HW 3

Group 3

2025-03-04

Question 1

Theoretical Background

Contour Plots

A **contour plot** is a graphical representation of a three-dimensional surface in two dimensions. It consists of contour lines, which connect points of equal function values. In the context of bivariate density estimation, contour plots are used to visualize the density function of a two-dimensional dataset.

Given a probability density function $f(x, y)$, the contour lines represent regions of constant density. Areas with closely spaced contour lines indicate steep changes in density, while widely spaced lines suggest gradual variations.

Bivariate Unimodal and Bimodal Distributions

A **bivariate distribution** describes the joint probability distribution of two random variables X and Y. The shape of its density function determines whether the distribution is **unimodal** or **bimodal**.

- **Unimodal Distribution**: The density function has a single peak, meaning the data clusters around one central region. An example is the **bivariate normal distribution**, where most of the data points are concentrated around the mean.
- **Bimodal Distribution**: The density function has two distinct peaks, indicating two separate regions of high data concentration. This often occurs when the data is generated from two different underlying processes or mixtures of two distributions.

A **multimodal distribution** extends this concept to more than two peaks.

Detecting Modes from Contour Plots

A **mode** in a distribution is a point where the density function attains a local maximum. In a **bivariate distribution**, modes correspond to density peaks. Contour plots help in detecting modes as regions where enclosed contour lines form distinct, separate peaks.

- **Unimodal Distribution:** A single peak appears in the contour plot.
- **Bimodal Distribution:** Two distinct peaks are present.
- **Multimodal Distribution:** More than two peaks can be observed.

While contour plots provide a visual method for identifying modes, other techniques like **Gaussian Mixture Models (GMM)** and **Kernel Density Estimation (KDE)** can also be used for quantitative mode detection.

In this report, we will analyze different bivariate datasets to study their modality, first through visual inspection of contour plots and then using numerical methods for mode estimation.

1.Simulating Bivariate Unimodal and Bimodal Distributions

A **bivariate unimodal distribution** has a single peak in its density function. The **bivariate normal distribution** $(X, Y) \sim \mathcal{N}(\mu, \Sigma)$ is a common example, where μ is the mean vector and Σ is the covariance matrix.

A **bimodal distribution** has two distinct peaks and can be modeled using a **mixture of two normal distributions**:

$$f(X, Y) = \lambda f_1(X, Y) + (1 - \lambda) f_2(X, Y)$$

where f_1, f_2 are normal densities and λ is a mixing weight.

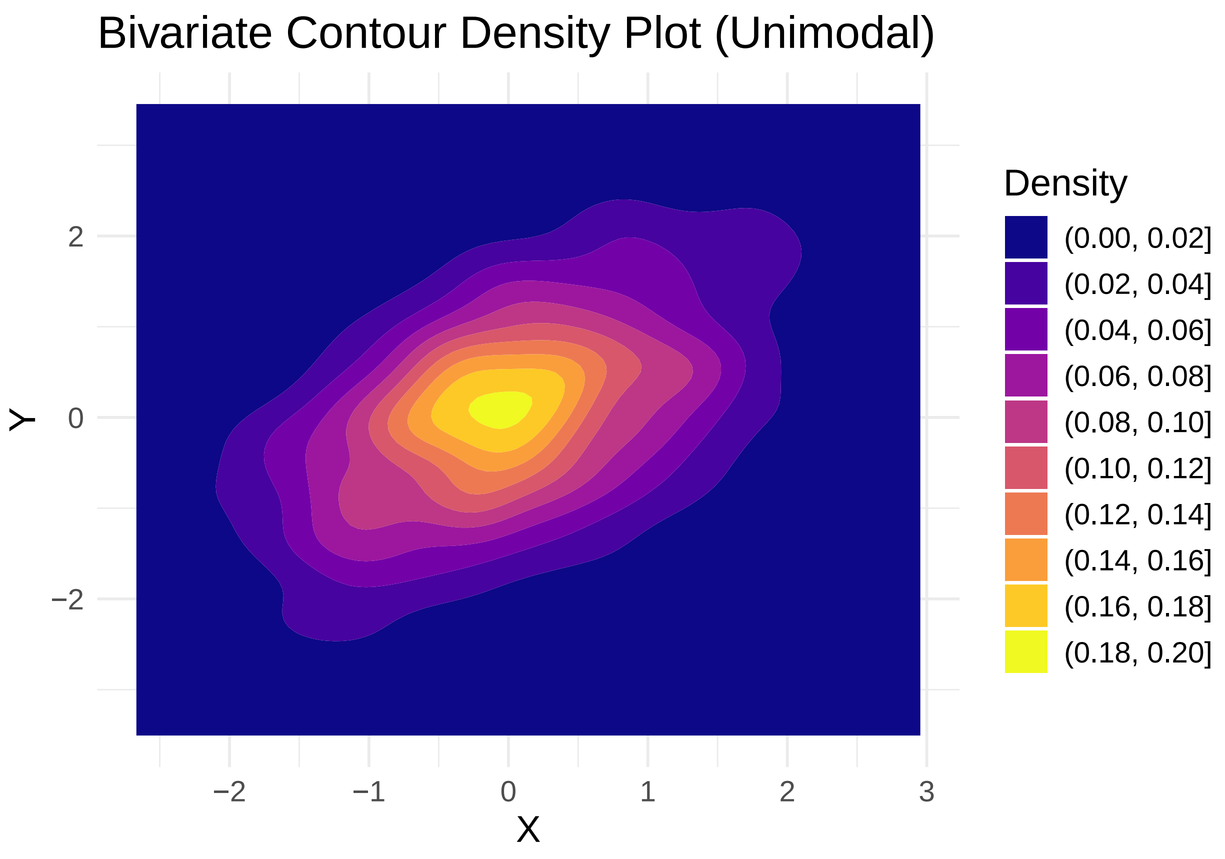
To detect modes, we use **contour plots**: - A **unimodal** distribution has one peak. - A **bimodal** distribution shows two separate peaks.

Contour Plot for Bivariate Unimodal Distribution

```r
# Unimodal distribution contour plot code here
# Load necessary libraries
library(MASS)
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.4.1
```

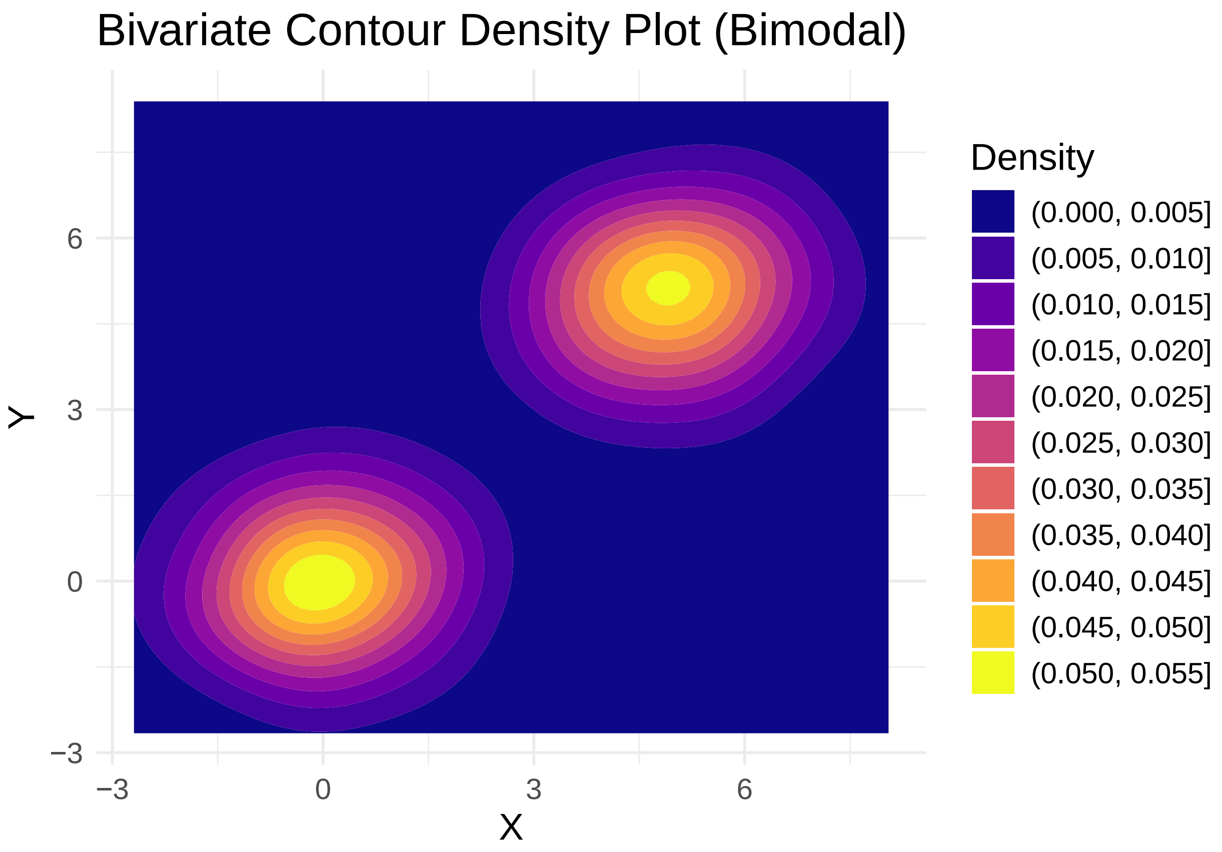
```r
# Function to generate a bivariate normal distribution
generate_bivariate_normal <- function(n, mu, sigma) {
  mvrnorm(n = n, mu = mu, Sigma = sigma)
}

# Set seed for reproducibility
set.seed(123)

# Generate unimodal data (Bivariate normal with correlation)
data_uni <- generate_bivariate_normal(1000, c(0, 0), matrix(c(1, 0.5, 0.5, 1), nrow = 2))

# Convert to data frame
df_uni <- data.frame(x = data_uni[,1], y = data_uni[,2])

# Contour density plot with full density coverage
unimodal_plot <- ggplot(df_uni, aes(x = x, y = y)) +
  geom_density_2d_filled() +  # Full density coverage
  scale_fill_viridis_d(option = "plasma") +  # Aesthetic color scheme
  theme_minimal() +
  theme(
    text = element_text(size = 14),
    legend.position = "right"
  ) +
  labs(title = "Bivariate Contour Density Plot (Unimodal)", x = "X", y = "Y", fill = "Density")

# Print plot
print(unimodal_plot)
```



Bimodal distribution contour plot code here

```
# Load necessary libraries
library(MASS)
library(ggplot2)

# Function to generate the bimodal distribution (Standard + Shifted Bivariate Normal)
generate_bivariate_bimodal <- function(n, lambda) {
  n1 <- round(lambda * n)
  n2 <- n - n1

  # Standard bivariate normal (mean = (0,0), identity covariance)
  data1 <- mvrnorm(n = n1, mu = c(0, 0), Sigma = diag(2))

  # Shifted bivariate normal (mean = (5,5), identity covariance)
  data2 <- mvrnorm(n = n2, mu = c(5, 5), Sigma = diag(2))
```

```r
  rbind(data1, data2)
}

# Set seed for reproducibility
set.seed(123)

# Generate bimodal data (Standard + Shifted Bivariate Normal)
data_bi <- generate_bivariate_bimodal(1000, lambda = 0.5)

# Convert to data frame
df_bi <- data.frame(x = data_bi[,1], y = data_bi[,2])

# Contour density plot with full density coverage
bimodal_plot <- ggplot(df_bi, aes(x = x, y = y)) +
  geom_density_2d_filled() +  # Full density coverage
  scale_fill_viridis_d(option = "plasma") +  # Aesthetic color scheme
  theme_minimal() +
  theme(
    text = element_text(size = 14),
    legend.position = "right"
  ) +
  labs(title = "Bivariate Contour Density Plot (Bimodal)", x = "X", y = "Y", fill = "Density")

# Print plot
print(bimodal_plot)
```



Observations

- The **contour plot of the unimodal distribution** shows a single peak, indicating a **single mode**.
- The **contour plot of the bimodal distribution** shows two distinct peaks, confirming **two modes**.

