Exercise 3 - Monte Carlo Simulation of areas

12433732 - Stefan Merdian

2024-11-08

Task 1

1)

We will use **Monte Carlo integration** with **uniformly distributed random variables** to approximate the integral $\int_1^6 e^{-x^3}\,dx$, and then compare the result with the exact value obtained using R's `integrate()` function.

```
set.seed(123)

#limits of integration
a <- 1
b <- 6

# Number of random samples
n <- 10000
x <- runif(n, min = a, max = b)

# Integralfunction from excersise
f <- function(x) {
  exp(-x^3)
}

# Monte Carlo Simulation
monte_carlo_estimate <- (b - a) * mean(f(x))
print(paste("Monte Carlo estimate of the integral:", monte_carlo_estimate))
```

```
## [1] "Monte Carlo estimate of the integral: 0.0867526312158127"
```

```
# integrate function for comparison
exact_integral <- integrate(f, lower = a, upper = b)
print(paste("Exact integral (using integrate function):", exact_integral$value))
```

```
## [1] "Exact integral (using integrate function): 0.0854683294295814"
```

2)

We use Monte Carlo integration to compute the integral for b = ∞ .

What would be a good density for the simulation in that case?

- The function we are integrating, $\int_1^{\infty} e^{-x^3}\,dx$, is defined over the interval [0,∞]. Therefore, we need a density function that also has support over the entire positive real line, like the exponential distribution. An exponential distribution works well here because it also decays exponentially with increasing values of x. It provides more samples in regions where the integrand is still meaningful, which improves efficiency. We will choose $\lambda = 1$, in this case, it is a reasonable starting point because it balances providing enough samples in the region where e^{-x^3} has significant values while decaying quickly enough to avoid wasting samples in low-contribution areas.

```
# Define the rate parameter for the exponential distribution
lambda <- 1

# Generate random numbers from an exponential distribution with rate lambda
x <- rexp(n, rate = lambda)

# Define the importance sampling function (f(x) / density of x)
# The density of the exponential distribution is d_exp(x) = lambda * exp(-lambda * x)
importance_weighted_values <- f(x) / (lambda * exp(-lambda * x))

# Calculate Monte Carlo estimate using importance sampling
monte_carlo_estimate <- mean(importance_weighted_values)
print(paste("Monte Carlo estimate of the integral:", monte_carlo_estimate))
```

```
## [1] "Monte Carlo estimate of the integral: 0.894973892888367"
```

```
# Use integrate function for comparison
exact_integral <- integrate(f, lower = 0, upper = Inf)
print(paste("Exact integral (using integrate function):", exact_integral$value))
```

```
## [1] "Exact integral (using integrate function): 0.892979511573142"
```

3)

Why Monte Carlo integration agrees in 2. with integrate but not so much in 1.? - The rapid decay of e^{-x^3} means that most contributions to the integral occur for smaller values of x. For the infinite interval, importance sampling with an exponential distribution better matches this behavior compared to uniform sampling in the finite interval. For the infinite interval, we used importance sampling with an exponential distribution, which allows us to allocate more samples to regions where the function has significant values, leading to lower variance and better accuracy. Uniform sampling over a finite interval where the function decays quickly leads to inefficient sampling, as many of the sampled points contribute almost nothing to the integral. In contrast, exponential sampling aligns well with the function's decay and improves efficiency. The importance sampling strategy in the infinite interval case minimizes the variance of the estimate and allows for faster and more reliable convergence compared to uniform sampling in the finite interval. The key difference lies in the sampling strategy. For the finite interval ($b = 6$), using a uniform distribution is less effective due to the rapid decay of the function, leading to higher variance and a less accurate estimate. For the infinite interval, using an exponential distribution with importance sampling allows us to focus sampling efforts where they matter most, leading to an accurate estimate that matches well with the exact value.

