An introduction to

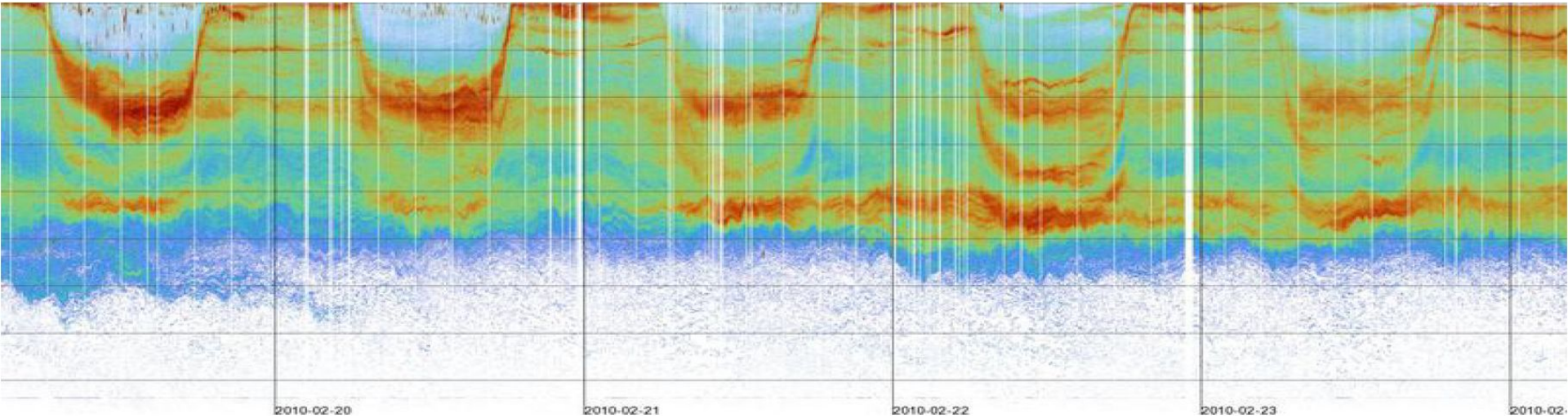
for quantitative ecologists

Sam Urmy, PhD
Research Fish Biologist, NOAA AFSC
SAFS Quantitative Seminar, May 7 2021

0. Introduction

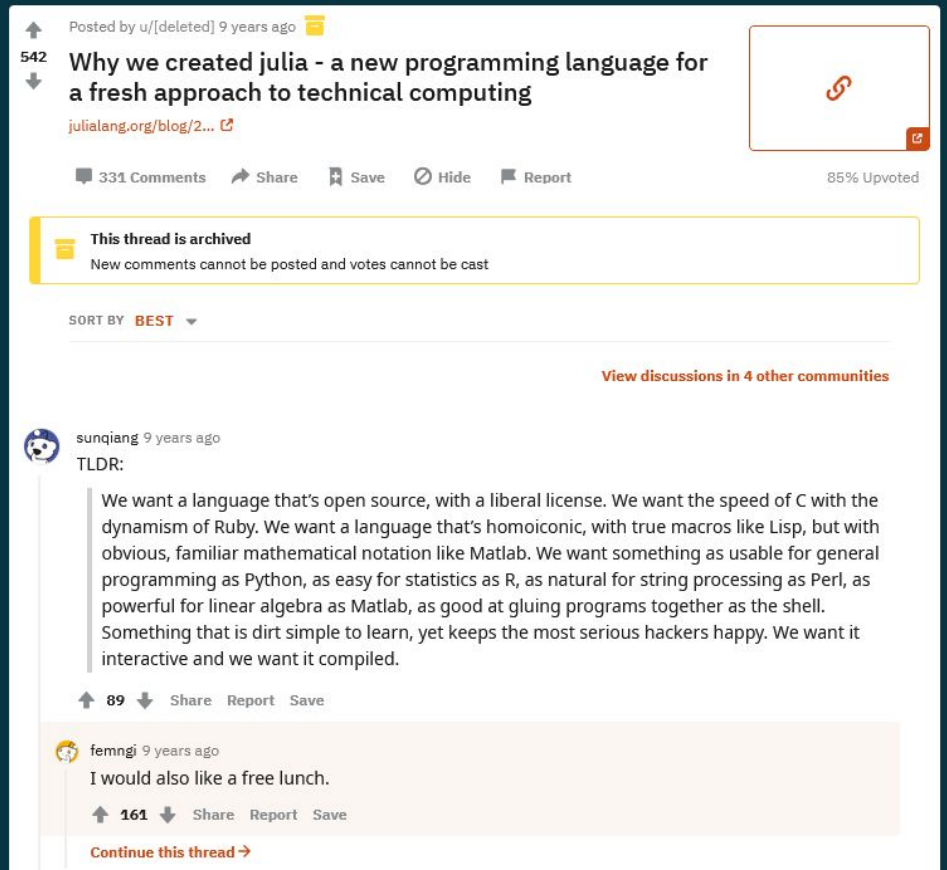
Nobody expects to become a programmer when they sign up to be a marine biologist...

- Learned Python in college, R at start of grad school



- Processing 18 months of high-resolution acoustic data
- Implementing custom algorithms and summary metrics
- Using Python and R, it was S L O W going

Three weeks after my MS defense:



Posted by u/[deleted] 9 years ago

542

Why we created julia - a new programming language for a fresh approach to technical computing

julialang.org/blog/2...

331 Comments Share Save Hide Report 85% Upvoted

This thread is archived

New comments cannot be posted and votes cannot be cast

SORT BY BEST

View discussions in 4 other communities

sunqiang 9 years ago

TLDR:

> 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.

89 Share Report Save

femngi 9 years ago

I would also like a free lunch.

161 Share Report Save

Continue this thread →

We are power Matlab users...Some are Pythonistas, others Rubyists, still others Perl hackers...We've generated more R plots than any sane person should. C is our desert island programming language.

We love all of these languages; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

We are greedy: we want more.

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 powerful for linear algebra as Matlab...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?)

89 Share Report Save

femngi 9 years ago

I would also like a free lunch.

161 Share Report Save

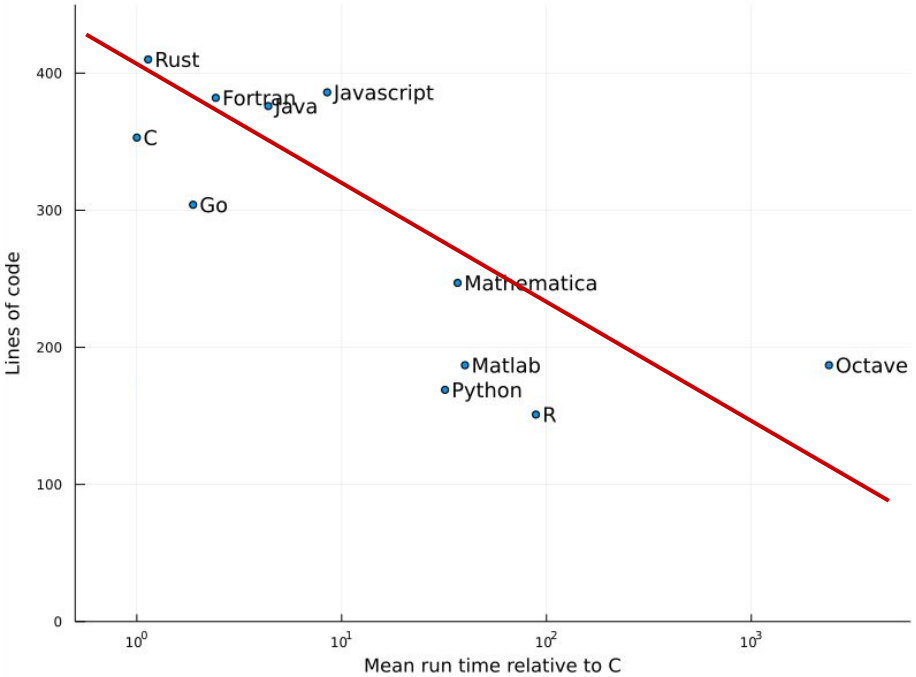
Continue this thread →

What is Julia?

- “A fresh approach to technical computing”
- Free and open-source
- Developed initially by a group at MIT
- Dynamic, optionally typed, just-in-time compiled
- Designed to solve the “two-language problem” in scientific computing

The two-language problem

Compiled, statically typed



Interactive, dynamically typed

https://github.com/JuliaLang/Microbenchmarks

Two language problem

- Write program in R/Matlab/Python
- Find slowest parts
- Rewrite those in C/C++/Fortran
- Call low-level program from high-level program

“Whatever, I just *use* the packages, who cares about this.”

If you have ever...

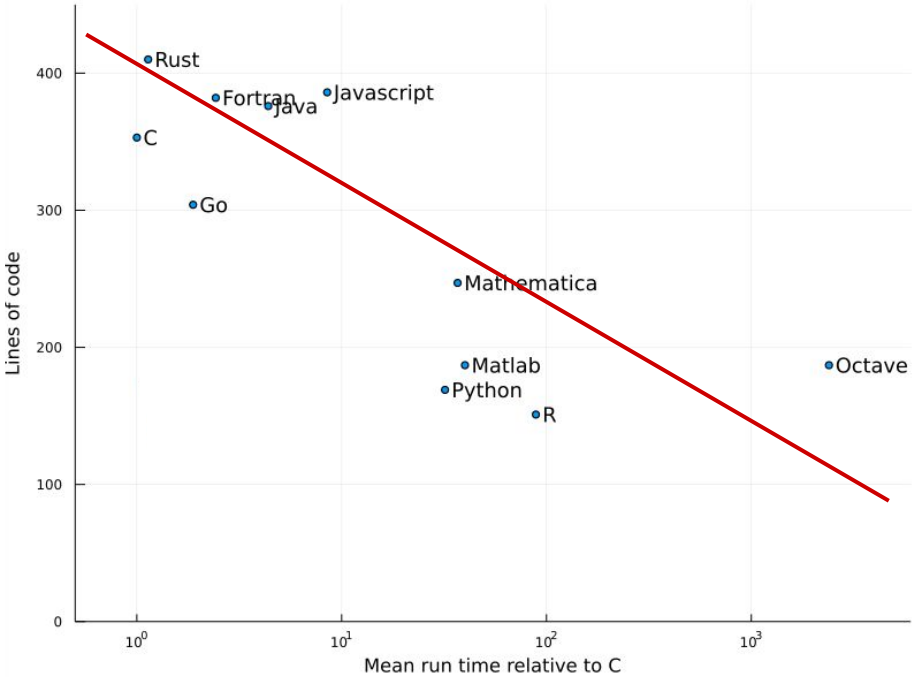
- Used a convoluted series of `sapply`'s and/or indexing tricks to avoid writing a simple `for`-loop…
- Struggled to learn (or remember) the syntax for JAGS, Stan, or TMB…
- Been stymied by an unintelligible C, C++, or Fortran error…
- Had to install `gcc` on Windows to build an R package…
- Wondered if you can *actually* trust the algorithm in the package you're using...
- Asked which part of your model is causing it to take days to converge…
- Wished you could get that model to converge in hours or (dare we even dream...) minutes???

...then YOU have suffered from the

two-language problem!

The two-language problem

Compiled, statically typed



Interactive, dynamically typed

https://github.com/JuliaLang/Microbenchmarks

Demo: hello, world!

My history programming in Julia

- 2012: See post on Reddit, download v0.1
- 2013-2014: Play around, use Julia in a couple of classes
- 2015: Start first package for my own use
- 2016: Write significant simulation model for my PhD research
- 2018: Julia v1.0 released. Using Julia 80% of the time
- 2021: Julia at v1.6
  - Using Julia almost exclusively
  - Ranting about it to anyone who will listen

But it’s not just me!

nature

View all journals Search Login

Explore content Journal information Publish with us Subscribe Sign up for alerts RSS feed

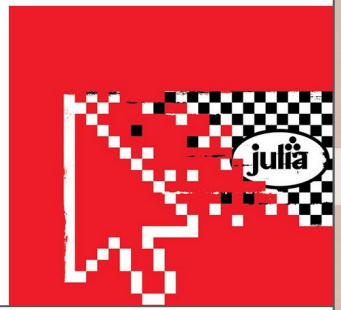
nature › toolbox › article

TOOLBOX · 30 JULY 2019

Julia: come for the syntax, stay for the speed

Researchers often find themselves coding algorithms in one programming language, only to have to rewrite them in a faster one. An up-and-coming language could be the answer.

Jeffrey M. Perkel



ars TECHNICA

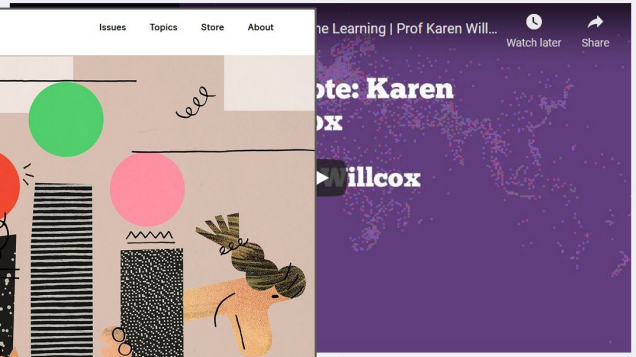
BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE FORUMS SUBSCRIBE SIGN IN

LOOKS LIKE MATH —

The unreasonable effectiveness of the Julia programming language

Fortran has ruled scientific computing, but Julia emerged for large-scale numerical work.

LEE PHILLIPS - 10/9/2020, 4:15 AM



party

d excited scientists lately. "Running into" in the virtual her opportunities to collide with scientists in meatspace ntists believe in the germ theory of disease.

icians are excited about a new tool. It's not a new particle ad, this exciting new tool for scientific research is... a

ARS VIDEO

IBM Research: Phase Change Memory

NEW Buy the print edition

Issues Topics Store About

CHRIS STOKEL-WALKER

Julia: The Goldilocks language

By blending speed and high-level functionality in a language for technical computing, Julia was designed to be just right.

PART OF ISSUE 5 APR 2018 Programming Languages

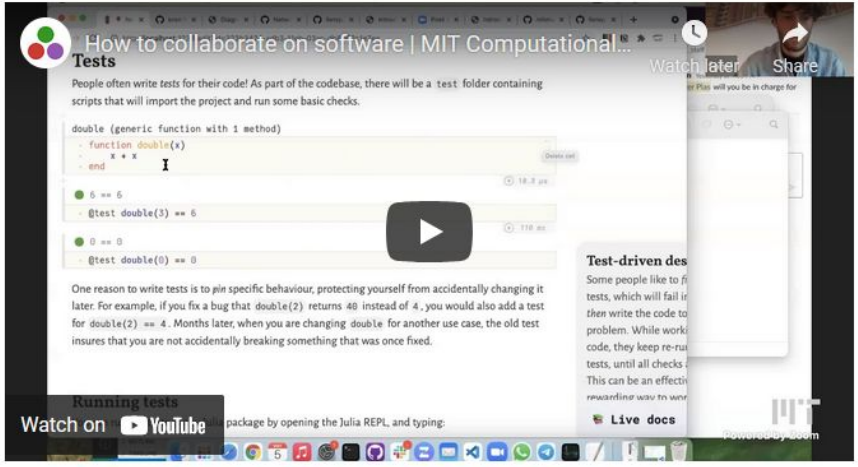
Introduction to Computational Thinking

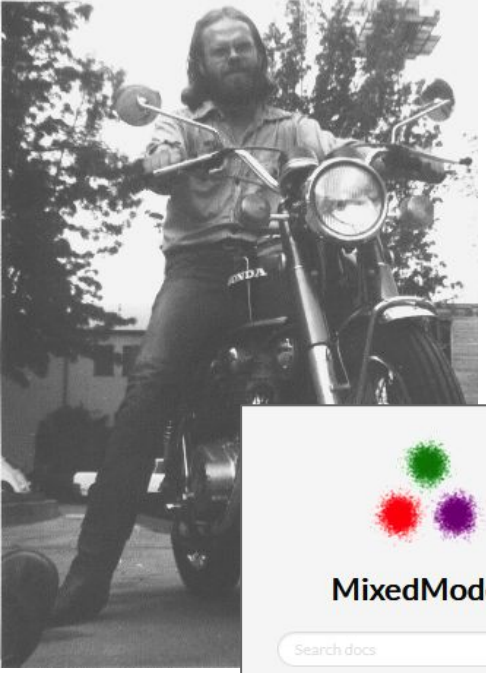
Welcome to **MIT 18.S191 aka 6.S083 aka 22.S092, Spring 2021** edition!

