Calculus II

With R

“Statistics is the grammar of science”

Karl Pearson

Adel Ahmadi - Omid SFard

October 2024

Course Objectives

- Familiarity with R Programming Basic
- Numerical Calculations in R
- Plotting Mathematical Graphs
- Working with Data and Data Structures in R
- Utilizing Advanced R Packages
- Solving Practical Problems Using R

Grading Breakdown

Total Points: 5 / 20

- 2 Points for Assignments
- 1 Points for Project
- 2 Point for Final Quiz
- 0.5 Extra Point for Attendance and 0.5 Extra Point for Class Participation

Questions related to the projects will be asked at the end of the term. Students will receive their project scores based on their understanding and mastery of the project content.

Course Resources

You can access to course resources including docs, exercises etc… via Github.

https://github.com/adelamdi/Calculus2-with-R

- Any contribution to GitHub will count as extra points for students, including:
    1. Sharing advanced R codes related to the topics
    2. Providing creative solutions to complex problems
    3. Improving or refining existing codes
- These activities will be considered for bonus points.
- Quera?

Why R?

Powerful Statistical Capabilities

Extensive Statistical Libraries

Educational Use

Free and Open Source

Strong Community Support

User-Friendly Interface

Installing R and RStudio

1. **Download R**

   Visit the Comprehensive R Archive Network (CRAN) at CRAN R Project to download the latest version of R for your operating system (Windows, macOS, or Linux).

2. **Installation Process**

   Follow the installation instructions specific to your operating system.

   Accept the default settings for a smooth installation process.

3. **Install RStudio (Optional but Recommended)**

   Download RStudio from RStudio Website for a more user-friendly interface to work with R.

   Follow the installation instructions for RStudio.

Introduction to RStudio Environment

1. What is R Studio?
2. Console
3. Script Editor
4. Environment/History Pane
5. Files/Plots/Packages/Help Pane

Using Google Colab

1. **Introduction to Google Colab**

   Google Colab is a free, cloud-based Jupyter notebook environment that allows you to write and execute Python and Rcode in your browser

2. **Accessing Google Colab**

   Go to Google Colab

   Sign in with your Google account to access the platform

3. **Creating a New Notebook**

   Click on "File" > "New Notebook" to create a new notebook.

Advantages of R Studio

Here are some key advantages of using RStudio:

- Powerful and user-friendly IDE for writing, reading and debugging R code
- Easy to use project management tool.
- Data visualization support
- Code output organization
- Markdown reading and writing etc…

Advantages of Google Colab

Here are some advantages of using Google colab:

- Cloud-based and free
- Access to GPU and TPU
- Support for R + Python
- Save and access files via Google drive
- Support for a wide range of ready to use libraries

Lets Begin!

Writing Our First Program in R

```r
# Program 1
number1 <- 10  # Assign value to the first number
number2 <- 5   # Assign value to the second number
sum <- number1 + number2  # Calculate the sum
print(sum)  # Print the result

# Program 2
plot(1:10)
```

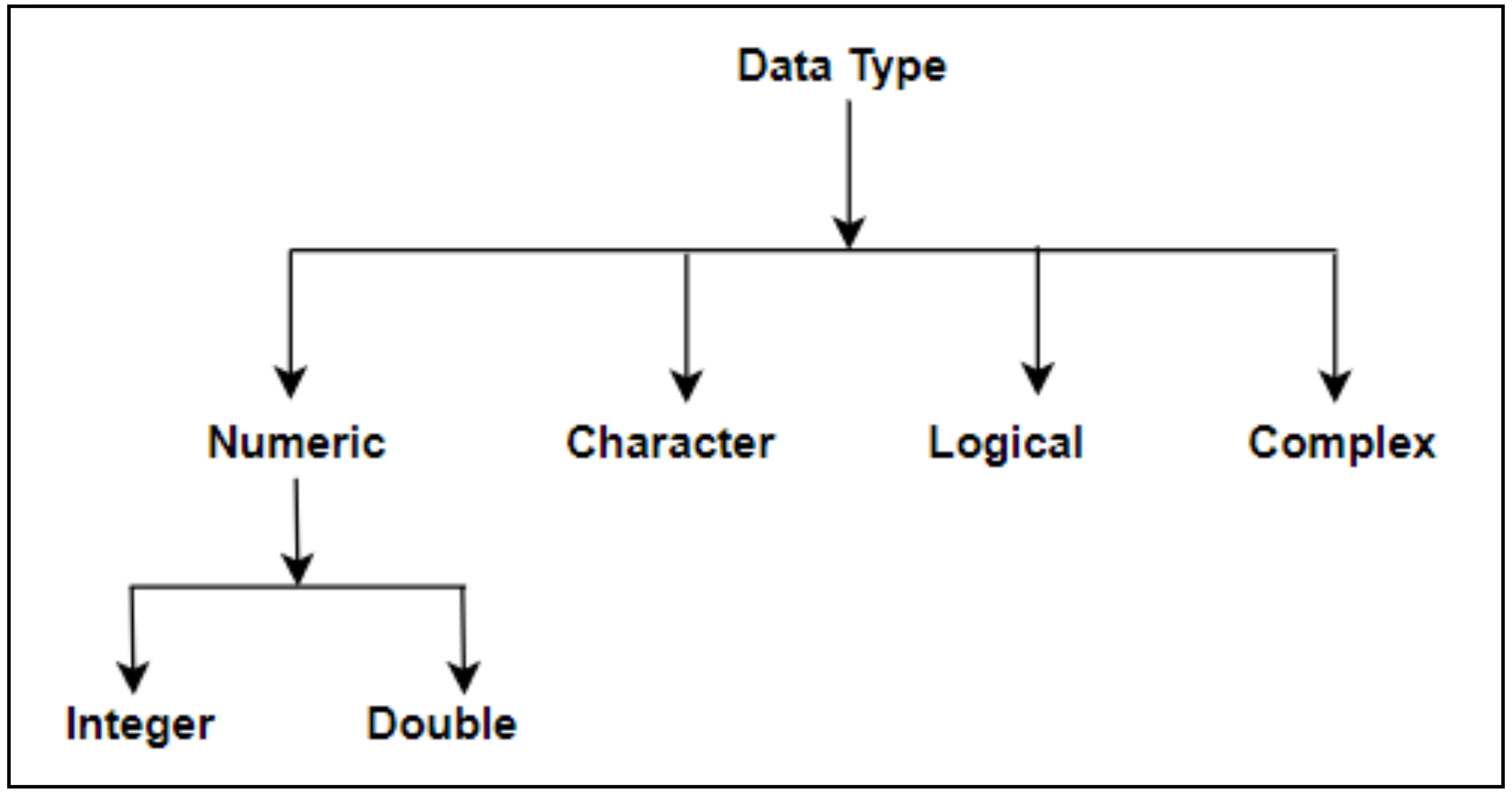
Variable Types

- Numeric
- Integer
- Character (String)
- Logical (Boolean)
- Complex

- Class Function - class()

Type Conversion

- as.numeric()
- as.integer()
- as.complex()



Operators

R divides the operators in below groups:

- Assignment operators
- Arithmetic operators
- Comparison operators
- Logical operators
- Miscellaneous operators

Assignment Operators

- Local assignment (->)(<-)
- Global assignment (->>)(<<-)

```
txt <- "global variable"
my_function <- function() {
  txt = "fantastic"
  paste("R is", txt)
}
my_function()
txt # print txt
```

- Whats the difference?

