Monte Carlo Simulations - The Art of Exploiting Randomness

Candidate Number: 229538

Word Count: 4558

Contents

1. R Version and Required Packages

1.1. R Version

When starting this project, I was using R version 4.0.3 (Bunny-Wunnies Freak Out) and all scripts work on this version. The only minor issue is that as R was updated, the use of packages throws a warning saying that the packages are built on a newer version of R. As a result, on 02/05/2021, I updated to R version 4.0.5 (Shake and Throw). This had no impact on the running of my developed Monte Carlo Simulations. I wrote the code within PyCharm on a Windows machine. I also tried executing the scripts within R's GUI and this worked as expected - the main difference is that all the code must be pasted in rather than encapsulating certain functions in their own file as I have written them. It is worth noting that for my figures, I changed the font of the plots to be 'serif' which may be dependent on the machine that the script is being run on. If the fonts changes are causing problems, you can comment out the following line in the scripts without affecting the results.

```
theme(text = element_text(size = 18,
    family = "serif"))
```

1.2. R Packages

The following packages are required to be installed if not already.

- Matrix - allows for easy initialisation of transition matrix and calculation of invariant distribution
- ggplot2 - allows plots with increased flexibility
- tictoc - allows for naive execution timing. I used this for my Poisson generator
- plyr - tools for splitting, applying and combining data. Used within my Chi-squared test

The following packages are extra and are not required to run the MCMC scripts

- markovchain - plots the transition matrix as a graph
- KSgeneral - this is an optional package. I originally explored the use of a discrete KS test for my goodness-of-fit test and as such, I have included my code. This is, however, still optional as the goodness-of-fit test I decided to present here is the Chi-squared test which does not require this package.
- lattice latticeExtra - These are used to create the rootogram plots found in my appendix

2. Generation of (Pseudo-) Random Numbers

The first step in all Monte Carlo methods is the generation of pseudo-random numbers from an arbitrary distribution. Whilst most programming languages offer this through builtin functions, we are able to replicate this idea from first principles, often through the transformation of uniformly distributed random variables in the interval [0,1]. To demonstrate this, I have chosen the discrete Poisson distribution which can be used to "describe the distribution of rare events in a large population" [28]. Interestingly, the classical example of a Poisson distribution is Bortkewitsch's analysis of 'deaths by horsekick' within the Prussian army [7]. Since then, however, the Poisson distribution has been applied to a range of different subject areas including rate of cell mutation [28], as well as more generally within Poisson processes where the Poisson distribution models the arrival times of an event of interest. The Poisson distribution has the following probability mass function (PMF):

$$\Pr(X = x) = \frac{\lambda^k e^{-\lambda}}{k!} \qquad (k \in \mathbb{N}_0) \tag{1}$$

Our parameter λ is the expected value of our random variable X as well as its variance. As a result, we can interpret this λ as the 'rate of occurrence' for a given event. k is our functions support and can be thought of as the number of occurrences of the given event. Due to the importance of the Poisson distribution, there are a plethora of algorithms to generate random variables such that $X \sim \text{Pois}(\lambda)$. Our algorithm implementation relies heavily on our choice of λ; I found it common for languages to implement a few different algorithms - one for low λ values and another for high λ values and sometimes one for fractional λ values [17].

2.0.1 Computational Considerations for Poisson Variable Generation

There are inherent problems with generating Poisson variables computationally that involve numerical stability and rounding errors. Firstly, $e^{-\lambda}$ can suffer from underflow as $\lambda \to \infty$ and conversely, $k!$ and λ^k can quickly lead to overflow. As a result, it is recommended to implement a numerically stable but mathematically equivalent PMF for computational implementations as follows [1]:

$$F(k, \lambda) = \exp(k \ln \lambda - \lambda - \ln \Gamma(k + 1)) \tag{2}$$

Where Γ is the gamma function. I will begin with the simpler example where $\lambda < 60$.

2.1. Generating Poisson Variables: $\lambda < 60$

The book by Devroye [9] is an excellent resource for a range of algorithms to generate Poisson random variables. For the majority of simple Poisson algorithms presented by Devroye, algorithms grow linearly with respect to λ - here we can use the 'big O' notation to denote this $O(\lambda)$. The simplicity of these algorithms and their time complexity make them ideal for small values of λ and as a result, I chose to compare two different $O(\lambda)$ algorithms. I decided to explore this because Hörmann [14] considered the number of uniform variables within the algorithm as a means of understanding algorithmic efficiency. Here I compare one algorithm that repeatedly generates uniform random variables to generate a Poisson variable whereas my second algorithm only generates a single uniform variable.

2.1.1 Poisson Generator: Multiplication of Uniform Random Variables

Devroye presents the following Lemma:

Let $U_1, U_2, \ldots$ be i.i.d uniform [0,1] random variables. Let X be the smallest integer such that

$$\prod_{i=1}^{X+1} U_i < e^{-\lambda}$$

Then X is Poisson (λ)

In other words, we can generate a Poisson random variable by comparing the cumulative product of uniform random variables with the constant $e^{-\lambda}$. This was an effective method for small values of λ but despite having time complexity $O(\lambda)$, it was outperformed by similar $O(\lambda)$ algorithms when the number of iterations became larger ($>$ 100,000 iterations).

2.1.2 Poisson Generator: Inversion by Sequential Search

This algorithm, despite also being $O(\lambda)$, is significantly faster than the previous method; I believe this is due to the fact that the previous method required the repeated generation of new uniform random variables which impacts the acceptance probability. The following method only requires a single uniform variable to be generated and is based on the following recurrence relation also presented by Devroye:

$$\frac{P(X = i + 1)}{P(X = 1)} = \frac{\lambda}{i + 1} \quad (i \geq 0)$$

Here we compare a single uniform random variable with the cumulative product of the constant $e^{-\lambda}$ and the ratio $\frac{\lambda}{i+1}$. Devroye does mention potential speed-ups of the sequential search algorithm by seeding the search with the Poisson's mode; whilst this does lead to a potential time complexity of $O(\sqrt{\lambda})$, the algorithms space complexity massively increases because we must pre-compute and store a table containing $P(X \leq \lambda)$. This lookup table method has seen a few variations however as pointed out by C.D Kemp, these methods become impractical for changing values of λ [17]. I tried implementing a naive version of this type of algorithm in R but the language lacks versatile hashmaps which are required to access each value in the table with complexity $O(1)$ and as a result, the algorithm ran slower than my naive sequential search.

2.1.3 Comparing Performance between Sequential Search and Uniform Multiplication

Using the *tictoc* library, I was able to record an approximate run time for these algorithms on my local machine - for reference, my desktop it relatively powerful with 6 CPU cores at 5.0 GHz and has 32GB of RAM. The script I created generated one million Poisson samples with parameter $\lambda = 29$ and was ran 5 times to calculate an average. The rationale behind $\lambda = 29$ was that this was the original cutoff between my naive Poisson algorithm and my efficient inversion algorithm and as such, was a worst-case scenario for my naive algorithms. The first algorithm which repeatedly multiplies uniforms took 33.026s ± 0.67 whereas the sequential search algorithm took 2.796s ± 0.031. To further illustrate the inefficiencies of the first method, I created a short script called **uniform_generation_counter.R** to count the number of uniforms required for each algorithm. With a sample size $N = 100,000$ and $\lambda = 15$, I ran my script 5 times to calculate an average. The sequential search produced 100,000 uniform random variables meaning 1 per Poisson variable. The multiplication of uniform random variables, however, produced 1,600,094 uniforms on average which equates roughly to 16 per Poisson variable.

