R Programming Language
Lab Programs

1. **Write an R program illustrated with an if-else statement and how it operates on vectors of variable length.**

Program :

```
# Define two vectors of different lengths
x <- c(1, -2, 3, -4, 5)  # A vector of length 5
y <- c(-3, 0, 2)         # A vector of length 3
# Using `ifelse()` to check if each element is positive or not for vector `x`
result_x <- ifelse(x > 0, "positive", "non-positive")
# Using `ifelse()` to check if each element is positive or not for vector `y`
result_y <- ifelse(y > 0, "positive", "non-positive")
# Print the results for each vector
print("Results for vector x:")
print(result_x)
print("Results for vector y:")
print(result_y)
```

Output :

```
[1] "Results for vector x:"
[1] "positive"     "non-positive" "positive"     "non-positive" "positive"

[1] "Results for vector y:"
[1] "non-positive" "non-positive" "positive"
```

Explanation:

- **Using `ifelse()`:**
  - The `ifelse()` function is used here to apply the condition `x > 0` or `y > 0` element-wise across the vectors.
  - For each element of `x` and `y`, `ifelse()` checks if the element is greater than 0.
  - If the condition is true, it assigns "positive" to that position; otherwise, it assigns "non-positive".

Key Takeaways:

- `ifelse()` is used to apply conditions element-wise over vectors, making it the proper way to handle conditions on vectors without producing warnings or errors.
- Unlike the standard `if` statement, `ifelse()` can directly operate on vectors of any length, returning a new vector of the same length with the results of the condition.

2. **Write an R program illustrate with for loop and stop on condition, to print the error message.**

Program :

```r
# Define a vector of numbers
numbers <- c(2, 4, -3, 7, 5)

# Iterate over the vector using a for loop
for (num in numbers) {
  # Check if the number is negative
  if (num < 0) {
    # Print an error message and stop the execution
    stop(paste("Error: Negative number encountered:", num))
  }

  # Print the number if it is non-negative
  print(paste("Processing number:", num))
}

# This part will not be reached if a negative number is encountered.
print("All numbers processed successfully.")
```

Output :

```
[1] "Processing number: 2"
[1] "Processing number: 4"
Error in eval(expr, envir, enclos): Error: Negative number encountered: -3
[1] "All numbers processed successfully."
```

Explanation:

- **Define a Vector**: A vector `numbers` is defined with some positive and negative values.
- **For Loop**: The `for` loop iterates over each element of the `numbers` vector.
- **Condition Check**: Inside the loop, the `if` statement checks if the current number (`num`) is negative.
- **Using `stop()`**: If a negative number is encountered, the `stop()` function is called with a custom error message. This immediately stops the execution of the loop and the program, printing the error message.
- **Processing Non-negative Numbers**: If a number is non-negative, it prints a message indicating that the number is being processed.
- **Stopping the Loop**: As soon as a negative number is encountered, the loop stops, and any code after `stop()` inside the loop will not be executed.

3. **Write an R program to find the factorial of a given number using recursion.**

Program :

```
# Define a recursive function to calculate factorial
factorial_recursive <- function(n) {
  # Base case: if n is 0 or 1, return 1
  if (n == 0 || n == 1) {
    return(1)
  } else if (n < 0) {
    # Handle negative input by returning an error message
    stop("Error: Factorial is not defined for negative numbers")
  } else {
    # Recursive case: n * factorial of (n - 1)
    return(n * factorial_recursive(n - 1))
  }
}

# Test the function with an example
number <- 5
result <- factorial_recursive(number)

# Print the result
print(paste("The factorial of", number, "is:", result))
```

Output :

```
[1] "The factorial of 5 is: 120"
```

Explanation:

- **Recursive Function**: The function `factorial_recursive()` takes a single argument `n` and calculates the factorial recursively.
- **Base Case**: When `n` is `0` or `1`, it returns `1` because the factorial of `0` and `1` is `1`.
- **Error Handling**: If `n` is negative, it uses the `stop()` function to print an error message, as the factorial is not defined for negative numbers.
- **Recursive Case**: For positive numbers greater than 1, it multiplies `n` by the result of `factorial_recursive(n - 1)`, effectively reducing the problem size with each recursive call.
- **Test the Function**: The function is tested with `number <- 5`, and the result is printed.

How the Recursion Works:

For `number = 5`, the function calls itself recursively as follows:

- `factorial_recursive(5)` calls `5 * factorial_recursive(4)`
- `factorial_recursive(4)` calls `4 * factorial_recursive(3)`
- `factorial_recursive(3)` calls `3 * factorial_recursive(2)`
- `factorial_recursive(2)` calls `2 * factorial_recursive(1)`
- `factorial_recursive(1)` returns `1` (base case)

The results are then multiplied as follows:

5 * 4 * 3 * 2 * 1 = 120

4. **Write an R program to implement T-Test**

A **T-Test** is a statistical test used to compare the means of two groups to determine if they are significantly different from each other. In R, you can use the built-in `t.test()` function to perform various types of T-tests, including:

- One-sample T-test
- Two-sample T-test (independent samples)
- Paired T-test

Program :

```
# Example data for two groups (independent samples)

group1 <- c(5.1, 5.5, 5.8, 6.0, 5.9)

group2 <- c(6.2, 6.4, 6.1, 6.3, 6.0)


# 1. Two-sample T-test (independent samples)

# Testing if the means of group1 and group2 are different

t_test_independent <- t.test(group1, group2, alternative = "two.sided")

print("Two-sample T-test (independent samples):")

print(t_test_independent)
```