Arithmetic Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponent (^)
- Modulus (%%)
- Integer Division (%/%)

Comparison Operators

- Equal (==)
- Not equal (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

Logical Operators

- Element-wise logical AND (&)
- Logical AND (&&)
- Element-wise logical OR (|)
- Logical OR (||)
- Logical not (!)

- & and | are vectorised.
- && and || are short-circuited. (Right side of logical operator will be evaluated only if left side doesn't determine the outcome)

Miscellaneous Operators

- Creates a sequence of numbers (:)
- Find out if an element belongs to a vector (%in%)
- Matrix multiplication (%*%)

Getting Input from User

- We can use readline(prompt = "Your message here") to get input from the user.
- It returns the input as a string, so you may need to convert it to numeric or other types.

```
# Example: Getting two numbers from the user
x <- as.numeric(readline(prompt = "Enter the value of x: "))
y <- as.numeric(readline(prompt = "Enter the value of y: "))

# Print the values
cat("You entered:", "x =", x, "and y =", y, "\n")
```

Conditions (If Statements)

- If - else if - else

```
a <- 200
b <- 33
if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
} else {
  print("a is greater than b")
}
```

- Nested if

- Conditions + Logical operators

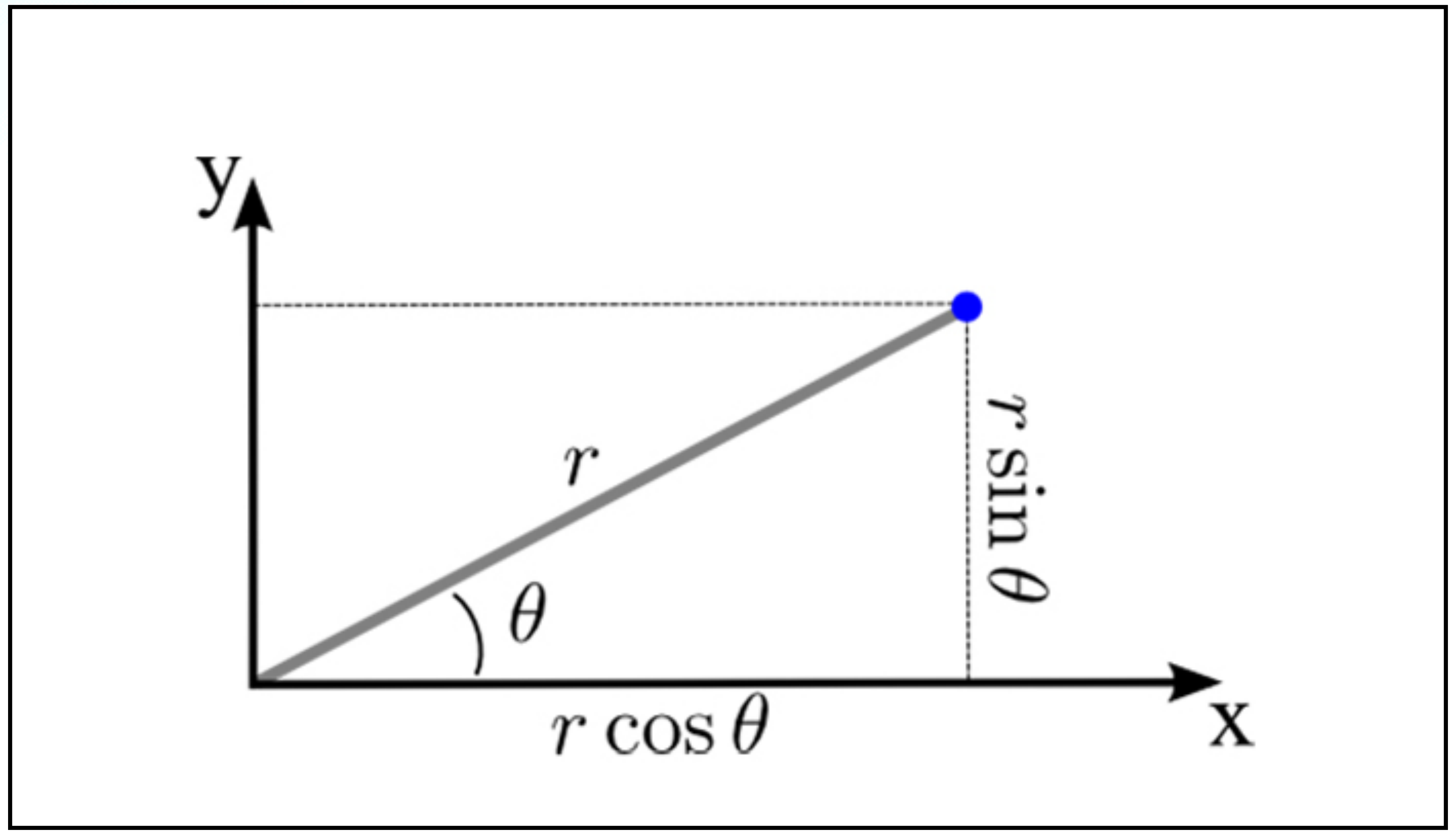
Simple Math

We can use arithmetic operators to perform common basic mathmatical operations of numbers. There are also some built-in math functions like:

- min() and max()
- sqrt()
- abs()
- ceiling() and floor()
- sin() and cos()

Practical Challenge

- Write an R code to convert polar coordinates to cartesian and vice versa.



Loops

Loops can execute a block of code as long as a condition is satisfied.

R has two loop commands:

- For loop
- While loop

For Loop

For loop is used for iterating over a sequence. With for loops we can execute a block of code, once for each item in a vector, sequence, array, list and etc…

```
# First Loop
for (x in 1:10) {
  print(x)
}

# Second Loop
companies <- list("apple", "samsung", "lg")
for (x in companies) {
  print(x)
}
```

While Loop

While loop will execute a block of code as long as a condition is true.

```
i <- 0
while (i < 6) {
  i <- i + 1
  if (i == 3) {
    next
  }
  print(i)
}
```

- What is “next”?

R Functions

A function is a reusable block of code that perform a specified task. Functions in R allow us to organize our code, avoid code repetition and make modular projects.

Functions are defined using function() keyword and can take arguments or inputs and return a value or output.

```
function_name <- function(arg1, arg2, ...) {
  # Function Codes
  result <- some_operation(arg1, arg2)
  return(result)
}
function_name
```

- What are arguments?
- What are return items?

