Introduction to Julia Programming

Sujit Sandipan Chaugule[1*], Dr. Amiya Ranjan Bhowmick[2]

[1*]Department of Pharmaceutical Sciences and Technology, Institute of Chemical Technology, Mumbai

[2]Department of Mathematics, Institute of Chemical Technology, Mumbai

A Glimpse into Julia Programming

To effectively learn any programming language, it is essential to grasp its basic data structures and functionalities. Julia is a high-performance, dynamically typed programming language specifically designed for numerical and scientific computing. It provides an easy-to-use syntax while ensuring high execution speed, making it an excellent choice for data analysis, statistics, and machine learning.

Installation

To install Julia, follow these steps:

- Go to the official Julia website: https://julialang.org/downloads/
- Download the appropriate version for your operating system (Windows, Linux, or macOS).
- Install Julia by following the on-screen instructions.
- Open Julia REPL (Read-Eval-Print Loop) or use Jupyter Notebook with the IJulia package for interactive coding.

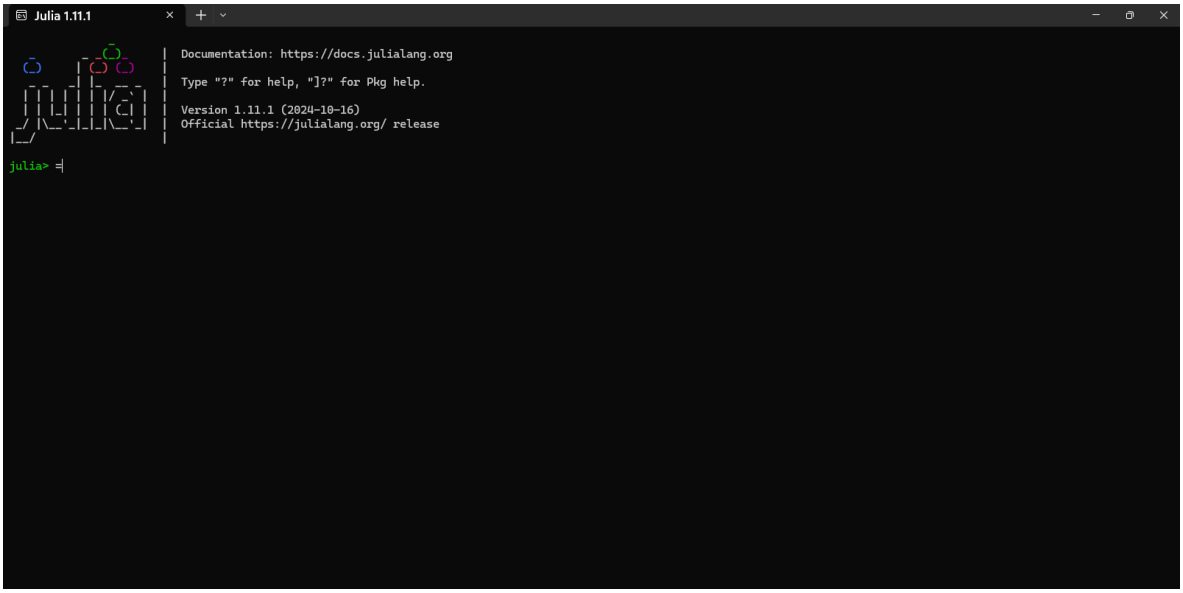

Figure 1: Julia REPL interface at startup, ready for executing commands.

To install Jupyter Notebook support, open Julia and run:

```
using Pkg
Pkg.add("IJulia")
```

Running Julia in VS Code

You can also use Visual Studio Code (VS Code) as an IDE for Julia development. To set it up:

- Install VS Code from https://code.visualstudio.com/.
- Open VS Code and go to the Extensions tab (Ctrl + Shift + X).
- Search for Julia and install the official Julia extension.
- Open Command Palette (Ctrl + Shift + P) and type Julia: Select Environment to set up your Julia installation.
- Open a new .jl file and start coding!

Running Julia in Google Colab

You can also run Julia in Google Colab, a cloud-based platform for interactive coding. To set it up:

- Open Google Colab.
- Create a new notebook and go to Runtime > Change runtime type.

