Statistical inference for SIS stochastic model: MLE and MH Tutorial in R

Dr Panayiota Touloupou

July 15, 2025

Abstract

Welcome to this R tutorial on fitting the Susceptible-Infected-Susceptible (SIS) model, considering it as a stochastic epidemic process based on the Binomial sampling scheme that have seen in our lectures. In this tutorial we will use two statistical methods: Maximum Likelihood Estimation (MLE) and Metropolis-Hastings (MH). This tutorial is designed for students new to R and will guide you through the process step-by-step.

1 Getting started with R (for students new to R)

R is a programming language and free software environment for statistical computing and graphics.

1.1 Install R and RStudio

We assume you don't yet have R on your computer. To get started, you will need to install two pieces of software:

- R, the actual programming language.
  - R has a homepage, `http://r-project.org/`, but the software itself is located for download on the CRAN, which you can find at `http://cran.r-project.org/`.
  - Chose your operating system (Macintosh, Windows, Linux or Unix) and select to install the most recent version.
- RStudio (you must have R installed to use RStudio), for working with R.
  - RStudio is simply an interface used to interact with R.
  - Install RStudio from `http://rstudio.org`.

1.2 RStudio interface (Brief Overview)

The RStudio screen is divided into 4 windows:

- Bottom left: **console** window (also called command window). This is the most important window since it is the active bit, where you can communicate directly with R. Here you can type simple commands after the " > " prompt and R will then execute your command. It is also where your non-graphical results appear.
- Top left: **editor** window (also called script window). Collections of commands (scripts) that can be edited and saved. Each script is basically just a text file where you can write longer sections of code. Also this area is a place where you can view datasets.
- Top right: environment and history window. In the environment window (also called workspace window) you can see what is in R's memory, i.e. a list of data and variables, which contain values that R has been told to save from previous commands. The history window shows a history of commands that have been typed on the console.
- Bottom right: files/plots/packages/help window. Here you can do everything else - open files, look at and manipulate plots, install and load packages or use the help function and read help files.

1.3 Basic R syntax

Let's try some basic commands. Type these into the **console** and press Enter (or into **editor** window and press the button Run).

```r
# This is a comment. R ignores anything after a '#' on a line.

# Assigning values to variables
x <- 10
y = 5 # ' = ' also works for assignment, but '<-' is preferred in R
print(x + y)

# Vectors (ordered collections of values of the same type)
my_vector <- c(1, 2, 3, 4, 5) # 'c()' combines values into a vector
print(my_vector)
print(my_vector * 2)
print(sum(my_vector))

# Functions
my_function <- function(a, b) {
  result <- a * b
  return(result)
}
print(my_function(x, y))

# Data frames (like tables or spreadsheets)
my_data <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 22),
  City = c("NY", "LA", "Chicago")
)
print(my_data)
print(my_data$Age) # Accessing a column by name
print(my_data[1, ]) # Accessing the first row
```

1.4 Installing and loading packages

R's power comes from its vast collection of packages. Packages are collections of functions, data, and compiled code in a well-defined format, designed to extend R's capabilities. Think of them as add-ons that provide specialized tools. With the standard R installation, most common packages are installed. At some point in the future if you want to use some functions that are not in the ordinary R installation, there is a good chance that there is a package that will fulfill your needs.

```
# Install packages (only need to do this once per R installation)
install.packages("ggplot2") # For nice plots
install.packages("coda")    # For MCMC diagnostics

# Load packages (need to do this every time you start a new R session
   and want to use them)
library(ggplot2)
library(coda)
```

1.5 Basic plotting

Visualizing data is crucial in statistics and modelling. R has built-in plotting functions, and packages like `ggplot2` offer more advanced and aesthetically pleasing options.

```
# Simple plot using R's base plotting system
# 'plot() ' is a versatile function, 'type = "l"' means draw lines.
# 'main ', 'xlab ', 'ylab ' are for setting the title and axis labels.
plot(my_vector, type = "l", main = "My Simple Line Plot",
  xlab = "Index", ylab = "Value")
```

2 The stochastic SIS model

The SIS model is an epidemic model where individuals can be in one of two states: Susceptible (S) or Infected (I). Infected individuals recover and immediately become susceptible again (no immunity).

