2925

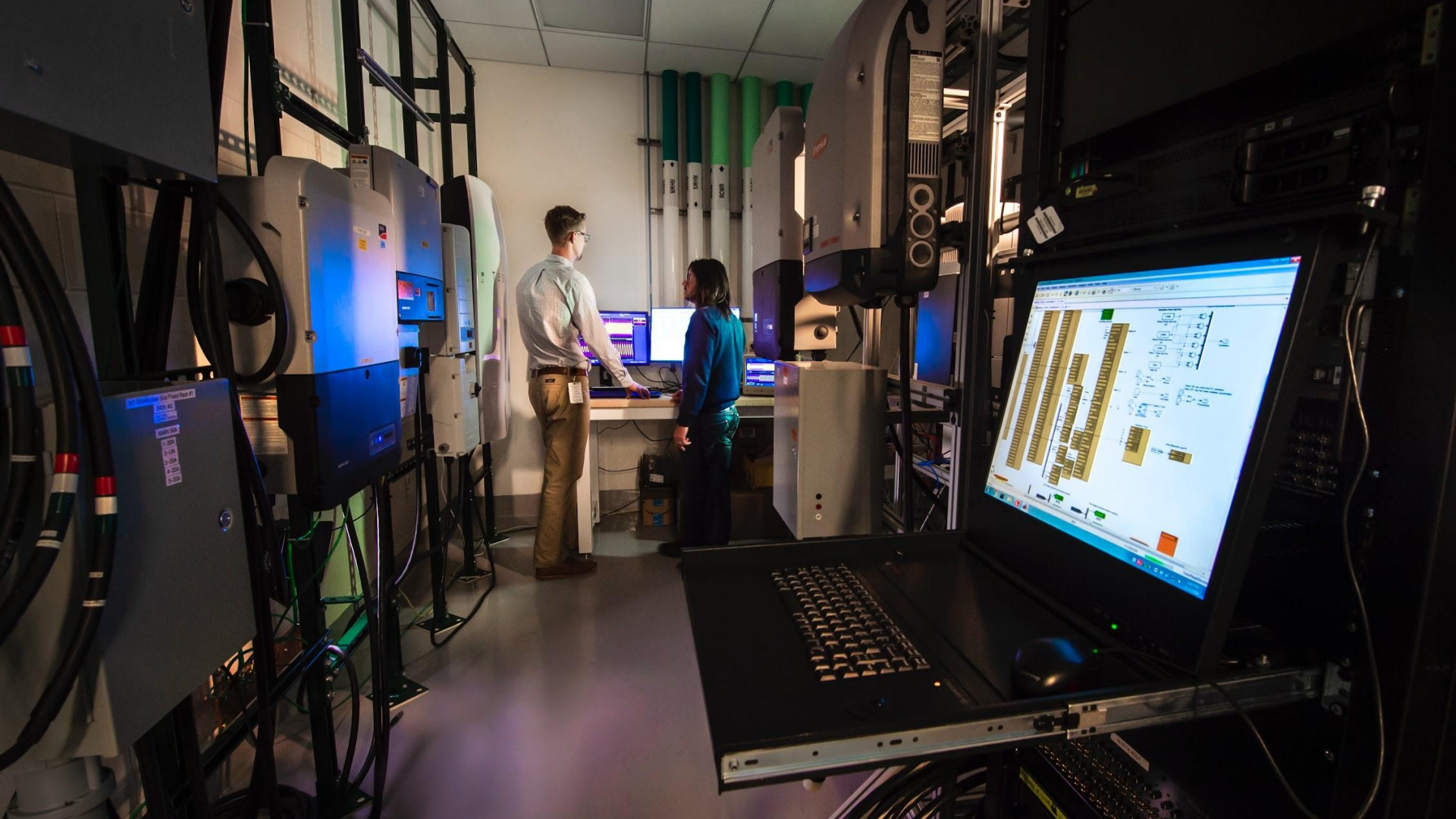
Michiel Stock
Bram De Jaegher
Daan Van Hauwermeiren

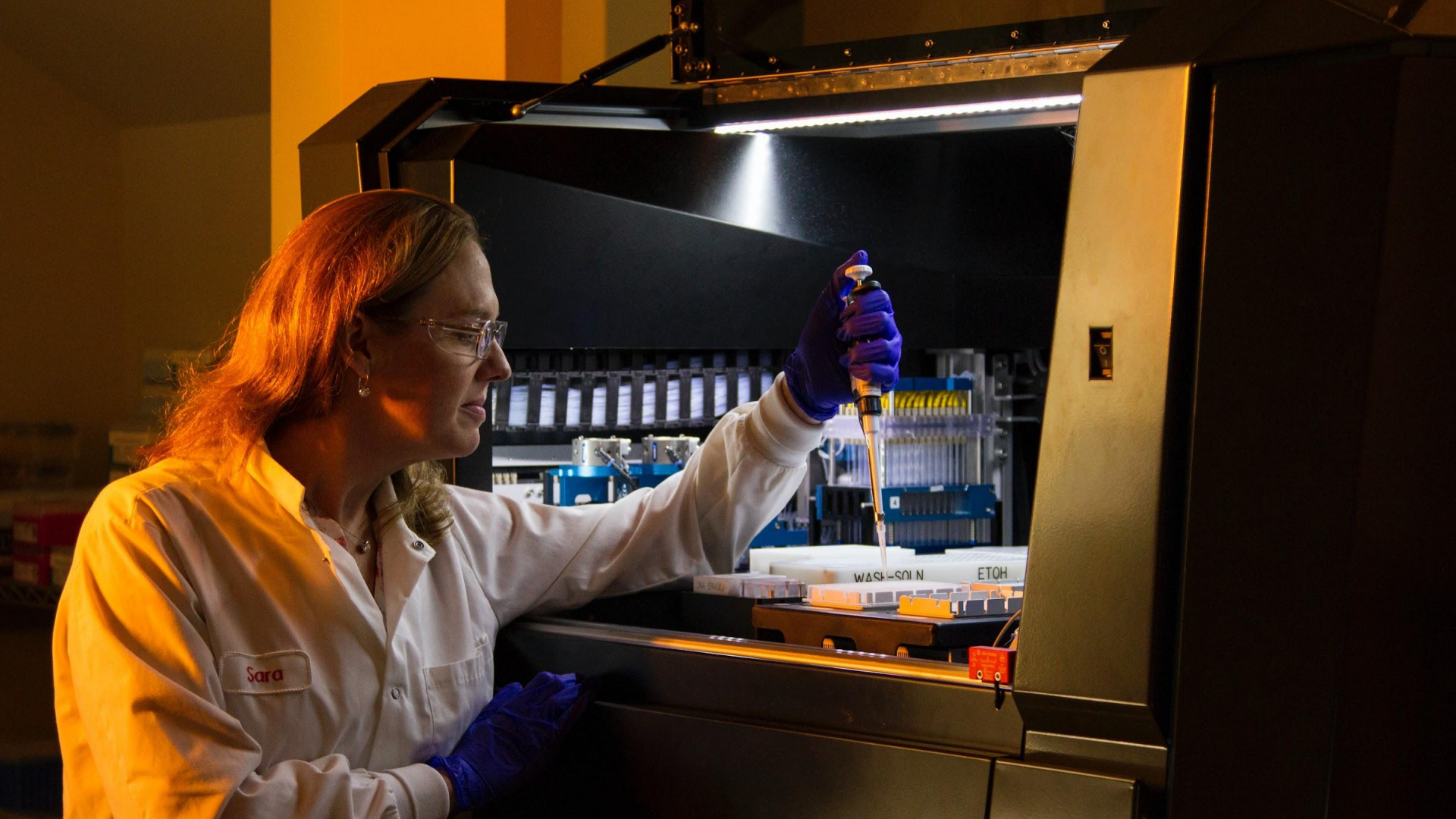
7-9 July 2025

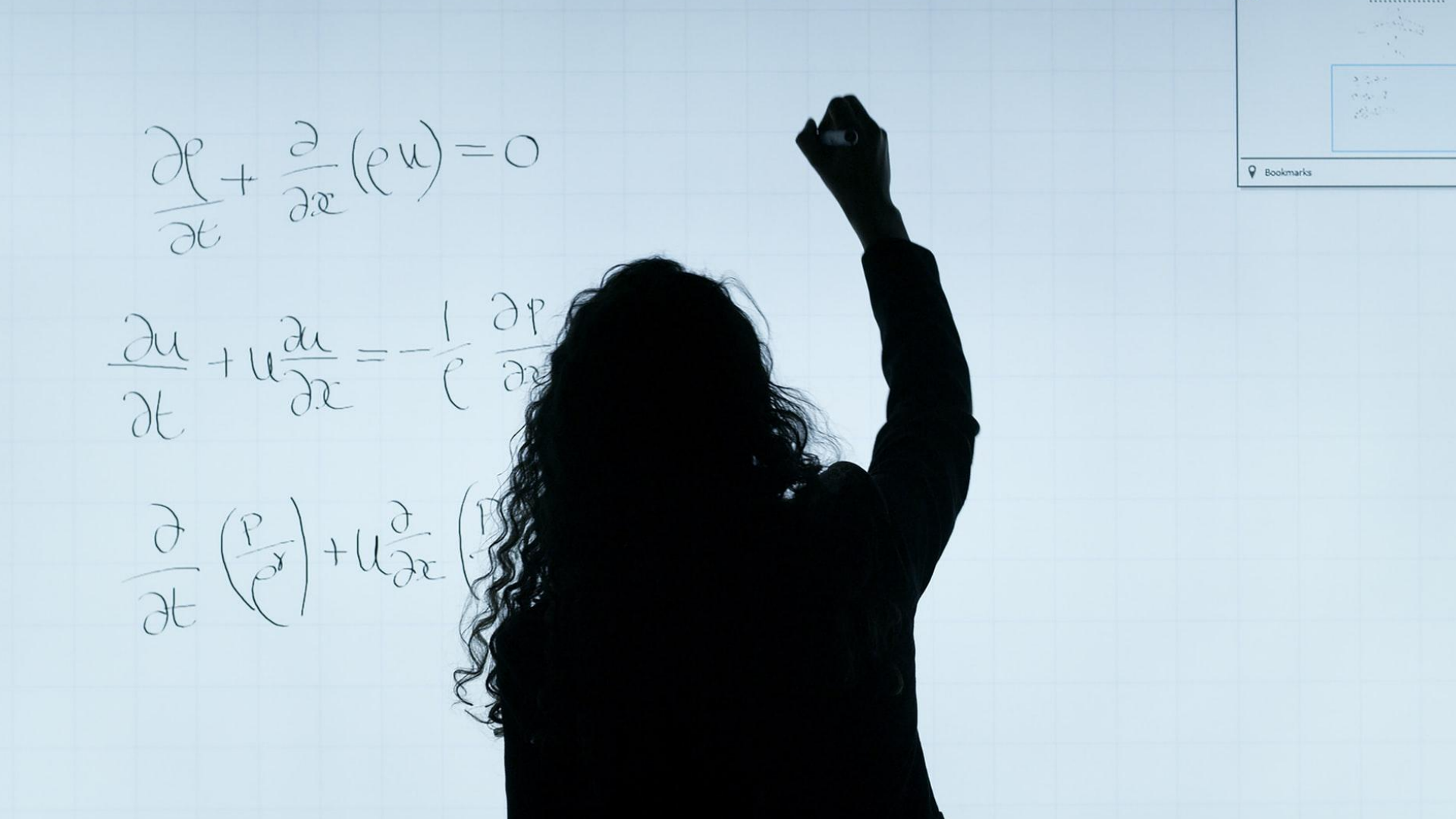
Flanders
State of the Art

RICHTING MORGEN
MET WETENSCHAP, TECHNOLOGIE, CREATIVITEIT & INNOVATIE

DOCTORAL SCHOOLS
GHENT UNIVERSITY




Sara
ETOH


Bookmarks

Scientists need to program for a variety of purposes

- Simulation (differential equations, IBMs etc)
- (Numerical) mathematics
- Reading, curation and plotting of data
- Statistics and machine learning
- Text processing
- Image analysis
- Using the shell to create a data-processing pipeline
- Organising files
- Creating figures
- Hosting a website
- ...

The programming tasks of a scientist are diverse and complex!

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

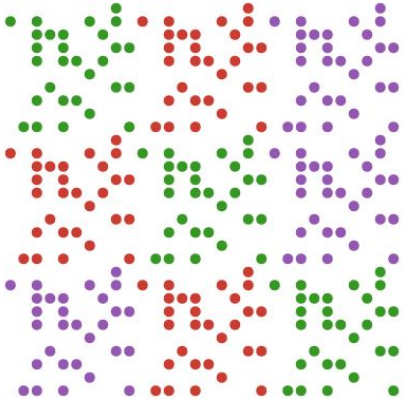
Jeff Bezanson, Stefan Karpinski,
Viral B. Shah, Alan Edelman

Who are you again?

Michiel

Professor at Faculty of BioScience Engineering, teaching modelling and simulation. Developing machine learning and optimization methods for plant sciences and synthetic biology. Contributed to several Julia packages.

Kronecker.jl



Experience with:

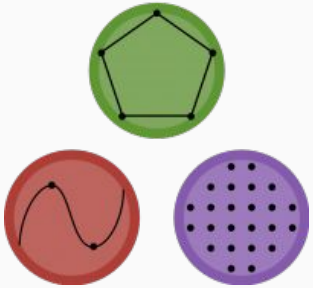
flux

DataFrames.jl

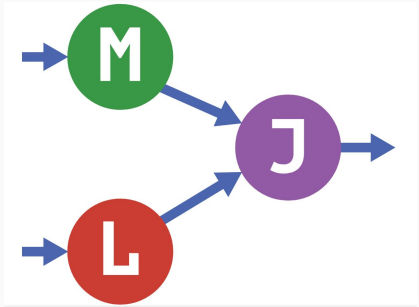
DifferentialEquations.jl
+ DiffEqFlux.jl

Pluto.jl

Experience with:



JuliaOpt.jl



MLJ.jl

DrWatson

The perfect sidekick to your scientific inquiries

Main focus at the moment:
Porting legacy Python code into the future

Day 1			
Time	Description	Learning outcomes	Method
9:00	*presentation: introduction*	What is julia? What are the main strengths, Introduction of teachers, scope and planning of course	presentation
9:30	*presentation: basic concepts*	Walkthrough of basic syntax	presentation
10:00	*break*		
10:15	*exercises:* 01-basics	introduce basic programming concepts in julia	exercises
11:00	*intermezzo:* plotting	learn how to use plotting.jl	demo
11:15	*exercises:* 01-basics	apply/implement basic programming concepts	exercises
12:00	*lunch break*		
12:45	*intermezzo: IDEs*	Explanation of how to use Julia (IDE, REPL, notebook)	presentation
12:55	*presentation*: collections	introduce basic collections	presentation
13:00	*exercises:* 02-collections	apply/implement basic collections	exercises
13:45	*break*		
14:00	*exercises:* 02-collections	apply/implement basic collections	exercises
15:00	presentation: synthesis exercises	introduce exercises, understand minimal requirements	presentation
15:15	*break*		
15:30	*exercises:* 03-exercises	synthesise concepts of day 1	exercises
17:00	end of day 1		

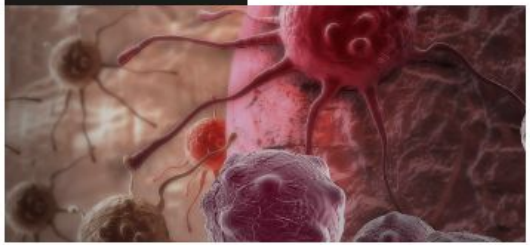
Day 2			
Time	**Description**	**Learning outcomes**	**Method**
9:00	*presentation:* overview day 2		presentation
9:05	*presentation: introduction types system*	learn the type system in julia	presentation
9:20	*exercises:* 01-types	apply the type system in julia	exercises
10:15	*break*		
10:30	*presentation: composite types*	what are composite types?	presentation
10:45	*exercises: 02-composite-types*	apply composite types	exercises
12:00	*optional: power of multiple dispatch + performance*	gain deeper insight into the julia type system	movie
12:30	*lunch break*		
13:00	*presentation:* introduction to macros	understand the power of macros	presentation
13:15	*presentation*: synthesis exercises	synthesise concepts of day 2	presentation
13:30	*synthesis exercises*	synthesise concepts of day 2	exercises
14:30	*break*		
14:15	*Intermezzo: useful tools in the julia landscape*	learn about the existence of optim.jl, diffeq, drwatson and others	demo
14:45	*synthesis exercises*	synthesise concepts of day 2	exercises
16:30	end of course		

Brief history of Julia

- 2009: project started by the founders at MIT
- 2012: launch of the language
- 2014: launch annual JuliaCon
- 2015: Julia Computing, Inc. (now JuliaHub) founded to support open-source Julia development
- 2018: v1.0 stable release 🎉
- 2019: James H. Wilkinson Prize for Numerical Software (awarded only every four years)
- 2021: major precompilation speed update v 1.6
- 2023: v1.9 native code caching = compiled code reuse
- 2024: v1.11 porting more low-level code to julia (Array, code parsing)
- 2025: Julia support in Google Collab

Julia for scientific computing

NATURE GENETICS Cancer Genomics



Modeling Cancer Evolution

UK cancer researchers use Julia to model tumor growth, informing interpretation of cancer genomes

ZIPLINE Drone Delivery

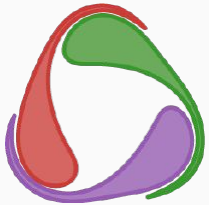
Emergency Medical Supplies by Drone

Zipline uses Julia for aircraft simulation to deliver life-saving emergency medical supplies via drone in Africa and worldwide

INVENIA Electrical Grid

Optimizing the Electrical Grid

Invenia Technical Computing is scaling up its energy intelligence system using Julia

