Monte Carlo Methods

Aksel Hiorth, the National IOR Centre & Institute for Energy Resources,

University of Stavanger

Feb 7, 2019

Contents

1 Monte Carlo Methods

Monte Carlo methods are named after the Monte carlo Casino in Monaco, this is because at its core it uses random numbers to solve problems. Monte Carlo methods are usually quite easy to program, and they are usually much more intuitive than a theoretical approach. If we would like to find the probability to get at least 5 on three dices after 5 throws there are methods from statistics that could tell us the probability. Using the Monte Carlo method, we would get the computer to pick a random integer between 1 and 6, three times, to represent one throw of the dices bla bla. later in this chapter Usually Usually we use differential equations to describe physical systems, the solution to these equations are continuous functions. In order for these solutions to be useful, they require that the differential equation describes our physical sufficiently. In many practical cases we have no control over many of the parameters entering the differential equation, or stated differently *our system is not deterministic*. This means that there could be some random fluctuations, occurring at different times and points in space, that we have no control over. In a practical situation we might would like to investigate how these fluctuations would affect the behavior of our system. A

2 Monte Carlo Integration

Let us start with a simple illustration of the Monte Carlo Method (MCM), Monte Carlo integration. To the left in figure 1 there is a shape of a pond. Imagine that we wanted to estimate the area of the pond, how could we do it? Assume further that you did not have your phone or any other electronic devices to help you.

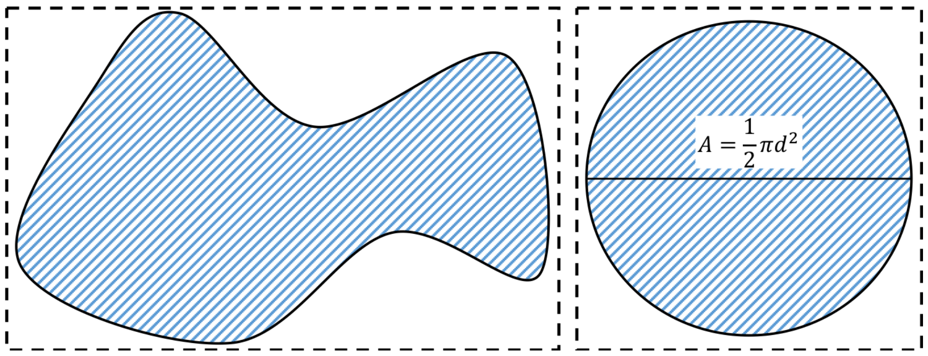

Figure 1: Two ponds to illustrate the MCM.

One possible approach is: First to walk around it, and put up some bands (illustrated by the black dotted line). Then estimate the area inside the bands (e.g. 4×3 meters). Then we would know that the area was less than e.g. 12m^2. Finally, and this is the difficult part, throw rocks *randomly* inside the bands. The number of rocks hitting the pond divided by the total number rocks thrown should be equal to the area of the pond divided by the total area inside the bands, i.e. the area of the pond should be equal to:

$$A \simeq \text{Area of rectangle} \times \frac{\text{Number of rocks hitting the pond}}{\text{Number of rocks thrown}}. \tag{1}$$

It is important that we throw the rocks randomly, otherwise equation (1) is not correct. Now, let us investigate this in more detail, and use the idea of throwing rocks to estimate π. To the right in figure 1, there is a well known shape, a circle. The area of the circle is $\pi d^2/4$, and the shape is given by $x^2 + y^2 = d^2/4$. Assume that the circle is inscribed in a square with sides of d. To throw rocks randomly inside the square, is equivalent pick random numbers with coordinates (x, y), where $x \in [0, d]$ and $y \in [0, d]$. We want all the $x-$ and $y-$values to be chosen with equal probability, which is equivalent to pick random numbers from a *uniform* distribution. Below is a Python implementation:

```
import numpy as np
import random

def estimate_pi(N,d):
#   random.seed(2)
    D2=d*d/4; dc=0.5*d
    A=0
    for k in range(0,N):
        x=random.uniform(0,d)
        y=random.uniform(0,d)
        if((x-dc)**2+(y-dc)**2 <= D2):
            A+=1
    # estimate area of circle: d*d*A/N
    return 4*A/N

N=1000;d=2
pi_est=estimate_pi(N,d)
print('Estimate for pi= ', pi_est,' Error=', np.pi-pi_est)
```

In the table below, we have run the code for $d = 1$ and different values of N.

MC estimate	Error	N	$1/\sqrt{N}$
3.04	-0.10159	10^2	0.100
3.176	0.03441	10^3	0.032
3.1584	0.01681	10^4	0.010
3.14072	-0.00087	10^5	0.003

We clearly see that a fair amount of rocks or numbers needs to be used in order to get a good estimate. If you run this code several times you will see that the results changes from time to time. This makes sense as the coordinates x and y are chosen at random.