> ***Announcement:*** *Next class is Wednesday, May 5th, 1:05pm EDT*

> *Today's Lecture Notebook.*

Currently live





Package ‘lme4’

December 1, 2020

Version 1.1-26

Title Linear Mixed-Effects Models using 'Eigen' and S4

Contact LME4 Authors <lme4-authors@lists.r-forge.r-project.org>

Description Fit linear and generalized linear mixed-effects models.
The models and their components are represented using S4 classes and

MixedModels

Search docs

- MixedModels.jl Documentation
- Model constructors
- Details of the parameter estimation
- Normalized Gauss-Hermite Quadrature
- Parametric bootstrap for mixed-effects models
- Rank deficiency in mixed-effects models

MixedModels.jl Documentation

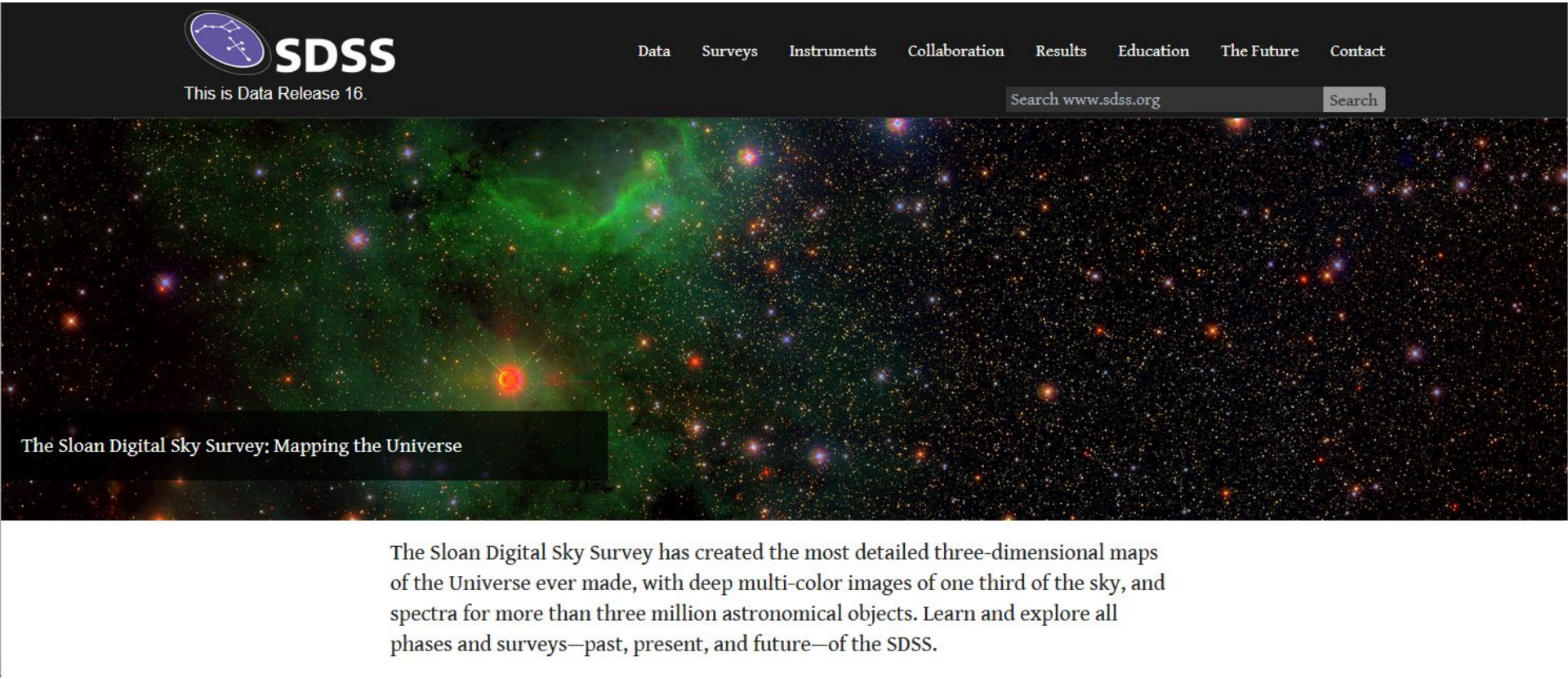
Edit on GitHub

MixedModels.jl Documentation

MixedModels.jl is a Julia package providing capabilities for fitting and examining linear and generalized linear mixed-effect models. It is similar in scope to the *lme4* package for `R`.

- Model constructors
  - Examples of linear mixed-effects model fits
  - Fitting generalized linear mixed models
- Extractor functions
  - Model-fit statistics
  - Fixed-effects parameter estimates
  - Covariance parameter estimates
  - Conditional modes of the random effects
  - Case-wise diagnostics and residual degrees of freedom
- Details of the parameter estimation

Sloan Digital Sky Survey



Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnany†, Keno Fischer‡, Andreas Noack§, Maximilian Lam*, Jarrett Revels§, Steve Howard¶, Ryan Giordano¶, David Schlegel‖, Jon McAuliffe¶, Rollin Thomas‖, Prabhat‖

*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
‡Julia Computing
§Computer Science and AI Laboratories, Massachusetts Institute of Technology
¶Department of Statistics, University of California, Berkeley
‖Lawrence Berkeley National Laboratory

***Abstract*—Astronomical catalogs derived from wide-field imaging surveys are an important tool for understanding the Universe. We construct an astronomical catalog from 55 TB of imaging data using Celeste, a Bayesian variational inference code written entirely in the high-productivity programming language Julia. Using over 1.3 million threads on 650,000 Intel Xeon Phi cores of the Cori Phase II supercomputer, Celeste achieves a peak rate of 1.54 DP PFLOP/s. Celeste is able to jointly optimize parameters for 188M stars and galaxies, loading and processing 178 TB across 8192 nodes in 14.6 minutes. To achieve this, Celeste exploits parallelism at multiple levels (cluster, node, and thread) and accelerates I/O through Cori's Burst Buffer. Julia's native performance enables Celeste to employ high-level constructs without resorting to hand-written or generated low-level code (C/C++/Fortran), and yet achieve petascale performance.**

***Keywords*-astronomy, Bayesian, variational inference, Julia, high performance computing**

I. Introduction

Modern astronomical surveys produce vast amounts of data. Our work uses images from the Sloan Digital Sky Survey (SDSS) [1] as a test case to demonstrate a new, highly scalable algorithm for constructing astronomical catalogs. SDSS produced 55 TB of images covering 35% of the sky and a catalog of 470 million unique light sources. Figure 1 shows image boundaries for a small region from SDSS. Advances in detector fabrication technology, computing power, and networking have enabled the development of a new generation of upcoming wide-field sky surveys orders of magnitude larger than SDSS. In 2020, the Large Synoptic Survey Telescope (LSST) will begin to obtain more than 15 TB of new images nightly [2]. The instrument will produce 10s–100s of PBs of imaging and catalog data over the lifetime of the project.

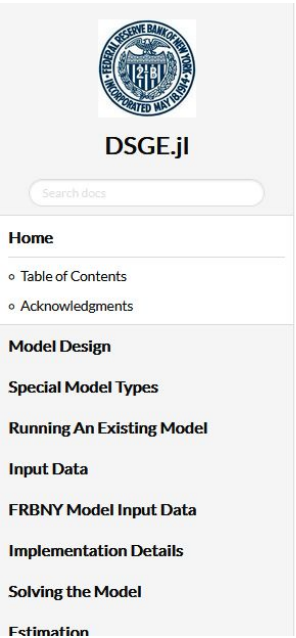
Producing an astronomical catalog is challenging in part because the parameters of overlapping light sources must be learned collectively: the optimal parameters for one light

JULIA LANGUAGE DELIVERS PETASCALE HPC PERFORMANCE

November 28, 2017 Rob Farber

Written in the productivity language **Julia**, the **Celeste project**—which aims to catalogue all of the telescope data for the stars and galaxies in in the visible universe—demonstrated the first Julia application to exceed 1 PF/s of double-precision floating-point performance (specifically 1.54 PF/s).

Federal Reserve Bank of New York



You may wonder why we wrote our code, which was originally in MATLAB and made available here, in Julia. MATLAB is a widely used, mature programming language that has served our purposes very well for many years. However, Julia has two main advantages from our perspective. First, as free software, Julia is more accessible to users from academic institutions or organizations without the resources for purchasing a license. Now anyone, from Kathmandu to Timbuktu, can run our code at no cost. Second, as the models that we use for forecasting and policy analysis grow more complicated, we need a language that can perform computations at a high speed. Julia boasts performance as fast as that of languages like C or Fortran, and is still simple to learn. (Read this post, written by the creators of the language, to understand why Julia fits the bill.) We want to address hard questions with our models—from understanding financial markets developments to modeling households' heterogeneity—and we can do so only if we are close to the frontier of programming.

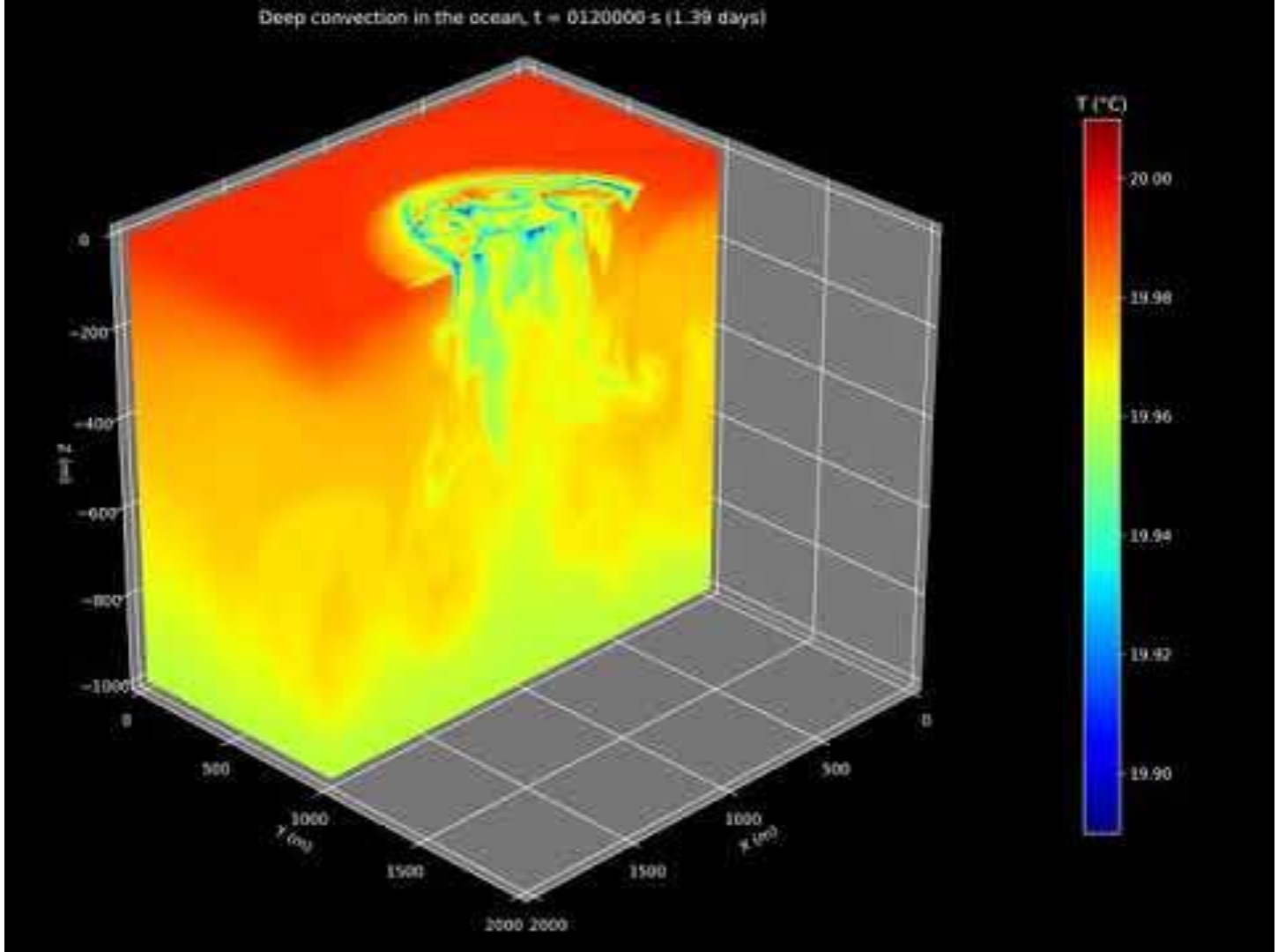
We tested our code and found that the model estimation is about ten times faster with Julia than before, a very large improvement. Our ports (computer lingo for "translations") of certain algorithms, such as Chris Sims's gensys (which computes the model solution), also ran about six times faster in Julia than the MATLAB versions we had previously used. (These results should not be interpreted as a broad assessment of the speed of Julia relative to MATLAB, as they apply only to the code we have written.) This document written by the New York Fed and QuantEcon collaborators, who did the real work on the port, documents the speed improvements and offers an overview of the hurdles encountered in the translation of a large codebase from one language to another. We hope it will be of use to other central banks and researchers embarking on similar projects.

https://libertystreeteconomics.newyorkfed.org/2015/12/the-frbny-dsge-model-meets-julia.html

CliMA
CLIMATE MODELING ALLIANCE

Home Our Team Workshops Publications For Students Press & Media Blog Join Us

CLIMATE MODELING ALLIANCE


Deep convection in the ocean, t = 0120000 s (1.39 days)
T (°C)
20.00
19.98
19.96
19.94
19.92
19.90
0
-200
-400
-600
-800
-1000
z (m)
0
500
1000
1500
2000
x (m)
0
500
1000
1500
2000
y (m)

São Paulo from Amazonia-1 Satellite

Why am I giving this talk?

- Selfish:
  - I think Julia is a very exciting tool, and want others to validate my excitement
  - I want more collaborators, contributors, and packages
- Altruistic:
  - I think Julia has potential to solve many of our field's problems
  - I think our field has a lot to contribute to Julia

Outline

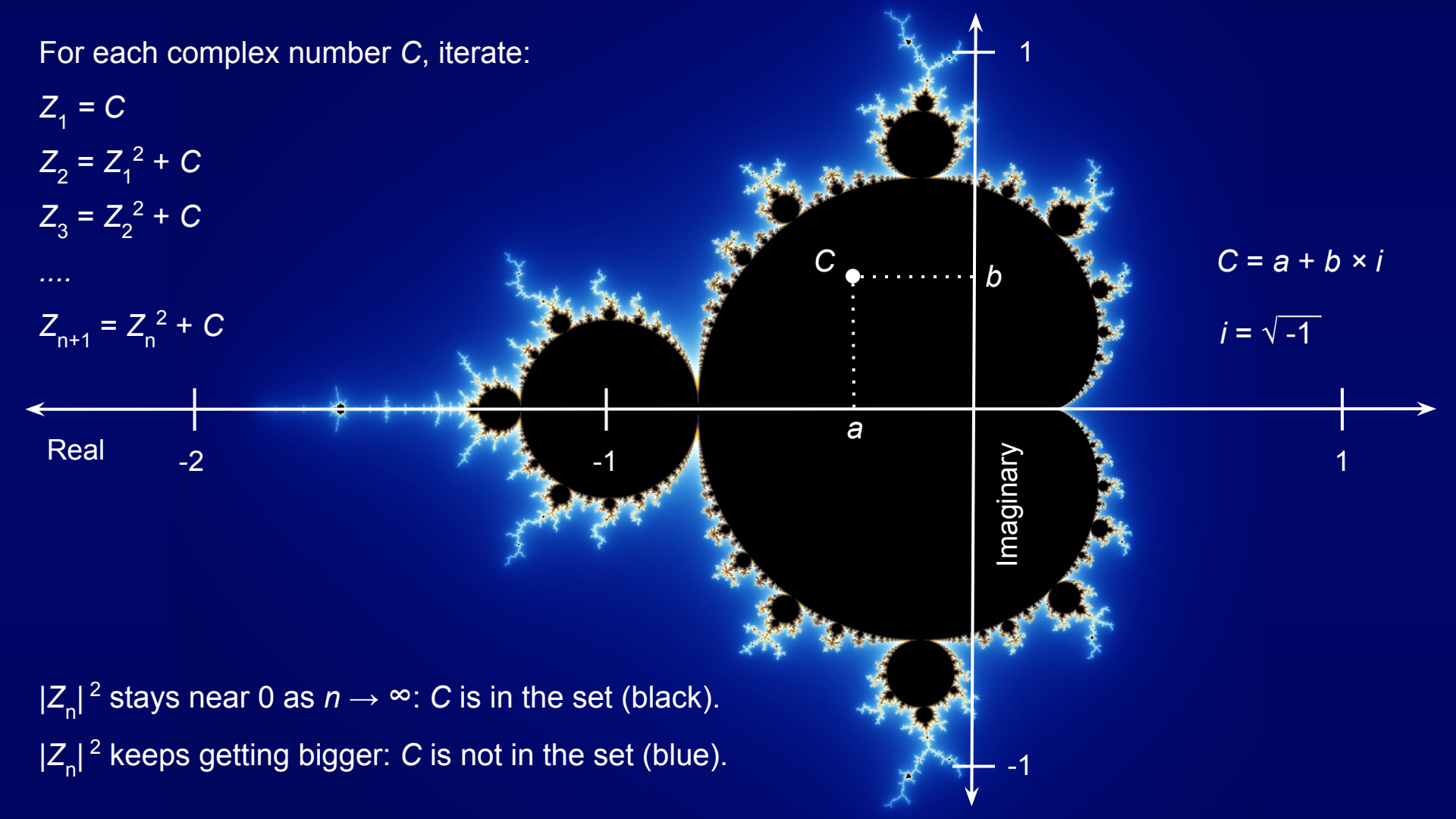
0. Introduction
1. Julia's strengths
    a. Speed
    b. Expressiveness
    c. Composability
2. Quick tour of some useful tools
3. Two case studies
    a. Simulating 55,550,000 birds
    b. Detecting 300,000,000 dolphin clicks
4. Concluding thoughts and advice

1. Julia’s strengths

1.1 *Speed*

Julia is fast.