Unlike deterministic models that describe average behavior with continuous equations, stochastic models incorporate randomness, reflecting that events (like infections or recoveries) happen probabilistically rather than with fixed rates. This is especially important in smaller populations where random fluctuations can significantly impact the epidemic trajectory.

2.1 Model events and probabilities

In this specific stochastic SIS model, we consider two types of events occurring at each discrete time step (e.g., daily, weekly):

1. **Number of new infections (X_t):** This is the number of susceptible individuals who become infected during the current time step.

- The number of new infections X_t is drawn from a Binomial distribution: $X_t \sim \text{Binomial}(S_t, p_{\text{inf}})$.
- The probability of infection for a single susceptible individual, p_{inf}, is given by: $1 - e^{-\frac{\beta I_t}{N}}$.
- A higher β or more I_t leads to a higher chance of infection.

2. **Number of new recoveries (Y_t):** This is the number of infected individuals who recover and return to the susceptible state during the current time step.
   - The number of new recoveries Y_t is drawn from a Binomial distribution: $Y_t \sim \text{Binomial}(I_t, p_{\text{rec}})$.
   - The probability of recovery for a single infected individual, p_{rec}, is given by: $1 - e^{-\gamma}$.
   - A higher γ means individuals recover faster.

Where:

- S_t and I_t are the number of susceptible and infected individuals at time t.
- $N = S_t + I_t$ is the total population size (assumed constant throughout the simulation).
- β is the transmission rate (a positive value).
- γ is the recovery rate (a positive value).

2.2 Implementing the stochastic SIS model in R

Now, let's translate these probabilistic rules into an R function. This function will simulate the SIS model step-by-step over a given time period.

Note that for the SIS model, calculating the likelihood based only on the total number of susceptible or infected individuals at each time point is not sufficient. We need additional information. This is because individuals can recover and return to the susceptible state. If we only observe S_t or I_t, we cannot tell whether a change in S (or I) is due to a new infection or a recovery.

For example, if S decreases, it could be due to an infection; but if I also decreases while S increases, it could be a recovery. Without explicitly tracking the number of new infections (`new_inf`) and recoveries (`new_rec`) at each time step, the likelihood function cannot correctly attribute the changes to the underlying processes. Therefore, we need to modify our simulation function (compared to the one we used in lectures) to return `new_inf_history` and `new_rec_history` to provide this crucial information for the likelihood calculation.

```r
# Define the Stochastic SIS model simulation function
# This function takes:
# N: Total population size
# beta: Transmission rate
```

```r
# gamma: Recovery rate
# I0: Initial number of infected individuals
# T: Total number of time steps to simulate

simulate_SIS <- function(N, beta, gamma, I0, T) {
  # Initialize vectors to store the number of Susceptible (S)
  # and Infected (I) individuals
  S <- numeric(T)
  I <- numeric(T)

  # Initialize vectors to store the *number of new infections* and
  # *new recoveries* that occur at each time step. These will be
  # crucial for calculating the likelihood function.
  new_inf_history <- numeric(T)
  new_rec_history <- numeric(T)

  # Set initial conditions for S and I at time t=0
  S[1] <- N - I0
  I[1] <- I0
  new_inf_history[1] <- 0 # No new events at time 0
  new_rec_history[1] <- 0 # No new events at time 0

  # Loop through each time step from t=1 to T-1
  for (t in 1:(T-1)) {
    # Calculate the probability of infection for a susceptible
    # individual based on the current number of infected individuals
    p_inf <- 1 - exp(-beta*I[t]/N)

    # Calculate the probability of recovery for an infected individual
    p_rec <- 1 - exp(-gamma)

    # Simulate number of events in this time step using binomial
    # distribution
    # rbinom(n, size, prob) generates n random values from a binomial
    # distribution size is the number of trials, prob is the
    #  probability of  success on each trial
    new_inf <- rbinom(1, S[t], p_inf)
    new_rec <- rbinom(1, I[t], p_rec)

    # Store the simulated number of events for this time step
    new_inf_history[t+1] <- new_inf
    new_rec_history[t+1] <- new_rec

    # Update the number of Susceptible and Infected individuals for the
    # next time step (t+1)
    # S decreases by new infections and increases by new recoveries
    S[t + 1] <- S[t] - new_inf + new_rec
    # I increases by new infections and decreases by new recoveries
    I[t + 1] <- I[t] + new_inf - new_rec
  }
  # Return a data frame
  return(data.frame(time = 0:(T-1), S = S, I = I,
    new_inf_obs = new_inf_history, new_rec_obs = new_rec_history))
}
```

