Monte Carlo Simulation of Areas

13-11-2024

Task 1

Approximate the given integral function for b = 6 and b = infinity using Monte Carlo Integration. Compare with R's integrate function.

The subsection below contains the following code:
- Integral approximation for b = 6 using Monte Carlo
- Integral approximation for b = inf using Monte Carlo
- Explanation on why Monte Carlo agrees with b = inf more than b = 6

The given integral function is

```
integrate(f(x), 1, b) = integrate(exp(-x^3), 1, b)
```

1. The code below calculates the Monte Carlo integration for b = 6

```
set.seed(1234)

## Given integral function.

f <- function(x) {
  return(base::exp(-(x^3)))
  ## Note: Added base::exp to avoid conflicts of exp with other packages
}

n <- 100000
x <- runif(n, min = 1, max = 6)

f_x <- f(x)

b <- 6
a <- 1

## Monte Carlo Integration
result <- (b-a) * mean(f_x)

## R Integrate function
actual_integral <- integrate(f, lower = 1, upper = 6)$value


cat("Monte Carlo estimate (b = 6) : ", result, "\n" )
```

```
## Monte Carlo estimate (b = 6) :  0.08559388
```

```
cat("Actual integral value (b = 6) : ", actual_integral, "\n")
```

```
## Actual integral value (b = 6) :  0.08546833
```

```
cat("Error : ", abs(actual_integral - result))
```

```
## Error :  0.0001255457
```

Observation: The error between the R integrate value and the Monte Carlo integrate value is very less (~0.000207), meaning that the Monte Carlo integration is very close to the actual integral value.

2. The code below calculates the Monte Carlo integration for b = infinity

```
x <-rexp(n)


## Monte Carlo Integration
f1 <- base::exp(-(x+1)^3) / dexp(x)
monte_carlo_result <- mean(f1)

## R Integration
actual_integral1 <- integrate(f, lower = 1, upper = Inf)$value

cat("Monte Carlo estimate (b = infinity): ", monte_carlo_result, "\n" )
```

```
## Monte Carlo estimate (b = infinity):  0.08524321
```

```
cat("Actual integral value (b = infinity) : ", actual_integral1, "\n")
```

```
## Actual integral value (b = infinity) :  0.08546833
```

```
cat("Error : ", abs(actual_integral1 - monte_carlo_result))
```

```
## Error :  0.0002251219
```

What is a good density for the simulation?
For integrating a given function, a good density function should have a shape that is similar to exp(-x^3) and decay for increasing x. Therefore, the exponential distribution with parameter, lambda = 1, is good because it decays exponentially.

3. Why does Monte Carlo integration agree in 2?

Using uniform distribution over a finite interval [1, 6] is not always the most efficient for functions that vary significantly over the interval, since it decays faster. But using exponential distribution is better for functions that decay quicker, like the function given in the question. That's why the error for b = inf is smaller than b = 6.

Task 2

Visualizing the area for different values of n, for the given function using Monte Carlo simulation.

