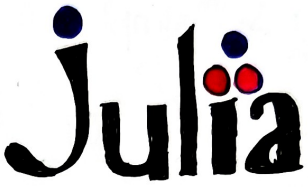
JULIA

Course Code : BDSL45CB
Teaching Hours / Week (L·T·P·S) : 0:0:2:0
Total Hours of Pedagogy : 24
Credits : 01
Semester : 4
CIE Marks : 50
SEE Marks : 50
Total Marks : 100



I. BASICS OF JULIA :

Julia is a programming language designed to be fast & easy to use, especially for tasks like maths, science & data analysis.

Simple example to understand :

Imagine you're working on a math or science project, & you need a tool that's as fast as calculator but as easy to write as a story That's Julia !!

It combines the speed of languages like C [which computers understand quickly] with the simplicity of languages like Python [which humans can write easily]

It's great for things like solving equations, analyzing data or creating simulations.

Conclusion : Julia is popular in research and machine learning because it handles complex problems efficiently while keeping the code clean & simple.

II. DEFINITION OF JULIA :

- It is a high performance, dynamic programming language designed for technical & numerical computing
- It is open source & combines the speed of low-level languages like C with the ease of use of high level languages like python.
- Julia is particularly suited for tasks in data science, maching learning, AI and scientific research.

III. HISTORY OF JULIA :

Julia was officially released in 2012.

Developed by : Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman at MIT.

Goal : The creators wanted a language that,
- Was fast like C
- Had the ease of use of python / MATLAB.

- Could handle math-heavy tasks effortlessly.
- Was open-source, making it accessible to everyone.

Conclusion : Since its release, Julia has gained popularity in fields like data science, ML and scientific computing due to it's unique ability to combine speed, simplicity & flexibility.

III. ADVANTAGES & DISADVANTAGES OF JULIA :

1. High Performance : Fast as C, making it ideal for heavy computations & numerical tasks.
2. Ease of Use : It's syntax is simple & readable similar to Python, which makes it begginer friendly.
3. Specialized for Numerical Computing :
   ↳ Support for complex mathematics, linear algebra & scientific computing.
4. Dynamic & Flexible :
   ↳ It allows dynamic typing, making development faster & more flexible.
5. Rich Ecosystem :
   ↳ Provides many libraries & tools for ML, Data visualization & more.

6. Open Source :

↳ Freely available to everyone with active community support & development.

7. Unified Language :

↳ Can handle all aspects of programming in a single language, from prototyping to production.

DISADVANTAGES :

1. Smaller Community :

↳ Compared to Python / R, Julia has fewer users which can limit the availability of resources.

2. Limited Libraries :

↳ Libraries are very limited compared to python.

3. Slower Start-up Time :

↳ Julia programs may take longer to start due to Just-in-time [JIT] compilation.

4. Less Industry Adoption :

↳ Despite it's benefits, Julia is not yet as widely used in industry as Python / Java.

IV. NOTABLE TOOLS AND LIBRARIES :

1. Plot.jl : Visualization library for creating diverse plots.
2. JuMP.jl : Optimization package for mathematical prgm.
3. Gadfly.jl : A library inspired by ggplot2 for statistical graphics.
4. Flux.jl : ML library.

V. STEP-BY-STEP INSTALLATION :

Step 1 : Go to google, search the latest version of Julia

Step 2 : Click on the windows for Julia download button

Step 3 : The new version is December 1, 2024
[PS : It keeps updating]

Step 4 : Click on 64-bit, Windows.

Step 5 : Once the slw [Julia] gets installed, select the directory & click the next button and click on Finish button.

Step 6 : Download Visual Studio Code [Latest version]
Open VSC → Open Extension → Download Julia.
+ other julia packages.

VI.

HOW TO RUN A PROGRAMME ??

Step 1 : Install latest Julia Version

Step 2 : Install Visual Studio Code [latest Version]

Step 3 : In Visual Studio Code → Extensions → Install Julia.

Step 4 : On Desktop, Create a Folder "XYZ"

Step 5 : Open Visual Studio Code → Open a folder → XYZ → Create a file i.e L1.jl and type the program.

Step 6 : Click on Run ▶

- Julia : Execute active File in REPL
- Julia : Run File in New Process
- Julia : Debug File in New Process.

You can use any of these to run the julia program ..

JULIA CODE EXPLANATION + VIVA QUESTIONS.