2.3 Simulating the stochastic SIS model

Let's set some parameters and initial conditions and simulate the model over time.

```r
# Set a seed for reproducibility of stochastic results.
# This ensures that if you run the code multiple times with the
# same seed, you will get the exact same sequence of random
# numbers, making your results reproducible.
set.seed(3)

# Simulate data using the 'simulate_SIS' function.
# N: Total population size
# beta: True transmission rate
# gamma: True recovery rate
# I0: Initial number of infected individuals
# T: Total number of time steps (e.g., days) for the simulation
SISdata <- simulate_SIS(N = 1000, beta = 0.3, gamma = 0.1, I0 = 10,
  T = 150)

# View the first few rows of the simulation output.
head(SISdata)

# Plotting the simulation results
plot(SISdata$time, SISdata$I, type = "l", col = "blue",
   ylab = "Number of individuals", xlab = "Time", ylim = c(0, 1000))
lines(SISdata$time, SISdata$S, col = "green")

# Add legend
legend("topright", legend = c("Infectious", "Susceptible"),
       lty = c(1, 1), col = c("blue", "green"))
```

2.4 Student activity:

- **Observe variability:** Run the simulation code several times. How do the individual simulation paths differ, even with the same parameters? This is the essence of stochasticity!
- **Experiment with parameters:** Change `beta` and `gamma` values in the simulation code. How do they influence the average behavior (e.g., the peak number of infected individuals, the endemic equilibrium) and the variability of the epidemic?
- **Change initial conditions:** Vary `I0` (initial infected). Does it affect the long-term behavior or the initial spread? What happens if you start with very few infected individuals (e.g., `I0 = 1`)?
- **Change population size `N`:** Does changing `N` (while keeping `I0` proportional or constant) alter the magnitude of stochastic fluctuations? **Comment:** In smaller populations, stochastic effects are often more pronounced. Why do you think this is the case? Consider what happens when counts become very small.

3 Maximum Likelihood Estimation (MLE) for stochastic SIS

Maximum Likelihood Estimation (MLE) is a widely used statistical method for estimating the parameters of a model. The core idea is to find the parameter values that make the observed data most "likely" to have occurred under the assumed model, i.e. the values that maximize the likelihood function (which is the probability of observing the given data under the model).

To test our MLE approach, we first need some "observed" data. In a real-world scenario, this would be actual epidemic data (e.g., daily reported cases). For this tutorial, we will generate synthetic data from our stochastic SIS model using known "true" parameters. We then pretend we don't know these true parameters and use MLE to try and recover them. The stochastic nature of the model itself will introduce the "noise" or variability that we would typically see in real data.

