Introduction to Monte Carlo Methods

Dr. Ananda Dasgupta
IISER Kolkata

Autumn Semester 2017

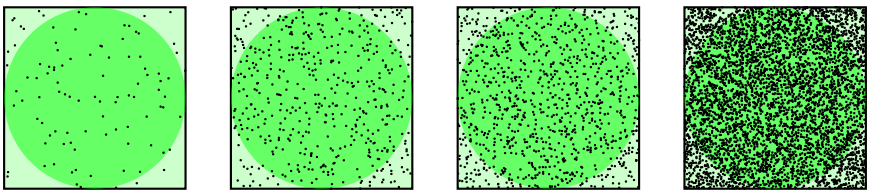
1 Monte Carlo Methods

The Monte Carlo methods of simulation use computer generated pseudo-random numbers to study naturally occurring random processes. It was first used in physics by von Neumann, Ulam and Metropolis near the end of the Second World War to study the diffusion of neutrons in fissionable material. The novel contribution of von Neumann and Ulam was to realize that determinate mathematical problems could be treated by finding a probabilistic analogue which is then solved by a stochastic sampling experiment.

2 A simple illustration

π by throwing pebbles

Imagine a child throwing pebbles at random on a square field. If she manages to do it entirely at random, the fraction of pebbles that lands inside the inscribed circle, in the limit of a very large number of pebbles, approaches the ratio of the areas of the circle and the square : $\frac{\pi}{4}$! The series of pictures below show this in action - although, out of fear of being accused of child abuse, we have enlisted the help of a computer in carrying out the throwing and counting!



$n = 100$, $\pi_{\text{est}} = 3.2$ | $n = 500$, $\pi_{\text{est}} = 3.172$ | $n = 1000$, $\pi_{\text{est}} = 3.108$ | $n = 5000$, $\pi_{\text{est}} = 3.1264$

π by throwing pebbles - the code

```
from random import random
```

```
N = input('Enter number of pebbles : ')
Nhits = 0

for i in range(N):
        x = 2*random()-1
        y = 2*random()-1
        if x*x+y*y < 1:
                Nhits += 1

print 'estimate for pi : ', 4*float(Nhits)/N
```

3 Monte Carlo integration

This method of estimating π is actually a determination of the area of the circle - which in turn is actually the double integral $\iint_C dxdy$. Monte Carlo techniques are a very good way for calculating higher dimensional integrals. The standard approach towards calculating a higher-dimensional integral is via iterative evaluation of one-dimensional integrals, *e.g*

$$\iiint_{B_3} f(x,y,z)\ dxdydz = \int_{-1}^{+1} dx \int_{-\sqrt{1-x^2}}^{+\sqrt{1-x^2}} dy \int_{-\sqrt{1-x^2-y^2}}^{+\sqrt{1-x^2-y^2}} dz f(x,y,z)$$

If you try to evaluate this numerically by using, say, 10-point Simpson method, you will need to evaluate the inner double integral (that over y and z) at 10 values of x - each of these will require the evaluation of the innermost (z) integral at 10 values of y, and these, in turn will require the evaluation of the integrand at 10 values of z. This is a total of 1000 function evaluations. The number of points where the integrand needs to be evaluated increases exponentially with dimension in all standard integration techniques (like trapezoidal, etc.) In Monte-Carlo methods the error is $\propto \frac{1}{\sqrt{N}}$ - independent of the number of dimensions! Thus, although evaluation of a lower dimensional integral will require a much larger number of points if you try Monte Carlo, as opposed to, say Simpson's method for the same level of accuracy - the situation is reversed for higher dimensional integrals.