Program
↳ 1a + 1b + 1c

⇒ What does the calculator() function do in the code ?

Ans. The calculator() function takes an arithmetic expression as i/p, parses the operands & operator performs the specified operation, & returns the result.

What are operands & operator ?

Operands : are the values / no's on which operations are performed, such as 5 & 3 in the expression 5+3

Operator : It is a symbol that specifies the operation to be performed. such as "+" for addition

"-" → subtraction

"*" → multiplication

"/" → Division.

How does the program handle user input for the arithmetic expression ?

The program uses readline() to get the i/p from the user, splits the i/p into two operands & an operator using split(), & then parses the operands as floating-point no.

What happens if the user enters an invalid operator ?

The program will prints "invalid operator" & doesn't show the result.

What error will occur if the user i/p's non-numeric values for operands ?

It'll throw an error.

Why is split(readline()) used ? & what it does ?

This splits the user's input into separate parts.

Eg: 5+3 ⟶ split ⟶ ["5", "+", "3"].

What's the purpose of parse(Float64,01)

It converts the string repr^n of 1st operand (01) into a floating point no.

Explain the purpose of elseif in this code.

- It allows checking xple conditions sequentially
- It ensures correct operation is executed depending on the operator

Difference b/w print & println

- Print → Displays the o/p on the same line w/o adding a new line

Eg: julia >> print("Hello")
print("World")

Output : HelloWorld

println : Displays the o/p & moves the cursor to the next line

Eg:- println ("Hello")
println ("World")

Output :	Hello World

What's the use of Float64

It is used to store decimal no's like 3.14 , 2.5

What's the use of o1

It stores the 1st no [operand] entered by the user.

What is parse ?

parse is a function in Julia used to convert a string into another data type , such as a no

1b ⇒

- This program performs arithmetic operations [+ , - , * , /] on two complex no's entered by the user .
- It reads the real & imaginary parts of the two no's , creates complex no's & then computes the result of the operations , displaying them in the o/p.

Eg:- Enter the first complex no 3 4
Enter the second complex no 1 2

Explanation :-

First complex no : $z_1 = 3 + 4i$

Second complex no : $z_2 = 1 + 2i$

Sum : $z_1 + z_2 = (3+1) + (4+2)i = 4 + 6i$

Difference : $z_1 - z_2 = (3-1) + (4-2)i = 2 + 2i$

Product : $z_1 \cdot z_2 = (3+4i)(1+2i) = 3+6i+4i+8(-1)$

$= -5 + 10i$ //

Quotient : $z_1 / z_2 = \frac{(3+4i)(1-2i)}{(1+2i)(1-2i)}$

$= \frac{11 - 2i}{5}$

$= 2.2 - 0.4i$ //

10 ⇒

- The program asks the user to enter a mathematical or other expression, evaluates it & prints the result.
- The eval() function is used to interpret & compute the value of the expression entered by the user.

Page No:

Eg:- Enter an expression with mixed types

$2 + 3 * 4$

$2 + (3*4)$ [BODMAS rule]

$2 + (12)$

$= 14$ //

2nd Programme :-

2a

- The function calculates how much to pay based on the hours worked & parts used.
- If the total is less than $150, the min charge will be 150
- The function makepay() calculates the total payment based on the hours worked & the cost of parts used, with certain conditions applied.

Eg :- Suppose the user works for 1 hr & the cost of parts is 30.

case (i)

$\Rightarrow$ $100 * 1 + 30 = 130$

$\because$ 130 is $<$ 150, the function will return the min total payment is 150

case (ii) Suppose the user works for 3 hr & the cost of parts is 80

$100 * 3 + 80 = 380$

$\because$ $380 > 150$, the function will return 380.

Conclusion :- The function ensures that the payment is atleast 150, but if the calculated amount exceeds 150, it will use the calculated value.

2b → This program calculates the total pay for an employee based on the no of hours worked & their hourly rate considering overtime pay for hours worked beyond 40.

1. User Input:

↳ The prgm 1st asks the user to enter the no of hours works

↳ Then, it asks for the rate of pay per hour (rate)

2. Pay Calculation

↳ If the no of hours worked is less than or equal to 40 the prgm calculates the regular pay.

↳ If the no of hours worked is greater than 40, the program calculates : regular pay for the first 40hrs

- Overtime pay for the extra hours.
- Then Total pay.