Output :

```
[1] "Two-sample T-test (independent samples):"

  Welch Two Sample t-test


data:  group1 and group2

t = -3.0377, df = 5.4524, p-value = 0.0258

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

 -0.98578388 -0.09421612

sample estimates:

mean of x mean of y

     5.66      6.20
```

```
# 2. Paired T-test

# Example data for paired samples (e.g., pre-test and post-test)

pre_test <- c(78, 85, 90, 88, 93)

post_test <- c(80, 87, 89, 91, 94)

# Testing if there is a significant difference between pre-test and post-test scores

t_test_paired <- t.test(pre_test, post_test, paired = TRUE)

print("Paired T-test:")

print(t_test_paired)
```

Output :

```
[1] "Paired T-test:"

Paired t-test

data:  pre_test and post_test

t = -2.0642, df = 4, p-value = 0.1079

alternative hypothesis: true mean difference is not equal to 0

95 percent confidence interval:

 -3.2830767  0.4830767

sample estimates:

mean difference

           -1.4
```

```
# 3. One-sample T-test

# Testing if the mean of group1 is significantly different from a hypothesized mean (e.g., 5.8)

t_test_one_sample <- t.test(group1, mu = 5.8, alternative = "two.sided")

print("One-sample T-test:")

print(t_test_one_sample)
```

Output :

```
[1] "One-sample T-test:"

 One Sample t-test

data:  group1
t = -0.8584, df = 4, p-value = 0.4391
alternative hypothesis: true mean is not equal to 5.8
95 percent confidence interval:
 5.207176 6.112824
sample estimates:
mean of x
     5.66
```

Explanation:

1. **Two-sample T-test (Independent Samples)**:
   - This test compares the means of two independent groups (`group1` and `group2`).
   - The `t.test()` function is called with `t.test(group1, group2)`.
   - `alternative = "two.sided"` indicates that we are testing if the means are different in either direction (greater or lesser).
   - The output will include the T-statistic, degrees of freedom, p-value, confidence interval, and means of the two groups.
   - Use this test when you have two separate groups and want to know if they have different means.
2. **Paired T-test**:
   - This test compares two sets of observations where each observation in one group is paired with a corresponding observation in the other group (e.g., pre-test and post-test scores).
   - The `t.test()` function is called with `paired = TRUE`.
   - This is useful when comparing measurements from the same subjects before and after a treatment.
3. **One-sample T-test**:
   - This test compares the mean of a single group (`group1`) to a hypothesized mean (`mu = 5.8`).
   - It checks whether the average value of `group1` is significantly different from `5.8`.
   - The `t.test()` function is called with `mu = 5.8`.

Output:

The output will include the results of each T-test, including the T-statistic, degrees of freedom, p-value, confidence interval, and mean of the data. The p-value helps determine if the results are statistically significant:

- **p-value < 0.05**: Reject the null hypothesis (significant difference).
- **p-value ≥ 0.05**: Fail to reject the null hypothesis (no significant difference).

The R program provides a comprehensive way to conduct T-tests depending on the nature of your data and the research question you aim to answer.

5. Write an R program to implement ANOVA.

ANOVA (Analysis of Variance) is used to compare means across multiple groups and determine if at least one group mean is different from the others. In R, you can perform ANOVA using the `aov()` function. Here's a program that demonstrates how to perform a one-way ANOVA, which is used when you have one independent variable with multiple groups.

Program: Implementing One-Way ANOVA in R

```r
# Example data for three groups

group1 <- c(5.1, 5.5, 5.8, 6.0, 5.9)

group2 <- c(6.2, 6.4, 6.1, 6.3, 6.0)

group3 <- c(5.7, 5.9, 5.8, 6.1, 5.6)


# Combine the data into a single data frame

data <- data.frame(

  values = c(group1, group2, group3),

  group = factor(rep(c("Group1", "Group2", "Group3"), each = 5))

)


# Perform one-way ANOVA

anova_result <- aov(values ~ group, data = data)


# Display the summary of the ANOVA results

print("ANOVA Summary:")

summary(anova_result)
```

Output :

```
          Df Sum Sq Mean Sq F value Pr(>F)

group       2  0.84  0.4200   4.5    0.03 *

Residuals   12  1.12  0.0933

---

Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Note :

- **Df**: Degrees of freedom.
- **Sum Sq**: Sum of squares between and within the groups.
- **Mean Sq**: Mean sum of squares.
- **F value**: The test statistic used in ANOVA.
- **Pr(>F)**: The p-value associated with the F statistic.

Interpreting the Results:

- If the p-value (`Pr(>F)`) is less than 0.05, there is a statistically significant difference between the means of the groups, and we reject the null hypothesis (which states that all group means are equal).
- If the p-value is greater than or equal to 0.05, there is no significant difference between the means of the groups, and we fail to reject the null hypothesis.

In the example output, if the p-value is `0.03`, it means that at least one group mean is significantly different from the others.

Explanation:

6. **Data Preparation**:
    a. Define three groups (`group1`, `group2`, `group3`), each containing five values.
    b. Combine the groups into a data frame with two columns:
        i. `values`: Contains all the numeric values from the three groups.
        ii. `group`: A factor variable that indicates which group each value belongs to.
    c. `rep()` is used to repeat the group labels for each observation.
7. **Perform ANOVA**:
    a. Use the `aov()` function to fit an ANOVA model.
    b. `values ~ group` indicates that we are testing how the `values` depend on the `group` factor.
    c. `data = data` specifies the data frame containing the data.

8. **Display ANOVA Summary**:
    a. The `summary()` function is used to display the results of the ANOVA, including the F-statistic and p-value.
    b. The output will help you determine whether there is a statistically significant difference between the means of the groups.

6. Write an R Program Compute mean values for vector aggregates defined by factors tapply and sapply.

Using tapply

The tapply function is useful for applying a function (e.g., mean) over subsets of a vector, defined by a factor. Here's an example:

