Julia language

outline

- Faster
- Much less code (sometimes)
- Creativity

- Generic programming
- Composable

2 apologies

Question 1: where does ML-in-Julia shine?

(A) Runtime speed.

Standard Julia story there, really. This is most noticable compared to PyTorch, at least when doing operations that aren't just BLAS/cuDNN/etc.-dominated. (JAX is generally faster in my experience.)

(B) Compilation speed.

No, really! Julia is substantially faster than JAX on this front. (It *really* doesn't help that JAX is essentially a compiler written in Python. JAX is a lovely framework, but IMO it would have been better to handle its program transformations in another language.)

It's been great watching the recent progress here in Julia.

(C) Introspection.

Julia offers tools like `@code_warntype`, `@code_native` etc. Meanwhile JAX offers almost nothing. (Once you hit the XLA backend, it becomes inscrutable.) For example I've recently had to navigate some serious performance bugs in the XLA compiler, essentially by trial-and-error.

(D) Julia is a programming language, not a DSL.

JAX/XLA have limitations like not being able to backpropagate while loops, or being able to specify when to modify a buffer in-place. As a "full" programming language, Julia doesn't share these limitations.

Julia offers native syntax, over e.g. `jax.lax.fori_loop(...)`.

(PyTorch does just fine on this front, though.)

Imagine working in an environment that has both the elegance of JAX (Julia has arrays, a jit compiler, and vmap all built-in to the language!) and the usability of PyTorch (Julia is a language, not a DSL!) Julia still has issues to fix, but come and help pitch in if this is a dream you want to see become reality.

https://discourse.julialang.org/t/state-of-machine-learning-in-julia/74385/4

Introduction: autodiff, adoption

- 2010: "Matlab has a huge number of libraries for numerics/machine learning. Python will never catch up."

- 2020: "Python has a huge number of libraries for numerics/machine learning. Julia will never catch up."

hold rebuttals till end?

Why Julia

I ran into Julia about five years ago. Back to those days, I was in love with Python which greatly increased my coding productivity. However, I was suffering from its slowness. Before Python, I developed scientific software using C++ which is fast but it is too complicated and sometimes it made me crazy to implement a specific numerical algorithm. You can check out those software packages I have developed **here**. I was wondering then if there was a programming language which combines the performance of C++ and productivity of Python. I tried to search terms like "speed and scripting programming language" in Google and I found Julia, which was in its very early stage but showed its great potential. After that, I payed special attention on it. Now Julia is in version 1.3. After ten years intensive developing, it matures into a stable language. Therefore I decide to give it a try. `Scattering.jl` is my first package written in Julia.

Besides its speed (check out a comparison of performance of various popular programming languages **here**), what attracts me most are listed below:

- The syntax is even more clean and concise than Python.
- Julia's mathematical syntax makes it an ideal way to express algorithms just as they are written in papers, owing to the support of Unicode characters and other syntax sugar added by the language. This drastically increase the readability and maintainability of the code.
- Multiple dispatch mechanism (allowing multiple functions to have the same name) allows you to write reusable codes more easily. And the functionality is smooth to be extended by others.
- High level support for **GPU computing** and parallel programming.
- Production ready numerical and machine learning packages: the state-of-the-art differential equations ecosystem (**DifferentialEquations.jl**), optimization tools (**JuMP.jl** and **Optim.jl**), iterative linear solvers (**IterativeSolvers.jl**), a robust framework for Fourier transforms (**AbstractFFTs.jl**), and powerful tools for deep learning with **automatic differentiation** and **GPU acceleration** (**Flux.jl**).

Nature published an article to promote Julia: **Julia: come for the syntax, stay for the speed**.

Let's get a little bit of taste of Julia:

Usage

Here is a simple example to call Python's `math.sin` function:

```
using PyCall
math = pyimport("math")
math.sin(math.pi / 4) # returns ≈ 1/√2 = 0.70710678...
```

Type conversions are automatically performed for numeric, boolean, string, IO stream, date/period, and function types, along with tuples, arrays/lists, and dictionaries of these types. (Python functions can be converted/passed to Julia functions and vice versa!) Other types are supported via the generic PyObject type, below.

Multidimensional arrays exploit the NumPy array interface for conversions between Python and Julia. By default, they are passed from Julia to Python without making a copy, but from Python to Julia a copy is made; no-copy conversion of Python to Julia arrays can be achieved with the `PyArray` type below.

Keyword arguments can also be passed. For example, matplotlib's pyplot uses keyword arguments to specify plot options, and this functionality is accessed from Julia by:

```
plt = pyimport("matplotlib.pyplot")
x = range(0;stop=2*pi,length=1000); y = sin.(3*x + 4*cos.(2*x));
plt.plot(x, y, color="red", linewidth=2.0, linestyle="--")
plt.show()
```

```
function mysum(A)
    thesum = 0
    for i=1:length(A)
        thesum += A[i]
    end
    return thesum
end
```

tensorflow: tf.reduce_sum(tf.multiply(tf.expand_dims(a,-1), w), axis=0)

```
julia> i*A*x
```

```
ξ = 1 / ((1/(1-λ)) + (y₀' * Bᵦ₀ * y₀))   # gradient scaling
```

```
Bᵦ₁ = A-(1-λ)*η*A*A/(1+(1-λ)*η*tr(A))   # updated inverse covariance matrix
```

with $\beta \in [0,1]$ often set to $\beta = 1 \times 10^{-4}$. Figure 4.1 illustrates this condition. If $\beta = 0$, then any decrease is acceptable. If $\beta = 1$, then the decrease has to be at least as much as what would be predicted by a first-order approximation.

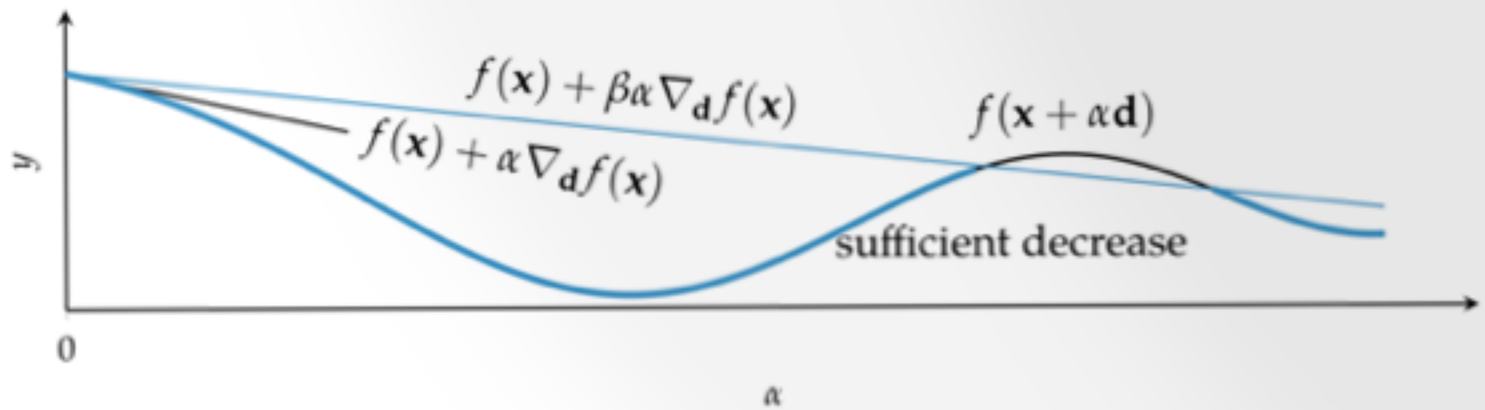


Figure 4.1. The sufficient decrease condition, the first Wolfe condition, can always be satisfied by a sufficiently small step size along a descent direction.

If $\mathbf{d}$ is a valid descent direction, then there must exist a sufficiently small step size that satisfies the sufficient decrease condition. We can thus start with a large step size and decrease it by a constant reduction factor until the sufficient decrease condition is satisfied. This algorithm is known as *backtracking line search*[6] because of how it backtracks along the descent direction. Backtracking line search is shown in figure 4.2 and implemented in algorithm 4.2. We walk through the procedure in example 4.2.

[6] Also known as *Armijo line search*, L. Armijo, "Minimization of Functions Having Lipschitz Continuous First Partial Derivatives," *Pacific Journal of Mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

```
function backtracking_line_search(f, ∇f, x, d, α; p=0.5, β=1e-4)
    y, g = f(x), ∇f(x)
    while f(x + α*d) > y + β*α*(g⋅d)
        α *= p
    end
    α
end
```

Algorithm 4.2. The backtracking line search algorithm, which takes objective function `f`, its gradient `∇f`, the current design point `x`, a descent direction `d`, and the maximum step size `α`. We can optionally specify the reduction factor `p` and the first Wolfe condition parameter `β`.

Note that the cdot character `⋅` aliases to the `dot` function such that `a⋅b` is equivalent to `dot(a,b)`. The symbol can be created by typing `\cdot` and hitting tab.

The first condition is insufficient to guarantee convergence to a local minimum. Very small step sizes will satisfy the first condition but can prematurely converge. Backtracking line search avoids premature convergence by accepting the largest

```julia
function backtracking_line_search(f, ∇f, x, d, α; p=0.5, β=1e-4)
    y, g = f(x), ∇f(x)
    while f(x + α*d) > y + β*α*(g⋅d)
        α *= p
    end
    α
end
```

latex/unicode variable names

In [6]:
```
function discriminantregularizer(y,labels,m; λ=LAMBDA, η=ETA, update=UPDATE)
    y = vec(y)
    M = size(m.μ,2)
    β = labels[1]      # β(n) class label for the nth sample
    μᵦ₀ = m.μ[:,β]     # μ[β(n)](n-1) exponentially weighted mean of class β(n) before the nth sam
ple
    Bᵦ₀ = m.B[:,:,β]  # B[β(n)](n-1) exponentially weighted inverse covariance matrix of class β(
n) before the nth sample
    μᵦ₁  = λ * μᵦ₀ + (1-λ) * y
    y₀ = y - μᵦ₀       # ybar[L-1](n) the centralized feature vector
    z = Bᵦ₀ * y₀       # unscaled gradient
    ξ = 1 / ((1/(1-λ)) + (y₀' * Bᵦ₀ * y₀))  # gradient scaling
    A = (1/λ)*(Bᵦ₀ - z*z'*ξ)
    Bᵦ₁ = A-(1-λ)*η*A*A/(1+(1-λ)*η*tr(A))  # updated inverse covariance matrix
    ∇g=0*y           # 0*y matches y's array type, zeros(size(y)) may not.
    g = m.g[1]
    for j=1:M
        if (j!=β)
            μⱼ=m.μ[:,j]
            Bⱼ=m.B[:,:,j]
            Δμⱼ₀=μᵦ₀-μⱼ
            Δμⱼ₁=μᵦ₁-μⱼ
            αⱼ₀=(Δμⱼ₀'*Bᵦ₀*Δμⱼ₀)
            αⱼ₁=(Δμⱼ₁'*Bᵦ₁*Δμⱼ₁)
            ζⱼ₀=(Δμⱼ₀'*Bⱼ*Δμⱼ₀)
            ζⱼ₁=(Δμⱼ₁'*Bⱼ*Δμⱼ₁)
            g=g-log(αⱼ₁)+log(αⱼ₀)-log(ζⱼ₁)+log(ζⱼ₀)
            qⱼ=Bᵦ₁*Δμⱼ₁
            ∇g+=Bⱼ*Δμⱼ₁/(Δμⱼ₁'*Bⱼ*Δμⱼ₁)+qⱼ*(1-qⱼ'*(y-μᵦ₁))/αⱼ₁
        end
    end

    if training()  # Store ∇g if differentiating
        m.∇g .= -2*(1-λ)*∇g
    end

    if update      # Update state if specified
        m.g[1] = g
        m.B[:,:,β] .= Bᵦ₁
        m.μ[:,β] .= μᵦ₁
    end
```

Symbolic derivative in 2D

Let's see what happens when we perturb by small amounts δ in the x direction and ϵ in the y direction around the point (a, b):

p = 0.07

$[a, b, \delta, \epsilon]$

```
@variables a, b, δ, ϵ
```

image = $\left[a + \delta + 0.07(b + \epsilon)^2, \quad b + \epsilon + 0.07(a + \delta)^2\right]$

```
image = expand.(T(p)( [ (a + δ), (b + ϵ) ] ))
```

```
2×2 Matrix{Text{Num}}:
 1.0    0.14b
 0.14a  1.0
```

```
jacobian(T(p), [a, b]) .|> Text
```

$[\delta + 0.14b\epsilon, \quad \epsilon + 0.14a\delta]$

```
jacobian(T(p), [a, b]) * [δ, ϵ]
```

Symbolics.Num[

1: $\delta - 0.07b^2 + 0.07(b + \epsilon)^2$

2: $\epsilon - 0.07a^2 + 0.07(a + \delta)^2$

]

```
image - T(p)([a, b])
```

$\left[-0.07b^2 + 0.07(b + \epsilon)^2 - 0.14b\epsilon, \quad -0.07a^2 + 0.07(a + \delta)^2 - 0.14a\delta\right]$

```
simplify.(expand.(image - T(p)([a, b]) - jacobian(T(p), [a, b]) * [δ, ϵ]))
```

Tools of the trade: Julia and Python

- Julia: Simple, unified interface with autodiff. Decent error messages.
  - More support from course materials.
- Python: Allowed, but initially can't use frameworks' network layers, initializers, or optimizers.
  - Suggested: Jax, PyTorch
  - Gotchas: need to learn both Python and a framework on top. Bad error messages.

```julia
function (a::Dense)(x::AbstractVecOrMat)
  W, b, σ = a.weight, a.bias, a.σ
  return σ.(W*x .+ b)
end
```

Question about source code of pytorch

linyu Aug '18

Where can I find the source code of torch.mm?

1

created Aug '18 | last reply Aug '18 | 2 replies | 348 views | 2 users | 2 likes | 1 link

SimonW Simon Wang Aug '18

It eventually dispatches to https://github.com/pytorch/pytorch/blob/2e0dd8690320fb1a7ecd548730824c1610207179/aten/src/ATen/native/LinearAlgebra.cpp#L136-L148 58, which calls blas gemm.

```julia
struct Dense{F,S,T}
  W::S
  b::T
  σ::F
end

Dense(W, b) = Dense(W, b, identity)

function Dense(in::Integer, out::Integer, σ = identity;
               initW = glorot_uniform, initb = zeros)
  return Dense(initW(out, in), initb(out), σ)
end

@functor Dense

function (a::Dense)(x::AbstractArray)
  W, b, σ = a.W, a.b, a.σ
  σ.(W*x .+ b)
end
```

```julia
mutable struct ADAM
  eta::Float64
  beta::Tuple{Float64,Float64}
  state::IdDict
end

ADAM(η = 0.001, β = (0.9, 0.999)) = ADAM(η, β, IdDict())

function apply!(o::ADAM, x, Δ)
  η, β = o.eta, o.beta
  mt, vt, βp = get!(o.state, x, (zero(x), zero(x), β))
  @. mt = β[1] * mt + (1 - β[1]) * Δ
  @. vt = β[2] * vt + (1 - β[2]) * Δ^2
  @. Δ =  mt / (1 - βp[1]) / (√(vt / (1 - βp[2])) + ϵ) * η
  o.state[x] = (mt, vt, βp .* β)
  return Δ
end
```

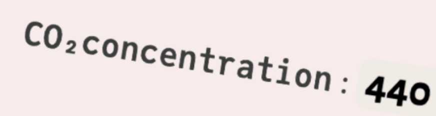
Flux: The Julia Machine Learning Library

Flux is a library for machine learning. It comes "batteries-included" with many useful tools built in, but also lets you use the full power of the Julia language where you need it. We follow a few key principles:

- **Doing the obvious thing**. Flux has relatively few explicit APIs for features like regularisation or embeddings. Instead, writing down the mathematical form will work – and be fast.
- **You could have written Flux**. All of it, from LSTMs to GPU kernels, is straightforward Julia code. When in doubt, it's well worth looking at the source. If you need something different, you can easily roll your own.
- **Play nicely with others**. Flux works well with Julia libraries from data frames and images to differential equation solvers, so you can easily build complex data processing pipelines that integrate Flux models.

computing more accessible and fun.

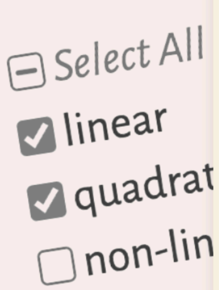
Simple, reactive programming environment for Julia



REACTIVITY

Interactivity as a fundamental principle

Just like a spreadsheet, Pluto understands variable links between code cells, and will re-run a cell when a dependency changes.



Reactivity means interactivity

Your programming environment becomes interactive by splitting your code into multiple cells! Changing one cell **instantly shows effects** on all other cells, giving you a fast and fun way to experiment with your model.

In this example, changing the parameter A and running the first cell will directly re-evaluate the second cell and display the new plot.

```
2×2 Array{Float64,2}:
 -0.4  -1.0
  1.0   0.41
```

```
A = [ -.4 -1
       1   .42]
```

Sliders, buttons and more!

Pluto lets you *bind* a Julia variable to an GUI element. Moving a slider from 0 to 100 will actually change one of your variables from `0` to `100`! Combined with reactivity, this is a very powerful tool!

```
@bind x html"<input type=range>"
```

```
[35, 70, 350]
```

```
[x, x*2, x*10]
```

It's that simple to make your Julia code come to life! That's because reactivity and widget interactivity are the same concept! Less to learn, more to discover.

The package PlutoUI.jl contains lots of common widgets like sliders, textfields and buttons. Need something different? PlutoUI.jl was made by us, but anyone can create their own special widgets! We give you full

6

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

Ain't no party like a programming language virtual conference party

I've been running into a lot of happy and excited scientists lately. "Running into" in the virtual sense, of course, as conferences and other opportunities to collide with scientists in meatspace have been all but eliminated. Most scientists believe in the germ theory of disease.

Anyway, these scientists and mathematicians are excited about a new tool. It's not a new particle accelerator nor a supercomputer. Instead, this exciting new tool for scientific research is… a computer language.

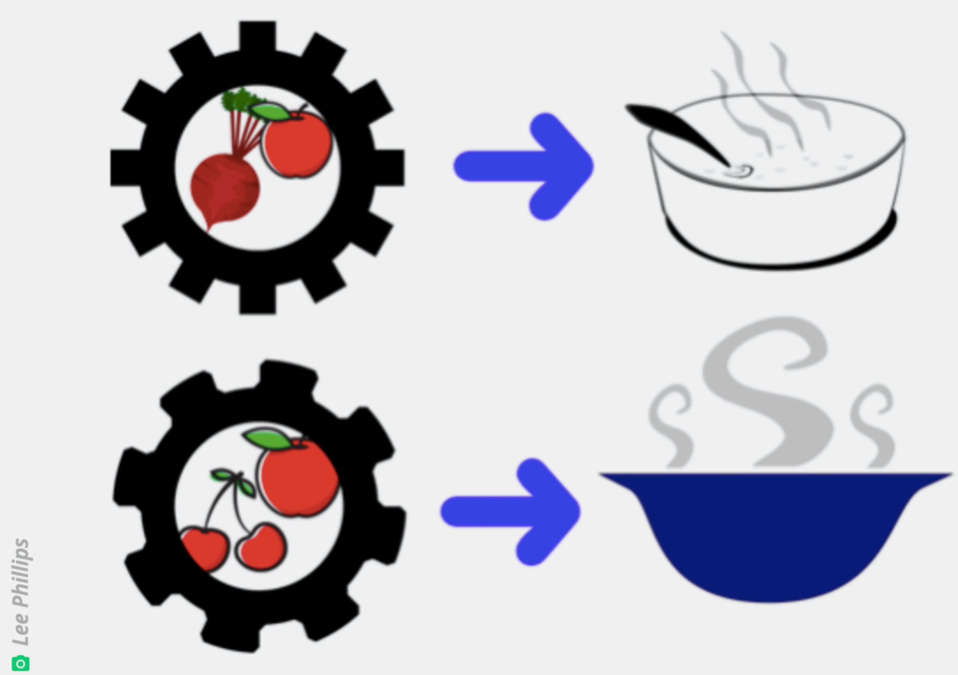
The Expression Problem, via extended analogy

The concept of the “expression problem” arises in the study of the design of computer languages. It is part of the domain of computer science, and so the existing explanations of its meaning, implications, and the various ways around the problem tend to be abstract and rely on a specialized terminology. But we can do better. It’s possible to describe all the issues involved by using an analogy to cooking.

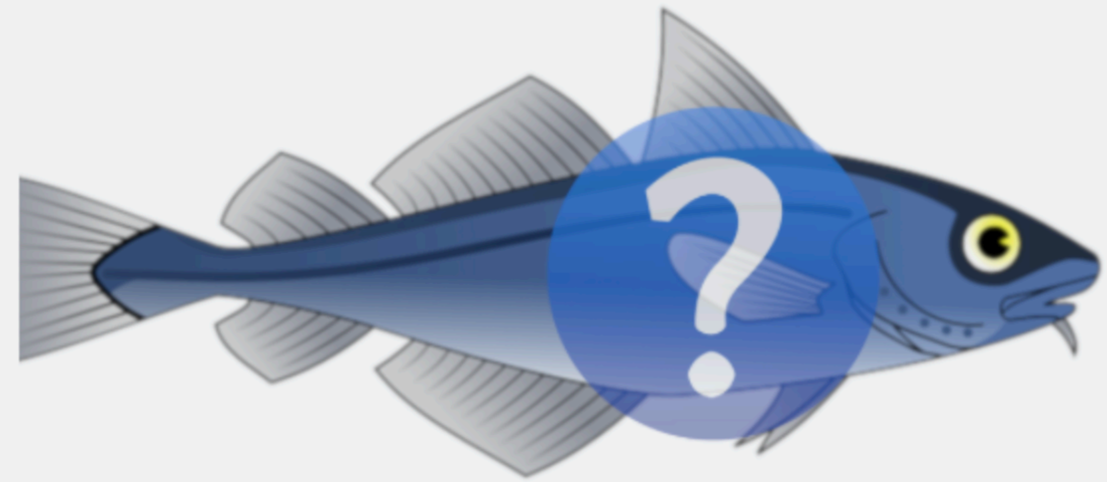
The computer science terms that we would like to analogize are *functions/programs, data types, and libraries/modules/packages*. Briefly, functions or programs are procedures for taking some input, doing something to it, and producing some output. Data types are collections of numbers or other information, which may have various kinds of structure, that the functions operate on. Libraries, etc., are collections of functions, along with descriptions of the data types that they work with, bundled together to perform a set of related tasks. An example of a library would be a set of functions for drawing graphs. The individual functions in the library might be for drawing different types of graphs, like pie charts and histograms. The data type for a pie chart, for example, would be a list of pairs of elements, with the first being a word or phrase and the second a percentage.

