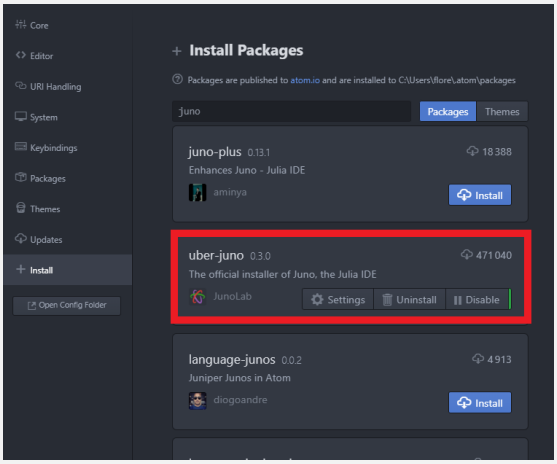
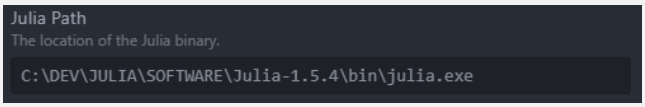


When searching for Juno in the bar, I look for the necessary pieces of Packages that would be useful, of which uber-juno and juno-client. © F. Poux

Note: *If the REPL does **not** seem to launch correctly and does not display the Julia logo, head over to the Juno Client Settings from the Package menu and verify the path to tour Julia executable.*



The path to my Julia executable. © Florent Poux

Bonus: Julia with Google Colab

You can also use a google Colab environment, but that needs a specif block of code to use Julia instead of Python. For this, you will find at this link a template that makes it possible to work directly within Colab. It also contains the main code of this tutorial.

Note: *Every time you want to use Julia on the Google Cloud, you need to run the first block, refresh the page, and continue directly to the second block without re-running the first until every element is ready for you.*

STEP 3: LOADING DATASETS

Great, so now we are ready to Julia code. First, let us discover where we are working, so our current working directory, using the command pwd(). Hmm, it looks like we are in the base directory, so let us change it to a project directory that you would create where you want to store most of your project (data, results, …) with `cd("C://DEV/JULIA/TUTORIALS/3D_PROJECT_1")`, followed by pwd() to check out the new path.

```
Out[2]: "C:\\DEV\\JULIA\\TUTORIALS\\3D_PROJECT_1"
```

Okay, we are all set. Let us download a dataset, for starters, a small noisy point cloud. For this, very handy, you can use the following command:

```
download("https://raw.githubusercontent.com/florentPoux/3D-small-datasets/main/tree_sample.csv","point_cloud_sample.csv")
```

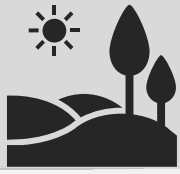
Note: *The download() command first takes the link to the data you want to download, which I dropped on my GitHub account, and then specifies its name locally after download.*

Great, we now have a local version of the data in our working directory that we specified with the cd() command. Now, how do we load it in the script? well, we will make use of a **Package** called DelimitedFiles.

Note: *A Package is a handy set of functions, methods, classes, and more that allow you to build on existing code without writing everything from scratch. The DelimitedFiles package allows manipulating (e.g., read and write) delimited files like our current point cloud at hand.*

For using a package, we first have to load the **package manager utility** by just typing using Pkg. To add a new package, it is pretty simple; we just write Pkg.add("DelimitedFiles"), and wait until the download + requirement checks are finished.

```
  Updating registry at `C:\Users\flore\.julia\registries\General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
Installed ImageMagick_jll ─ v6.9.12+0
Installed StructTypes ──── v1.8.1
Installed JSON3 ────────── v1.9.2
  Updating `C:\Users\flore\.julia\environments\v1.6\Project.toml`
```



What is great about this is that you do not have to worry about dependencies (other packages needed by the current) as all is handled for you! Cool, huh? And on top, we can create excellent packages easily to ensure reproducibility of the results, for example, and independent environments, but that is for another tutorial 😁.

Note: *Managing packages is then pretty straightforward, and we have a bunch of functions to update the packages, to know their current status, if we have any conflicts between them (rare), or to load unregistered packages by other like-minded coders, even one of your future local package* 😁. *I usually manage them by using the REPL and entering the package manager with the command] in the right environment. To exit the package manager, one needs just to do Ctrl+C .*

Ok, now that the package is installed (you only need to run this once per environment). You can use it in your current project by typing using DelimitedFiles, and also, if there are no conflicts of function names, you do not need to write from which package a function comes. to read a delimited file DelimitedFiles.readdlm() is equivalent to readdlm().

From there, let us read the point cloud at hand and store the data in the variable pointlist:

```
pointlist=readdlm("point_cloud_sam
ple.csv",';')
```

The first lines should look like below.

```
100000×6 Matrix{Float64}:
 41.6179  46.8634  18.311   0.357621  -0.003791  0.933859
 38.3256  48.8711  15.896  -0.709508  -0.073998  0.700802
 37.5081  49.965   18.053  -0.69199   -0.426478  0.582466
 37.1348  53.5679  17.27    0.104579  -0.212938  0.971453
 39.4802  54.7671   2.407   0.155142  -0.276558  0.948392
 31.3828  46.837    2.279  -0.256003   0.16218   0.952974
 37.8124  49.3977   8.908   0.908389   0.285243  0.305722
 36.4778  50.1293  11.803  -0.952869  -0.296249  0.065406
```

STEP 4: FIRST PRE-PROCESSING OPERATIONS

Okay, pretty cool up until now, hun? And now, let us see the first real surprise if you are used to other programming languages: **indexing**. You can try doing pointlist[0] to retrieve the first element. What are we getting? a *BoundsError*.

Haha, in Julia, **indexes start at 1**, so if you want to retrieve the first element (the X coordinate of the first point), you just input pointlist[1] that returns 41.61793137. A bit confusing at first, it is pretty handy and logical, at least from a scientific point of view 😅. So now, if you want to retrieve the first point, then you need to know that indexes work on the first axis (rows) first, followed by the second axis (column(s)), and so on. Thus, to retrieve the first point (first row and every column):

```
pointlist[1,:]
```

```
41.6179  46.8634  18.311   0.357621  -0.003791  0.933859
```

Very cool, and now, to get further, if we want to store the coordinates in the points variables and the normals in the normals variable, we just do:

```
points  = pointlist[:,1:3]
normals = pointlist[:,4:6]
```

Note: *If you want to know the type of a variable, typeof() is what you are looking for. typeof(points)will display that we deal with Matrices, which are an alias of 2-dimensional arrays. Also Float64 is a computer number format, usually occupying 64 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point. Double precision may be chosen when the range or precision of single-precision (Float32) would be insufficient.*

One last straightforward pre-processing step would be to know how to quickly sample the variable extracting, let say, 1 line every tenth. For this, we can do the following (a bit like