Julia Starter Pack

Installation

Julia executables can be downloaded from the official Julia website. Just follow the instructions for your platform.

Workflows

There are multiple ways of writing Julia code that correspond to different needs. As far as text editors/IDEs go, the best option is VSCode with the Julia extension, which is the best supported editor, as it provides, linting, suggestions, live documentation and executing your code interactively. This workflow is best for working on multiple files, say when you want to implement a bunch of functions.

For stuff like data analysis or live presentations, Notebooks are what you want. Julia can be used in Jupyter Notebooks thanks to the IJulia package (there's also a VSCode extension that allows editing Jupyter Notebooks). Another cool package is Pluto, which is a pure Julia Notebook that is meant for interactive prototyping and exploration. It differs from Jupyter in that notebook cells don't save their result, so every cell is run again when you reload the notebook. You can check out the Introduction to Computational Thinking videos by 3Blue1Brown on the Julia Youtube channel to see what it looks like.

Note: Previously, the best editor for Julia was Atom, with Julia Plugins, which could be found prepackaged as Juno. Since Microsoft acquired Github (the people who made Atom), the editor support has shifted to VSCode (which is an open source project by Microsoft).

Tidbits

This document is meant to get people started by giving them pointers. There are plenty of tutorials on Julia in various formats. If you know your way around programming, consulting the Julia Documentation and looking on stackoverlow as you code should be mostly enough.

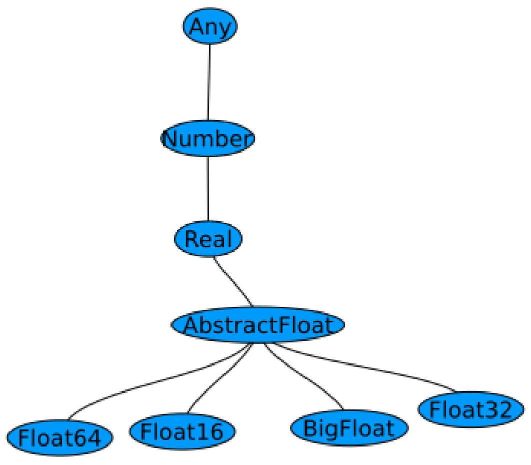
Types

One of the most important parts of Julia, which sets it apart from Python or Matlab is its type system. Julia has a rich type system which is an integral part of it's speed. As such, making use of the type system is crucial for getting the most out of Julia.

The Julia expression `x::T` means that variable `x` is of type `T`. For example, `n::Float32` means that variable `n` is a 32 bits floating point number (also known as `float` in C). Type annotations like this are optional, as the Julia compiler will do its best to infer the type of variables.

Julia types come in two varieties. Abstract types, which don't hold any data, and concrete types, which do. There is a partial order relation on types, denoted by `S <: T`, where we say that `S` is a subtype of `T`. Only abstract types can have subtypes, and there is an uppermost type called `Any` which is a supertype of every type. The hierarchy of types can be seen as a tree, with `Any` as the root, and concrete types at the leaves.

For example, the following diagram shows all the subtypes and supertypes of `AbstractFloat`, which is the abstract type for all floating point number types. Its supertypes are `Real` which is the abstract type for representing real numbers, then `Number` which is the abstract type for numbers in general (e.g. complex numbers, but it can be any kind of algebraic ring/field), and finally `Any`.



As you may gather from this example, abstract types are used to represent groups of types that represent the same kinds of objects (e.g. real numbers), or follow similar rules (e.g. algebraic structures like groups, rings, ...). We'll see later how this hierarchy of types is put to use.

Arrays

Julia likes arrays, and provides a bunch of tools to use them efficiently. First of all, arrays start at 1, and you can pass ranges of indices, like

```
a[1:n]
```

Like in Matlab, you select the last element of an array with `a[end]`, and all elements along a dimension with `a[i,:]`.

In order to create an array with specific dimensions (which is always recommended), you can use

```
a = Array{T,N}(undef, dims)
```

where `T` is the element type of your array (e.g. `Float64` for double precision floating point numbers), and `N` is the dimension of your array, with `dims` being a tuple of integers denoting the size of each dimension. For example, to create a 50 by 100 array of integers, you can use

```
a = Array{Int,2}(undef, 50, 100)
```

There are also special constructors like `zeros` and `ones` that work the same way as their Matlab equivalent. For creating an array filled with a given value, use `fill`.