2.Real-World Dataset: Detecting Modes Using Contour Plots

Dataset-1 Description

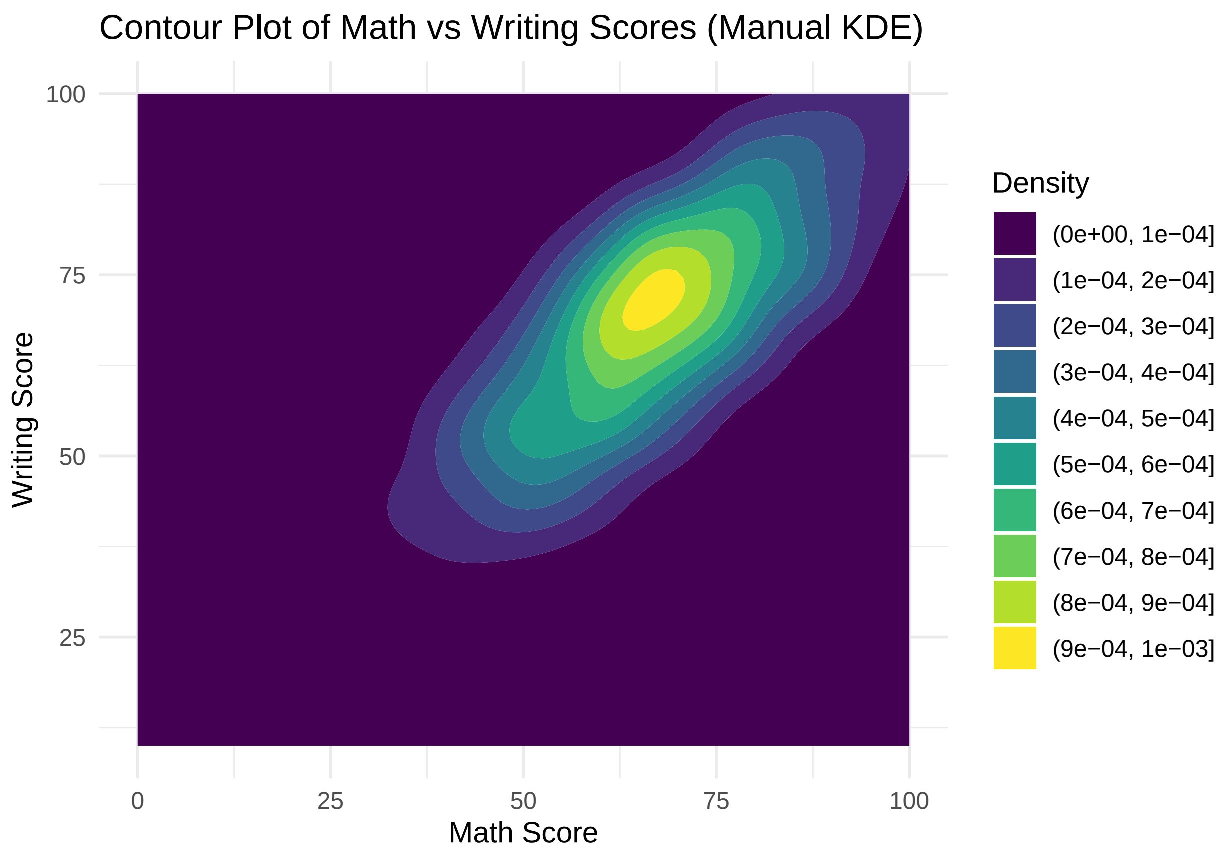
We use a real-world dataset containing **students' performance scores** in three subjects: math, reading, and writing. To analyze modality, we focus on the **math and writing scores** as our bivariate dataset.

Methodology

1. **Density Estimation**: Since we do not have the true density function, we estimate it **manually** using a kernel density estimator (KDE) without inbuilt R functions.
2. **Contour Plot**: We visualize the estimated density using a contour plot. Peaks in the contour plot indicate **modes** of the distribution.
3. **Mode Detection**: We examine the number of peaks in the contour plot to determine if the data is **unimodal or bimodal**.

Next, we provide the R code to perform these steps.

```r
# Read the CSV file
data <- read.csv("StudentsPerformance.csv")


# Extract relevant columns
x <- data$math.score
y <- data$writing.score

# Define bandwidth selection (Scott's Rule of Thumb)
bw_x <- 1.06 * sd(x) * length(x)^(-1/5)
bw_y <- 1.06 * sd(y) * length(y)^(-1/5)

# Define grid for density estimation
n_grid <- 100
x_grid <- seq(min(x), max(x), length.out = n_grid)
y_grid <- seq(min(y), max(y), length.out = n_grid)

# Function to compute 2D kernel density estimation
kde2d_manual <- function(x, y, x_grid, y_grid, bw_x, bw_y) {
  density_matrix <- matrix(0, nrow = length(x_grid), ncol = length(y_grid))
  n <- length(x)

  # Gaussian kernel function
  gaussian_kernel <- function(x) exp(-0.5 * x^2) / sqrt(2 * pi)

  # Compute density for each grid point
  for (i in 1:length(x_grid)) {
    for (j in 1:length(y_grid)) {
      # Compute weighted sum of Gaussian kernels
      kde_value <- sum(gaussian_kernel((x_grid[i] - x) / bw_x) *
                         gaussian_kernel((y_grid[j] - y) / bw_y))
      density_matrix[i, j] <- kde_value / (n * bw_x * bw_y)
    }
  }

  return(list(x = x_grid, y = y_grid, z = density_matrix))
}

# Compute the KDE manually
kde_result <- kde2d_manual(x, y, x_grid, y_grid, bw_x, bw_y)

# Convert to dataframe for ggplot
density_df <- expand.grid(x = kde_result$x, y = kde_result$y)
density_df$z <- as.vector(kde_result$z)

# Plot using ggplot2
ggplot(density_df, aes(x = x, y = y, z = z)) +
  geom_contour_filled() +
  labs(title = "Contour Plot of Math vs Writing Scores (Manual KDE)",
       x = "Math Score",
       y = "Writing Score",
       fill = "Density") +
  theme_minimal()
```



Observation

The selected variables, **math score** and **writing score**, exhibit a strong correlation. The contour plot shows a single peak, indicating a **unimodal distribution**. This result aligns with expectations, as writing and math scores are often related, leading to a smooth and continuous density distribution with one dominant mode.

Dataset-2 Description

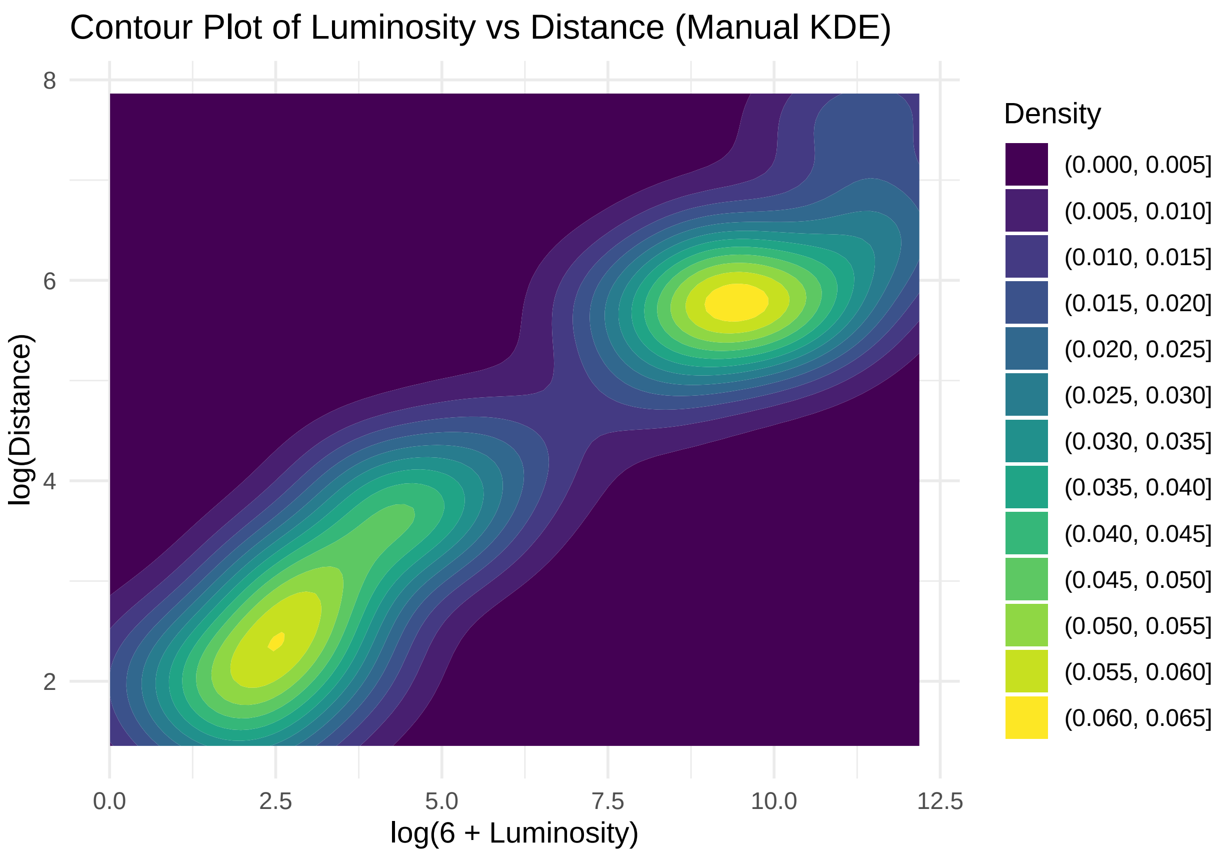
The dataset contains astronomical properties of stars, including luminosity, distance, radius, temperature, and spectral class. We use **Luminosity** and **Distance** to analyze the density distribution.

To handle the wide range of values, we apply a **log transformation**:
- $x = \log(6 + \text{Luminosity})$
- $y = \log(\text{Distance})$

This transformation ensures a more stable and interpretable distribution.

```r
# Load necessary libraries
library(ggplot2)

# Read the CSV file
data <- read.csv("star_dataset.csv")

# Extract relevant columns
x <- log(6 + data$Luminosity..L.Lo.)
y <- log(data$Distance..ly.)
```

```r
# Define bandwidth selection (Scott's Rule of Thumb)
bw_x <- 1.06 * sd(x) * length(x)^(-1/5)
bw_y <- 1.06 * sd(y) * length(y)^(-1/5)

# Define grid for density estimation
n_grid <- 100
x_grid <- seq(min(x), max(x), length.out = n_grid)
y_grid <- seq(min(y), max(y), length.out = n_grid)

# Function to compute 2D kernel density estimation
kde2d_manual <- function(x, y, x_grid, y_grid, bw_x, bw_y) {
  density_matrix <- matrix(0, nrow = length(x_grid), ncol = length(y_grid))
  n <- length(x)

  # Gaussian kernel function
  gaussian_kernel <- function(x) exp(-0.5 * x^2) / sqrt(2 * pi)

  # Compute density for each grid point
  for (i in 1:length(x_grid)) {
    for (j in 1:length(y_grid)) {
      # Compute weighted sum of Gaussian kernels
      kde_value <- sum(gaussian_kernel((x_grid[i] - x) / bw_x) *
                       gaussian_kernel((y_grid[j] - y) / bw_y))
      density_matrix[i, j] <- kde_value / (n * bw_x * bw_y)
    }
  }

  return(list(x = x_grid, y = y_grid, z = density_matrix))
}

# Compute the KDE manually
kde_result <- kde2d_manual(x, y, x_grid, y_grid, bw_x, bw_y)

# Convert to dataframe for ggplot
density_df <- expand.grid(x = kde_result$x, y = kde_result$y)
density_df$z <- as.vector(kde_result$z)

# Plot using ggplot2
ggplot(density_df, aes(x = x, y = y, z = z)) +
  geom_contour_filled() +
  labs(title = "Contour Plot of Luminosity vs Distance (Manual KDE)",
       x = "log(6 + Luminosity)",
       y = "log(Distance)",
       fill = "Density") +
  theme_minimal()
```



Observations
The contour plot shows a **bimodal density structure** with two distinct but slightly merged loops, indicating **two underlying clusters** in the data.

Possible Reasons for Bimodal Behavior

- **Two Star Populations**: Likely represents **main-sequence stars** (closer, lower luminosity) and **giants/supergiants** (farther, highly luminous).
- **Spectral Classification**: Bright stars belong to earlier spectral types (O, B, A), while fainter stars are later types (F, G, K, M), forming separate clusters.
- **Selection Effects**: The dataset may favor **distant, bright supergiants** and **nearby, faint dwarfs**, leading to two peaks.
- **Stellar Evolution**: The peaks could correspond to stars at different **evolutionary stages** in their lifecycle.