2.2. Poisson Variable Generation: $\lambda \geq 60$

Because of the prevalence of Poisson distributions, there already exists a range of different 'uniformly fast generators' - this means that the algorithms scale independently of λ, thus giving a $O(1)$ time complexity. Whilst the sequential search algorithm performed well up to and exceeding $\lambda = 500$, the slowdown in algorithm performance was noticeable. To overcome this, many programming languages implement a second $O(1)$ algorithm for larger λ values. NumPy is a commonly used mathematical library for the language Python - their implementation of a Poisson generator has the cutoff between Poisson algorithms at $\lambda \geq 10$ [26]. Hörmann [14] was a valuable resource in understanding the differences between a range of high-performance Poisson generators. It compared algorithms developed by Kemp and Kemp [18], Ahrens and Dieter [4], [5] and Schmeiser and Kachitvichyanukul [15] with two of their own algorithms. A common theme for the fastest $O(1)$ Poisson generators is that their implementation is long and cumbersome; as a result, I decided to implement Hörmann's Poisson Transform Rejection with Squeeze (PTRS) algorithm because it had comparable performance to their faster Transform Rejection with Decomposition (PTRD) algorithm but was simpler to implement. One other important consideration raised by Hörmann is that the PTRS algorithm can be safely used for $10 \leq \lambda \leq 10^7$ which was more than sufficient for our needs. Whilst Hörmann offers an extensive proof of concept in his paper, the reason for these bounds on λ is due to the fact that the transformed rejection algorithm relies on an approximation of the inverse distribution function which becomes less stable at high values of λ.

2.3. My Poisson Generator

Similar to other popular programming languages, I decided to encapsulate both algorithms within a single 'Poisson Generator' which can be found in **poisson_generator.R**. For my implementation, I found that $\lambda \geq 60$ was a suitable cutoff between my sequential search algorithm and the PTRS algorithm. Figure 1 and 2 illustrates both of my implemented algorithms and how they compare the R's builtin **rpois** function. I have utilised R's *tictoc* library to give an intuition into how fast these algorithms are - it is especially important when illustrating the speed of the PTRS algorithm. These plots can be produced using the **Poisson GGPLOT.R** script.

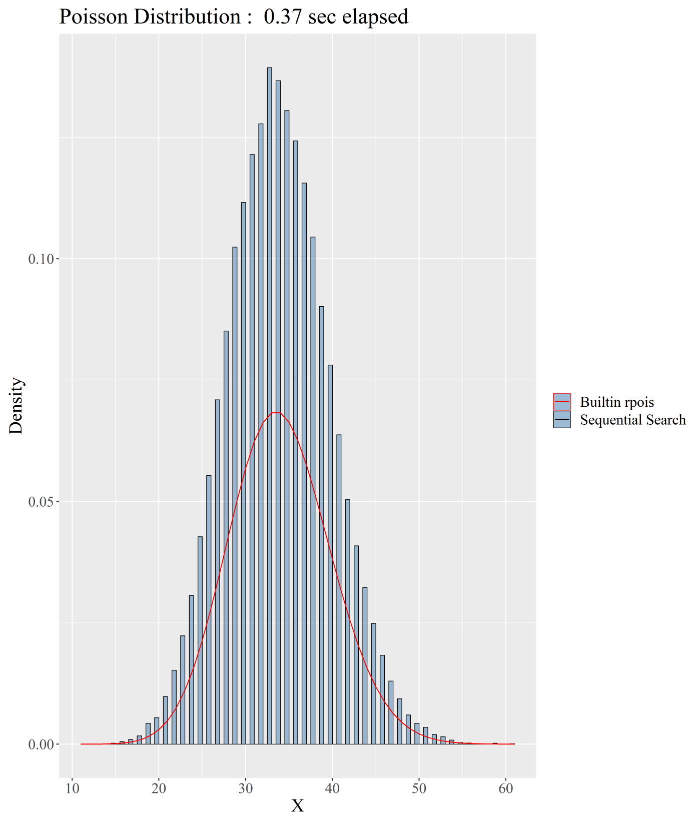

Figure 1. A plot to compare R's builtin Poisson generator with my own. N = 100,000. This is a sample of Pois(34) meaning that my sequential search algorithm is being used. We can see that the builtin **rpois** is very similar to my generated variates.

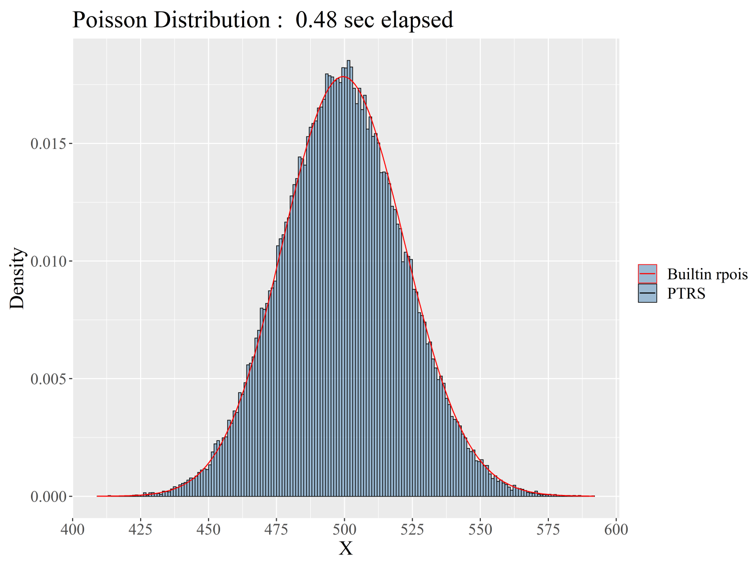

Figure 2. A plot to compare R's builtin Poisson generator with my own. N = 100,000. This is a sample from Pois(500) which will use my implementation of the PTRS algorithm. We can see that despite having a λ almost double that in Figure 1, the run time is virtually identical. We can see that my PTRS algorithm produced variates exceptionally close to the builtin **rpois** function.

3. Goodness-of-Fit Tests

Whilst Figure 1 and 2 do suggest a close relationship between my random variates and the builtin **rpois**, to quantitatively measure the similarity we need to perform a 'Goodness-of-fit' test. Because my distribution is discrete, the typical test for goodness-of-fit is the Chi-squared ($\mathcal{X}^2$) test which is a measure of how close your observed data is from the expected data. For my own learning, however, I wanted to explore other options for assessing goodness-of-fit as well as the Chi-squared test - this lead me to explore the Kolmogorov-Smirnov (KS) test, a test that is classically based on the assumption that the distributions of interest are continuous. There are, however, papers that explore how the KS test can be adapted for discrete and grouped data [23] and crucially, there are in fact some recent developments within R that have begun to incorporate a discrete version of the language's builtin **ks.test** [6], [11]. I decided to primarily use Chi-squared due to the inherest flaws of the discrete KS test, however, I have included my code for completeness - **Kolmogorov-Smirnov Test.R** and **Poisson KS Visualisation.R**.

3.1. Chi-Squared Tests

Chi-squared is a very popular familiy of statistical tests. In R, this has been implemented as the **chisq.test** and it allows for both goodness-of-fit as well as statistical independence testing. The calculation of the Chi-squared statistic is as follows:

$$\mathcal{X}^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Where O and E are the observed and expected counts for the generated variates. The two major assumptions with this test is that the observations are independent and that calculated frequency of the 'expected' column is at least 5 [20]. This presented a problem for our application of a goodness-of-fit test because it means that we will have to 'bin' some of the variates in order to ensure the second assumption is satisfied. I could not derive a robust method for automating this binning process and as a result, I decided to naively remove all random variates that had an expected frequency of 5 or less. This is of course not ideal, however the percentage of columns with a frequency less than 5 was negligible in the overall sample. The observed counts can be found easily with R's *plyr* package using its **count** function. To generate the expected counts, however, we must use the Poisson's PMF. This gives us a probability for each unique variate which we can then multiple by our sample size N to attain our expected frequencies. For a Chi-squared test, our null hypothesis is that our data is consistent with the specified distribution. If P is the distribution of our sample and P_0 is our reference Poisson distribution using **rpois**, we can state this as the following:

$$H_0 : P = P_0$$

$$H_1 : P \neq P_0$$

As with the majority of statistical tests, I will set my confidence level $\alpha = 0.05$. To mirror the plots I presented above, I set the sample size to $N = 100,000$ and performed a Chi-squared test at both $\lambda = 34$ and $\lambda = 500$ to ensure both of my algorithms were performing as expected. Using a Chi-squared test, my sequential search and PTRS algorithms produced a p-value of 0.85 and 0.8 respectively, which is significantly higher than my 0.05 confidence level - as a result we fail to reject our null hypothesis and thus, I am confident that my Poisson generator was producing random variables according to the specified distribution. My Chi-sqaured test is encapsulated within the **Chi_Squared_Test.R** script and is utilised within my **Poisson GGPLOT.R** script.