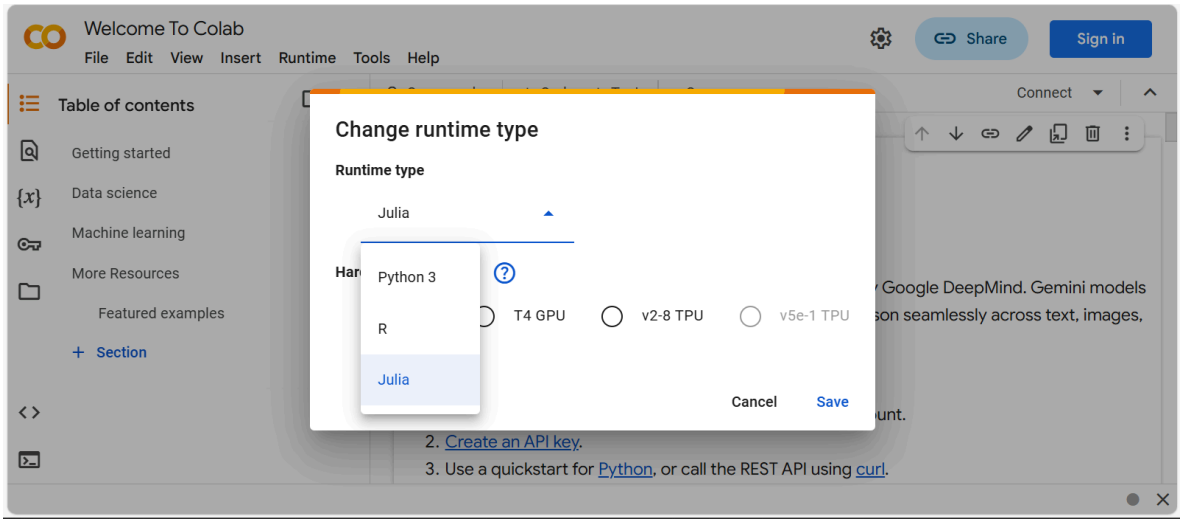

Figure 2: Google Colab

- Changing the runtime type in Google Colab to Julia. The dropdown menu in the "Change runtime type" dialog allows users to select between Python 3, R, and Julia for their notebook environment.

```
!apt update
!apt install julia -y
!julia -e 'using Pkg; Pkg.add(["IJulia"]); using IJulia'
```

- Restart the runtime and start running Julia code inside Colab.

Basic data Structures in Julia

Vectors (1D Arrays)

A vector in Julia is created using square brackets:

In [1]:
```
x = [1, 2, 3, 4, 5]
println(x)
```

```
[1, 2, 3, 4, 5]
```

You can also use the range notation:

In [2]:
```
x = 1:5
println(x)
```

```
1:5
```

Matrices (2D Arrays)

Julia allows easy creation of matrices:

In [3]:
```
A = [1 2 3; 4 5 6; 7 8 9]  # 3x3 matrix
A
```

Out[3]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

Accessing elements

In [4]:
```
println(A[1, 2])  # First row, second column
println(A[:, 2])  # Entire second column
```

```
2
[2, 5, 8]
```

Tuples

Tuples are immutable sequences of elements:

In [5]:
```
tuple_example = ("A", 10, true)
println(tuple_example)
```

```
("A", 10, true)
```

Dictionaries

Dictionaries store key-value pairs:

In [6]:
```
dict_example = Dict("A" => 10, "B" => 20)
println(dict_example["A"])  # Access value associated with key "A"
```

```
10
```

Writing functions in Julia

Functions in Julia are defined using the `function` keyword:

In [7]:
```
function add_numbers(a, b)
    return a + b
end
```

```
println(add_numbers(3, 5))
```

```
8
```

Alternatively, use short-form syntax

In [8]:
```
add_numbers(a, b) = a + b
println(add_numbers(3, 5))
```

```
8
```

We can also do customization of the function. Suppose instead of numeric value someone enters a character in the entry, then the summation will not be computed and result in error. Inside the function body, we can add more comments

In [9]:
```
# Function to determine the nature of roots of a quadratic equation
function quadratic_roots(a, b, c)
    discriminant = b^2 - 4*a*c

    if discriminant > 0
        return "Two distinct real roots"
    elseif discriminant == 0
        return "One real root (repeated)"
    else
        return "Complex conjugate roots"
    end
end

# Test the function
println(quadratic_roots(1, -3, 2))  # Output: Two distinct real roots
println(quadratic_roots(1, -2, 1))  # Output: One real root (repeated)
println(quadratic_roots(1, 2, 5))   # Output: Complex conjugate roots
```

```
Two distinct real roots
One real root (repeated)
Complex conjugate roots
```

You can also write down mathematical functions. The common algebraic and trigonometric functions are available in Julia. Some common functions are `sin()`, `cos()`, `log()`, etc. In the following, we show an example. The `plot()` function from the `Plots.jl` package is very important to understand when dealing with mathematical functions.