- Yes, even for-loops!
- Well-written Julia code generally within ~2-5 x speed of C or Fortran
  - Maybe faster!
- Speed allows almost all of Julia to be written in Julia
- It *is* possible to write slow Julia code...
  - Worst case: about as slow as R/Python
  - Easy to avoid by following a few simple rules (see “Performance tips” in Julia manual)
- Julia developers have high expectations
  - Poor performance seen as bug


For each complex number C, iterate:
$Z_1 = C$
$Z_2 = Z_1^2 + C$
$Z_3 = Z_2^2 + C$
....
$Z_{n+1} = Z_n^2 + C$
$C = a + b \times i$
$i = \sqrt{-1}$
C
b
a
Real
-2
-1
1
Imaginary
1
-1
$|Z_n|^2$ stays near 0 as $n \to \infty$: C is in the set (black).
$|Z_n|^2$ keeps getting bigger: C is not in the set (blue).

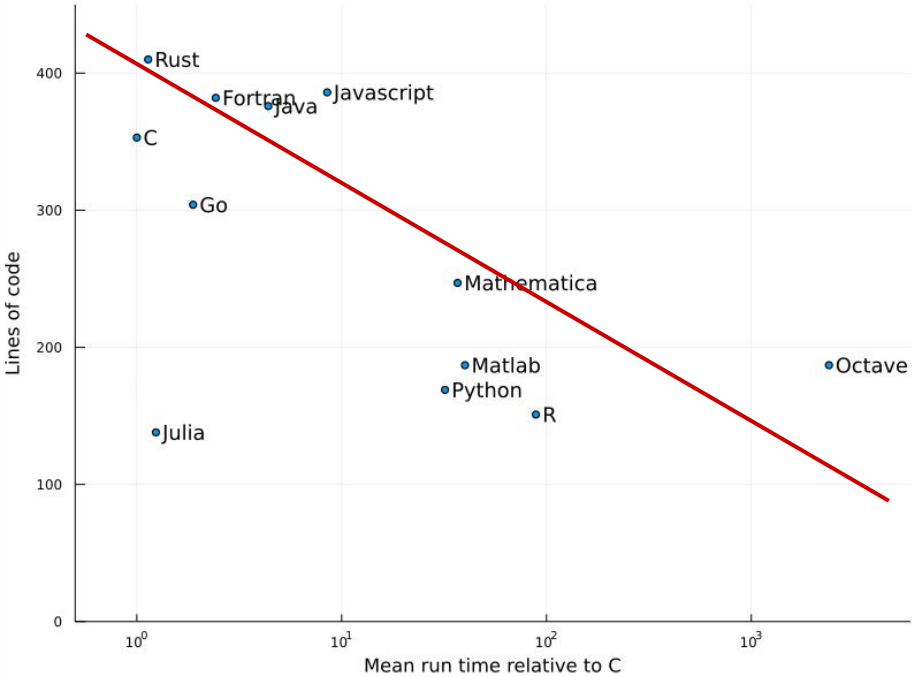
Demo: Mandelbrot speed test

How is this possible?

- Just-in-time *compiler*
- Precise, extensible system of *data types*
- *Multiple dispatch* for functions

The two-language problem

Compiled, statically typed



Interactive, dynamically typed

https://github.com/JuliaLang/Microbenchmarks

(Side note: what is a compiler, anyway?)

(Side side note: what even is a programming language?)

```
01010010101101001010100010
01011001010101111001011010
10110100101010101111001110
11010111000011101011010010
01110110100100111101010001
11100010110111100101111010
00000010100101011010010101
00101010110010101011111001
11010101101001010101011110
11101110101110000111010110
00101011101101001001111010
00010111000101101111001101
01001001111010100010111000
01101111001011110101000000
...
```

```
integer function mandel(z0) result(r)
  complex(dp), intent(in) :: z0
  complex(dp) :: c, z
  integer :: n, maxiter
  maxiter = 200
  z = z0
  c = z0
  do n = 1, maxiter
    if (abs2(z) > 4) then
      r = n-1
      return
    end if
    z = z**2 + c
  end do
  r = maxiter
end function
```

“Take a complex number, square it, and add it to itself. Then square that result and add it to the original number, and keep doing that until it gets too big or you hit 200 iterations.”

(Side note: what is a compiler, anyway?)

```
01010010101110100101010001 01
01011001010101111001011010
10110100101010101111001110 1
11010111000011101011010010 1
01110110100100111101010001 0
11100010110111100101111010 1
00000010100101011010010101 0
00101010110010101011111001 0
11010101101001010101011110 0
11101110101110000111010110 1
00101011101101001001111010 1
00010111000101101111001101 1
01001001111010100010111000 1
01101111001011110101000000 0
```


```
integer function mandel(z0) result(r)
  complex(dp), intent(in) :: z0
  complex(dp) :: c, z
  integer :: n, maxiter
  maxiter = 200
  z = z0
  c = z0
  do n = 1, maxiter
    if (abs2(z) > 4) then
      r = n-1
      return
    end if
    z = z**2 + c
  end do
  r = maxiter
end function
```

High-level vs. low-level programming

```
mandelbrot <- function(c, maxiter=200) {
  z <- c
  for (n in 1:maxiter) {
      if (abs(z)^2 > 4) {
          return(n - 1)
      }
      z <- z^2 + c
  }
  return(maxiter)
}
```


```
integer function mandel(z0) result(r)
  complex(dp), intent(in) :: z0
  complex(dp) :: c, z
  integer :: n, maxiter
  maxiter = 200
  z = z0
  c = z0
  do n = 1, maxiter
    if (abs2(z) > 4) then
      r = n-1
      return
    end if
    z = z**2 + c
  end do
  r = maxiter
end function
```

High-level vs. low-level programming

```
mandelbrot <- function(c, maxiter=200) {
  z <- c
  for (n in 1:maxiter) {
      if (abs(z)^2 > 4) {
          return(n - 1)
      }
      z <- z^2 + c
  }
  return(maxiter)
}
```

What is "c" at run-time?


```
integer function mandel(z0) result(r)
  complex(dp), intent(in) :: z0
  complex(dp) :: c, z
  integer :: n, maxiter
  maxiter = 200
  z = z0
  c = z0
  do n = 1, maxiter
    if (abs2(z) > 4) then
      r = n-1
      return
    end if
    z = z**2 + c
  end do
  r = maxiter
end function
```

Julia’s compromise: JIT compiler + multiple dispatch

```
mandelbrot <- function(c, maxiter=200) {
  z <- c
  for (n in 1:maxiter) {
      if (abs(z)^2 > 4) {
          return(n - 1)
      }
      z <- z^2 + c
  }
  return(maxiter)
}
```

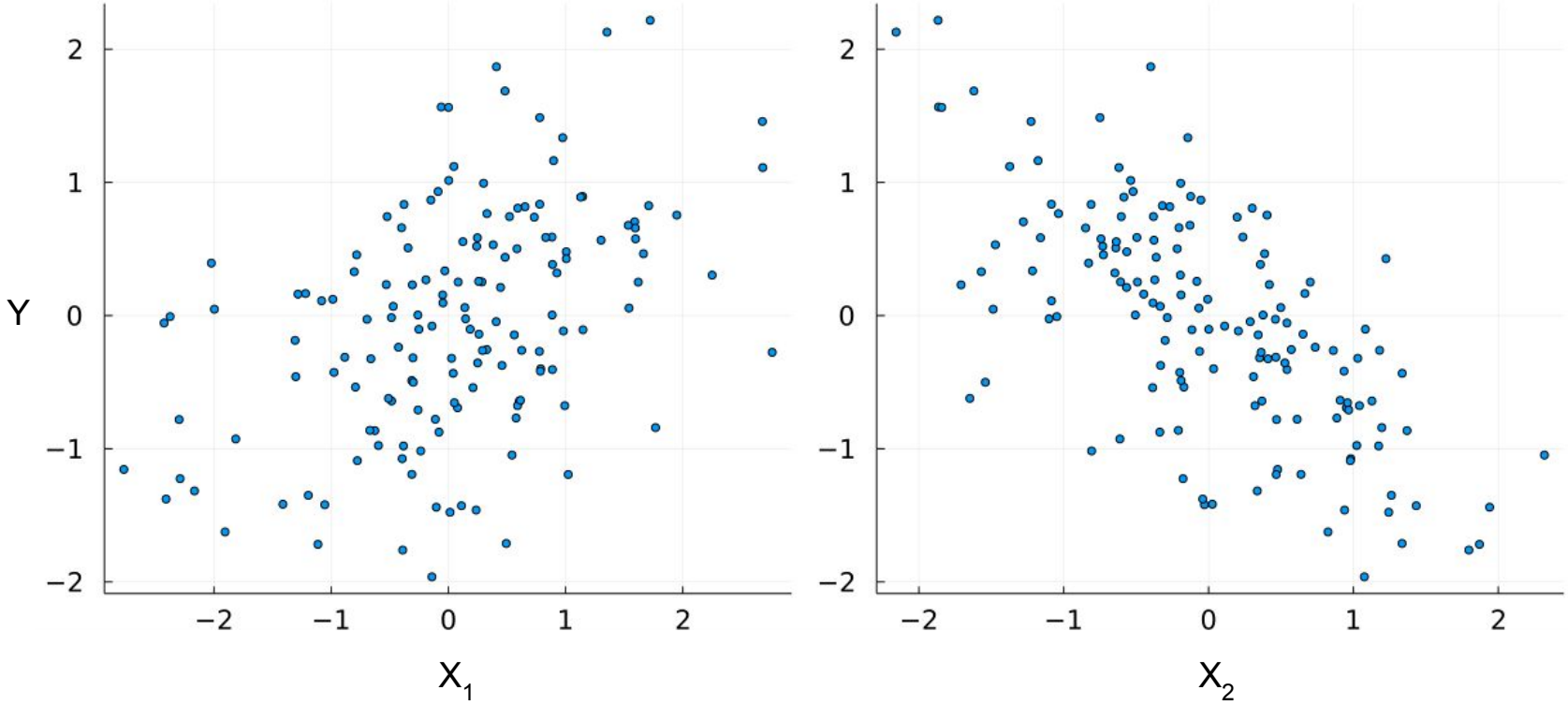
```
function mandelbrot(c, maxiter=200)
  z = c
  for n in 1:maxiter
      if abs(z)^2 > 4
          return n-1
      end
      z = z^2 + c
  end
  return maxiter
end

mandelbrot(1)
mandelbrot(1.0)
mandelbrot(1.0 + 0im)
```

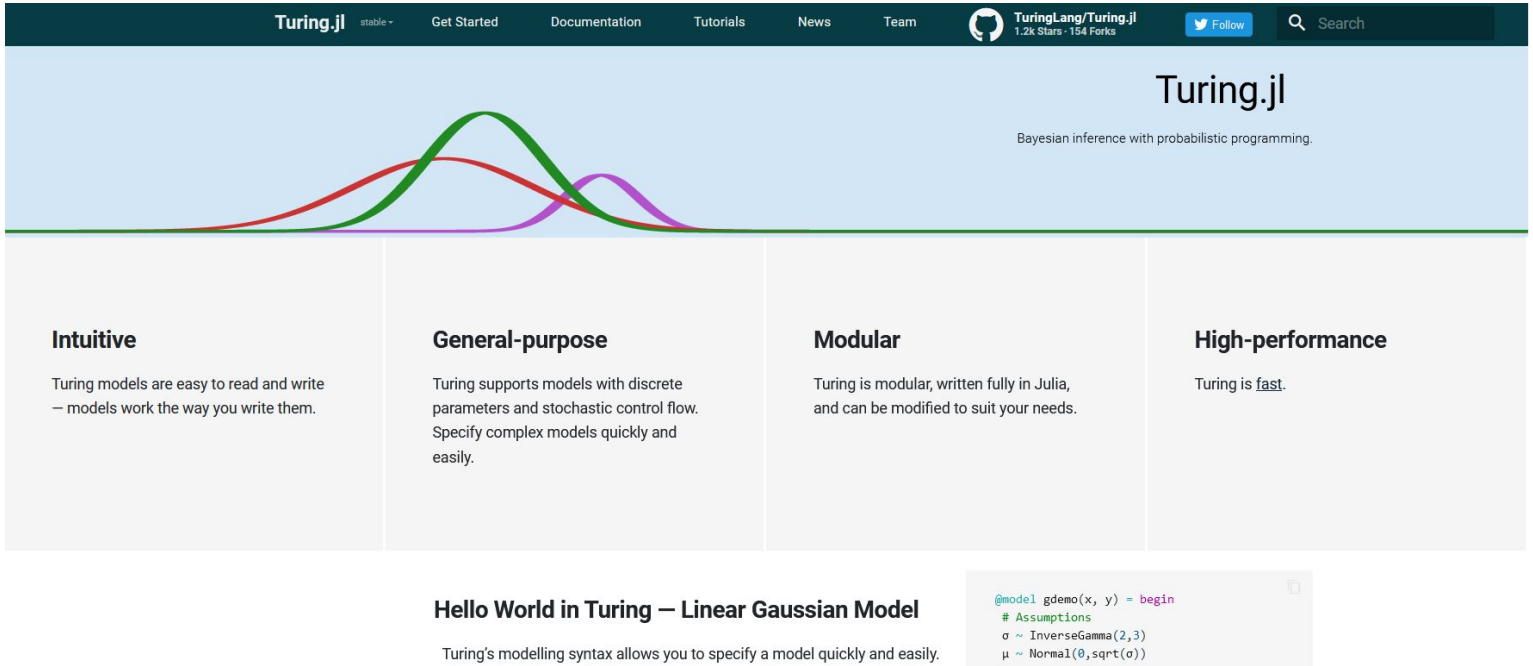
Demo: multiple dispatch

1.1 *Speed*

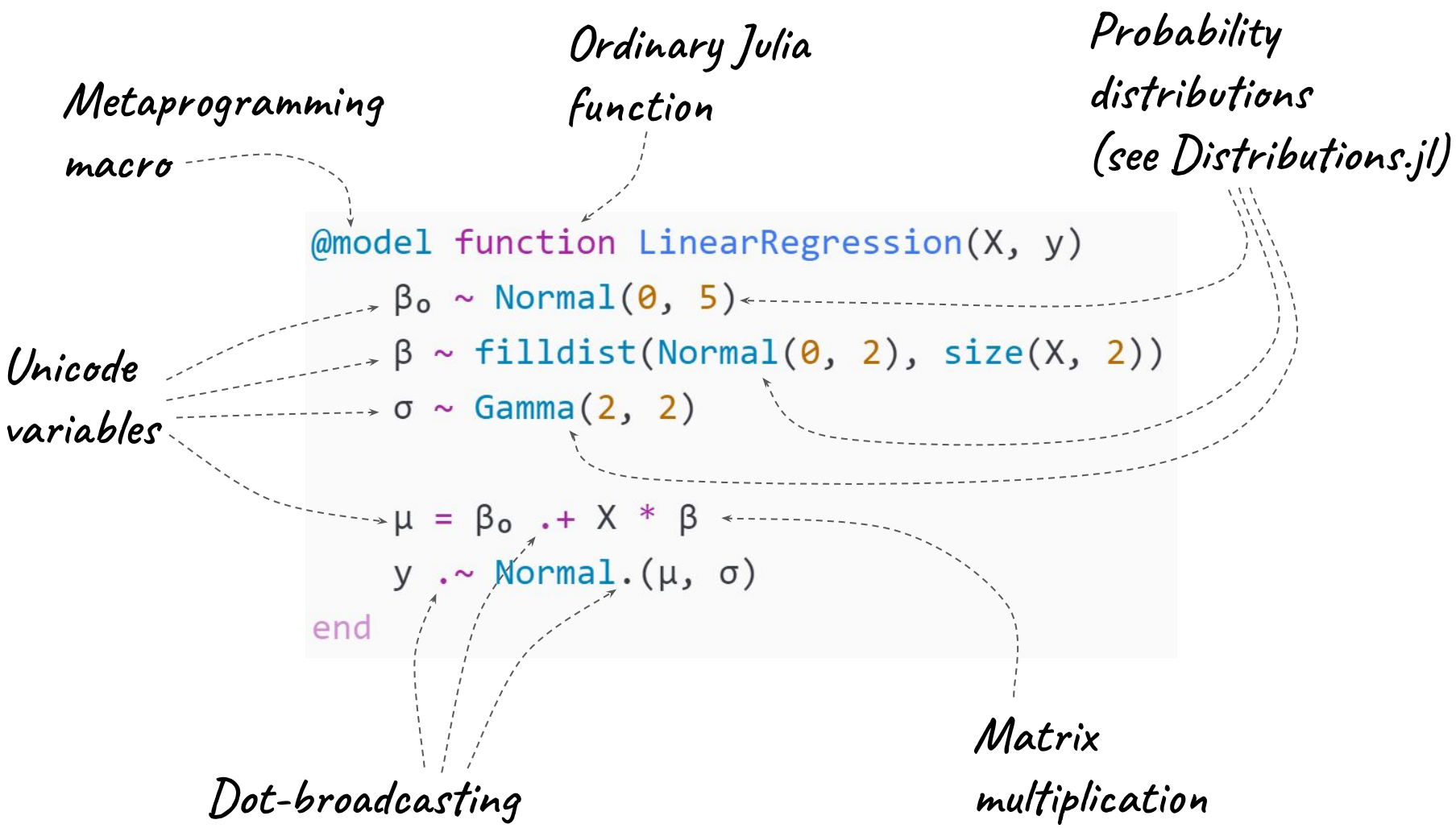
1.2 *Expressiveness*

Expressiveness example 1: Bayesian linear regression





https://turing.ml


Metaprogramming macro
Ordinary Julia function
Probability distributions (see Distributions.jl)
Unicode variables
@model function LinearRegression(X, y)
β₀ ~ Normal(0, 5)
β ~ filldist(Normal(0, 2), size(X, 2))
σ ~ Gamma(2, 2)
μ = β₀ .+ X * β
y .~ Normal.(μ, σ)
end
Dot-broadcasting
Matrix multiplication

