STAT 431

The Metropolis Algorithm: A Special Case of Metropolis-Hastings

Henry Szklanny, Aditya Kakarla, Xiangru An, Zexi Chen

December 2024

Introduction

The Metropolis-Hastings algorithm is a widely used method that is part of the MCMC methodologies. It plays an essential role in modern statistical methods, especially in Bayesian inference, statistical physics, and optimization. The algorithm, named partly after Nicholas Metropolis, originated in a 1954 paper titled "Equation of State Calculations by Fast Computing Machines", which introduced a method for random sampling using symmetric distributions, called the Metropolis algorithm. In 1970, Wilfred Keith Hastings expanded his ideas to handle more general cases with asymmetric proposals, and together these contributions form what we now call the Metropolis-Hastings Algorithm.

The primary objective of the algorithm is to generate random samples from probability distributions that are complex or high-dimensional that cause direct sampling to be unfeasible or analytically impossible. At its core, MH is a tool that is used to explore a distribution that is often called the target distribution. By generating a sequence of samples, the algorithm creates a sequence of Markov Chains. These samples are then used to approximate integrals, probabilities, and other statistics.

To illustrate the special topic of the Metropolis-Hastings (MH) algorithm, it is also important to remember that MH is used to obtain samples from probability distributions that have unusual shapes. Bimodal distributions are not as common as unimodal distributions, they are not famous enough to have easily recognizable density functions or have their own names, and they also have an unusual shape defined by two peaks. If the posterior has multiple modes offset from each other, this can make sampling difficult for MCMC methods like the Gibber Sampler. It is hoped that the samples generated by Gibbs converge in

distribution to the posterior, and do not get stuck in one place, like around one mode (Reich & Ghosh, 2021). Therefore, to illustrate the methodology of MH, we motivate its use for the example case of a bimodal distribution and then generate probabilities and summary statistics.

Compared to the Gibbs Sampler, MH is more generally applicable to more complicated problems. Furthermore, MH is actually a generalization of a special case of what is just called the Metropolis Algorithm. One of the steps involved in using MH is to suggest a proposal distribution from which candidate points will be sampled from (which is why the proposal distribution is sometimes also called the candidate distribution). If the proposal distribution is symmetric, the MH algorithm becomes just the Metropolis Algorithm. Our goal and motivation for using the algorithm is unchanged despite this small technical distinction, and henceforth we will refer to it as just the Metropolis Algorithm.

The Algorithm

1. Initialization: Start with an initial state x_t (e.g. $x_{t=0} = 0$).

2. Proposal Step: At the current state x_t, generate a candidate x_c from the proposal distribution $\rho(x_c \mid x_t)$.

3. Compute the Acceptance Probability: Calculate the acceptance probability, which is $\alpha = \min\left(1, \frac{f(x_c)}{f(x_t)}\right)$. This formula calculates the likelihood of the proposed state x_c relative to the current state x_t, adjusted for the proposal distribution.

4. Accept or Reject: Draw a random number U from a uniform distribution $U \sim \text{Uniform}(0, 1)$. If $\alpha > U$, accept the proposed state: $x_{t+1} = x_c$. Otherwise reject the proposed state: $x_{t+1} = x_t$.

5. Repeat: Iterate the proposal and acceptance steps for a sufficient number of iterations to allow the Markov chains to explore the target distribution and generate a representative sample from the target distribution.

Mathematical Foundations of MH and Key Components

MH fulfills its goal by generating a sequence of Markov chains that converge to the target distribution. In other words, the stationary distribution of the

Markov chains matches the desired target distribution. It does this by iteratively generating proposed states and then deciding whether to accept or reject them.

- Target Distribution: This is the probability distribution that we want to sample from.
- Proposal Distribution: This is a conditional distribution that specifies how to propose a new state given the current state.
- Acceptance Probability: This determines whether to move to the proposed state or stay with the current state. It is computed as

$$\alpha = \min\left(1, \frac{f(x_c)}{f(x_t)} \times \frac{\rho(x_t \mid x_c)}{\rho(x_c \mid x_t)}\right)$$

  This is calculated to ensure that the chains converge to the target distribution. For symmetric proposals it simplifies to $\alpha = \min\left(1, \frac{f(x_c)}{f(x_t)}\right)$.

- Detailed Balance Criterion: An important step in the Metropolis Algorithm is to show that the detailed balance criterion holds:

$$f(x_t) \times \alpha(x_t \to x_c) = f(x_c) \times \alpha(x_c \to x_t)$$

  where the left side of the equation represents the probability of transitioning from x_t to x_c and the right side represents the probability of transitioning from x_c to x_t. The MH algorithm ensures convergence to the target distribution if the detailed balance condition holds.

Example Case

Define a Bimodal Target Distribution

We begin our illustration of MH with the first step: defining a bimodal target distribution. We define the target distribution as the sum of two Normal distributions with different means and same standard deviations. This target should clearly resemble a bimodal distribution with peaks at -3 and 3

$$f(x_t) = 0.5 \times N(x_t \mid \mu_1, \sigma_1^2) + 0.5 \times N(x_t \mid \mu_2, \sigma_2^2)$$

where

- $N(x \mid \mu, \sigma^2)$ denotes a Normal distribution
- $\mu_1 = -3, \sigma_1 = 0.5$
- $\mu_2 = 3, \sigma_2 = 0.5$

Suggest a Proposal Distribution

The second step is to suggest a proposal distribution. We suggest that our proposal $\rho(x_c \mid x_t)$ be a Normal centered about the current or most recent state x_t

$$\rho(x_c \mid x_t) = N(x_c \mid x_t, \sigma_\rho^2)$$

where σ_ρ is the standard deviation of the proposal distribution. A common choice for the proposal is a random-walk Normal candidate distribution. This would be an acceptable and even a good proposal in the bimodal case because it adds jitter to the current state of the Markov chain (Reich & Ghosh, 2021). This is desirable because if the tuning parameter σ_ρ of the candidate distribution is appropriately tuned, then the size of the jitters will cover a large parameter space of the target distribution. In other words, the Markov chain should not become "stuck" around one mode. A large σ_ρ will enable faster exploration of the parameter space and greater coverage by "jumping" between both modes. Using a Normal candidate distribution also simplifies the acceptance ratio, which will be explained in the very next step. The obvious question then concerns what σ_ρ should be. According to Reich and Ghosh (2021), "a rule of thumb is to tune the algorithm so that it accepts 30–50% of the candidates" (p. 91). Reich and Ghosh (2021) also note that it can be difficult to know what value(s) of σ_ρ will lead to this acceptance rate range before starting the sampling process. Interestingly, Roberts et al. (1997) show that "the ideal variance in the proposal is twice the variance of the target or, equivalently, that the acceptance rate should be close to $\frac{1}{4}$" (as cited in Robert, 2016, p. 6). According to (Robert, 2016), although "this rule is only an indication (in the sense that it was primarily designed for a specific and asymptotic Gaussian environment), it provides a golden rule for the default calibration of random walk Metropolis–Hastings algorithms" (p. 6). The second step ends with us initializing with an arbitrary starting point $x_{t=0} = 0$. We also choose to generate $\mathcal{S} = 1,000,000$ samples for this simulation.

