Julia

Hong Kong Machine Learning Meetup

Emmanuel Rialland

April, 29th 2020

Plan for today

- What is Julia?
- Data Science - COVID 19 as an example
- Machine Learning - the Julia `SciML.jl` machine learning stack

What is Julia?

High-level features

Easy to use: Julia has high level syntax, making it an accessible language for programmers from any background or experience level.

Dynamic: Julia is dynamically-typed, feels like a scripting language, and has good support for interactive use.

Optionally typed: Julia has a rich language of descriptive datatypes, and type declarations can be used to clarify and solidify programs.

General: Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns.

Julia is fast!: Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Open source: Julia is provided under the MIT license, free for everyone to use. All source code is publicly viewable on GitHub.

Why We Created Julia

(from the authors)

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?)

We never want to mention types when we don't feel like it. But when we need **polymorphic functions**, we want to use **generic programming** to write an algorithm just once and apply it to an infinite lattice of types; we want to use **multiple dispatch** to efficiently pick the best method for all of a function's arguments, from dozens of method definitions, providing common functionality across drastically different types. Despite all this power, we want the language to be **simple and clean**.

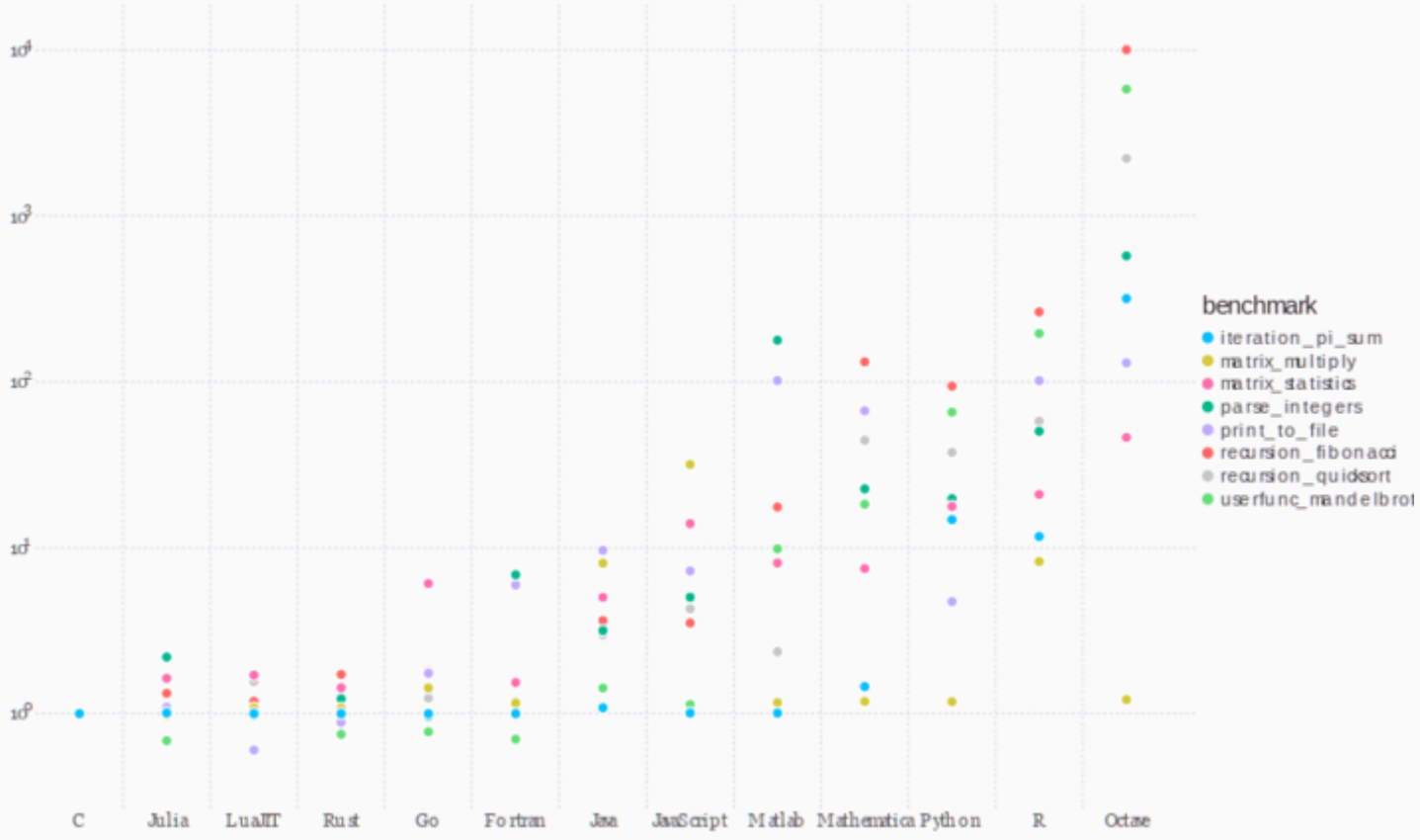
Source: https://julialang.org/blog/2012/02/why-we-created-julia/

And they did it!

Easy to use

- Interactive REPL like Python or R. Julia is the *Ju* in Jupyter
- Your preferred IDE supports it. Atom is the most mature, but VS Code, VIM, Emacs, SublimeText work fine. (This presentation is made with RStudio which can include Julia in its markdown.)
- Great community active and helpful on Discourse and Slack.

It is fast - (Old) Benchmark



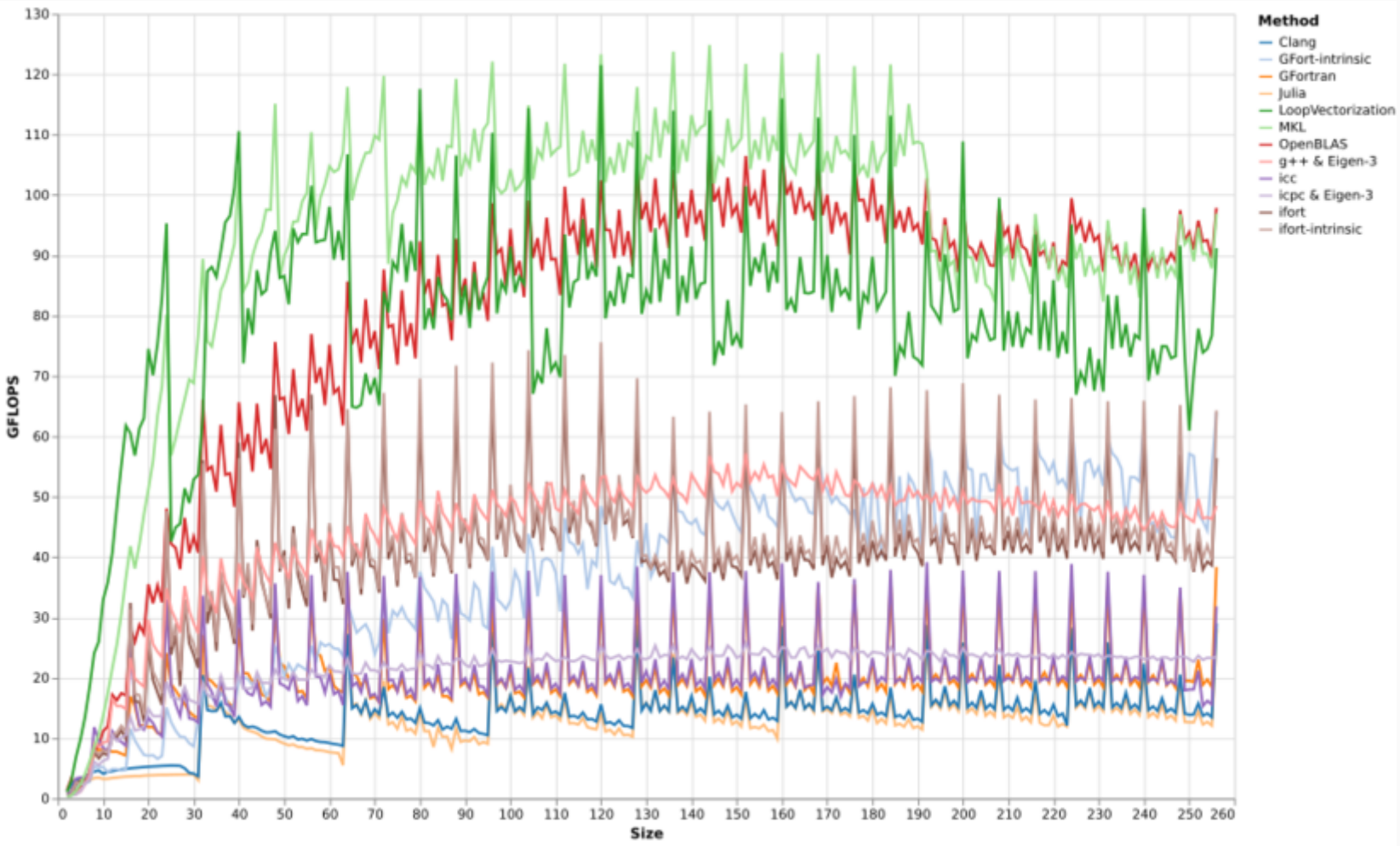
Julia v1.0.0, SciLua v1.0.0-b12, Rust 1.27.0, Go 1.9, Java 1.8.017, Javascript V8 6.2.414.54, Matlab R2018a, Python 3.6.3 (NumPy v1.14.0), R 3.5.0, and Octave 4.2.2. C and Fortran are compiled with gcc 7.3.1. See https://julialang.org/benchmarks/

