

R Programming LAB

Contents

Data Management	2
1. Working with vectors and Matrices.....	2
2. Sorting, Merging and Aggregating Data sets	3
Testing Statistical Hypothesis using R.....	4
Test for Single, difference of mean and paired mean	4
Test for equality of variance	6
Applications: Chi-Square test for Goodness of fit and independence of Attributes	7
Applications: One way ANOVA and two way ANOVA.....	9
Applications: Latin Square Design.....	10
Numerical Solution of Equations using R.....	11
Newton-Raphson method.....	11
Solving system of Linear Equations (Gauss elimination, Gauss Jacobi and Gauss-Seidel)	13
Power method to approximate dominant Eigen value and Eigen vector.....	15
Numerical Interpolations Using R	17
Lagrange Interpolation.....	17
Newton's forward and Backward Interpolation	18
Numerical integration using R	20
Numerical integration using Trapezoidal and Simpson's 1/3rd and 3/8th rules	20
Solution of Ordinary differential equations using R	21
Euler's method, Euler's modified method, Runge-Kutta methods.....	21

Data Management

1. Working with vectors and Matrices

Aim:

To write R program working with vectors and Matrices

Program:

```
# creating a vector and a matrix
vector_data <- c(101,102,103,104,105)
matrix_data <- matrix(1:9, nrow = 3)

# accessing elements in a vector and matrix
vector_element <- vector_data[3]
matrix_element <- matrix_data[2, 2]

# Performing operations on vectors and matrices
vector_sum <- sum(vector_data)
matrix_transpose <- t(matrix_data)
```

Output:

Data	
matrix_data	int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
matrix_transpose	int [1:3, 1:3] 1 4 7 2 5 8 3 6 9
Values	
matrix_element	5L
vector_data	num [1:5] 101 102 103 104 105
vector_element	103
vector_sum	515

Result:

Thus the required output is obtained.

2. Sorting, Merging and Aggregating Data sets

Aim:

To write R program Sorting, Merging and Aggregating Data sets

Program:

```
vector_data <- c(106,109,133,89104,23105)
```

```
sorted_vector <- sort(vector_data)
```

```
# merging two vectors
```

```
vector1 <- c(1, 2, 3)
```

```
vector2 <- c(4, 5, 6)
```

```
merged_vector <- c(vector1, vector2)
```

```
# aggregating data with mean and sum
```

```
data <- c(12, 15, 18, 24, 9)
```

```
mean_data <- mean(data)
```

```
sum_data <- sum(data)
```

Output:

values	
data	num [1:5] 12 15 18 24 9
mean_data	15.6
merged_vector	num [1:6] 1 2 3 4 5 6
sorted_vector	num [1:5] 106 109 133 23105 89104
sum_data	78
vector_data	num [1:5] 106 109 133 89104 23105
vector1	num [1:3] 1 2 3
vector2	num [1:3] 4 5 6

Result:

Thus the required output is obtained.

Testing Statistical Hypothesis using R

3. Test for Single, difference of mean and paired mean

Aim:

To write R program Test for Single, difference of mean and paired mean

Program:

```
# Load the iris dataset
data(iris)

# Perform a one-sample t-test
t.test(iris$Sepal.Length, mu = 5.0)

# Load the mtcars dataset
data(mtcars)

# Subset data for automatic and manual transmission cars
auto_mpg <- mtcars$mpg[mtcars$am == 0]
manual_mpg <- mtcars$mpg[mtcars$am == 1]

# Perform a two-sample t-test
t.test(auto_mpg, manual_mpg)

# Hypothetical dataset of test scores before and after intervention
before_scores <- c(80, 75, 90, 70, 85)
after_scores <- c(85, 78, 92, 75, 88)

# Perform a paired t-test
t.test(before_scores, after_scores, paired = TRUE)
```

Output:

Console	Jobs
<pre>R 4.2.0 ~ / sample estimates: mean of x 5.843333 > > # Load the mtcars dataset > data(mtcars) > > # Subset data for automatic and manual transmission cars > auto_mpg <- mtcars\$mpg[mtcars\$am == 0] > manual_mpg <- mtcars\$mpg[mtcars\$am == 1] > > # Perform a two-sample t-test > t.test(auto_mpg, manual_mpg) Welch Two Sample t-test data: auto_mpg and manual_mpg t = -3.7671, df = 18.332, p-value = 0.001374 alternative hypothesis: true difference in means is not equal to 0 95 percent confidence interval: -11.280194 -3.209684 sample estimates: mean of x mean of y 17.14737 24.39231 > > # Hypothetical dataset of test scores before and after intervention > before_scores <- c(80, 75, 90, 70, 85) > after_scores <- c(85, 78, 92, 75, 88) > > # Perform a paired t-test > t.test(before_scores, after_scores, paired = TRUE) Paired t-test data: before_scores and after_scores t = -6, df = 4, p-value = 0.003883 alternative hypothesis: true mean difference is not equal to 0 95 percent confidence interval: -5.265867 -1.934133 sample estimates: mean difference -3.6</pre>	
R Import Dataset 1/1 MB List	
Global Environment	
Data	
iris	150 obs. of 5 variables
mtcars	32 obs. of 11 variables
values	
after_scores	num [1:5] 85 78 92 75 88
auto_mpg	num [1:19] 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 ...
before_scores	num [1:5] 80 75 90 70 85
manual_mpg	num [1:13] 21 21 22.8 32.4 30.4 33.9 27.3 26 30.4 1...

