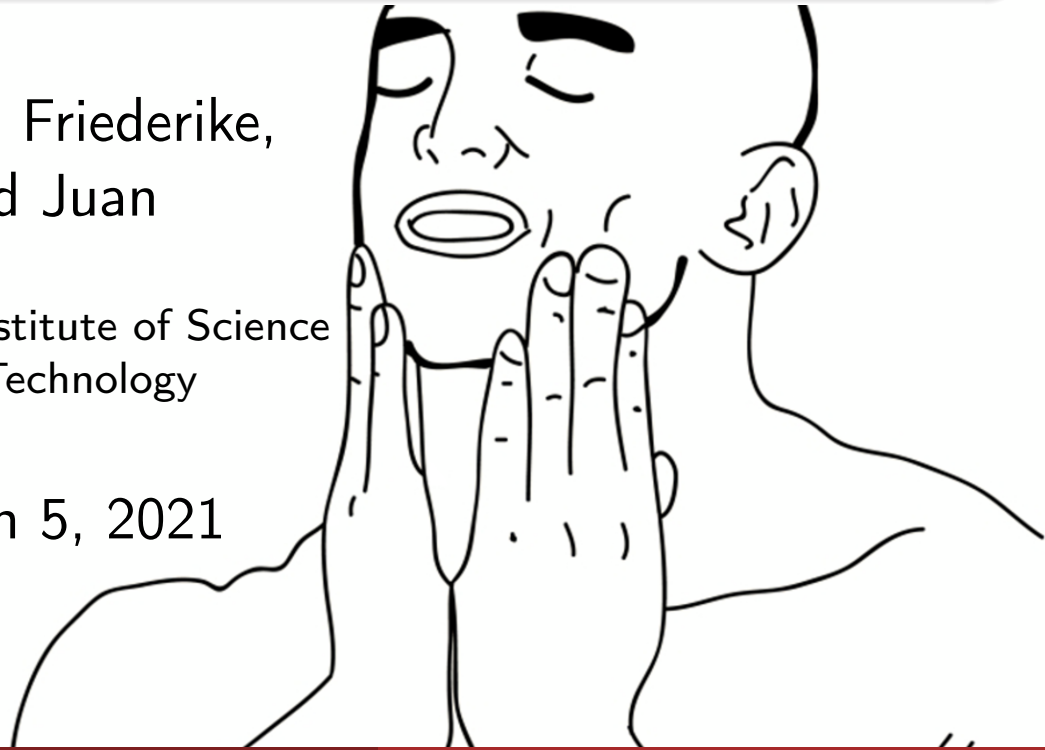
Mini course: Julia

Lecture 3: IDEs, Data Structures, Coding Practices, and Algorithms

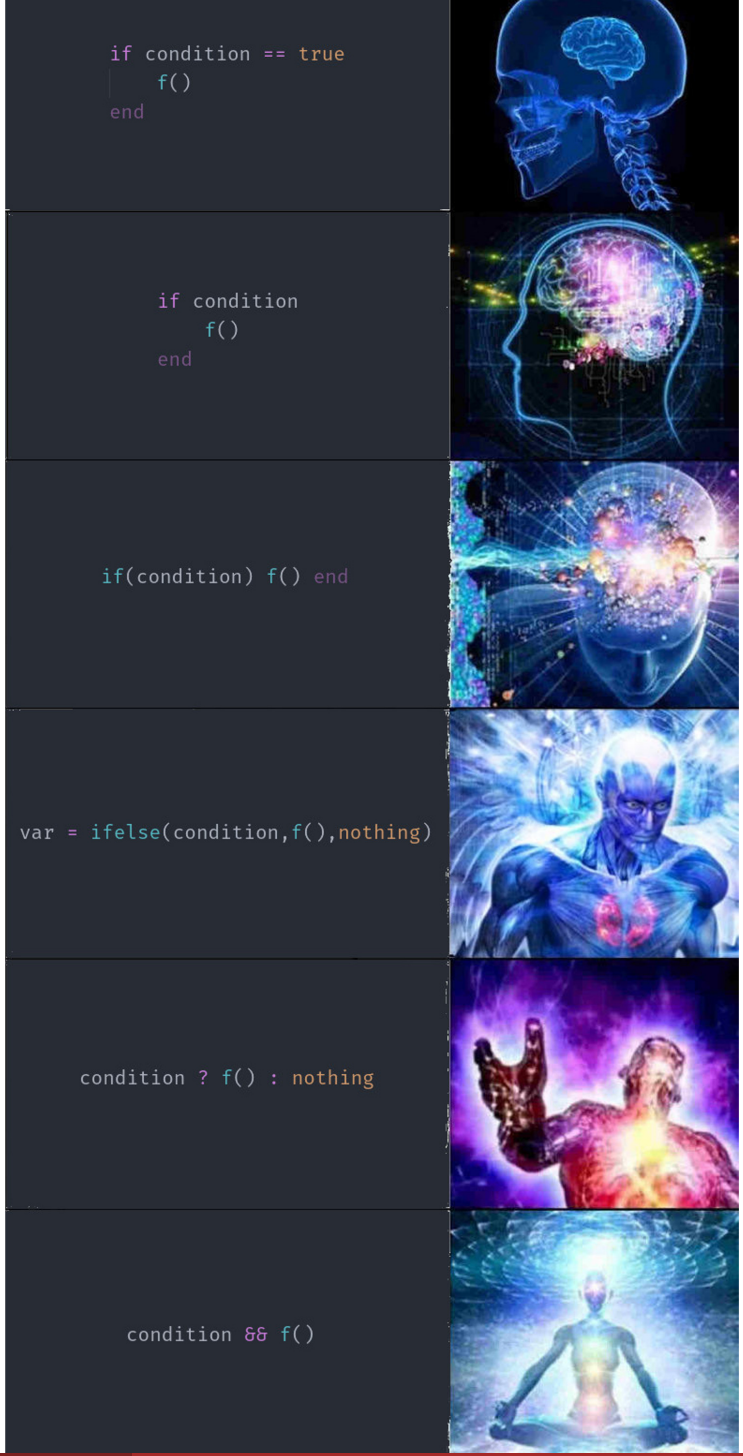
Gaston, Friederike,
and Juan

Okinawa Institute of Science
and Technology

March 5, 2021



1. IDEs & remote workflow
2. Data Structures
3. Coding Practices
4. Algorithms



Disclaimer

When designing this Skill Pill, we assumed the following

- You have your own workflow in regards to which software and libraries you work with.
- You have seen and are familiar with common data structures.
- You know how to program and use programming as part of your daily work.

As such, we have designed today's lesson so that you may begin using Julia in your work as soon as possible.

Workflow

Julia in VS Code

Jupyter notebook

- Step 1: Install Julia.
- Step 2: Run]add IJulia on Julia's REPL.
- Step 3: Install Anaconda distribution
- Step 4: Launch Jupyter lab/notebook from the Anaconda Navigator
- Step 5: Create a new jupyter notebook with Julia kernel.

Atom + Juno

- Step 1: Install Julia, and Atom.
- Step 2: Search in Atom for the packages: uber-juno, language-julia, julia-client, ink, and indent-detective.
- Step 3: Restart and launch Julia in the REPL inside Atom. It will gather all remaining necessary packages.

Visual Studio Code + Julia for VS

- Step 1: Install Julia, and Visual Studio Code.
- Step 2: Inside VSCode, go to extensions and search "julia". Install Julia for VSCode and restart.

Remote connection to your workstation

Depending on your workflow, you may want to have access to data, more computational resources, or just to avoid ~~stressing out the crappy Macbooks~~ using your personal laptop for running research code.
On Jupyter notebook, this is done very easily. Make sure you have installed Jupyter notebook on both the local and remote computers.
Assuming you have access (via VPN) and know the IP of your workstation to run the following ...

On remote host terminal:

```
$ jupyter notebook –no-browser –port=8889
```

Remote connection to your workstation

In the **local host**, run the following in the terminal (cmd for Windows):

To connect:

```
$ ssh -N -f -L localhost:8888:localhost:8889 user@remote_host_IP
```

Make sure to change *user* to your real username in **remote host**, and change *your_remote_host_name* to your address of your working station.