As a final note somewhat related to arrays, Julia has a nice syntax for nesting for loops:

```
for i in 1:n, j in 1:m
    a[i,j] = rand(Int)
end
```

Functions

There are multiple ways to define a function, for example

```
function foo(x,y)
    return x^2 - y
end
```

By default, Julia returns the result of the last expression in the function body, so the above is equivalent to

```
function foo(x,y)
    x^2 - y
end
```

For simple expressions like that, you can also use

```
foo(x,y) = x^2 - y
```

You can also use anonymous functions, also known as *lambda expressions*. These don't need to be assigned to a variable, and can just be passed as is to a function. (You can also pass any function name as a function argument)

```
foo = function(x,y)
    return x^2 - y
end

bar = (x,y) -> y^2 - x
```

You can provide default values for arguments of your function, as well as keyword arguments:

```
function henrici(A::AbstractMatrix; normalize=true)
    λ = eigvals(A)
    frob = norm(A)
    # by default, calling norm on a matrix returns the Frobenius norm
    if normalize
        return sqrt( frob^2 - sum(abs.(λ).^2) ) / frob
    else
        return sqrt( frob^2 - sum(abs.(λ).^2) )
    end
end
```

Calling this function like `henrici(A)` will return the normalized Henrici's departure from normality, while calling `henrici(A, normalize=false)` will return the un-normalized one (you don't need to what this means, it's just an example).

As a side note, You can use Unicode characters in Julia code, in the REPL and editor (including Jupyter and Pluto), special characters like `λ` can be obtained by typing `\lambda<TAB>`. These characters can also be used in functions names, which means Julia code quite often closely resembles the math it's supposed to implement.

One important thing about functions is that functions that modify their input are denoted with a `!` at the end of their name, for example `push!(xs, x)` is a function that appends `x` at the end of the array (or collection) `xs`. These *in-place* functions are pretty important for maximizing performance, as it is very costly to allocate memory to store the result of a function call.

Finally, there's a special syntax to evaluate a function element-wise on an array, for example

```
 zs .= foo.(xs,ys) .- xs .* ys
```

As you can see, this also works for assignment and infix operators like `-` and `*`. This behavior is called *broadcasting*, and puts everything in a loop, so the previous line is equivalent to

```
for i in 1:length(zs)
    zs[i] = foo(xs[i], ys[i]) - xs[i] * ys[i]
end
```

Unlike Matlab, this rule is applied consistently for every function, so calling `sin.(A)` on a matrix `A` computes the sines of the elements of that matrix, while `sin(A)` computes the matrix sine (like the matrix exponential, but with sine instead)

Finally, there's a useful macro for broadcasting long expression called `@.`. The previous example can be rewritten as

```
@. zs = foo(xs,ys) - xs * ys
```

which is much easier to read.

Methods

In the previous section, I've omitted discussing how types are used in relation to functions. In reality, there is a distinction between *functions*, which are really just a name, and *methods*, which are implementations of a given function for a specific combination of argument types.

To illustrate this, let's revisit the `henrici` function. When we defined the header

```
function henrici(A::AbstractMatrix, normalize=true)
```

we used the abstract type `AbstractMatrix` to specify that this is a generic implementation for any matrix type. Now, if there was a specific subtype of `AbstractMatrix` for which we knew a better implementation than the generic one, we'd like to use that implementation for matrices of that type. This can be done by overloading the `henrici` function by defining a new method like this

```julia
function henrici(A::MySpecificMatrixType, normalize=true)
    # insert implementation for MySpecificMatrixType
end
```

To decide which method to use for a given function call, the Julia compiler looks at the types of the arguments and looks for the most specific method for that particular combination of types. If no method exists for the exact types, it takes the method from the supertypes if it can find one, otherwise it raises an error.

This approach is called *multiple dispatch* and is one of the cornerstones of Julia's design, as it allows flexibility between using a default implementation and optimized implementations for specific types, without having to write the same method for each individual type.

As a final note, the Julia compiler only performs optimizations when compiling methods, so using methods is very important for performance.

Documentation

For the Julia core, as well as community packages, documentation can be accessed in two different ways. You can read the documentation in the manual/website (if the package developer has made one), or you can access the in-code documentation similar to Matlab's help command by entering

```
? <function-name>
```

in the Julia REPL. VSCode also has a feature that allows you to view the documentation of a function by hovering over its name in your code, as well as a documentation tab. Pluto

If all else fails, you can always look at the package's code on Github, but you should probably take that as a sign that the package isn't mature enough.