Task 2

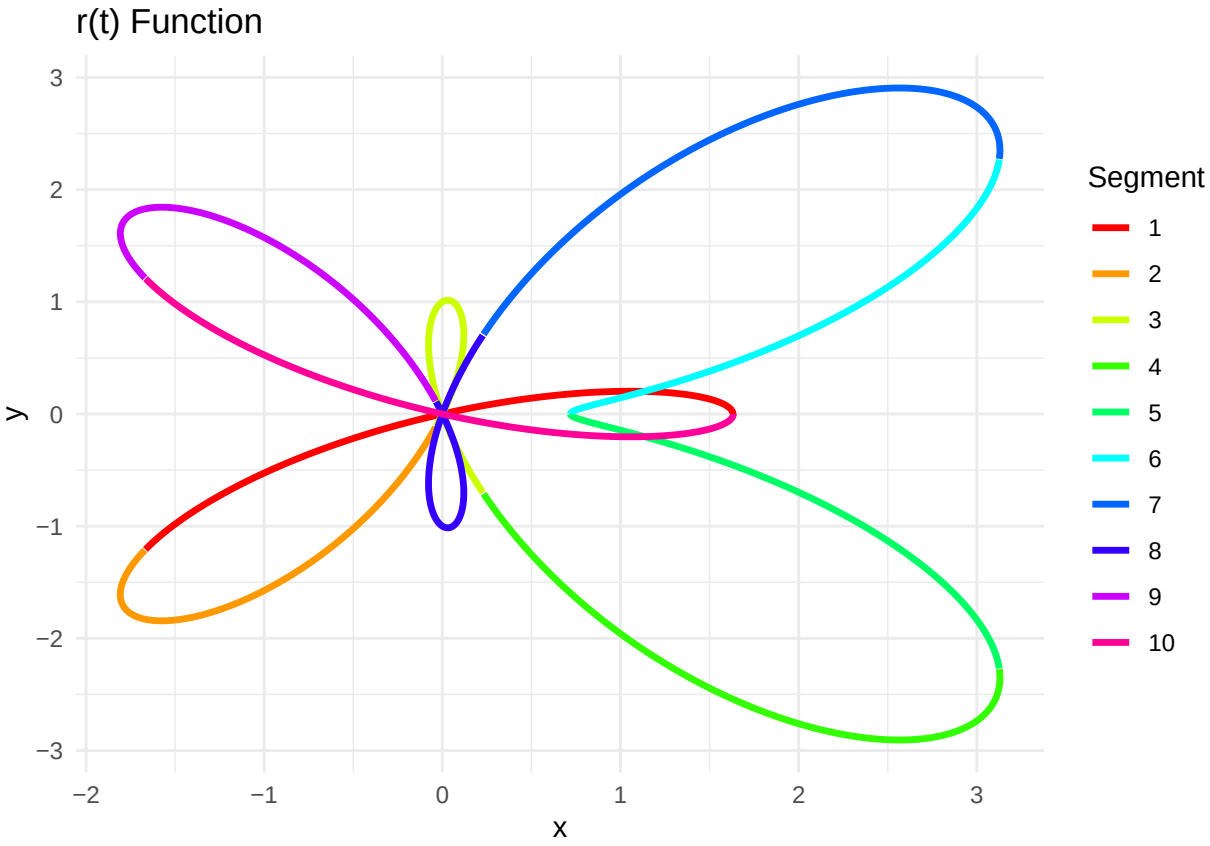
1)

Next, we create a sequence of numbers ranging from — to . Using these values, we calculate the corresponding Cartesian coordinates (x and y) from the polar coordinates defined by the function. Finally, we plot the resulting function.

With 1000 evenly spaced values between — — and , we obtain a dense set of points, allowing us to accurately visualize the entire function.

I divided the graph into 10 segments to better understand which θ values correspond to different parts of the curve. Segment 1 starts at $-\pi$, and Segment 10 ends at π.

```r
library(ggplot2)

r <- function(t) {
  exp(cos(t)) - 2 * cos(4 * t) - (sin(t / 12))^5
}

# Generate sequence of theta values in the interval [-pi, pi]
t_val <- seq(-pi, pi, length.out = 5000)

# Convert polar coordinates to Cartesian coordinates
x_values <- r(t_val) * cos(t_val)
y_values <- r(t_val) * sin(t_val)

# Create a data frame so I can divide the coordinates in segments
data <- data.frame(
  x = x_values,
  y = y_values,
  group = rep(1:10, each = 500)
)

ggplot(data, aes(x = x, y = y, color = factor(group))) +
  geom_path(linewidth = 1.2) +
  labs(
    x = "x",
    y = "y",
    title = "r(t) Function",
    color = "Segment"
  ) +
  theme_minimal() +
  scale_color_manual(values = rainbow(10)) +
  theme(legend.position = "right")
```



2)

We will generate random points that are spread evenly across a rectangle. The rectangle has an **x-range** from -2 to 3.5 and a **y-range** from -3 to 3. Each point will have an **x-coordinate** and a **y-coordinate** that fall somewhere within this rectangle.

```
x_min <- -2
x_max <- 3.5
y_min <- -3
y_max <- 3

bounding_area <- (x_max - x_min) * (y_max - y_min)

n_points <- 1000
random_coordinates <- function(n_points) {
  x_rand <- runif(n_points, min = x_min, max = x_max)
  y_rand <- runif(n_points, min = y_min, max = y_max)

  return(data.frame(x = x_rand, y = y_rand))
}
```

The `is_inside` Function:

1. **Calculate the Angle (θ):**
   - For each random point, we find the angle (θ) between the point and the origin.