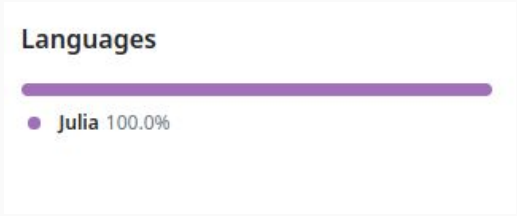


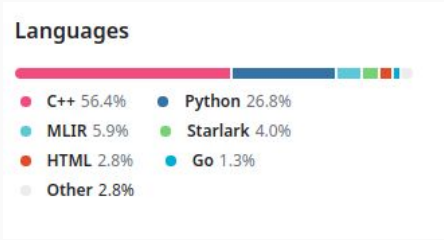
Julia for machine learning

Zygote

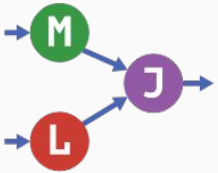
flux

Enzyme.jl

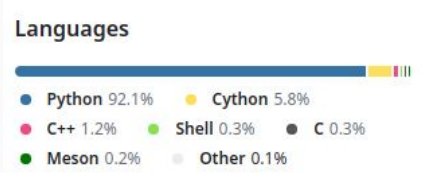




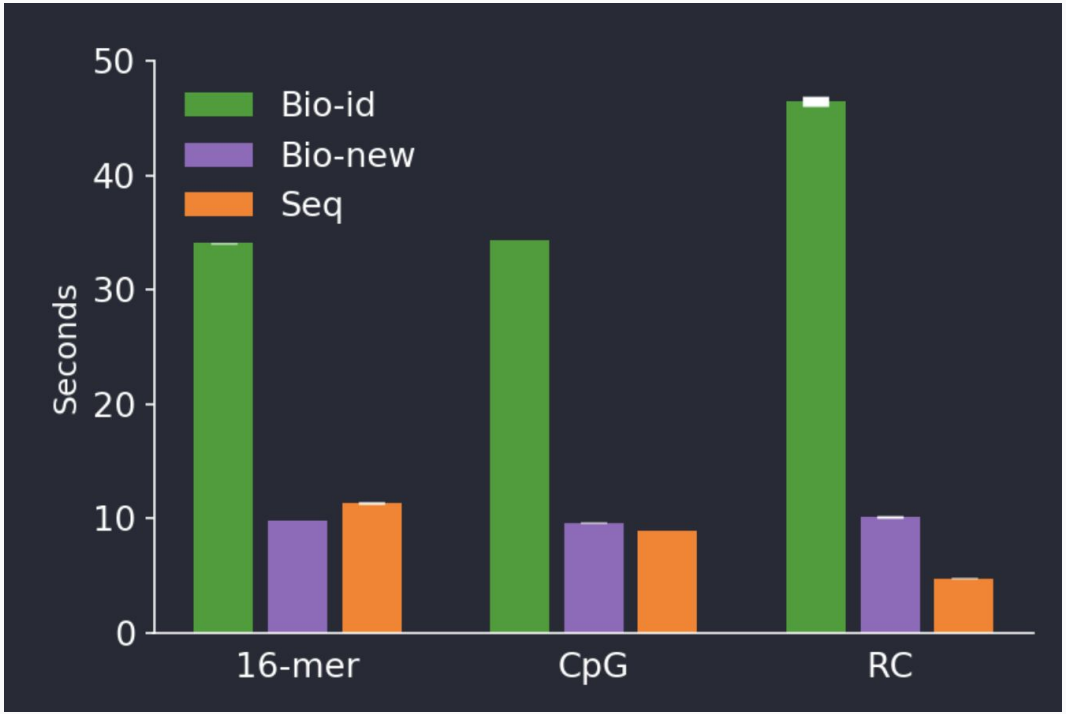
SciMLSensitivity.jl



scikit learn



Julia for bioinformatics



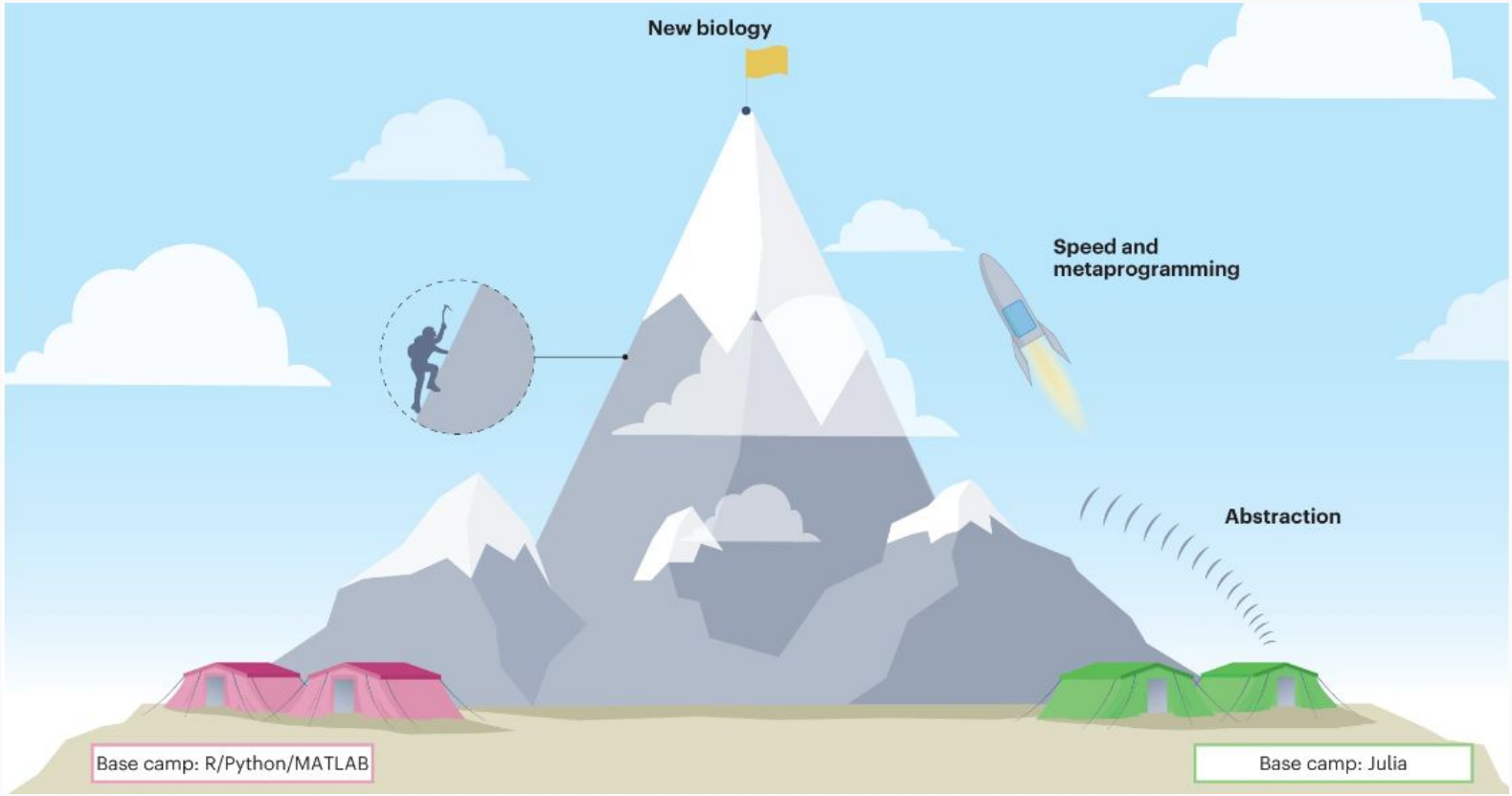
Julia performance is comparable to specialised languages for bioinformatics

https://biojulia.dev/posts/seq-lang/

The secret sauce of JuliaLang

Type-based dispatch allows for the generation of highly optimised and specialised methods (functions) for a given input combination.

Julia gives you superpowers!



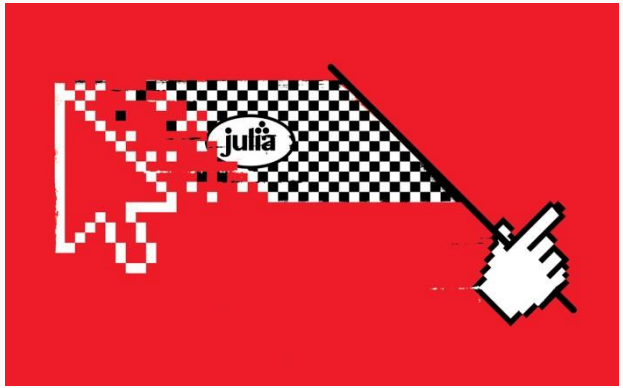
Roesch, E., Greener, J. G., MacLean, A. L., Nassar, H., Rackauckas, C., Holy, T. E., & Stumpf, M. P. H. (2021). Julia for Biologists. Retrieved from http://arxiv.org/abs/2109.09973

Three main strengths of Julia:

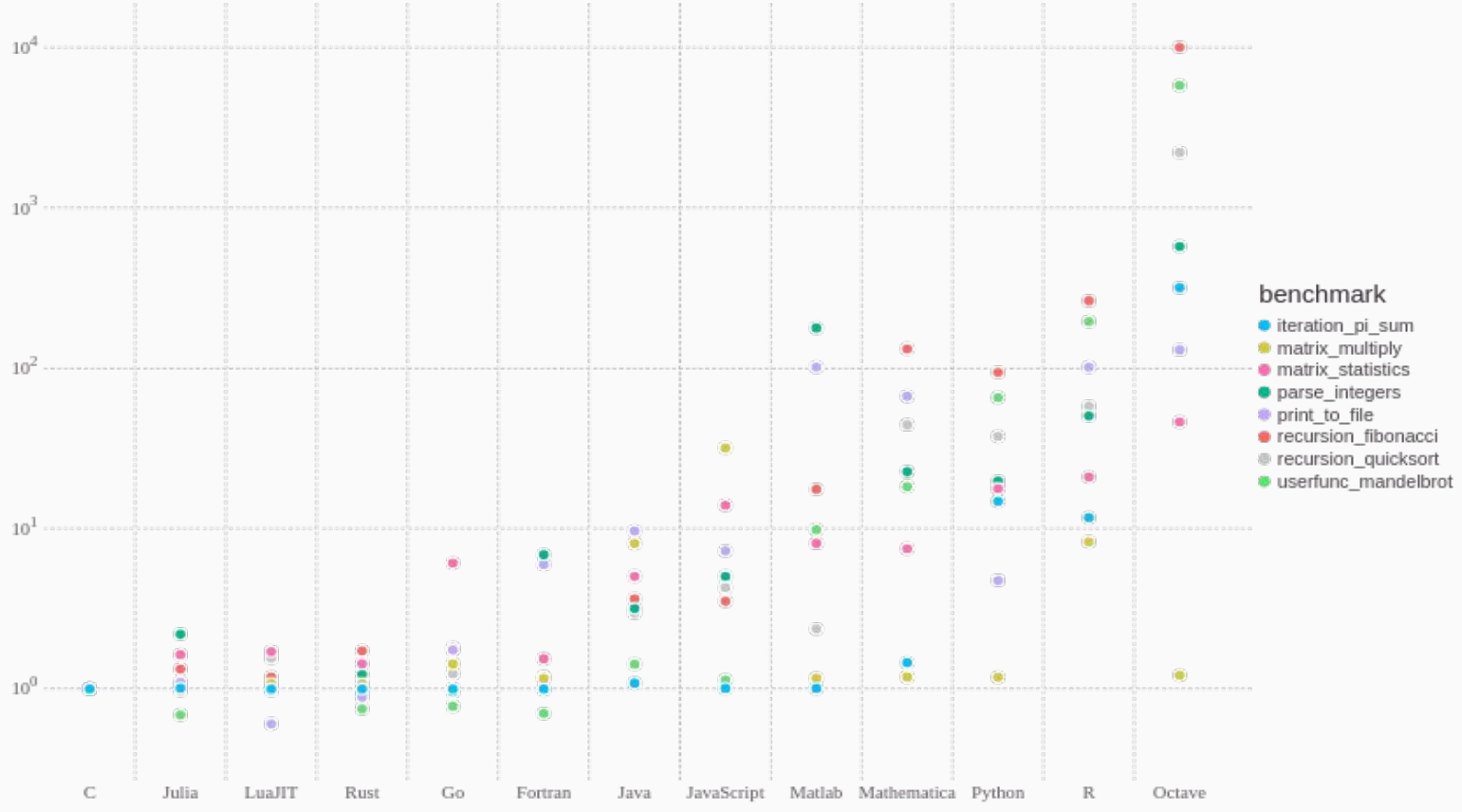
1. Speed
2. Abstraction
3. Metaprogramming

1. Speed

- Typically much (50x) faster than Python, R, …
- Close to the speed of C, if code is properly optimized (not necessarily easy)
- Can be made *faster* than C and the like, because one can use high-level tricks such as metaprogramming



Julia is fast!



Of course we still like Julia most

Comparison of gradient descent in Python and Julia

Pseudocode for the gradient descent method

given a starting point $\mathbf{x}$

repeat

1. $\Delta\mathbf{x} := -P\mathbf{x} - \mathbf{q}$
2. *Line search*. Choose $t > 0$.
3. *Update*. $\mathbf{x} := \mathbf{x} + t\Delta\mathbf{x}$.

until stopping criterion is reached.

Comparison: gradient descent in Python and Julia

Python

```python
import numpy as np

def gradient_descent(P, q, x0,
                     alpha=0.01, beta=0.8, maxiter=1000, eps=1e-5):
    x = np.copy(x0)
    nabla_f = lambda x : P.dot(x) + q
    Dx = -nabla_f(x)
    for iter in range(maxiter):
        x += alpha * Dx
        Dx = beta * Dx - (1-beta) * nabla_f(x)
        if np.linalg.norm(Dx) < eps:
            break
    return x

P = np.array([[10, -1, 0],
              [-1, 1, 0],
              [0, 0, 5]])

q = np.array([0, -10, 20])

x0 = np.zeros(3)

gradient_descent(P, q, x0)
```

Julia

```julia
using LinearAlgebra

function gradient_descent(P, q, x₀;
                          α=0.01, β=0.8, maxiter=1000, ε=1e-5)
    x = copy(x₀)
    ∇f = x -> P * x .+ q
    Δx = -∇f(x)
    for iter in 1:maxiter
        x .+= α .* Δx
        Δx .=  β .* Δx .- (1-β) .* ∇f(x)
        norm(Δx) < ε && break
    end
    return x
end

P = [10 -1 0;
    -1 1 0;
    0 0 5]

q = [0, -10, 20]

x₀ = zeros(3)

gradient_descent(P, q, x₀)
```

Comparison: gradient descent in Python and Julia

Python

```
In [2]: %timeit gradient_descent(P, q, x0)
3.33 ms ± 72.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Julia

```
julia> @btime gradient_descent($P, $q, $x₀)
  38.292 µs (4008 allocations: 156.56 KiB)
3-element Vector{Float64}:
  1.110998451882383
 11.110084811197943
 -3.999999999999998
```

2. Abstraction by type system and dispatch

Julia allows users to construct their own types within the native type system.

Multiple dispatch allows to specify certain functions for certain combinations of types.

This is **extremely** powerful!

Composability

Code of different sources can easily be combined into novel ways:

- Zygote.jl + Colors.jl = differentiation on colors
- DifferentialEquations.jl + Measurements.jl = ODEs with error bars
- DifferentialEquations.jl + Flux.jl = neural ODEs

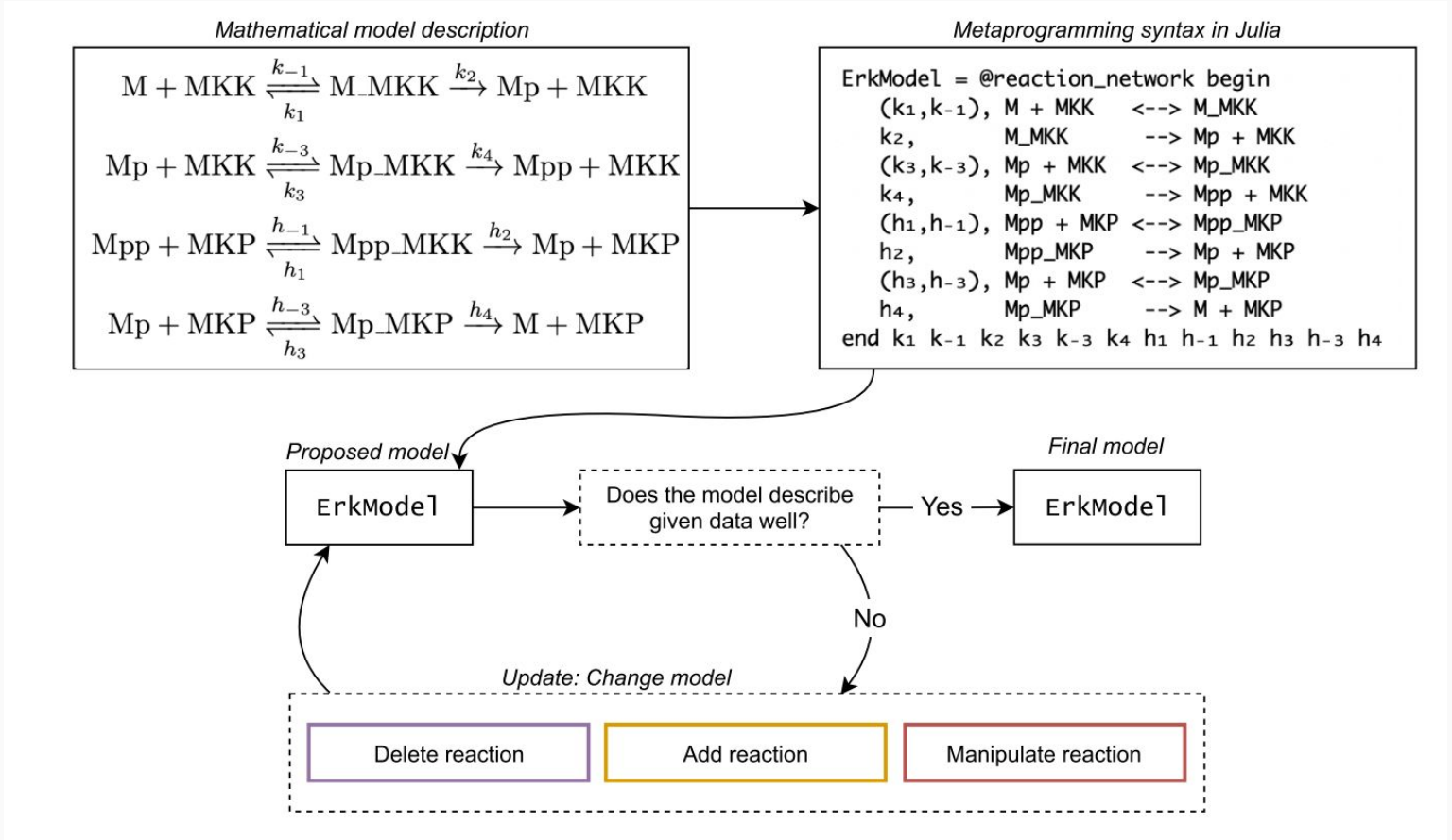
3. Metaprogramming

Metaprogramming is using a programming language to write new code.

Julia is **homoiconic**: Julia code can be represented as a Julia data structure (similar to languages such as LISP)

More about this tomorrow!

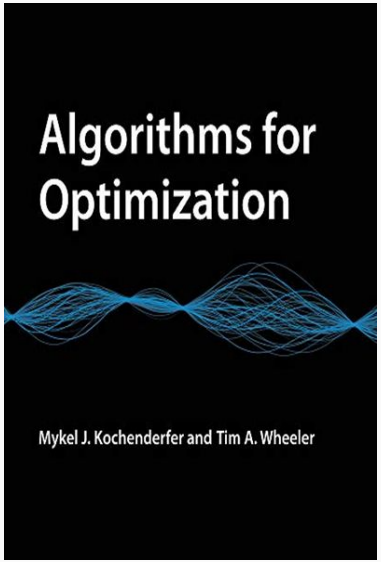
Domain-specific languages (Catalyst.jl)



Extra: code readable by scientists

Bayesian model:

```
@model function measurement(x)
    n = length(x)
    σ²_μ ~ InverseGamma(10, 0.1)
    μ ~ Normal(0.0, √(σ²_μ))
    σ² = Vector(undef, n)
    for i in 1:n
        σ²[i] ~ InverseGamma(10, 0.1)
        x[i] ~ Normal(μ, √(σ²[i]))
    end
end
```