3.1 Generating Synthetic Data

To test our MLE approach, we'll first generate some "observed" data from our stochastic SIS model. The stochastic nature itself will provide the "noise".

```
# True parameters (these are what we want to estimate)
true_beta <- 0.3
true_gamma <- 0.1

# Initial conditions and time for data generation
N_data <- 1000
I0_data <- 5
S0_data <- N_data - I0_data
T_data <- 150

# Simulate the true SIS stochastic model to generate observed data
set.seed(123) # For reproducibility
observed_data <-  simulate_SIS(N = N_data, beta = true_beta,
  gamma = true_gamma, I0 = I0_data,  T = T_data)
```

3.2 Defining the log likelihood function for SIS

The likelihood of observing a specific number of new infections (X_t) at time t is given by a Binomial distribution with parameters S_t and p_{inf} (probability of infection). Similarly, for new recoveries (Y_t), it's a Binomial distribution with parameters I_t and p_{rec} (probability of recovery).

The total log-likelihood is the sum of the log-likelihood of all observed events across all time steps.

```
# Define the log-likelihood Function for the SIS model
# This function takes:
# params: A vector of parameters (e.g., c(beta, gamma))
```

```r
# data: The observed data.

loglik_SIS <- function(params, data) {
  beta = params[1]
  gamma = params[2]

  # Ensure parameters are valid (e.g., positive).
  # If any parameter is non-positive, return -Inf for the
  # log-likelihood.
  # This tells the optimizer that these parameters are invalid.
  if (beta <= 0 || gamma <= 0) {
     return(-Inf)
   }

  loglik <- 0 # Initialize the total log-likelihood to zero
  # Calculate the total population size from the initial states
  N <- data$S[1] + data$I[1]

  # Loop through each time step in the observed data
  for (t in 1:(nrow(data) - 1)) {

    # Get the number of Susceptible and Infected individuals
    # at time t and t+1
    St <- data$S[t]
    It <- data$I[t]

    # Get the observed number of new infections and new recoveries
    # that occurred between time t and time t+1. These come directly
    # from our observed data.
    new_inf <-  data$new_inf_obs[t + 1] # Events observed at t+1,
                                        # from state at t
    new_rec <- data$new_rec_obs[t + 1] # Events observed at t+1, from
                                       # state at t

    # Calculate probabilities
    p_inf <- 1 - exp(-beta*It/N)
    p_rec <- 1 - exp(-gamma)

    # Avoid probabilities of exactly 0 or 1, which can cause log(0) or
    # log(1-1) issues in hte dbinom function.
    # We clip them to be slightly away from the boundaries.
    p_inf <- min(max(p_inf, 1e-10), 1 - 1e-10)
    p_rec <- min(max(p_rec, 1e-10), 1 - 1e-10)

    # Add log-likelihood contribution for this step
    # dbinom(x, size, prob, log = TRUE) gives the log of the
    # binomial probability mass function.
    loglik <- loglik +
      dbinom(new_inf, St, p_inf, log = TRUE) +
      dbinom(new_rec, It, p_rec, log = TRUE)
  }
  return(loglik) # Return the total log-likelihood
}
```

3.3 Performing MLE using `optim`

R's `optim` function is a powerful general-purpose optimization tool. It attempts to find the set of parameters that minimize a given function. In our case, it will maximize the log-likelihood by using `control = list(fnscale = -1)`, thereby finding the parameters that maximize the likelihood of our observed data. Write `?optim` in console to find more details.

```
?optim

# Set Initial guesses for parameters
# These are the starting points for the optimizer's search.
initial_params  <- c(beta = 0.2, gamma = 0.05)

# Run the optimization using 'optim()'.
# 'par': The initial guesses for the parameters.
# 'fn': The function to be optimized (our log-likelihood function).
# 'data': Our simulated 'SISdata' that contains the observed S, I,
# new_inf_obs, new_rec_obs.
# 'control = list(fnscale = -1)': This is crucial! It tells 'optim'
# to maximize 'fn' instead of minimizing it.
mle_result <- optim(par = initial_params,
                    fn = loglik_SIS, data = observed_data,
                    control = list(fnscale = -1))
print("MLE Results (initial run):")
print(mle_result)
# Look for 'convergence = 0' which indicates successful convergence.
# 'par' will give you the estimated parameter values.
# 'value' is the maximum log-likelihood found.

# Extract estimated parameters
estimated_beta_mle <- mle_result$par[1]
estimated_gamma_mle <- mle_result$par[2]

cat("\nEstimated Beta (MLE):", estimated_beta_mle, "\n")
cat("Estimated Gamma (MLE):", estimated_gamma_mle, "\n")
cat("True Beta:", true_beta, "\n")
cat("True Gamma:", true_gamma, "\n")
```