Iterate the Sampling Process

The third step is to iterate the sampling process. To do this, we sample a candidate point x_c from the proposal distribution. In other words, we draw a candidate $x_c \sim N(x_t, \sigma_\rho^2)$. Continue for each iteration $t = 1, \ldots, \mathcal{S}$ by drawing a new sample. Then, compute $f(x_t)$ and $f(x_c)$. So, the chain starts at

$$f(x_0) = 0.5 \times N(x_0 \mid -3, 0.5^2) + 0.5 \times N(x_0 \mid 3, 0.5^2)$$

and

$$f(x_c) = 0.5 \times N(x_c \mid -3, 0.5^2) + 0.5 \times N(x_c \mid 3, 0.5^2)$$

Next, we compute an acceptance ratio. A special property of random-walk candidate distributions like the Normal is the fact that the acceptance ratio simplifies to

$$\alpha = \min\left(1, \frac{f(x_c)}{f(x_t)}\right)$$

According to Reich and Ghosh (2021), if the candidate distribution is a Normal or any symmetric distribution, the MH algorithm reduces to the Metropolis algorithm. The only difference between MH and the Metropolis algorithm is the fact that the latter samples from symmetric distributions, and therefore have different acceptance ratios. In fact, the Metropolis Algorithm is a special case of pure MH. In Metropolis-Hastings, the acceptance ratio is slightly more involved. It is

$$\alpha = \min\left(1, \frac{f(x_c)}{f(x_t)} \times \frac{\rho(x_t \mid x_c)}{\rho(x_c \mid x_t)}\right)$$

where again $\rho(x \mid y)$ is some type of proposal distribution. The ratio of the candidate distributions cancels out. The next part of the third step is the acceptance or rejection of the sample x_c. The accept/reject step generates a number from $U \sim \text{Uniform}(0, 1)$. If $\alpha > U$, the candidate is accepted, otherwise the candidate is rejected. If the candidate is accepted, we set $x_{t+1} = x_c$, meaning the next most recent value is the candidate. If the candidate is rejected, we set $x_{t+1} = x_t$, meaning the next most recent value will just be the current value. Equivalently,

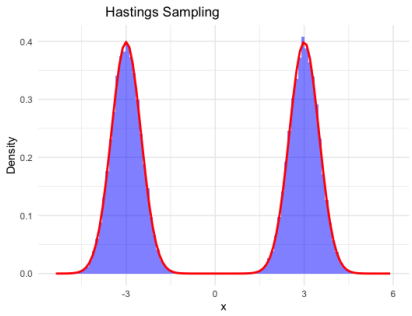
$$x_{t+1} = \begin{cases} x_c & \text{if } \alpha > U \\ x_t & \text{otherwise} \end{cases}$$

Repeat

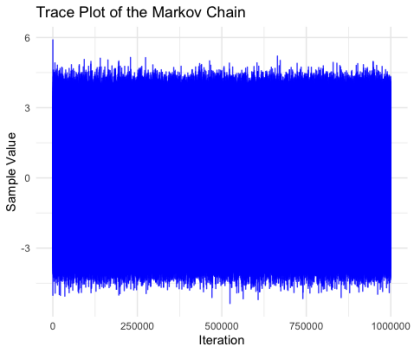
The fourth step is to repeat the above steps for $\mathcal{S}$ iterations to generate a sequence of samples $x_0, \ldots, x_{\mathcal{S}}$.

The Tuning Parameter σ_ρ

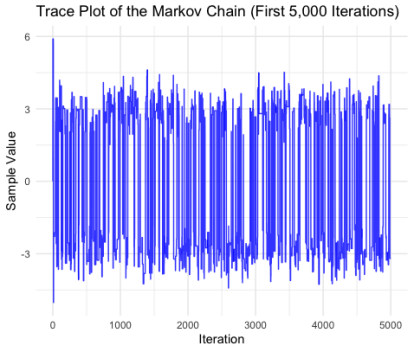
We now return to the topic regarding the tuning parameter σ_ρ and discuss the advantages and disadvantages of a high and low acceptance rate. There is no hard and fast rule on what the ideal acceptance rate should be. Different sources in the literature have varying ideas. A high acceptance rate generally means that the samples are very close to the current value of the chain. This can be desirable if we are sampling from a narrow distribution like a high-peaked unimodal. Obviously, a high acceptance rate (low σ_ρ) for our example would be a bad suggestion. But we also need to be cautious about a very low acceptance rate. A low acceptance rate means that the proposed samples are more divergent from the current value. This can be desirable if the goal is to explore a large parameter space and obtain greater coverage. Obviously, this is what we want in our case: a fairly low acceptance rate and fairly high σ_ρ. We follow the rule of Roberts et al. (1997), and we choose our $\sigma_\rho = \sqrt{2 \times 9.25} \approx 4.3$. This value of σ_ρ results in an acceptance rate of about 20.27%. Although this rate is slightly smaller than what Roberts et al. (1997) envisioned, we find no issue because the trace plot shows quick convergence. We can know that the MH algorithm is properly generating samples from the target distribution based on visualization tools and convergence diagnostics. According to Reich and Ghosh (2021), "verifying that the MCMC chains have converged and long run enough to sufficiently explore the posterior is often done informally by visual inspection of trace plots" (p. 103). The algorithm generates a Markov chain that must settle into the target distribution after a burn-in phase. The trace plot shows how the chain explores the parameter space. For the bimodal example, the trace plot should show the chain oscillating between the two modes. As $s \to \infty$, the algorithm does a better job approximating the target distribution. So, the samples should approximately match the target distribution when we look at a histogram.



(a) Histogram of samples



(b) Trace plot of 1,000,000 iterations



(c) Trace plot of first 5,000 iterations

Detailed Balance Criterion

Here we show that the detailed balance criterion for the Markov Chain is met. We cannot possibly provide the output to show that it is met for all one million iterations, so we truncate the output in the appendix to show the first 100. The code is written to show the difference between the left side and right side of the equation, and it should be 0. This necessarily implies that the probability of transitioning from x_t to x_c and the probability of transitioning from x_c to x_t are the same, which they are.

```
Iteration 1 :
LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.004432
RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.004432
Discrepancy (LHS - RHS): 0
```

Detailed Balance Criterion `R` Output for the First Iteration

Summary Statistics and Probabilities

We are now in a position to approximate expected values, variances, and probabilities. The expected value should not be difficult to deduce. Just by intuition we know the expected value is zero because it is the balance point of the target. Although perhaps it is not initially obvious, the variance is actually 9.25. Interestingly, while the standard Normal distribution cannot be expressed as a closed-form solution, the sum of two Normals like in this example can be. We can still, however, use MCMC methods to compute the same variance, which will be provided in the `R` code. According to Corona (2017), the variance of a mixture of k distributions with weights w_i, means μ_i, variances σ_i^2, and average mean of all k distributions $\bar{\mu}$ is

$$\sum_{i=1}^{k} w_i \sigma_i^2 + \sum_{i=1}^{k} w_i (\mu_i - \bar{\mu})^2$$

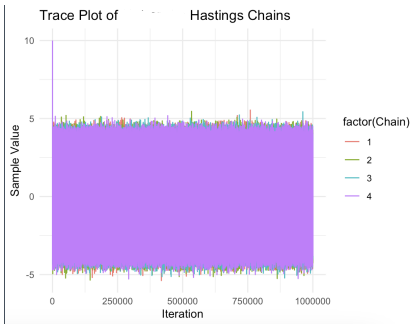
which if we compute will result in 9.25 (for $k = 2$). To compute probabilities, we use $P(-\pi < X < \pi)$ as an example. Computing this probability in `R` gives a result of approximately 0.610052.