```
function Base.rand(ϕ::Factor)
    tot, p, w = 0.0, rand(), sum(values(ϕ.table))
    for (a,v) in ϕ.table
        tot += v/w
        if tot >= p
            return a
        end
    end
    return NamedTuple()
end

function Base.rand(bn::BayesianNetwork)
    a = NamedTuple()
    for i in topological_sort(bn.graph)
        name, ϕ = bn.vars[i].name, bn.factors[i]
        value = rand(condition(ϕ, a))[name]
        a = merge(a, namedtuple(name)(value))
    end
    return a
end
```

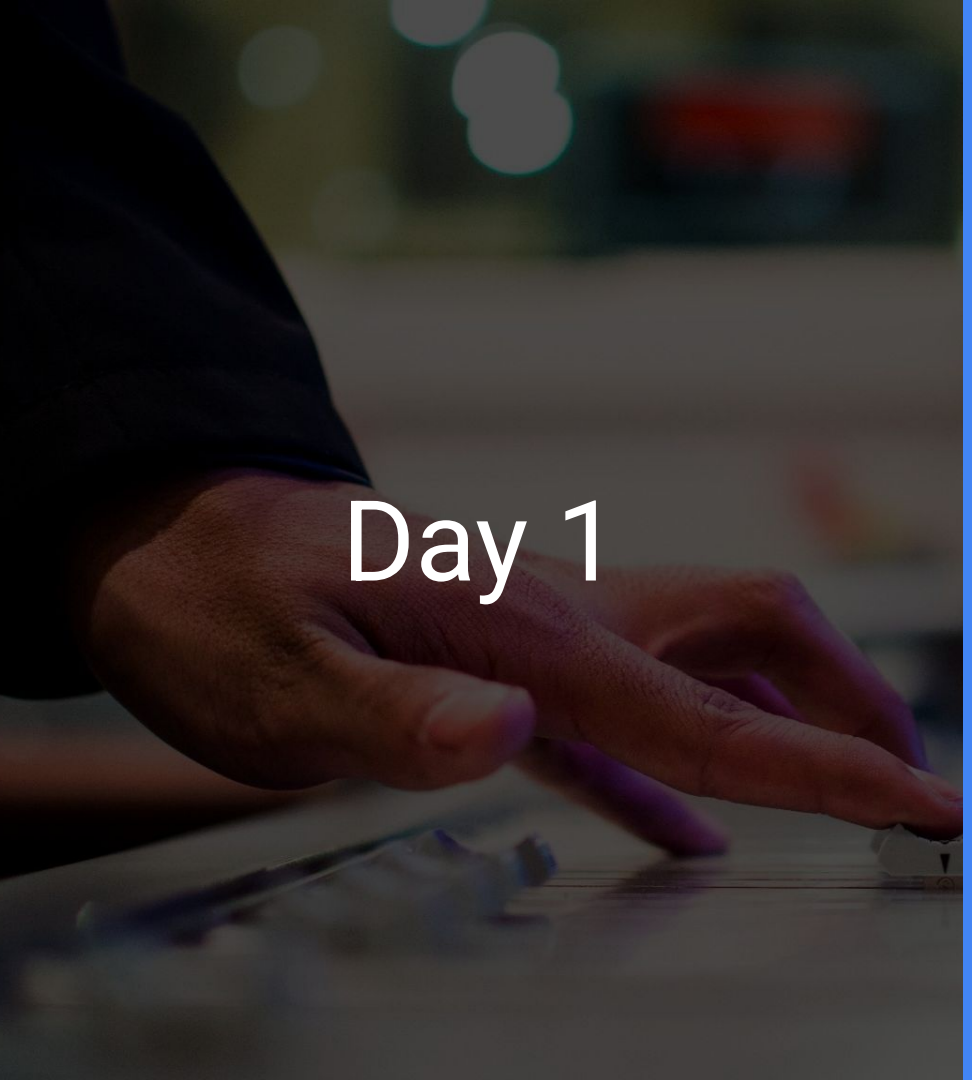
All algorithms given in pure julia code

Warning: strawman!

In this course, we will often compare features of Julia with other languages (often Python) to illustrate the difference.

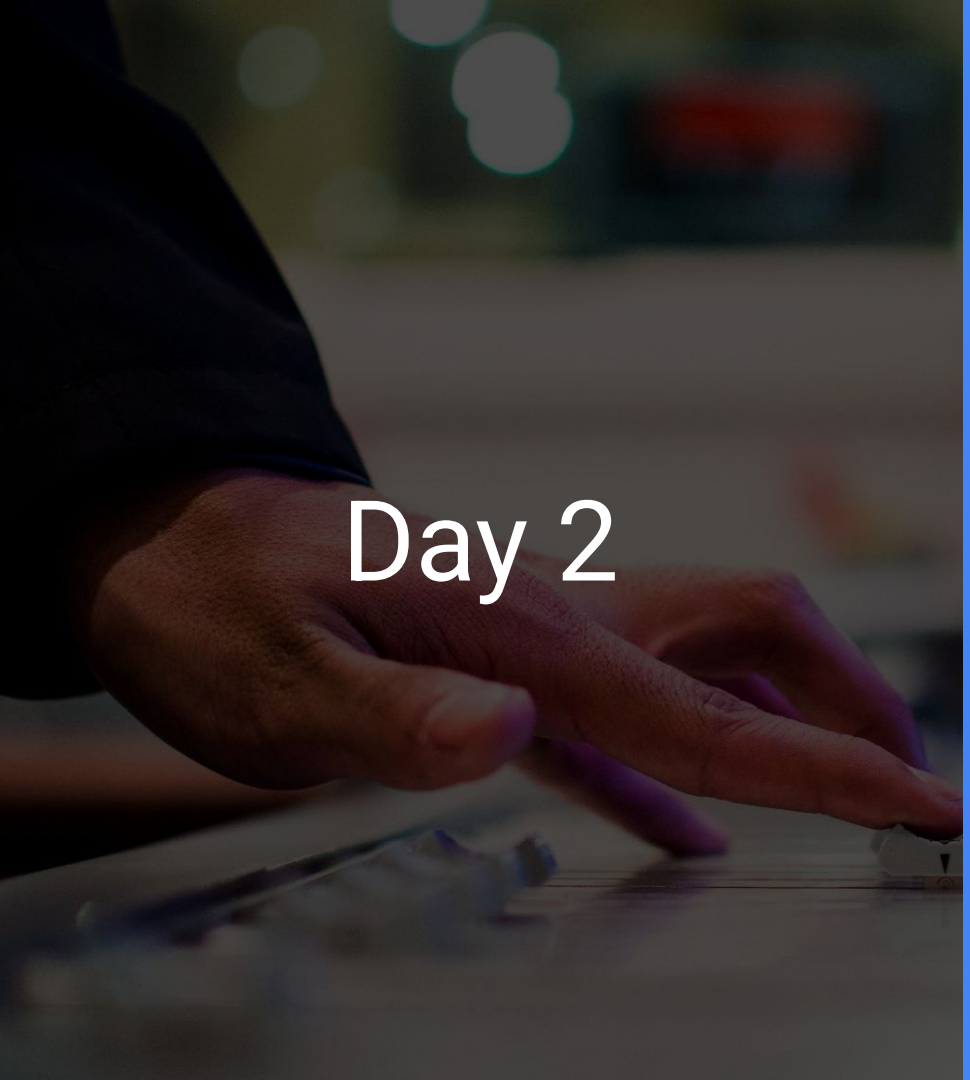
Though we love Julia, we do **not** believe it is inherently superior to other languages such as Python, R, Matlab, C, Mathematica… Each language has its own strengths, best use cases and user preferences.

We will often compare with Python, because it is popular, **not** because it is a bad language.



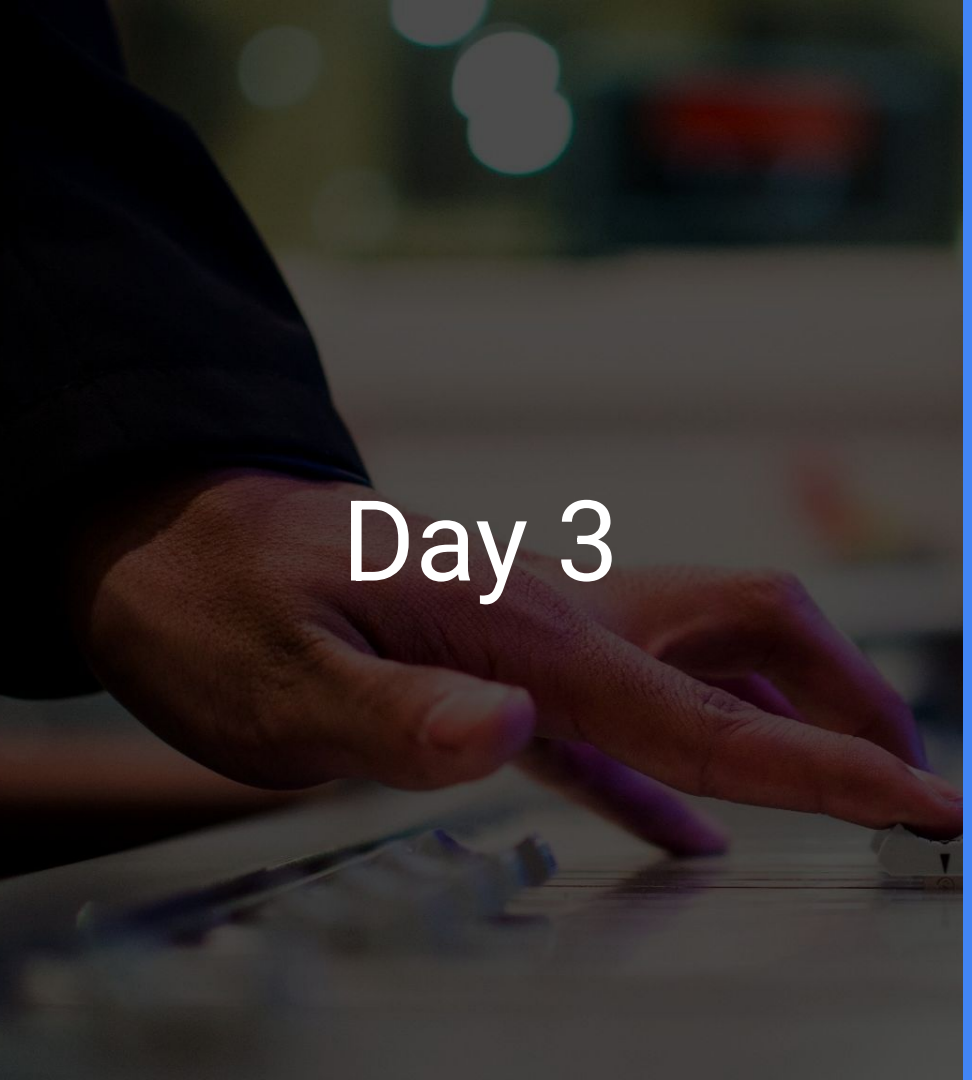
Day 1

- Go over the fundamentals of the Julia language
- Plotting
- In-depth exploration of collections such as arrays
- Project to practice writing scientific code and data visualisation



Day 2

- Detailed coverage of the type system and multiple dispatch
- Metaprogramming
- Large synthesis exercise
- Overview of some cool Julia projects



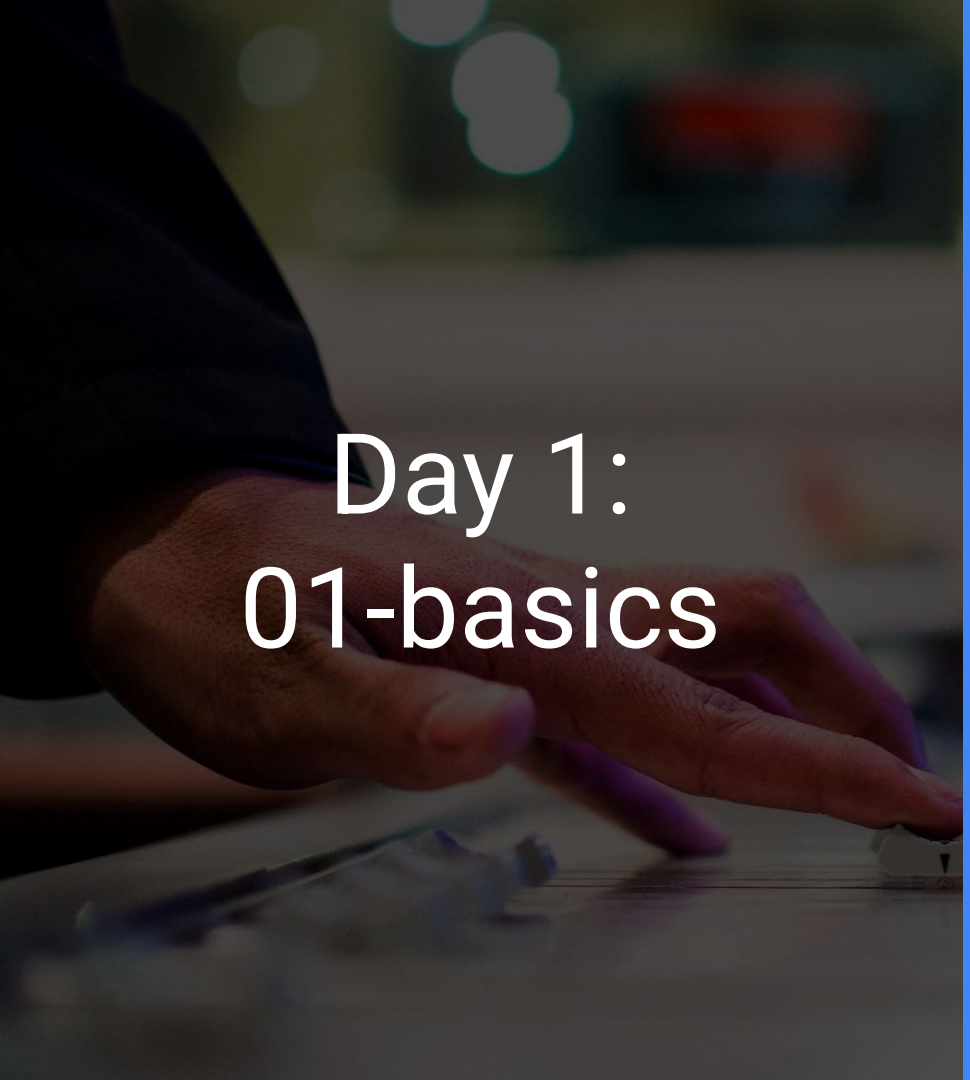
Day 3

- DrWatson overview
- Projects
- Optional: Git and Github

Requirement for passing the doctoral schools

- Fill in standard survey at the end of the course.
- Mail Michiel, Daan and Bram the HTMLs of your completed notebooks by Monday 14th of July.

Slides and other course material will be made available at the end of the course.



Day 1: 01-basics

- Basics
- Logical statements
- Control flow
- Looping
- Functions
- (intermezzo on) types
- Multiple dispatch
- (small teaser on) macros
- plotting

Basics, logical statements & control flow

Let’s start with some basic mathematical operations!

```
3
  1 + 2        # adding integers

3.0
  1.0 + 2.0    # adding floats

3.0
  1 + 2.0      # adding a float to an integer...

0.5
  2 / 4        # standard division
```

```
0
  div(2, 4)    # Computes 2/4 truncated to an integer

0
  2 ÷ 4        # looks nices but exactly the same!

1
  7 % 3        # get the remainder of the integer division

0.2
  35 \ 7       # inverse division

1//3
  1 // 3       # fractions, gives result as a rational
```

Basics, logical statements & control flow

```
2.0 + 3.0im
```

```
2.0 + 3.0im  # complex numbers
```

```
'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

```
'c'          # characters (unicode)
```

```
:symbol
```

```
:symbol      # symbols, mostly used for macros
```

```
:ζ
```

```
:ζ           # any LaTeX symbol
```

```
:🐛
```

```
:🐛          # or Unicode emoji
```

```
mystery = "life, the universe and everything"
```

```
mystery = "life, the universe and everything"
```

Try to use include some LaTeX or emojis in the exercises!

For instance type \alpha and <TAB> to get the greek lowercase alpha

Basics, logical statements & control flow

Julia will print the result of every statement by default (like Matlab)

Suppress with ;

```
a = 10;  # not printed...
```

```
10
```

```
a  # ...but still defined
```

Basics, logical statements & control flow

Julia uses `true` and `false` for Boolean variables.

```
false
```

```
2 == 1
```

```
true
```

```
2 != 1
```

```
true
```

```
2 >= 2  # or 2 ≥ 2  (\ge<TAB>)
```

```
true
```

```
# Comparisons can be chained
1 < 2 < 3
```

The Boolean logic operators:

- `&&` (AND)
- `||` (OR)
- `⊻` (XOR, exclusive or)

```
false
```

```
true && false
```

```
false
```

```
false || false
```

```
true
```

```
true ⊻ false
```

Basics, logical statements & control flow

if, else, elseif-statement is instrumental to any programming language!

Note: control flow is ended with an end statement

Note #2: tabs are only for clarity !

```
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
if 4 > 3
  'A'
elseif 3 > 4
  'B'
else
  'C'
end
```

Julia allows for a very condensed if/else condition:

y = condition ? valueiftrue : valueiffalse

Looping, functions, types and multiple dispatch

Repeated evaluation: `for` loops and `while`

Statement ends with an `end`!

(you can use `in`, `=` and `∈`)

```
for i in [1, 2, 3]
  println(i)
end
```

```
while false
    # do something
end
```

Functions are defined with the `function` keyword, a function name and round brackets.

Statement ends with an `end`!

Note: `return` is not explicitly needed

```
square (generic function with 1 method)
```

```
function square(x)
  result = x * x
  return result
end
```

Looping, functions, types and multiple dispatch

All Julia objects have a type!

Code is specialised for a specific type.