For anyone who has spent time in the kitchen creating dishes from recipes, this analogy will be fairly direct and natural. The library or package becomes the recipe book; imagine a somewhat focused book about making desserts, or soups, for example. The functions or programs can be thought of either as complete recipes for making a dish or as techniques or procedures, such as how to sauté. We can visualize them as gears, as they are the machinery for processing raw ingredients. The data types are the raw ingredients in this exercise.

Imagine our recipe book is organized in such a way that recipes only work with certain ingredients. For example, you can look up “how to sauté” and find the procedure, the set of steps, for sautéing onions or sautéing shrimp. All these procedures are *different,* as they use different ingredients. If recipes work like a computer language, the ingredient lists are part of, in fact enclosed within, the recipes.

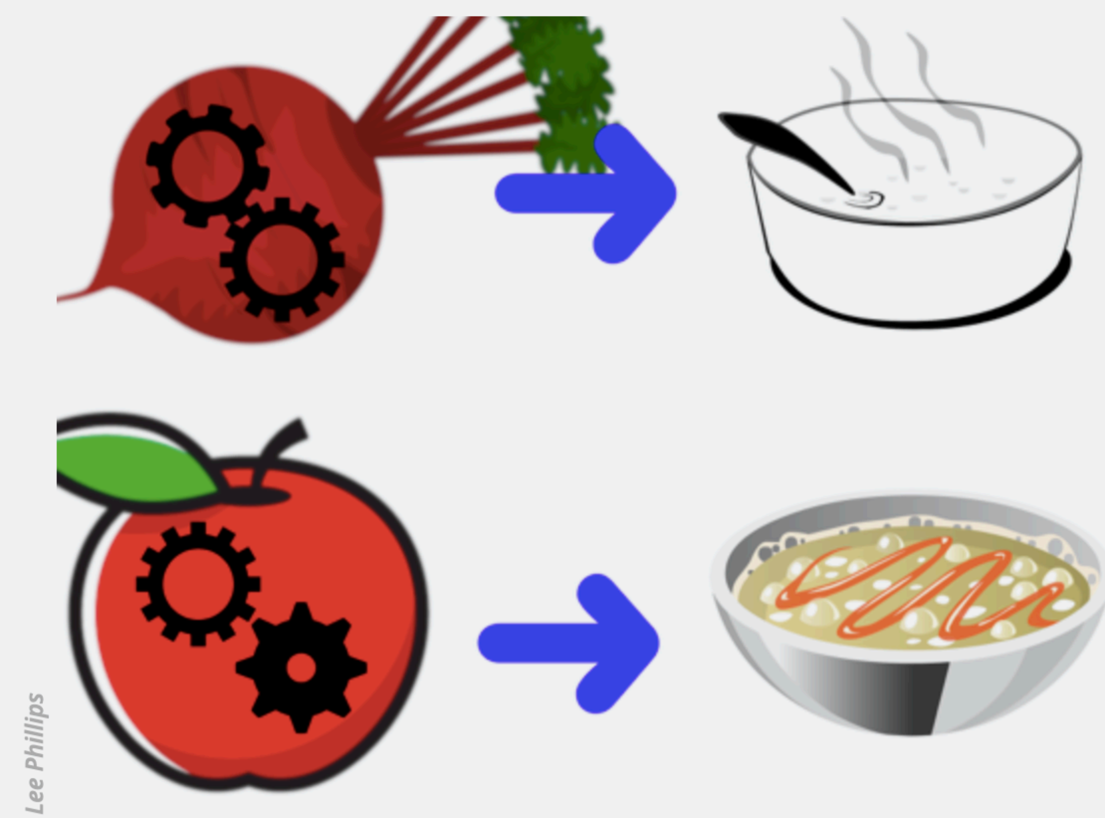


Recipes that only work with specified ingredients.



A new ingredient

There is more than one way to organize a recipe book, however. What if it were organized arou ingredients, rather than around methods of cooking? For each ingredient, there would be a se techniques or methods that go with it. Continuing with our iconography, this could be represented with this picture:



adoption

The rapid adoption of Julia, the open source, high level programing language with roots at MIT, shows no sign of slowing according to data from Julialang.org. In 2020, the **number of downloads jumped 87 percent to more than 24 million** (2020 v. 2019) and the number of available packages rose 73 percent to roughly 4800. Jan 13, 2021

Julia Update: Adoption Keeps Climbing; Is It a Python Challenger?

By John Russell

January 13, 2021

The rapid adoption of Julia, the open source, high level programing language with roots at MIT, shows no sign of slowing according to data from Julialang.org. In 2020, the number of downloads jumped 87 percent to more than 24 million (2020 v. 2019) and the number of available packages rose 73 percent to roughly 4800. Last year (2019 v. 2018) the number of downloads jumped 77 percent. In the most recent TIOBE index, Julia jumped from #47 to #23 and TIOBE CEO Paul Jansen said Julia is the top candidate to jump into the top 20 (used languages) next year.

Julia is hot.

“

That’s one of the things that makes Julia so powerful in the solution of these problems [...] This integration gives Julia an advantage over other languages [...] we have been able to develop these solutions in a very short period of time:

León Alday, molecular modeling

“

Julia is really the language that allows such a project to exist:

George Datseris, Dr. Watson, a scientific assistant

“

Julia is a joy to program in:

Mauro Werder, Glacier ice thickness

“

The Julia language [...] is a particularly agile tool:

Valeri Vasquez, Disease vector dynamics

“

Julia was the obvious choice:

Rafael Schouten, Spatial simulations

“

[Julia allows] me to harness tools from across disciplines to advance cancer research:

Meghan Ferrall-Fairbanks, Tumor dynamics

“

This work has been very nice to do in Julia because of the nice abstractions that allow very general code:

Vilim Štih, Zebrafish brain dynamics

“

SemiseparableMatrices.jl

A Julia package to represent semiseparable and almost banded matrices

build passing

codecov 92%

`SemiseparableMatrix`

A semiseparable matrix of semiseparability rank `r` has the form

```
tril(A,-l-1) + triu(B,u+1)
```

e can construct a semiseparable matrix as follows:

```
FillArrays
```

HierarchicalMatrices.jl

build passing build passing

This package provides a flexible framework for hierarchical data types in Julia.

Create your own hierarchical matrix as simply as:

```
julia> using HierarchicalMatrices

julia> @hierarchical MyHierarchicalMatrix LowRankMatrix Matrix
```

The invocation of the `@hierarchical` macro creates an abstract supertype `AbstractMyHie`
`AbstractMatrix{T}` and the immutable type `MyHierarchicalMatrix`, endowing it with fielc
`HierarchicalMatrixblocks`, `LowRankMatrixblocks`, `Matrixblocks`, and a matrix of integ
which type of block is active. The package comes pre-loaded with a `HierarchicalMatrix`.

See the example on speeding up the matrix-vector product with Cauchy matrices.

PaddedMatrices

docs stable | docs latest | build error | build unknown | codecov 17%

Usage

This library provides a few array types, as well as pure-Julia matrix multiplication.

The native types are optionally statically sized, and optionally given padding to ensure that all col
following chart shows benchmarks on a 10980XE CPU, comparing:

- `SMatrix` and `MMatrix` multiplication from StaticArrays.jl.
- `FixedSizeArray` from this library without any padding.
- `FixedSizeArray` from this library with padding, named `PaddedArray` in the legend.
- The base `Matrix{Float64}` type, using the `PaddedMatrices.jmul!` method.

BlockDiagonals.jl

docs stable | docs dev | build passing | codecov 95%

Functionality for working efficiently with block diagonal matrices.

`BlockDiagonals.BlockDiagonal` — *Type*.

```
BlockDiagonal{T, V<:AbstractMatrix{T}} <: AbstractMatrix{T}
```

A matrix with matrices on the diagonal, and zeros off the diagonal.

LazyBandedMatrices.jl

A Julia package for lazy banded matrices

build passing

codecov 67%

This package supports lazy banded and block-banded matrices, for exam

```
julia> using LazyBandedMatrices, La

julia> A = brand(10,10,1,1);

julia> ApplyMatrix(*, A, A)
10×10 ApplyArray{Float64,2,typeof(*
 0.191109   0.379118   0.318899   ·
 0.329746   0.728074   1.12126    0.
 0.0341854  0.138194   0.95911    0.
  ·         0.0561613  0.823235   1.
```

BlockBandedMatrices.jl

A Julia package for representing block-block-banded matrices and banded-block-banded matrices

build passing build passing codecov 71%

docs stable docs latest

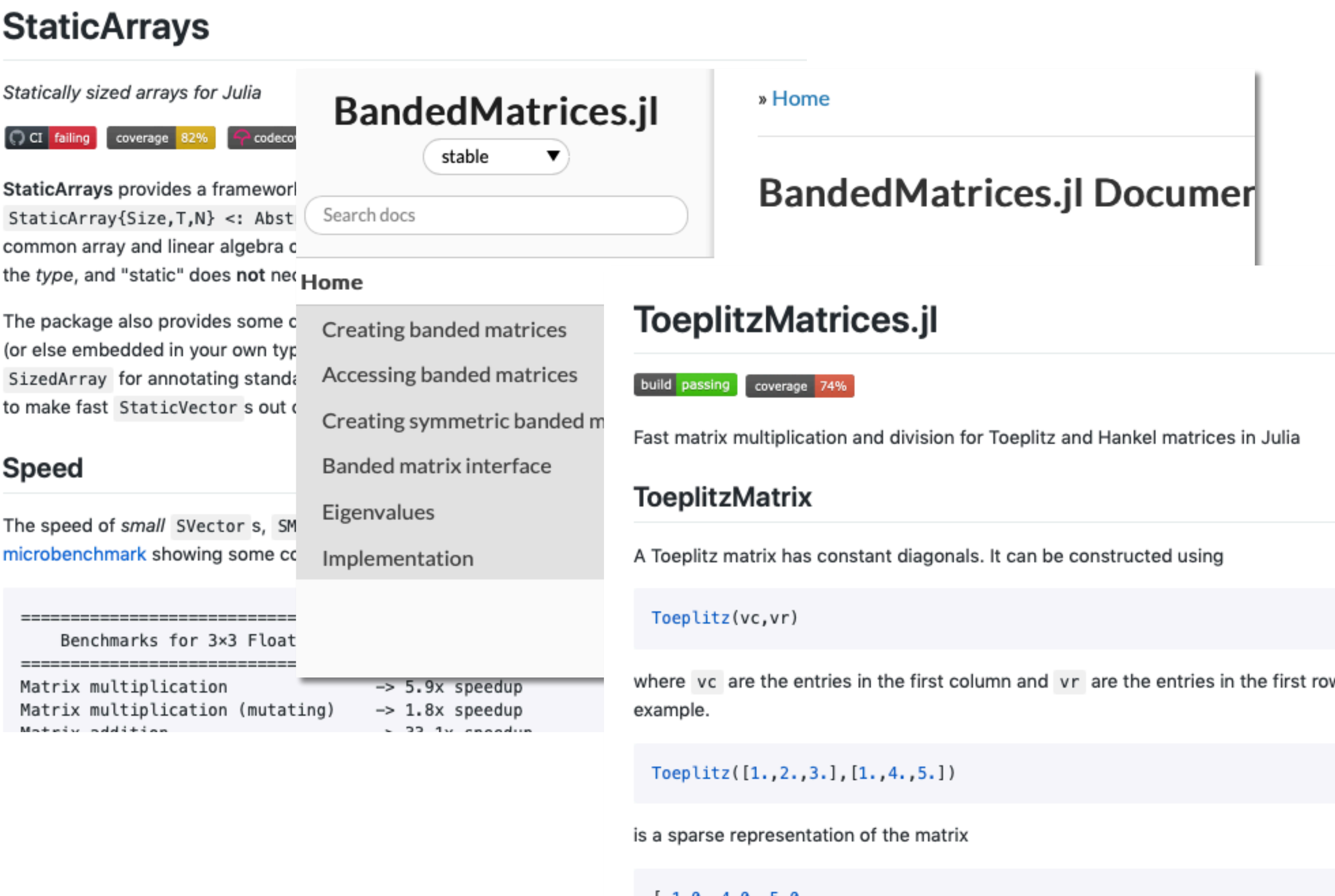
This package supports representing block-banded and banded-block-banded matrices by only storing t
non-zero bands.

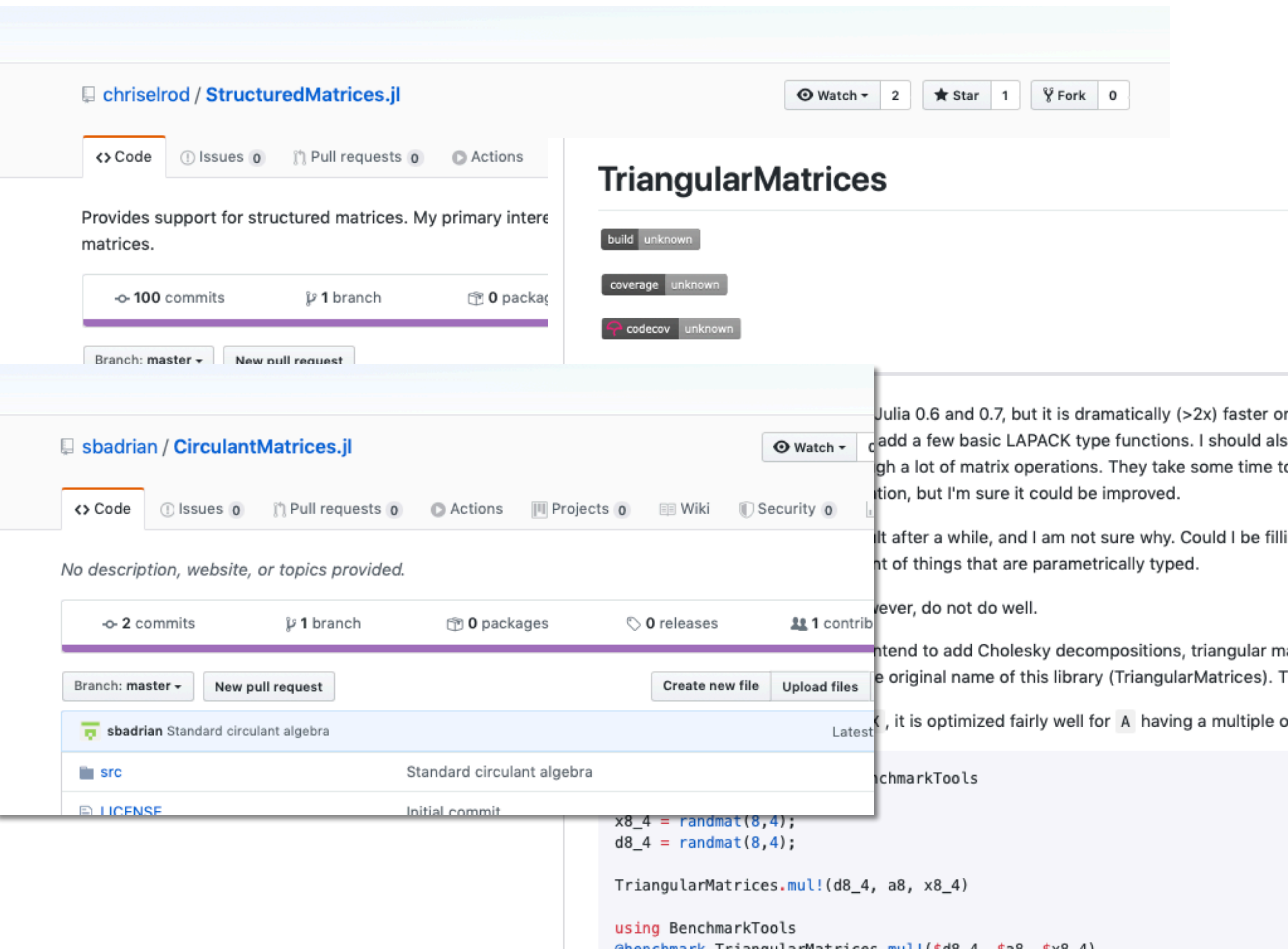
A `BlockBandedMatrix` is a subtype of `BlockMatrix` of BlockArrays.jl whose non-zero blocks are band
construct a `BlockBandedMatrix` as follows:

```
l,u = 2,1          # block bandwidths
N = M = 4          # number of row/column blocks
cols = rows = 1:N  # block sizes

BlockBandedMatrix(Zeros(sum(rows),sum(cols)), rows,cols, (l,u)) # creates a block-banded
BlockBandedMatrix(Ones(sum(rows),sum(cols)), rows,cols, (l,u)) # creates a block-banded 
BlockBandedMatrix(I, rows,cols, (l,u))                          # creates a block-banded
```

A `BandedBlockBandedMatrix` has the added structure that the blocks themselves are banded, and cont


StaticArrays
Statically sized arrays for Julia
CI failing
coverage 82%
StaticArrays provides a framewor
StaticArray{Size,T,N} <: Abst
common array and linear algebra
the type, and "static" does not ne
The package also provides some
(or else embedded in your own typ
SizedArray for annotating stand
to make fast StaticVector s out
Speed
The speed of small SVector s, SM
microbenchmark showing some c
Benchmarks for 3×3 Float
Matrix multiplication -> 5.9x speedup
Matrix multiplication (mutating) -> 1.8x speedup
BandedMatrices.jl
stable
Search docs
Home
Creating banded matrices
Accessing banded matrices
Creating symmetric banded m
Banded matrix interface
Eigenvalues
Implementation
» Home
BandedMatrices.jl Documen
ToeplitzMatrices.jl
build passing
coverage 74%
Fast matrix multiplication and division for Toeplitz and Hankel matrices in Julia
ToeplitzMatrix
A Toeplitz matrix has constant diagonals. It can be constructed using
Toeplitz(vc,vr)
where vc are the entries in the first column and vr are the entries in the first ro
example.
Toeplitz([1.,2.,3.],[1.,4.,5.])
is a sparse representation of the matrix


chriselrod / StructuredMatrices.jl
Watch 2
Star 1
Fork 0
Code
Issues 0
Pull requests 0
Actions
Provides support for structured matrices. My primary intere
matrices.
100 commits
1 branch
0 packag
Branch: master
New pull request
TriangularMatrices
build unknown
coverage unknown
codecov unknown
Julia 0.6 and 0.7, but it is dramatically (>2x) faster or
add a few basic LAPACK type functions. I should als
gh a lot of matrix operations. They take some time t
tion, but I'm sure it could be improved.
lt after a while, and I am not sure why. Could I be filli
nt of things that are parametrically typed.
ever, do not do well.
ntend to add Cholesky decompositions, triangular ma
e original name of this library (TriangularMatrices). T
, it is optimized fairly well for A having a multiple o
nchmarkTools
x8_4 = randmat(8,4);
d8_4 = randmat(8,4);
TriangularMatrices.mul!(d8_4, a8, x8_4)
using BenchmarkTools
sbadrian / CirculantMatrices.jl
Watch
Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security 0
No description, website, or topics provided.
2 commits
1 branch
0 packages
0 releases
1 contrib
Branch: master
New pull request
Create new file
Upload files
sbadrian Standard circulant algebra
Latest
src
Standard circulant algebra
LICENSE
Initial commit

HMatrices.jl

A package for assembling and factoring hierarchical matrices

docs stable | docs dev | CI passing | codecov 72% | lifecycle experimental

Installation

Install from the Pkg REPL:

```
pkg> add HMatrices
```

Overview

This package provides some functionality for assembling as well as for doing linear algebra with hierarchical matrices with a strong focus in applications arising in **boundary integral equation** methods.