In [10]:
```
using Plots, LaTeXStrings
```

In [11]:
```
f(x) = x^2*sin(10*x)  # body of the function
p1 = plot(f, -1, 1, color = "red", xlabel = L"x", lw = 3,
ylabel = L"f(x)", title = L"x^3\sin(10x)", label = "")

g(x) = x^3
h(x) = sin(10*x)
p2 = plot(g, -1, 1, color = "red", lw = 3, linestyle = :dash,
ylims = (-1.2, 1.6), label = L"g(x)")
plot!(h, color = "blue", lw = 2, linestyle = :dash, xlabel = "x",
    label = L"h(x)")
plot(p1, p2, layout = (1,2), size = (800, 400))
```

Out[11]:

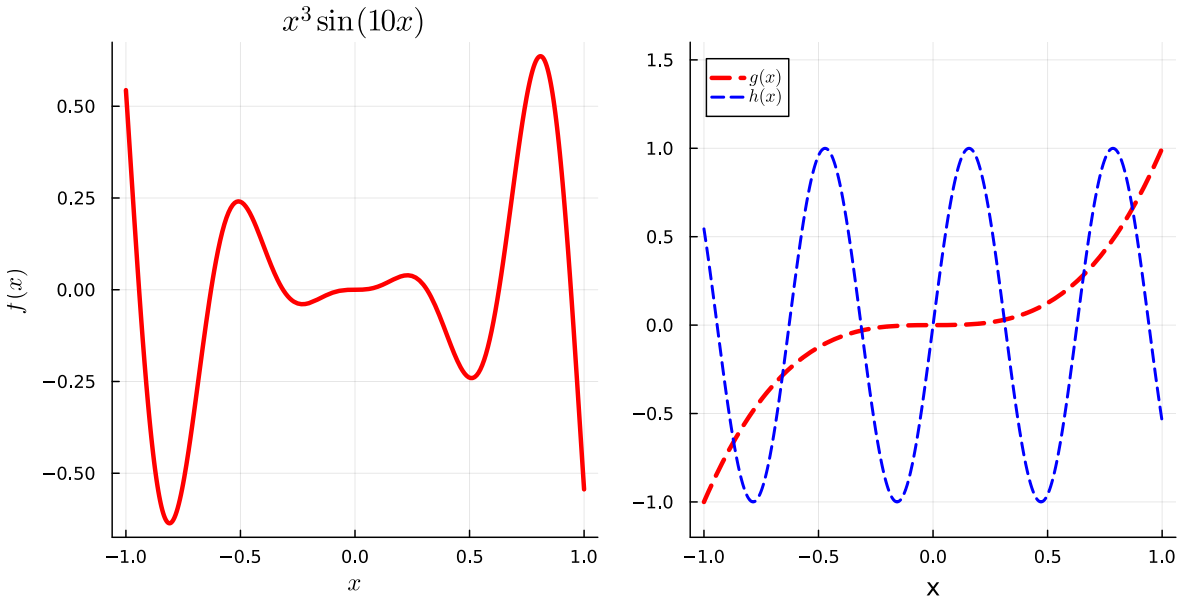

Figure 3: The curve function can be efficiently used to plot mathematical functions in an effective way. You can also perform mathematical annotations on the plot using the `LaTeXStrings.L""` function.

Some important functions in Julia

In [12]:
```julia
x = vcat(fill(1, 3), reverse(1:5))
println(x)

println(argmax(x))
println(argmin(x))

println(findall(x .== 1))
println(findall(x .!= 1))

println(sum(x .== 1))

println(x[x .== 1])

println(20)

println(x[x .!= 1])
println(x[x .> 1])
```

```
[1, 1, 1, 5, 4, 3, 2, 1]
4
1
[1, 2, 3, 8]
[4, 5, 6, 7]
4
[1, 1, 1, 1]
20
[5, 4, 3, 2]
[5, 4, 3, 2]
```

Symbolic Computation

Julia also provides some options for performing symbolic computation using `Symbolics.jl` package. In the following, we perform the symbolic derivative of the function $f_s(x) = \frac{2x}{3+x}$ to compute $f_s'(x)$ using Julia. Another example is also given with $g_s(x) = \frac{2x^2}{1+x^2}$ We need to

define the function symbolically using the `@variables` and `@derivatives` macros before differentiation.