3.1.1 Rootograms

Whilst not strictly required, and certainly not as rigorous, visual methods can sometimes be a nice addition to reinforce goodness-of-fit tests. I have attached in the appendix Figures 14 and 15 where I have used the *extraLattice* package to develop some simple rootogram plots, originally a technique developed by Tukey [30]. This allows for a visual comparison of different values for λ to see how well my Poisson generator can produce the expected sample. Figure 14 illustrates my sequential search algorithm whereas the PTRS algorithm is illustrated in Figure 15. The code for these plots can be found in **GoF Plotting.R**.

4. Markov Chains and Markov Chain Monte Carlo

4.1. Introduction to Markov Chain Theory

In order to introduce the theory supporting Markov chains before exploring their role in Monte Carlo simulations, I referred to the following material [16], [19], [24], [25], [27]. Markov chains and their application within Markov chain Monte Carlo (MCMC) comprise a collection of statistical algorithms that can be used to sample from a distribution of interest. Due to their versatility, MCMC techniques are utilised in a wide range of different fields; Diaconis suggests that any area of science is likely to have literature utilising MCMC methods [10].

A stochastic process is said to be Markovian when its future is independent of the past, given the present [27]. This is known as the Markov property and is defined as the following:

$$\Pr(X_{t+1} = x_{t+1} | X_0 = x_0, ..., X_t = x_t) = \Pr(X_{t+1} = x_{t+1} | X_t = x_t)$$

A *discrete-time Markov chain* can be defined as a collection of random variables $X_0, X_1, X_2 ...$, where each X_t takes a value in a finite set of states $\mathcal{S}$. With each discrete time step $t = 0, 1, 2, ...$, we say that X_t is in state i if $X_t = i$. Given $X_t = i$, the probability that $X_{t+1} = j$ is the one-step transition probability and is defined as:

$$P_{i,j} = \Pr(X_{t+1} = j | X_t = i)$$

This can be generalised to n-step transition probabilities as:

$$P^n_{i,j} = \Pr(X_{t+n} = j | X_t = i)$$

Because we have a countable state space, we can naturally represent our transition probabilities as an $\mathcal{S} \times \mathcal{S}$ transition matrix denoted P.

$$P = P_{i,j} = \begin{pmatrix} P_{0,0} & P_{0,1} & ... & P_{0,j} & ... \\ P_{1,0} & P_{1,1} & ... & P_{1,j} & ... \\ \vdots & \vdots & \ddots & \vdots & \ddots \\ P_{i,0} & P_{i,1} & ... & P_{i,j} & ... \\ \vdots & \vdots & \ddots & \vdots & \ddots \end{pmatrix}$$

When our transition probabilities are independent of our time step t, we refer to these Markov chains as *homogeneous*. A chain is *irreducible* if any state in $\mathcal{S}$ is reachable from any other state in $\mathcal{S}$. In other words, for any pair of states $i, j \in \mathcal{S}$, there exists a non-zero probability of transitioning from state i to state j in n steps. An irreducible Markov chain has period D, where D is the greatest common divisor of $\{n \geq 1 : P^n_{i,i} > 0\}$ - if $D = 1$, then the chain is said to be *aperiodic*. A quick way to verify aperiodicity is to see if the transition matrix has some non-zero probabilities on the main diagonal. Unless explicitly specified, all Markov chains discussed further will be homogeneous, irreducible and aperiodic chains.

4.1.1 The Invariant Distribution

If we run our Markov chains for a sufficiently long time, there will be a $\pi_i \simeq \Pr(X_t = i)$ that is independent of the initial distribution. We can intuitively interpret this as the the number of times our chain equalled state i over all discrete-time steps t as t approaches infinity.

$$\pi_i = \lim_{t\to\infty} \frac{\sum_{n=1}^{t}(X_n = i)}{t}$$

In fact, for every pair $i, j \in \mathcal{S}$ there exists the limit $\lim_{n\to\infty} P^n_{i,j}$ that is independent of i and is often denoted as π_j. Further to this, if $\mathcal{S}$ is countable, we can write:

$$\sum_{j\in\mathcal{S}} \pi_j = 1 \text{ and } \sum_{i\in\mathcal{S}} \pi_i P_{i,j} = \pi_j \quad (\forall j \in \mathbf{S})$$

If this is satisfied, we can describe the Markov chain as a ”positive recurrent” chain where π is our *equilibrium* or *steady-state* distribution. This means that if X_0 has distribution π, X_t will also have distribution π for every time-step t. In addition, our chain is said to be *reversible* if it satisfies detailed balance with respect to our invariant distribution

$$\pi_i p_{i,j} = \pi_j p_{j,i}$$

4.2. Implementation of a Markov Chain Monte Carlo Simulation in R

Our task was to implement a system that is characterised by three independent states defined as $\mathcal{S} =$

$\{A, B, C\}$. To illustrate this hypothetical system, we can use a weighted, directed graph defined $G = (V, E)$ where V and E are the set of vertices and edges respectively (Figure 3). With each node of the graph representing a discrete state in $\mathcal{S}$, the edges connecting the vertices are weighted with the respective probability of transitioning from the current state to a viable future state. Because we are representing a relatively simple system with this model, we can quickly verify the irreducibility of the system by seeing that in Figure 3, any state is able to reach any other state. Because we have a finite state space, we can naturally represent our graph as a 3×3 transition matrix; here we can also verify that our Markov chain is aperiodic if the chain has at least one non-zero probability on the main diagonal of our transition matrix.

$$P = \begin{pmatrix} 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 1/2 \\ 0 & 1 & 0 \end{pmatrix}$$

Before we can confidently assess the validity of our Monte Carlo model, we need to first calculate the theoretical invariant distribution (π) for our transition matrix P. In appendix B, I have attached my calculation of the invariant distribution however, we can also solve this using R's **solve** function for a set of linear equations - I created the function **inv_solver** that automatically calculates the invariant distribution for a transition matrix. For this transition matrix, the calculated invariant distribution is $\pi_A = 0.3$, $\pi_B = 0.4$ and $\pi_C = 0.3$. With this, we now have a benchmark to determine the accuracy of our Monte Carlo simulation.

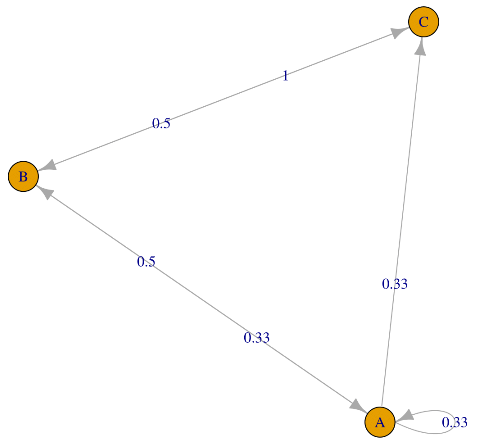

Figure 3. A graphical representation of the transition matrix for my 3-state system. This was made using the *markovchain* package for R.