Put covariates in
n x 2 matrix

Read in example dataset as
a data frame

```
data = CSV.read("linear_regression.csv", DataFrame)
X = Matrix(data[:, [:x1, :x2]])
y = data.y
model = LinearRegression(X, y)
chain = sample(model, NUTS(), 2000)
```

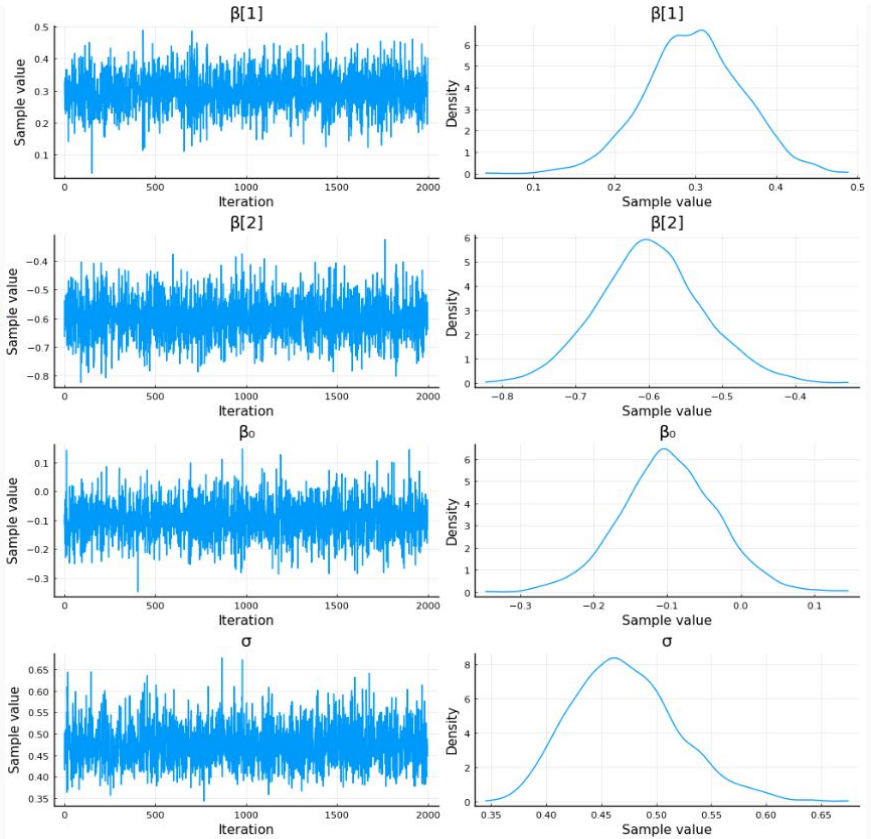
Create probabilistic
model, conditioned
on X and y

Sample posterior
2,000 times
using MCMC

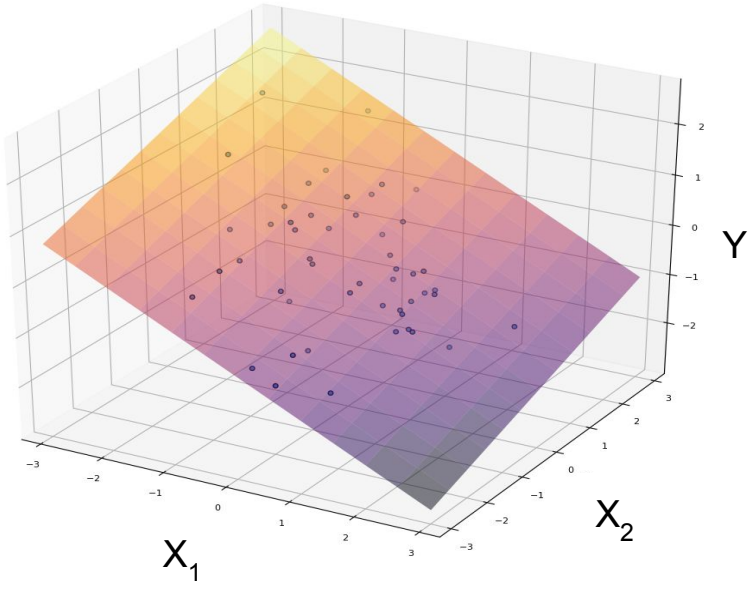
No-U-Turn Sampler

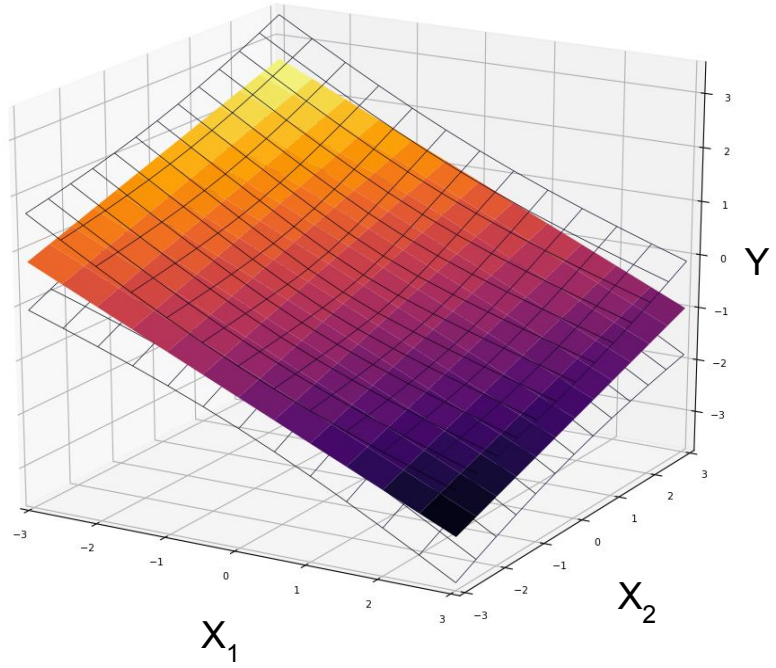
MCMC Results

```
julia> chain
Chains MCMC chain (2000×16×1 Array{Float64, 3}):

Start time        = 2021-05-05T22:34:55.471
Stop time         = 2021-05-05T22:34:55.746
Wall duration     = 0.28 seconds
Iterations        = 1:2000
Thinning interval = 1
Chains            = 1
Samples per chain = 2000
parameters        = β[1], β[2], β₀, σ
internals         = acceptance_rate, hamiltonian_energy, hamiltonian_energy_error, is_accept,

Summary Statistics
  parameters      mean       std   naive_se      mcse         ess      rhat   ess_per_sec
      Symbol   Float64   Float64    Float64   Float64     Float64   Float64       Float64

        β[1]    0.2985    0.0601     0.0013    0.0012   2777.9233    1.0002   10101.5393
        β[2]   -0.5975    0.0709     0.0016    0.0015   3099.4395    0.9998   11270.6891
          β₀   -0.0974    0.0656     0.0015    0.0012   2994.5459    1.0002   10889.2579
           σ    0.4719    0.0481     0.0011    0.0007   2363.2962    0.9996    8593.8043

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

        β[1]    0.1789    0.2598    0.2983    0.3384    0.4154
        β[2]   -0.7327   -0.6441   -0.5995   -0.5539   -0.4507
          β₀   -0.2286   -0.1395   -0.0988   -0.0545    0.0326
           σ    0.3905    0.4380    0.4676    0.5017    0.5778
```



Linear prediction surface and 96% credible interval





Turing.jl

```julia
using CSV, DataFrames, Turing
using StatsPlots

data = CSV.read("linear_regression.csv", DataFrame)
X = Matrix(data[:, [:x1, :x2]])
y = data.y

@model function LinearRegression(X, y)
    β₀ ~ Normal(0, 5)
    β ~ filldist(Normal(0, 2), size(X, 2))
    σ ~ Gamma(2, 2)

    μ = β₀ .+ X * β
    y .~ Normal.(μ, σ)
end

model = LinearRegression(X, y)
chain = sample(model, NUTS(), 2000)
plot(chain)
```

And what about speed...?

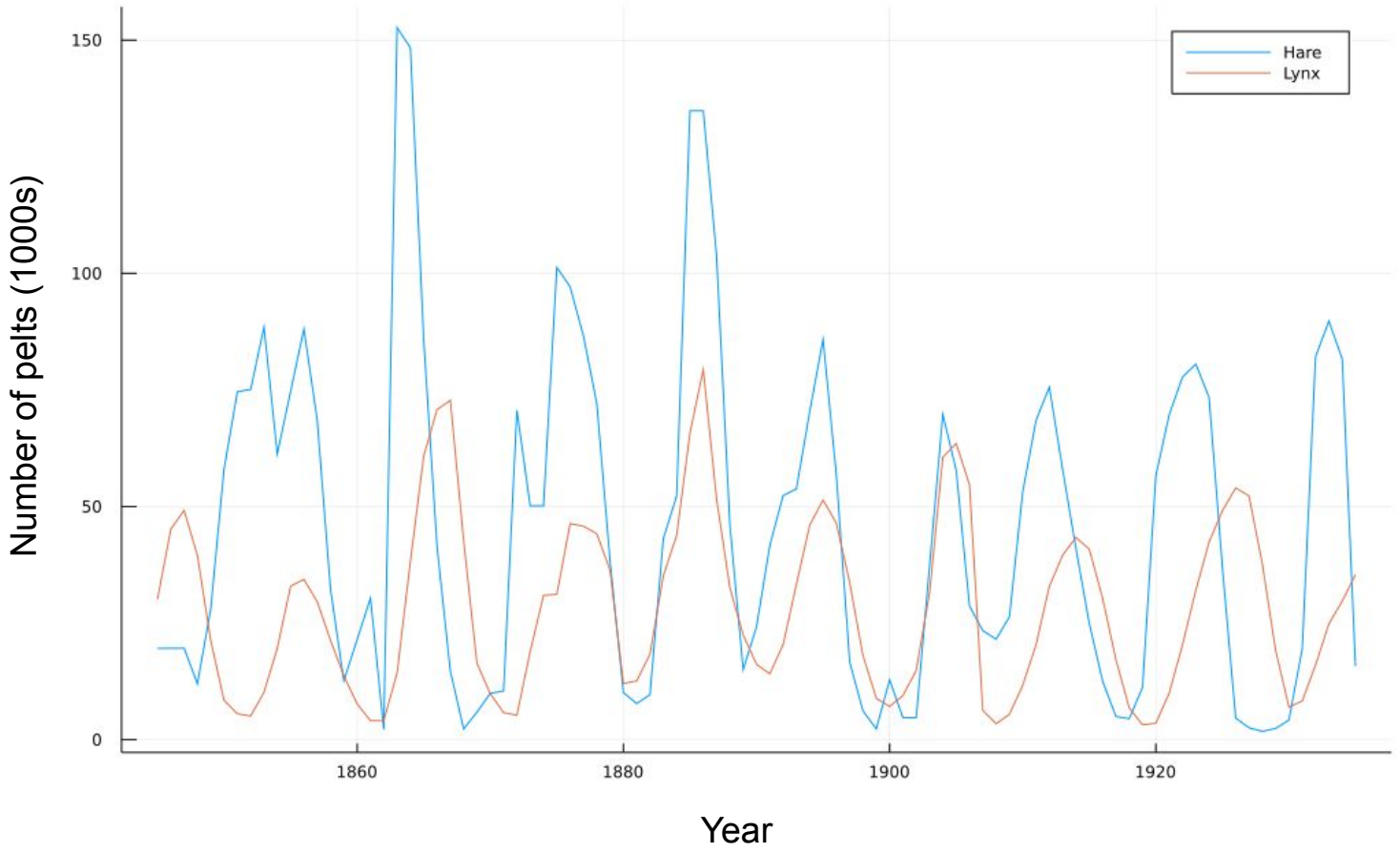
After compiling both models:

Turing.jl: 89 µs / sample
Stan: 263 µs / sample

Stan + R

```r
library(rstan)

data <- read.csv("linear_regression.csv")
X <- data[c("x1", "x2")]
y <- data$y

stan_code <- "
data {
  int<lower=0> N;
  int<lower=0> K;
  matrix[N, K] x;
  vector[N] y;
}

parameters {
  real beta_0;
  vector[K] beta;
  real<lower=0> sigma;
}

transformed parameters {
  vector[N] mu;
  mu = beta_0 + x * beta;
}

model {
  beta_0 ~ normal(0, 0.2);
  beta ~ normal(0, 0.5);
  sigma ~ gamma(2, 0.5);
  y ~ normal(mu, sigma);
}"

linear_regression <- stan_model(model_code=stan_code)
chain <- sampling(linear_regression,
                  data=list(y=y, x=X, N=nrow(data), K=ncol(X)),
                  warmup=1000, iter=3000)
plot(chain)
```

Expressiveness example 2: Two-species Lotka-Volterra model


Hare
Lynx
Number of pelts (1000s)
150
100
50
0
1860
1880
1900
1920
Year

Reproduction

Mortality (~predation)

$$\frac{dN_1}{dt} = \alpha N_1 - \beta N_1 N_2$$

$$\frac{dN_2}{dt} = \delta N_1 N_2 - \gamma N_2$$

Rates of change

Reproduction (~feeding)

Mortality

Ordinary Julia function

"!" indicates function mutates an argument

Unpack populations from "N"

Unpack parameters

```julia
using DifferentialEquations

function lotka_volterra!(dNdt, N, params, t)
    🐇 = N[1]
    🐆 = N[2]
    α, β, γ, δ = params
    dNdt[1] = α*🐇 - β*🐇*🐆
    dNdt[2] = δ*🐇*🐆 - γ*🐆
end
```

Calculate rates of change, update dN/dt in-place

Set parameters and initial populations

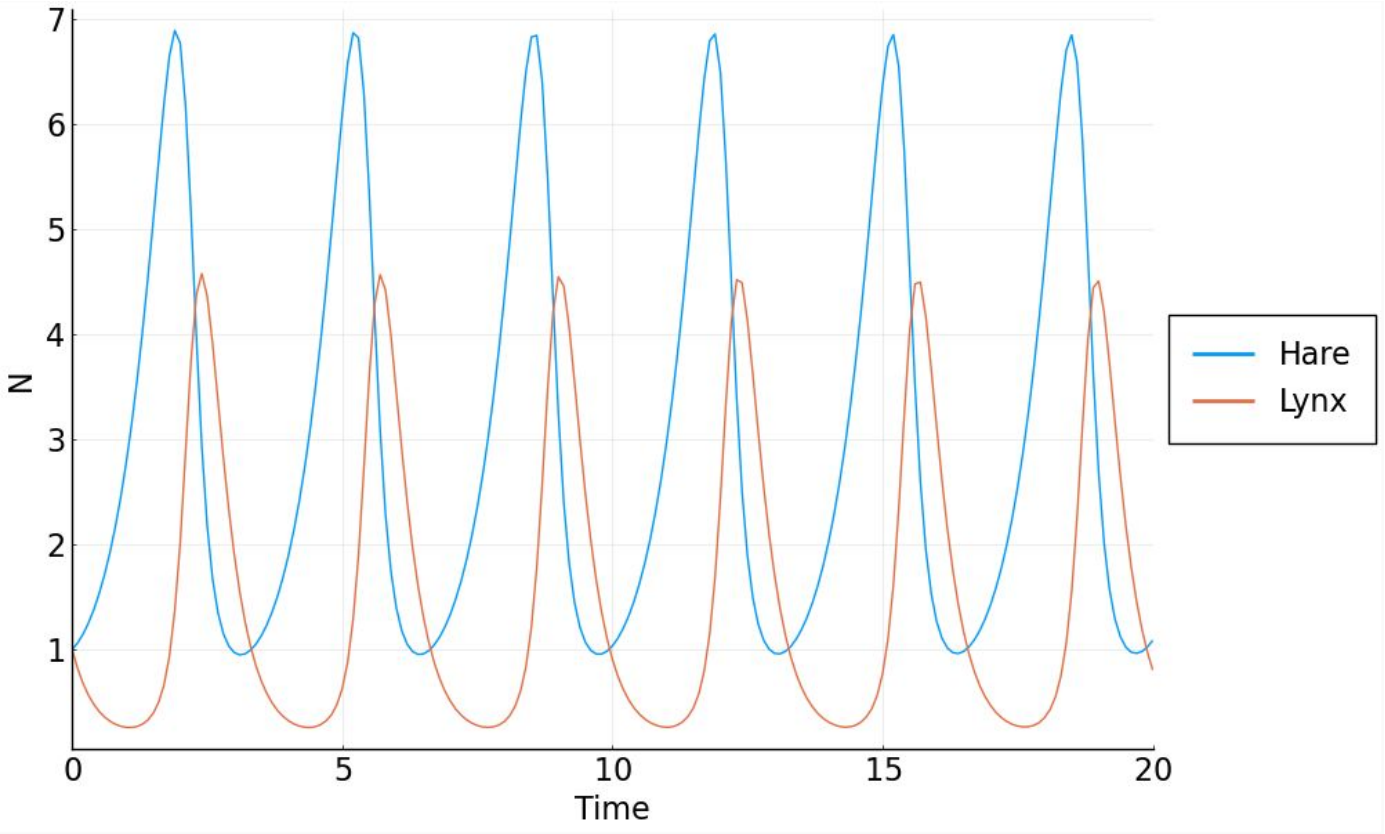
Time-span to solve equations over

```
10
11  params = [1.5, 1.0, 3.0, 1.0]
12  N0 = [1.0,1.0]
13  tspan = [0.0, 10.0]
14  prob1 = ODEProblem(lotka_volterra!, N0, tspan, params)
15  sol = solve(prob1, saveat=0.1)
```

