Julia Lab Manual

BDSL456D

Prof. Nelvita L F

Introduction

Julia language is a Free and opensource programming language, developed by MIT professors

This language developed with the below features in mind by the creators

- Julia's code Speed is the same as C, C++, and Fortran code
- Similar to Ruby in Dynamic types
- Works as same as Matlab in mathematical computations
- Works as same as Python for general programming language
- Similar to Statistics computing like R

Julia File Extension

- Julia's code is written in a file with the extension. jl.

What language is Julia written in?

- Julia language compiler is written using C and C++ code and Julia compiler is a free software compiler later many contributed to the release of multiple versions.

Installation Julia

STEP1: Download and install Julia by following the instructions at https://julialang.org/downloads/.

STEP2: Once you open the link you will be able to see this page

Current stable release: v1.10.2 (March 1, 2024)

Checksums for this release are available in both SHA256 and MD5 formats.

Platform	64-bit	32-bit
Windows [help]	installer, portable	installer, portable
macOS x86 (Intel or Rosetta) [help]	.dmg, .tar.gz	
macOS (Apple Silicon) [help]	.dmg, .tar.gz	
Generic Linux on x86 [help]	glibc (GPG), musl[1] (GPG)	glibc (GPG)
Generic Linux on ARM [help]	AArch64 (GPG)	
Generic FreeBSD on x86 [help]	.tar.gz (GPG)	

Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

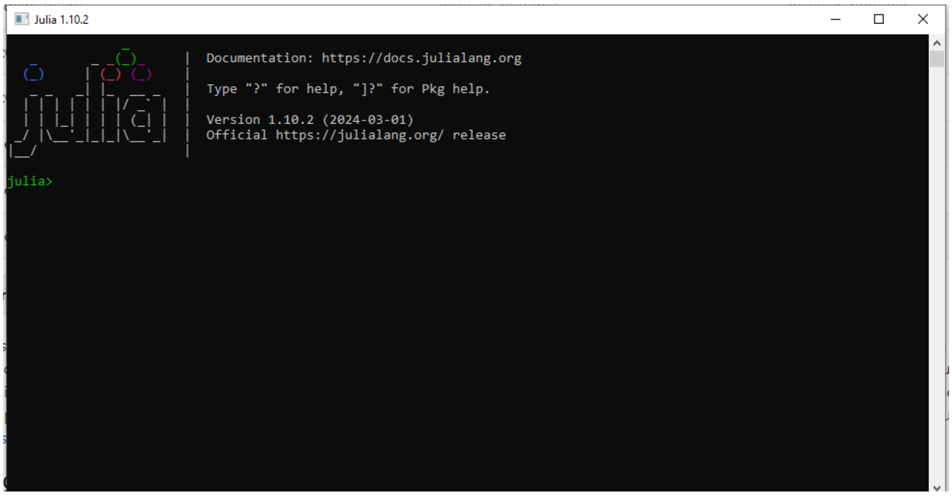
STEP3: Now according to our system specification we can select windows platform. So am using 64bit specifications

Current stable release: v1.10.2 (March 1, 2024)

Checksums for this release are available in both SHA256 and MD5 formats.

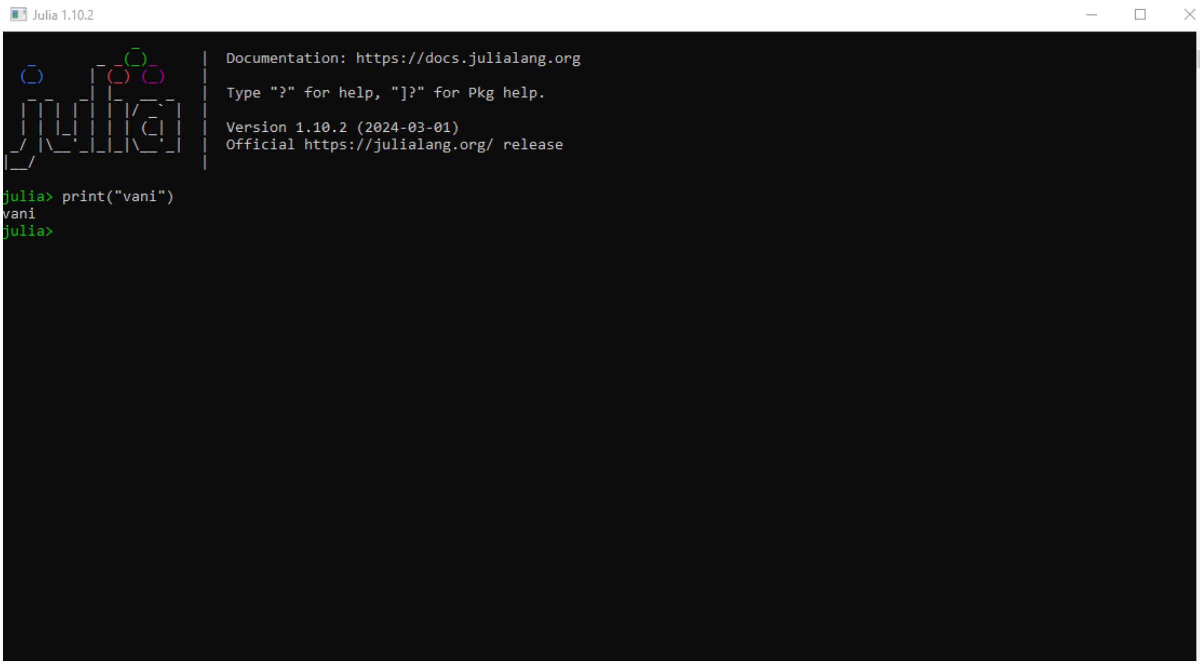
Platform	64-bit	32-bit
Windows [help]	installer, portable	installer, portable

STEP4: Now Julia start downloading, So open Julia platform



This is how Julia platform looks like

STEP5: Now we will just try “hello world” program to check whether our platform is working perfectly



Pluto.jl is a Julia programming environment designed for *learning and teaching*, and it is a great way to get started with Julia programming, packages and visualisation.

Install Pluto

```
import Pkg; Pkg.add("Pluto")
```

This will use Julia's package manager to install the Pluto package.

Run Pluto

In the Julia terminal, type:

```
import Pluto; Pluto.run()
```

Pluto will automatically open your browser when it's ready.

How do I install multiple packages in Julia?