Example:

```
$ ssh -N -f -L localhost:8888:localhost:8889 gaston@10.2.80.145
```

Now open web browser and type:

Web browser address bar:

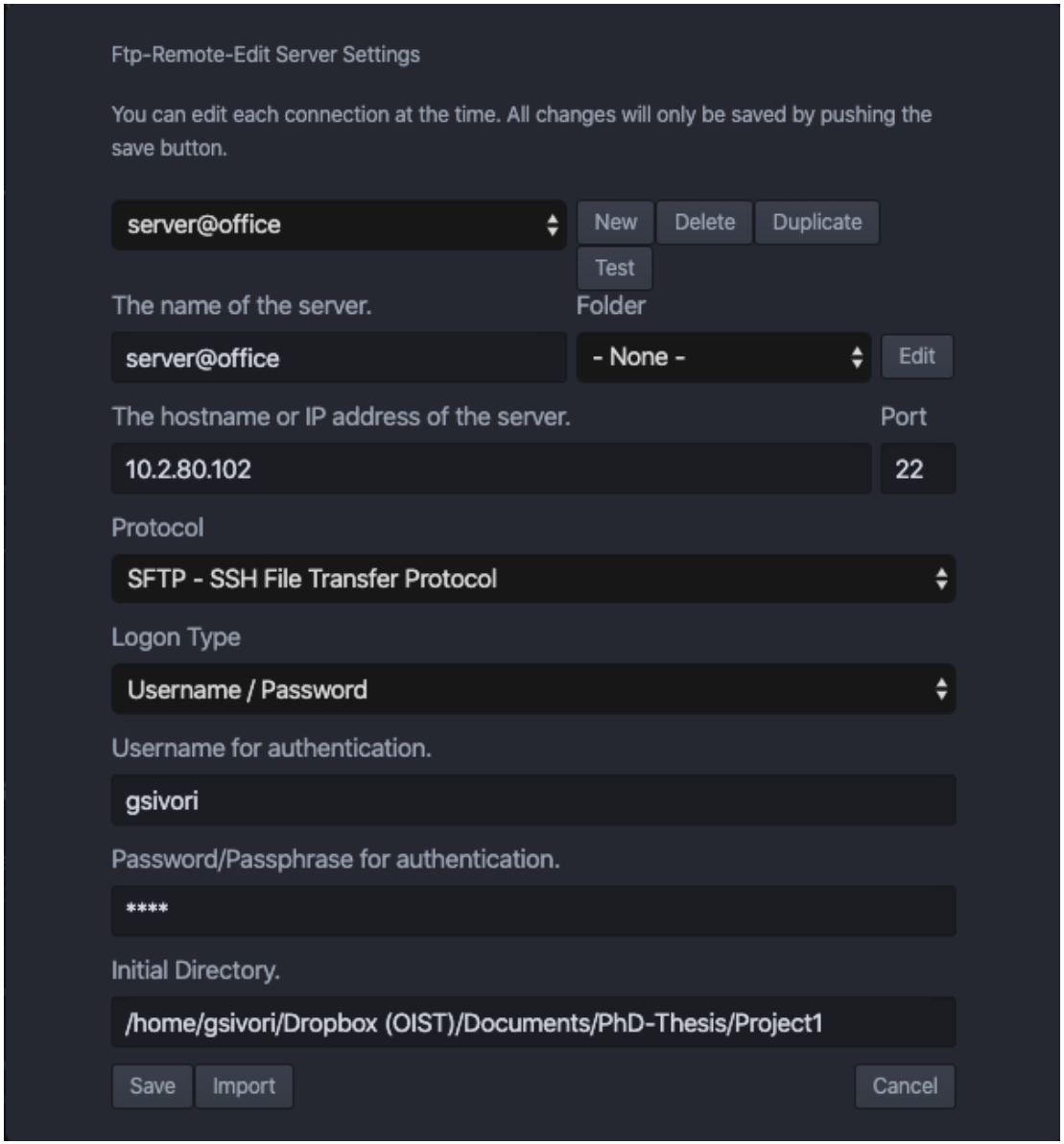
```
localhost:8888
```

Remote connection to your workstation

In Atom, the setup for remote workflow is very simple.