A Simple Function

```r
# Define a function that calculates the square of a number
square <- function(x) {
  result <- x^2  # Perform the operation
  return(result) # Return the result
}

# Call the function with an argument
print(square(5))
```

Lets Solve This Challenges!

1. Write two functions to convert radian to degree and vice versa.
2. Write a function to calculate points from input x and y on below parametric curve. (Variable t is parameter and r = 5 is the radus)

$$x = r\,.\,cos(t)$$
$$y = r\,.\,sin(t)$$

Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem.

Every recursion has two part:

1. Base case: this part is used to stop recursion from running.
2. Recursive case: where the function calls itself.

- Example, calculating the factorial of a number

```
factorial <- function(n) {
  if (n == 1) {
    return(1)  # Base case
  } else {
    return(n * factorial(n - 1))  # Recursive case
  }
}
```

Recursion Applications

Recursion can be useful for problems like:

- Exponentiation

```r
power <- function(base, exp) {
  if (exp == 0) {
    return(1)  # Base case
  } else {
    return(base * power(base, exp - 1))  # Recursive call
  }
}

power(2, 3)  # Output: 8
```

Recursion Applications

- Fibonacci sequences

```r
fibonacci <- function(n) {
  if (n == 0) {
    return(1)  # Base case
  } else if (n == 1) {
    return(1)  # Base case
  } else {
    return(fibonacci(n - 1) + fibonacci(n - 2))  # Recursive call
  }
}

fibonacci(6)  # Output: 8
```

Vectors

A vector contains a list of items with the same type separated by a comma. We use c() keyword to declare vectors.

```
# Example One
# Vector of strings
fruits <- c("banana", "apple", "orange")
# Print fruits
fruits

# Example Two
values = c(TRUE,FALSE,FALSE)
# Print values
values
```

Access and Modify Vectors

- We can access a vector item by its index number inside brackets. Indexes start from 1,2,…,,n.

```
fruits <- c("banana", "apple", "mango")

# Accessing the first item
fruits[1]
```

- We also can change a vector item by its index.

```
fruits <- c("banana", "apple", "orange")

# Change the first item (banana)
fruits[1] = "Mango"
fruits
```

Sort Vectors

- We can use sort() function to sort items alphabetically or numerically.

```
fruits <- c("banana", "apple")
numbers <- c(2, 3, 51, 1, 20, 4)

sort(fruits)  # Sort a string
sort(numbers) # Sort numbers
```

Lists

- Lists in R are like vectors but we can to store many different data types in lists.

```
 # List of strings
thislist <- list("apple", "banana", "cherry",2,TRUE)

# Print the list
thislist
```

List Functions

- Access a specific item and change it

```
ls1 <- list("apple", "banana", "cherry")
ls1[1]
ls1[1] <- "blackcurrant"
ls1[1]
# Print the updated list
ls1
```

- Add item to list

```
ls1 <- list("apple", "banana", "cherry")
ls1
ls1 = append(ls1,'mango')
# Print the updated list
ls1
```

List Functions

- Check if item exist in list

```
ls1 <- list("apple", "banana", "cherry","mango")
if ("mango" %in% ls1){
  print('There is mango')
}
```

- Remove from list with index or value

```
ls1 <- list("apple", "banana", "cherry","mango")
# Delete with value
ls2 <- ls1[ls1 != "mango"]
# Delete with index
ls3 <- ls1[-4]
# Print modified lists
ls2
ls3
}
```

List Functions

- Loop through a list

```
ls1 = list("BMW","Mercedes","Hyundai","Lexus")
for(car in ls1){
  print(paste('Car:',car))
}
```

- Join lists

```
ls1 = list("BMW","Mercedes","Hyundai","Lexus")
ls2 = list("Ninja","Honda")
ls3 = c(ls1,ls2)
# Print Joined list
ls3
```

Matrices

Matrix is a two dimensional data with rows and columns. We can use matrix() keyword to create a matrix and specify number of rows and columns with nrow and ncol.

```
m1 = matrix(c(1,2,3,4,5,6,7,8,9),nrow = 3,ncol = 3,byrow = TRUE)
# Printing the matrix
m1
```

- What is “byrow=TRUE”?

Matrix Functions

- Access and change values

```
m1 = matrix(c(1,2,3,4,5,6,7,8,9),nrow = 3,ncol = 3,byrow = TRUE)
# Changing row 1, col 2 index
m1[1,2] = 10
m1
```

- Add rows and columns

```
m1 = matrix(c(1,2,3,4),nrow = 2,ncol = 2,byrow = TRUE)
# Add a column
m2 <- cbind(m1, c("cbind1", "cbind2"))
# Add a row
m3 <- rbind(m1,c("rbind1","rbind2"))
# Printing new matrixes
m2
m3
```

Matrix Functions

- Remove rows and columns

```
m1 = matrix(c(1,2,3,4,5,6),nrow = 2,ncol = 3,byrow = TRUE)
m2 <- m1[-c(1),] # Remove first row
m3 <- m1[,-c(1)] # Remove first column
# Print modified matrices
m2
m3
```

- Loop through a matrix

```
m1 = matrix(c(1,2,3,4,5,6),nrow = 2,ncol = 3,byrow = TRUE)
for(row in 1:nrow(m1)){
  for(col in 1:ncol(m1)){
    print(m1[row,col])
  }
}
```

Arrays

Unlike matrices, arrays can have more than 2 dimensions in R. We use array() function to declare an array and dim keyword to determine array dimensions. Remember that array can only store data with same data types.

```
# An array with more than one dimension created from a vector at first
vector1 = c(1:36)
arr1 <- array(vector1, dim = c(4, 3, 3))
arr1
```

- How can we access and modify array items?

DataFrames

- Data frames are data displayed in a format like table. We can store data with different data types in a data frame but each column in data frame can contain data with same data type. We use data.frame() keyword to declare a data frame.

```
# Create a data frame
df1 <- data.frame (
 ID = c(1 , 2, 3),
 Name = c("Alex", "John", "Alice"),
 Age = c(60, 30, 45)
)

# Print the data frame
df1
```

- What does summary() keyword do?

DataFrame Functions

- Access and modify values

```
# Create a data frame
df1 <- data.frame (
  ID = c(1, 2, 3),
  Name = c("Alex", "John", "Alice"),
  Age = c(60, 30, 45)
)
# Access to values
df1[2]
df1$Name
# Modify a value
df1$Name[2] = "Richard"
df1
```

Other functions like add and remove rows and columns, combine data frames and etc.. are like arrays function.

Plotting

We can use plot() function to draw points in a diagram. This function has two main arguments, x-axis and y-axis. For example, you can plot two numbers (1,4) and (2,9) with code below:

```
plot(c(1, 2), c(4, 9))
```

- Plot a simple sequence

```
plot(1:10)
```

- Drawing a line

```
plot(1:10, type="l")
```

- Labeling the plot

```
plot(1:10, main="Graph 1", xlab="X-axis", ylab="Y-axis")
```

Lets solve some questions!