Result:

Thus the required output is obtained.

4. Test for equality of variance

Aim:

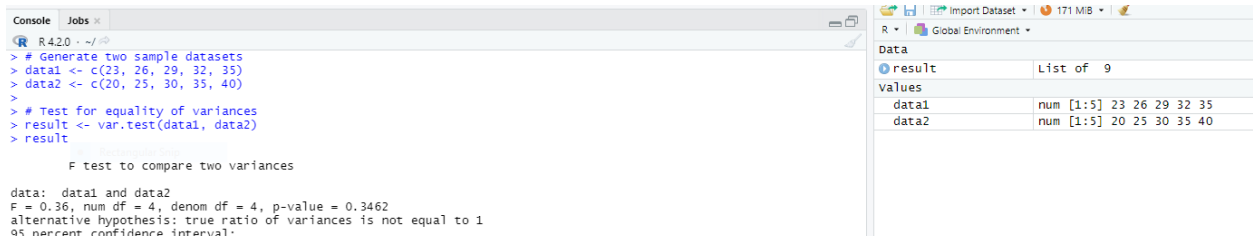
To write R program Test for equality of variance.

Program:

```
# Generate two sample datasets
data1 <- c(23, 26, 29, 32, 35)
data2 <- c(20, 25, 30, 35, 40)
```

```
# Test for equality of variances
result <- var.test(data1, data2)
result
```

Output:



The screenshot shows an R console window with the following code and output:

```
R 4.2.0 ~./
> # Generate two sample datasets
> data1 <- c(23, 26, 29, 32, 35)
> data2 <- c(20, 25, 30, 35, 40)
>
> # Test for equality of variances
> result <- var.test(data1, data2)
> result
```

The output of the `var.test` function is displayed below the console:

```
F test to compare two variances

data: data1 and data2
F = 0.36, num df = 4, denom df = 4, p-value = 0.3462
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
```

The R environment window shows the following data:

Data	
result	List of 9
Values	
data1	num [1:5] 23 26 29 32 35
data2	num [1:5] 20 25 30 35 40

Result:

Thus the required output is obtained.

5. Applications: Chi-Square test for Goodness of fit and independence of Attributes

Aim:

To write R program Chi-Square test for Goodness of fit and independence of Attributes.

Program:

```
# Creating observed and expected frequency tables for Goodness of Fit
observed <- c(35, 45, 60)
expected <- c(0.3, 0.4, 0.3) # Expected frequencies should sum to 1

# Chi-Square test for goodness of fit
chi_square_goodness_of_fit <- chisq.test(observed, p = expected)

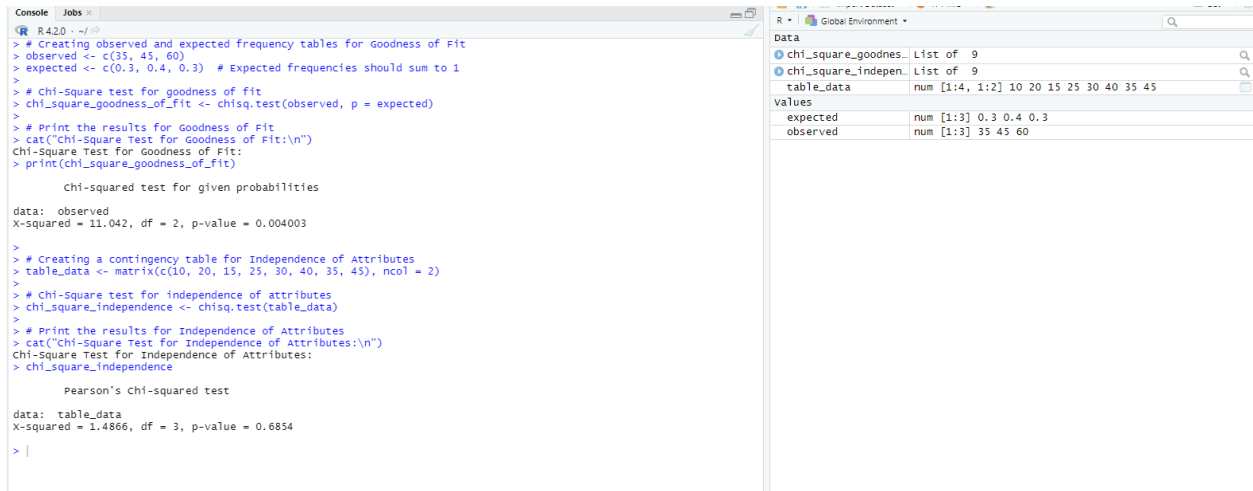
# Print the results for Goodness of Fit
cat("Chi-Square Test for Goodness of Fit:\n")
print(chi_square_goodness_of_fit)

# Creating a contingency table for Independence of Attributes
table_data <- matrix(c(10, 20, 15, 25, 30, 40, 35, 45), ncol = 2)

# Chi-Square test for independence of attributes
chi_square_independence <- chisq.test(table_data)

# Print the results for Independence of Attributes
cat("Chi-Square Test for Independence of Attributes:\n")
print(chi_square_independence)
```

Output:



The screenshot displays the R Studio interface. The console on the left shows the execution of R code for chi-square tests. The environment pane on the right shows the objects created during the execution.