Fast means *really* fast

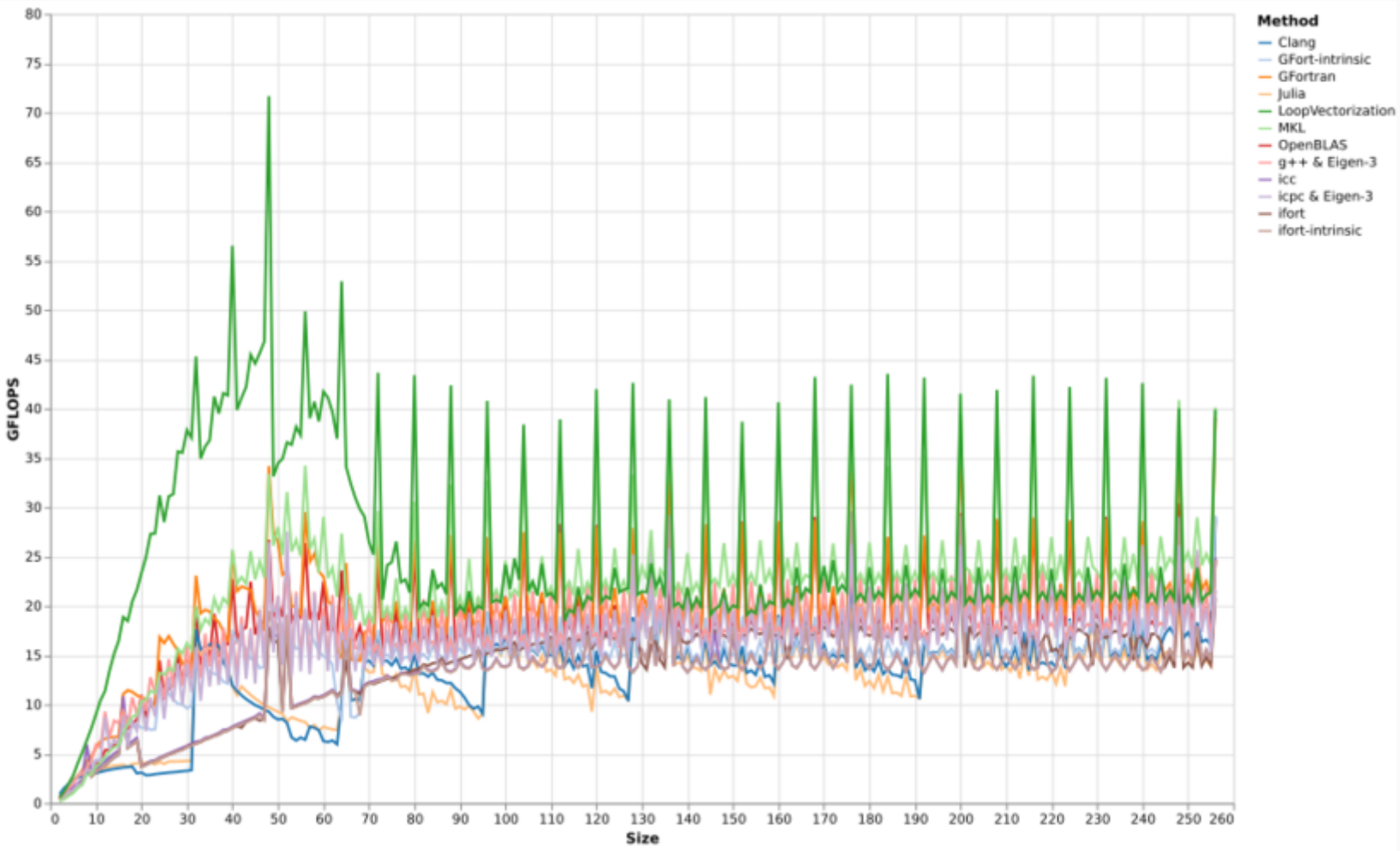
`LoopVectorization.jl`

(Recent library, pure Julia, available on any platform supported by LLVM)

Matrix multiplication



Matrix-vector multiplication



Sum-squared error

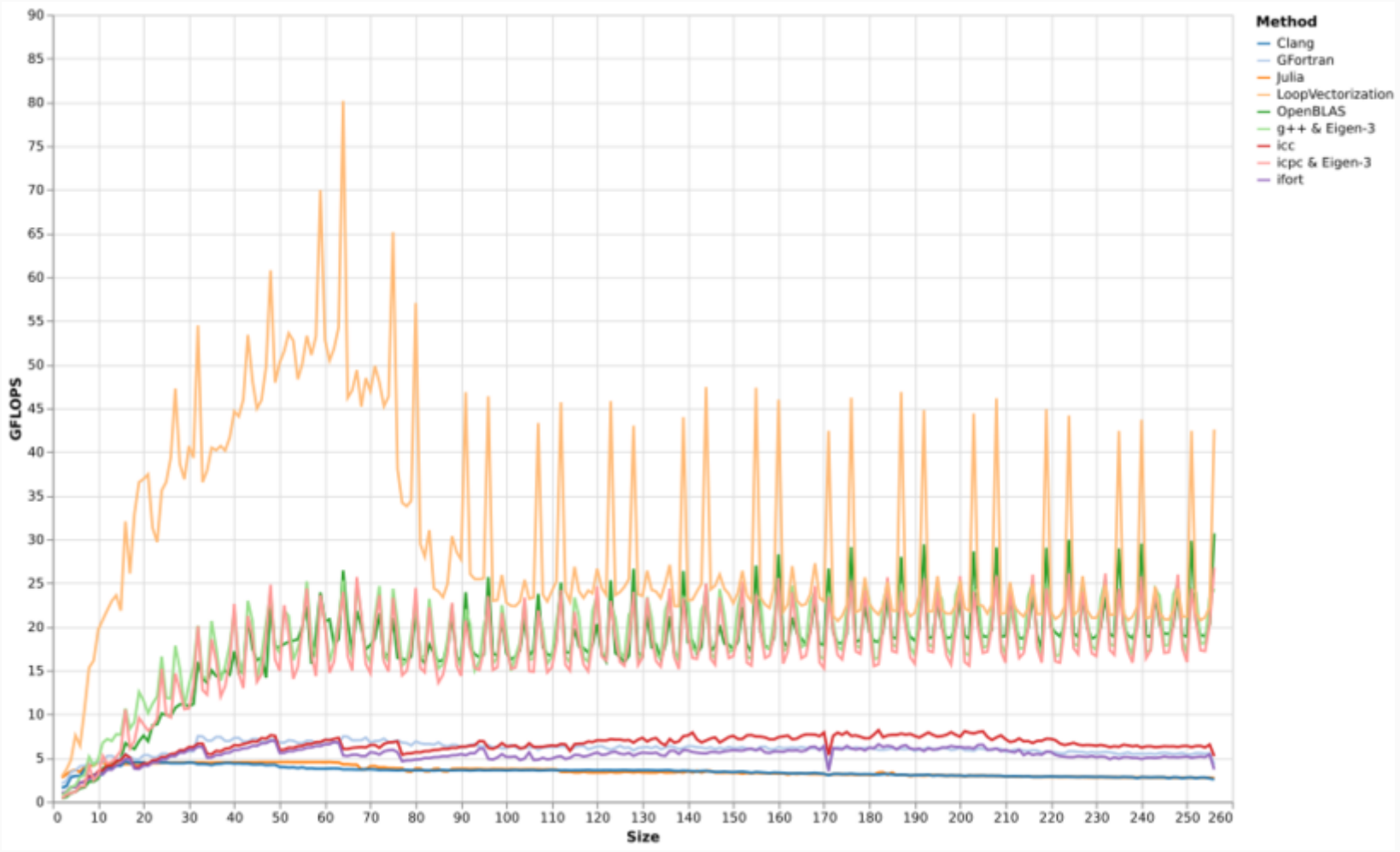
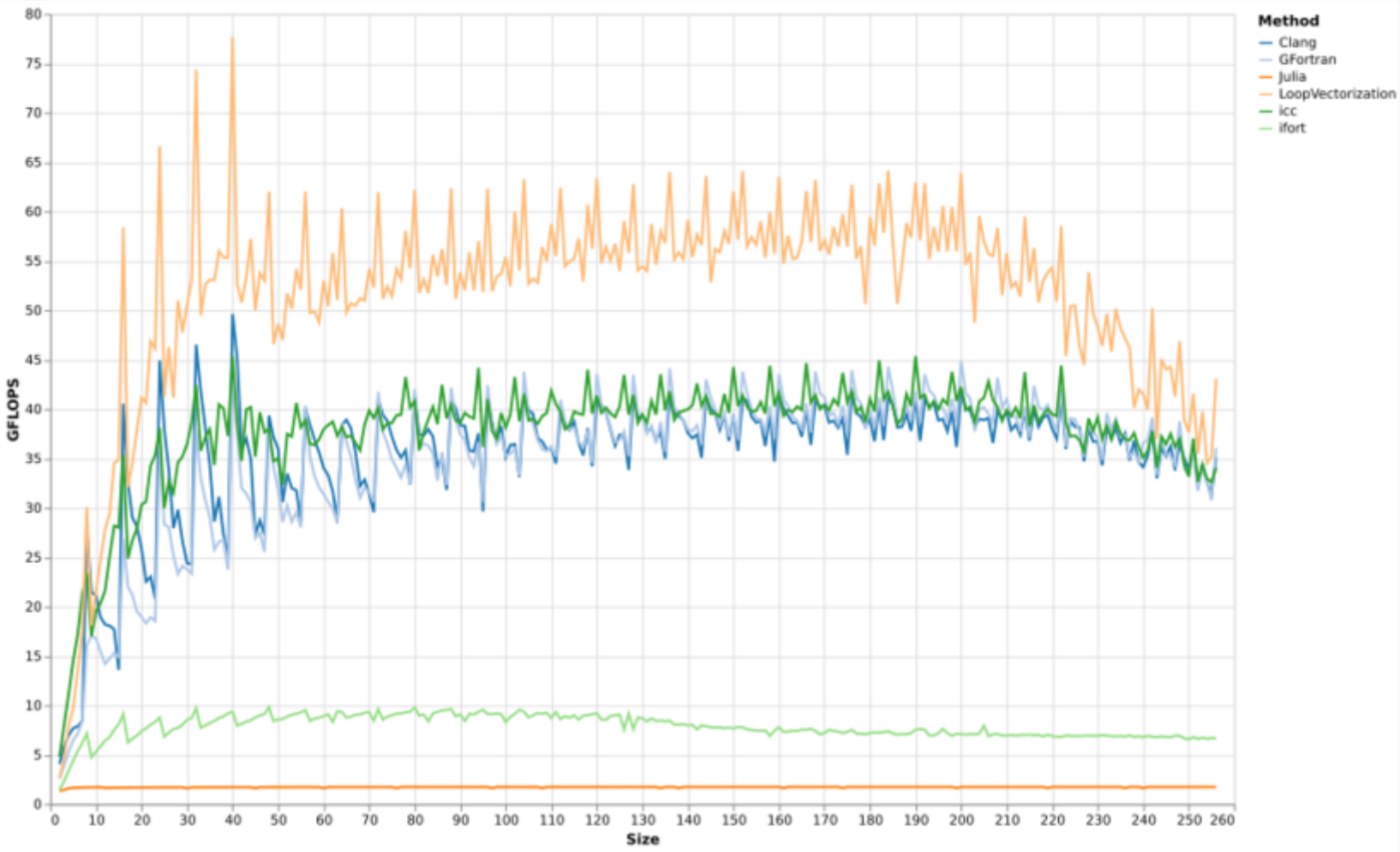
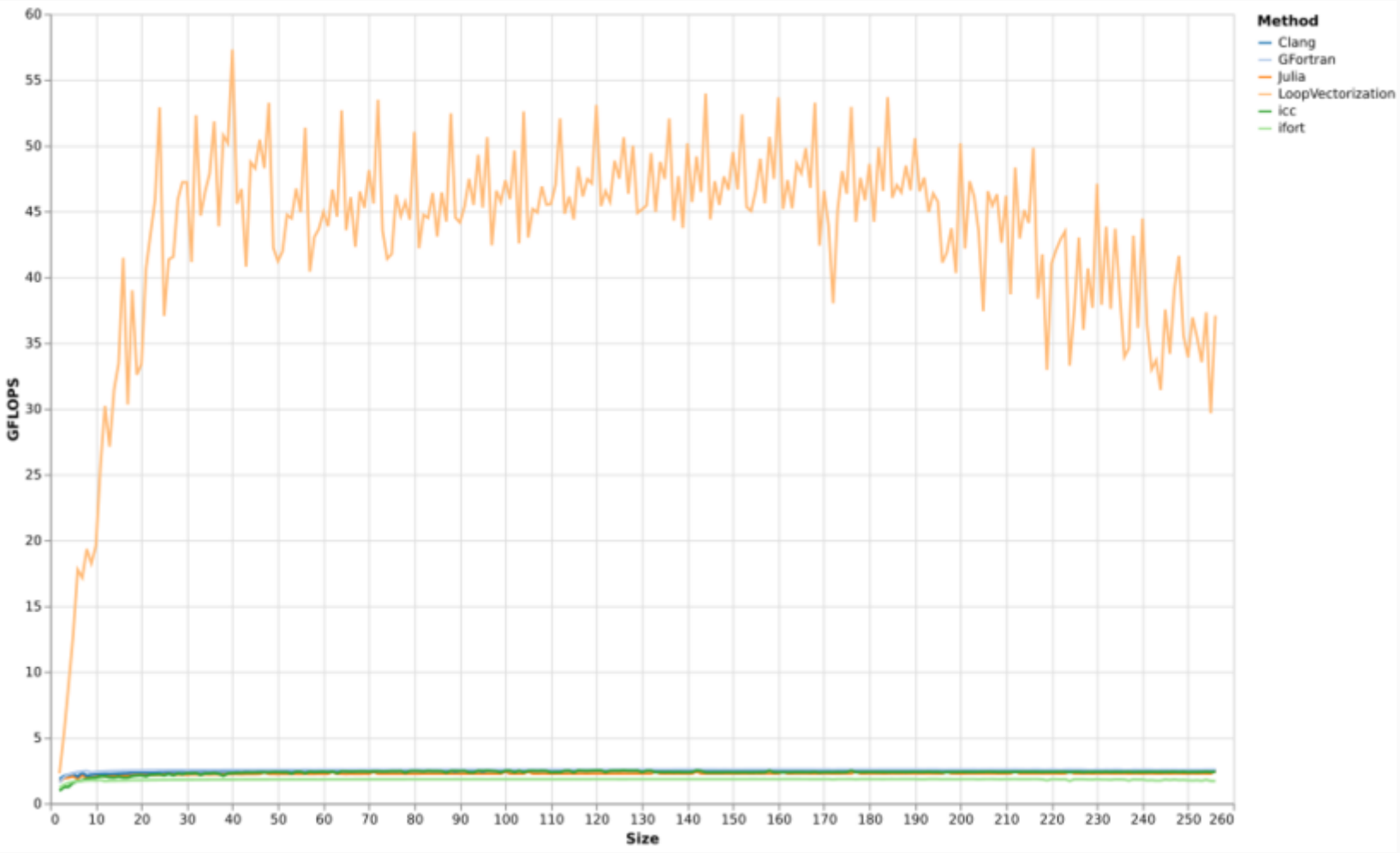