4.2.1 The Metropolis-Hastings Algorithm

The Metropolis algorithm [21] is perhaps the most straightforward MCMC algorithm to understand and implement. This technique generalises well and can provide a good starting point for creating a Markov chain that samples a target distribution providing that the transition matrix is symmetric. A further generalisation to this algorithm was proposed by Hastings in 1970 [13] which adds the 'Hasting's ratio' allowing for the transition matrix to be asymmetric. Both of these algorithms share the same core concepts and to implement them, we need to define a collection of objects:

- A finite state space $\mathcal{S}$
- A transition probability matrix Q.
- A target distribution π on $\mathcal{S}$ where $\pi_i > 0$ for all $i \in \mathcal{S}$

Repeat the following for *k* steps:
If we have a Markov chain which has a current state $X_t = i$ in $\mathcal{S}$, the successor state X_{t+1} is determined using the following criteria:

1. Randomly choose $Y \in \mathcal{S}$ according to our transition matrix Q. Y is called the *proposal*
2. Define a threshold α - this is the *acceptance probability*:
   - If Q is symmetric such that $q_{i,j} = q_{j,i}$ for every pair of states i, j, we can use the simplified Metropolis algorithm where $\alpha = \min(1, \pi_Y/\pi_i)$
   - If Q is asymmetric, we define $\alpha = \min(1, \frac{\pi_Y Q(X_t|Y)}{\pi_i Q(Y|X_t)})$

3. Generate a temporary random variable $U \sim U[0,1]$.

$$X_{t+1} = \begin{cases} Y & \text{if } U \leq \alpha \\ X_t & \text{otherwise} \end{cases}$$

Whilst the fundamental principles of these algorithms are straightforward to apply, the versatility of these techniques arises from the fact that we do not need a normalising constant for π, we just need to know the ratio between π_Y/π_i.

4.2.2 Implementing Metropolis-Hastings in R

We are now able to implement a simple MCMC simulation in R to confirm that our Markov chain does sample from our target distribution π - this is implemented in my script called **MCMC GGPLOT.R**. As we can see in Figure 4, we can expect a reasonably accurate Monte Carlo distribution after 50,000 iterations - we can estimate the amount of "Monte Carlo" standard error by using the following formula:

$$\frac{\hat{\sigma}_N}{\sqrt{N}} \tag{3}$$

I have included this metric in Figure 4, 5 and 6 [8]. Using [19] as a reference, we can also think about calculating a 95% confidence interval for the expected state within our Markov chain. To achieve this, we split our N observations from our stationary sequence into b non-overlapping batches - Madras later says that 20 or 30 batches are often sufficient. Once I had generated the 50,000 observations from my Markov chain, I saved it to a .csv file for convenience. To be conservative, and because I had such a large number of observations, I split my complete sample into 500 batches, each containing 100 observations. When L is sufficiently large where $L = N/b$, each of the b batches are approximately independent and as such, we can use classical sample statistics as follows:

$$\bar{Y}_b = \frac{1}{b}\sum_{i=1}^{b} Y_i \tag{Sample Mean}$$

$$S_b^2 = \frac{1}{b-1}\sum_{k=1}^{b}(Y_k - \bar{Y}_b)^2 \tag{Sample Variance}$$

For a large enough number of batches, we can define our 95% confidence intervals as the following:

$$\bar{Y}_b \pm t_{b-1}, 0.025\sqrt{S_b^2/b} \tag{95\% CI}$$

where t_{b-1} is a two-tailed student's t-distribution with $b-1$ degrees of freedom. Because it is a two-tailed test, we halve our alpha value of 0.05 giving 0.025. For my 50,000 observations, this results in the 95% confidence interval of $[1.9812, 2.0062]$. This means that 95% of the time, our Markov chain's expected state is state 2 (state B) and this coincides with Figure 6. Figure 4 also suggests that state B is the most probable, further reinforcing our confidence interval. My confidence interval calculations are found within the **CI via Batch Means.R** script.

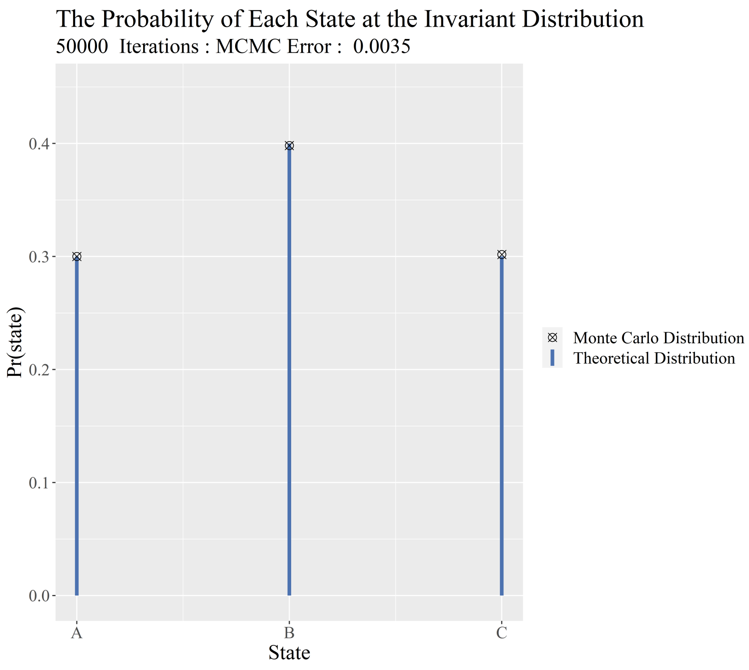

Figure 4. A plot illustrating the performance of my Monte Carlo algorithm after 50,000 iterations. We can see a relatively close agreement between the theoretical distribution and the distribution produced my my MCMC sampler. We also see that state 2 (state B) is the most likely of the three states.

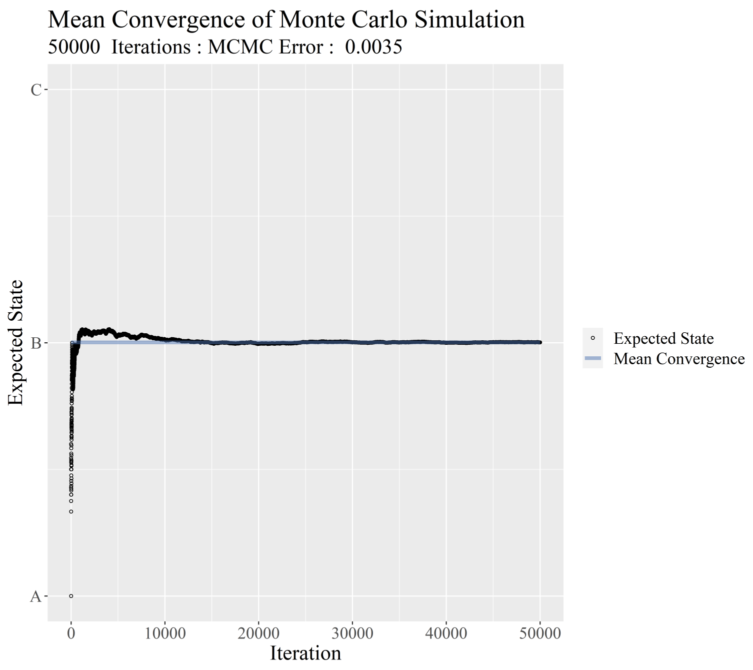

Figure 5. A plot showing how my Markov chain converges towards state B - after looking at Figure 4, this is expected as state B is the most probable.

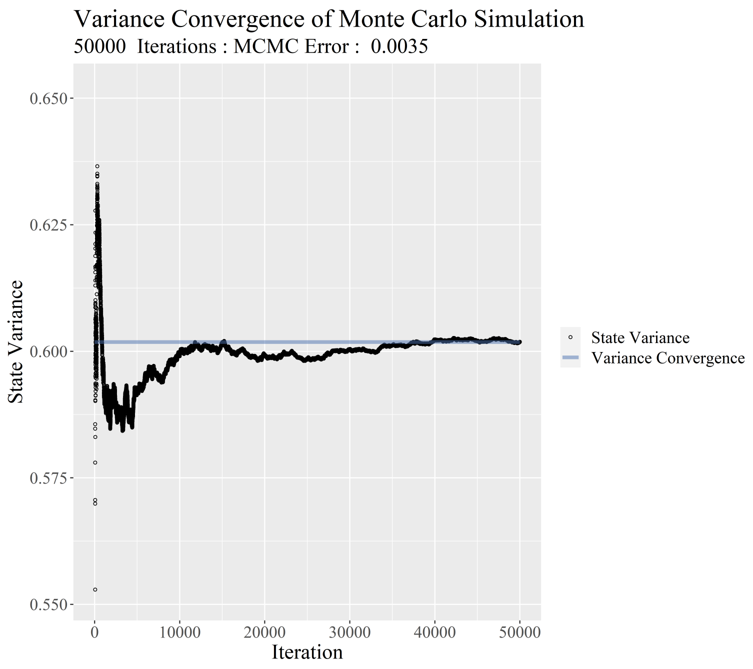

Figure 6. A plot showing the convergence of variance for my Markov chain. We can see that this converges at a much slower rate compared to the mean convergence in Figure 6.