Useful in combination with multiple dispatch = functions work differently depending on the input types

Want to know the type of an instance? Use typeof!

```
Float64
```

```
typeof(3.)
```

More details: tomorrow morning!

```
translate (generic function with 3 methods)
```

```
begin
    translate(xy::Array) = xy .+ [1 3]
    translate(s::String) = Markdown.parse("[Ctrl + click here](https://translate.google.com/?sl=en&tl=nl&text=$(s)&op=translate)")
    translate(anything::Any) = "I'm lost in translation..."
end
```

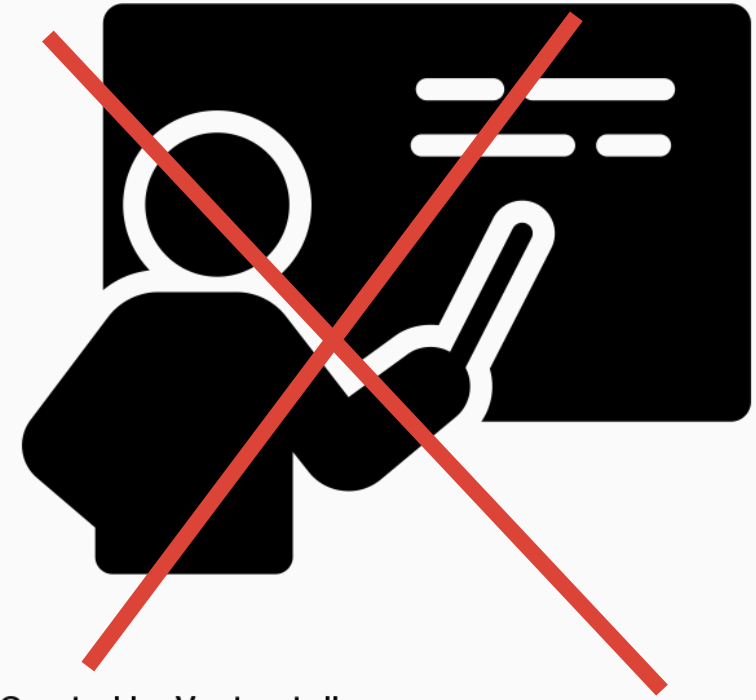
```
"I'm lost in translation..."
```

```
translate(1.0)
```

```
1×2 Array{Float64,2}:
 1.0  3.0
```

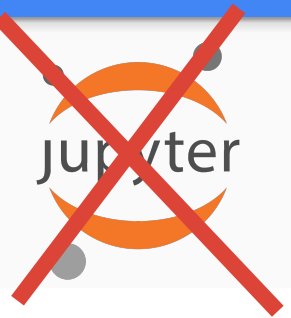
```
translate([0.0 0.0])
```

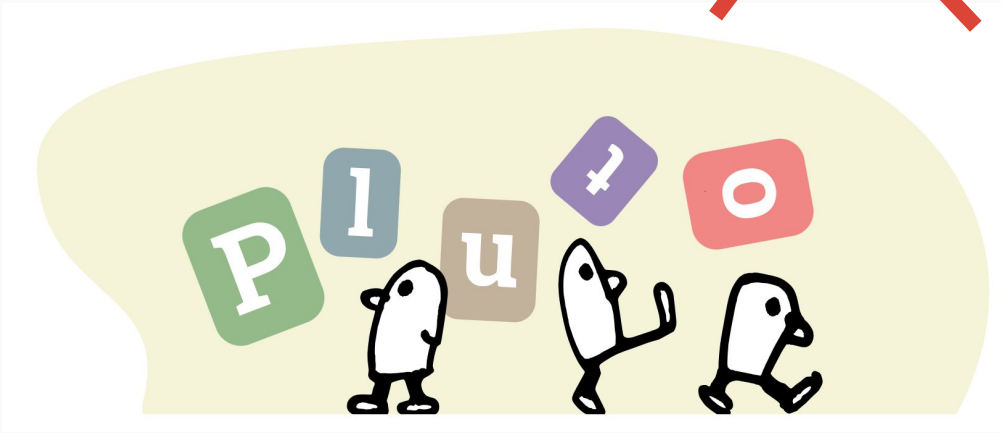
Course setup



Created by Vectorstall
from Noun Project

Guided exercises in



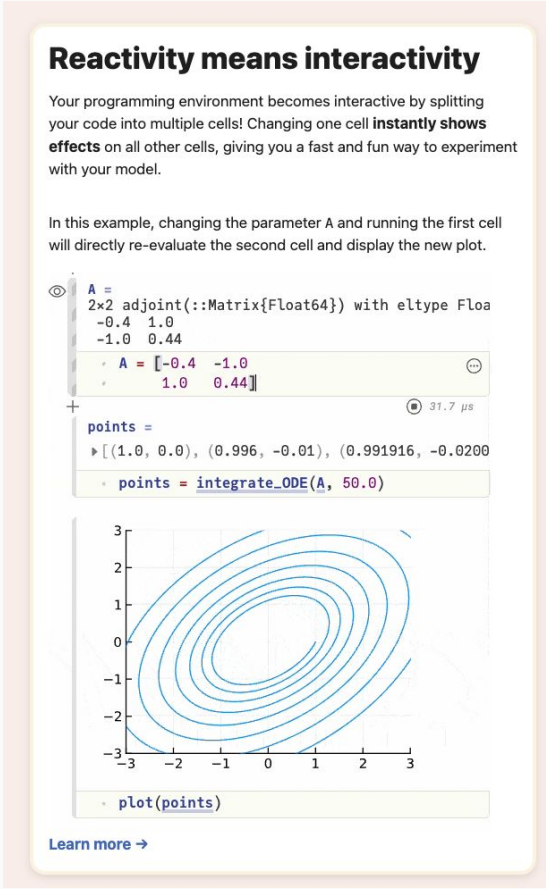


You can work together and ask us as many questions as you wish!

Pluto.jl: Reactive, Literate, and Reproducible Julia Notebooks

Pluto.jl is a unique interactive Julia environment

- Unlike traditional notebooks (like Jupyter), It's **reactive** (unlike Jupyter): changing any code automatically re-evaluates dependent cells.
- This ensures **always-in-sync output**, eliminating stale results and greatly improving **reproducibility**.
- Limitations: only one statement per line and needing unique variable names.
- Written in pure Julia, equipped with its own package manager.

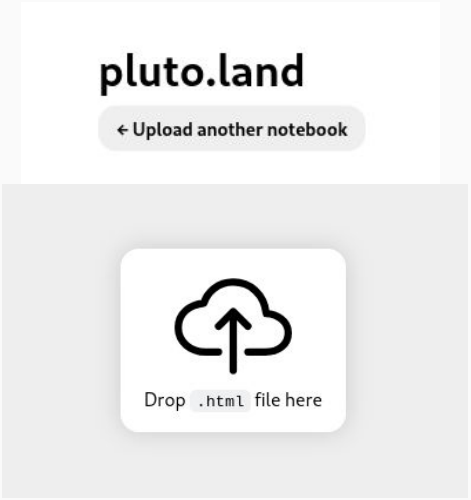


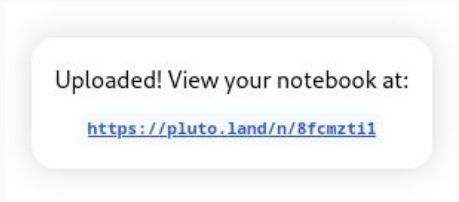
Pluto.jl: Sharing notebooks with friends and family

Pluto notebooks can be shared online.

https://pluto.land/

Could be useful if you are collaborating with you neighbours during
the doctoral schools

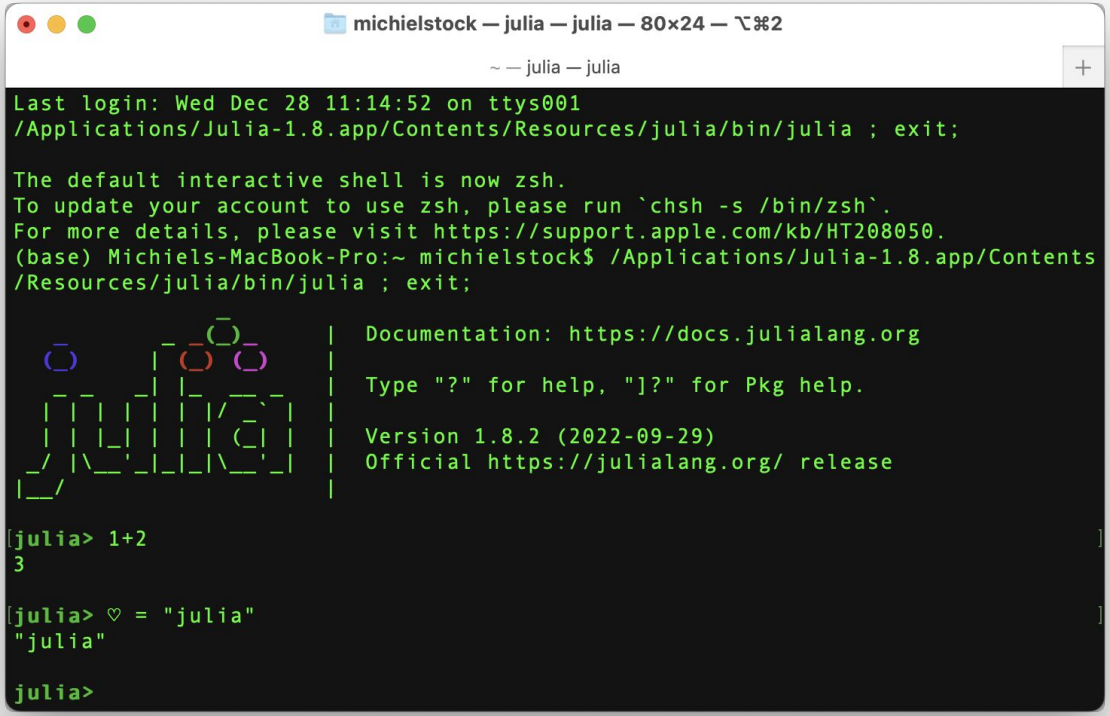




Lunchbreak

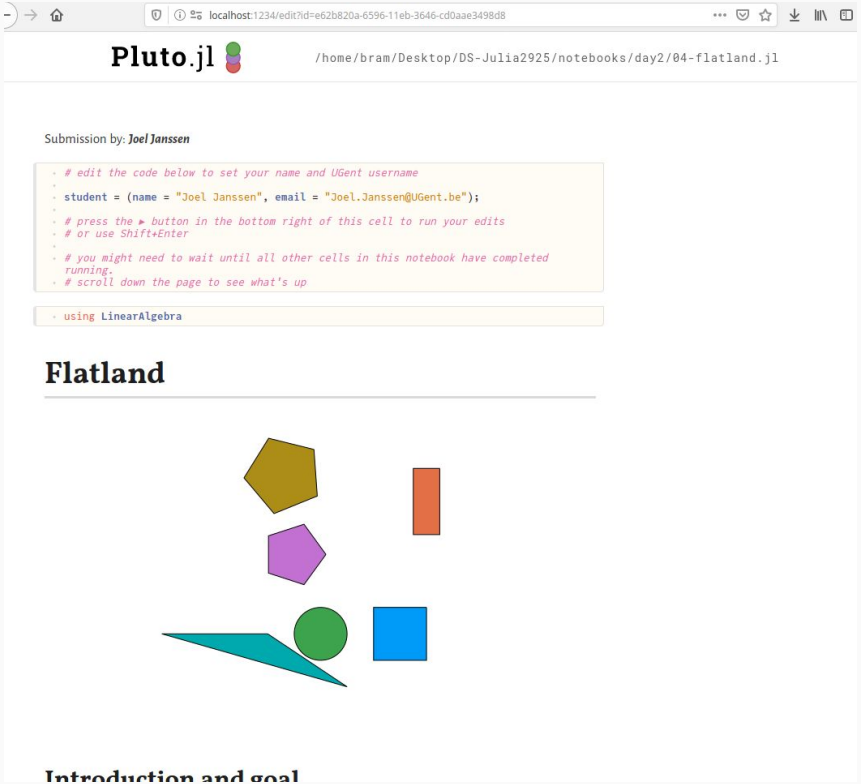
What tools do I use to code Julia?

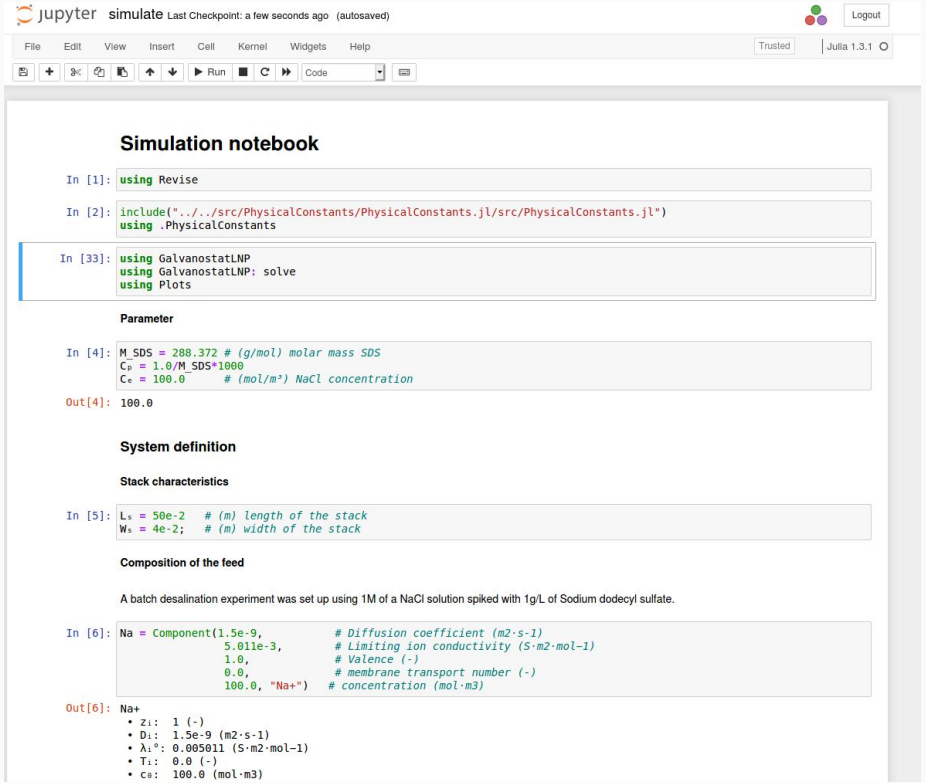
REPL: Quick calculations



What tools do I use to code Julia?

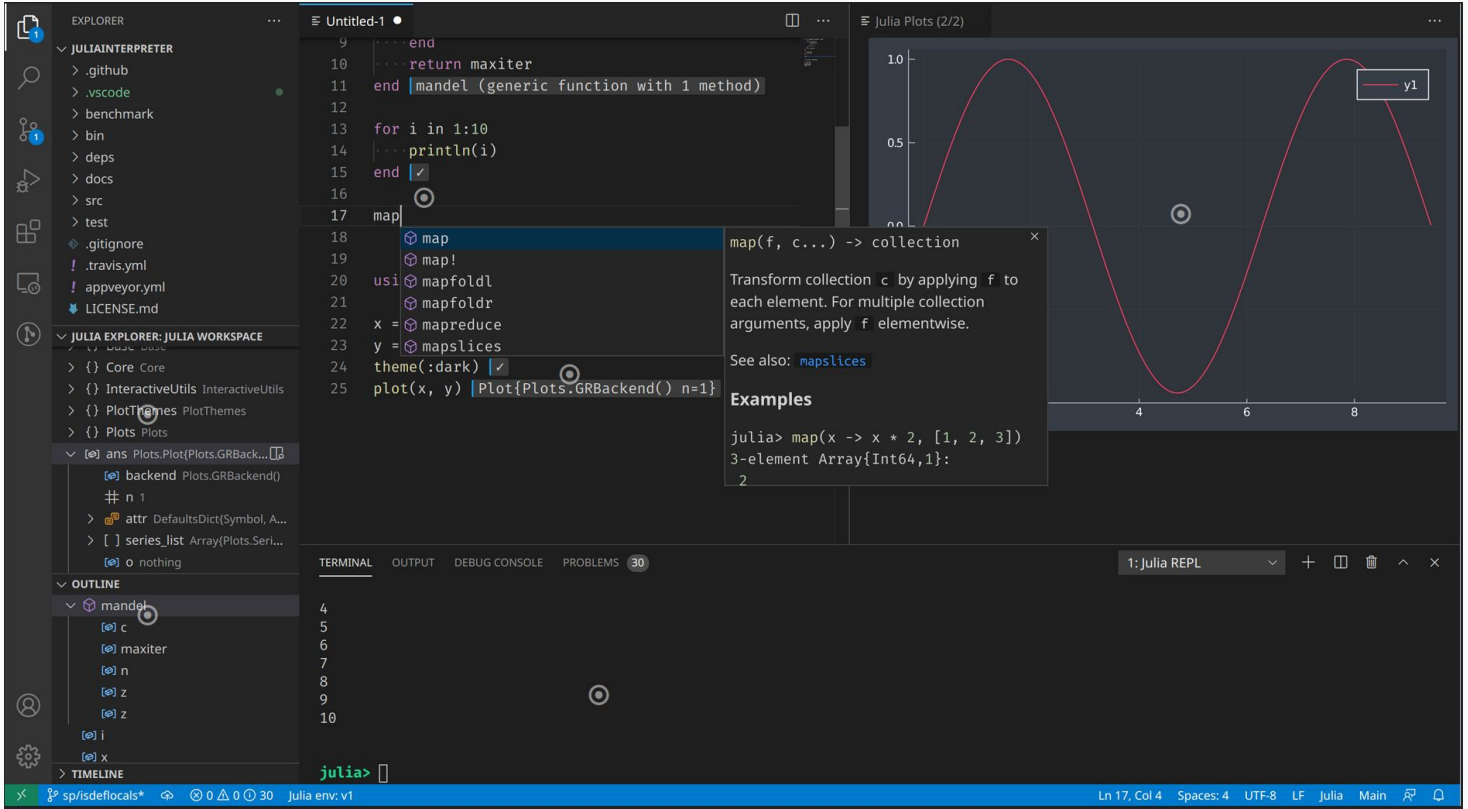
Notebooks: teaching, workshops, reporting





What tools do I use to code Julia?

VSCode IDE: workhorse, code development



Day 1: 02-collections

- Arrays
- Indexing and slicing
- Initialisation
- Comprehensions
- Matrices
- Concatenation
- Element-wise operation
- Higher-dimensional arrays
- Other collections

Collections?

A collection or container is a type that **collects** a group of data items.

Each collection type has its own properties (and use cases), e.g.,

- Lists, Matrices and Arrays (variable size)
- Tuples (fixed size)
- Dictionaries (fast look up)
- ...

Arrays and matrices

Vectors are **one-dimensional Arrays**,

```
X = ▸ Int64[1, 3, -5, 7]
  X = [1, 3, -5, 7] # array of integers
```

and behave like a **list** in **Python**.

Vectors are initialized by separating elements by *commas*.

Matrices are **two-dimensional,**

```
P = 4×3 Array{Int64,2}:
 0   1   1
 2   3   5
 8  13  21
34  55  89
  P = [0 1 1; 2 3 5; 8 13 21; 34 55 89]
```

For matrices, elements in a row are separated by *spaces* and rows by *semicolons*

Arrays and matrices

Arrays and **matrices** can be indexed and sliced,

```
1
  X[1]
```

```
▸ Int64[3, -5, 7]
  X[2:end]
```

but there are a plethora of other **operations** to be explored.

- Concatenation
- Comprehensions
- Pushing and popping (arrays)
- ...