- Install the following package inside Atom: **ftp-remote-edit**.
- Go to Packages → Ftp-Remote-Edit → Toggle.
- Right click on Remote tab → Edit server.
- Name the server and input the necessary details.
- To use tmux. Go to **julia-client** package settings → Remote settings → Use persistent tmux session. Also check that the command to execute Julia remotely is appropiate.
- Start the remote server by going to Juno → Start Remote Julia Process.

Remote connection to your workstation



Datastructures.jl

DataStructures.jl has the following data structures:

- Deque (based on block-list)
- Stacks and Queues
- Accumulators and Counters
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- Dictionaries with Defaults
- Trie (Tree)
- Linked List
- Sorted Dict, Multi-Dict and Set
- Priority Queue

Datastructures.jl

DataStructures.jl has the following data structures:

- Deque (based on block-list)
- Stacks and Queues
- Accumulators and Counters
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- Dictionaries with Defaults
- Trie (Tree)
- Linked List
- Sorted Dict, Multi-Dict and Set
- Priority Queue

All information regarding these Data Structures can be found Here: `http://datastructuresjl.readthedocs.io/en/latest/index.html/`

All of these algorithms can be viewed with `@edit`. We'll use two more before hopping into algorithms: `Binary Trees` and `Priority Queues`

Deque

Based on contiguous blocks on memory. Similar to Julia's **Vector** implementation, but, as it grows, it avoids copying all values to a new Vector.

```
a = Deque{Int}()
isempty(a)            # test whether the dequeue is empty
length(a)             # get the number of elements
push!(a, 10)          # add an element to the back
pop!(a)               # remove an element from the back
pushfirst!(a, 20)     # add an element to the front
popfirst!(a)          # remove an element from the front
first(a)              # get the element at the front
last(a)               # get the element at the back
```

Circular Buffer

A circular buffer of fixed capacity. New items are pushed to the back of the list, overwriting values circularly.

```
cb = CircularBuffer{Int}(n) # n-element circular buffer
isfull(cb) # test whether the buffer is full
isempty(cb) # test whether the buffer is empty
empty!(cb) # reset the buffer
capacity(cb) # return capacity
size(cb) # same as length(cb)
push!(cb, 10) # overwrites front if full
pop!(cb) # remove the element at the back
pushfirst!(cb, 10) # overwrites back if full
popfirst!(cb)  # remove the element at the front
append!(cb, [1, 2, 3, 4]) # push at most last `capacity` items
convert(Vector{Float64}, cb) # convert items to type Float64
eltype(cb) # return type of items
fill!(cb, data) # grows the buffer up-to capacity
```

Stacks and Queues

```
s = Stack{Int}()  # create a stack
eltype(s)         # get the type of elements
push!(s, 1)       # push back a item
first(s)          # get an item from the top of stack
pop!(s)           # get and remove a first item
iterate(s::Stack) # Get a LIFO iterator of a stack
Iterators.reverse(s::Stack{T}) # Get a FILO iterator of a stack
s1 == s2 # check whether the two stacks are same
```

```
q = Queue{Int}()
enqueue!(q, x)
x = first(q)
x = last(q)
x = dequeue!(q)
```

PriorityQueue

A type of data structure that enqueues items by priority.

```
julia> # Julia code
       pq = PriorityQueue();
julia> # Insert keys with associated priorities
       pq["a"] = 10; pq["b"] = 5; pq["c"] = 15; pq
PriorityQueue{Any,Any,Base.Order.ForwardOrdering} with 3 entries:
  "b" => 5
  "a" => 10
  "c" => 15
julia> # Change the priority of an existing key
       pq["a"] = 0; pq
PriorityQueue{Any,Any,Base.Order.ForwardOrdering} with 3 entries:
  "a" => 0
  "b" => 5
  "c" => 15
```

Red-black Tree

A type of self-balancing binary tree that has an extra bit of information to make searching, inserting, and deleting operations with $O(log(n))$:
`https://en.wikipedia.org/wiki/Red%E2%80%93black_tree`

```
julia> tree = RBTree{Int}();
julia> for k in 1:2:20
         push!(tree, k)
       end
julia> haskey(tree, 3)
true
julia> tree[4]
7
julia> for k in 1:2:10
         delete!(tree, k)
       end
julia> haskey(tree, 5)
false
```

Coding Practices

Some recommended coding practices to consider with Julia.