We can solve this questions just with R basics:

- Plotting a Parametric Circle
  $x(t) = cos(t), y(t) = sin(t), for\ t \in [0,2\pi]$
- Approximate Arc Length of a Curve
  curve: $x(t) = 3t, y(t) = 2\sqrt{(}t), for\ t \in [1,4]$
  hint: we can use integrate() function for calculating integrals.

Previous Slide Solutions in R

1. First Question Answer:

```r
t <- seq(0, 2 * pi, length.out = 100)
x <- cos(t)
y <- sin(t)
plot(x, y, type = "l", col = "blue", xlab = "x", ylab = "y", main = "Parametric Circle")
```

Previous Slide Solutions in R

1. Second Question Solution:

```
# Define functions x(t) and y(t)
x <- function(t) { 3 * t }
y <- function(t) { 2 * sqrt(t) }

# Define the derivatives dx/dt and dy/dt
dx <- function(t) { 3 }
dy <- function(t) { 1 / sqrt(t) }

# Define the integrand for arc length
integrand <- function(t) {
  sqrt((dx(t))^2 + (dy(t))^2)
}

# Calculate the arc length from t = 1 to t = 4
arc_length <- integrate(integrand, lower = 1, upper = 4)$value
cat("Arc Length:", arc_length, "\n")
```

Basic Statistics in R

R provides straightforward functions to compute basic statistics on numerical data, making it ideal for quick data summaries and analyses.

- Maximum and Minimum
- Mean (Average)
- Median
- Percentiles and Quantiles
- Summary Statistics

Basic Statistics in R

- Maximum and Minimum
  We have functions to find the largest and smallest values of dataset.

```
data <- c(5, 12, 3, 19, 8)
max_value <- max(data)  # 19
min_value <- min(data)  # 3
```

- Mean (Average)
  Mean function calculate the average of dataset.

```
mean_value <- mean(data)
```

Basic Statistics in R

- Median
  We have functions to find median value of a dataset. (Middle value of a dataset when ordered)

  ```
  mean_value <- median(data)
  ```

- Percentiles and Quantiles
  There is function to find values at specific percentiles within a dataset. This can help us to achieve a better understanding of data distribution.

  ```
  quantiles <- quantile(data, probs = c(0.25, 0.5, 0.75))
  ```

- Summary Statistics
  Provides a summary of data, including Min, 1st Quartile, Median, Mean, 3rd Quartile, and Max.

  ```
  summary(data)
  # Output includes Min, 1st Qu., Median, Mean, 3rd Qu., Max
  ```

Detect Outliers Using Quantiles

Outliers can skew data analysis results, so detecting and potentially removing them can improve the accuracy of your analyses. Quantiles are commonly used to identify these outliers.

Steps to Detect Outliers Using Quantiles:

1. **Calculate Quantiles**: Identify the first (Q1) and third (Q3) quartiles.
2. **Compute the Interquartile Range (IQR)**: Calculate the range within which most of the data falls.
   IQR = Q3 - Q1
3. **Define Outliers**: Values below $Q1 - 1.5 * IQR$ or above $Q3 + 1.5 * IQR$ are often considered outliers.

Detect Outliers Using Quantiles

```r
# Sample data
data <- c(5, 12, 3, 19, 8, 25, 35, 30, 7, 14)

# Calculate Q1, Q3, and IQR
Q1 <- quantile(data, 0.25)
Q3 <- quantile(data, 0.75)
IQR <- Q3 - Q1

# Define outlier boundaries
lower_bound <- Q1 - 1.5 * IQR
upper_bound <- Q3 + 1.5 * IQR

# Detect and remove outliers
outliers <- data[data < lower_bound | data > upper_bound]
cleaned_data <- data[data >= lower_bound & data <= upper_bound]

# Output results
outliers  # Values considered outliers
cleaned_data  # Data with outliers removed
```

Installing and Loading R Packages

- We can install R packages with below command:
  `install.packages("package_name")`

- For Adding a package to R project environment:
  `library(package_name)`

- Some common R Packages:
  dplyr: Data manipulation and transformation
  ggplot2: Excellent package for data visualization
  tidyr: easily reshape and tidy data for analysis
  lubridate: easily work with date and time
  shiny: build interactive web apps

Ggplot2: Data Visualization in R

- Ggplot2 is an excellent library for creating a wide range of visualization. This package allows us to layering, customization and detailed control over plots.
- Simple code for ggplot2

```r
# Load ggplot2 package
library(ggplot2)

# Sample Data
data <- data.frame(x = 1:10, y = c(3, 7, 8, 5, 6, 9, 12, 11, 15, 18))

# Scatter Plot
ggplot(data, aes(x = x, y = y)) +
geom_point(color = "blue") +
labs(title = "Simple Scatter Plot", x = "X-axis", y = "Y-axis") +
theme_minimal()
```

Ggplot2 Options

- Data: The data argument specifies the dataset for the plot.
- Aes: This option defines aesthetic mappings, connecting data to visual properties like x, y, color, size, and shape.
- Geom: This option defines the type of plot you create, like geom_point (scatter), geom_line (line), or geom_bar (bar plot).
- Labs: The labs() function customizes labels for titles, axis labels, and legends.
- Theme: Theme functions are used to control the appearance of non-data elements, like backgrounds, grids, and text formatting. (theme_light() , theme_dark(), theme_void())