2.1 Random number generators

There are much to be said about random number generators. The MCM depends on a good random number generator, otherwise we cannot use the results from statistics to develop our algorithms. Below, we briefly summarize some important points that you should be aware of:

1. Random number generators are generally of two types: *hardware random number generator* (HRNG) or *pseudo random number generator* (PRNG).

2. HRNG uses a physical process to generate random numbers, this could atmospheric noise, radioactive decay, microscopic fluctuations, which is translated to an electrical signal. The electrical signal is converted to a digital number (1 or 0), by sampling the random signal random numbers can be generated. The HRNG are often named *true random number generators*, and their main use are in *cryptography*.

3. PRNG uses a mathematical algorithm to generate an (apparent) random sequence. The algorithm uses an initial number, or a *seed*, to start the sequence of random number. The sequence is deterministic, and it will

generate the same sequence of numbers if the same seed is used. At some point the algorithm will reproduce itself, i.e. it will have certain period. For some seeds the period may be much shorter.

4. Many of the PRNG are not considered to be cryptographically secure, because if a sufficiently long sequence of random numbers are generated from them, the rest of the sequence can be predicted.

5. Python uses the Mersenne Twister[1] algorithm to generate random numbers, and has a period of $2^{19937} - 1 \simeq 4.3 \cdot 10^{6001}$. It is not considered to be cryptographically secure.

In Pythons `random.uniform` function, a random seed is chosen each time the code is run, but if we set e.g. `random.seed(2)`, the code will generate the same sequence of numbers each time it is called.

2.2 Encryption

This section can be skipped as it is not relevant for development of the numerical algorithms, but it is a good place to explain the basic idea behind encryption of messages. A very simple, but not a very good encryption, is to replace all the letters in the alphabet with a number, e.g. A=1, B=2, C=3, etc. This is what is know as a *substitution cipher*, it does not need to be a number it could be a letter, a sequence of letters, letters and numbers etc. The receiver can solve the code by doing the reverse operation.

The weakness of this approach is that it can fairly easily be cracked, by the following approach: First we analyze the encrypted message and find the frequency of each of the symbols. Assume that we know that the message is written in English, then the frequency of symbols can be compared with the frequency of letters from a known English text (the most common is `E` (12%), then `T` (9%), etc.). We would then guess that the most occurring symbol probably is an `E` or `T`. When some of the letters are in place, we can compare with the frequency of words, and so on. By the help of computers this process can easily be automated.

A much better algorithm is *to not replace a letter with the same symbol.* To make it more clear, consider our simple example where A=1, B=2, C=3, ... If we know say that A=1 but we add a *random number*, then our code would be much harder to crack. Then the letter A could be several places in the message but represented as a complete different number. Thus we could not use the frequency of the various symbols to crack the message.

How can the receiver decrypt the message? Obviously, it can be done if both the sender and receiver have the same sequence of random numbers (or the *key*). This can be achieved quite simple with random number generators, if we know the seed used we can generate the same sequence of random numbers. If Alice where to send a message to Bob without Eve knowing what it is, Alice

[1]`https://en.wikipedia.org/wiki/Mersenne_Twister`

and Bob could agree to send a message that was scrambled using Pythons Mersenne-Twister algorithm with seed=2.

The weakness of this approach is of course that Eve could convince Alice or Bob to give her the seed or the key. Another possibility is that Eve could write a program that tested different random number generators and seeds to decipher the message. How to avoid this?

Let us assume that Alice and Bob each had their own hardware random generator. This generator generated random numbers that was truly random, and the sequence could not be guessed by any outsider. Alice do not want to share her key (sequence of random numbers) with Bob, and Bob would not share his key with Alice. How can they send a message without sharing the key? One possible way of doing it is as follows: Alice write a message and encrypt it with her key, she send the message to Bob. Bob then encrypt the message with his key, he sends it back to Alice. Alice then decrypt the message with her key and send it back to Bob. Now, Bob can decrypt it with his own key and read the message. The whole process can be visualized by thinking of the message as box with the message. Alice but her padlock on the box (keeps her key for her self), she sends the message to Bob. Bob locks the box with his padlock, now there are two padlocks on the box. He sends the box back to Alice, Alice unlocks her padlock with her key, and sends it back to Bob. The box now only has Bob's key, he can unlock the box and read the message. The important point is that the box was never unlocked throughout the transaction, and Alice and Bob never had to share the key with anyone.

