A few tips to start programming in Julia

(caveat: this was written for Julia v0.6 – some differences going to v1.0 may apply)

Download source code:

https://julialang.org/downloads/
https://docs.julialang.org/en/v1/index.html ä new 1.0 version

MAC ä click on self-installing pkg
LINUX ä untar and find exec in
> julia-0d7248e2ff/bin/julia
./julia to launch Julia's shell.

Inside the julia command line, one can write simple programs of multiple lines with loops etc (the program will be launched only after it's closed with a final "end")

OR

./julia -p (nprocs) file.jl to execute file.jl without opening the shell on nprocs processors
./julia -p (nprocs) file.jl to execute file.jl arg1 arg2... to give arguments

OR

julia> include(path/"script.jl") to run script.jl within Julia shell.

TROUBLES WITH PACKAGES

1) Sometimes, when the installation of Julia (or of its packages) goes wrong because of internet glitches or other (the available memory on the destination system is over or else) it is necesary to clean up Julia's cache, which is usually well hidden...
In these cases, one should do
rm -rf ~/.julia/.cache/Plots
rm ~/.julia/lib/v0.4/Plots.ji
(or similar, after having located the installed version and packages in the cache giving problems)

2) one can change julia's package directory by following these steps:
run export JULIA_PKGDIR=/your/directory in shell(or manually add a new environment variable JULIA_PKGDIR on windows)
run Pkg.init() in julia to initialize a new package system
copy REQUIRE from old directory to the new one
run Pkg.resolve() in julia
*copy the old package folder to the new directory only works for some packages.

Help while coding:

- Within the shell, at any time one can type
  julia> ?<command> -> help from the command line
- also typing <comm and then TAB will suggest possible completions
  e.g. sq \tab -> suggests sqrt sqrtm squeeze etc

Syntax miscellanea (just stuff I found useful for first tests/debugging)

- Legal/illegal statements:

```
x=1
print(x)
print(2x) -> 2  legal!
print(x2) -> error
print(xcos(x)) -> not legal
print((x)cos(x)) -> legal !
```

- Comments with #

- Printing stuff on screen:

println(x) -> print stuff on a new line (opposite to print(x))

- Stopping the program at some point:

inserting *error()* at any point in the batch script will stop the execution (annoyingly, with a message seeming like an error)

- Exiting

quit or exit() will stop the execution but also exit Julia's shell.

- ARRAYS/VECTORS

1-N ordering (not 0,N-1 as in C, IDL)
Entries stored in Fortran (column-major) ordering
Index ranges: start: stride:finish

```
name="abcd"   #defines a string of letter
print(name) -> abcd
print(name[2]) -> b
print(name[2:3])-> bc
print,name -> (print, "abcd")

x=10.0
print,x -> (print, 10.0)
print,x[1] -> (print, 10.0)
print,x[2] -> error
print(x) -> 10.0
```

1-dimensional vectors (they are intialised to ~0)

```
a=Vector{Float64}(10)  -> 2.38972e-314 2.38972e-314 etc..
a=Vector{Float32}(10)  ->  0  0 etc
a=Vector{Float16}(10)  ->fuzzy stuff which I don't understand
a=Vector{Float32}(1:10) -> 1.0 2.0 etc
a=Vector{Float16}(1:10) -> 1.0 2.0 etc
a=Vector{Float64}(1:10) -> 1.0 2.0 etc
a=Vector{Int64}(1:10) -> 1 2 etc
a=Vector{Complex64}(10) ->  8.98138f-19+1.4013f-45im etc
a=Vector{Complex64}(1:10) -> 1.0+0.0im  2.0+0.0im etc
```

Arrays:

```
 a=Array{Float64}(n,n)
a=Array{Float32}(n,n,n)
b=similar(a)    # creates an array with same indexing of a (but empty)
c=similar(a,Int) # as above, but c is of Integer type
```