Diagnostics

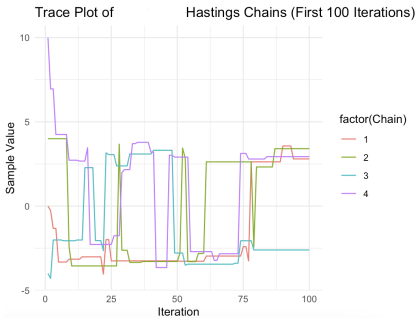
We wish to briefly return to the diagnostics. We chose our initial value for the chain to be the arbitrary starting point $x_{t=0} = 0$. But the better approach to running MCMC methods is to choose several initial starting values. Then, for each initial value, a separate chain is run for $\mathcal{S}$ iterations. The chains should then be monitored for convergence and the first $\mathcal{S}_b$ iterates should be part of the burn-in phase. We wish to implement this to be complete and thorough.

We define four chains with initial states $0, 4, -4, 10$, respectively. The chain with initial value 0 is simply the chain we wrote above, and the others start in like fashion. The number of samples to generate for each chain is still $\mathcal{S} = 1,000,000$. The proposed standard deviation is still $\sigma_\rho = \sqrt{2 \times 9.25} \approx 4.3$ for each chain. As before, each chain generates an acceptance rate and should be similar. For chain 1 with initial starting value 0, the acceptance rate is 20.19%. This is slightly different from before due to sampling variability. For chain 2 with initial starting value 4, the acceptance rate is 20.32%. For chain 3 with initial starting value -4, the acceptance rate is 20.27%. For chain 4 with initial starting value 10,

the acceptance rate is 20.22%. When we create a trace plot of the four chains, we observe a similar trace plot to the one before. We do not include the burn-in phase because burn-in was so immediate.



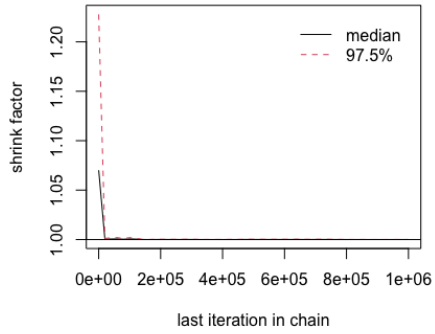
(a) Trace plot of 4 chains for 1,000,000 iterations



(b) Trace plot of 4 chains for first 100 iterations

Just as before, the trace plots show the chains oscillating between the two modes.

Further diagnostics include using the Gelman-Rubin plot to monitor convergence. This is especially useful when using multiple chains with different starting points. When the chains have converged, the Gelman-Rubin statistic should be close to 1, which the plot below shows.



Now we run the summary statistics of the combined samples in R. The mean is -0.006239 (which is about zero) and the standard deviation is 3.041483 (which is about $\sqrt{9.25}$). We again run $P(-\pi < X < \pi)$, and the approximate solution is 0.6111697 (which is about 0.610052). All of these statistics are very close to those solutions when we only used zero as our starting point for a single chain. This then begs the question of why we went through the trouble of using multiple chains to use MH. The initial value might be a very bad starting point. As Reich and Ghosh (2021) point out, "Surely we cannot trust that the initial value is a sample from the posterior because it is selected subjectively by the user. It would also be dangerous to trust that the first random sample follows the posterior distribution because it depends on the initial values, which might be far from the posterior" (p. 79). So, if a single starting value is used, there is the possibility that the chain might get "stuck" around one mode. For even more extremely unusual distributions, choosing several different overly dispersed initial values and running separate chains is the practice.

Summary and Contributions

In the code portion of this report, we implement this methodology. As expected, MH successfully obtains samples from the bimodal distribution. We now conclude this example by providing the R code that was used to create this simulation. We also provide the code with outputs in an R Markdown file. The steps are provided with brief commentary.

Henry Szklanny proposed the example for this report, specifically the use of a bimodal distribution to illustrate the Metropolis Algorithm. Henry worked on pages 2-9 of this report. Aditya contributed in writing the introduction, specifically explaining what the algorithm is and how it works before introducing any mathematical notation or applications. Aditya worked on pages 1-3 of this report. Xiangru and Zexi also contributed in providing ideas for the introduction and further contributed by writing the summary and compiling the references sections.

References

[1] Corona, F. (2017, February). Mixture distributions: Useful distributions. Lecture.

[2] Reich, B. J., & Ghosh, S. K. (2021). Bayesian statistical methods. CRC Press.

[3] Robert, C. P. (2017). The Metropolis-Hastings Algorithm, 1–15. https://doi.org/https://doi.org/10.48550/arXiv.1504.01896

MH Final Example

Henry Szklanny

2024-11-21

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```r
# Step 1: Define the bimodal target distribution
target_density <- function(x) {
  # Combine two normal distributions with equal weights
  w1 <- 0.5
  w2 <- 0.5
  mu1 <- -3
  sigma1 <- 0.5
  mu2 <- 3
  sigma2 <- 0.5
  w1 * dnorm(x, mean = mu1, sd = sigma1) + w2 * dnorm(x, mean = mu2, sd = sigma2)
}
```

```r
# Step 2: Define the Metropolis-Hastings function, suggest a proposal, initialize
metropolis_hastings <- function(num_samples, proposal_sd, x0, target_density) {
  # Initialize storage for samples
  samples <- numeric(num_samples)
  samples[1] <- x0  # Starting value

  num_accepted <- 0

  # Sampling process
  for (t in 2:num_samples) {
    current_x <- samples[t - 1]  # Current value

    # Generate a proposal from the normal distribution
    proposed_x <- rnorm(1, mean = current_x, sd = proposal_sd)

    # Compute the acceptance ratio
    acceptance_ratio <- target_density(proposed_x) / target_density(current_x)
    acceptance_ratio <- min(1, acceptance_ratio)  # Ensure ratio is in [0, 1]

    # Acceptance or Rejection
    if (runif(1) < acceptance_ratio) {
      samples[t] <- proposed_x  # Accept the proposal
      num_accepted <- num_accepted + 1
    } else {
```