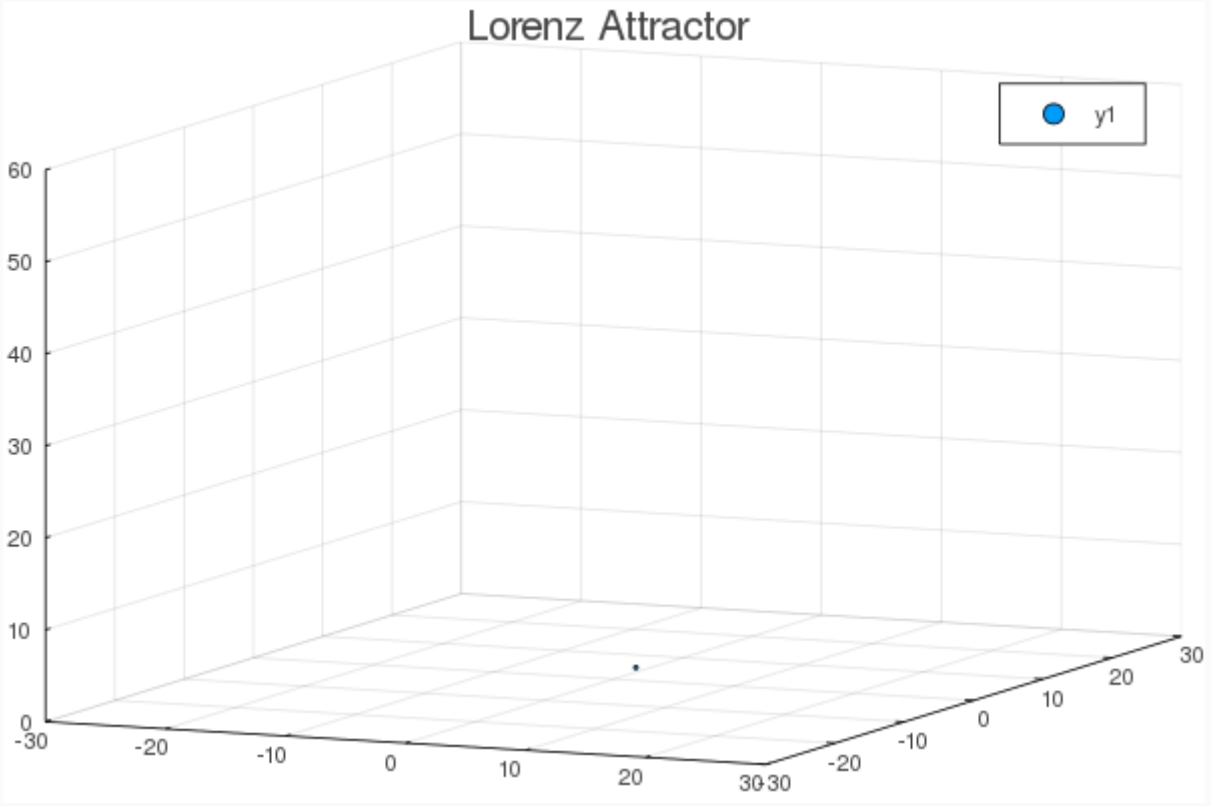
Image kernel convolution



No cheating - unknown kernel size at compile time



It is also pretty!



https://docs.juliaplots.org/

Code walk

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

```
using Plots

# define a structure to gather the Lorenz attractor's parameters
Base.@kwdef mutable struct LorenzAttractor
    x::Float64 = 1;         y::Float64 = 1;         z::Float64 = 1;
    σ::Float64 = 10;        ρ::Float64 = 28;        β::Float64 = 8/3;   dt::Float64 = 0.02
end

l = LorenzAttractor()

function step!(l::LorenzAttractor)
    dx = l.σ * (l.y - l.x); dy = l.x * (l.ρ - l.z) - l.y;  dz = l.x * l.y - l.β * l.z;
    l.x += l.dt * dx;       l.y += l.dt * dy;              l.z += l.dt * dz
end

# initialize a 3D plot with 1 empty series
plt = plot3d(1, title = "Lorenz Attractor", marker = 2,
               xlim = (-30, 30),   ylim = (-30, 30),   zlim = (0, 60))

# build an animated gif by pushing new points to the plot, saving every 10th frame
@gif for i = 1:1500
    step!(l)                            # Calculate the next point
    push!(plt, l.x, l.y, l.z)           # Add that point to the plot
end every 10
```

Generic function / Multiple dispatch

- Generic functions (Common Lisp, R S3, R S4) - Not message passing (e.g. Java, C++, Python, R Reference Classes and S6)
- Multiple dispatch on the entire type signature of the method (Common Lisp, R S4), not just on the first one (e.g. R S3, Python)
- Better at code reuse than message-passing OO:
    - No need to sub-class: no new class, can provide completely different representation.
    - No need to reimplement all methods.
- More natural fit for scientific programming where functions (of any number of parameters) are everywhere.
- Recommended viewing: *The Unreasonable Effectiveness of Multiple Dispatch* by Stefan Karpinski

This is why you can write this

$$a, b \in \mathbb{R}$$
$$A, B \in \mathbb{R}^{m \times n}$$

$$aA + bB$$

Python (and Numpy)

```python
import numpy
A = B = [[1, 2], [3, 4]]

# 2.A + 3.B
numpy.add(numpy.multiply(2, A),
          numpy.multiply(3, B))
```

Julia

```julia
A = B = [1 2; 3 4]
2 * A + 3 * B
```

or better looking (the language specifies how to enter many useful Unicode characters)

```julia
Γ = Λ = [1 2; 3 4]
2 * Γ + 3 * Λ
```

Typing and multiple dispatch example: addition (1/2)

Julia first compiles to LLVM

```
function add_mult(a::Int64, b::Int64)::Int64
  return a + b * 7
end
```

```
## add_mult (generic function with 2 methods)
```

```
@code_llvm debuginfo=:none add_mult(2, 3)
```

```
##
## define i64 @julia_add_mult_17880(i64, i64) {
## top:
##   %2 = mul i64 %1, 7
##   %3 = add i64 %2, %0
##   ret i64 %3
## }
```

```
function add_mult(a::Float64, b::Float64)::Float64
  return a + b * 7.0
end
```

```
## add_mult (generic function with 2 methods)
```

```
@code_llvm debuginfo=:none add_mult(2.0, 3.0)
```

```
##
## define double @julia_add_mult_17901(double, double) {
## top:
##   %2 = fmul double %1, 7.000000e+00
##   %3 = fadd double %2, %0
##   ret double %3
## }
```

Compilation is not Ahead of Time or Just in Time. It is Just Ahead of Time.