Define problem

Solve problem

```
17  using Plots
18  plot(sol, xlabel="Time", ylabel="N", label=["Hare" "Lynx"],
19      legend=:outerright, size=(1000, 600), tickfontsize=16,
20      legendfontsize=16, guidefontsize=16)
```



Expressiveness

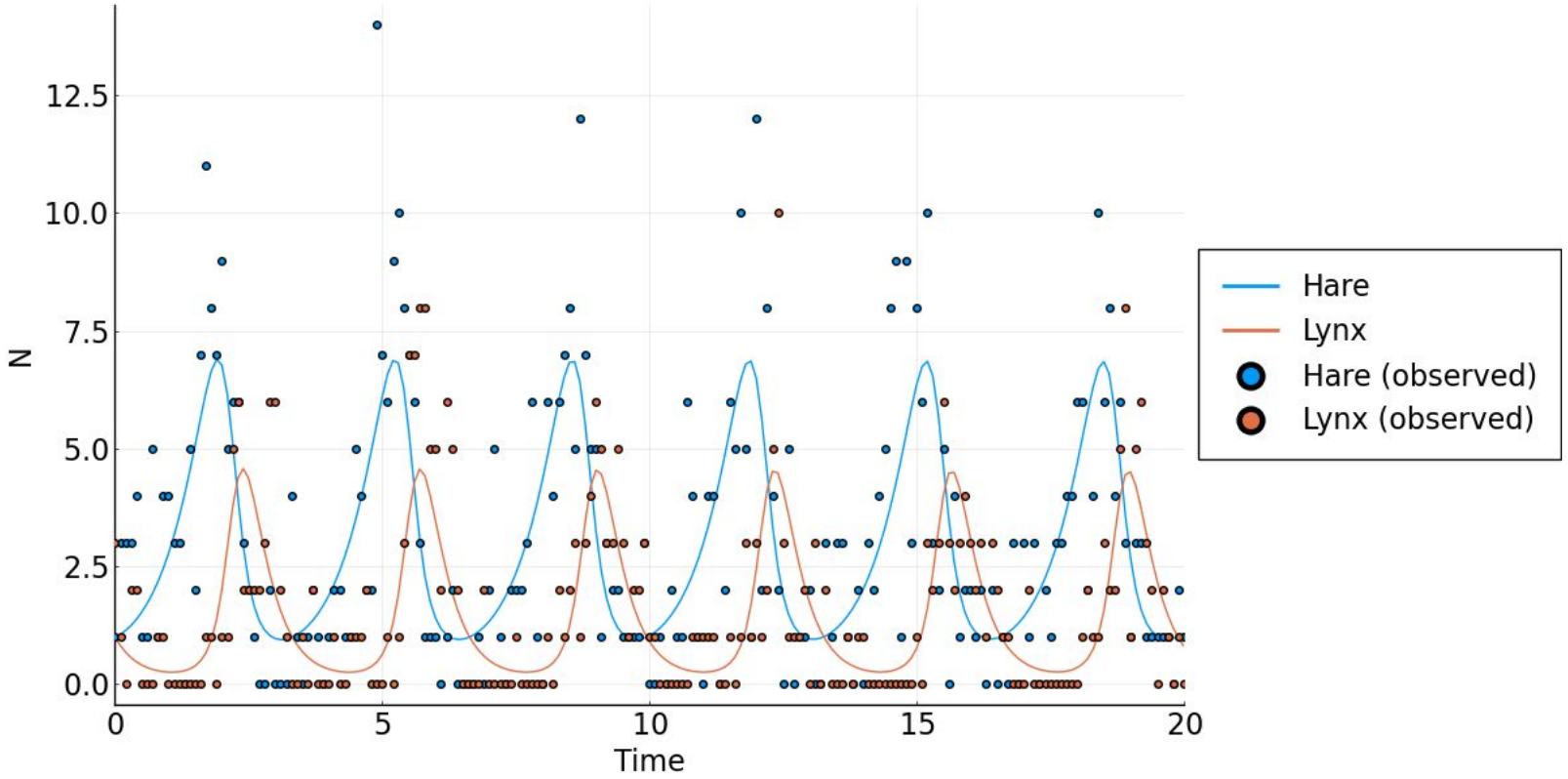
- Concise, intuitive syntax
- Natural mathematical notation
  - Matrix operations
  - Unicode symbols
- Native speed: can write everything in plain Julia
- Metaprogramming allows specialized syntax
  - E.g., Turing.jl `@model` definitions

1.2 *Expressiveness*

1.3 *Composability*

What if we wanted to fit the Lotka-Volterra model to data?

```
33  odedata = rand.(Poisson.(Array(sol)))
```



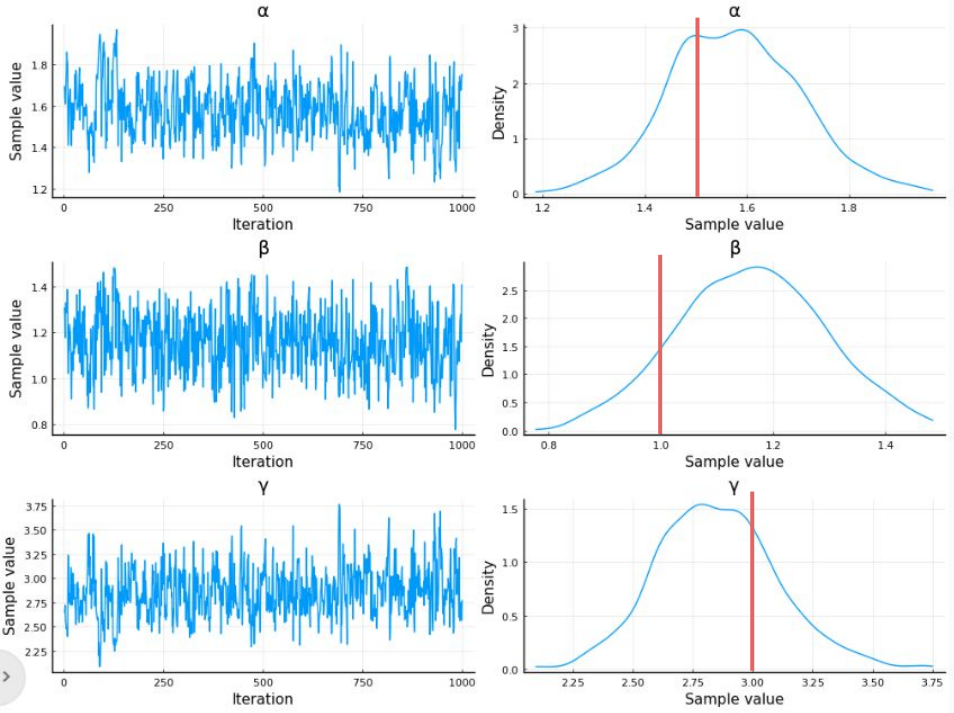
```julia
@model function LotkaVolterraModel(N_obs)
    # Set priors on parameters and initial populations
    α ~ truncated(Normal(1.5, 0.5), 1.0, 2.0)
    β ~ truncated(Normal(1.2, 0.5), 0.5,1.5)
    γ ~ truncated(Normal(3.0, 0.5), 2, 4)
    δ ~ truncated(Normal(1.0, 0.5), 0.5, 1.5)
    🐇 ~ truncated(Normal(2, 2), 0.1, 10)
    🐆 ~ truncated(Normal(1, 2), 0.1, 10)

    # Run the model, conditional on the parameters
    params = [α, β, γ, δ]
    N0 = [🐇, 🐆]
    prob = ODEProblem(lotka_volterra!, N0, tspan, params)
    predicted = solve(prob, saveat=0.1)

    # Calculate the likelihood of the observed data
    for i in 1:length(predicted)
        N_obs[:, i] .~ Poisson.(predicted[i])
    end
end

lv_model = LotkaVolterraModel(odedata)
lv_chain = sample(lv_model, NUTS(), 1000)
```

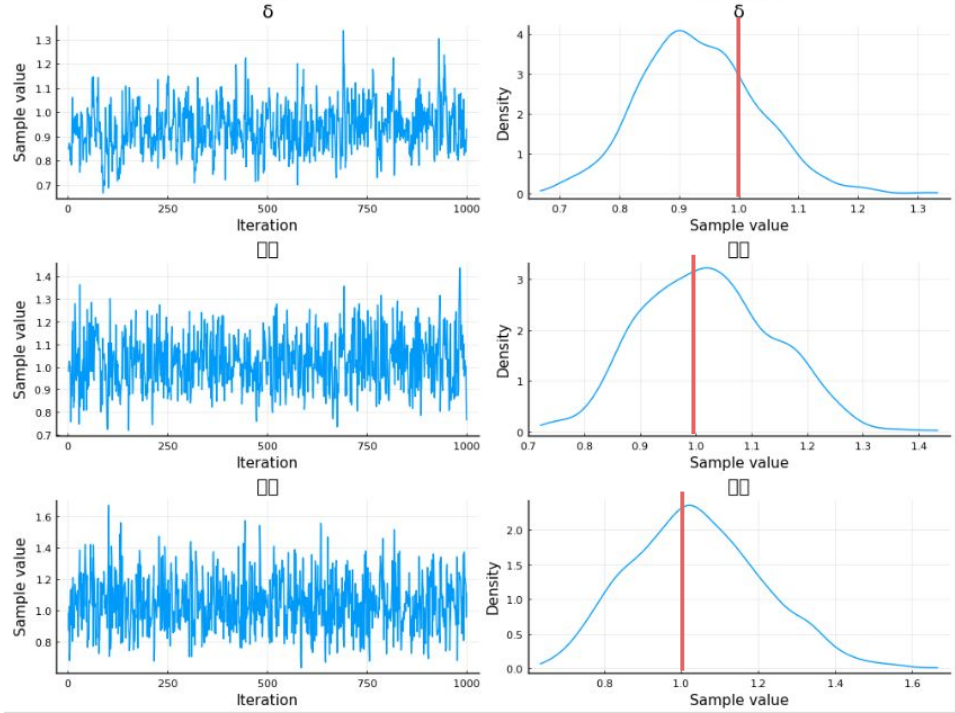
Define new Turing model, takes count data as input

Exactly the same code we used before to solve the L-V system

Still using NUTS: auto-diff can take derivative of ODE(!)

Did we infer the right parameter values?





Hey! You got MCMC in my differential equations!

No way! You got differential equations in my Hamiltonian Monte Carlo!

Composability

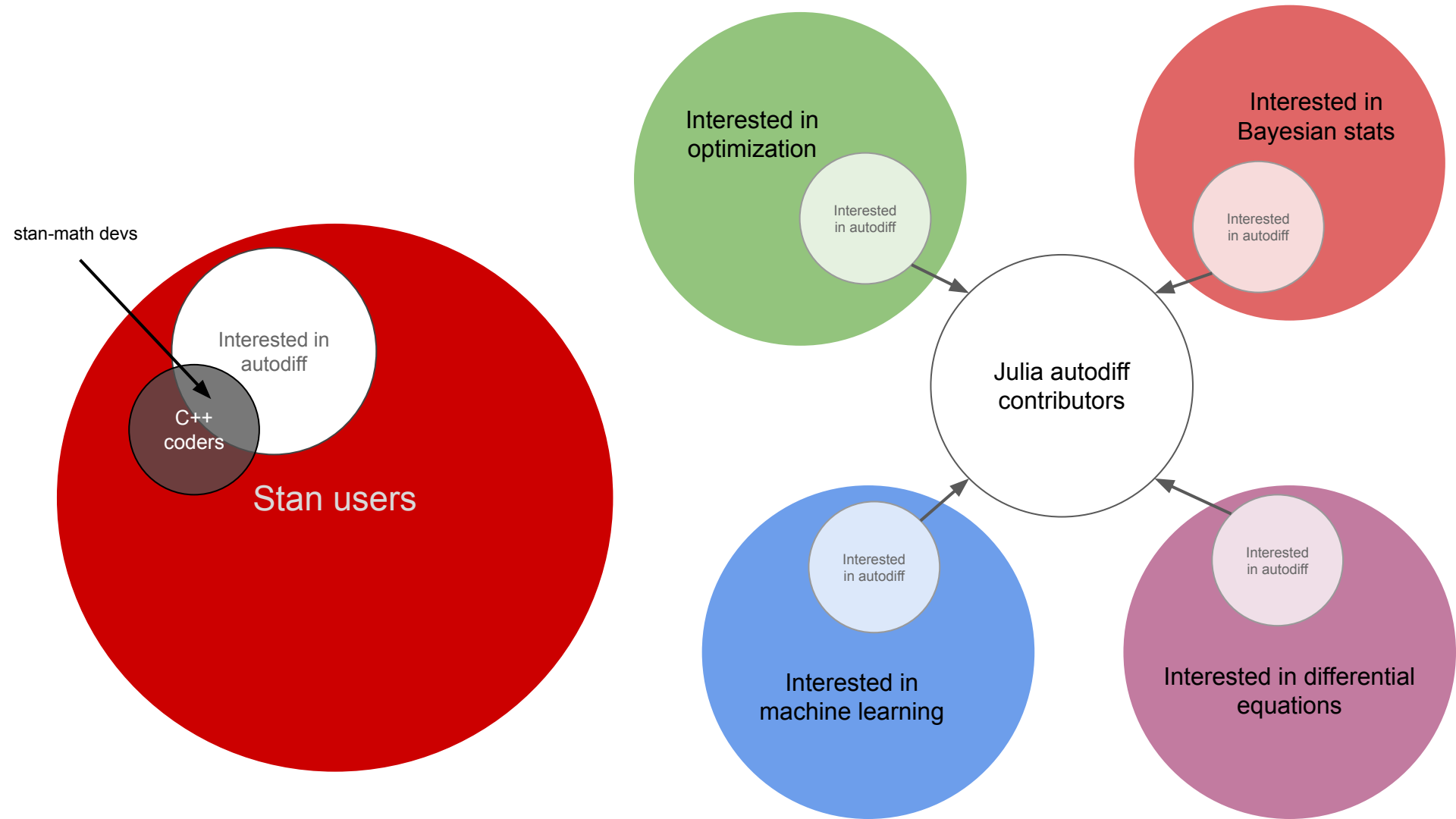
- Turing, Distributions, DifferentialEquations, and ForwardDiff:
  - All written 100% in Julia
  - Mostly “just work” together
  - *Developed independently, by different groups!*
- Combinatorial package mashups possible
- *Easy to think at right level of abstraction*

- Benefits of composability weren’t fully anticipated by Julia’s creators
  - “The unreasonable effectiveness of multiple dispatch” (Stefan Karpinsky): https://www.youtube.com/watch?v=kc9HwsxE1OY
- Leads to virtuous cycles of code re-use and collaboration
  - E.g., Automatic differentiation ecosystem

Automatic differentiation

- Computer program that does your calculus homework for you
- Used in:
  - Hamiltonian Monte Carlo (Stan, PyMC)
  - Complex optimization (e.g. the “AD” in ADMB)
  - Machine learning (e.g. the “Flow” in TensorFlow)
  - Solving differential equations
- Not trivial to implement

- So what does this have to do with composability?


stan-math devs
Interested in autodiff
C++ coders
Stan users
Interested in optimization
Interested in autodiff
Interested in Bayesian stats
Interested in autodiff
Julia autodiff contributors
Interested in autodiff
Interested in machine learning
Interested in autodiff
Interested in differential equations