```
      samples[t] <- current_x  # Reject the proposal
    }
  }

  # Calculate and display the acceptance percentage
  acceptance_percentage <- (num_accepted / (num_samples - 1)) * 100
  cat("Acceptance Percentage:", round(acceptance_percentage, 2), "%\n")

  # Return the generated samples
  return(samples)
}
```

```
# Step 3: Set parameters and run the algorithm, iterate the sampling process
set.seed(42)  # Set seed for reproducibility

num_samples <- 1000000  # Number of iterations
proposal_sd <- sqrt(2 * 9.25)     # Proposal standard deviation
x0 <- 0                 # Initial state
```

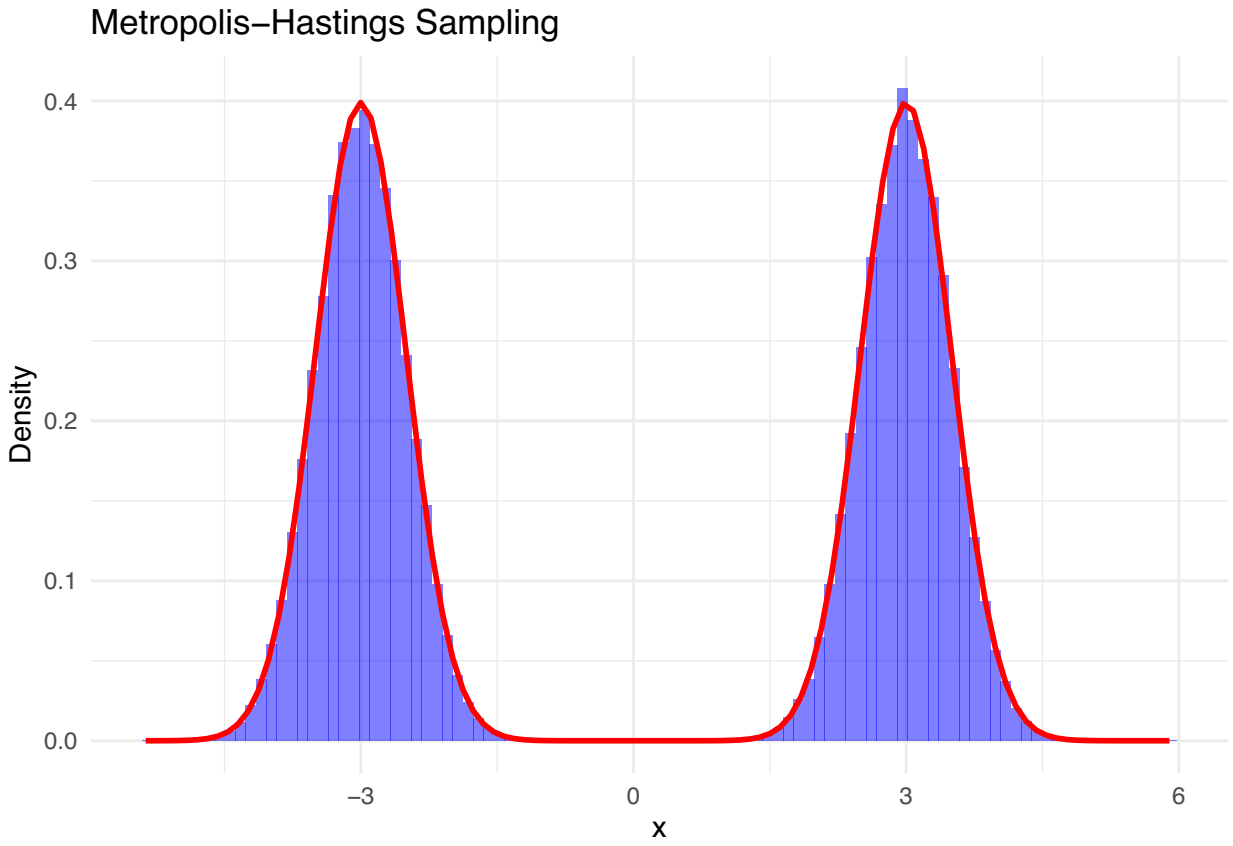
```
# Step 4: Repeat
samples <- metropolis_hastings(num_samples, proposal_sd, x0, target_density)
```

```
## Acceptance Percentage: 20.27 %
```

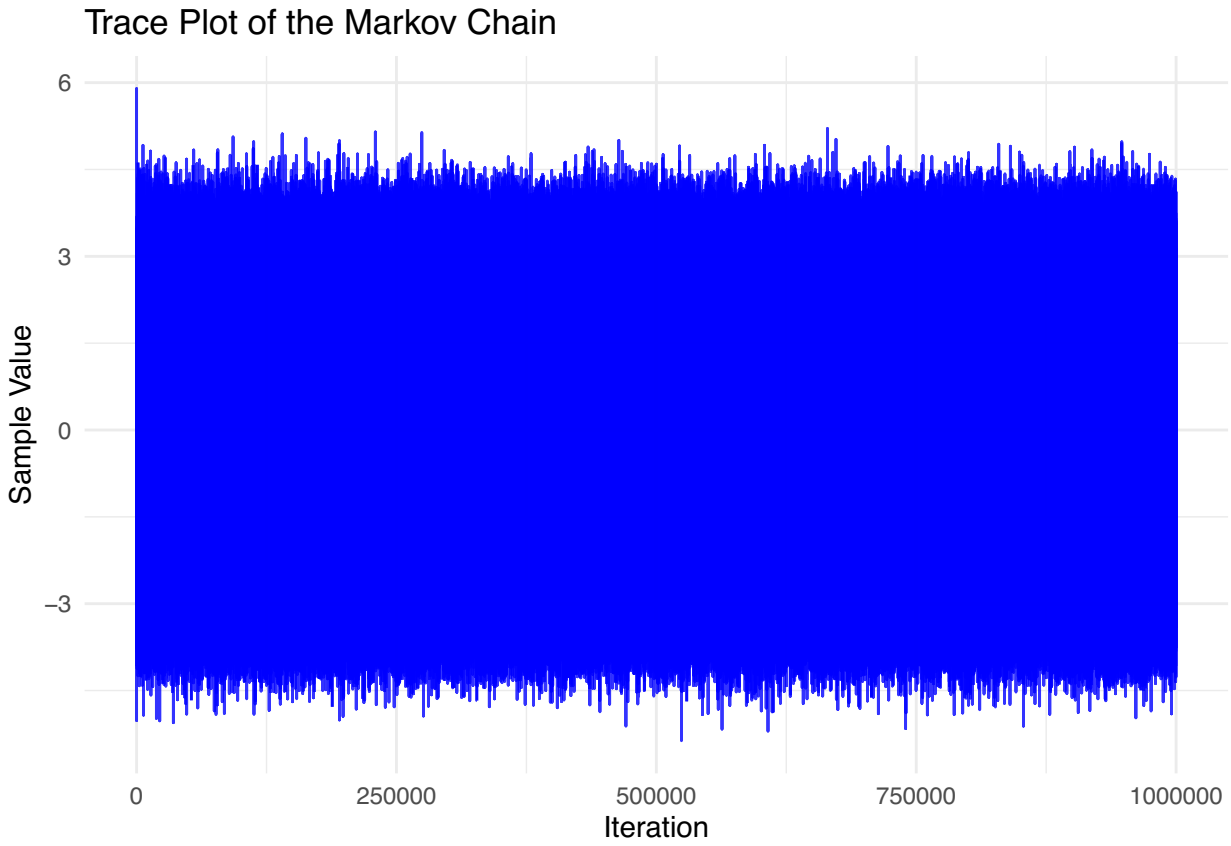
```
#Visualization and Statistics
library(ggplot2)