3.4 Repeated MLE estimation and variability

In the previous section, you performed MLE once. However, because our data is generated from a stochastic model, each time you simulate data, you'll get a slightly different trajectory. This means that if you run the MLE process on different simulated datasets, you'll get slightly different parameter estimates.

Let's explore this variability by repeating the data simulation and MLE estimation process multiple times.

```
# Define the number of times to repeat the MLE process
num_repetitions <- 25

# Create empty vectors to store the estimated beta and gamma from
```

```r
#  each repetition
estimated_betas <- numeric(num_repetitions)
estimated_gammas <- numeric(num_repetitions)

# Set the fixed parameters for data generation for this exercise
fixed_N <- 1000
fixed_I0 <- 10
fixed_T <- 150
fixed_true_beta <- 0.3
fixed_true_gamma <- 0.1

# Loop to repeat the simulation and MLE estimation
for (i in 1:num_repetitions) {
  # Set a unique seed for each repetition to get different stochastic
  # data realizations
  set.seed(100 + i)

  # Simulate new SIS data for this repetition
  current_SISdata <- simulate_SIS(N = fixed_N, beta = fixed_true_beta,
    gamma = fixed_true_gamma,  I0 = fixed_I0, T = fixed_T)

  # Run MLE for the current simulated data
  # Use the same initial guesses and bounds as before
  current_mle_result <- optim(par = initial_params,
                              fn = loglik_SIS, data = current_SISdata,
                              control = list(fnscale = -1))

  # Store the estimated parameters
  estimated_betas[i] <- current_mle_result$par[1]
  estimated_gammas[i] <- current_mle_result$par[2]

}

# Now, visualize the distribution of your 25 estimates using boxplots.
# Boxplots are great for showing the median, quartiles, and outliers
# of a distribution.
# Set up a 1x2 plotting layout
par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))

# Boxplot for estimated Beta values
boxplot(estimated_betas, main = "Distribution of Estimated Beta",
        ylab = "Estimated Beta", col = "lightblue", border = "black")
# Add a line for the true beta value
abline(h = fixed_true_beta, col = "red", lwd = 2, lty = 2)

# Boxplot for estimated Gamma values
boxplot(estimated_gammas, main = "Distribution of Estimated Gamma",
        ylab = "Estimated Gamma", col = "lightgreen", border = "black")
# Add a line for the true gamma value
abline(h = fixed_true_gamma, col = "red", lwd = 2, lty = 2)

# Reset plotting layout
par(mfrow = c(1, 1))
```

3.5 Student Activities for MLE:

- Execute the "Repeated MLE Estimation and Boxplots" code block.

- Observe the boxplots:
  - Where is the median (the line inside the box) of your boxplots located relative to the true parameter values (red dashed line)?
  - How wide are the boxes? This indicates the interquartile range (middle 50% of your estimates).
  - Are there any "outliers" (points beyond the whiskers)?
  - **Comment:** What does the spread of the boxplots tell you about the precision of your MLE estimates for this stochastic model? How does this relate to the concept of statistical uncertainty?
- Vary `num_repetitions`: Try increasing `num_repetitions`: (e.g., to 100 or 500). Does the distribution of estimates change? Does it become more concentrated around the true value?
- Vary `N` (Population Size): Change `N` (e.g., to 100, 1000, 10000) when generating the data, then re-run the "Repeated MLE Estimation and Boxplots" section.