2. Quick tour of useful tools

Package manager

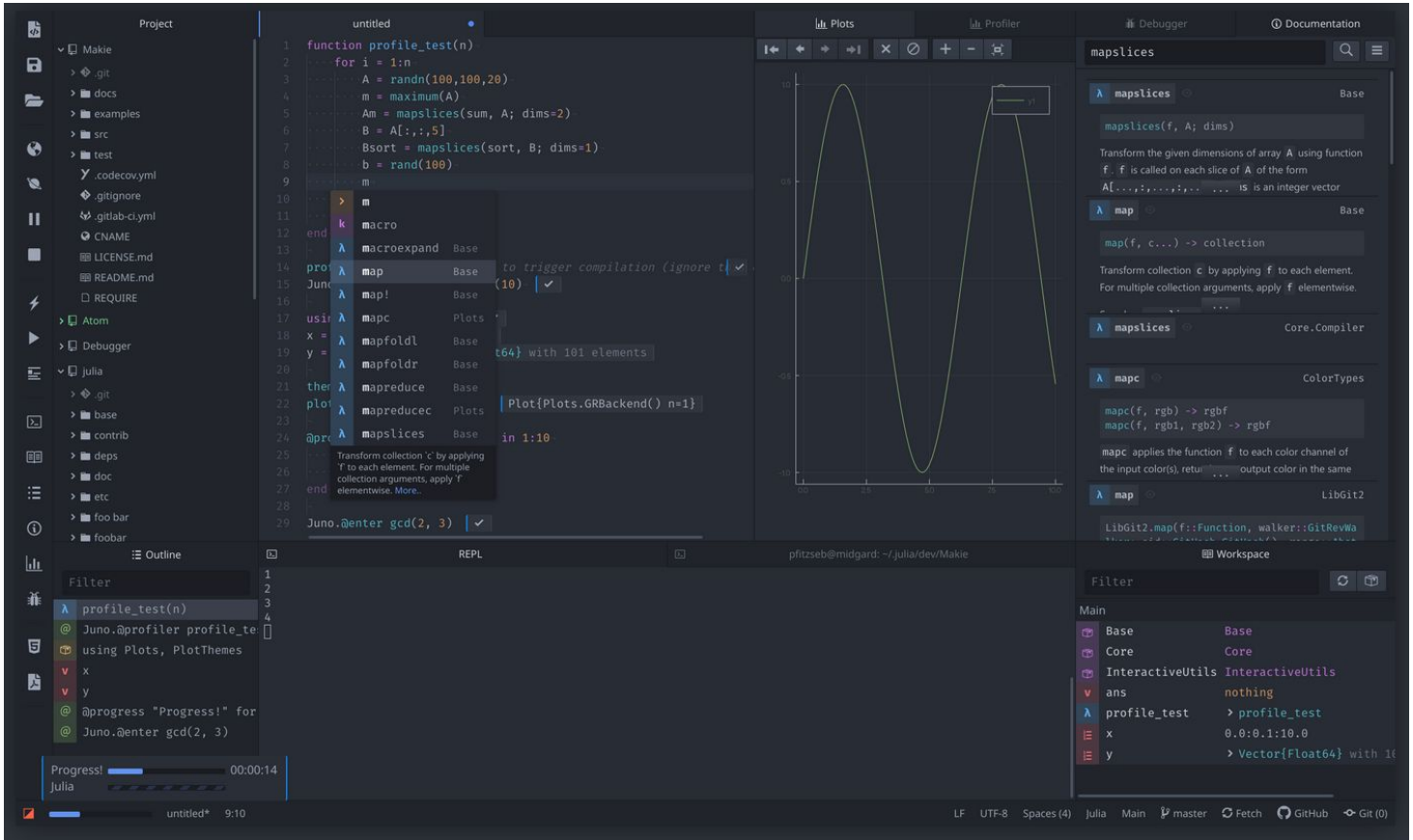
- Built-in utility to add, remove, update packages
- Type “]” at command prompt to enter package mode (backspace to exit)

```
julia>

(@v1.6) pkg> add GLM
   Resolving package versions...
    Updating `C:\Users\sam.urmy\.julia\environments\v1.6\Project.toml`
  [38e38edf] + GLM v1.4.2
    Updating `C:\Users\sam.urmy\.julia\environments\v1.6\Manifest.toml`
  [38e38edf] + GLM v1.4.2
  [1277b4bf] + ShiftedArrays v1.0.0
  [3eaba693] + StatsModels v0.6.22

julia>
```

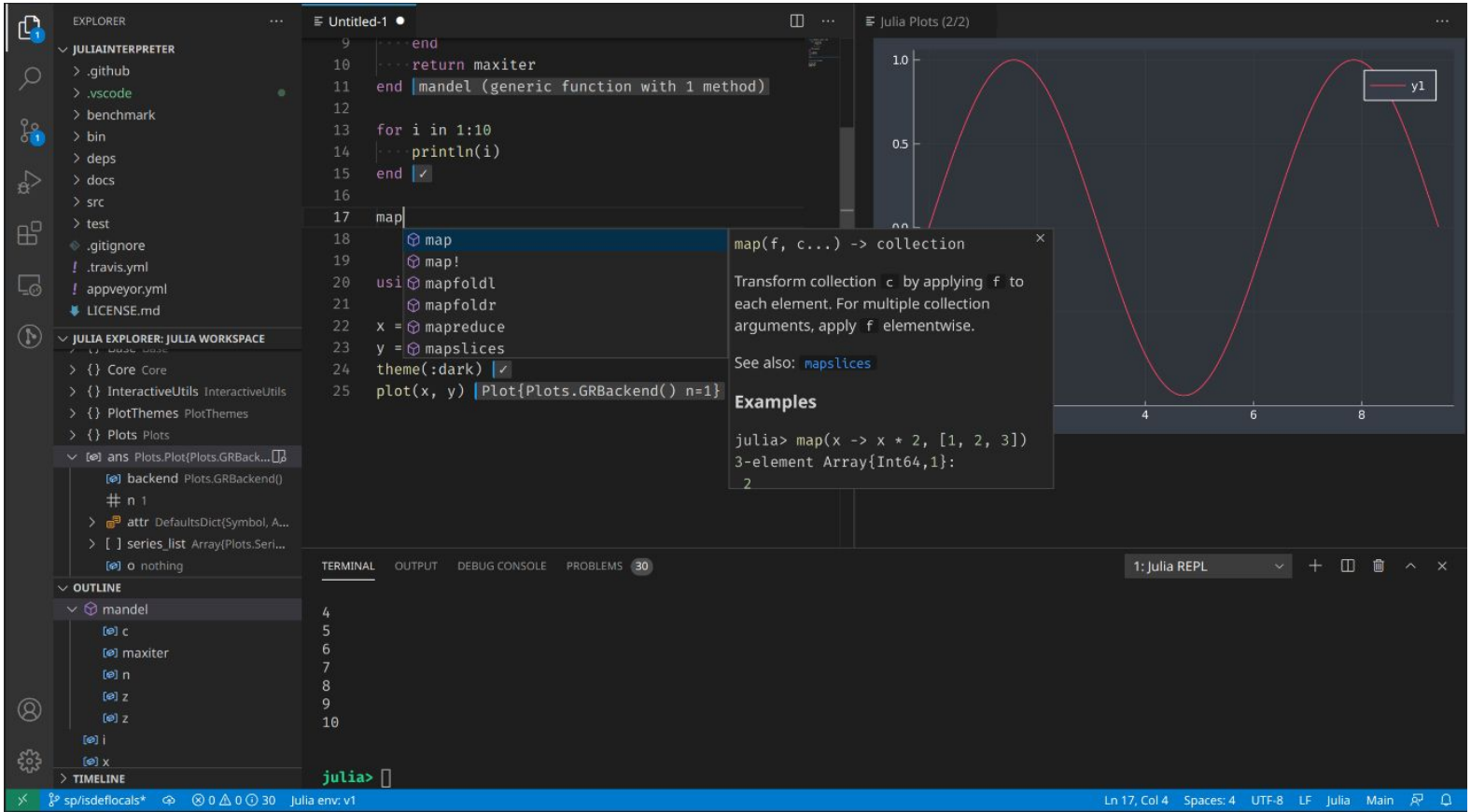
- Easy to set up self-contained environments (reproducibility!)

Code editors and integrated development environments

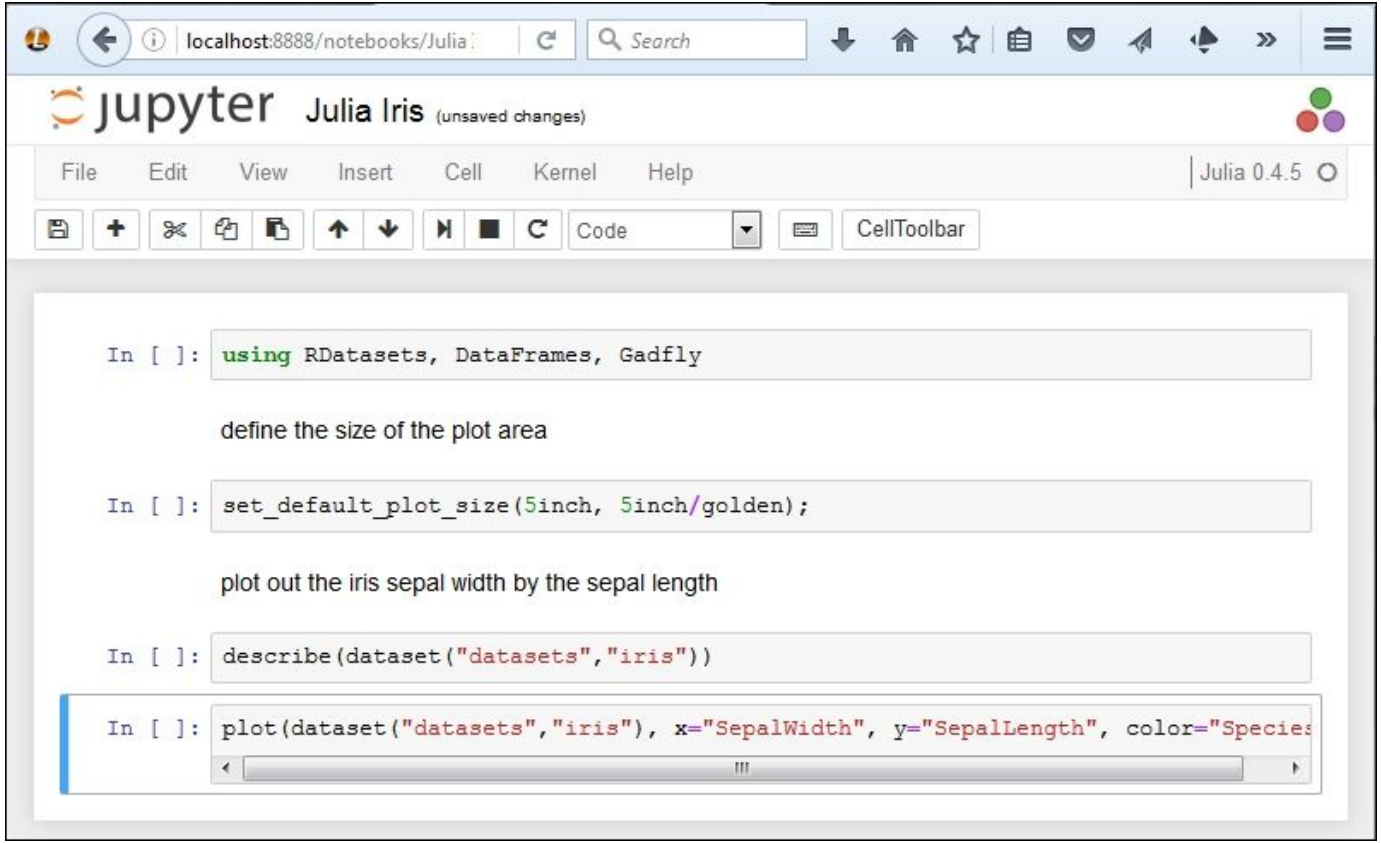
Juno: Julia plugin for Atom text editor



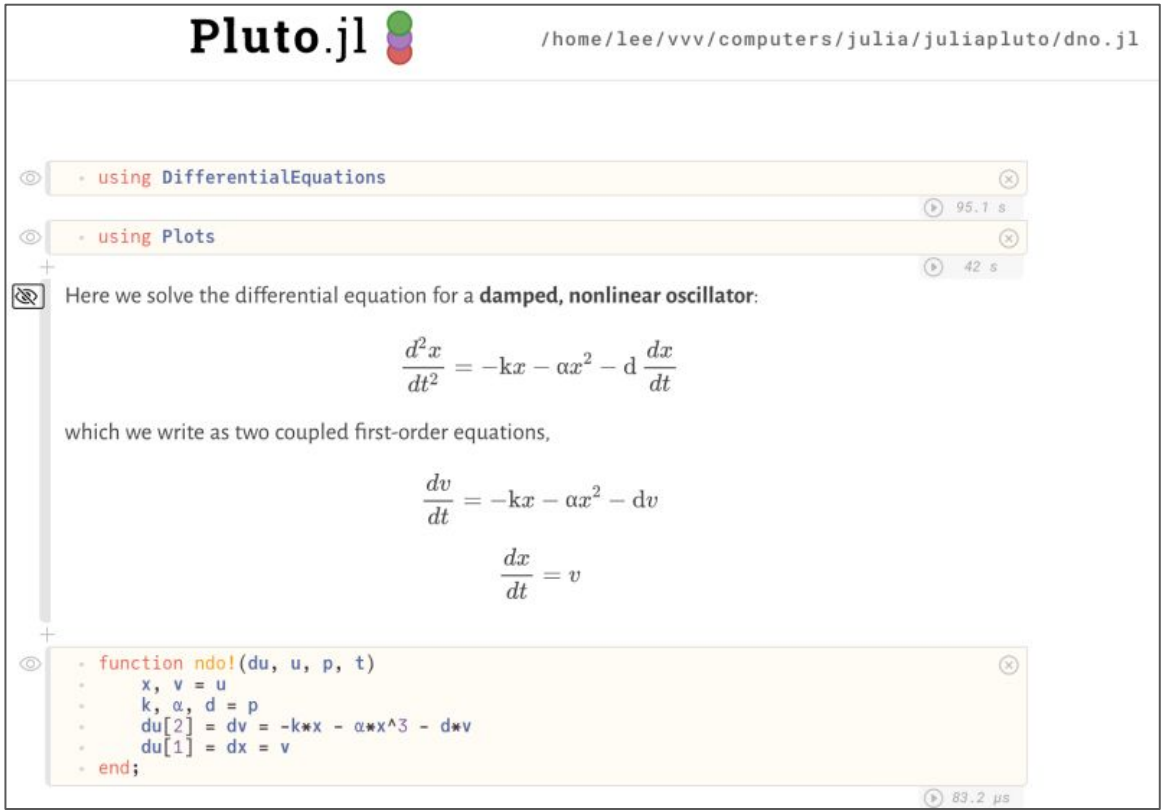
Julia-VSCode: plugin for Visual Studio Code



Julia puts the “ju” in in Jupyter



Pluto: lightweight, reactive notebooks

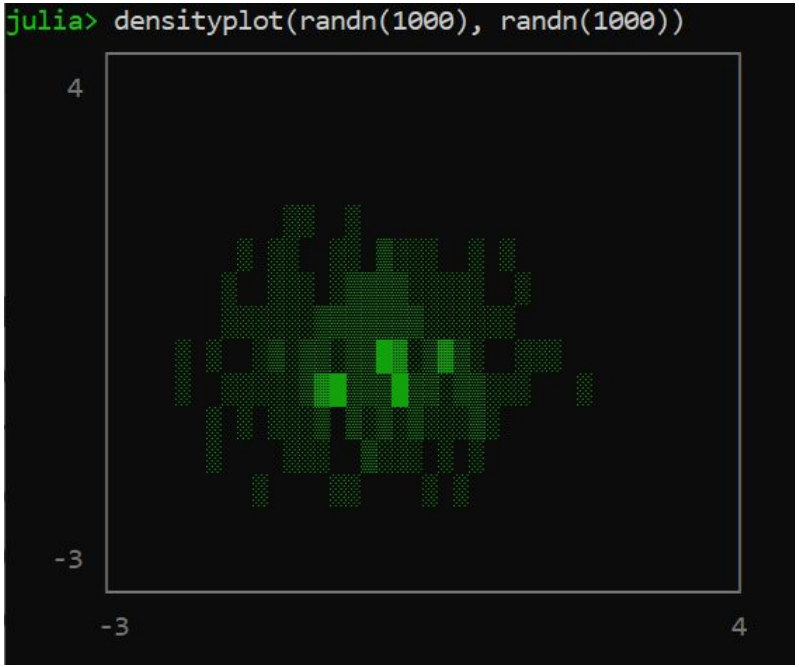


Coding tools

- *Debugger*
- *BenchmarkTools*
- Macros for code introspection
  - `@which`: find the specific method that
  - `@code_warntype`: Find type-instabilities
  - `@code_lowered, @code_llvm, @code_native`: How was your Julia code translated into assembly/machine code?
- Profiler
  - Which parts of your code are slow?