#Histogram of the samples
hist_data <- data.frame(samples = samples)
ggplot(hist_data, aes(x = samples)) +
  geom_histogram(aes(y = ..density..), bins = 100, fill = "blue", alpha = 0.5) +
  stat_function(fun = function(x) target_density(x), color = "red", size = 1) +
  labs(title = "Metropolis-Hastings Sampling",
       x = "x",
       y = "Density") +
  theme_minimal()
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

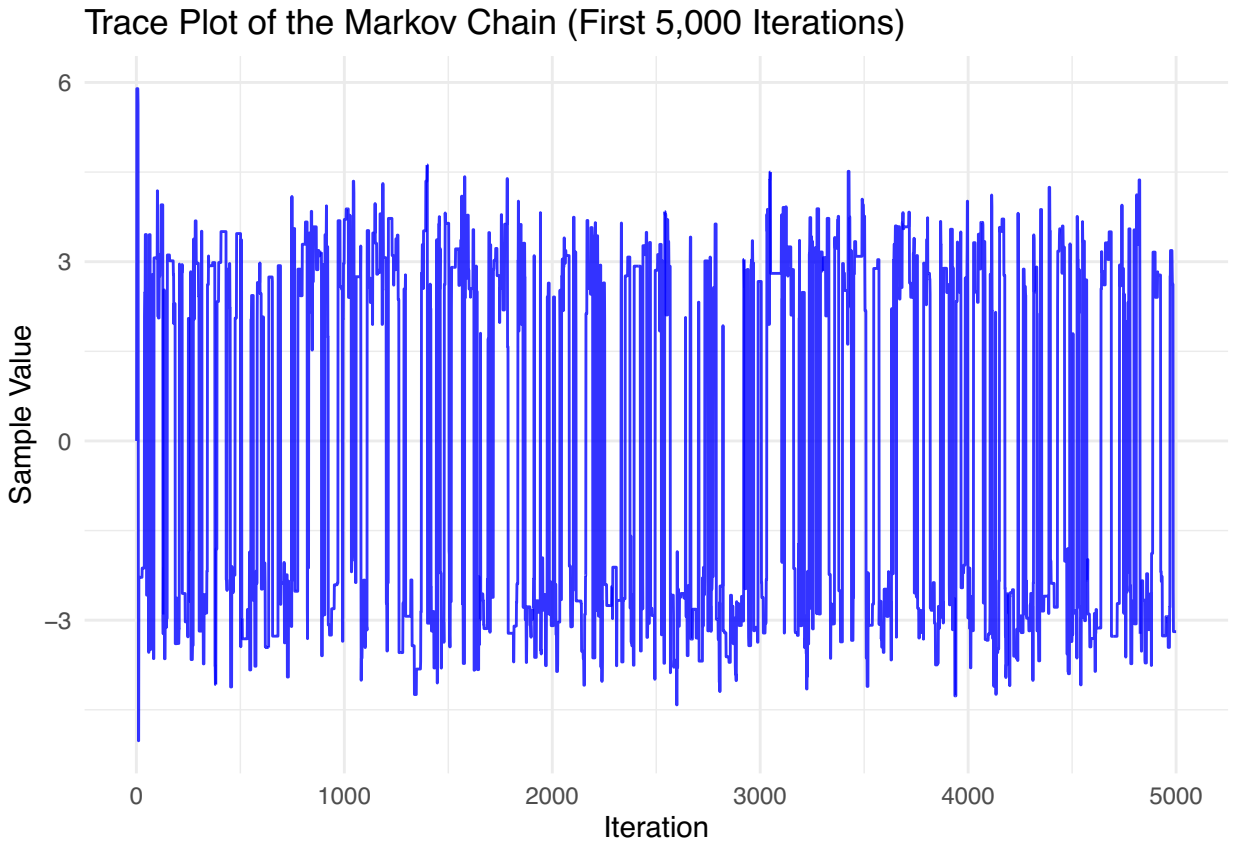
```
## Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(density)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```




```
#Trace Plot for all 1,000,000 iterations
trace_data <- data.frame(Iteration = 1:num_samples, Value = samples)

ggplot(trace_data, aes(x = Iteration, y = Value)) +
  geom_line(color = "blue", size = 0.5, alpha = 0.8) +
  labs(title = "Trace Plot of the Markov Chain",
       x = "Iteration",
       y = "Sample Value") +
  theme_minimal()
```




```
#Trace Plot for the first 5,000 iterations
trace_data <- data.frame(Iteration = 1:num_samples, Value = samples)
trace_data_subset <- trace_data[1:5000, ]

ggplot(trace_data_subset, aes(x = Iteration, y = Value)) +
  geom_line(color = "blue", size = 0.5, alpha = 0.8) +
  labs(title = "Trace Plot of the Markov Chain (First 5,000 Iterations)",
       x = "Iteration",
       y = "Sample Value") +
  theme_minimal()
```




```
#Expected value
expected_value <- mean(samples)
cat("Expected Value (E[X]):", round(expected_value, 4), "\n")
```

```
## Expected Value (E[X]): -0.001
```

```
#Variance
variance <- var(samples)
cat("Variance:", round(variance, 4), "\n")
```

```
## Variance: 9.25
```

```
#Probability between two points (for example, -pi to pi)
lower_bound <- -pi
upper_bound <- pi

probability_between <- mean(samples >= lower_bound & samples <= upper_bound)

cat("Probability P(-pi <= X <= pi):", round(probability_between, 10), "\n")
```

```
## Probability P(-pi <= X <= pi): 0.610052
```

```
# Step 5: Verify detailed balance condition and print both sides
verify_detailed_balance <- function(samples, target_density) {
  # Initialize counters for detailed balance check
  detailed_balance_check <- numeric(length(samples) - 1)

  # Determine the number of iterations to output (limit to the first 100)
  num_iterations_to_output <- min(100, length(samples) - 1)
```

```r
  # Loop through each pair of adjacent samples to check the detailed balance
  for (i in 1:num_iterations_to_output) {
    x_t <- samples[i]         # Current state
    x_c <- samples[i + 1]     # Proposed state (next state)

    # Calculate the acceptance ratio for moving from x_t to x_c
    alpha_tx <- min(1, target_density(x_c) / target_density(x_t))