Typing and multiple dispatch example: addition (2/2)

Then LLVM to Assembly

```
function add_mult(a::Int64, b::Int64)::Int64
  return a + b * 7
end
```

```
@code_native debuginfo=:none add_mult(2, 3)
```

```
##     .text
##     leaq    (,%rsi,8), %rax
##     subq    %rsi, %rax
##     addq    %rdi, %rax
##     retq
##     nop
```

```
function add_mult(a::Float64, b::Float64)::Float64
  return a + b * 7.0
end
```

```
@code_native debuginfo=:none add_mult(2.0, 3.0)
```

```
##     .text
##     movabsq    $139848117152440, %rax   # imm = 0x7F30ED
##     vmulsd    (%rax), %xmm1, %xmm1
##     vaddsd    %xmm0, %xmm1, %xmm0
##     retq
##     nopw    %cs:(%rax,%rax)
##     nopl    (%rax)
```

Multiple dispatch is used extensively

For example, the `+` function:

```
length(methods(+))
```

```
## 170
```

```
@show first(methods(+))
```

```
## first(methods(+)) = +(x::Bool, z::Complex{Bool}) in Base at complex.jl:286
```

```
## +(x::Bool, z::Complex{Bool}) in Base at complex.jl:286
```

Typing

If we had just defined `function add(x, y) return x + y end`, same result.

But typing catches bugs, the compiler can skip sorting through methods, and yields clearer code.

- Julia has a whole zoo of different types: Primitive, Abstract vs. Concrete, Immutable vs. Mutable Composite, Union, Parametric, Aliases...
- Key points:
  - Dynamic typing when early development, strong typing to catch bugs later.
  - High-level code is easy to express with Abstract and Parametric types:
  - `Int64` is a subtype of `Integer`, and `Float` is a subtype of `AbstractFloat`, which are subtypes of `Number`.
  - You can have `Matrix(Float64)` or `Matrix{Int64}`
- Sub-typing (and generic functions) allows easy algorithmic specialisation and high speed
  - Easy to specialise on `DenseArray` or `SparseArray` for specific algorithms, but default to generic algorithms for others (just an example).

Data science with COVD19

Background

This part is about fitting a model to a set of data. It is not machine learning is the sense of guessing a model that best fits data.

It was born after reading a report from Imperial College London and viewing a forecasting model by NeherLab.

There are 4 types of epidemiological models:

- **Agent-based models** simulate a population that mimics a real population. The disease is then propagated via agents through the simulated population. Stochastic simulation.
- **Machine learning/regression models** are models that learn historical patterns and leverage those patterns for forecasting. This group includes such approaches as statistical time series, linear or regularized regression, clustering, and nonparametric approaches, often characterized by the absence of a mechanistic model.
- **Mechanistic models** are differential-equation model descriptions of the disease transmission mechanism. They include a class of models referred to as compartmental models that partition a population into compartments and mathematically describe how individuals in the population move between compartments.
- **Data-assimilation/dynamic models** usually involve embedding a mechanistic model into a probabilistic framework, allowing for the explicit modeling of the disease transmission process and observational noise with stochastic and/or Bayesian formalism. That is, the dynamic modeling approach combines two sources of uncertainty in the modeling; parametric uncertainty in the mechanistic model and random uncertainty in the observations.

(Only snippets of code included.)

Overview

The **mechanistic** model works as follows:

- susceptible individuals are exposed and infected through contact with contagious individuals. Each contagious individual causes on average R_0 new infections.
- Transmissibility of the virus could have seasonal variation which is parameterised with the parameter "seasonal forcing" (amplitude) and "peak month" (month of most active transmission).

```
# Seasonal forcing parameter ϵ
const ϵ = Dict(:north => 0.2, :tropical => 0.0, :south => 0.2)

# Gives R_0 at a given date
function R₀(d; r₀ = baseR₀, latitude = :north)
    eps = ϵ[latitude]
    peak = peakDate[latitude]

    return r₀ * (1 + eps * cos(2.0 * π * (d - peak) / 365.25))
end
```

- Exposed individuals progress through sequential conditions after an average latency: asymptomatic → symptomatic → severe → critical → death. At each stage, an individual can recover.

Age cohorts

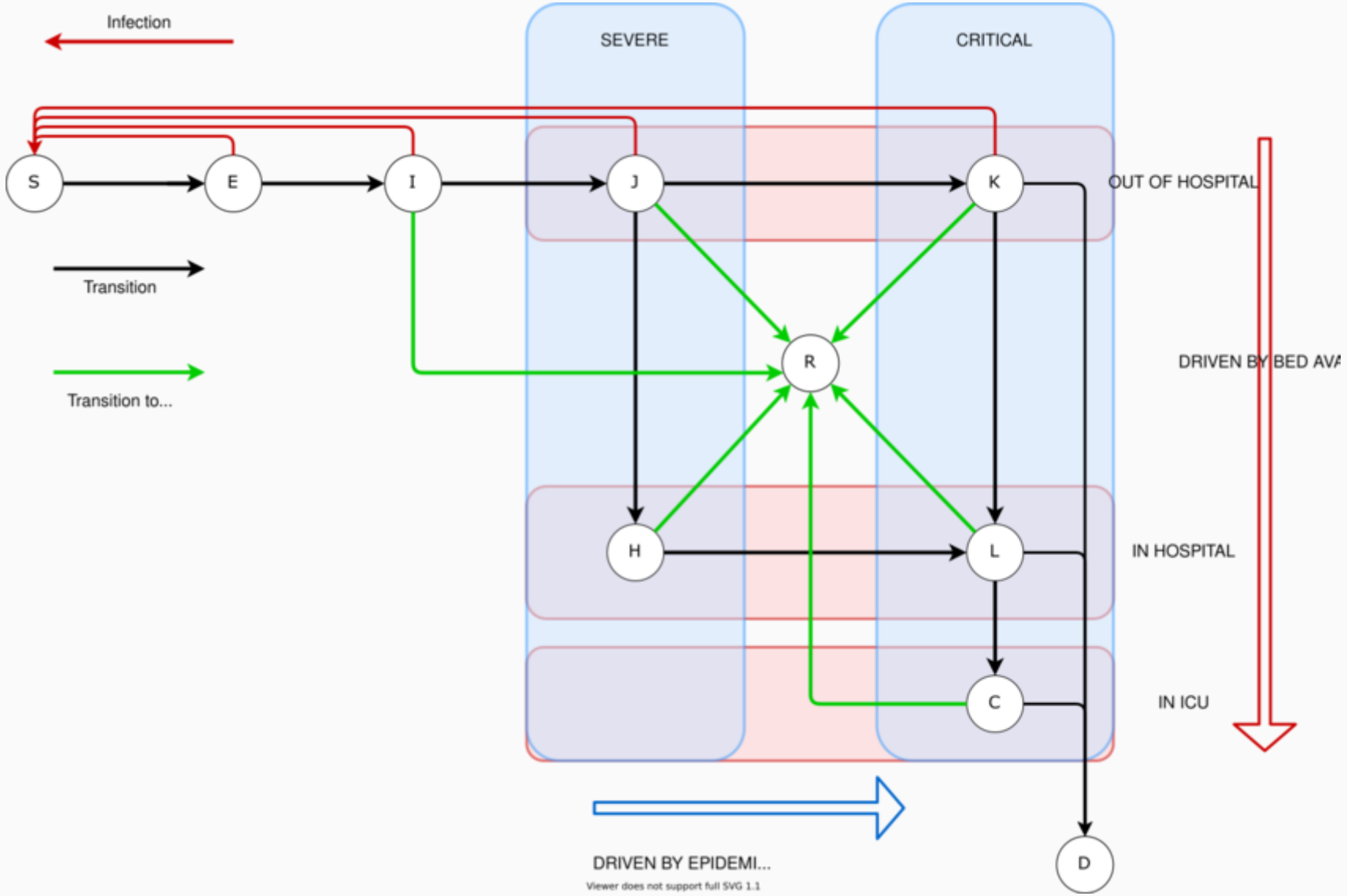
- The proportion/probability of passing from condition to condition depends on age
- How to use what is seen in one country in another country? Age pyramid accounts for a lot of the differences.
- COVID-19 is much more severe in the elderly and proportion of elderly in a community is therefore an important determinant of the overall burden on the health care system and the death toll. We collected age distributions for many countries from data provided by the UN and make those available as input parameters. Furthermore, we use data provided by the epidemiology group by the Chinese CDC to estimate the fraction of severe and fatal cases by age group.

```
#-- Susceptibility to contagion and transition from a compartment to another
const AgeGroup = ["0-9", "10-19", "20-29", "30-39", "40-49", "50-59", "60-69", "70-79", "80+"]
const z_a =       [0.05,   0.05,   0.10,    0.15,    0.20,    0.25,    0.30,    0.40,    0.50]
const m_a =       [0.01,   0.03,   0.03,    0.03,    0.06,    0.10,    0.25,    0.35,    0.50]
const c_a =       [0.05,   0.10,   0.10,    0.15,    0.20,    0.25,    0.35,    0.45,    0.55]
const f_a =       [0.30,   0.30,   0.30,    0.30,    0.30,    0.40,    0.40,    0.50,    0.50]
```

Infrastructure

- Hospital beds and ICU units are limited.
- The probability of a condition becoming more severe is increased if appropriate care is not available: being in critical condition at home vs. in an ICU unit.