2. **Find the Radius**:
   - Using this angle, we calculate the radius of the boundary at that angle.
3. **Convert to Cartesian Coordinates**:
   - Using the radius and angle, we convert the polar coordinates to **Cartesian coordinates** (x and y) to find the **boundary point**.
4. **Calculate Distances**:
   - We calculate the distance from the origin to the **boundary point**, and also the distance from the origin to the **random point**.
5. **Check if the Point is Inside**:
   - Finally, we compare the distances: if the random point is **closer** to the origin than the boundary point, we know that the random point is **inside** the area.

In short, we are checking if the **random point** is **closer** to the origin compared to the **boundary point** at the same angle. If it is, that means the point is **inside**.

```
is_inside <- function(x, y) {

  t <- atan2(y, x)


  radius <- r(t)

  x_border <- radius * cos(t)
  y_border <- radius * sin(t)

  border_distance <- sqrt(x_border^2 + y_border^2)

  point_distance <- sqrt(x^2 + y^2)

  return(point_distance <= border_distance)
}
```

3)

Now we create a list with the values `(100, 1000, 10000, 100000)` to represent the number of random points we want to use in different simulations.

We also create a data frame to keep track of the results for each of these simulations. Specifically, we will store the **percentage of points** that fall inside the area and the **estimated size of the area** for each different number of random points. This helps us see how the accuracy of our area estimate changes as we use more points.

```
n_points_list <- c(100, 1000, 10000, 100000)

summary_table <- data.frame(
  n_points = integer(),
  percentage_inside = numeric(),
  estimated_area = numeric()
)
```

We use Monte Carlo simulation to estimate the area of a shape.

1. **Generate Random Points**: We create sets of random points (100, 1000, 10,000, 100,000) within a bounding box.
2. **Check if Points are Inside**: We use the `is_inside()` function to see if each point is inside the shape.
3. **Calculate Area**: We estimate the area by finding the percentage of points inside and multiplying by the area of the bounding box.
4. **Store Results**: We store the number of points, percentage inside, and estimated area for each set in a table to compare results.

The more points we use, the more accurate the estimate becomes.

```
points_list <- list()

for (n_points in n_points_list) {
  random_points <- random_coordinates(n_points)
  inside_indicator <- is_inside(random_points$x, random_points$y)
  percentage_inside <- (sum(inside_indicator) / n_points) * 100
  estimated_area <- (sum(inside_indicator) / n_points) * bounding_area

  summary_table <- rbind(summary_table, data.frame(
    n_points = n_points,
    percentage_inside = percentage_inside,
    estimated_area = estimated_area
  ))

  points_list[[as.character(n_points)]] <- list(
    x = random_points$x,
    y = random_points$y,
    inside = inside_indicator
  )
}
```

Summarized Table:

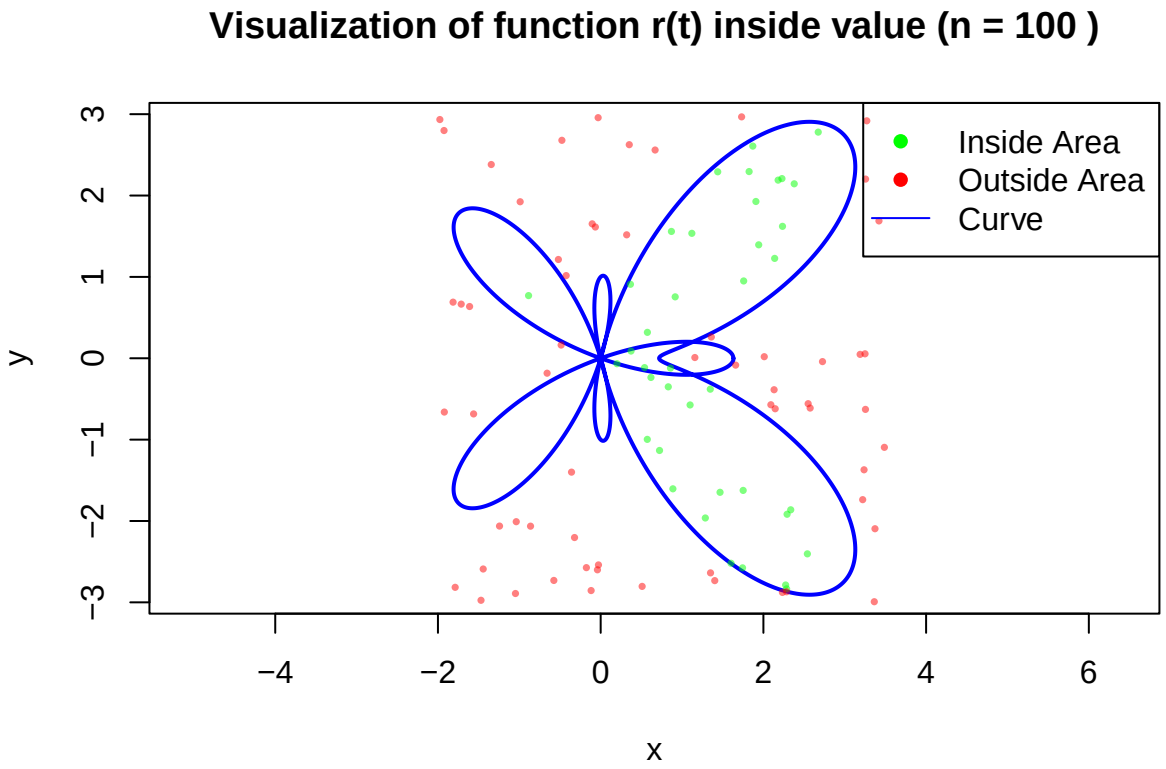
```
print(summary_table)
```

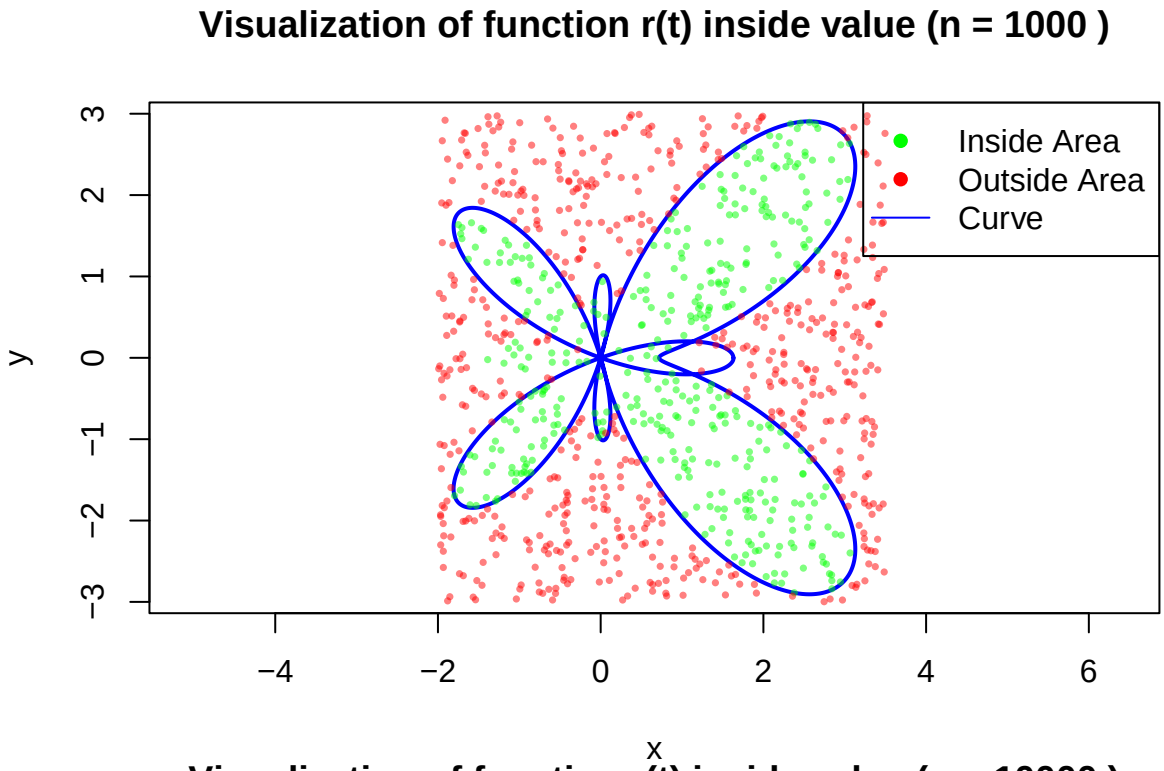
```
##   n_points percentage_inside estimated_area
## 1    1e+02            39.000       12.87000
## 2    1e+03            40.700       13.43100
## 3    1e+04            40.280       13.29240
## 4    1e+05            40.656       13.41648
```