2.3 Errors on Monte Carlo Integration and the Binomial Distribution

How many rocks do we need to throw in order to reach a certain accuracy? To answer this question we need some results from statistics. Our problem of calculating the integral is closely related to the *binomial distribution.* When we throw a rock one of two things can happen i) the rock falls into the water, or ii) it falls outside the pond. If we denote the probability that the rock falls into the pond as p, then the probability that it falls outside the pond, q, has to be $q = 1 - p$. This is simply because there are no other possibilities and the sum of the two probabilities has to be one: $p + q = p + (1 - p) = 1$. The binomial distribution is given by:

$$p(k) = \frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}. \tag{2}$$

$p(k)$ is the probability that an event happens k times after n trials. The mean, μ, and the variance, σ^2, of the binomial distribution is:

$$\mu = \sum_{k=0}^{n-1} kp(k) = np, \tag{3}$$

$$\sigma^2 = \sum_{k=0}^{n-1} (k-\mu)^2 p(k) = np(1-p). \tag{4}$$

Mean and variance.

The mean of a distribution is simply the *sum* divided by the *count*, the symbol μ is usually used. For n observations, x_i, $\mu = \sum_i x_i/n$. The mean is just an average, it could e.g. be the sum of all the heights of students in the class divided by the number of students. The mean would then be the average height of all the students in the class.

The variance is calculated by taking the difference between each of the data points and the mean, square it, and sum over all data points. Usually the symbol σ^2 is used. $\sigma^2 = \sum_i(\mu - x_i)^2/n$. Defined in this way it is quite easy to understand that the variance tells us something about the spread in the data. Usually we use the *standard deviation*, $\sigma = \sqrt{\sigma^2}$, as an estimate of the uncertainty in our estimate of the mean, i.e. $\mu \pm \sigma$.

Before we proceed, we should take a moment and look a little more into the meaning of equation (2) to appreciate its usefulness. A classical example of the use of the binomial formula is to toss a coin, if the coin is fair it will have an equal probability of giving us a head or tail, hence $p = 1/2$. Equation (2), can answer questions like: ”What is the probability to get only heads after 4 tosses?”. Let us calculate this answer using equation (2), the number of tosses is 4, the number of success is 4 (only heads each time)

$$p(k=4) = \frac{4!}{4!(4-4)!}\frac{1}{2}^4(1-\frac{1}{2})^{4-4} = \frac{1}{2^4} = \frac{1}{16}. \tag{5}$$

”What is the probability to get three heads in four tosses?”, using the same equation, we find:

$$p(k=3) = \frac{4!}{3!(4-3)!}\frac{1}{2}^3(1-\frac{1}{2})^{4-3} = \frac{4}{2^4} = \frac{1}{4}. \tag{6}$$

In figure 2, all the possibilities are shown. The number of possibilities are 16, and there are only one possibility that we get only heads, i.e. the probability is 1/16 as calculated in equation (5). In the figure we also see that there are 4 possible ways we can get three heads, hence the probability is 4/16=1/4 as calculated in equation (6).

