Julia Workshop 2024

- Subplots
- Axes: Limits, Aspect Ratio & Frames
- Log and Semilog plots
- 3D and Cantours
- Plot Recipes

Julia Type System
- Abstract vs Concrete types
- Custom Types
- Abstract Types
- Parametric Types

Multiple Dispatch
- How it works
- Example
- Example: OneHot Vector
- Excercises
  - 1) Type Hierarchies
- 2) Indexing of a range type
- 3) Rational Numbers

Linear Algebra

Differential Equations
- ODEs
  - Example 1
  - Example 2: Simple Pendulum

Optimization
- Example 1
- Defining the Objective Function
  - Derivative-Free solvers
  - Gradient-Based Solvers
  - Hessian-Based Solvers
- Constraints
  - Example 2:

Linear Solvers
- `LinearSolve.jl`

Nonlinear Solvers
- Example 1

Symbolic Computation

Graph Theory

Data Science

By:

1. Mohammed Alshahrani,
2. Norah Al-Muraysil
3. Hajar Alshaikh

Source: Julia Learn

Introduction

Audience of this Workshop

This workshop is targeting an audience familiar with programming that wants to transition to Julia and use Julia effectively.

> We prefer you bring your own **laptop** with you to the workshop?

Why Julia?

In **February 14th, 2012**, the creators of Julia: Jeff Bezanson (computer scientist), Stefan Karpinski (computer scientist), Viral B. Shah (computer scientist) and Alan Edelman (mathematician) announced **Julia** in a Blog Post titled **"Why We Created Julia"**.

They wrote

> We love all of these languages [Matlab, Lisp, Python, Ruby, Perl, Mathematica, R and C]; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off. **We are greedy: we want more.**
>
> - We want a language that's open source, with a liberal license.
> - We want the speed of C with the dynamism of Ruby.
> - We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab.
> - We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell.
> - Something that is dirt simple to learn, yet keeps the most serious hackers happy.
> - We want it interactive and we want it compiled.

Julia Highlights

- **Julia is Fast**: Julia was designed for **high performance**. Julia programs automatically compile to efficient native code via LLVM, and support multiple platforms.
- **Julia is Dynamic**: Julia is dynamically typed, feels like a scripting language, and has good support for interactive use, but can also optionally be separately compiled.

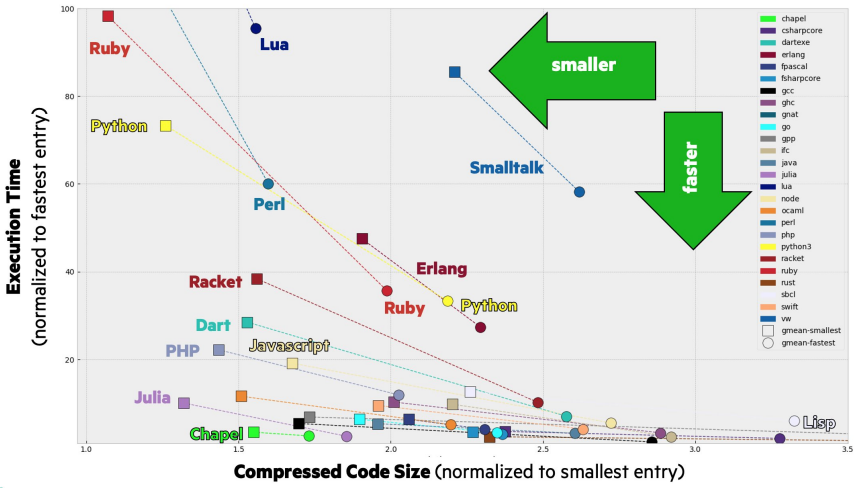
- **Julia is Reproducible**: Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.
- **Julia is Composable**: Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.
- **Julia is General**: Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.
- **Julia is Open source**: Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

Advantages of Julia

See source

1. **Julia solves the two language problem.**
2. **Julia occupies the "sweet spot" of high performance and simple code.**

CLBG: ALL-LANGUAGE SUMMARY (FEB 7, 2023)



3. **Julia's syntax is intuitive and as close to math as possible.**
4. **Multiple dispatch.**
5. **Unprecedented code re-use and inter-package communication.**
6. **Julia is written in Julia.**
7. **Julia's package ecosystem is already top-of-the-class in some scientific disciplines.**
8. **Developer communities around seemingly every area of science.**
9. **Interoperability with other languages.**
10. **Exceptionally strong integrated package manager.**
11. **Welcoming and responsive community.**
12. **Many large-scale projects and organizations have already adopted Julia**
13. **Easy installs and pre-built binary dependencies.**

Wait a minute! why do you try so hard to convince people?

It should be obvious that I am not funded or affiliated by Julia in any way. I try to convice people to use Julia because

> "I genuinly believe that Julia is the best programming language for Scientific computing and it can accelerate progress and increases openness in academia"

Installing Julia

Option 1: For Windows 10 and 11 users:

1. Open the Microsoft Store and search for "Julia" in the search area.
2. Download the Julia app from the store. This will install both Julia and Juliaup simultaneously.
3. Verify the installation:
    - Open a command prompt window and run the following command to verify that Julia and Juliaup installed correctly:

```
julia --version
```

```
juliaup --version
```

Option 2: For users with other operating systems:

1. Visit the Julia Language website.
2. Follow the instructions provided on the website to download and install Julia on your specific operating system

Developing in Julia

- You can develop your Julia programs by typing directly in the REPL. Note that the history of everything you type in the REPL is stored, even across restarts. You can access this history using the up-arrow key, or by using ctrl-r for searching.
- If you want a more graphical environment that also saves your programs, you can use Julia from within your favorite text editor: Visual Studio Code, Vim, Emacs, IntelliJ IDEA. Minimally, a text editor like Vim or Notepad will allow you to create a document where you can write programs but without automatically adding a bunch of formatting information.
- To make coding easier, there are lots of “integrated development environments” (IDEs) out there that offer more than a simple text editor. These IDEs will allow you not only to write and store programs you’ve written, but also to run them, making it easier to test and experiment as you

write. If you don't yet have a favorite editor or IDE, we would suggest using **VS code along with the Julia plugin**.

- Pluto: Simple, reactive programming environment for Julia
- Jupyter Notebook

Julia Basics

Basic syntax

Assignment

- You can assign anything to a variable binding. This includes functions, modules, data types, or whatever you can come up with.

```
x = 1
```

```
σ = 5
```

```
# Variable names can include practically any Unicode character. Even using LaTeX syntax
σ = 5 # type \sigma and then tab!
```

- You can assign multiple variables to multiple values using commas.

```
(1, 0, -1)
```

```
😺, 😄, 😢 = 1, 0, -1
# to use emojis type \: then tab to show list of available emojis.
```

```
true
```

```
😺 + 😢 == 😄
# legitimate code. Not good for readability though ;)
```

- Strings are created between double quotes " while the single quotes ' are used for characters only.

```
md"- Strings are created between double quotes \" while the single quotes ' are used for characters only."
```

```
س = "بسم الله"
```

```
بسم الله" = س"
```

```
y = 'ش': Unicode U+0636 (category Lo: Letter, other)
```

```
y = 'ش'
```

```
ص = 3
```

```
3=ص

```

```
6
```

```
3 + ص
```

- Since assignment returns the value, by default this value is printed. This is AMAZING, but you can also silence printing by adding ; to the end of the expression:

```
1 z = 4;
```

- you can interpolate any expression into a string using $(expression)

```
"The value of the crying face (😿) is 1"
```

```
1 "The value of the crying face (😿) is $(😿)"
```

```
"You can do math inside a string: 1.0"
```

```
1 "You can do math inside a string: $(cos(π) + 2)"
```

Note on Pluto

Simple, reactive programming environment for Julia

Workspace variables Pluto offers an environment where changed code takes effect instantly and where deleted code leaves no trace. Unlike Jupyter or Matlab, there is no mutable workspace, but rather, an important guarantee:

At any instant, the program state is completely described by the code you see.

No hidden state, no hidden bugs.

Pluto doesn't support multiple expressions per cell. This is a conscious choice - this restriction helps you write less buggy code once you get used to it. To fix the code, you can either split the above cell, or wrap it in a begin ... end block. Both work.

```
x = 4
```

```
1 x = 4
```

```
2
```

```
1 let
2     x = 3
3     y = 2
4 end
```

Multiple expressions in one cell.
How would you like to fix it?

- Split this cell into 2 cells, or
- Wrap all code in a *begin ... end* block.

1. **top-level scope** @ *none:1*
2. **top-level scope** @ *none:1*

```
1     x +2
2     y + 20
3
```

```
1 Enter cell code...
```

```
1 Enter cell code...
```

Math Operations

Basic math operators are +, -, *, / and ^ for power.

```
10.04510856630514
```

```
begin
    a = 3
    b = a^2.1
    # Most julia operators have their = version, which updates something with its own value
    # a += 3
    # a -= 3
    # a *= 3
    # a /= 3
end
```

```
19.99823744
```

```
let
    # Literal numbers can multiply anything without having to put * inbetween, as long as the number is on the left side:
    5x - 12.24a * 1.2e-5x
end
```

Type Basics

Everything that exists in Julia has a certain Type. (e.g. numbers can be integers, floats, rationals). This "type system" is instrumental for the inner workings of Julia, and is mainly what enables Julia to have performance matching static languages like C.

The type system also enables Multiple Dispatch, one of Julia's greatest features, which we will cover in the second lecture.

To find the type of a thing in Julia you simply use `typeof(thing)`:

```
Float32
```

```
let
    # Integers
    # Floats
    # Strings
    # Characters
    x = 1.5f0
    typeof(x)
    # typemax(x)
    # typemax(Int64)
    # eps(1.)
end
```

Basic collection datastructures

- Index a collection with brakets `A[1]`.
- indexing in Julia starts from 1

Tuples

Tuples are immutable ordered collections of elements of any type. They are most useful when the elements are not of the same type with each other and are intended only for small collections.

```
Tuple{String, Cmd, Irrational{:π}, Symbol}
```

```
let
    mythings = ("blue", `🧁`, π, :math)
    mythings[4]
    # You can extract multiple values into variables from any collection using commas.
    mycolor, mycake,_, mysubject = mythings
    mycolor, mycake, mysubject
    # The type of the tuple is the type of its constituents.
    typeof(mythings)
end
```

Dictionaries

Dictionaries are unordered mutable collections of pairs key-value. They are intended for sets of relational data, and typically you want the data to be of the same type. Syntax:

```
Dict(key1 => value1, key2 => value2, ...)
```

A good example of a dictionary is a contacts list, where we associate names with phone numbers.

```
Dict{String, String}
```

```
begin
    myphonebook = Dict("Ali" => "860-2931", "Mohammed" => "860-7748")
    # myphonebook["Ali"]
    # New entries can be added to the above dictionary, because it is mutable
    myphonebook["Khalid"]="860-4415"
    myphonebook
    typeof(myphonebook)
end
```

Named tuples

These are exactly like tuples but also assign a name to each variable they contain. Hence, they are an immutable collection of ordered and named elements. They rest between the Tuple and Dict type in their use.

Their syntax is:

```
(key1 = val1, key2 = val2, ...)
```

For example:

```
:math
```

```
let
    mythings = (color="blue", cake=`🧁`, number=π, subject=:math)
    mythings[4]
    mythings[:subject]
    mythings.subject

    # mycolor, mycake,_, mysubject = mythings
    # mycolor, mycake, mysubject
    # # The type of the tuple is the type of its constituents.
    # typeof(mythings)
end
```

Arrays

The standard Julia Array is a **mutable and ordered collection of items of the same type**. The dimensionality of the Julia array is important. A **Matrix** is an array of dimension 2. A **Vector** is an array of dimension 1. The element type of what an array contains is irrelevant to its dimension!

i.e. a Vector of Vectors of Numbers and a Matrix of Numbers are two totally different things!

The syntax to make a vector is enclosing elements in brackets

```
▸ [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

```
let
    prime_numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
    # typeof(prime_numbers)
    # prime_numbers[end]=29
    # push!(prime_numbers,31)
    pop!(prime_numbers);prime_numbers

end
```

```
Vector{Any} (alias for Array{Any, 1})
```

```
let
    mixture = [1, 1, 2, 3, "Norah", "Ali"]
    typeof(mixture)
end
```

As mentioned, the type of the elements of an array must be the same. Yet above we mix numbers with strings! I wasn't lying though; the above vector is an unoptimized version that can hold any thing. You can see this in the type of the vector, `Vector{Any}`.

Arrays of other data structures, e.g. vectors or dictionaries, or anything, as well as multi-dimensional arrays are possible:

- If you want to make a matrix, two ways are the most common: (1) specify each entry one by one

```
md"""
As mentioned, the type of the elements of an array must be the same. Yet above we mix numbers with strings! I wasn't lying though; the above vector is an unoptimized version that can hold any thing. You can see this in the type of the vector, `Vector{Any}`.

Arrays of other data structures, e.g. vectors or dictionaries, or anything, as well as multi-dimensional arrays are possible:

- If you want to make a matrix, two ways are the most common: (1) specify each entry one by one
"""
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

```
let
    matrix = [1 2 3; # elements in same row separated by space
          4 5 6; # semicolon means "go to next row"
          7 8 9]
    typeof(matrix)
end
```

- or (2), you use a function that initializes a matrix. E.g. `rand(n, m)` will create an n×m matrix with uniformly random numbers

```
md"""
- or (2), you use a function that initializes a matrix. E.g. `rand(n, m)` will create an n×m matrix with uniformly random numbers
"""

```

```
0.8109921897371506
```

```
let
    R = rand(4,3)
    R[1,2]

end
```

- Lastly, for multidimensional arrays, the : symbol is useful, which means to "select all elements in this dimension".

```
md"""
- Lastly, for multidimensional arrays, the : symbol is useful, which means to "select all elements in this dimension".
"""
```

```
▸ [0.84188, 0.734578, 0.975231, 0.0842755]
```

```
let
    R = rand(4,6)
    R[:,1] # it means to select the first column
end
```

Ranges

Ranges are useful shorthand notations that define a "vector" (one dimensional array) with equi-spaced entries. They are created with the following syntax:

```
start:stop # mainly for integers
start:step:stop
range(start, stop, length)
range(start, stop; step = ...)
```

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}, Int64}
```

```
let
    r = 0:0.1:5
    typeof(r)
end
```

- Ranges always include the first element and step until they do not exceed the ending element. If possible, they include the stop element (as above).

