UNIVERSITY OF CALGARY

PHYSICS 581

COMPUTATIONAL PHYSICS III

Lab #1

Authors
Kunal Naidu 30020999
Ruand Hennawi 30024661

Instructor
Dr. Rachid Ouyed

June 5, 2023

Contents

1. Introduction

In this report, we explored various techniques related to Random Number Generators (RNGs) and Monte Carlo Method. We started by studying Pseudo RNGs by looking at the linear congruential method, and the correlation plot to test the RNG, and then applied an Auto-Correlation test for independence to evaluate the randomness of the RNG. Next, we looked at Probability Distribution Functions (PDFs) and the Accept/Reject method, and studied the Radiative Transfer of a photon and its fate. We also explored the Markov Chain Monte Carlo with a classic example of sunny and rainy weather, and the steady state distributions. Additionally, we examined the Metropolis-Hasting algorithm, as well as the Burn-in phase and the Simulated Annealing and Hill climbing algorithms.

2. Methods

Monte Carlo

Monte Carlo Method is a probabilistic numerical technique used to estimate the outcome of a given, uncertain and stochastic process. This means that it solves problems where the behavior of a system or process is uncertain or subject to randomness. This is usually a case when we have random variables in our processes. It works by generating a large number of random samples, based on a probability distribution of the inputs, and using these samples to estimate the desired outputs. By repeating this process multiple times and averaging the results (*The Law of Large Numbers*), the Monte Carlo method can provide a robust and accurate solution to problems that would be otherwise difficult or impossible to solve analytically.

The Law of Large Numbers describes what happens when performing an same experiment many times; the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. The Monte Carlo Method is a way to apply the Law of Large Numbers in practice, by using large numbers of simulations to estimate the expected value of a random variable.

Random Number Generators (RNGs)

Monte Carlo methods relies heavily on random number generation schemes, generally on **Pseudo Random Number Generators (PRNGs)**. The most well known algorithm for generating pseudo random sequences of integers is the **Linear Congruential Method**. It is a simple and efficient method for generating a sequence of numbers that appears random, but is actually not.

The basic idea behind the Linear Congruential Method is to generate a sequence of numbers using a linear equation, where each number in the sequence is determined by the previous number. The formula for the linear congruential method is

$$I_{n+1} = (AI_n + C)\ mod\ (M) \tag{1}$$

where A is the multiplier, C is the increment, and M is the modulus. They produce a sequence of integers between 0 and $M-1$. I_{n+1} is a random integer yielded from $(AI_n + C)$ and $mod\ (M)$. The method requires an initial seed, I_0 to initialize the sequence.

RNG Tests

Two important statistical properties of random numbers is **Uniformity** and **Independence**. First, for *uniformity*, χ^2 test is done to check if the random sequence is independently and identically distributed (**iid**). This is done by measuring how a model compares to real observed data; compares the observed frequencies with the expected frequencies under the assumption of a uniform distribution tests to see if they match.

Second, the auto-correlation test for *indepen-*

dence. It is used to identify non-randomness in a data set. Moreover, auto-correlation plots are a commonly used tool for checking randomness in a data set. To evaluate the serial correlation, auto-correlation can be used which for a sequence of random numbers determines how related the sequence is to itself shifted by k numbers and is given by

$$\begin{aligned} AC(k) &= \frac{E[(x_t - \bar{x})(x_{t+k} - \bar{x})]}{\sigma^2} \\ &\approx \frac{1}{N-k} \frac{\sum_{t=1}^{t=N-k} (x_t - \bar{x})(x_{t+k} - \bar{x})}{\sigma^2} \end{aligned} \tag{2}$$

where $\bar{x}$ is the average value of the random number sequence, and $\sigma^2 = \frac{1}{N-k} \sum_{t=1}^{t=N-k} (x_t - \bar{x})^2$ is its variance.

The Fundamental Principle

The essence of the Monte Carlo Method is sampling from **Probability Distribution Functions (PDFs)**.

A **Cumulative Distribution Function (CDFs)** describes the probability that a random variable is less than or equal to a certain value. A **Probability Distribution Function (PDF)** describes the probability of the occurrence of that random variable within a given range. A CDF is obtained from the integral of a PDF.

Consider a random variate x_0, so that the integral of a PDF, $P(x)$, gives a CDF denoted as $\psi(x_0)$. In this case,

$$\psi(x_0) = \frac{\int_a^{x_0} P(x)\, dx}{\int_a^b P(x)\, dx} \tag{3}$$

As x_0 ranges from a to b, $\psi(x_0)$ ranges from 0 to 1 uniformly. Thus, to sample a random variate x_0, we just need to call a random number generator that samples from 0 to 1 uniformly (we call this "uniform random deviate" ζ), and invert equation above to get x_0. If the CDF of $P(x)$ is difficult to estimate, the **Accept/Reject Method** can be used.

The Accept/Reject Method is a method, used in Monte Carlo, that generates random samples from a known, easy to sample distribution, and then either accepts or rejects those samples based on their probability of being from the desired distribution. In other words, provided that the peak of a PDF is known, then the algorithm used is to first sample x_0 from a uniform distribution within the range of the PDF.Then sample y from a uniform distribution between zero and the peak value of the PDF, y_{max} and calculate $y_0 = P(x_0)$. If $y > y_0$, then reject this x_0 and return to Step 1. Otherwise, accept this x_0.

Radiative Transfer

The **Radiative Transfer** is when a photon is emitted, then travels a distance, and something happens to it.

Consider a slab with constant density, n. A photon will be injected in only upward directions with $z = z_{min} = 0$ being the bottom of the slab with a physical distance, L, that the photon has travelled and is given by

$$\tau = \int_0^L n\sigma\, ds = n\sigma L \tag{4}$$

where τ is the optical depth. From this equation we get

$$L = \frac{\tau}{n\sigma} \tag{5}$$

and

$$\begin{aligned} \tau_{max} &= \int_0^{z_{max}} n\sigma\, dz \\ \tau_{max} &= n\sigma\, z_{max} \\ \frac{\tau_{max}}{z_{max}} &= n\sigma \end{aligned} \tag{6}$$

then, substituting (5) into (4) gives

$$L = \frac{\tau\, z_{max}}{\tau_{max}} \tag{7}$$

The probability that the photon travels an optical depth τ is

$$P(\tau)\, d\tau = \exp^{-\tau}\, d\tau \tag{8}$$

with $\tau = 0$ being the bottom of the slab at $z = z_{min} = 0$ and $\tau = \tau_{max}$ being the top of the slab at $z = z_{max}$. Using equation (3), we can find τ to be

$$\begin{aligned}
\psi(x_0) &= \frac{\int_a^{x_0} P(x)\,dx}{\int_a^b P(x)\,dx} \\
\zeta &= \frac{\int_0^{\tau} e^{-\tau}\,d\tau}{\int_0^{\tau_{max}} e^{-\tau}\,d\tau} \\
\zeta &= \frac{1 - e^{-\tau}}{1 - e^{-\tau_{max}}} \\
e^{-\tau} &= 1 - \zeta(1 - e^{\tau_{max}}) \\
\tau &= -\ln\big(1 - \zeta(1 - e^{-\tau_{max}})\big) \qquad (9)
\end{aligned}$$

The initial photon position is at the origin, $(x, y, z) = (0, 0, 0)$, and the initial direction of the photon is

$$n_x = sin\theta cos\phi, \quad n_y = sin\theta sin\phi, \quad n_z = cos\theta \qquad (10)$$

In an isotropic scattering atmosphere the photons are scattered uniformly into 4π steradians. The associated PDFs are

$$P(\theta) = \frac{1}{2} sin(\theta) \qquad (11)$$

$$P(\phi) = \frac{1}{2\pi} \qquad (12)$$

$P(\theta)$ is normalized over $[0, \pi]$ and $P(\phi)$ normalized over $[0, 2\pi]$. Using the *Fundamental Principle*, one can generate the initial and subsequent photon isotropic directions.