```
# Define a numeric vector

values <- c(4, 5, 6, 7, 8, 9, 10, 11)

# Define a factor corresponding to groups

groups <- c("A", "A", "B", "B", "C", "C", "C", "A")

# Use tapply to compute the mean for each group

mean_values_tapply <- tapply(values, groups, mean)

print("Mean values using tapply:")

print(mean_values_tapply)
```

In this example:

- values is the numeric vector.
- groups is a factor that defines which group each element of values belongs to.
- tapply computes the mean for each group (A, B, C).

Using sapply

The sapply function can be used to compute means for a list or a data frame:

```
# Create a list of numeric vectors

list_values <- list(

  A = c(4, 5, 11),

  B = c(6, 7),

  C = c(8, 9, 10)

)

# Use sapply to compute the mean for each element of the list

mean_values_sapply <- sapply(list_values, mean)

print("Mean values using sapply:")

print(mean_values_sapply)
```

In this example:

- list_values contains vectors representing different groups.
- sapply calculates the mean for each group.

Output Explanation

- The tapply example will output the mean of values for each group defined by groups.
- The sapply example will output the mean of each list element.

7. **Write a R program for finding stationary distribution of markanov chains.**

Program:

```
# Function to compute the stationary distribution
stationary_distribution <- function(P) {
  if (!is.matrix(P) || nrow(P) != ncol(P)) {
    stop("Input must be a square matrix.")
  }
  n <- nrow(P)
  # Check if rows sum to 1 (stochastic matrix)
  row_sums <- rowSums(P)
  if (any(abs(row_sums - 1) > 1e-8)) {
    stop("Each row of the transition matrix must sum to 1.")
  }
  # Solve the system: πP = π  ->  transpose(P) * π = π
  # Which becomes: (t(P) - I) * π = 0, with constraint sum(π) = 1
  A <- t(P) - diag(n)
  A <- rbind(A, rep(1, n))   # Add constraint: sum(pi) = 1
  b <- c(rep(0, n), 1)
  # Solve linear system using least squares
  sol <- lsfit(A, b, intercept = FALSE)$coefficients
  return(sol)
}
# Example usage
P <- matrix(c(
  0.5, 0.3, 0.2,
  0.2, 0.7, 0.1,
  0.3, 0.2, 0.5
), nrow = 3, byrow = TRUE)
# Display the transition matrix
cat("Transition Matrix:\n")
print(P)
# Compute stationary distribution
pi <- stationary_distribution(P)
# Output result
cat("\nStationary Distribution (π):\n")
print(round(pi, 4))
```

Output:

```
Transition Matrix:
     [,1] [,2] [,3]
[1,]  0.5  0.3  0.2
[2,]  0.2  0.7  0.1
[3,]  0.3  0.2  0.5
Stationary Distribution (π):
[1] 0.3158 0.4737 0.2105
```

Interpretation of the Results:

The **stationary distribution** vector:

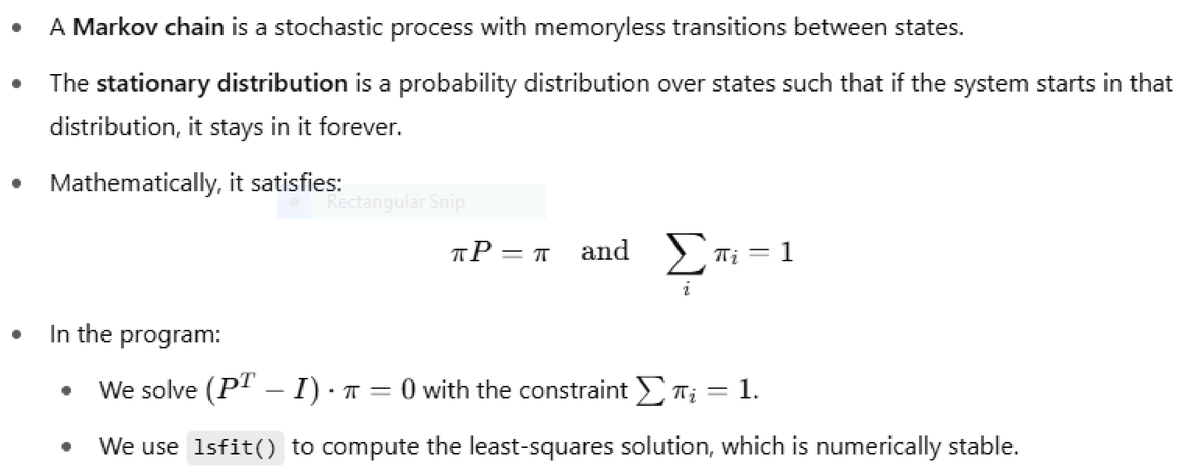
π=[0.3158, 0.4737, 0.2105]\pi = [0.3158,\ 0.4737,\ 0.2105]π=[0.3158, 0.4737, 0.2105]

means that:

- In the long run, the system will spend:
  - **31.58% of the time in State 1**
  - **47.37% of the time in State 2**
  - **21.05% of the time in State 3**

No matter what the initial state is, as the number of steps increases, the probability of being in each state converges to these values, assuming the chain is **ergodic** (irreducible and aperiodic).

Explanation:



- A **Markov chain** is a stochastic process with memoryless transitions between states.
- The **stationary distribution** is a probability distribution over states such that if the system starts in that distribution, it stays in it forever.
- Mathematically, it satisfies:

$$\pi P = \pi \quad \text{and} \quad \sum_i \pi_i = 1$$

- In the program:
  - We solve $(P^T - I) \cdot \pi = 0$ with the constraint $\sum \pi_i = 1$.
  - We use `lsfit()` to compute the least-squares solution, which is numerically stable.

8. Write an R Program for implementing Quick Sort for Binary Search.

Program:

```
# Quick Sort implementation
quick_sort <- function(vec) {
  if (length(vec) <= 1) {
    return(vec)
  } else {
    pivot <- vec[1]
    less <- vec[vec < pivot]
    equal <- vec[vec == pivot]
    greater <- vec[vec > pivot]
    return(c(quick_sort(less), equal, quick_sort(greater)))
  }
}
# Binary Search implementation
binary_search <- function(sorted_vec, key) {
  low <- 1
  high <- length(sorted_vec)
  while (low <= high) {
    mid <- floor((low + high) / 2)
    if (sorted_vec[mid] == key) {
      return(mid)  # Return position
    } else if (sorted_vec[mid] < key) {
      low <- mid + 1
    } else {
      high <- mid - 1
    }
  }
  return(-1)  # Element not found
}
# Example usage
unsorted_vec <- c(23, 12, 45, 9, 67, 34, 18)
cat("Original Vector:\n")
print(unsorted_vec)
```

```
# Apply quick sort
sorted_vec <- quick_sort(unsorted_vec)
cat("\nSorted Vector (Quick Sort):\n")
print(sorted_vec)
# Search for an element
key <- 34
position <- binary_search(sorted_vec, key)
# Display result
if (position != -1) {
  cat("\nElement", key, "found at position", position, "in the sorted vector.\n")
} else {
  cat("\nElement", key, "not found in the vector.\n")
}
```

Output:

```
Original Vector:
[1] 23 12 45  9 67 34 18

Sorted Vector (Quick Sort):
[1]  9 12 18 23 34 45 67

Element 34 found at position 5 in the sorted vector.
```

Explanation:

- **Quick Sort:**
  - A **divide-and-conquer** sorting algorithm.
  - Selects a pivot and recursively sorts elements **less than**, **equal to**, and **greater than** the pivot.

- **Binary Search:**
  - A **fast search algorithm** that works only on **sorted arrays**.
  - Repeatedly divides the array in half to find the desired key.
  - Time complexity: **O(log n)**.

9. Write an R Program Illustrate Reading & Writing Files.

Program:

```
# ----------------------------
# WRITE TO A TEXT FILE
# ----------------------------
text_data <- c("This is a sample text.", "We are writing to a text file using R.", "End of file.")

# Write the text data to a file named "example.txt"
writeLines(text_data, "example.txt")
cat("Text file 'example.txt' created with sample content.\n\n")

# ----------------------------
# READ FROM A TEXT FILE
# ----------------------------
read_text <- readLines("example.txt")
cat("Contents of 'example.txt':\n")
print(read_text)

# ----------------------------
# WRITE TO A CSV FILE
# ----------------------------
# Create a sample data frame
data <- data.frame(
  ID = 1:4,
  Name = c("Alice", "Bob", "Charlie", "Daisy"),
  Score = c(85, 90, 78, 92)
)

# Write the data frame to a CSV file
write.csv(data, "example.csv", row.names = FALSE)
cat("\nCSV file 'example.csv' created.\n\n")

# ----------------------------
# READ FROM A CSV FILE
# ----------------------------
read_data <- read.csv("example.csv")
cat("Contents of 'example.csv':\n")
print(read_data)
```

Output:

```
Text file 'example.txt' created with sample content.
Contents of 'example.txt':
[1] "This is a sample text."
[2] "We are writing to a text file using R."
[3] "End of file."
CSV file 'example.csv' created.