```
4.8
```

```
let
    r = 0:0.1:5
    r[end-2] # use `end` as index for the final element
    # r[2:10:end] |> collect
end
```

- Ranges are not unique to numeric data, and can be used with anything that extends their interface, e.g.

```
▸ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', '
```

```
let
    letterragnge = 'a':'z'
    letterragnge[2]
    # As ranges are printed in this short form, to see all their elements you can use
    collect, to transform the range into a Vector.
    collect(letterragnge)
end
```

Ranges are cool because they do not store all elements in memory like Vectors. Instead they produce the elements on the fly when necessary, and therefore are in general preferred over Vectors if the data is equi-spaced.

- Lastly, ranges are typically used to index into arrays. One can type `A[1:3]` to get the first `3` elements of `A`, or `A[end-2:end]` to get the last three elements of `A`. If `A` is multidimensional, the same type of indexing can be done for any dimension:

▸ [5, 4, 8]

```
let
    # rng = MersenneTwister(123)
    A = rand(0:9,4,4) # creates a random 4 x 4 matrix from the range given
    display(A)
    A[2:4,1]
end
```

```
4×4 Matrix{Int64}:
 1  4  4  1
 5  0  0  2
 4  1  0  2
 8  3  4  0
```

Iteration

Iteration in Julia is high-level. This means that not only it has an intuitive and simple syntax, but also iteration works with anything that can be iterated. Iteration can also be extended

`for` loops

A `for` loop iterates over a container and executes a piece of code, until the iteration has gone through all the elements of the container. The syntax for a for loop is

```
for *var(s)* in *loop iterable*
    *loop body*
end
```

you will notice that all Julia code-blocks end with end

```
for n in 1:5
    println(n)
end
```

```
1
2
3
4
5
```

- The nature of the iterating variable depends on what the iterating container has. For example, when iterating over a dictionary one iterates over pairs of key-value.

```
for pair in myphonebook
    println(pair)
end
```

```
"Ali" => "860-2931"
"Mohammed" => "860-7748"
"Khalid" => "860-4415"
```

```
for (key,val) in myphonebook
    println("The number of $key is $val")
end
```

```
The number of Ali is 860-2931
The number of Mohammed is 860-7748
The number of Khalid is 860-4415
```

- In the context of for loops, the enumerate iterator is often useful. It takes in an iterable and returns pairs of the index and the iterable value.

```
for (i, v) in enumerate(rand(3))
    println("value of index $i is ($v)")
end
```

```
value of index 1 is (0.24274588179092793)
value of index 2 is (0.5329753949915015)
value of index 3 is (0.9549335575750545)
```

command termination

Julia has a modern syntax parser that automatically understands when a command starts and ends. It does not require identation (like `Python`) or the `;` character (like `C`) to establish the end of a command. The following two are totally valid and syntactically identical Julia codes

```
        for (key,
val) in
                myphonebook
  println(
"The number of $key is
$val"
        ) end

```

```
The number of Ali is
860-2931
The number of Mohammed is
860-7748
The number of Khalid is
860-4415
```

```
for (key, val) in myphonebook println("The number of $key is $val") end
```

```
The number of Ali is 860-2931
The number of Mohammed is 860-7748
The number of Khalid is 860-4415
```

However, code readability is important so it is strongly recommended to properly ident your code even if it is not enforced by the language!

`while` loops

A `while` loop executes a code block until a boolean condition check (that happens at the start of the block) becomes `false`. Then the loop terminates (without executing the block again). The syntax for a standard `while` loop is

```
while *condition*
    *loop body*
end
```

```
md"""
### `while` loops
A `while` loop executes a code block until a boolean condition check (that happens at the start of the block) becomes `false`. Then the loop terminates (without executing the block again). The syntax for a standard `while` loop is
```julia
while *condition*
    *loop body*
end
```
"""
```

```
let
    n = 0
    while n < 5
        n += 1
        println(n)
    end
end
```

```
1
2
3
4
5
```

Conditionals

Conditionals execute a specific code block depending on what is the outcome of a given boolean check. The `&`, `|` are the boolean `and`, `or` operators.

`if` block

In Julia, the syntax

```
if *condition 1*
    *option 1*
elseif *condition 2*
    *option 2*
else
    *option 3*
end
```

evaluates the conditions sequentially and executes the code-block of the first true condition.

```
md"""
## Conditionals
Conditionals execute a specific code block depending on what is the outcome of a given boolean check. The `&`, `|` are the boolean `and`, `or` operators.

### `if` block
In Julia, the syntax
```julia
if *condition 1*
    *option 1*
elseif *condition 2*
    *option 2*
else
    *option 3*
end
```

evaluates the conditions sequentially and executes the code-block of the first true condition.
"""
```

6

```
let
    x, y = 5, 6
    if x > y
        x
    else
        y
    end
end
```

Ternary operator

The ternary operator (named for having three arguments) is a convenience syntax for small `if` blocks with only two clauses.

Specifically, the syntax

```
condition ? if_true : if_false
```

is syntactically equivalent to

```
if condition
    if_true
else
    if_false
end
```

For example

```
"yes"
```

```
5 == 5.0 ? "yes" : "no"
```

break and continue

The keywords `continue` and `break` are often used with conditionals to skip an iteration or completely stop the iteration code block.

```
let
    N = 1:100
    for n in N
        isodd(n) && continue
        println(n)
        n > 10 && break
    end
end
```

```
2
4
6
8
10
12
```

List comprehension

The list comprehension syntax

```
[expression(a) for a in collection if condition(a)]
```

is available as a convenience way to make a `Vector`. The `if part` is optional.

```
md"""
## List comprehension
The list comprehension syntax
```julia
[expression(a) for a in collection if condition(a)]
```
is available as a convenience way to make a `Vector`. The `if part` is optional.

"""
```

```
[4, 16, 36, 64, 100]
```

```
[    a^2 for a in 1:10 if iseven(a)      ]
```

Functions

Functions are the bread and butter of Julia, which heavily supports functional programming.

Function declaration

Functions are declared with two ways. First, the verbose

4

```
begin
    # First, the verbose
    function f1(x)
        # function body
        return x^2 # While `return` is not necessary, it is recommended for clarity
    end
    # Or, you can define functions with the short form (best used for functions that only take up a single line of code)
    f2(x) = x^2
    # Functions are called using their name and parenthesis () enclosing the calling arguments:
    f2(2)
end
```

- Functions in Julia support optional positional arguments, as well as keyword arguments. The **positional** arguments are always given by their order, while **keyword** arguments are **always given by their keyword**. Keyword arguments are all the arguments defined in a function after the symbol ;. Example:

```
md"""
- Functions in Julia support optional positional arguments, as well as keyword arguments. The __positional__ arguments are always given by their order, while __keyword__ arguments are __always given by their keyword__. Keyword arguments are all the arguments defined in a function after the symbol ;. Example:
"""
```

60

```
let
    function g(x, y = 5; z = 2, w = 1)
        return x*z*y*w
    end
    g(5,4,w=3, z=1)
end
```

Passing by reference: mutating vs. non-mutating functions

You can divide Julia variables into two categories:

- **Mutable**: meaning that the values of your data can be changed in-place, i.e. literally in the place in memory the variable is stored in the computer.
- **Immutable**: immutable data cannot be changed after creation, and thus the only way to change part of immutable data is to actually make a brand new immutable object from scratch.

Use isimmutable(v) to check if value v is immutable or not.

For example, Vectors are mutable in Julia:

```
true
```

```
let
    x = [5, 5, 5]
    x[1] = 6 # change first entry of x
    x
    y = (5, 5, 5)
    isimmutable(y)
    # y[1] = 6
end
```

- Julia **passes values by reference**. This means that if a mutable object is given to a function, and this object is mutated inside the function, the final result is kept at the passed object. E.g.:

```
▸ [8, 5, 5]
```

```
let
    function add3!(x)
        x[1] += 3
        return x
    end

    x = [5, 5, 5]
    # y = (5, 5, 5)
    add3!(x)
    x
end
```

- **By convention**, functions with name ending in ! alter their (mutable) arguments and functions lacking ! do not. Typically the first argument of a function that ends in ! is mutated.

For example, let's look at the difference between sort and sort!.

```
▸ ([2, 3, 5], [2, 3, 5])
```

```
let
    v = [3, 5, 2]
    # sort(v),  v
    sort!(v), v
end
```

Functions as arguments

Functions, like literally anything else in Julia, are objects that can be passed around like any other value. Including giving them as arguments to other functions.

A typical application of this is with the `findall` and related functions, that find the indices of all elements in a collection that return `true` for a particular expression.

```
▸ [1, 2, 3, 4, 5, 17, 18, 19, 20, 21]
```

```
let
    expression(x) = (x < 0.5) | (x > 1.5)
    x = 0:0.1:2
    valid_indices = findall(expression, x)
end
```

The help system

Typing ? followed by a function (or type) name will display its documentation string. Alternatively you can type @doc and then the function name.

For example

```
# @doc findall
```

Broadcasting

Broadcasting is a convenient syntax for applying any function over the elements of an iterable input. I.e., the result is a new iterable whose elements is the function application of the elements of the input.

Broadcasting is done via the simple syntax of adding a dot . before the parenthesis in the function call: `g.(x)`.

```
3×3 Matrix{Int64}:
 2  1  1
 1  2  1
 1  1  2
```

```
let
    h(x, y = 1) = x + y
    x = [1,2,3]
    t = (1,2,3)
    m = I(3)
    h.(m,1)
end
```

Exercises - basics

- Important note for all exercises: when an exercise says "use function function_name to do something", you need to first learn how to use the function. For this, you access the function's documentation string, using the help mode (type ? or @doc in the Julia console and then type the function name)!

Babylonian square root

To get the square root of y Babylonians used the algorithm

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right)$$

iteratively starting from some value x_0. Then $x_n \to \sqrt{y}$ as $n \to \infty$.

Implement this algorithm in a function `babylonian(y, ε, x0 = 1)` (default optional argument `x0`), that takes some convergence tolerance `ε` to compare with the built-in `sqrt(y)`. The function should return the steps it took to reach the square root value within given tolerance.

Counting nucleotides

Create a function that given a DNA strand (as a `String`, e.g. `"AGAGAGATCCCTTA"`) it counts how much of each nucleotide (A G T or C) is present in the strand and returns the result as a dictionary mapping the nucleotides to their counts. The function should throw an error (using the `error` function) if an invalid nucleotide is encountered. Test your result with `"ATATATAGGCCAX"` and `"ATATATAGGCCAA"`.

Hint: Strings are iterables! They iterate over the characters they contain.

Fibonacci numbers

Using recursion (a function that calls itself) create a function that given an integer `n` it returns the `n-th` Fibonacci number. Apply it using map to the range 1:8 to get the result [1,1,2,3,5,8,13].

Hamming distance

Create a function that calculates the Hamming distance of two equal DNA strands, given as strings. This distance is defined by counting (sequentially) the number of non-equal letters in the two strands, e.g. "ATA" and "ATC" have distance of 1, while "ATC" and "CAT" have distance of 3.

Hint: this exercise has a one-liner solution, using the zip and count functions.

- Plots
- User-defined types
- Linear Algebra & Differential Equations
- Numeical Methods
- Data Science
- Optimization

Plots

Source: Julia Programming: A Hands-On Tutorial by Martín D. Maas

Plots.jl

`Plots.jl` library is considered the standard plotting tool in the Julia ecosystem. It provides a single API to access multiple "backends", which include:

- Matplotlib (Pyplot)
- Plotly
- GR

To select one of these different backends, we simply call the corresponding command. For example, to load `Plots.jl` and use the `GR` backend (which is the default), we would do:

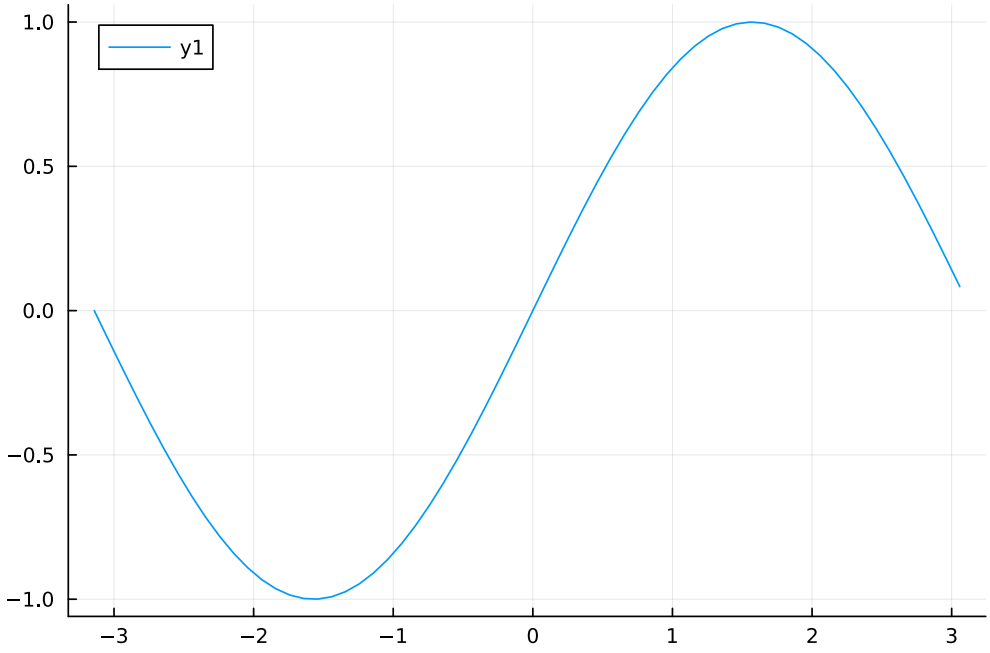
```
using Pkg
Pkg.add("Plots")
using Plots
gr()
```

Other plotting packages for Julia

- StatsPlots.jl
- Makie.jl
- VegaLite
- Gadfly

Simple Plots

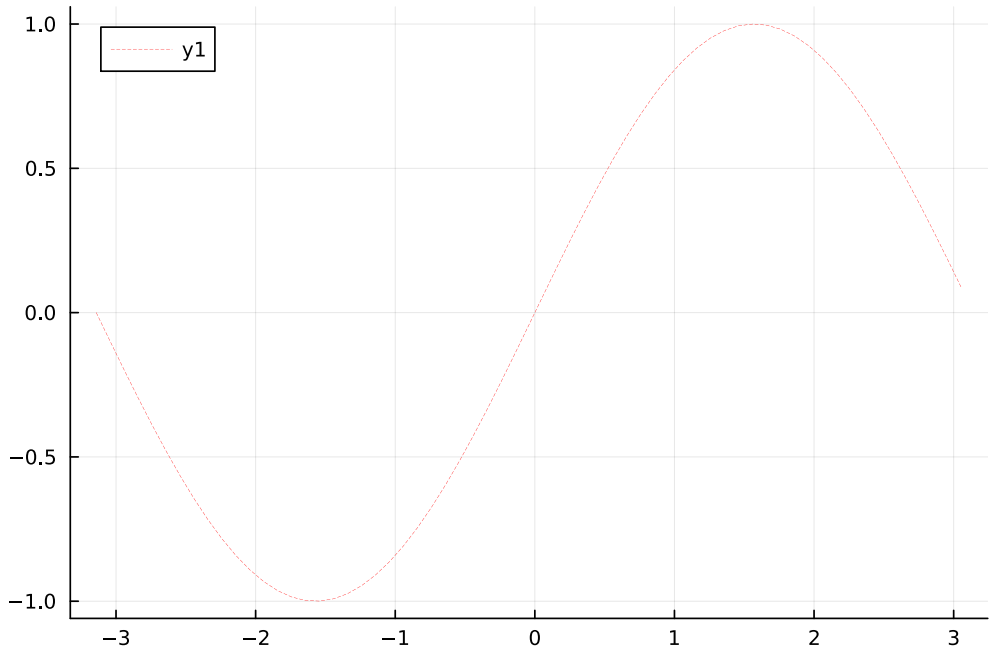
Let's plot the function $f(x) = \sin(x)$ over the interval $[-\pi, \pi]$.




