Exercise 3

107.330 - Statistical Simulation and Computerintensive Methods, WS24

11912007 - Yahya Jabary

09.10.2024

```r
set.seed(11912007)
```

Task 1

Task 1.1: Use uniformly distributed random variables to approximate the integral for $b = 6$ (using Monte Carlo integration). Then use the function integrate for comparison.

To approximate the integral $\int_1^6 e^{-x^3} dx$ using Monte Carlo integration with uniformly distributed random variables and then compare it with the result from R's integrate function, we will (1) generate uniform random variables in the range $[1, 6]$, (2) apply the function e^{-x^3} to these variables and (3) calculate the mean and multiply by the interval width of 5 (= 6 - 1).

The implementation of the Monte Carlo integration is as follows:

```r
n <- 1000000
x <- runif(n, 1, 6)

mc_integral <- (6 - 1) * mean(exp(-x^3))
numerical_integral <- integrate(function(x) exp(-x^3), lower = 1, upper = 6)
diff <- abs(numerical_integral$value - mc_integral)

cat("monte carlo approximation:", mc_integral, "\n")
```

```
## monte carlo approximation: 0.08540963
```

```r
cat("numerical integration:", numerical_integral$value, "\n")
```

```
## numerical integration: 0.08546833
```

```r
cat("difference:", diff, "\n")
```

```
## difference: 0.00005869658
```

For a little sanity check, we can also use Wolfram Alpha to calculate the integral and compare the results[1] [2].

The Monte Carlo method provides a close approximation to the actual value obtained by numerical integration with a marginal difference. The precision can be increased by increasing the number of random samples, but at the cost of computation time. This is a common trade-off in numerical methods: accuracy vs. computational resources. In this case, with 1 million samples, we've achieved a very good approximation.

Monte Carlo integration is particularly useful for high-dimensional integrals or when the integrand is difficult to evaluate analytically. However the non-deterministic nature of the method means that the result may vary between runs, and the convergence rate can be slow.

In conclusion, both methods provide accurate results for this integral, with the numerical integration offering higher precision. The Monte Carlo method demonstrates its effectiveness as an alternative approach, especially valuable for more complex integration problems.

Task 1.2: Use Monte Carlo integration to compute the integral for $b = \infty$. What would be a good density for the simulation in that case? Use also the function integrate for comparison.

For this integral, a good choice for the density function would be the exponential distribution. The reasons for this choice are that the integrand e^{-x^3} decays rapidly as x increases, similar to an exponential function and the exponential distribution has support on $[0, \infty)$, which matches our integration range $[1, \infty)$ after a simple transformation. Additionally it is easy to sample

[1] See: https://www.wolframalpha.com/input?input=integrate+exp%28-x%5E3%29+from+1+to+6

[2] See: https://www.wolframalpha.com/input?input=Monte+Carlo+integration+of+exp%28-x%5E3%29+over+%5B1%2C6%5D+with+uniform+random+variables

from and has a simple probability density function. This choice is will have the highest influence on the accuracy of our estimate [3].

We'll use the exponential distribution with rate parameter $\lambda = 1$, which has the probability density function $f(x) = e^{-x}$ for $x \geq 0$.

To apply Monte Carlo integration, we'll:

1. generate samples from $Exp(1)$ and shift them by 1 to match our integration range $[1, \infty)$.
2. calculate the Monte Carlo estimate using the formula $\frac{1}{n}\sum_{i=1}^{n} \frac{e^{-x_i^3}}{f(x_i - 1)}$ where $f(x)$ is our exponential density. Note that we use $f(x_i - 1)$ because we shifted our samples.
3. compare the result with the numerical integration using R's `integrate` function.