For the purpose of illustration, let us consider an abstract matrix `K` with entry `i,j` given by the evaluation of some *kernel function* `G` on points `X[i]` and `Y[j]`, where `X` and `Y` are vector of points (in 3D here); that is, `K[i,j]=G(X[i],Y[j])`. This object can be constructed as follows:

```
using HMatrices, LinearAlgebra, StaticArrays
const Point3D = SVector{3,Float64}
# sample some points on a sphere
m = 100_000
X = Y = [Point3D(sin(θ)cos(ϕ),sin(θ)*sin(ϕ),cos(θ)) for (θ,ϕ) in zip(π*rand(m),2π*rand(m))]
function G(x,y)
  d = norm(x-y) + 1e-8
  1/(4π*d)
end
K = KernelMatrix(G,X,Y)
```

where we took `G` to be the free-space Greens function of Laplace's equation in 3D (to avoid division-by-zero we added `1e-8` to the distance between points).

```
# struct AutoregressiveMatrixAdjoint{T,V <: AbstractVector} <: AbstractAutoregressiveMatrixAdjoint{T,V}
#     ρ::T
#     ρᵗ::TV
#     invOmρ²ᵗ::TV
#     rinvOmρ²ᵗ::TV
#     τ::V
# end

function AutoregressiveMatrixLowerCholeskyInverse(ρ::T, τ::AbstractUnitRange) where {T}
    invOmρ²ᵗ = 1 / ( 1 - ρ*ρ )
    rinvOmρ²ᵗ = sqrt(invOmρ²ᵗ)
    AutoregressiveMatrixLowerCholeskyInverse(
        ρ, τ, EvenSpacing(nothing, invOmρ²ᵗ, rinvOmρ²ᵗ)#, - ρ * rinvOmρ²ᵗ)
    )
end
function AutoregressiveMatrixLowerCholeskyInverse(ρ::T, τ::AbstractRange) where {T}
    ρᵗ = copysign(abs(ρ)^(step(τ)), ρ)
    invOmρ²ᵗ = 1 / ( 1 - ρᵗ*ρᵗ )
    rinvOmρ²ᵗ = sqrt(invOmρ²ᵗ)
    AutoregressiveMatrixLowerCholeskyInverse(
        ρ, τ, EvenSpacing(ρᵗ, invOmρ²ᵗ, rinvOmρ²ᵗ)#, - ρᵗ * rinvOmρ²ᵗ)
    )
end
```

```julia
function lmul!(S::Adjoint{T,SplitCholesky{T,Symmetric{T,M}}}, B::AbstractVecOrMat{T}) where {T,M<:BandedMatrix{T}}
    require_one_based_indexing(B)
    n, nrhs = size(B, 1), size(B, 2)
    if size(S, 1) != n
        throw(DimensionMismatch("Matrix has dimensions $(size(S)) but right hand side has first dimension $n"))
    end
    A = S.parent.factors
    b = bandwidth(A, 1)
    m = (n+b)÷2
    @inbounds for l = 1:nrhs
        for j = m:-1:1
            t = zero(T)
            @simd for k = max(1,j-b):j
                t += A[k,j]*B[k,l]
            end
            B[j,l] = t
        end
        for j = m-b+1:m
            t = zero(T)
            @simd for k = m+1:j+b
                t += A[k,j]*B[k,l]
            end
            B[j,l] += t
        end
        for j = m+1:n
            t = zero(T)
            @simd for k = j:min(j+b,n)
                t += A[k,j]*B[k,l]
            end
            B[j,l] = t
```

Introduction to Applied Linear Algebra

Vectors, Matrices, and Least Squares

Julia Language Companion

Stephen Boyd and Lieven Vandenberghe

DRAFT August 26, 2018

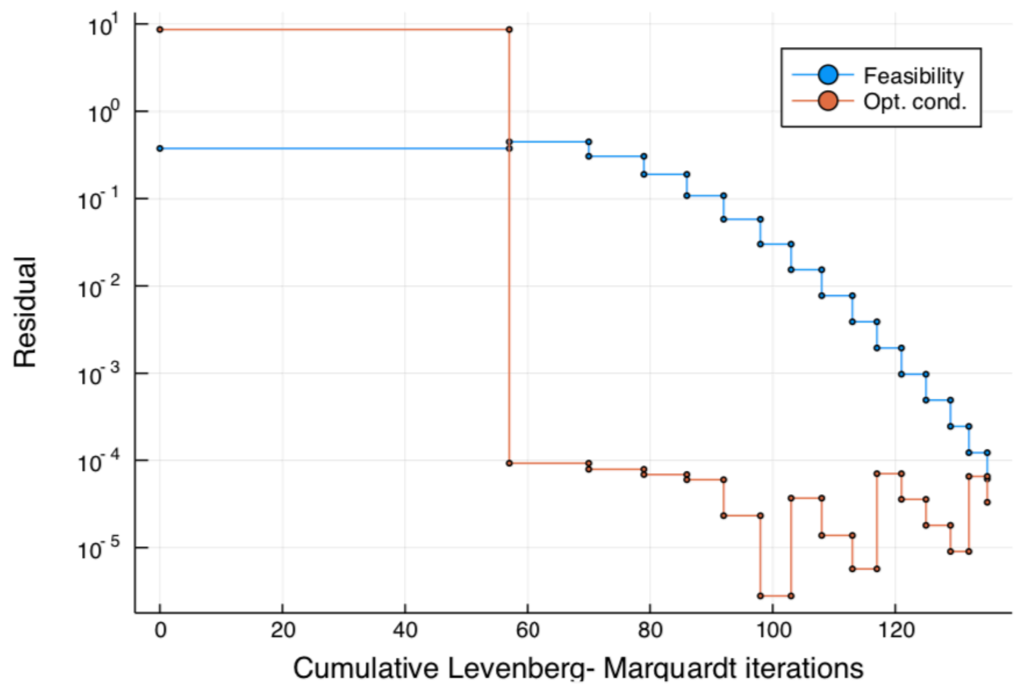


Figure 19.1 Feasibility and optimality condition errors versus the cumulative number of Levenberg–Marquardt iterations in the penalty algorithm.

```
           [r;r] for r in hist["oc_res"][1:end-1]]...,
           hist["oc_res"][end]);
julia> plot(itr, feas_res, shape=:circle, label = "Feasibility")
julia> plot!(itr, oc_res, shape=:circle, label = "Opt. cond.")
julia> plot!(yscale = :log10,
        xlabel = "Cumulative Levenberg--Marquardt iterations",
        ylabel = "Residual")
```

19.3 Augmented Lagrangian algorithm

```
function aug_lag_method(f, Df, g, Dg, x1, lambda1; kmax = 100,
```

Huchette article receives Beale-Orchard-Hays Prize from MOS

JuMP: A Modeling Language for Mathematical Optimization' was published in the SIAM Review in 2017.

An article co-authored by Joey Huchette, an adjunct faculty member in computational and applied mathematics at Rice University, has received the Beale-Orchard-Hays Prize from the Mathematical Optimization Society (MOS).

"JuMP: A Modeling Language for Mathematical Optimization" was published in the SIAM (Society for Industrial and Applied Mathematics) Review in 2017. Huchette's co-authors are Iain Dunning, team lead and researcher with Hudson River Trading, and Miles Lubin, research scientist in the algorithms and optimization team at Google.

ocw.mit.edu › 18-06-linear-algebra-spring-2010

Linear Algebra | Mathematics | MIT OpenCourseWare

Used with permission.) Instructor(s). Prof. Gilbert Strang. **MIT** Course Number. **18.06**. As Taught In.

Video Lectures · Introduction to Linear Algebra · Syllabus · Readings

mitmath / 1806

Watch 95 | Star

Code | Issues 0 | Pull requests 1 | Actions | Projects 0 | Wiki | Security 0 | Insights

Branch: master | 1806 / lectures / | Create new file | Upload fil

alanedelman syllabus | Latest comm

..	
1806overview.pdf	lecture 1 updates
1806overview.pptx	lecture 1 updates
Chutes-and-Ladders.ipynb	lecture updates
Circulant-Matrices.ipynb	typos
Complexity.ipynb	lecture updates
Conditioning.ipynb	typos
Dense-and-Sparse.ipynb	lecture updates
Determinants.ipynb	lecture updates
Diagonalization.ipynb	updates
Eigenvalue-Intro.ipynb	updates from lecture
Eigenvalue-Polynomials.ipynb	lecture updates
Elimination-matrices.ipynb	class
Fibonacci.ipynb	updates

stanford.edu › class › julia

EE103: Software - Stanford University

Julia. In this **course** we will be using the relatively new language **Julia**. Keep in mind that you are not expected to have a strong background in programming ...

ee104.stanford.edu › julia

Julia - EE104 - Stanford University

EE104/CME107: Introduction to Machine Learning. **Stanford** University, Spring Quarter 2020. In this **course**, you will use the **Julia** language to create short ...

web.stanford.edu › class › cgi-bin › julia

Julia | AA228/CS238 - Stanford University

Although this **course** is language agnostic, we will use **Julia** to demonstrate various algorithms. It is a high-level language for scientific computing that provides ...

ee103.stanford.edu

EE103/CME103: Introduction to Matrix Methods - Stanford ...

In this **course**, students use a relatively new language called **Julia** to do computations with vectors and matrices. The **course** is suitable for any undergraduate with ...

explorecourses.stanford.edu › search › q=CME 257: Ad...

CME 257: Advanced Topics in Scientific Computing with Julia

CME 257: Advanced Topics in Scientific Computing with **Julia**. This **course** will rapidly introduce students to the **Julia** programming language, with the goal of ...

stanford.edu › class › courseinfo

EE103: Course Information - Stanford University

Sections will be 2 hours long, with the first hour spent on problem solving and **Julia** programming, and the remaining time used as office hours. Mark Nishimura: ...

explorecourses.stanford.edu › search › q=CME 257: Ad...

CME 257: Advanced Topics in Scientific Computing with Julia

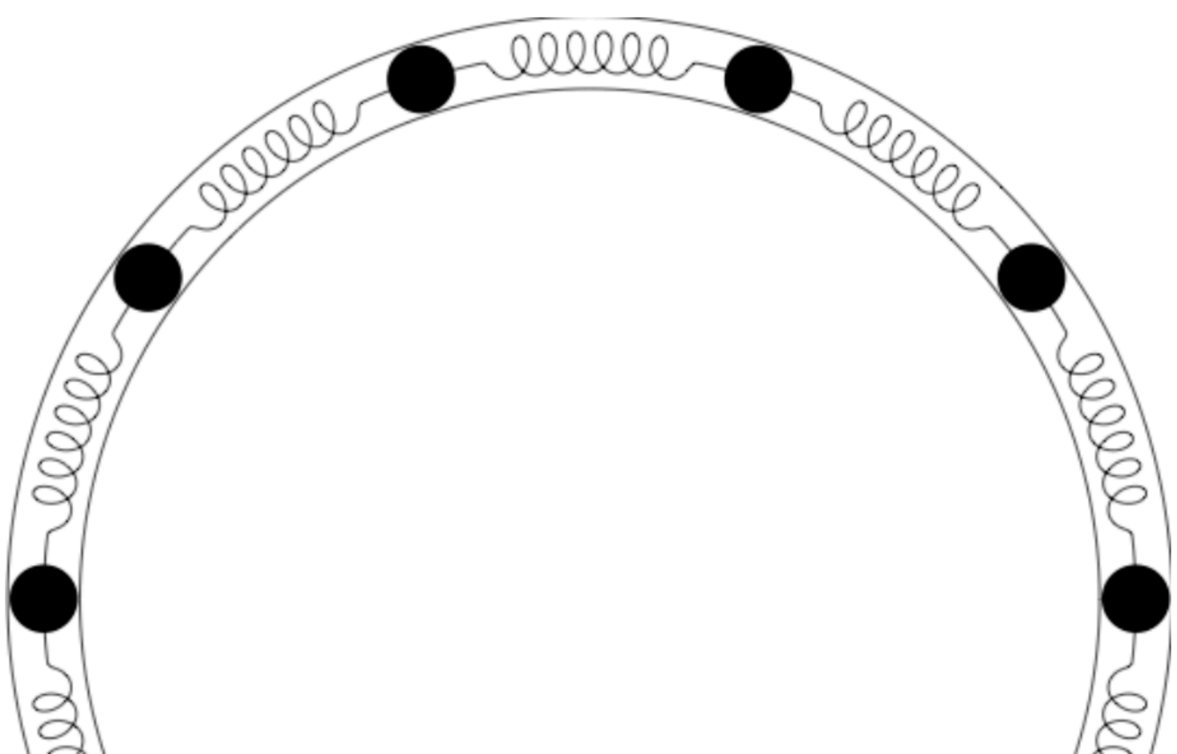
Several MIT courses involving numerical computation, including 18.06, 18.303, 18.330, 18.335/6.337, 18.337/6.338, and 18.338, are beginning to use Julia, a fairly new language for technical computing. This page is intended to supplement the

In [1]:
```
using PyPlot, Interact
```

```
INFO: Interact.jl: using new nbwidgetsextension protocol
```

Circulant Matrices

In this lecture, I want to introduce you to a new type of matrix: **circulant** matrices. Like Hermitian matrices, they have orthonormal eigenvectors, but unlike Hermitian matrices we know *exactly* what their eigenvectors are! Moreover, their eigenvectors are closely related to the famous Fourier transform and Fourier series. Even more importantly, it turns out that circulant matrices and the eigenvectors lend themselves to **incredibly efficient** algorithms called FFTs, that play a central role in much of computational science and engineering.



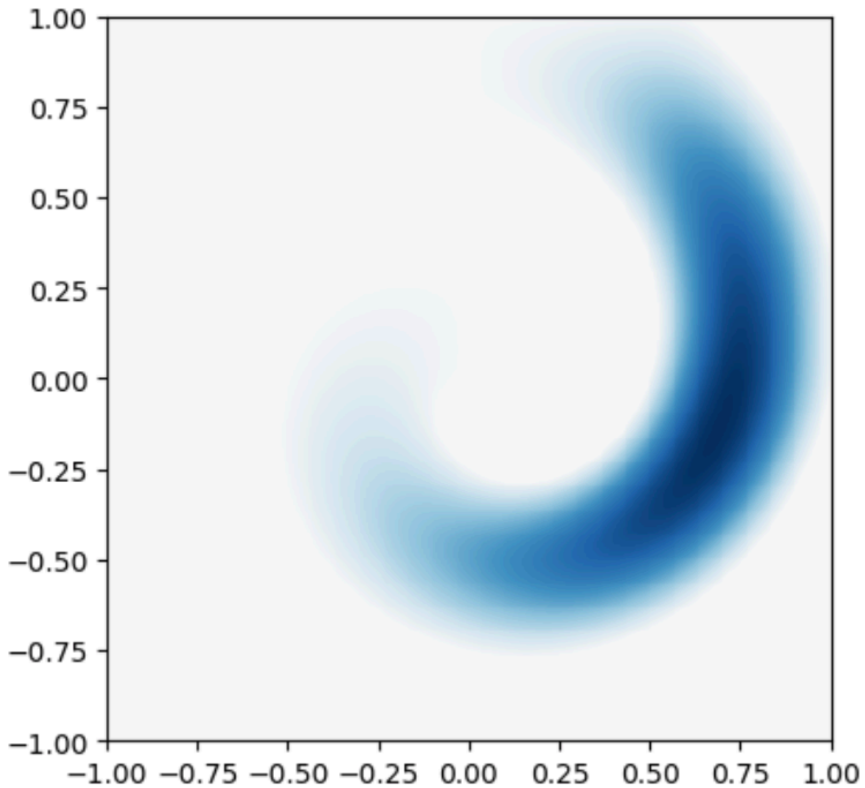
Now we'll compute a few of the smallest-|λ| eigenvectors using `eigs`. We'll use the [Interact package] to interactively decide which eigenvalue to plot.

In [13]:
```
u = zeros(N,N)
@time λ, X = eigs(A, nev=20, which=:SM);

f = figure()
@manipulate for which_eig in slider(1:20, value=1)
    withfig(f) do
        u[i] = X[:,which_eig]
        umax = maximum(abs, u)
        imshow(u, extent=[-1,1,-1,1], vmin=-umax,vmax=+umax, cmap="RdBu")
    end
end
```

```
  3.791509 seconds (1.65 M allocations: 313.391 MiB, 3.50% gc time)
```

Out[13]:



Notice that it took **less than two seconds** to solve for 20 eigenvectors and eigenvalues!

This is because `eigs` is essentially using an algorithm like the power method, that only uses repeated multiplication by A.

Or, sometimes, particularly to find the smallest $|\lambda|$ eigenvectors, it might repeatedly divide by A, i.e. solve $Ax = b$ for $b = A^{-1}x$. But it can't actually compute the inverse matrix, and I said that LU factorization was $\sim n^3$ in general. So, what is happening?

Sparse-direct solvers for Ax=b

Even if A is a sparse matrix, A^{-1} is generally not sparse. However, if you arrange things cleverly, often the L and U factors *are* still sparse!

This leads to something called **sparse-direct solvers**: they solve $Ax = b$ by ordinary Gaussian elimination to find $A = LU$, but they take advantage of sparse A to avoid computing with zeros. Moreover, they first re-order the rows and columns

In the Classroom

Julia is ready for the classroom. We encourage instructors to participate in the Julia community for questions about Julia or specific packages. This page puts together various resources that instructors and students alike may find useful. See where Julia is being taught today.

Julia in the classroom

Julia is now being used in several universities and onlin

- AGH University of Science and Technology, Polan
  - Signal processing in medical diagnostic sys
- Arizona State University
  - MAT 423, Numerical Analysis (Prof. Clemen
- Azad University, Science and Research Branch
  - CE 3820, Modeling and Evaluation (Dr. Arm
- Brown University
  - CSCI 1810, Computational Molecular Biolog
- Budapest University of Technology and Economic
  - Applications of Differential Equations and V
- City University of New York
  - MTH 229, Calculus Computer Laboratory (P
- Cornell University
  - CS 5220, Applications of Parallel Computer
- École Polytechnique Fédérale de Lausanne
  - [CIVIL 557] Decision-aid methodologies in t
- Einaudi Institute for Economics and Finance, Rom
  - Econometrics of DSGE Models (Giuseppe R
- Emory University
  - MATH 346, Introduction to Optimization Th
  - MATH 516, Numerical Analysis II (Prof. Lars
- Federal Rural University of Rio de Janeiro (UFRRJ
  - TM429, Introduction to Recommender Syst
- Federal University of Alagoas (*Universidade Fede*
  - COMP272, Distributed Systems (Prof. Andre
- Federal University of Paraná (*Universidade Feder*
  - CM103, Laboratório de Matemática Aplicad
  - CMM014, Cálculo Numérico (Prof. Abel Soa
  - CM106/CMI043/CMM204/MNUM7079, Otin
- Federal University of Uberlândia, Institute of Phys
  - GFM050, Física Computacional (Prof. Gersc
- Hadsel High School, Stokmarknes, Nordland, Nor
  - AnsattOversikt, [REA3034] Programmering Olav A Marschall, M.sc. Computer Science)
- IIT Indore
  - ApplNLA, Modern Applications of Numerica
- Iowa State University
  - STAT 590F, Topics in Statistical Computing:
- Luiss University Rome, Department of Economics
  - Econometric Theory (Giuseppe Ragusa)
- Lund University, Sweden, Department of Automat
  - Julia for Scientific Computing
  - Optimization for Learning
- Massachusetts Institute of Technology (MIT)
  - 6.251 / 15.081, Introduction to Mathematical
  - 18.06, Linear Algebra: Fall 2015, Dr. Alex To
  - 18.303, Linear Partial Differential Equations:
  - 18.337 / 6.338, Numerical Computing with
  - 18.085 / 0851, Computational Science And
  - 18.330, Introduction to Numerical Analysis (
- Northeastern University, Fall 2016
  - MTH3300: Applied Probability & Statistics
- Óbuda University, John von Neumann Faculty of Inf
  - [Intelligent Development Tools (Hungarian)]
  - [Intelligent Development Tools (English)]
  - [Fundamental Mathematical Methods (English
- Pennsylvania State University
  - ASTRO 585, Seminar: High-Performance Scie github repo
  - ASTRO 528, High-Performance Scientific Cor
- Politecnico di Torino (Torino, Italy)
  - Algorithms for Optimization, Inference and Le
  - Inference in Biological Systems, (Prof. A. Gam
  - Stochastic Simulation Methods In Physics, (Pr
- Polytech Nice Sophia
  - Mathematics for the engineer (Prof. J.-B. Caill
- Pontifical Catholic University of Rio de Janeiro (PUC
  - Programming in Julia (Prof. Thuener Silva), Su
  - Linear Optimization (Prof. Alexandre Street), S
  - Decision and Risk Analysis (Prof. Davi Valladã
- Purdue University
  - CS51400, Numerical Analysis (Prof. David Gle
- Royal Military Academy (Brussels)
  - ES123, Computer Algorithms and Programmin
  - ES313, Mathematical modelling and Compute
- "Sapienza" University of Rome, Italy
  - Operations Research (Giampaolo Liuzzi), Spri
  - Optimization for Complex Systems (Giampaol
- Sciences Po Paris, Department of Economics, Sprin
  - Computational Economics for PhDs (Florian O
- SGH Warsaw School of Economics, Poland
  - 223490-0286, Statistical Learning Methods (
  - 234900-0286, Agent-Based Modeling (Bogu
  - 239420-0553, *Introduction to Deep Learning*
- Southcentral Kentucky Community and Technical C
  - CIT 120 Computational Thinking (Inst. Bryan K
- Stanford University
  - AA222, Introduction to Multidisciplinary Desig
  - AA228/CS238, Decision Making under Uncert
  - EE103, Introduction to Matrix Methods (Prof. S
  - CME 257, Advanced Topics in Scientific Comp
  - EE266, Stochastic Control (Prof. Sanjay Lall),
- Tec de Monterrey, Santa Fe Campus, Mexico City
  - IN2022, Modelos de Optimización (Prof. Marz
- Tokyo Metropolitan University, Tokyo, Japan
  - L0407, Exercises in Programming I for Mechar (scheduled), in Japanese
- TU Dortmund / SFB 823, Germany
  - One week introductory course into Julia with a
- Universidad Adolfo Ibáñez, Chile
  - ING747, Integer programming, Fall 2018-2019
  - DIIO06, Advanced linear optimization, Spring
- Universidad del Norte, Barranquilla, Colombia
  - ELP 4076, Ingeniería de Ríos y Costas (Prof. G
  - ICI 4083, Hidráulica (Prof. Germán Rivillas)
- Universidad Nacional Autónoma de México
  - Física computacional (Prof. David P. Sanders), Fall 2014
  - Métodos numéricos para sistemas dinámicos (Prof. Luis Benet), Fall 2014
  - Métodos numéricos avanzados (Prof. David P. Sanders and Prof. Luis Benet), Spring 2015
  - Métodos computacionales para la física estadística (Prof. David P. Sanders), Spring 2015
- Universidad Nacional Pedro Ruiz Gallo, Lambayeque, Perú
  - Julia: el lenguaje del futuro, Semana de Integración de Ingeniería Electrónica, (Oscar William N
- Universidad Veracruzana, México
  - Algoritmos Evolutivos y de Inteligencia Colectiva (Jesús A. Mejía-de-Dios), Fall 2019
- University at Buffalo
  - IE 572 Linear Programming (Prof. Changhyun Kwon), Fall 2014
- University of Antwerp, Faculty of Pharmaceutical, Biomedical, Veterinary Sciences, October 2016
  - Computational Neuroscience (2070FBDBMW), Master of Biomedical Sciences, of Biochemistr
- University of Basel, Department of Physics
  - Classical Mechanics (Prof. Christoph Bruder), Fall 2020
- University of California, Los Angeles (UCLA)
  - Biostat 257, Computational Methods for Biostatistical Research, Spring 2021 (Prof. Hua Zhou)
- University of Cologne, Institute for Theoretical Physics
  - Computational Physics (Prof. Simon Trebst), Summer 2016
  - Computational Physics (Prof. Ralf Bulla), Summer 2017
  - Statistical Physics (Prof. Simon Trebst), Winter 2017
  - Computational Many-Body Physics (Prof. Simon Trebst), Summer 2018
  - Advanced Julia Workshop (MSc. Carsten Bauer), Fall 2018
  - Computational Physics (Prof. Simon Trebst), Summer 2019
  - Advanced Julia Workshop (MSc. Carsten Bauer), Fall 2019
- University of Connecticut, Storrs
  - CHEG 5395, Metaheuristic and Heuristic Methods in Chemical Engineering (Prof. Ranjan Sriva
- University of Edinburgh
  - Spring 2017, MATH11146, Modern optimization methods for big data problems (Prof. Peter Ric
  - Spring 2016, MATH11146, Modern optimization methods for big data problems (Prof. Peter Ric
- University of Glasgow, School of Mathematics and Statistics
  - An Introduction to Julia, course of Online Master of Science (MSc) in Data Analytics (Theodore
- University of Oulu
  - Invited Advanced Julia Workshop (MSc. Carsten Bauer, University of Cologne), Spring 2020
- University of South Florida
  - ESI 4312, Deterministic Operations Research (Prof. Changhyun Kwon), Fall 2017–Fall 2020
  - ESI 6410, Optimization in O.R. (Prof. Changhyun Kwon), Spring 2021
  - ESI 6491, Linear Programming and Network Optimization (Prof. Changhyun Kwon), Fall 2015–F
  - EIN 6945, Nonlinear Optimization and Game Theory (Prof. Changhyun Kwon), Spring 2016, 20
- University of Sydney
  - MATH3076/3976, Mathematical Computing (Assoc. Prof. Sheehan Olver), Fall 2016
- Université Paul Sabatier, Toulouse
  - Optimization in Machine Learning, (Prof. Peter Richtarik), Fall 2015
- Université de Liège
  - MATH0462, Discrete Optimization (Prof. Quentin Louveaux), Fall 2016
  - MATH0461, Introduction to Numerical Optimization (Prof. Quentin Louveaux), Fall 2016
  - MATH0462, Discrete Optimization (Prof. Quentin Louveaux), Fall 2015
- Université de Montréal
  - IFT1575, Modèles de recherche opérationnelle (Prof. Bernard Gendron), Fall 2017
  - IFT3245, Simulation et modèles (Prof. Fabian Bastin), Fall 2017
  - IFT3515, Optimisation non linéaire (Prof. Fabian Bastin), Winter 2017-2018
  - IFT6512, Programmation stochastique (Prof. Fabian Bastin), Winter 2018
- University of Washington
  - AMATH 586, Numerical analysis of time-dependent problems (Prof. Tom Trogdon), Spring 202
- Western University Canada

The Big List

This is a big list of Julia Automatic Differentiation (AD) packages and related tooling. As you can see there is a lot going on here. As with any such big lists it rapidly becomes out-dated. When you notice something that is out of date, or just plain wrong, please submit a PR.

