Monte Carlo Simulations

(Pseudo) Random Number Generator

Randomness loosely means that we can't expect what is going to happen next. But, sometime you have heard the word uniform random number which mean you want every number to have equal chance of happening. This, however, does not mean that the uniform random number generator for number $\in [1, 10]$, after giving 1-9 will have to give 10 on the next turn. The statement only says about probability which means that over infinitely long amount of time each number should appear equally.

This however cannot be achieved perfectly without a good external source. For example, one could take CPU temperature and pattern of disk/network usage as source of randomness. However, if we want a good randomness from these quantity, we will have to wait for sometime between requesting each number. If we poll the temperature two times in a millsecond, the two numbers are going to be pretty much the same. This makes external source a bad idea for generating a large amount of random numbers.[1]

So we settle with the idea of *pseudo random number generator*. We make something determinitic and fast that gives us a seemingly uniform and seemingly random.

To get uniformity, we could use modulo. For example, if we want number from 0 to 10, we just take a number and take modulo of 11. So we could do something like

$$x_i = i \mod 11$$

This however will make our nubmer generator create a cyclic sequential number which makes is very lousy number generator. This can be fixed by using the concept of state. Which means the next number depends on the previous one. The number for x_0 is called the *seed*. [2]

$$x_{n+1} = x_n \mod 11$$

We can customize it a little bit by adding constants.

$$x_n = ax_n + b \mod 11$$

[1] There are people who sell result of dice as cryptographically secure random number generator. `http://arstechnica.com/business/2015/10/this-11-year-old-is-selling-cryptographically-secure-passwords-for-2-each/`

[2] Remember all those `np.random.seed` you saw in the exercise and homework? This is what it means.

This however doesn't fix the cycle length problem. This can be seen from the fact that once we get to $x_n = 3$ the next number is fixed. So the cycle length is still 11.

Some of the library really use this function. For example unix `rand()` actually uses

$$x_{n+1} = (1103514245 x_n + 12345) \mod 2^{31}$$

to generate 32 bit random number.

The cycle length can be make longer by having multiple states and xor all the results. This random number generator does not produce a good result and has many artifact. See. `https://en.wikipedia.org/wiki/Linear_congruential_generator`.

For scientific purpose, people usually use Mersene twister [3] to generate random numbers. This has a much longer cycle. Python `random.random()`, numpy `np.random()` both use Mersene Twister.

Monte Carlo Integration

One of the use for random number is for Integration. Let us consider the problem of evaluating triple integral

$$I = \iiint_C \exp(xyz)\, dx\, dy\, dz$$

over a cube $x \in [0,1]$, $y \in [0,1]$ and $z \in [0,1]$.

The trapezoid rule and the Simpson's rule we learned previously does not help much.

Let us consider another strategy of integration. Recall that integration is just the sum of the product of the function and volume element.

$$I \approx \sum f(x,y,z)\Delta V$$

where ΔV is the volume element. This would work perfectly if we us uniform points and the more points we have the better accuracy it becomes.

We can exploit this fact for numerical purpose. Instead of using uniform points, we can use uniform *random* points within a cube. So, the idea is generating a bunch of random points for each one of them you calculate $f(x,y,z)$ and you multiply associate volume element of each point. To determine the volume element of each point, we can just take the whole volume and divide by N. Specifically,

$$I \approx Q_N = \frac{V}{N} \sum f(x,y,z)$$

Non Rectangular Boundary

Sometime we don't always have rectangular bounday. Suppose you are integrating over a volume C. This can simply be done by using a modified function.

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

$$g(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } x \in C \\ 0 & \text{otherwise} \end{cases}$$

This function works because it just ignore all the volume outside our bound of integral.

For example, let's suppose we want to perform the following integration.

$$\iiint_C e^{xyz}\, dx\, dy\, dz$$

over C is the unit sphere centered at the origin.