3.Image-Based Data Analysis using Contour Plots

Objective

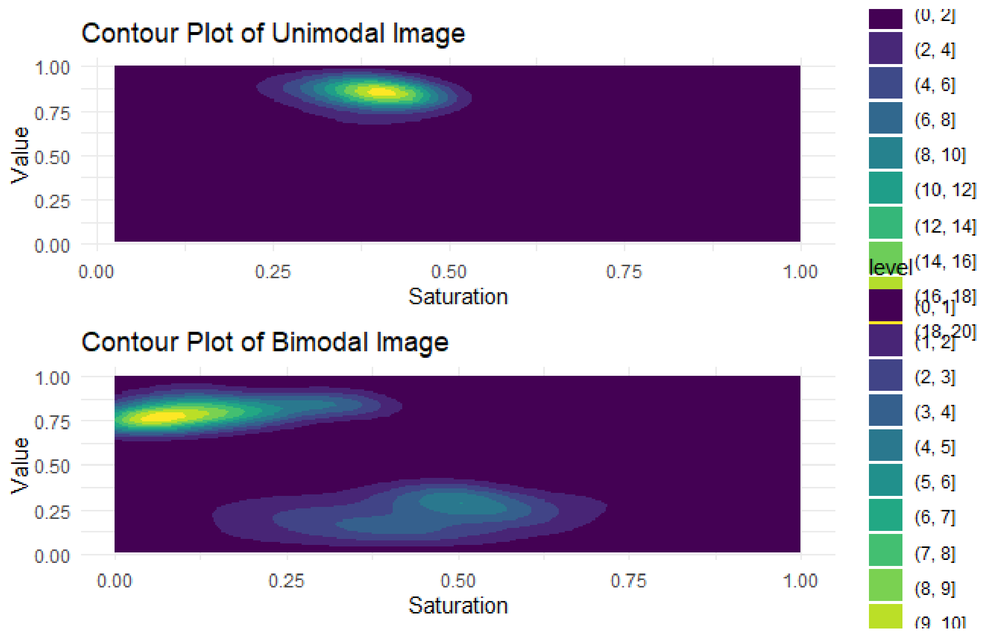
In this section, we analyze bivariate distributions derived from image data using contour plots. Specifically, we extract color features from images and apply Kernel Density Estimation (KDE) to detect mode structures. The goal is to determine whether the color distribution of an image exhibits unimodal or bimodal behavior.

We process images by converting them into the HSV (Hue, Saturation, Value) color space and extracting the **Saturation** and **Value (Brightness)** components. These extracted values are then used to estimate density distributions through KDE, allowing us to visualize contour structures.

Code Implementation

```r
library(imager)
library(ggplot2)
library(gridExtra)

# Function to process image and return KDE data frame
process_image <- function(image_path, grid_size = 100, bandwidth = 0.05) {
  img <- load.image(image_path)         # Load image
  img_hsv <- RGBtoHSV(img)              # Convert to HSV

  sat <- as.vector(channel(img_hsv,2))  # Extract Saturation
  value <- as.vector(channel(img_hsv,3))# Extract Value (Brightness)

  df <- na.omit(data.frame(Saturation = sat, Value = value)) # Remove NA values
  df <- df[sample(nrow(df), min(10000, nrow(df))), ]   # Downsample for efficiency

  # Define grid for KDE evaluation
  x_range <- range(df$Saturation, na.rm = TRUE)
  y_range <- range(df$Value, na.rm = TRUE)

  x_seq <- seq(x_range[1], x_range[2], length.out = grid_size)
  y_seq <- seq(y_range[1], y_range[2], length.out = grid_size)
  grid <- expand.grid(Saturation = x_seq, Value = y_seq)

  # Compute KDE manually using Gaussian kernel
  kde_values <- numeric(nrow(grid))
  n <- nrow(df)

  for (i in 1:nrow(grid)) {
    kde_values[i] <- sum(exp(-((df$Saturation - grid$Saturation[i])^2 +
                                 (df$Value - grid$Value[i])^2) / (2 * bandwidth^2)))
  }

  # Normalize density values
  kde_values <- kde_values / (n * 2 * pi * bandwidth^2)

  # Attach density values to grid
  grid$Density <- kde_values
  return(grid)
}

# Define file paths

unimodal_image_path <- "C:/Users/Jaini Patel/Downloads/Sun Temple.jpg"
bimodal_image_path <- "C:/Users/Jaini Patel/Downloads/Sky photo IITK .jpg"
```

```r
# Process both images
unimodal_kde <- process_image(unimodal_image_path)
bimodal_kde <- process_image(bimodal_image_path)

# Generate contour plots
plot_unimodal <- ggplot(unimodal_kde, aes(x = Saturation, y = Value, z = Density)) +
  geom_contour_filled() +
  labs(title = "Contour Plot of Unimodal Image",
       x = "Saturation", y = "Value") +
  theme_minimal()

plot_bimodal <- ggplot(bimodal_kde, aes(x = Saturation, y = Value, z = Density)) +
  geom_contour_filled() +
  labs(title = "Contour Plot of Bimodal Image",
       x = "Saturation", y = "Value") +
  theme_minimal()

# Arrange plots side by side
grid.arrange(plot_unimodal, plot_bimodal, nrow = 2)
```



Observations

The contour plots reveal distinct density structures:

- The **unimodal image** shows a **single dense region**, indicating a **uniform color distribution**.
- The **bimodal image** exhibits **two distinct peaks**, suggesting **two dominant color clusters**.

Possible Reasons

1. **Image Content** – A **single-colored** region forms a unimodal distribution, while **contrasting areas** create bimodal density.

2. **Lighting & Shadows** – **Uniform lighting** leads to one peak, while **varying light sources or shadows** introduce a second mode.

3. **Color Composition** – **Simple hues** yield unimodal patterns; **diverse colors** result in bimodal distributions.

These findings highlight how **color structure influences density patterns in images**.

Question 2

1.Monte Carlo Estimation using Uniform Sampling

We estimate the given integral using **Monte Carlo integration** with uniform random sampling. The integral to be computed is:

$$I = \int_0^1 x^8(1-x)^8 \frac{25+816x^2}{3164(1+x^2)} dx.$$

Since $X \sim U(0,1)$, we can rewrite the integral as an expectation:

$$I = \mathbb{E}[f(X)], \quad X \sim U(0,1).$$

The Monte Carlo estimate for I is given by:

$$I_N = \frac{1}{N}\sum_{i=1}^{N} f(X_i),$$

where X_i are **i.i.d.** uniform samples from $[0,1]$.

The **expected value** of the integral, computed analytically, is:

$$I = \frac{9940-3164\pi}{3164} \approx 2.6676419 \times 10^{-7}.$$

This value serves as a reference for evaluating the accuracy of our Monte Carlo estimates.

```r
set.seed(123)  # Ensuring reproducibility
# Define the function inside the integral
f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816*x^2)) / (3164 * (1 + x^2))
}

# Monte Carlo estimation function
monte_carlo_integral <- function(N) {
  x_samples <- runif(N)  # Generate N uniform samples in [0,1]
  I_N <- mean(f(x_samples))  # Compute the Monte Carlo estimate
  return(I_N)
}

# Example estimation for N = 500
N <- 500
estimate <- monte_carlo_integral(N)
cat("Monte Carlo Estimate for N =", N, ":", estimate, "\n")
```

```
## Monte Carlo Estimate for N = 500 : 2.609823e-07
```

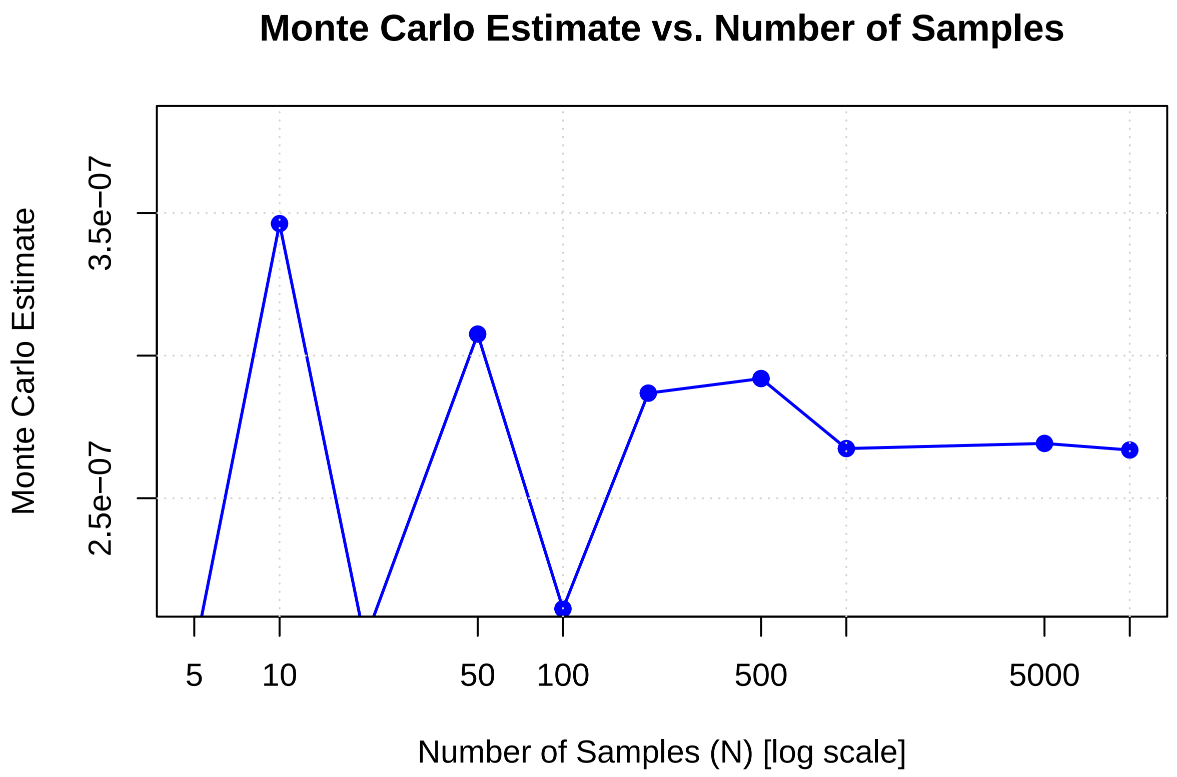
observation

For $N = 500$, the Monte Carlo estimate obtained is 2.61×10^{-7}. This value is an approximation of the true integral, which converges as N increases. However, due to the stochastic nature of Monte Carlo estimation, the result may vary slightly across different runs.

Monte Carlo integration is subject to **statistical error**, and the accuracy improves with larger N. To systematically analyze how the estimate evolves, we now compute it for various values of N and visualize the trend.

To understand how the estimate behaves as N increases, we compute the Monte Carlo estimate for various values of N and visualize the trend.

```r
# Values of N to analyze (more evenly spread for better scaling)
N_values <- c(5, 10, 20, 50, 100, 200, 500, 1000, 5000, 10000)

# Compute estimates for each N
estimates <- sapply(N_values, monte_carlo_integral)

# Improved trace-style plot with logarithmic x-axis
plot(N_values, estimates, type="o", log="x", pch=19, col="blue",
     xlab="Number of Samples (N) [log scale]", ylab="Monte Carlo Estimate",
     main="Monte Carlo Estimate vs. Number of Samples",
     ylim=c(min(estimates) * 1.1, max(estimates) * 1.1), lwd=1.5)

# Adding a reference line at 0 for better visualization
abline(h=0, col="red", lty=2)

# Grid for better readability
grid()
```



Observation on Convergence :

From the plot, we observe that as N increases, the Monte Carlo estimate stabilizes and converges towards a consistent value. While fluctuations are noticeable for smaller N, the estimates become more stable for $N \geq 1000$, indicating convergence.

This behavior aligns with the theoretical expectation that the Monte Carlo estimate approximates the true integral more accurately as the number of samples increases. However, due to randomness, minor variations persist even for large N.

Analysis of Variance in Monte Carlo Estimates

Theoretical Background Monte Carlo estimation relies on the **Law of Large Numbers**, ensuring that the sample mean of independent random variables converges to the expected value as the number of samples increases. However, this convergence is associated with **statistical variance**, which decreases as N grows.