5. Monte Carlo Random Walk

The Monte Carlo Random Walk is one of the central algorithms within MCMC. Given initial conditions, a random walk algorithm will take random steps around the sampling space and given a suitably high number of iterations, will converge towards the desired sampling distribution. Whilst there are a number of algorithms to choose from, the simplest rendition of a random walk can be implemented using the Metropolis-Hastings that I have already illustrated. As a proof of concept, I will use the Poisson random variable that I introduced in the first section to implement a MCMC Poisson sampler. These plots can be produced using my **Poisson MCMC GGPLOT.R** script which in turn requires my MCMC sampler found in **Poisson MH_MCMC Sampler.R**

5.1. Initial Conditions

A significant issue with all MCMC methods is the time it takes for them to converge to the invariant distribution - this is also known as the mixing time or ’burn in’ period. Burn in periods are often incorporated into MCMC algorithms by naively removing the first k observations from your overall sample; whilst this is effective, it is clear to see how this simply makes the algorithm less efficient. If you wanted to generate 10,000 observations from your target distribution, you would have to generate 10,000 + k samples to account for the burn in period. Whilst there is the argument that your burn in period is usually a negligible fraction of your overall sample size, it is still a consideration when designing an MCMC algorithm. Another naive solution to this issue is to ’seed’ your sampler with the expected value (if known) of your distribution - with a Poisson distribution $\sim \text{Pois}(\lambda)$, this is trivial as both $\mathbb{E}(\text{X})$ and $\text{Var}(\text{X})$ equal λ. Another option for seeding your sampler is to set the initial value to a value ’you dont mind having in the sample’ [8]. Whilst there are other more advanced MCMC algorithms that reduce the mixing time such as the Gibbs sampler or even Hamiltonian Monte Carlo, using my simplistic Metropolis-Hastings algorithm, I decided to implement functionality for both a user-specified burn in period as well as the ability to seed the sampler with a desired initial value. Figures 7 and 8 clearly illustrates the reason why burn in periods are common and why the speed at with the sampler approaches its steady-state distribution is worth considering.

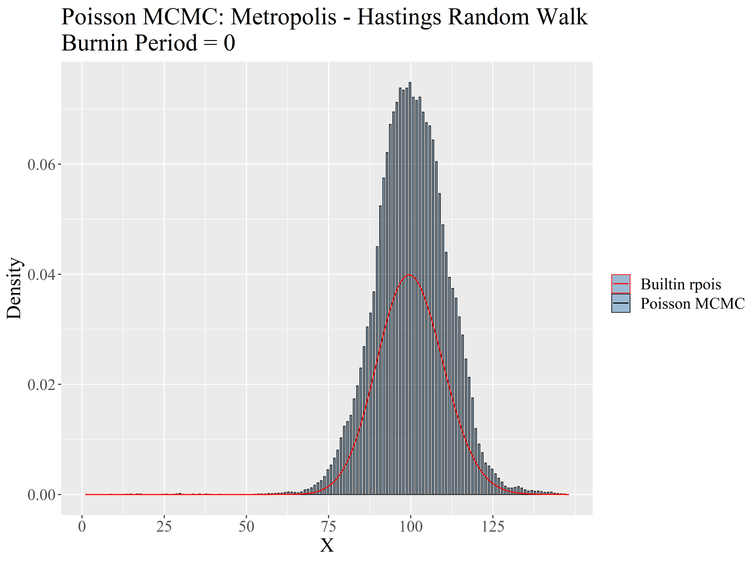

Figure 7. N = 100,000, $\lambda = 100$. We can see that whilst my sampler ’walked’ towards the desired distribution, the first part of the sample is unwanted and can introduce initialisation bias. This would make our estimations of the expected value skewed unless the sampler was run for a considerably long time.

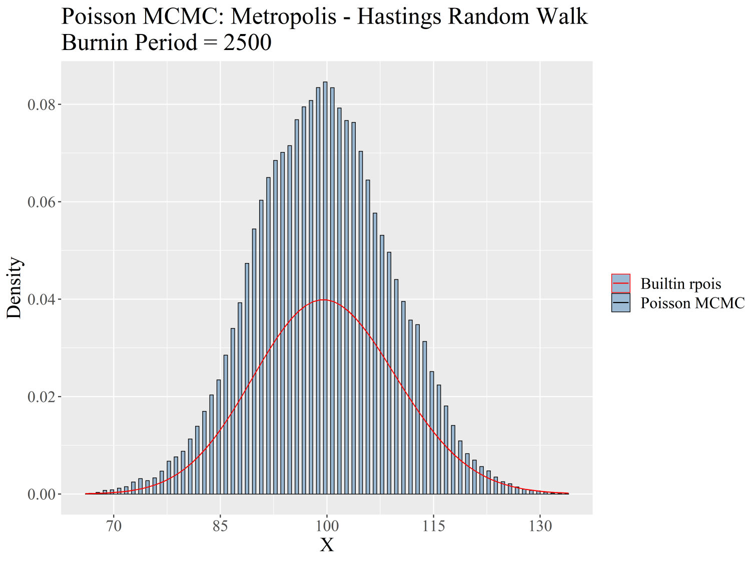

Figure 8. N = 100,000, $\lambda = 100$. With a burn in period of 2500 iterations, this distribution is a more accurate representation of Pois($\lambda = 100$) without the initialisation bias.

5.2. The Addition of N Poisson Variables

Given a sequence of N i.i.d random variables $\{X_i\}_{i=1}^{N} \sim \text{Pois}(\lambda)$, we were tasked with using our MCMC sampler to sample from Z_N such that:

$$Z_N = \sum_{i=1}^{N} X_i \tag{4}$$

The summation of i.i.d random variables is a common problem in probability theory when concerned with complex 'joint-events'. Madras offers an extensive exploration into the common example of pump failure within a power plant; here we can use a MCMC algorithm to determine quantities of interest such as the pump failure rate per unit time [19]. In order to successfully use my MCMC sampler, we need to derive a joint PMF for my Poisson variables which will act as out target distribution function within the Metropolis-Hasting algorithm. The derivations below are based on a range of online resources that I used to gain a better understanding of the mathematics used in solving these problems [2], [3], [12], [22], [29]. Before we see how the theory extends to N Poisson variables, I will begin by illustrating the expected joint distribution between two i.i.d Poisson variables.