Plotting functions

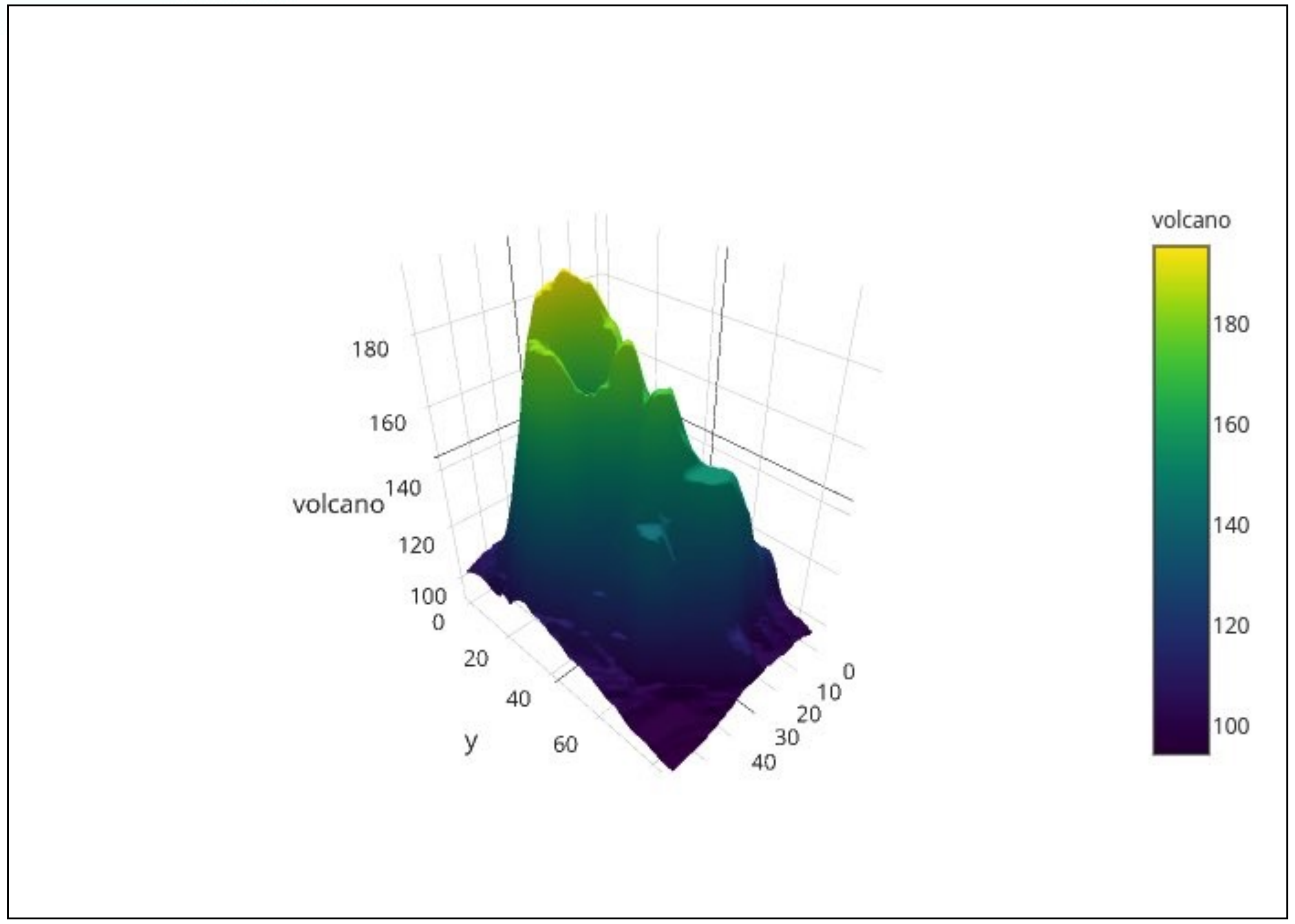
- Lets review a simple example of how to plot a mathematical function:

```
# Load necessary libraries
 library(ggplot2)

 # Define the functions
 f1 <- function(x) { return(x^3) }
 f2 <- function(x) { return(abs(x)) }

 # Create a data frame with a sequence of x values
 x_values <- seq(-2, 2, length.out = 100)  # More points for smooth curves

 # Create the ggplot
 ggplot(data.frame(x = x_values), aes(x = x)) +
   stat_function(fun = f1 , aes(color = "x^3")) +
   stat_function(fun = f2, aes(color = "abs")) +
   labs(
     title = "Plot of Functions f(x) = x^3 and g(x) = abs(x)",
     x = "x",
     y = "f(x) and g(x)"
   ) +
   scale_color_manual(
     name = "Functions",
     values = c("x^3" = "black", "abs" = "green")  # Corrected color mapping
   ) +
   theme_minimal()
```

Whats Plotly?

- Plotly is a package for R that enables creation of high-quality, interactive visualizations. Some key features include:
    1. Interactive Plots
    2. 3D Plotting
    3. Wide Range of Charts
    4. Customization Options

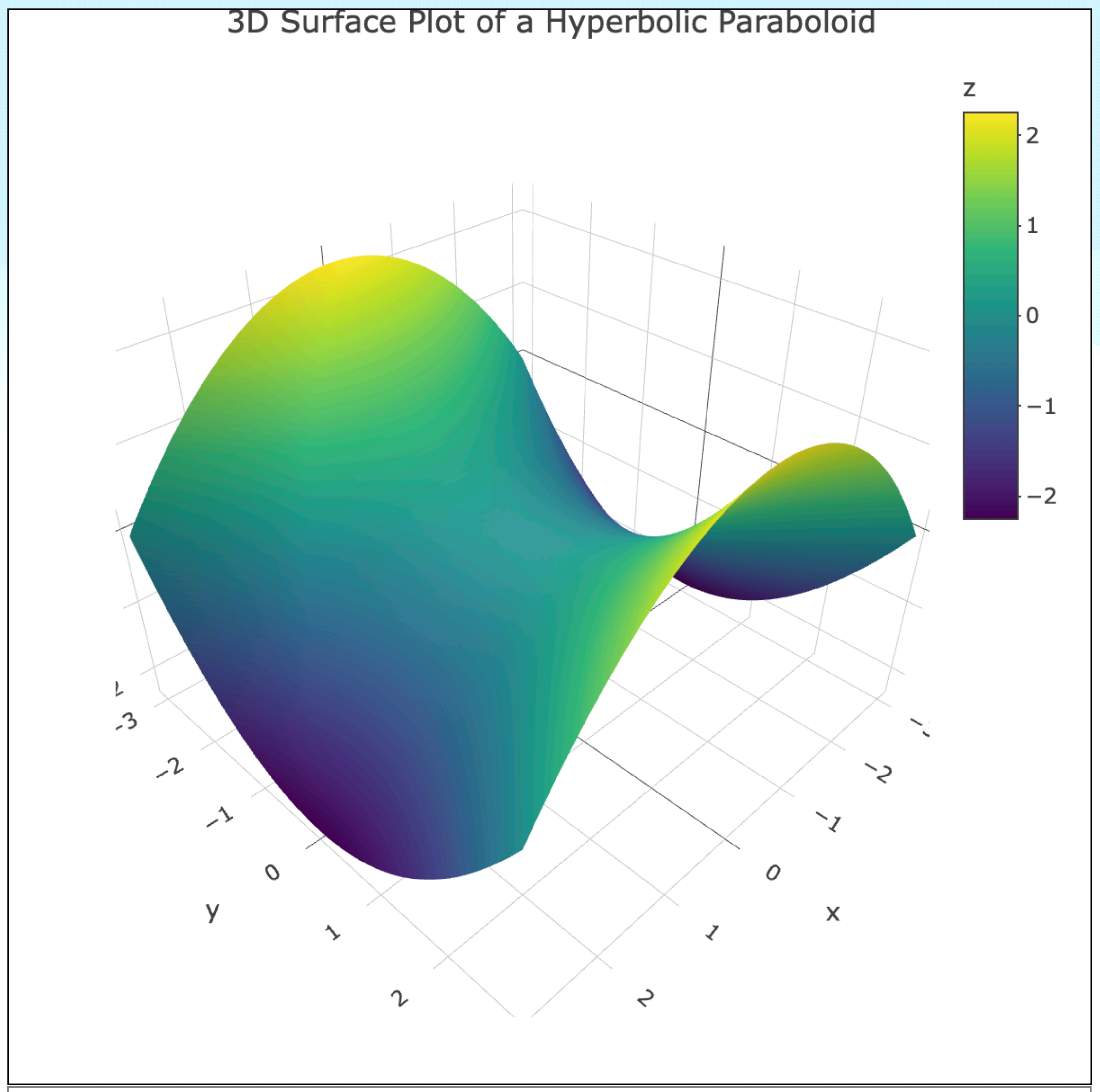
- What is outer() function? The outer function in R is a powerful tool used to compute the outer product of two arrays and can also be utilized for applying a specified function across all combinations of the elements of the input arrays.