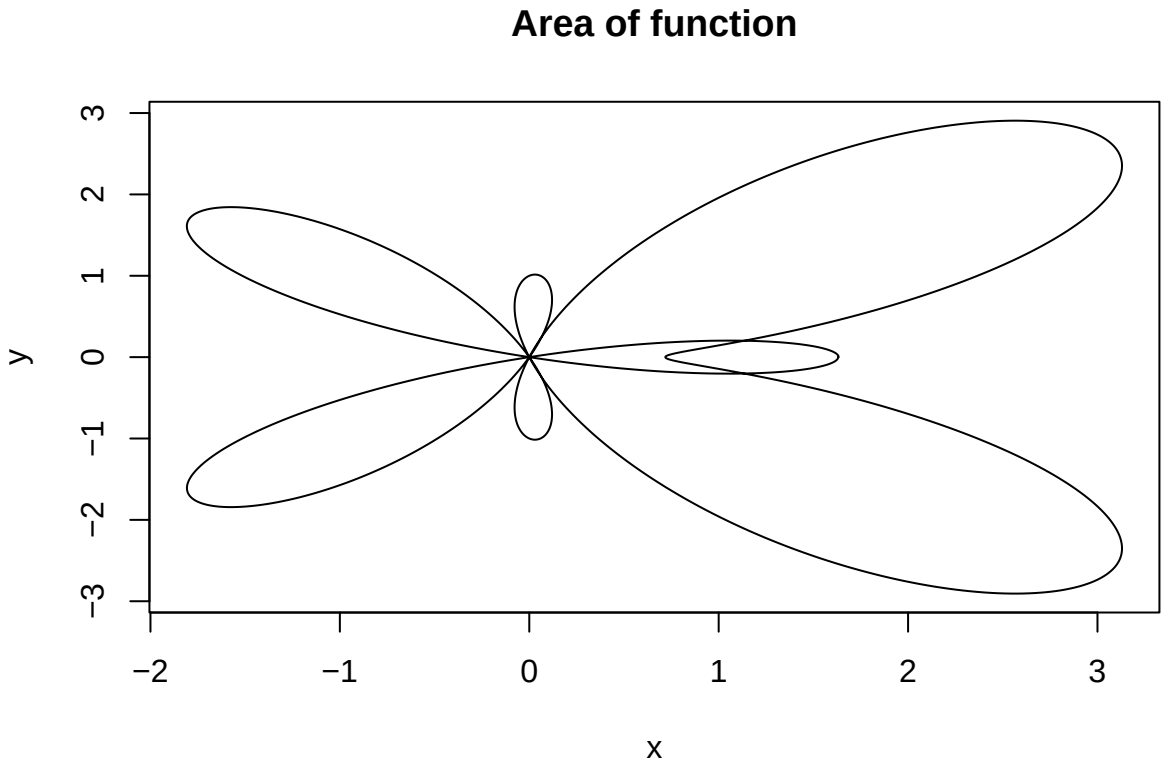
Given function:
r(t) = (exp(cos(t)) - 2 * cos(4*t) - sim(t/12)^5)

for t in [-pi, pi],
using polar coordinates x = r(t) * cos(t) and y = r(t) * sin(t)

The subsection below contains the following code:
- Visualizing the function and the area
- Generating a table with area and percent of points within the area, for different values of n.
- Explanation of the functionality of Monte Carlo simulation