For a Monte Carlo estimate $\hat{I}_N$ of an integral, the variance is given by:

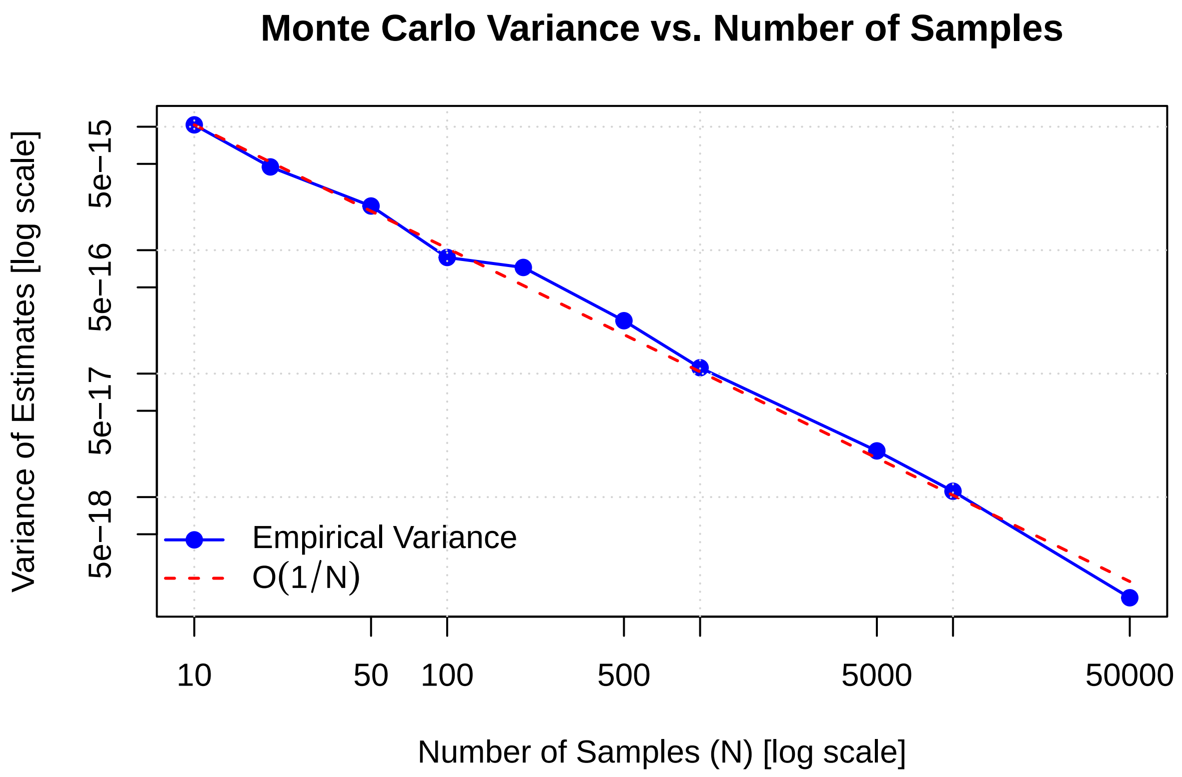
$$\mathrm{Var}(\hat{I}_N) = \frac{\sigma^2}{N}$$

where σ^2 is the inherent variance of the function being integrated. This indicates that **the variance should scale as** $O(1/N)$, meaning that the uncertainty in our estimate decreases proportionally to the number of samples.

Since we have already observed that the Monte Carlo estimate stabilizes for sufficiently large N, we now analyze how its variance behaves as N increases. If the empirical variance follows the expected $O(1/N)$ decay, it will confirm the theoretical convergence properties of Monte Carlo integration.

In the next section, we compute the variance of Monte Carlo estimates for different values of N and visualize its behavior.

```r
set.seed(123)  # Ensuring reproducibility

# Function to compute variance of Monte Carlo estimates for a given N
compute_variance <- function(N, repetitions = 100) {
  estimates <- replicate(repetitions, monte_carlo_integral(N))
  return(var(estimates))
}

# Values of N to analyze (more values for better trend visualization)
N_values <- c(10, 20, 50, 100, 200, 500, 1000, 5000, 10000, 50000)

# Compute variance for each N
variances <- sapply(N_values, compute_variance)

# Plot variance vs. N (log-log scale for power-law behavior)
plot(N_values, variances, type="o", log="xy", pch=19, col="blue",
     xlab="Number of Samples (N) [log scale]", ylab="Variance of Estimates [log scale]",
     main="Monte Carlo Variance vs. Number of Samples",
     lwd=1.5)

# Adding a reference line for theoretical O(1/N) decay (scaled to match variance magnitude)
lines(N_values, variances[1] * (N_values[1] / N_values), col="red", lty=2, lwd=1.5)

# Enhancing visualization with grid
grid()

# Adding legend for clarity
legend("bottomleft", legend=c("Empirical Variance", expression(O(1/N))),
       col=c("blue", "red"), pch=c(19, NA), lty=c(1,2), lwd=c(1.5,1.5), bty="n")
```



Observation on Variance Behavior

The variance of Monte Carlo estimates decreases as N increases, following the expected $O(1/N)$ trend.

Empirical Findings:
- The variance closely follows the theoretical $O(1/N)$ decay.
- No significant deviations from the expected trend are observed across different values of N.
- This confirms that Monte Carlo estimates become more stable with increasing N, as predicted theoretically.

The results validate the expected variance behavior in Monte Carlo estimation.

Scaling Analysis of Variance: $N^c \cdot \mathbf{Var}(\hat{I}_N)$

To further analyze the behavior of variance, we examine how it scales when multiplied by N^c for different values of c.

Theoretical Justification
From Monte Carlo theory, the variance of the estimate follows:

$$\text{Var}(\hat{I}_N) = O(1/N)$$

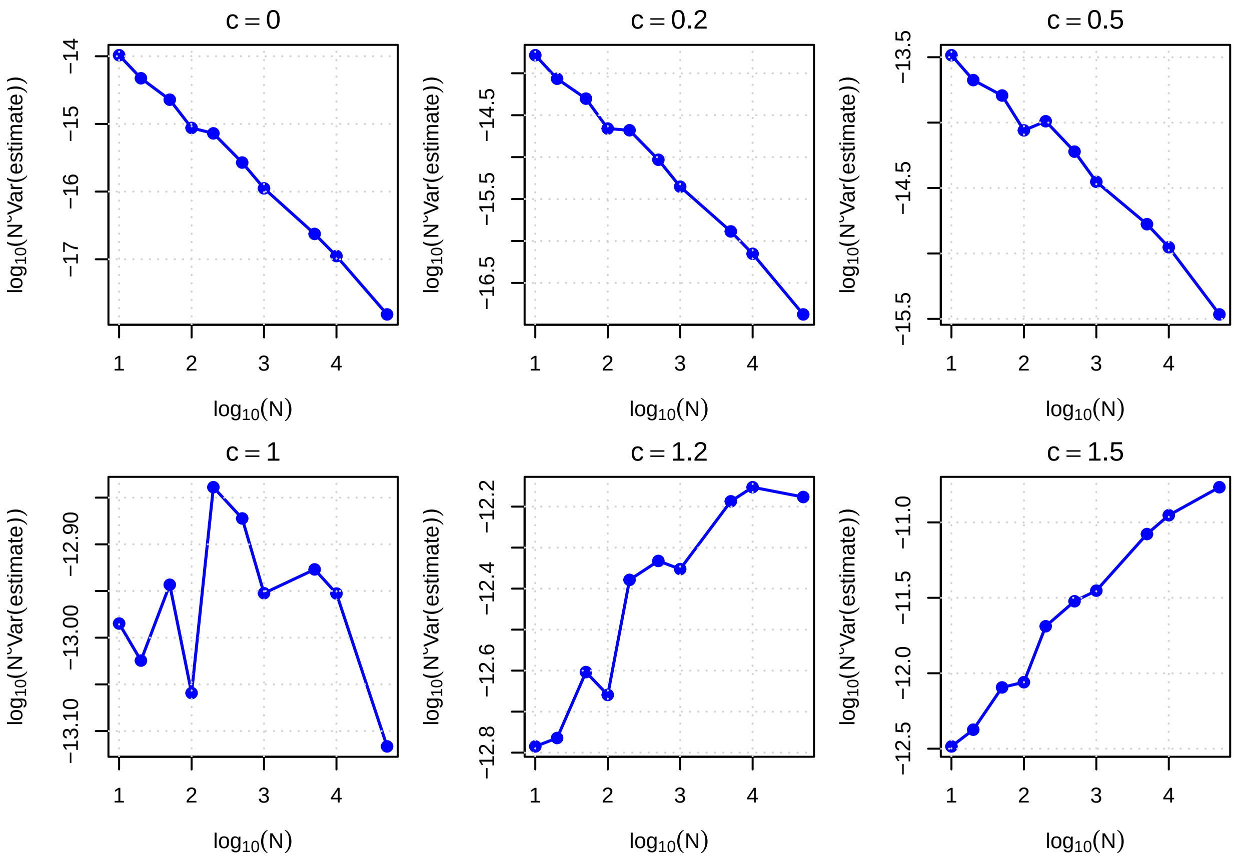
Multiplying by N^c gives:

$$N^c \cdot \text{Var}(\hat{I}_N) = O(N^{c-1})$$

- If $c = 1$, the term remains approximately constant, suggesting the expected $O(1/N)$ decay.
- If $c < 1$, the term should decay towards zero as N increases.
- If $c > 1$, the term should diverge, indicating increasing variance contribution.

By plotting this quantity for different c, we verify whether the empirical trend aligns with theoretical predictions.

```r
set.seed(123)  # Ensuring reproducibility

# Function to compute variance of Monte Carlo estimates for a given N
compute_variance <- function(N, repetitions = 100) {
  estimates <- replicate(repetitions, monte_carlo_integral(N))
  return(var(estimates))
}

# Values of N to analyze
N_values <- c(10, 20, 50, 100, 200, 500, 1000, 5000, 10000, 50000)

# Compute variance for each N
variances <- sapply(N_values, compute_variance)

# Values of c to analyze
c_values <- c(0, 0.2, 0.5, 1, 1.2, 1.5)

# Set up plotting layout for 2 rows and 3 columns
par(mfrow=c(2,3), mar=c(4,4,2,1))

# Generate separate plots for each value of c
for (c_val in c_values) {
  scaled_variance <- N_values^c_val * variances

  plot(log10(N_values), log10(scaled_variance), type="o", col="blue", pch=19, lwd=1.5,
       xlab=expression(log[10](N)), ylab=expression(log[10](N^c * Var(estimate))),
       main=bquote(c == .(c_val)))

  grid()
}
```



Observation on Scaling Analysis

From the plots, we observe that the behavior of $N^c \times \mathrm{Var}(\hat{I}_N)$ closely follows theoretical expectations for different values of c. The curves exhibit a general trend consistent with the predicted scaling behavior.

However, minor fluctuations are present, possibly due to the scaling of the graphs and the inherent randomness in Monte Carlo estimation. Despite these variations, the results reinforce the expected $O(1/N)$ variance decay.

Conclusion on Monte Carlo Estimation

Monte Carlo integration provides a simple yet effective method for approximating integrals, particularly in higher dimensions where traditional numerical techniques become infeasible. Our analysis confirms the expected properties:

- The estimate converges as N increases, with statistical fluctuations decreasing.
- The variance follows the predicted $O(1/N)$ decay, improving stability with larger N.
- However, achieving high accuracy requires a large number of samples, making Monte Carlo inefficient for some integrals.

Motivation for Importance Sampling

A key limitation of uniform sampling is that it treats all regions of the domain equally, even if some contribute little to the integral. This inefficiency can lead to slow convergence, especially when the function has sharp peaks or most of its contribution comes from a small region.

Importance Sampling addresses this by sampling from a more suitable distribution, reducing variance and improving efficiency. Instead of using uniform samples, we choose a distribution that better matches the shape of the integrand, leading to more accurate estimates with fewer samples.

In the next section, we explore how Importance Sampling can enhance Monte Carlo estimation.

2.Importance Sampling with Beta(9,9) Distribution

Theoretical Background

While standard Monte Carlo integration samples uniformly, **Importance Sampling** improves efficiency by sampling from a more suitable probability distribution. This reduces variance by concentrating samples in regions that contribute most to the integral.

Given an integral of the form:

$$I = \int_a^b f(x)dx$$

we rewrite it using a probability density function $g(x)$:

$$I = \int_a^b \frac{f(x)}{g(x)} g(x)dx = \mathbb{E}_g\left[\frac{f(X)}{g(X)}\right]$$

where $X \sim g(x)$. Instead of sampling X uniformly, we choose $g(x)$ to approximate the shape of $f(x)$, leading to more efficient estimates.

Choice of Beta(9,9) Distribution

For our integral, we select **Beta(9,9)** as the importance sampling distribution because:

- The function $f(x)$ has a similar shape to Beta(9,9), ensuring that samples are concentrated where $f(x)$ is significant.
- The Beta distribution is naturally defined on $[0, 1]$, aligning with our integral's domain.

Expected Advantages

- **Faster convergence**: Since more samples are drawn from important regions, the estimate stabilizes quicker than uniform sampling.
- **Lower variance**: The variance of the estimate should be smaller compared to standard Monte Carlo.

