Scientific Programming - Introduction to Julia Programming

Institute for Biomedical Imaging, Hamburg University of Technology

- 🧑‍🏫 Prof. Dr.-Ing. Tobias Knopp
- 🧑‍🏫 Dr. rer. nat. Martin Möddel

Table of Contents

Getting Started with the REPL

Julia code can be run in various ways:

- The easiest way is to start the Julia REPL. REPL stands for Read Eval Print Loop and is what is often known as a terminal or console. The REPL is characterized by being an interactive environoment. One can basically inspect the code while running it. When you start Julia from the command line you will get a promt like this

```
julia> 1 + 1
2
```

What happens is that the *expression* `1+1` is *read*. Then it is *evaluated*. And finally the result is *printed*. All this is repeated in an infinite *loop*. So its a Read Eval Print Loop.

- It is also possible to execute a Julia script that has the extension .jl by running

```
 > julia myscript.jl
```

Note

One common question being asked when learning a new programming language is: What is a good IDE (Integrated Development Environment) for the language?

We actually encourage that you start with a simple text editor and the REPL and learn how to master programming in this way. As editors we suggest

```
* vim
* vscode
* gedit
```

just to name a few. `vim` is a tool, which you should get used to anyway since you will have situations where you are on a server and want to change some code without a graphical frontend.

Note that `vscode` is actually a full featured Julia IDE with the appropriate plugin being installed. So if you want to give that a go feel free to play around with its features.

Notebooks

In this course we use *Pluto.jl* for both the lecture and the exercise. *Pluto* is a so-called *notebook* system which is helpful in an interactive setting. Pluto is similar to Jupyter or Mathematica but has some advantages compared to Jupyter. In particular the current document is a plain Julia file that you can also execute in the Julia REPL or open in a plain text editor like vim or vscode. Here are some Pluto features:

- One can mix code, text, and images freely.
- Notebooks are an implementation of literate programming proposed by Donald Knuth. Notebooks are actually a little bit more, the term *literate computing* is often used for notebook systems.
- In notebook systems one has *cells* that are executed in order.
- Notebooks are also handy to capture the *plotting results* being generated during a calculation.
- Pluto also has a nice help function (see live docs in the lower right)

But we should note that despite these many advantages, notebook systems also have disadvantages:

- They do not scale well when modularizing code.
- Video explaining several disadvantage of notebook systems on YouTube.

Note

We encourage you to use the REPL early although the first exercises are all done using Pluto notebooks. The reason is that this help us putting the tasks into small chunks that you can solve step by step. In a real project you will develop the code in a package and only use notebooks as a frontend.

Note

Within this notebook you will sometime see expressions like

```
with_terminal() do
    ...
end
```

or

```
begin
    ...
end
```

or

```
let ...
    ...
end
```

You can simply ignore these enclosing code blocks. Both are necessary in *Pluto* to display multiple expressions within a single cell. From time to time Julia code is not entered as real code but just as text as is done within this note. We do this when we don't want the code to run, for example because the above code is not complete.

Assignments

Values can be assigned to variables using the assignment operator

```
a = 30
a = 30
```

Coming from a language with static typing one immediately observes that the type of the variable does not need to be declared. This is usual for a language with a *dynamic type system*. But of course `a` still has a concrete type. Lets inspect that:

```
Int64
typeof(a)
```

This makes sense since the literal `30` apparently is an integer. Julia's default integer type has 64 bit. Here, are some elementary types:

```
Float64
typeof(0.4)
```

```
String
typeof("Test")
```

```
Bool
```

```julia
typeof(true)
```

```
Irrational{:π}
```

```julia
typeof(pi)
```

That is interesting, why isn't the type of the constant `pi` a `Float64`? Maybe because pi does not fit into a 64-bit floating point number. Lets convert it

```
3.141592653589793
```

```julia
convert(Float64, pi)
```

This brings us to an interesting aspect. What if we want a variable `b` to be of a specific type. Can this somehow be enforced? The answer is yes and the syntax is as follows

```
π = 3.1415926535897...
```

```julia
b::Float64 = pi
```

```
Float64
```

```julia
typeof(b)
```

We call this explicit typing. You can also see in the example that the conversion from the type `Irrational{:π}` to the type `Float64` is done implicitly. This is because the function `convert(::Type{Float64}, ::Irrational{:π})` exists. Implicit conversion is very handy. You are certainly used to it from other programming languages

Printing

The Julia REPL prints the value of an entered expression by default. One can omit this by putting a `;` at the end of an expression.