Substituting equation (11) into equation (3), taking into consideration that $P(\theta)$ is normalized, we get

$$\begin{aligned}
\psi(\theta) &= \int_0^{\theta} P(\theta)\,d\theta \\
\zeta &= \int_0^{\theta} \frac{1}{2} sin(\theta)\,d\theta \\
\zeta &= \frac{1}{2}(1 - cos(\theta)) \\
2\zeta &= 1 - cos(\theta) \\
\theta &= cos^{-1}(1 - 2\zeta) \qquad (13)
\end{aligned}$$

Now, substituting equation (12) into equation (3), taking into consideration that $P(\phi)$ is normalized, we get

$$\begin{aligned}
\psi(\phi) &= \int_0^{\phi} P(\phi)\,d\phi \\
\zeta &= \int_0^{\phi} \frac{1}{2\pi}\,d\phi \\
\zeta &= \frac{\phi}{2\pi} \\
\phi &= 2\pi\zeta \qquad (14)
\end{aligned}$$

Then, with equation (9) and $\mu = cos(\theta)$, and after moving a distance L, the photon's position becomes

$$\begin{aligned}
x &= x + Lsin(\theta)cos(\phi) \\
y &= y + Lsin(\theta)sin(\phi) \qquad (15) \\
z &= z + Lcos(\theta)
\end{aligned}$$

The absorb-or-scatter condition depends on the following ratio

$$\text{prob} = \frac{P_s}{P_s + P_a}$$

where P_s is the probability for a scattering event and P_a is the probability of an absorption event. For a constant density slab, the probabilities P_a and P_s are the absorption and scattering cross-sections, $P_a = \sigma_a$ and $Ps = \sigma_s$, considering continuum absorption only. To decide if a photon is scattered, one has to generate a number from $[0, 1]$ (ζ). If $\zeta <$ prob then the photon is scattered, otherwise the photon is absorbed. So, we can say that the range of prob is $[0, 1]$, and when $P_s = 1$ and $P_a = 0$ then it's full scattering, and when $P_s = 0$ and $P_a = 1$ then it's full absorption.

Markov Chain Monte Carlo

A **Markov Chain** is a mathematical model for a process which moves step by step through various states. The idea is that the probability of moving from one state to another is only dependent on the current state, and not on the past states, and this applies to the transition probability. A Markov chain consists of states and transition probabilities. A transition matrix is a matrix that is used to rep-

resent the transition probabilities between the states of a Markov chain. Each transition probability is the probability of moving from one state to another in one step. For example, the probabilities of weather conditions, given the weather on the preceding day, can be represented by a transition matrix as follows

$$\mathbf{P} = \begin{array}{c} \\ \text{Sunny} \\ \text{Rainy} \end{array} \begin{array}{c} \begin{array}{cc} \text{Sunny} & \text{Rainy} \end{array} \\ \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} \end{array} \tag{16}$$

Here, notice that the rows of P sum to 1. This is because P is a **Stochastic Matrix**. This matrix represents the probability that the next day is rainy, given today is sunny, is 10%. Therefore, it follows that the probability that the next day is sunny, given that today is sunny is 90%. It also represents the probability that the next day is sunny, given today is rainy, is 50% Since, the next day's weather is either sunny or rainy it follows that the probability next day is rainy, given today is rainy 50%.

The general rule for predicting the weather for day n is

$$x^{(n)} = x^{(n-1)}\, P \tag{17}$$

The weather on day 0 is known to be sunny. This is represented by a vector in which the "sunny" entry is 100%, and the "rainy" entry is 0%:

$$x^{(0)} = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{18}$$

So, if the weather is sunny today then what is the weather likely to be tomorrow? Using equation (17) for $n = 1$, we get

$$\begin{aligned} x^{(1)} &= x^{(1-1)}\, P \\ &= \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} \\ &= \begin{pmatrix} 0.9 & 0.1 \end{pmatrix} \end{aligned}$$

this means that there's a 90% chance that the next day is gonna be sunny and 10% that its gonna be rainy. What about two days from today? Using equation (17) for $n = 2$ and $n = 3$ we get

$$\begin{aligned} x^{(2)} &= x^{(2-1)}\, P \\ &= \begin{pmatrix} 0.9 & 0.1 \end{pmatrix} \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} \\ &= \begin{pmatrix} 0.86 & 0.14 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} x^{(3)} &= x^{(3-1)}\, P \\ &= \begin{pmatrix} 0.86 & 0.14 \end{pmatrix} \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} \\ &= \begin{pmatrix} 0.844 & 0.156 \end{pmatrix} \end{aligned}$$

This means that there's a 84.4% chance that in two days its gonna be sunny and 15.6% that its gonna be rainy.

However, if one keeps forecasting weather like this, eventually the nth day forecast, where n gets very large, reaches an equilibrium probability. In other words, your forecast for the n^{th} day and the $(n+1)^{th}$ day remain the same. You would get the same forecast for the weather if you start by assuming that the weather today is either sunny or rainy. This is called the **steady state** of the Markov process.

The **steady state** is a probability distribution over the states of the system such that the probabilities of future states are constant over time and are independent of the initial state, provided the transition matrix P is such that every state is eventually reachable from any initial state. The steady state condition is given by

$$x^{\infty} = P\, x^{\infty} \tag{19}$$

and the the sum of x^{∞} components is 1.

$$\sum_{i=1}^{N} x_i^{\infty} = 1 \tag{20}$$

So, using the matrix in equation (16), we can find x^{∞} by solving the following set of equations

$$\begin{aligned} x_1 &= 0.9x_1 + 0.5x_2 \\ x_2 &= 0.1x_1 + 0.5x_2 \\ 1 &= x_1 + x_2 \end{aligned}$$

so we find,

$$x_1 \approx 0.833333, \quad x_2 \approx 0.166667$$
$$\rightarrow x^{\infty} = (0.833333 \quad 0.166667)$$

Applying equation (19) gives

$$\begin{pmatrix} 0.833333 & 0.166667 \end{pmatrix} \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.833333 & 0.166667 \end{pmatrix}$$

and equation (20) gives

$$0.833333 + 0.166667 = 1$$

The Random-Walk Metropolis and The Metropolis-Hastings Algorithm

The **Random-Walk Metropolis** is an implementation of the **Metropolis-Hastings Algorithm**, where the proposal distribution is chosen as a symmetric random walk. In a Random-Walk Metropolis, the proposal distribution $q(x_n|x^*)$ is defined as a normal distribution centered at x_n.

The **Metropolis-Hastings Algorithm** is a Markov chain Monte Carlo method used for generating samples from a target probability distribution. It works by proposing new states for the system based on a proposal distribution and accepting or rejecting them with a probability proportional to their likelihood under the target distribution.

Let $P(x)$ be the target distribution. The Metropolis-Hastings algorithm starts with selecting a proposal distribution $q(x^*|x_n)$ and choosing an initial value for the state variable, x_0. Next, generate a proposal value, x^*, from the proposal distribution, $q(x_n|x^*)$, where x_n is the current state. Then, calculate the importance ratio given by

$$r = \frac{p(x^*)\, q(x_n|x^*)}{p(x_n)\, q(x^*|x_n)} \tag{21}$$

as well as the acceptance probability, given by

$$\alpha(x_n, x^*) \leftarrow min\left(1, \frac{p(x^*)\, q(x_n|x^*)}{p(x_n)\, q(x^*|x_n)}\right) \tag{22}$$

Generate a uniform random number u $[0, 1]$. If $\alpha(x_n, x^*)$ is greater than u, accept x^* as the new state. If alpha is less than u, the reject the candidate and set it as x_n instead.

$$x_{n+1} = \begin{cases} x^*, & \text{if} \quad u \geq \alpha(x_n, x^*) \\ x_n, & \text{if} \quad \text{Otherwise} \end{cases} \tag{23}$$

Here, $q(x^*|x_n)$ describes the probability of generating x^* as the candidate given that the current state is x_n, whereas $q(x_n|x^*)$ describes the probability of generating x_n as the candidate x^* being the current state. Moreover, if one chooses the proposal distribution $q(x_n|x^*)$, then the probability that x_n would be generated as the candidate value is

$$q(x_n|x^*) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x^* - x_n}{\sigma})^2} \tag{24}$$

If one chooses the proposal distribution $q(x^*|x_n)$, then the probability that x^* would be generated as the candidate value is

$$q(x^*|x_n) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x_n - x^*}{\sigma})^2} \tag{25}$$

We can see that, since the powers of the exponential in equations (24) and (25) are squared, this will yield the same answer for $q(x_n|x^*)$ and $q(x^*|x_n)$, and therefore, $q(x_n|x^*) = q(x^*|x_n)$. This shows that for the proposal distribution that's chosen, the Metropolis-Hastings algorithm simply becomes the Metropolis algorithm that is known as the Random-Walk Metropolis.