You can use Pkg to install multiple packages. For instance: Pkg. add(["Combinatorics", "TaylorSeries"]) . If your list sits in a text file, you can just load it and pass it to Pkg.

important packages and libraries that you may want to install in Julia are:

1. **Plots: This is a plotting library that allows you to create various types of plots in Julia. You can install it using the following command:**

using Pkg

Pkg.add("Plots")

2. **LinearAlgebra: This package provides a collection of functions for linear algebra operations, such as matrix multiplication, determinants, and solving linear systems. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("LinearAlgebra")

3. **Statistics: This package provides functions for performing statistical analysis, such as calculating the mean, median, and standard deviation of a data set. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("Statistics")

4. **Optim: This package provides various optimization algorithms, such as gradient descent and Nelder-Mead, that can be used to find the minimum or maximum of a function. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("Optim")

5. **Distributions: This package provides a collection of probability distributions that can be used for various statistical applications, such as generating random numbers from a distribution or computing the probability density of a value. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("Distributions")

6. **DataFrames: This package provides a data structure for working with tabular data in Julia, similar to the DataFrame type in Python's pandas library. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("DataFrames")

7. **CSV: This package provides functions for reading and writing CSV files in Julia. You can install it using the following command:**

julia

EditFull ScreenCopy code

1using Pkg

2Pkg.add("CSV")

These are just a few examples of important packages and libraries that you may want to install in Julia. The Julia ecosystem is vast, and there are many other packages and libraries available that you can explore based on your specific needs.

using Pkg

Pkg.add("ArgParse")

HOW TO EXECUTE:

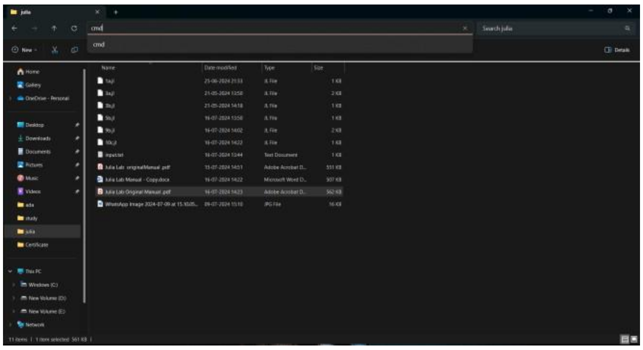
Step 1: Type the Code in Notepad:

```
function read_and_print_words(filename)
    # Open the file for reading
    file = open(filename, "r")

    # Initialize a variable to store the current word
    current_word = ""

    # Loop through each line in the file
    for line in eachline(file)
        # Loop through each character in the line
        for char in line
            if isletter(char)
                current_word *= char
            else
                if current_word != ""
                    println(current_word)
                    current_word = ""
                end
            end
        end
        # If line ends with a word, print it
        if current_word != ""
            println(current_word)
            current_word = ""
        end
    end

    close(file)
end

read_and_print_words("input.txt")
```

Step 2: Save the file with extension ".jl" . Open

cmd for the same folder. Type CMD in the

Search bar



Step 3: Type " julia <filename.jl>" to execute the program.

Experiments 1

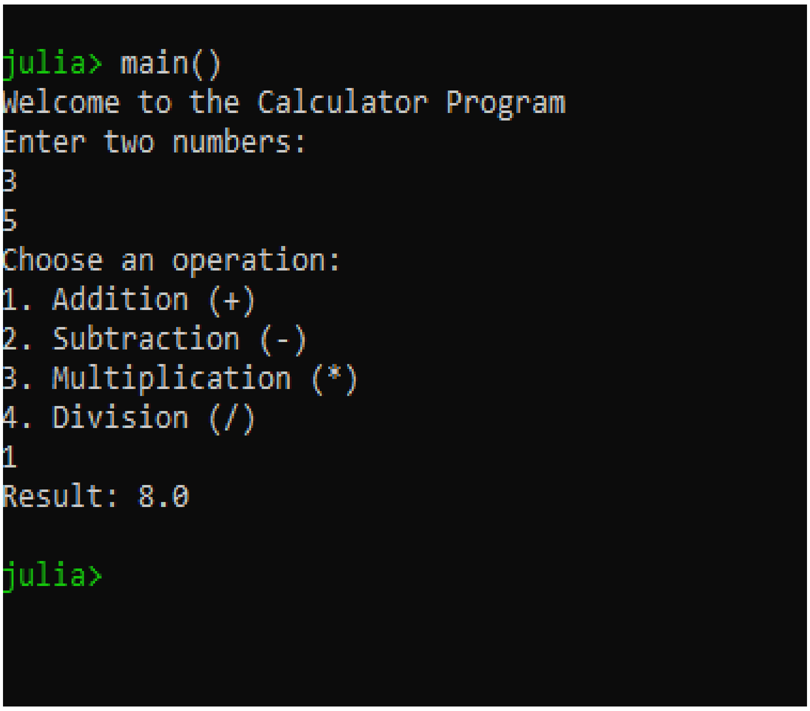
1a) Develop a Julia program to simulate a calculator (for integer and real numbers).

```
# Function to perform addition
function add(x, y)
    return x + y
end

# Function to perform subtraction
function subtract(x, y)
    return x - y
end

# Function to perform multiplication
function multiply(x, y)
```

```
    return x * y
end

# Function to perform division
function divide(x, y)
    if y == 0
        println("Error: Division by zero")
        return NaN
    else
        return x / y
    end
end

# Main function
function main()
    println("Welcome to the Calculator Program")
    println("Enter two numbers:")
    num1 = parse(Float64, readline()) # Input first number
    num2 = parse(Float64, readline())  # Input second number

    println("Choose an operation:")
    println("1. Addition (+)")
    println("2. Subtraction (-)")
    println("3. Multiplication (*)")
    println("4. Division (/)")

    operation = readline()

    if operation == "1"
        result = add(num1, num2)
        println("Result: ", result)
    elseif operation == "2"
        result = subtract(num1, num2)
        println("Result: ", result)
    elseif operation == "3"
        result = multiply(num1, num2)
        println("Result: ", result)
    elseif operation == "4"
        result = divide(num1, num2)
        println("Result: ", result)
```

```
    else
        println("Invalid operation selected")
    end
end

# Call the main function
main()
```

Output



```
julia> main()
Welcome to the Calculator Program
Enter two numbers:
3
5
Choose an operation:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
1
Result: 8.0

julia>
```

1b) Develop a Julia program to add, subtract, multiply and divide complex numbers.

```
function add_complex(c1, c2)
    return complex(c1.re + c2.re, c1.im + c2.im)
end

function subtract_complex(c1, c2)
    return complex(c1.re - c2.re, c1.im - c2.im)
```

```
end

function multiply_complex(c1, c2)
    return complex(c1.re * c2.re - c1.im * c2.im, c1.re * c2.im + c1.im *
c2.re)
end

function divide_complex(c1, c2)
    denominator = c2.re^2 + c2.im^2
    return complex((c1.re * c2.re + c1.im * c2.im) / denominator, (c1.im *
c2.re - c1.re * c2.im) / denominator)
end

# Example usage
c1 = complex(12, 8)
c2 = complex(3, 4)
println("c1 = ", c1)
println("c2 = ", c2)
println("c1 + c2 = ", add_complex(c1, c2))
println("c1 - c2 = ", subtract_complex(c1, c2))
println("c1 * c2 = ", multiply_complex(c1, c2))
println("c1 / c2 = ", divide_complex(c1, c2))
```