Packages

Unlike Matlab, most of the cool features from Julia are made by the community as open source packages. This has several implications. First, while that means the Julia ecosystem can grow much faster than if it were done in closed source by a team of company developers, it also means that some packages are not as well tested and documented as others. Nonetheless, some big packages such as DifferentialEquations are very mature, and are probably one the main reasons for people starting to use Julia in their projects.

Second, developing and publishing your own package is quite easy, and most open source packages are hosted on Github where you can read the code and documentation. You can also browse registered packages here.

Installing packages in Julia is very easy. In the Julia console, one can press `]` to open the package manager. Installing a package is simply done by entering

```
add <package-name>
```

so for example,

```
add DifferentialEquations
```

adds DifferentialEquations to your main environment. Some packages aren't registered, but can be installed from Github (or any git repository) by specifying the Github repository, so for instance,

```
add "https://github.com/csimal/NetworkEpidemics.jl.git"
```

installs my own WIP package (I've been working on it on and off, so I wouldn't recommend installing it just yet)

Alternatively, you can install packages from the Julia REPL by entering

```
using Pkg; Pkg.add("<package-name>")
```

This is useful if you want to make a script to install a bunch of packages.

Note that packages share their dependencies, so when you install a new package, it may change the version of a dependency from another package, leading to that package having to be recompiled, or in the worst case, a version incompatibility, meaning those two packages just won't work together. For that reason, it's a good idea to create a new environment for each project.

To create a new environment, or switch to an existing one, simply enter

```
activate @<env-name>
```

in the package manager. The default environment for Julia 1.6 is `@v1.6`. The `@` here tells Julia to look for that environment in its home directory (e.g. `C:\Users\cedric\.julia\environments` on Windows, or `/home/csimal/.julia/environments` on Linux), otherwise it looks in the current working directory.

Before moving to the last part, it's important to know that while Julia is very fast at runtime, installing packages, especially for the first time, can take a while for large packages. This is because there are a lot of dependencies to install as well as precompilation to make loading packages at runtime faster. Once you've installed a couple of large packages, the process will become faster.

Here's a list of must-have packages, as well as some extras. Note that the convention is to append `.jl` to the name of your package on Github, so to find DifferentialEquations on Github, try googling `julia DifferentialEquations.jl`.

DifferentialEquations

THE package for all your ODE solver needs. Quite possibly the best suite of solvers in the market today. Seriously! It has solvers for stiff/non-stiff ODEs, Hamiltonian systems, Stochastic DEs, Delay DEs, you name it! They're also blazingly fast!

It also has a growing ecosystem of packages for various things such as sticking Neural Networks into ODES and vice-versa.

Also, check out `ModelingToolkit`, which allows defining an ODE symbolically and automagically spits out an optimized problem that you can then pass to your solver.

DynamicalSystems

A companion package to DifferentialEquations that implements a bunch of tools for analyzing chaotic dynamical systems, including Lyapunov exponents, bifurcation diagrams, Poincaré maps, ... It is pretty dope.

Plots

Plots is the standard plotting package for Julia. It's actually a metapackage that allows switching between multiple plotting backends such as

- `GR`: the default backend.
- `PyPlot`: Matplotlib's pyplot library. I use it when GR is being derpy, or to use the xkcd theme.
- `PlotlyJS` : A Javascript backend for interactive plots. Useful in notebooks.
- `PGFPlotsX` : A LaTeX based plotting package. A bit slower than the other ones, but the plots look really nice.

Note that you'll have to install them separately (they are standalone packages) to be able to use them as backends in `Plots`.

There are several other plotting packages other than Plots. Most notably

- `Makie` : more geared towards interactivity than Plots. I haven't used it much because I couldn't get it to work for the longest time. It also supports viewing 3d models out of the box.
- `Compose`/`Gadfly` : More inspired by R's ggplot, so this is more geared towards data analysis.

Graphs

The main network analysis package. Feature wise, it's not quite on par with Python's networkx, and some parts of the documentation are meh. See also the companion packages `SimpleWeightedGraphs` and `GraphRecipes` for weighted graphs and plotting graphs in Plots, respectively.

There's also `NetworkDynamics` that allows defining dynamical systems on networks, but it's still in early development and poorly documented. I've tried using it recently, and it turns out it doesn't handle directed networks correctly for my purposes (I'll probably bother them about it at some point).

Note: Graphs was formerly known as LightGraphs, which has just been deprecated. Most packages in the JuliaGraphs ecosystem have switched to Graphs, but some of them still use LightGraphs.

LinearAlgebra

Standard Linear Algebra functions. 'nuff said.

This one is actually part of the Julia standard library, so it's always installed. See also `Arpack` and IterativeSolvers for sparse linear algebra routines and iterative algorithms respectively.