Note

Unlike Matlab Julia does not require semicolons in scripts that are included to suppress printing a value. When you include a script what is printed is the last expression being evaluated.

In Julia code in files one thus needs functions to print variables and expressions. There are various possibilities for that:

```
3
y = 3
[ Info: 3
┌ Warning: What is going on?
└ @ Main.var"workspace#3" ~/tuhhCloud/Lectures/WS2223/ImageProcessing/tutorials/1. In
```

```julia
with_terminal() do
    y = 3

    println(y) # if you omit the ln, there will be no line break
    @show y # this macro is handy since it can operate on expressions
    @info y # this goes in direction of the logging infrastructure
    @warn "What is going on?"

end
```

Of course we can also read user input:

```
julia> a = readline();
Hallo

julia> @show a
a = "Hallo"
"Hallo"
```

Functions

One of the core concepts in all successful programming languages is the possibility of grouping functionality in smaller, reusable code units called functions. Functions typically have arguments, a function body, and return values.

Lets define a function that calculates square numbers

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

```julia
function square(a)
  b = a*a
  return b
end
```

```
16
```

```julia
square(4)
```

Seems to work. Here, `a` is the function argument, `b` is the return value and everything between the first and the last line is the function body. Julia also has a short notation:

```julia
square(a) = a*a
```

which is often handy when defining small functions.

Note

Functions in Julia are as for most procedural languages not the same as mathematical functions since they can have side effects. Julia has the possibility of marking functions as pure, which allows the compiler to apply additional optimization that are not allowed for unpure functions

Higher Order Functions

In Julia functions are regular objects of type `Function`. Thus they can be stored or passed around freely, i.e. functions in Julia are first-class objects. This allows to implement higher order functions in Julia:

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

```julia
function foo(f::Function, arg)
    return f(arg)
end
```

```
16
```

```julia
foo(square, 4)
```

Such higher order functions are for instance relevant when wanting to apply scalar function to a collection of values:

```
[1, 4, 9]
```

```julia
map(square, [1, 2, 3])
```

Anonymous Functions

Functions do not necessary need to have a name (like `square` in the previous example). We call a function without a name an *anonymous* function. It can be defined like this:

```
[1, 4, 9]
```

```julia
map(a -> a*a, [1, 2, 3])
```

(more information: Julia Documentation)

Optional Arguments

In many cases the arguments of a function can be divided into *mandatory* and *optional* arguments. This can be done like this:

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

```julia
mylog(a,b=e) = log(a,b)
```

Here we defined a function `mylog` that by default uses the exponent `e` for the calculation of the logarithm. We thus can call it like this:

```
0.9102392266268373
```

```julia
mylog(3.0)
```

I.e. the second argument has been omitted. We note that the second argument to `log` actually already is a positional argument.

Note

Positional arguments are usually used when a small number of arguments is optional and when there is a clear hierachy, which argument is *more* optional.

(more information: Julia Documentation)

Keyword Arguments

While positional arguments are very useful, they have limitations. Lets consider the function

```julia
function foo(a=1, b=2, c=3)
...
end
```

One can easily call this function as

```julia
foo()
foo(1)
foo(1,2)
foo(1,2,3)
```

But what is if we just want to pass `c` to `foo` and use the default values for `a` and `b`? This is not possible.

Keyword arguments can solve this problem and are nowadays a crucial element in many Julia APIs. The idea is to give the argument a name (i.e. key) and then it is possible to change the order of arguments.

As an example we consider the a function

```
plot(x,y)
```

that is supposed to plot a graph with coordinates `x,y`. Such a function can have several parameters such as

- the line width
- the line style
- the color of the graph

In Julia we can define `plot` like this:

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

```julia
function myplot(x,y; width=1, style="solid", color="green")
    @info width, style, color
    # here do the actual plotting
end
```

Let us see how we can call this function.

```
[ Info: (1, "solid", "green")
[ Info: (1, "solid", "blue")
[ Info: (3, "dashed", "green")
```

```julia
with_terminal() do
    x = collect(1:10)
    y = collect(1:10)

    myplot(x,y)
    myplot(x,y;color="blue")
    myplot(x,y;style="dashed",width=3)
end
```

Note

The semicolon is necessary when defining the keyword arguments but they are not necessary when calling it.

