampling scheme, the estimate is no longer a simple average of all points sampled but it is a weighted average.

- It should be noted that this type of sampling introduces a bias which must be eliminated. Importance sampling is discussed in greater detail in Module 2.



Probability Distribution Functions

- As illustrated above, Monte Carlo methods work by sampling according to a probability distribution function (PDF).
- In the previous examples, using simple sampling, we have in effect been sampling from a “uniform distribution function” in which every selection is made with equal probability.
- In statistics, the most important PDF is probably the Gaussian or Normal Distribution:

$$f(x) = \frac{e^{-(x-m)^2 / 2s^2}}{\sqrt{2\pi s}}$$



Probability Distribution Functions (contd)

where μ is the mean of the distribution and σ^2 is the variance.

This function has the well-known “bell-shape” and gives a good representation of such things as the distribution of the intelligence quotient (IQ) in the community centred on a “normal” value of IQ = 100 (hopefully we are a biased sample with IQs to the right of this value!).

- In scientific applications, a well-known distribution function is the Boltzmann distribution, which relates the fraction of particles N_i with energy E_i to the temperature (T) and Boltzmann’s constant (k):



Probability Distribution Functions (contd)

$$N_i / N = e^{-E_i / kT}$$

Indeed, this distributed will play a central role in many of the Monte Carlo algorithms presented in subsequent modules.



Problems

1. Consider the function $f(x) = x^{10}-1$. Use a simple sampling MC technique to integrate this function between $x = a$ and $x = 2$.
2. Implement the MC program to calculate π using the standard `rand()` function in place of `myRand()`. Compare the accuracy of your results after 30,000, 100000 and 1000000 samples. Does the accuracy improve? Hint: depending on the compiler, you may have to convert the numbers to lie on the 0 – 1 range.
3. Repeat problem (2) using `myRand()` and compare the results.
4. Generalise the method used to solve the three-dimensional integral to solve the following 10-D integral:



Problems (continued)

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \dots \int_0^1 dx_{10} (x_1 + x_2 + \dots + x_{10})^2$$

5. Write your own simple function to generate random numbers using the linear congruent method. Test your generator with the values $a = 111$, $c = 2$, $M = 256$ and $R_1 = 1$ and determine the repetition period.
6. A simple random number generator can be obtained using the following relationship:

$$R_n = R_{n-p} R_{n-q}$$

Where $p = 17$, $q = 5$ are “off-set” values. Write a function to compute random numbers in this way and compare the output with the random numbers generated your compiler’s random number generator.



Problems (continued)

7. Sometimes, it is useful to generate random numbers not uniformly but according to some pre-determined distribution. Numbers on the Guassian distribution can be generated via Box-Muller method. This involves generating two numbers (x_1 and x_2) on a uniform distribution and transforming them using:

$$y_1 = \sqrt{-2\ln x_1} \cos(2\pi x_2)$$

$$y_2 = \sqrt{-2\ln x_1} \sin(2\pi x_2)$$

Write a function that calculates numbers on the Guassian distribution.



Reading Material

The material covered in this module is discussed in greater detail in the following books:

M.P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, OUP, Oxford, 1987, pages 110-112.

D. P. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, CUP, 2000, pages 1-4 and 48-53.

D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications*, Academic Press, San Diego, 1996, pages 19-28.

R.J. Sadus, *Molecular Simulation of Fluids: Theory, Algorithm and Object-Oriented*, Elsevier, Amsterdam, 1999.

W.H. Press, B.F. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, CUP, Cambridge, 1988, pages 204-213.

R.H. Landau and M.J. Páez, *Computational Physics: Problem Solving with Computers*, Wiley, New York, 1997, pages 93-108.