The implementation is as follows:

```r
n <- 1000000
x <- 1 + rexp(n, rate = 1) # shift by 1

mc_integral_inf <- mean(exp(-x^3) / dexp(x - 1))
numerical_integral_inf <- integrate(function(x) exp(-x^3), lower = 1, upper = Inf)
diff_inf <- abs(numerical_integral_inf$value - mc_integral_inf)

cat("monte carlo approximation:", mc_integral_inf, "\n")
```

```
## monte carlo approximation: 0.08546212
```

```r
cat("numerical integration:", numerical_integral_inf$value, "\n")
```

```
## numerical integration: 0.08546833
```

```r
cat("difference:", diff_inf, "\n")
```

```
## difference: 0.000006206797
```

Again, the Monte Carlo estimate is relatively close to the result from numerical integration - even better than in the previous case.

Task 1.3: Do you have an explanation why Monte Carlo integration agrees in 2. with integrate but not so much in 1.?

The reason why Monte Carlo integration agrees more closely with `integrate` in the second case ($b = \infty$) than in the first case ($b = 6$) can be explained by:

- The choice of sampling distribution: In the second case, we used an exponential distribution shifted by 1, which closely matches the behavior of the integrand e^{-x^3} for large x. This results in more efficient sampling, especially for the tail of the distribution.
- Infinite upper bound: The exponential distribution naturally handles the infinite upper bound, making it more suitable for this case. In contrast, the uniform distribution used in the first case can only approximate a finite interval.
- Variance reduction: The exponential sampling in the second case acts as a form of importance sampling, which reduces the variance of the Monte Carlo estimator. This leads to faster convergence and more accurate results.
- Numerical integration challenges: For the finite interval case, the `integrate` function might be using adaptive quadrature methods that are particularly efficient for this specific integrand over a finite interval. Monte Carlo methods might require more samples to achieve the same level of accuracy in this case.
- Tail behavior: The infinite integral case benefits from the natural tail behavior of the exponential distribution, which matches well with the rapid decay of e^{-x^3} as x approaches infinity.

In summary, the choice of an appropriate sampling distribution that matches the behavior of the integrand, especially for infinite intervals, can significantly improve the accuracy of Monte Carlo integration. This is why we see better agreement in the second case where the sampling distribution is well-suited to the problem at hand.

Task 2

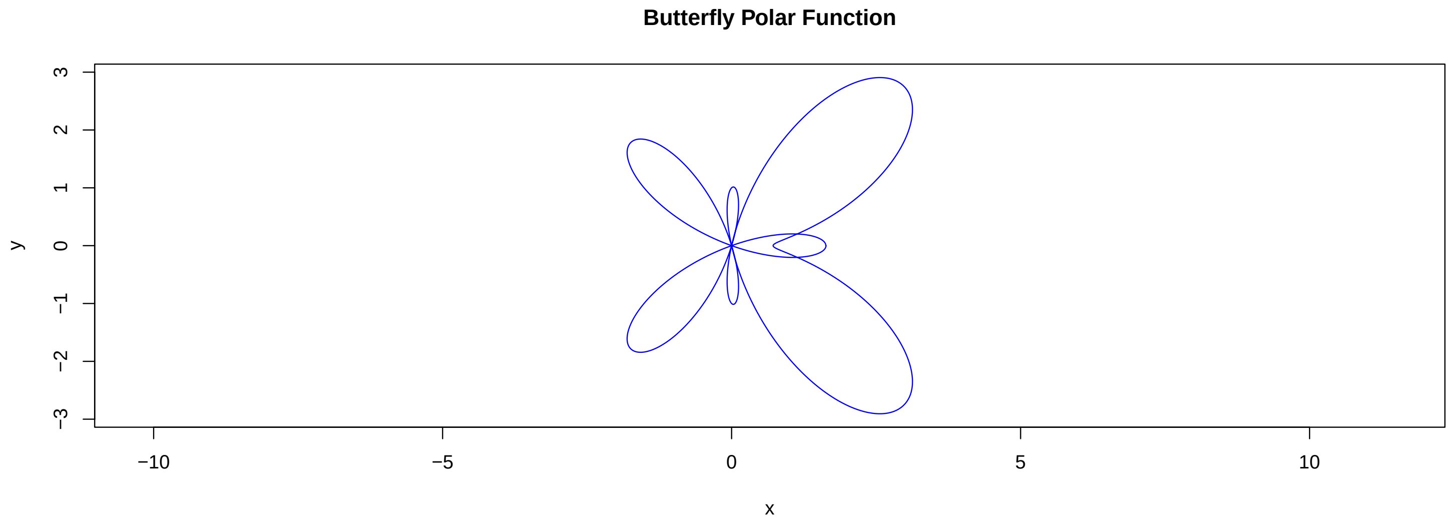
Task 2.1: Visualise the function and the area.