π again - this time from a sphere

The code

```
import numpy as np

N = input('Enter number of points : ')
```

```
Nhits = 0

for i in range(N):
        pt = np.random.rand(3)
        if pt.dot(pt)<1:
                Nhits += 1

print 'Volume  : ',float(Nhits)/N
```

4 Throwing pebbles again - but locally

Going back to our child throwing pebbles , consider, now, a situation where the field is big! It may not be possible for the poor child to throw pebbles uniformly over such a large area. So, let's modify the game a bit! Imagine that the kid now starts at some fixed point inside the square field and throws a pebble at random with eyes closed. If the pebble lies inside the field, she walks over to where the pebble has landed, and throws another pebble from there. She keeps up at this game for a large number of throws. If at any stage the pebble lands outside the field, the kid just places **another** pebble at her current location and tries again! At the end of the game, she counts the number of pebbles that are inside the circle. The ratio of this number to the total number of throws gives an estimate for the ratio of the areas of the circle and the field.

Note that in this case too, somehow the pebbles end up being distributed uniformly all over the field. The probability of a pebble being added to a given point in the $n+1$th step, depends only on the position of the nth pebble! This is the essence of a **Markov Chain**!

The idea of putting down another pebble whenever one is thrown out of the field sounds counter-intuitive at first. It seems that we are, in a way, rewarding failure! Notice that if the child has a small reach, then she is most likely to throw a pebble out of the field if she is standing close to an edge (even more so if she is near a corner). It is more likely that pebbles will pile up on top of each other at such points, rather than at a point in the middle of the filed. On the other hand, a region near the middle of the field can have pebbles coming in from all directions (depending, of course, on the position of the kid in the last step)- the option is much less if you are near the edge.In other words, the parts of the field that are less likely to be reached are also the ones harder to get out of! The two effects balance out to give the uniform distribution.

The code

```
from random import random
x,y = 1.0,1.0
```

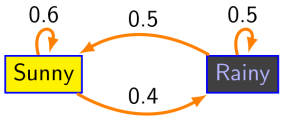
```
n,NN = 0,1000000

d = 0.5
for i in xrange(NN):
        dx = (random()*2-1)*d
        dy = (random()*2-1)*d
        if abs(x+dx)<=1 and abs(y+dy)<=1:
                x = x+dx
                y = y+dy
        if x**2+y**2<1:
                n += 1
print 4*float(n)/NN
```

5 A tale of two (types of) days

In the fictitious City of Markovia - each day can only be either of two kinds - either sunny or rainy! If today is sunny, the chance that tomorrow will be rainy is 40%, which, of course means that the chance of tomorrow being sunny is 60%. On the other hand, if today is rainy, then chances are even that tomorrow will be either sunny or rainy. The question that we are trying to answer is : in the long run, what is the chance that a given day will be rainy or sunny?

In such problems it helps to think in terms of **ensembles**. Imagine a universe where you have millions and millions of copies of Markovia - identical cities in which the same rules are going to play out over time. Physics would have been a very costly affair, of course, if we really had to build this - but thoughts are free, after all! The state of affairs in Markovia can be depicted by the picture on the left below. The equations on the right are relevant for our multi-Markovia ensemble. Here $P_R^{(n)}$ and $P_S^{(n)}$ denote the probabilities that an arbitrary city in the ensemble has a rainy or a sunny n-th day, respectively. The equations simply translates the diagram into maths! Such an equation that describes how probabilities evolve at each step in a stochastic process is called a **master equation**.



$$P_S^{(n+1)} = 0.6P_S^{(n)} + 0.5P_R^{(n)}$$
$$P_R^{(n+1)} = 0.4P_S^{(n)} + 0.5P_R^{(n)}$$

In case you find it a bit difficult to follow the equation, here's a cartoon version of the argument that can be used to understand it. If the number of cities in our multi-Markovia ensemble is N_{ens}, then on the n-th day, the number of cities that will have a sunny day is $P_S^{(n)}N_{\text{ens}}$, while the rest, $P_R^{(n)}N_{\text{ens}}$ will have a rainy day (of course, here we are making the sin of replacing the fraction of cities by the probabilities - but, as you must be aware, this is not a bad thing to

do if the total numbers are huge, as we assume here). Out of the $P_S^{(n)} N_{\rm ens}$ which are having a sunny n-th day, 60%, or $0.6 P_S^{(n)} N_{\rm ens}$ will continue to have a sunny $n+1$-th day. On the other hand, half of the other $P_R^{(n)} N_{\rm ens}$ will also have a sunny $n+1$-th day. So, the total number of cities that will have a sunny $n+1$-th day is

$$0.6 P_S^{(n)} N_{\rm ens} + 0.5 P_S^{(n)} N_{\rm ens}$$

and so on ...