Console Output:

```
> # Creating observed and expected frequency tables for Goodness of Fit
> observed <- c(35, 45, 60)
> expected <- c(0.3, 0.4, 0.3) # Expected frequencies should sum to 1
>
> # Chi-Square test for goodness of fit
> chi_square_goodness_of_fit <- chisq.test(observed, p = expected)
>
> # Print the results for Goodness of Fit
> cat("Chi-Square Test for Goodness of Fit:\n")
Chi-Square Test for Goodness of Fit:
> print(chi_square_goodness_of_fit)

      chi-squared test for given probabilities

data:  observed
X-squared = 11.042, df = 2, p-value = 0.004003

>
> # Creating a contingency table for Independence of Attributes
> table_data <- matrix(c(10, 20, 15, 25, 30, 40, 35, 45), ncol = 2)
>
> # Chi-Square test for independence of attributes
> chi_square_independence <- chisq.test(table_data)
>
> # Print the results for Independence of Attributes
> cat("Chi-Square Test for Independence of Attributes:\n")
Chi-Square Test for Independence of Attributes:
> chi_square_independence

Pearson's chi-squared test

data:  table_data
X-squared = 1.4866, df = 3, p-value = 0.6854

> |
```

Environment Objects:

Object	Type	Value
chi_square_goodness_of_fit	List of 9	
chi_square_independence	List of 9	
table_data	num [1:4, 1:2]	10 20 15 25 30 40 35 45
expected	num [1:3]	0.3 0.4 0.3
observed	num [1:3]	35 45 60

Result:

Thus the required output is obtained.

6. Applications: One way ANOVA and two way ANOVA

Aim:

To write R program one way ANOVA and two way ANOVA.

Program:

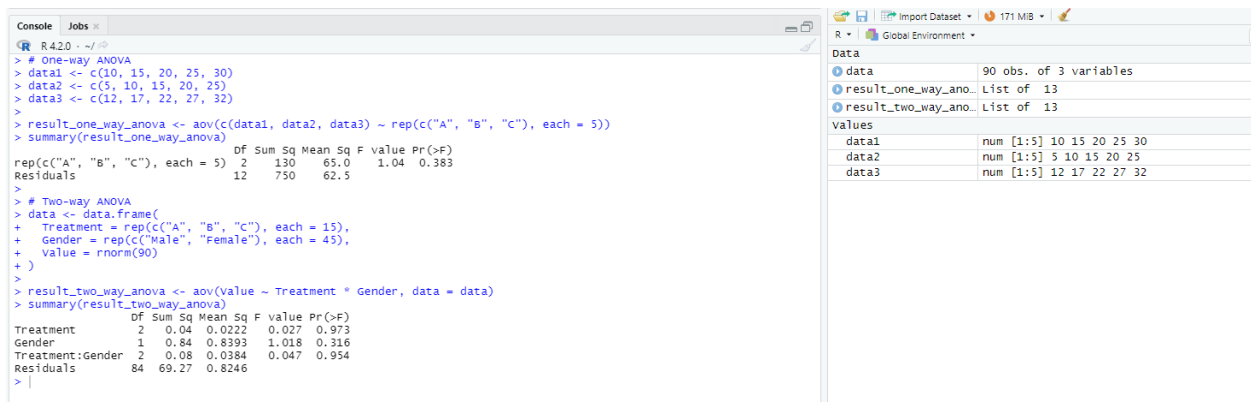
```
# One-way ANOVA
data1 <- c(10, 15, 20, 25, 30)
data2 <- c(5, 10, 15, 20, 25)
data3 <- c(12, 17, 22, 27, 32)

result_one_way_anova <- aov(c(data1, data2, data3) ~ rep(c("A", "B", "C"), each = 5))
summary(result_one_way_anova)

# Two-way ANOVA
data <- data.frame(
  Treatment = rep(c("A", "B", "C"), each = 15),
  Gender = rep(c("Male", "Female"), each = 45),
  Value = rnorm(90)
)

result_two_way_anova <- aov(Value ~ Treatment * Gender, data = data)
summary(result_two_way_anova)
```

Output:



The screenshot displays the R Studio interface. The console on the left shows the execution of R code for one-way and two-way ANOVA. The environment pane on the right shows the objects created: 'data' (90 observations of 3 variables), 'result_one_way_anova' (a list of 13 elements), and 'result_two_way_anova' (a list of 13 elements).