Output

```
julia> # Example usage

julia> c1 = complex(12, 8)
12 + 8im

julia> c2 = complex(3, 4)
3 + 4im

julia> println("c1 = ", c1)
c1 = 12 + 8im

julia> println("c2 = ", c2)
c2 = 3 + 4im

julia> println("c1 + c2 = ", add_complex(c1, c2))
c1 + c2 = 15 + 12im

julia> println("c1 - c2 = ", subtract_complex(c1, c2))
c1 - c2 = 9 + 4im

julia> println("c1 * c2 = ", multiply_complex(c1, c2))
c1 * c2 = 4 + 72im

julia> println("c1 / c2 = ", divide_complex(c1, c2))
```

1c) Develop a Julia program to evaluate expressions having mixed datatypes (integer, real, floating-point number and complex).

```
# Function to evaluate an expression
function evaluate_expression(expression)
    try
        result = eval(Meta.parse(expression))
        println("Result: $result")
    catch e
        println("Error: $e")
```

```
    end
end

# Main function
function main()
    println("Enter an expression to evaluate:")
    expression = readline()

    evaluate_expression(expression)
end

# Call the main function
main()
```

Output

```
julia>

julia> # Call the main function

julia> main()
Enter an expression to evaluate:
10.0+6*45
Result: 280.0

julia>
```

2a) Develop a Julia program for the following problem: A computer repair shop charges $100 per hour for labour plus the cost of any parts used in the repair.

However, the minimum charge for any job is $150. Prompt for the number of hours worked and the cost of parts and print the charge for the job.

```
# Function to calculate the repair charge
function calculate_charge(hours_worked, parts_cost)
    labor_cost = max(100 * hours_worked, 150)   # Ensure minimum charge of $150
    total_cost = labor_cost + parts_cost
    return total_cost
end

# Main function
function main()
    println("Welcome to the Computer Repair Shop!")
    println("Enter the number of hours worked:")
    hours_worked = parse(Float64, readline())

    println("Enter the cost of parts:")
    parts_cost = parse(Float64, readline())

    total_charge = calculate_charge(hours_worked, parts_cost)
    println("The total charge for the repair job is:", total_charge)
end

# Call the main function
main()
```

Output

```
julia> main()
Welcome to the Computer Repair Shop!
Enter the number of hours worked:
65
Enter the cost of parts:
120000
The total charge for the repair job is:126500.0

julia>
```

2b) Develop a Julia program to calculate a person's regular pay, overtime pay and gross pay bases on the following: if hours worked is less than or equal to 40,regular pay is calculated by multiplying hours worked by rate of pay, and overtime pay is 0.If hours worked is greater than 40 regular pay is calculated by multiplying 40 by the rate of pay, and overtime pay is calculated by multiplying the hours in excess of 40 by the rate of pay by 1.5.Gross pay is calculated by adding regular pay and overtime pay.

```
function calculate_pay(hours_worked, rate_of_pay)
    regular_pay = 0
    overtime_pay = 0


    if hours_worked <= 40
        regular_pay = hours_worked * rate_of_pay
    else
```

```
        regular_pay = 40 * rate_of_pay
        overtime_hours = hours_worked - 40
        overtime_pay = overtime_hours * rate_of_pay * 1.5
    end

    gross_pay = regular_pay + overtime_pay

    return regular_pay, overtime_pay, gross_pay
end

# Example usage
hours_worked = 45
rate_of_pay = 10

regular_pay, overtime_pay, gross_pay = calculate_pay(hours_worked, rate_of_pay)

println("Regular Pay: ", regular_pay)
println("Overtime Pay: ", overtime_pay)
println("Gross Pay: ", gross_pay)
```

Output

```
julia> regular_pay, overtime_pay, gross_pay = calculate_pay(hours_worked, rate_of_pay)
(400, 75.0, 475.0)

julia>

julia> println("Regular Pay: ", regular_pay)
Regular Pay: 400

julia> println("Overtime Pay: ", overtime_pay)
Overtime Pay: 75.0

julia> println("Gross Pay: ", gross_pay)
Gross Pay: 475.0

julia>
```

3a) An amount of money P(for principal) is put into an account which earns interest at r% per annum. So, at the end of one year, the amount becomes P+P*/100. This becomes the principal for the next year. Develop a Julia program to print the amount at the end of each year for the next 10 years. However, if the amount ever exceeds 2P, stop any further printing. Your program should prompt for the values of P and r.

```
using ArgParse

function parse_commandline()
    s = ArgParseSettings()
```

```
    @add_arg_table s begin
        "P"
            arg_type = Float64
            required = true
            help = "The principal amount."
        "r"
            arg_type = Float64
            required = true
            help = "The interest rate percentage."
    end

    return parse_args(s)
end

function print_amounts(P, r)
    println("Year 10 amount: ", P)
    for year in 2:10
        P = P + P * r / 100
        println("Year $year amount: ", P)
        if P > 2 * parse(Float64, get(ARGS, "P", "")), "")
            println("Amount exceeded 2P, stopping further printing.")
            break
        end
    end
end

function main()
```

```
    parsed_args = parse_commandline()
    P = parse(Float64, get(ARGS, "P", ""), "")
    r = parse(Float64, get(ARGS, "r", ""), "")

    if isempty(P) || isempty(r)
        println("Enter the principal amount (P): ")
        P = parse(Float64, readline())
        println("Enter the interest rate (r) in percentage: ")
        r = parse(Float64, readline())
    end

    print_amounts(P, r)
end

main()
```

```
D:\study\julia>julia 3a.jl
Enter the principal amount (P):
100000
Enter the interest rate (r) in percentage:
13
Year 1 amount: 100000.0
Year 2 amount: 113000.0
Year 3 amount: 127690.0
Year 4 amount: 144289.7
Year 5 amount: 163047.361
Year 6 amount: 184243.51793
Year 7 amount: 208195.1752609
Amount exceeded 2P, stopping further printing.

D:\study\julia>S
```

