Monte Carlo and The Metropolis Algorithm

1 Sampling Probability Distributions

In Physics, one often encounters the following situation: given a probability distribution $P(x)$ over a random variable x (where x is usually an observable), one wishes to compute the mean value of a physical quantity f, which is a function of the variable x. If x takes values $x_1, x_2, ..x_N$, the average is given by

$$\bar{f} = \sum_i f(x_i) P(x_i) \tag{1}$$

Often, this average cannot be computed analytically, and needs to be computed numerically. Then, one can in principle generate this distribution on the computer and calculate the average. One way to go about it is to generate the various values x_i with the corresponding probabilities $P(x_i)$. Then, if x_i has been generated M_i times, the average of f is given by

$$\bar{f} = \frac{\sum_i M_i \times f(x_i)}{M}$$

where $M = \sum_i M_i$. This can also be directly wriiten as

$$\bar{f} = \frac{1}{M} \sum_r f_r \tag{2}$$

where here r refers to a 'trial' in which the variable x is generated with probabiity $P(x)$, f_r is the value of f for that trial and M is the total number of trials. In this expression, a given value f_r will usually occur multiple times, in proportion to the probability $P(x_r)$ of x_r occuring. We can use a random number generator, which can generate the values of x with the given probability to achieve this. This technique is called 'direct sampling', since it directly samples the probability distribution. For simple probability distributions, this method works well. However, in more complicated cases, this is impracticle. For instance, in Statistical Mechanics, given a system in thermal equilibrium at temperature T, the probability that it is in a microscopic state (microstate) x is given by the Canonical distribution

$$P(x) \propto e^{-\beta E(x)} \tag{3}$$

where $E(x)$ is the energy of the system in microstste x and $\beta = 1/(k_B T)$, k_B being the Boltzmann constant. At first look, eqn.(3) seems to be a very simple probability distribution. However, the problem is in enumerating the microstates x. In a Classical description, the microstate x is specified by specifying the position coordinates and momentum components of typically N_A particles, where N_A is the Avogadro number $\sim 10^{23}$. This results in one needing to specify $6N_A$ numbers to specify a single microstate of the system! Given this and the number of microstates accessible to a macroscopic system (which is usually a number so large that it cannot be compared with the largest numbers one encounters in Physics), it is clearly impossible to directly sample any substantial set of microstates with the probability eqn.(3). It is here that direct sampling fails.

2 Monte Carlo Sampling

Given that it is often impossible to even enumerate the possible values of the random variable x (which, as the example above shows, could itself be a very large set of other variables), the following alternative sampling is often employed. It relies on sampling the *relevant* values of x, the values which appreciably contribute to the mean value in eqn.(2). It starts with identifying a relevant value of x, and computing its contribution to eqn.(2). Next, one takes a random walk in the space of possible values of x, a walk which is generated such that after a 'large enough' number of steps of the walk, different values of x have been 'stepped-on' by the virual walker a number of times proportional to the probability which which they occur in the distribution $P(x)$. Such a probabilistic random walk is an example of a *Markov Chain*. To visualise this, it is convenient to imagine the space of values of x being divided into cells with any given cell having a certain number of nearest neighbors (these neighboring cells storing values of x 'closest' to the value in the cell in question). The walker hops from one cell to a nearest neighbor with a certain probability, such that any one cell is connected to its neighbors by a set of probabilities with which it steps on them. This set of probabilities forms the algorithm which eventually covers the cells with the overall probability $P(x)$

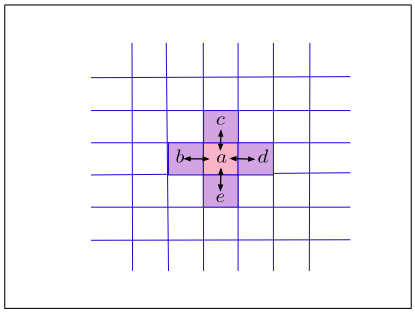

Figure 1: Cell a with four nearest neighbors

Say, there are N cells, the walker starts in cell 1. Then, with probability $p(1 \longrightarrow i)$, he steps on the i^{th} cell. Let us write the various cell probabilities as a column vector. Then, at the beginning, the probability vector will have to form

$$p_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ .. \\ 0 \end{pmatrix}$$