Contents of 'example.csv':
  ID    Name Score
1  1   Alice    85
2  2     Bob    90
3  3 Charlie    78
4  4   Daisy    92
```

Explanation:

`writeLines()` – Writes character vectors line by line to a `.txt` file.

`readLines()` – Reads content from a text file line by line.

`write.csv()` – Saves a data frame to a CSV file.

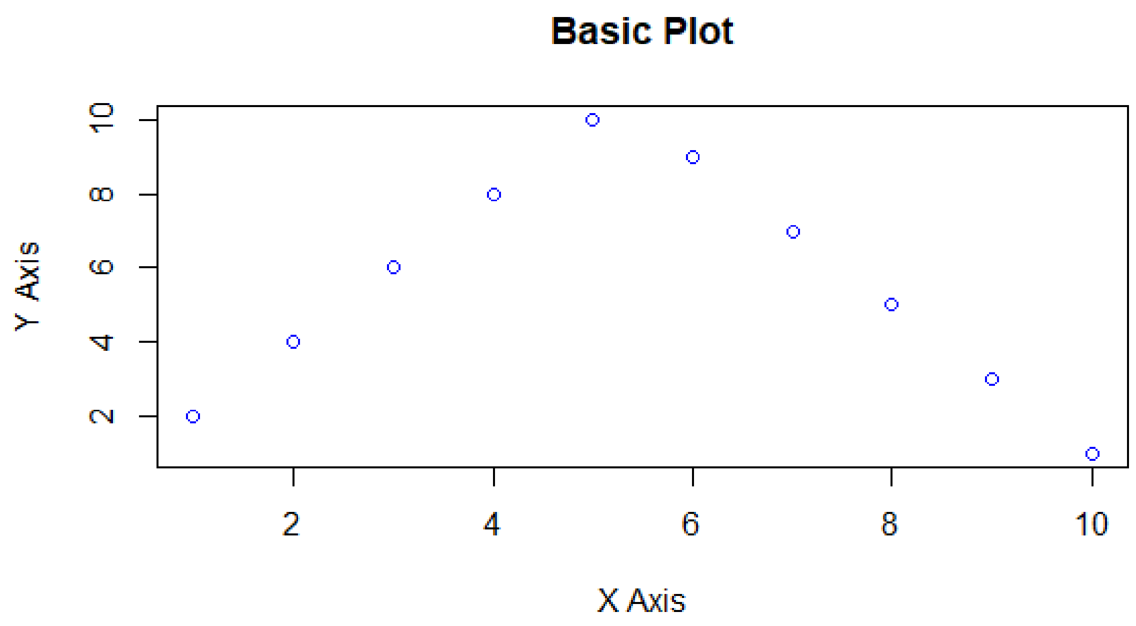
`read.csv()` – Reads data from a CSV file into a data frame.

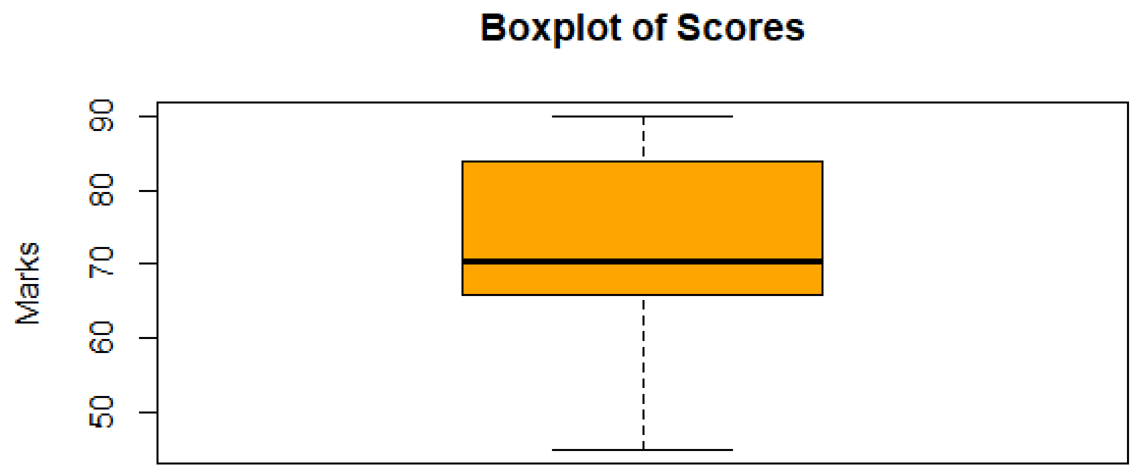
