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

```
# Define function for addition
function add(x, y)
    return x + y
end

# Define function for subtraction
function subtract(x, y)
    return x - y
end

# Define function for multiplication
function multiply(x, y)
    return x * y
end

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

# Main function to perform calculator operations
function calculator()
    println("Welcome to the calculator simulation!")
    println("Please select an operation:")
    println("1. Addition (+)")
    println("2. Subtraction (-)")
    println("3. Multiplication (*)")
    println("4. Division (/)")
    operation = readline()

    println("Enter first number:")
    num1 = parse(Float64, readline())

    println("Enter second number:")

    num2 = parse(Float64, readline())

    if operation == "+" || operation == "1"
```

```
        result = add(num1, num2)
        println("Result: $result")
    elseif operation == "-" || operation == "2"
        result = subtract(num1, num2)
        println("Result: $result")
    elseif operation == "*" || operation == "3"
        result = multiply(num1, num2)
        println("Result: $result")
    elseif operation == "/" || operation == "4"
        result = divide(num1, num2)
        println("Result: $result")
    else
        println("Invalid operation selected!")
    end
end
calculator()
```

Output

```
Welcome to the calculator simulation!
Please select an operation:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
stdin>  1
Enter first number:
stdin>  10
Enter second number:
stdin>  20
Result: 30.0
```

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

```
function complex_add(z1::Complex, z2::Complex)
    return z1 + z2
end

# Define function for complex number subtraction
function complex_subtract(z1::Complex, z2::Complex)
    return z1 - z2
end

# Define function for complex number multiplication
function complex_multiply(z1::Complex, z2::Complex)
    return z1 * z2
end

# Define function for complex number division
function complex_divide(z1::Complex, z2::Complex)
    return z1 / z2
end

# Main function to perform complex number operations
function complex_calculator()
    println("Welcome to the complex number calculator!")
    println("Please select an operation:")
    println("1. Addition (+)")
    println("2. Subtraction (-)")
    println("3. Multiplication (*)")
    println("4. Division (/)")
    operation = readline()

    println("Enter the real part of the first complex number:")
    real1 = parse(Float64, readline())

    println("Enter the imaginary part of the first complex number:")
    imag1 = parse(Float64, readline())

    println("Enter the real part of the second complex number:")
    real2 = parse(Float64, readline())

    println("Enter the imaginary part of the second complex number:")
    imag2 = parse(Float64, readline())

    z1 = complex(real1, imag1)
    z2 = complex(real2, imag2)
```

```
    if operation == "+" || operation == "1"
        result = complex_add(z1, z2)
        println("Result: $result")
    elseif operation == "-" || operation == "2"
        result = complex_subtract(z1, z2)
        println("Result: $result")
    elseif operation == "*" || operation == "3"
        result = complex_multiply(z1, z2)
        println("Result: $result")
    elseif operation == "/" || operation == "4"
        result = complex_divide(z1, z2)

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

Output

```
Welcome to the complex number calculator!
Please select an operation:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
stdin>  1
Enter the real part of the first complex number:
stdin>  2
Enter the imaginary part of the first complex number:
stdin>  2
Enter the real part of the second complex number:
stdin>  -1
Enter the imaginary part of the second complex number:
stdin>  4
Result: 1.0 + 6.0im
```

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

```
function evaluate_expression(expr::String)
    try
        # Use Meta.parse to parse the input expression
        parsed_expr = Meta.parse(expr)

        # Evaluate the parsed expression
        result = eval(parsed_expr)

        return result
    catch e
        println("Error: $e")
        return nothing
    end
end

function main()
    println("Welcome to the mixed data type expression evaluator!")
    println("Please enter the expression to evaluate:")
    expr = readline()

    result = evaluate_expression(expr)
    if result !== nothing
        println("Result: $result")
    end
end

# Main entry point of the program
main()
```

Output

```
Welcome to the mixed data type expression evaluator!
Please enter the expression to evaluate:
stdin>  6+5.6
Result: 11.6
```

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 (which could be $0) and print the charge for the job.