This is because since the walker is in the first cell, he is there with probability unity. After the first step, the probability vector becomes

$$p_1 = \begin{pmatrix} p(1 \longrightarrow 1) \\ p(1 \longrightarrow 2) \\ p(1 \longrightarrow 3) \\ .. \\ p(1 \longrightarrow N) \end{pmatrix}$$

Similarly, if he were to start in cell 2, the probabilities would be $p(2 \longrightarrow 1), p(2 \longrightarrow 2), .., p(2 \longrightarrow N)$ and so on. We can arrange these probablilities in the form of a matrix

$$P = \begin{pmatrix} p(1 \longrightarrow 1) & p(2 \longrightarrow 1) & .. & p(N \longrightarrow 1) \\ p(1 \longrightarrow 2) & p(2 \longrightarrow 2) & .. & p(N \longrightarrow 2) \\ .. & .. & .. & .. \\ .. & .. & .. & .. \\ p(1 \longrightarrow N) & p(2 \longrightarrow N) & .. & p(N \longrightarrow N) \end{pmatrix} \tag{4}$$

Then, obtaining p_1 from p_0 reduces to the matrix operation

$$p_1 = P\ p_0$$

The successive steps simply lead to successive matrix operations by the probability matrix P. Then, given any initial state p_0, after m steps, the state is

$$p_m = (P)^m \; p_0$$

Alternatively, given the state after n steps, the state after $n+1$ steps is

$$p_{n+1} = P \; p_n \tag{5}$$

It should be noted that the matrix P is independent of the state of probabilities at any given step. Further, since the total probability of going from a given cell to some cell (which could be itself or a different cell) must be equal to unity, it follows that the sum of elements of any column must be equal to 1

$$\sum_{j=1}^{N} p(i \longrightarrow j) = 1 \; \forall i \tag{6}$$

Such a matrix of probabilities is called a *Markov* matrix. Given the probability distribution $P(x)$ that we wish to generate through this Markovian walk, the matrix P should be such that there should exist a steady probability vector

$$p_s = \begin{pmatrix} P(x_1) \\ P(x_2) \\ P(x_3) \\ .. \\ P(x_N) \end{pmatrix} \tag{7}$$

which should satisfy

$$P \; p_s = p_s \tag{8}$$

That is, eventually, the process should generate a probability vector whose entries are the probabilities we wish to sample, and this vector should be such that there is no furthur change in its entries under application of the probability matrix P. Mathematically, eqn.(7) implies that this steady vector should be an eigenvector of P with eigenvalue 1. Therefore, the problem reduces to constructing the probability matrix P which encodes the rules of the Markovian walk and which is such that the probability distribution we wish to generate is its eigenvector with unit eigenvalue. This is a challenging mathematical task, and we do not know if such a matrix P is even unique. So, we resort to constructing the simplest possible matrix P which guarantees that the probability vector (7) is unchanged under its application. Let us refer to figure 1. which shows a cell with four nearest neighbors. We will carry out the analysis for this special case, but it is easy to convince oneself that the result is not sensitive to the number of neighboring cells. Say, we wish to know after n steps of the Markov walk the probability of stepping onto cell marked a at the $n+1$ th step. Let the probability values in the cells a, b, c, d, e at the end of n steps be $P(a), P(b), P(c), P(d)$ and $P(e)$. Since we can step into cell a only from a, b, c, d or e, the probability $P'(a)$ that we step onto cell a at the $n+1$ th step is given by

$$P'(a) = P(a) \times p(a \longrightarrow a) + P(b) \times p(b \longrightarrow a) + P(c) \times p(c \longrightarrow a) + P(d) \times p(d \longrightarrow a) + P(e) \times p(e \longrightarrow a)$$

However, if we have already attained the steady probability vector, $P'(a) = P(a)$. Substituting this and rearranging terms, we get

$$P(a)\,[1 - p(a \longrightarrow a)] = P(b) \times p(b \longrightarrow a) + P(c) \times p(c \longrightarrow a) + P(d) \times p(d \longrightarrow a) + P(e) \times p(e \longrightarrow a) \tag{9}$$