For purists, `LinearAlgebra` also provides low level wrappers to LAPACK functions, though in practice, there is little need to use them directly (Speaking of which, you can natively call C or Fortran functions out of the box). You can even switch between different BLAS implementations at runtime.

DataFrames

Julia's equivalent to Python's pandas. It has been slowly crawling its way up various dataframe package performance benchmarks.

Flux

A Deep Learning framework entirely written in Julia. It also has tight integration with DifferentialEquations through the `DiffEqFlux` package, making it a must for anything involving neural networks and ODEs.

Other than Flux, `Knet` is another pure Julia Deep Learning framework, and there are bindings to Tensorflow and other frameworks.

For non Deep Learning stuff, there are many packages implementing well-known machine learning models. Check out MLJ and ScikitLearn for a good starting point.

JuMP

JuMP is the equivalent of DifferentialEquations or Plots, but for numerical optimisation. It provides ways to specify various types of optimisation problems and bindings to solvers both in Julia and to foreign libraries.

If you don't need that kind of industrial grade package, check out `Optim` and `BlackBoxOptim`.

PyCall

Python interop. It actually handles converting Julia objects to Python objects for you, so you can literally use *any* Python package with

```
const pkg = pyimport("<pkg-name>")
```

Note that the package needs to be installed in the python installation used by PyCall. By default, it uses its own installation, but you can tell it to use a specific one instead.

I've actually used it for a project where I wanted to find communities in an empirical network, and since the community detection algorithm from Graphs kinda sucks, I used the one from networkx. It literally **just worked**, with minimal fuss.

Most package that provide an interface to Python libraries (e.g. `PyPlot`, `ScikitLearn`, ...) are just thin wrappers around PyCall.

MATLAB

Call MATLAB from Julia! Like PyCall it handles converting Julia data to MATLAB for you. It might not work out of the box, so check the instructions on the package's Github page in case that happens.

If you just want to read `.mat` files from Julia, see MAT.

LaTeXStrings

This is what you use to have LaTex in your plot labels. For math formulas, you can simply use

```
L"<latex-math-expression>"
```

to get a LaTeX formatted string.

IJulia

Julia Kernel for Jupyter (speaking of which, the "Ju" in Jupyter is actually for Julia)

Pluto

Pure Julia notebook system. It's more lightweight than Jupyter, so I like using it for simple interactive exploration. The main difference between Pluto and Jupyter is that Pluto notebooks are *reactive*, in the sense that when rerunning a cell, all cells who depend on the result of that cell are rerun as well.

BenchmarkTools

Being able to measure how fast your code is running is critical for developing high performance programs. The Julia standard library already provides some simple macros to measure the execution time and memory use of a function call, but for proper benchmarking, you should use the `BenchmarkTools` package, which will run your function multiple times and give you a bunch of statistics, such as mean and median execution time.

Other useful packages include `Profile` and `PProf` for more insight into your functions.

Read this blog post for a practical introduction.

Example

The following is an example script showcasing how to use Plots and DifferentialEquations. Note that it may take a while to use these packages for the first time. It will be faster the next time you run it.

```
using DifferentialEquations
using Plots
#using MATLAB
using Graphs
using LinearAlgebra
using LaTeXStrings

 # generate a graph with 10 nodes using the Watts-Strogatz model
g = watts_strogatz(10, 4, 0.1)
L = - laplacian_matrix(g) # we negate L now for the heat equation

f! = function(du, u, p, t)
    mul!(du, L, u)
    # This is equivalent to du = L*u,
    # but it doesn't allocate memory for the result of L*u.
    # instead it puts the result directly into du
end

u0 = randn(10) # set the initial condition to be normally distributed
tspan = (0.0, 2.0) # time span for the ODE solver

# This constructs the ODE
prob = ODEProblem(f!, u0, tspan)

# and this solves it.
# Tsit5() is a 4-5 order Runge-Kutta method with adaptive timestep
sol = solve(prob, Tsit5())

plot(sol) # We can just plot the solution

f(t, u...) = (t, norm(u))
# You can also plot a function of the solution. The 0 index here is for time
plot(sol, vars=(f, 0:10...), label=L"\| u \|")

#= NB. This is a block comment
# Let's use Matlab to plot this instead
# expressions enclosed in $() are converted to Matlab format
mat"""
figure
plot($(sol.t), $(map(norm, sol.u)))
legend({'\$\\| u \\| \$'}, 'Interpreter', 'latex')
"""
=#
```