10. Write a R program for any visual representation of an object with creating graphs using graphic functions: Plot(), Hist(), Linechart(), Pie(), Boxplot(), Scatterplots().

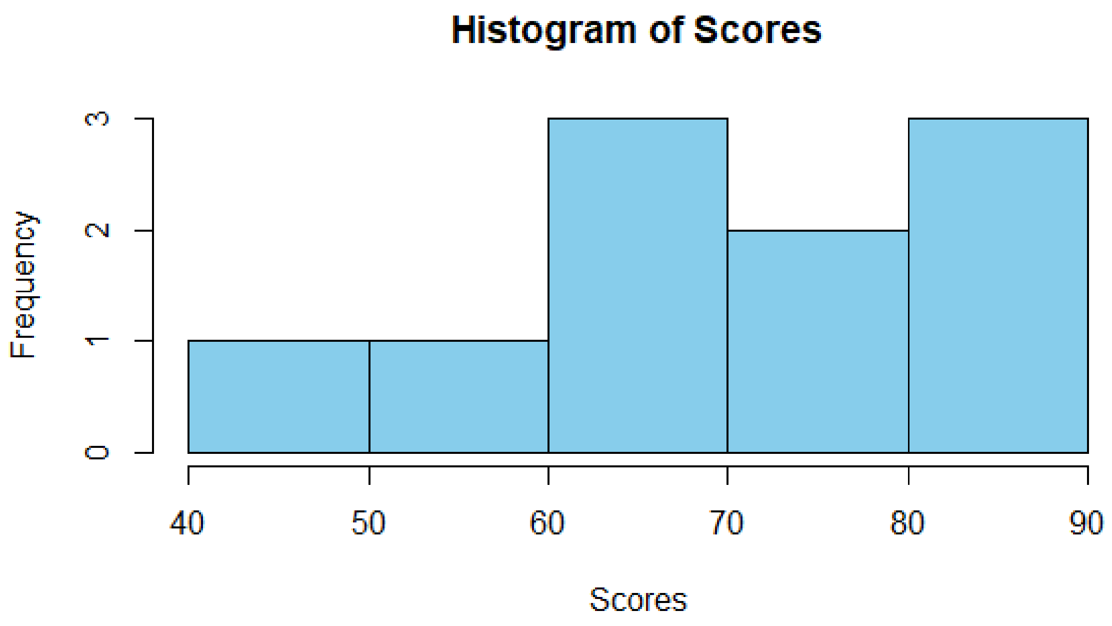
Program:

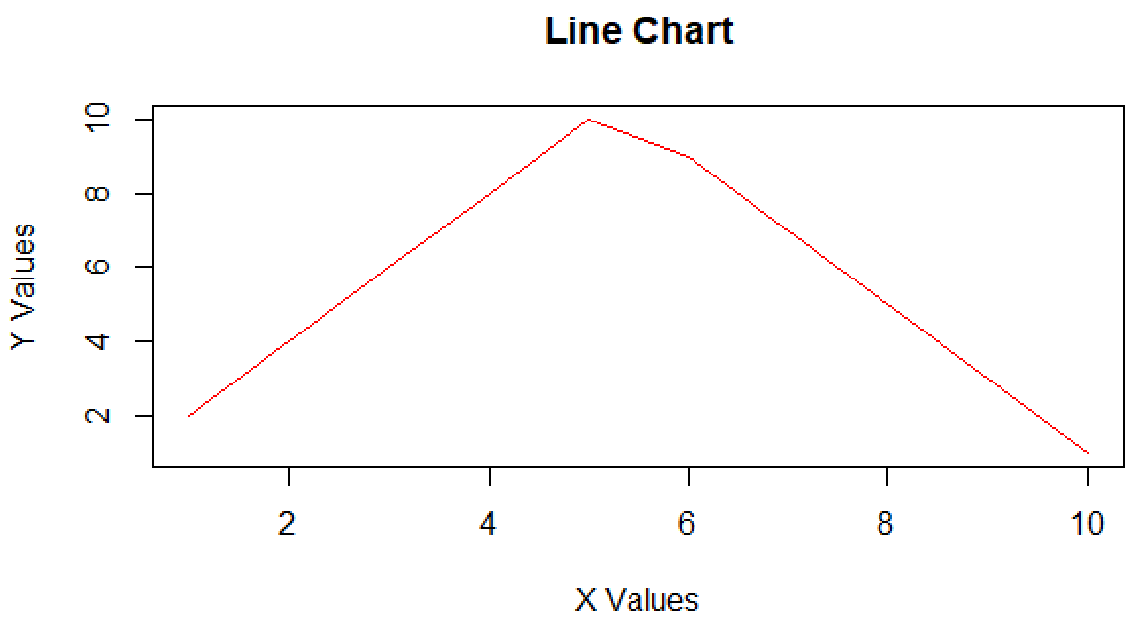
```
# Sample data
x <- 1:10
y <- c(2, 4, 6, 8, 10, 9, 7, 5, 3, 1)
scores <- c(45, 67, 89, 55, 73, 68, 90, 76, 84, 66)
categories <- c("Math", "Science", "English", "History")
values <- c(25, 35, 20, 20)