3b)Develop a Julia program which reads numbers from a file (input.txt) and finds the largest number, smallest number, count, sum and average of numbers.

```
function read_numbers_from_file(filename)
    numbers = Float64[]

    # Read numbers from file
    open(filename, "r") do file
```

```
for line in eachline(file)
    push!(numbers, parse(Float64, line))
```

```
    end
    end

    return numbers
end

function calculate_statistics(numbers)
    if isempty(numbers)
        println("No numbers found in the file.")
        return
    end

    max_number = maximum(numbers)
    min_number = minimum(numbers)
    count = length(numbers)
    total_sum = sum(numbers)
    average = total_sum / count

    println("Largest Number: ", max_number)
    println("Smallest Number: ", min_number)
    println("Count: ", count)
    println("Sum: ", total_sum)
    println("Average: ", average)
end

# Main program
filename = C:\Users\LENOVO\Desktop # Change this to the path of your file if it's located elsewhere
```

```
numbers = read_numbers_from_file(“input.txt”)
calculate_statistics(numbers)


function read_numbers(input.txt)
    numbers = Int[]
    open(filename) do file
        for line in eachline(file)
            push!(numbers, parse(Int, line))
        end
    end
    return numbers
end
```

```
D:\study\julia>julia 3b.jl
Largest Number: 4.3432111e7
Smallest Number: 1.0
Count: 13
Sum: 4.4224071e7
Average: 3.4018516153846155e6

D:\study\julia>
```

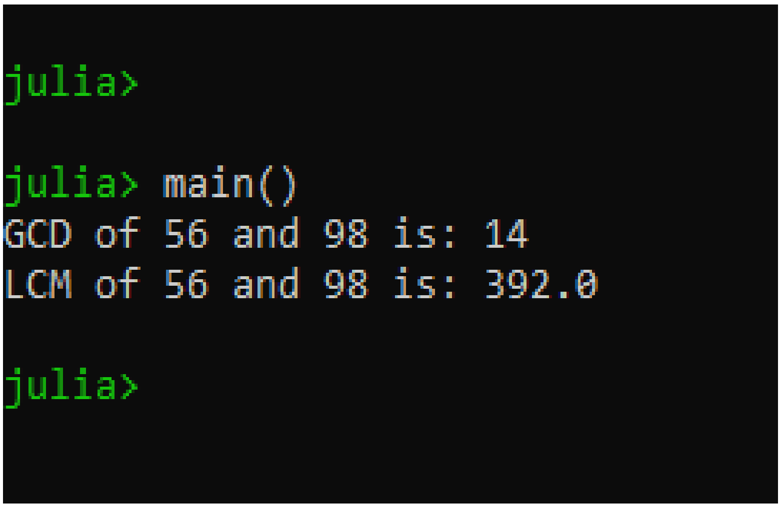
4a)Develop a Julia program and two separate functions to calculate GCD and LCM.

```
function gcd(a, b)
    while b != 0
        a, b = b, a % b
    end
    return abs(a)
end
```

```
function lcm(a, b)
    return a * b / gcd(a, b)
end

function main()
```

```
    a = 56
    b = 98
    println("GCD of $(a) and $(b) is: ", gcd(a, b))
    println("LCM of $(a) and $(b) is: ", lcm(a, b))
end

main()
```

OUTPUT

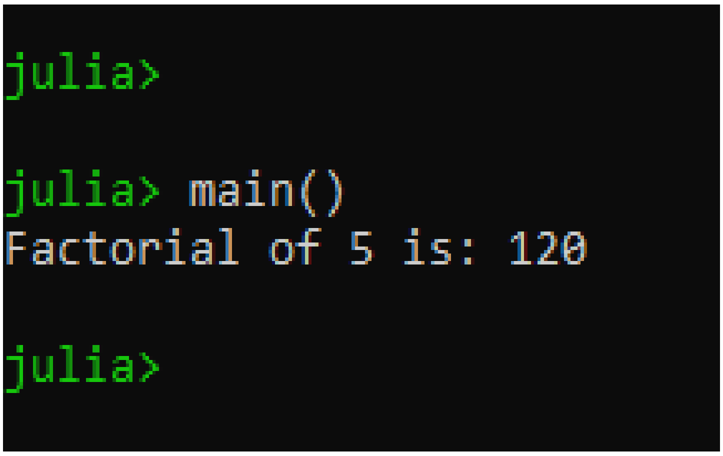


4b)Develop a Julia program and a recursive function to calculate factorial of a number.

```
function factorial(n)
    if n == 0
        return 1
    else
        return n * factorial(n - 1)
    end
end
```

```
function main()
    n = 5
    println("Factorial of $(n) is: ", factorial(n))
end

main()
```

OUTPUT

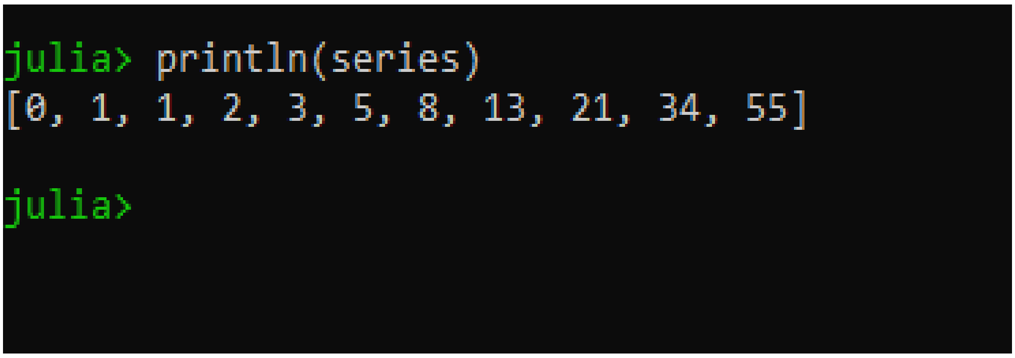


4C)Develop a Julia program and a recursive function to generate Fibonacci Series.

```
function fibonacci(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fibonacci(n - 1) + fibonacci(n - 2)
    end
end
```

```
function generate_fibonacci_series(n)
    series = [fibonacci(i) for i in 0:n]
    return series
end

n = 10
series = generate_fibonacci_series(n)
println("Fibonacci series of length $(n+1):")
println(series)
```