```
let
    f(x) = sin(x)
    x = -π:0.1:π
    # plot(f)
    # plot(ϵ->sin(ϵ))
    plot(x,f.(x))
end
```

Lines Styles and Markers

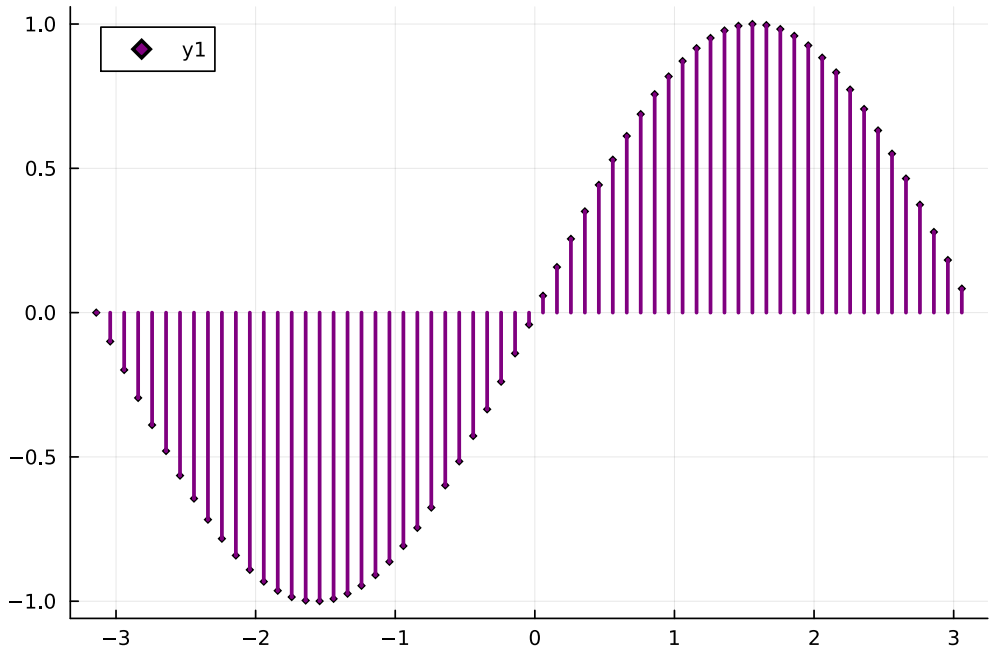
Series attributes	Type
linestyle	:solid, :dash, :dot, :dashdot, :dashdotdot
lw	Float
seriescolor	Color Type
seriestype	:path, :sticks, :scatter, :bar




```
let
    f(x) = sin(x)
    x = -π:0.1:π
    y = f.(x)
    plot(
        x,y,
        linestyle=:dash,
        seriescolor=:red,
        lw=0.2,
        seriestype=:path
    )
end
```

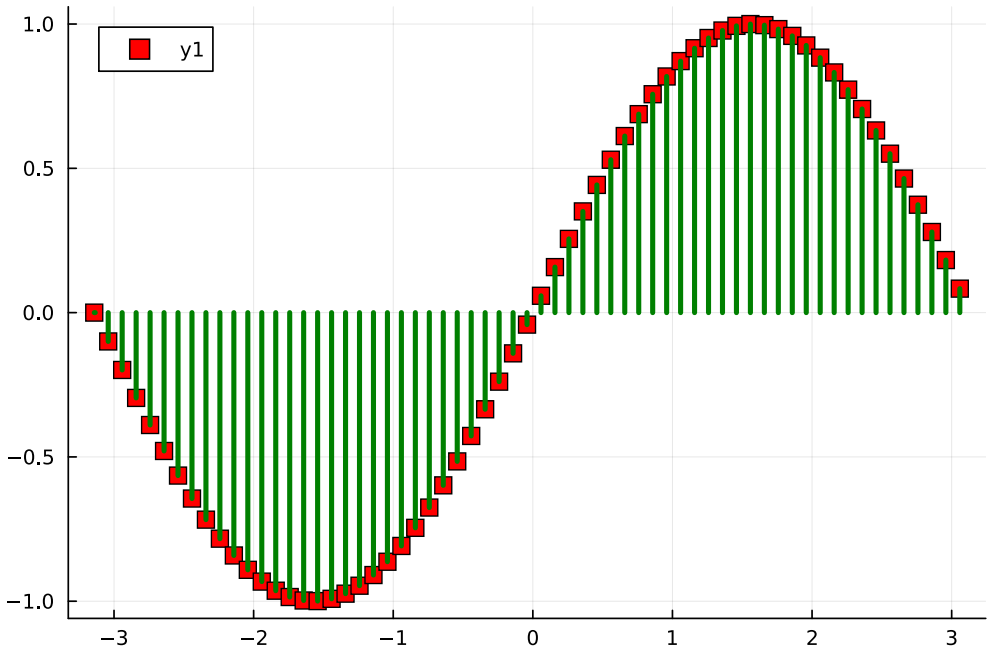
Markers

Series attributes	Type
marker	:d, :hex,
markersize	Float
markerstrokecolor	Color Type




```
let
    f(x) = sin(x)
    x = -π:0.1:π
    y = f.(x)
    plot(
        x,y,
        seriescolor=:purple,
        lw=2.2,
        seriestype=:sticks,
        marker=:diamond, # m alias
        markersize=2
    )
end
```

Alternatively we may define a line and a marker as tuples. For examle




```
let
    f(x) = sin(x)
    x = -π:0.1:π
    y = f.(x)
    plot(
        x,y,
        line=(:sticks,3,:green), # order does not matter.
        marker=(:square,:red,5)
    )
end
```

Text, Font & LaTeX

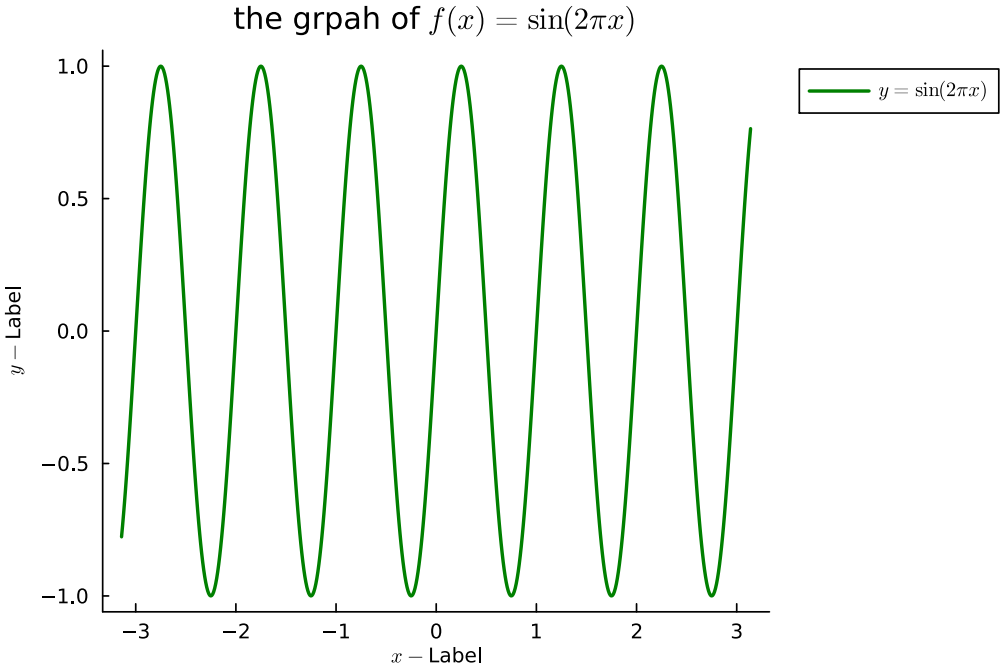
Subplot Attribute	Type
title	string
titlefontsize	Int
xlabel	string
ylabel	string
guidefontsize	Int
label	string
legendfontsize	Int
tickfontsize	Int

We can use LaTeX in titles and labels with the package `LaTeXStrings`. Just use the macro literal `L"latex expression"`

Some possible values for the legend position are:

- :right
- :left
- :top
- :bottom
- :topright

- :topleft
- :bottomright
- :bottomleft
- :outerright
- :outerleft
- :outertopright
- :outertopleft
- :outerbottomright
- :outerbottomleft

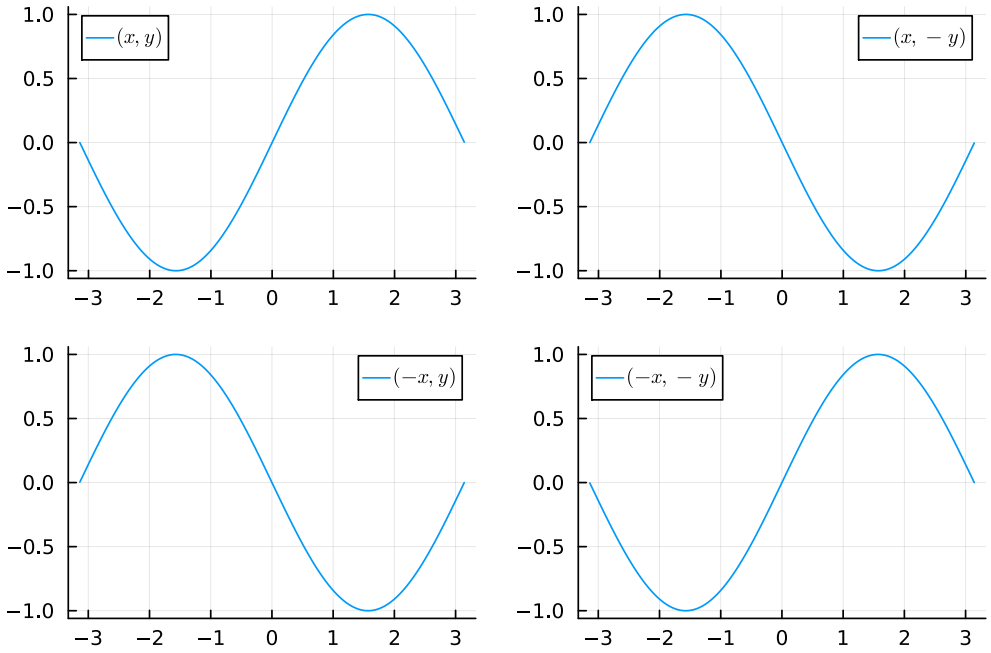



```
let
    f(x) = sin(2π*x)
    x = -π:0.01:π
    y = f.(x)
    plot(
        x,y,
        line=(:path,2,:green), # order does not matter.
        # marker=(:square,:red,5),
        title="the grpah of $(L"f(x)=\sin(2\pi x)")",
        titlefontsize=12,
        label=L"y=\sin(2\pi x)",
        legend=:outertopright,
        xlabel=L"x-"*"Label",
        ylabel=L"y-"*"Label",
        labelfontsize=8,
        legendfontsize=8, #  guidefontsize=8, #for both
        grid=false,
        # ylimits=(-2,2)

    )
end
```

Subplots

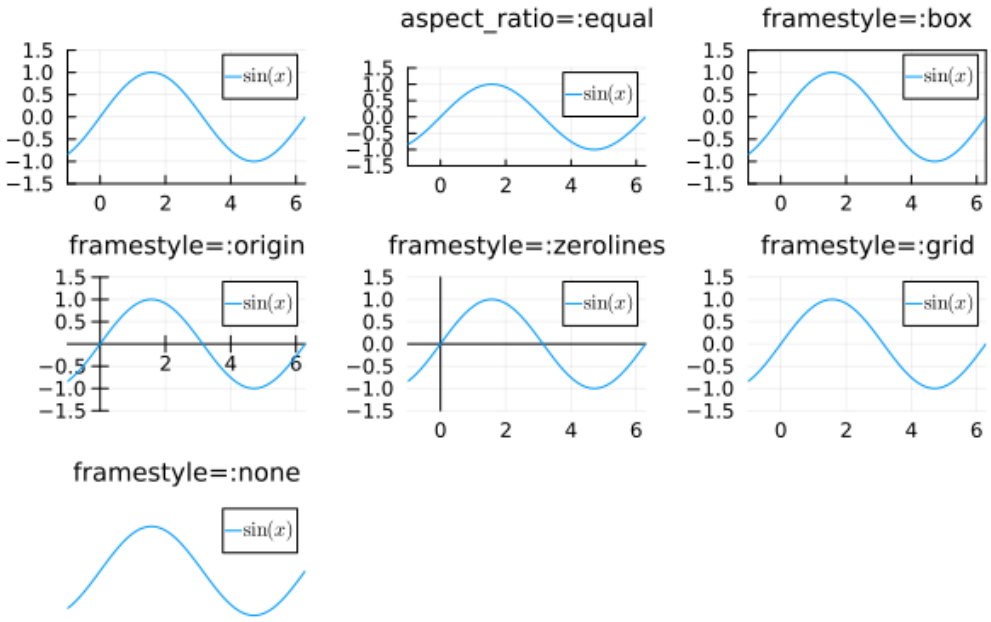
We can use `layout` to produce multiple plots in one figure. There many ways to produce multiple plots. We give an example of here (see the `Plots.jl` documentation under layout for more).