In [13]:
```
using Symbolics, LaTeXStrings
```

In [14]:
```
@variables x
f_s = 2*x/(3+x)
```

Out[14]:

$$\frac{2x}{3+x} \tag{1}$$

In [15]:
```
df_s = Symbolics.derivative(f_s, x) # derivative w.r.t x
```

Out[15]:

$$\frac{-2x}{(3+x)^2} + \frac{2}{3+x} \tag{2}$$

In [16]:
```
@variables x
g_s =  ((2*x^2)/(1+x^2))
```

Out[16]:

$$\frac{2x^2}{1+x^2} \tag{3}$$

In [17]:
```
dg_s = Symbolics.derivative(g_s, x)
```

Out[17]:

$$\frac{4x}{1+x^2} - 2x\frac{2x^2}{(1+x^2)^2} \tag{4}$$

In [18]:
```
p1 = plot(df_s, 0.001, 10, color = "red", lw = 3,
xlabel = L"x" , ylabel = L"\nabla(f_s(x))", label = "")

p2 = plot(dg_s, 0.001, 10, color = "blue", lw = 3,
xlabel = L"x" , ylabel = L"\nabla(g_s(x))", label = "")

plot(p1, p2, layout = (1,2), size = (800, 500))
```

Out[18]:

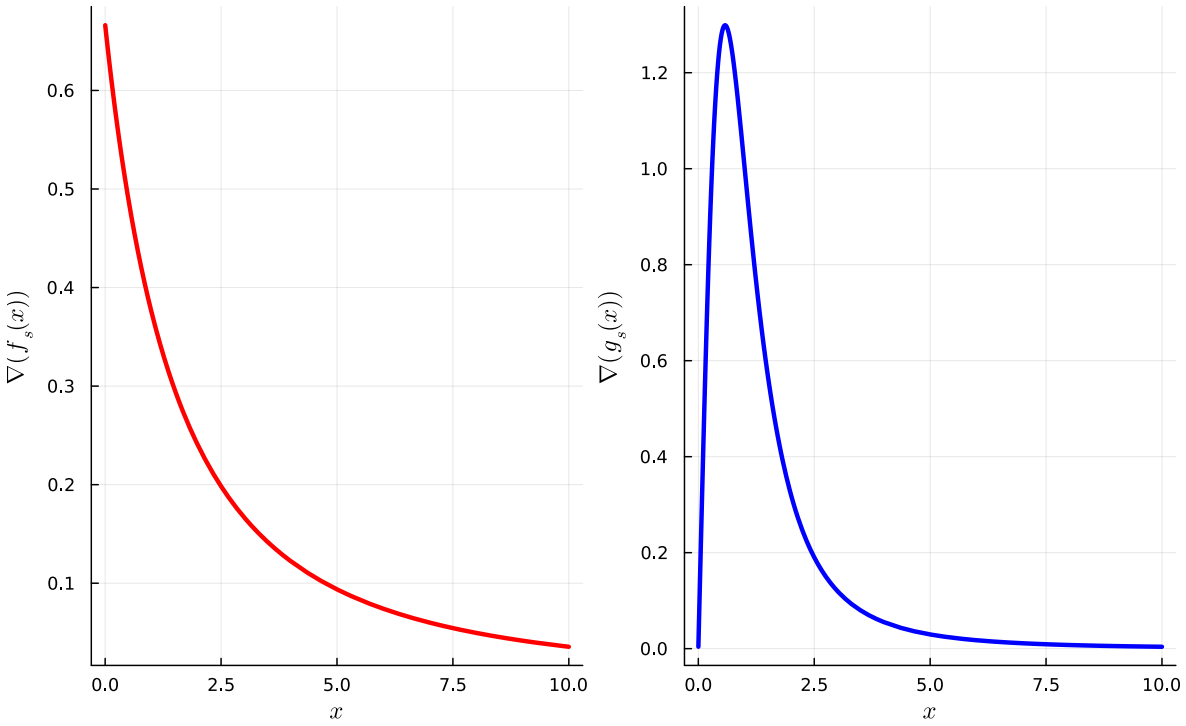

Figure 4: The derivative of the functions $f_s(x)$ and $g_s(x)$ for $x \in (0, \infty)$.

References

- Nagar, S. (2017). *Beginning Julia programming: For engineers and scientists.* Apress. https://doi.org/10.1007/978-1-4842-3171-5
- Lauwens, B., & Downey, A. B. (2019). *Think Julia: How to think like a computer scientist.* O'Reilly Media.
- Kwon, C. (2016). *Julia programming for operations research: A primer on computing.* CreateSpace Independent Publishing Platform.