```
# Function to calculate the total repair cost
function calculate_repair_cost(hours_worked::Float64, parts_cost::Float64)
        # Constants
        labor_rate = 100.0  # Labor cost per hour
        minimum_charge = 150.0  # Minimum charge for any job

        # Calculate labor cost
        labor_cost = hours_worked * labor_rate

        # Calculate total cost
        total_cost = labor_cost + parts_cost

        # Apply minimum charge rule
        if total_cost < minimum_charge
        total_cost = minimum_charge
        end

        return total_cost
end

# Main script
function main()
        println("Enter the number of hours worked:")
        hours = parse(Float64, readline())

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

        # Calculate the charge
        total_charge = calculate_repair_cost(hours, parts_cost)

        # Print the total charge
        println("The total charge for the job is \$$total_charge")
end

# Run the program
main()
```

Output

```
Enter the number of hours worked:
stdin>  10
Enter the cost of parts used in the repair:
stdin>  500
The total charge for the job is $1500.0
```

2b.Develop a Julia program to calculate a person's regular pay, overtime pay and gross pay based 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 to calculate pay details
function calculate_pay(hours_worked::Float64, hourly_rate::Float64)
        # Initialize pay variables
        regular_pay, overtime_pay = 0.0, 0.0

        # Check hours and calculate pay
        if hours_worked <= 40
        # All hours are regular
        regular_pay = hours_worked * hourly_rate
        else
        # First 40 hours are regular
        regular_pay = 40 * hourly_rate
        # Overtime hours
        overtime_hours = hours_worked - 40
        overtime_pay = overtime_hours * hourly_rate * 1.5
        end

        # Calculate gross pay
        gross_pay = regular_pay + overtime_pay

        return regular_pay, overtime_pay, gross_pay
end

# Main function to interact with the user
function main()
        println("Enter the number of hours worked:")
        hours = parse(Float64, readline())

        println("Enter the hourly rate of pay:")
        rate = parse(Float64, readline())
```

```
        # Calculate the pay details
        regular_pay, overtime_pay, gross_pay = calculate_pay(hours, rate)

        # Display the results
        println("Regular Pay: \$$(round(regular_pay, digits=2))")
        println("Overtime Pay: \$$(round(overtime_pay, digits=2))")
        println("Gross Pay: \$$(round(gross_pay, digits=2))")
end

# Execute the main function
main()
```

Output

```
Enter the number of hours worked:
stdin>  30
Enter the hourly rate of pay:
stdin>  20
Regular Pay: $600.0
Overtime Pay: $0.0
Gross Pay: $600.0
```

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×r/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, ifthe amount ever exceeds 2P, stop any further printing. Your program should prompt for the values of Pand r.

```
Define a function to calculate and print the amount at the end of each year
function calculate_amount(P, r)
        # Print the amount at the end of the first year
        println("Year 1: ", P + P*r/100)
        # Set the principal for the next year
        principal = P + P*r/100
        # Iterate over the next 9 years
        for year in 2:10
        # Calculate the amount at the end of the current year
        principal = principal + principal*r/100
        # Print the amount at the end of the current year
        println("Year $year: ", principal)
        # Check if the amount exceeds 2P
        if principal > 2*P
        # If the amount exceeds 2P, print a message and break out of the loop
        println("The amount exceeds 2P. Stopping further calculations.")
        break
        end
        end
end

# Prompt the user to enter the principal amount (P)
println("Enter the principal amount (P): ")
# Read the input from the user and parse it as a Float64
P = parse(Float64, readline())

# Prompt the user to enter the interest rate (r) in percentage
println("Enter the interest rate (r) in percentage: ")
# Read the input from the user and parse it as a Float64
r = parse(Float64, readline())

# Call the calculate_amount function with the user-provided values of P and r
calculate_amount(P, r)
```