In the next sections, we implement Importance Sampling using Beta(9,9) and compare its performance with standard Monte Carlo integration.

```r
set.seed(123)  # Ensuring reproducibility

# Define the function inside the integral
f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816*x^2)) / (3164 * (1 + x^2))
}

# Probability density function of Beta(9,9)
g <- function(x) {
  dbeta(x, 9, 9)
}

# Importance Sampling estimation function using Beta(9,9)
importance_sampling_integral <- function(N) {
  x_samples <- rbeta(N, 9, 9)  # Generate N samples from Beta(9,9)
  weights <- f(x_samples) / g(x_samples)  # Compute importance weights
  I_N <- mean(weights)  # Compute the Importance Sampling estimate
  return(I_N)
}

# Example estimation for N = 500
N <- 500
importance_estimate <- importance_sampling_integral(N)
cat("Importance Sampling Estimate for N =", N, ":", importance_estimate, "\n")
```

```
## Importance Sampling Estimate for N = 500 : 2.705805e-07
```

Observation for $N = 500$

For $N = 500$:
- **Monte Carlo Estimate**: 2.6098×10^{-7}
- **Importance Sampling Estimate**: 2.7058×10^{-7}
- **Expected Value**:

$$\frac{9940 - 3164\pi}{3164} \approx 2.6676419 \times 10^{-7}.$$

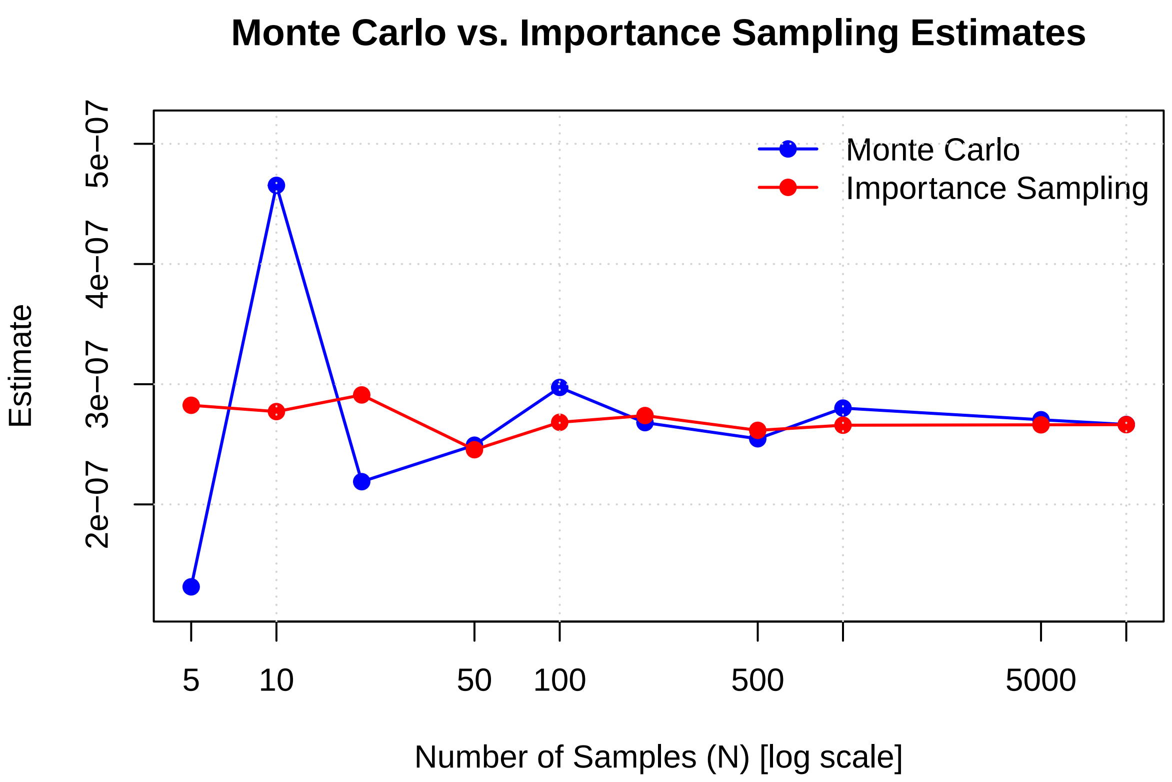
Key Observations:

- The Importance Sampling estimate is **closer to the expected value** than the Monte Carlo estimate.
- The Monte Carlo estimate shows a slightly higher deviation, indicating **higher variance** in standard sampling.
- Importance Sampling reduces variance by sampling from a more suitable distribution (Beta$(9, 9)$), leading to **faster convergence**.
- While both methods approximate the integral well, Importance Sampling demonstrates **better efficiency** at the same sample size.

Next, we analyze the behavior of estimates across different values of N.

```
set.seed(123)  # Ensuring reproducibility

# Define the function inside the integral
f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816*x^2)) / (3164 * (1 + x^2))
}
```

```r
# Define the importance sampling density (Beta(9,9))
g <- function(x) {
  dbeta(x, 9, 9)
}

# Importance Sampling estimation function
importance_sampling_integral <- function(N) {
  x_samples <- rbeta(N, 9, 9)  # Sample from Beta(9,9)
  weights <- f(x_samples) / g(x_samples)  # Compute importance weights
  I_N <- mean(weights)  # Compute the Importance Sampling estimate
  return(I_N)
}

# Values of N to analyze
N_values <- c(5, 10, 20, 50, 100, 200, 500, 1000, 5000, 10000)

# Compute estimates for Monte Carlo and Importance Sampling
mc_estimates <- sapply(N_values, monte_carlo_integral)
is_estimates <- sapply(N_values, importance_sampling_integral)

# Plotting estimates vs. N (log scale)
plot(N_values, mc_estimates, type="o", log="x", pch=19, col="blue",
     xlab="Number of Samples (N) [log scale]", ylab="Estimate",
     main="Monte Carlo vs. Importance Sampling Estimates",
     ylim=c(min(mc_estimates, is_estimates) * 0.9, max(mc_estimates, is_estimates) * 1.1),
     lwd=1.5)
lines(N_values, is_estimates, type="o", col="red", pch=19, lwd=1.5)

# Adding legend
legend("topright", legend=c("Monte Carlo", "Importance Sampling"),
       col=c("blue", "red"), pch=19, lty=1, lwd=1.5, bty="n")

# Grid for better visualization
grid()
```



Observation: Importance Sampling vs. Monte Carlo

From the comparison of estimates, we observe key differences between **Monte Carlo estimation** and **Importance Sampling estimation**:

- **Less fluctuation for small** N: Importance Sampling exhibits reduced variability for lower values of N, while Monte Carlo estimates fluctuate more significantly.
- **Relatively smoother convergence**: The Importance Sampling estimates appear to stabilize more smoothly as N increases, unlike Monte Carlo, which shows more oscillations.
- **Faster convergence**: Importance Sampling reaches stable estimates more quickly, suggesting that fewer samples are required for a reliable approximation compared to Monte Carlo.

This confirms that **Importance Sampling improves efficiency** by leveraging a better-suited sampling distribution, leading to a more stable and reliable estimation process.

Variance in Importance Sampling Estimation

Theoretical Background

Variance reduction is a key advantage of **Importance Sampling** over standard Monte Carlo methods. The variance of an Importance Sampling estimate is given by:

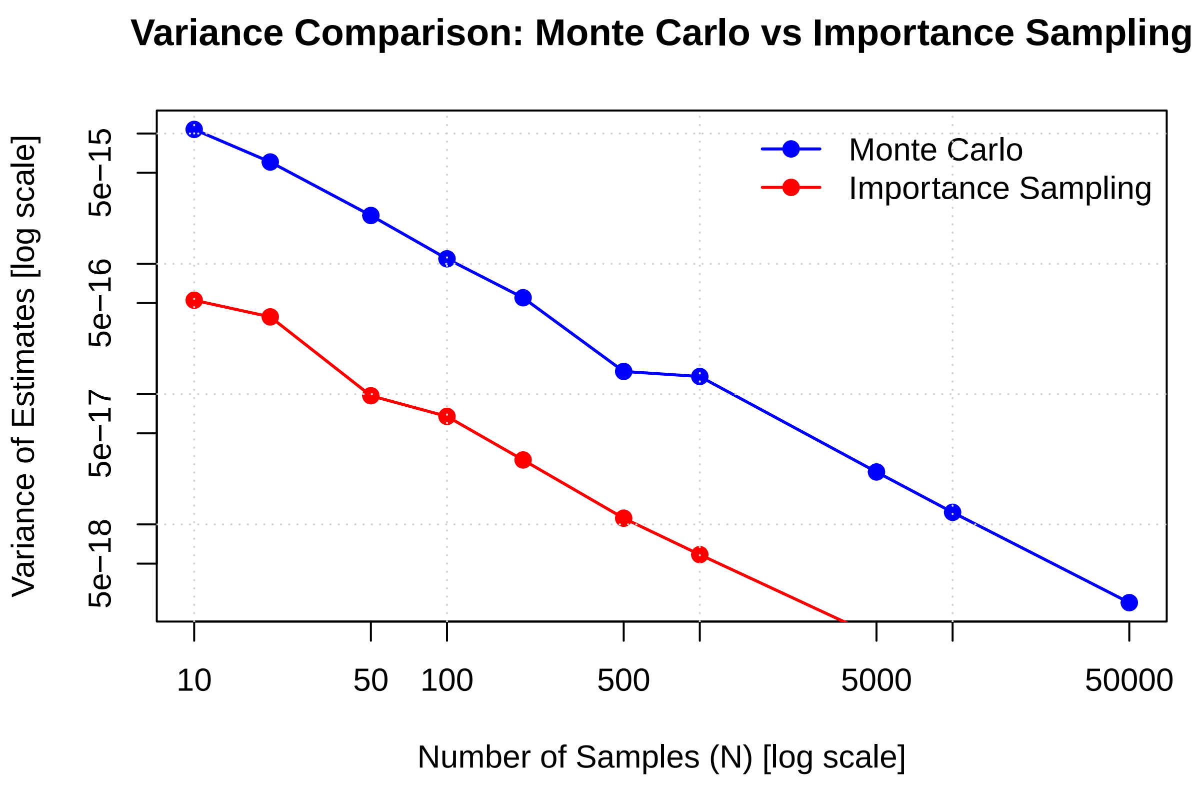
$$\mathrm{Var}(\hat{I}_N) = \frac{1}{N}\mathbb{E}\left[\left(\frac{f(X)}{g(X)}\right)^2\right] - I^2$$

where $g(x)$ is the importance distribution. Since $g(x)$ is chosen to resemble the function being integrated, the variance is often **significantly lower** compared to uniform sampling.

Compared to Monte Carlo:
- The variance should be lower for Importance Sampling, especially for small N.
- We expect a similar $O(1/N)$ scaling but with a **smaller constant factor**, leading to faster convergence.

Now, we compute the variance of Importance Sampling estimates for different values of N and compare it with Monte Carlo variance.

```r
set.seed(123)  # Ensuring reproducibility

# Function to compute variance of Importance Sampling estimates for a given N
compute_variance_is <- function(N, repetitions = 100) {
  estimates <- replicate(repetitions, importance_sampling_integral(N))
  return(var(estimates))
}

# Values of N to analyze
N_values <- c(10, 20, 50, 100, 200, 500, 1000, 5000, 10000, 50000)

# Compute variance for each N using Importance Sampling
variances_is <- sapply(N_values, compute_variance_is)

# Compute variance for Monte Carlo estimates (from previous results)
variances_mc <- sapply(N_values, compute_variance)

# Plot variance comparison on a log-log scale
plot(N_values, variances_mc, type="o", log="xy", pch=19, col="blue",
     xlab="Number of Samples (N) [log scale]", ylab="Variance of Estimates [log scale]",
     main="Variance Comparison: Monte Carlo vs Importance Sampling", lwd=1.5)

lines(N_values, variances_is, type="o", col="red", pch=19, lwd=1.5)

# Adding legend
legend("topright", legend=c("Monte Carlo", "Importance Sampling"), col=c("blue", "red"),
       pch=19, lty=1, lwd=1.5, bty="n")

# Enhancing visualization with grid
grid()
```



Observation on Variance Behavior

- **Lower variance for small N:** Importance Sampling shows significantly less fluctuation than Monte Carlo.
- **Smoother convergence:** Variance decreases more consistently, indicating better stability.
- **Faster reduction in variance:** Importance Sampling achieves lower variance with fewer samples, confirming its efficiency.
- **Leftward shift:** The variance curve is shifted left, meaning faster convergence compared to Monte Carlo.

Overall, Importance Sampling provides **more stable estimates with fewer samples**, making it a more efficient approach.