The code and results

The code below simulates the multi-Markovia ensemble. The array days stores the state of a given day (1 codes for "sunny", and 0 for "rainy") in each of the NN cities in the ensemble. The inner loop visits each city (actually, each element of the array, and determines probabilistically if the state (sunny or rainy) should be changed. The trailing end of the output of the code, which shows the fraction of cities in multi-Markovia which are sunny on a given day is shown alongside. Although here we have started with an ensemble where all the cities have a sunny day to begin with, in the long run things seem to settle down to a situation where about 55% of the cities are sunny on a given day.

```
import numpy as np

NN = 100000
Ndays = 1000

days = np.ones(NN,dtype="int")

for i in xrange(Ndays):
    for j in xrange(NN):
        r = np.random.rand()
        day = days[j]
        if day == 1 and r<0.4:
            days[j] = 0
        if day == 0 and r<0.5:
            days[j] = 1
       if i% 20 == 0:
            print i,'\t',float(sum(days))/NN
```

```
680  0.55554
700  0.55448
720  0.55439
740  0.55703
760  0.55663
780  0.5576
800  0.5546
820  0.55438
840  0.55712
860  0.55564
880  0.55533
900  0.55672
920  0.55539
940  0.55478
960  0.55498
980  0.55609
```

Note that we could have cheated and just used the master equation to see how the probabilties evolve over time. A simple line of python like

```
P_r,P_s = 0.4*P_s+0.5*P_r,0.6*P_s+0.5*P_r
```

inside the loop would have sufficed.

6 Markov chains

Some simple math

The master equation for Markovian weather

$$\begin{aligned} P_S^{(n+1)} &= 0.6P_S^{(n)} + 0.5P_R^{(n)} \\ P_R^{(n+1)} &= 0.4P_S^{(n)} + 0.5P_R^{(n)} \end{aligned}$$

can be re-written in the matrix form below :

$$\mathbf{P}^{(n+1)} = \mathbf{M}\,\mathbf{P}^{(n+1)}, \qquad \mathbf{P}^{(n)} \equiv \begin{pmatrix} P_S^{(n)} \\ P_R^{(n)} \end{pmatrix},\ \mathbf{M} \equiv \begin{pmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{pmatrix}$$

This means that

$$\mathbf{P}^{(n)} = \mathbf{M}^n \mathbf{P}^{(0)}$$

and thus, the evolution of probabilities reduces to a problem of linear algebra!

Obviously, all the elements of $\mathbf{M}$ must be non-negative. that the ij element of $\mathbf{M}$ is the conditional probability that given the system (in our example, the weather in a given day in Markovia) is in state j (two choices in our example) on the n-th day, it is in state i on the $n+1$-th. We often denote such "transition probabilities" by $W_{i\leftarrow j}$. We obviously must have $\sum_i M_{ij} = \sum_i W_{i\leftarrow j} = 1$, the system must be in one of the states i on the $n+1$-th day! Matrices obeying this condition are called **Markovian** matrices or **stochastic** matrices.

If $\mathbf{M}$, which is a $\nu \times \nu$ matrix (for Markovia, ν is just 2 - but we are leaving open scope for generalization here) has ν eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_\nu$ corresponding to the eigenvectors $v_1, v_2, \ldots, v_\nu$. Then we can expand the initial column vector $\mathbf{P^{(0)}}$ in the form[1]

$$\mathbf{P^{(0)}} = c_1 v_1 + c_2 v_2 + \ldots + c_\nu v_\nu$$

Since $\mathbf{M}^n v_i = \lambda_i^n v_i$, we have

$$\mathbf{M}^n \mathbf{P^{(0)}} = c_1 \lambda_1^n v_1 + c_2 \lambda_2^n v_2 + \ldots + c_\nu \lambda_\nu^n v_\nu$$

If one of the eigenvalues, say λ_1, is larger in magnitude than *all* the others, then for large n we will have the first term on the left dominating the rest.