Output

```
Enter the principal amount (P):
stdin>  1000
Enter the interest rate (r) in percentage:
stdin>  5
Year 1: 1050.0
Year 2: 1102.5
Year 3: 1157.625
Year 4: 1215.50625
Year 5: 1276.2815624999998
Year 6: 1340.0956406249998
Year 7: 1407.1004226562497
Year 8: 1477.4554437890622
Year 9: 1551.3282159785153
Year 10: 1628.894626777441
```

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.

```
# Define a function to calculate statistics from numbers in a file
function calculate_statistics(filename)
        # Open the file for reading
        file = open(filename, "r")

        # Initialize variables to store statistics
        largest = typemin(Float64)  # Initialize largest number as the smallest possible
Float64 value
        smallest = typemax(Float64) # Initialize smallest number as the largest possible
Float64 value
        count = 0               # Initialize count of numbers to 0
        total = 0.0             # Initialize sum of numbers to 0.0 (Float64)

        # Read numbers from the file line by line
        for line in eachline(file)
        # Parse each line as a Float64
        number = parse(Float64, line)

        # Update largest and smallest numbers if necessary
        if number > largest
        largest = number
        end
        if number < smallest
        smallest = number
        end
```

```
        # Update count and total sum
        count += 1
        total += number
        end

        # Close the file
        close(file)

        # Calculate the average
        average = total / count

        # Print the results
        println("Largest number: ", largest)
        println("Smallest number: ", smallest)
        println("Count: ", count)
        println("Sum: ", total)
        println("Average: ", average)
end

# Call the function with the filename "input.txt"
calculate_statistics("input.txt")
```

Input.txt

```
10
20
30
40
2
7
90
```

Output

```
Largest number: 90.0
Smallest number: 2.0
Count: 7
Sum: 199.0
Average: 28.428571428571427
```

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

```
# Function to calculate the Greatest Common Divisor (GCD) using Euclid's algorithm
function gcd(a, b)
        while b != 0
        a, b = b, a % b
        end
        return a
end

# Function to calculate the Least Common Multiple (LCM)
function lcm(a, b)
        return abs(a * b) ÷ gcd(a, b)  # LCM = (|a * b|) / GCD(a, b)
end

# Main program
println("Enter two numbers separated by space: ")
input = readline()
a, b = parse.(Int, split(input))

gcd_result = gcd(a, b)
lcm_result = lcm(a, b)

println("GCD of $a and $b is: ", gcd_result)
println("LCM of $a and $b is: ", lcm_result)
```

OUTPUT

```
Enter two numbers separated by space:
stdin>  18 24
GCD of 18 and 24 is: 6
LCM of 18 and 24 is: 72
```

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

```
# Recursive function to calculate factorial
function factorial_recursive(n)
        if n == 0 || n == 1
        return 1
        else
        return n * factorial_recursive(n - 1)
        end
```

```
end

# Main program
println("Enter a number to calculate its factorial: ")
num = parse(Int, readline())

if num < 0
        println("Factorial is not defined for negative numbers.")
else
        result = factorial_recursive(num)
        println("Factorial of $num is: ", result)
end
```

Output:

```
Enter a number to calculate its factorial:
stdin>  5
Factorial of 5 is: 120
```

4c. Develop a Julia program and a recursive function to generate Fibonacci series.

```
# Recursive function to generate Fibonacci series
function fibonacci_recursive(n)
        if n == 0
        return 0
        elseif n == 1
        return 1
        else
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
        end
end

# Main program to print Fibonacci series
println("Enter the number of terms for Fibonacci series: ")
num_terms = parse(Int, readline())

if num_terms < 1
        println("Number of terms should be greater than 0.")
else
        println("Fibonacci series:")
        for i in 0:num_terms-1
        println(fibonacci_recursive(i))
        end
end
```

Output:

```
Enter the number of terms for Fibonacci series:
stdin>  5
Fibonacci series:
0
1
1
2
3
```

5a.develop a julia program which reads a String (word) and prints whether the word is palindrome

```
function is_palindrome(word)
        # Convert the word to lowercase to ignore case sensitivity
        word = lowercase(word)

        # Check if the word is equal to its reverse
        return word == reverse(word)
end

# Main program
println("Enter a word to check if it's a palindrome: ")
word = readline()

if is_palindrome(word)
        println("'$word' is a palindrome.")
else
        println("'$word' is not a palindrome.")
end
```

output:

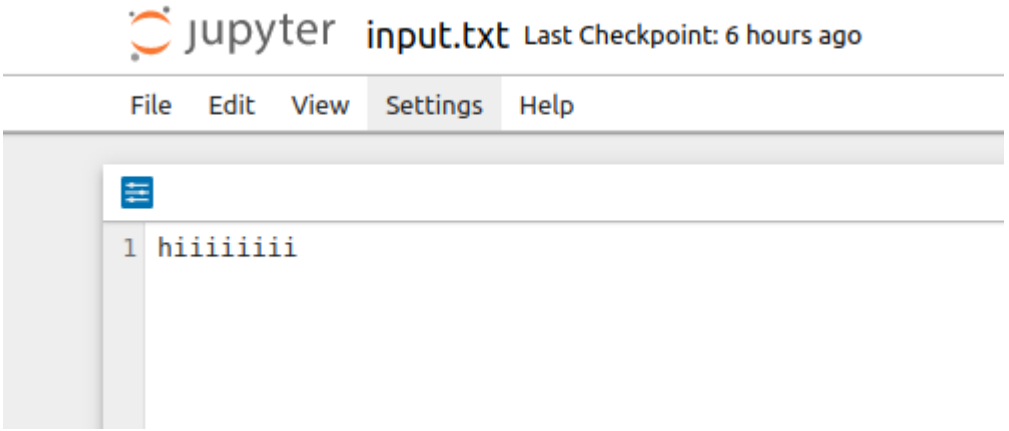
```
Enter a word to check if it's a palindrome:
stdin>  aba
'aba' is a palindrome.
```

```
Enter a word to check if it's a palindrome:
stdin> pass
'pass' is not a palindrome.
```

5b .develop a julia program which reads and prints words prsent in file (input.txt) having random data in which words are dispersed ramdomly(Assumption:a word is a contiguous sequence of letters.a word is delimited by any non-letter chracter or end of line)

```
function extract_words_from_file(filename)
        words = String[]

        # Open the file
        file = open(filename, "r")

        # Read the file line by line
        for line in eachline(file)
        # Split the line into words using regular expression
        words_in_line = split(line, r"[^a-zA-Z]+")

        # Append each word to the words array
        append!(words, words_in_line)
        end

        # Close the file
        close(file)

        return words
end

# Main program
filename = "input.txt"

words = extract_words_from_file(filename)

# Print the extracted words
println("Words present in the file:")
for word in words
        println(word)
end
```

input.txt



output

```
Words present in the file:
hiiiiiiii
```

6.aDevelop 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 count_letters(text::String)
        # Define an empty dictionary to store letter frequencies
        letter_freq = Dict{Char, Int}()

        # Iterate through each character in the text
        for char in text
        # Check if the character is a letter
        if isletter(char)
        # Convert the letter to lowercase
        char = lowercase(char)
        # Update the frequency count for the letter
        if haskey(letter_freq, char)
                letter_freq[char] += 1
        else
                letter_freq[char] = 1
        end
        end
        end

        # Print the letter frequencies
        for (letter, frequency) in sort(collect(letter_freq))
        println("$letter : $frequency")
        end
end

# Example usage
line_of_text = "This is an example sentenceee."
count_letters(line_of_text)
```

Output:

```
a : 2
c : 1
e : 7
h : 1
i : 2
l : 1
m : 1
n : 3
p : 1
s : 3
t : 2
x : 1
```

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 find_most_popular_artist(artists::Vector{String}, votes::Vector{Int})
        # Create a dictionary to store the vote count for each artist
        vote_count = Dict{String, Int}()

        # Initialize vote count for each artist to 0
        for artist in artists
        vote_count[artist] = 0
        end

        # Count the votes for each artist
        for vote in votes
        artist = artists[vote]
        vote_count[artist] += 1
        end

        # Find the most popular artist
        max_votes = maximum(values(vote_count))
        popular_artists = [artist for (artist, votes) in vote_count if votes == max_votes]

        println("Most popular artist(s): ", popular_artists)
end

# Example usage
artists = ["Artist 1", "Artist 2", "Artist 3", "Artist 4", "Artist 5",
        "Artist 6", "Artist 7", "Artist 8", "Artist 9", "Artist 10"]
votes = [2, 3, 1, 2, 5, 3, 7, 8, 1, 3]  # Example votes
```

find_most_popular_artist(artists, votes)