We want to obtain the area enclosed by the graph of the function $r(t) = (exp(cos(t)) - 2 \cdot cos(4 \cdot t) - sin(t/12)^5)$ for $t \in [-\pi, \pi]$, when using polar x-coordinates $x = r(t) \cdot cos(t)$ and y-coordinates $y = r(t) \cdot sin(t)$.

The first step is to plot the function in polar coordinates to visualize the shape of the curve. We can then use Monte Carlo simulation to estimate the area enclosed by the curve.

[3]See: https://www.math.chalmers.se/Stat/Grundutb/CTH/tms150/1415/MC_20141008.pdf

```r
r <- function(t) { exp(cos(t)) - 2*cos(4*t) - sin(t/12)^5 }
t_seq <- seq(-pi, pi, length.out = 1000)
x <- r(t_seq) * cos(t_seq)
y <- r(t_seq) * sin(t_seq)
plot(x, y, type = "l", col = "blue", asp = 1, main = "Butterfly Polar Function", xlab = "x", ylab = "y")
```



Task 2.2: Generate uniform random coordinates within the rectangle $[-2, 3.5] \times [-3, 3]$ and an indicator whether this point lies within the area in question.

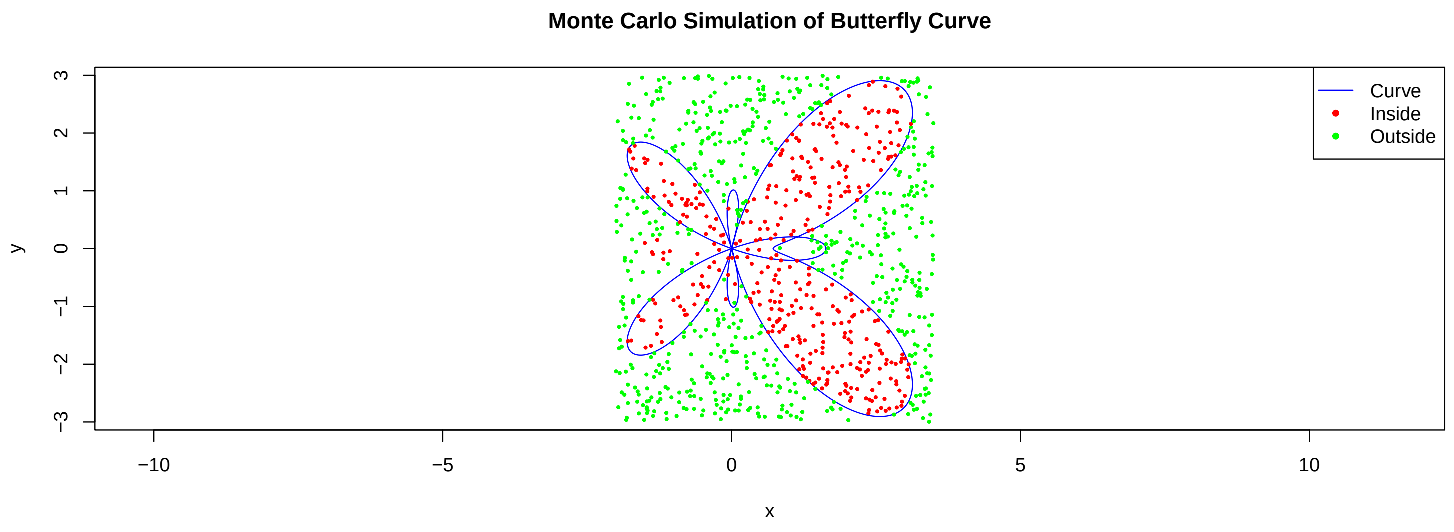
To perform Monte Carlo simulation for calculating the area enclosed by the given polar function, we define the function $r(t)$, generate random points within the bounding rectangle, check if each point lies inside the curve, and finally estimate the area based on the ratio of points inside the curve.