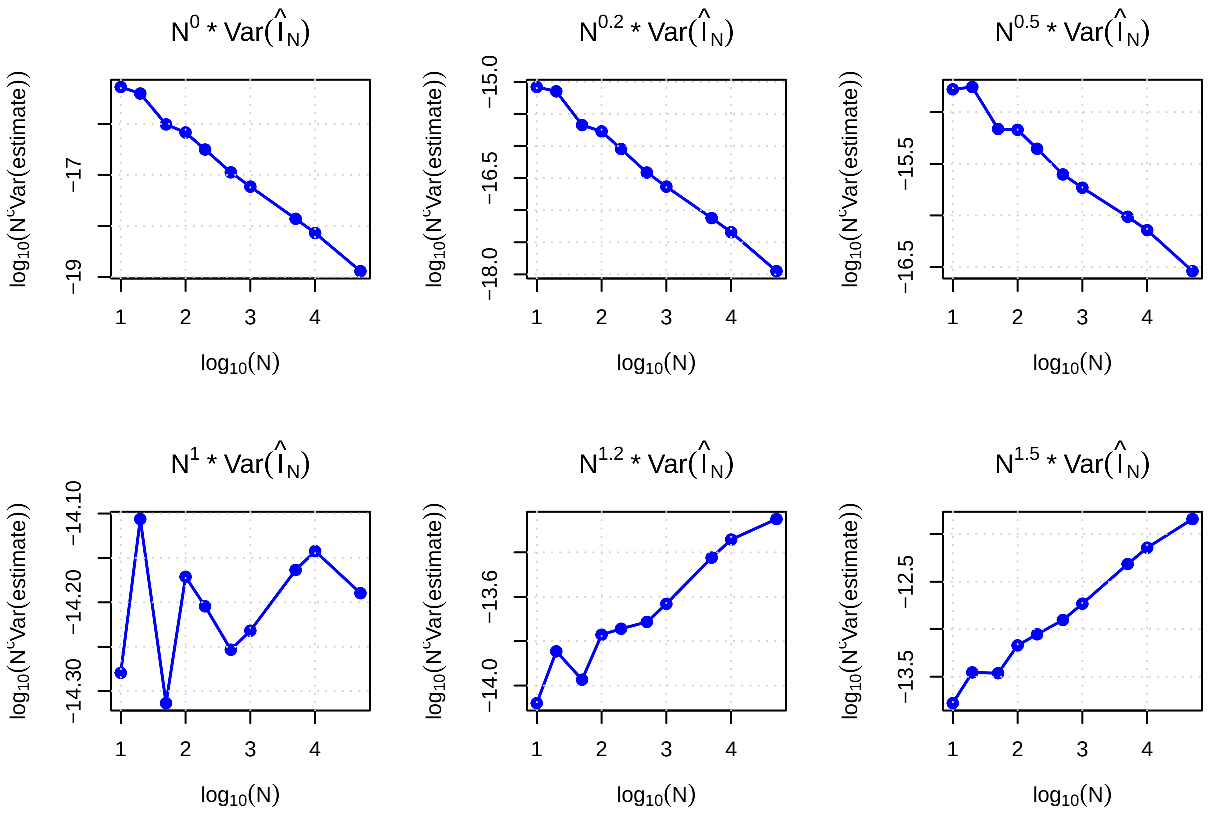
Scaling Analysis in Importance Sampling

To assess the variance behavior in **importance sampling**, we analyze how it scales with N by examining $N^c \cdot \mathrm{Var}(\hat{I}_N)$ for different values of c. This helps us verify if the expected $O(1/N)$ decay holds.

Unlike **Monte Carlo estimation**, where variance reduction is slow, importance sampling is designed to **reduce variance by sampling more efficiently** from a relevant distribution. The variance should still follow a power-law behavior, but with reduced fluctuations and smoother convergence.

We compute variance for different N, multiply it by N^c, and compare its stability across different values of c. The goal is to observe if variance decays faster or exhibits more consistent scaling compared to Monte Carlo.

```r
set.seed(123)  # Ensuring reproducibility

# Function to compute variance of importance sampling estimates for a given N
compute_variance_IS <- function(N, repetitions = 100) {
  estimates <- replicate(repetitions, importance_sampling_integral(N))
  return(var(estimates))
}

# Values of N to analyze
N_values <- c(10, 20, 50, 100, 200, 500, 1000, 5000, 10000, 50000)

# Compute variance for each N
variances_IS <- sapply(N_values, compute_variance_IS)

# Values of c to analyze
c_values <- c(0, 0.2, 0.5, 1, 1.2, 1.5)

# Plotting N^c * Var(estimate) for different c (6 separate plots)
par(mfrow=c(2,3))  # Arrange plots in a 2x3 grid
for (c_val in c_values) {
  scaled_variance_IS <- N_values^c_val * variances_IS

  plot(log10(N_values), log10(scaled_variance_IS), type="o", col="blue", pch=19, lwd=1.5,
       xlab=expression(log[10](N)), ylab=expression(log[10](N^c * Var(estimate))),
       main=bquote(N^.(c_val) ~ "*" ~ Var(hat(I)[N])))

  grid()
}
```



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

Observation on Variance Scaling

The variance in **importance sampling** follows the same overall $O(1/N)$ trend as Monte Carlo estimation. However, we observe:

- **Reduced fluctuation** in variance compared to Monte Carlo, leading to a **smoother trend**.
- **More stable scaling behavior**, confirming the efficiency of importance sampling in variance reduction.

These findings reinforce the advantage of importance sampling in achieving more reliable estimates with less variability.

Final Conclusion

Based on our observations, **importance sampling** demonstrates a **clear advantage** over standard Monte Carlo estimation for this integral.

- **Faster convergence**: Importance sampling reaches stable estimates with **smaller sample sizes** compared to Monte Carlo.
- **Reduced variance**: Variance decreases more **smoothly** with N, leading to **less fluctuation** in estimates.

- **Improved efficiency**: Given the same computational effort, importance sampling provides **more reliable estimates**.

While these findings align with theoretical expectations, the effectiveness of importance sampling **depends on the choice of the proposal distribution**. In this case, the Beta(9,9) function led to better results by concentrating samples where the integrand had higher contributions.

Thus, while Monte Carlo remains a **general-purpose** approach, importance sampling can be a **more efficient alternative** when a suitable proposal distribution is available.

3.Using Ordered Statistics for Numerical Integration

Traditional numerical integration methods use **fixed grid points**. However, a more **statistical** approach involves generating **random samples** from a uniform distribution and using their **order statistics** as integration points.

Why Use Ordered Statistics?

1. **Statistical Interpretation**
   - Instead of a fixed partition, we generate `\(N+1\)` **random samples** from `\(U(0,1)\)` and **sort** them.

   - This aligns with **Monte Carlo principles**, where randomness is incorporated into numerical methods.
2. **Adaptive Step Size**
   - Normally, a **fixed step size `\(h\)`** is used in numerical integration.

   - Here, `\(h\)` is estimated from the **differences between consecutive ordered samples**, capturing **natural variations** in sample density.
3. **Better Approximation**:
   - By **randomizing grid points**, we approximate the integral with a **probabilistic approach**, reducing biases from deterministic grid placement.

Simpson's 1/3 Rule with Ordered Statistics

Simpson's **1/3 Rule** approximates integration using **quadratic interpolation**. The integral is estimated as:

$$I \approx \frac{h}{3}\left[f(X_{(1)}) + 4\sum f(X_{(i)}) + 2\sum f(X_{(j)}) + f(X_{(N+1)})\right]$$

where: - $X_{(i)}$ are **ordered statistics** from $U(0,1)$. - **Summations** follow Simpson's 1/3 rule structure. - h is estimated from **ordered sample differences**.

Why is This More Statistical?

- By **randomly selecting grid points**, integration follows a **Monte Carlo-like method**.
- This **reduces deterministic bias** and introduces **statistical randomness** in estimation.

```r
set.seed(123)  # For reproducibility

# Define function to integrate
f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816 * x^2)) / (3164 * (1 + x^2))
}

# Simpson's 1/3 Rule with Ordered Uniform Samples
simpson_1_3_stat <- function(N) {
  if (N %% 2 != 0) stop("N must be even for Simpson's 1/3 rule")

  x_samples <- sort(runif(N + 1))  # Generate ordered U(0,1) samples
  h <- mean(diff(x_samples))  # Approximate step size using differences
  y <- f(x_samples)  # Function values at ordered statistics

  integral <- (h / 3) * (y[1] + 4 * sum(y[seq(2, N, by = 2)]) +
                         2 * sum(y[seq(3, N-1, by = 2)]) + y[N+1])

  return(integral)
}
```

###**Simpson's 3/8 Rule with Ordered Statistics** Simpson's **3/8 Rule** improves accuracy by using **cubic interpolation**. The formula is:

$$I \approx \frac{3h}{8}\left[f(X_{(1)}) + 3\sum f(X_{(i)}) + 3\sum f(X_{(j)}) + 2\sum f(X_{(k)}) + f(X_{(N+1)})\right]$$

where: - $X_{(i)}$ are **ordered statistics** from $U(0,1)$. - Summations are structured according to **3/8 Rule**. - h is estimated from **ordered sample differences**.

Why is This More Statistical?

- **Same benefits as 1/3 Rule**, but **higher-order accuracy**.
- By using **ordered statistics**, this rule adapts to the **randomness** in numerical integration.

```r
set.seed(123)  # For reproducibility

# Define function to integrate
f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816 * x^2)) / (3164 * (1 + x^2))
}

# Simpson's 3/8 Rule with Ordered Uniform Samples
simpson_3_8_stat <- function(N) {
  if (N %% 3 != 0) stop("N must be a multiple of 3 for Simpson's 3/8 rule")

  x_samples <- sort(runif(N + 1))  # Generate ordered U(0,1) samples
  h <- mean(diff(x_samples))  # Approximate step size using differences
  y <- f(x_samples)  # Function values at ordered statistics

  integral <- (3 * h / 8) * (y[1] +
                             3 * sum(y[seq(2, N, by = 3)]) +  # Fixed
```

```
                              3 * sum(y[seq(3, N-1, by = 3)]) +  # Fixed
                              2 * sum(y[seq(4, N-2, by = 3)]) +  # Fixed
                              y[N+1])

  return(integral)
}
```

Results

For the interval $[0, 1]$, we use:

- $N = 1000$ for Simpson's 1/3 Rule (must be even).
- $N = 999$ for Simpson's 3/8 Rule (must be a multiple of 3).

```
N1 <- 1000  # Must be even
I_simpson_1_3_stat <- simpson_1_3_stat(N1)
cat("Simpson's 1/3 Rule Estimate (Statistical Approach):", I_simpson_1_3_stat, "\n")
```

```
## Simpson's 1/3 Rule Estimate (Statistical Approach): 2.663829e-07
```

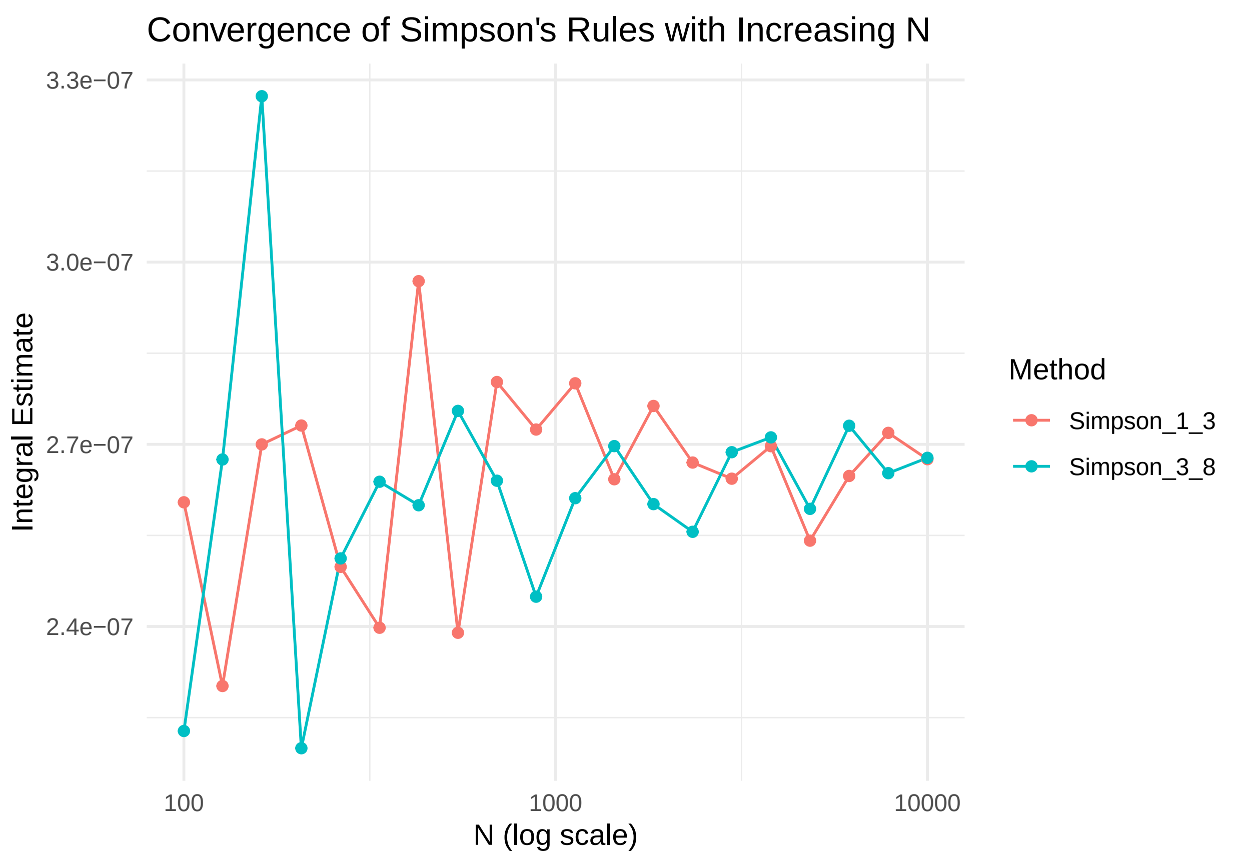
```
# Example with N = 999
N2 <- 999  # Must be a multiple of 3
I_simpson_3_8_stat <- simpson_3_8_stat(N2)
cat("Simpson's 3/8 Rule Estimate (Statistical Approach):", I_simpson_3_8_stat, "\n")
```

```
## Simpson's 3/8 Rule Estimate (Statistical Approach): 2.825014e-07
```

After computing the integral estimates using Simpson's 1/3 and 3/8 rules, it is useful to visualize how these estimates behave as the number of sample points (N) increases.