Output:

Most popular artist(s): ["Artist 3"]

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 count_letters(text)
    letter_count = Dict{Char, Int}()

    # Initialize letter_count with 0 for all alphabets
    for c in 'a':'z'
        letter_count[c] = 0
    end

    # Count occurrences of each letter
    for char in text
        if isletter(char)  # Check if the character is a letter
            char = lowercase(char)  # Convert to lowercase to count regardless of case
            letter_count[char] += 1
        end
    end

    return letter_count
end

# Example usage
text = "This is a sample text to demonstrate the program."
result = count_letters(text)

# Display the result
for (letter, count) in result
    println("$letter: $count")
end
```

Output

```
n: 1
f: 0
w: 0
d: 1
e: 5
o: 3
h: 2
j: 0
i: 2
k: 0
r: 3
s: 4
t: 7
q: 0
y: 0
a: 4
c: 0
p: 2
m: 3
z: 0
g: 1
v: 0
l: 1
u: 0
x: 1
b: 0
```

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: e.g. to and To are treated as the same word).

```
function fetch_words(filename)
    words = Set{String}()

    # Read the file line by line
    for line in eachline(filename)
        # Split the line into words and remove punctuation
        for word in split(line, r"[^a-zA-Z]+")
            # Convert word to lowercase for case insensitivity
            word = lowercase(word)
            if !isempty(word)
                push!(words, word)
            end
        end
    end

    return words
end

# Example usage
filename = "sample_text.txt"  # Replace with your file path
unique_words = fetch_words(filename)

# Display the unique words
println("Unique words found in the file:")
for word in unique_words
    println(word)
end
```

Output

```
Unique words found in the file:
by
is
are
whitespace
punctuation
ignored
words
separated
and
```

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

Install using Pkg

Pkg.add("Symbolics") in the julia terminal.

```
function evaluate_expression(expression)
    return eval(Meta.parse(expression))
end

# Get expression input from the user
println("Enter the expression to evaluate:")
expression = readline()

# Evaluate the expression
result = evaluate_expression(expression)

# Display the result
println("Result of the expression '$expression' is: ", result)
```

Output

```
Enter the expression to evaluate:
stdin>  sqrt(2) + 3 * (1/2) + exp(1)
Result of the expression 'sqrt(2) + 3 * (1/2) + exp(1)' is: 5.6324953908321405
```

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(matrix)
    properties = Dict{String, Any}()

    # Determinant
    properties["Determinant"] = det(matrix)

    # Check if the matrix is invertible
    if properties["Determinant"] != 0
        # Inverse
        properties["Inverse"] = inv(matrix)
    else
        properties["Inverse"] = "Matrix is singular and not invertible"
    end

    # Rank
    properties["Rank"] = rank(matrix)

    # Upper triangular matrix
    properties["Upper Triangular Matrix"] = UpperTriangular(matrix)

    # Lower triangular matrix
    properties["Lower Triangular Matrix"] = LowerTriangular(matrix)

    # Diagonal elements
    properties["Diagonal Elements"] = diag(matrix)

    # Euclidean norm
    properties["Euclidean Norm"] = norm(matrix)

    # Square root of a matrix
    properties["Square Root of Matrix"] = LinearAlgebra.sqrt(matrix)

    return properties
end

# Function to get matrix input from user
function get_matrix_input()
    println("Enter the dimensions of the matrix (rows columns):")
    dims = split(readline())
```

```
    rows = parse(Int, dims[1])
    cols = parse(Int, dims[2])

    println("Enter the elements of the matrix row-wise:")
    matrix = [parse(Float64, x) for x in split(readline())]
    reshape(matrix, rows, cols)
end

# Get matrix input from the user
matrix = get_matrix_input()

# Calculate properties
properties = matrix_properties(matrix)

# Display properties
for (property, value) in properties
    println("$property: $value")
end
```