To check whether $\vec{x} \in C$ we can just check whether $x^2 + y^2 + z^2 < 1$ or not. That means

$$g(\vec{x}) = \begin{cases} e^{xyz} & \text{if } x^2 + y^2 + z^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

Then we can generate the random points over the cube $x \in [-1, 1]$, $y \in [-1, 1]$, $z \in [-1, 1]$. For each one of them we sum them up. Then multiply by the volume $(2 \times 2 \times 2 = 8)$ and divide by the number of points we generate.

Convergence

The convergence of Monte Carlo integration is a little bit different from what we learn about convergence. Since our method is inherently random, every time we did the experiment we are going to get a different answer. Thus, unlike trapezoid rule, we can't say with absolute certainty that our answer is going to be within some bound from the real answer.

We can thing about the result from each run as being drawn from some distribution which has some width. This width of the distribution that our answer is drawn from can be used to measure the convergence of our answer since the mean of our Q_N over infinite number of experiment(using N points) is I.

Let us find the variance of Q_N. Our random variable is $f(\vec{x}_i)$. This is the thing that will change if we repeat the experiment. Recall from Discrete Math that

1. $\text{Var}(kR) = k^2 \,\text{Var}(R)$ if k is a constant.
2. $\text{Var}(A + B) = \text{Var}(A) + \text{Var}(B)$ if A and B are independent.

This means

$$\begin{aligned}\mathrm{Var}(Q_N) &= \mathrm{Var}\left(\frac{V}{N}\sum f(\vec{x}_i)\right)\\ &= \frac{V^2}{N^2}\mathrm{Var}(\sum f(\vec{x}_i))\\ &= \frac{V^2}{N^2}N \times \mathrm{Var}(f)\\ &= \frac{V^2}{N}\mathrm{Var}(f)\end{aligned}$$

Thus the standard deviation of Q_N over repeated experiment is given by

$$\sigma_{Q_N} = V\sigma_f \times \frac{1}{\sqrt{N}}$$

where σ_f is the standard deviation of f over the volume of our random points generation.

The most important part of this expression is that the standard deviation drops like $1/\sqrt{N}$. This is independent of the number dimension of integration which is a very neat property. Recall from trapezoid that the error or integration depends on the size of each piece. Suppose that we divide each dimension in to N pieces. If we want to do d dimensions and keep the same size on each dimension, the number of pieces we need would be N^d. This grows really fast with the dimension. The fact that convergence of monte carlo integration is independent from the number of dimension d makes it very attractive for performing multi-dimension integration.[4].

Simulations

Loaded Dice

Another use for the Monte Carlo method is in performing simulations. Let us consider the problem of loaded die.

Outcome	1	2	3	4	5	6
Probability	0.3	0.2	0.1	0.2	0.1	0.1

There are a lot of things we can do with this. First, we could ask what is the probability that when we toss this dice two time, the sum of the value will add up to 6. Of course, you could do this by hands[5]. But, let us use computer to do it.

So first we need to simulate the process of dice rolling. This can be done by generate a random number x in the range of $(0, 1)$. Then we compare the value against the cumulative distribution. Specifically, if $x \in (0, 0.3)$ then we make the result 1. If $x \in [0.3, 0.5)$ then we make the result 2. If $x \in [0.5, 0.7)$ then we make the result 3 and so on.

[4]This is found a lot when we try to normalize distributions in the fit.

[5]Right!?

Now we need to keep tossing the two dice and count the number of times the result is 6. The probability would then be given by

$$P[\text{Sum is 6}] = \frac{\#\text{ of times we get 6}}{\text{Total number of times we roll}}$$

You can imagine playing this game with multiplying the two number or even dividing the two. You can even go extreme and consider something like monopoly game and use a similar simulation to figure out the probability of landing on each spot.

Simulating Stock Price

Sometimes the sample we want is not just a simple number but a series of number. For example, one thing we might be interested in is stock price. We could ask about given a model for a stock. What is the probability that the stock will go down below a certain value? Or what is the expected profit/loss for a given buy/sell stragey on a certain stock. Of course we only know the stock price in the past but not in the future. We want to know about the future.