Conservation of probability gives

$$p(a \longrightarrow a) + p(a \longrightarrow b) + p(a \longrightarrow c) + p(a \longrightarrow d) + p(a \longrightarrow e) = 1$$

such that

$$1 - p(a \longrightarrow a) = p(a \longrightarrow b) + p(a \longrightarrow c) + p(a \longrightarrow d) + p(a \longrightarrow e)$$

substituting this in eqn.(9), we get

$$\begin{aligned} P(a)p(a \longrightarrow b) + P(a)p(a \longrightarrow c) + P(a)p(a \longrightarrow d) + P(a)p(a \longrightarrow e) &= P(b)p(b \longrightarrow a) \\ &+ P(c)p(c \longrightarrow a) \\ &+ P(d)p(d \longrightarrow a) \\ &+ P(e)p(a \longrightarrow a) \end{aligned} \tag{10}$$

This is the condition which the probability matrix elements should satisfy such that its action on the equilibrium probability vector consisting of entries $P(a), P(b), P(c)$..... (which together are just $P(x)$) leaves it unchanged. However, given the equilibrium distribution (which determined $P(a), P(b), ...$), this equation (and similar ones for the other cells) does not uniquely determine the probability matrix elements. However, we can easily construct *one* solution: the equation will be satisfied if given any two cells i and j with probabilities $P(i)$ and $P(j)$, the matrix elements $p(i \longrightarrow j)$ and $p(j \longrightarrow i)$ satisfy

$$P(i)p(i \longrightarrow j) = P(j)p(j \longrightarrow i) \tag{11}$$

which is the equation for *detailed balance*. This equation states something very simple: the probabilities stay steady because the flow of probability from one cell to another is equal to the flow in the other direction. This implies that given the probability distribution $P(x)$, the probability matrix which *dynamically* attains this probability distribution should be such that

$$\frac{p(x_i \longrightarrow x_j)}{p(x_j \longrightarrow x_i)} = \frac{P(x_j)}{P(x_i)} \tag{12}$$

Again, this equation does not uniquely determine $p(x_i \longrightarrow x_j)$. However, the following ansatz works

$$p(x_i \longrightarrow x_j) = min\left(1, \frac{P(x_j)}{P(x_i)}\right) \tag{13}$$

This ansatz is known as the *Metropolis Algorithm*. The way it works is as folows: say, during our random walk through the space of the possible values of the random variable x, we are in a 'cell' where the value is x_i. We take the next step by randomly choosing a neighboring cell, and 'proposing' a move to that cell. Say, the cell corresponds to value x_j. If the ratio $P(x_j)/P(x_i) > 1$, we accept the move and step onto that cell (this is because following eqn.(13), $p(x_i \longrightarrow x_j) = 1$). If however $P(x_j)/P(x_i) < 1$, then $p(x_i \longrightarrow x_j) = P(x_j)/P(x_i)$. To realize this probability, we generate a random number between 0 and 1. If the number is less than $P(x_j)/P(x_i)$, we accept the move and step into the new cell. Else, we stay in the original cell. During this random walk, we are evaluating the average of some observable f. As we step into cell x_r, the function picks the value $f(x_r)$. To step into the next cell, as discussed, we propose a move according to the Metropolis ansatz. If the move is accepted and we step into another cell with say $x = x_s$, the function picks up the value $f(x_s)$. However, if the move is rejected, we stay in cell x_r and *once again* the function picks the vlaue $f(x_r)$. Thus, on rejection of a move, we stay in the cell we occupy and for that (rejected) move, the mean value picks a contribution from that cell itself. If this is not done and say, we keep proposing moves till one is accepted, we will violate the condition of detailed balance.