Element-wise operation and matrix multiplication

Matrix multiplication is the default,

```
E = 2×3 Array{Int64,2}:
     2  4  3
     3  1  5
```

```
E = [2 4 3; 3 1 5]
```

```
R = 3×2 Array{Int64,2}:
     3  10
     4   1
     7   1
```

```
R = [ 3 10; 4 1 ;7 1]
```

```
2×2 Array{Int64,2}:
 43  27
 48  36
```

```
E * R
```

Element-wise operation and matrix multiplication

The **dot-notation** is used for **element-wise** operation,

```
T = 2×3 Array{Int64,2}:
     10  10  10
     20  20  20
```

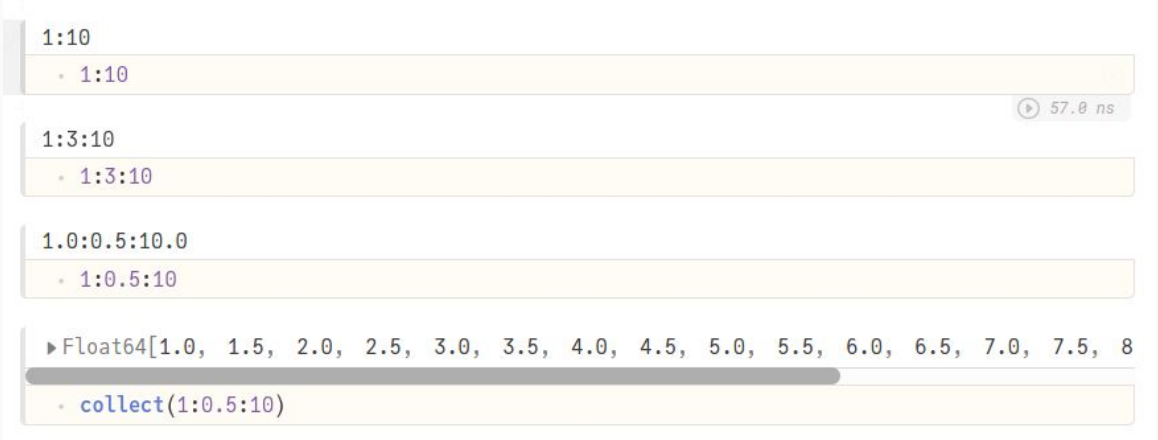
```
T = [10 10 10; 20 20 20]
```

```
2×3 Array{Int64,2}:
 100  100  100
 400  400  400
```

```
T.^2
```

Ranges and UnitRanges

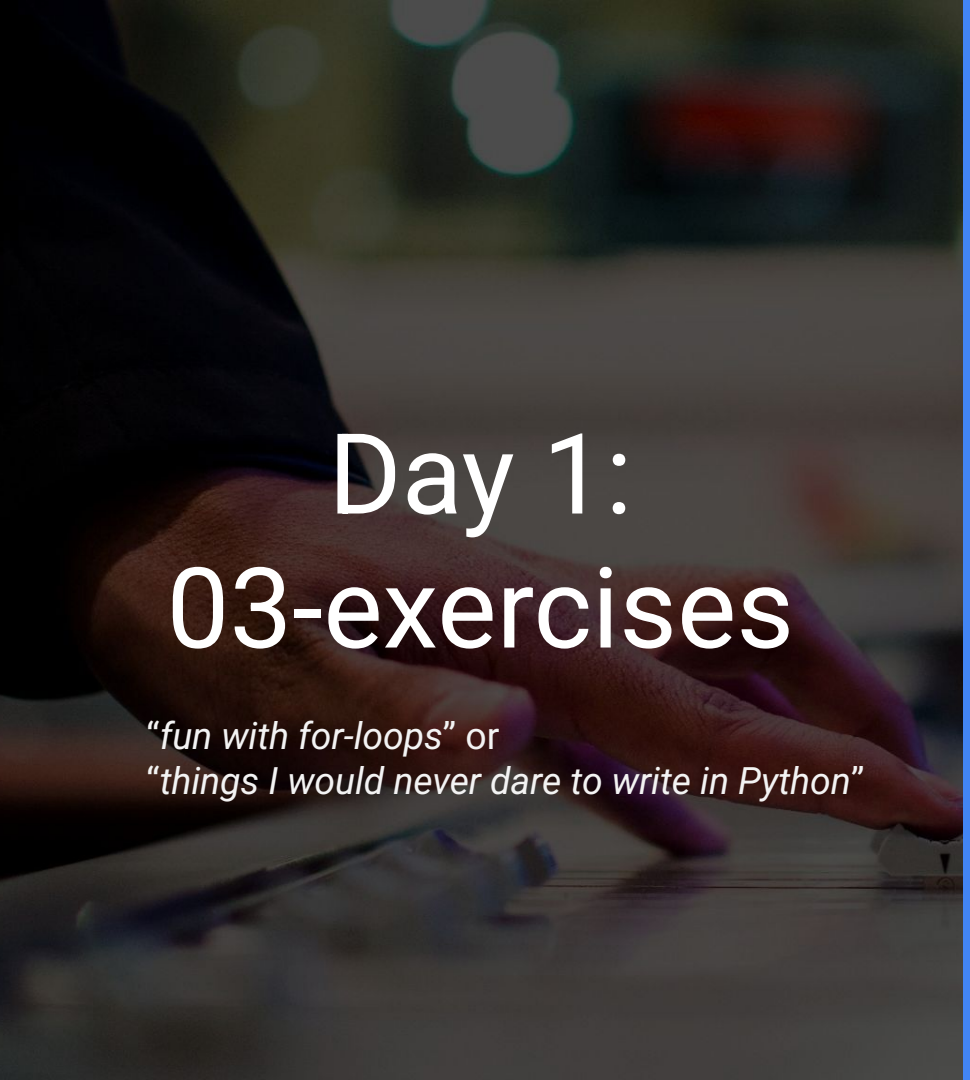
The **colon operator :** is used to define ranges which are **lazy arrays,**



You can also use the function `range`, which has slightly more functionality

these **Ranges** only store the first, the stepsize, and the last values and are only converted to arrays when it is necessary.

There is a lot more to discover. Go and explore *02-collections.jl* !



Day 1: 03-exercises

“fun with for-loops” or
“things I would never dare to write in Python”

images and cellular automata

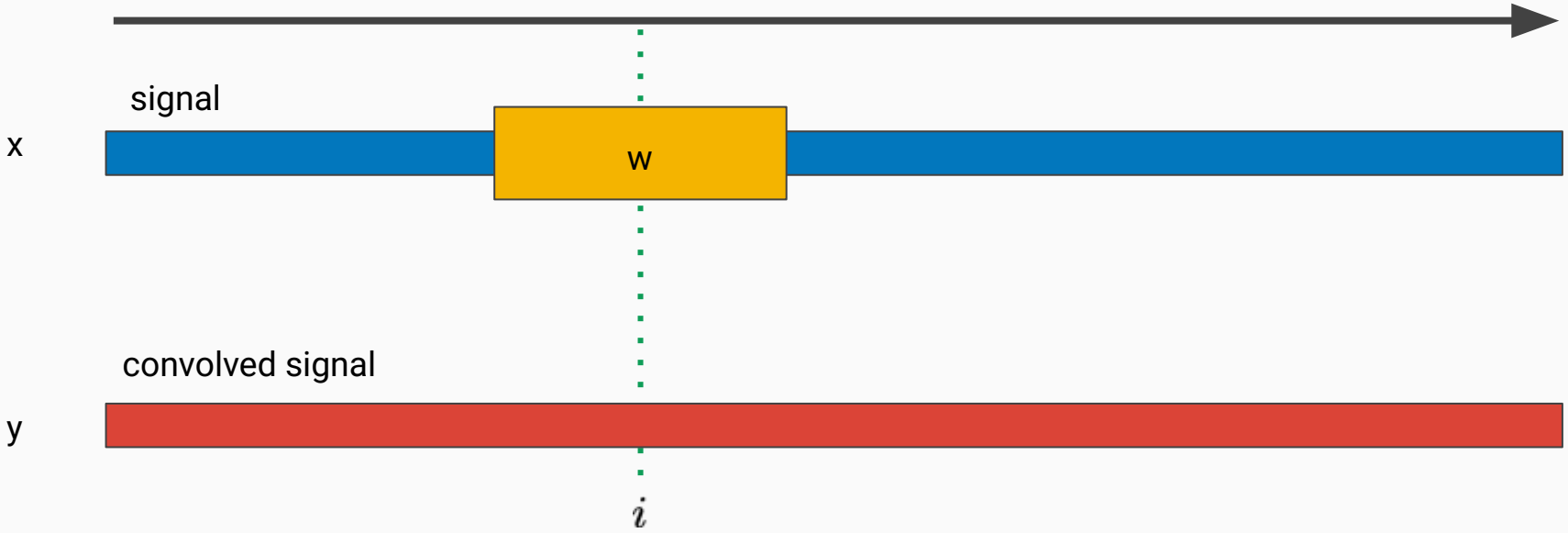
We will implement some interesting algorithms in Julia.

Goals:

- Code reuse;
- Implementing scientific code;
- Getting a good performance;
- Introduction to Colors.jl and Images.jl.

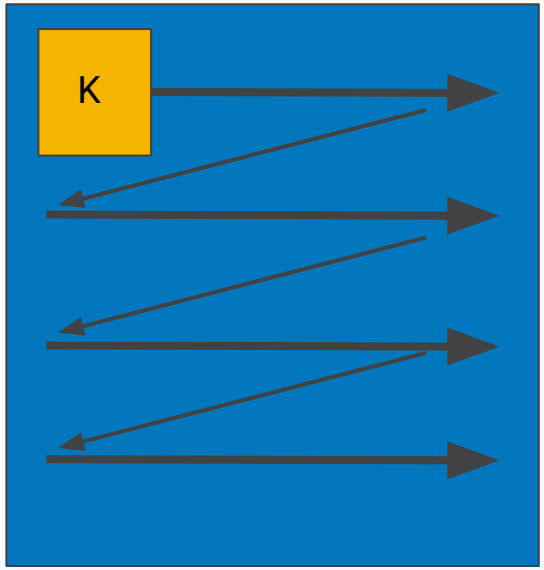
1-D convolutions

$$y_i = \sum_{k=-m}^{m} x_{i+k} w_{m+k+1}$$

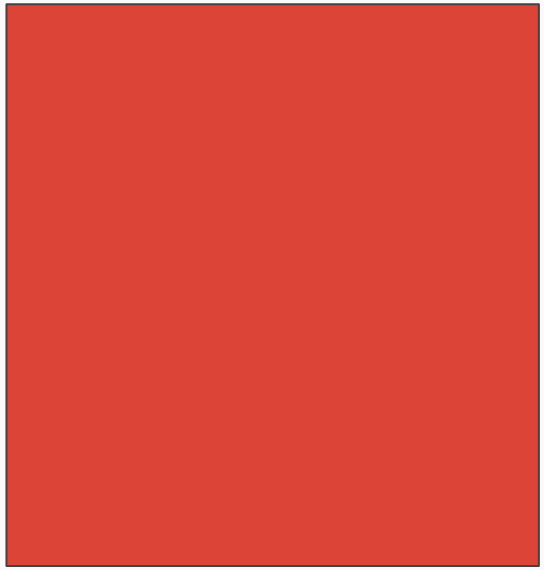


2-D convolutions

$$Y_{i,j} = \sum_{k=-m}^{m} \sum_{l=-m}^{m} X_{i+k,\, j+l}\, K_{m+(k+1),\, m+(l+1)}$$



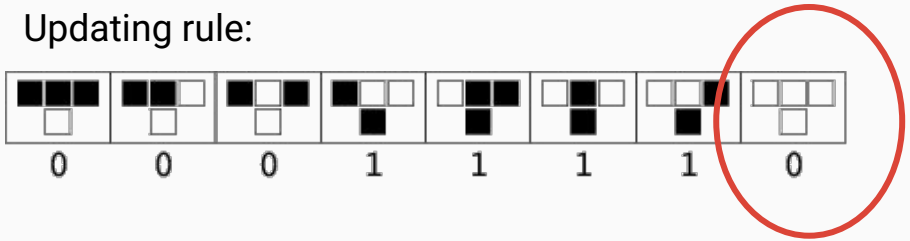
x



y

Elementary cellular automata (optional)

Updating rule:

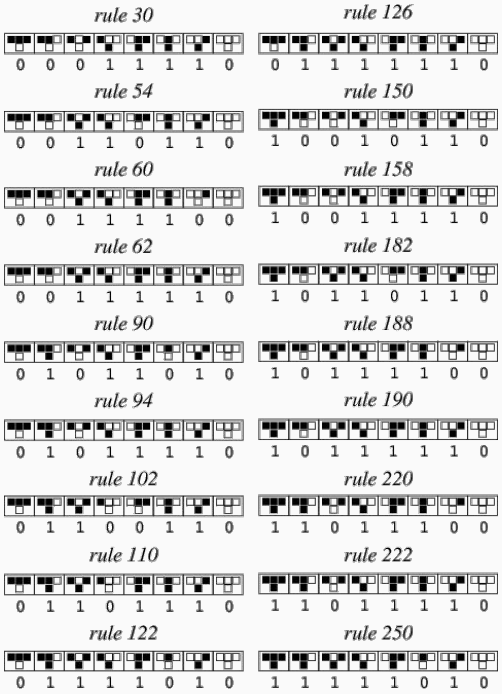


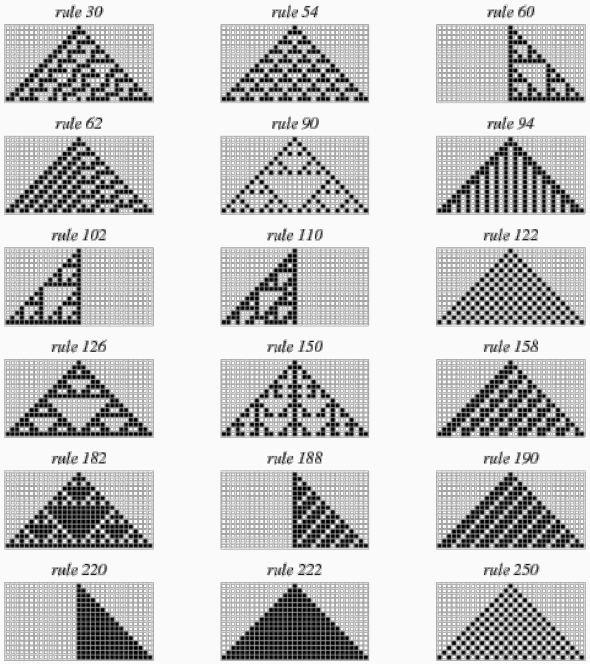
Next state depends on:
- State of cell
- State of the two neighbours

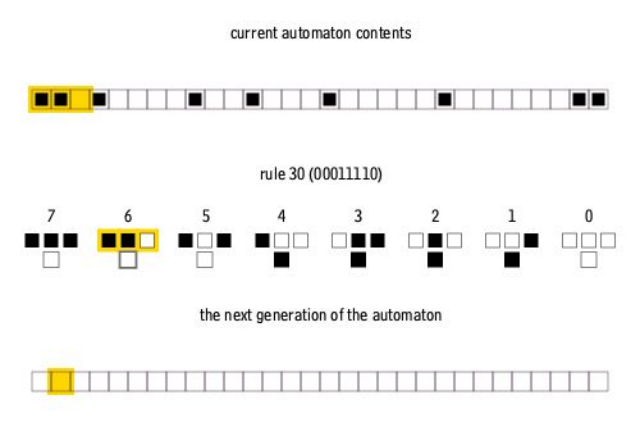
1. Take a bitstring: *001011***001011***001011* (cyclic boundary)
2. Split in overlapping groups of three: 1**0**0 0**0**1 0**1**0 1**0**1 0**1**1 1**1**0
3. Update according to the rules: 111010
4. repeat

Every rule can be written as an unsigned 8-bit number: 00011110_2 = 30
There are only 2^8 = 256 unique rules. Let's explore them all!

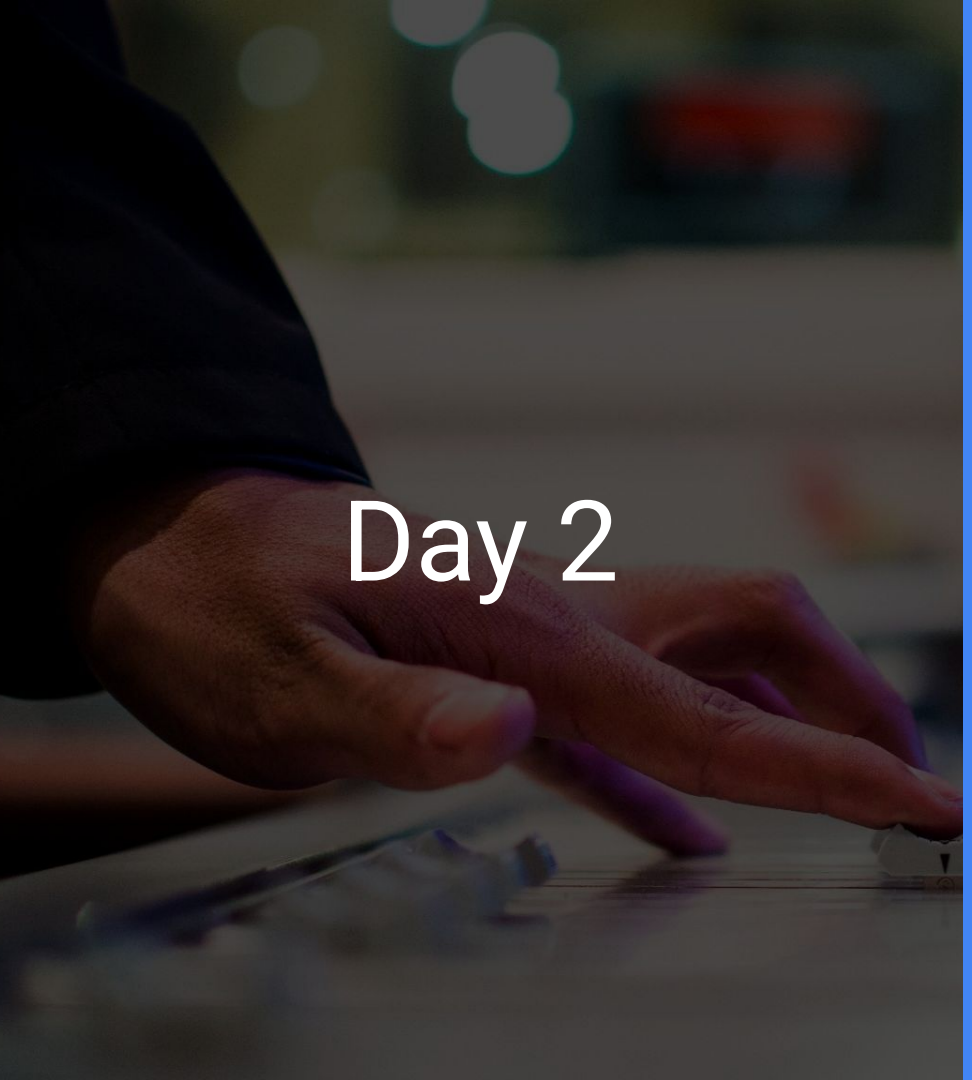
Examples of elementary cellular automata







https://mathworld.wolfram.com/ElementaryCellularAutomaton.html



Day 2

- Detailed coverage of the type system and multiple dispatch.
- Metaprogramming
- Synthesis exercise
- Overview of some cool Julia projects

Day 1			
Time	Description	Learning outcomes	Method
9:00	*presentation: introduction*	What is julia? What are the main strengths, Introduction of teachers, scope and planning of course	presentation
9:30	*presentation: basic concepts*	Walkthrough of basic syntax	presentation
10:00	*break*		
10:15	*exercises:* 01-basics	introduce basic programming concepts in julia	exercises
11:00	*intermezzo:* plotting	learn how to use plotting.jl	demo
11:15	*exercises:* 01-basics	apply/implement basic programming concepts	exercises
12:15	*lunch break*		
12:45	*intermezzo: IDEs*	Explanation of how to use Julia (IDE, REPL, notebook)	presentation
12:55	*presentation*: collections	introduce basic collections	presentation
13:00	*exercises:* 02-collections	apply/implement basic collections	exercises
13:45	*break*		
14:00	*exercises:* 02-collections	apply/implement basic collections	exercises
15:00	presentation: synthesis exercises	introduce exercises, understand minimal requirements	presentation
15:15	*break*		
15:30	*exercises:* 03-exercises	synthesise concepts of day 1	exercises
17:00	end of day 1		

Day 2			
Time	**Description**	**Learning outcomes**	**Method**
9:00	*presentation:* overview day 2		presentation
9:05	*presentation: introduction types system*	learn the type system in julia	presentation
9:20	*exercises:* 01-types	apply the type system in julia	exercises
10:15	*break*		
10:30	*presentation: composite types*	what are composite types?	presentation
10:45	*excercises: 02-composite-types*	apply composite types	exercises
12:00	*optional: power of multiple dispatch + performance*	gain deeper insight into the julia type system	movie
12:30	*lunch break*		
13:00	*presentation:* introduction to macros	understand the power of macros	presentation
13:15	*presentation*: synthesis exercises	synthesise concepts of day 2	presentation
13:30	*synthesis exercises*	synthesise concepts of day 2	exercises
14:30	*break*		
14:15	*Intermezzo: useful tools in the julia landscape*	learn about the existence of optim.jl, diffeq, drwatson and others	demo
14:45	*synthesis exercises*	synthesise concepts of day 2	exercises
16:30	end of course		

Minimal requirements.