The idea is the following. From the past, we know some characteristic from the stock like the variance or the mean return. Then given these characteristics we want to generate a bunch of scenario of what the price can happen in the future. This will have a specific distribution. From these samples of various scenarios of stock price, we can then find probability of various type of event from these sample.

One of the popular way to generate stock price is called Geometric Brownian Motion[6]. In a nutshell, it just says that the percentage change of stock price is normal distributed[7]. The fomula for this is given by

$$s(t+dt) = s(t) + s(t)\left(\mu \times dt + \epsilon\sigma\sqrt{dt}\right)$$

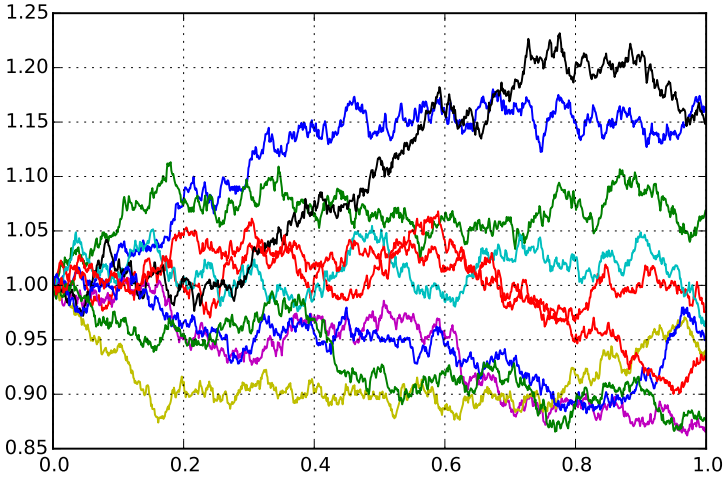
where

- $s(t)$ is the stock price at time t.
- dt is the step in time. (Ex: 1 day).
- μ is the mean rate of return. Ex. (2% per year.). When we plug this in the formul we need to make sure the unit of dt and μ matches. If μ is typically given in per year unit, so your dt should be per year unit.
- ϵ is a normal distributed random number of mean 0 and standard deviation and 1.
- σ is called volatility. It is the width of distribution. It loosely represent how far/how likely it is to deviate from the mean rate of returns. The it has a funny unit of percent/$\sqrt{\text{time}}$. If you look up for this number on the internet, the unit is normally unspecified but they mean %/$\sqrt{\text{year}}$.

Using this you can generate a bunch of price trend. The result is shown below.

[6] One of the technology Physics export to finance world.

[7] The real correct way to say this is that the log of return is normally distributed.



Why $\sqrt{dt}$? (Just for your curiosity)

Everything looks like it is what we expect except for the part where we have $\sqrt{dt}$. The $\sqrt{dt}$ is the right expression since it preserve the scale invairant of time. This can be seen by suppose that we split the time in to n pieces we should get the same width of the distribution.

For if we use one time period dT. The fractional change of the stock price is normally distributed with mean $\mu \times dT$ with standard deviation $\sigma\sqrt{dT}$. That is of the variance $\sigma^2 dT$

$$\frac{s(t+dT)-s(t)}{s(t)} = \mu \times dT + \epsilon\sigma\sqrt{dT}$$

Therefore,

$$\begin{aligned}\mathrm{Var}\left[\frac{s(t+dT)-s(t)}{s(t)}\right] &= \mathrm{Var}[\mu \times dT + \epsilon\sigma\sqrt{dT}] \\ &= \mathrm{Var}[\epsilon\sigma\sqrt{dT}] \\ &= \sigma^2 dT\,\mathrm{Var}[\epsilon] \\ &= \sigma^2 dT\end{aligned}$$

which means that the variance grows linearly with time. The same thing can be done with the mean.