3 Example: Gaussian Distribution

As a simple example, let us do a Markov sampling of a Gaussian probability distribution

$$P(x, y) = \left(\frac{1}{2\pi\sigma^2}\right) e^{-(x^2+y^2)/2\sigma^2} \tag{14}$$

and compute the average of the function $f(x, y) = x^2y^2$. This example is so simple that $\bar{f}$ is analytically computable

$$\begin{aligned} \bar{f} &= \int_{-\infty}^{\infty} dxdy\ P(x, y)\ x^2y^2 \\ &= \sigma^4 \end{aligned} \tag{15}$$

To compute the average using the Metropolis algorithm, we need to (a) choose a starting point for the walk (b) decide on a step size δ (this is equivalent to converting the space of values of x and y into 'cells'). The algorithm then proceeds as follows:

1. Start at point (x_0, y_0). Compute $f(x_0, y_0)$.
2. Generate two random numbers Δx and Δy between $-\delta$ and $+\delta$. Propose a move to the point $(x_0 + \Delta x, y_0 + \Delta y)$.
3. The probability of acceptance is given by

$$p(x_0, y_0 \longrightarrow x_0 + \Delta x, y_0 + \Delta y) = min\left(1, \frac{P(x_0 + \Delta x, y_0 + \Delta y)}{P(x_0, y_0)}\right)$$

   where $P(x, y)$ is the given probability distribution (Gaussian, here).
4. If the move is rejected, go to step 1.
5. If the move is accepted, label the new point (x_0, y_0) and go to step 1.

There is a clear ambiguity in this algorithm: the step size of the random walk. What is an ideal step size? If it is too small, in a given number of steps, we will not cover enough of the space of possible values of the random variable(s). This will be reflected in most of the proposed moves being accepted. On the other hand, if it is too large, though we will be able to cover much more of the space, most moves will be rejected. There is a rule of thumb which is often used to fix the step size: use a step size such that the acceptance ratio is between 0.3-0.5. A computer implementation of the Metropolis algorithm should compute this acceptance ratio and the step size should be tweaked to obtain this range. The following Python program implements this algorithm to sample the 2D Gaussian distribution, eqn.(14), and grphically shows the random walk

```
from pylab import *
import pylab
import random
import math

###### This program samples a 2D Gaussian distribution #################

### Setup of graphic visualisation. The distribution if visualised on a canvas, a square of side 2.

L = 2.

pylab.cla()
pylab.axis([-L/2, L/2, -L/2, L/2])
pylab.setp(pylab.gca(), xticks=[-L/2, L/2], yticks=[-L/2, L/2])


def Gauss(x,y,sigma): #### 2D Gaussian distribution
    return (1./(2*math.pi*sigma**2))*math.exp(-(x**2 + y**2 )/(2.*sigma**2))


x, y = 0., 0. ## Starting point
sigma = 0.25 ## Distribution width
delta = 0.5  ## Monte Carlo step size
n_trials = 5000 ## Number of steps
n_accept = 0 ## Number of moves accepted
positions = [(x,y)] ## List of sampled points
for i in range(n_trials):
    del_x, del_y = random.uniform(-delta, delta), random.uniform(-delta, delta) ## Random step
    if random.uniform(0.0, 1.0) < Gauss(x + del_x, y + del_y, sigma)/Gauss(x, y, sigma):  #### Metropolis
        acceptance condition
        x, y = x + del_x, y + del_y ## Move is accepted
        n_accept += 1
    positions.append((x,y)) #### New position appended if move is accepted, else old position retained
        and appended.

#### Graphic visualisation of the Markov chain.
for i in range(1,len(positions)):
    pylab.arrow(positions[i-1][0], positions[i-1][1] , positions[i][0] - positions[i-1][0] , positions[i]
        [1] - positions[i-1][1], head_width=0.02, head_length=0.02, fc='r', ec='r')

print "Acceptance Ratio = ", 1.*n_accept/n_trials
pylab.show()
```

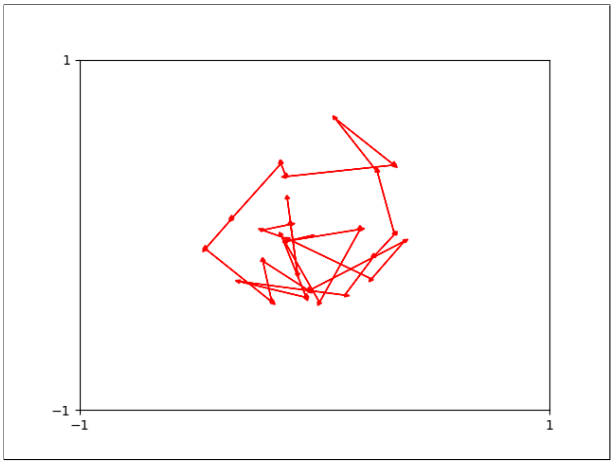

Figure 2: Walk with 50 moves.