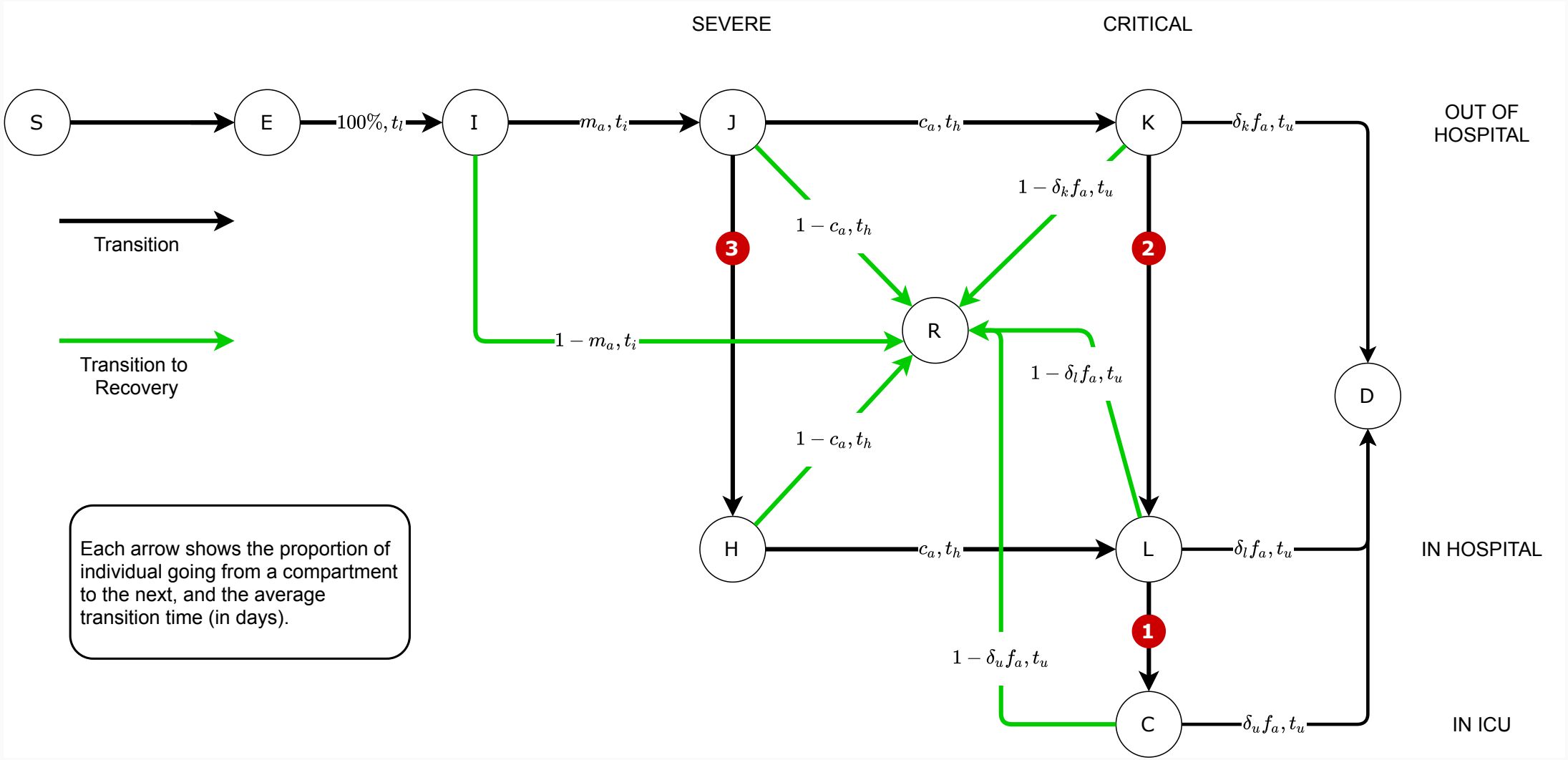
Compartments



Compartments

```
const COMPARTMENTS = [["S",                 "Susceptible"],
                      ["E",                 "Exposed"],
                      ["I",                 "Infectious"],
                      ["J",                 "Severe"],
                      ["H",                 "Hospitalised"],
                      ["C",                 "ICU"],
                      ["R",                 "Recovered"],
                      ["F",                 "Fully_Recovered"],
                      ["D",                 "Dead"],
                      ["K",                 "Critical"],
                      ["L",                 "Critical_Hospitalised"]]

const COMPARTMENTS_LIST = [v[1] for v in COMPARTMENTS]
const COMPARTMENTS_N    = length(COMPARTMENTS_LIST)
```

Transition rates



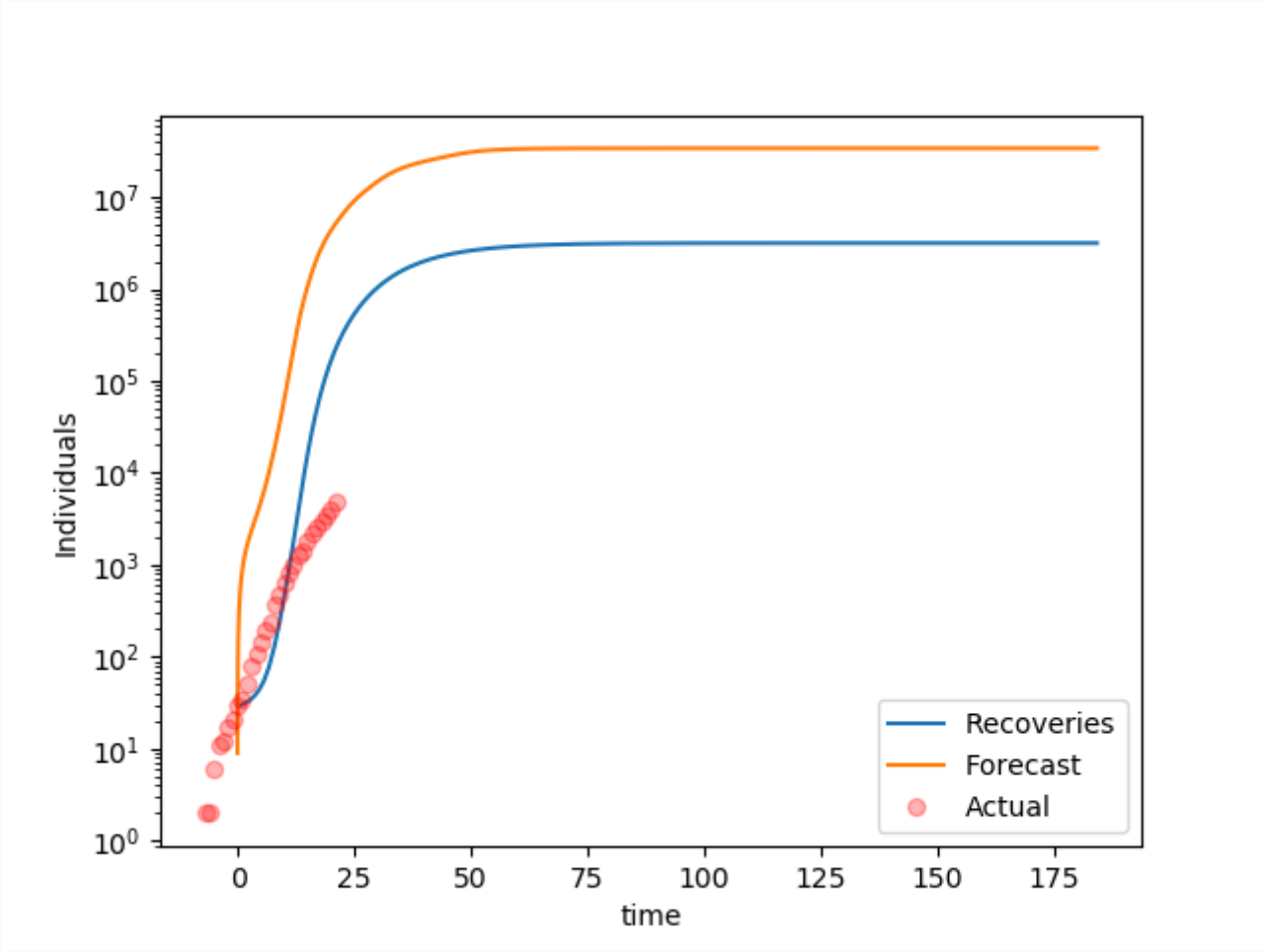
Code: Differential equation for the evolution of the epidemy

```
function epiDynamics!(dP, P, params, t)
    # Destructuring of the compartments with all age groups (P is just one very long vector!)
    c = 0
    S = P[c*nAgeGroup + 1:c*nAgeGroup + nAgeGroup]; c += 1

    ...

    # Destructuring of the individual parameters
    r₀, tₗ, tᵢ, tₕ, tᵤ, tᵣ,..,.  BED_max, ICU_max, Population = params

    ...

    # EI means flow from compartment E to compartment I
    EI = ones(nAgeGroup) .* E / tₗ;  EI = max.(EI, 0.0001);     IE = -EI
    IJ = mₐ              .* I / tᵢ;  IJ = max.(IJ, 0.0001);     JI = -IJ

    ...

    # dS is the decrease of compartment S: how many people are infected.
    dS = - sum(γₑ.*E + γᵢ.*I + γⱼ.*J + γₖ.*K + γᵣ.*R)  / Population .* (S .* β)
    # Exposed
    dE = -dS + IE

    ...

    # Vector change of population and update in place
    result = vcat(dS, dE, dI, dJ, dH, dC, dR, dF, dD, dK, dL)
end
```

ODE solution

```
function calculateSolution(country, diseaseparams, countryparams;
                           finalDate::Union{Nothing, Date} = nothing)

    ...

    ## Compartment vectors of the initial conditions
    S0 = Age_Pyramid .- InfectedAtStart .- InfectiousAtStart .- DeathsAtStart
    E0 = InfectedAtStart
    I0 = InfectiousAtStart
    J0 = 0.0001 .* ones(Float64, nAgeGroup)

    ...

    P0 = vcat(S0, E0, I0, J0, H0, C0, R0, F0, D0, K0, L0)

    # Differential equation solver
    model = ODEProblem(epiDynamics!, P0, tSpan, model_params)

    sol = solve(model, Tsit5(); progress = false)

    return sol
end
```