- Measure performance with @time and check for memory allocations.
- Break a function into multiple definitions (proper use of multiple dispatcher)
- Consider variable type stability for compiler optimizations.
- Access arrays in memory order (along columns in Julia)
- Pre-allocate output.
- Use vectorized operation "@." for readability.
- Consider using @views to look at big arrays.
- Read through Profile.jl, BenchmarkingTools.jl, and Cthulu.jl packages for further optimizations and debugging.
- Diferences with other languages: `https://docs.julialang.org/en/v1/manual/noteworthy-differences/`

@time

Measure performance with @time. Notice how using global variable allocates unnecesary memory making this example run longer.

```
julia> x = rand(1000);
julia> function sum_global()
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;
julia> @time sum_global()
  0.017705 seconds (15.28 k allocations: 694.484 KiB)
496.84883432553846
julia> @time sum_global()
  0.000140 seconds (3.49 k allocations: 70.313 KiB)
496.84883432553846
```

@time

Compared to passing it as an argument:

```
julia> x = rand(1000);
julia> function sum_arg(x)
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;
julia> @time sum_arg(x)
  0.007701 seconds (821 allocations: 43.059 KiB)
496.84883432553846
julia> @time sum_arg(x)
  0.000006 seconds (5 allocations: 176 bytes)
496.84883432553846
```

Break function in many definitions depending input

```
using LinearAlgebra
function mynorm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svdvals(A))
    else
        error("mynorm: invalid argument")
    end
end
```

This is better:

```
norm(x::Vector) = sqrt(real(dot(x, x)))
norm(A::Matrix) = maximum(svdvals(A))
```

Consider variable type

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    return x
end
```

```
function foo()
    x::Float64 = 1.0 / rand()
    for i = 2:10
        x /= rand()
    end
    return x
end
```

Access arrays in memory order

Multi-dimensional arrays are stored in column-major order (arrays are stacked one column at a time).

```
julia> x = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
julia> x[:]
4-element Array{Int64,1}:
 1
 3
 2
 4
```

A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Looping will be faster if the inner-most loop index is the first to appear in a slice expression.

Pre-allocate output

If your function returns an Array or some other complex type, it may have to allocate memory. Unfortunately, often times allocation and its converse, garbage collection, are substantial bottlenecks. Compare this:

```
julia> function xinc(x)
           return [x, x+1, x+2]
       end;
julia> function loopinc()
           y = 0
           for i = 1:10^7
               ret = xinc(i)
               y += ret[2]
           end
           return y
       end;
```

Pre-allocate output

To this:

```
julia> function xinc!(ret::AbstractVector{T}, x::T) where T
           ret[1] = x
           ret[2] = x+1
           ret[3] = x+2
           nothing
       end;
julia> function loopinc_prealloc()
           ret = Vector{Int}(undef, 3)
           y = 0
           for i = 1:10^7
               xinc!(ret, i)
               y += ret[2]
           end
           return y
       end;
```

Pre-allocate output

This changes a lot.

```
julia> @time loopinc()
  0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc
      time)
50000015000000
julia> @time loopinc_prealloc()
  0.030850 seconds (6 allocations: 288 bytes)
50000015000000
```

Use vectorized operation for readability

Using . for broadcast opearation messes up readability. Use "@." for vectorized operations.

```
julia> f(x) = 3x.^2 + 4x + 7x.^3;

julia> fdot(x) = @. 3x^2 + 4x + 7x^3
```

If x is an Array, fdot(x) will have significantly faster performance.

Consider @views to look into slices of big arrays

```
julia> x = rand(10^6);

julia> fcopy(x) = sum(x[2:end-1]);

julia> @views fview(x) = sum(x[2:end-1]);

julia> @time fcopy(x);
  0.003051 seconds (7 allocations: 7.630 MB)

julia> @time fview(x);
  0.001020 seconds (6 allocations: 224 bytes)
```

Exercise

Implement your favorite algorithm in Julia

I will be coding the Lorentz Attractor as an example. Lorentz function given by:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Solve this for $t = 1 : 150$. Initial conditions $[1, 1, 1]$.
$\sigma = 10, \rho = 28, \beta = 8/3$.

Thanks for joining us!