Now, it is obvious that the $1 \times \nu$ row vector $(1, 1, \ldots, 1)$ acted upon by $\mathbf{M}$ from the right is a row vector whose j-th term is $\sum_i M_{ij} = 1$. So,

$$(1, 1, \ldots, 1)\mathbf{M} = (1, 1, \ldots, 1) \qquad \Longrightarrow \qquad \mathbf{M}^T \begin{pmatrix} 1 \\ 1 \\ \ldots \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \ldots \\ 1 \end{pmatrix}$$

[1] Note that we have made a simplifying assumption that $\mathbf{M}$ has a complete set of eigenvectors - this is by no means guaranteed to happen!

Since a square matrix and its transpose has the same eigenvalues, it follows that one of the eigenvalues of $\mathbf{M}$ must be 1.

Moreover, it is also easy to see that any eigenvalue of a Markov matrix satisfies $|\lambda| \leq 1$. To prove this, let $v = (v_1, v_2, \ldots, v_\nu)^T$ be the corresponding eigenvector. Let v_k be the element with the largest magnitude. Then $\lambda v_k = \sum_j M_{kj} v_j \implies$

$$\begin{aligned} |\lambda|\ |v_k| &= \left| \sum_j M_{kj} v_j \right| \\ &\leq \sum_j |M_{kj}|\, |v_j| \\ &\leq \sum_j |M_{kj}|\, |v_k| \\ &\leq \left(\sum_j |M_{kj}| \right) |v_k| = |v_k| \end{aligned}$$

Since an eigenvector must be non-null, $|v_k| > 0$, which shows that $|\lambda| \leq 1$.

The two results above seems to indicate that after a large number of steps, the system will settle down to a steady state probabilty distribution - that corresponding to the eigenvalue 1. There are some caveats, though. What we had proven so far does not imply that all other eigenvalues (other than 1, that is) have magnitudes less than 1. Again, we have not proved that the eigenvalue of 1 is non-degenerate. Modulo these difficulties, it is usually true that the probability distribution settles down to a steady vale in the long run.

If this happens, then the elements π_i of $\lim_{n\to\infty} \mathbf{P}^{(n)}$ must obey

$$\pi_i = \sum_j W_{i \leftarrow j} \pi_j$$

Since $\sum_j W_{j \leftarrow i} = 1$, we have

$$\sum_j W_{j \leftarrow i} \pi_i = \sum_j W_{i \leftarrow j} \pi_j$$

This can always be satisfied if we demand **detailed balance**

$$W_{j \leftarrow i} \pi_i = W_{i \leftarrow j} \pi_j$$

Note that while this condition is not necessary for the system to achieve steady state, it (along with a technical ciondition called ergodicity) is sufficient to ensure that it does reach steady state. What is also really nice about this principle is that if you need to sample states according to some probability distribution $\{\pi_i\}$, you can design a Markov chain that does that, by simply chosing

$$\frac{W_{j \leftarrow i}}{W_{i \leftarrow j}} = \frac{\pi_j}{\pi_i}$$

The added bonus here is that we don't need to know the actual probabilities - only the relative probability will do! This is especially important in statistical mechanics - because there we know the relative probability of two states as long as we can determine their energies - while the actual value of the probabilities may be very difficult to determine.

A famous Markov chain Monte Carlo algorithm - the **Metropolis Algorithm** - ensures detailed balance by first proposing a new state at random, and then accepting a proposed change $i \to j$ with the probability

$$p_{i \to j} = \min \left[1, \frac{\pi_j}{\pi_i} \right]$$

You can easily check that this method obeys detailed balance, while maximizing the chance that a proposed change will be accepted.