Consider the following target probability distribution,

$$P(x) = \frac{1}{2\sqrt{\pi}}(sin\, 5x + sin\, 2x + 2)e^{-x^2} \tag{26}$$

taking into account that the current state had is x^*, and x_n is the candidate value, then the expression

for the acceptance probability for equation (26) is

$$A(x^* \to x_n) = \min\left(1, \frac{1}{2\sqrt{\pi}}\left(\frac{(\sin, 5x_n + \sin, 2x_n + 2)e^{-x_n^2}}{(\sin, 5x^* + \sin, 2x^* + 2)e^{-x^{*2}}}\right)\right) \tag{27}$$

The Burn-in phase

A key issue in the successful implementation of Metropolis-Hastings or any other Markov Chain Monte Carlo sampler is the number of runs until the chain approaches stationarity (the length of the burn-in period). The burn-in phase is typically set to a large number of iterations, and the samples generated during this phase are thrown out typically the first 1000 to 5000 elements. After the burn-in phase, the samples generated are used for to assess whether the stationary state has indeed been reached.

Simulated Annealing

Imagine letting a ball roll downhill on the function surface, this is like descent for minimization. Now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking, this is like **Simulated Annealing**. Simulated Annealing is one of the most widely used optimization techniques. This method is used to find the minimum or maximum of a function by gradually cooling the temperature, T, of a metal. The goal is to bring the system, from an arbitrary initial state, to a state with the minimum possible energy.

Simulated annealing is very closely related to Metropolis sampling, differing only in that the acceptance probability is given by

$$A(x_n \to x^*) = min\left(1, \left(\frac{P(x^*)}{P(x_n)}\right)^{\frac{1}{T(n)}}\right) \tag{28}$$

where the function $T(n)$ is called the cooling schedule, and the particular value of T at any point in the chain is the temperature.

The Metropolis-Hasting algorithm is recovered when setting T = 1;

$$\begin{aligned} A(x_n \to x^*) &= min\left(1, \left(\frac{P(x^*)}{P(x_n)}\right)^{\frac{1}{1}}\right) \\ &= min\left(1, \left(\frac{P(x^*)}{P(x_n)}\right)\right) \end{aligned}$$

However, suppose we start with $P(x^*) = 0.5$ and $T = 100$ (high temperature or strong $P(x_n)$ "shaking" or "very bouncy ball") this gives a probability of

$$A(x_n \to x^*) = min\left(1, (0.5)^{\frac{1}{100}}\right) = 0.993$$

But what is obtained when one cools down to T = 1?

$$A(x_n \to x^*) = min\left(1, (0.5)^{\frac{1}{1}}\right) = 0.500$$

We can see that as the temperature T decreases, the acceptance probability decreases, which means the temperature has cooled down and the system is less "shaking", leading to fewer downhill moves. In other words, the probability of moving to a higher energy state, instead of lower is

$$p = e^{(\frac{-\Delta E}{kT})} \tag{29}$$

where ΔE is the positive change in energy level, T is the temperature, and k is Boltzmann's constant. As the temperature becomes lower kT becomes lower, $\frac{\Delta E}{kT}$ gets bigger, and so $e^{(\frac{-\Delta E}{kT})}$ gets smaller. How about when you cool to T = 0.1?

$$A(x_n \to x^*) = min\left(1, (0.5)^{\frac{1}{0.1}}\right) = 9.765 \times 10^{-4}$$

We can see that the acceptance probability decreases a lot.

ΔE represents the change in the value of the objective function. Since the physical relationships no longer applies, then the Boltzmann factor, k, is

dropped. So, equation (28) becomes

$$p = e^{(\frac{-\Delta E}{T})} \tag{30}$$

Let's say we give the acceptance probability for $P(x) = e^{-E}$. Then,

$$A(x_n \rightarrow x^*) = min\left(1, \left(\frac{e^{-E^*}}{e^{-E_n}}\right)^{\frac{1}{T(n)}}\right)$$

$$A(x_n \rightarrow x^*) = min\left(1, \left(e^{-\frac{1}{T}(E^* - E_n)}\right)\right) \tag{31}$$

We can see that this is the same as equation (29), and so the acceptance probability is the probability of moving to a higher energy state.

3. Results and Discussion

We first started by implementing the linear congruential method. For the parameters $I_0 = 3$, $A = 7$, $C = 0$, and $M = 10$ we saw the output to be

$$(3, 1, 7, 9, 3, 1, 7, 9, 3, 1)$$

We noticed that the values 3, 1, 7, 9 repeated in that order infinitely. We then plotted the unique values and their occurrences.

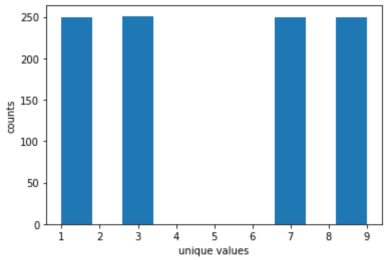

Figure 1: Comparing unique value to counts for the linear congruential method with the parameters $I_0 = 3$, $A = 7$, $C = 0$, and $M = 10$.

We again can confirm that the values 1, 3, 7, and 9 are the only ones to repeat. This periodicity in numbers seems to be caused due to the modulus. The random number generated can never increase pas the value of M. The value of M can be seen as the max value value. The ratio of A to M determines the period of the RNG.

We then looked at testing the method. We first used a correlation plot. We looked at the coefficients $I_0 = 1$,$A = 106$, $C = 1283$, and $M = 6075$.

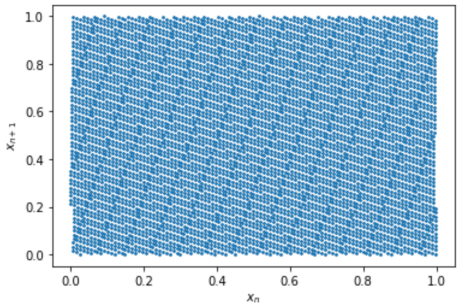

Figure 2: Correlation plot for the first 10^4 pairs using the linear congruential method with the values $I_0 = 1$, $A = 106$, $C = 1283$, and $M = 6075$.

When looking at figure 2, we see that there is a pattern that arises. This pattern is there due to an underlying preference in numbers. With these values we see that RNG has a much larger period but still has cyclical tendencies.

We then looked at the similar correlation plot with coefficients $I_0 = 1$,$A = 107$, $C = 1283$, and $M = 6075$.

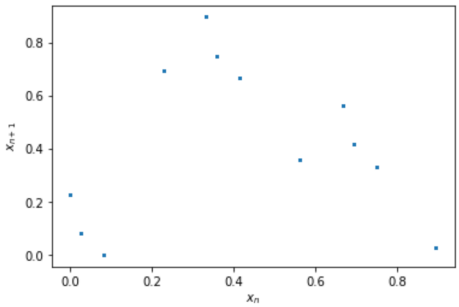

Figure 3: Correlation plot for the first 10^4 pairs using the linear congruential method with the values $I_0 = 1$,$A = 107$, $C = 1283$, and $M = 6075$.

When looking at figure 3, we see that there significantly less plot points compared with figure 2. We see that the change of A from 106 to 107 caused there to be significantly less unique values. This tells

us that for $I_0 = 1$,$A = 107$, $C = 1283$, and $M = 6075$, there are significantly less unique values within this RNG.

We next looked at the similar correlation plot with coefficients $I_0 = 1$,$A = 1103515245$, $C = 12345$, and $M = 32768$.

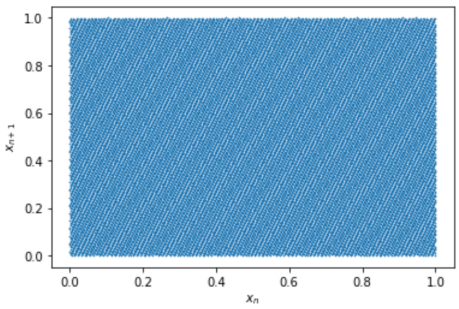

Figure 4: Correlation plot for the first 10^4 pairs using the linear congruential method with the values $I_0 = 1$,$A = 1103515245$, $C = 12345$, and $M = 32768$.

Looking at figure 4 we see that it almost is fully covered. But we can see these lines white lines that go diagonally across. This shows how there is still an underlying pattern to this RNG.

We then looked at the auto-correlation test for independence of the linear congruential method with the values $I_0 = 1$, $A = 106$, $C = 1283$, and $M = 6075$.