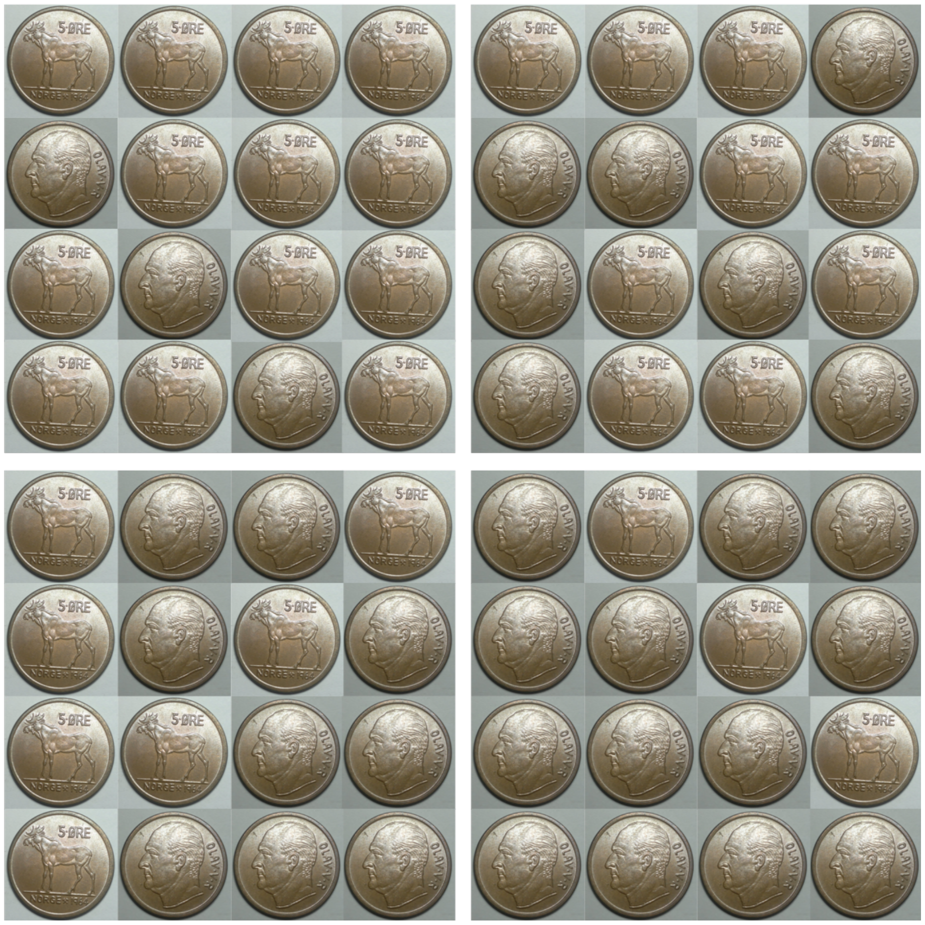

Figure 2: The famous Norwegian Moose coin, and possible outcomes of four coin flips in a row.

Now, let us return to our original question, "What is the error on our estimate of the integral, when using the MCM?". Before we continue we should also clean up our notation, let I be the value of the true integral, A is our *estimate* of the integral, and I_N is the area of the rectangle. First, let us show that the mean or expectation value of the binomial distribution is related to our estimate of the area of the pond or the circle, A. In our case we draw $n = N$ random numbers, and k times the coordinate falls inside the circle, equation (3) tells us that the mean value is np. p is the probability that the coordinate is within the area to be integrated, hence as before p is equal to the area to be integrated divided by the area of the total domain, thus:

$$\mu = np = N\frac{A}{I_N}, \tag{7}$$

or

$$A = I_N \frac{\mu}{N}. \tag{8}$$

Equation (4), gives us an estimate of the variance of the mean value. Assume for simplicity that we can replace $1 - p \simeq p$, this is of course only correct if the area of the rectangle is twice as big as our pond, but we are only interested in an estimate of the error, hence $\sigma^2 \simeq np^2$. We can now use the standard deviation as an estimate of the error of our integral:

$$\begin{aligned} I &\simeq I_N \frac{\mu \pm \sigma}{n} = I_N \frac{Np \pm \sqrt{N}p}{N} \\ &\simeq I_N(p \pm \frac{p}{\sqrt{N}}) = A \pm \frac{A}{\sqrt{N}}. \end{aligned} \tag{9}$$

In the last equation we have replaced p with A/I_N. Hence, the error of our integral is inversely proportional to the square root of the number of points.

2.4 The mean value method

How does our previous method compare with some of our standard methods, like the midpoint rule? The step length, h, is related to the number of points as $h = (b - a)/n$, where b and a are the integration limit. Thus our MCM scales as $1/\sqrt{n} \sim h^{1/2}$, this is actually worse than the midpoint or trapezoidal rule, which scaled as h.

The MCM can be improved, we will first describe the mean value method. In the last section we calculated the area of a circle by picking random numbers inside a square and estimated the fraction of points inside the circle. This is equivalent to calculate the area of a half circle, and multiply with 2:

$$I = 2\int_{-d/2}^{d/2} \sqrt{(d/2)^2 - x^2}dx = \frac{\pi d^2}{4}. \tag{10}$$

The half-circle is now centered at the origin. Before we proceed we write our integral in a general form as:

$$I = \int_a^b f(x)dx. \tag{11}$$

Instead of counting the number of points inside the curve given by $f(x)$, we could instead estimate the expectation value or the mean of the function. In order to transform the mean value to an integral we need to multiply with the width of the integration domain $b - a$, hence:

$$I = \int_a^b f(x)dx \simeq \frac{b-a}{n}\sum_{i=0}^{n-1} f(x_i). \tag{12}$$

Note that this formula is similar to the midpoint rule, but now the function is not evaluated at the midpoint, but at several points and we use the average value. Below is an implementation:

```
import numpy as np
import random

def f(x,D2):
    return 2*(D2-x*x)**0.5

def mcm_mean(N,d):
#   random.seed(2)
    D2=d*d/4
    A=0
    for k in range(0,N):
        x=random.uniform(-d/2,d/2)
        A+=f(x,D2)
    # estimate for area: A/N
    return d*A/N
```

In the table below we have compared the mean value method with the method of counting the fraction of points inside the circle. We see that the mean value method performs somewhat better, but there are some random fluctuations and in some cases it performs poorer.

MC-mean	Error	MC	Error	N
3.1706	0.0290	3.1200	-0.0216	10^2
3.1375	-0.0041	3.1560	0.0144	10^3
3.1499	0.0083	3.1224	-0.0192	10^4
3.1424	0.0008	3.1402	-0.0014	10^5
3.1414	-0.0002	3.1437	0.0021	10^6

We also see that the error scales as $1/\sqrt{N}$