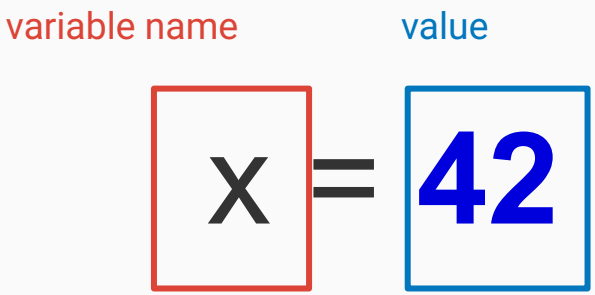
- *day1/01-basics.jl*
  - Questions 1 to 5 and 7.1 (no terminal input)

- *day1/02-collections.jl*
  - Question 1: Riemann sum
  - Question 2: N-rook problem
  - Question 3: Estimating pi
  - Question 4: Password

- *day1/03-exercises.jl*
  - Everything except cellular automata

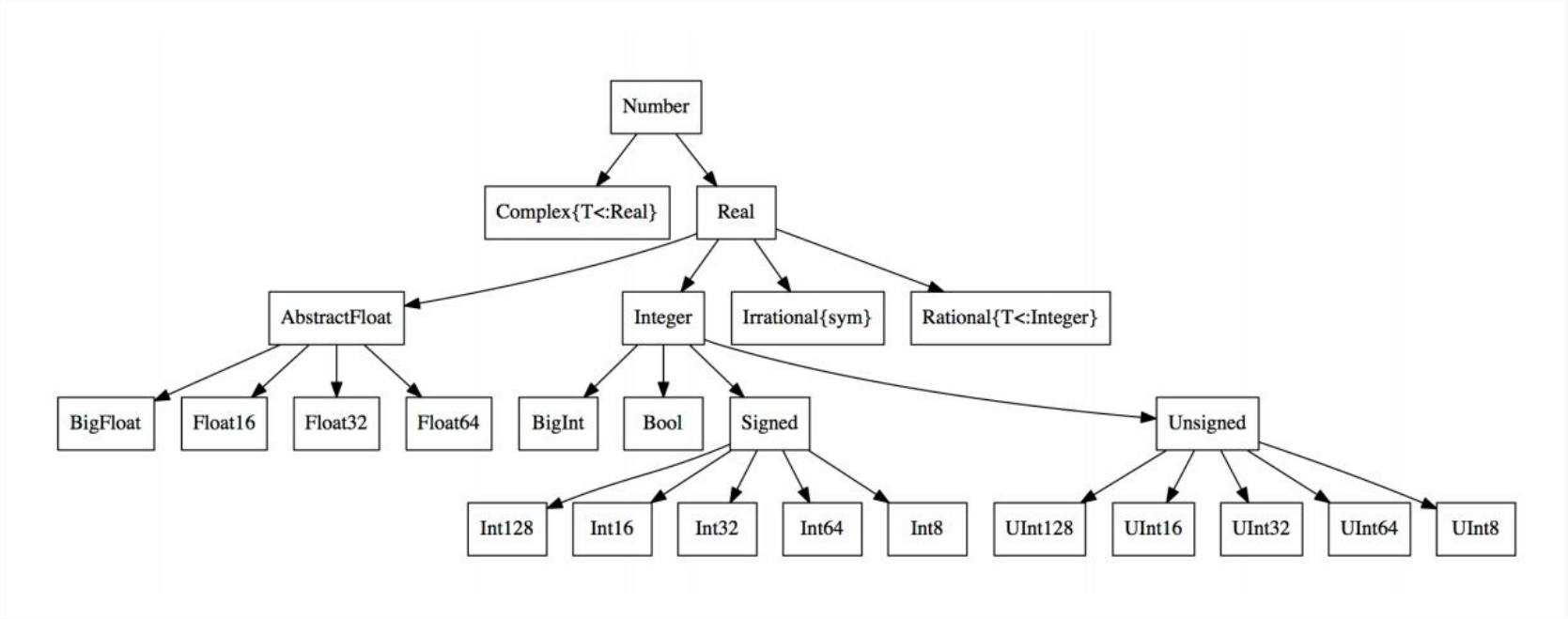
- *day2/02-composites-types.jl*
  - Vandermonde matrix **AND** Wizarding currency + Molecules OR Colors
- day2/03-fluky_fields.jl
  - One type of shape!

Fill in attendance sheet!




```
julia> typeof(x)
Int64
```

What is a number?



A combinatorial explosion

```
julia> @code_llvm 1 + 2

;  @ int.jl:86 within `+'
define i64 @"julia_+_385"(i64, i64) {
top:
  %2 = add i64 %1, %0
  ret i64 %2
}
```

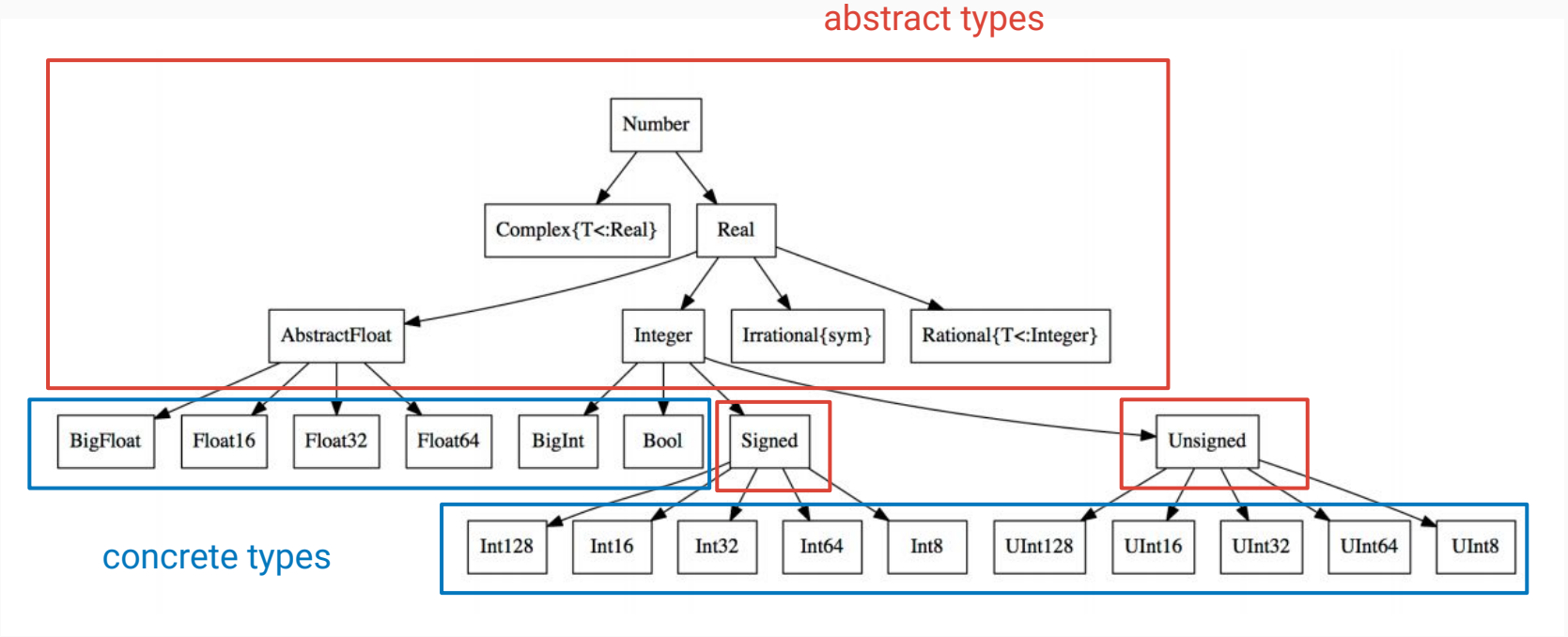
```
julia> @code_llvm 1.0 + 2.0

;  @ float.jl:401 within `+'
define double @"julia_+_386"(double, double) {
top:
  %2 = fadd double %0, %1
  ret double %2
}
```

```
julia> @code_llvm 1 + 2.0

;  @ promotion.jl:311 within `+'
define double @"julia_+_387"(i64, double) {
top:
; ┌ @ promotion.jl:282 within `promote'
; │┌ @ promotion.jl:259 within `_promote'
; ││ ┌ @ number.jl:7 within `convert'
; │││ ┌ @ float.jl:60 within `Float64'
     %2 = sitofp i64 %0 to double
; └└└└
;  @ promotion.jl:311 within `+' @ float.jl:401
  %3 = fadd double %2, %1
;  @ promotion.jl:311 within `+'
  ret double %3
}
```

Why is Julia so fast?

- Whenever a function is run with a particular type signature, it is compiled by the LLVM compiler (this takes time and is the reason why Julia is slow on the first call);
- For this process to occur, the function has to be **type stable**: for a given combination of inputs it should always return the same types of outputs.
- As most interpreted languages (Python, ...) cannot determine the type of the output, a lot of checks have to be done every step, making these languages slow.

What is a number?

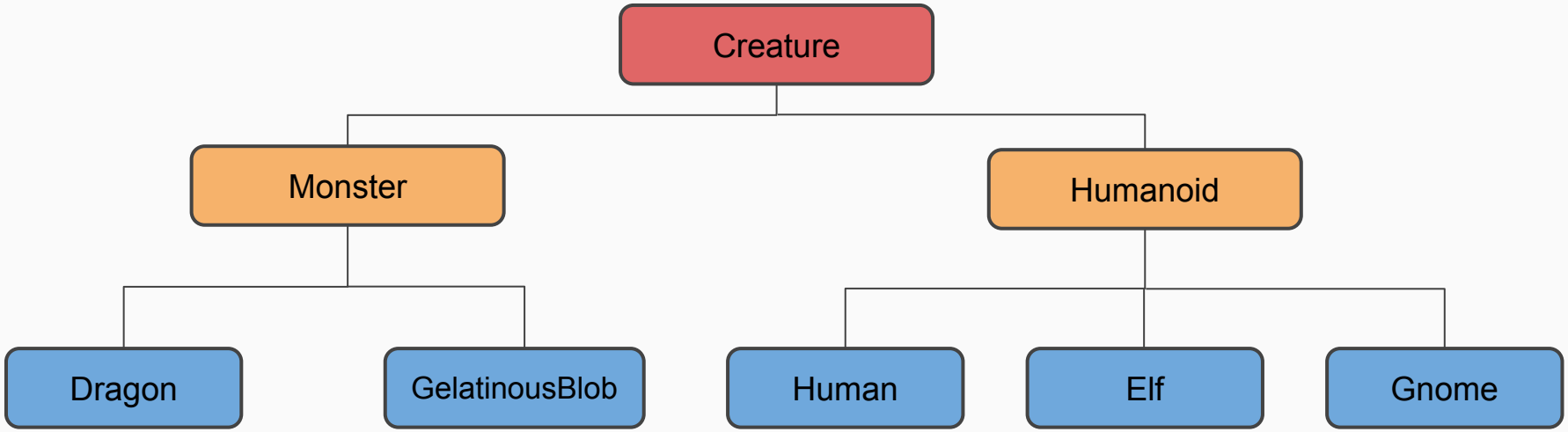


Julia’s type system in brief

- Abstract types: used to define the concrete type system
- Concrete types: can be instantiated
  - Primitive types: consists of bits (Int, Float64, UInt…)
  - Composite types: bind other data
    - static: fields cannot be changed
    - mutable: fields can be changed

Parametric types can take parameters in their types, making them very flexible. For example, matrices can work with any number!

Example: Dungeons and Dragons type system




```
abstract type Creature end

abstract type Humanoid <: Creature end

abstract type Monster <: Creature end
```

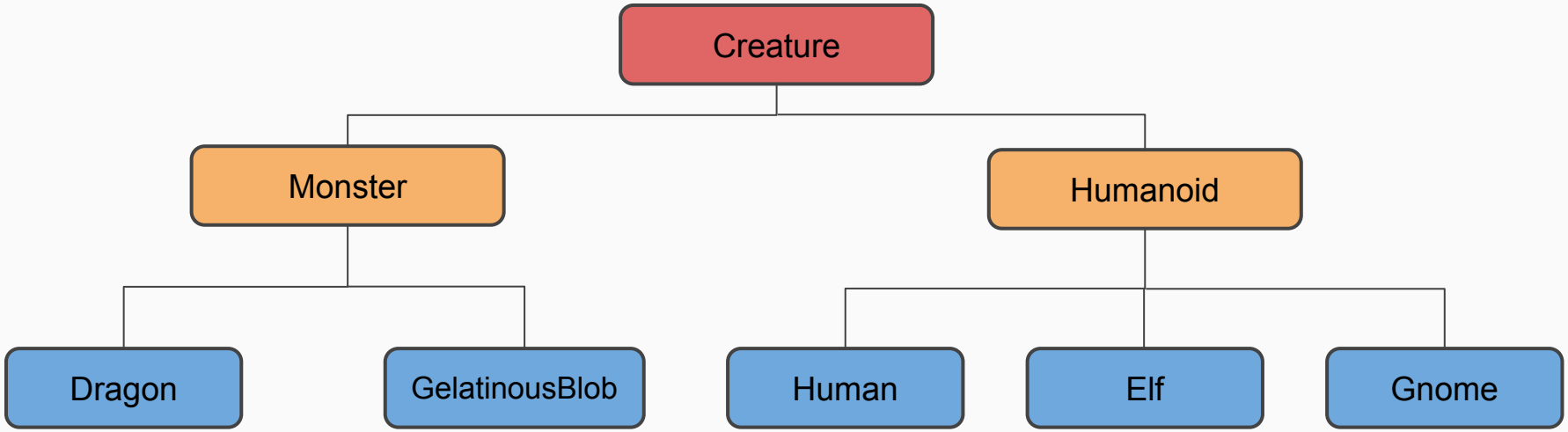
Methods

- Functions are automatically compiled for their type signature
- You can restrict the inputs using `::` signatures, for example f(x::Number, y::Number) only accepts two numbers as input.
- You are free to create several version with different type signatures, these are called **methods**:
  - repeat(x::Number, n::Int) = [x for i in 1:n]
  - repeat(x::Number) = [x, x]
  - repeat(x::String, n::Int) = x^n
- The fact that you are free to define methods for any arbitrary input type combination is called **multiple dispatch**.

methods(*)


I'M A JULIA EXPERT
OH YEAH?
NAME 328 METHODS
*
That's on me, I set
the bar too low.

Example: Dungeons and Dragons type system



languages(::Humanoid) = "Common Tongue"
languages(::Elf) = "Common Tongue, Elvish"

encounter(::Creature, ::Creature) = nothing
encounter(::Humanoid, ::Monster) = "run!"
encounter(::Monster, ::Humanoid) = "attack!"
encounter(::Humanoid, ::Humanoid) = "talk"
encounter(::Gnome, ::Humanoid) = "steals gold..."

Concrete types and constructors

Concrete types are defined using `struct … end` or `mutable … end`

Constructors are functions that can initialize an object of a type:

- **Outer constructors** are defined outside the type definition;
- **Inner constructors** are defined in the type using the keyword `new`

```
mutable struct Human <: Humanoid
    name::String
    hp::Int
    level::Int
    gold::Int
end


Human(name) = Human(name, 10, 1, rand(20:50))
```

```
mutable struct Elf <: Humanoid
    name::String
    hp::Int
    level::Int
    gold::Int
    function Elf(name)
        return new(name, 12, 1, rand(10:40))
    end
end
```

Constructors are optional, e.g. `Human(“Drax”, 10, 2, 100)` works out of the box.

Example of a parametric type

There are two types of monsters:

- HP is recorded as the number of hitpoints;
- HP is recorded as the fraction of the maximum HP, i.e. in [0, 1]

Both cases can be captured using **parametric types**

```
mutable struct Dragon{T<:Union{Float64,Int}} <: Creature
    hp::T
    strength::Int
    magic::Int
end

mutable struct GelatinousCube{T<:Union{Float64,Int}} <: Creature
    hp::T
    strength::Int
    object::String
end
```

```
damage(opponent, dragon::Dragon{Int}) = max(0, opponent.level - dragon.strength)
damage(opponent, dragon::Dragon{Float64}) = max(0.0, (opponent.level - dragon.strength) / dragon.strength)
```

Case study: Kronecker products

If **A** is an $m \times n$ matrix and **B** is a $p \times q$ matrix, then the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the $pm \times qn$ block matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix},$$

more explicitly:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$

```
A = rand(Bool, 100, 100)
B = randn(50, 50)
K = A ⊗ B
Kdense = kron(A, B)  # native julia version

Base.inv(K::Kronecker) = Kronecker(inv(K.A), inv(K.B))

@time inv(K)  # inverting two small matrices => fast

@time inv(Kdense) # inverting a large matrix => slow
```

```
struct Kronecker{T,TA,TB} <: AbstractMatrix{T}
    A::TA  # first matrix
    B::TB  # second matrix
    # constructor
    function Kronecker(A::TA, B::TB) where
{TA<:AbstractMatrix,TB<:AbstractMatrix}
        T = promote_type(eltype(A), eltype(B))  # general type
        return new{T,TA,TB}(A, B)
    end
end

Base.eltype(K::Kronecker{T,TA,TB}) where {T,TA,TB} = T
Base.size(K::Kronecker) = size(K.A) .* size(K.B)

function  Base.getindex(K::Kronecker, i, j)
    A, B = K.A, K.B
    n, m = size(A)
    p, q = size(B)
    return A[(i-1)÷p+1,(j-1)÷q+1] * B[(i-1)%p+1,(j-1)%q+1]
end

# syntactic sugar
⊗ = Kronecker
```