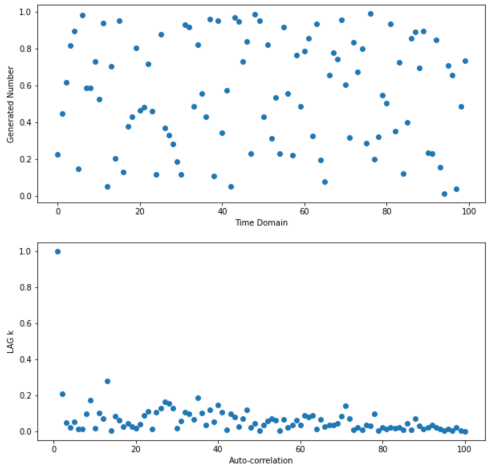

Figure 5: Auto-correlation test for independence for the linear congruential method with values with the values $I_0 = 1$, $A = 106$, $C = 1283$, and $M = 6075$

Using the auto-correlation test we expect the LAG to be equal to zero always if the data set is completely random. When looking at 5 we see that the values aren't all zero. We even notice some parabolic shapes when looking at the top figure where the generated numbers over the time domain are plotted.

We then used the following random number generator in the numpy library to generate a random number between 0 and 1

```
np.random.random()
```

We used to the auto-correlation test on this random number generator

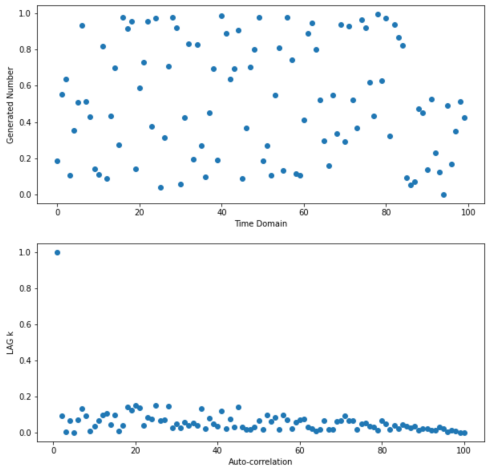

Figure 6: Auto-correlation test for independence for the random function in the numpy library

We see that for the top graph in figure 6 we see that there is no visual patterns within the data values. Although when looking at the bottom graph we see that we have non zero values for LAG meaning that our random number generator is not actually random but yet fixed. We now can see that both the method in the numpy library and our method both are not very random and that these are pseudo-random numbers. This is where the number are usually random enough for you to not notice visually when looking at the generated numbers but when doing an analysis on the output we see that the numbers are actually not random.

We next looked at radiative transfer. We first sampled random optical depths τ, and the angle μ and ϕ.

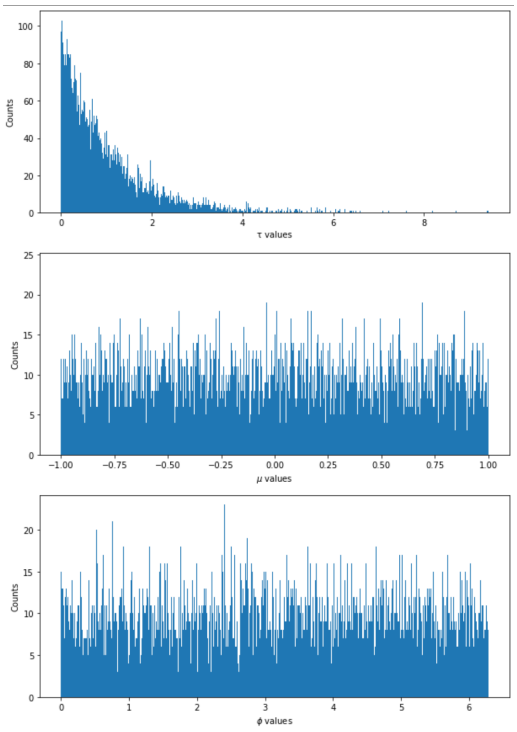

Figure 7: The histogram for sampling random optical depths τ, and the angle μ and ϕ for 10^8 picks

Looking at the top graph for figure 7 we see that the τ value exponentially drop off from $\tau = 0$. Since τ which is seen in equation 9. Looking at the middle graph for figure 7 we see that μ goes from -1 to 1 which is expected. Looking at the bottom graph for figure 7 we see that ϕ goes from 0 to 2π. These graphs are used to make sure the parameters for radiative transfer are being updated properly.

After we ran our code for optical depth $\tau_{max} = 10$ and $N_\mu = 20$ for a pure scattering slab. We then plotted the intensity as a function of exit angle.

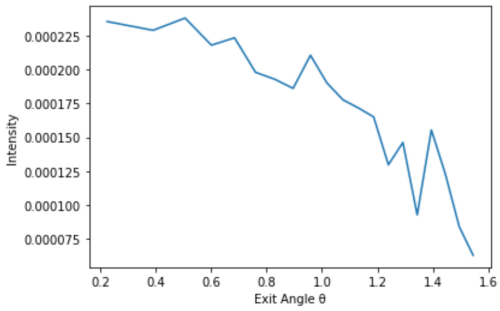

Figure 8: Comparing the intensity as a function of exit angle for optical depth $\tau_{max} = 10$ and $N_\mu = 20$ for a pure scattering slab

We see in figure 8 that as exit angle increases the intensity decreases. We will compare to the theoretical function

$$\bar{I} \approx 0.0244(51.6 - 0.0043\theta^2)$$

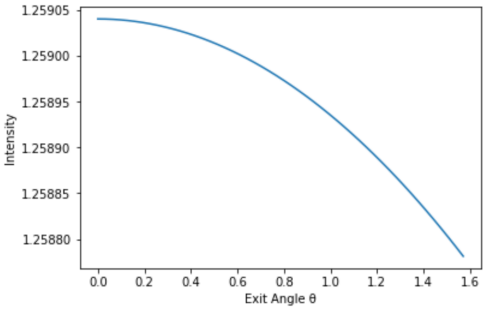

Figure 9: Theoretical function for comparing the intensity as a function of exit angle for optical depth $\tau_{max} = 10$ and $N_\mu = 20$ for a pure scattering slab

Comparing figure 8 to figure 9, we see that we achieved the general shape compared to the theoretical. Thus we can say that our method is quite accurate in calculating the intensity.

Next we looked at a slab with half probability to scatter and half probability to absorb. The intensity as a function of exit angle was

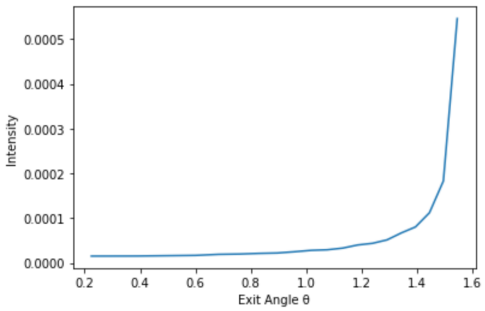

Figure 10: Comparing the intensity as a function of exit angle for optical depth $\tau_{max} = 10$ and $N_\mu = 20$ for a half scattering and half absorbing slab.

We can see that in figure 10 that as exit angle increases that intensity increases exponentially.

We then looked at the Markov Chains looking at the weather on a given day and using a transition matrix to find the probabilities that the forecast will be rainy or sunny after n days.

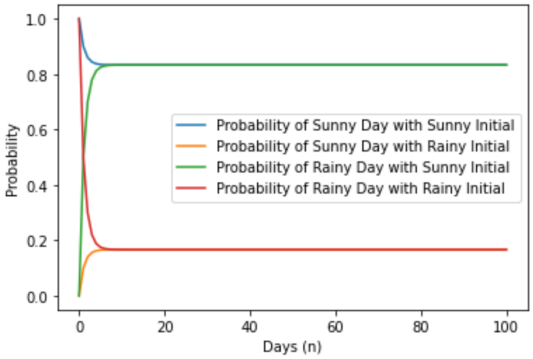

Figure 11: Plotting x^n versus n for $n = 100$ days with initial probability that there is 100% chance it rains and other initial probability that there is 100% chance it will be sunny.

Looking at figure 11 we see that no matter if we have a sunny initial state or a rainy initial state that our probability both collapses to 0.83 and 0.167. We also noticed a burn-in where it takes i certain amount of days to reach the steady state probability. It takes around 10 days to reach the steady state. Even when we look at the transition matrix we see that it eventually also reaches a steady state where the transition matrix does not change after multiplications of itself. The $P_1 00$ or steady state transition matrix was

$$P_{100} = \begin{pmatrix} 0.833 & 0.1667 \\ 0.833 & 0.1667 \end{pmatrix}$$

We see that the matrix values collapsed to the values that we saw in figure 11.

