Plotting Graphs

This tutorial will cover the topic of plotting in Julia-Lang.

```
md"This tutorial will cover the topic of plotting in Julia-Lang. "
```

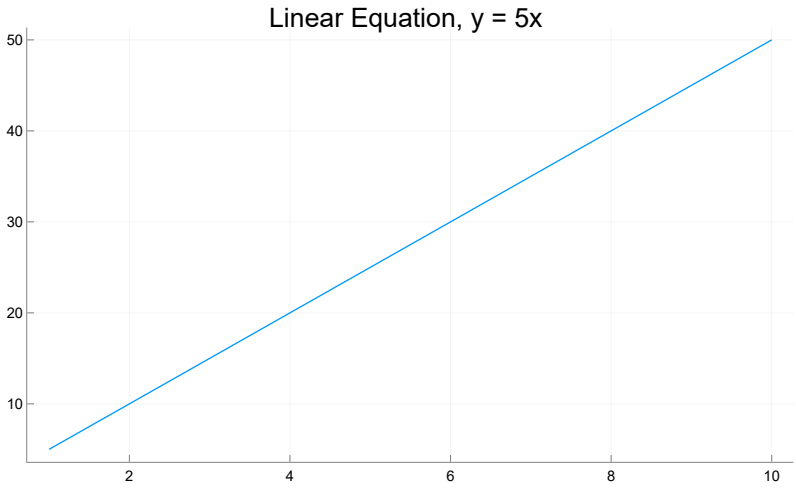
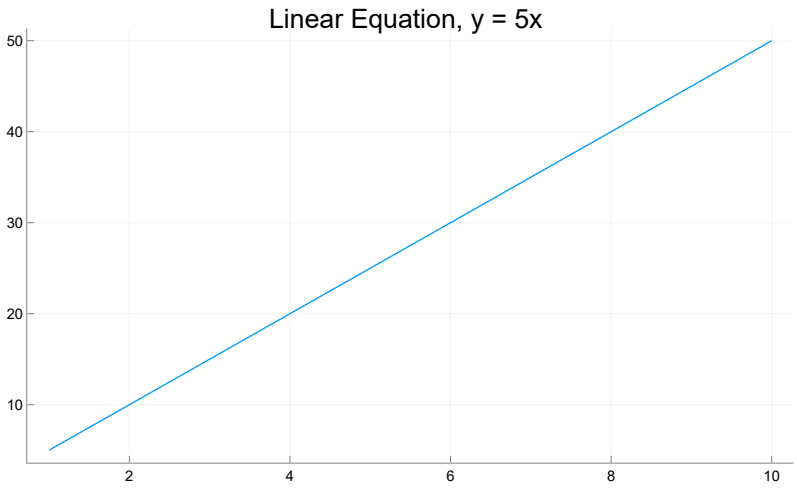
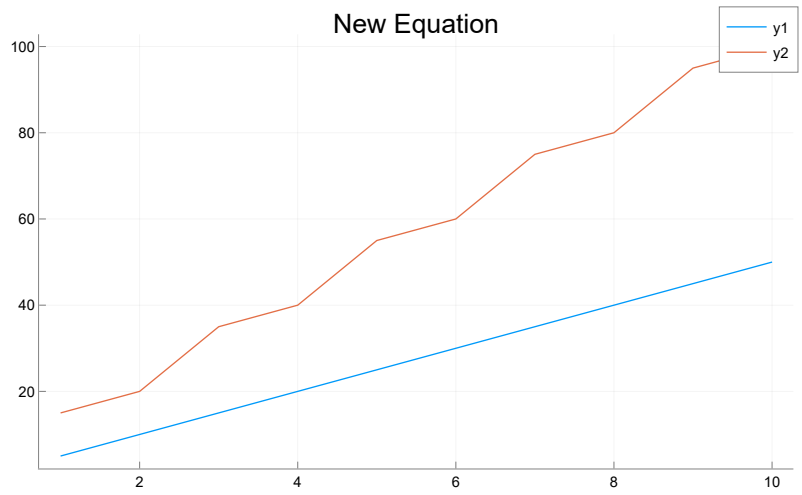
Steps to install the package "Plots".(manually)

```
md"Steps to install the package \"Plots\".(manually)"
```