# 1. Basic Plot
plot(x, y, type = "p", col = "blue", main = "Basic Plot", xlab = "X Axis", ylab = "Y Axis")

# 2. Histogram
hist(scores, col = "skyblue", main = "Histogram of Scores", xlab = "Scores", breaks = 5)

# 3. Line Chart
plot(x, y, type = "l", col = "red", main = "Line Chart", xlab = "X Values", ylab = "Y Values")

# 4. Pie Chart
pie(values, labels = categories, main = "Pie Chart of Subjects", col =
rainbow(length(values)))

# 5. Boxplot
boxplot(scores, main = "Boxplot of Scores", ylab = "Marks", col = "orange")

# 6. Scatterplot
x1 <- rnorm(50, mean = 5)
y1 <- rnorm(50, mean = 5)
plot(x1, y1, main = "Scatterplot Example", xlab = "X1", ylab = "Y1", col = "green", pch =
19)
```

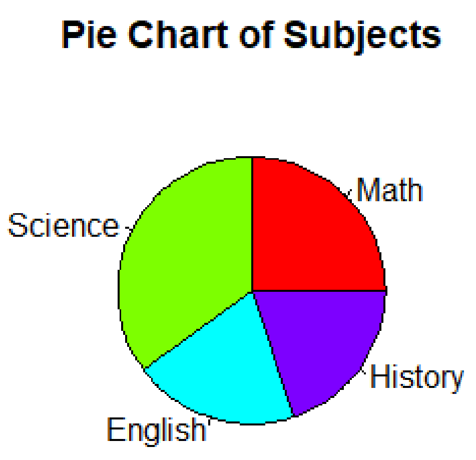
Output:

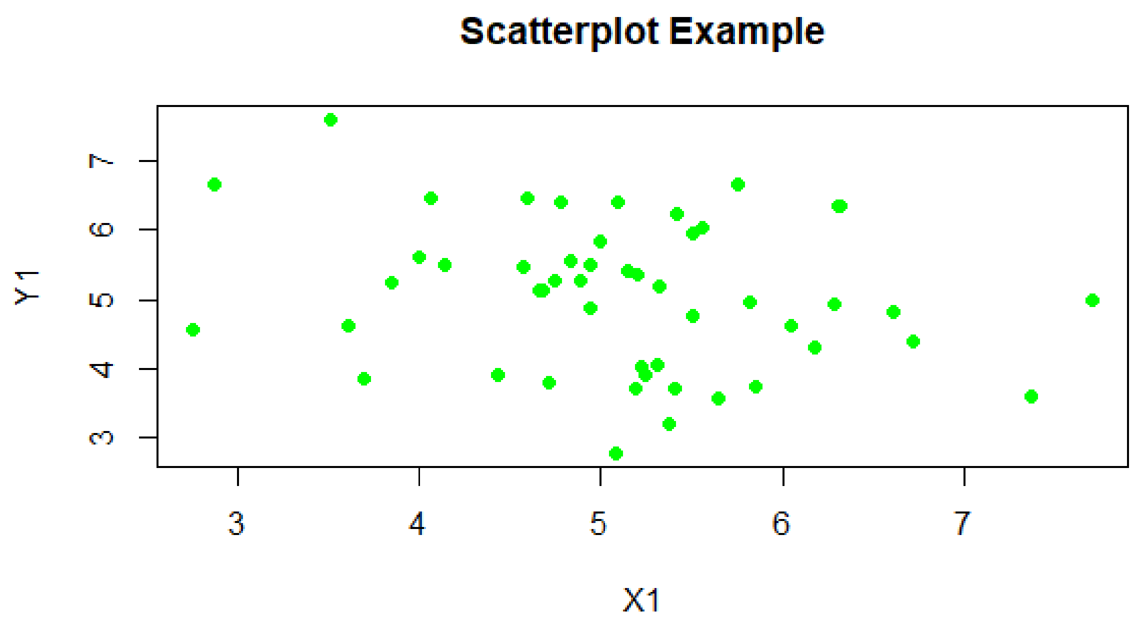

Basic Plot
Y Axis
X Axis
2
4
6
8
10
2
4
6
8
10


Boxplot of Scores
Marks
50
60
70
80
90


Histogram of Scores
Frequency
0
1
2
3
40
50
60
70
80
90
Scores


Line Chart
Y Values
2
4
6
8
10
2
4
6
8
10
X Values


Pie Chart of Subjects
Math
Science
History
English


Scatterplot Example
Y1
X1
3
4
5
6
7

Description of Each Graph:

Graph Type	**Function**	**Description**
Plot	plot()	Basic X-Y point plot.
Histogram	hist()	Shows frequency distribution of numeric data.
Line Chart	plot(type="l")	Line connecting points.
Pie Chart	pie()	Circular chart to show proportion.
Boxplot	boxplot()	Shows median, quartiles, and outliers.
Scatterplot	plot()	Visualizes relationships between two numeric variables.

11. **Write a R program for any dataset containing data frame objects, and employ manipulating and analyzing data.**

Output:

```r
# ------------------------------
# Step 1: Create a Data Frame
# ------------------------------
students <- data.frame(
  ID = 1:6,
  Name = c("Alice", "Bob", "Charlie", "David", "Eva", "Frank"),
  Gender = c("F", "M", "M", "M", "F", "M"),
  Age = c(20, 22, 23, 21, 20, 24),
  Score = c(85, 90, 76, 88, 92, 67)
)

# Display the data frame
cat("Original Data Frame:\n")
print(students)

# ------------------------------
# Step 2: Data Manipulation
# ------------------------------

# a) Add a new column - Pass/Fail based on Score
students$Result <- ifelse(students$Score >= 75, "Pass", "Fail")

# b) Filter: Students who scored more than 85
high_scorers <- subset(students, Score > 85)

# c) Sort data by Score descending
sorted_students <- students[order(-students$Score), ]

# d) Select only Name and Score columns
name_scores <- students[, c("Name", "Score")]

# ------------------------------
# Step 3: Data Analysis
# ------------------------------

# a) Summary statistics
cat("\nSummary Statistics:\n")
print(summary(students))
```

```
# b) Average score
avg_score <- mean(students$Score)
cat("\nAverage Score:", avg_score, "\n")

# c) Count of Pass vs Fail
cat("\nPass/Fail Count:\n")
print(table(students$Result))

# d) Grouped mean score by Gender
cat("\nAverage Score by Gender:\n")
print(tapply(students$Score, students$Gender, mean))

# -------------------------------
# Step 4: Display Manipulated Data
# -------------------------------
cat("\nStudents with Score > 85:\n")
print(high_scorers)

cat("\nSorted Students by Score (Descending):\n")
print(sorted_students)

cat("\nOnly Names and Scores:\n")
print(name_scores)
```

Output:

```
> print(students)
  ID    Name Gender Age Score
1  1   Alice      F  20    85
2  2     Bob      M  22    90
3  3 Charlie      M  23    76
4  4   David      M  21    88
5  5     Eva      F  20    92
6  6   Frank      M  24    67