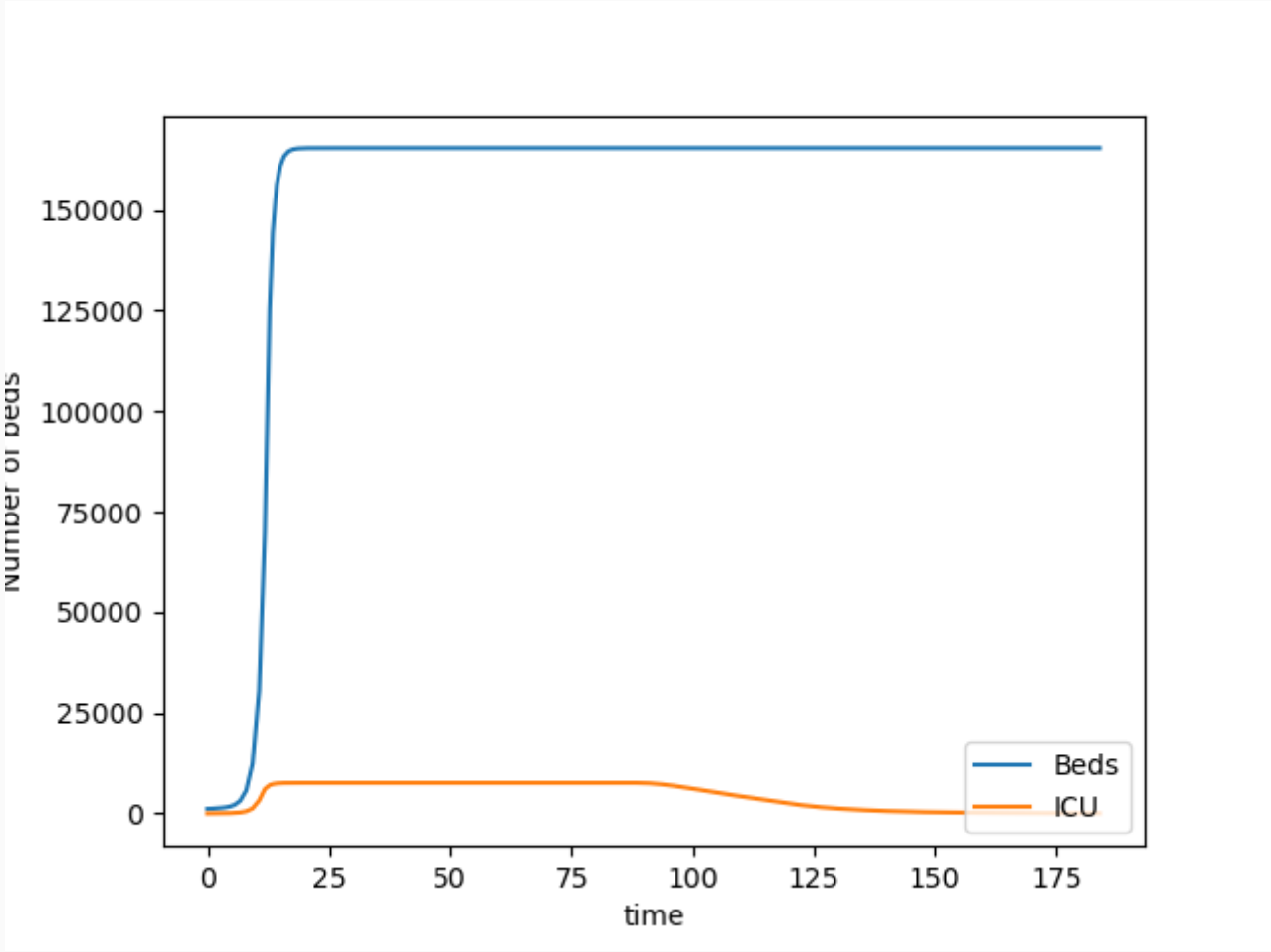
Initial results for Italy



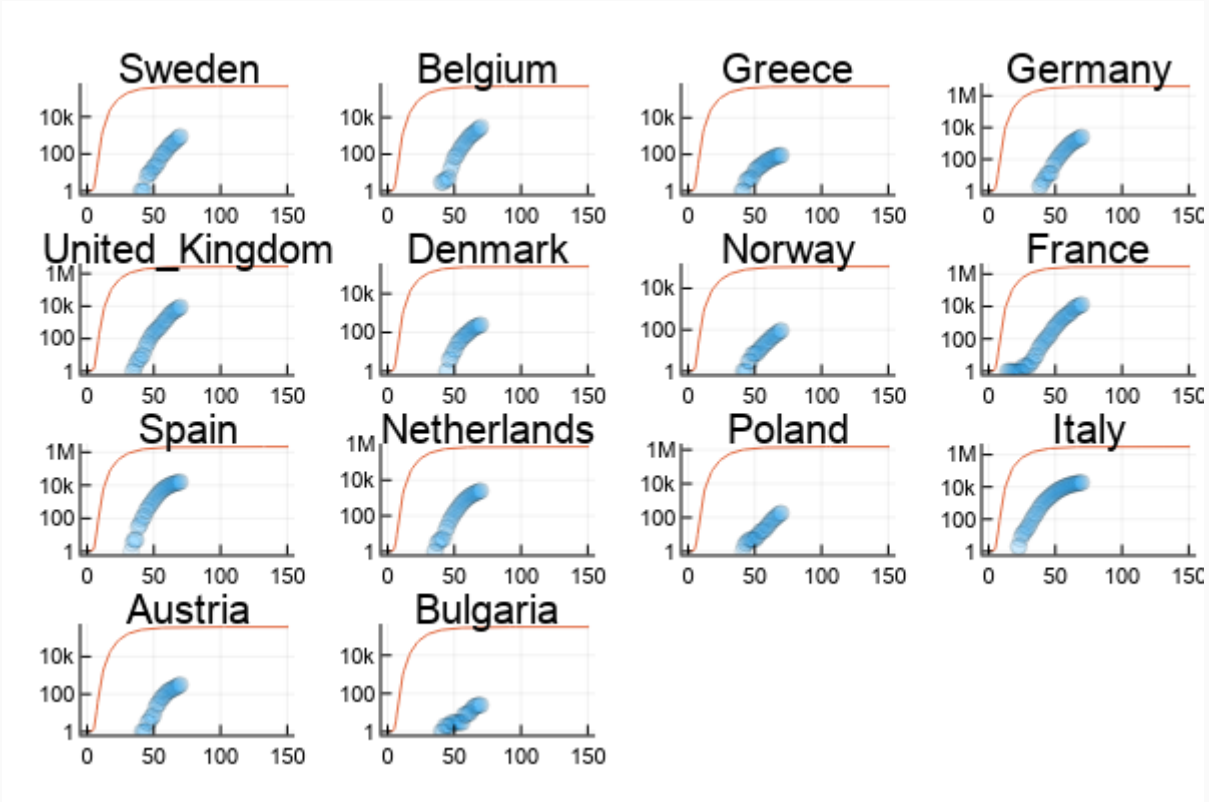
Plot country

```
function plotCountry(country::String; finalDate = Date(2020, 7, 1))
    plotly()

    sol = calculateSolution(country, DiseaseParameters,.......

    p = Plots.plot(title = country)

    xvar = countryData[country][:cases].t
    yvar = countryData[country][:cases].deaths
    p = Plots.scatter!(xvar, yvar, label = "", marker = :circle, markeralpha = 0.30)

    xvar = timeModel2Real.(sol.t, country)
    totalInCompartments = 0.0 .* getSummedCompartment(sol, "S")
    for c in COMPARTMENTS_LIST
        yvar = getSummedCompartment(sol, c)
        totalInCompartments = totalInCompartments .+ yvar

        p = Plots.plot!(xvar, yvar, label = c)
    end

    p = Plots.plot!(xvar, totalInCompartments, label = "Total")

    p = Plots.xaxis!("")
    p = Plots.yaxis!("", :log10)

    return Plots.plot(p)
end
```

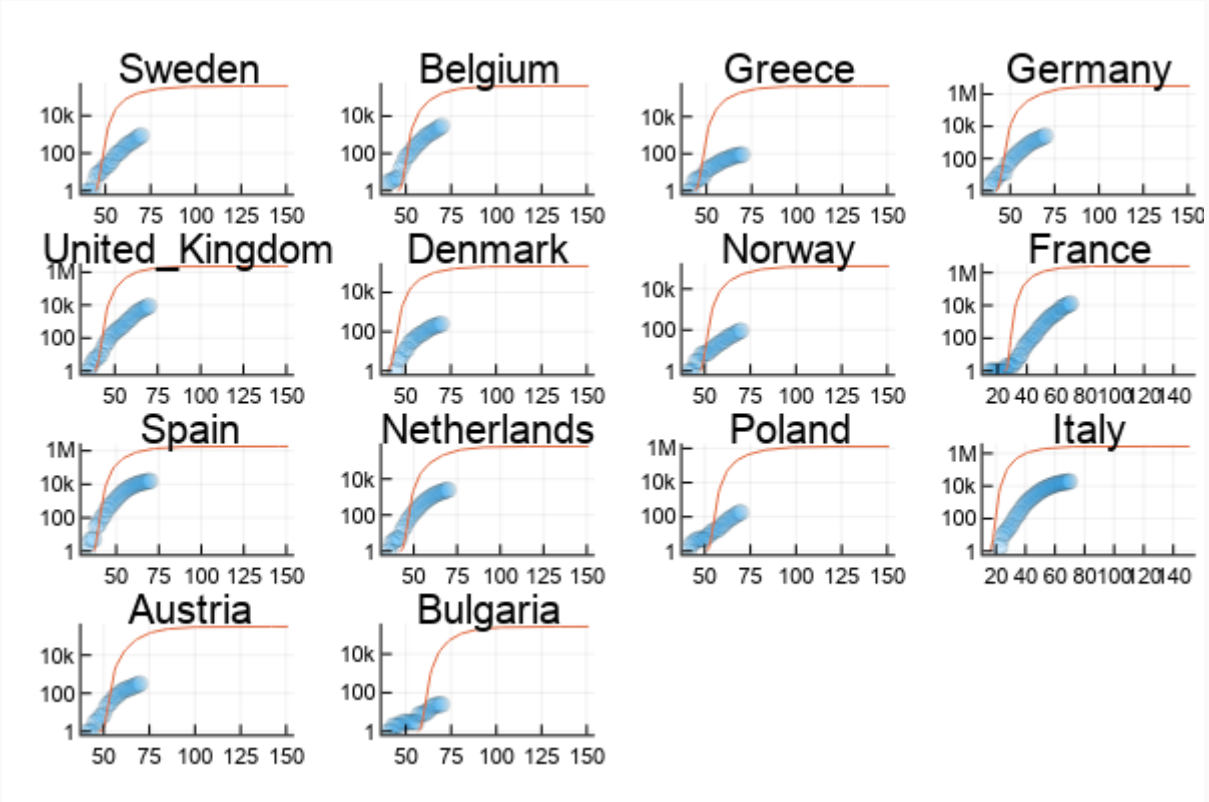
Bed usage


Number of beds
150000
125000
100000
75000
50000
25000
0
0
25
50
75
100
125
150
175
time
Beds
ICU

Before optimisation



After 3 minute optimisation



Optimise every country

```
using BlackBoxOptim

function updateEveryCountry(; maxtime = 60)
    for (country, _) in COUNTRY_LIST

      ...

      # Determine optimal parameters for each countryw
      result = bboptimize(p -> singleCountryLoss(country, DiseaseParameters, p),
                          SearchRange = countryRange;
                          Method = :adaptive_de_rand_1_bin,
                          MaxTime = maxtime,
                          TargetFitness = 2.0,
                          TraceMode = :compact)

      ...

    end
end
```

Limitations

- This is just a model. The map is not the territory. SEIR models are notorious for over-inflating the spread of epidemies.
- Modelling the effect of social measures / different countries, especially their change over time.
- Herd effect.
- More compartments (post-recovery contagion).
- Bayesian approach and probabilistic models account better for incremental information.