```
let
    f(x) = sin(x)
    x = -π:0.01:π
    y = f.(x)
    p1 = plot(x,y, label=L"(x,y)")
    p2 = plot(x,-y, label=L"(x,-y)", legend=:topright)
    p3 = plot(-x,y, label=L"(-x,y)", legend=:topright)
    p4 = plot(-x,-y, label=L"(-x,-y)")
    plot(p1,p2,p3,p4, layout=(2,2))
end
```

Axes: Limits, Aspect Ratio & Frames

- To control the feasible part of the axes, we use `xlimits` and `ylimits` for the x—axis and the y—axis respectively.
- To control the scales of the axes, we use `aspect_raio`.
- To control the figure frame and the location of the axis, we use `framestyle`.

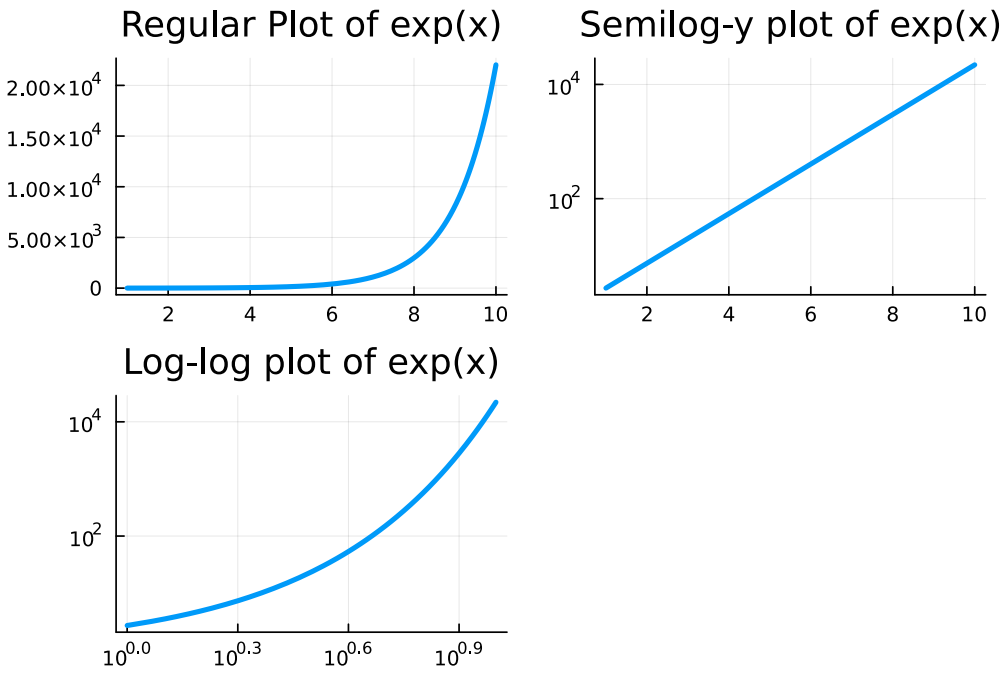



```
let
    f(x) = sin(x)
    x = -π:0.01:3π
    y = f.(x)
    p1 = plot(x,y,
            xlimits=(-1,2π),
            ylimits=(-1.5,1.5),
            label=L"\sin(x)",
            titlefontsize=10
        )
    p2= plot(p1,aspect_ratio=:equal, title="aspect_ratio=:equal")
    p3= plot(p1,framestyle=:box, title="framestyle=:box")
    p4= plot(p1,framestyle=:origin, title="framestyle=:origin")
    p5= plot(p1,framestyle=:zerolines, title="framestyle=:zerolines")
    p6= plot(p1,framestyle=:grid, title="framestyle=:grid")
    p7= plot(p1,framestyle=:none, title="framestyle=:none")

    p = plot(p1,p2,p3,p4,p5,p6,p7, layout=8)
end
```

Log and Semilog plots

We use `xscale=:log10` and/or `yscale=:log10` to set one of both axes to log scale. For example



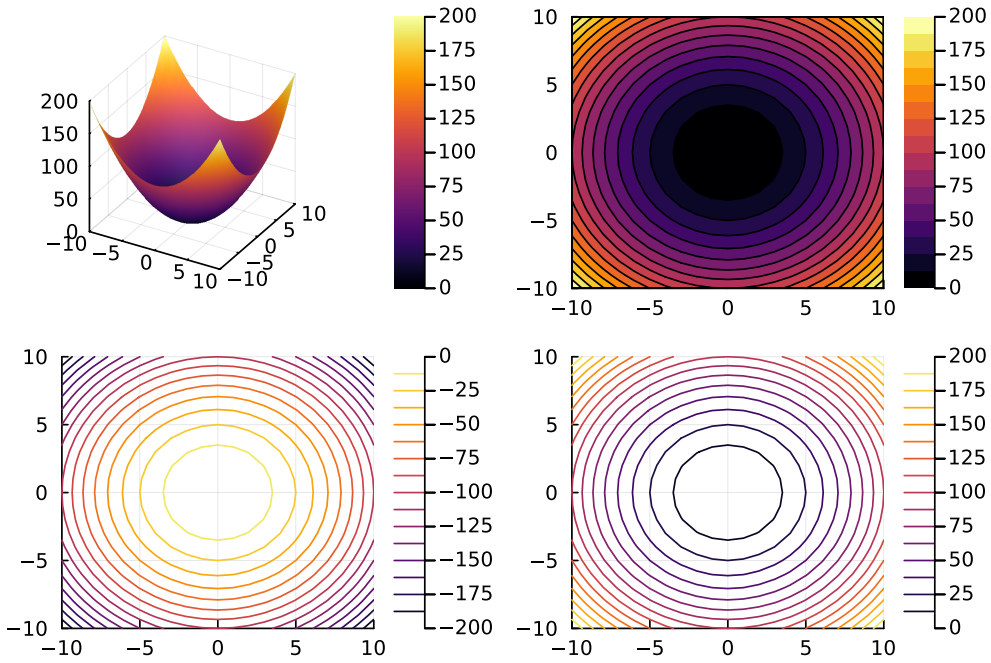
3D and Cantours

3D plots work in a similar way as 2D plots, we need to provide `x`, `y` and `z` values. Here is an example.

To plot

$$f(x, y) = x^2 + y^2$$

we may use `plot` with `linetype=:surface` or the shortcut `surface`. For countours we use `countour`



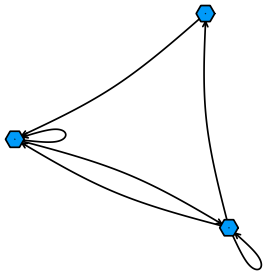
```
let
    f(x,y) = x^2 + y^2
    x = -10:10
    y = x
    # either using mesh-like
    X = repeat(x',length(y),1)
    Y = repeat(y,1,length(x))
    z = f.(X,Y)
    surface(x, y, z)
    # or directly as
    p0=surface(x, y, f)
    # plot(x, y, f, linetype=:surface)
    p1 = contour(x, y, f, fill = true)
    p2 = contour(x, y, (x,y)->-f(x,y))
    p3 = plot(x, y, z, linetype=:contour)
    plot(p0,p1,p2,p3, layout=4)
end
```

Plot Recipes

Recipes are a way of defining visualizations in your own packages and code, without having to depend on Plots. Here we give an example with plotting graphs (as in Graph theory).

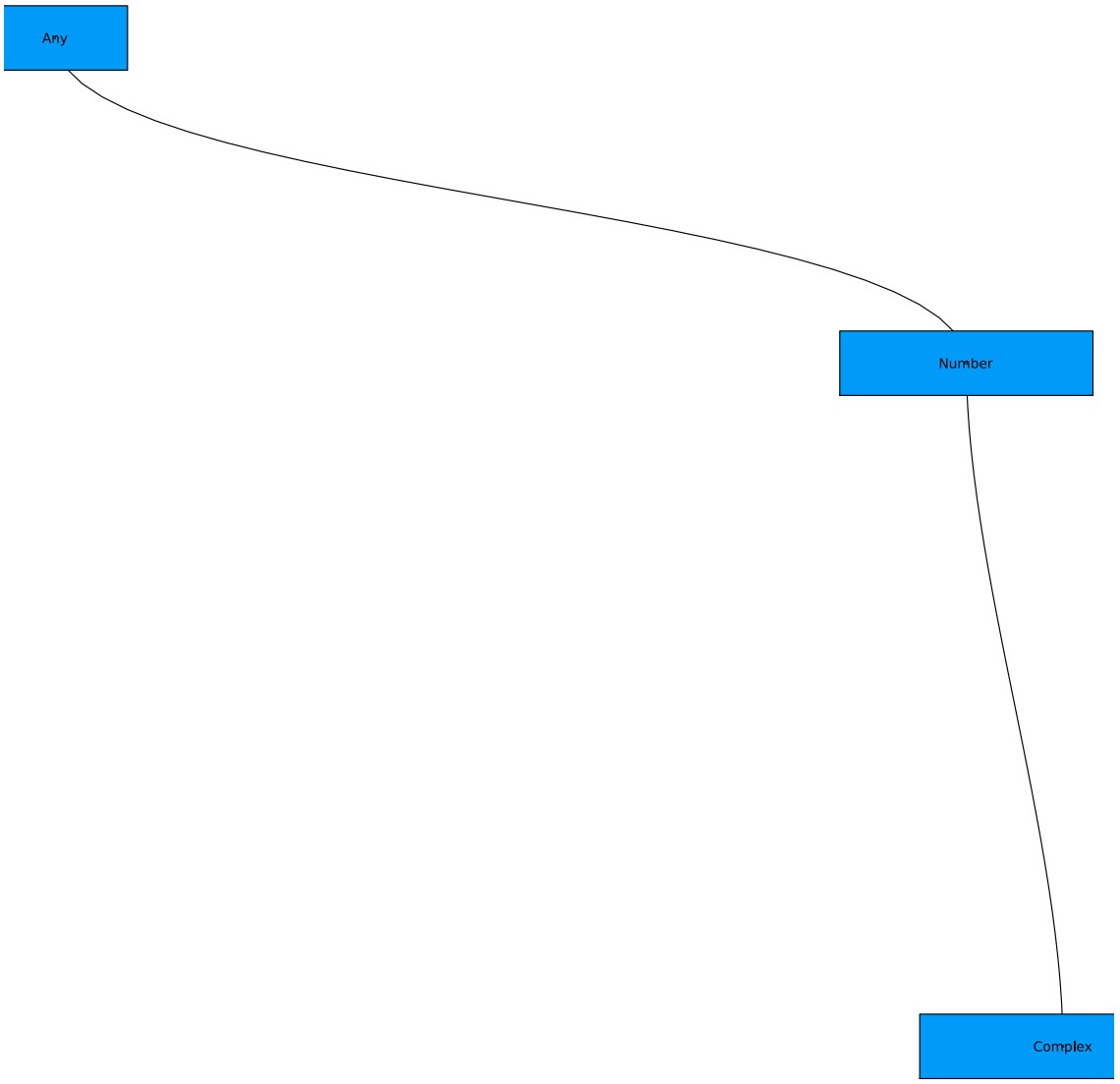
Example: Draw the DiGraph whose adjacency matrix is

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$



```
let
    # using Graphs and GraphRecipes
    g = [1 1 1;
        0 0 1;
        1 0 1]
    graphplot(DiGraph(g), self_edge_size=0.2, size=(200,200))
end
```

Example: Draw the type tree of julia type `Integer`




```
let
    plot(Complex,
            method=:tree,
            fontsize=8,
            axis_buffer=0.05,
            nodeshape=:rect,
            nodesize=0.12,
            # nodecolor=:purple,
            # shorten=0.02,
        size=(1000, 1000))
end
```

Julia Type System

In Julia every element has a type. The type system is a hierarchical structure: at the top of the tree there is the type `Any`, which means that every element belongs to it, then there are many other subtypes, for example `Number` which includes `Real` and `Complex`, and `Real` contains for example `Int` (integer) numbers and `Float64` numbers.

The functions `subtypes` and `supertype` show the children or the parent of a node in this tree.

▸ [Bool, OffsetInteger, OffsetInteger, UpperBoundedInteger, ChainedVectorIndex, Signed, Unsi

```
begin
    subtypes(Integer)
    # supertypes(Integer)
end
```

You can also use the operator <: to check if something is a "subtype of" something else, for example

```
cm"""You can also use the operator <: to check if something is a "subtype of"
something else, for example"""
```

true

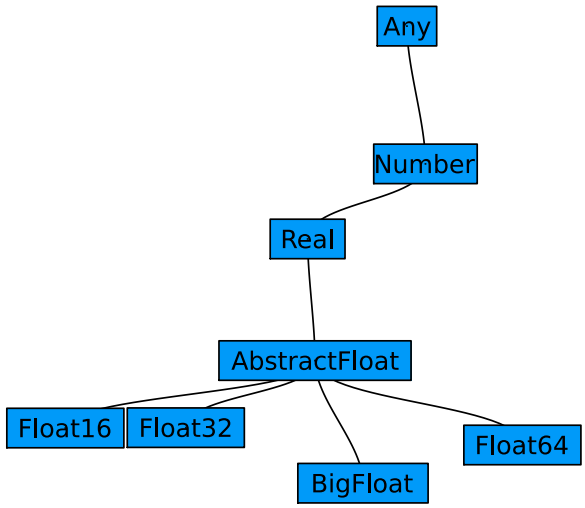
```
Float32 <: Number
```

Abstract vs Concrete types

- Concrete types are anything that can be actually instantiated. Any value that exists in Julia code that is running always has a concrete type.
- Abstract types exist only to establish hierarchical relations between the concrete types. At the moment we don't have a reason for hierarchical relations, but once we talk about **Multiple Dispatch** this reason will become apparent.

> "In terms of **mathematical reasoning**, the distinction between abstract and concrete is simple: concrete types are leaves of the Type tree, while abstract types are anything else.

Based on the below tree of the Number type, one can see that Number, Real, AbstractFloat are all abstract types, while Float64, Float32 are concrete.