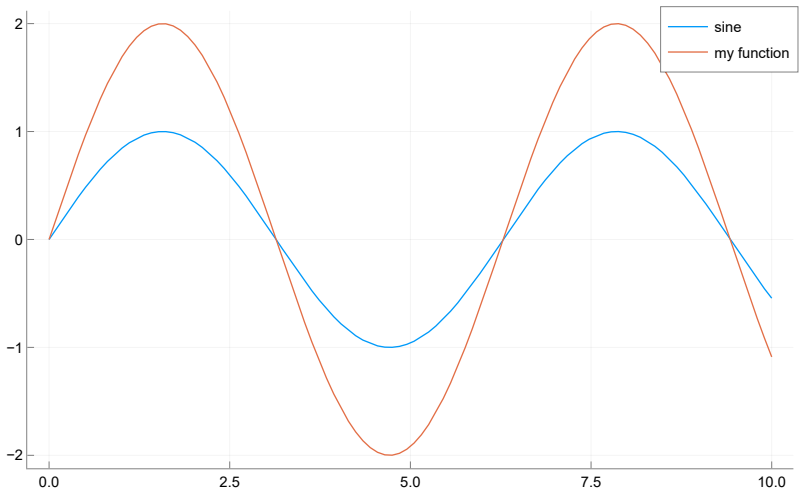
1. Open your terminal.
2. Launch "julia".
3. In the Julia REPL (Read-Eval-Print Loop), press the ']' key on your keyboard to enter the package manager mode.
4. Once inside the package manager mode, run the following command to add the "Plots" package:

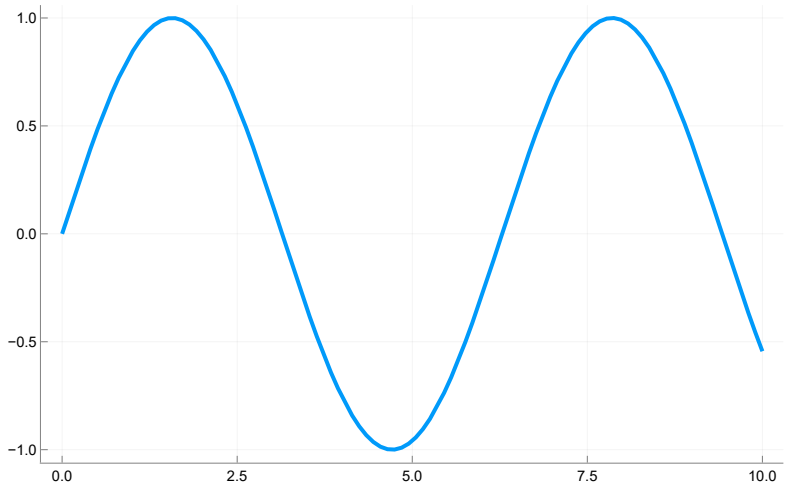
```
import Pkg
Pkg.add("Plots")
```

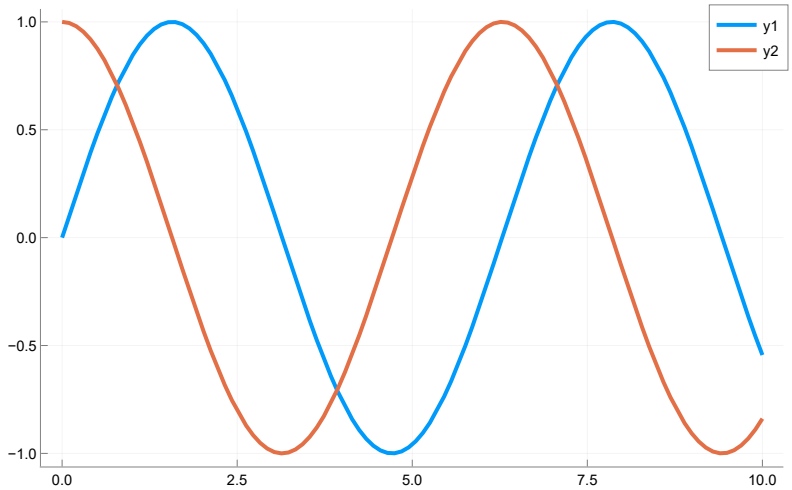
Now you have successfully set up plotting in Julia. You can use the "Plots" package for creating visualizations.