Now let us split dT into N pieces. If dT/N is small enough, the percentage change for each step is small. We can use the approximation that we can add the percentage change together.[8]

[8] If you want to be rigourous about this, take the ratio and take log

$$\frac{s(t+dT)-s(t)}{s(t)} \approx \underbrace{\left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_1 + \ldots + \left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_N}_{N\text{terms}}$$

The variance of the expression above is given by

$$\text{Var}\left[\frac{s(t+dT)-s(t)}{s(t)}\right] = \text{Var}\left[\left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_1 + \ldots + \left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_N\right]$$

Using the fact that $\text{Var}[A+B] = \text{Var}[A] + \text{Var}[B]$ if A and B are independent we have

$$\text{Var}\left[\frac{s(t+dT)-s(t)}{s(t)}\right] = \text{Var}\left[\left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_1\right] + \ldots + \text{Var}\left[\left(\mu \times dT/N + \epsilon\sigma\sqrt{dT/N}\right)_N\right].$$

each one of the term is just $\sigma^2 dT/N \,\text{Var}[\epsilon] = \sigma^2 dT/N$. Therefore,

$$\text{Var}\left[\frac{s(t+dT)-s(t)}{s(t)}\right] = N \times \sigma^2 \frac{dT}{N} = \sigma^2 dT$$

which has the right scaling behavior.

Hill Climbing

Another application of random number is in optimization. The idea is similar to gradient descent. The basic idea goes as follows

1. Start somewhere
2. Walk randomly from the place you are.
3. If the new place you end up with has lower cost function go there.
4. Keep walking(randomly).
5. "Eventually" we will end up in a very good place.

At first this algorithm sounds really stupid why would one walk randomly when we have information from gradient. The algorithm can be summarized as greedily blind. However, gradient is not always available and that is where this method shines. For example, if we want to find a graph(as in edges and vertices) which gives the minimum score. There is no concept of derivative of edges and vertices. Let us look at a couple examples.

Largest Triangle

Let us consider the problem of finding the largest triangle in a given set of points. Given a bunch of points in 2D planes, the task is to figure out the thre points which gives the largest area.

We can start solving this problem by picking 3 random points. The cost function is then just the negative of the area[9]. The move can be defined by picking one point from the three and replace it with another random points. If the points give you more area, we keep it. If not, discard it.

Simulated Annealing

One of the problem we find in hill climbing algorithm is that it usually get stuck at the local minimum while we are trying to find global minimum.

This can be fixed by allowing some bad configuration as our move. However if we always allow bad move moves we will not reach the minimum. So, the idea is that we want to allow some bad moves in the beginning. In addition, if possible, we want to also allow drastic move(whatever drastic means in the context) in the beginning. All these are there to avoid getting stuck at the local minimum in the beginning. If we stuck at least we want it to be a deep minimum. Which means the probability of accepting a bad move should be some function of how bad that moves is. If the move is really bad then we want to reject it but if it is not so much worse from the configuration we are at we may want to accept it. Then the second thing we want is so that is probability should die down as the time goes on[10].

One popular way to do that is something we borrow from Physics called simulated annealing. The probability of accepting an answer goes as follow for minimization problem.

$$P(\Delta s) = \begin{cases} 1 & \text{if } \Delta s < 0 \\ \exp\left(-\frac{\Delta s}{T}\right) & \text{if } \Delta s > 0 \end{cases} \tag{1}$$

where Δs is the difference in the score and T is the temperature. For maximize problem we can just flip the sign on the exponent. To make sense out of thise function. First notice that it does the right thing if the the score improves $\Delta s < 0$; it just accept the new state. If the new state is worse than the old state the probability exponenotially falls off with the how worse off we are with new state. The temperature T is the parameter that scales the probability. This is the parameter that changes with time. If the temperature is high, the probability will be close to 1 which means that we accept almost all the new states regardless of the improvement/diminishment. If the temperature is low, the probability is going to be close to zero which means the only thing we accept is state that improve the score.