This list aims to be comprehensive in coverage. By necessity, this means it is not comprehensive in detail. It is worth investigating each package yourself to really understand its ins and outs, and pros and cons of its competitors.

Reverse-mode

- ReverseDiff.jl: Operator overloading reverse-mode AD. Very well-established.
- Nabla.jl: Operator overloading reverse-mode AD. Used in (its maintainer) Invenia's systems.
- Tracker.jl: Operator overloading reverse-mode AD. Most well-known for having been the AD used in earlier versions of the machine learning package Flux.jl. No longer used by Flux.jl, but still used in several places in the Julia ecosystem.
- AutoGrad.jl: Operator overloading reverse-mode AD. Originally a port of the Python Autograd package. Primarily used in Knet.jl.
- Zygote.jl: IR-level source to source reverse-mode AD. Very widely used. Particularly notable for being the AD used by Flux.jl. Also features a secret experimental source to source forward-mode AD.
- Yota.jl: IR-level source to source reverse-mode AD.
- XGrad.jl: AST-level source to source reverse-mode AD. Not currently in active development.
- ReversePropagation.jl: Scalar, tracing-based source to source reverse-mode AD.
- Enzyme.jl: Scalar, LLVM source to source reverse-mode AD. Experimental.
- Diffractor.jl: Next-gen IR-level source to source reverse-mode (and forward-mode) AD. In development.

Forward-mode

- ForwardDiff.jl: Scalar, operator overloading forward-mode AD. Very stable. Very well-established.
- ForwardDiff2: Experimental, non-scalar hybrid operator-overloading/source-to-source forward-mode AD. Not currently in development.
- Diffractor.jl: Next-gen IR-level source to source forward-mode (and reverse-mode) AD. In development.

Symbolic:

- Symbolics.jl: A pure Julia computer algebra system. While its docs focus on some particular domain use-case it is a fully general purpose system.

Exotic

- TaylorSeries.jl: Computes polynomial expansions; which is the generalization of forward-mode AD to nth-order derivatives.
- NiLang.jl: Reversible computing DSL, where everything is differentiable by reversing.
- TaylorDiff.jl: an efficient, linear-scaling implementation for higher-order directional derivatives, implemented with operator-overloading on statically-typed Taylor polynomials. In development.

Finite Differencing

Yes, we said at the start to stop approximating derivatives, but these packages are faster and more accurate than you would expect finite differencing to ever achieve. If you really need finite differencing, use these packages rather than implementing your own.

- FiniteDifferences.jl: High-accuracy finite differencing with support for almost any type (not just arrays and numbers).
- FiniteDiff.jl: High-accuracy finite differencing with support for efficient calculation of sparse Jacobians via coloring vectors.
- Calculus.jl: Largely deprecated, legacy package. New users should look to FiniteDifferences.jl and FiniteDiff.jl instead.

Rulesets

Packages providing collections of derivatives of functions which can be used in AD packages.

- ChainRules: Extensible, AD-independent rules.
    - ChainRulesCore.jl: Core API for user to extend to add rules to their package.
    - ChainRules.jl: Rules for Julia Base and standard libraries.
    - ChainRulesTestUtils.jl: Tools for testing rules defined with ChainRulesCore.jl.
- DiffRules.jl: An earlier set of AD-independent rules, for scalar functions. Used as the primary source for ForwardDiff.jl, and in part by other packages.
- ZygoteRules.jl: Lightweight package for defining rules for Zygote.jl. Largely deprecated in favour of the AD-independent ChainRulesCore.jl.

Sparsity

- SparsityDetection.jl: Automatic Jacobian and Hessian sparsity pattern

speed

Speed Gotchas

- JIT and "time to first plot"
- Only functions are compiled REPL code is NOT compiled
- type stability
- vs hand-coded fused cuda kernels (pytorch)

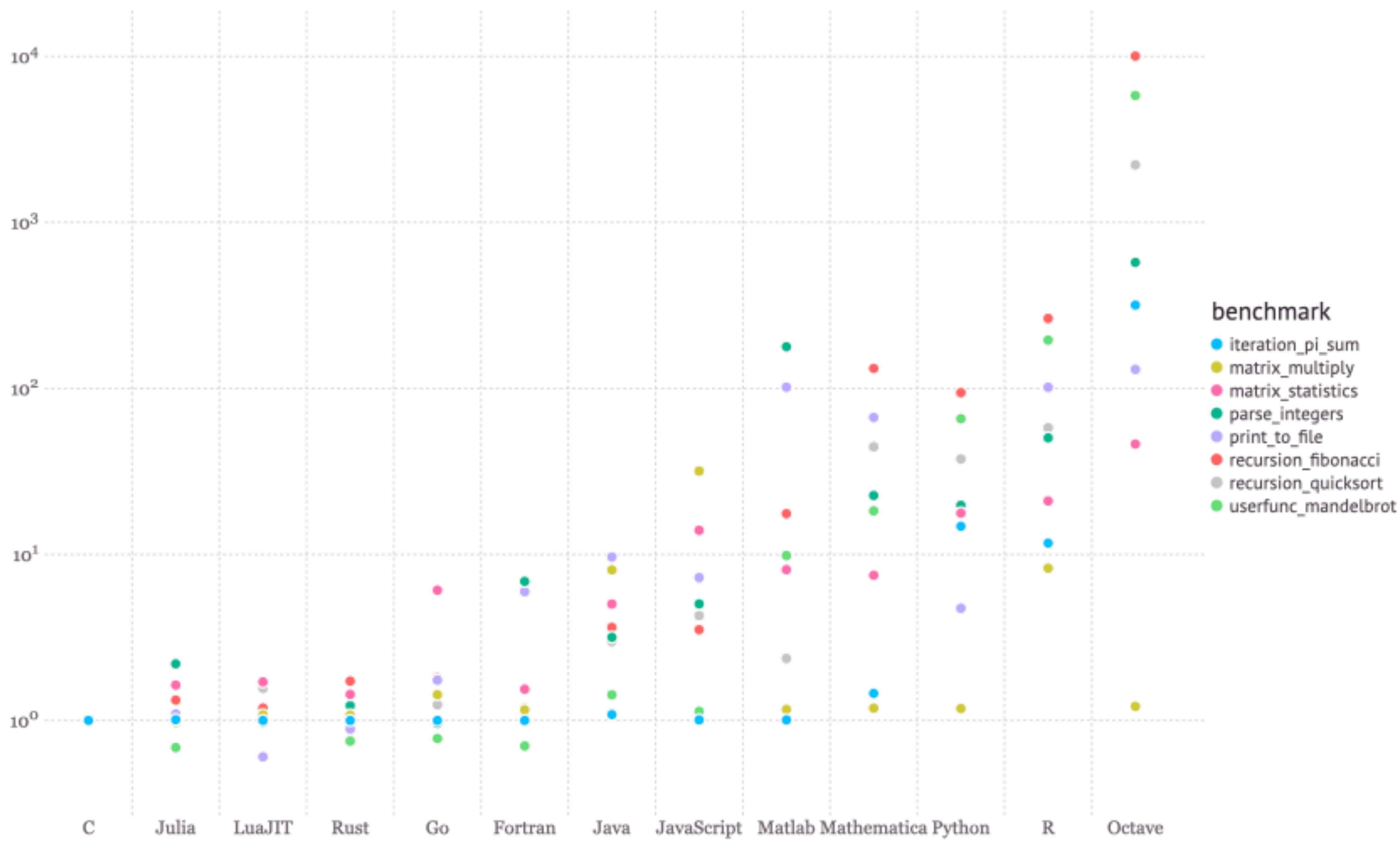
TYPE STABILITY

- @code_warntype

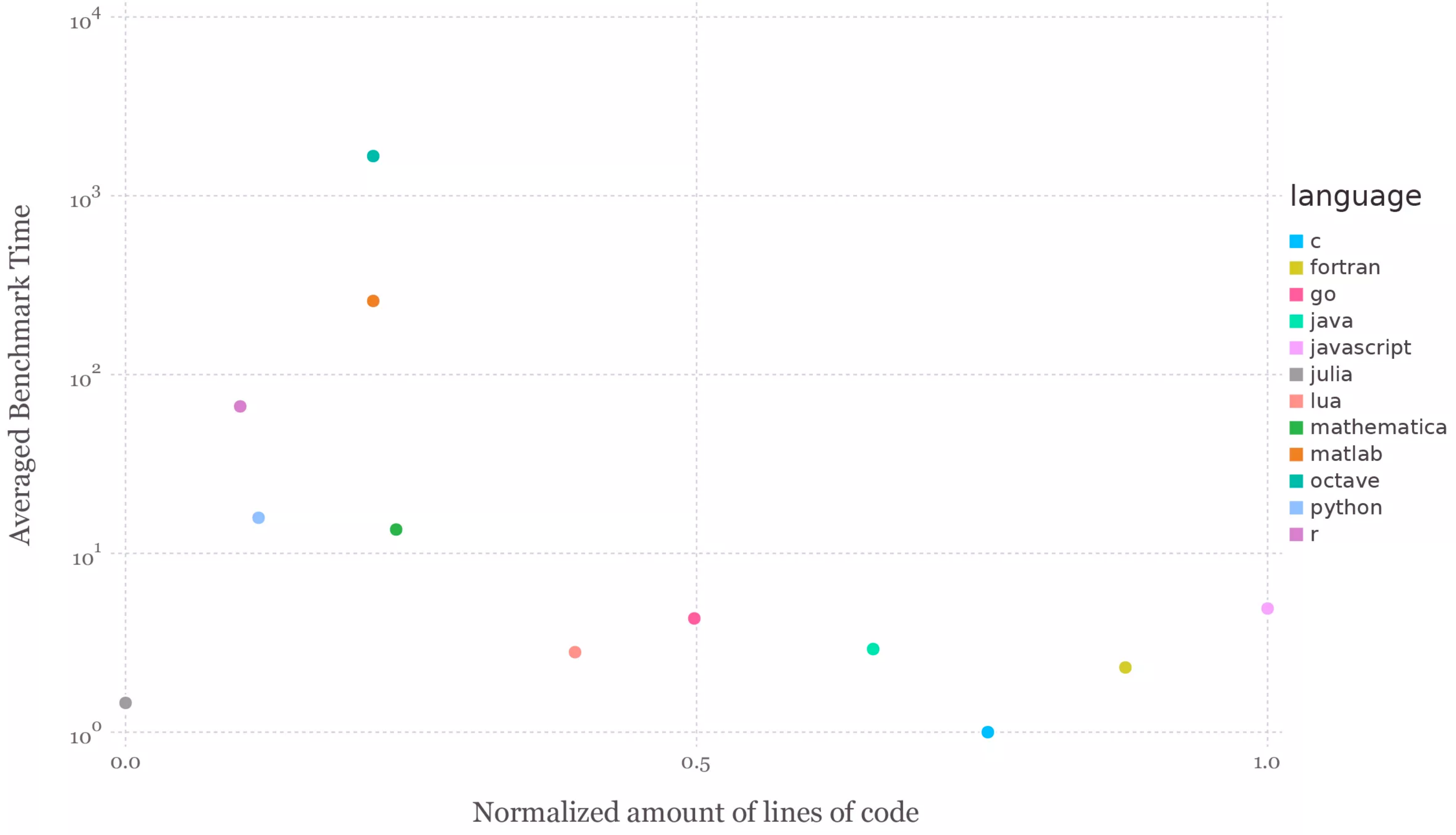
```
function bad(i)
  i = i + 1
  i = “bad”
end
```

-

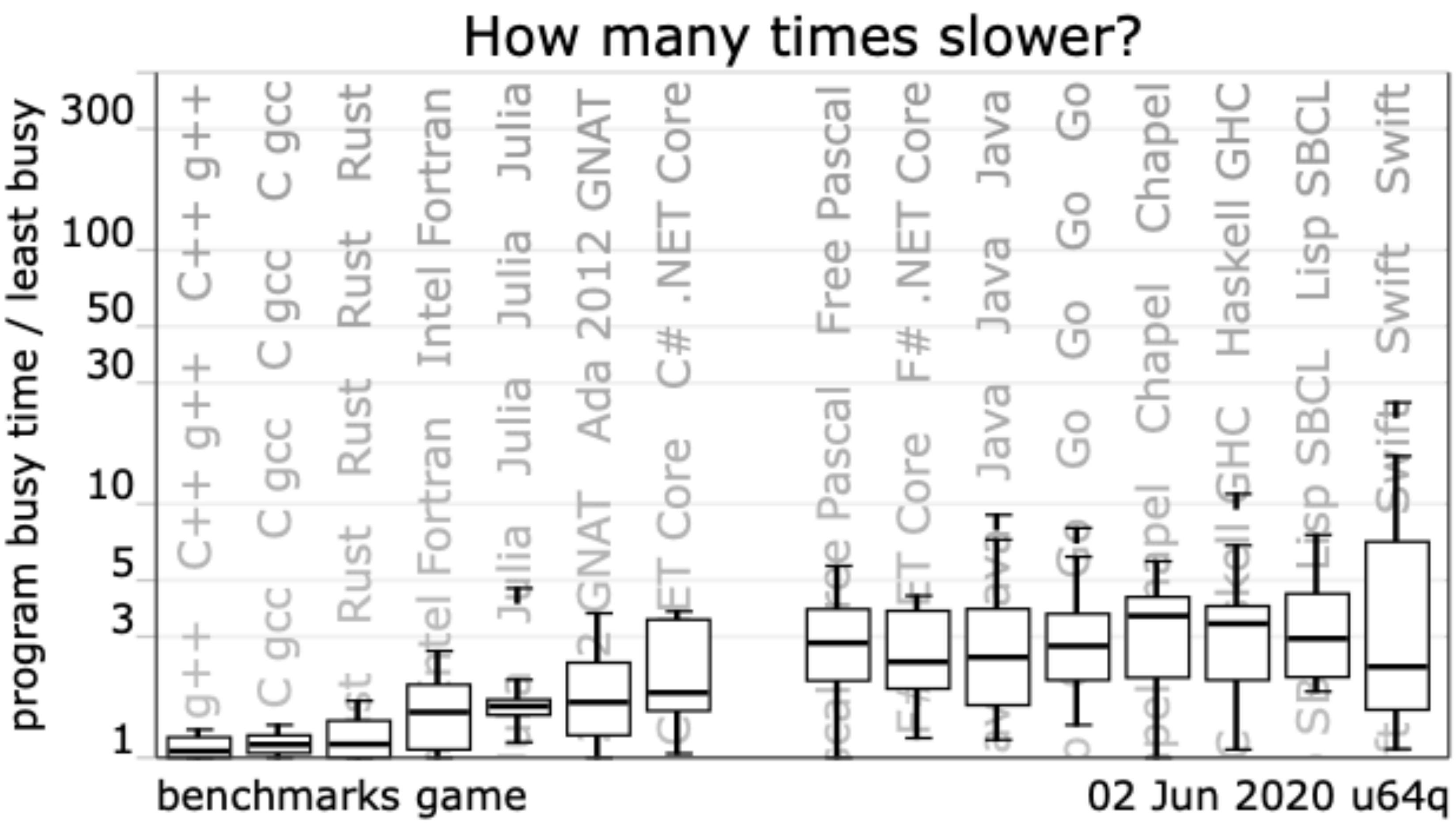
Julia Joins the Petaflop Club: Celeste joins the rarified list of applications **to exceed 1 petaflop per second performance,** and is the first to do so in a dynamic high-level language. The Celeste research team processed 55 terabytes of visual data and classified 188 million astronomical objects in just 15 minutes, resulting in the first comprehensive catalog of all visible objects from the Sloan Digital Sky Survey. This is one of the largest problems in mathematical optimization ever solved. The Celeste team, which includes researchers from UC Berkeley, Lawrence Berkeley National Laboratory, National Energy Research Supercomputing Center, Intel, Julia Computing and the Julia Lab at MIT, used 9,300 Knights Landing (KNL) nodes on the NERSC Cori Phase II supercomputer to execute 1.3 million threads on 650,000 KNL cores.



10^4
10^3
10^2
10^1
10^0
C
Julia
LuaJIT
Rust
Go
Fortran
Java
JavaScript
Matlab
Mathematica
Python
R
Octave
benchmark
iteration_pi_sum
matrix_multiply
matrix_statistics
parse_integers
print_to_file
recursion_fibonacci
recursion_quicksort
userfunc_mandelbrot



10^4
10^3
10^2
10^1
10^0
Averaged Benchmark Time
0.0
0.5
1.0
Normalized amount of lines of code
language
c
fortran
go
java
javascript
julia
lua
mathematica
matlab
octave
python
r



Including jit time

https://benchmarksgame-team.pages.debian.net/

< MatecDev >   Tutorials   Blog   About

Python, Julia, Fortran

Testing Julia: Fast as Fortran, Versatile as Python

by Martin D. Maas, Ph.D
@MartinDMaas
Last updated: 2021-09-21

I'm super enthusiastic about Julia after running this comparison of Julia vs Numpy vs Fortran, for performance and code simplicity.

ACCELERATED COMPUTING HPC

High-Performance GPU Computing in the Julia Programming Language

By Tim Besard | October 25, 2017 Tags: Compilation, Julia, Programming Languages and Compilers

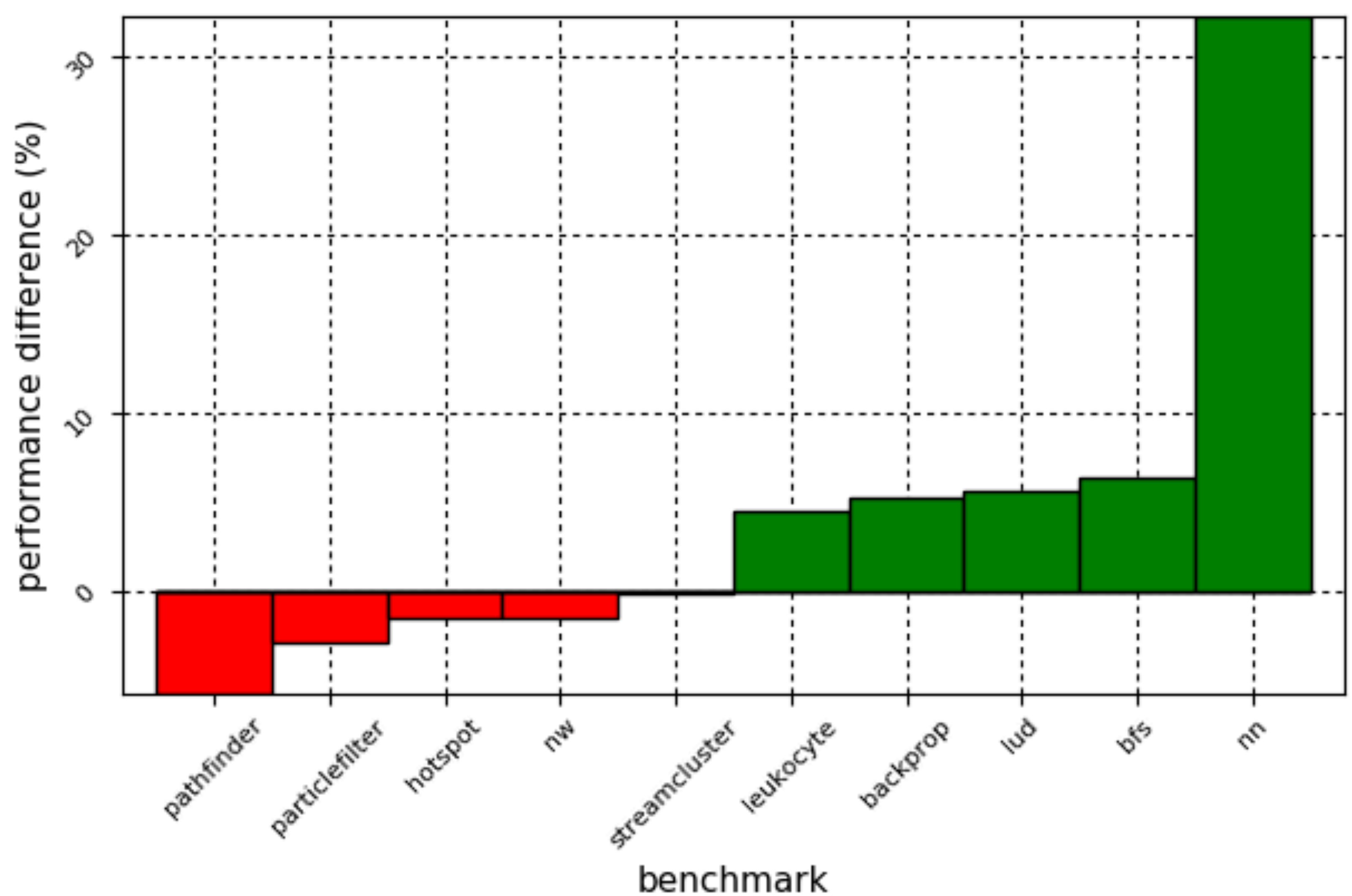
Julia is a high-level programming language for mathematical computing that is as easy to use as Python, but as fast as C. The language has been created with performance in mind, and combines careful language design with a sophisticated LLVM-based compiler [Bezanson et al. 2017].

