MA3227 Numerical Analysis II

Introduction to Julia

Simon Etter, 2019/2020

Both class, lab and homework assignments for this module will require you to be familiar with the Julia programming language. This document will help you getting started. You can either work through it on your own, or you can attend the "Getting started with Julia" session on Wednesday 15 January from 9-10am in S17-04-06.

1 Installing Julia

Please go to

`https://julialang.org/downloads`

and follow the instructions there to install Julia v1.3.1 on your computer. Any reasonably recent computer should be 64-bit, so it is best to download this version first and only try the 32-bit version if 64-bit did not work.

I highly recommend the Juno Integrated Development Environment (IDE) for writing Julia code. Juno is built on top of Atom, which is a general purpose text editor; hence in order to install Juno, you will have to install Atom first. A complete guide on how to install both Atom and Juno can be found under

`http://docs.junolab.org/latest/man/installation`.

2 Using Julia

Starting Julia

The main interface to Julia is the REPL (Read-Eval-Print-Loop). Have a look at Step 4 in the installation instructions linked above regarding how to open a Julia REPL in Atom.

Once the REPL is open, you can type simple mathematical formulae to check that everything is working. For example, typing `exp(1)` should give you the following.

```
julia> exp(1)
2.718281828459045
```

Loading code

The REPL is convenient to play around with short, one-line segments of code. For more advanced programming, you will want to write your code in a text file and then execute that file. To illustrate this process, create a file `test.jl` on your desktop and copy the following line to this file:

```
println("hello world")
```

The command to execute all code in a file is `include(path_to_file)`. On Windows, the path to the `test.jl` file that we just created should be

```
path_to_file = "C:\\Users\\<username>\\Desktop\\test.jl"   # Windows
```

(note that you need to type double backslashes because \ is a special character in a Julia string) while on Mac the path should be

```
path_to_file = "/Users/<username>/Desktop/test.jl"   # Mac
```

where in both cases you should replace `<username>` with your user name. If everything is working correctly, you should be able to do the following:

```
julia> include("/home/ettersi/Desktop/test.jl")   # Linux
hello world
```

Please contact me if you have issues getting this to work.

Typing the whole file path every time you want to run a file is inconvenient. To overcome this issue, the Julia REPL has the concept of a working directory: if you start a file path with something other than `C:\\` (Windows) or `/` (Mac), the REPL will add the path of the working directory to the beginning of the file path. A simple example should clarify the idea:

```
julia> pwd()   # Path to working directory.
               # At this point, this could be anything.
"/home/ettersi"

julia> cd("/home/ettersi/Desktop/")
       # Change the working directory to Desktop.

julia> pwd()   # Check that the working directory has been updated.
"/home/ettersi/Desktop"

julia> include("test.jl")   # No need to put full file path anymore.
hello world
```

Basic syntax

Julia's syntax is very similar to Matlab, so if you are familiar with Matlab you should have no problem writing Julia code. However, there are a few noteworthy differences.

- Julia arrays are indexed with square brackets, i.e. write `A[i,j]` instead of `A(i,j)`.
- Comment lines start with `#` rather than `%`.
- In Matlab, `zeros(5)` returns a 5×5 matrix of zeros. In Julia, `zeros(5)` returns a vector of length 5.

- To apply a function to each entry of an array, you have to add a `.` after the function name, otherwise you will get an error message. Example:

  ```
  julia> exp([0,1])
  ERROR: MethodError: no method matching exp(::Array{Int64,1})

  julia> exp.([0,1])
  2-element Array{Float64,1}:
   1.0
   2.718281828459045
  ```

- Unlike Matlab, Julia distinguishes between integers and floating point numbers. Some examples:

  ```
  julia> a = -1; 2^a
  ERROR: DomainError with -1:
  Cannot raise an integer x to a negative power -1.
  Make x or -1 a float by adding a zero decimal (e.g., 2.0^-1 or
  2^-1.0 instead of 2^-1), or write 1/x^1, float(x)^-1, x^float(-1) or
  (x//1)^-1

  julia> a = [1,2]; a[1.0]
  ERROR: ArgumentError: invalid index: 1.0 of type Float64
  ```