Uncertainty propagation

Function	Variance	Standard Deviation
$f = aA$	$\sigma_f^2 = a^2\sigma_A^2$	$\sigma_f = \lvert a\rvert\sigma_A$
$f = aA + bB$	$\sigma_f^2 = a^2\sigma_A^2 + b^2\sigma_B^2 + 2ab\,\sigma_{AB}$	$\sigma_f = \sqrt{a^2\sigma_A^2 + b^2\sigma_B^2 + 2ab\,\sigma_{AB}}$
$f = aA - bB$	$\sigma_f^2 = a^2\sigma_A^2 + b^2\sigma_B^2 - 2ab\,\sigma_{AB}$	$\sigma_f = \sqrt{a^2\sigma_A^2 + b^2\sigma_B^2 - 2ab\,\sigma_{AB}}$
$f = A - B,$	$\sigma_f^2 = \sigma_A^2 + \sigma_B^2 - 2\sigma_{AB}$	$\sigma_f = \sqrt{\sigma_A^2 + \sigma_B^2 - 2\sigma_{AB}}$
$f = aA - aA,$	$\sigma_f^2 = 2a^2\sigma_A^2(1-\rho_A)$	$\sigma_f = \sqrt{2}\,\lvert a\rvert\,\sigma_A(1-\rho_A)^{1/2}$
$f = AB$	$\sigma_f^2 \approx f^2\left[\left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2 + 2\frac{\sigma_{AB}}{AB}\right]$ [9][10]	$\sigma_f \approx \lvert f\rvert\sqrt{\left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2 + 2\frac{\sigma_{AB}}{AB}}$
$f = \frac{A}{B}$	$\sigma_f^2 \approx f^2\left[\left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2 - 2\frac{\sigma_{AB}}{AB}\right]$ [11]	$\sigma_f \approx \lvert f\rvert\sqrt{\left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2 - 2\frac{\sigma_{AB}}{AB}}$
$f = aA^b$	$\sigma_f^2 \approx \left(abA^{b-1}\sigma_A\right)^2 = \left(\frac{fb\sigma_A}{A}\right)^2$	$\sigma_f \approx \left\lvert abA^{b-1}\sigma_A\right\rvert = \left\lvert\frac{fb\sigma_A}{A}\right\rvert$
$f = a\ln(bA)$	$\sigma_f^2 \approx \left(a\frac{\sigma_A}{A}\right)^2$ [12]	$\sigma_f \approx \left\lvert a\frac{\sigma_A}{A}\right\rvert$
$f = a\log_{10}(bA)$	$\sigma_f^2 \approx \left(a\frac{\sigma_A}{A\ln(10)}\right)^2$ [12]	$\sigma_f \approx \left\lvert a\frac{\sigma_A}{A\ln(10)}\right\rvert$
$f = ae^{bA}$	$\sigma_f^2 \approx f^2(b\sigma_A)^2$ [13]	$\sigma_f \approx \lvert f\rvert\,\lvert(b\sigma_A)\rvert$
$f = a^{bA}$	$\sigma_f^2 \approx f^2(b\ln(a)\sigma_A)^2$	$\sigma_f \approx \lvert f\rvert\,\lvert(b\ln(a)\sigma_A)\rvert$
$f = a\sin(bA)$	$\sigma_f^2 \approx [ab\cos(bA)\sigma_A]^2$	$\sigma_f \approx \lvert ab\cos(bA)\sigma_A\rvert$
$f = a\cos(bA)$	$\sigma_f^2 \approx [ab\sin(bA)\sigma_A]^2$	$\sigma_f \approx \lvert ab\sin(bA)\sigma_A\rvert$
$f = a\tan(bA)$	$\sigma_f^2 \approx \left[ab\sec^2(bA)\sigma_A\right]^2$	$\sigma_f \approx \left\lvert ab\sec^2(bA)\sigma_A\right\rvert$
$f = A^B$	$\sigma_f^2 \approx f^2\left[\left(\frac{B}{A}\sigma_A\right)^2 + (\ln(A)\sigma_B)^2 + 2\frac{B\ln(A)}{A}\sigma_{AB}\right]$	$\sigma_f \approx \lvert f\rvert\sqrt{\left(\frac{B}{A}\sigma_A\right)^2 + (\ln(A)\sigma_B)^2 + 2\frac{B\ln(A)}{A}\sigma_{AB}}$
$f = \sqrt{aA^2 \pm bB^2}$	$\sigma_f^2 \approx \left(\frac{A}{f}\right)^2 a^2\sigma_A^2 + \left(\frac{B}{f}\right)^2 b^2\sigma_B^2 \pm 2ab\frac{AB}{f^2}\sigma_{AB}$	$\sigma_f \approx \sqrt{\left(\frac{A}{f}\right)^2 a^2\sigma_A^2 + \left(\frac{B}{f}\right)^2 b^2\sigma_B^2 \pm 2ab\frac{AB}{f^2}\sigma_{AB}}$

Case study: error propagation

```
struct Measurement{T<:AbstractFloat} <: Number
    x::T
    σ::T
    function Measurement(x::T, σ::T) where
{T<:AbstractFloat}
        if σ < zero(σ)
            error("Measurement error should be
non-zero")
        end
        new{T}(x, σ)
    end
end

val(m::Measurement) = m.x;
err(m::Measurement) = m.σ;

Base.show(io::IO, measurement::Measurement) =
print(io, "$(measurement.x) ± $(measurement.
σ)");


±(x, σ) = Measurement(x, σ);
```

```
# scalar multiplication
Base.:*(a::Real, m::Measurement) = a * m.x ± abs(a) * m.σ;
Base.:/(m::Measurement, a::Real) = inv(a) * m;
# adding and substracting measurements
Base.:+(m1::Measurement, m2::Measurement) = (m1.x + m2.x) ± √(m1.σ^2 +
m2.σ^2);
Base.:-(m1::Measurement, m2::Measurement) = (m1.x - m2.x) ± √(m1.σ^2 +
m2.σ^2);
Base.:-(m::Measurement) = -m.x ± m.σ;
# adding a constant
Base.:+(m::Measurement, a::Real) = m + (a ± zero(a));
Base.:+(a::Real, m::Measurement) = m + a;
# multiplying two measurements
Base.:*(m1::Measurement, m2::Measurement) = m1.x * m2.x ± (m1.x * m2.x) *
√((m1.σ / m1.x))
```

Case study: type-based dispatch

The type system can be used to mix-and-match different algorithms, this is called **type-based dispatch**. It is extremely useful in scientific code:

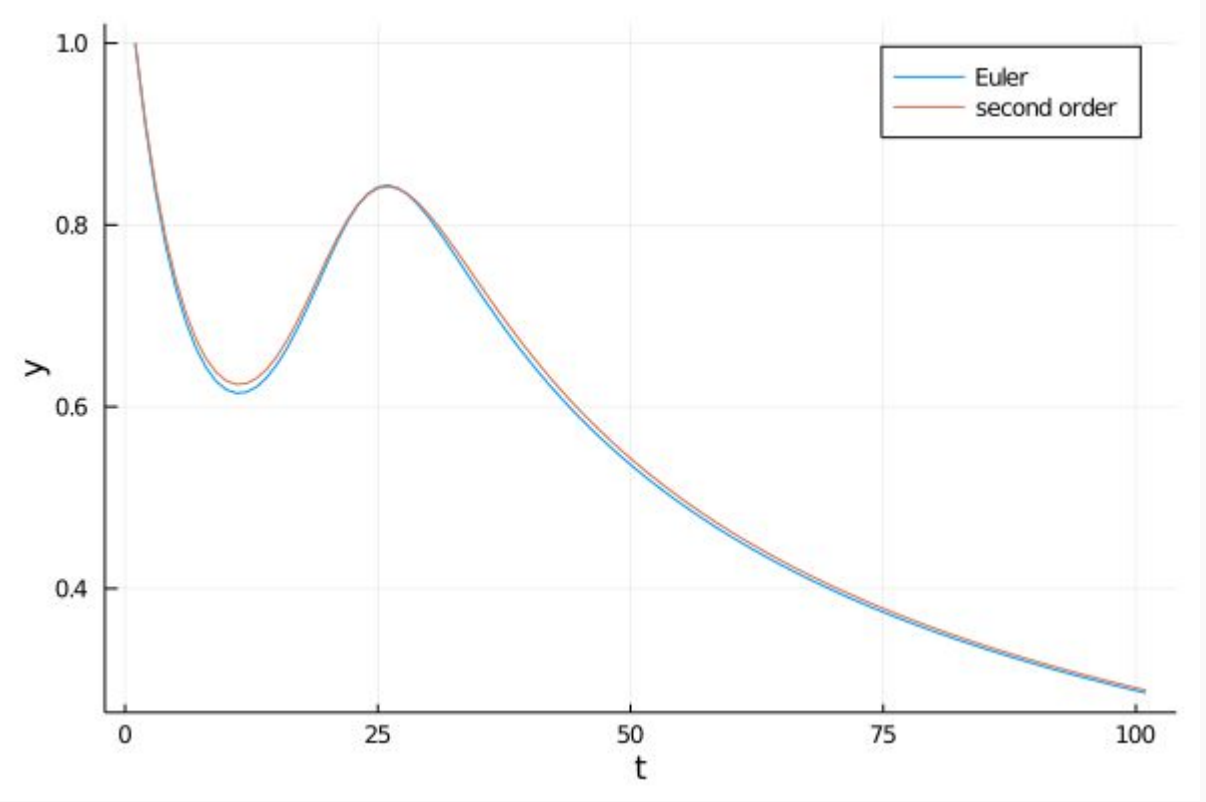
```
abstract type ODESolver end

struct Euler <: ODESolver end
struct SecondOrder <: ODESolver end

function solve_ode(f′, (t₀, tₑ), y₀, Δt, method::ODESolver)
    tsteps = t₀:Δt:tₑ
    # initialize solution vector
    y = zeros(length(tsteps))
    # set initial value
    y[1] = y₀
    for (i, t) in enumerate(tsteps)
        i == 1 && continue
        yᵢ₋₁ = y[i-1]
        yᵢ = step(f′, t, yᵢ₋₁, Δt, method)
        y[i] = yᵢ
    end
    return y
end
```

```
step(f′, t, yᵢ₋₁, Δt, ::Euler) = yᵢ₋₁ + Δt * f′(t, yᵢ₋₁)

function step(f′, t, yᵢ₋₁, Δt, ::SecondOrder)
    y = yᵢ₋₁ + 0.5Δt * f′(t, yᵢ₋₁)  # half step
    return yᵢ₋₁ + Δt * f′(t, y)
end
```

```
f′(t, y) = -y + sin(t * y)

y_euler = solve_ode(f′, (0, 10), 1.0, 0.1, Euler())
y_second_order = solve_ode(f′, (0, 10), 1.0, 0.1, SecondOrder())

using Plots
plot(y_euler, label="Euler")
plot!(y_second_order, label="second order")
```


a few
nanoseconds
later*
* if you use Julia

Case study: type-based dispatch



Object-oriented vs functional programming

Object-oriented

Define a bicycle object prototype

attributes:

- speed
- gear

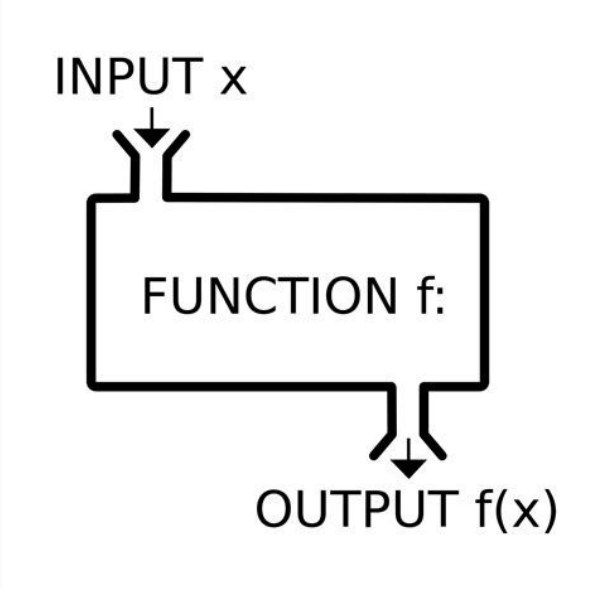
behaviours:

- speed up
- apply brake
- change gear

```
class bicycle:
    ''' properties'''
    # Class variables.
    gear = 1
    speed = 0

    def __init__(self, gear, speed):
        self.gear = gear
        self.speed = speed

    def speedUp(self, increase):
        self.speed += increase

    def changeGear(self,newGear):
        self.gear = newGear

    def applyBrake(self, decrease):
        self.speed -= decrease
```

"noun"-oriented programming
Easy to make new objects, hard to make new operations

Verb-based



"verb"-oriented programming
Easy to make new operations, hard to make new objects


juliacon
Baltimore 2019
Julia computing
J.P.Morgan
intel
relational AI
Google

Performance tips

- No global variables! (or use `const` if you must)
- Profile, mainly using @time or @btime from BenchmarkTools
- Always use concrete types for collections, e.g. use `Float64[1.0, 2.0]` rather than `Real[1.0, 2.0]`
- Write type-stable functions, e.g. `relu(x) = x < 0 ? zero(x) : x` rather than `relu(x) = x < 0 ? 0 : x`
- Don’t change the type of a variable in your program
- Pre-allocate inputs (but this is always a good idea)
- Use dots! Vectorization of your code might be slower than for-loops!
- Use `@inbounds`, `@fastmath` and `@simd` (but use them wisely)

https://docs.julialang.org/en/v1/manual/performance-tips/

Make it easy for the compiler to create fast code!

Lunchbreak

Introduction to metaprogramming

Meta-programming is when you write Julia code to process and modify Julia code

In Julia, the execution of raw source code takes place in two stages

1. Your raw Julia code is parsed, result is an abstract syntax tree
2. Your parsed code is executed

With Julia's metaprogramming facilities, you can access the code **after** it's been parsed but **before** it's evaluated.

This lets you do things that you can't normally do.

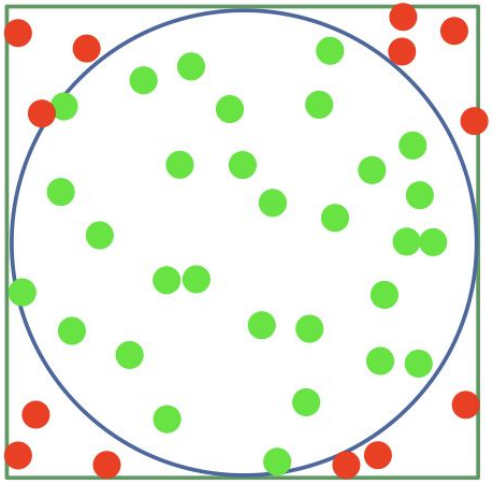
See notebook: `day2/04-metaprogramming.jl`

Project 2: Fluky Fields

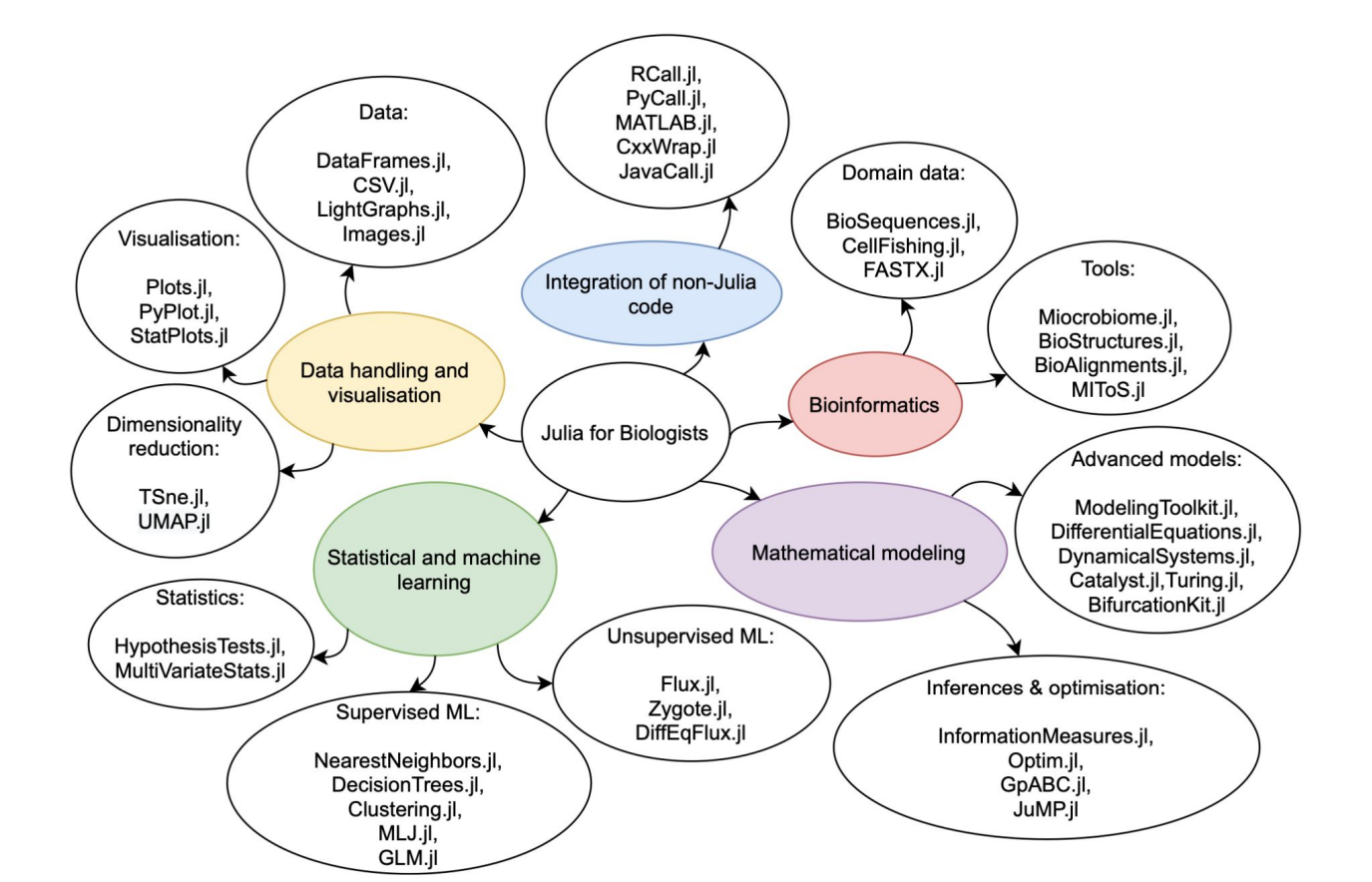
Project 2 goals

Implement a shape type (Circle, Rectangle or Triangle) with the following functionality:

1. area method;
2. a method that checks if two shapes overlap
3. a function to plot the shapes;
4. The total area of the intersection of all the objects



Your basic Julia toolkit



Closing remarks

Useful and **mature** Julia packages**,**

- *DifferentialEquations.jl*: extensive library for solving differential equations;
- *Optim.jl* and *JuMP*: univariate and multivariate optimisation in library;
- Agents.jl: agent-based modelling
- DataFrames.jl: tabular data manipulation;
- *Flux.jl:* lightweight machine learning library (strong focus on neural networks);
- Turing.jl: probabilistic programming;
- Plots.jl: plotting library;
- Zygote.jl: automatic differentiation;
- Symbolics.jl and ModellingToolkit.jl: symbolic reasoning about your model;
- IJulia.jl: Jupyter notebook kernel for julia.

Julia communications channels for **help,**

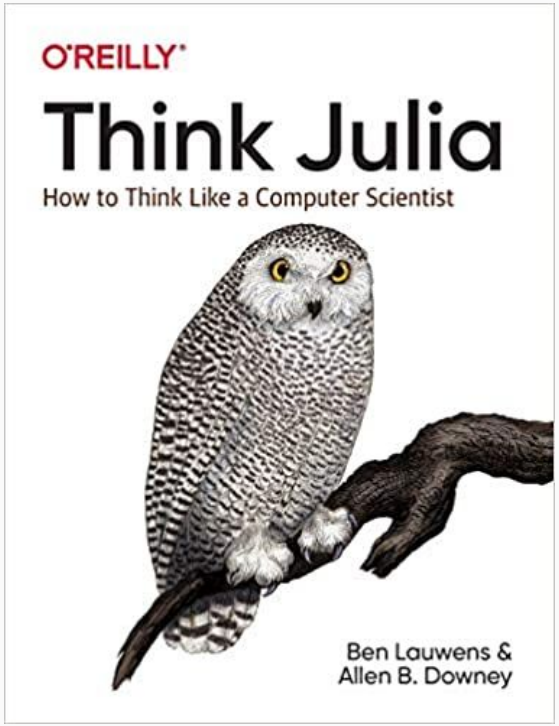
- The Julia Language Slack;
- Julia Discourse;
- A list of official communication channels and Julia-related events.