The code below plots the area of the given function

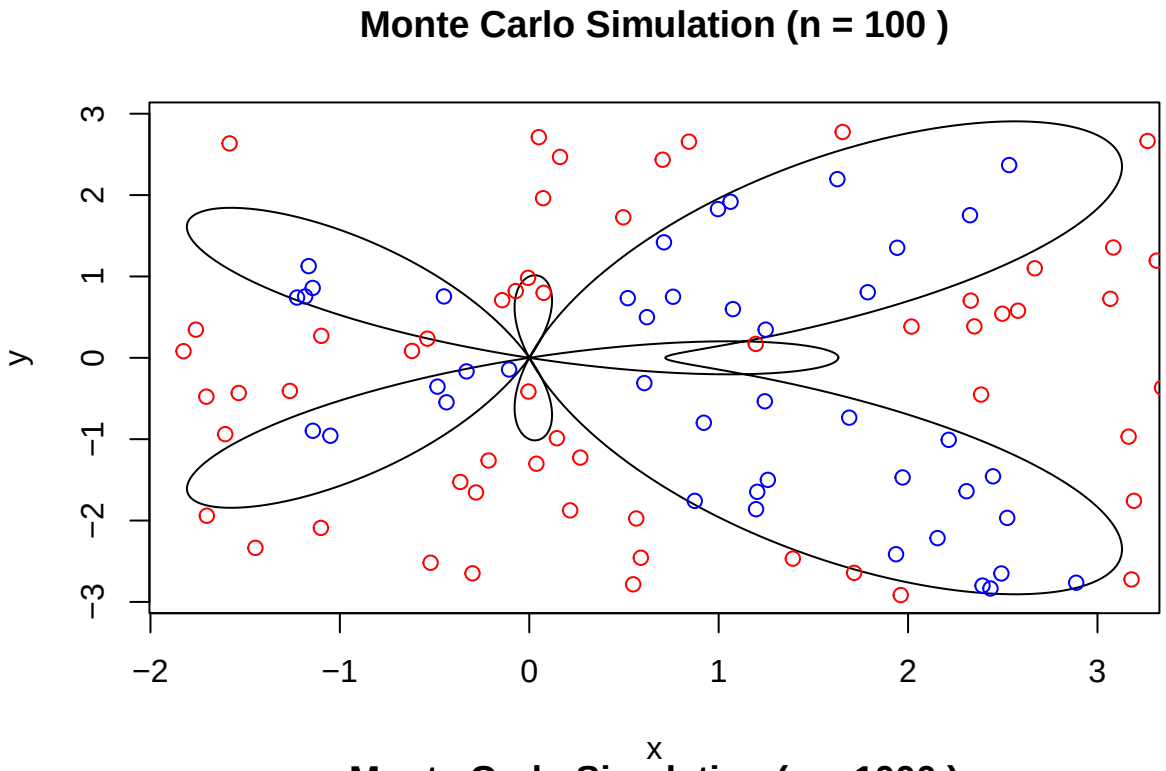
```r
r <- function(t) {
  return(base::exp(cos(t)) - (2 * cos(4 * t) - (sin(t/12)^5)))
}

t_val <- seq(-pi, pi, length.out = 1000)

## x and y values using polar coordinates
x <- r(t_val) * cos(t_val)
y <- r(t_val) * sin(t_val)

plot(x, y, type = 'l', xlab = 'x', ylab = 'y', main = "Area of function")
```