```
begin
    plot(AbstractFloat, method=:tree,
            fontsize=10,
            # axis_buffer=0.05,
            nodeshape=:rect,
            # nodesize=0.12,
            # nodecolor=:purple,
            # shorten=0.02,
        # size=(500, 500)
    )
end
```

Custom Types

We can of course, create our own Types. We can do this using `struct` and `mutable struct` keywords.

Types then are instantiated by calling them as if they are functions, with arguments their fields (this is the "default" constructor, and one can make any arbitrary constructor they want).

```
▸ B(3, 2)
```

```
let
    struct A
        x
        y
    end
    mutable struct B
        x
        y
    end
    a = A(1,2)
    # a.x = 3
    b = B(1,2)
    b.x = 3
    b
end
```

- The way we defined A, B here is **bad for performance**.
  - The fields x, y can have any possible Type, as we added no restrictions to them. This means that once a function gets an instance of A, the compiler cannot optimize this function, as it is impossible to know by the type A what are the actual data structures involved.
- To avoid this, use the type assertion operator :: to enforce a Type to the fields of a struct definition.

```
struct Agood
    x::Float64
    y::Float64
    label::String
end
```

Abstract Types

- As we specified no subtyping rule for `A, B`, they are by definition subtypes of the central node of the tree, `Any`.
- Abstract types are meaningful for subtyping relations.
- Abstract types are defined with the **`abstract type`** keyword

```
abstract type C end
```

```
MethodError: no constructors have been defined for Main.var"workspace#3".C
```

1. **top-level scope** @ *Local: 1* [inlined]

```
C(5)
```

- To make `C` useful, we can create some subtypes of `C` using the subtyping operator **`<:`**

```
struct D <: C
    d::Number
end
```

▸ (D(5), D(1.5), D(1.1+2.2im))

```
D(5),D(1.5), D(1.1+2.2im)
```

Now `D` is a subtype of `C`:

```
cm" Now `D` is a subtype of `C`:"
```

▸ [D]

```
subtypes(C)
```

Parametric Types

Julia types can be parameterized based on other types, meaning that you can indeed make a field of a type to be "anything that is a number", and still be type stable!

This is very useful for reducing code replication as well as leveraging multiple dispatch more. Type-parameterization means that when defining a struct, the fields of the struct could be of an arbitrary Type. Type-parameterization uses the curly brackets syntax. For example:

```
struct Alpha{T}
    a::T
    b::T
end

```

- Above `Alpha` has two fields `a` and `b` with parametric type `T`. `T` here can be any type. So we may instantiate `Alpha` as `Alpha{Int64}(value1, value2)` and that we create an object of type `Alpha` whose fields are of type `Int64`.

▸ Alpha(1, 2)

```
let
    a = Alpha{Int64}(1,2)
end
```

- The high level constructor Alpha(val1, val2) also exists, and will try to deduce what Type should be based on the arguments:

▸ Alpha(1.2, 2.0)

```
let
    a = Alpha(1.2,2.0)
end
```

- In some case we may need that each field has its own type. For example

```
struct Beta{T1, T2}
    a::T1
    b::T2
end
```

▸ Beta(1, "Mohammed")

```
Beta(1,"Mohammed")
```

▸ Beta(2.0, 1 + 1im)

```
Beta(2.0,1+im)
```

- Here is another more complex example

```
struct Gamma{T1 <: Real, T2 <: Union{String, Complex}}
    a:: T1
    b:: T2
    c::Int
end
```

▸ (Gamma(1, "Hi", 5), Gamma(1, 1+1im, 5))

```
Gamma(1,"Hi",5), Gamma(1,1+im,5)
```

Multiple Dispatch

Multiple dispatch is the core programming paradigm of Julia. This notebook presents simple examples that can teach you multiple dispatch.

What is Multiple Dispatch?

We should distinguish between two terms.

- **function**: The name of the function or process we refer to. In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, because they can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

```
function f(x,y)
   x + y
end
```

- **method**: what actually happens when we call the function with a specific combination of input arguments.It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the + function.

- **Dispatch** means that when a function call occurs, the language decides somehow which of the function methods have to be used.
- **No dispatch**: In no dispatch, as in e.g. C, there is nothing to be decided. The method and the function are one and the same.
- **single dispatch**, as in most object-oriented languages (like Python), it is possible for the same function name to have different methods:

```
array.set_size(args...)
axis.set_size(args...)
```

- where `array` could be an instance of something from `numpy` while `axis` could come from `matplotlib`. Here the language dispatches the function `set_size`, depending on the first argument, which is `array` or `axis`. It is important to note that in most object oriented languages, the method is a property of the type.
- Julia use **mutilpe dispatch** which occurs based on **every single argument**. For example

```
set_size(a::Array, args...) = ...
set_size(a::Axis, args...) = ...
set_size(s, a::Array, args...) = ...
set_size(a::Array, b::Vector) = ...
set_size(a::Array, x::Real, y::Real, z::Real) = ...
```

How it works

Keeping things simple, **multiple dispatch** follows one really basic rule: ***the most specific method that is applicable to the input arguments is the one chosen!***

- Upon calling a function, **Julia** will try to find the method that is most specific across all arguments.
- This means that if a method is defined for both the abstract type combination, as well as the concrete type combination, Julia will always call the more concrete one. - This rule also applies to e.g. parametric types, since Vector{Float64} is more specialized than Vector a method defined explicitly for Vector{Float64} is more specific.
- This rule also applies to unions, since Float64 is more specialized than Union{Float64, String}.

Two important points:

- **methods** do not belong to the Types!
- new methods can be defined after the Types have been defined!

Example

```
▸ Cat("Cat 2")
```

```
begin
    abstract type Animal end
    struct Dog <: Animal
        name::String
    end
    struct Cat <: Animal
        name::String
    end
    ## let's instaniate 4  animals
    d1 = Dog("Dog 1")
    d2 = Dog("Dog 2")
    c1 = Cat("Cat 1")
    c2 = Cat("Cat 2")
end
```

Let's now define functions that works for both cats and dogs by taking advantage of the `Animal` abstract type.

```
meets (generic function with 7 methods)
```

```
begin
    function encounter(a::Animal, b::Animal)
        verb = meets(a,b)
        println("$(a.name) meets $(b.name) and $verb")
    end
    meets(a::Animal, b::Animal) = "passes by"
end
```

```
encounter(c1,d2)
```

```
Cat 1 meets Dog 2 and hisses
```

Let's now be more specific.

```
meets (generic function with 4 methods)
```

```
begin
    meets(a::Dog, b::Dog) = "sniffs"
    meets(a::Dog, b::Cat) = "chases"
    meets(a::Cat, b::Dog) = "hisses"
    meets(a::Cat, b::Cat) = "slinks"
end
```

```
begin
    encounter(d1,d2)
    encounter(d1,c2)
    encounter(c1,c2)
    encounter(c1,d2)
end
```

```
Dog 1 meets Dog 2 and sniffs
Dog 1 meets Cat 2 and chases
Cat 1 meets Cat 2 and slinks
Cat 1 meets Dog 2 and hisses
```

- What is we need to define a new animal? say turtle

```
▸ Turtle("Turtle 1")
```

```
begin
    struct Turtle <: Animal
        name::String
    end
    meets(a::Dog, b::Turtle) = "barks"
    meets(a::Turtle, b::Dog) = "hides head"
    turtle1 = Turtle("Turtle 1")



end
```

```
begin
    encounter(turtle1, d1)
    encounter(d2,turtle1)
    encounter(c1,turtle1)
end
```

```
Turtle 1 meets Dog 1 and hides head
Dog 2 meets Turtle 1 and barks
Cat 1 meets Turtle 1 and passes by
```

- We can check how many methods a function has using `methods`
- We can check whcih method is used for a particalur call by `@which` macro

```
md"""
- We can check how many methods a function has using `methods`
-  We can check whcih method is used for a particalur call by `@which` macro
"""
```

meets(a::**Main.var"workspace#3".Cat**, b::**Main.var"workspace#3".Dog**) in Main.var"workspace#3" at C:\Users\mmogi\KFUPM\Julia Workshop 2024 - General\JuliaWorkShopJan2024\src\main_nb.jl#==#40df631e-7e04-475f-a66e-03457f370c5a:4

```
begin
    # methods(meets)
    @which meets(c1, d2)
end
```

Example: OneHot Vector

In various fields that use linear algebra, like machine learning, a special vector datastracture is used often, called a "OneHot" vector. This vector has a 1 in a single entry, and 0 in all other entries. Having a dedicated structure for this vector is super efficient, because you only need to store 2 numbers to store it in memory, irrespectively of how large the vector is!

Let's create this in Julia. To make our life easy, we will make this new Type a subtype of `AbstractVector`. This means that all functions that are defined on the abstract level for `AbstractVector` will work with our type as well (provided we implement all the necessary methods).

```
struct OneHotVector <: AbstractVector{Bool}
    len::Int
    ind::Int
end
```

- In order for our new vector to behave like a vector, we need to define some methods; mainly `size` and `getindex`

```
begin
    Base.size(v::OneHotVector) = (v.len,)
    Base.getindex(v::OneHotVector,i::Integer) = i == v.ind
end
```

```
1
```

```
begin
    oh1 = OneHotVector(10,3)
    oh2 = OneHotVector(10,3)
    size(oh1)
    sum(oh1)
    dot(oh1,oh2)
end
```

Excercises

1) Type Hierarchies

Create a function `person_info(p)` that takes in any type of a **"person"** and prints their name and potentially some extra information. For a normal person, only the name is printed. If the person is a student, it should print their name and grade. If it is a group leader, then print their name and their group name. If the input to person_info is something other than a "person", it should error.

Solve this exercise without using a single if statement; only multiple dispatch. These kind of problems seem to be "natural" to solve using if statements. However using multiple dispatch instead makes the code clearer, more easy to extend and (typically) more performant, because of how Julia compiles code.

Hint: given the rules of the exercise, you do not actually need abstract types to solve it. However, it is in general good practice to use them.

2) Indexing of a range type

Create your own `Range` object, which is an efficient iterable container defined by a (start, step, stop), which gives numeric values incremented by step starting from start, while ensuring that the result never exceeds stop. You only need to extend (add a new method) to the Base.getindex(r::Range, i::Int) function for this exercise.

Test your range by initializing a range like `r = Range(0, 0.5, 1)`, and checking if `r[1]`, `r[2]`, `r[3]` give you indeed `0.0, 0.5, 1.0` as they should.

This object can efficiently represent this range while storing only these 3 numbers, instead of all elements that theoretically belong to the range.

Hints:

- As a first step, define your `Range struct`, preferably by making it a parametric type of 1 type-parameter (think about how many fields it should have).
- The Julia syntax `A[i]` is translated to `getindex(A, i)`. That's why you need to extend the Base.getindex function. Use the help functionality on getindex to learn more.
- Implement indexing for your Range type via extending `getindex`. Index `1` giving `start`. Be sure it errors for incorrect indices (where the value given by the range would exceed `stop`).

3) Rational Numbers

Implement the type `RationalNumber <: Real`, similar to Julia's `Rational`, but without being a parametric type (i.e. both fields are just `Int`). Using the following function

```
function simplify(n::Integer, d::Integer)
    g = gcd(n, d)
    m = d < 0 ? -1 : 1
    (m * n ÷ g, m * d ÷ g)
end
```

define an **inner constructor**: that is a function `RationalNumber` inside the type-definition, which returns `new(simplify(n, d)...)`. `new` is a special keyword and stands for the type to be created. For example:

```
struct A
    a::Float64
    function A(a)
```

```
        x = cos(a)
        return new(x) # This creates an instance of A with the value x
    end
end
```

Inner constructors are used to perform specific tests or transformations necessary before instantiating a type. In this exercise the inner constructor simply ensures that if you try to define the rational number 5/10 you in fact create the equivalent (but simpler) 1/2.

Then, implement the operations `+, -, *, /` between `RationalNumber` s, by extending e.g. `Base.:+(r1::RationalNumber, r2::RationalNumber)`.

Test your code by ensuring that `RationalNumber(5, 10)` indeed has fields `(1, 2)`. Then test all operations with the rationals `1/2` and `1/3`.

extra point: Find out the minimum extension you need to make to your number type so that all trigonometric functions like cos, tanh, etc. work out of the box with it. (Hint: it is only one more method definition)

Linear Algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with `using LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported:

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
 1.0        0.0   0.0
 0.571429   1.0   0.0
 0.142857  -0.24  1.0
U factor:
3×3 Matrix{Float64}:
 7.0   8.0      1.0
 0.0  -3.57143  5.42857
 0.0   0.0      4.16
```

```
let
    # using LinearAlgebra
    A = [1 2 3;4 1 6;7 8 1]
    tr(A)
    inv(A)
    det(A)
    eigvals(A)
    eigvecs(A)
    factorize(A)
end
```

Differential Equations

Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl--a performant and feature-rich ecosystem for solving differential equations in Julia. Journal of Open Research Software, 5(1).