Eg: Enter the no of hours : 30
Enter the rate of pay : 10

The Employee worked 30hrs, which is less than 40, so there is no overtime.

Regular pay = 30hrs * 10 (rate) = 300

Output : Regular Pay = 300
Overtime pay = 0
Gross pay = 300

Eg:- Enter the no of hours : 50
Enter the rate of pay : 12

↳ The employee worked 50hrs . The 1st 40 hrs are paid & the regular rate , & the remaining 10hrs are considered overtime.

- Regular pay = 40 * rate
  = 40 * 12 = 480

- Overtime Pay = (hours - 40) * rate * 1.5
  = (50 - 40) * 12 * 1.5
  = 10 * 12 * 1.5
  = 180

- Grosspay = Regular Pay + Overtime pay
  = 480 + 180
  = 660 //

Output : Regular Pay = 480
Overtime pay = 180
Gross Pay = 660

3a ⇒ The code Calculates how money grows with interest . It starts with a principal amount & interest rate , then keeps adding interest to the money . The loop stops when the money doubles or after 10 steps . & then it shows the final amount .

P = Principal amount
r = interest rate .

Eg:- Principal (P) : 1000

Interest rate (r) : 5%

In the first year:

- Interest = 1000 * 5% = 50
- New Principal = 1000 + 50 = 1050

In the seund year:

- Interest = 1050 * 5% = 52.5
- New Principal = 1050 + 52.5 = 1102.5

The process continues & the principal grows each year.

`i = 1` → Counter variable i do 1, which will be used to track how many times the loop runs.

`while i <= 10` → This starts a loop, that will keep running as long as "i" is less than or equal to 10, so, it will run upto 10 times.

`Inside the loop` → The principal is updated by adding interest.

`i = i+1` → ries the value of "i" by 1 after each iteration so, the loop moves to the next cycle.

`P >= 2 * OP` → The loop will stop early, if the principal amount doubles, but if not, it'll stop after 10 iterations.

P = 1000
r = 20%

3b ⇒ This Julia program reads no's from a file (inp.txt) & calculates the following :

- Largest no : Find the max no in the file.
- Smallest no: Finds the min no in the file.
- Count : Tracks how many no's are in the file.
- Sum : Adds all the no's together.
- Average : Calculates the avg of the no.

It then prints these statistics : largest, smallest, count, sum & avg.

4th Program :-

4a ⇒ GCD = Greatest Common Divisor
LCM = Least Common multiple

* GCD Function (gcd(m,n)):
  ↳ This finds the greatest divisor of the two no's using the Euclidean algorithm.

* The Lcm function finds the common xple using the formula.

$$LCM = \frac{m \times n}{GCD(m,n)}$$

* The program repeatedly ask the user for two numbers, computes & displays their GCD & LCM & stops when any number is zero / negative.

Eg:- Enter the value of m & n

12 15

Output : GCD = 3
LCM = 60

The program calculates:

GCD (12,15) = 3

LCM (12,15) = $\frac{(12 * 15)}{3}$

= 60//

Input (Invalid Input to Stop)

Enter the value of m & n

-5 7

Output : Since one of the numbers is -ve, the loop stops.

4b :- ⇒

This program calculates the factorial of a no. using a recursive function.

Factorial :- The factorial of a no "n" is the product of all +ve integers from 1 to n.

Eg:- 5! = 5 x 4 x 3 x 2 x 1 = 120

3! = 3 x 2 x 1 = 6

- function factorial (n) ↗ It takes one i/p parameter, n
- if n<=1
  return 1
  end

  } If n is 1 or less than 1, the function returns 1. This is becz the factorial of 0 or 1 is always 1

- return n * factorial (n-1) } If n is > 1, the function calls itself with the value n-1 This is recursion.

println ("Enter a no")
n = parse (BigInt, readline())
↳ takes input as text and parse (BigInt, ---) converts it to a BigInt (used for large no's)

Eg :- 1! = 1
2! = 2x1 = 2
3! = 3x2x1 = 6

4c Program :-

Fibonacci Sequence :- Is a series of numbers where :
↳ The first two no's are 1 & 1
↳ Each subsequent no is the sum of the two preceding no

Eg :- 1, 1, 2, 3, 5, 8, 13

- fibonacc (n) } calculates n^{th} Fibonacci no
- If n == 0 || n == 1
  ↳ If the i/p n is 0 or 1, the function directly returns 1, becz the first two no's in the Fibonacci sequence are always 1.
- If n > 1, the function calls itself twice (Recursive case)
  ↳ Once for fibonacc (n-1)
  ↳ Once for fibonacc (n-2)

Input :- Enter the no 5.