We then plotted probability versus n days with random initial states.

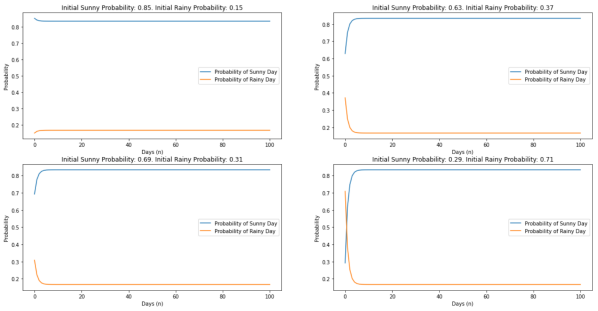

Figure 12: Comparing the probability for n days using random initial states

Looking at figure 12 we see that irrespective of the the initial started state the system eventually reached an equilibrium probability distribution of states of 0.83 and 0.167.

We then looked at the Random-Walk Metropolis where we applied the Metropolis-Hasting algorithm to the function

$$P(x) = \frac{1}{2\sqrt{2\pi}}(sin(5x) + sin(2x) + 2)e^{-x^2} \quad (32)$$

We first started by plotting for $x_0 = -1$ and $\sigma = 0.025, 0, 50$ for 100 iterations

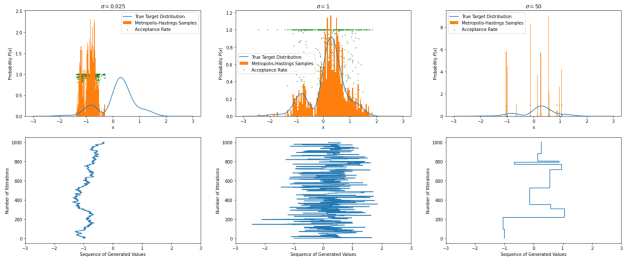

Figure 13: The Metropolis-Hasting algorithm output for $x_0 = -1$ and $\sigma = 0.025, 0, 50$ for 100 iterations for the $P(x)$ in equation 32. The top graph is the probability versus x and the bottom graph is the Sequence of iterations versus the number of iterations

We see that for $\sigma = 0.025$ that the Metropolis-Hasting Samples get stuck in the first hump of the target distribution. This causes it to only sample a small portion of the target distribution. We see that number of iterations for the sequence of generated values is really tight. We see that only values within the first hump get sampled. When looking at the acceptance rate, we see that its all clumped up and hovering over the small hump. When we go to $\sigma = 1$, we saw that the samples form more closely to the target distribution but sometimes overshoot. We see that the acceptance rate plot in the area around the target distribution. When we looking at the sequence of generated values over the number of iterations, we see that there are large horizontal lines going across and they are more dense around the global maxima. With this we can see that the algorithm samples more of the target distribution. When we go to $\sigma = 50$ we see that the samples do not adhere to the target distribution, there's random spikes. We see the acceptance rate is scattered around, sometimes the acceptance rate is within the target distribution. When we look at the sequence of generated values to the number of iterations we see that the there is a significantly smaller amount of samples. It takes significantly larger number of iterations to get a single sample. The samples do vary largely over x and span the whole target distribution.

Next we repeated the same process but for 50,000 iterations

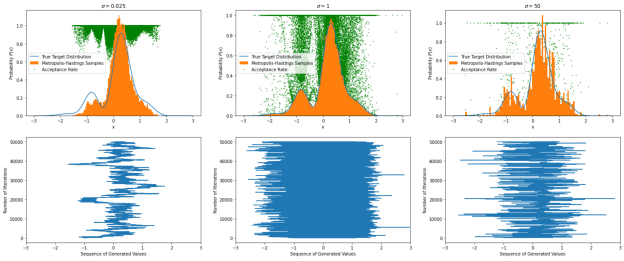

Figure 14: The Metropolis-Hasting algorithm output for $x_0 = -1$ and $\sigma = 0.025, 0, 50$ for 50,000 iterations for the $P(x)$ in equation 32. The top graph is the probability versus x and the bottom graph is the Sequence of iterations versus the number of iterations

When we increased the number of iterations we noticed an overall better fit of our samples to our target distribution. We noticed for $\sigma = 1$ that our samples almost perfectly fit our target distribution and for the sequence of generated values compared to the number of iterations, we notice that we get very dense horizontal lines which mean that our generated samples are almost evenly sampling the whole target distribution. Also as we increased our iterations, the acceptance rate really formed around the target distribution.

We then looked at burn in phase. We started at $x_0 = -3$ with $\sigma = 0.2$ for 1000 iterations.

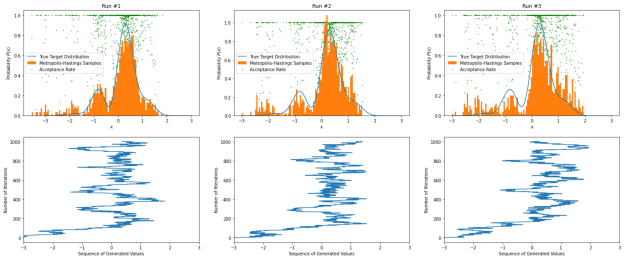

Figure 15: The Metropolis-Hasting algorithm output for $x_0 = -3$ and $\sigma = 0.025, 0, 50$ for 1,000 iterations for the $P(x)$ in equation 32. The top graph is the probability versus x and the bottom graph is the Sequence of iterations versus the number of iterations

We then removed the first 200 iterations and then re plotted the same graphs.

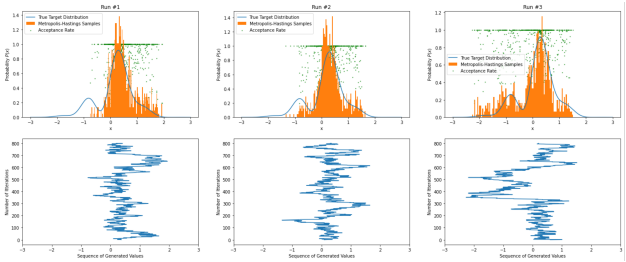

Figure 16: The Metropolis-Hasting algorithm output for $x_0 = -3$ and $\sigma = 0.025, 0, 50$ for 1,000 iterations for the $P(x)$ in equation 32. The first 200 iterations were removed. The top graph is the probability versus x and the bottom graph is the Sequence of iterations versus the number of iterations

Comparing figure 15 to figure 16, we see that figure 15 samples have more emphasis on the values around $x = -3$. When we look at the sequence of generated values for the number of iterations, we see that they all start at $x = -3$ and move towards the middle. By including the burn in period we notice that there is bias in our sampling toward our initial x value. When we remove the burn in period, we exclude the bias in our sampling.

Finally we looked at simulated annealing, we first looked at the min and max of the function

$$f(x, y) = e^{-(x^2+y^2)} \tag{33}$$

We found the global maximum to be $(6.95, 8.65)$ and the minimum to be $(0.0185, -10.26137)$.

When then looked at the function

$$f(x, y) = e^{-(x^2+y^2)} + 2e^{-[(x-1.7)^2+(y-1.7)^2]} \tag{34}$$

We found the global maximum to be $(-17.186, -12.232485)$ and the minimum to be $(-9.25, 4.25)$.

When then looked at the function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{35}$$

We found the global maximum to be $(2.03, 4.08)$ and the minimum to be $(-2.37, 5.89)$.

We were not able to get the global maximum or minimum for the desired functions. Every time we run the program we were getting different values. We weren't able to find the code provided to run the simulated annealing so we created our own code using the concepts provided. We were unsure if it had to do with the temperature scale or if we had a systematic error. But we tried multiple different temperature scales to no avail. We assume we made a systematic error.