Source

`DifferentialEquations.jl` is by far the best free and open source differential equations solver (not just for Julia, for any language): it is orders of magnitude faster and has orders of mangitude more features than anything else out there. It can solve standard ODEs, Delay-DEs, stochastic DEs, PDEs, ADEs, event handling, 1000s of solvers for all tese DEs and many other features. It is the central part of a whole organization focused on scientific machine learning (SciML) and the basis for ModelingToolkit.jl, a library for building simulations from symbolic definitions.

Here we will focus on Ordinary Differential Equations (ODEs) solving and use the module OrdinaryDiffEq only.

The general workflow for using the package is as follows:

- Define a problem
- Solve the problem
- Analyze the output

ODEs

Defining and solving some ODEs

The way `DifferentialEquations.jl` works is quite straightforward:

- Make your set of ODEs a Julia function `f`
- Put `f`, an initial state and a parameter container into an `ODEProblem`.
- Choose the solvers and the arguments of the solvers you will use (e.g. tolerances, etc.)
- Give the ODEProblem as well as the auxiliary arguments to the function solve!

Example 1

Soving Scalar Equations

Solve

$$\frac{du}{dt} = 1.01u, \quad u(0) = \frac{1}{2}, \quad t \in [0,1],$$

Here,

- u is the current state variable,
- t is the current time.
- u_0 is the initial value of u.

```
md"""
### Example 1
Soving Scalar Equations

Solve
```math
\frac{du}{dt} = 1.01u, \; \; u(0) = \frac{1}{2}, \quad t\in [0,1],
```
Here,
- ``u`` is the current state variable,
- ``t`` is the current time.
- ``u_0`` is the initial value of ``u``.
"""
```

- After reading the **documentation**. An ordinary differential equation is defined by

$$\frac{du}{dt} = f(u,p,t)$$

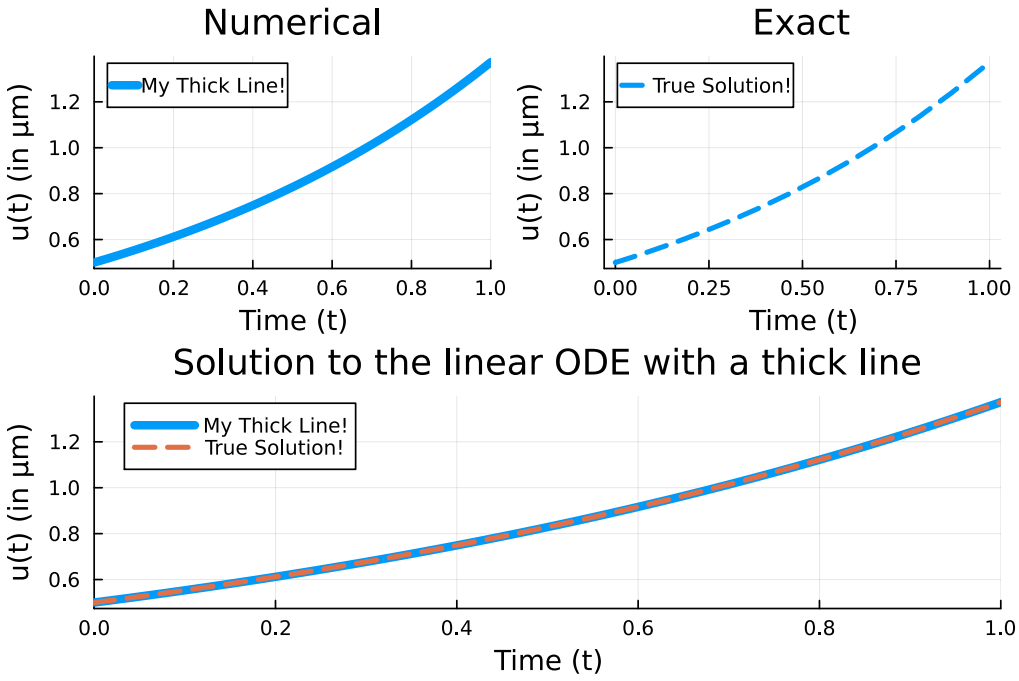
over some time interval `tspan` with some initial condition $u0$, and therefore the `ODEProblem` type is defined by those components:

```
oroblem = ODEProblem(f,u0,tspan)
```

Then we solve using a unified interface via `solve`

```
solution = solve(problem,alg;kwargs)
```

```
# @doc ODEProblem
```



Example 2: Simple Pendulum

We will start by solving the pendulum problem. In the physics class, we often solve this problem by small angle approximation, i.e. $\sin(\theta) \approx \theta$, because otherwise, we get an elliptic integral which doesn't have an analytic solution. The linearized form is

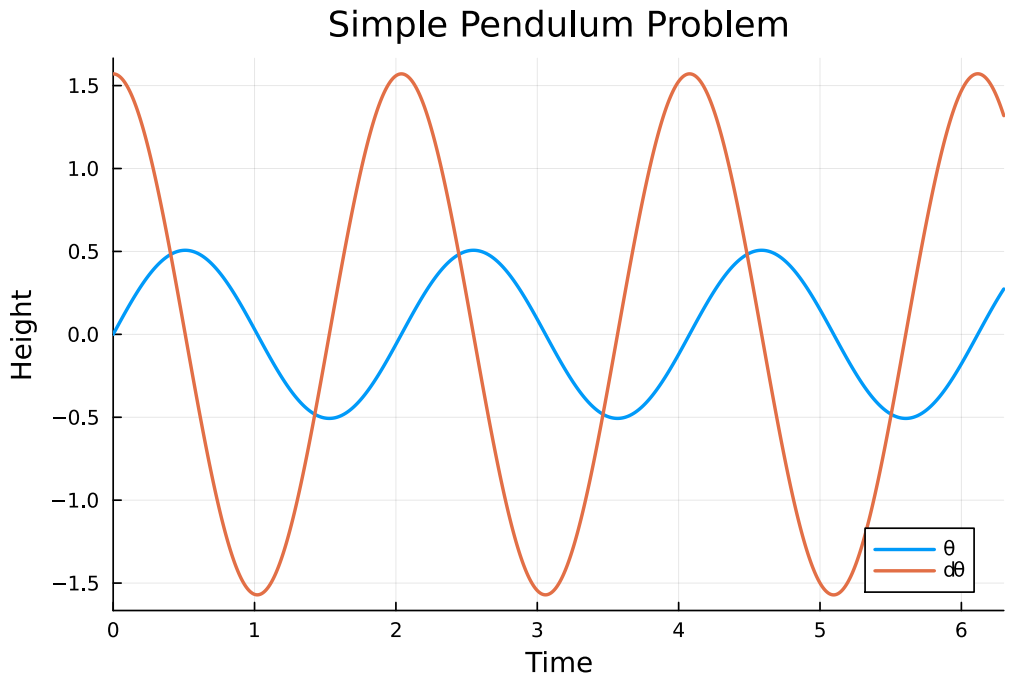
$$\ddot{\theta} + \frac{g}{L}\theta = 0$$

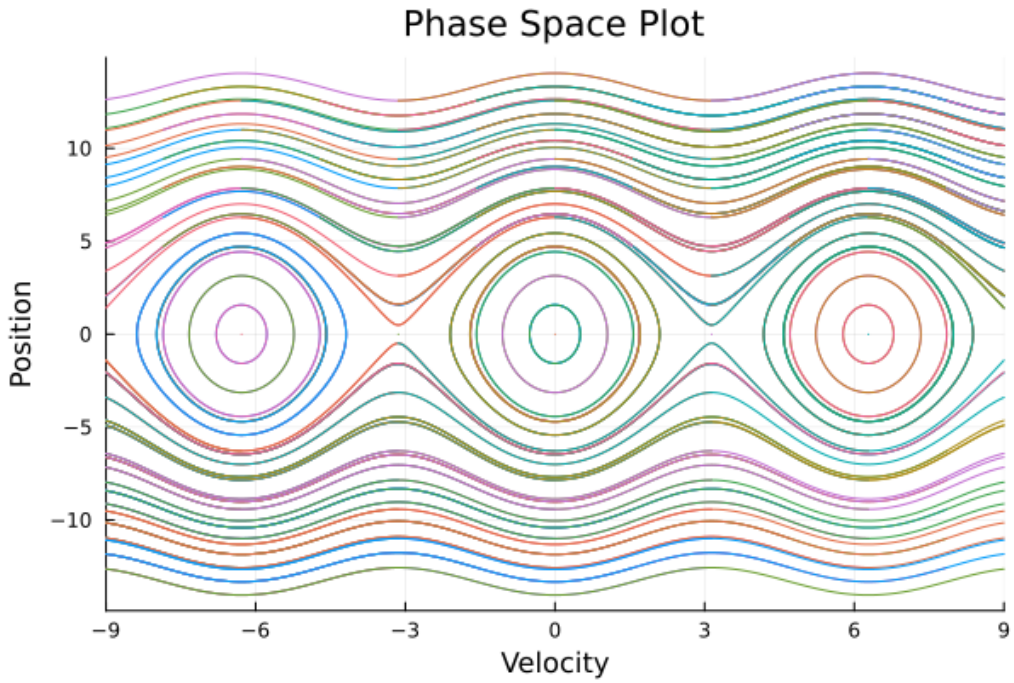
But we have numerical ODE solvers! Why not solve the real pendulum?

$$\ddot{\theta} + \frac{g}{L}\sin\theta = 0$$

Notice that now we have a second order ODE. In order to use the same method as above, we need to transform it into a system of first order ODEs by employing the notation

$$\begin{aligned} \dot{\theta} &= d\theta \\ \dot{d\theta} &= -\frac{g}{L}\sin\theta \end{aligned}$$






```julia
let
    p = plot(sol, idxs = (1, 2), xlims = (-9, 9), title = "Phase Space Plot",
    xaxis = "Velocity", yaxis = "Position", leg = false)
function phase_plot(prob, u0, p, tspan = 2pi)
    _prob = ODEProblem(prob.f, u0, (0.0, tspan))
    sol = solve(_prob, Vern9()) # Use Vern9 solver for higher accuracy
    plot!(p, sol, idxs = (1, 2), xlims = :none, ylims = :none)
end
for i in (-4pi):(pi / 2):(4π)
    for j in (-4pi):(pi / 2):(4π)
        phase_plot(prob, [j, i], p)
    end
end
plot(p, xlims = (-9, 9))
end
```

Optimization

Dixit, V. K., & Rackauckas, C. (2023). Optimization.jl: A Unified Optimization Package (Version v3.12.1). doi:10.5281/zenodo.7738525

Source

`Optimization.jl` provides

- the easiest way to create an optimization problem and solve it.
- a uniform interface to >25 optimization libraries, hence 100+ optimization solvers encompassing almost all classes of optimization algorithms such as global, mixed-integer, non-convex, second-order local, constrained, etc.
- the ability to choose an Automatic Differentiation (AD) backend by simply passing an argument to indicate the package to use and automatically generates the efficient derivatives of the objective and constraints while giving you the flexibility to switch between different AD engines as per your problem. Additionally,

`Optimization.jl` takes care of passing problem specific information to solvers that can leverage it such as the sparsity pattern of the hessian or constraint jacobian and the expression graph.

Install it by

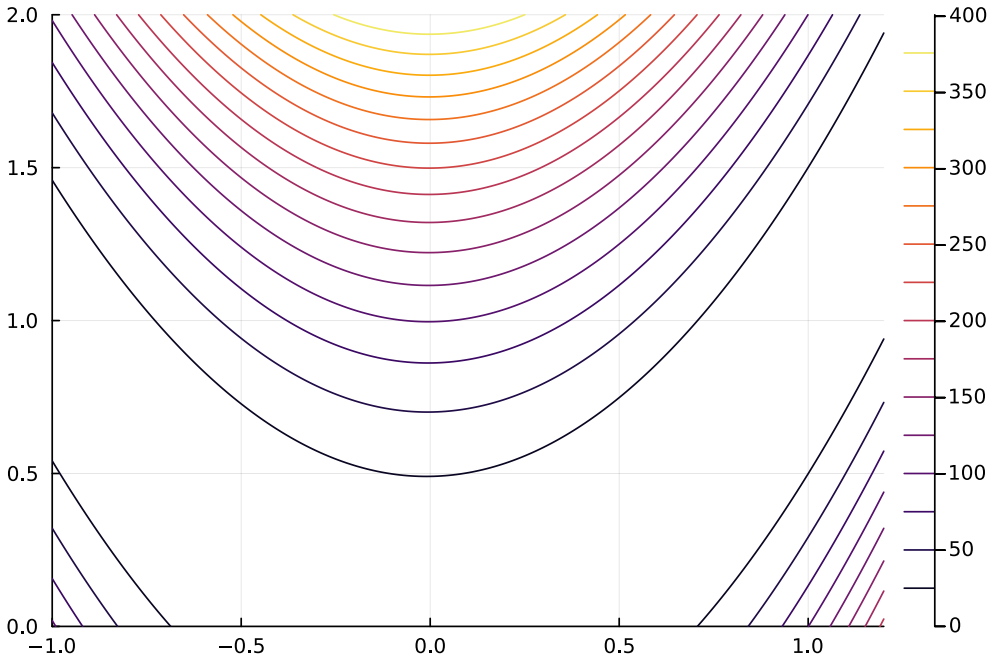
```
import Pkg
Pkg.add("Optimization")
```

Example 1

Find the minimum value of Rosenbrock function (2d) defined as

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

This function is non-convex and it has a global minimum at (a, a^2).



Defining the Objective Function

The objective function in **`Optimization.jl`** is assumed to have the signature `objective(x,p)`.

- `x` is optimization variables.
- `p` is a vector of any other parameters.