Plotting

- *Plots*: most popular
- *Makie*: next-gen plotting package
- *Gadfly*: grammar-of-graphics
- *VegaLite*: another grammar-of-graphics
- *PyPlot*: wraps Python's *matplotlib*
- *PGFPlotsX*: wraps LaTeX library *PGFPlots*
- *UnicodePlots*: ASCII plots at command line



Data frames and data manipulation

- *DataFrames*: main package for tabular data
- *DataFramesMeta*: data manipulation, comparable to dplyr
- *Query*: alternative data manipulation framework
- *CSV*, *Feather*, *RData*, *ODBC*: read in data in various formats
- *RDatasets*: all the “classic” datasets from R

Statistics

- *StatsBase*, *StatsFuns*: stats basics (`std`, `mean`, `probit`, `logistic`, etc.)
- *HypothesisTests*
- *GLM* and *MixedModels*: linear models, similar interface to R

```
julia> ols = lm(@formula(Y ~ X), data)
```

- *MultivariateStats*: PCA, factor analysis, etc.
- *Clustering*
- *Turing* and *Soss*: Bayesian modeling
- *JuliaGaussianProcesses*: Gaussian fields (compatible with Turing)
- *GeoStats*: spatial data, variograms, kriging
- *Measurements*, *MonteCarloMeasurements*: uncertainty propagation

Numerical tools

- *LinearAlgebra*, *SparseArrays*
- Automatic differentiation
  - *ForwardDiff*, *ReverseDiff*, *FiniteDiff*, *Zygote*, *ChainRules*
- *SparseDiffTools*: sparsity detection
- *Symbolics*: very new symbolic math library
- *DifferentialEquations*
- *ModelingToolkit*: growing ecosystem of numerical tools
- *Optim*: general optimization
- *JuMP*: linear programming
- Built-in support for parallel and distributed computing

Interoperability with other languages

- Built in `ccall` function
- *JuliaInterop* organization on GitHub
- *RCall* / *JuliaCall*
  - *RData*: load .Rdata files
- *PyCall* / *PyJulia*
- *MATLAB*: call Matlab scripts
  - *MAT*: load mat files

Community resources

- Official forum: discourse.julialang.org
- Slack and Zulip channels
- StackOverflow ([julia] tag)
- #JuliaLang, @JuliaLanguage
- reddit.com/r/Julia

3. Case studies

3.1 *Simulating 55,550,000 birds*

Reply | Reply List | Forward | Archive | Junk | Delete | More

From Chloe Bracis <cbracis@uw.edu>

Subject **[safsgrads] [Safs_quantitativeseminar] Quantitative Seminar: Charlotte Boyd, Nov. 18th, 12:30 in FSH 203** 11/15/2011 12:44 PM

To safs_quantitativeseminar@u.washington.edu

Good morning everyone,

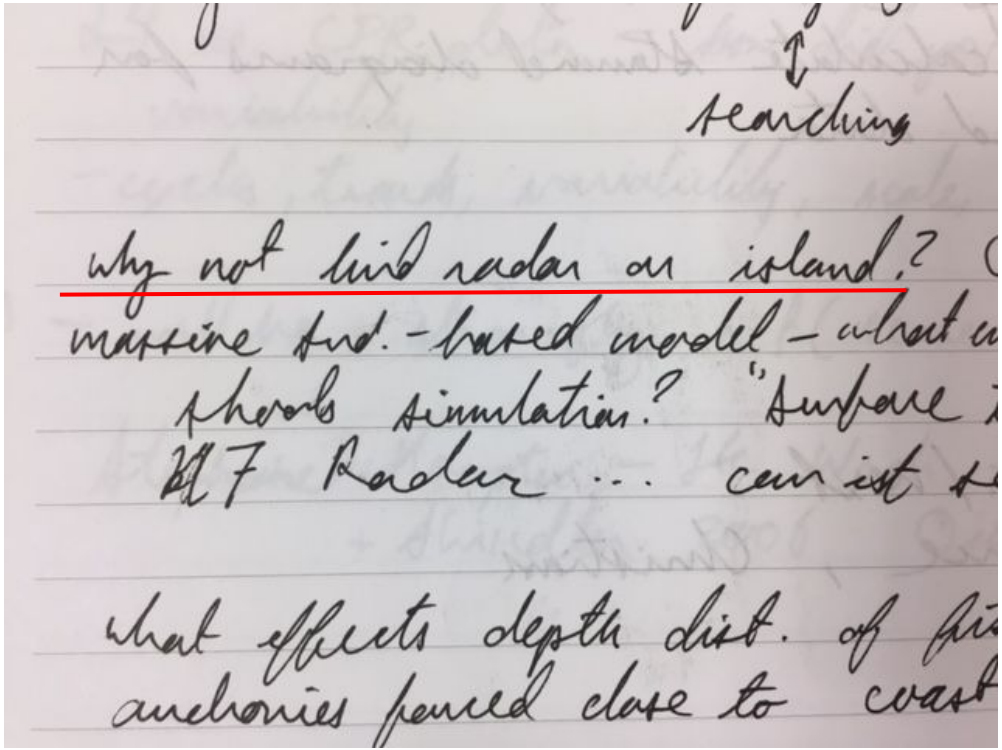
Please join us for this week's quantitative seminar on Friday, Nov. 18th from 12:30 to 1:30 in FSH 203.

Charlotte Boyd, SAFS, UW
A spatially-explicit individual-based foraging model for central place foragers

You can see the rest of this quarter's schedule on the website: http://fish.washington.edu/quantsem.
See you Friday!

Regards,
Chloe

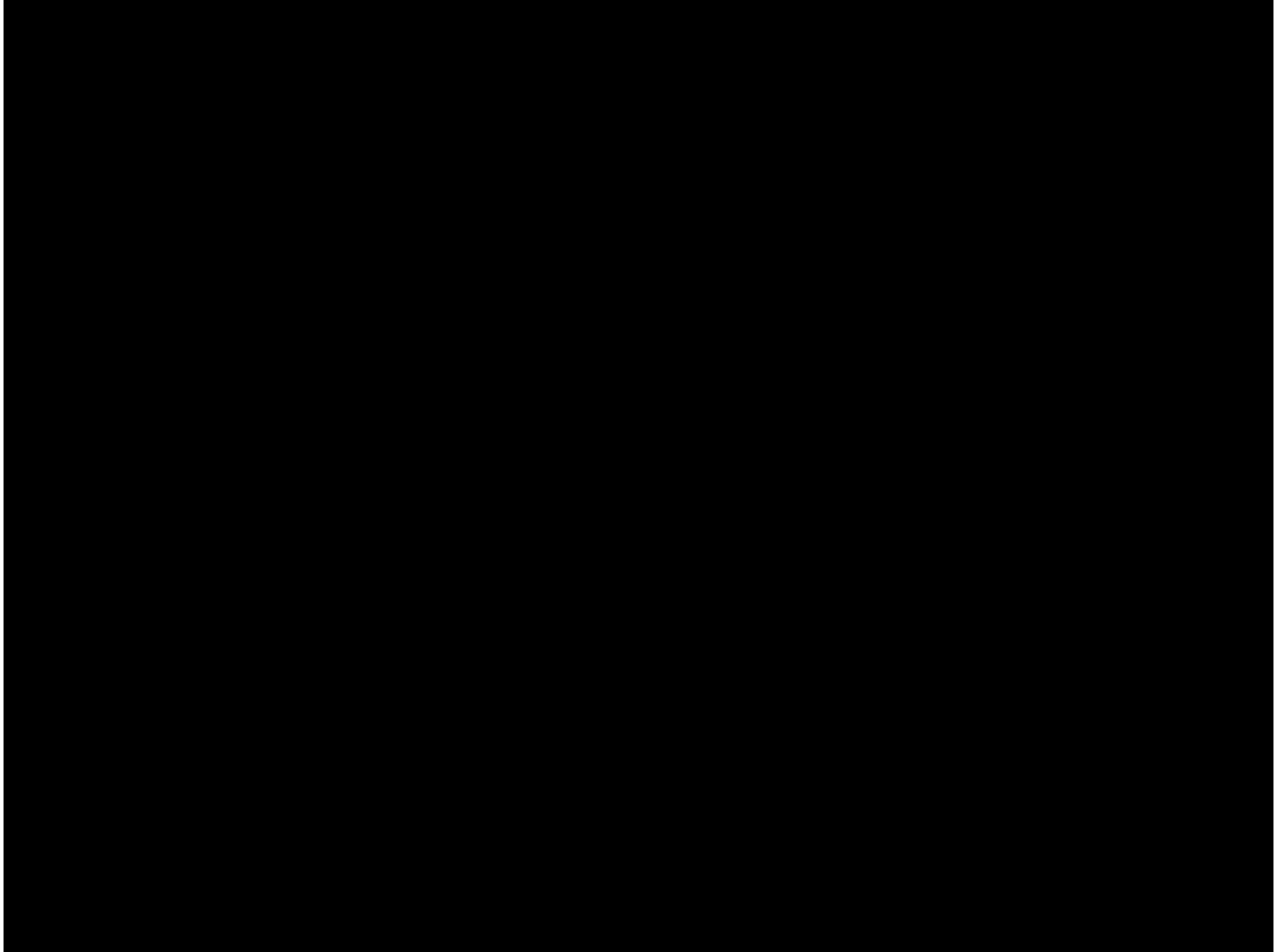
Abstract:
Reduced food availability has been identified as a threat to a number of globally threatened or near-threatened marine species, especially central place foragers in regions where large commercial fisheries target forage fish. The design of effective conservation and management responses depends on our understanding of how changes in the abundance and distribution of food resources impact foraging success. I will describe a spatially-explicit individual-based foraging model (IBFM) designed to investigate these questions, and present results for Peruvian Boobies (*Sula variegata*) and Guanay Cormorants (*Phalacrocorax bougainvillii*) foraging on a variable stock of Peruvian anchoveta (*Engraulis ringens*). The IBFM is designed to be a flexible and accessible tool that can be adapted to other central place foragers, including fishing vessels.



Great Gull Island, New York

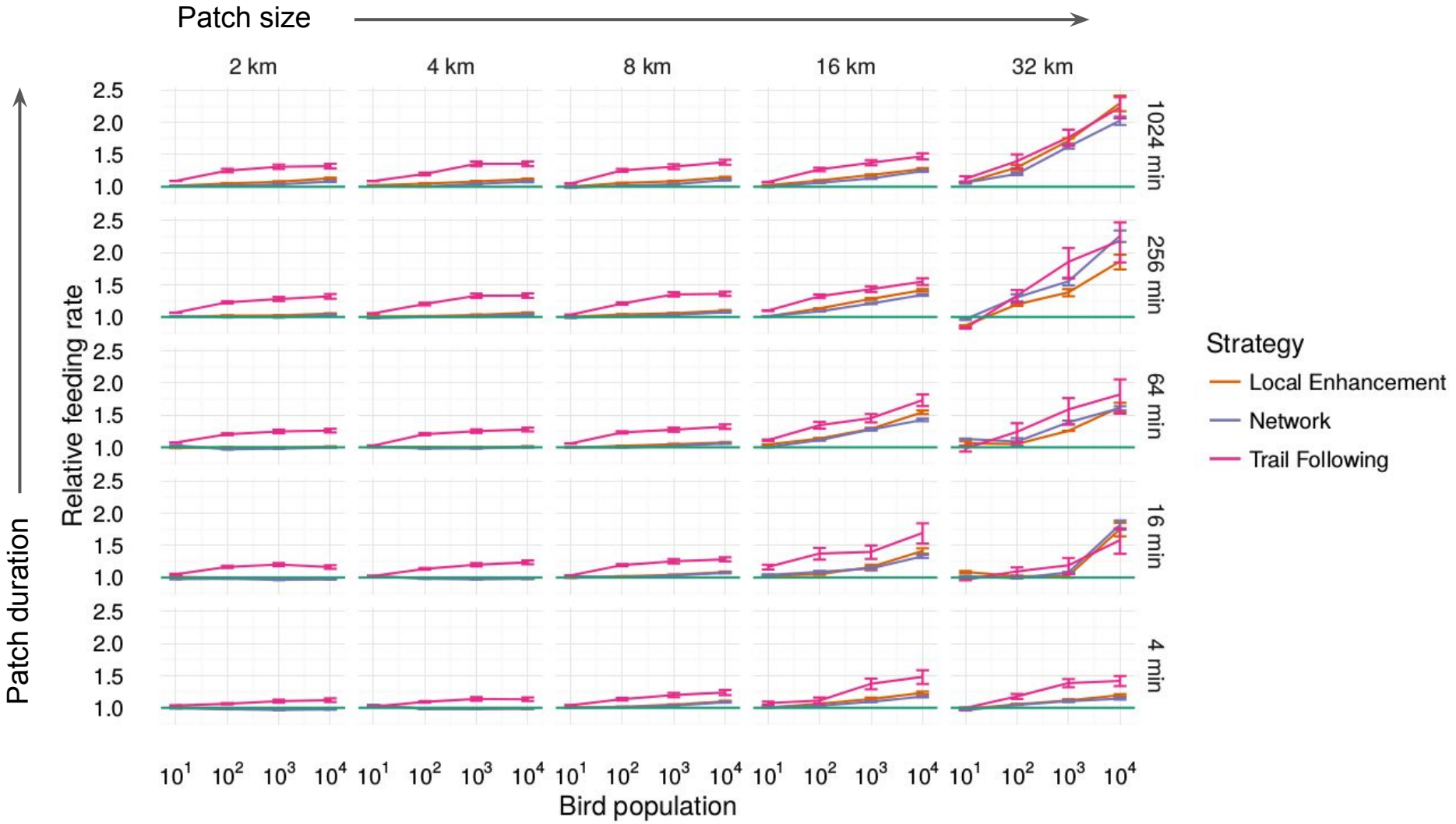
- Former Army fort
- Current AMNH research station
- Summer home of ~22,000 terns


Bird radar on island



4 foraging strategies
x 5 spatial scales
x 5 temporal scales
x 4 population sizes
x 200 replicates

80,000 simulations


Patch size
2 km
4 km
8 km
16 km
32 km
Patch duration
1024 min
256 min
64 min
16 min
4 min
Relative feeding rate
2.5
2.0
1.5
1.0
10^1
10^2
10^3
10^4
Bird population
Strategy
Local Enhancement
Network
Trail Following

Ecological Monographs, 91(1), 2021, e01429

Visual trail following in colonial seabirds: theory, simulation, and remote observations

Samuel S. Urmy

School of Marine and Atmospheric Sciences, Stony Brook University, 239 Montauk Highway, Southampton, New York 11968 USA

Citation: Urmy, S. S. 2021. Visual trail following in colonial seabirds: theory, simulation, and remote observations. Ecological Monographs 91(1):e01429. 10.1002/ecm.1429

Abstract. Seabirds have long been thought to exploit social information when searching for their prey, the distribution of which is often patchy and variable. The fact that most seabirds breed colonially has led to speculation that colonies serve as "information centers," allowing their inhabitants to learn about the distribution of food by observing or following other successful foragers, though this hypothesis is controversial and the evidence for it is mixed. However, several recent studies have documented behaviors that suggest some seabirds do exploit social orientation cues at or near their colonies in order to orient toward food. In this paper, I explore in-depth one such social orientation behavior, which I call "visual trail following." I derived a simple model of information transfer and showed that trail following should be favored over other commonly hypothesized foraging behaviors. An individual-based simulation model was then used to test this theoretical prediction against several other foraging strategies while varying prey patchiness and colony size. The model's results showed that trail following was the optimal strategy across a wide range of conditions. Finally, I used radar data recorded at a tern colony in coastal New York to demonstrate evidence for trail following in the movements of wild seabirds. These results show that trail following and similar behaviors are effective foraging strategies that are likely important for seabirds and other colonial animals.

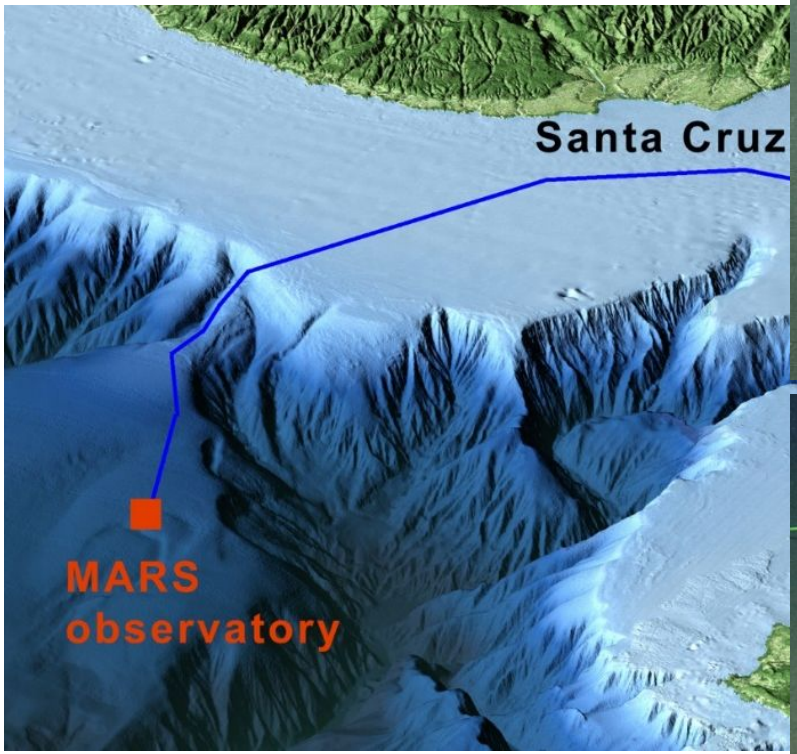
Key words: agent-based model; collective movement; Common Terns; fisheries acoustics; Great Gull Island; patchiness; pattern-oriented modeling; public information; radar ornithology.