3D Plotting

The equation $z = \frac{x}{4} - \frac{y}{9}$ represents a hyperbolic paraboloid.

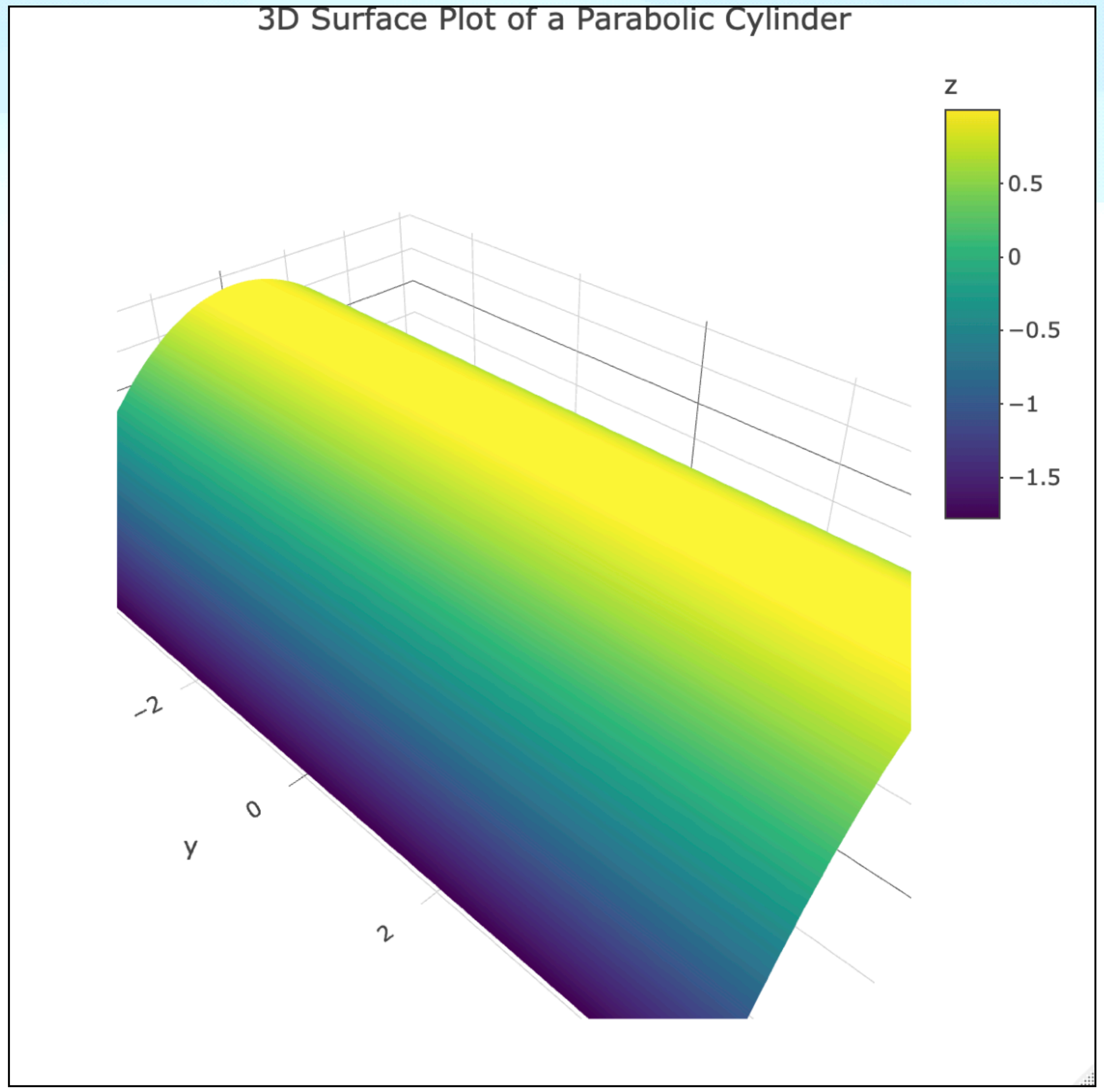
```
# Define x and y ranges
x <- seq(-3, 3, length.out = 100)
y <- seq(-3, 3, length.out = 100)

# Calculate z values for hyperbolic paraboloid
z <- outer(x, y, function(x, y) (x^2 / 4 - y^2 / 9))

# Create plotly surface plot
plot_ly(x = ~x, y = ~y, z = ~z, type = "surface") %>%
  layout(
    title = "3D Surface Plot of a Hyperbolic Paraboloid",
    scene = list(
      xaxis = list(title = "x"),
      yaxis = list(title = "y"),
      zaxis = list(title = "z")
    )
  )
```