OUTPUT

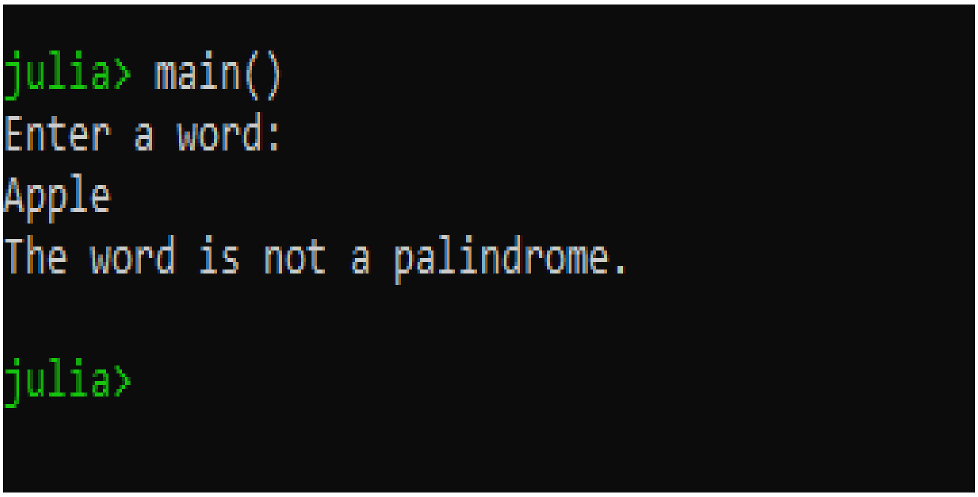


5a)Develop a Julia program which reads a string (word) and prints whether the word is palindrome.

```
function is_palindrome(word)
    n = length(word)
    for i in 1:n÷2
        if word[i] != word[n-i+1]
            return false
        end
    end
```

```
    return true
end

function main()
    println("Enter a word:")
    word = readline()
    if is_palindrome(word)
        println("The word is a palindrome.")
    else
        println("The word is not a palindrome.")
    end
end

main()
```

OUTPUT



5b)Develop a Julia program which reads and prints the words present in a file(input.txt) having Random Data in which words are dispersed randomly(Assumption: a word is a contiguous sequence of letters. A word is delimited by any non-letter character or end-of-line)

```
function
read_and_print_words(filename)
    # Open the file for reading
    file = open(filename, "r")

    # Initialize a variable to store the
current word
    current_word = ""

    # Loop through each line in the
file
    for line in eachline(file)
```

```
        # Loop through each character in the line
        for char in line
            if isletter(char)
                current_word *= char
            else
                if current_word != ""
                    println(current_word)
                    current_word = ""
                end
            end
        end
        # If line ends with a word, print it
        if current_word != ""
            println(current_word)
            current_word = ""
        end
    end

    close(file)
end

read_and_print_words("input.txt")
```

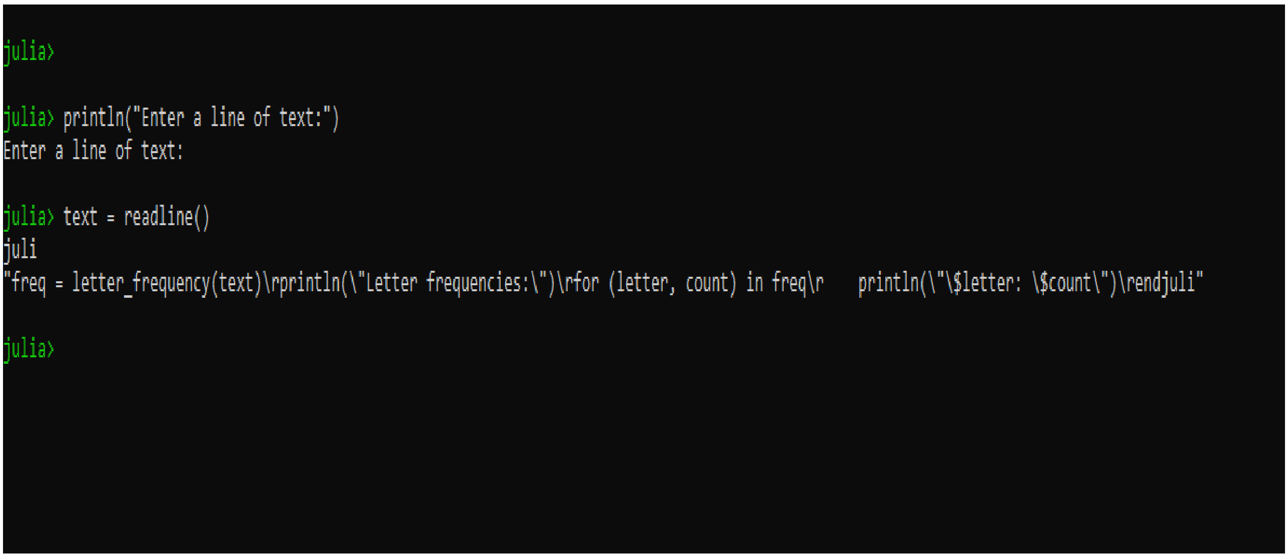
```
julia> include("5B.jl")
hello
world
goodbye
world
```

6a)Develop a Julia program to determine and print the frequency with which each letter of the alphabet is used in a given line of text.

```
function letter_frequency(text)
    freq = Dict{Char, Int}()
    for c in text
        if isletter(c)
            lc = lowercase(c)
            if haskey(freq, lc)
                freq[lc] += 1
            else
```

```
                freq[lc] = 1
            end
        end
    end
    return freq
end

println("Enter a line of text:")
text = readline()
freq  =  letter_frequency(text)
println("Letter  frequencies:")
for (letter, count) in freq
    println("$letter:  $count")
end
```

OUTPUT



```
julia>

julia> println("Enter a line of text:")
Enter a line of text:

julia> text = readline()
juli
"freq = letter_frequency(text)\rprintln(\"Letter frequencies:\")\rfor (letter, count) in freq\r    println(\"\$letter: \$count\")\rendjuli"

julia>
```

6b) A survey of 10 pop artists is made. Each person votes for an artist by specifying the number of the artist(a value from 1 to 10). Develop a julia program to read the names of the artists,followed by the votes, and find out which artist is the most popular.

```
function read_votes(n)
    votes = Dict{String, Int}()
    for i in 1:n
        print("Enter the name of artist $i: ")
        name = readline()
        print("Enter the number of votes for artist $i: ")
        votes[name] = parse(Int, readline())
    end
    return votes
end

function find_most_popular(votes)
    max_votes = 0
    most_popular = ""
    for (name, votes) in votes
        if votes > max_votes
            max_votes = votes
            most_popular = name
        end
    end
    return most_popular
end

n = 10
votes = read_votes(n)
```

most_popular = find_most_popular(votes)

println("The most popular artist is $most_popular")