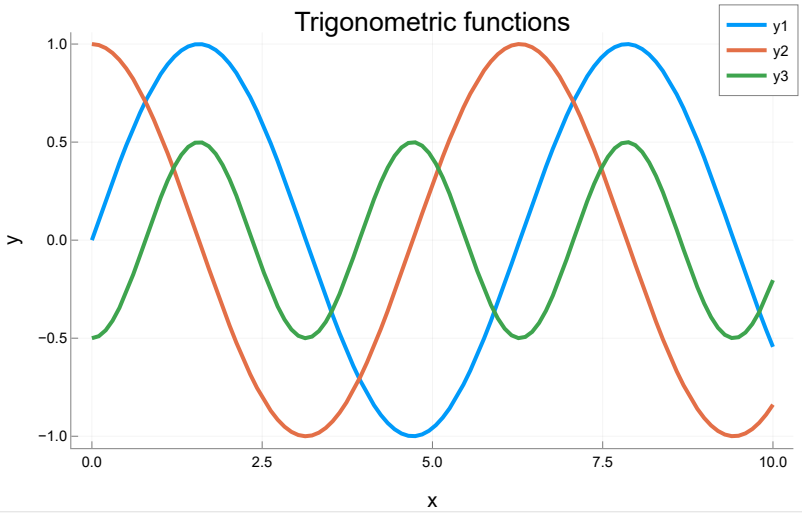
```
md"""
1. Open your terminal.
2. Launch "julia".
3. In the Julia REPL (Read-Eval-Print Loop), press the ']' key on your keyboard to enter the package manager mode.
4. Once inside the package manager mode, run the following command to add the "Plots" package:
```julia
import Pkg
Pkg.add("Plots")
```
Now you have successfully set up plotting in Julia. You can use the "Plots" package for creating visualizations.
"""
```

Initialize "Plots" package

```
md"### Initialize \"Plots\" package"
```

```
using Plots
```

Pluto's package manager automatically takes care of installing and tracking dependencies, so we do not really need to do the above stated, but it adds some knowledge.

```
md"Pluto's package manager automatically takes care of installing and tracking dependencies, so we do not really need to do the above stated, but it adds some knowledge."
```

In the context of plotting in Julia (and other data visualization libraries), choosing a backend allows you to specify how and where your plots are displayed or saved.

```
md"In the context of plotting in Julia (and other data visualization libraries), choosing a backend allows you to specify how and where your plots are displayed or saved."
```

```
PlotlyBackend()
```

```
plotly()
```

This can also be done in a single line.

```
using Plots; plotly()
```

```
md"""This can also be done in a single line.
```julia
using Plots; plotly()
```
"""
```

Getting into Plots

```
md"## Getting into Plots"
```

```
x = 1:10
```

```
x=1:10
```

```
y =  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

```
y= [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

We are going to plot the linear equation, $y = 5x$

```
md"We are going to plot the linear equation, $y=5x$"
```




```
plot(x, y, legend=false, title="Linear Equation, y = 5x ")
```

This can also be written in seperate lines,

```
begin
    plot(years, apples)
    plot!(legend=false)
    plot!(title="Number of apples per year")
end
```

Here,

```
 plot!
```

means, that the command is modifying the current plot.

```
md"""
This can also be written in seperate lines,
```julia
begin
    plot(years, apples)
    plot!(legend=false)
    plot!(title="Number of apples per year")
end
```
Here,
```julia
 plot!
```
means, that the command is modifying the current plot.
"""
```




```
begin
    plot(x, y)
    plot!(legend=false)
    plot!(title="Linear Equation, y = 5x ")
end
```

As you can see, both gives the same result.

```
md"As you can see, both gives the same result."
```

We can use this approach to graph multiple plots in the same diagram, let's see!

```
md"We can use this approach to graph multiple plots in the same diagram, let's see!"
```

```
x1 = 1:10
```

```
x1 =1:10
```

```
y1 =   Int64[
          1:  15
          2:  20
          3:  35
          4:  40
          5:  55
          6:  60
          7:  75
          8:  80
          9:  95
          10:  100
    ]
```

```
y1=[15, 20, 35, 40, 55, 60, 75, 80, 95, 100]
```




```
begin
 plot(x, y, legend=true, title="Linear Equation, y = 5x ")
 plot!(x1,y1, legend=true, title="New Equation")
end
```

There are different types of plotting available to us.