3D Plotting

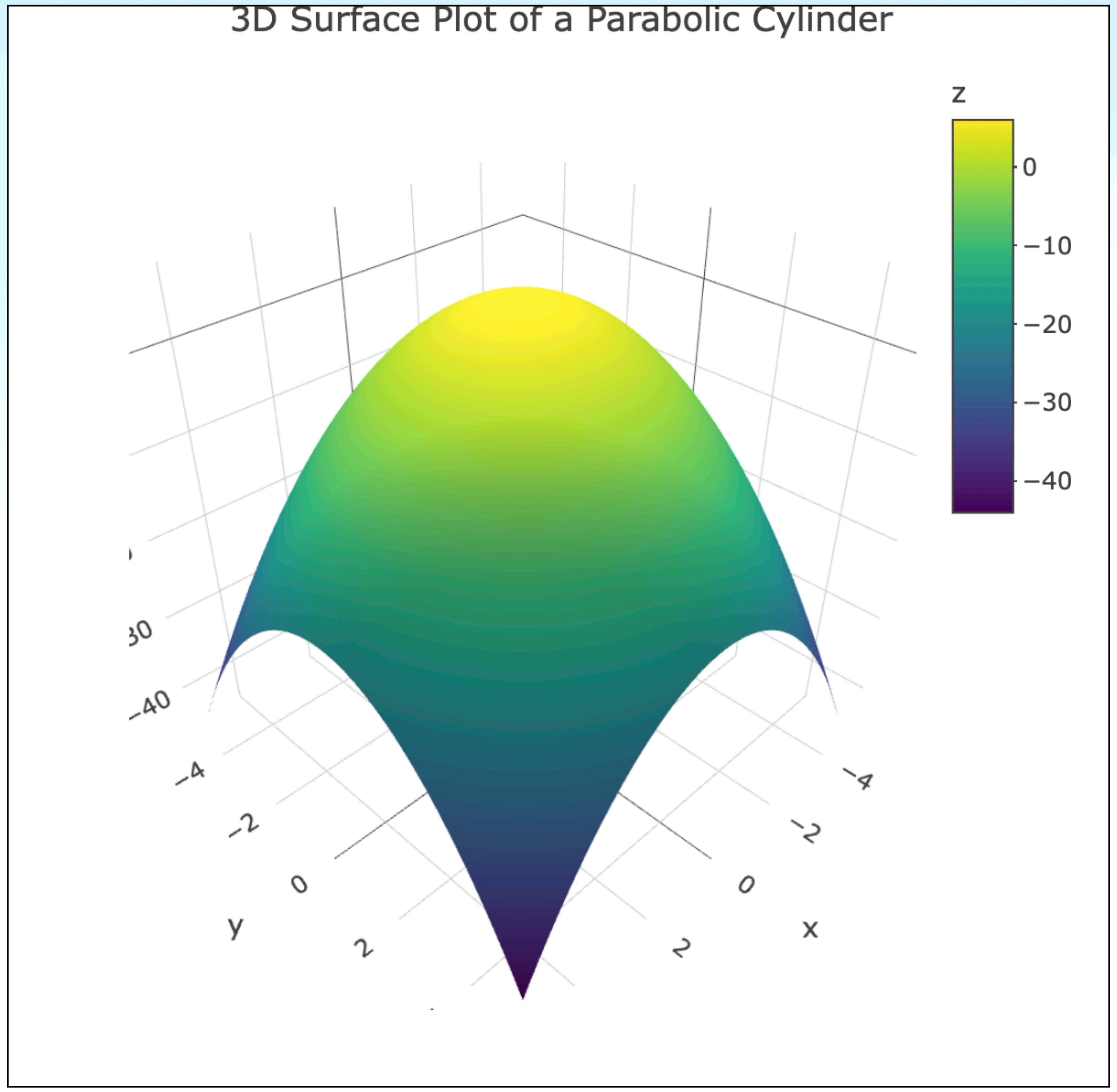
The equation $1 = \frac{y^2}{9} + z$ represents a parabolic cylinder. With a little rearranging we have $z = 1 - \frac{y^2}{9}$.

```r
y <- seq(-5, 5, length.out = 100)
x <- seq(-3, 3, length.out = 100)

# Calculate z values for the surface
z <- outer(x, y, function(x, y) 1 - (y^2 / 9))

# Create the plotly surface plot
plot_ly(x = ~x, y = ~y, z = ~z, type = "surface") %>%
  layout(
    title = "3D Surface Plot of a Parabolic Cylinder",
    scene = list(
      xaxis = list(title = "x"),
      yaxis = list(title = "y"),
      zaxis = list(title = "z")
    )
  )
```



Practical Challenge

Write an R code that plot $z = -x^2 - y^2 + 6$. Final result should be something like below image:



Previous Slide Answer

```r
# Install plotly if not installed
if (!require(plotly)) install.packages("plotly")
library(plotly)

# Define x and y ranges
x <- seq(-3, 3, length.out = 100)
y <- seq(-3, 3, length.out = 100)

# Calculate z values for the surface
z <- outer(x, y, function(x, y) -x^2 - y^2 + 6)

# Create the 3D surface plot
plot_ly(x = ~x, y = ~y, z = ~z, type = "surface") %>%
  layout(
    title = "3D Surface Plot of z = -x^2 - y^2 + 6",
    scene = list(
      xaxis = list(title = "x"),
      yaxis = list(title = "y"),
      zaxis = list(title = "z")
    )
  )
```

Understanding Sequences

- A sequence is an ordered list of numbers following a specific pattern.
  Example:
  Arithmetic sequence: $a_n = a_1 + (n-1)d$
  Geometric sequence: $a_n = a \,.\, r^{(n-1)}$

```r
# Arithmetic sequence
a <- 2  # First Number
 d <- 3  # Common difference
 n <- 10 # Number of terms
 seq <- a + (0:(n-1)) * d
 print(seq)

 # Geometric sequence
 a <- 2  # First term
 r <- 3  # Common ratio
 n <- 10 # Number of terms
 seq <- a * r^(0:(n-1))
 print(seq)
```

Series

- A series is the summation of the terms of a sequence.
  Example:

```
# Arithmetic series
a <- 2  # First term
d <- 3  # Common difference
n <- 10 # Number of terms
seq <- a + (0:(n-1)) * d
sum(seq)

# Geometric series
a <- 2  # First term
r <- 0.5  # Common ratio
n <- 10 # Number of terms
seq <- a * r^(0:(n-1))
sum(seq)
```

Convergence

- A series converges if the sequence of its partial sums approaches a finite value.
- Divergence: A series diverges if the partial sums grow without bound.

```
a <- 1  # First term
r <- 0.5  # Common ratio
n <- 50  # Number of terms

seq <- a * r^(0:(n-1))
partial_sums <- cumsum(seq)

plot(partial_sums, type = "b", main = "Partial Sums of Geometric Series")
```

Ratio Test

- For a series $\sum a_n$ the ratio test states:

  if $\lim_{n\to\infty} \frac{a_{n+1}}{a_n} < 1$, the series converges. Else, the series diverges.

```
a_n <- function(n) { 1 / factorial(n) }  # Example: Exponential series
ratio <- function(n) { a_n(n+1) / a_n(n) }
ratio_values <- sapply(1:10, ratio)
print(ratio_values)
```

Power Series

- A power series is of the form:

$$f(x) = \sum_{n=0}^{\infty} c_n (x-a)^n$$

Example: $f(x) = 1 - x + x^2 - x^3 + x^4 \ldots$

```
power_series <- function(x, n, coeffs) {
  # Ensure coeffs has at least n+1 elements
  if (length(coeffs) < n + 1) {
    stop("The length of coeffs must be at least n+1")
  }

  # Calculate the power series
  series_value <- 0

  for (i in 0:n) {
    series_value <- series_value + coeffs[i + 1] * x^i
  }
  return(series_value)
}

# Example usage
x <- 2  # The value at which to evaluate the series
n <- 5  # Degree of the series
coeffs <- c(1, -1, 1, -1, 1, -1)  # Coefficients for each power of x (a_0 to a_5)

result <- power_series(x, n, coeffs)
print(paste("Power series value at x =", n, "is:", result))
```

Partial Derivative

- Partial derivatives measure how a function changes as one variable keeping others constant.
- There are several ways to calculate partial derivatives:
  - Central difference method
  - Numerical methods (packages like numDeriv for gradients)
  - Symbolic methods (packages like derive to compute exact formulas)

Finite Difference Method

- Mathematical formula for finite difference (central difference) is:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Example

```
# Define the function
f <- function(x, y) x^2 + y^3

# Compute partial derivative with respect to x
partial_x <- function(x, y, h = 1e-5) {
  (f(x + h, y) - f(x - h, y)) / (2 * h)
}

partial_x(1, 2) # Output: approximately 2
```

Numerical Derivatives

- The numDeriv package is one of most commonly used packages designed for numerical differentiation. We can use grad() function to compute gradient or partial derivatives.