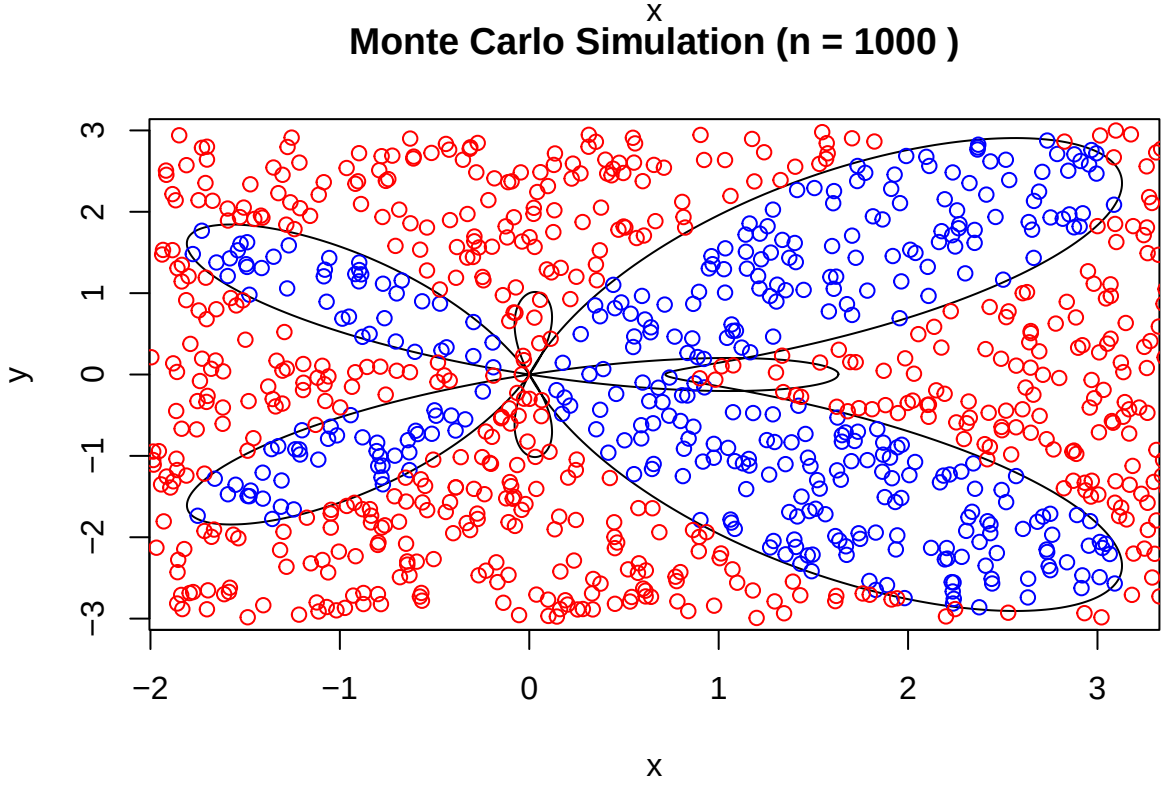


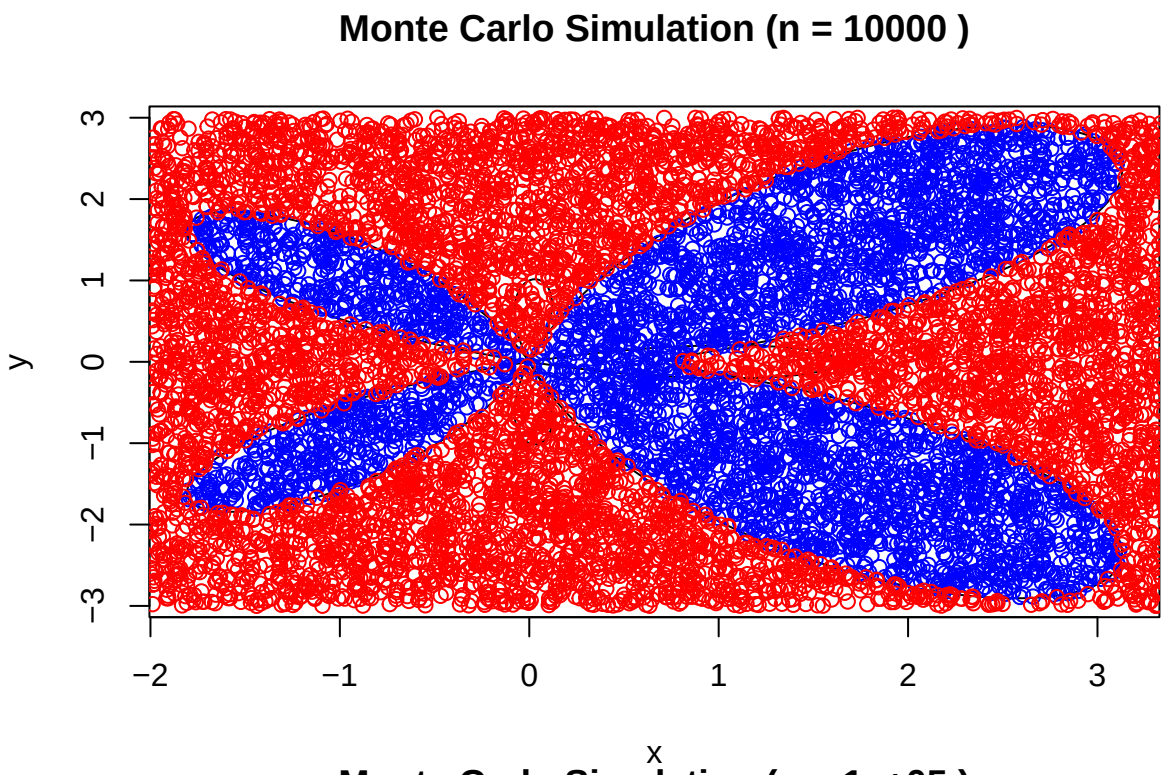
Calculating percent of points for given values of n (100, 1000, 10000, 100000)