When having a chain of functions it can be convenient to bundle keyword arguments and pass them further. This can be done like this:

```
[ Info: (10, Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}())
[ Info: (5, Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}())
[ Info: (5, Base.Pairs(:mysecondarg => 13))
```

```julia
with_terminal() do
    function bar(x; myarg=10, kargs...)
        @info myarg, kargs
    end

    function foo(x; kargs...)
        bar(x; kargs...)
    end


    # Lets call it without keyword arguments
    foo(0)
    # Lets change myarg
    foo(0, myarg=5)
    # And now lets pass another keyword argument
    foo(0, myarg=5, mysecondarg=13)
end
```

In practice this means that it is possible to tunnel arguments into low-level functions without the need to change the high-level interface.

(more information: Julia Documentation)

Evaluation Strategy

An interesting question when using functions is what evaluation strategy is used. Julia has call-by-value semantics, which means that a copy is being made when passing a value to a function. It also means that the arguments are first evaluated before they are passed to the function.

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

```julia
function evaluationStrategy(a)
  a = 7
  return
end
```

```
[ Info: 3
```

```julia
with_terminal() do
    y = 3
    evaluationStrategy(y)
    @info y
end
```

Hence, one can see that a copy of `y` has been made when passing it to the function `evaluationStrategy`.

However, for many types in Julia (so-called mutable types) the variable actually stores only a reference to the data. This is similar like in C where a pointer is often used to store a reference to a complex data type. Lets see this in action using the array datatype.

```
evaluationStrategy2 (generic function with 1 method)
```

```julia
function evaluationStrategy2(a)
  a[1] = 7
  return
end
```

```
[ Info: [7]
```

```julia
with_terminal() do
    z = [3]
    evaluationStrategy2(z)
    @info z
end
```

You can see, what has changed is the data behind the reference `z`, its not `z` itself.

Methods

Functions in Julia map arguments to return values, whereas the *type* of the arguments matters. Thus we can have:

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

```julia
begin
    foobar(x::Int) = x
    foobar(x::Float64) = -x
end
```

Lets call them:

```
[ Info: 1
[ Info: -1.0
```

```julia
with_terminal() do
    @info foobar(1)
    @info foobar(1.0)
end
```

So it seems that Julia allows the same function name for different argument types. Both functions are still related and we call `foobar` a *method*. You can also list all functions associated with it:

```
┌ Info: # 2 methods for generic function "foobar":
│ [1] foobar(x::Int64) in Main.var"workspace#3" at /Users/knopp/tuhhCloud/Lectures/WS2
└ [2] foobar(x::Float64) in Main.var"workspace#3" at /Users/knopp/tuhhCloud/Lectures/W
```

```julia
with_terminal() do
    @info methods(foobar)
end
```

We will not go deeper into methods at this point but will have a dedicated lecture on this important topic.

Arithmetic Operators

Julia offers the well known arithmetic operators `+,-,*,/,^`. So you can do simple calculations:

```
ϕ = 45
```

```julia
ϕ = 3 * 4 + 17 + 2^4
```

In Julia operators are nothing special but regular functions. One can either use the infix notation

```julia
3 + 4
```

or the prefix notation

```julia
+(3,4)
```

The later is important if you want to implement an operator for a new type. In other programming languages this is called `operator overloading`, in Julia the transformation from infix to postfix notation is just syntactic sugar and done during code lowering.

Here is an example of implementing the addition for a new type:

```
▸MySuperInt(7)
```

```julia
begin
    import Base.+ # we need to import a method if we want to add a new function

    mutable struct MySuperInt # ignore this definition for the moment
        val::Int
    end

    function +(a::MySuperInt, b::MySuperInt)
      return MySuperInt(a.val + b.val)
    end

    MySuperInt(3) + MySuperInt(4)
end
```

Ranges and Vectors

We discuss array/vectors/ranges in more detail in the upcoming lectures. At this point we just want to briefly introduce some basics.

Vectors

Vectors can be created and handled like this:

```
a = [1, 2, 3, 4]
a[3] = 3
a = [1, 2, -10, 4]
```

```julia
with_terminal() do
    a = [1,2,3,4] # this is a length-4 vector

    @show a
    @show a[3] # we access elements using square brackets
    a[3] = -10 # setting a value
    @show a
    nothing
end
```

At this point you should already have seen that Julia uses 1-based indexing like Matlab.