The `SciML.jl` library

Scientific Machine Learning Stack

Modern machine learning - Differentiable programming

Describing a model

- `Flux.jl` is a small library (few 1,000's of lines compared to millions for TF and PyTorch) and written in a single language (no Python, C, C++...)
- Provides an easy way to describe neural networks (Keras-style)

Transform it into a system of differential equations

- Deep learning speed improvement have heavily relied on the introduction of automatic differentiation (calculation of a derivative at the cost of a single function call).
- This is normally restricted to the differentiation of mathematic formulas.
- `Zygote.jl` adds algorithmic differentiation: for example `for` loops, `if` statements. This dramatically extends the universe of possibilities.

Solving the equations

- `DiffEqFlux.jl` bridges `DifferentialEquations.jl` and `Flux.jl`: Universal neural differential equations with O(1) backprop, GPUs/TPUs backends, and stiff and non-stiff DE solvers.

Simple code sample (1/2)

Simple NN with 2 dense layers.

```
using Flux

# Define a model of a dense layer
# σ is the activation function.
dense(W, b, σ = identity) = x -> σ.(W * x .+ b)

# For backpropagation, derivatives are calculated
# starting from the end.
#  ∘ denotes function composition
chain(f...) = foldl(∘, reverse(f))

# Easy multilayer perceptron.
# Final model output is the sum of the activations
mlp = chain(
    dense(randn(5, 10), randn(5), tanh),
    dense(randn(2, 5), randn(2)))

# do notation is one way to declare an anonymous (lambda) function.
# Equivalent to: δm = gradient(m -> sum(m(x)), mlp)
x = rand(10)
δm = gradient(mlp) do m
    sum(m(x))
end

# Gradient descent - η: learning rate
m -= η * δm
```

Simple code sample (2/2)

Let's offload that to the GPU with `Cuda`...

```
using CuArrays

cuda() do
    δm = gradient(mlp) do m
        sum(m(x))
    end
end
```

... or use a model defined in `Python`

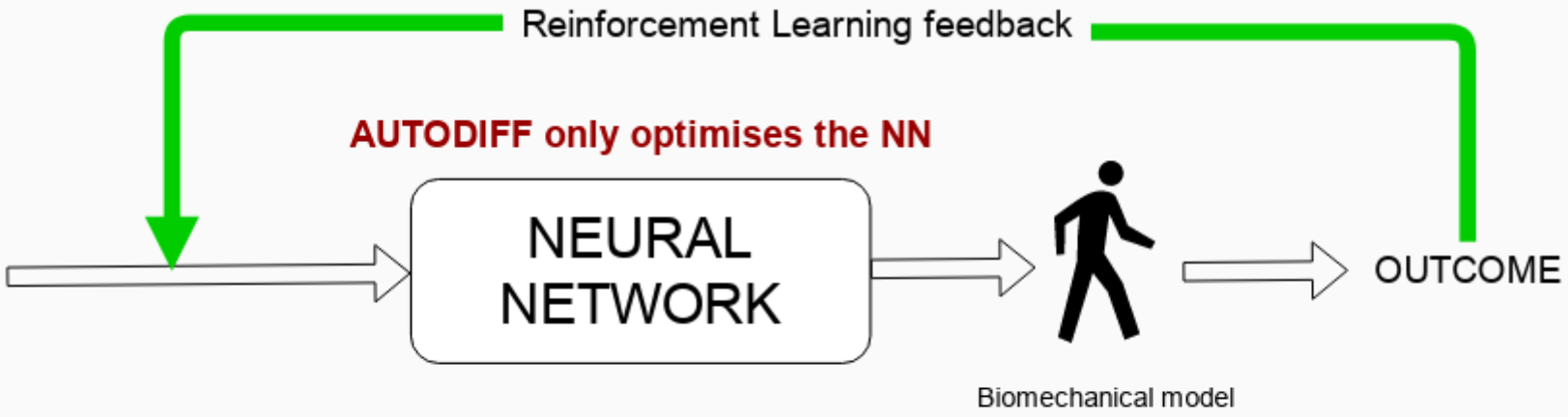
```
using PyCall

py"""
  import torch.nn.functional as F
  def foo(W, b, x):
      return F.sigmoid(W @ x + b)
"""

δW, δb = gradient(W, b) do W, b
    sum( (foo(W, b, x) .- [0.0, 1.0]).^2 )
end
```

Machine Learning is more than neural networks

- A neural network is a conceptually simple formula: optimise $Loss(y - \hat{y})$ for $y = \sigma(W.x + b)$. Autodiff is enough.

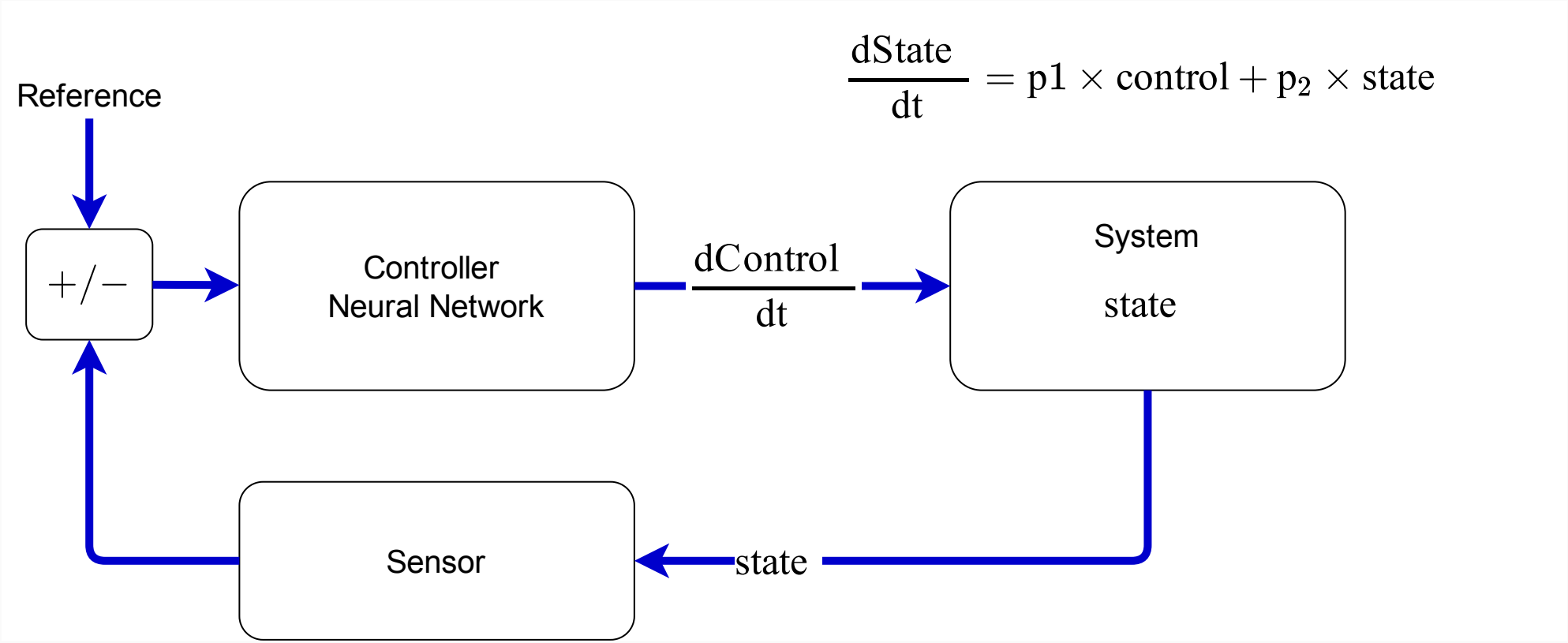
But this doesn't work for richer models:

- Image recognition with detailed physical model of lenses.
- X-Ray images feeding into a pharmacokinetics model to optimise drug dosage.
- Self-driving car with realistic physical model of car given various road conditions.
- Example: Biomechanical model



FLUX can optimise both the NN and the model

Example: Neural network used for optimal control



Harder code sample - using a NN for optimal control (1/2)

```
using DiffEqFlux, Flux, Optim, OrdinaryDiffEq

# Create a NN and create a vector of all its parameters
aNN = FastChain(FastDense(2, 16, tanh), FastDense(16, 16, tanh), FastDense(16, 1))
pNN = initial_params(aNN)

# Parameters for the physical model to control
pModel1 = 0.5; pModel2 = -0.5
pModel = [pModel1; pModel2]

pAll = [pNN; pModel1; pModel2]

function dudt!(du, u, p, t)
    control, state = u
    pnn = p[1:length(pNN)]; pm1 = p[end-1]; pm2 = p[end]

    # Change of control depending on conditions
    dcontrol= ann(u, pnn)[1]

    # Change of the physical model state
    dstate = pm1 * control + pm2 * state

    du[1] = dcontrol
    du[2] = dstate
end

t_span = (0.0, 25.0); t_steps = 0.0:1.0:25.0
control0 = 0.0; state0 = 1.1
mixedProblem = ODEProblem(dudt!, [control0, state0], t_span, pModel)
```

Harder code sample - using a NN for optimal control (2/2)

```
# Global variable containing total world/model state
θ = [state0; pAll]

function predict_adjoint(θ)
  state = θ[1]
  p     = θ[2:end]
  solution = Array(concrete_solve(mixedProblem, Tsit5(), [0, state], p, saveat = t_steps))
  return solution[2, :]  # Only interested in state, not control
end

REFERENCE_VALUE = 1.0
loss_adjoint(θ) = sum(abs2, predict_adjoint(θ) .- REFERENCE_VALUE)

result = DiffEqFlux.sciml_train(loss_adjoint, θ, BFGS(initial_stepnorm = 0.01))
```