Julia is already well regarded for programming multicore CPUs and large parallel computing systems, but recent developments make the language suited for GPU computing as well. The performance possibilities of GPUs can be democratized by providing more high-level tools that are easy to u … ing programmers. In this blog post, I will focus on native G … with native PTX code generation capabilities: CUDAnat

```
using CUDAdrv, CUDAnative

function kernel_vadd(a, b, c)
    i = threadIdx().x
    c[i] = a[i] + b[i]
    return
end
```

The chart in Figure 3 compares the performance of the original CUDA C++ implementations of these benchmarks against our Julia ports. The Julia versions are almost verbatim ports, that is, with no algorithmic changes and not introducing high-level concepts, in order to assess compiler performance differences as exactly as possible. As you can see, using Julia for GPU computing doesn't suffer from any broad performance penalty. The only outlier is the nn benchmark, which performs significantly better with CUDAnative.jl due to slightly better register usage. On average, the CUDAnative.jl ports perform identical to statically compiled CUDA C++ (the difference is ~2% in favor of CUDAnative.jl, excluding nn). This is in part because of the work by Google on the NVPTX LLVM back-end.



It is comparing Finite Element solver, which is an often used algorithm in material research and therefore represents a relevant use case for Julia.

N	JULIA	FENICS(PYTHON + C++)	FREEFEM++(C++)
121	0.99	0.67	0.01
2601	1.07	0.76	0.05
10201	1.37	1.00	0.23
40401	2.63	2.09	1.05
123201	6.29	5.88	4.03
251001	12.28	12.16	9.09

(taken from codeproject.)
These are remarkable results, considering that the author states it was not a big effort to achieve this. After all, the other libraries are established FEM solvers written in C++, which should not be easy to compete with.

Torchdiffeq vs DifferentialEquations.jl (/ DiffEqFlux.jl) Benchmarks

Benchmark: Solve the Lorenz equations from 0 to 100 with abstol=reltol=1e-8

Absolute Timings

- DifferentialEquations.jl: 1.675 ms
- diffeqpy (DifferentialEquations.jl called from Python): 3.473 ms
- SciPy+Numba: 50.99 ms
- SciPy: 110.6 ms
- torchdiffeq: 48 seconds
- torchscript torchdiffeq: 48 seconds

Timings Relative to DifferentialEquations.jl

- DifferentialEquations.jl: 1x
- diffeqpy (DifferentialEquations.jl called from Python): 2.07x Slower
- SciPy+Numba: 30x Slower
- SciPy: 66x Slower
- torchdiffeq: 30,000x Slower
- torchscript torchdiffeq: 30,000x Slower

The torchscript versions are kept as separate scripts to allow for the JITing process to occur, and are called before timing to exclude JIT timing, as per the PyTorch documentation suggestions. Python results were scaled by the number of times ran in timeit.

https://gist.github.com/ChrisRackauckas/cc6ac746e2dfd285c28e0584a2bfd320

Deep Learning: Exploring High Level APIs of Knet.jl and Flux.jl in comparison to Tensorflow-Keras

Jun 20, 2019 *by* Al-Ahmadgaid B. Asaad

Training CNN (VGG-style) on CIFAR-10 - Image Recognition

DL Library	Test Accuracy (%)	Training Time (s)
MXNet	77	145
Caffe2	79	148
Gluon	76	152
Knet(Julia)	78	159
Chainer	79	162
CNTK	78	163
PyTorch	78	169
Tensorflow	78	173
Keras(CNTK)	77	194
Keras(TF)	77	241
Lasagne(Theano)	77	253
Keras(Theano)	78	269

model	dataset	epochs	batch	Knet	Theano	Torch	Caffe	TFlow
LinReg	Housing	10K	506	2.85	1.88	2.66	2.37	5.92
Softmax	MNIST	10	100	2.35	1.40	2.88	2.82	5.57
MLP	MNIST	10	100	3.68	2.31	4.03	3.75	6.94
LeNet	MNIST	1	100	3.59	3.03	1.69	3.54	8.77
CharLM	Hiawatha	1	128	2.25	4.57	2.23	–	2.86

Deep Learning: Exploring High Level APIs of Knet.jl and Flux.jl in comparison to Tensorflow-Keras

Jun 20, 2019 *by* Al-Ahmadgaid B. Asaad

An Open Source Machine Learning Framework for Everyone https://tensorflow.org

tensorflow machine-learning python deep-learning deep-neural-networks neural-network ml distributed

C++ 53.0% Python 38.4% HTML 3.7% Jupyter Notebook 1.3% Go 1.3% Java 0.7% Other 1.6%

Koç University deep learning framework.

knet deep-learning julia machine-learning neural-networks data-science

Julia 50.6% Jupyter Notebook 41.4% Cuda 5.1% C 0.9% Makefile 0.9% MATLAB 0.9% Other 0.2%

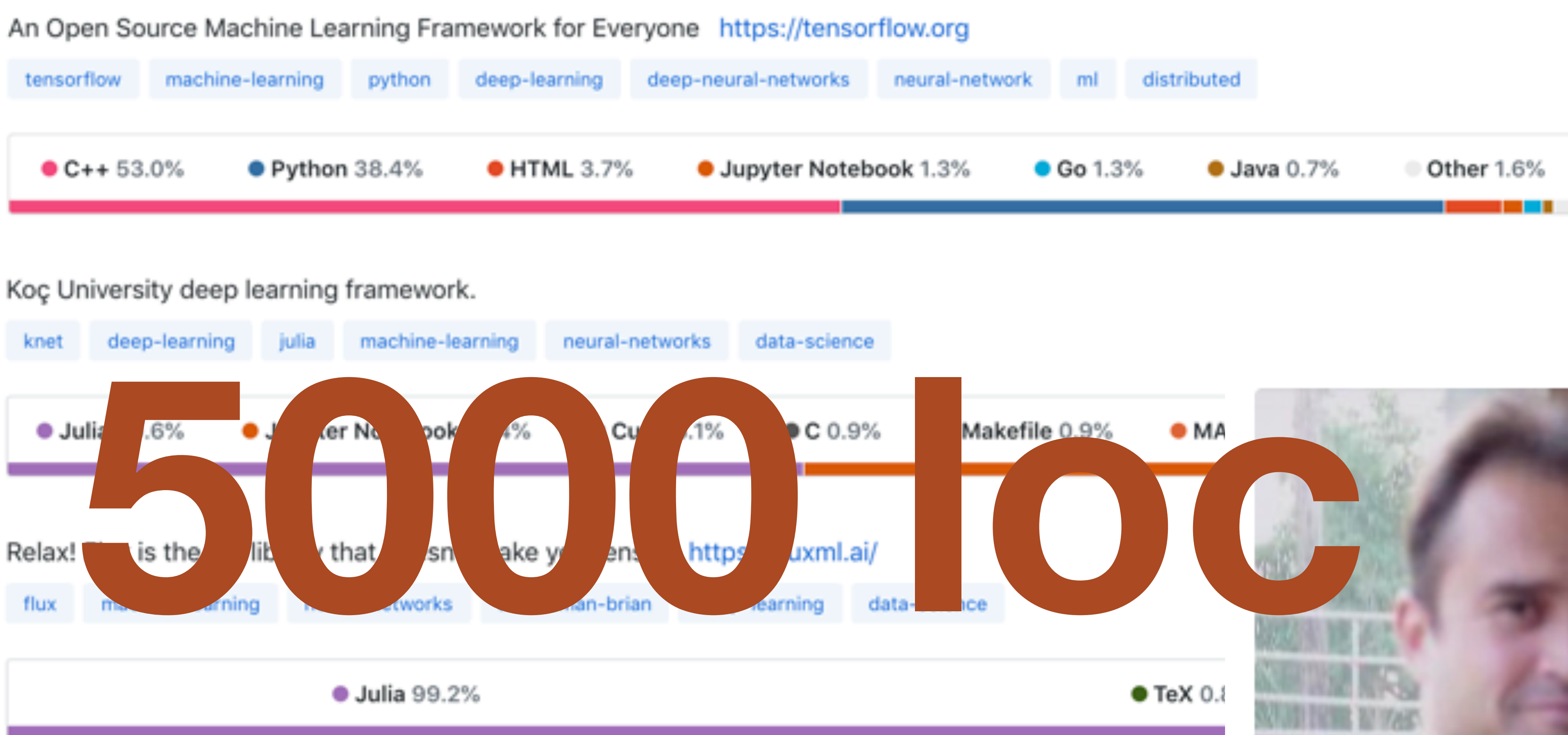
Relax! Flux is the ML library that doesn't make you tensor https://fluxml.ai/

flux machine-learning neural-networks the-human-brian deep-learning data-science

Julia 99.2% TeX 0.8%

Deep Learning: Exploring High Level APIs of Knet.jl and Flux.jl in comparison to Tensorflow-Keras

Jun 20, 2019 *by* Al-Ahmadgaid B. Asaad

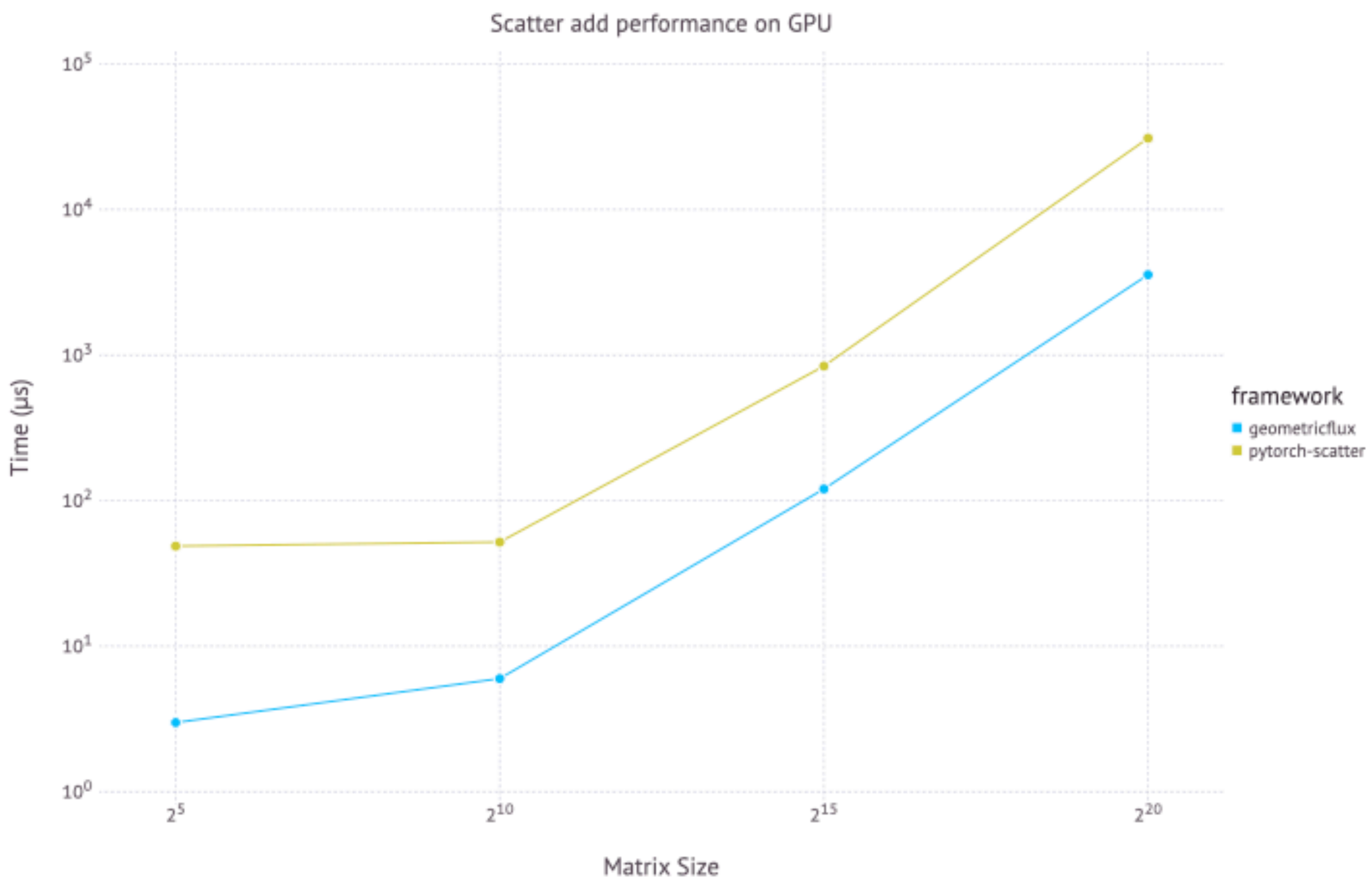


< 5000 loc

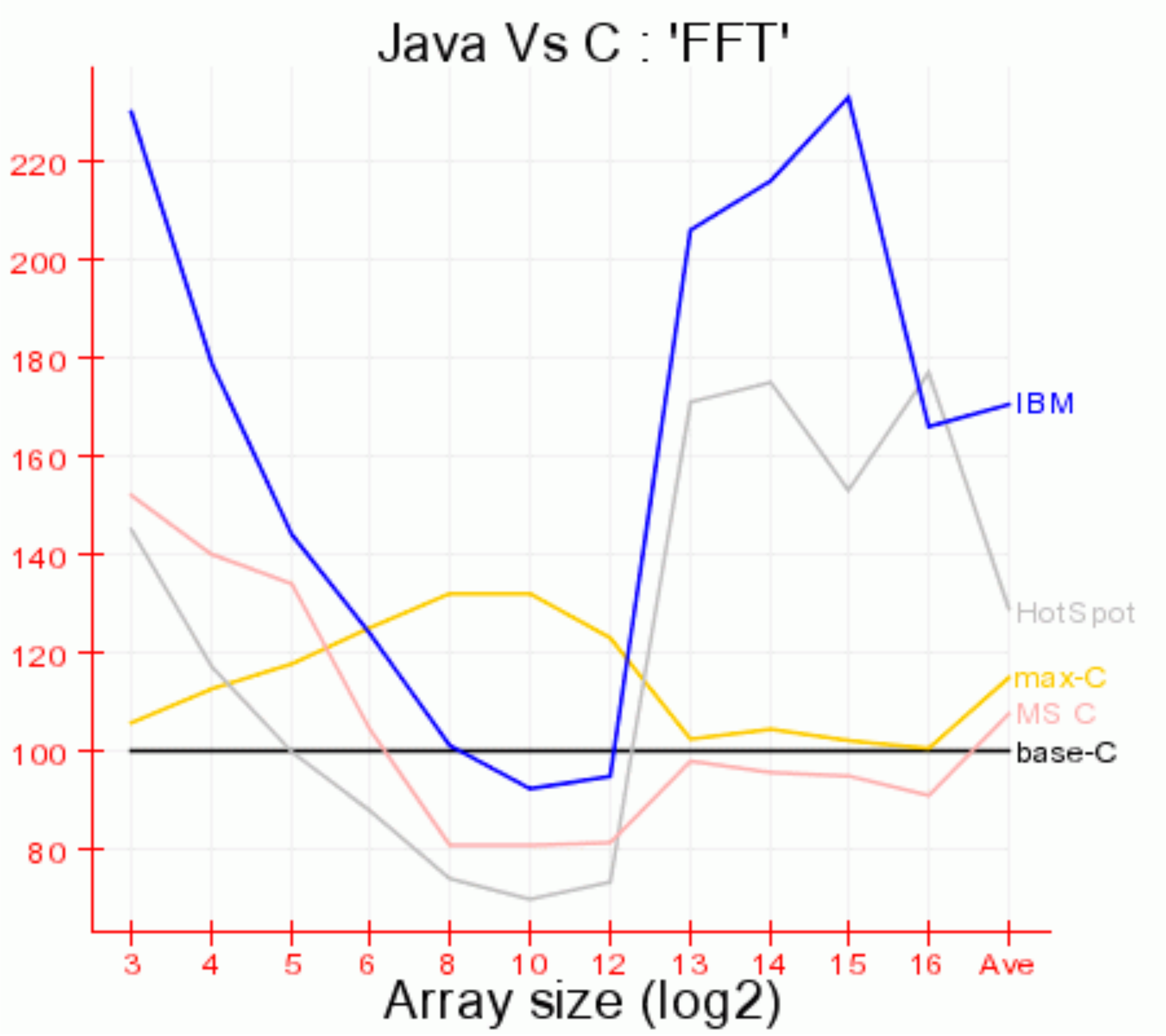
denizyuret

Benchmark

Scatter operations are fundamental to GeometricFlux.jl and they are implemented in CPU and CUDA version. Benchmarks of scatter operations are done with scripts in benchmark folder. Statistics, includes max, min and mean, are shown in the following plots.



but be careful of benchmarks!



speed how? by type annotation?

```julia
function mysum(A)
    thesum = 0
    for i=1:length(A)
        thesum += A[i]
    end
    return thesum
end
```

```julia
function (a::Dense)(x::AbstractVecOrMat)
  W, b, σ = a.weight, a.bias, a.σ
  return σ.(W*x .+ b)
end
```

- mantra: strictly type your types,

  loosely type your functions.

speed how? by type annotation?

```julia
function mysum(A)
    thesum = 0
    for i=1:length(A)
        thesum += A[i]
    end
    return thesum
end
```

```julia
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

```julia
(a::Dense{<:Any,W})(x::AbstractArray{<:AbstractFloat}) where {T <: Union{Float32,Float64}, W <: AbstractArray{T}} =
  a(T.(x))
```

speed how?

- compile, on the fly, for every encountered type

Type-Dispatch Design: Post Object-Oriented Programming for Julia

May 29 2017 in Julia, Programming | Tags: | Author: Christopher Rackauckas

In this post I am going to try to explain in detail the type-dispatch design which is used in Julian

Categories

- Mathematics
  - Differential Equations
  - Stochastics
- Programming

```
my_square(x) = x^2
```

then we see that this function will be efficient for the types that we give it. Looking at the generated code:

```
@code_llvm my_square(1)
define i64 @julia_my_square_72669(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

speed how

```
my_square(x) = x^2
```

Thus we don't need to restrict the types we allow in functions in order to get performance. That means that

```
my_restricted_square(x::Int) = x^2
```

is no more efficient than the version above, and actually generates the same exact compiled code:

```
@code_llvm my_restricted_square(1)

define i64 @julia_my_restricted_square_72686(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

```
@code_llvm my_square(1)
define i64 @julia_my_square_72669(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

```
@code_llvm my_square(1.0)
define double @julia_my_square_72684(double) #0 {
top:
  %1 = fmul double %0, %0
  ret double %1
}
```

```
function mysum(A)
    thesum = 0.
    for i=1:N
        thesum += A[i]
    end
    return thesum
end

const M = [1.,2.]
```

```
      (thesum = 0.0)
%2  = (1:Main.N)::Any
      (@_4 = Base.iterate(%2))
%4  = (@_4 === nothing)::Bool
%5  = Base.not_int(%4)::Bool
      goto #4 if not %5
%7  = @_4::Any
      (i = Core.getfield(%7, 1))
%9  = Core.getfield(%7, 2)::Any
```

less code

```julia
function (a::Dense)(x::AbstractVecOrMat)
  W, b, σ = a.weight, a.bias, a.σ
  return σ.(W*x .+ b)
end
```

Question about source code of pytorch

linyu Aug '18

Where can I find the source code of torch.mm?

1

created	last reply	replies	views	users	likes	link
Aug '18	Aug '18	2	348	2	2	1

SimonW Simon Wang Aug '18

It eventually dispatches to https://github.com/pytorch/pytorch/blob/2e0dd8690320fb1a7ecd548730824c1610207179/aten/src/ATen/native/LinearAlgebra.cpp#L136-L148 58, which calls blas gemm.

metaprogramming

the only feature a language needs

- “homoiconic”, vs. C++ metaprogramming

- program is a convenient data structure, available for manipulation at compile time

metaprogramming

```
julia> ex = :(a + b)
:(a + b)

julia> typeof(ex)
Expr

julia> ex.head
:call

julia> ex.args
3-element Array{Any,1}:
 :+
 :a
 :b
```

metaprogramming

```
julia> ex = :(a + b)
:(a + b)

julia> ex.args[2],ex.args[3] = ex.args[3],ex.args[2]
(:b,:a)

julia> ex.args[1] = :*
:*

julia> ex
:(b * a)
```

metaprogramming   no more copy/paste/bug

Typical example: need nearly parallel code for data structure with .X, .Y, .Z fields
Loop over X,Y,Z fields, generate 3 functions at compile time
each of which sums or averages one of the coordinates.

```
type Point{T <: Number}
  X::T
  Y::T
  Z::T
end

Pts = Array{Point{Float32}}(3)
Pts[1] = Point{Float32}(1,2,3)  # etc

for (name,field) in ((:sumX, :X), (:sumY, :Y), (:sumZ, :Z))
  @eval begin

    function $name(ptarr)                    # note $name
      thesum = 0
      for i = 1:length(ptarr)
        println(ptarr[i].$fieldname)
          thesum += ptarr[i].$fieldname      # note $fieldname
      end
      thesum
    end

  end
end
```

example: metaprogramming vs o-o

- no O-O? metaprogram it in an afternoon (example: Paul Graham book)
- O-O but no metaprogramming? oh well

Paul Graham example

- inheritance in 6 lines of code
- extend to before/after methods, method combination, appropriate syntax
  … in an afternoon

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (unless (meth-p (gethash ',name ,gobj))
         (setf (gethash ',name ,gobj) (make-meth)))
       (setf (,(symb 'meth- type) (gethash ',name ,gobj))
             ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
  (let ((gargs (gensym)))
    `#'(lambda (&rest ,gargs)
         (labels
           ((call-next ()
              ,(if (or (eq type :primary)
                       (eq type :around))
                   `(cnm ,gobj ',name (cdr ,gargs) ,type)
                   '(error "Illegal call-next.")))
            (next-p ()
              ,(case type
                 (:around
                  `(or (rget ,gobj ',name :around 1)
                       (rget ,gobj ',name :primary)))
                 (:primary
                  `(rget ,gobj ',name :primary 1))
                 (t nil))))
           (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
  (case type
    (:around  (let ((ar (rget obj name :around 1)))
                (if ar
                    (apply ar obj args)
```

metaprogramming

Because @def works at compile-time, there is no cost associated with this. Similar metaprogramming can be used to build an "inheritance feature" for Julia. One package which does this is ConcreteAbstractions.jl which allows you to add fields to abstract types and make the child types inherit the fields:

```
# The abstract type
@base type AbstractFoo{T}
    a
    b::Int
    c::T
    d::Vector{T}
end

# Inheritance
@extend type Foo <: AbstractFoo
    e::T
end
```

```
type Foo{T} <: AbstractFoo
    a
    b::Int
    c::T
    d::Vector{T}
    e::T
end
```

Imagine that we want following JSON notation to build a nested dictionary/list(vector).