Ranges

Ranges are vector-like objects, where the elements are not stored in memory but are created on the fly. We use them extensively for indexing and iteration. Here are some examples ranges:

```
a = 1:4
a[3] = 3
collect(a) = [1, 2, 3, 4]
collect(1:2:9) = [1, 3, 5, 7, 9]
```

```julia
with_terminal() do
    a = 1:4

    @show a
    @show a[3] # we access elements using square brackets
    @show collect(a) # convert a range into a dense vector using collect
    @show collect(1:2:9) # every second element
    nothing
end
```

In addition to the special range syntax, there is also a function syntax that is in many cases more flexible:

```
collect(range(1, 4, length = 5)) = [1.0, 1.75, 2.5, 3.25, 4.0]
collect(range(1, length = 5, step = 0.5)) = [1.0, 1.5, 2.0, 2.5, 3.0]
```

```julia
with_terminal() do

    @show collect( range(1, 4, length=5) )
    @show collect( range(1, length=5, step=0.5) )
    nothing
end
```

Control Flow

Next we discuss the control flow expressions that are available in Julia

Conditional Evaluation

Quite often one wants to execute different code path depending on the *value* of an expression. This is done using the well-known `if` statement:

```
"$(s) is a student!" = "Peter is a student!"
```

```julia
with_terminal() do

    students = ["Marie", "Peter", "Klaus", "Waltraut"]
    teachers = ["Hugo", "Jane"]

    s = "Peter"
    if s in students
        @show "$s is a student!"
    elseif s in teachers
        @show "$s is a teacher!"
    else
        @show "$s is neither a teacher nor a student"
    end
    nothing
end
```

Also the ternary operator is available:

```
(b, c) = (14, 7)
```

```
with_terminal() do

    a = true

    b = a ? 14 : 7
    c = !a ? 14 : 7

    @show b,c
    nothing
end
```

One interesting question for `if` statements is whether they have their own scope. Lets consider the following C code:

```
if( ... ) {
  int i = 10;
} else {
  int i = 20
}

// do something with i
```

This will not work, since the variable `i` is only locally declared in the if statement. Instead one needs to first declare the variable:

```
int i;
if( ... ) {
  i = 10;
} else {
  i = 20
}

// do something with i
```

In Julia `if` statements do not have their own scope and thus we can do:

```
i = 10
```

```
with_terminal() do
    a = true

    if a
        i = 10
    else
        i = 20
    end

    @show i
    nothing
end
```

Short-Circuit Evaluation

The boolean operations `&&` and `||` are usually used within `if` statements:

```
don't something with Any[]
```

```
with_terminal() do

    a = [] # I am an empty array

    if !isempty(a) && a[14] == 7
        println("do something with $a")
    else
        println("don't something with $a")
    end
end
```

You can see that the second part in the `if` clause is not evaluated since `a[14]` actually would throw an error. This is because of the following rules:

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to true.
- In the expression `a || b` the subexpression `b` is only evaluated if `a` evaluates to false.

But actually this is not limited to `if` statements, one can actually also use this for regular control flow:

```
This is only shown in debug mode!
No error happened, we can proceed!
```

```
with_terminal() do

    debug = true

    debug && println("This is only shown in debug mode!")

    error = false

    error || println("No error happened, we can proceed!")
end
```

Note

In many cases the code remains actually more readable when sticking to simple `if` statements.

Repeated Evaluation

Another very important control flow mechanism is repeated evaluation or loops. They allow to repeat code fragments for a certain number of times. Julia supports two types, `for` loops and `while` loops. Lets see both in action:

```
1 2 3 4
4 6 8
```

```julia
with_terminal() do
  for i=1:4
    print("$i ")
  end

    println("")

  l = 4
  while l< 10
    print("$l ")
    l += 2
  end

end
```

Both should be pretty clear. `for` loops are very powerful in Julia and can be used to loop over containers quite easily.

```
[ Info: Hans
[ Info: Judith
[ Info: Peter
[ Info: Angelina
```

```julia
with_terminal() do
  global names = ["Hans", "Judith", "Peter", "Angelina"]

  for name in names
    @info name
  end
end
```

Now what is if you need both the element and the index in the `for` loop?

```
[ Info: ("Hans", 1)
[ Info: ("Judith", 2)
[ Info: ("Peter", 3)
[ Info: ("Angelina", 4)
```

```julia
with_terminal() do
  for (i,name) in enumerate(names)
    @info name, i
  end
end
```

Thats cool, isn't it?