5.2.1 The Distribution of the Sum of Two Poisson Random Variables

Assuming independence:

$$\text{Pois}(\lambda_1) + \text{Pois}(\lambda_2) = \text{Pois}(\lambda_1 + \lambda_2) \tag{5}$$

We can verify this a number of ways: directly from the PMF, from the characteristic function, the probability generating function (PGF) as well as the moment generating function (MGF). After exploring various online resources, I found that the PGF is a powerful tool when trying to characterise the joint distribution of discrete random variables. The Poisson PGF is as follows:

$$G_X(s) = \mathbb{E}(s^X) = e^{-\lambda}e^{(\lambda s)} = e^{\lambda(s-1)}$$

PGFs are useful because they transform a sum into a product, making them easier to manipulate. Before showing how PGFs are useful for the sums of discrete variables, I must first present a useful property of PGFs - the uniqueness property. If two random variables have the same PGF, we can say that the two random variables have the same PMF and thus, the same distribution. If X and Y are discrete random variables with PGFs G_X and G_Y then

$$G_X(s) = G_Y(s) \text{ for all } s \iff P(X = k) = P(Y = k) \quad (k \in \mathbb{N}_0)$$

Following this, let $X \sim \text{Pois}(\lambda_1)$ and $Y \sim \text{Pois}(\lambda_2)$ be independent and let $Z = X + Y$. We can find the distribution of Z as follows:

$$\begin{aligned} G_{X+Y}(s) &= G_X(s) \cdot G_Y(s) \\ &= e^{\lambda_1(s-1)}e^{\lambda_2(s-1)} \\ &= e^{(\lambda_1+\lambda_2)(s-1)} \end{aligned}$$

Through the uniqueness property of PGFs, we can say that $X + Y \sim \text{Pois}(\lambda_1 + \lambda_2)$.

5.2.2 The Distribution of the Sum of N Poisson Random Variables

Following on from two Poisson random variables, if we have N i.i.d Poisson variables (Eq. 4), and equation 5 holds, we can say that the distribution of N random variables $\sim \text{Pois}(\lambda)$ has the following distribution:

$$Z_N \sim \text{Pois}(\lambda_1 + \lambda_2 + \lambda_3 + ... + \lambda_N)$$

Within our program we can simplify this by finding a new 'temporary' λ value which equals $N\lambda$. Within the Poisson PMF and CDF, we would have the following:

$$\frac{N\lambda^k e^{-N\lambda}}{k!} \tag{6}$$

$$\sum_{i=1}^{k} \frac{N\lambda^k e^{-N\lambda}}{i!} \tag{7}$$

For my purposes, I decided to sample the sum of 10 Poisson variables with parameter ($\lambda = 3$). If the theory holds, this should sample the distribution $\text{Pois}(10 \cdot 3 = 30)$ - we should be able to verify this easily as the expectation also scales linearly with N meaning that 30 should be our new expectation and variance. This can be seen in **N Poisson Variables.R**.

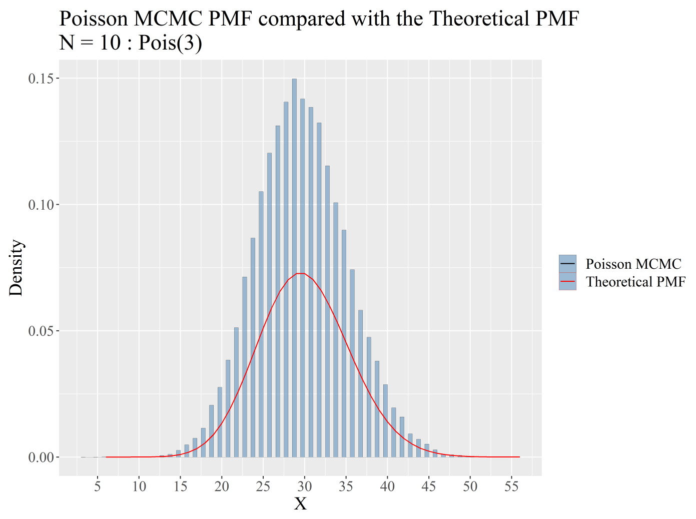

Figure 9. A plot illustrating 50,000 samples of Z_N where $N = 10$ and $\lambda = 3$ using my MCMC sampler. There was no burnin period for this sample.

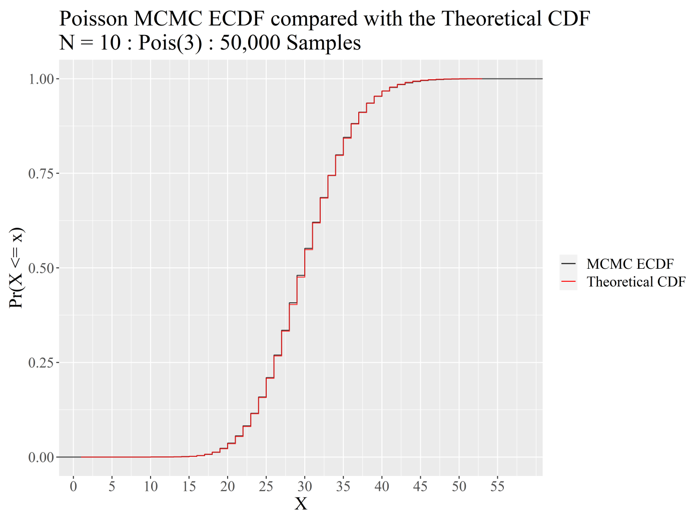

Figure 11. A plot comparing the empirical cumulative distribution function of my MCMC sample with the theoretically calculated CDF shown above. With 50,000 samples, we can see how closely my MCMC sample is to the theoretical CDF.

Figure 9 illustrates how my MCMC algorithm is able to sample Z_N when provided with the updated PMF. Figure 10 and 11 compare the ECDF of my MCMC sample with the theoretical CDF shown above. I included two figures with a varied number of samples to highlight how the respective CDFs converge as the number of iterations increase.

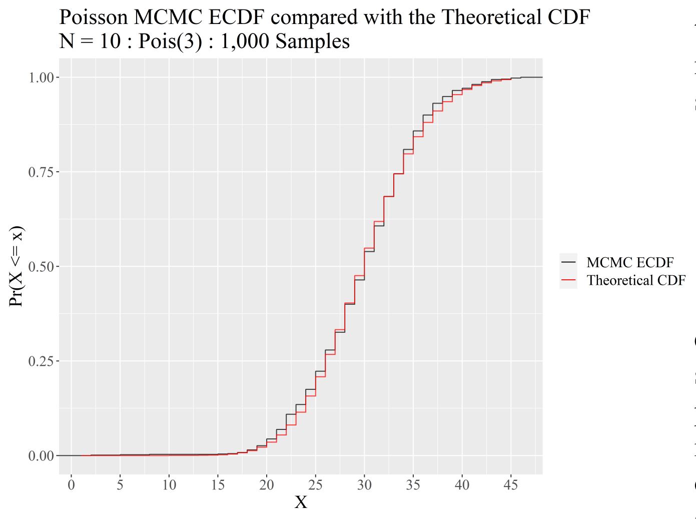

Figure 10. A plot comparing the empirical cumulative distribution function of my MCMC sample with the theoretically calculated CDF shown above. With only 1,000 samples, we still see some disagreement between the two quantities.

5.3. The Convergence to the Standard Normal Distribution

Before we looked at the sum of two and ten Poisson random variables respectively; when $N \to \infty$, the random variable Z_N should approximate a normal distribution. Further to this, the central limit theorem dictates that providing $\mathrm{E}(X) < \infty$ and $\mathrm{Var}(X) < \infty$, a new random variable U_N should converge towards a standard normal distribution defined as the following:

$$U_N = \frac{Z_N - N\mathbb{E}(X)}{\sqrt{N\mathrm{Var}(X)}} \tag{8}$$

Due to issues with numerical stability when using extremely high λ values, I decided to alter my MCMC sampling algorithm. Rather than providing the sampler with the theoretical PMF (Eq. 6), I decided to put my sampler within its own **for** loop where the number of iterations of the loop equates to the number of Poisson random variables in the overall sum. Whilst this is considerably slower, it allows for the calculation of U_N when N is very large. This is implemented in my **Standard Normal MCMC.R** script. Figure 12 shows the results from my MCMC algorithm when we sample 30,000 observations of the sum of 500 individual

Poisson variables with parameter $\lambda = 100$. Figure 13 shows how this can be transformed using equation 8 to approximate a standard normal distribution.