Let's now solve **Example 1**.

```
""
```

```
begin
    rosenbrock(x, p) = (p[1] - x[1])^2 + p[2] * (x[2] - x[1]^2)^2
    x0 = zeros(2)
    p = [1.0, 100.0]
    ""
end
```

Derivative-Free solvers

```
OptimizationProblem. In-place: true
u0: 2-element Vector{Float64}:
 0.0
 0.0
```

```
let
    # Import the package and define the problem to optimize
    # using Optimization


    prob = OptimizationProblem(rosenbrock, x0, p)

    # 1
    # Import a solver package and solve the optimization problem
    # using OptimizationOptimJL # solver package for Optim.jl
    # sol = solve(prob, NelderMead())
    # summary
    # sol.original

    # 2
    # Import a different solver package and solve the optimization problem a different way
    # using OptimizationBBO # solve package for BlackBoxOptim.jl.
    # prob = OptimizationProblem(rosenbrock, x0, p, lb = [-1.0, -1.0], ub = [1.0, 1.0])
    # sol = solve(prob, BBO_adaptive_de_rand_1_bin_radiuslimited())
    # sol.original


end
```

Gradient-Based Solvers

In the above example, niether of the used methods require the gradient. However, often first order optimization (i.e., using gradients) is much more efficient. Defining gradients can be done in two ways. One way is to manually provide a gradient definition in the `OptimizationFunction` constructor. However, the more convenient way to obtain gradients is to provide an AD backend type.

For example let's us Broyden–Fletcher–Goldfarb–Shanno (BFGS) method in `OptimizationOptimJL`. This method is first-order method. So to solve **Example 1**, we need to provide the gradient either directely or using automatic differentiation, forward-mode, (`using ForwardDiff`)

```
* Status: success

* Candidate solution
   Final objective value:     7.645684e-21

* Found with
   Algorithm:     BFGS

* Convergence measures
   |x - x'|               = 3.48e-07 ≰ 0.0e+00
   |x - x'|/|x'|          = 3.48e-07 ≰ 0.0e+00
   |f(x) - f(x')|         = 6.91e-14 ≰ 0.0e+00
   |f(x) - f(x')|/|f(x')| = 9.03e+06 ≰ 0.0e+00
   |g(x)|                 = 2.32e-09 ≤ 1.0e-08

* Work counters
   Seconds run:   0  (vs limit Inf)
   Iterations:    16
   f(x) calls:    53
   ∇f(x) calls:   53
```

```julia
let
    # using ForwardDiff
    optf = OptimizationFunction(rosenbrock, Optimization.AutoForwardDiff())
    prob = OptimizationProblem(optf, x0, p)
    sol = solve(prob, BFGS())
    sol.original
end
```

Hessian-Based Solvers

```
( * Status: success                              , [1.0, 1.0])
```

```julia
let
    # using ForwardDiff
    optf = OptimizationFunction(rosenbrock, Optimization.AutoForwardDiff())
    prob = OptimizationProblem(optf, x0, p)
    sol = solve(prob, Newton())
    sol.original, sol.u
end
```

Constraints

Example 2:

Solve

$$\begin{aligned} \min \quad & (a-x)^2 + b(y-x^2)^2 \\ \text{subject to} \quad & \\ & x^2 + y^2 \leq 0.8 \\ & -1.0 \leq xy \leq 2.0 \end{aligned}$$

We need a solver that handles nonlinear constraints. `MathOptInterface` and `Optim`'s `IPNewton` is such solver. **`Optimization.jl`** provides a simple interface to define the constraint as a Julia function and then specify the bounds for the output in `OptimizationFunction` to indicate if it's an equality or inequality constraint.

0.49999999985827304

```
let
    # using Optimization,  OptimizationOptimJL
    rosenbrock(x, p) = (p[1] - x[1])^2 + p[2] * (x[2] - x[1]^2)^2
    x0 = zeros(2)
    p = [1.0, 1.0]
    # constraint functions.
    cons(res, x, p) = (res .= [x[1]^2 + x[2]^2, x[1] * x[2]])
    optprob = OptimizationFunction(rosenbrock, Optimization.AutoForwardDiff(), cons = cons)
    prob = OptimizationProblem(optprob, x0, p, lcons = [-Inf, -1.0], ucons = [0.8, 2.0])
    sol = solve(prob, IPNewton())
    sol.u
    ## check the constraints
    cons(x0,sol.u,p)
    sol.objective, prob.f(sol.u,p)
    # if we change the constrains to
    # x^2 + y^2 =1
    # xy = 0.5
    prob = OptimizationProblem(optprob, x0, p, lcons = [1, 0.50], ucons = [1, 0.5])
    sol = solve(prob, IPNewton())
    sol.u[1]*sol.u[2]
end
```

Linear Solvers

To solve a linear system $Ax = b$, one might use the slash operator \. This is the same as in MATLAB. For example, to solve

$$\begin{aligned} 2x + y - 2z &= -1 \\ 3x - 3y - z &= 5 \\ x - 2y + 3z &= 6 \end{aligned}$$

▸ [1.0, -1.0, 1.0]

```
let
    A = [2. 1. -2
         3 -3 -1
        1 -2 3]
    b= [-1;5;6]
    x = A\b

end
```

LinearSolve.jl

LinearSolve.jl: High-Performance Unified Linear Solvers

Source

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

However we can also use the package **`LinearSolve.jl`** which is a unified interface for the linear solving packages of Julia. It interfaces with other packages of the Julia ecosystem to make it easy to test alternative solver packages and pass small types to control algorithm swapping. It also interfaces with the ModelingToolkit.jl world of symbolic modeling to allow for automatically generating high-performance code.

▸ LinearCache(3×3 Matrix{Float64}:, [-1.0, 5.0, 6.0], [1.0, -1.0, 1.0], NullParameters(), Kr
2.0 1.0 -2.0

```
let
    # using LinearSolve
    A = Float64.([2 1 -2
         3 -3 -1
        1 -2 3])
    b= Float64.([-1;5;6])
    prob = LinearProblem(A,b)
    # sol = solve(prob)
    sol = solve(prob,KrylovJL_GMRES())
    sol.u
    sol.cache
end
```

Nonlinear Solvers

NonlinearSolve.jl: High-Performance Unified Nonlinear Solvers

Source

Installation

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

A nonlinear system $f(u) = 0$ is specified by defining a function $f(u, p)$, where p are the parameters of the system.

Example 1

Solve the vector equation $f(u) = u^2 - p$ for a vector of equations. Here $u \in \mathbb{R}^2$

```
▸ TrustRegion(AutoForwardDiff(Tag()), nothing, DEFAULT_PRECS (generic function with 1 metho
```

```
let
    # using NonlinearSolve
    f(u,p) = u.^2 .- p
    u0 = [1.0, 1.0]
    p = 2.0
    prob = NonlinearProblem(f, u0, p)
    sol = solve(prob, TrustRegion())
    u = sol.u
    f(u, p)
    sol.stats
    sol.alg
end
```

Symbolic Computation

Gowda, S., Ma, Y., Cheli, A., Gwozdz, M., Shah, V. B., Edelman, A., & Rackauckas, C. (2021). High-performance symbolic-numerics via multiple dispatch. arXiv Preprint arXiv:2105. 03949.

Source

`Symbolics.jl` is a symbolic modeling language.

Installation

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

```
2×2 Matrix{Int64}:
 4  6
 4  0
```

```
let
    # using Symbolics
    # Building Symbolic Expressions
    @variables x y
    # Define an expression
    z = x^2 + y
    # Define an array of expressions
    A = [x^2+y 0 2x
        0     0 2y
        y^2+x 0 0
    ]
    # We can use normal Julia functions
    function f(u)
        [u[1]-u[3],u[1]^2-u[2],u[3]+u[2]]
    end
    f([x,y,z])

    # We can define an array variable
    @variables u[1:4]
    f(u)

    # Derivatives
    @variables t
    # Define the differential operator d/dt
    D = Differential(t)
    # we can now differentiate an expression in t
    d = D(t^2+t)
    # The result is lazily computed to expand we use
    expand_derivatives(d)
    # We can also compute the Jacobian of an array
    Symbolics.jacobian(A[1,:],[x,y])

    # We can simplify expression
    # using SymbolicUtils
    simplify(2x + 2y)
    # or
    B = simplify.([t + t^2 + t + t^2  2t + 4t
                x + y + y + 2t     x^2 - x^2 + y^2])
    # We can then use substitute to change values of an expression around:
    simplify.(substitute.(B, (Dict(x => 2, t=>1, y=>0),)))

    # We substitute
    V = substitute.(B, (Dict(x => 2, t=>1, y=>0),))
    Symbolics.value.(V)
end
```

Graph Theory

JuliaGraphs

Source

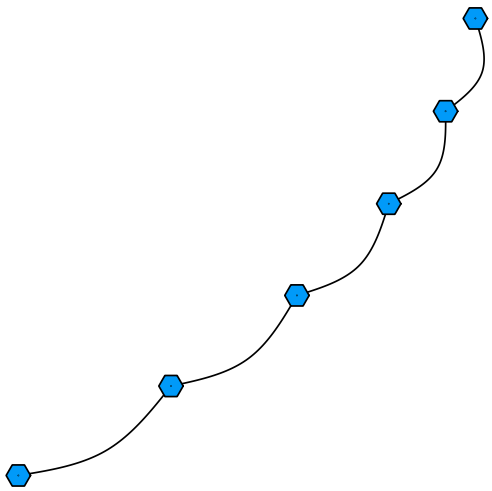
The central package of the ecosystem is `Graphs.jl`. It contains a standard graph interface and some basic types for unweighted graphs, as well as a set of combinatorial algorithms like shortest paths.

Installation

```
import Pkg
Pkg.add("Graphs.jl)
```

and for Visualization we install `GraphRecipes.jl` as well.

Let's start by creating a path graph and answer some questions about it.



```
begin
    pg = path_graph(6)
    graphplot(pg)
end
```

How many vertices?

6

```
nv(pg)
```

How many edges?

5

```
ne(pg)
```

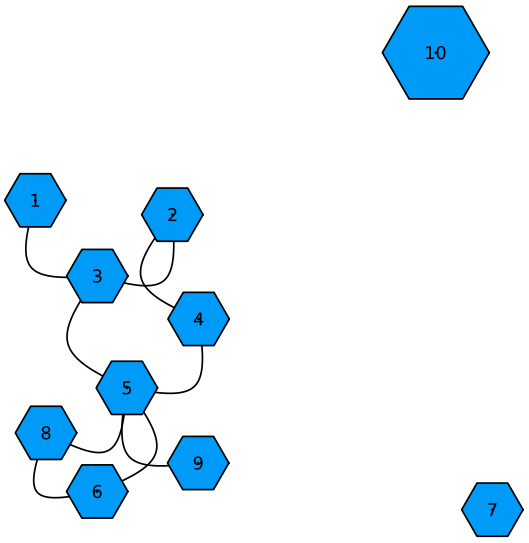
Let's add an edge

▸ [1, 3]

Let's try a simple graph

```
generateRandomGraph (generic function with 2 methods)
```

```
function generateRandomGraph(n,c=5)
    g = SimpleGraph(n)
    m = Int(n//2)+1
    v1 = rand(1:m,c)
    v2 = rand(1:n,c)
    # length.([v1,v2])
    for i ∈ 1:c
        add_edge!(g,v1[i],v2[i])
    end
    g
end
```




```
begin
    smplGraph = generateRandomGraph(10,10)
    # =SimpleGraph(10) # vertices with no edges
    # add_vertex!(smplGraph)
    # add_edge!(smplGraph, 3, 11)
    # # g2=SimpleDiGraph(10)
    graphplot(smplGraph, names=1:10,
        markersize = 0.4,
        curvature_scalar=0.1)

end
```

▸ DijkstraState([3, 0, 2, 2, 3, 5, 0, 5, 5, 0], [2, 0, 1, 1, 2, 3, 9223372036854775807, 3, 3,

```
dijkstra_shortest_paths(smplGraph,2)
```

```
10×10 Matrix{Int64}:
 0  0  1  0  0  0  0  0  0  0
 0  0  1  1  0  0  0  0  0  0
 1  1  0  0  1  0  0  0  0  0
 0  1  0  0  1  0  0  0  0  0
 0  0  1  1  0  1  0  1  1  0
 0  0  0  0  1  0  0  1  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  0  0  0  0
 0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
```

```
Matrix(adjacency_matrix(smplGraph))
```

```
▸ [(2, 1), (3, 2), (4, 2), (4, 3), (1, 4), (5, 1)]
```

```
begin
    elist = [(1,2),(2,3),(2,4),(3,4),(4,1),(1,5)]

    g2 = SimpleGraph(Graphs.SimpleEdge.(reverse.(elist)))
    graphplot(g2,names=1:vertices(g2)[end])
    reverse.(elist)
end
```

```
▸ [[2, 4, 1], [2, 3, 4]]
```

```
cycle_basis(g2)
```

Data Science

We start with loading the required packages and reading in the `CSV` file to a `DataFrame`:

```
using CSV
using DataFrames
using Dates
using Plots
```

We will also use `Downloads` package to download the data. The data is for USA Airport Dataset available from `kaggle` here.

```
"C:\\Users\\mmogi\\AppData\\Local\\Temp\\jl_OH25AV8fat"
```

```
begin
    raw_data =
    Downloads.download("https://www.dropbox.com/scl/fi/hb0sq53m2isd5cd5suqyp/Airports2
    .csv?rlkey=l1ggqiy8078bpya4casabokeg&raw=1")

end
```

```
df =
```

	Origin_airport	Destination_airport	Origin_city	Destination_city
1	"MHK"	"AMW"	"Manhattan, KS"	"Ames, IA"
2	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
3	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
4	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
5	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
6	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
7	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
8	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
9	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
10	"SEA"	"RDM"	"Seattle, WA"	"Bend, OR"
⋮	more			
3606803	"FWA"	"OH1"	"Fort Wayne, IN"	"Washington Court House, OI

```
df = CSV.read(raw_data,DataFrame,missingstring="NA")
```