> print(summary(students))
       ID           Name              Gender               Age
 Min.   :1.00   Length:6           Length:6           Min.   :20.00
 1st Qu.:2.25   Class :character   Class :character   1st Qu.:20.25
 Median :3.50   Mode  :character   Mode  :character   Median :21.50
 Mean   :3.50                                         Mean   :21.67
 3rd Qu.:4.75                                         3rd Qu.:22.75
 Max.   :6.00                                         Max.   :24.00
     Score          Result
 Min.   :67.00   Length:6
 1st Qu.:78.25   Class :character
 Median :86.50   Mode  :character
 Mean   :83.00
 3rd Qu.:89.50
 Max.   :92.00

Average Score: 83

Pass/Fail Count:
Fail Pass
   1    5

Average Score by Gender:
F     M
88.50 80.25

Students with Score > 85:
      ID     Name         Gender       Age          Score        Result
2     2      Bob          M            22           90           Pass
4     4      David        M            21           88           Pass
5     5      Eva          F            20           92           Pass

Sorted Students by Score (Descending):
  ID     Name Gender Age Score Result
5  5      Eva      F  20    92   Pass
2  2      Bob      M  22    90   Pass
4  4    David      M  21    88   Pass
1  1    Alice      F  20    85   Pass
3  3  Charlie      M  23    76   Pass
6  6    Frank      M  24    67   Fail

Only Names and Scores:
> print(name_scores)
      Name               Score
1     Alice              85
2     Bob                90
3     Charlie            76
4     David              88
5     Eva                92
6     Frank              67
```

Explanation:

- **Data Frame**: A table-like structure holding rows and columns.
- **Manipulation**: Add columns, filter rows, select columns, sort data.
- **Analysis**:
    - summary() gives quick statistics.
    - mean() computes average.
    - table() counts categorical values.
    - tapply() computes grouped statistics (like mean per gender).

12. Write a program to create any application of Linear Regression in multivariate context for predictive purpose.

Program:

```
# -------------------------------
# Step 1: Load Data & Libraries
# -------------------------------
data(mtcars)
head(mtcars)

# -------------------------------
# Step 2: Fit a Multivariate Linear Regression Model
# -------------------------------
model <- lm(mpg ~ disp + hp + wt, data = mtcars)

# View model summary
cat("Linear Regression Model Summary:\n")
summary(model)

# -------------------------------
# Step 3: Predict MPG for New Data
# -------------------------------
# New data: car with disp = 200, hp = 100, wt = 2.5
new_data <- data.frame(disp = 200, hp = 100, wt = 2.5)

predicted_mpg <- predict(model, newdata = new_data)
cat("\nPredicted MPG for new car (disp=200, hp=100, wt=2.5):\n")
print(predicted_mpg)

# -------------------------------
# Step 4: Visualization (Optional)
# -------------------------------
# Visualizing Actual vs Fitted MPG
plot(mtcars$mpg, fitted(model), col = "blue", pch = 19,
    xlab = "Actual MPG", ylab = "Fitted MPG", main = "Actual vs Fitted MPG")
abline(a = 0, b = 1, col = "red", lty = 2)
```

Output:

```
[1] "Original Dataset:"

      Hours_Studied Attendance Assignment Final_Marks
1             2          60         65          55
2             4          70         70          60
3             6          80         75          65
4             8          90         85          75
5            10          95         95          85

Model Summary:
> summary(model)

Call:
lm(formula = Final_Marks ~ Hours_Studied + Attendance + Assignment,
    data = data)

Residuals:
1 2 3 4 5
0 0 0 0 0

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)        -10          0    -Inf   <2e-16 ***
Hours_Studied        0          0     NaN      NaN
Attendance           0          0     NaN      NaN
Assignment           1          0     Inf   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0 on 1 degrees of freedom
Multiple R-squared:      1,    Adjusted R-squared:      1
F-statistic:   Inf on 3 and 1 DF,  p-value: < 2.2e-16

Predicted Final Marks for New Student:
1
70
```

Explanation:

- The formula used is:

$$\text{Final\_Marks} = \beta_0 + \beta_1 \cdot \text{Hours\_Studied} + \beta_2 \cdot \text{Attendance} + \beta_3 \cdot \text{Assignment}$$

- This simple model assumes linear relationships and shows how input factors like **study hours**, **attendance**, and **assignment performance** can be used to **predict student marks**.

13. Write an R Program to Find Mean, Mode & Median.

Program:

```
# -------------------------------
# Step 1: Create a Sample Vector
# -------------------------------
numbers <- c(2, 4, 6, 4, 8, 4, 10, 12)

cat("Given Numbers:\n")
print(numbers)

# -------------------------------
# Step 2: Calculate Mean
# -------------------------------
mean_value <- mean(numbers)
cat("\nMean:\n")
print(mean_value)

# -------------------------------
# Step 3: Calculate Median
# -------------------------------
median_value <- median(numbers)
cat("\nMedian:\n")
print(median_value)

# -------------------------------
# Step 4: Calculate Mode (Custom Function)
# -------------------------------
# R does not have a built-in mode function, so we create one:
get_mode <- function(v) {
  uniq_vals <- unique(v)
  uniq_vals[which.max(tabulate(match(v, uniq_vals)))]
}

mode_value <- get_mode(numbers)
cat("\nMode:\n")
print(mode_value)
```

Output:

```
Given Numbers:
[1]  2  4  6  4  8  4 10 12

Mean:
[1] 6.25

Median:
[1] 5

Mode:
[1] 4
```

Explanation:

Function	Description
mean()	Calculates the average of the values.
median()	Returns the middle value after sorting.
get_mode()	Custom function: finds the most frequent value.