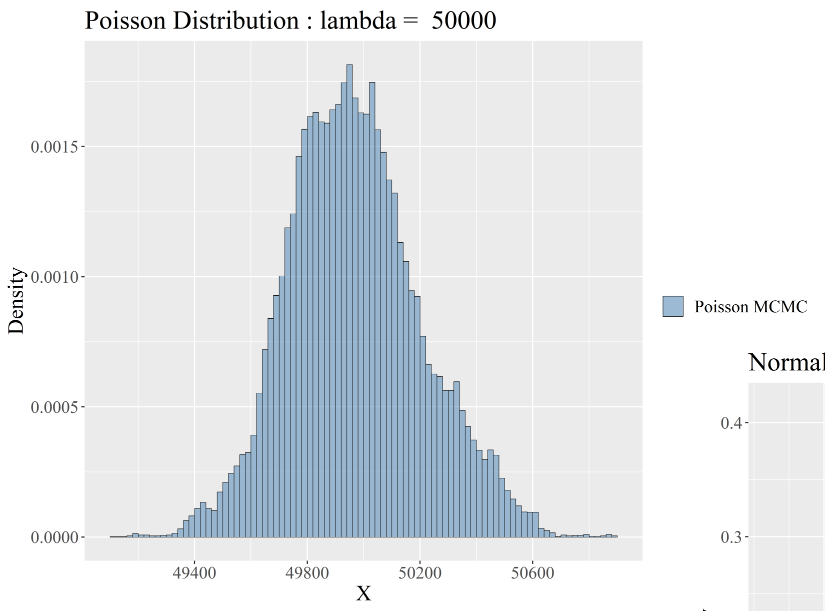

Figure 12. A plot of results from my MCMC algorithm. Here we have 500 individual Poisson random variables, all with $\lambda = 100$ resulting in a plot of Pois(50,000).

Using similar batch means techniques as shown before, we can once again construct a confidence interval for my Monte Carlo simulation in order to assess it's accuracy. First, calculating the Monte Carlo standard error (Eq. 3) gives us an error of 1.354. Using 500 mini-batches for my 30,000 observations, I calculated the confidence interval [49968.72, 50008.25] for my MCMC sampler.

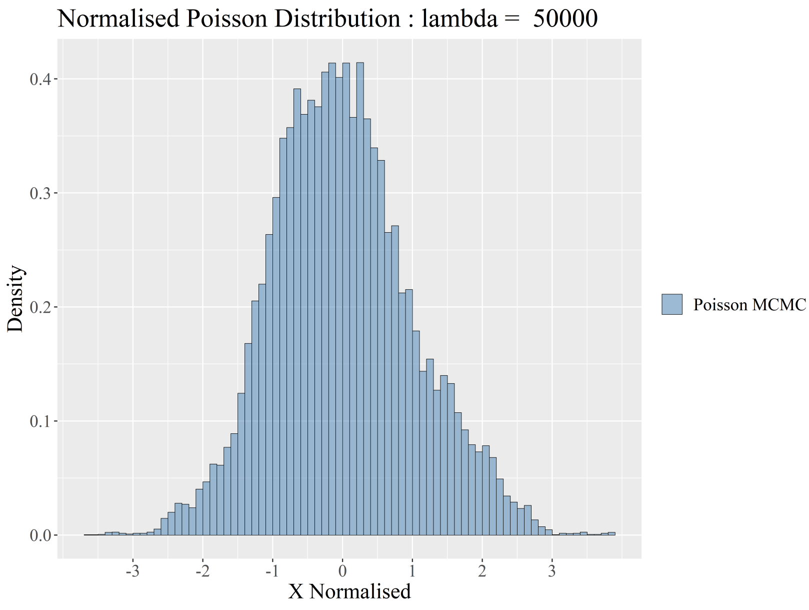

Figure 13. A plot of results from my MCMC algorithm. Here we have 500 individual Poisson random variables, all with $\lambda = 100$ resulting in a plot of Pois(50,000).

References

[1] Poisson distribution. *Wikipedia*. 3

[2] Poisson distribution of sum of two random independent variables. https://bit.ly/3tsDZLY, 2021. 12

[3] Probability generating function. https://random-walks.org/content/prob-intro/ch04/content.html, 2021. 12

[4] J. Ahrens and U. Dieter. A convenient sampling method with bounded computatin times for poisson distributions. In *The First International Conference on Statistical Computing*, pages 4–17. . 4

[5] J. H. Ahrens and U. Dieter. Computer generation of poisson deviates from modified normal distributions. *ACM Transactions on Mathematical Software (TOMS)*, 8(2):163–179, 1982. 4

[6] T. Arnold and J. Emerson. Nonparametric goodness-of-fit tests for discrete null distributions. *R Journal*, 3, 2011. 5

[7] L. Bortkewitsch. *Das Gesetz der kleinen Zahlen*. Teubner, 1898. 3

[8] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011. 9, 11

[9] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986. 3

[10] P. Diaconis. The markov chain monte carlo revolution. *Bulletin of the American Mathematical Society*, 46:179textendash205, 2009. 7

[11] D. S. Dimitrova, V. K. Kaishev, and S. Tan. Computing the kolmogorov-smirnov distribution when the underlying cdf is purely discrete, mixed, or continuous. *Journal of Statistical Software; Vol 1, Issue 10 (2020)*, 2020. 5

[12] R. Fewster. Stats 325 - stochastic process: Chapter 4. generating functions. https://www.stat.auckland.ac.nz/~fewster/325/notes/ch4.pdf, 2014. 12

[13] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. 8

[14] W. Hörmann. The transformed rejection method for generating poisson random variables. *Insurance: Mathematics and Economics*, 12(1):39–45, 1993. 3, 4

[15] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Commun. ACM*, 31(2):216–222, 1988. 4

[16] S. Karlin and H. M. Taylor. *Chapter 2 - MARKOV CHAINS*, pages 45–80. Academic Press, Boston, 1975. 7

[17] C. D. Kemp. New algorithms for generating poisson variates. *Journal of Computational and Applied Mathematics*, 31(1):133–137, 1990. 3, 4

[18] C. D. Kemp and A. W. Kemp. Rapid generation of frequency tables. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 36(3):277–282, 1987. 4

[19] N. N. Madras. *Lectures on Monte Carlo Methods*. Fields Institute Monographs. American Mathematical Society, 2002. 7, 9, 12

[20] M. L. McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013. 6

[21] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. 8

[22] S. J. Miller. Sums of poisson random variables. https://web.williams.edu/Mathematics/sjmiller/public_html/BrownClasses/162/SumPoisson.pdf, 2021. 12

[23] A. N. Pettitt and M. A. Stephens. The kolmogorov-smirnov goodness-of-fit statistic with discrete and grouped data. *Technometrics*, 19(2):205–210, 1977. 5

[24] E. Scalas. Summary of results on markov chains. 2007. 7

[25] E. Scalas. Markov chain monte carlo. 2021. 7

[26] K. Sheppard, W. Matti Picus, Weckesser, R. Monton, E. Wieser, and E. Franzella. https://github.com/numpy/numpy/blob/main/numpy/random/src/distributions/distributions.c. GitHub. 4

[27] K. Siegrist. Probability, mathematical statistics, stochastic processes. http://www.randomservices.org/random/, 2021. 7

[28] F. H. Stephenson. *Chapter 3 - Cell growth*, pages 45–81. Academic Press, Boston, 2010. 3

[29] M. Taboga. Sums of independent random variables. *Lectures on probability theory and mathematical statistics*, 2017. 12

[30] J. W. Tukey. Some graphic and semigraphic displays. *Statistical papers in honor of George W. Snedecor*, 5:293–316, 1972. 6