- Fast ways of initialising sequential arrays

```
x=collect(1:5)
```

```
print(x) -> 1 2 3 4 5   [Integer]
x=collect(1.5:5.5) -> 1.5 2.5 etc [Float64]

x=collect(1.1:0.1:2)
print(x) -> 1.1  1.2  1.3  1.4 etc.   [MIDDLE NUMBER SPECIFIES
INCREMENT - otherwise 1 by default]

x=[sqrt(2*i) for i=1:10]  -> 1.41421  2.0  2.44949 ... 4.472
x=[i+sqrt(j) for i=1:3, j=1:2] -> 3 x 2 Array{Float64,2}

x=zeros(n,n,n)   #zero valued nxnxn array
x=ones(n,n,n)    #1-valued
x=rand(n,n,n)    #random values
```

- IF construct

```
if (value < 5)
value=10
else
value=20
end
```

- DO LOOPS

```
value=0
 for i in 1:10
       value+=1
       print(" ",value)
       end

list=[1,2,3,4,10,3]
value=0
for i in list
  value+=1
  print(" ",value)

end
# the above will print 1,2,3,4,5,6  because the list is made of
elements.
#Notice that
value=0
for i in list
  print(" ",list[i])
end
 will fail! Because list[list[5]] = list[10] does not exist!


for idx in eachindex(d)  #if d is an array (even multi-dimensional)
this give an easy way of looping over its elements
```

- FUNCTIONS

Typing the initial letters of a function in the command line will show possible alternatives and syntax. Also ?function_name will open a tutorial

- PACKAGES

Any available package is downloaded from the command like like

```
Pkg.add("namepackage") to install a package from the command line
```

```
Pkg.status()   to check what is currently installed
Pkg.update()   updates
```

If a package is not in the "official" Julia distribution (i.e. Some package developed by someone)

```
Pkg.clone("https://github.com/blabla.jl.git")
```

Very useful packages for our stuff:

```
Pkg.add("HDF5")  #support for HDF5 reading/writing
Pkg.add("FITSIO") #support for FITS reading/writing
Pkg.add("Optim")  #optimisation of code
Pkg.add("Devectorize") #devectorisation for speedup
Pkg.add("DistributedArrays") #domain decomposition for parallel
```

- PLOTTING

```
Pkg.add("Winston")   #very basic stuff
using Winston
→ https://github.com/JuliaGraphics/Winston.jl
Pkg.add("PyPlot")    #very sophisticated stuff
using PyPlot
→ https://github.com/JuliaPy/PyPlot.jl
Pkg.add("ImageView")  #image filtering/manipulation
```

- PARALLEL (to be properly written)

Use `DistributedArrays` for domain decomposition

- SYNTAX HIGHLIGHTING (in emacs)

in the .emacs file in the home directory add

```
 add-to-list 'load-path"/Users/francovazza/Downloads/julia-emacs-master/")
(require 'julia-mode)
```

where the julia-emacs master can be downloaded here https://github.com/JuliaEditorSupport/julia-emacs

- MAKING IT FASTER

  See useful guides here
- http://www.stochasticlifestyle.com/7-julia-gotchas-handle/
- http://www.stochasticlifestyle.com/optimizing-julia-performance-practical-example/
- http://www.exegetic.biz/blog/2015/10/monthofjulia-day-37-fourier-techniques/

General rules (might be subject to revision with experience…)

- when possible, wrap the code into functions ä this allows the compiler to know where the types cannot change and to best speed up.
- be consistent with the type of variables throughout the code; never implicitly change from integer to float or else ä this would speed up the code 10-fold;
- find "type instabilities" with @code_warntype in front of the call of your function (this works only if your function is "small enough" for Julia to

anlayze). Example: @code_warntype div(vx,vy,vz)...

- declare all possible variables as constant ä huge speedup
- pay attention to how equations are computed.
- Don't vectorise (unlike many other languages). Well written loops are much faster.
- Use the broadcast "." syntax ä x .= x .+ f.(x) is much faster than x=x+f.(x) [f is a function and x a vector], although they do the same thing, because the first equates ot a element by element loop.
- Try using "Optim"/"Devectorize" packages (but check performances: for well written codes they actually slow down the execution!)
- use @inbounds in front of loops where you are sure all indices are not exceeding the limits ä it makes the compiler skipping this check;
- use @simd before (indipendent) loops, @fastmath in front of operations
- limit the use of temporary arrays.
- Use @views when dealing with v[:,:,1]=etc... "slices" through larger arrays;
- Use @. for long expressions like fx=x^2.+sqrt(x)+x*b ....
- Pre-allocate arrays before they are first allocated within a function.