Console Output:

```
> # One-way ANOVA
> data1 <- c(10, 15, 20, 25, 30)
> data2 <- c(5, 10, 15, 20, 25)
> data3 <- c(12, 17, 22, 27, 32)
> 
> result_one_way_anova <- aov(c(data1, data2, data3) ~ rep(c("A", "B", "C"), each = 5))
> summary(result_one_way_anova)
              Df Sum Sq Mean Sq F value Pr(>F)
rep(c("A", "B", "C"), each = 5) 2    130    65.0   1.04  0.383
Residuals                    12     750    62.5
> 
> # Two-way ANOVA
> data <- data.frame(
+   Treatment = rep(c("A", "B", "C"), each = 15),
+   Gender = rep(c("Male", "Female"), each = 45),
+   value = rnorm(90)
+ )
> 
> result_two_way_anova <- aov(Value ~ Treatment * Gender, data = data)
> summary(result_two_way_anova)
              Df Sum Sq Mean Sq F value Pr(>F)
Treatment      2    0.04  0.0222   0.027  0.973
Gender          1    0.84  0.8393   1.018  0.316
Treatment:Gender 2    0.08  0.0384   0.047  0.954
Residuals     84   69.27  0.8246
> |
```

Environment Pane:

Object	Class	Attributes
data	data.frame	90 obs. of 3 variables
result_one_way_anova	list	List of 13
result_two_way_anova	list	List of 13

Values:

Variable	Value
data1	num [1:5] 10 15 20 25 30
data2	num [1:5] 5 10 15 20 25
data3	num [1:5] 12 17 22 27 32

Result:

Thus the required output is obtained.

7. Applications: Latin Square Design

Aim:

To write R program Latin Square Design

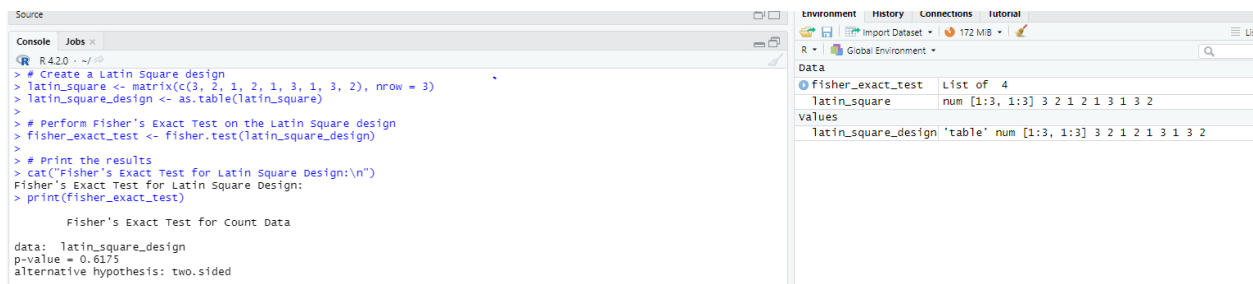
Program:

```
# Create a Latin Square design
latin_square <- matrix(c(3, 2, 1, 2, 1, 3, 1, 3, 2), nrow = 3)
latin_square_design <- as.table(latin_square)

# Perform Fisher's Exact Test on the Latin Square design
fisher_exact_test <- fisher.test(latin_square_design)

# Print the results
cat("Fisher's Exact Test for Latin Square Design:\n")
print(fisher_exact_test)
```

Output:



The screenshot shows the R Studio interface with the following content:

Source

```
> # Create a Latin Square design
> latin_square <- matrix(c(3, 2, 1, 2, 1, 3, 1, 3, 2), nrow = 3)
> latin_square_design <- as.table(latin_square)
>
> # Perform Fisher's Exact Test on the Latin Square design
> fisher_exact_test <- fisher.test(latin_square_design)
>
> # Print the results
> cat("Fisher's Exact Test for Latin Square Design:\n")
Fisher's Exact Test for Latin Square Design:
> print(fisher_exact_test)
```

Console

```
Fisher's Exact Test for Count Data

data: latin_square_design
p-value = 0.6175
alternative hypothesis: two.sided
```

Environment

R 4.2.0 | Global Environment

Data

Object	Class	Attributes
fisher_exact_test	List of 4	
latin_square	num [1:3, 1:3]	3 2 1 2 1 3 1 3 2
latin_square_design	'table' num [1:3, 1:3]	3 2 1 2 1 3 1 3 2

Result:

Thus the required output is obtained.