    # Calculate the acceptance ratio for moving from x_c to x_t
    alpha_ct <- min(1, target_density(x_t) / target_density(x_c))

    # Compute both sides of the detailed balance equation
    lhs <- target_density(x_t) * alpha_tx
    rhs <- target_density(x_c) * alpha_ct

    # Print both sides for each comparison
    cat("Iteration", i, ":\n")
    cat("LHS = target_density(x_t) * alpha(x_t -> x_c) =", round(lhs, 6), "\n")
    cat("RHS = target_density(x_c) * alpha(x_c -> x_t) =", round(rhs, 6), "\n")
    cat("Discrepancy (LHS - RHS):", round(lhs - rhs, 6), "\n\n")

    # Store the maximum discrepancy
    detailed_balance_check[i] <- abs(lhs - rhs)
  }

  # Calculate and display the maximum discrepancy
  max_discrepancy <- max(detailed_balance_check)

  if (max_discrepancy < 1e-6) {
    cat("Detailed balance criterion is satisfied with a maximum discrepancy of", round(max_discrepancy,
  } else {
    cat("Detailed balance criterion is not satisfied with a maximum discrepancy of", round(max_discrepa
  }
}

# After generating samples with metropolis_hastings
verify_detailed_balance(samples, target_density)
```

```
## Iteration 1 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 2 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 3 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 4 :
```

```
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 5 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 6 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 7 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 8 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0
## Discrepancy (LHS - RHS): 0
##
## Iteration 9 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 9.3e-05
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 9.3e-05
## Discrepancy (LHS - RHS): 0
##
## Iteration 10 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.000115
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.000115
## Discrepancy (LHS - RHS): 0
##
## Iteration 11 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.000115
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.000115
## Discrepancy (LHS - RHS): 0
##
## Iteration 12 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 13 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 14 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
```

```
## Iteration 15 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 16 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 17 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 18 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 19 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 20 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 21 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 22 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 23 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 24 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 25 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
```

```
##
## Iteration 26 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.142706
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.142706
## Discrepancy (LHS - RHS): 0
##
## Iteration 27 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 28 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 29 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 30 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 31 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 32 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 33 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 34 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 35 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 36 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
```

```
## Discrepancy (LHS - RHS): 0
##
## Iteration 37 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.08709
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.08709
## Discrepancy (LHS - RHS): 0
##
## Iteration 38 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.077109
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.077109
## Discrepancy (LHS - RHS): 0
##
## Iteration 39 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.077109
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.077109
## Discrepancy (LHS - RHS): 0
##
## Iteration 40 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.232226
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.232226
## Discrepancy (LHS - RHS): 0
##
## Iteration 41 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.232226
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.232226
## Discrepancy (LHS - RHS): 0
##
## Iteration 42 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.232226
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.232226
## Discrepancy (LHS - RHS): 0
##
## Iteration 43 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 44 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 45 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 46 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 47 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
```

```
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 48 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.263687
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.263687
## Discrepancy (LHS - RHS): 0
##
## Iteration 49 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 50 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 51 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 52 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 53 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 54 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 55 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 56 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 57 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.280719
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.280719
## Discrepancy (LHS - RHS): 0
##
## Iteration 58 :
```

```
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.223229
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.223229
## Discrepancy (LHS - RHS): 0
##
## Iteration 59 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.223229
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.223229
## Discrepancy (LHS - RHS): 0
##
## Iteration 60 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.223229
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.223229
## Discrepancy (LHS - RHS): 0
##
## Iteration 61 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.25868
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.25868
## Discrepancy (LHS - RHS): 0
##
## Iteration 62 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.264582
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.264582
## Discrepancy (LHS - RHS): 0
##
## Iteration 63 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.264582
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.264582
## Discrepancy (LHS - RHS): 0
##
## Iteration 64 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.264582
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.264582
## Discrepancy (LHS - RHS): 0
##
## Iteration 65 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.264582
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.264582
## Discrepancy (LHS - RHS): 0
##
## Iteration 66 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.244744
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.244744
## Discrepancy (LHS - RHS): 0
##
## Iteration 67 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.244744
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.244744
## Discrepancy (LHS - RHS): 0
##
## Iteration 68 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.244744
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.244744
## Discrepancy (LHS - RHS): 0
##
```

```
## Iteration 69 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.329294
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.329294
## Discrepancy (LHS - RHS): 0
##
## Iteration 70 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.253607
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.253607
## Discrepancy (LHS - RHS): 0
##
## Iteration 71 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.253607
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.253607
## Discrepancy (LHS - RHS): 0
##
## Iteration 72 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.398553
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.398553
## Discrepancy (LHS - RHS): 0
##
## Iteration 73 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.398553
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.398553
## Discrepancy (LHS - RHS): 0
##
## Iteration 74 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.156567
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.156567
## Discrepancy (LHS - RHS): 0
##
## Iteration 75 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.156567
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.156567
## Discrepancy (LHS - RHS): 0
##
## Iteration 76 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.362445
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.362445
## Discrepancy (LHS - RHS): 0
##
## Iteration 77 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.362445
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.362445
## Discrepancy (LHS - RHS): 0
##
## Iteration 78 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.362445
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.362445
## Discrepancy (LHS - RHS): 0
##
## Iteration 79 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.362445
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.362445
## Discrepancy (LHS - RHS): 0
```

```
##
## Iteration 80 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.362445
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.362445
## Discrepancy (LHS - RHS): 0
##
## Iteration 81 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.174343
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.174343
## Discrepancy (LHS - RHS): 0
##
## Iteration 82 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.174343
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.174343
## Discrepancy (LHS - RHS): 0
##
## Iteration 83 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.174343
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.174343
## Discrepancy (LHS - RHS): 0
##
## Iteration 84 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.174343
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.174343
## Discrepancy (LHS - RHS): 0
##
## Iteration 85 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395641
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395641
## Discrepancy (LHS - RHS): 0
##
## Iteration 86 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395641
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395641
## Discrepancy (LHS - RHS): 0
##
## Iteration 87 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395641
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395641
## Discrepancy (LHS - RHS): 0
##
## Iteration 88 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395641
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395641
## Discrepancy (LHS - RHS): 0
##
## Iteration 89 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 90 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
```

```
## Discrepancy (LHS - RHS): 0
##
## Iteration 91 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 92 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 93 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 94 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 95 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 96 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 97 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 98 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 99 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.395613
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.395613
## Discrepancy (LHS - RHS): 0
##
## Iteration 100 :
## LHS = target_density(x_t) * alpha(x_t -> x_c) = 0.023772
## RHS = target_density(x_c) * alpha(x_c -> x_t) = 0.023772
## Discrepancy (LHS - RHS): 0
##
## Detailed balance criterion is satisfied with a maximum discrepancy of 0
```

Untitled

Henry Szklanny

2024-12-04

```r
#Additional code for Gelman-Rubin Diagnostic
library(coda)