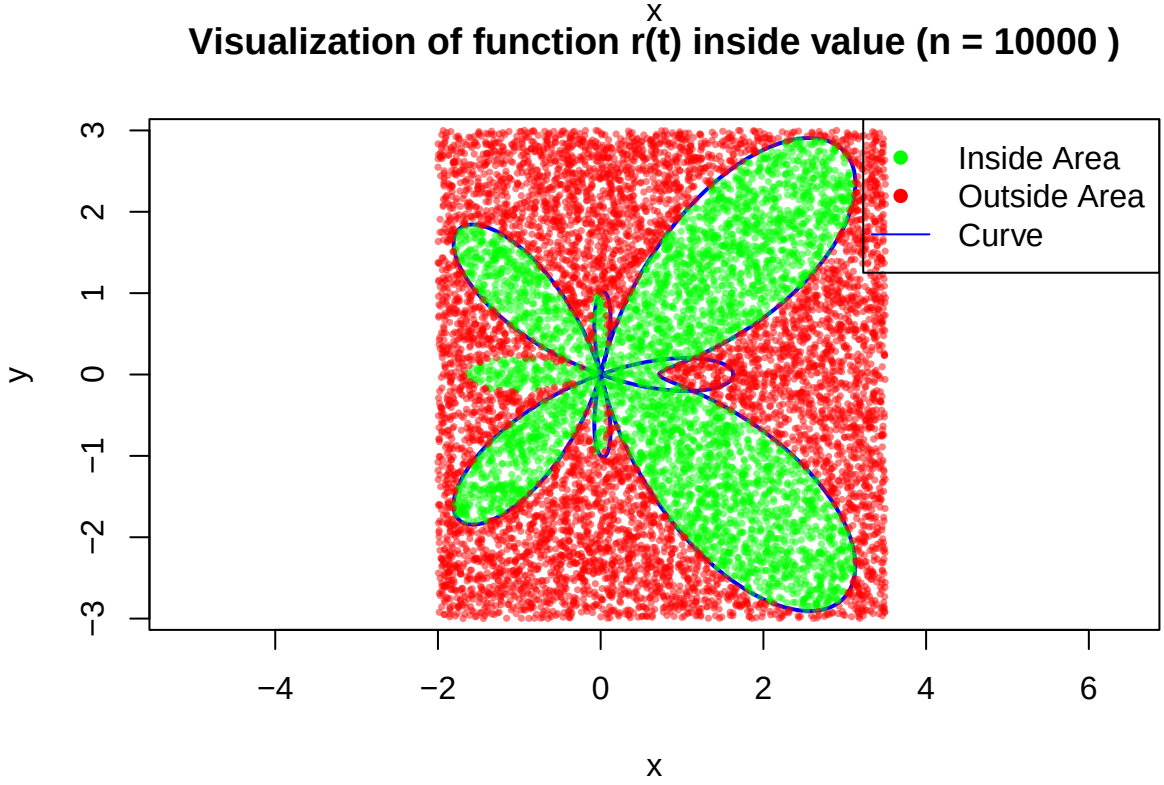
We will plot the function along with the random points for each set of values to visualize which points are inside or outside the area. Each plot will show the function curve in blue, with points inside in green and points outside in red, helping to illustrate how the Monte Carlo estimate improves as we increase the number of points.