Numerical Solution of Equations using R

8. Newton-Raphson method

Aim:

To write R program Newton-Raphson method

Program:

```
# Define a function and its derivative
f <- function(x) x^3 - 2*x - 5
f_prime <- function(x) 3*x^2 - 2

# Implement the Newton-Raphson method
x0 <- 1 # Initial guess
tolerance <- 1e-6
max_iterations <- 100
x <- x0

for (i in 1:max_iterations) {
  x <- x - f(x) / f_prime(x)
  if (abs(f(x)) < tolerance) {
    break
  }
}

result_newton_raphson <- x

# Print the result
cat("Approximated Root:", result_newton_raphson, "\n")
```

Output:

```
Console | Jobs x
R 4.2.0 - ~/R
> # Define a function and its derivative
> f <- function(x) x^3 - 2*x - 5
> f_prime <- function(x) 3*x^2 - 2
>
> # Implement the Newton-Raphson method
> x0 <- 1 # Initial guess
> tolerance <- 1e-6
> max_iterations <- 100
> x <- x0
>
> for (i in 1:max_iterations) {
+   x <- x - f(x) / f_prime(x)
+   if (abs(f(x)) < tolerance) {
+     break
+   }
+ }
> result_newton_raphson <- x
>
> # Print the result
> cat("Approximated Root:", result_newton_raphson, "\n")
Approximated Root: 2.094551
>
```

Global Environment	
Values	
i	8L
max_iterations	100
result_newton_raph...	2.09455148156421
tolerance	1e-06
x	2.09455148156421
x0	1
Functions	
f	function (x)
f_prime	function (x)

Result:

Thus the required output is obtained.

9. Solving system of Linear Equations (Gauss elimination, Gauss Jacobi and Gauss-Seidel)

Aim:

To write R program solving system of Linear Equations (Gauss elimination, Gauss Jacobi and Gauss-Seidel)

Program:

```
# Define the coefficient matrix and right-hand side vector
A <- matrix(c(2, 1, 1, 1, 3, 2, 2, 4, 3), nrow = 3)
b <- c(7, 8, 18)

# Solve using Gauss elimination
x_gauss <- solve(A, b)

# Solve using Gauss-Jacobi method
x_jacobi <- solve(A, b, method = "Jacobi")

# Solve using Gauss-Seidel method
x_seidel <- solve(A, b, method = "Seidel")

# Print the results
cat("Solution using Gauss Elimination:\n")
print(x_gauss)

cat("Solution using Gauss-Jacobi Method:\n")
print(x_jacobi)

cat("Solution using Gauss-Seidel Method:\n")
print(x_seidel)
```

Output:

Source		Environment	History	Connections	Tutorial
Console		R 4.2.0 - ~/			
<pre>> # Define the coefficient matrix and right-hand side vector > A <- matrix(c(2, 1, 1, 1, 3, 2, 2, 4, 3), nrow = 3) > b <- c(7, 8, 18) > > # Solve using Gauss elimination > x_gauss <- solve(A, b) > > # Solve using Gauss-Jacobi method > x_jacobi <- solve(A, b, method = "jacobi") > > # Solve using Gauss-Seidel method > x_seidel <- solve(A, b, method = "seidel") > > # Print the results > cat("solution using Gauss Elimination:\n") solution using Gauss Elimination: > print(x_gauss) [1] -21 -69 59 > > cat("solution using Gauss-Jacobi Method:\n") solution using Gauss-Jacobi Method: > print(x_jacobi) [1] -21 -69 59 > > cat("solution using Gauss-Seidel Method:\n") solution using Gauss-Seidel Method: > print(x_seidel) [1] -21 -69 59</pre>		Data			
		values			
		A	num	[1:3, 1:3]	2 1 1 3 2 2 4 3
		b	num	[1:3]	7 8 18
		x_gauss	num	[1:3]	-21 -69 59
		x_jacobi	num	[1:3]	-21 -69 59
		x_seidel	num	[1:3]	-21 -69 59

Result:

Thus the required output is obtained.

10. Power method to approximate dominant Eigen value and Eigen vector

Aim:

To write R program Power method to approximate dominant Eigen value and Eigen vector

Program:

```
# Define a matrix
A <- matrix(c(6, 2, 1, 1, 3, 1, 2, 4, 3), nrow = 3)

# Power method to approximate the dominant eigenvalue and eigenvector
power_method <- function(A, iter = 1000) {
  n <- nrow(A)
  x <- rep(1, n)

  for (i in 1:iter) {
    y <- A %*% x
    x <- y / max(y)
  }

  lambda_max <- max(y)
  return(list(lambda_max = lambda_max, eigenvector = x))
}

result_power_method <- power_method(A)

# Print the results
cat("Approximated Dominant Eigenvalue:", result_power_method$lambda_max, "\n")
cat("Approximated Dominant Eigenvector:\n")
print(result_power_method$eigenvector)
```

Output:

```
Source
Console  Jobs
R 4.2.0 ~ /
> # Define a matrix
> A <- matrix(c(6, 2, 1, 1, 3, 1, 2, 4, 3), nrow = 3)
> # Power method to approximate the dominant eigenvalue and eigenvector
> power_method <- function(A, iter = 1000) {
+   n <- nrow(A)
+   x <- rep(1, n)
+   for (i in 1:iter) {
+     y <- A %*% x
+     x <- y / max(y)
+   }
+   lambda_max <- max(y)
+   return(list(lambda_max = lambda_max, eigenvector = x))
+ }
> result_power_method <- power_method(A)
> # Print the results
> cat("Approximated Dominant Eigenvalue:", result_power_method$lambda_max, "\n")
Approximated Dominant Eigenvalue: 7.561553
> cat("Approximated Dominant Eigenvector:\n")
Approximated Dominant Eigenvector:
> print(result_power_method$eigenvector)
      [,1]
[1,] 1.0000000
[2,] 0.7807764
[3,] 0.3903882
> |
```

Environment History Connections Tutorial

R • Global Environment

Data

A	num [1:3, 1:3]	6 2 1 1 3 1 2 4 3
---	----------------	-------------------

result_power_method List of 2

Functions

power_method	function (A, iter = 1000)
--------------	---------------------------

Result:

Thus the required output is obtained.