Output

Enter the dimensions of the matrix (rows columns):

stdin> 3 3

Enter the elements of the matrix row-wise:

stdin> 1 2 3 4 5 6 7 8 9

Inverse: Matrix is singular and not invertible
Determinant: 0.0
Square Root of Matrix: ComplexF64[0.44975636349737436 + 0.7622786048165151im 1.0185207327387167 + 0.08415135991834113im 1.5872851019800591 - 0.5939758611037105im; 0.5526217397721798 + 0.20679584461712824im 1.2514702292219844 + 0.0228291414450686im 1.9503187186717892 - 0.161137609479231im; 0.6554871160469856 - 0.34868689170613776im 1.4844197257052523 - 0.038493124780444im 2.3133523353635193 + 0.2717006660213682im]
Upper Triangular Matrix: [1.0 4.0 7.0; 0.0 5.0 8.0; 0.0 0.0 9.0]
Diagonal Elements: [1.0, 5.0, 9.0]
Lower Triangular Matrix: [1.0 0.0 0.0; 2.0 5.0 0.0; 3.0 6.0 9.0]
Euclidean Norm: 16.881943016134134
Rank: 2

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

```
function matrix_addition(matrix1, matrix2)
    if size(matrix1) != size(matrix2)
        println("Matrices must have the same dimensions for addition.")
        return nothing
    end

    result = matrix1 .+ matrix2
    return result
end

function matrix_subtraction(matrix1, matrix2)
    if size(matrix1) != size(matrix2)
        println("Matrices must have the same dimensions for subtraction.")
        return nothing
    end

    result = matrix1 .- matrix2
    return result
end

# Example usage
matrix1 = [1 2 3; 4 5 6; 7 8 9]
matrix2 = [9 8 7; 6 5 4; 3 2 1]

# Perform addition
addition_result = matrix_addition(matrix1, matrix2)
println("Result of addition:")
println(addition_result)

# Perform subtraction
subtraction_result = matrix_subtraction(matrix1, matrix2)
println("\nResult of subtraction:")
println(subtraction_result)
```

Output

```
Result of addition:
[10 10 10; 10 10 10; 10 10 10]

Result of subtraction:
[-8 -6 -4; -2 0 2; 4 6 8]
```

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

```
using LinearAlgebra

function scalar_multiplication(matrix, scalar)
    return matrix * scalar
end

function elementwise_multiplication(matrix1, matrix2)
    return matrix1 .* matrix2
end

function dot_product(matrix1, matrix2)
    return dot(matrix1, matrix2)
end

function cross_product(matrix1, matrix2)
    if size(matrix1) != (3, 3) || size(matrix2) != (3, 3)
        println("Cross product is only defined for 3x3 matrices.")
        return nothing
    end

    result = Vector{Vector{Int}}(undef, 3)
    for i in 1:3
        vector1 = matrix1[:, i]
        vector2 = matrix2[:, i]
        result[i] = [vector1[2]*vector2[3] - vector1[3]*vector2[2],
                     vector1[3]*vector2[1] - vector1[1]*vector2[3],
                     vector1[1]*vector2[2] - vector1[2]*vector2[1]]
    end

    return result
end

# Example usage
matrix1 = [1 2 3; 4 5 6; 7 8 9]
matrix2 = [9 8 7; 6 5 4; 3 2 1]
scalar = 2

# Scalar multiplication
scalar_result = scalar_multiplication(matrix1, scalar)
println("Scalar multiplication:")
println(scalar_result)
```

```
# Element-wise multiplication
elementwise_result = elementwise_multiplication(matrix1, matrix2)
println("\nElement-wise multiplication:")
println(elementwise_result)

# Dot product
dot_product_result = dot_product(matrix1, matrix2)
println("\nDot product:")
println(dot_product_result)

# Cross product
cross_product_result = cross_product(matrix1, matrix2)
println("\nCross product:")
for i in 1:3
    println("Cross product of column $i:")
    println(cross_product_result[i])
end
```