```
@json {
a: 1,
b: [2, 3 * 3],
c : {
    d: "doubly-quoted string",
    e
},
f: g
}
```

The implementation is:

```
using MLStyle
json(node) =
    @match node begin
        :({ $(kvs...) }) =>
                let f =
                    @λ (k :: Symbol && Do(v = k) ||
                        :($k : $v)) -> Expr(:call, =>, stri
                    Expr(:call, Dict, (f(kv) for kv in kvs :
                end
        :[$(elts...)] => Expr(:vect, map(json, elts)...)
        a => a
    end

macro json(expr)
    json(expr) |> esc
end
```

Transducers.jl: Efficient transducers for Julia

docs stable | docs dev | build passing | codecov 90% | coverage 94% | Aqua.jl

Transducers.jl provides composable algorithms on "sequence" of inputs. They are called *transducers*, first introduced in Clojure language by Rich Hickey.

Using transducers is quite straightforward, especially if you already know similar concepts in iterator libraries:

```
using Transducers
xf = Partition(7) |> Filter(x -> prod(x) % 11 == 0) |> Cat() |> Scan(+)
foldl(+, xf, 1:40)
```

However, the protocol used for the transducers is quite different from iterators and results in a better performance for complex compositions. Furthermore, some transducers support parallel execution. If a transducer is composed of such transducers, it can be automatically re-used both in sequential (`foldl` etc.) and parallel (`reduce` etc.) contexts.

Vectorized Constraints and Objective

We can also add constraints and objective to JuMP using vectorized linear algebra. We'll illustrate this by solving an LP in standard form i.e.

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \succeq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

In [32]:

```
vector_model = Model(GLPK.Optimizer)

A= [ 1 1 9  5;
     3 5 0  8;
     2 0 6 13]

b = [7; 3; 5]

c = [1; 3; 5; 2]

@variable(vector_model, x[1:4] >= 0)
@constraint(vector_model, A * x .== b)
@objective(vector_model, Min, c' * x)

optimize!(vector_model)

@show objective_va
```

```
objective_value(ve
```

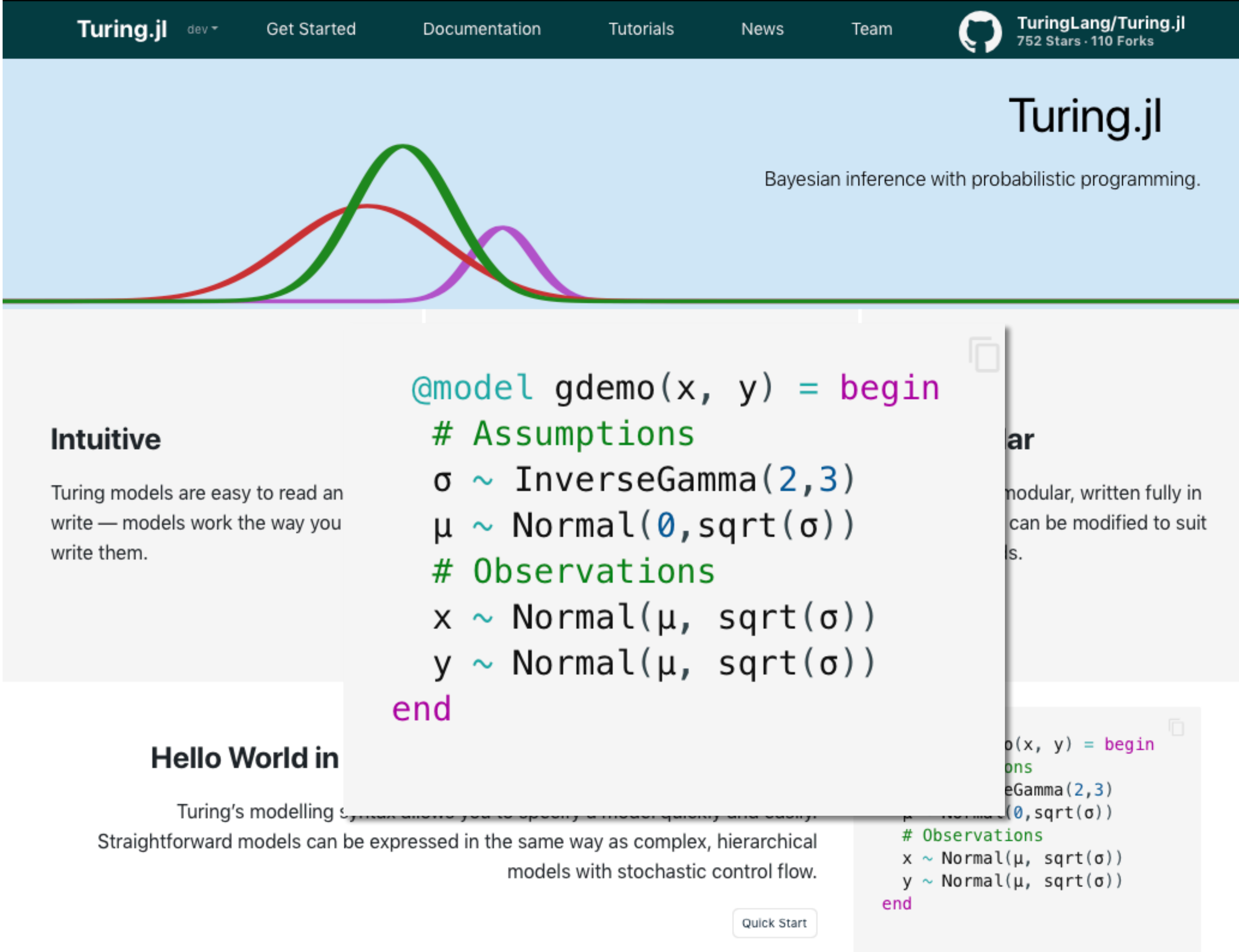
What package[s] are state-of-the art OR attract you to Julia, an

Usage

ExpandingMan 3 May '18

Definitely JuMP. The Python equivalents are a joke.

I think some of the simple stuff is really underrated. I can't express to you how strongly I prefer Julia DataFrames over pandas. They are so lightweight in simple, it's so easy to work on them just using functions from `Base`. As I've said elsewhere, for the most part the only really specialized functions I use

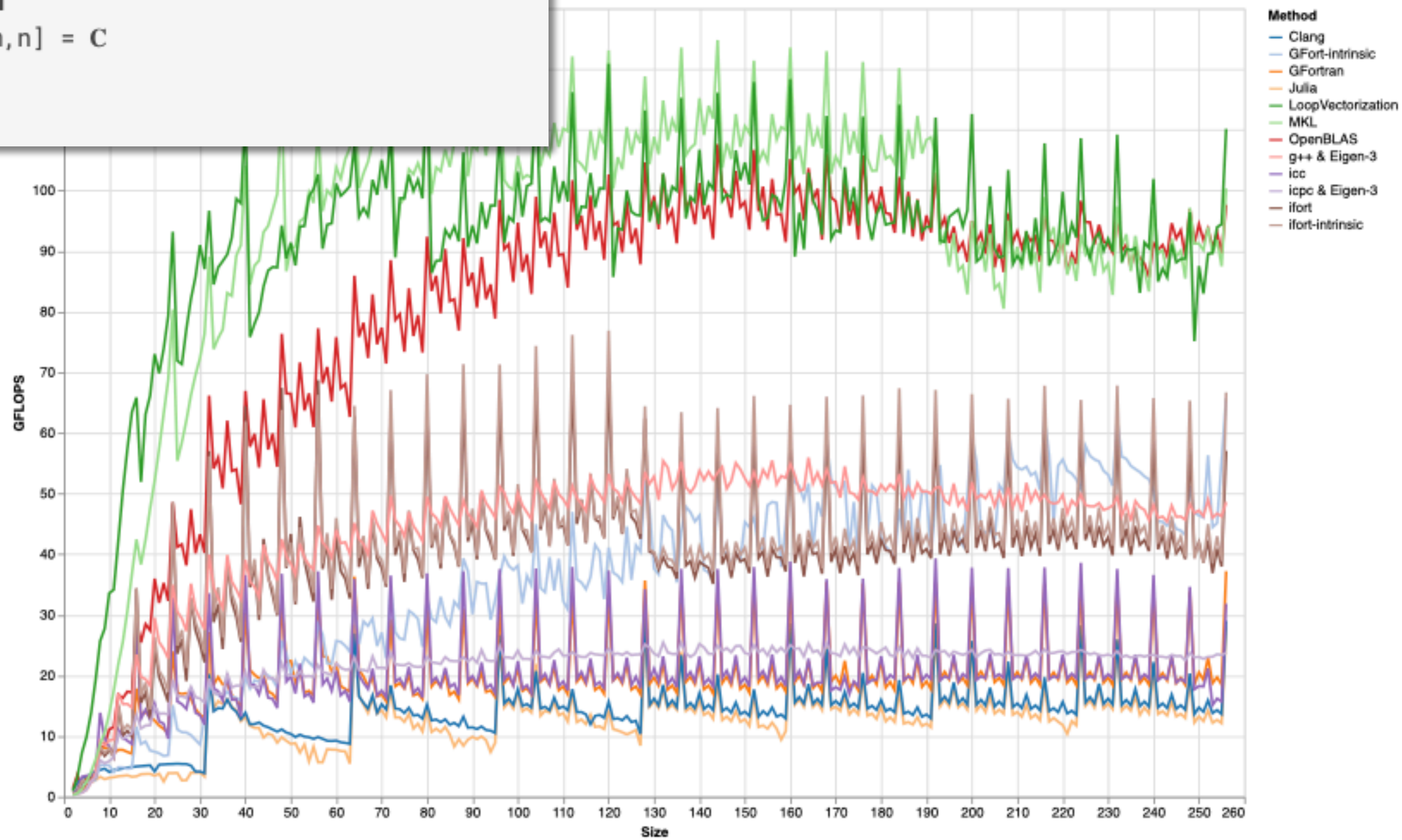

Turing.jl
dev
Get Started
Documentation
Tutorials
News
Team
TuringLang/Turing.jl
752 Stars · 110 Forks
Turing.jl
Bayesian inference with probabilistic programming.
@model gdemo(x, y) = begin
Assumptions
σ ~ InverseGamma(2,3)
μ ~ Normal(0,sqrt(σ))
Observations
x ~ Normal(μ, sqrt(σ))
y ~ Normal(μ, sqrt(σ))
end
Intuitive
Turing models are easy to read an
write — models work the way you
write them.
modular, written fully in
can be modified to suit
Hello World in
Turing's modelling
Straightforward models can be expressed in the same way as complex, hierarchical
models with stochastic control flow.
Quick Start

domain-specific minicompilers

- Loop unroll
- memory padding
- conv kernel edge conditions

LoopVectorization.jl

```julia
function A_mul_B!(C, A, B)
    @avx for m ∈ 1:size(A,1), n ∈ 1:size(B,2)
        C   = zero(eltype(C))
        for k ∈ 1:size(A,2)
            C   += A[m,k] * B[k,n]
        end
        C[m,n] = C
    end
end
```



This is classic GEMM, `C = A * B`. GFortran's intrinsic `matmul` function does fairly well, as does Clang-Polly, because Polly is designed to specfically recognize GEMM-like loops and optimize them. But all the compilers are well behind LoopVectorization here, which falls behind MKL's `gemm` beyond `56 × 56`. The problem imposed by alignment is also

Generic GPU Kernels in Julia

Julia has a library called CUDAnative, which hacks the compiler to run your code on GPUs.

```
using CuArrays, CUDAnative

xs, ys, zs = CuArray(rand(1024)), CuArray(rand(1024)), CuArray(zeros(1024))

function kernel_vadd(out, a, b)
  i = (blockIdx().x-1) * blockDim().x + threadIdx().x
  out[i] = a[i] + b[i]
  return
end

@cuda (1, length(xs)) kernel_vadd(zs, xs, ys)

@assert zs == xs + ys
```

Is this better than writing CUDA C? At first, it's easy to mistake this for simple syntactic convenience, but I'm convinced that it brings something fundamentally new to the table. Julia's powerful array abstractions turn out to be a great fit for GPU programming, and it should be of interest to GPGPU hackers regardless of whether they use the language already.

http://mikeinnes.github.io/2017/08/24/cudanative.html

For example, take a CPU kernel that adds two 2D arrays:

```
function add!(out, a, b)
  for i = 1:size(a, 1)
    for j = 1:size(a, 2)
      out[i,j] = a[i,j] + b[i,j]
    end
  end
end
```

This kernel is fast, but hard to generalise across different numbers of dimensions. The change needed to support 3D arrays, for example, is small and mechanical (add an extra inner loop), but we can't write it using normal functions.

Julia's code generation enables an elegant, if slightly arcane, solution:

http://mikeinnes.github.io/2017/08/24/cudanative.html

Julia's code generation enables an elegant, if slightly arcane, solution:

```
using Base.Cartesian

@generated function add!(out, a, b)
  N = ndims(out)
  quote
    @nloops $N i out begin
      @nref($N, out, i) = @nref($N, a, i) + @nref($N, b, i)
    end
  end
end
```

The `@generated` annotation allows us to hook into Julia's code specialisation; when the function receives matrices as input, our custom code generation will create and run a twice-nested loop. This will behave the same as our `add!` function above, but for arrays of any dimension. If you remove `@generated` you can see the internals.

http://mikeinnes.github.io/2017/08/24/cudanative.html

If you try it with, say, a seven dimensional input, you'll be glad you didn't have to write the code yourself.

```
for i_7 = indices(out, 7)
  for i_6 = indices(out, 6)
    for i_5 = indices(out, 5)
      for i_4 = indices(out, 4)
        for i_3 = indices(out, 3)
          for i_2 = indices(out, 2)
            for i_1 = indices(out, 1)
              out[i_1, i_2, i_3, i_4, i_5, i_6, i_7] = a[i_1, i_2, i_3, i_4, i_5, i_6, i_7] +
# Some output omitted
```

http://mikeinnes.github.io/2017/08/24/cudanative.html

Functions for Nothing

Julia has more tricks up its sleeve. It automatically specialises higher-order functions, which means that if we write:

```
function kernel_zip2(f, out, a, b)
  i = (blockIdx().x-1) * blockDim().x + threadIdx().x
  out[i] = f(a[i], b[i])
  return
end

@cuda (1, length(xs)) kernel_zip2(+, zs, xs, ys)
```

It behaves and performs *exactly* like `kernel_vadd`; but we can use any binary function without extra code. For example, we can now subtract two arrays:

```
@cuda (1, length(xs)) kernel_zip2(-, zs, xs, ys)
```

http://mikeinnes.github.io/2017/08/24/cudanative.html

There's no hint of it in our code, but Julia will compile a custom GPU kernel to run this high-level expression. Julia will also fuse multiple broadcasts together, so if we write an expression like

```
y .= σ.(Wx .+ b)
```

This creates a single kernel call, with no memory allocation or temporary arrays required. Pretty cool – and well out of the reach any other system I know of.

http://mikeinnes.github.io/2017/08/24/cudanative.html

& Derivatives for Free

If you look at the original `kernel_vadd` above, you'll notice that there are no types mentioned. Julia is duck typed, even on the GPU, and this kernel will work for anything that supports the right operations.

For example, the inputs don't *have* to be `CuArray`s, as long as they look like arrays and can be transferred to the GPU. If we add a range of numbers to a `CuArray` like so:

```
@cuda (1, length(xs)) kernel_vadd(xs, xs, 1:1024)
```

The range `1:1024` is never actually allocated in memory; the elements `[1, 2, ..., 1024]` are computed on-the-fly as needed on the GPU. The element type of the array is also generic, and only needs to support `+`; so `Int + Float64` works, as above, but we can also use user-defined number types.

A powerful example is the dual number. A dual number is really a pair of numbers, like a complex number; it's a value that carries around its own derivative.

http://mikeinnes.github.io/2017/08/24/cudanative.html

The full broadcasting machinery in CuArrays is *60 lines long*. While not completely trivial, this is an incredible amount of functionality to get from this much code. CuArrays itself is under 400 source lines, while providing almost all general array operations (indexing, concatenation, permutedims etc) in a similarly generic way.

http://mikeinnes.github.io/2017/08/24/cudanative.html

deep learning / differentiation for free

differentiable code for free

- no need to write with special data types or restricted operations

Researchers, users, and developers of Flux

UCL

UNIVERSITY OF TORONTO

Julia computing

The Alan Turing Institute

THE UNIVERSITY of EDINBURGH

relationalAI

UNIVERSITY of WASHINGTON

UNIVERSITY OF CAMBRIDGE

Carnegie Mellon University

INVENIA

BEACON BIOSIGNALS

AMD

Convolutional Conditional Neural Processes

Jonathan Gordon, Wessel P. Bruinsma, Andrew Y. K. Foong, James Requeima, Yann Dubois, Richard E. Turner

25 Sep 2019 (modified: 11 Mar 2020) ICLR 2020 Conference Blind Submission Readers: Everyone Show Bibtex Show Revisions

Original Pdf: pdf

TL;DR: We extend deep sets to functional embeddings and Neural Processes to include translation equivariant members

Abstract: We introduce the Convolutional Conditional Neural Process (ConvCNP), a new member of the Neural Process family that models translation equivariance in the data. Translation equivariance is an important inductive bias for many learning problems including time series modelling, spatial data, and images. The model embeds data sets into an infinite-dimensional function space, as opposed to finite-dimensional vector spaces. To formalize this notion, we extend the theory of neural representations of sets to include functional representations, and demonstrate that any translation-equivariant embedding can be represented using a convolutional deep-set. We evaluate ConvCNPs in several settings, demonstrating that they achieve state-of-the-art performance compared to existing NPs. We demonstrate that building in translation equivariance enables zero-shot generalization to challenging, out-of-domain tasks.

Keywords: Neural Processes, Deep Sets, Translation Equivariance

Code: https://github.com/cambridge-mlg/convcnp

11 Replies Add **Public Comment**

Show all from everybody

[–] **Paper Decision**

ICLR 2020 Conference Program Chairs

19 Dec 2019 (modified: 19 Dec 2019) ICLR 2020 Conference Paper2232

Decision Readers: Everyone

Decision: Accept (Talk)

Comment: This paper presents Convolutional Conditional Neural Process (ConvCNP), a new member of the neural

Using the Hello World guide, you'll start a branch, write comments, and ope

Wessel wesselb

Hey, 👋! I am a PhD student at the Machine Learning Group at the University of Cambridge, supervised by Dr Richard Turner.

Read the guide

wesselb / **ConvCNPs.jl**

Watch

<> Code | Issues 0 | Pull requests 0 | Actions | Projects 0 | Wiki | Security

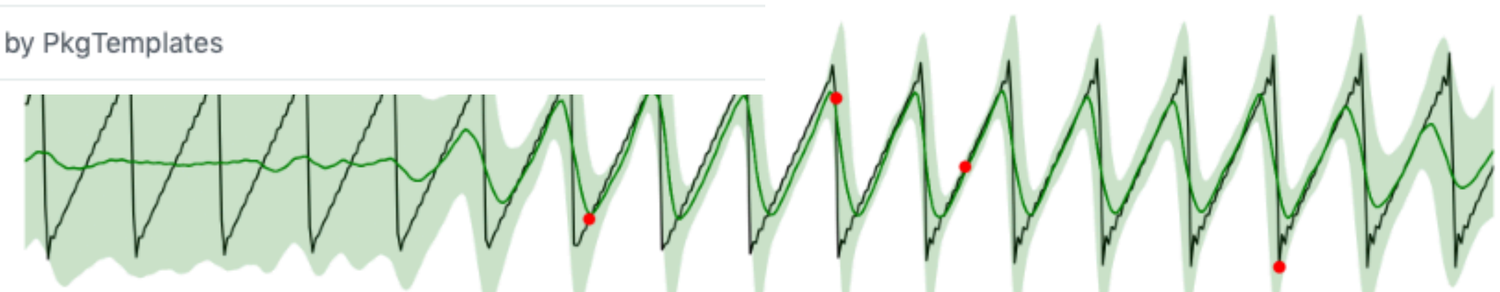
Implementation of the ConvCNP in Julia

141 commits | **3** branches | **0** packages | **0** releases | **2** co

Tree: f5b0061a34 | New pull request | Create new file | Upload

W.P. Bruinsma Adjust experiment and add model

src	Revert "Automatically calibrate BayesianConvCNP"	13 c
test	Add Sawtooth and change construtor of DataGenerator	13 c
.gitignore	Add checkpoints and swap files to gitignore	17 c
LICENSE	Files generated by PkgTemplates	13 c
	del	13 c
	sawtooth	14 c



ConvCNPs.jl

Implementation of the Convolutional Conditional Neural Process.

Generative Ratio Matching Networks

Akash Srivastava, Kai Xu, Michael U. Gutmann, Charles Sutton

25 Sep 2019 (modified: 11 Mar 2020) ICLR 2020 Conference Blind Submission Readers: Everyone Show Bibtex Show Revisions

Original Pdf: pdf

Keywords: deep generative model, deep learning, maximum mean discrepancy, density ratio estimation

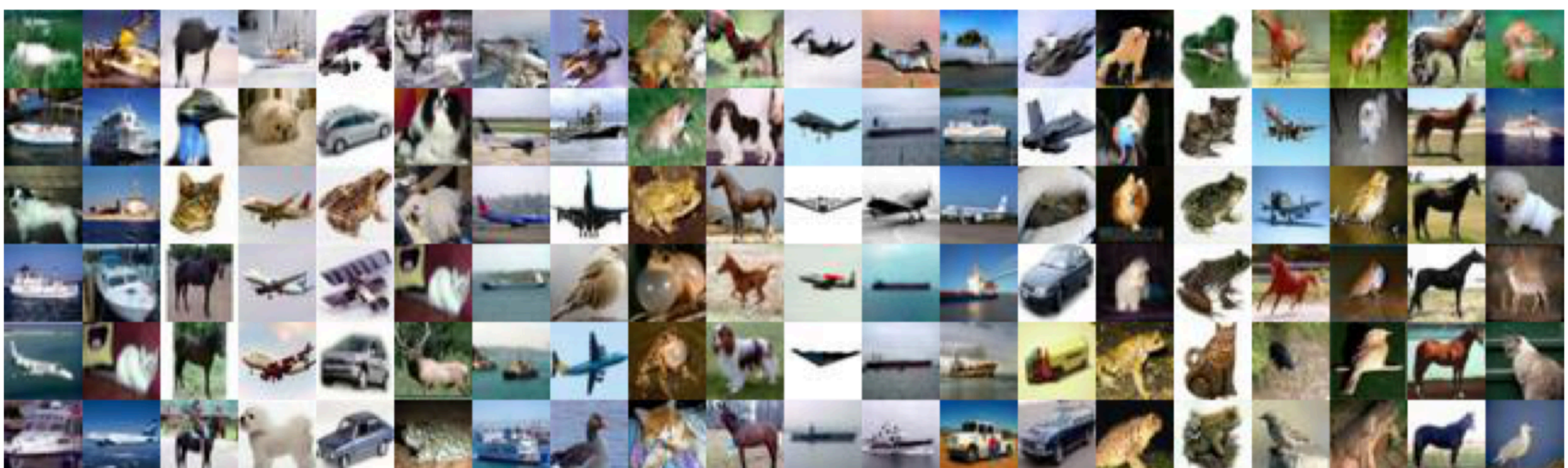
Abstract: Deep generative models can learn to generate realistic-looking images, but many of the most effective methods are adversarial and involve a saddlepoint optimiza
careful balancing of training between a generator network and a critic network. Maximum mean discrepancy networks (MMD-nets) avoid this issue by using kernel as a fixed a
unfortunately, they have not on their own been able to match the generative quality of adversarial training. In this work, we take their insight of using kernels as fixed adversa
novel method for training deep generative models that does not involve saddlepoint optimization. We call our method generative ratio matching or GRAM for short. In GRAM,
networks do not play a zero-sum game against each other, instead, they do so against a fixed kernel. Thus GRAM networks are not only stable to train like MMD-nets but they
generative quality of adversarially trained generative networks.

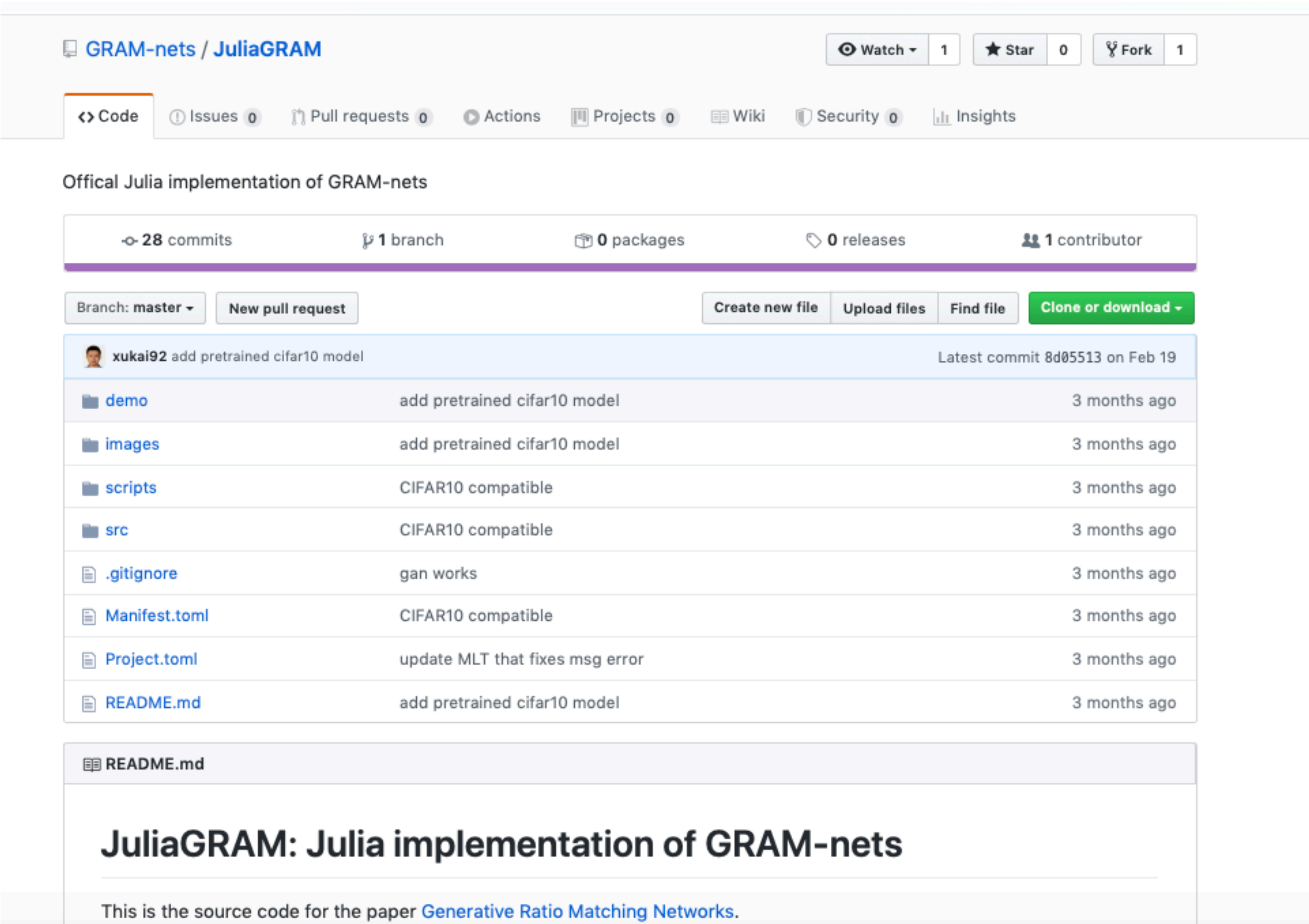
Code: https://github.com/GRAM-nets

TL;DR: MMD-based, saddle-point optimisation free, stable-to-train generative model that beats GAN on generative quality without playing any zero-sum games.

14 Replies

Figure 4: Nearest training images to samples from a GRAM-net trained on Cifar10. In each column, the top image is a sample from the generator, and the images below it are the nearest neighbors.




GRAM-nets / JuliaGRAM
Watch 1
Star 0
Fork 1
Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security 0
Insights
Offical Julia implementation of GRAM-nets
28 commits
1 branch
0 packages
0 releases
1 contributor
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
xukai92 add pretrained cifar10 model
Latest commit 8d05513 on Feb 19
demo add pretrained cifar10 model 3 months ago
images add pretrained cifar10 model 3 months ago
scripts CIFAR10 compatible 3 months ago
src CIFAR10 compatible 3 months ago
.gitignore gan works 3 months ago
Manifest.toml CIFAR10 compatible 3 months ago
Project.toml update MLT that fixes msg error 3 months ago
README.md add pretrained cifar10 model 3 months ago
README.md
JuliaGRAM: Julia implementation of GRAM-nets
This is the source code for the paper Generative Ratio Matching Networks.

https://forums.fast.ai › ann-announcing-fastai-jl-fastai-f...

[ANN] Announcing FastAI.jl: fastai for Julia - fast.ai Forum

Jul 27, 2021 — **jl**, a **fastai**-like library for deep learning in Julia. Features include high-level training loops with hyperparameter scheduling and callbacks, a ...

Julia ML Community Call and **FastAI.jl** - fast.ai Forum	Sep 6, 2020
Why swift instead of Julia - San Francisco - fast.ai Forum	Mar 6, 2019
Any thoughts on fast.ai architecture vs. MLJ or Flux (Julia)?	Sep 12, 2020

More results from forums.fast.ai

Flux: The Julia Machine Learning Library

Flux is a library for machine learning. It comes "batteries-included" with many useful tools built in, but also lets you use the full power of the Julia language where you need it. We follow a few key principles:

- **Doing the obvious thing**. Flux has relatively few explicit APIs for features like regularisation or embeddings. Instead, writing down the mathematical form will work – and be fast.
- **You could have written Flux**. All of it, from LSTMs to GPU kernels, is straightforward Julia code. When in doubt, it's well worth looking at the source. If you need something different, you can easily roll your own.
- **Play nicely with others**. Flux works well with Julia libraries from data frames and images to differential equation solvers, so you can easily build complex data processing pipelines that integrate Flux models.

```julia
struct Dense{F,S,T}
  W::S
  b::T
  σ::F
end

Dense(W, b) = Dense(W, b, identity)

function Dense(in::Integer, out::Integer, σ = identity;
               initW = glorot_uniform, initb = zeros)
  return Dense(initW(out, in), initb(out), σ)
end

@functor Dense

function (a::Dense)(x::AbstractArray)
  W, b, σ = a.W, a.b, a.σ
  σ.(W*x .+ b)
end
```

```julia
mutable struct ADAM
  eta::Float64
  beta::Tuple{Float64,Float64}
  state::IdDict
end

ADAM(η = 0.001, β = (0.9, 0.999)) = ADAM(η, β, IdDict())

function apply!(o::ADAM, x, Δ)
  η, β = o.eta, o.beta
  mt, vt, βp = get!(o.state, x, (zero(x), zero(x), β))
  @. mt = β[1] * mt + (1 - β[1]) * Δ
  @. vt = β[2] * vt + (1 - β[2]) * Δ^2
  @. Δ =  mt / (1 - βp[1]) / (√(vt / (1 - βp[2])) + ϵ) * η
  o.state[x] = (mt, vt, βp .* β)
  return Δ
end
```

```julia
mutable struct Nesterov
  eta::Float64
  rho::Float64
  velocity::IdDict
end

Nesterov(η = 0.001, ρ = 0.9) = Nesterov(η, ρ, IdDict())

function apply!(o::Nesterov, x, Δ)
  η, ρ = o.eta, o.rho
  v = get!(o.velocity, x, zero(x))::typeof(x)
  d = @. ρ^2 * v - (1+ρ) * η * Δ
  @. v = ρ*v - η*Δ
  @. Δ = -d
end
```

DON'T UNROLL ADJOINT: DIFFERENTIATING SSA-FORM PROGRAMS

Michael J Innes [1]

ABSTRACT

This paper presents reverse-mode algorithmic differentiation (AD) based on source code transformation, in particular of the Static Single Assignment (SSA) form used by modern compilers. The approach can support control flow, nesting, mutation, recursion, data structures, higher-order functions, and other language constructs, and the output is given to an existing compiler to produce highly efficient differentiated code. Our implementation is a new AD tool for the Julia language, called Zygote, which presents high-level dynamic semantics while transparently compiling adjoint code under the hood. We discuss the benefits of this approach to both the usability and performance of AD tools.

1 INTRODUCTION

We additionally introduce Zygote, a working implementa-

In the **Flux paper**, we demonstrate the ease with which one is able to take advantage of the underlying ecosystem to express ideas and complicated thoughts. One example is how Flux models can be learned with custom training loops that can house arbitrary logic, including more complex gradient flows than a typical machine learning framework might support.

```
for x, c, d in training_set
    c_hat, d_hat = model(x)
    c_loss = loss(c_hat, y) + λ*loss(d_hat, 1 - d)
    d_loss = loss(d_hat, d)
    back!(c_loss)
    back!(d_loss)
    opt()
end
```

Flux.jl has been shown to run on par with contemporary deep learning libraries while being dramatically simpler, providing intelligent abstractions and maintaining a minimalist API.

```
>>> @code_llvm derivative(x -> 5x+3, 1)
define i64 @"julia_#625_38792"(i64)
{ top:
    ret i64 5
}
```

TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning

Akshay Agrawal [1] Akshay Naresh Modi [1] Alexandre Passos [1] Allen Lavoie [1] Ashish Agarwal [1] Asim Shankar [1] Igor Ganichev [1] Josh Levenberg [1] Mingsheng Hong [1] Rajat Monga [1] Shanqing Cai [1]

Abstract

TensorFlow Eager is a multi-stage, Python-embedded domain-specific language for hardware-accelerated machine learning, suitable for both interactive research and production. TensorFlow, which TensorFlow Eager extends, requires users to represent computations as dataflow graphs; this permits compiler optimizations and simplifies

The paper on TF 2.0 49, which shares many ideas with Jax, discusses this a bit as well:

> In TensorFlow Eager, users must manually stage computations, which might require refactoring code. An ideal framework for differentiable programming would automatically stage computations, without programmer intervention. One way to accomplish this is to embed the framework in a compiled procedural language and implement graph extraction and automatic differentiation as compiler rewrites; this is what, e.g., DLVM, Swift for TensorFlow, and Zygote do. Python's flexibility makes it difficult for DSLs embedded in it to use such an approach.

In TensorFlow Eager, users must manually stage computa- tions, which might require refactoring code (see §4.1). An ideal framework for differentiable programm stage computations, without programmer interven- tion. One way to accomplish this is to embed the framework in a compiled procedural language and implement automatic differentiation as compiler rewrites; this is what, e.g., DLVM, Swift for TensorFlow, and Zygote do (Wei et al., 2017; Lattner & the Swift for TensorFlov 2019). Python's flexibility makes it diffi- cult for DSLs embedded in it to use such an approach.

differentiable code for free

- no need to write with special data types (tf.*, torch.*) or restricted operations (jax)

composable

Reverse-mode

- ReverseDiff.jl: Operator overloading reverse-mode AD. Very well-established.
- Nabla.jl: Operator overloading reverse-mode AD. Used in (its maintainer) Invenia's systems.
- Tracker.jl: Operator overloading reverse-mode AD. Most well-known for having been the AD used in earlier versions of the machine learning package Flux.jl. No longer used by Flux.jl, but still used in several place in the Julia ecosystem.
- AutoGrad.jl: Operator overloading reverse-mode AD. Originally a port of the Python Autograd package. Primarily used in Knet.jl.
- Zygote.jl: IR-level source to source reverse-mode AD. Very widely use Particularly notable for being the AD used by Flux.jl. Also features a secret experimental source to source forward-mode AD.
- Yota.jl: IR-level source to source reverse-mode AD.
- XGrad.jl: AST-level source to source reverse-mode AD. Not currently in active development.
- ReversePropagation.jl: Scalar, tracing-based source to source reverse-mode AD.
- Enzyme.jl: Scalar, LLVM source to source reverse-mode AD. Experimental.
- Diffractor.jl: Next-gen IR-level source to source reverse-mode (and forward-mode) AD. In development.

Forward-mode

- ForwardDiff.jl: Scalar, operator overloading forward-mode AD. Very stable. Very well-established.
- ForwardDiff2: Experimental, non-scalar hybrid operator-overloading/source-to-source forward-mode AD. Not currently in development.
- Diffractor.jl: Next-gen IR-level source to source forward-mode (and reverse-mode) AD. In development.

Symbolic:

- Symbolics.jl: A pure Julia computer algebra system. While its docs focus on some particular domain use-case it is a fully general purpose system.

Exotic

- TaylorSeries.jl: Computes polynomial expansions; which is the generalization of forward-mode AD to nth-order derivatives.
- NiLang.jl: Reversible computing DSL, where everything is differentiable by reversing.

Sparsity

- SparsityDetection.jl: Automatic Jacobian and Hessian sparsity pattern detection.
- SparseDiffTools.jl: Exploiting sparsity to speed up FiniteDiff.jl and ForwardDiff.jl, as well as other algorithms.

https://juliadiff.org

- ChainRules: Extensible, AD-independent rules.
    - ChainRulesCore.jl: Core API for user to extend to add rules to their package.
    - ChainRules.jl: Rules for Julia Base and standard libraries.
    - ChainRulesTestUtils.jl: Tools for testing rules defined with ChainRulesCore.jl.

```julia
#####
##### `dot`
#####

function frule((_, Δx, Δy), ::typeof(dot), x, y)
    return dot(x, y), dot(Δx, y) + dot(x, Δy)
end