In the following section, we plot the estimated integral values for different values of N to analyze the convergence of these numerical integration methods.

```
library(ggplot2)
library(tidyr)

# Generate a sequence of N values (logarithmically spaced)
N_values <- round(10^seq(2, 4, length.out = 20))  # From 10^2 to 10^4

# Ensure even N for Simpson's 1/3 rule and multiples of 3 for 3/8 rule
N_values_1_3 <- N_values + (N_values %% 2)  # Make sure N is even
N_values_3_8 <- N_values + (3 - N_values %% 3) %% 3  # Make N a multiple of 3

# Compute estimates
simpson_1_3_estimates <- sapply(N_values_1_3, simpson_1_3_stat)
simpson_3_8_estimates <- sapply(N_values_3_8, simpson_3_8_stat)

# Create a data frame for plotting
df <- data.frame(
  logN = log10(N_values),
  N = N_values,
  Simpson_1_3 = simpson_1_3_estimates,
  Simpson_3_8 = simpson_3_8_estimates
)

# Convert data to long format for ggplot
df_long <- df %>% pivot_longer(cols = c(Simpson_1_3, Simpson_3_8),
                               names_to = "Method",
                               values_to = "Estimate")

# Plot with log-scaled x-axis
ggplot(df_long, aes(x = N, y = Estimate, color = Method)) +
  geom_line() +
  geom_point() +
  scale_x_log10() +  # Log-scale for better visualization
  labs(title = "Convergence of Simpson's Rules with Increasing N",
       x = "N (log scale)",
       y = "Integral Estimate") +
  theme_minimal()
```



###**Observations**

Upon plotting the integral estimates for Simpson's 1/3 and 3/8 rules using ordered uniform samples, we notice that even for large values of N, the convergence is not smooth. The estimates exhibit significant fluctuations, making it difficult to determine which method provides a more accurate approximation.

Additionally, the expected reduction in error with increasing N is not clearly visible, which suggests the presence of an additional source of error beyond the standard Simpson's rule error.

Possible Reasons

1. **Error Due to Ordered Statistics Approximation**
   - In deterministic quadrature, the integration points `\(x_i\)` are fixed and evenly spaced as `\(x_i = \frac{i}{N}\)`.
   - In the statistical approach, the ordered uniform samples introduce variability, leading to irregular spacing between points.
   - The maximum possible difference between consecutive ordered statistics is approximately `\(\frac{1}{N}\)`, introducing an additional source of error.
2. **Propagation of Sampling Variability**
   - Unlike deterministic integration rules, where step size `\(h\)` is constant, here `\(h\)` is estimated as the mean of differences in ordered statistics, leading to fluctuations.
   - The presence of randomness affects higher-order accuracy, leading to larger deviations from the true integral.
3. **Impact on Convergence**
   - The error introduced by randomness does not necessarily diminish at the expected rate of Simpson's rule.
   - Instead of a smooth decrease, the fluctuations persist even as `\(N\)` increases, making the performance comparison between the 1/3 and 3/8 rules unclear.

Conclusion

The results indicate that incorporating numerical integration methods, such as Simpson's rules, with a statistical approach using ordered uniform samples does not yield very reliable results. The additional randomness introduced by the statistical sampling disrupts the structured error behavior expected in deterministic numerical integration.

In particular, the fluctuations in the integral estimates suggest that the statistical variability dominates the higher-order accuracy of Simpson's rules, preventing clear convergence. This highlights a fundamental challenge in combining numerical quadrature with random sampling—while Monte Carlo methods naturally handle randomness, Simpson's rule relies on precise spacing of points, which is lost when using ordered statistics.

To improve the approach, alternative techniques such as importance sampling or variance reduction methods could be explored to mitigate the effects of sampling variability and better utilize structured quadrature rules in a stochastic setting.

4.Riemann Sum Method

We use the **Riemann sum method** with randomly chosen partition points to approximate the integral. Additionally, we analyze the variance of our estimates across multiple trials to assess convergence behavior.

Theory

The **Riemann sum** is a fundamental approach to approximating definite integrals. Given a function $f(x)$ defined on an interval $[a, b]$, the integral can be approximated as:

$$I \approx \sum_{i=1}^{N} f(x_i^*)\Delta x_i,$$

where: - x_i^* is a sample point in each partition subinterval, - $\Delta x_i = x_i - x_{i-1}$ represents the width of the partition, - N is the number of partitions.

The accuracy of this approximation improves as N increases, leading to a better estimate of the true integral.

In our approach, instead of using **uniform partitions**, we generate random partitions x_i from a uniform distribution and sort them and work with the corresponding ordered statistics.

We also normalize the estimates by dividing by 31640, which is derived from numerical integration results.

Methodology

The integral is computed using the following steps: 1. Generate a sequence of **random partition points** x_i drawn from a uniform distribution in $[0, 1]$. 2. Sort the points in ascending order to define partitions. 3. Compute the function value at these partition points. 4. Multiply the function value by the corresponding **bin width** Δx_i and sum over all bins to approximate the integral. 5. Repeat the estimation multiple times to obtain variance statistics. 6. Analyze how the estimated value converges and how variance behaves as the number of partition points N increases.

Implementation

```r
library(ggplot2)
library(gridExtra)
```

```
## Warning: package 'gridExtra' was built under R version 4.4.2
```

```r
f <- function(x) {
  (10 * x^8 * (1 - x)^8 * (25 + 816 * x^2)) / (1 + x^2)
}
```

Defining the Function

```r
N_values <- round(10^(seq(1, 5, length.out = 500)))  # Log-spaced N values
M <- 20  # Number of trials for variance computation

estimated_integrals <- numeric(length(N_values))
variances <- numeric(length(N_values))

for (i in seq_along(N_values)) {
  N <- N_values[i]
  integral_estimates <- numeric(M)  # Store multiple trials
  for (j in 1:M) {
    x_rand <- sort(runif(N, 0, 1))  # Generate and sort random partitions
    dx <- diff(x_rand)  # Compute bin widths (Δx)
    f_values <- f(x_rand[-1])  # Evaluate function at bins
    integral_estimates[j] <- sum(f_values * dx) / 31640  # Normalize
  }

  estimated_integrals[i] <- mean(integral_estimates)
  variances[i] <- var(integral_estimates)
}
```

Statistical Estimation

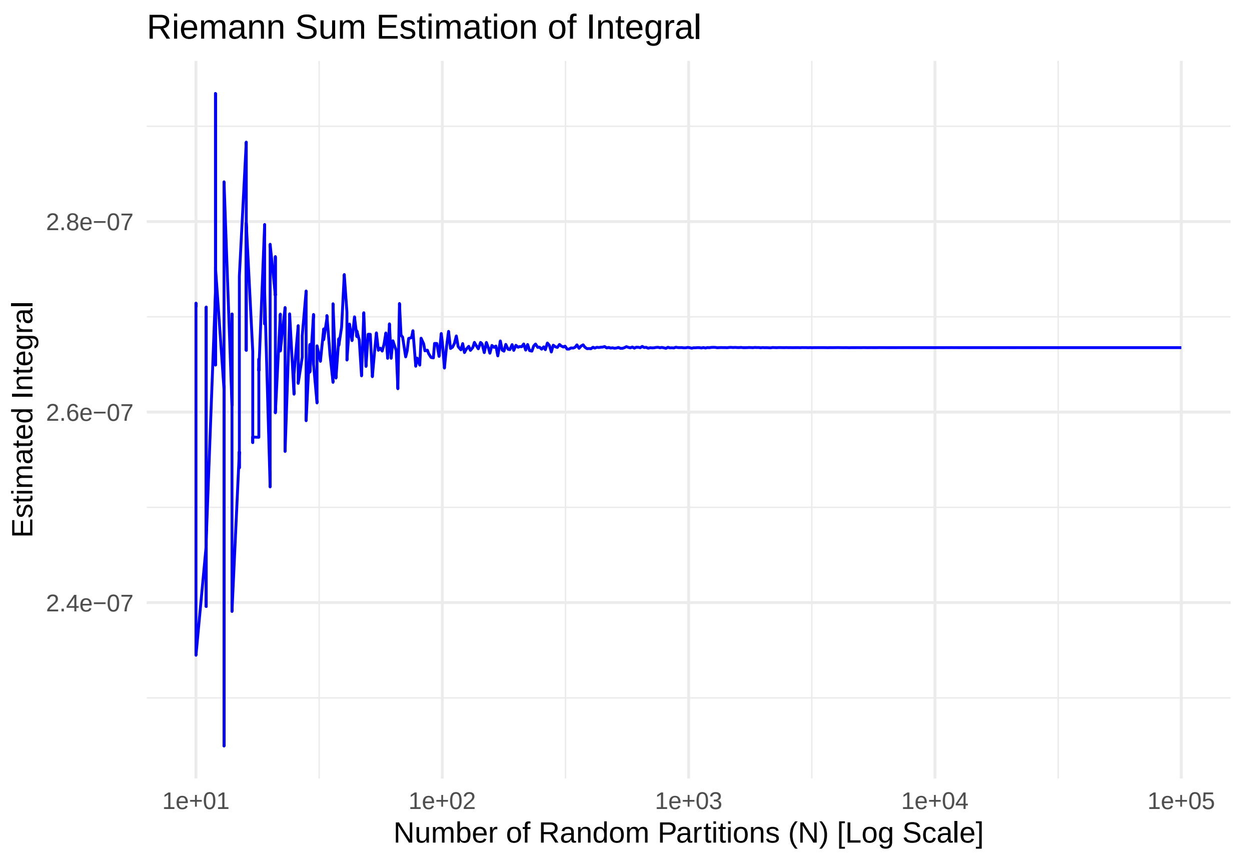
Results

```r
results <- data.frame(N = N_values, Integral_Estimate = estimated_integrals, Variance = variances)

p1 <- ggplot(results, aes(x = N, y = Integral_Estimate)) +
  geom_line(color = "blue", size = 0.5) +
  scale_x_log10() +
  labs(title = "Riemann Sum Estimation of Integral",
       x = "Number of Random Partitions (N) [Log Scale]",
       y = "Estimated Integral") +
  theme_minimal()
```