  Comment: How does increasing the population size (`N`) affect the variability (spread) of your boxplots? Does more data generally lead to more precise estimates for stochastic models?
- Vary `I0` (Initial Infected): Change `I0` (e.g., to 1, 50, 100) when generating the data, then re-run the "Repeated MLE Estimation and Boxplots" section.

  Comment: How does the initial number of infected individuals affect the estimation? Is it harder to estimate parameters if `I0` is very small (e.g., leading to early extinction of the epidemic in some stochastic runs)?

4 Metropolis-Hastings (MH) for stochastic SIS

While MLE provides a single "best" point estimate for parameters, it doesn't directly tell us about the uncertainty around those estimates. Metropolis-Hastings (MH) is a Markov Chain Monte Carlo (MCMC) algorithm that allows us to sample from complex probability distributions, particularly the posterior distribution in Bayesian inference. Instead of a single estimate, MH gives us a distribution of plausible parameter values, which is incredibly valuable for understanding uncertainty.

4.1 Introduction to Bayesian Inference (Recap)

- **Prior distribution $p(\theta)$:** Our beliefs about the parameters ($\theta = (\beta, \gamma)$) before seeing the data.
- **Likelihood function $p(Data|\theta)$:** The probability of observing the data given the parameters (same as in MLE, but now based on the stochastic model).
- **Posterior distribution $p(\theta|Data)$:** Our updated beliefs about the parameters after seeing the data. It's proportional to Prior $\times$ Likelihood.

The MH algorithm constructs a Markov chain whose stationary distribution is the target posterior distribution. By running the chain for a long time, the samples generated will approximate samples from the posterior.

4.2 Implementing the MH Algorithm

The core idea of MH is to propose a new set of parameters, calculate an acceptance ratio, and either accept or reject the new parameters.

```r
# Log-Prior Function (assuming exponential priors)
log_prior <- function(beta, gamma, lambda_beta = 1, lambda_gamma = 1) {
  dexp(beta, rate = lambda_beta, log = TRUE) +
    dexp(gamma, rate = lambda_gamma, log = TRUE)
}


# Metropolis-Hastings Algorithm
MH_sampler_SIS <- function(data, n_iter, beta_init, gamma_init,
                           lambda_beta, lambda_gamma, proposal_sd) {

  # Initialize current parameters
  beta <- beta_init
  gamma <- gamma_init
  samples <- matrix(NA, n_iter, 2)

  # Counter for accepted proposals
  accepted_count <- 0

  for (i in 1:n_iter) {
    beta_prop <- rnorm(1, beta, proposal_sd)
    gamma_prop <- rnorm(1, gamma, proposal_sd)

    if (beta_prop > 0 && gamma_prop > 0) {
      loglik_curr <- loglik_SIS(c(beta, gamma), data)
      loglik_prop <- loglik_SIS(c(beta_prop, gamma_prop), data)

      logprior_curr <- log_prior(beta, gamma, lambda_beta,
            lambda_gamma)
      logprior_prop <- log_prior(beta_prop, gamma_prop, lambda_beta,
            lambda_gamma)

      log_accept_ratio <- (loglik_prop + logprior_prop) -
          (loglik_curr + logprior_curr)

      if (log(runif(1)) < log_accept_ratio) {
       # accept
        beta <- beta_prop
        gamma <- gamma_prop
        accepted_count <- accepted_count + 1
      }
    }

    # Store the current (accepted or re-used) parameters in the chain
    samples[i, ] <- c(beta, gamma)
  }

    cat("Acceptance Rate:", accepted_count / n_iter, "\n")

  colnames(samples) <- c("beta", "gamma")
  return(as.data.frame(samples))
}
```