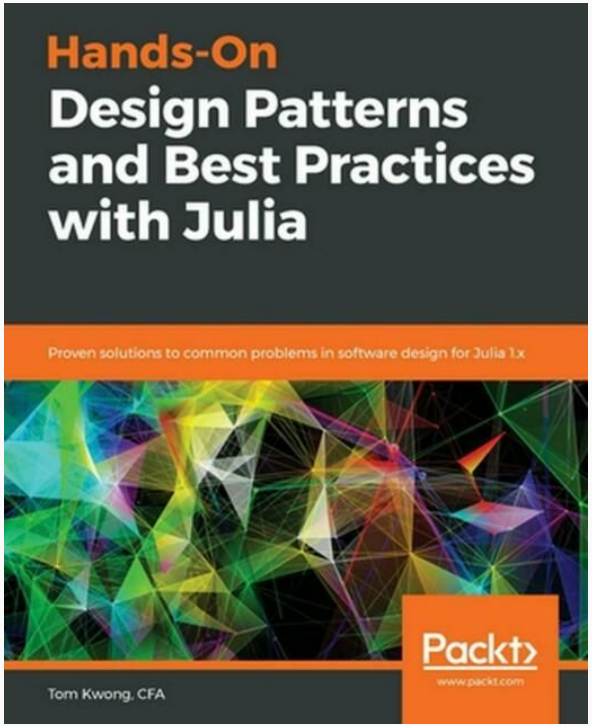
juliacon 2024

Conference presentations can be viewed afterwards on
www.youtube.com/@TheJuliaLanguage

Roesch, E., Greener, J. G., MacLean, A. L., Nassar, H., Rackauckas, C., Holy, T. E., & Stumpf, M. P. H. (2021). Julia for Biologists. Retrieved from http://arxiv.org/abs/2109.09973

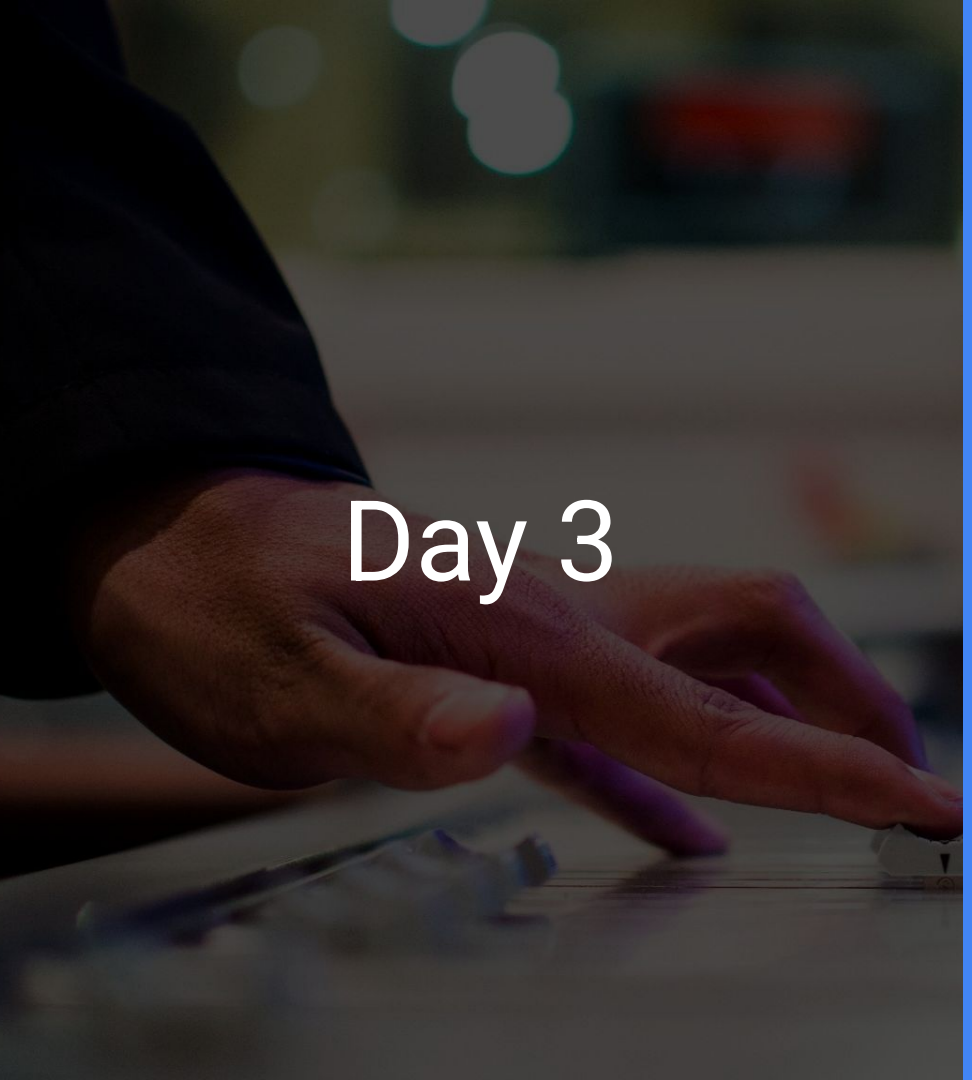
Recommended references





www.youtube.com/@doggodotjl

https://benlauwens.github.io/ThinkJulia.jl/latest/book.html



Day 3

- DrWatson overview
- Projects overview
- Optional: Git and Github

Interesting package: DrWatson

DrWatson

The perfect sidekick to your scientific inquiries

https://github.com/JuliaDynamics/DrWatson.jl
https://youtu.be/jKATlEAu8eE

Interesting package: DrWatson

- DrWatson = scientific project assist software

Have you thought things like:

- Urgh, I moved my folders and now my load commands don't work anymore!
- Hold on, haven't I run this simulation already?
- Do I have to produce a dataframe of my finished simulations AGAIN?!
- Wait, are those experiments already processed?
- PFffffff I am tired of typing savename = "w=$w_f=$f_x=$x.txt", can't I do it automatically?
- I wish I could just make a dataframe out of all my simulations with one command!
- Yeah you've sent me your project but none of the scripts work...
- It would be so nice to automatically integrate git information to all the data I save...

DrWatson tries to eradicate such bad thoughts and bedtime nightmares.

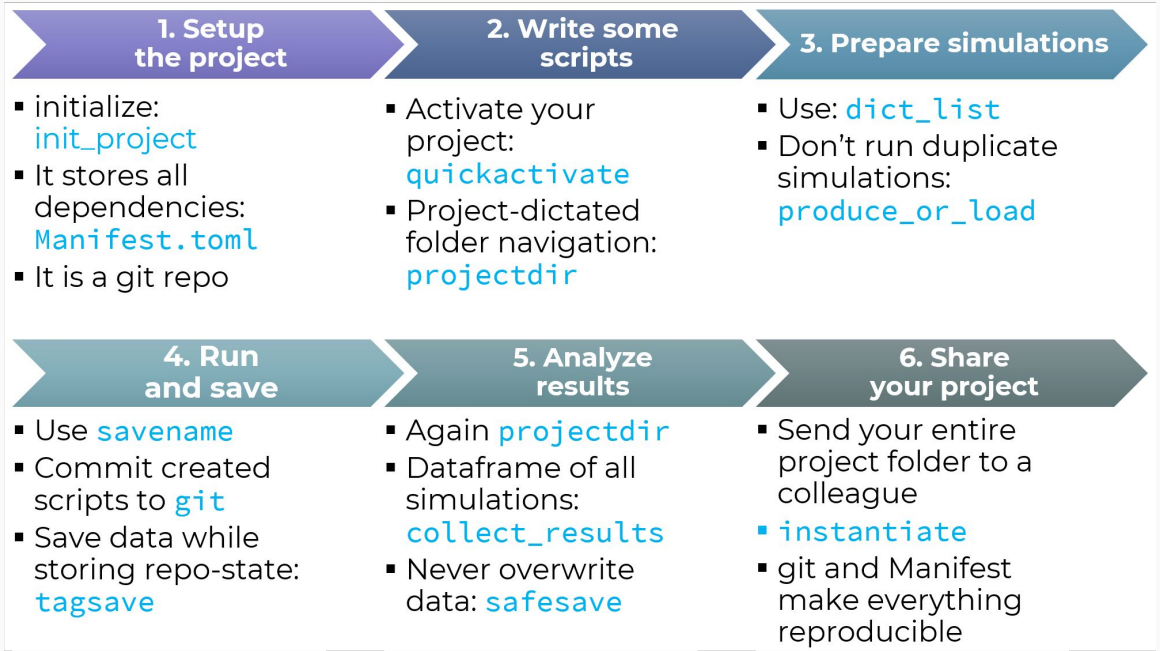
Interesting package: DrWatson

- Project Setup :
  A universal project structure and functions that allow you to consistently and robustly navigate through your project, no matter where it is located.
- Naming Simulations :
  A robust and deterministic scheme for naming and handling your containers.
- Saving Tools :
  Tools for safely saving and loading your data, tagging the Git commit ID to your saved files, safety when tagging with dirty repos, and more.
- Running & Listing Simulations:
  Tools for producing tables of existing simulations/data, adding new simulation results to the tables, preparing batch parameter containers, and more.

IMPORTANT: DrWatson is

- **not a data management system**.
- **not a Julia package creator** like PkgTemplates.jl **nor a package development tool**.

Interesting package: DrWatson



Full overview here:
https://juliadynamics.github.io/DrWatson.jl/dev/workflow/

Walkthrough making a data science project

```
# Note: using temp to avoid polluting my system environment
using Pkg;Pkg.activate(;temp=true)
Pkg.add("DrWatson")
Pkg.add("Documenter")

using DrWatson

DrWatson.initialize_project("DrWatsonExample";
    readme=true,
    authors=["Jos", "Pierre"],
    force=false,
    git=true,
    add_test=true,
    add_docs=true,
    template=DrWatson.DEFAULT_TEMPLATE,
    placeholder=true
)
```

Walkthrough making a package

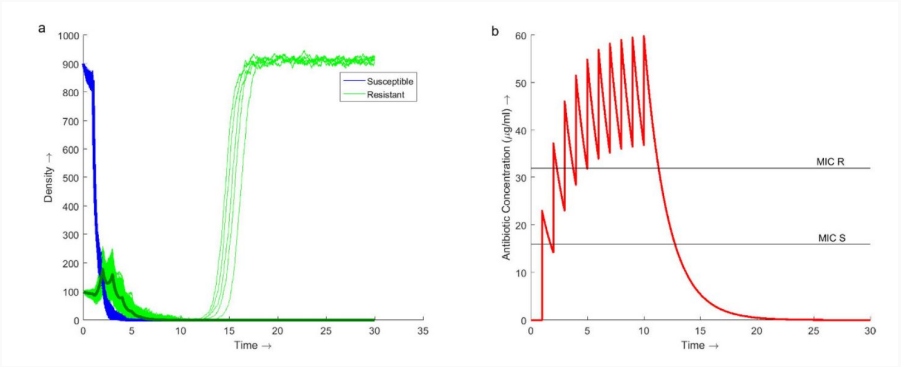
```
# Note: using temp to avoid polluting my system environment
using Pkg;Pkg.activate(;temp=true)
Pkg.add("PkgTemplates")
using PkgTemplates
# Create a new package template
t = Template(;
        user="my-username",
        dir="./code",
        authors="Acme Corp",
        julia=v"1.10",
        plugins=[
        License(; name="MIT"),
        Git(; manifest=true, ssh=true),
        GitHubActions(; x86=true),
        ],
)
t("YourPackageName") # generating a package

# add to src/functions.jl
function greet_your_package_name()
        return "Hello YourPackageName!"
end
# add to src/YourPackageName.jl
export greet_your_package_name
include("functions.jl")
# add to tests
@test YourPackageName.greet_your_package_name() == "Hello YourPackageName!"
@test YourPackageName.greet_your_package_name() != "Hello world!"

# julia shell
"""
; cd code/YourPackageName
] activate .
] test
"""
```

Project 1: modelling antibiotics treatment

This project explores the impact of antibiotic dosing on a bacterial system using a simple ordinary differential equation (ODE) model. By optimizing dosage and timing, we aim to minimize bacterial infection while minimizing overall antibiotic usage.

Optimising Antibiotic Usage to Treat Bacterial Infections

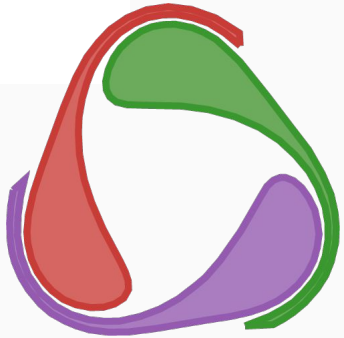
Iona K. Paterson[1], Andy Hoyle[1], Gabriela Ochoa[1], Craig Baker-Austin[2] & Nick G. H. Taylor[2]



$$\frac{dS}{dt} = \underbrace{rS\left(1 - \frac{S+R}{K}\right) - \theta S}_{\text{Natural Growth}} - \underbrace{\beta SR}_{\text{HGT}} - \underbrace{A_S(C)S}_{\text{AB Death}}$$

$$\frac{dR}{dt} = \underbrace{rR\left(1 - \frac{S+R}{K}\right)(1-a) - \theta R}_{\text{Natural Growth}} + \underbrace{\beta SR}_{\text{HGT}} - \underbrace{A_R(C)R}_{\text{AB Death}}$$

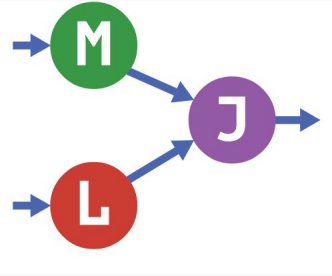
$$\frac{dC}{dt} = \underbrace{\sum_{n=1}^{10} D_n \delta(t - \hat{t}_n)}_{\text{Antibiotic Doses}} - \underbrace{gC}_{\text{Degredation}}$$



Project 2: machine learning

In this project, you will use Julia to go through the different steps of a data science or machine learning project. To be more specific, you are going to implement a supervised learning task. For this, you will need to follow these steps:

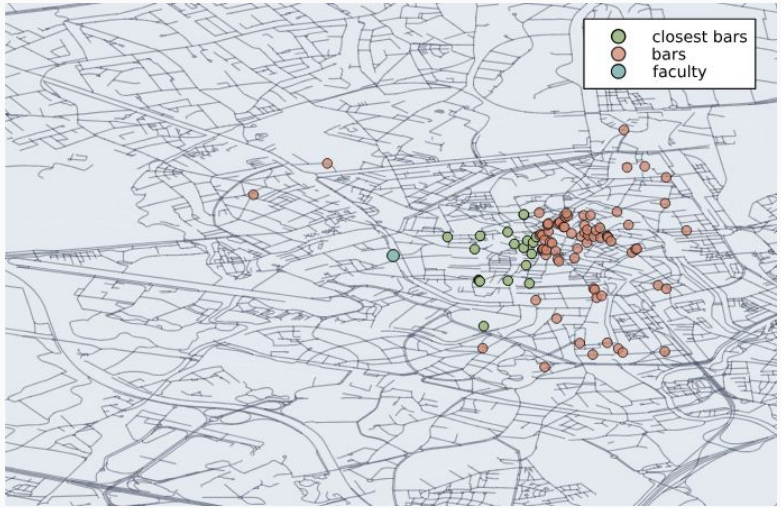
- get and process the data
- assess the data
- pose the modelling question
- train the model
- verify the model



Project extreme bar crawl

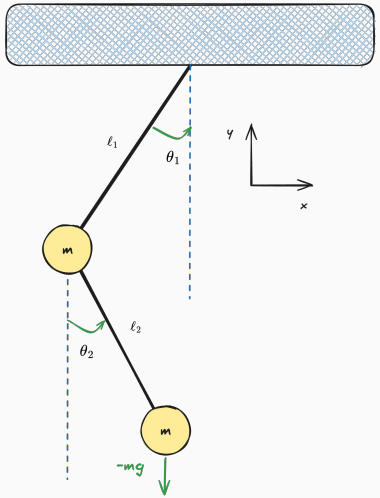
Day 3 of the Julia doctoral schools is a cover to explore some of Ghent's excellent bars. An efficient algorithm will be developed for computing the ideal bar crawl route. Ghent's Open Data Portal will be used to obtain an up-to-date overview of the bars and real-time availability of bicycles in a bicycle-sharing system. You will,

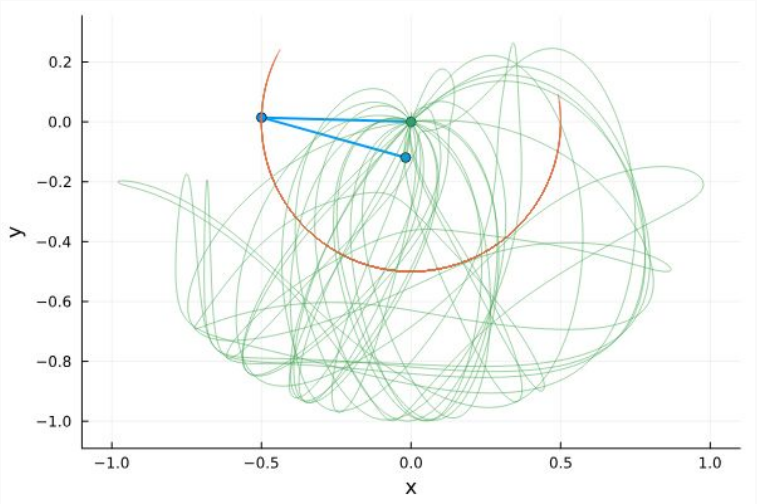
- fetch data API endpoints,
- data wrangling,
- create a map,
- solve an optimisation problem.



Project 3: mechanistic modelling

ModellingToolkit.jl (MTK) is a powerful acausial modelling software, that can automatically derive the model equations based on its performant computer algebra system. This project invites you to develop your own (physical) system based on ODE's together with the visualizations.

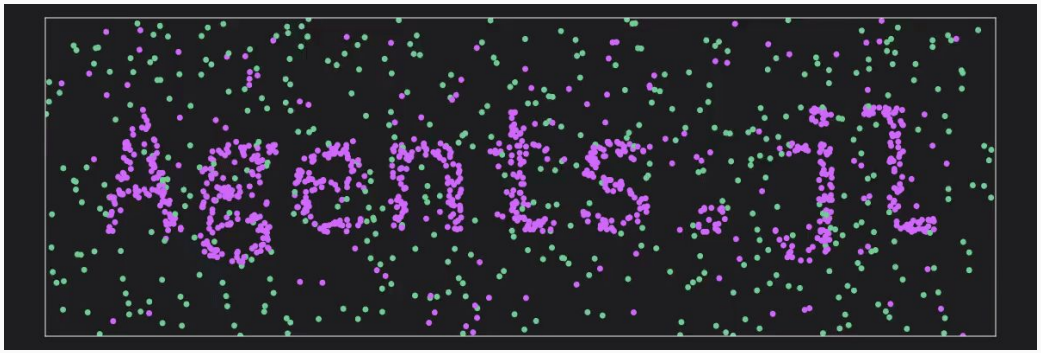




Idea: make [double pendulum fractal]

Project 4: Zombie outbreak using Agents.jl

This project explores an agent-based model (ABM) designed to simulate a zombie outbreak by modifying a SIR model.



https://juliadynamics.github.io/AgentsExampleZoo.jl/dev/

Project 5: Advent of Code

Advent of Code is a yearly programming challenge with daily puzzles. For this project you have to solve one day of choice, earning you two ⭐.

2024 Day 2: Red-Nosed Reports	order of lists	*
2024 Day 3: Mul it over	regex parsing	**
2024 Day 4: Ceres Search	find xmas words	***
2024 Day 5: Print Queue	custom sort, quite fun	*
2024 Day 7: Bridge Repair	going over combinations of operators and implementing an new operator, fun	...
2024 Day 9: Disk Fragmenter	working in dense format, quite a bit of work	***
2024 Day 11: Plutonian Pebbles	stones that change, tricky	**

julia

Goals for today

Minimum requirements:

- Complete 2 projects (AoC is one project)

Hand in your projects as a zip file
For AoC: send us your username so that we can check the stars.

That’s all folks!

Thanks for your interest in our course!

Please fill in the evaluation for the DS:

P.S.: if you find a bug or mistake please report it here.