4. Conclusion

In conclusion, we first studied the behavior of random numbers generated using the linear congruential method. The parameters I_0, A, C, and M were manipulated to observe the impact on the periodicity and independence of the generated random numbers. The results showed that the period of the random number generator is determined by the ratio of A to M. We also found that the underlying pattern still exists despite a larger period and the change in parameters. The auto-correlation test was also performed to test the independence of the random number generator and found that our random number generator is not actually random but yet fixed. We also analyzed the radiative transfer in a pure scattering slab and a half scattering and half absorbing slab. We found that the intensity decreased as the exit angle increased for the pure scattering slab and increased exponentially as the exit angle increased for the half scattering and half absorbing slab. Moreover, We looked at the weather on a given day and used Markov Chains to find the probabilities of the weather being rainy or sunny after n days. We found that the probabilities eventually reached a steady state and the transition matrix also reached a steady state. Furthermore, the results obtained from the Metropolis-Hasting algorithm applied to the function $P(x)$ showed that the accep-

tance rate and the samples generated were dependent on the value of σ and the number of iterations. When $\sigma = 0.025$, the samples generated were stuck in the first hump of the target distribution and had a low acceptance rate. When $\sigma = 1$, the samples formed closer to the target distribution, but sometimes overshot, and had a higher acceptance rate. When $\sigma = 50$, the samples did not adhere to the target distribution and had a scattered acceptance rate. Increasing the number of iterations led to a better fit of the samples to the target distribution. Then, looking at the The burn-in phase, we found that by including the burn in period we notice that there is bias in our sampling toward our initial x value. When we remove the burn in period, we exclude the bias in our sampling. Finally, we studied the simulated annealing. We found the global maximum to be $(2.03, 4.08)$ and the minimum to be $(-2.37, 5.89)$, However, We were not able to get the global maximum or minimum for the desired functions.

5. References

[1] University of Calgary. Department of Physics and Astronomy. *PHYS581-MC-Lab-2023*. Personal Communication. 2023.