1. Scatter : It will give us a discrete plot of our input.

```
scatter(x, y, legend=true, title="Linear Equation, y = 5x ")
```

2. Linear : It will give us a discrete plot of our input, connected with straight lines.

```
plot(x, y, legend=true, title="Linear Equation, y = 5x ")
```

```
md"""
There are different types of plotting available to us.

1. Scatter : It will give us a discrete plot of our input.
```julia
 scatter(x, y, legend=true, title="Linear Equation, y = 5x ")
```

2. Linear : It will give us a discrete plot of our input, connected with straight lines.
```julia
 plot(x, y, legend=true, title="Linear Equation, y = 5x ")
```
"""
```

Plotting more functions

```
md"Plotting more functions"
```




```
begin
    x3 = range(0, 10, length=100)
    function equation(x)
        y3 = 2*sin.(x)
        return y3
    end

    plot(x3, sin, label="sine") #graphs sin(x3)
    plot!(x3, equation, label="my function") #graphs 2*sin(x)
end
```

Plotting functions in a different way

```
md"Plotting functions in a different way"
```




```
begin
 x4 = range(0, 10, length=100)
 y4 = sin.(x4)
 plot(x4, y4, legend=false, linewidth=3)
end
```

In Plots.jl, the structure of your data is key to creating series for plotting. Each column in a matrix represents a series, comprising a set of interconnected data points that are used to generate lines, surfaces, or other graphical elements. This flexibility allows you to visualize multiple lines simultaneously by organizing your data in a matrix, where each column corresponds to an individual line.

For instance, consider a matrix like [y1 y2]. This forms a 100x2 matrix, signifying it contains 100 data points distributed across 2 columns. In this context, each column serves as an independent line or series in your plot, enabling you to visualize and compare multiple series within the same plot.

```
md"""
In Plots.jl, the structure of your data is key to creating series for plotting. Each column in a matrix represents a series, comprising a set of interconnected data points that are used to generate lines, surfaces, or other graphical elements. This flexibility allows you to visualize multiple lines simultaneously by organizing your data in a matrix, where each column corresponds to an individual line.

For instance, consider a matrix like [y1 y2]. This forms a 100x2 matrix, signifying it contains 100 data points distributed across 2 columns. In this context, each column serves as an independent line or series in your plot, enabling you to visualize and compare multiple series within the same plot.
"""
```




```
begin
 x5 = range(0, 10, length=100)
 y51 = sin.(x5)
 y52 = cos.(x5)
 plot(x5, [y51 y52], linewidth=3, minorgrid=true)
end
```

```
    @.
```

This is a convenience macro that inserts dots for every function call to the right of the macro, ensuring that the entire expression is to be evaluated in an element-wise manner. If we inputted the dots manually, we would need three of them for the sine, exponent, and subtraction, and the resulting code would be less readable.

```
md"""
```julia
    @.
```
This is a convenience macro that inserts dots for every function call to the right of the macro, ensuring that the entire expression is to be evaluated in an element-wise manner. If we inputted the dots manually, we would need three of them for the sine, exponent, and subtraction, and the resulting code would be less readable.
"""
```




```
begin
 y53 = @. sin(x5)^2 - 1/2  # equivalent to y3 = sin.(x).^2 .- 1/2, the entire
                           # expression is to be evaluated in an element-wise manner.

 plot!(x5, y53, linewidth=3)
 xlabel!("x")
 ylabel!("y")

 title!("Trigonometric functions")
end
```

Until now, you may have noticed something unique about Pluto. When one cell gets executed, Pluto automatically executes every corresponding cell that shares even one element with it.

```
md"Until now, you may have noticed something unique about Pluto. When one cell gets executed, Pluto automatically executes every corresponding cell that shares even one element with it."
```