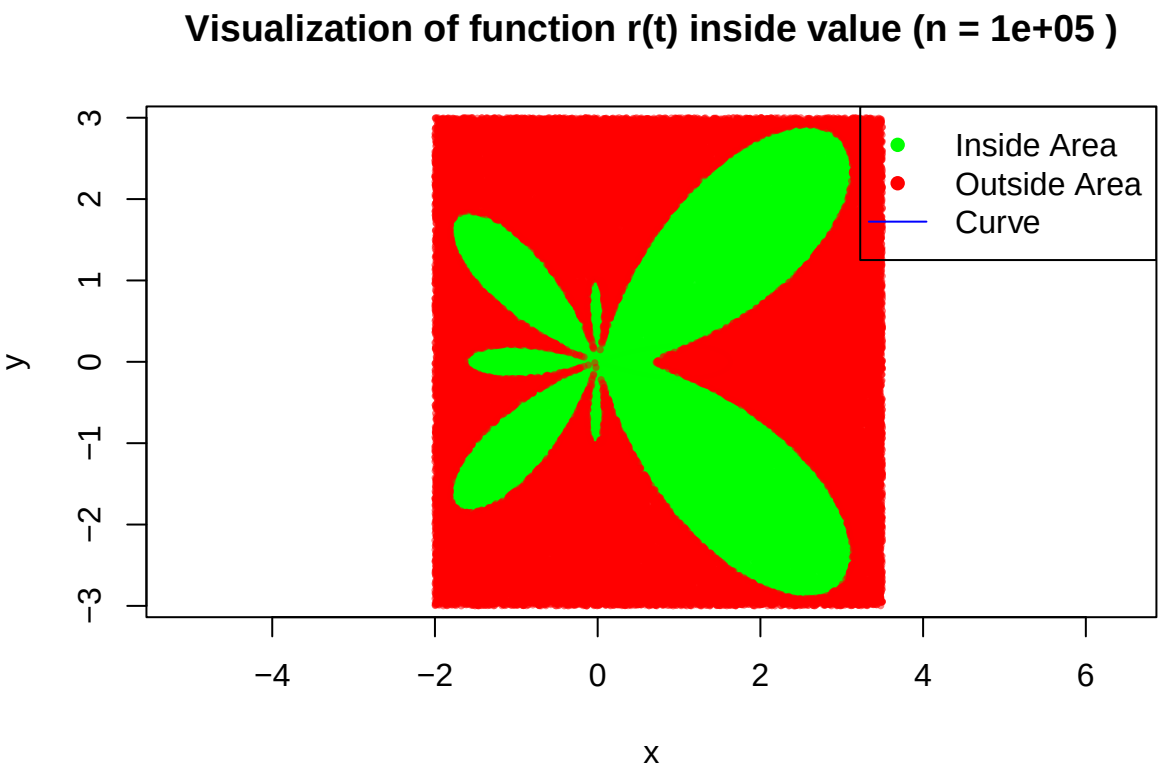
```
for (n_points in n_points_list) {

  current_points <- points_list[[as.character(n_points)]]
  x_rand <- current_points$x
  y_rand <- current_points$y
```

```r
  inside_indicator <- current_points$inside


  plot(x_values, y_values, type = "l", col = "blue", lwd = 2,
       xlab = "x", ylab = "y", main = paste("Visualization of function r(t) inside value (n =", n_points
       asp = 1)

  points(x_rand[inside_indicator], y_rand[inside_indicator], col = rgb(0, 1, 0, 0.5), pch = 16, cex = 0

  points(x_rand[!inside_indicator], y_rand[!inside_indicator], col = rgb(1, 0, 0, 0.5), pch = 16, cex =

  legend("topright", legend = c("Inside Area", "Outside Area", "Curve"),
         col = c("green", "red", "blue"), pch = c(16, 16, NA), lty = c(NA, NA, 1))
}
```




Visualization of function r(t) inside value (n = 1000)
Inside Area
Outside Area
Curve
y
x
−3
−2
−1
0
1
2
3
−4
−2
0
2
4
6


Visualization of function r(t) inside value (n = 10000)
Inside Area
Outside Area
Curve
y
x
−3
−2
−1
0
1
2
3
−4
−2
0
2
4
6



The `is_inside()` function calculates the **distance** from the origin to determine if a point is **inside or outside**. However, due to the way **polar coordinates** handle negative x-values, the function sometimes **misinterprets** the distances, especially on the **left side** of the graph, causing incorrect classifications. This leads to the image looking **flipped** on the left, but the overall point about **estimating area** using random sampling is still valid, despite these minor visual inconsistencies.

4)

Monte Carlo simulation is a technique that uses **random sampling** to estimate a value, like the area of a complex shape like in our case. Instead of directly calculating the exact area using complicated formulas, we can use random points to get a good estimate.

What we did:

1. We generate **random points** within a bounding box that completely contains the shape. For each random point, we check if it lies **inside** the shape or **outside**.
2. If many points are inside, we estimate the area of the shape based on the **proportion of points** that fall inside compared to the total points. As we add more points, our estimate becomes more accurate.