Appendix

Code

```
    # -*- coding: utf-8 -*-
"""Lab 1.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1DK-Ywqa53K_cQUmvbcsCOK3uXeww_Ltk
"""

import numpy as np
import matplotlib.pyplot as plt
import random

"""# RNG"""

class RNG:
  """
  randomly generates number based on parameters
  """
  def __init__(self, seed, A, C, M) -> None:
    self.seed = seed
    self.A = A
    self.C = C
```

```
    self.M = M

  def generate(self, I_0):
    A = self.A
    C = self.C
    M = self.M

    return (A*I_0 + C) % M

  def x_n(self, steps = int(10e4)):
    I = []

    I.append(self.seed)

    for i in range(steps):
      I.append(self.generate(I[-1]))

    return np.array(I)[1:] / self.M


  def correlation(self, steps = int(10e4)):
    I_1 = []
    I_2 = []

    I = self.seed

    for i in range(steps):
      I_1.append(I)
      I = self.generate(I)
      I_2.append(I)

    # divide by x
    x_1, x_2 = np.array(I_1) / self.M , np.array(I_2) / self.M

    return x_1, x_2


def auto_correlation(x):
  # set k values
  k_values = np.arange(0, 100)

  #values for auto correlation
  AC = []
```

```
  for k in k_values:

    #initialize values
    sum_1 = 0
    sum_2 = 0

    x_mean = np.mean(x)

    N = len(x)

    # numerator sum
    t = 1
    while (t < N-k):

      x_t = x[t-1]
      x_tk = x[t+k-1]
      sum_1 += (x_t - x_mean) * (x_tk - x_mean)

      t += 1

    #denominator sum
    t = 1
    while (t < N):
      x_t = x[t-1]
      sum_2 += (x_t - x_mean)**2

      t += 1

    # auto correlation for specific k
    current_AC = sum_1 / sum_2

    AC.append(current_AC)


  return AC

I_0, A, C, M = 3, 7, 0, 10

sequence = []
sequence.append(I_0)

RNG_1 = RNG(I_0, A, C, M)
```

```
for i in range(1000):
  sequence.append(RNG_1.generate(sequence[-1]))

print(sequence[:10])



plt.hist(sequence)
plt.xlabel("unique values")
plt.ylabel("counts")

"""We notice peroidicy in the generated numbers. This is due to the modulus.
The random number generated can never increase past the value of M.
The value of m can be seen as the max valaue.
The ratio of A to M determines the peroid of the rng.
"""

I_0, A, C, M = 1, 106, 1283, 6075

RNG_1 = RNG(I_0, A, C, M)
x_1, x_2 = RNG_1.correlation()

plt.scatter(x_1, x_2, s=0.5)
plt.xlabel("$x_n$")
plt.ylabel("$x_{n+1}$")

I_0, A, C, M = 1, 107, 1283, 6075

RNG_1 = RNG(I_0, A, C, M)
x_1, x_2 = RNG_1.correlation()

plt.scatter(x_1, x_2, s=0.5)
plt.xlabel("$x_n$")
plt.ylabel("$x_{n+1}$")

DI_0, A, C, M = 1, 1103515245, 12345, 32768

RNG_1 = RNG(I_0, A, C, M)
x_1, x_2 = RNG_1.correlation()

plt.scatter(x_1, x_2, s=0.1)
plt.xlabel("$x_n$")
```

```
plt.ylabel("$x_{n+1}$")

I_0, A, C, M = 1, 106, 1283, 6075

RNG_1 = RNG(I_0, A, C, M)
x_1 = RNG_1.x_n(steps = 100)


correlation = auto_correlation(x_1)


fig, axes = plt.subplots(2, 1,figsize=(10,10))

axes[0].scatter(range(len(x_1)), x_1)
axes[0].set_xlabel("Time Domain")
axes[0].set_ylabel("Generated Number")

axes[1].scatter(np.arange(1,101), np.abs(correlation))
axes[1].set_xlabel("Auto-correlation")
axes[1].set_ylabel("LAG k")

x_1 = np.random.random(size=100)

correlation = auto_correlation(x_1)

fig, axes = plt.subplots(2, 1,figsize=(10,10))

axes[0].scatter(range(len(x_1)), x_1)
axes[0].set_xlabel("Time Domain")
axes[0].set_ylabel("Generated Number")

axes[1].scatter(np.arange(1,101), np.abs(correlation))
axes[1].set_xlabel("Auto-correlation")
axes[1].set_ylabel("LAG k")

"""# Radiative Transfer"""

class Photon:
  def __init__(self) -> None:
    self.position = np.zeros(3)
    self.direction = np.zeros(3)

    self.history = []
```

```
    self.history.append(np.copy(self.position))

  def update_x_direction(self, theta, phi):
    return np.sin(theta) * np.cos(phi)

  def update_y_direction(self, theta, phi):
    return np.sin(theta) * np.sin(phi)

  def update_z_direction(self, theta):
    return np.cos(theta)

  def update_direction(self, theta, phi):

    x = self.update_x_direction(theta,phi)
    y = self.update_y_direction(theta,phi)
    z = self.update_z_direction(theta)

    self.direction[0], self.direction[1], self.direction[2] = x, y , z

    return

  def update_position(self, L):

    self.position[0] += L * self.direction[0]
    self.position[1] += L * self.direction[1]
    self.position[2] += L * self.direction[2]

    self.history.append(np.copy(self.position))
    return

class RadiativeTransfer:
  def __init__(self, tau_max, sigma_s, sigma_a, N):

    #initialize first photon
    self.photon = Photon()

    # initialize constants
    self.tau_max = tau_max

    self.sigma_s = sigma_s
    self.sigma_a = sigma_a
    self.prob = sigma_s / (sigma_a + sigma_s)
```

```
  self.z_max = 1

  #for binning
  self.bins = np.linspace(0,1, N+1)
  self.counting_bins = np.zeros(N)
  self.N = N


def optical_depth(self):
  zeta = np.random.random()
  tau_max = self.tau_max

  return - np.log(1 - zeta * (1 - np.exp(-tau_max)))

def theta(self):
  zeta = np.random.random()
  return np.arccos(1-2*zeta)

def phi(self):
  zeta = np.random.random()
  return 2 * np.pi * zeta

def distance_travelled(self, tau):
  z_max = self.z_max
  tau_max = self.tau_max
  return tau * z_max /  tau_max


def out_of_bounds(self):
  photon = self.photon

  if photon.position[2] < 0 or photon.position[2] > self.z_max:
    return True

  else:
    return False

def absorbed(self):
  zeta = np.random.random()

  prob = self.prob

  if prob < zeta:
```

```
    return True

  else:
    return False

def simulate(self):
  loop = 10**5
  i = 1
  empty = False
  # number of itterations
  while(i < loop and empty == False):

    #check to see if photon is in slab
    if self.out_of_bounds() or self.absorbed():


      # check to see if itterations have maxed
      # if maxed stop sending photons

      if (i < loop):
        mu = np.cos(current_theta)
        if mu> 0 and mu < 1:
          self.binning(mu)

        self.photon = Photon()
        i += 1

      else:
        empty = True
        # log $\mu$
        mu = np.cos(current_theta)
        if mu> 0 and mu < 1:
          self.binning(mu)


    # keep scattering
    else:
      current_tau = self.optical_depth()
      current_theta = self.theta()
      current_phi = self.phi()
      current_L = self.distance_travelled(current_tau)

      self.photon.update_direction(current_theta, current_phi)
```

```
        self.photon.update_position(current_L)

    return

  def binning(self, mu):

    counting_bins = self.counting_bins
    bins = self.bins

    for i in range(len(bins)):
      if (mu < bins[i+1]):
        counting_bins[i] += 1
        break

    self.counting_bins = counting_bins
    return

  def energy(self):
    counting_bins = self.counting_bins
    energy = counting_bins / np.sum(counting_bins)

    self.energy_values = energy

    return

  def intensity(self):
    energy = self.energy_values
    counting_bins = self.counting_bins
    bins = self.bins

    mu_i = np.zeros(len(counting_bins))

    for i in range(len(mu_i)):
      mu_i[i] = (bins[i] + bins[i+1]) / 2

    self.mu_i = mu_i

    intensity = energy * self.N/ (2*np.sum(counting_bins)*mu_i)

    self.intensity_values = intensity
    return

a = RadiativeTransfer(10, 1, 0, 1)
```

```
tau_list = []
mu_list = []
phi_list = []

for i in range(10**4):
  tau_list.append(a.optical_depth())
  mu_list.append(np.cos(a.theta()))
  phi_list.append(a.phi())

fig, axes = plt.subplots(3, 1,figsize=(10,15))
axes[0].hist(tau_list, bins= 1000)
axes[0].set_xlabel(" values")
axes[0].set_ylabel("Counts")

axes[1].hist(mu_list, bins=1000)
axes[1].set_xlabel("$\mu$ values")
axes[1].set_ylabel("Counts")

axes[2].hist(phi_list, bins=1000)
axes[2].set_xlabel("$\phi$ values")
axes[2].set_ylabel("Counts")

tau_max = 10
N_mu = 20

sigma_a = 0
sigma_s = 1


pure_scatter = RadiativeTransfer(tau_max, sigma_s, sigma_a, N_mu)
pure_scatter.simulate()
pure_scatter.energy()
pure_scatter.intensity()

plt.plot(np.arccos(pure_scatter.mu_i), pure_scatter.intensity_values)
plt.xlabel("Exit Angle ")
plt.ylabel("Intensity")

theta = np.linspace(0, np.pi/2)

def analytical(theta):
  return 0.0244 * (51.6 - 0.0043 * theta**2)
```

```
plt.plot(theta, analytical(theta))
plt.xlabel("Exit Angle ")
plt.ylabel("Intensity")

tau_max = 10
N_mu = 20

sigma_a = 0.5
sigma_s = 0.5


pure_scatter = RadiativeTransfer(tau_max, sigma_s, sigma_a, N_mu)
pure_scatter.simulate()
pure_scatter.energy()
pure_scatter.intensity()

plt.plot(np.arccos(pure_scatter.mu_i), pure_scatter.intensity_values)
plt.xlabel("Exit Angle ")
plt.ylabel("Intensity")

"""# Markov Chain"""

def forcast(initial_state, n):

  transition_matrix = np.array([[0.9, 0.1],[0.5, 0.5]])
  power = np.linalg.matrix_power(transition_matrix, n)
  return np.matmul(initial_state, power)

sunny_day = np.array([1, 0])
rainy_day = np.array([0, 1])


sunny_day_sunny_day = []
sunny_day_rainy_day = []

rainy_day_sunny_day = []
rainy_day_rainy_day = []

for n in range(101):
  sunny_day_sunny_day.append(forcast(sunny_day, n)[0])
  sunny_day_rainy_day.append(forcast(sunny_day, n)[1])
  rainy_day_sunny_day.append(forcast(rainy_day, n)[0])
  rainy_day_rainy_day.append(forcast(rainy_day, n)[1])
```

```
plt.plot(range(101), sunny_day_sunny_day, label= "Probability of Sunny Day "+
         "with Sunny Initial")
plt.plot(range(101), sunny_day_rainy_day, label="Probability of Sunny Day "+
         "with Rainy Initial")
plt.plot(range(101), rainy_day_sunny_day, label="Probability of Rainy Day "+
         "with Sunny Initial")
plt.plot(range(101), rainy_day_rainy_day, label="Probability of Rainy Day "+
         "with Rainy Initial")
plt.xlabel("Days (n)")
plt.ylabel("Probability")
plt.legend()


P_matrix = np.linalg.matrix_power(np.matrix([[0.9,0.1],[0.5,0.5]]),10000)
print("P_100 matrix is: \n",P_matrix)

data = []
for i in range(4):
  sun_initial = np.random.random()
  rain_initial = 1 - sun_initial

  state = np.array([sun_initial, rain_initial])

  prob_sunny = []
  prob_rain = []

  for i in range(101):
    prob_sunny.append(forcast(state, i)[0])
    prob_rain.append(forcast(state, i)[1])

  data.append([state, prob_sunny, prob_rain])