Custom and Composite Types

In many situations one wants to create new types that group several values together. In Julia this is done by `structs`. For instance we can create a struct `Student` that has two members `name` and `age`:

```julia
struct Student
    name::String
    age::Int
end
```

Lets create a student

```
s = ▸Student("Tobi", 21)
```

```julia
s = Student("Tobi", 21)
```

We can access the members using the dot syntax

```
"Tobi"
```

```julia
s.name
```

```
21
```

```julia
s.age
```

Immutability

The struct that we discussed so far is a so-called *immutable* struct. Immutable means that we cannot change it

```
setfield!: immutable struct of type Student cannot be changed

 1. setproperty!(::Main.var"workspace#3".Student, ::Symbol, ::String) @ Base.jl:39
 2. top-level scope @ Local: 1  [inlined]
```

```julia
s.name = "Franz"
```

It might not be clear at this point what the advantage of immutability is but you can basically think of this as a hint for the compiler that allows for several optimizations that improve the execution speed.

If you need mutability just use a mutable struct

```julia
mutable struct MutableStudent
    name::String
    age::Int
end
```

```
c = ▶MutableStudent("Karl", 40)
```

```julia
c = MutableStudent("Karl", 40)
```

```
10
```

```julia
c.age = 10
```

What is important for `structs` and `mutable structs` is that you specify the types. If you do not do this, it is possible that members can change their type during lifetime, which will prevent emitting efficient code.

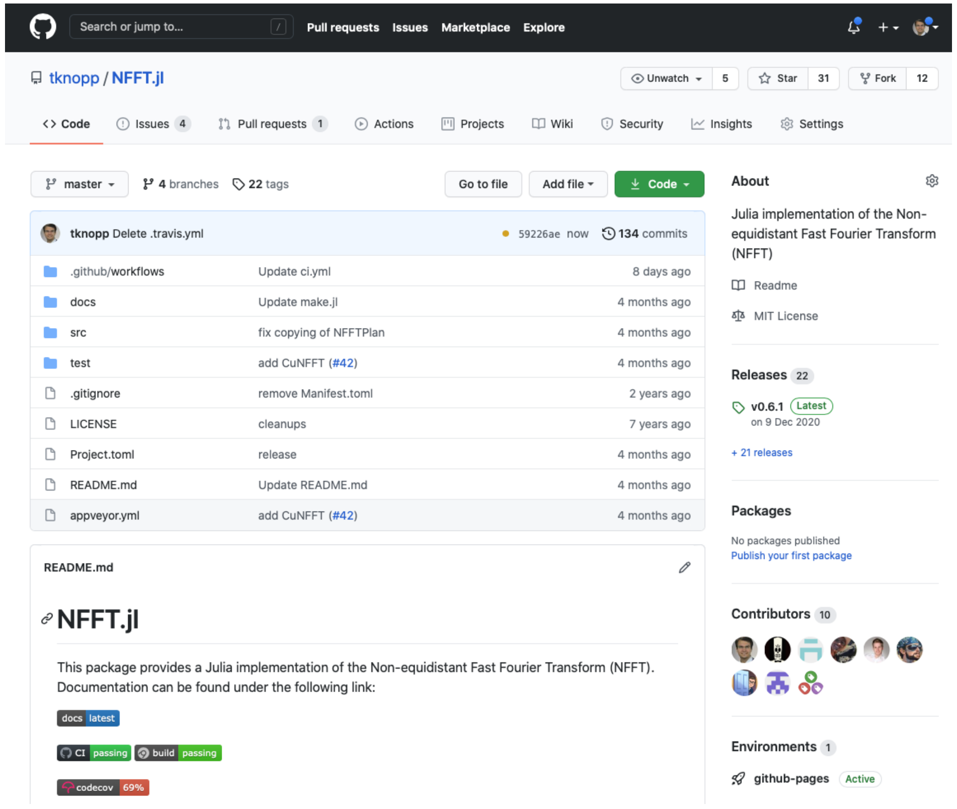
```
[ Info: true
[ Info: false
```

```julia
with_terminal() do
    @info isimmutable(s)
    @info isimmutable(c)
end
```

Packages and Modules

When developing a large codebase it is important to structure the code in a way that common functionality is grouped together. In Julia this is done using `modules` . At this point in the lecture we will not go into details of modules.