Step 1 :- fibonacc (4) becz (5-1) = 4
fibonacc (4) = frb (3) + frb (2)

Step 2 : frb (3) = fib (2) + frb (1)

fib(2) = fib(1) + fib(0)

here fib(1) = 1
fib(0) = 1

Result : fib(2) = 1+1 = 2

back to fib(3) → fib(2) = 2
fib(1) = 1
fib(3) = 2+1 = 3//

Back to fib(4) → fib(3) = 3
fib(2) = 2
fib(4) = 3+2 = 5

Output :- The program prints 5.

Example :- 0 1 1 2 3 5 8

- Every term is the sum of previous two term
- $a_1 = 1$, $a_2 = 1$ } starting two digits will be 1

$$a_n = a_{n-1} + a_{n-2} \quad , \quad n > 2$$

n = 3 $a_3 = a_2 + a_1$
$a_3 = 1 + 1 = 2$//
$a_4 = a_3 + a_2$
$2 + 1 = 3$//

5th Program :-

5a ⇒

- The program checks . if a word is palindrome or not
- Palindrome is a word that reads the same forwards and backwards.

Eg: MADAM , RACECAR

5b ⇒ This program is designed to extract & print individual words from a text file named "input-txt".

Step 1 : Read the file content

```
mydata = read ("input-txt", String)
```

} This reads entire content from input-txt as a string & stores in the variable mydata .

Step 2 : Initialize a variable to store words

```
word = " "
```

The variable word is initialized as an empty string. It will be used to build words letter by letter.

Step 3 : Iterate through each character

```
for ch in mydata
```

} The loop goes through each character ch in the file content (mydata)

Step 4 : Check if the character is a letter :

```
if isletter(ch)
```

} This function checks whether the character is an alphabet like ('a','b','c') etc).

Step 5: If it is a letter { word = word + ch. } add the letter ch to the current word.

If it is not a letter (like a space, !, ", ? etc)

Step 6: Function Call

extractwordsfromfile

↳ This calls the function & runs the program to extract & print all words from the file.

Eg:- Suppose the file input.txt contains

Hello, world! Julia is awesome

The program will extract & print

Hello world Julia is awesome,

6th Program :-

5a ⇒ The given program calculates the frequency of each letter in a line of text.

Step 1 :- Initialization

↳ The freq array is created with 26 slots, each initialized to 0.

↳ Each slot corresponds to a letter in the alphabet [a to z]

Step 2 : Lowercase Conversion

↳ The i/p file line is converted to all lowercase letters using `lowercase(line)`. This ensures the program treats uppercase & lowercase letters the same.

Step 3 : Iterating Through the Line :

↳ The program goes through each character in the line one by one.

↳ It checks if the character is a letter using `isletter(ch)`. If it's not a letter (eg. digit, punctuation) it skips it.

Step 4 : Printing Frequencies :

↳ The program loops through the letters a to z.

↳ It prints the fq of each letter from freq array

Example :- Hello World

freq(a) = 0
freq(b) = 0
freq(c) = 1
freq(d) = 1
freq(e) = 0
freq(f) = 0
freq(g) = 1
freq(h) = 0
freq(i) = 0
freq(j) = 0
freq(k) = 3
freq(l) = 0
freq(m) = 0
freq(n) = 2
freq(o) = 0
freq(p) = 0
freq(q) = 1
freq(r) = 0
freq(s) = 0
freq(t) = 0
freq(u) = 0
freq(v) = 1
freq(w) = 0
freq(x) = 0
freq(y) = 0
freq(z) = 0

h = 1
e = 1
l = 3
o = 2
w = 1
r = 1
d = 1

6b :- ⟹ The given program simulates a voting system and determines the winner(s) based on the highest no of votes.

Step 1 :- Get i/p for Candidates & Votes

↳ The prgm asks how many candidates (n) are participating in the election & stores their name in a list called candidates.

↳ It also asks for the total no of votes (m).

Step 2 :- Initialize Storage

↳ A list votes of size n is created to store the no of votes each candidate receive. Initially, all values in the list are set to 0

Step 3 :- Input Candidate Name

↳ The program loops n times to allow the user to enter the names of all candidates, storing each name in the candidates list.