fig, axes = plt.subplots(2,2, figsize=(20,10))

for i in range(2):
  axes[0][i].plot(range(101), data[i][1], label = "Probability of Sunny Day")
  axes[0][i].plot(range(101), data[i][2], label = "Probability of Rainy Day")
  axes[0][i].set_xlabel("Days (n)")
  axes[0][i].set_ylabel("Probability")
  axes[0][i].legend()
  axes[0][i].title.set_text("Initial Sunny Probability: " +
                            str(round(data[i][0][0], 2)) +
```

```
                            ". Initial Rainy Probability: " +
                            str(round(data[i][0][1], 2)))

for i in range(2):
  axes[1][i].plot(range(101), data[i+2][1], label = "Probability of Sunny Day")
  axes[1][i].plot(range(101), data[i+2][2], label = "Probability of Rainy Day")
  axes[1][i].set_xlabel("Days (n)")
  axes[1][i].set_ylabel("Probability")
  axes[1][i].legend()
  axes[1][i].title.set_text("Initial Sunny Probability: " +
                            str(round(data[i+2][0][0], 2)) +
                            ". Initial Rainy Probability: " +
                            str(round(data[i+2][0][1], 2)))

class Metropolis_Hasting:
  def __init__(self, target_pdf, sigma) -> None:
    self.target_pdf = target_pdf
    self.sigma = sigma
    self.acceptance_data = []

  def proposal_pdf(self, current_x):
    sigma = self.sigma
    return random.normalvariate(current_x, sigma)

  def acceptance(self, current_x, proposed_x, target_pdf, proposal_pdf):
    return min(1, target_pdf(proposed_x) / target_pdf(current_x))

  def run(self, x_1, max_itteration, output_ignore):
    samples = []
    samples.append(x_1)
    acceptance_data = []

    for i in range(1, max_itteration):
      current_x = samples[-1]
      y = self.proposal_pdf(current_x)
      alpha = self.acceptance(current_x, y, self.target_pdf, self.proposal_pdf)
      acceptance_data.append(alpha)
      u = np.random.uniform(low = 0, high = 1)
      if u < alpha:
        samples.append(y)
      else:
        samples.append(current_x)
    self.acceptance_data = acceptance_data[output_ignore:]
```

```
    return samples[output_ignore:]

def target_pdf(x):
  return 1/(2 * np.sqrt(np.pi)) * (np.sin(5 * x) + np.sin(2 * x) + 2) *\
   np.exp(-x**2)

example_25 = Metropolis_Hasting(target_pdf, 0.025)
data_25 = example_25.run(x_1=-1, max_itteration=1000, output_ignore=0)


example_1 = Metropolis_Hasting(target_pdf, 1)
data_1 = example_1.run(x_1=-1, max_itteration=1000, output_ignore=0)

example_50 = Metropolis_Hasting(target_pdf, 50)
data_50 = example_50.run(x_1=-1, max_itteration=1000, output_ignore=0)


fig, axes = plt.subplots(2,3, figsize=(25,10))
data = [data_25,data_1, data_50]
examples = [example_25, example_1, example_50]
sigma = [0.025, 1, 50]

for i in range(3):
  axes[0][i].plot(np.linspace(-3,3, 200), target_pdf(np.linspace(-3,3, 200)),
                label="True Target Distribution")
  axes[0][i].hist(data[i], bins=100, density=True, label="Metropolis-Hastings Samples")
  axes[0][i].scatter(data[i][:-1], examples[i].acceptance_data, color = "green",s=1 ,
                     label = "Acceptance Rate")
  axes[0][i].set_ylabel("Probability $P(x)$")
  axes[0][i].set_xlabel("x")
  axes[0][i].title.set_text("$\sigma = $" + str(sigma[i]))
  axes[0][i].legend()

  axes[1][i].plot(data[i], range(len(data[i])))
  axes[1][i].set_xlabel("Sequence of Generated Values")
  axes[1][i].set_ylabel("Number of Itterations")
  axes[1][i].set_xlim(-3,3)

"""Increased Itterations from 1,000 to 50,000"""

example_25 = Metropolis_Hasting(target_pdf, 0.025)
data_25 = example_25.run(x_1=-1, max_itteration=50000, output_ignore=0)
```

```
example_1 = Metropolis_Hasting(target_pdf, 1)
data_1 = example_1.run(x_1=-1, max_itteration=50000, output_ignore=0)

example_50 = Metropolis_Hasting(target_pdf, 50)
data_50 = example_50.run(x_1=-1, max_itteration=50000, output_ignore=0)


fig, axes = plt.subplots(2,3, figsize=(25,10))
data = [data_25,data_1, data_50]
examples = [example_25, example_1, example_50]
sigma = [0.025, 1, 50]

for i in range(3):
  axes[0][i].plot(np.linspace(-3,3, 200), target_pdf(np.linspace(-3,3, 200)),
                label="True Target Distribution")
  axes[0][i].hist(data[i], bins=100, density=True, label="Metropolis-Hastings Samples")
  axes[0][i].scatter(data[i][:-1], examples[i].acceptance_data, color = "green",s=1
                    ,label = "Acceptance Rate")
  axes[0][i].set_ylabel("Probability $P(x)$")
  axes[0][i].set_xlabel("x")
  axes[0][i].title.set_text("$\sigma = $" + str(sigma[i]))
  axes[0][i].legend()

  axes[1][i].plot(data[i], range(len(data[i])))
  axes[1][i].set_xlabel("Sequence of Generated Values")
  axes[1][i].set_ylabel("Number of Itterations")
  axes[1][i].set_xlim(-3,3)

"""Burn in phase"""

example_25 = Metropolis_Hasting(target_pdf, 0.2)
data_25 = example_25.run(x_1=-3, max_itteration=1000, output_ignore=0)


example_1 = Metropolis_Hasting(target_pdf, 0.2)
data_1 = example_1.run(x_1=-3, max_itteration=1000, output_ignore=0)

example_50 = Metropolis_Hasting(target_pdf, 0.2)
data_50 = example_50.run(x_1=-3, max_itteration=1000, output_ignore=0)


fig, axes = plt.subplots(2,3, figsize=(25,10))
```

```
data = [data_25,data_1, data_50]
examples = [example_25, example_1, example_50]
sigma = [0.2, 0.2, 0.2]

for i in range(3):
  axes[0][i].plot(np.linspace(-3,3, 200), target_pdf(np.linspace(-3,3, 200)),
                label="True Target Distribution")
  axes[0][i].hist(data[i], bins=100, density=True, label="Metropolis-Hastings Samples")
  axes[0][i].scatter(data[i][:-1], examples[i].acceptance_data, color = "green",s=1 ,
                    label = "Acceptance Rate")
  axes[0][i].set_ylabel("Probability $P(x)$")
  axes[0][i].set_xlabel("x")
  axes[0][i].title.set_text("Run #" + str(i+1))
  axes[0][i].legend()

  axes[1][i].plot(data[i], range(len(data[i])))
  axes[1][i].set_xlabel("Sequence of Generated Values")
  axes[1][i].set_ylabel("Number of Itterations")
  axes[1][i].set_xlim(-3,3)

"""Ignore Initial 200 Values"""

example_25 = Metropolis_Hasting(target_pdf, 0.2)
data_25 = example_25.run(x_1=-3, max_itteration=1000, output_ignore=200)



example_1 = Metropolis_Hasting(target_pdf, 0.2)
data_1 = example_1.run(x_1=-3, max_itteration=1000, output_ignore=200)

example_50 = Metropolis_Hasting(target_pdf, 0.2)
data_50 = example_50.run(x_1=-3, max_itteration=1000, output_ignore=200)



fig, axes = plt.subplots(2,3, figsize=(25,10))
data = [data_25,data_1, data_50]
examples = [example_25, example_1, example_50]
sigma = [0.2, 0.2, 0.2]

for i in range(3):
  axes[0][i].plot(np.linspace(-3,3, 200), target_pdf(np.linspace(-3,3, 200)),
                                label="True Target Distribution")
  axes[0][i].hist(data[i], bins=100, density=True, label="Metropolis-Hastings Samples")
  axes[0][i].scatter(data[i][:-1], examples[i].acceptance_data, color = "green",s=1 ,
```

```
                        label = "Acceptance Rate")
  axes[0][i].set_ylabel("Probability $P(x)$")
  axes[0][i].set_xlabel("x")
  axes[0][i].title.set_text("Run #" + str(i+1))
  axes[0][i].legend()

  axes[1][i].plot(data[i], range(len(data[i])))
  axes[1][i].set_xlabel("Sequence of Generated Values")
  axes[1][i].set_ylabel("Number of Itterations")
  axes[1][i].set_xlim(-3,3)

"""
Simulated Annealing
"""
class Simulated_Annealing:
  def __init__(self, target_function, sigma) -> None:
    self.target_function = target_function
    self.sigma = sigma

  def proposal_pdf(self, current_x):
    sigma = self.sigma
    return random.normalvariate(current_x, sigma)

  def acceptance(self, delta_E, T):
    return min(1, np.exp(-delta_E/ T))

  def cooling_schedule(self, t, T_0, T_f, n):
    return max(T_f, T_0 * (T_f / T_0) ** (t / n))

  def run(self, x_0, y_0, max_itteration, T_0, T_f, time_itterations):
    current_x = x_0
    current_y = y_0

    t = 0
    T = self.cooling_schedule(t, T_0, T_f, time_itterations)
    while T > T_f:

      for i in range(int(max_itteration)):
        x_w = self.proposal_pdf(current_x)
        y_w = self.proposal_pdf(current_y)

        #cost function
        delta_E = self.target_function(x_w, y_w) - self.target_function(current_x, current_y)
```

```
        if delta_E <= 0:
          current_x = x_w
          current_y = y_w

        else:
          r = np.random.random()
          if r <= self.acceptance(delta_E, T):
            current_x = x_w
            current_y = y_w


      t += 1
      T = self.cooling_schedule(t, T_0, T_f, time_itterations)

    return current_x, current_y

def function_1(x, y):
  return np.exp(-(x**2 + y**2))

def function_1_min(x, y):
  return -np.exp(-(x**2 + y**2))

def function_2(x, y):
  return np.exp(-(x**2 + y**2)) + 2 * np.exp(-((x-1.7)**2 + (y-1.7)**2))

def function_2_min(x, y):
  return -np.exp(-(x**2 + y**2)) + 2 * np.exp(-((x-1.7)**2 + (y-1.7)**2))

def function_3(x, y):
  return (1-x)**2 + 100 * (y-x**2)**2

def function_3_min(x,y):
  return -(1-x)**2 + 100 * (y-x**2)**2

#function 1
max_function_1 = Simulated_Annealing(function_1, 0.5)
max_x_function_1, max_y_function_1 = max_function_1.run(0, 0, 100, 2, 1, 10)

min_function_1 = Simulated_Annealing(function_1_min, 0.5)
min_x_function_1, min_y_function_1 = min_function_1.run(0, 0, 100, 2, 1, 10)
```

```
#function 2
max_function_2 = Simulated_Annealing(function_2, 0.5)
max_x_function_2, max_y_function_2 = max_function_2.run(0, 0, 100, 2, 1, 10)

min_function_2 = Simulated_Annealing(function_2_min, 0.5)
min_x_function_2, min_y_function_2 = min_function_2.run(0, 0, 100, 2, 1, 10)

#function 3
max_function_3 = Simulated_Annealing(function_3, 0.5)
max_x_function_3, max_y_function_3 = max_function_3.run(0, 0, 100, 2, 1, 10)

min_function_3 = Simulated_Annealing(function_3_min, 0.5)
min_x_function_3, min_y_function_3 = min_function_3.run(0, 0, 100, 2, 1, 10)

#function 1
print("function 1")
print(max_x_function_1, max_y_function_1)
print(min_x_function_1, min_y_function_1)

#function 2
print("function 2")
print(max_x_function_2, max_y_function_2)
print(min_x_function_2, min_y_function_2)

#function 3
print("function 3")
print(max_x_function_3, max_y_function_3)
print(min_x_function_3, min_y_function_3)
```