Estimated Integral Convergence

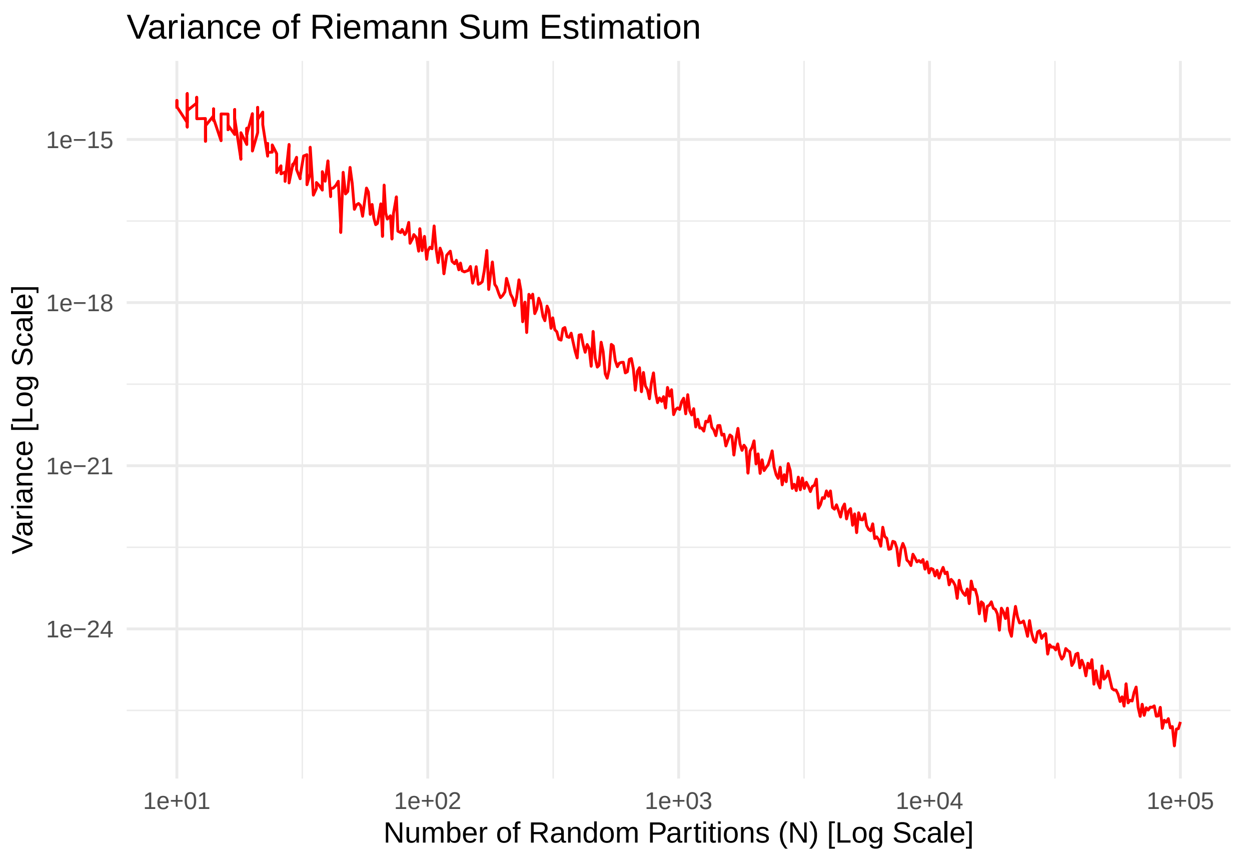
```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
print(p1)
```



Variance Convergence

```r
p2 <- ggplot(results, aes(x = N, y = Variance)) +
  geom_line(color = "red", size = 0.5) +
  scale_x_log10() +
  scale_y_log10() +  # Log scale for variance
  labs(title = "Variance of Riemann Sum Estimation",
       x = "Number of Random Partitions (N) [Log Scale]",
       y = "Variance [Log Scale]") +
  theme_minimal()

print(p2)
```



Observations

The results from the numerical experiment reveal several important insights:

- The estimated integral **converges** to a stable value as the number of partitions N increases, which is consistent with the expected behavior of the Riemann sum.
- The **variance decreases** with increasing N, indicating that larger sample sizes provide more stable and reliable estimates.
- Initially, the variance exhibits some fluctuations, which can be attributed to the randomness in the choice of partition points. However, as N grows, these fluctuations diminish, and the variance stabilizes at much lower values.
- The log-log plot of variance shows a clear downward trend, confirming that increasing the number of partitions leads to better precision in integral estimation.
- Compared to other statistical integration methods such as Monte Carlo, the Riemann sum approach with random partitions provides a good balance between accuracy and computational efficiency, as well as exceptionally low values of variance while giving a very accurate estimation.

Overall, this method demonstrates the effectiveness of using random partitions in the Riemann sum for statistical estimation of integration and highlights how variance analysis provides a deeper understanding of the estimation stability.

```r
cat("Estimated value of the integral:", estimated_integrals[length(estimated_integrals)], "\n")
```

```
## Estimated value of the integral: 2.667642e-07
```

```
cat("Variance for the largest N (", max(N_values), "):", variances[length(variances)], "\n")
```

```
## Variance for the largest N ( 1e+05 ): 1.947205e-26
```

4.Monte Carlo Integration Using Rejection Sampling

###Theory

Monte Carlo integration estimates the integral of a function $f(x)$ over an interval $[a, b]$ by randomly sampling points x_i and computing the fraction of points that fall below the curve:

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

where N is the number of random samples.

Rejection Sampling Approach In rejection sampling, random points (x_i, y_i) are generated within a bounding region, typically a rectangle $[a, b] \times [0, M]$, where M is an upper bound for $f(x)$. The integral is then estimated as the fraction of points that fall below the curve, scaled by the area of the bounding region.

$$I \approx \frac{\text{count of points under } f(x)}{N} \times M(b-a)$$

Numerical Stability Considerations To improve numerical stability, we normalize $f(x)$ by dividing it by its maximum value $f_{\max}$, ensuring that the sampling remains effective. Without this normalization, if $f(x)$ has a very small range, the probability of sampling points under the curve can become too low, leading to unreliable estimates. The final integral estimate is then obtained by scaling back using the normalization factor.

This method allows for robust Monte Carlo integration while minimizing skewed estimations due to rare samples.

Methodology

We implement Monte Carlo integration using rejection sampling within the unit square $[0, 1] \times [0, 1]$. The integral is estimated by generating uniform random points (x_i, y_i) in this domain and determining the fraction of points that fall below the function $f(x)$. This fraction, adjusted by a scaling factor, provides the estimated integral.

This approach can be interpreted as a Bernoulli process, where each sampled point either lies below the curve (success) or above it (failure). Given this probabilistic nature, the variance of the estimate is given by:

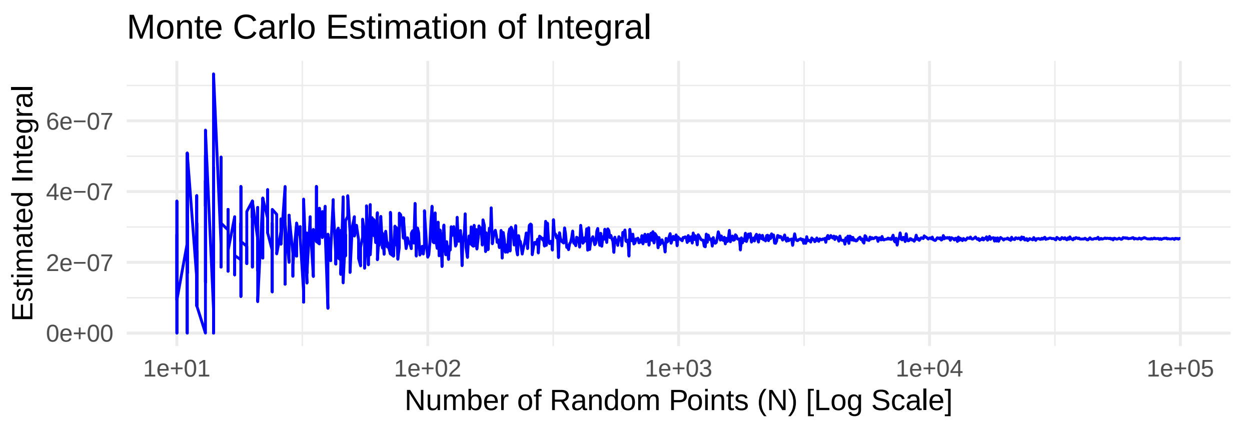
$$\sigma^2 = \frac{I(1-I)}{N}$$

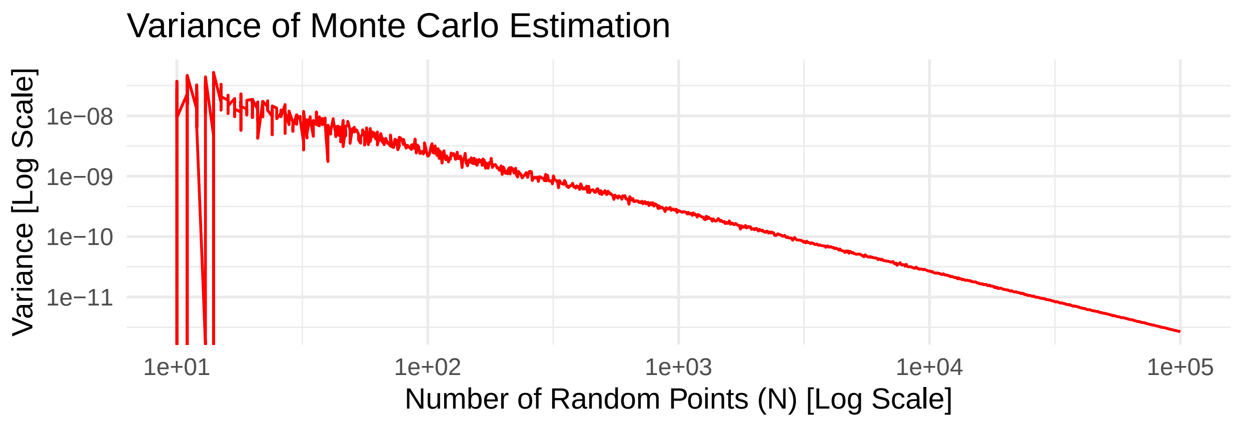
where I is the estimated integral, and N is the number of samples. The variance decreases as N increases, improving the accuracy of the estimate.

To enhance numerical stability, we normalize $f(x)$ by dividing it by its maximum value before performing rejection sampling. This ensures that the sampled points are distributed effectively across the domain, preventing rare event biases that can lead to highly skewed estimates.

```r
library(ggplot2)
library(gridExtra)

f <- function(x) {
  (x^8 * (1 - x)^8 * (25 + 816 * x^2)) / (3164 * (1 + x^2))
}

x_vals <- seq(0, 1, length.out = 10000)
f_max <- max(f(x_vals))

scaling_factor <- 1 / f_max

g <- function(x) scaling_factor * f(x)

N_values <- round(10^(seq(1, 5, length.out = 1000)))
estimated_integrals <- numeric(length(N_values))
variances <- numeric(length(N_values))

for (i in seq_along(N_values)) {
  N <- N_values[i]
  x_rand <- runif(N, 0, 1)
  y_rand <- runif(N, 0, 1)
  under_curve <- (y_rand <= g(x_rand))
  estimated_integrals[i] <- mean(under_curve) / scaling_factor
  variances[i] <- (estimated_integrals[i] * (1 - estimated_integrals[i])) / N
}

results <- data.frame(N = N_values, Integral_Estimate = estimated_integrals, Variance = variances)

p1 <- ggplot(results, aes(x = N, y = Integral_Estimate)) +
  geom_line(color = "blue", size = 0.5) +
  scale_x_log10() +
  labs(title = "Monte Carlo Estimation of Integral", x = "Number of Random Points (N) [Log Scale]", y =
  theme_minimal()

p2 <- ggplot(results, aes(x = N, y = Variance)) +
  geom_line(color = "red", size = 0.5) +
  scale_x_log10() +
  scale_y_log10() +
  labs(title = "Variance of Monte Carlo Estimation", x = "Number of Random Points (N) [Log Scale]", y =
  theme_minimal()

grid.arrange(p1, p2, ncol = 1)
```

```
## Warning in scale_y_log10(): log-10 transformation introduced infinite values.
```





```r
cat("Estimated value of the integral:", estimated_integrals[length(estimated_integrals)], "\n")
```

```
## Estimated value of the integral: 2.675069e-07
```

```r
cat("Variance for the largest N (", max(N_values), "):", variances[length(variances)], "\n")
```

```
## Variance for the largest N ( 1e+05 ): 2.675068e-12
```

Observations

The estimated integral converges as the number of random points increases, demonstrating the expected behavior of Monte Carlo integration using rejection sampling. The variance of the estimate decreases with increasing sample size, as predicted by the theoretical variance formula.

Variance Analysis

- The variance follows the expected relationship $\sigma^2 = \frac{I(1-I)}{N}$, decreasing as N increases.
- Early estimates fluctuate significantly due to randomness in sampling, but these fluctuations diminish with larger N.
- The log-log plot of variance vs. N confirms the inverse proportionality to sample size.

Comparison with Deterministic Methods

- Unlike deterministic methods such as the Riemann sum, which systematically partitions the domain, the Monte Carlo approach relies on random sampling, leading to inherent variability.
- Monte Carlo exhibited greater fluctuations in early iterations compared to the Riemann sum method.
- While both methods ultimately provided similar integral estimates, Monte Carlo required significantly more samples to achieve comparable precision.
- The variance in Monte Carlo integration, though decreasing, remained higher than that observed in the Riemann sum approach.

Conclusion

Monte Carlo integration using rejection sampling effectively estimates the given integral. However, due to its reliance on random sampling, it exhibits higher variance compared to deterministic methods such as the Riemann sum. While Monte Carlo is advantageous in high-dimensional cases where deterministic integration becomes infeasible, for one-dimensional integrals like this, the Riemann sum method proved to be more efficient, achieving smoother convergence with fewer samples.

The variance analysis further highlights that while increasing the sample size reduces uncertainty, Monte Carlo integration requires significantly larger N to achieve the same precision as structured numerical methods. This underscores the trade-off between general applicability and computational efficiency in numerical integration techniques.