#Here we define the number of chains
num_chains=4
#So we will have 4 chains, and here we we define the initial states for each chain
initial_states <- c(0, 4, -4, 10)

# Number of samples to generate per each chain
num_samples <- 1000000 #this is part of step 2
proposal_sd <- sqrt(2 * 9.25)  # Proposal standard deviation; same as before

#Define the bimodal target distribution
target_density <- function(x) {
  # Combine two normal distributions with equal weights
  w1 <- 0.5
  w2 <- 0.5
  mu1 <- -3
  sigma1 <- 0.5
  mu2 <- 3
  sigma2 <- 0.5
  w1 * dnorm(x, mean = mu1, sd = sigma1) + w2 * dnorm(x, mean = mu2, sd = sigma2)
}



#Now we need to define the Metropolis-Hastings function for each of the 4 chains
metropolis_hastings <- function(num_samples, proposal_sd, x0, target_density) {
  samples <- numeric(num_samples)
  samples[1] <- x0

  num_accepted <- 0

  for (t in 2:num_samples) {
    current_x <- samples[t - 1]  # Current value

    # Generate a proposal from the normal distribution
    proposed_x <- rnorm(1, mean = current_x, sd = proposal_sd)

    # Compute the acceptance ratio
    acceptance_ratio <- target_density(proposed_x) / target_density(current_x)
    acceptance_ratio <- min(1, acceptance_ratio)  # Ensure ratio is in [0, 1]
```

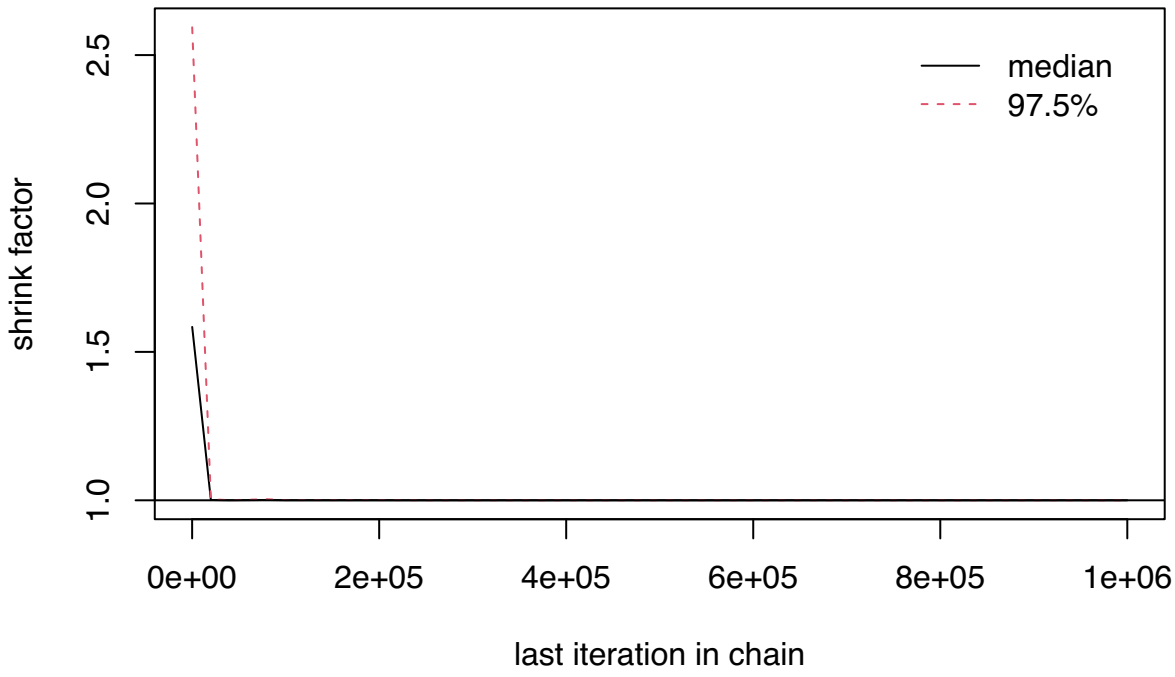
```
    # Acceptance or Rejection
    if (runif(1) < acceptance_ratio) {
      samples[t] <- proposed_x  # Accept the proposal
      num_accepted <- num_accepted + 1
    } else {
      samples[t] <- current_x  # Reject the proposal
    }
  }

  # Calculate and display the acceptance percentage
  acceptance_percentage <- (num_accepted / (num_samples - 1)) * 100
  cat("Acceptance Percentage:", round(acceptance_percentage, 2), "%\n")
  print(acceptance_percentage)
  # Return the generated samples
  return(samples)
}

#So for each chain, we run; run multiple chains with different initial states
samples_list <- list()
for (i in 1:num_chains) {
  # Run Metropolis-Hastings for each chain
  samples_list[[i]] <- metropolis_hastings(num_samples, proposal_sd, initial_states[i], target_density)
}
```