With this behavior of temperature T, we want the temperature to be high in the beginning and zeror at the end. This means the algorithm will get more and more greedy as we go. The function that dictate how fast the temperature goes down with iteration is called temperature scheduling. There are many ways to do it. The slower the schedule

[9]The negative is so that the problem becomes minimization.

[10]The step we take, if possible, should be less drastic.

is the more likely we are going to get to the global minimum. Yet, at the same time such schedule will waste more time. One popular way is to use a linear function.

$$T = T_0 - T_0 \frac{k}{k_{max}}$$

where k is iteration number and k_{max} is maximum number of iterations. There are many other choices that has the right behavior. The slower the temperature cools down the more likely we are of getting the global maximum but that also means slower convergence. Another way to do it is

$$T = \frac{T_0}{\log(1+k)}$$

where k is the number of iteration. This one cool down very very slowly. Use this one if global minimum is really desired. There is no right answer for temperature scheduling as long as it has the right behaving of cooling down as time goes on. Pick one or come up with one that suit the goal.

There is also a problem of picking T_0. The rule of thumb is to pick T_0 such that the probability that we accept a worse solution at the very beginning is about 80%. This can be done by experimenting the first iteration a couple times.

Scheduling Exam

MUIC runs many classes a term. Being the only true Liberal Art University in Thailand, students register for various combination of classes. At the end of each term the school needs to schedule the final exam. There are limited number of time slots ($\approx$25) in total. If possible we would like to find a way to schedule final exam such that no student have exam conflict. If there is one we want to minimize the number of students who have exam conflict.

If you learn a things or two from discrete math, this sounds like a graph coloring problem on steriod. We do not even know if such thing is possible if it's not possible then we want a configuration that minimize the number of students who have time conflict.

This is a perfect problem for heuristic algorithm like Hill Climbing. We are not looking for a perfect solution. We are looking for a "decent" solution not neccessarily perfect. Hill Climing algorithm will give a decent solution if not a perfect solution.

To solve this we need to define the state, the cost function and the "move". The state are just the color of each class. The cost function is simply the number of students who has conflict. For the move, we can just find a random point and assign it a random color. Then if the random color we assign gives us better score we accept it, if not we discard it with some probability according to the temperature.

Label Placement

Sometimes the cost function is not so obvious. We need to come up with one from a broad description of the problem. Let us considr the problem of Automatic Label Placement(ALP). Placing city labels on a map is actually not so easy. There are two

things we want from label placement. First, we want the label to be near the location it represents. At the same time, we want to avoid having two labels on top of each other.

Again this is one of the problem that we do not need the optimal answer. In this case, we do not have the exact definition of perfect either. We just want anything presentable. Thus, all our cost function should do is to represent the presentability. This has two folds as described in the previous paragraph.

1. The score should go down if the label is close to the location on the map. Something like the square difference of the two location works.

2. The score should go really high up if the two labels overlap. Some constant times the overlap area works quite well.

If we combined the two, we would come up with the score that looks something like

$$\text{cost(labels locations)} = \sum_{\text{all labels}} |\vec{x}_{\text{label}} - \vec{x}_{\text{location}}|^2 + k \times \sum_{\text{all pair of labels}} \text{Overlapping Area} \quad (2)$$

where k is a large postive number[11].

It is possible to come up with different cost function that have the same behavior. There is really no "the correct answer". Anything that has the right behavior will get you to a decent solution. But one must be careful about the value of k. This is because of the trade off between being farther from the location and having overlapping region. The distance should be under almost no circumstance be trade off with overlapping area. This tell us that k has to be large.

The move would be quite obvious on this one. Pick a random label. Move it a little bit from the previous state. In this case we can make the move smaller and smaller as the time goes on the same way we did gradient descent.

It should be mentioned that there is another way to do this using Physics. We can use the concept of force to do this. We can just make the label push each other like the way the charges pushing each other. Then, since the label want to be near the location we can put a spring there. This actually allows us to write an ODE and write time evolution. We will cover how to do something like this next week.

[11] We still need to make sure that the distance does not get lost beyond machine precision.