- Example:

```
library(numDeriv)

# Define the function
f <- function(x) x[1]^2 + x[2]^3

# Compute partial derivatives with respect to x1 and x2
grad(f, c(1, 2))
# Output: [1] 2 12
```

Symbolic Derivatives

- With Deriv package we can symbolically compute derivatives and return exact formulas.
- Example:

```
library(Deriv)

# Define the function
f <- function(x, y) x^2 + y^3 + x

# Compute partial derivatives
df_dx <- Deriv(f, "x")
df_dy <- Deriv(f, "y")

# Results
df_dx # Output: function(x, y) 2 * x
df_dy # Output: function(x, y) 3 * y^2
```

Practical Challenge

- Consider below function:

$$f(x, y) = 2^{x^2+y^2} \sin(xy)$$

- Compute $\frac{\partial f}{\partial x}$ with one of previous slide methods.

Previous Slide Answer

```
library(Deriv)

# Define the function
f <- function(x, y) 2^((x^2)+(y^2))*sin(x*y)

# Compute partial derivatives
df_dx <- Deriv(f, "x")

# Results
df_dx
```

-

Numerical Integration

- Numerical integration approximates the value of definite integrals when an analytical solution is difficult or impossible to compute.
- In R, the integrate() function is a powerful tool for performing numerical integration with high precision.
- Example:

```
 f <- function(x) exp(-x^2)
result <- integrate(f, lower = 0, upper = 1)
print(result$value)
```

•The integrate() function provides both the integral value and an error estimate.

What is Linear Regression?

- Linear regression is a statistical method used to model and analyze the relationship between one or more independent variables (predictors) and a dependent variable (outcome). It predicts a continuous outcome.

- **Why Use Linear Regression?**

  1. To quantify relationships between variables.
  2. To predict outcomes based on input data.
  3. To identify important predictors affecting the outcome.

- **Types of Linear Regression**

  1. **Simple Linear Regression**: One independent variable.
  2. **Multiple Linear Regression**: Two or more independent variables.

Correlation Coefficient

- Measures the strength and direction of the linear relationship between two variables.
- Values range from -1 to 1:
  1. $r > 0$: Positive correlation.
  2. $r < 0$: Negative correlation
  3. $r = 0$: No correlation.

- Formula:

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \cdot \sum (y_i - \bar{y})^2}}$$

Assumptions of Linear Regression

1. **Linearity**: Relationship between predictors and outcome is linear.
2. **Independence**: Observations are independent of each other.
3. **Homoscedasticity**: Equal variance of residuals across all levels of the predictors.
4. **Normality**: Residuals (errors) are normally distributed.
5. **No Multicollinearity**: Predictors should not be highly correlated with each other.

How to Check Assumptions?

Running Linear Regression on a Dataset

- **Dataset Overview**

  Predictor (x): Hours of study.

  Outcome (y): Exam score.

```r
# Load data
data <- data.frame(hours = c(1, 2, 3, 4, 5), score = c(50, 55, 60, 65, 70))

# Fit linear model
model <- lm(score ~ hours, data = data)

# Summary of the model
summary(model)

# Plot
plot(data$hours, data$score, main = "Linear Regression", xlab = "Hours of Study", ylab = "Exam Score", pch = 19)
abline(model, col = "blue", lwd = 2)
```

Predicting with Linear Regression

- Now we can predict new exam points based on study hours with our linear regression model.

```
# Create a data frame with new input values
new_data <- data.frame(hours = c(4.0, 4.5, 5.0))

# Predict using the linear regression model
predicted_scores <- predict(lm.r, newdata = new_data)

# Display the predicted salaries
print(predicted_scores)
```

Checking Assumptions of Linear Regression

- **Assumption 1: Linearity**

  The relationship between the independent variable (Hours) and dependent variable (Score) should be linear.

  # Scatterplot with regression line

```r
plot(data$hours, data$score, main = "Linearity Check", xlab = "Hours of Study", ylab = "Exam Score",
pch = 19)
abline(model, col = "blue", lwd = 2)
```

Checking Assumptions of Linear Regression

- **Assumption 2: Homoscedasticity**

  The variance of residuals should be constant across all levels of the predictors.

```r
# Residual plot
plot(model$fitted.values, model$residuals,
    main = "Homoscedasticity Check",
    xlab = "Fitted Values",
    ylab = "Residuals",
    pch = 19)
abline(h = 0, col = "red", lwd = 2)
```

Checking Assumptions of Linear Regression

- **Assumption 3: Normality of Residuals**

  Residuals should follow a normal distribution.

# Histogram of residuals

```
hist(model$residuals, main = "Normality of Residuals", xlab = "Residuals", breaks = 10, col =
"lightblue")

# Q-Q plot
qqnorm(model$residuals, main = "Q-Q Plot of Residuals")
qqline(model$residuals, col = "red", lwd = 2)
```

Checking Assumptions of Linear Regression

- **Assumption 4: Independence**

  Observations should be independent of each other.

  # Durbin-Watson test for independence

```
install.packages("lmtest")
library(lmtest)
dwtest(model)
```

- Interpretation
  - p-value ≤ 0.05: Residuals are not independent.
  - p-value > 0.05: Residuals are independent.

Checking Assumptions of Linear Regression

- **Assumption 5: No Multicollinearity**

  Predictors should not be highly correlated with each other.

  # Check Variance Inflation Factor (VIF)

```
install.packages("car")
vif(model)
```

- **Interpretation**
  - VIF ≈ 1: No multicollinearity.
  - VIF > 5: High multicollinearity, consider removing variables.

In this example, we have just one predictor.

Do the assumptions of linear regression hold true for every dataset?

The Reality

- The 5 assumptions of linear regression are not always true for every dataset.
- It is crucial to validate these assumptions to ensure the reliability of the regression results.

1. Linearity

   The relationship between predictors and response may not be linear. Example: Polynomial or exponential trends.

2. Independence

   Observations might be dependent in time series or hierarchical datasets.

3. Homoscedasticity

   Variance of residuals might increase/decrease with the predicted values, leading to heteroscedasticity.

4. Normality of Errors

   Residuals may not follow a normal distribution, especially with small sample sizes or outliers.

5. No Multicollinearity

   Predictors might be highly correlated, violating this assumption.

The Reality

- **Biased coefficients**: Predictions might not represent the true relationship.
- **Incorrect significance tests**: p-values and confidence intervals may be invalid.
- **Suboptimal predictions**: Predictions could lack reliability.

- How to address this issues?
  1. Transform variables (e.g., log, square root, or polynomial terms).
  2. Use **robust regression** or other models like Ridge/Lasso.
  3. Check multicollinearity and remove/reduce redundant predictors.
  4. Etc…