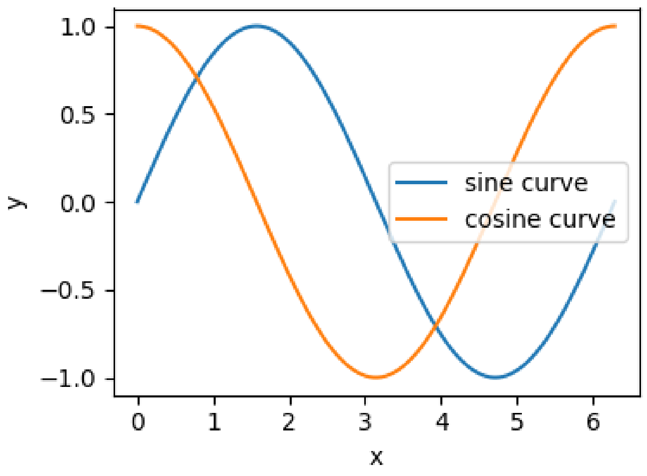
Exercise: Write a function `squared(n)` which returns the vector `[1,4,9,...,n^2]`. *Hint.* Type `?function` and hit enter to access the documentation for defining functions.

3 Creating plots using PyPlot

Plots can be created in Julia using the `PyPlot` package. To install this package, open the REPL, type `]` followed by `add PyPlot` and press enter. Once the installation has completed, you can load the package by typing `using PyPlot`.

The following example illustrates most of the commands that you will need for this class:

```
using PyPlot
figure(figsize=(4,3)) # Only needed to specify the figure size
x = LinRange(0,2pi,1000) # 1000 equispaced points in [0,2pi]
plot(x, sin.(x), label="sine curve")
plot(x, cos.(x), label="cosine curve")
legend(loc="best")
xlabel("x")
ylabel("y")
tight_layout()  # resizes the plot so x and y label are not cut off
savefig("test.png")
```

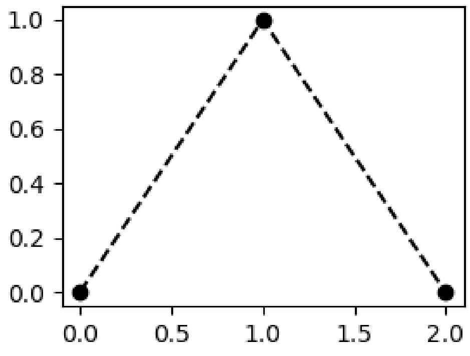


Exercise: Create a plot of $f(x) = x^2$ on $[-1, 1]$.

Other commands which may prove useful are:

- `semilogx()`: use logarithmic scale for x axis
- `semilogy()`: use logarithmic scale for y axis.
- `loglog()`: use logarithmic scale for both axes.
- `plot(x,y,format,...)` with `format` being a combination of the following allows you to change the line format.
  - Line styles: `-` (solid), `--` (dashed), `-.` (dash-dotted), `:` (dotted).
  - Markers: see `https://matplotlib.org/3.1.1/api/markers_api.html`.
  - Colours: `C0` to `C9` (default colour palette), `k` (black)

  Example: `plot([0,1,2], [0,1,0], "k--o")` yields the following.



4 Where to get help

Please do not hesitate to contact me (ettersi@nus.edu.sg) if you run into any issues with Julia, but please do so well before the submission deadlines. Julia-related issues will not be accepted as an excuse for late/incomplete submission.

Other useful resources:

- Type `?` followed by a function name or a keyword to access the documentation for that function or keyword.
- Julia documentation: `https://docs.julialang.org/`
- Google. Recommended websites are discourse.julialang.org and stackoverflow.com.