Step 4 :- Collect Votes

↳ The program loops m times to collect each vote

↳ If the vote is valid b/w 1 & n, the vote count in the votes list is incremented.

Step 5 :- Find the Maximum votes

↳ After all votes are collected, the program finds the highest no of votes (maxvote) using the maximum function

Step 6 :- Determine and Print Winner(s)

↳ The program loops through the votes list to check ~~the~~ which candidate(s) received maxvote.

Step 7 :- It prints the name(s) of the candidate(s) with the highest vote

7th Program :-

7a ⟹ The program calculates how many times each letter appears in a given line of text.

Step 1 : Prepare to Count letters

↳ A dictionary freq is created to store letters as keys & their frequencies (how many times they appear) as values

↳ Initially, it's empty.

Step 2 : Convert text to lowercase :

↳ converted using lowercase(line).

↳ This makes sure that both uppercase & lowercase letters are treated the same. eg H to h

Step 3 : Loop Through Each Character :

↳ Check if it's a letter. If it's a letter (isletter(ch)), the

program updates its count in the dictionary.

↳ If the letter is already in the dictionary, its count is increased by 1

↳ If not, then it is added with a count of 1.

↳ It ignores characters like spaces, no's, punctuation.

Step 4 : Sort & Display Frequencies

↳ The program collects all the letters (keys) from the dictionary & sorts them alphabetically

Eg: User enters : Hello World!

converts to lower case → hello world

Result : Frequency of d = 1
—"— e = 1
—"— h = 1
—"— l = 3
—"— o = 2
—"— r = 1
—"— w = 1

7b ⇒ The program reads texts from a file and identifies all the unique words in it, ignoring punctuation, spaces & case diffs.

Eg: Input files → Hello, world! Hello OpenUp.

Output → Set ("hello", "world", "OpenUp").

↳ Handles Case Insensitivity → converts all text to lowercase

↳ Handles Non-Letters → Words are split whenever a non-letter character is encountered.

↳ Unique Words → Uses a Set to ensure no duplicates

↳ Dynamic Word Formation → Builds words letter by letter and adds them only when completed.

8th Program :-

8a

This program evaluates a mathematical / logical expression entered by the user & returns the result. If there's an error in the expression, it displays an error message instead of crashing.

Step 1 :- Get User Input

↳ The program asks the user to enter a mathematical expr^n Eg: (2+ 3*4) & stores it in the variable `expr`

Step 2 : Evaluate the Expression

↳ The function evaluate_expression(expr) is called with the i/p expression (expr) as its argument.

↳ Inside the function Meta.parse(expr) → converts the i/p string into a form the computer can understand as a mathematical / logical expression

↳ eval(...) Executes the parsed expression & calculates the result.

Eg:- Input → 3+5*2
Output → 13

Eg! Input → 3+*2
Output → Error.

Eg: Input → 10/2
Output → 5.0

Eg! (3+5)*(2-1)
Output 8

∵ The prgm works for any valid mathematical expression & provides an error msg for invalid I/Ps.

8b ⇒ This program works with matrices & performs various operations like computing the determinant, inverse, rank & more while displaying the results in a neat table format using the PrettyTables package.

Eg:- Find the Determinant of a matrix

$$A = \begin{bmatrix} 6 & 6 \\ -2 & 4 \end{bmatrix}$$ Find $|A|$

Soln :- $$|A| = \begin{vmatrix} 6 & 6 \\ -2 & 4 \end{vmatrix}$$

$$= (6 \times 4) - (-2 \times 6)$$
$$= (24) - (-12)$$
$$= +36$$

9th Program :-

9a → This program performs basic matrix operations like addⁿ, subⁿ on two matrices A & B and displays the results

↳ The matrixop function performs the matrix operations and the results are displayed for the user to see.

Eg:- $A = \begin{bmatrix} 2 & 4 \\ 6 & 7 \end{bmatrix}$ $B = \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}$

$$A+B = \begin{bmatrix} (2+4) & (4+3) \\ (6+2) & (7+1) \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 \\ 8 & 8 \end{bmatrix}$$

Eg:- If $A = \begin{bmatrix} 5 & 5 \\ 4 & 6 \end{bmatrix}$ and $B = \begin{bmatrix} 3 & 4 \\ 6 & 2 \end{bmatrix}$

$$(A-B) = \begin{bmatrix} (5-3) & (5-4) \\ (4-6) & (6-2) \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 1 \\ -2 & 4 \end{bmatrix}$$

9b → This program performs several operations on two matrices / vectors A & B. It then displays the results of those operations

Step 1 : Matrix Scaling [C = 2*A]

↳ This xplies every no in matrix A by 2

↳ If A = [1 3; 4 2] then C = 2*A

C = [2 6 ; 8 4]

Step 2 : Element-wise Multiplication [D = A * B]

↳ This xplies each element of A by the corresponding element of B.