4.3 Running the MH Sampler

Now, let's run the Metropolis-Hastings algorithm to generate samples from the posterior distribution of our β and γ parameters.

```
# True parameters (these are what we want to estimate)
true_beta <- 0.3
true_gamma <- 0.1

# Initial conditions and time for data generation
N_data <- 1000
I0_data <- 5
S0_data <- N_data - I0_data
T_data <- 150

# Simulate the true SIS stochastic model to generate observed data
set.seed(123) # For reproducibility
observed_data <-  simulate_SIS(N = N_data, beta = true_beta,
          gamma = true_gamma, I0 = I0_data,  T = T_data)


# Run the MH sampler
mh_chain <- MH_sampler_SIS(data = observed_data,
      n_iter = 10000, beta_init = 0.2,  gamma_init = 0.2,
      lambda_beta = 1, lambda_gamma = 1,  proposal_sd = 0.01)
```

4.4 Analyzing MCMC Output

It's crucial to analyze the MCMC chain to ensure it has converged and is sampling effectively.

```
# Discard burn-in period (e.g., first 10% of iterations)
burn_in <- n_iter * 0.1
mh_chain_post_burnin <- mh_chain[-(1:burn_in), ]

#install.packages("coda")
library("coda")

# Convert to 'mcmc' object for coda package functions
mcmc_object <- as.mcmc(mh_chain_post_burnin)

# 1. Trace Plots: Show the values of parameters over iterations.
# Should look like "fuzzy caterpillars".
plot(mcmc_object)

# 2. Summary Statistics: Mean, median, credible intervals
# (similar to confidence intervals).
summary(mcmc_object)

# 3. Autocorrelation Plots: Show correlation between
# samples at different lags. Should drop quickly.
autocorr.plot(mcmc_object)

# 4. Effective Sample Size (ESS): How many independent samples you
```

```r
# effectively have. Higher is better.
effectiveSize(mcmc_object)

# Compare MH estimates (mean of posterior) with MLE and true values
cat("\nMH Posterior Means:\n")
print(colMeans(mh_chain_post_burnin))
cat("\nMLE Estimates:\n")
print(mle_result$par)
cat("\nTrue Parameters:\n")
print(c(beta = true_beta, gamma = true_gamma))
```

4.5 Student Activities for MH:

1. **Multiple Chains:**
   - It's good practice to run multiple independent MCMC chains from different starting points to check for convergence.
   - Run the `MH_sampler_SIS` function multiple times (e.g., 2-3 times), each with a different `set.seed()` and different initial parameters, `beta_init` and `gamma_init`.
   - Combine the chains (e.g., using `rbind` or `mcmc.list` from `coda`) and plot them together using `plot(as.mcmc(combined_chain))`.
   - **Comment:** Do the chains converge to the same region of the parameter space? What does this tell you about the reliability of your MCMC results and whether the algorithm has found the true posterior?
2. **Different proposal distributions:**
   - Experiment with `proposal_sd` values (e.g., `c(0.001, 0.0005)` for smaller steps, or `c(0.05, 0.02)` for larger steps).
   - Observe the "Acceptance Rate" printed by the function.
   - **Comment:** How does the `proposal_sd` affect the acceptance rate? What happens to the trace plots and autocorrelation plots if the acceptance rate is very low or very high? (Aim for an acceptance rate between 20-50%).
3. **Comparing MLE and MH:**
   - Compare the point estimates from MLE (`mle_result$par`) with the mean/median of the posterior distributions from MH (`colMeans(mh_chain_post_burnin)` or `summary(mcmc_object)`).
   - **Comment:** Are they similar? What are the advantages of having a full posterior distribution (from MH) compared to a single point estimate (from MLE) when dealing with stochastic models and uncertainty? (Hint: Think about how each method quantifies or represents uncertainty).

Experiment with the code and the suggested activities to deepen your understanding!!!