```
## Acceptance Percentage: 20.19 %
## [1] 20.19122
## Acceptance Percentage: 20.32 %
## [1] 20.31792
## Acceptance Percentage: 20.27 %
## [1] 20.26722
## Acceptance Percentage: 20.22 %
## [1] 20.22342
```

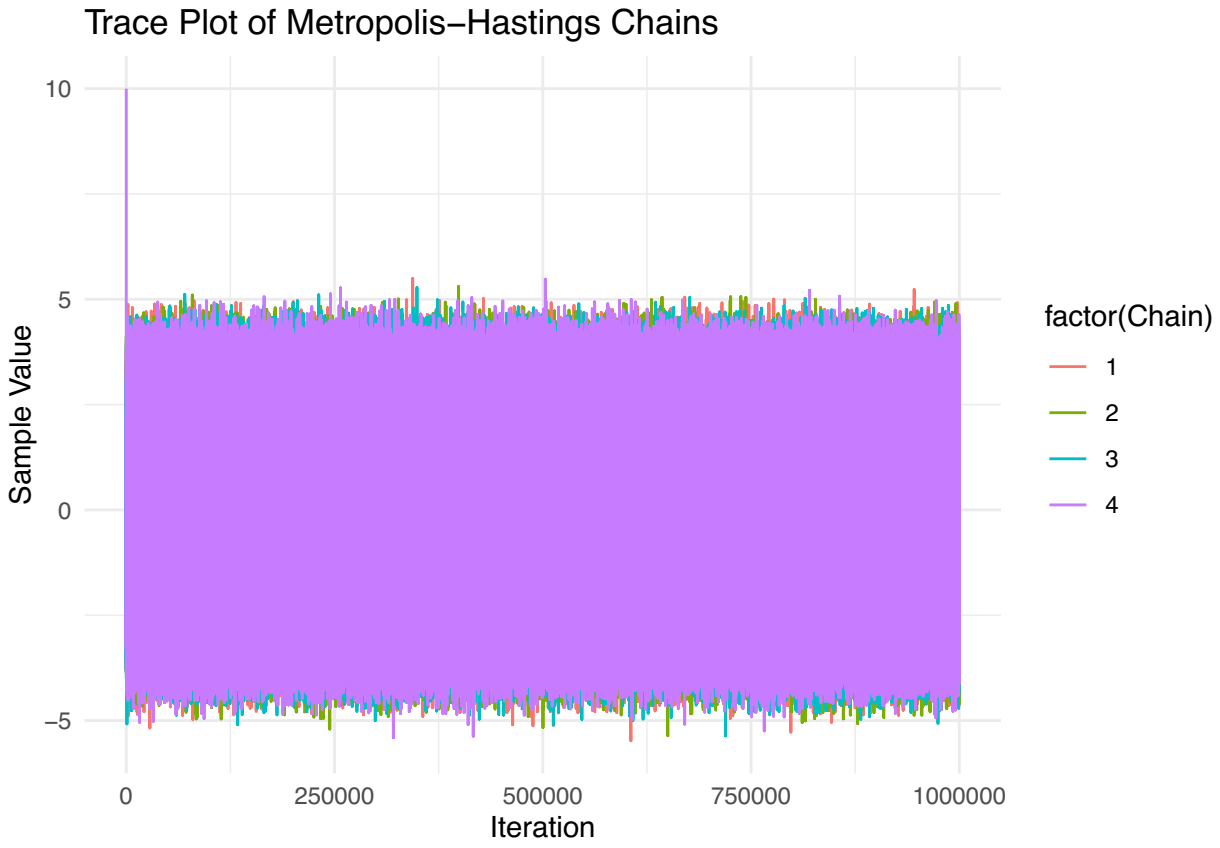
```
combined_samples <- mcmc.list(lapply(samples_list, mcmc))
gelman_rubin_diag <- gelman.diag(combined_samples)
print(gelman_rubin_diag)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]          1          1
```

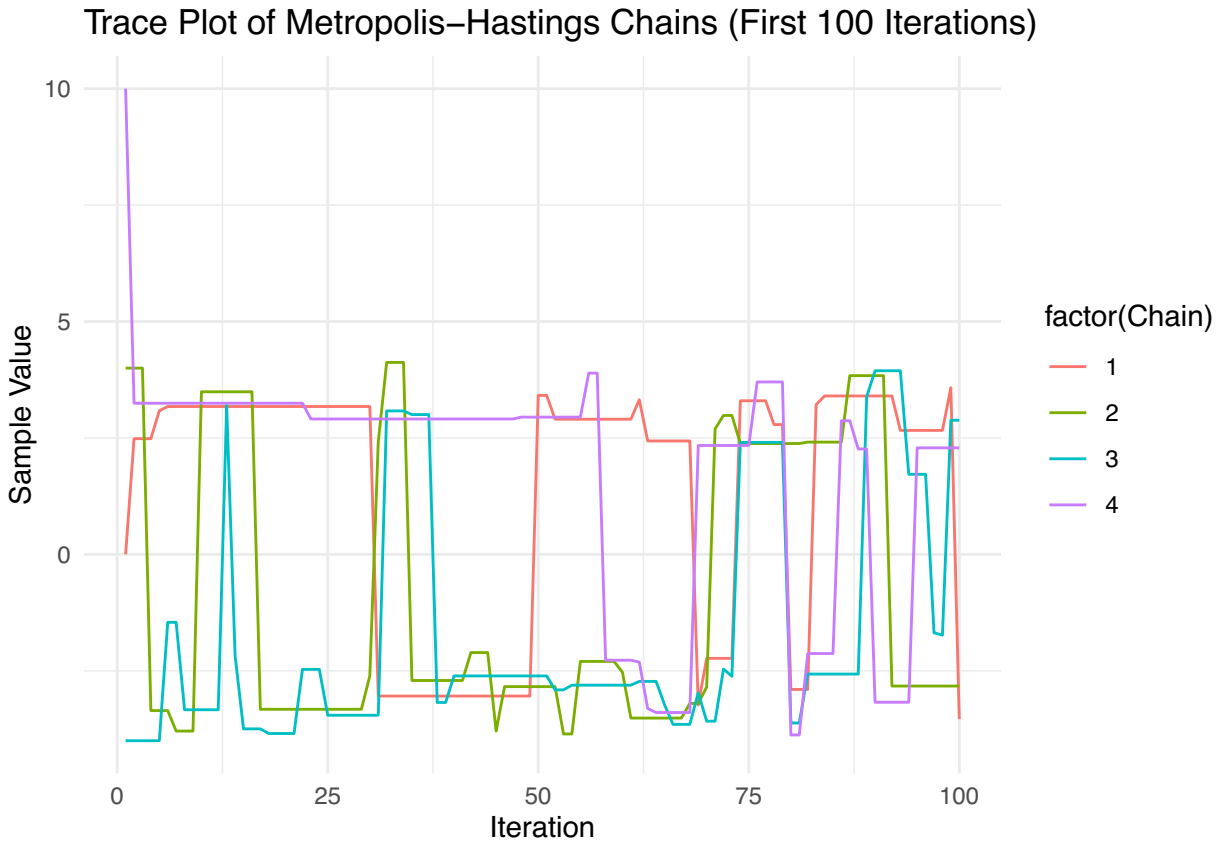
```
gelman.plot(combined_samples)
```




```r
library(ggplot2)

trace_data <- data.frame(Iteration = rep(1:num_samples, num_chains),
                         Value = unlist(samples_list),
                         Chain = rep(1:num_chains, each = num_samples))

ggplot(trace_data, aes(x = Iteration, y = Value, color = factor(Chain))) +
  geom_line() +
  labs(title = "Trace Plot of Metropolis-Hastings Chains",
       x = "Iteration",
       y = "Sample Value") +
  theme_minimal()
```




```r
# Plot the trace plot for the first 100 iterations
trace_data_subset <- trace_data[trace_data$Iteration <= 100, ]
ggplot(trace_data_subset, aes(x = Iteration, y = Value, color = factor(Chain))) +
  geom_line() +
  labs(
    title = "Trace Plot of Metropolis-Hastings Chains (First 100 Iterations)",
    x = "Iteration",
    y = "Sample Value"
  ) +
  theme_minimal()
```




```r
summary(combined_samples)
```

```
##
## Iterations = 1:1e+06
## Thinning interval = 1
## Number of chains = 4
## Sample size per chain = 1e+06
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##      -0.006239       3.041483       0.001521       0.006372
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
## -3.827 -3.000 -1.579  3.001  3.820
```

```r
# Unlist the combined samples to get a numeric vector
samples_combined <- unlist(samples_list)

# Calculate the probability that the sample lies between -pi and pi
lower_bound <- -pi
upper_bound <- pi

# Compute the proportion of samples within the specified range
probability_between <- mean(samples_combined >= lower_bound & samples_combined <= upper_bound)
```

```r
# Print the result
cat("Probability P(-pi <= X <= pi):", round(probability_between, 10), "\n")
```

```
## Probability P(-pi <= X <= pi): 0.6111697
```