OUTPUT

```
Enter the name of artist 1: jim
Enter the number of votes for artist 1: 10
Enter the name of artist 2: joy
Enter the number of votes for artist 2: 9
Enter the name of artist 3: lim
Enter the number of votes for artist 3: 6
Enter the name of artist 4: ray
Enter the number of votes for artist 4: 6
Enter the name of artist 5: joey
Enter the number of votes for artist 5: 8
Enter the name of artist 6: rayo
Enter the number of votes for artist 6: 9
Enter the name of artist 7: hon
Enter the number of votes for artist 7: 8
Enter the name of artist 8: joyer
Enter the number of votes for artist 8: 6
Enter the name of artist 9: doey
Enter the number of votes for artist 9: 5
Enter the name of artist 10: weeve
Enter the number of votes for artist 10: 7
Dict{String, Int64} with 10 entries:
  "most_popular = find_most_popular(votes)\rprintln(\"The most popular artist is \$most_popular\")jim" => 10
  "doey"                                                                                              => 5
  "lim"                                                                                               => 6
  "rayo"                                                                                              => 9
  "joyer"                                                                                             => 6
  "joey"                                                                                              => 8
  "hon"                                                                                               => 8
  "joy"                                                                                               => 9
  "ray"                                                                                               => 6
  "weeve"                                                                                             => 7
```

7a) Given a line of text as input, develop a Julia program to determine the frequency with which each letter of the alphabet is used(make use of dictionary).

```
function letter_frequency(text)
    freq = Dict{Char, Int}()
    for char in text
```

```
        if isletter(char)
            if haskey(freq, char)
                freq[char] += 1
            else
                freq[char] = 1
            end
        end
    end
    return freq
end

println("Enter a line of text:")
text = readline()
freq = letter_frequency(text)
println("Letter frequencies:")
for (letter, count) in freq
    println("$letter: $(count)")
end
```

OUTPUT

```
julia> include("7a.jl")
Enter a line of text:
hello world
Letter frequencies:
w: 1
h: 1
d: 1
l: 3
e: 1
r: 1
o: 2
```

7b) Develop a Julia program to fetch words from a file with arbitrary punctuation and keep track of all the different words found(make use of set and ignore the case of the letters: eg to and To are treated as the same word).

```
function fetch_words(file_path::AbstractString)
    # Initialize an empty set to store unique words
    unique_words = Set{String}()

    # Open the file for reading
    open(file_path) do file
        # Read each line from the file
        for line in eachline(file)
            # Remove punctuation and split the line into words
            words = split(replace(lowercase(line), r"[[:punct:]]" => ""), " ")

            # Add each word to the set
```

```
            for word in words
                push!(unique_words, word)
            end
        end
    end

    return unique_words
end

# Example usage
file_path = " C:\Users\LENOVO\Desktop" # Replace with your file path
words_set = fetch_words(file_path)
println("Unique words found in the file:")
for word in words_set
    println(word)
end
```

```
julia> include("7b.jl")
Unique words found in file:
goodbye
hello
world
```

8a) Develop a Julia program to evaluate expressions consisting of rational,irrational number and floating point numbers.

```
function eval_expr(expr)
    # Parse the expression using Julia's built-in parser
    ast = Meta.parse(expr)
```

```
# Evaluate the expression using Julia's built-in evaluator
result = eval(ast)
```

```
    # Return the result as a Float64
    return Float64(result)
end

# Test cases
println(eval_expr("2 + 3"))
println(eval_expr("sqrt(2) * pi"))
println(eval_expr("exp(1) / 2"))
println(eval_expr("sin(pi / 4)"))
println(eval_expr("cos(pi / 3)"))
println(eval_expr("1 / 3 + 2 / 5"))
println(eval_expr("(1 + sqrt(2)) * (1 - sqrt(2))"))
```

OUTPUT

```
julia> println(eval_expr("2 + 3"))
5.0

julia> println(eval_expr("sqrt(2) * pi"))
4.442882938158366

julia> println(eval_expr("exp(1) / 2"))
1.3591409142295225

julia> println(eval_expr("sin(pi / 4)"))
0.7071067811865475

julia> println(eval_expr("cos(pi / 3)"))
0.5000000000000001

julia> println(eval_expr("1 / 3 + 2 / 5"))
0.7333333333333334

julia> println(eval_expr("(1 + sqrt(2)) * (1 - sqrt(2))"))
-1.0000000000000002

julia>
```

8b) Develop a Julia program to determine the following properties of a matrix: determinant,inverse,rank,upper& lower triangular matrix, diagonal elements,Euclidean norm and square root of a matrix..

```
using LinearAlgebra

function matrix_properties(A)
    n = size(A, 1)

    # Determinant
    detA = det(A)

    # Inverse
    invA = inv(A)

    # Rank
    rankA = rank(A)

    # Upper and Lower Triangular Matrices
    U, L = triu(A)

    # Diagonal Elements
    diagA = diag(A)

    # Euclidean Norm
    normA = norm(A)
```

```
    # Square Root
    sqrtA = sqrt(A)

    return detA, invA, rankA, U, L, diagA, normA, sqrtA
end

A = [3.0 2.0 -1.0; 1.0 0.0 2.0; 0.0 -1.0 1.0]

detA, invA, rankA, U, L, diagA, normA, sqrtA = matrix_properties(A)

println("Determinant: ", detA)
println("Inverse: ", invA)
println("Rank: ", rankA)
println("Upper Triangular Matrix: ", U)
println("Lower Triangular Matrix: ", L)
println("Diagonal Elements: ", diagA)
println("Euclidean Norm: ", normA)
println("Square Root: ", sqrtA)
```

OUTPUT

```
julia>

julia> A = [3.0 2.0 -1.0; 1.0 0.0 2.0; 0.0 -1.0 1.0]
3×3 Matrix{Float64}:
 3.0   2.0  -1.0
 1.0   0.0   2.0
 0.0  -1.0   1.0

julia>

julia> detA, invA, rankA, U, L, diagA, normA, sqrtA = matrix_properties(A)
```

9a) Develop a Julia program to determine addition and substraction of two matrices(element-wise).

```
function elementwise_add(A, B)
    n, m = size(A)
    C = similar(A)
    for i in 1:n
        for j in 1:m
            C[i, j] = A[i, j] + B[i, j]
        end
    end
    return C
end

function elementwise_sub(A, B)
    n, m = size(A)
    C = similar(A)
    for i in 1:n
        for j in 1:m
            C[i, j] = A[i, j] - B[i, j]
        end
    end
    return C
end

A = [1.0 2.0; 3.0 4.0]
B = [5.0 6.0; 7.0 8.0]
```

```
C = elementwise_add(A, B)
println("Element-wise addition:")
println(C)

D = elementwise_sub(A, B)
println("Element-wise subtraction:")
println(D)
```

OUTPUT

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> B = [5.0 6.0; 7.0 8.0]
2×2 Matrix{Float64}:
 5.0  6.0
 7.0  8.0

julia>

julia> C = elementwise_add(A, B)
2×2 Matrix{Float64}:
  6.0   8.0
 10.0  12.0

julia> println("Element-wise addition:")
Element-wise addition:

julia> println(C)
[6.0 8.0; 10.0 12.0]

julia>

julia> D = elementwise_sub(A, B)
2×2 Matrix{Float64}:
 -4.0  -4.0
 -4.0  -4.0

julia> println("Element-wise subtraction:")
Element-wise subtraction:

julia> println(D)
[-4.0 -4.0; -4.0 -4.0]

julia>
```

9b) Develop a Julia program to perform multiplication operation on matrices: Scalar multiplication, element-wise multiplication, dot product, cross product.