Eg' A = [1 3 ; 4 2]

B = [1 6 ; 2 1]

$$D = A * B = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} (1*1) & (3*6) \\ (4*2) & (2*1) \end{bmatrix}$$

$$D = \begin{bmatrix} 1 & 18 \\ 8 & 2 \end{bmatrix}$$

Step 3 : Dot Product [E = dot (A, B)]

↳ This operation is used with vectors (1D arrays)

↳ It xplies corresponding elements of A & B & then adds the results

Eg' A = [1, 3] B = [4, 5]

E = (1*4) + (3*5) = 4 + 15 = 19.

Step 4 : Cross Product [F = cross(A, B)]

↳ This operation is only for 3D vectors

↳ It gives a new vector that is Ter (perpendicular) to both A & B

Eg: A = [1, 2, 3]
B = [4, 5, 6]

Cross Product F = [-3, 6, -3]

(i) ⇒ A = A_1 A_2 A_3
B = B_1 B_2 B_3

F = $(A_2 \cdot B_3) - (A_3 \cdot B_2)$
= (2·6) − (3·5)
= (12) − (15)
= −3//

(ii) ⇒ $(A_3 \cdot B_1) - (A_1 \cdot B_3)$
= (3·4) − (1·6)
= (12) − (6)
= 6//

(iii) ⇒ $(A_1 * B_2) - (A_2 \cdot B_1)$
= (1·5) − (2·4)
= (5) − (8)
= −3//

10th Program :-

10 a ⇒ Pkg.add("Plots")

↳ used for creating plots & graphs in Julia.

Import Pkg

↳ allows to use Julia's package manager & Pkg.add() is used to install packages.

Using Plots

↳ You can use its functions to create & customize plots.

savefig("line_plot.png")

↳ saves the plot as an image

This prgm creates a simple line plot that shows the relationship b/w x & y, where y is the square of x.

plot(x, y, label = "square", xlabel = "X-axis", ylabel = "Y axis)

↳ plot(x,y) : Plots the values of x (horizontal) & y (vertical)

↳ label = square : The graph will appear in square box

↳ xlable = "X-axis" : Labels the x-axis as "X-axis"

↳ title = "Line-Plot" : Adds the title "Line Plot" at the top of the plot

- x = Array([1, 2, 3, 4, 5])
  y = x.^2

  y = [1^2, 2^2, 3^2, 4^2, 5^2]
  = [1, 4, 9, 16, 25]

sq of 1 = 1
sq of 2 = 4
sq of 3 = $(3)^2$ => 9
sq of 4 = 16
sq of 5 = 25

10 b

⇒ eq(x) = sind(x) + sind(2x)

↙

define function that takes i/p x & calculates the value of expression sind(x) + sind(2x)

↳ sind(x) → is a sine function. but instead of taking i/p's in radians, it uses degrees. So, sind(x) gives the sine of x, where x is in degrees

Eg:- x = 30

then eq(30) = sind(30) + sind(60)

sin 0 = 0
sin 90 = 1
sin 30 = 0.5 i.e 1/2
sin 60 = √3/2
sin 45 = √2/2

10] c

⟹ plot (eq, 1:500)

↳ This generates a plot of the function eq(x) over the range from 1 to 500

↳ The 1:500 means, the function will be plotted for values of x b/w 1 & 500

↳ eq1(x) = sind(x) + sind(3x)

↳ This defines another function eq1(x) that returns the sum of sind(x) + sind(3x)

↳ plot! (eq1, 1:500)

→ This adds the plot of second function eq1(x) to the same graph as the 1st one.

→ The "!" → indicates the existing plot will be modified (i,e the new plot will be overlaid on the original one)

In short, the code plots two sine functions, one with sind(x) + sind(2x) and the other with sind(x) + sind(3x) and saves the result as an image.

Thank you!!
All the Best.