Let us now get some summary statistics of our data set using the `describe` function

```
md"Let us now get some summary statistics of our data set using the `describe`
function"
```

	variable	mean	min	median	max
1	:Origin_airport	nothing	"1B1"	nothing	"ZZV"
2	:Destination_airport	nothing	"1B1"	nothing	"ZZV"
3	:Origin_city	nothing	"Aberdeen, SD"	nothing	"Zanesville, OH"
4	:Destination_city	nothing	"Aberdeen, SD"	nothing	"Zanesville, OH"
5	:Passengers	2688.91	0	1118.0	89597
6	:Seats	4048.3	0	1998.0	147062
7	:Flights	37.2289	0	25.0	1128
8	:Distance	697.319	0	519.0	5095
9	:Fly_date	nothing	1990-01-01	2001-11-01	2009-12-01
10	:Origin_population	5.8715e6	13005	2.40019e6	38139592
11	:Destination_population	5.89798e6	12887	2.40019e6	38139592
12	:Org_airport_lat	37.7503	19.7214	38.8521	64.8375
13	:Org_airport_long	-91.8618	-157.922	-87.7524	-68.8281
14	:Dest_airport_lat	37.7409	19.7214	38.8521	64.8375
15	:Dest_airport_long	-91.8343	-157.922	-87.7524	-68.8281

```
describe(df)
```

From the source `kaggle`, the data consists of the following columns:

1. **:Origin_airport**: Three letter airport code of the origin airport
2. **:Destination_airport**: Three letter airport code of the destination airport
3. **:Origin_city**: Origin city name
4. **:Destination_city**: Destination city name
5. **:Passengers**: Number of passengers transported from origin to destination
6. **:Seats**: Number of seats available on flights from origin to destination
7. **:Flights**: Number of flights between origin and destination (multiple records for one month, many with flights > 1)
8. **:Distance**: Distance (to nearest mile) flown between origin and destination
9. **:Fly_date**: The date (yyyymm) of flight
10. **:Origin_population**: Origin city's population as reported by US Census
11. **:Destination_population**: Destination city's population as reported by US Census

Let's find out more about our data

```
(3606803, 15)
```

```
size(df)
```

The data consists of

- 3606803 rows
- 15 cols
- The names of the columns as string: Origin_airport, Destination_airport, Origin_city, Destination_city, Passengers, Seats, Flights, Distance, Fly_date, Origin_population, Destination_population, Org_airport_lat, Org_airport_long, Dest_airport_lat, Dest_airport_long

```
cm"""
The data consists of

- $(nrow(df)) rows

- $(ncol(df))  cols

- The names of the columns as string: $(join(names(df),",\n"))
"""
```

3606803

```
# number of rows
nrow(df)
```

15

```
# number of cols
ncol(df)
```

▸ ["Origin_airport", "Destination_airport", "Origin_city", "Destination_city", "Passengers"

```
# names of the columns are
names(df)
```

▸ [:Origin_airport, :Destination_airport, :Origin_city, :Destination_city, :Passengers, :Sea

```
# names of the columns (as Symbols) are
propertynames(df)
```

In particular note that `CSV.jl` correctly idenfitied `:Fly_date` column as being a `Date` type.

When we investigate the summary statistics we note that there are flights that potentially have 0 passengers, 0 seats, or 0 flights.

In practice if you get such data it is good to investigate it, as it shows some potential data quality issues.

So, let us investigate it:

	Passengers	Seats	Flights	nrow	proprow
1	false	false	false	11239	0.00311606
2	false	false	true	322787	0.0894939
3	false	true	false	94	2.60619e-5
4	false	true	true	51772	0.014354
5	true	false	false	7	1.94078e-6
6	true	false	true	3	8.31762e-7
7	true	true	false	56	1.55262e-5
8	true	true	true	3220845	0.892992

```
df |>
    x -> select(x, :Passengers, :Seats, :Flights) |>
    x -> mapcols(ByRow(>(0)), x) |> # or: mapcols(ByRow( x -> x > 0 ), _ )
    x-> groupby(x,:, sort=true) |>
    x-> combine(x, nrow, proprow)
# we are using the pipe operator '|>'
# let
#   df1 = select(df, :Passengers, :Seats, :Flights)
#   df2 = mapcols(ByRow(>(0)), df1)
#   df3 = groupby(df2,:, sort=true)
#   df4 = combine(df3, nrow, proprow)
# end
```

Now going to our results - some of entries can be explained, like 0 passenges, but some seats and some flights (I guess it means that just no one took some flight).

However cases like some passengers, but 0 seats and 0 flights are probably a mistake in data (we have 7 rows that have this combination of values).

Let us try to find these 7 rows in two ways:

```
md"""
Now going to our results - some of entries can be explained, like 0 passenges, but some seats and some flights (I guess it means that just no one took some flight).

However cases like some passengers, but 0 seats and 0 flights are probably a mistake in data (we have 7 rows that have this combination of values).

Let us try to find these 7 rows in two ways:
"""
```

	Origin_airport	Destination_airport	Origin_city	Destination_city	Passengers	S
1	"ABR"	"PIR"	"Aberdeen, SD"	"Pierre, SD"	11	0
2	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	2	0
3	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	1	0
4	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	2	0
5	"BOS"	"LAS"	"Boston, MA"	"Las Vegas, NV"	9	0
6	"BKW"	"IAD"	"Beckley, WV"	"Washington, DC"	7	0
7	"CMI"	"BMI"	"Champaign, IL"	"Bloomington, IL"	6	0

```
# 1 using `filter`
filter(row->row.Passengers>0 && row.Seats==0 && row.Flights==0, df)
```

	Origin_airport	Destination_airport	Origin_city	Destination_city	Passengers	S
1	"ABR"	"PIR"	"Aberdeen, SD"	"Pierre, SD"	11	0
2	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	2	0
3	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	1	0
4	"IAD"	"BKW"	"Washington, DC"	"Beckley, WV"	2	0
5	"BOS"	"LAS"	"Boston, MA"	"Las Vegas, NV"	9	0
6	"BKW"	"IAD"	"Beckley, WV"	"Washington, DC"	7	0
7	"CMI"	"BMI"	"Champaign, IL"	"Bloomington, IL"	6	0

```
# 2 using subset
subset(df, :Passengers=>ByRow(>(0)), :Seats=>ByRow(==(0)), :Flights=> ByRow(==(0)))
```

df2 =

	Origin_airport	Destination_airport	Origin_city	Destination_city
1	"MHK"	"AMW"	"Manhattan, KS"	"Ames, IA"
2	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
3	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
4	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
5	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
6	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
7	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
8	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
9	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
10	"SEA"	"RDM"	"Seattle, WA"	"Bend, OR"
⋮ more				
3606796	"FWA"	"OH1"	"Fort Wayne, IN"	"Washington Court House, OH

```
df2 = filter(row->row.Passengers==0 || row.Seats>0 || row.Flights>0, df)

```

Now let us move to another analysis.

We want to get an information about the occupancy of each flight. It can be calculated as ratio of the number of passengers and number of seats time number of flights.

$$\text{occupancy} = \frac{\text{passengers}}{\text{seats} + \text{flights}}$$

The problem is that if there are 0 seats or flights we would be dividing by 0 and get a NaN result. We prefer to get a missing value in this case. Therefore first define a helper function:

```
get_occupied (generic function with 1 method)
```

```
function get_occupied(passengers, seats, flights)
    if seats == 0 || flights == 0
        return missing
    else
        return passengers / (seats * flights)
    end
end
```

and now we use it to add a new column to our data frame. Additionally we create three new columns:

1. **:ost** : state of the origin city
2. **:dst** : state of the destination city
3. **:year** : year of flight

df3 =

	Origin_airport	Destination_airport	Origin_city	Destination_city
1	"MHK"	"AMW"	"Manhattan, KS"	"Ames, IA"
2	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
3	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
4	"EUG"	"RDM"	"Eugene, OR"	"Bend, OR"
5	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
6	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
7	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
8	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
9	"MFR"	"RDM"	"Medford, OR"	"Bend, OR"
10	"SEA"	"RDM"	"Seattle, WA"	"Bend, OR"
⋮	more			
3606803	"FWA"	"OH1"	"Fort Wayne, IN"	"Washington Court House, OI

```
# year is available through Dates package
df3 = transform(df,
    [:Passengers, :Seats, :Flights] => ByRow(get_occupied) => :occupied,
    [:Origin_city, :Destination_city] .=> ByRow(x -> last(x, 2)) .=> [:ost, :dst],
                :Fly_date => ByRow(year) => :year)
```

Notice again that in the computation we used ByRow wrapper that instructs **DataFrames.jl** to apply the function for each row of the passed data.

It is instructive to disect this pipeline step by step. When we said

```
[:Passengers, :Seats, :Flights] => ByRow(get_occupied) => :occupied
```

remember that the **get_occupied** function required **3** input arguments. The initial `[:Passengers, :Seats, :Flights]` selector makes sure we pass the required **3** columns, and in the correct order. Finally, having `ByRow(get_occupied) => :occupied` means that our function returns a single output vector, and we supply a single name, `:occupied` for it.

In the next statement, we wrote

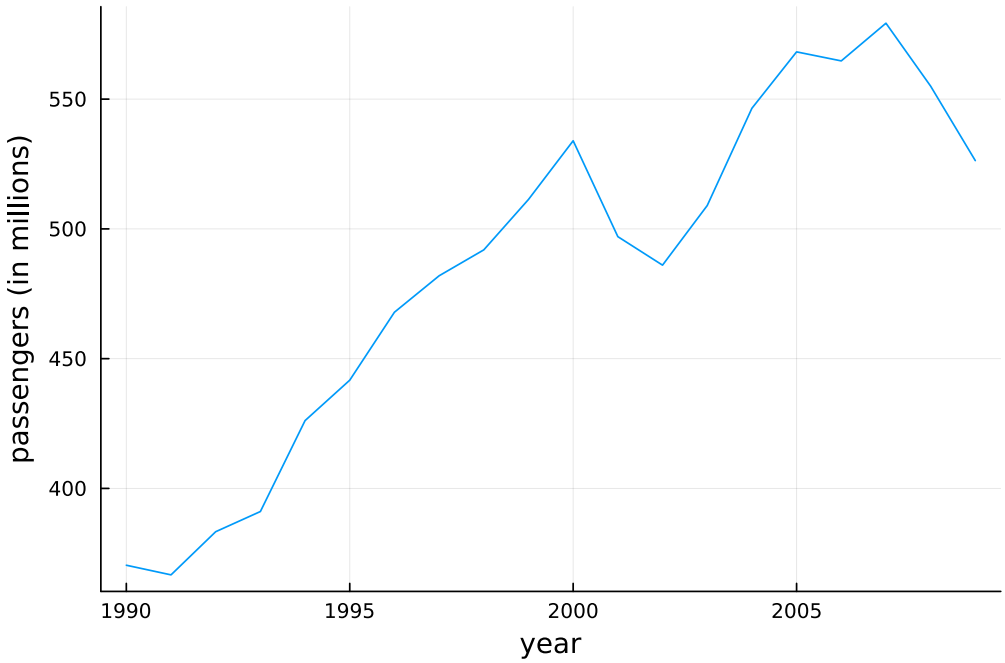
```
[:Origin_city, :Destination_city] .=> ByRow(x -> last(x, 2)) .=> [:ost, :dst]
```

which looks similar, but has a crucial difference: the Pair operator `=>` is being broadcasted this time (note the `.` in front of the `=>`)! In simple words, here we will apply our transformation to each input column, outputting as well **2** columns (ensured by the second `.=>`). One noteworthy feature of `DataFrames.jl` is that the so-written components of a data pipeline are valid julia code (and not part of a package-specific DSL). For instance, you can evaluate this line in the REPL to investigate the transformation we are building in this step:

```
▸ [:Origin_city ⇒ ByRow{var"#25#26"{typeof(last)}}(#25) ⇒ :ost, :Destination_city ⇒ ByRo
```

```
[:Origin_city, :Destination_city] .=> ByRow(x -> last(x, 2)) .=> [:ost, :dst]
```

Finally, let's do some visualiziation by plotting the total number of passengers that flew by year




```
df3 |>
    x-> groupby(x,:year, sort=true) |>
    x -> combine(x,:Passengers => sum=> :Total_Passengers) |>
    x -> plot(x.year,x.Total_Passengers./10^6, legend=nothing,
            xlab="year", ylab="passengers (in millions)"
            )

```

In the following example we will want to find pairs of origin-destination states that have the highest occupancy:

df4 =

	ost	dst	occupied	Passengers_sum
1	"NE"	"AK"	1.0	180
2	"MD"	"ID"	1.0	150
3	"ID"	"AR"	1.0	124
4	"ID"	"MS"	1.0	181
5	"ND"	"RI"	1.0	50
6	"WY"	"NY"	0.991935	123
7	"OR"	"RI"	0.991935	123
8	"WV"	"NE"	0.984615	176
9	"KS"	"ID"	0.983871	122
10	"AK"	"TN"	0.981723	376
⋮ more				
2369	"AK"	"VA"	0.0	0

```
# mean from Statistics pacakge
df4 = df3 |>
x -> groupby(x,[:ost,:dst]) |>
# x -> combine(x,:occupied => (y->mean(skipmissing(y))) => :occupied, :Passengers => sum) |>
x -> combine(x,:occupied => mean∘skipmissing => :occupied, :Passengers => sum) |>
x -> filter(y->isfinite(y.occupied),x)  |>
x -> sort(x,:occupied, rev=true)

```

```
begin
    using PlutoUI, CommonMark
    # using FileIO, ImageIO, ImageShow, ImageTransformations
    # using SymPy
    using Plots, PlotThemes, LaTeXStrings
    using Latexify
    using GraphRecipes
    using Graphs
    using HypertextLiteral: @htl, @htl_str
    # using Colors
    using Random
    using LinearAlgebra
    using DifferentialEquations
    using Optimization, OptimizationOptimJL, OptimizationBBO
    # using OptimizationMOI, Ipopt
    using ForwardDiff
    using LinearSolve
    using NonlinearSolve
    using Symbolics
    using SymbolicUtils
    using DataFrames
    using CSV
    using Downloads
    using Dates
    using Statistics

#   import Pkg
# Pkg.activate(@__DIR__)
# Pkg.status()
end
```

```
cite (generic function with 1 method)
```

```
begin

    function cite(paper, link)
        @htl("""
        <div style="display: flex; justify-content:space-between; flex-direction:
        row; gap: 10px;">

        <p style="margin-right: 10px;padding-right: 10px;">

        $paper

        </p>

        <p style="border-left: 2px solid red;">

        <a style="margin-left: 10px;padding-right: 10px;" href="$(link)"
        target="_blank" rel="noopener noreferrer">Source</a>

        </p>
        </div>
        """)
    end
end
```