```
function scalar_multiplication(matrix::Matrix{T}, scalar::T) where T
    return matrix * scalar
end


function element_wise_multiplication(matrix1::Matrix{T}, matrix2::Matrix{T}) where T
```

```
    return matrix1 .* matrix2
end

function dot_product(vector1::Vector{T}, vector2::Vector{T}) where T
    if length(vector1) != length(vector2)
        error("Vectors must have the same length for dot product operation.")
    end
    return dot(vector1, vector2)
end

function cross_product(vector1::Vector{T}, vector2::Vector{T}) where T
    if length(vector1) != 3 || length(vector2) != 3
        error("Cross product is only defined for 3-dimensional vectors.")
    end
    return cross(vector1, vector2)
end

# Example usage
A = [1 2; 3 4]
B = [5 6; 7 8]

println("Scalar Multiplication:")
scalar = 2
println("A * $scalar:")
println(scalar_multiplication(A, scalar))
```

```
println("\nElement-wise Multiplication:")
println("A .* B:")
println(element_wise_multiplication(A, B))

println("\nDot Product:")
v1 = [1, 2, 3]
v2 = [4, 5, 6]
println("Dot Product of $v1 and $v2:")
println(dot_product(v1, v2))

println("\nCross Product:")
v3 = [1, 2, 3]
v4 = [4, 5, 6]
println("Cross Product of $v3 and $v4:")
println(cross_product(v3, v4))
```

OUTPUT

```
julia> scalar = 2
2

julia> println("A * $scalar:")
A * 2:

julia> println(scalar_multiplication(A, scalar))
[2 4; 6 8]

julia>

julia> println("\nElement-wise Multiplication:")

Element-wise Multiplication:

julia> println("A .* B:")
A .* B:

julia> println(element_wise_multiplication(A, B))
[5 12; 21 32]

julia>

julia> println("\nDot Product:")

Dot Product:

julia> v1 = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> v2 = [4, 5, 6]
3-element Vector{Int64}:
 4
 5
 6

julia> println("Dot Product of $v1 and $v2:")
Dot Product of [1, 2, 3] and [4, 5, 6]:

julia> println(dot_product(v1, v2))
```

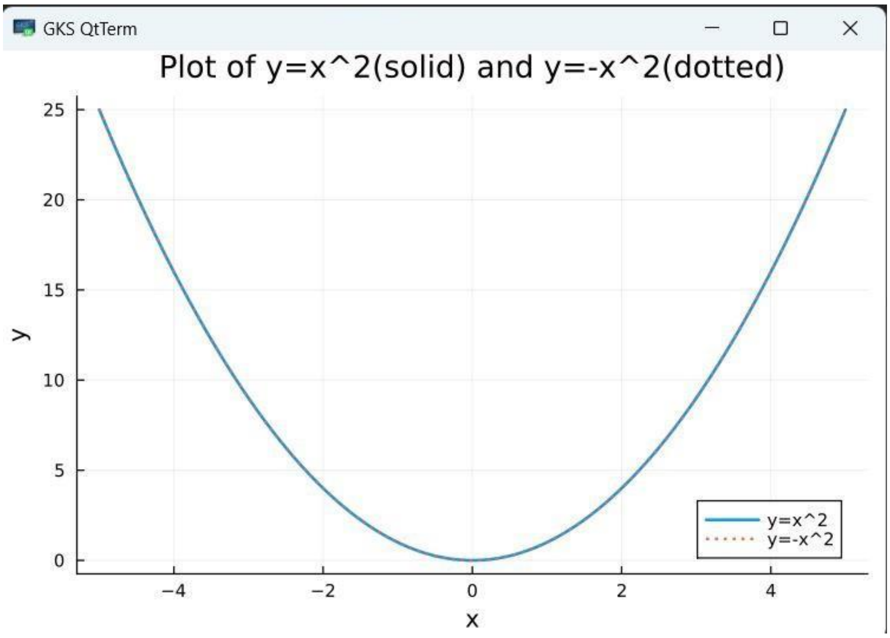
10a) Develop a Julia program to generate a plot of (solid & dotted) a function:

y= x^2(use suitable data points for x)

using Plots

```
# Generate data points for x in the range -5 to 5

x = range(-5, 5, length=100)



# Calculate corresponding y values for the function y = x²

y1 = x.^2



# Calculate corresponding y values for the function y = -x²

y2 = -x.^2



# Create a plot

plt = plot(x, y1, label="y = x²", linestyle=:solid, linewidth=2, xlabel="x",

ylabel="y", title="Plot of y = x² (solid) and y = -x² (dotted)")



# Add a dotted line to the plot

plot!(x, y2, label="y = -x²", linestyle=:dot, linewidth=2)