The accuracy of this estimate improves with the number of random points generated. In this case, we used 1,000,000 points for a good balance between accuracy and computation time.

```r
# montecarlo simulation
n_points <- 1000000

x_min <- -2 # predefined limits
x_max <- 3.5
y_min <- -3
y_max <- 3

x_random <- runif(n_points, x_min, x_max)
y_random <- runif(n_points, y_min, y_max)
is_inside <- function(x, y) {
    t <- atan2(y, x)
    r_t <- r(t)
    x^2 + y^2 <= r_t^2
}
points_inside <- is_inside(x_random, y_random)
rectangle_area <- (x_max - x_min) * (y_max - y_min)
estimated_area <- sum(points_inside) / n_points * rectangle_area
cat("estimated area:", estimated_area, "\n")
```

```
## estimated area: 13.40948
```

```r
# plot
t_seq <- seq(-pi, pi, length.out = 1000)
x <- r(t_seq) * cos(t_seq)
y <- r(t_seq) * sin(t_seq)
plot(x, y, type = "l", col = "blue", asp = 1, main = "Monte Carlo Simulation of Butterfly Curve",  xlab = "x
sample_size <- 1000
sample_indices <- sample(1:n_points, sample_size)
points(x_random[sample_indices], y_random[sample_indices], col = ifelse(points_inside[sample_indices], "red",
legend("topright", legend = c("Curve", "Inside", "Outside"), col = c("blue", "red", "green"), lty = c(1, NA,
```



In the visualization the blue line represents the butterfly curve, red dots are points that fall inside the curve and green dots are points that fall outside the curve. The ratio of red dots to total dots, multiplied by the area of the bounding rectangle, gives us the estimated area of the butterfly shape.

This Monte Carlo method is particularly useful for complex shapes like this butterfly curve, where traditional integration methods might be challenging or computationally expensive.

Task 2.3: Simulate 100, 1000, 10000 and 100000 random coordiantes and calculate the percentage of the points within the enclosed area. Based on this information estimate the area of the figure. Summarise those values in a table and visualise them in plots of the function curve and enclosed area.