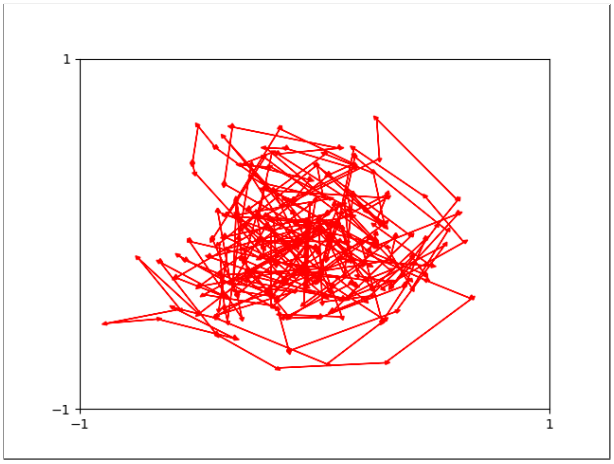

Figure 3: Walk with 500 moves.

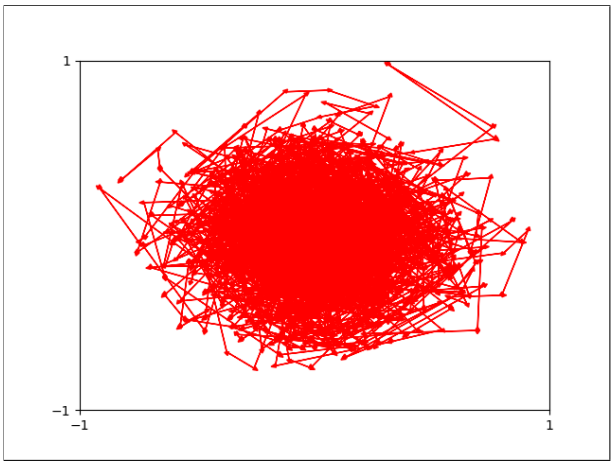

Figure 4: Walk with 5000 moves.

The following program computes the average of the function $f(x, y) = x^2y^2$

```
import random
import math


def f(x,y):  #### Function to be averaged
    return (x**2)*(y**2)

def Gauss(x,y,sigma): ### 2D Gaussian distribution
    return (1./(2*math.pi*sigma**2))*math.exp(-(x**2 + y**2 )/(2.*sigma**2))



x, y = 0., 0. ### Start at the centre of the Gaussian (`relevant' point)
sigma = 0.2 #### Gaussian width
delta = 0.5  ##### Random walk step size
n_trials = 80000 ## Number of steps
n_accept = 0 ## Number of moves accepted
positions = [(x,y)] ## List of points sampled
for i in range(n_trials):
    del_x, del_y = random.uniform(-delta, delta), random.uniform(-delta, delta)## Random step
    if random.uniform(0.0, 1.0) < Gauss(x + del_x, y + del_y, sigma)/Gauss(x, y, sigma): ## Met
        x, y = x + del_x, y + del_y  #### Accept the move
        n_accept += 1
    positions.append((x,y))   ### New position appended if accepted, else old position appended



### Averaging process ####
sum = 0.
for x,y in positions:
    sum += f(x,y)   ##### Sum of values of function

print "Acceptance Ratio = ", 1.*n_accept/n_trials
print " Monte Carlo average of function = ", sum/n_trials
print "Analytically computed average = ", sigma**4
```

```
Acceptance Ratio =  0.3647125
 Monte Carlo average of function =  0.00162164700055
Analytically computed average =  0.0016
```