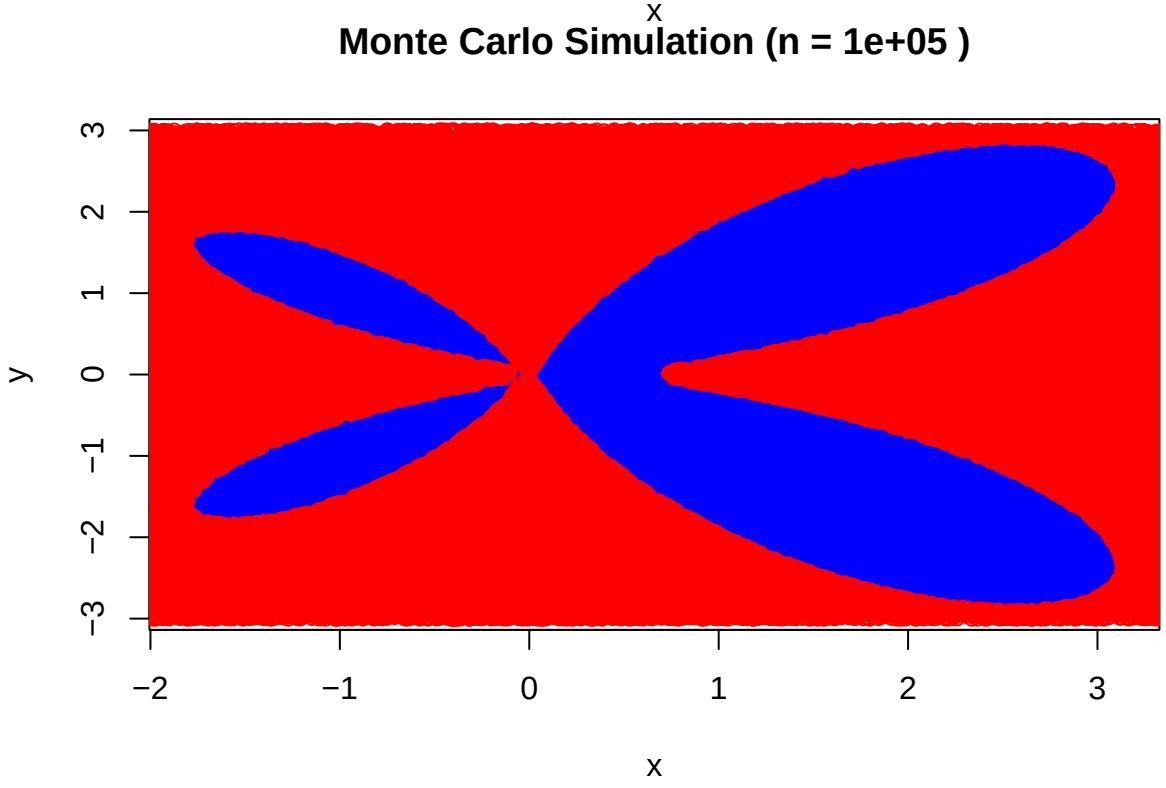
```r
r <- function(t) {
  return(base::exp(cos(t)) - (2 * cos(4 * t) - (sin(t/12)^5)))
}

t_val <- seq(-pi, pi, length.out = 1000)
x <- r(t_val) * cos(t_val)
y <- r(t_val) * sin(t_val)

## x and y intervals

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

```r
## Checking if a point (x, y) is within the area

location <- function(x, y) {
  angle <- atan2(y, x) # angle from -pi to pi
  r_point <- sqrt(x^2 + y^2) # distance from origin to point
  area <- r(angle)
  check <- r_point <= area
  return(check)
}

n_values <- c(100, 1000, 10000, 100000)

## Table to store area and percentage of points
results <- data.frame(N = n_values, Percent_of_points = NA, Area_Estimate = NA)

for (i in 1:length(n_values)) {
  n <- n_values[i]
  x_val <- runif(n, x_min, x_max)
  y_val <- runif(n, y_min, y_max)
  internal_points <- mapply(location, x_val, y_val)
  total_points <- sum(internal_points)

  ## Calculate the percentage of points and the area estimation
  percent_points <- (total_points / n)*100
  area <- ((x_max - x_min) * (y_max - y_min) * total_points) / n

  ## Storing values to table
  results$Percent_of_points[i] <- percent_points
  results$Area_Estimate[i] <- area

  plot(x, y, type = 'l', xlab = 'x', ylab = 'y', main = paste("Monte Carlo Simulation (n =", n, ")"))
  points(x_val[internal_points], y_val[internal_points], col = 'blue')
  points(x_val[!internal_points], y_val[!internal_points], col = 'red')
}
```


Monte Carlo Simulation (n = 100)
y
x
−2
−1
0
1
2
3
−3


Monte Carlo Simulation (n = 1000)
y
x
−2
−1
0
1
2
3
−3






```
print(results)
```

```
##       N Percent_of_points Area_Estimate
## 1 1e+02            43.000      14.19000
## 2 1e+03            38.200      12.60600
## 3 1e+04            38.050      12.55650
```

```
## 4 1e+05            38.355      12.65715
```

Functionality of Monte Carlo Simulation:

The Monte Carlo simulation uses random sampling to estimate unknown values. In the given exercise, we use the method to estimate the area of a given function. The following steps are used:

1. Using the rectangular bounds, we generate random uniform coordinates, that contains the area defined by the function.

2. For each generated point, we check if it falls within the area of the function.

3. The amount of points that fits within the area is used to calculate the percentage of points. Multiplying the total amount of points within the bounds also gives us the area of the function.

Observation: When increasing the number of random points, the accuracy of the Monte Carlo estimation increases. This is also shown in the plots- with increasing values of n, the area becomes smoother around the edges. This shows that the Monte Carlo estimation works very well for shapes that are irregular.