Appendices

A. Extra Plots

A.1. Rootograms

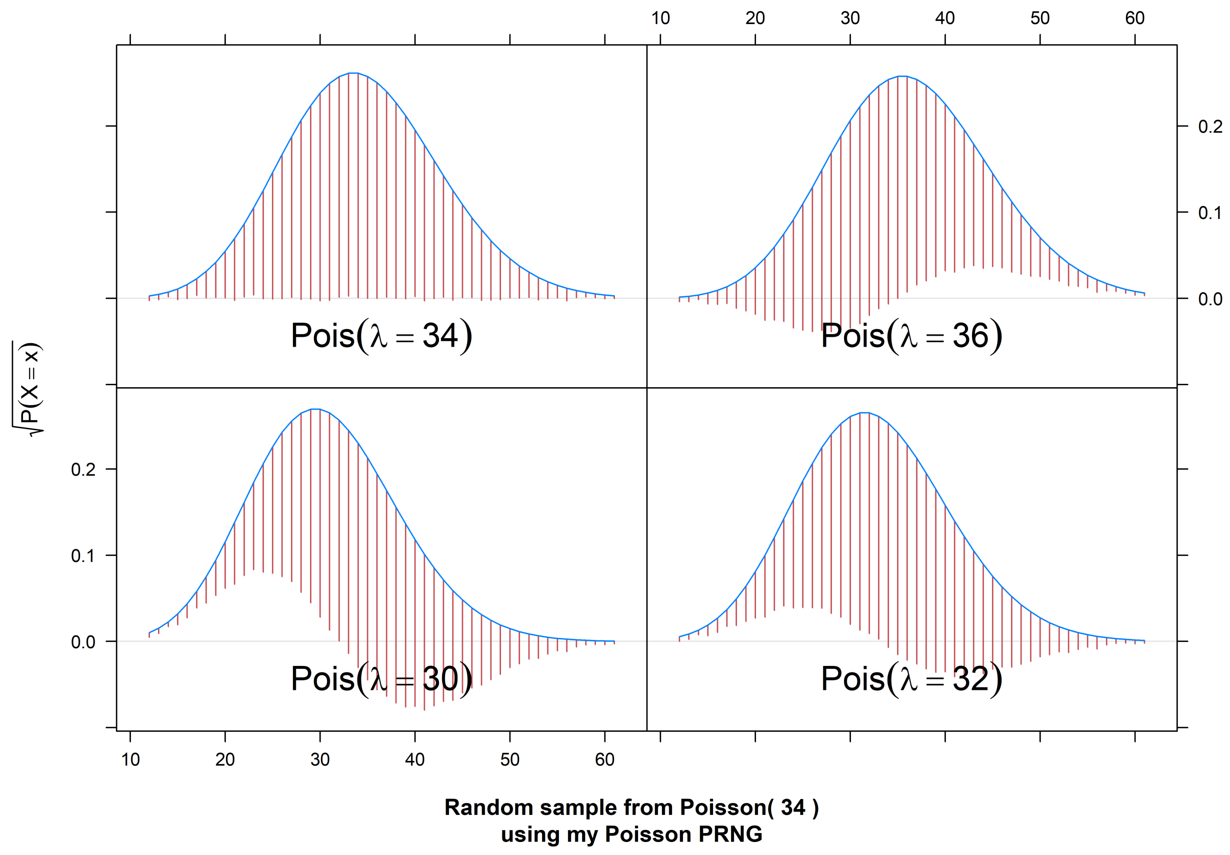

Figure 14. A plot using the R package *latticeExtra*. This compares the output of my sequential search algorithm at different values of λ.

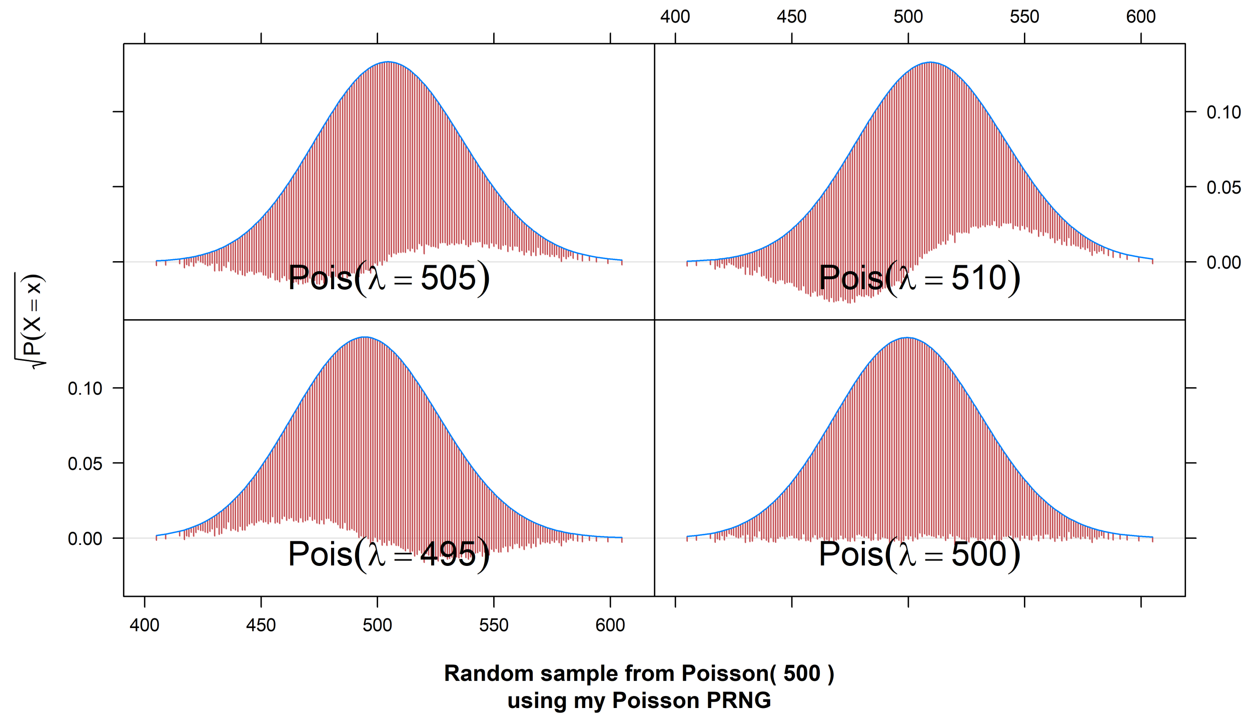

Figure 15. A plot using the R package *latticeExtra*. This compares the output of my PTRS algorithm at different values of λ.

B. Extended Mathematics

B.1. Calculating the Invariant Distribution

Given the following transition matrix P, calculate the invariant distribution

$$P = \begin{pmatrix} 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 1/2 \\ 0 & 1 & 0 \end{pmatrix}$$

We can represent P as a system of linear equations

$$\frac{1}{3}\pi_A + \frac{1}{2}\pi_B = \pi_A \tag{i}$$

$$\frac{1}{3}\pi_A + \pi_C = \pi_A \tag{ii}$$

$$\frac{1}{3}\pi_A + \frac{1}{2}\pi_B = \pi_C \tag{iii}$$

We also know that as π_A, π_B and π_C are probabilities, they must sum to 1.

$$\pi_A + \pi_B + \pi_C = 1 \tag{iv}$$

Notice that (i) and (iii) are equivalent $\therefore$ $\pi_A = \pi_C$. Rearrange (i)

$$\begin{aligned} &\frac{1}{2}\pi_B = \frac{2}{3}\pi_A \\ &3\pi_B = 4\pi_A && \text{(multiply by common factor)} \\ &\frac{3}{4}\pi_B = \pi_A = \pi_C && \text{(rearrange in terms of B)} \end{aligned}$$

Substitute into (iv)

$$\begin{aligned} \frac{3}{4}\pi_B + \pi_B + \frac{3}{4}\pi_B &= 1 \\ \frac{5}{2}\pi_B &= 1 && \text{(group terms)} \\ 5\pi_B &= 2 \\ \pi_B &= \frac{2}{5} = 0.4 \end{aligned}$$

Find π_A and π_C

$$\begin{aligned} \pi_A + 0.4 + \pi_C &= 1 \\ \pi_A + \pi_C &= 1 - 0.4 \therefore \\ \pi_A = \pi_C &= 0.3 \end{aligned}$$

This gives our invariant distribution of $[\pi_A, \pi_B, \pi_C] = [0.3, 0.4, 0.3]$