Output

```
Scalar multiplication:
[2 4 6; 8 10 12; 14 16 18]

Element-wise multiplication:
[9 16 21; 24 25 24; 21 16 9]

Dot product:
165

Cross product:
Cross product of column 1:
[-30, 60, -30]
Cross product of column 2:
[-30, 60, -30]
Cross product of column 3:
[-30, 60, -30]
```

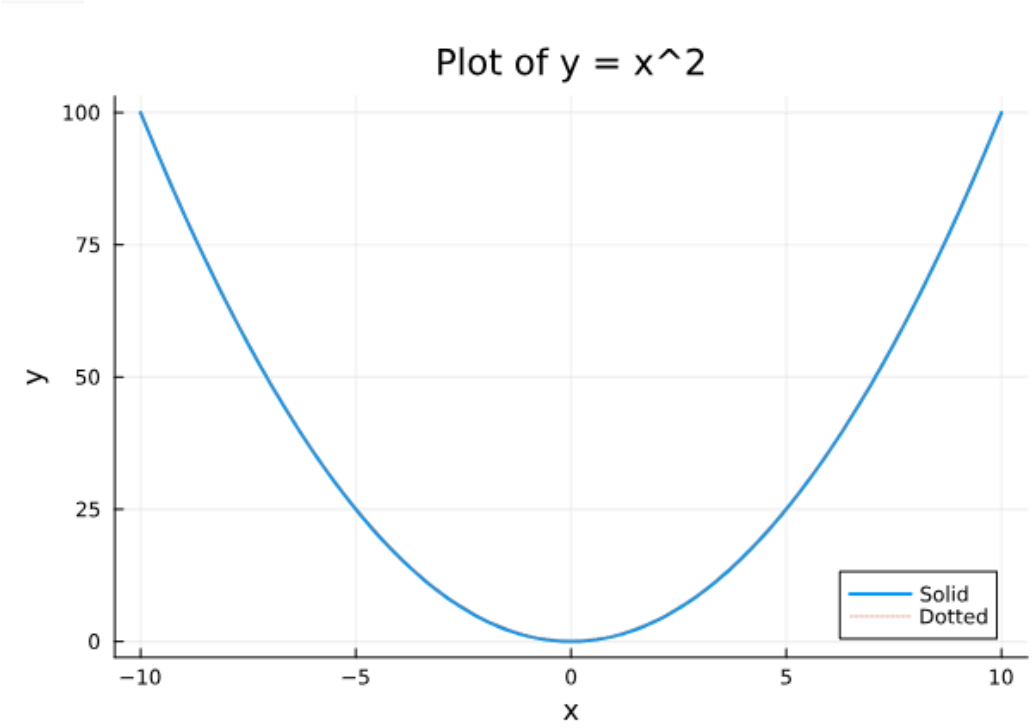
10a. Develop a Julia program to generate a plot of (solid & dotted) a function: y=x2 (use suitable data points for x

Install using Pkg
Pkg.add("Plots") in the julia terminal.

```
# Import the Plots package
using Plots

# Generate x values from -10 to 10 with a step size of 0.1
x = -10:0.1:10

# Calculate y values for the solid line using the function y = x^2
y_solid = x .^ 2

# Generate x values for the dotted line with a larger step size
x_dotted = -10:1:10

# Calculate y values for the dotted line using the function y = x^2
y_dotted = x_dotted .^ 2

# Plot the solid line with a label "Solid" and thicker line width
plot(x, y_solid, label="Solid", linewidth=2)

# Plot the dotted line with a label "Dotted" and using dotted linestyle
plot!(x_dotted, y_dotted, label="Dotted", linestyle=:dot)

# Add x-axis label
xlabel!("x")

# Add y-axis label
ylabel!("y")

# Add title to the plot
title!("Plot of y = x^2")

# Display the plot
plot!()
```

Output

Install using Pkg
Pkg.add("Plots") in julia terminal.