In Julia a package is a more general concept. A package can consist of one or several modules that is grouped together such that other people can use it. Packages in Julia usually correspond to a git repository (hosted e.g. at GitHub). Here is an example package *NFFT.jl* that allows for carrying a generalization of the Fast Fourier Transform (FFT):



tknopp / NFFT.jl

tknopp Delete .travis.yml • 59226ae now 134 commits

.github/workflows	Update ci.yml	8 days ago
docs	Update make.jl	4 months ago
src	fix copying of NFFTPlan	4 months ago
test	add CuNFFT (#42)	4 months ago
.gitignore	remove Manifest.toml	2 years ago
LICENSE	cleanups	7 years ago
Project.toml	release	4 months ago
README.md	Update README.md	4 months ago
appveyor.yml	add CuNFFT (#42)	4 months ago

About

Julia implementation of the Non-equidistant Fast Fourier Transform (NFFT)

Readme

MIT License

Releases 22

v0.6.1 Latest
on 9 Dec 2020

+ 21 releases

Packages

No packages published
Publish your first package

Contributors 10

Environments 1

github-pages Active

README.md

NFFT.jl

This package provides a Julia implementation of the Non-equidistant Fast Fourier Transform (NFFT). Documentation can be found under the following link:

One can see that the files and folders are structured in a certain way. Furthermore one can see some tooling that is offered in addition plain git:

- Continuous Integration (CI)
- Automatic generation of documentation
- Code coverage

Package Ecosystem

Julia has a very large package ecosystem and thus many things to not need to be implemented by yourself. For instance:

- You need to do Spline interpolation on arbitrary dimensioned arrays? Just use Interpolations.jl
- Need to solve differential equations? Just use DifferentialEquations.jl
- You need to develop a neural network? Just use Flux.jl

It should be noted that the entire Julia ecosystem is, similar to wikipedia, a collection of open source projects. Thus, the quality of packages varies greatly. But there is a certain fraction of high-quality packages out there that are already more sophisticated than similar packages in Python or Matlab. If you want to explore the package ecosystem you can visit JuliaHub.

Using Packages

At this point, again, we do not detail, what a package is exactly. Instead we just want to make you aware that they exist and how they can be used.

To install a package you go into package mode by entering `]` and then enter (if you want to install *Interpolations.jl*):

```
pkg> add Interpolations
```

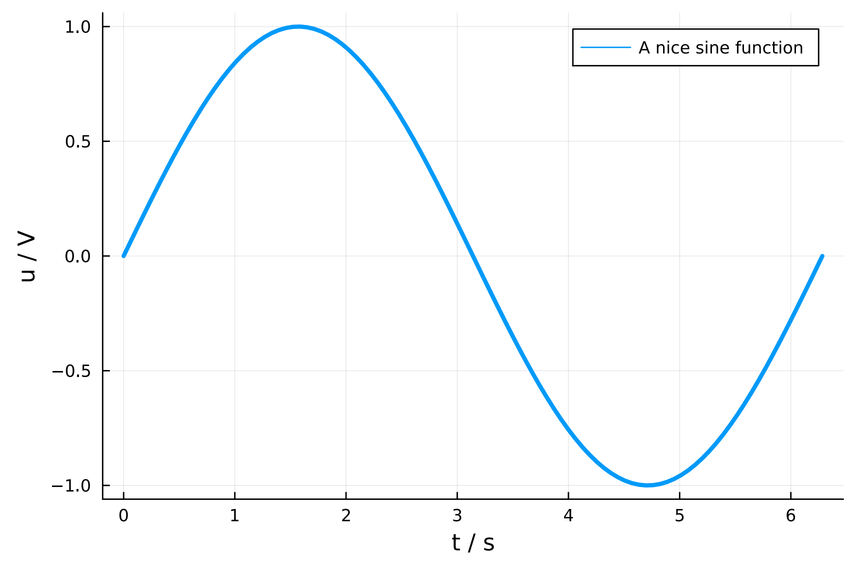
That's it. After you have installed it you need to load the package. This is done in Julia by:

```
using Interpolations
```

Then all functions exported by *Interpolations* are brought into the current namespace and can be used.

Visualization

There are various packages in Julia to perform visualization, i.e. plotting of functions or displaying images. Two popular packages you should be aware of are *Plots.jl* and *PyPlot.jl*. We will mostly use *Plots.jl*.



```
begin
    t = range(0, 2π, length=100)
    u = sin.(t)
    p = plot(t, u, label="A nice sine function", lw=3)
    xlabel!(p, "t / s")
    ylabel!(p, "u / V")
end
```