Comparing Monte Carlo to Numerical Integration

- We will use now a method called **numerical integration** to calculate the exact area of the comlex function we got.

```
area_function <- function(t) {
  0.5 * (r(t))^2
}
```

```r
# Calculate the integral from -pi to pi
exact_area <- integrate(area_function, lower = -pi, upper = pi)

print(paste("Exact area of the figure:", exact_area$value))
```

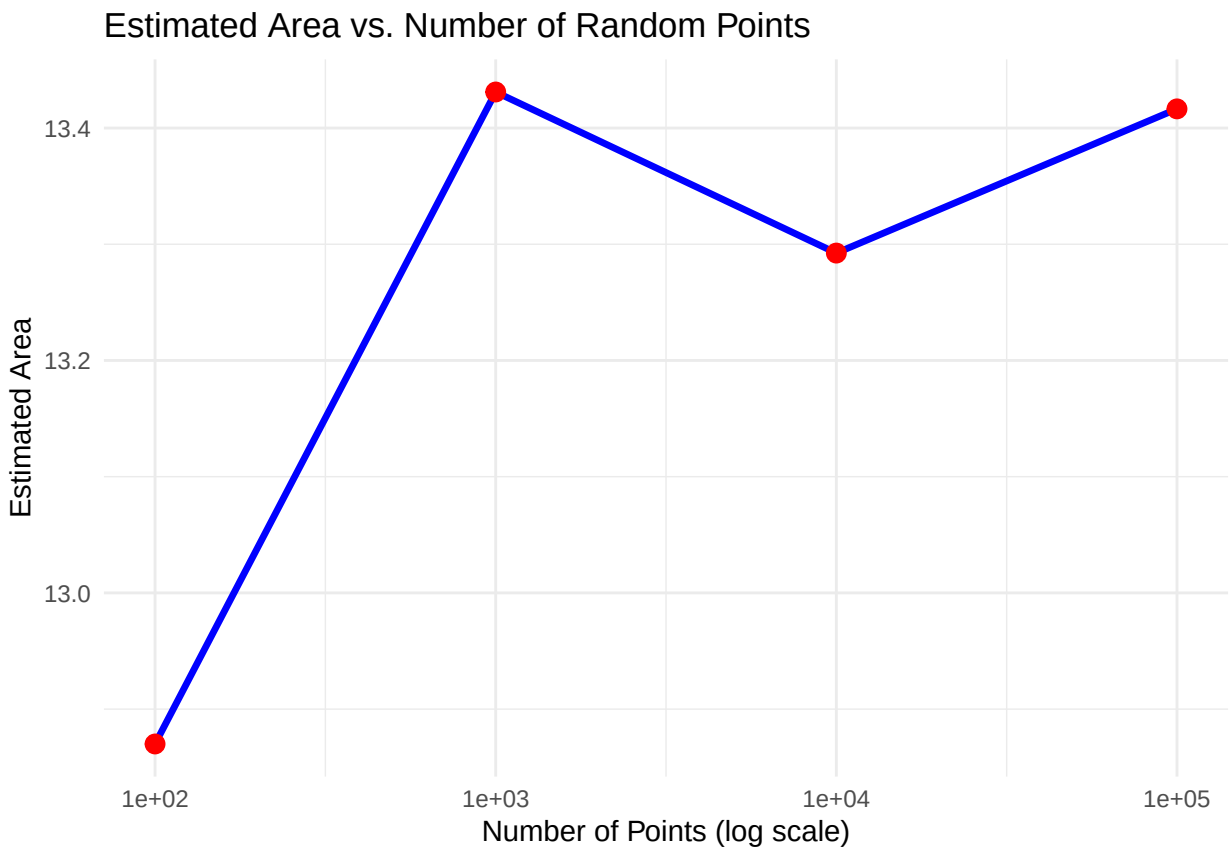
```
## [1] "Exact area of the figure: 13.4103184712479"
```

The value we got is **13.41**.