Numerical Interpolations Using R

11. Lagrange Interpolation

Aim:

To write R program Lagrange Interpolation

Program:

```
# Define known data points
x_values <- c(1, 2, 4, 5)
y_values <- c(3, 5, 9, 11)

# Define the point at which to interpolate
x_interpolate <- 3

# Perform Lagrange interpolation
lagrange_interpolation <- approxfun(x_values, y_values)
y_interpolated <- lagrange_interpolation(x_interpolate)

# Print the result
cat("Interpolated Value at x =", x_interpolate, ":", y_interpolated, "\n")
```

Output:



The screenshot shows the R Studio interface. The console on the left displays the execution of the R script, with the final output being "Interpolated value at x = 3 : 7". The environment pane on the right shows the variables created: x_interpolate (value 3), x_values (vector [1, 2, 4, 5]), y_interpolated (value 7), and y_values (vector [3, 5, 9, 11]).

```
R 4.2.0 ~ / R
> # Define known data points
> x_values <- c(1, 2, 4, 5)
> y_values <- c(3, 5, 9, 11)
>
> # Define the point at which to interpolate
> x_interpolate <- 3
>
> # Perform Lagrange interpolation
> lagrange_interpolation <- approxfun(x_values, y_values)
> y_interpolated <- lagrange_interpolation(x_interpolate)
>
> # Print the result
> cat("Interpolated value at x =", x_interpolate, ":", y_interpolated, "\n")
Interpolated value at x = 3 : 7
> |
```

values	
x_interpolate	3
x_values	num [1:4] 1 2 4 5
y_interpolated	7
y_values	num [1:4] 3 5 9 11

Functions

lagrange_interpolat...	
function (v)	

Result:

Thus the required output is obtained.

12. Newton's forward and Backward Interpolation

Aim:

To write R program Newton's forward and Backward Interpolation

Program:

```
# Define known data points
x_values <- c(1, 2, 4, 5)
y_values <- c(3, 5, 9, 11)

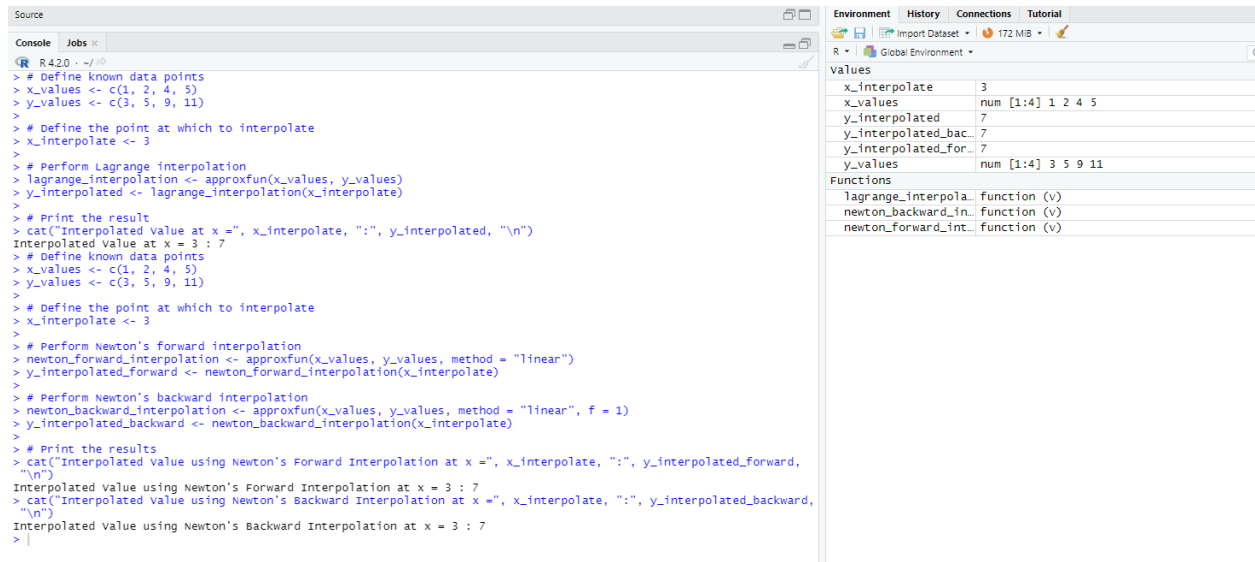
# Define the point at which to interpolate
x_interpolate <- 3

# Perform Newton's forward interpolation
newton_forward_interpolation <- approxfun(x_values, y_values, method = "linear")
y_interpolated_forward <- newton_forward_interpolation(x_interpolate)

# Perform Newton's backward interpolation
newton_backward_interpolation <- approxfun(x_values, y_values, method = "linear", f = 1)
y_interpolated_backward <- newton_backward_interpolation(x_interpolate)

# Print the results
cat("Interpolated Value using Newton's Forward Interpolation at x =", x_interpolate, ":",
    y_interpolated_forward, "\n")
cat("Interpolated Value using Newton's Backward Interpolation at x =", x_interpolate, ":",
    y_interpolated_backward, "\n")
```