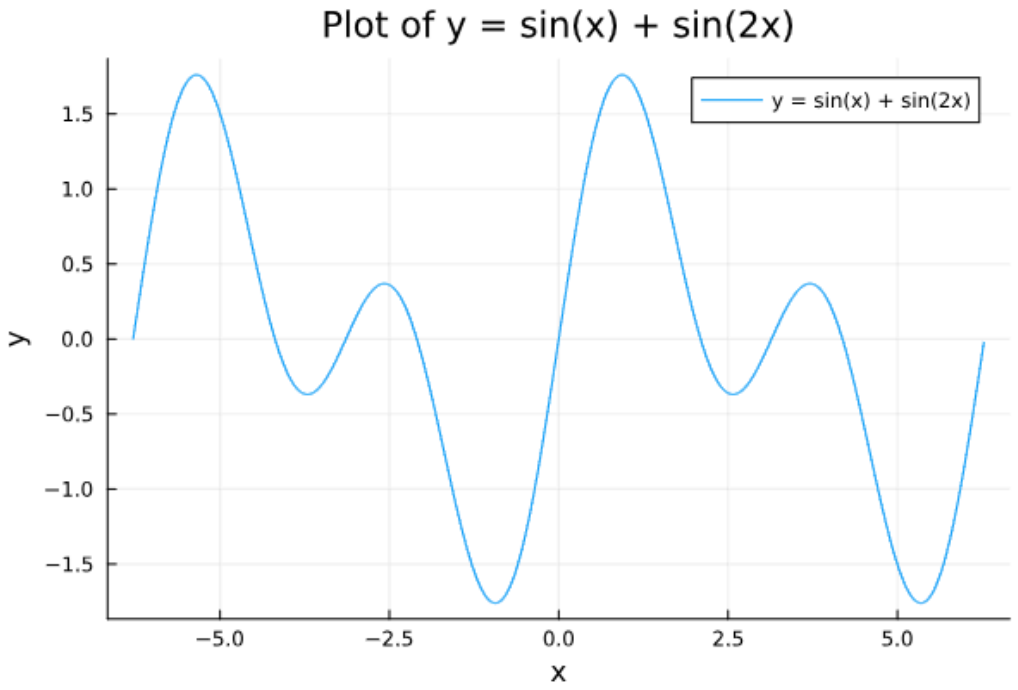
10b. Develop a Julia program to generate a plot of mathematical equation: y = sin(x) + sin(2x).

```
# Import the Plots package
using Plots

# Generate x values from -2π to 2π with a step size of 0.01
x = -2*pi:0.01:2*pi

# Calculate y values for the equation y = sin(x) + sin(2x)
y = sin.(x) + sin.(2 .* x)

# Plot the function with a label
plot(x, y, label="y = sin(x) + sin(2x)")

# Add x-axis label
xlabel!("x")

# Add y-axis label
ylabel!("y")

# Add title to the plot
```

```
title!("Plot of y = sin(x) + sin(2x)")

# Display the plot
plot!()
```

Output

3]:



10c. Develop a Julia program to generate multiple plots of mathematical equations: y = sin(x) + sin(2x) and y = sin(2x) + sin(3x).

```
# Import the Plots package
using Plots

# Define the functions
f1(x) = sin(x) + sin(2x)
f2(x) = sin(2x) + sin(3x)

# Generate data points for x
x = -2π:0.01:2π

# Compute the corresponding y values for each function
y1 = f1.(x)
y2 = f2.(x)

# Create a single plot and plot both functions
plot(x, y1, label="y = sin(x) + sin(2x)", lw=2)
plot!(x, y2, label="y = sin(2x) + sin(3x)", lw=2)
```

```
# Add x-axis label
xlabel!("x")

# Add y-axis label
ylabel!("y")

# Add title to the plot
title!("Plot of y = sin(x) + sin(2x) and y = sin(2x) + sin(3x)")
```

Output