Optimisation output

```
Status: success

Candidate solution
 Minimizer: [1.00e+00, 4.33e-02, 3.72e-01,  ...]
 Minimum:   6.572520e-13

Found with
 Algorithm:     BFGS
 Initial Point: [1.10e+00, 4.18e-02, 3.64e-01,  ...]

Convergence measures
 |x - x'|               = 0.00e+00 ≤ 0.0e+00
 |x - x'|/|x'|          = 0.00e+00 ≤ 0.0e+00
 |f(x) - f(x')|         = 0.00e+00 ≤ 0.0e+00
 |f(x) - f(x')|/|f(x')| = 0.00e+00 ≤ 0.0e+00
 |g(x)|                 = 5.45e-06 ≰ 1.0e-08

Work counters
 Seconds run:   8  (vs limit Inf)
 Iterations:    23
 f(x) calls:    172
 ∇f(x) calls:   172
```

Metalhead

For computer vision models, see `Metalhead.jl`

```
using Metalhead
using Metalhead: classify

vgg = VGG19()
img = "Dog.jpg"


vgg.layers

# Chain(Conv((3, 3), 3=>64, relu),
#       Conv((3, 3), 64=>64, relu),
#       MaxPool((2, 2), pad = (0, 0, 0, 0),
#       stride = (2, 2)),
#       .....
#       Dense(25088, 4096, relu), Dropout(0.5),
#       Dense(4096, 4096, relu), Dropout(0.5),
#       Dense(4096, 1000), softmax))
```

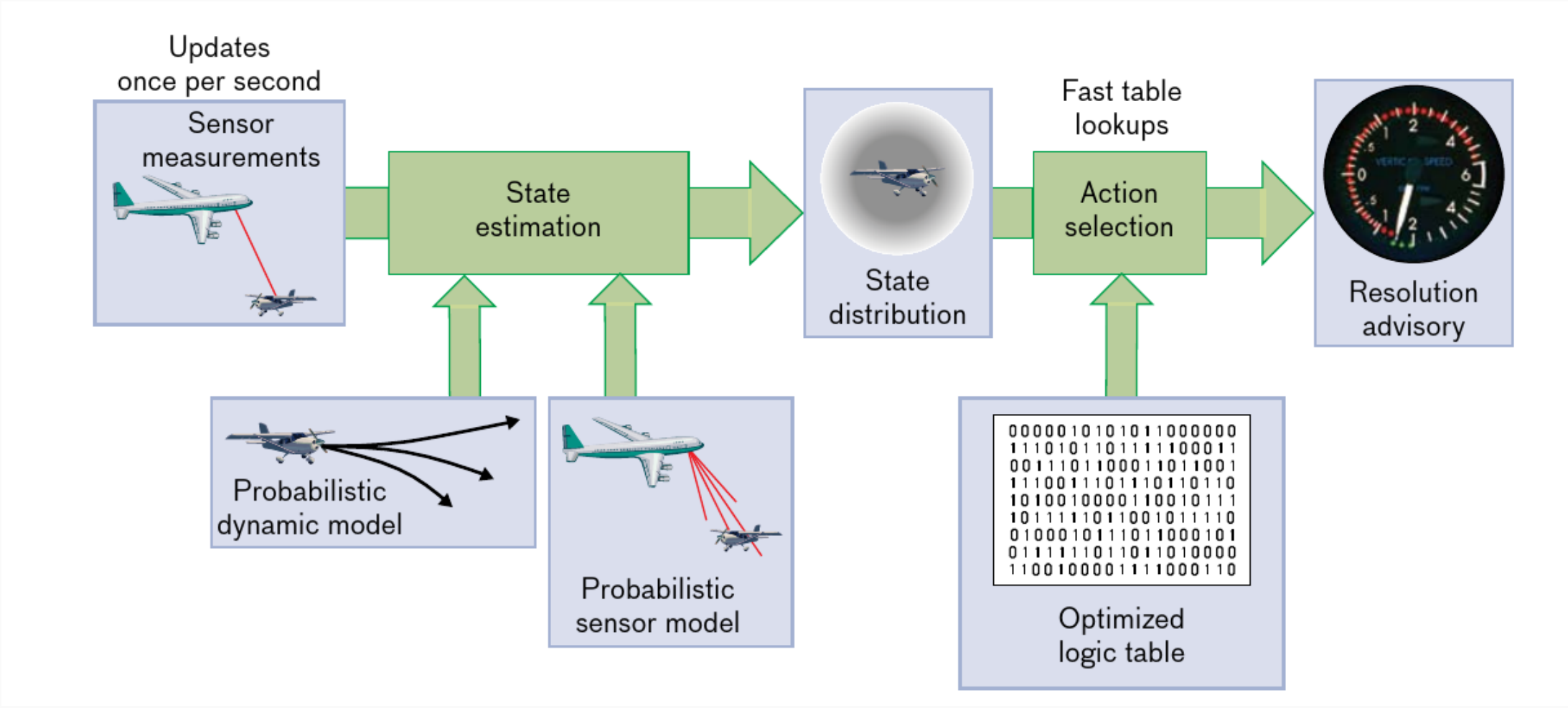
(This is a random Internet dog...)

```
Result: "Labrador retriever
```

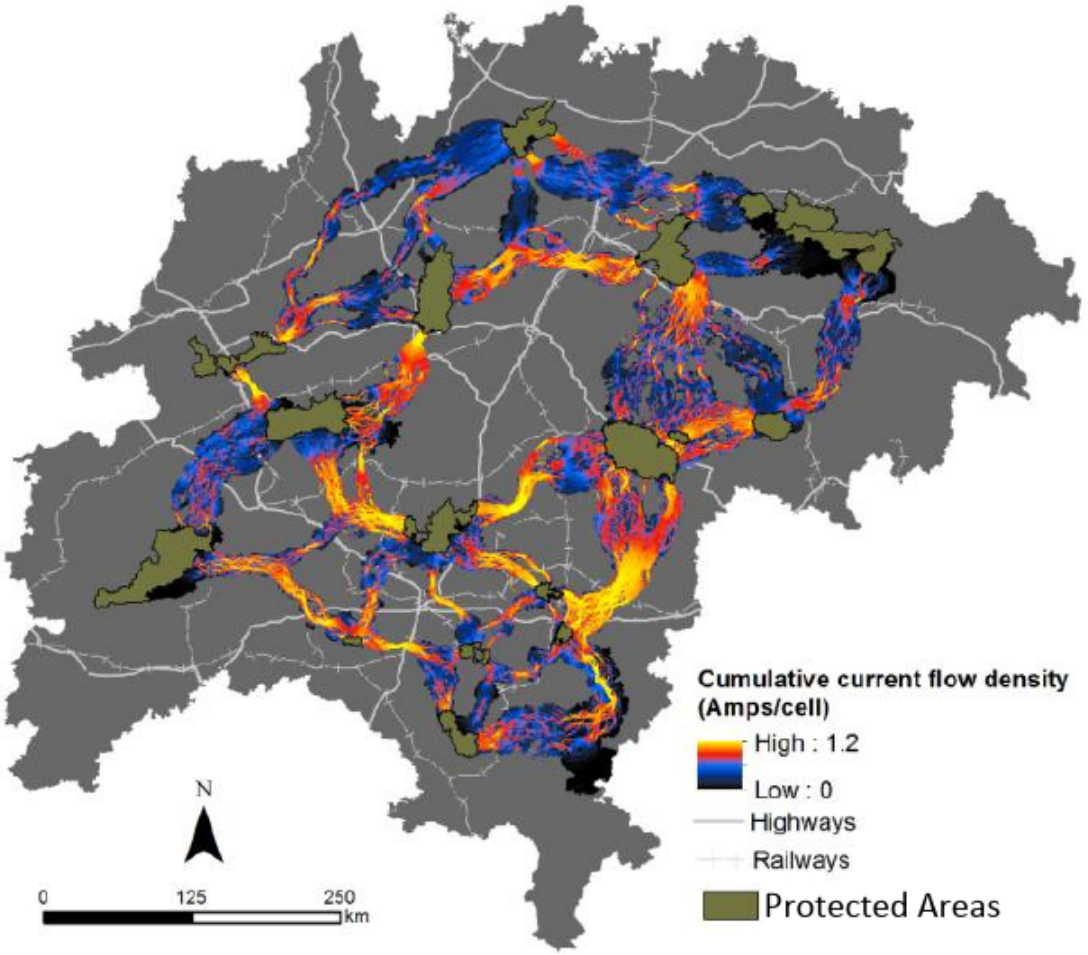
From now on, you will think about Julia when...

... you look at the sky: *Approximate inference for constructing astronomical catalogs from images*

… you are in a plane: *Next-Generation Airborne Collision Avoidance System*



… see migrating species: *Circuitscape: modeling landscape connectivity to promote conservation and human health*



Dutta et al. (2015) combined Circuitscape with least-cost corridor methods to map pinch points within corridors connecting protected areas for tigers in central India. Areas with high current flow are most important for tiger movements and keeping the network connected.

Thanks and links

Julia: https://julialang.org ; Youtube channel: https://www.youtube.com/user/JuliaLanguage ; SciML: https://sciml.ai/

LoopVectorization: https://chriselrod.github.io/LoopVectorization.jl/latest/ ; Differentiable programming https://fluxml.ai/2019/02/07/what-is-differentiable-programming.html

NeherLab: https://neherlab.org/covid19/ ; Julia case studies: https://juliacomputing.com/case-studies/

References:

JuliaCon 2019 | The Unreasonable Effectiveness of Multiple Dispatch | Stefan Karpinski https://www.youtube.com/watch?v=kc9HwsxE1OY

Osthus, D., Gattiker, J., Priedhorsky, R., & Del Valle, S. Y. (2019). *Dynamic Bayesian influenza forecasting in the United States with hierarchical discrepancy (with discussion)*. Bayesian Analysis, 14(1), 261-312. https://arxiv.org/abs/1708.09481

Introduction to SEIR Models by Nakul Chitnis, Workshop on Mathematical Models of Climate Variability, Environmental Change and Infectious Diseases http://indico.ictp.it/event/7960/session/3/contribution/19/material/slides/0.pdf

---

Code written in **Atom** with the **JunoLab** extension ; Slides written in **RStudio** in R Markdown) and formatted with the **Xaringan**) package.