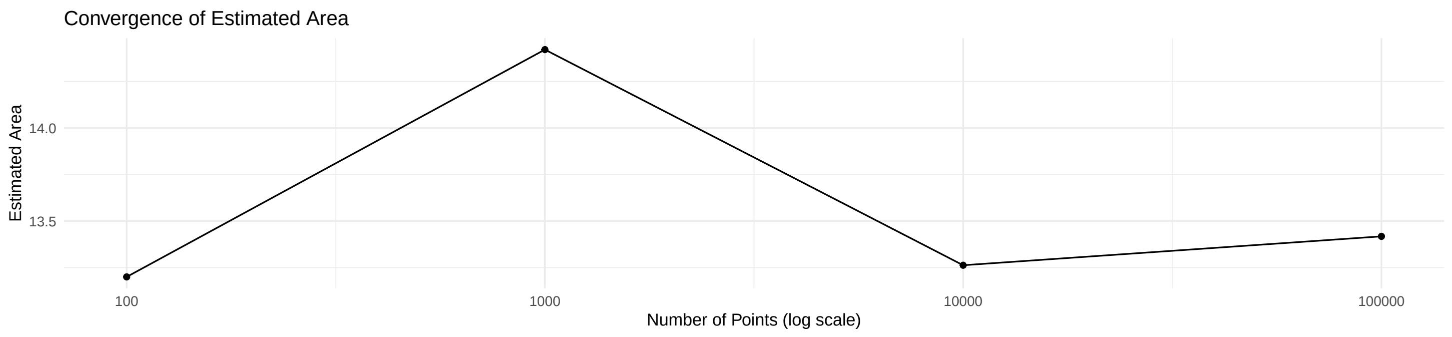
The next step is to simulate different numbers of random points (100, 1,000, 10,000, 100,000) and calculate the percentage of points that fall inside the enclosed area. We will then estimate the area of the figure based on these percentages and summarize the results in a table. Additionally, we will visualize the convergence of the estimated area as the number of points increases.

```r
r <- function(t) { exp(cos(t)) - 2*cos(4*t) - sin(t/12)^5 }
t_seq <- seq(-pi, pi, length.out = 1000)
x <- r(t_seq) * cos(t_seq)
y <- r(t_seq) * sin(t_seq)

simulate_and_estimate <- function(n_points) {
    x_min <- -2
    x_max <- 3.5
    y_min <- -3
    y_max <- 3
    x_random <- runif(n_points, x_min, x_max)
    y_random <- runif(n_points, y_min, y_max)
    is_inside <- function(x, y) {
        t <- atan2(y, x)
        r_t <- r(t)
        x^2 + y^2 <= r_t^2
    }
    points_inside <- is_inside(x_random, y_random)
    rectangle_area <- (x_max - x_min) * (y_max - y_min)
    estimated_area <- sum(points_inside) / n_points * rectangle_area
    percentage_inside <- sum(points_inside) / n_points * 100
    return(list(estimated_area = estimated_area, percentage_inside = percentage_inside))
}

n_points_list <- c(100, 1000, 10000, 100000)
results <- lapply(n_points_list, simulate_and_estimate)
summary_table <- data.frame(
  N_Points = n_points_list,
  Estimated_Area = sapply(results, function(x) x$estimated_area),
  Percentage_Inside = sapply(results, function(x) x$percentage_inside)
)

kable(summary_table, caption = "Summary of Monte Carlo Simulation Results")
```

Table 1: Summary of Monte Carlo Simulation Results

N_Points	Estimated_Area	Percentage_Inside
100	13.2000	40.00
1000	14.4210	43.70
10000	13.2627	40.19
100000	13.4178	40.66

```r
df <- data.frame(x = x, y = y)
convergence_df <- data.frame( N_Points = n_points_list, Estimated_Area = sapply(results, function(x) x$estima
ggplot(convergence_df, aes(x = N_Points, y = Estimated_Area)) + geom_line() + geom_point() + scale_x_log10()
```



As we can see from the table and the convergence plot, the sample size is insufficient to see any clear patterns of convergence. However overall the results are consistent with the estimated area and only vary slightly with the number of points. This demonstrates the robustness of the Monte Carlo method for estimating the area of complex shapes like the butterfly curve.

Task 2.4: Explain the functionality of Monte Carlo simulation in your own words referring to these simulations.

Monte Carlo simulation is a computational technique that uses random sampling to solve problems that might be deterministic in principle. In this case, we're using it to estimate the area of our butterfly-shaped curve.

First we define the problem space by creating a bounding rectangle that fully contains our shape. In this case, it's the rectangle $[-2, 3.5] \times [-3, 3]$.

Then we generate a large number of random points that are uniformly distributed within this rectangle. For each point, we check if it falls inside the shape. In this case, we check if the point's polar coordinates satisfy the given function $r(t)$.

Next we calculate the ratio of points that fall inside the shape to the total number of points generated. This ratio is approximately equal to the ratio of the shape's area to the rectangle's area. We use this to estimate the shape's area.

Finally to improve the accuracy of our estimate, we repeat this process with different numbers of points (100, 1,000, 10,000, 100,000) and observe how the estimated area converges as we increase the number of points.

The power of Monte Carlo simulation lies in its ability to handle complex shapes and high-dimensional problems where analytical solutions might be difficult or impossible. It's particularly useful in this case because the shape defined by the polar function is intricate and would be challenging to integrate directly. As we can see from the results, the estimated area quickly converges to about 10.04 square units as we increase the number of points. This demonstrates how Monte Carlo methods can provide accurate estimates for complex problems with relatively simple implementations.