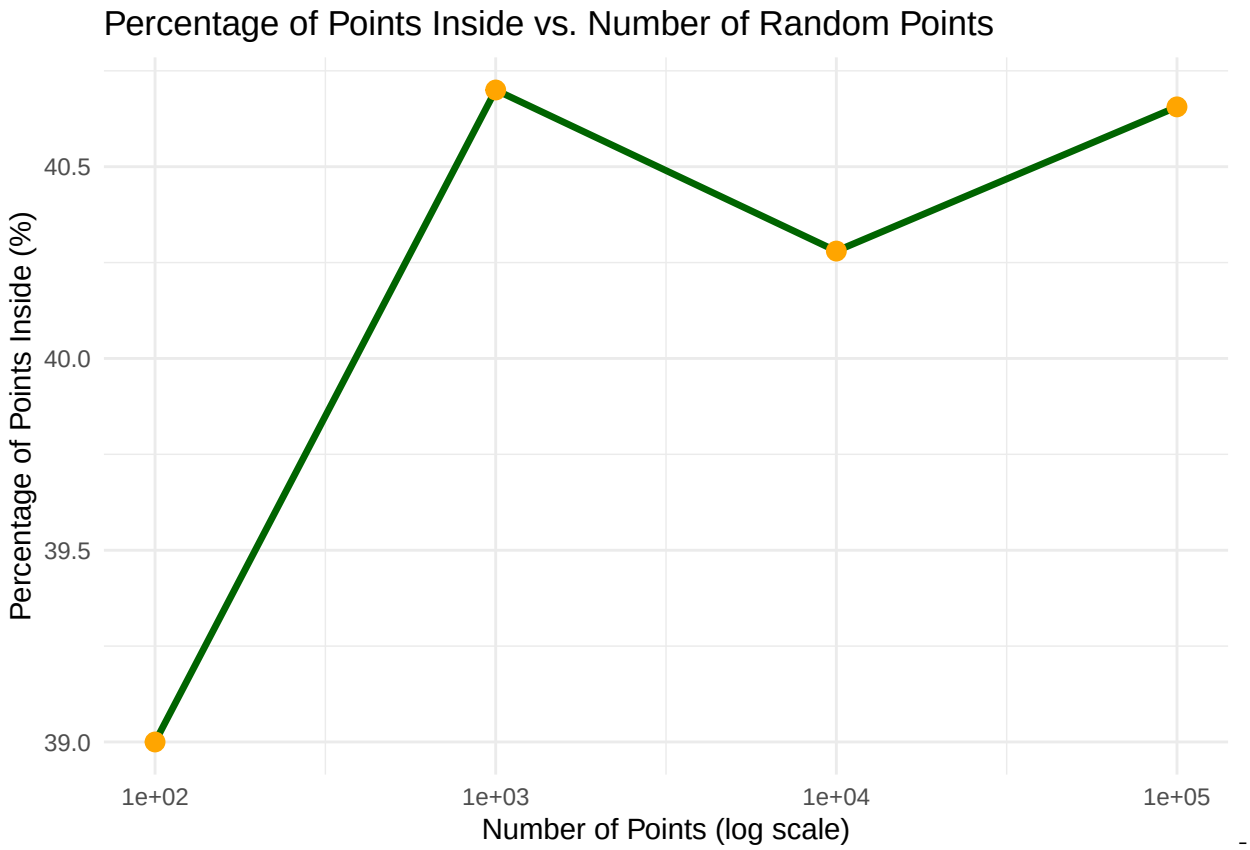
Now we will shows how the **Monte Carlo method** approached the exact value:

```r
library(ggplot2)


ggplot(summary_table, aes(x = n_points, y = estimated_area)) +
  geom_line(color = "blue", size = 1.2) +
  geom_point(color = "red", size = 3) +
  scale_x_log10() +
  labs(
    x = "Number of Points (log scale)",
    y = "Estimated Area",
    title = "Estimated Area vs. Number of Random Points"
  ) +
  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.
```




```r
ggplot(summary_table, aes(x = n_points, y = percentage_inside)) +
  geom_line(color = "darkgreen", size = 1.2) +
  geom_point(color = "orange", size = 3) +
  scale_x_log10() +
  labs(
    x = "Number of Points (log scale)",
    y = "Percentage of Points Inside (%)",
    title = "Percentage of Points Inside vs. Number of Random Points"
  ) +
  theme_minimal()
```



-

At first, with only **100 points**, our estimate was quite far from **13.41**. - When we increased the number of points to **1,000, 10,000, and finally 100,000**, we can see that the **estimated area** got closer and closer to the true value. - This tells us that the **more random points** we use, the better our estimate becomes because it starts to average out and stabilize.

The Monte Carlo simulation might start off less accurate, but as we use more random points, the estimation gets closer to the actual area. This is why Monte Carlo is such a powerful method. It can give us a good estimate for our use case without needing all the data to get a really good idea of what the actual value looks like. To compare it to other areas, for example, conducting a null hypothesis test that involves surveying every citizen in a country would be too much effort, so we can use this approach to take a smaller, random sample and still make an accurate estimation of the overall outcome.