# Show the plot

display(plt)
```

OUTPUT



10b) Develop a Julia program to generate a plot of mathematical equation:

y=sin(x)+sin(2x).

using Plots

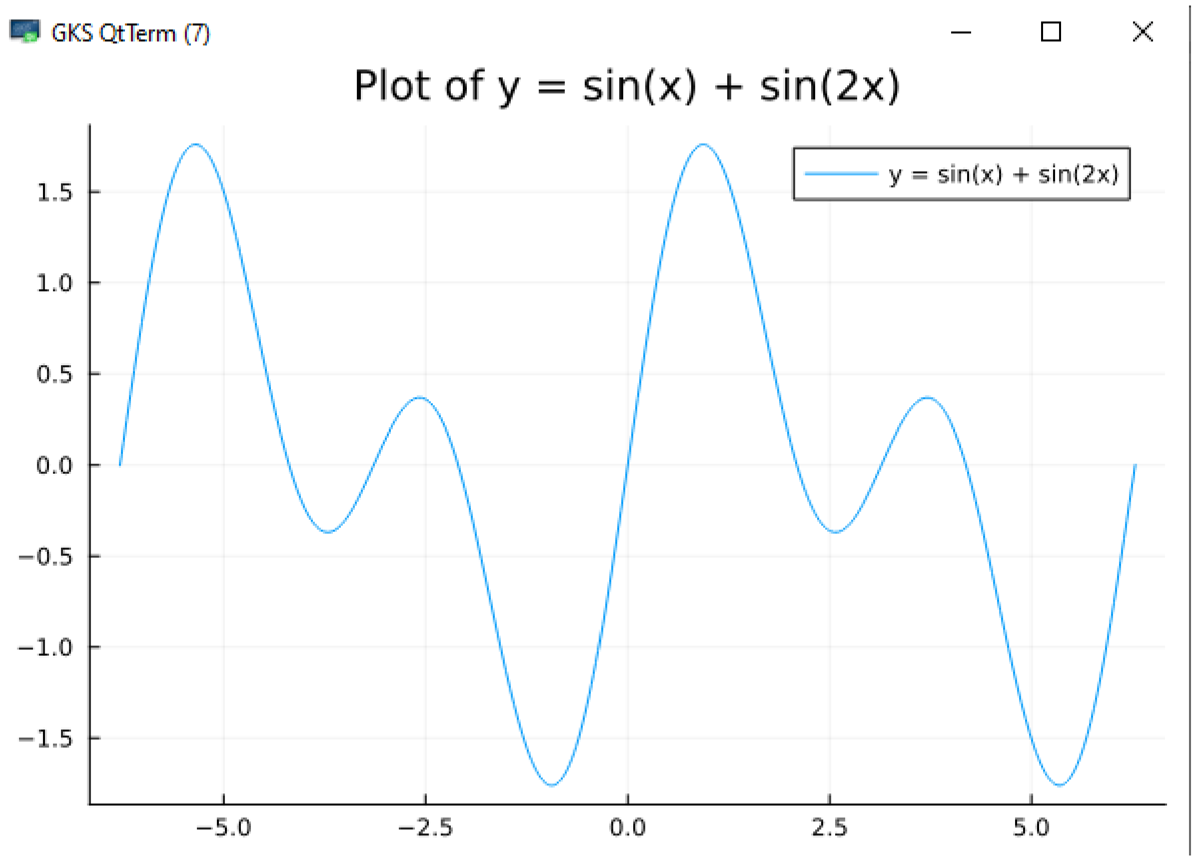
Define the function y = sin(x) + sin(2x)

f(x) = sin(x) + sin(2x)

Generate the plot

```
plot(f, -2π, 2π, label="y = sin(x) + sin(2x)", title="Plot of y = sin(x) + sin(2x)")
```

OUTPUT

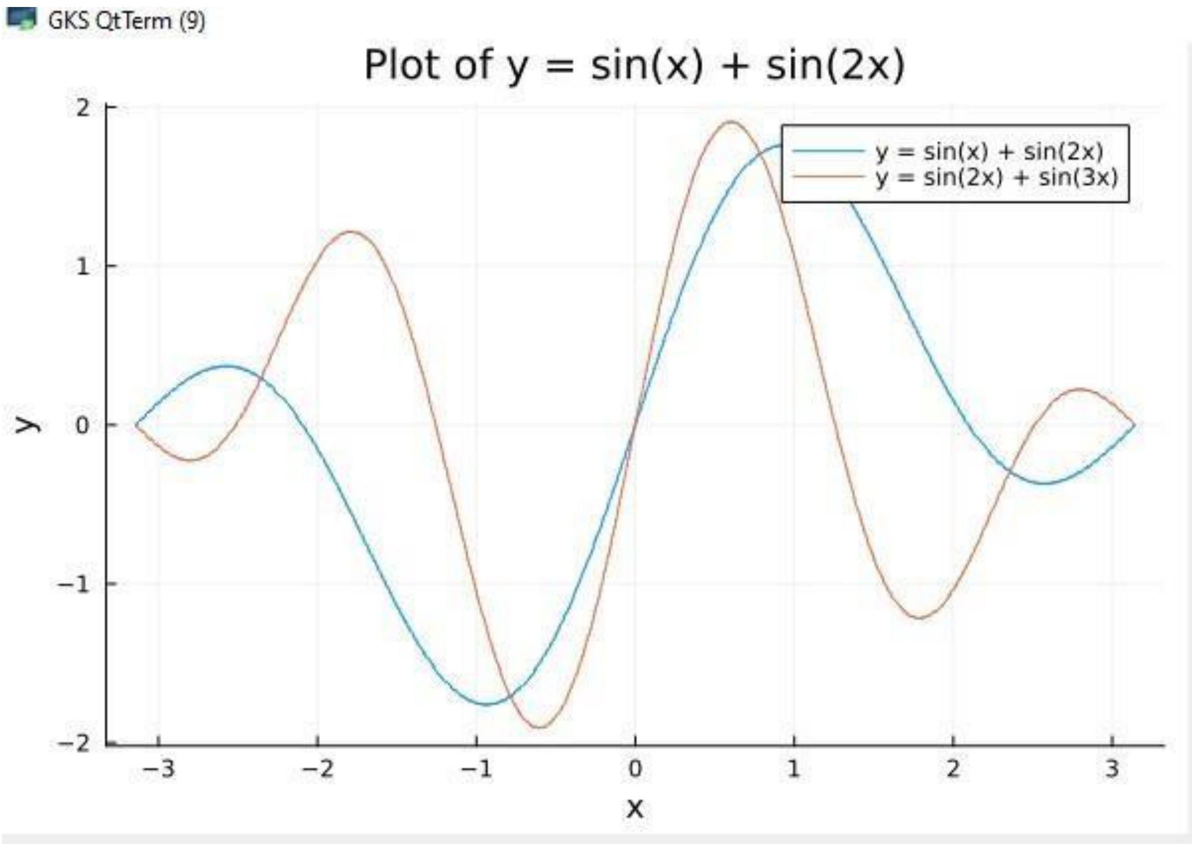


10c)Develop a Julia program to generate multiple plots of mathematical

equations: y=sin(x)+sin(2x) and y=sin(2x)+sin(3x)

```
using Plots
x_values1 = -pi:0.01:pi
y11(x) = sin(x) + sin(2x)
y12(x) = sin(2x) + sin(3x)
y_values1 = y11.(x_values1)
y_values2 = y12.(x_values1)
plot(x_values1, y_values1, label="y = sin(x) + sin(2x)", xlabel="x", ylabel="y", title="Plot of y = sin(x) + sin(2x)")
plot!(x_values1, y_values2, label="y = sin(2x) + sin(3x)")
```

OUTPUT


GKS QtTerm (9)
Plot of y = sin(x) + sin(2x)
y = sin(x) + sin(2x)
y = sin(2x) + sin(3x)
y
x
2
1
0
−1
−2
−3
−2
−1
0
1
2
3