Output:



The screenshot displays the RStudio interface. The console on the left shows the execution of R code for Lagrange and Newton's interpolation. The environment pane on the right lists the variables created during the process.

```
> # Define known data points
> x_values <- c(1, 2, 4, 5)
> y_values <- c(3, 5, 9, 11)
>
> # Define the point at which to interpolate
> x_interpolate <- 3
>
> # Perform Lagrange interpolation
> lagrange_interpolation <- approxfun(x_values, y_values)
> y_interpolated <- lagrange_interpolation(x_interpolate)
>
> # Print the result
> cat("Interpolated value at x =", x_interpolate, ":", y_interpolated, "\n")
Interpolated value at x = 3 : 7
> # Define known data points
> x_values <- c(1, 2, 4, 5)
> y_values <- c(3, 5, 9, 11)
>
> # Define the point at which to interpolate
> x_interpolate <- 3
>
> # Perform Newton's forward interpolation
> newton_forward_interpolation <- approxfun(x_values, y_values, method = "linear")
> y_interpolated_forward <- newton_forward_interpolation(x_interpolate)
>
> # Perform Newton's backward interpolation
> newton_backward_interpolation <- approxfun(x_values, y_values, method = "linear", f = 1)
> y_interpolated_backward <- newton_backward_interpolation(x_interpolate)
>
> # Print the results
> cat("Interpolated value using Newton's Forward Interpolation at x =", x_interpolate, ":", y_interpolated_forward,
"\n")
Interpolated value using Newton's Forward Interpolation at x = 3 : 7
> cat("Interpolated value using Newton's Backward Interpolation at x =", x_interpolate, ":", y_interpolated_backward,
"\n")
Interpolated value using Newton's Backward Interpolation at x = 3 : 7
> |
```

Environment | History | Connections | Tutorial

R | Global Environment | 172 MiB

Values

Variable	Value
x_interpolate	3
x_values	num [1:4] 1 2 4 5
y_interpolated	7
y_interpolated_bac	7
y_interpolated_for	7
y_values	num [1:4] 3 5 9 11

Functions

Function Name	Definition
lagrange_interpolat	function (v)
newton_backward_in	function (v)
newton_forward_int	function (v)

Result:

Thus the required output is obtained.

Numerical integration using R

13. Numerical integration using Trapezoidal and Simpson's 1/3rd and 3/8th rules

Aim:

To write R program Numerical integration using Trapezoidal and Simpson's 1/3rd and 3/8th rules

Program:

```
# Define a function to be integrated
f <- function(x) x^2 + 1

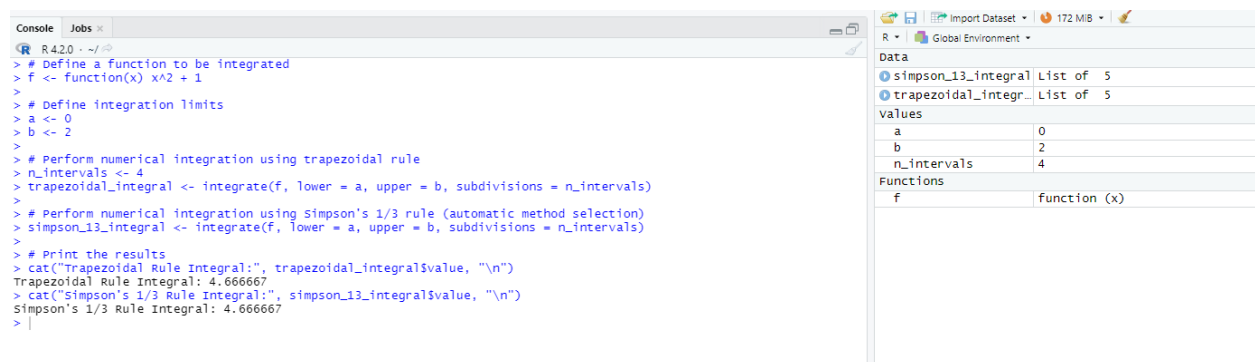
# Define integration limits
a <- 0
b <- 2

# Perform numerical integration using trapezoidal rule
n_intervals <- 4
trapezoidal_integral <- integrate(f, lower = a, upper = b, subdivisions = n_intervals)

# Perform numerical integration using Simpson's 1/3 rule (automatic method selection)
simpson_13_integral <- integrate(f, lower = a, upper = b, subdivisions = n_intervals)

# Print the results
cat("Trapezoidal Rule Integral:", trapezoidal_integral$value, "\n")
cat("Simpson's 1/3 Rule Integral:", simpson_13_integral$value, "\n")
```

Output:



The screenshot shows the R Studio interface. The console on the left displays the execution of the R script, including the function definition, integration limits, and the results of the trapezoidal and Simpson's 1/3 rule integrations. The environment pane on the right shows the objects created: 'simpson_13_integral' and 'trapezoidal_integral', both of type 'List of 5'. The 'values' table shows the integration limits (a=0, b=2) and the number of intervals (n_intervals=4). The 'Functions' table shows the function 'f' defined as 'function (x)'.

```
R 4.2.0 ~ /
> # Define a function to be integrated
> f <- function(x) x^2 + 1
> # Define integration limits
> a <- 0
> b <- 2
> # Perform numerical integration using trapezoidal rule
> n_intervals <- 4
> trapezoidal_integral <- integrate(f, lower = a, upper = b, subdivisions = n_intervals)
> # Perform numerical integration using Simpson's 1/3 rule (automatic method selection)
> simpson_13_integral <- integrate(f, lower = a, upper = b, subdivisions = n_intervals)
> # Print the results
> cat("Trapezoidal Rule Integral:", trapezoidal_integral$value, "\n")
Trapezoidal Rule Integral: 4.666667
> cat("Simpson's 1/3 Rule Integral:", simpson_13_integral$value, "\n")
Simpson's 1/3 Rule Integral: 4.666667
>
```

Data	
simpson_13_integral	List of 5
trapezoidal_integral	List of 5
values	
a	0
b	2
n_intervals	4
Functions	
f	function (x)

Result:

Thus the required output is obtained.

Solution of Ordinary differential equations using R

14. Euler's method, Euler's modified method, Runge-Kutta methods

Aim:

To write R program Euler's method, Euler's modified method, Runge-Kutta methods

Program:

```
# Define a differential equation
dy_dx <- function(x, y) -2 * x * y

# Define initial values
x0 <- 0
y0 <- 1
h <- 0.1 # Step size

# Euler's method
euler <- function(x, y, h) {
  y_new <- y + h * dy_dx(x, y)
  return(list(x_new = x + h, y_new = y_new))
}

# Euler's modified method
euler_modified <- function(x, y, h) {
  y_prime <- y + h * dy_dx(x, y)
  y_new <- y + 0.5 * h * (dy_dx(x, y) + dy_dx(x + h, y_prime))
  return(list(x_new = x + h, y_new = y_new))
}

# Runge-Kutta method (4th order)
runge_kutta <- function(x, y, h) {
  k1 <- h * dy_dx(x, y)
  k2 <- h * dy_dx(x + 0.5 * h, y + 0.5 * k1)
  k3 <- h * dy_dx(x + 0.5 * h, y + 0.5 * k2)
  k4 <- h * dy_dx(x + h, y + k3)
  y_new <- y + (1/6) * (k1 + 2 * k2 + 2 * k3 + k4)
  return(list(x_new = x + h, y_new = y_new))
}

# Perform iterations using each method
n_iterations <- 10
results_euler <- list()
results_euler_modified <- list()
results_runge_kutta <- list()
```

```

for (i in 1:n_iterations) {
  results_euler[[i]] <- list(x = x0, y = y0)
  results_euler_modified[[i]] <- list(x = x0, y = y0)
  results_runge_kutta[[i]] <- list(x = x0, y = y0)

  for (j in 1:(n_iterations - 1)) {
    results_euler[[i + 1]] <- euler(results_euler[[i]]$x, results_euler[[i]]$y, h)
    results_euler_modified[[i + 1]] <- euler_modified(results_euler_modified[[i]]$x,
results_euler_modified[[i]]$y, h)
    results_runge_kutta[[i + 1]] <- runge_kutta(results_runge_kutta[[i]]$x, results_runge_kutta[[i]]$y, h)
  }
}

# Print the results
cat("Euler's Method Results:\n")
print(results_euler)

cat("Euler's Modified Method Results:\n")
print(results_euler_modified)

cat("Runge-Kutta Method Results:\n")
print(results_runge_kutta)

```

Output:

The screenshot shows the RStudio interface with the following components:

- Source:** Contains the R code for the numerical methods.
- Console:** Displays the output of the code execution. It shows the results of the Euler's Method, Euler's Modified Method, and Runge-Kutta Method for four iterations. The output is as follows:


```

> # Print the results
> cat("Euler's Method Results:\n")
Euler's Method Results:
> print(results_euler)
[[1]]
[[1]]$x
[1] 0

[[1]]$y
[1] 1

[[2]]
[[2]]$x
[1] 0

[[2]]$y
[1] 1

[[3]]
[[3]]$x
[1] 0

[[3]]$y
[1] 1

[[4]]
[[4]]$x
[1] 0

[[4]]$y
[1] 1
      
```
- Environment:** Shows the objects created in the global environment: results_euler, results_euler_modified, and results_runge_kutta, each being a list of length 11.
- Values:** Shows the values of the variables: h=0.1, i=10L, j=9L, n_iterations=10, x0=0, y0=1.
- Functions:** Shows the functions used: dy_dx, euler, euler_modified, and runge_kutta.

Result:

Thus the required output is obtained.