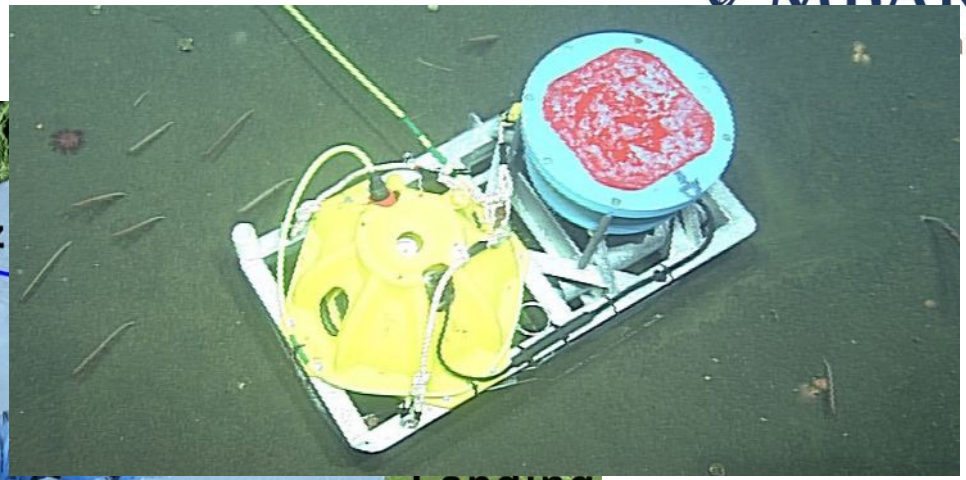
How did Julia help?

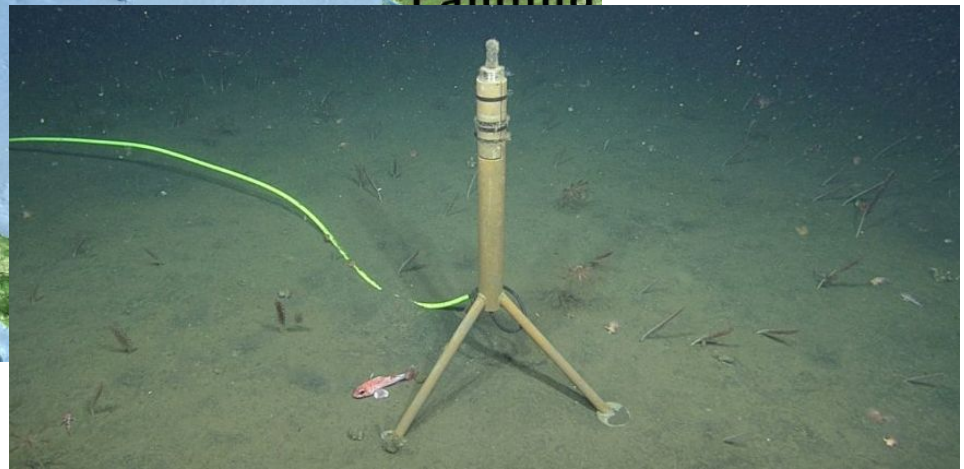
- Multiple dispatch
  - Easy to program bird types with different behaviors
- Speed
  - (Duh)
- Built-in parallel computing capabilities
  - *Identical* Julia code for 4-core desktop and 96-core cluster.

3.2 *Detecting 300,000,000 dolphin clicks*

Back to the bottom of Monterey Bay...







Dolphin foraging: easier heard than seen

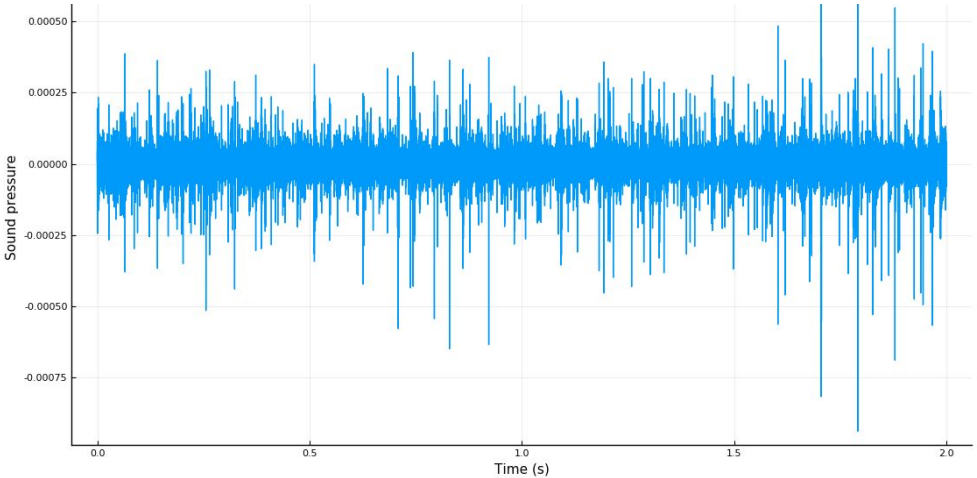
Common dolphin

Risso’s dolphin

Pacific white-sided dolphin

- All echolocate to find prey
- Short, loud, ultrasonic clicks

Searching for needles in 75 TB of hay



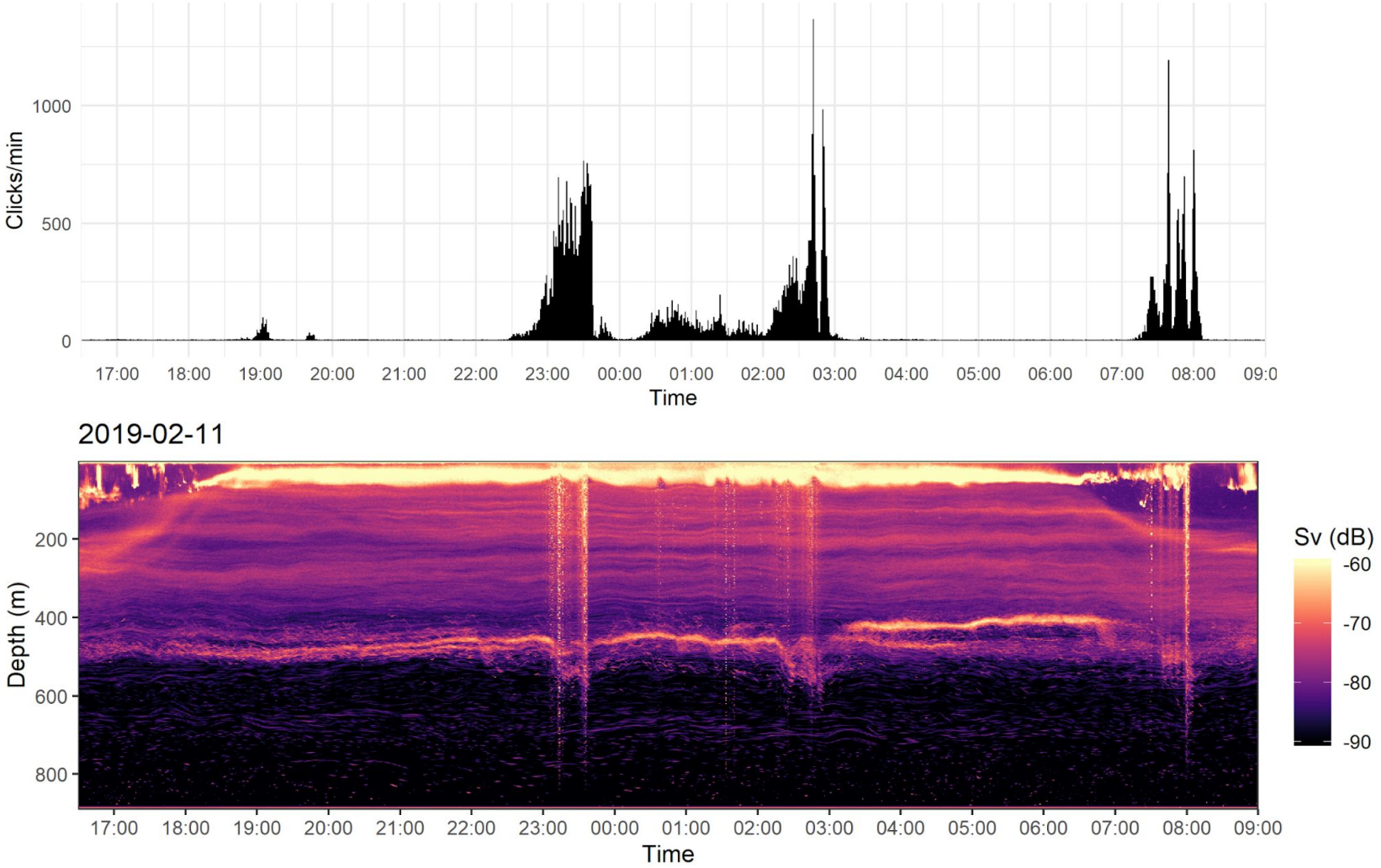
- Clicks ~ 1 μs long
- *Three years* of continuous recordings!
- “Teager energy operator” commonly used to detect clicks

$$T = x(t)^2 - x(t-1)\,x(t+1)$$

- Every language has a fast library for linear signal filtering…
  - But Teager operator is nonlinear!
- Existing passive acoustic tools either expensive, hard to use, or not up to four 75 TB of continuous data

How did Julia help?

- Speed
  - Teager operator is trivial to code, Julia lets it be fast
- Great digital signal processing library (*DSP.jl*)
- Again, parallelism
  - Easy to process data simultaneously on all cores
- Ended up detecting ~300,000,000 clicks

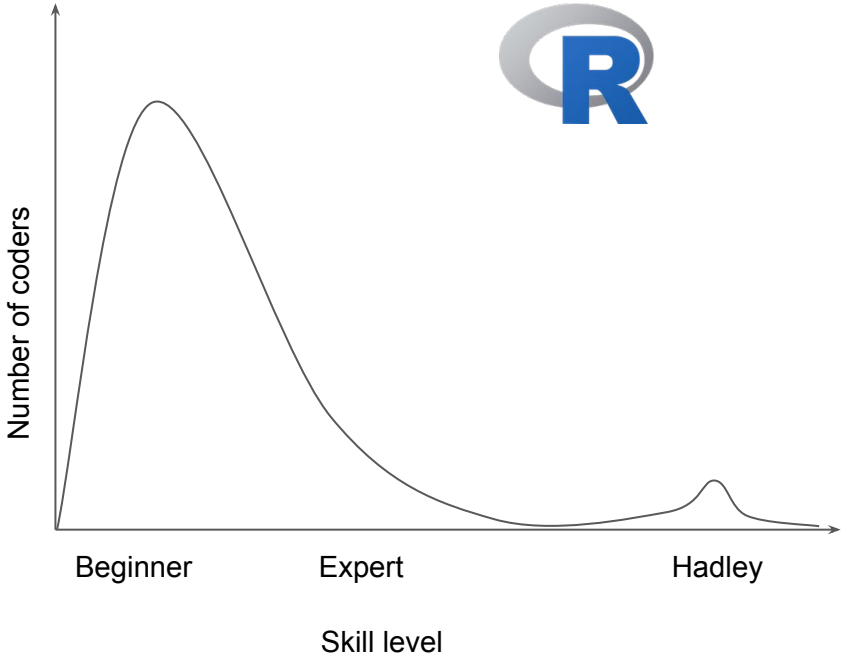

Clicks/min
1000
500
0
17:00
18:00
19:00
20:00
21:00
22:00
23:00
00:00
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
09:0
Time
2019-02-11
Depth (m)
200
400
600
800
17:00
18:00
19:00
20:00
21:00
22:00
23:00
00:00
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
09:00
Time
Sv (dB)
-60
-70
-80
-90

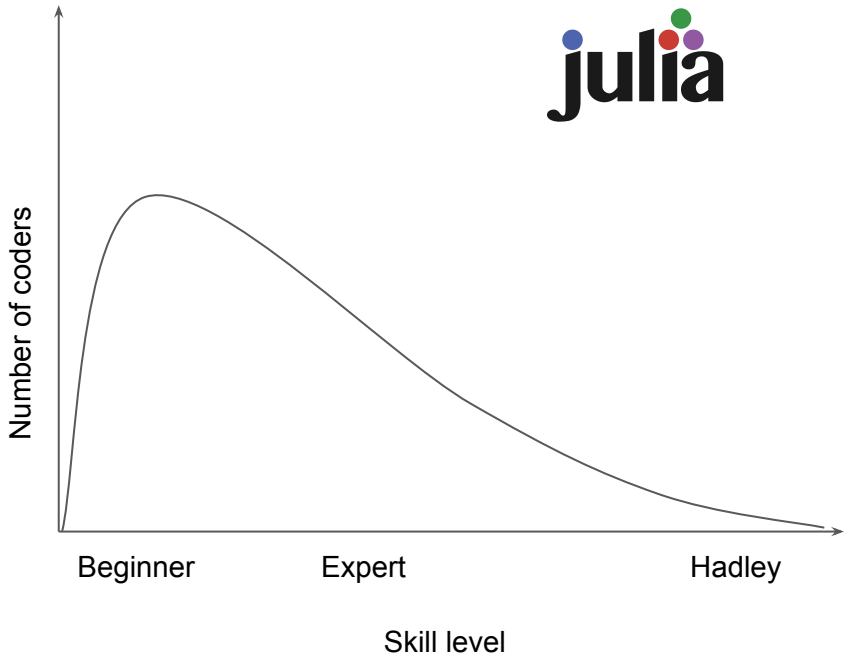
4. Concluding thoughts and advice

What’s bad about Julia

- Relatively small community and user base
  - Harder to find answers on StackOverflow, etc.
  - Less likely to find a local expert
- Young language, growing and developing fast
  - Smaller ecosystem than R
  - Some rough edges, sparse docs, breaking changes
- Compilation time: first time using a function can be slow
  - A.k.a “time-to-first-plot problem” (TTFPP)
  - Current priority, improving fast
- Learning curve (moreso for advanced coders)
  - Types and multiple dispatch make Julia great, but take some (un)learning

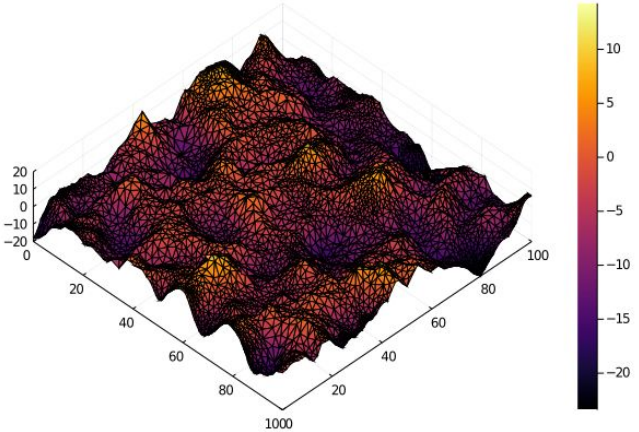
A more continuous learning curve

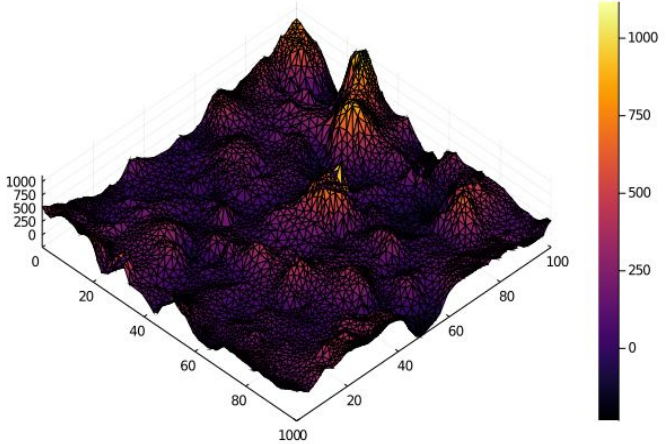




Missing tools for quantitative ecologists

- No out-of-the-box TMB, VAST, or INLA equivalents
  - Pieces all there, just need to put them together
- Works-in-progress (myself and John Best):
  - https://github.com/ElOceanografo/SPDEPrecisionMatrices.jl
  - https://github.com/ElOceanografo/GaussianMarkovRandomFields.jl
  - https://github.com/ElOceanografo/MarginalLogDensities.jl





Looking forwards

- Should you learn Julia?
  - Happy with your current tools? Why would you?
  - Intrigued but unconvinced? Why not?
  - Pushing the limits and dissatisfied? You owe it to yourself!
- Yes, you *can* learn a second language!

- Julia has a lot to offer our field
  - One language from basic plots to high-performance modeling on 1,000s of cores
  - Faster, easier algorithm development and testing
  - Eroding barrier between mechanistic and statistical modeling
  - Closer collaborations with climate/ocean scientists?
- Quantitative ecologists have a lot to offer Julia


```
library(tRucks)
```

```
#include <stdlib.h>
#include <jigsaw.h>
#include <lathe.h>
```

using Trucks

using Wheels
using Windows
using Technic

sam.urmy@noaa.gov