function rrule(::typeof(dot), x::AbstractArray, y::AbstractArray)
    project_x = ProjectTo(x)
    project_y = ProjectTo(y)
    function dot_pullback(Ω̄)
        ΔΩ = unthunk(Ω̄)
        x̄ = @thunk(project_x(reshape(y .* ΔΩ', axes(x))))
        ȳ = @thunk(project_y(reshape(x .* ΔΩ, axes(y))))
        return (NoTangent(), x̄, ȳ)
    end
    return dot(x, y), dot_pullback
end
```

creativity

Programming languages teach you not to want what they don’t provide.

Paul Graham

what is art, and why does it exist

of the visual cortex. Specifically, we propose that a broad subset of visual art can be defined as *patterns that are exciting to a visual brain.* Resting on the finding that artificial neural networks trained on visual tasks can provide predictive models of processing in the visual cortex, our definition is operationalized by using a trained deep net as a surrogate "visual brain", where "exciting" is defined as the activation energy of particular layers of this net. We find that this definition

Our methodology rests on the recent discovery that artificial deep nets trained on visual tasks are surprisingly accurate predictive models of both cortical spiking and population aggregate responses of primate visual brains [KRK14, Kri15, GvG15, CKP+16, YD16, WSZ+17]. By making use of this correspondence, we obtain a computational realization of the proposed definition that would not be possible using alternative methods such as brain imaging.

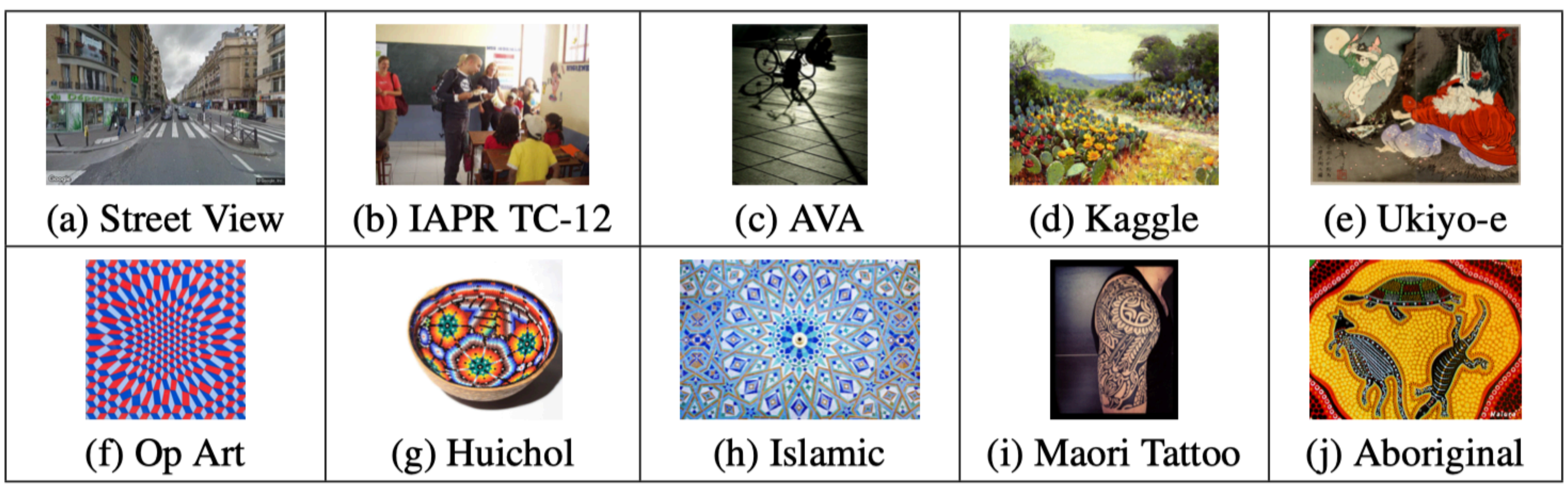
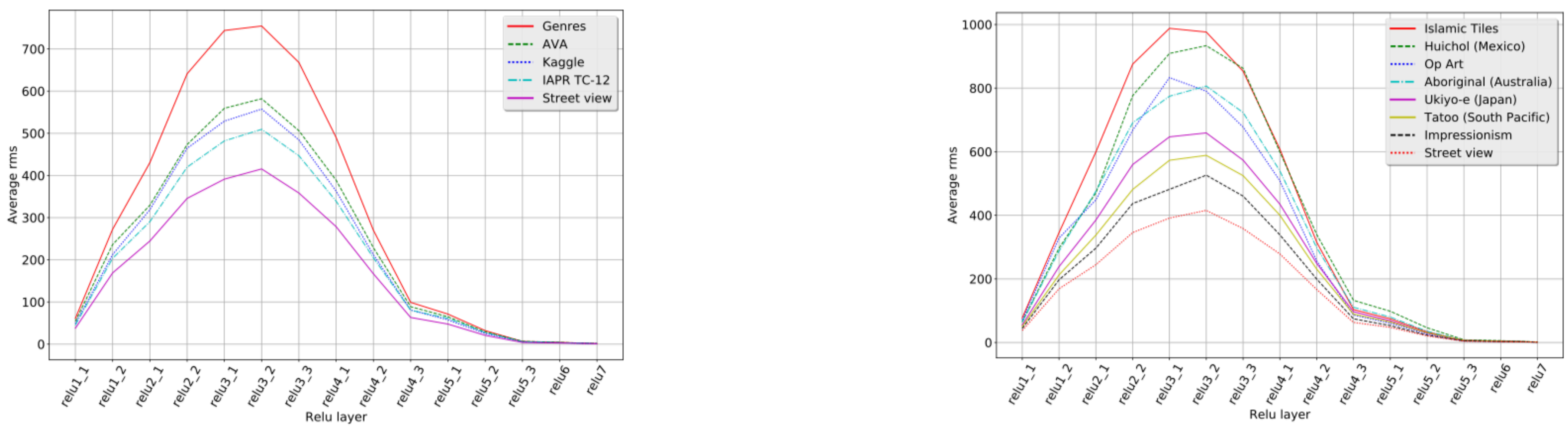


Figure 1: Examples of art and non-art image data used in this study: (a) Google Street View, (b) IAPR TC-12 Benchmark, (c) AVA (artistic photos), (d) Kaggle subset of Wikiart (Impressionism category), (e) Ukiyo-e (Japan), (f) Op Art, (g) Huichol (Mexico), (h) Islamic tile, (i) Maori tattoo (New Zealand), (j) Aboriginal (Australia). Please enlarge to see details.



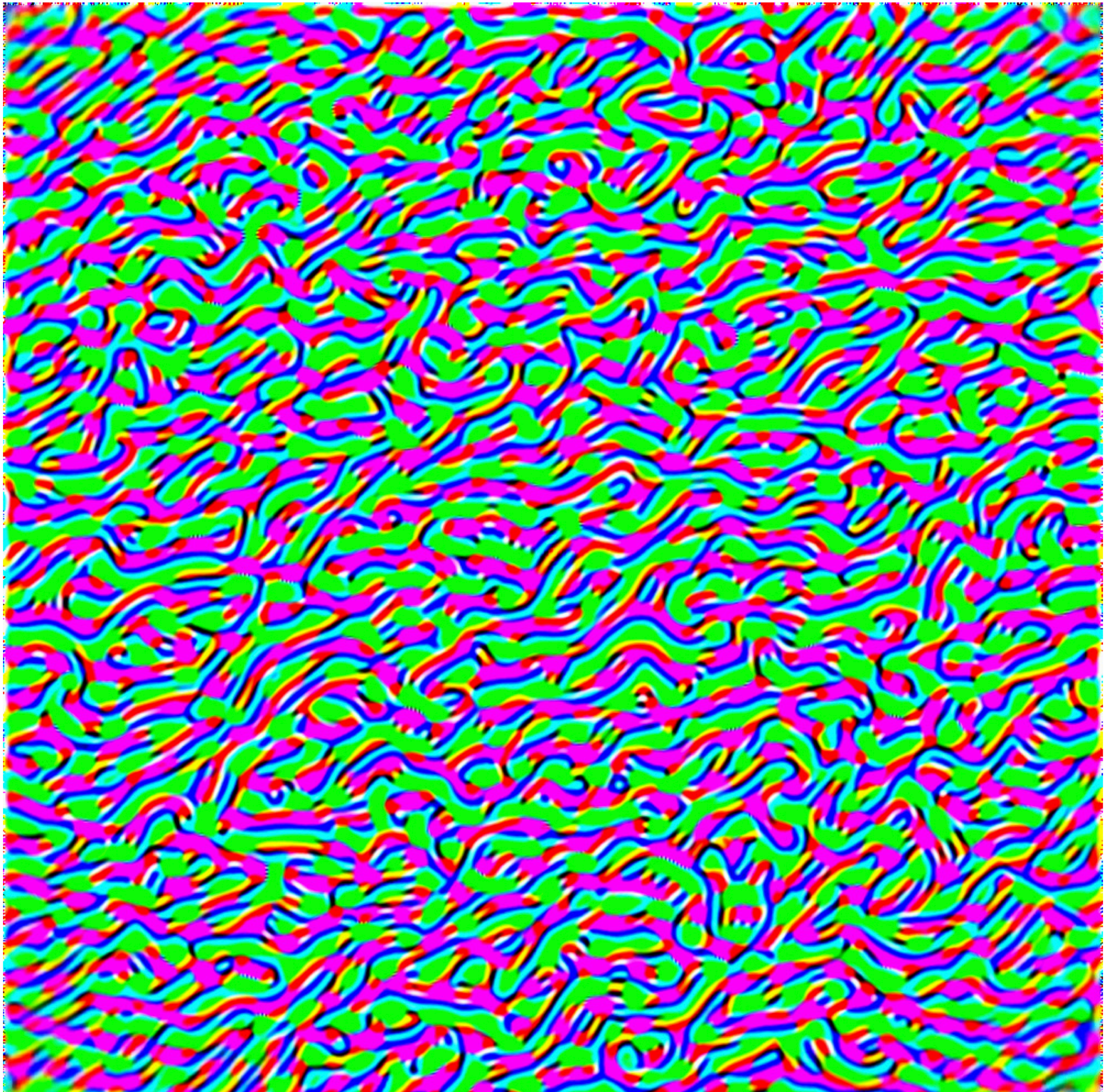
(a) Overall categories

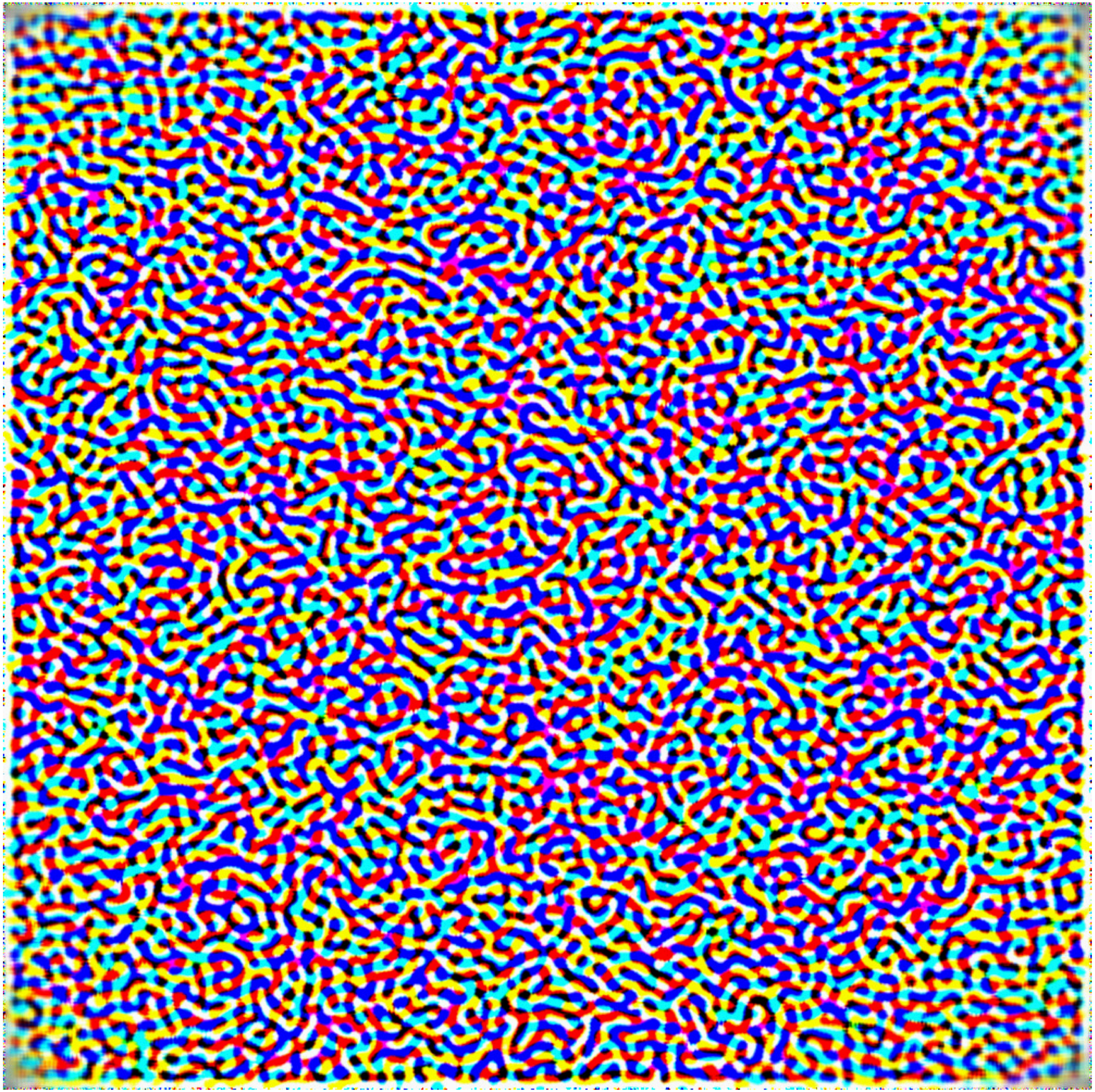
(b) Individual genres

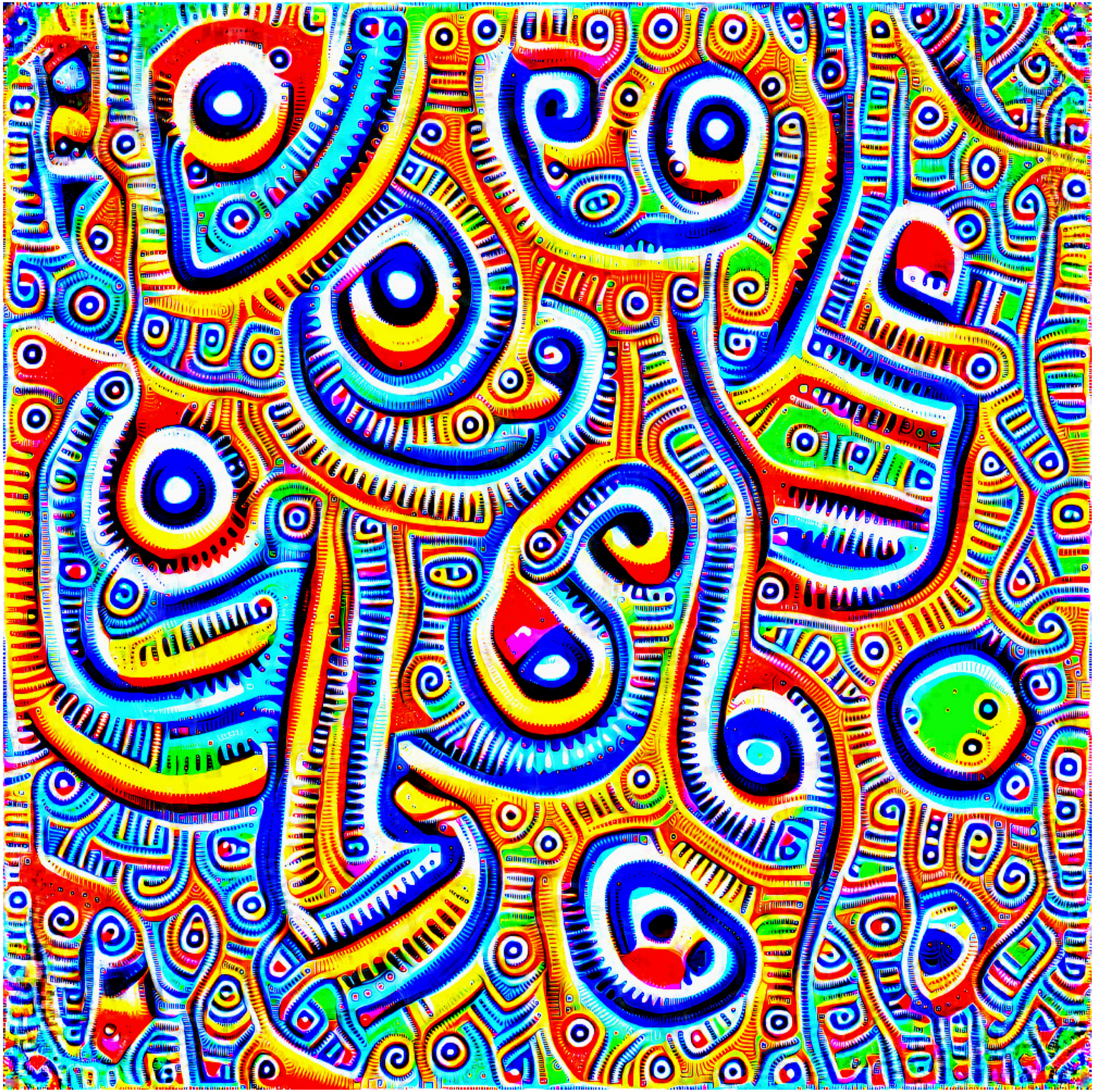
Figure 3: RMS activation at several relu layers of VGG, averaged across images.

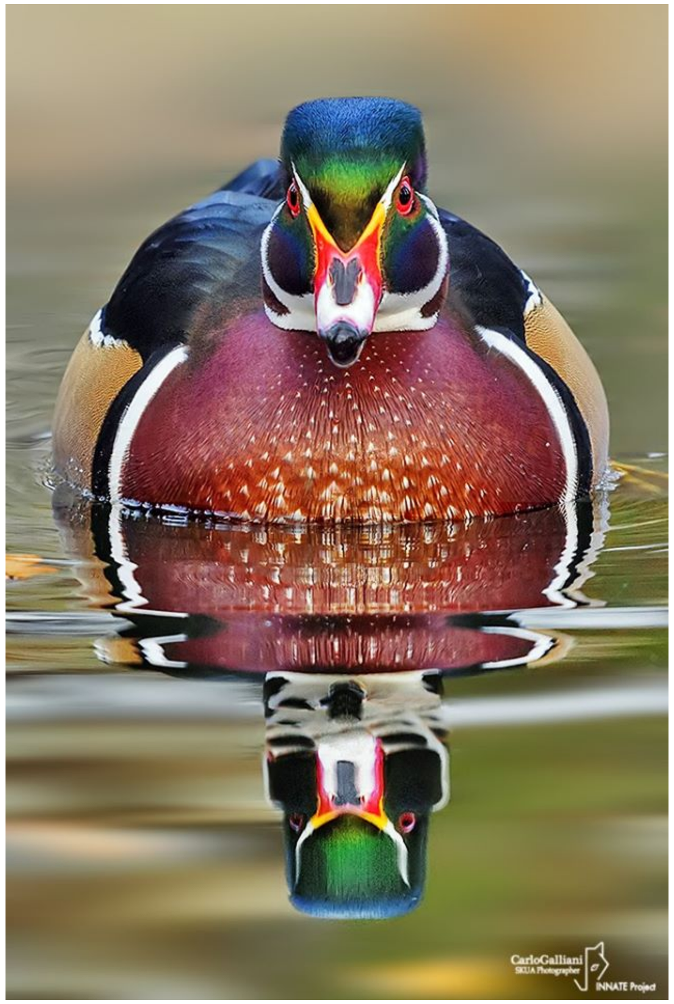
Table 1: One-way permutation analysis

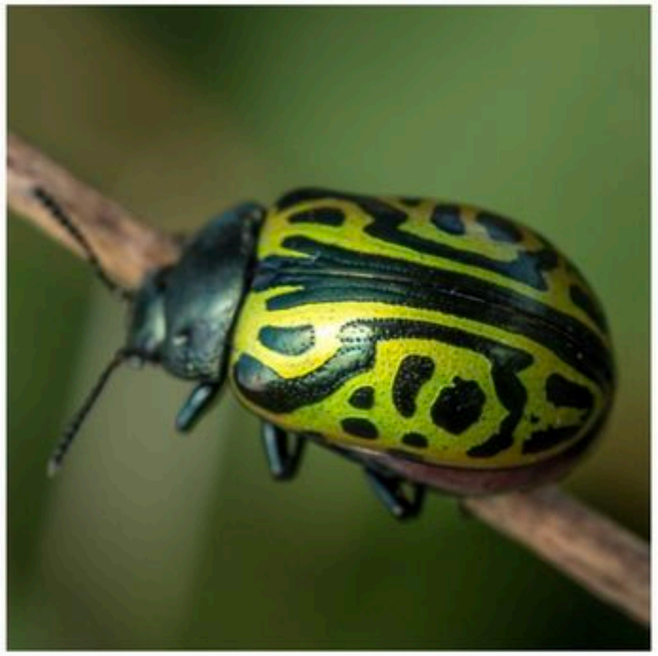
Comparison	Stat	p.value	p.adjust
AVA/IAPR = 0	13.97	0	0.00
AVA/Kaggle = 0	4.317	1.581e-05	0.00
AVA/Genres = 0	-21.13	4.382e-99	0.00
AVA/Street View = 0	30.6	0	0.00
IAPR/Kaggle = 0	-9.28	1.697e-20	0.00
IAPR/Genres = 0	-29.05	1.464e-185	0.00
IAPR/Street View = 0	23.08	0	0.00
Kaggle/Genres = 0	-23.44	1.829e-121	0.00
Kaggle/Street View = 0	26.95	0	0.00
Genres/Street View = 0	36.69	0	0.00

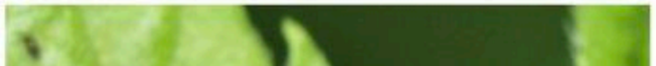







CarloGalliani
SKUA Photographer
INNATE Project





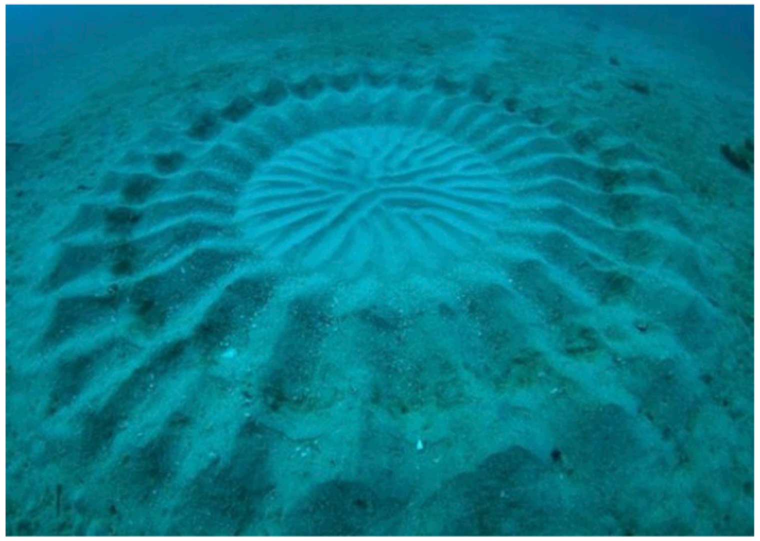


Figure 5: Pattern created by a male pufferfish [KOI13]. Permission to reproduce Figure 3 of [KOI13] obtained from Rightslink copyright clearance center.

composability, multiple dispatch, other

software engineering is not

- O-O: state acts like globals to all (inherited) functions of a class
- Violates **referential transparency.** Hard for humans, hard for compilers.
- Arbitrarily asymmetric: dispatch only on the type of the first arg: `a.mul(b)`
- Julia: dispatch on all types, avoid state wherever possible

multiple dispatch

```
julia> *
* (generic function with 149 methods)

julia> function *(a::String, n::Integer)
         accum = ""
         for i = 1:n
           accum = accum + a                # after defining +
         end
         return accum
       end
* (generic function with 150 methods)

julia> "abc" * 4
"abcabcabcabc"
```

Type magic compositionality

- symbolic for free
- cuarray

type magic:
convert numeric code to symbolic for free

Stochastic Lifestyle

A Random Blog About Math and Life

Home Current Projects Personal Website RSS

Some Fun With Julia Types: Symbolic Expressions in the ODE Solver

May 4 2017 in Differential Equations, Julia, Mathematics, Uncategorized | Tags: | Author: Christopher Rackauckas

In Julia, you can naturally write generic algorithms which work on any type which has specific "actions". For example, an "AbstractArray" is a type which has a specific set of functions implemented. This means that in any generically-written algorithm that wants an array, you can give it an AbstractArray and it will "just work". This kind of abstraction makes it easy to write a simple algorithm and then use that same exact code for other purposes. For example, distributed computing can be done by just passing in a DistributedArray, and the algorithm can be accomplished on the GPU by using a GPUArrays. Because Julia's functions will auto-specialize on the types you give it, Julia automatically makes efficient versions specifically for the types you pass in which, at compile-time, strips away the costs of the abstraction.

Categories

- Mathematics
  - Differential Equations
  - Stochastics
- Programming
  - C
  - CUDA
  - FEM
  - HPC
  - Julia
  - Mathematica
  - MATLAB
  - Xeon Phi

The ODE solvers for Julia are in the package DifferentialEquations.jl. Let's solve the linear ODE:

$$\frac{dy}{dt} = 2y$$

with an initial condition which is a symbolic variable. Following the tutorial, let's swap out the numbers for symbolic expressions. To do this, we simply make the problem type and solve it:

```
using DifferentialEquations, SymEngine
y0 = symbols(:y0)
u0 = y0
f = (t,y) -> 2y
prob = ODEProblem(f,u0,(0.0,1.0))
sol = solve(prob,RK4(),dt=1/10)

println(sol)
# SymEngine.Basic[y0,1.2214*y0,1.49181796*y0,1.822106456344*y0,2.22552082577856*y0,2.71825113
```

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

Ain't no party like a programming language virtual conference party

I've been running into a lot of happy and excited scientists lately. "Running into" in the virtual sense, of course, as conferences and other opportunities to collide with scientists in meatspace have been all but eliminated. Most scientists believe in the germ theory of disease.

Anyway, these scientists and mathematicians are excited about a new tool. It's not a new particle accelerator nor a supercomputer. Instead, this exciting new tool for scientific research is... a computer language.

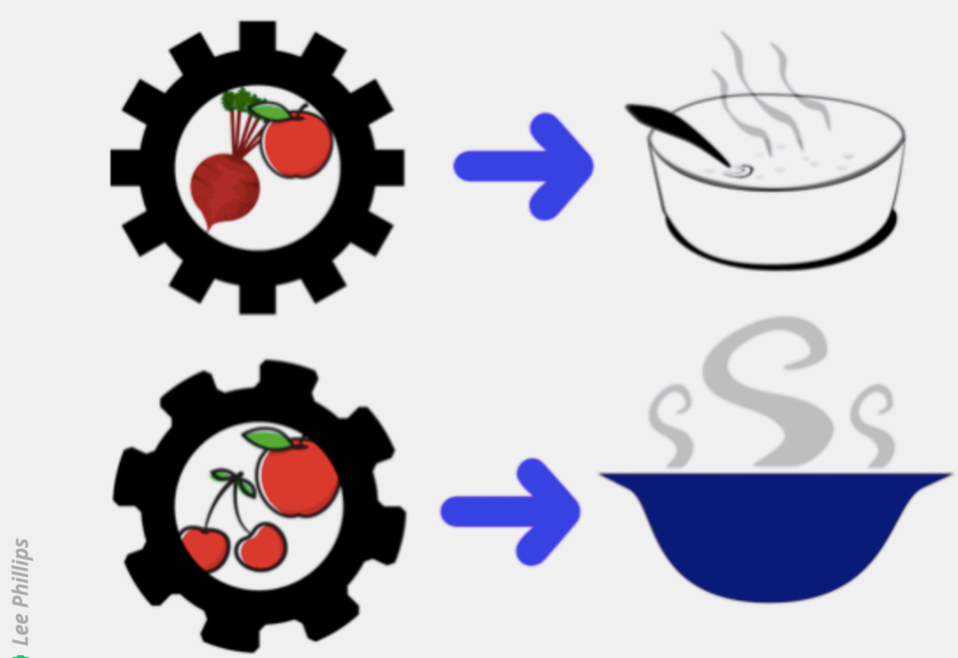
The Expression Problem, via extended analogy

The concept of the "expression problem" arises in the study of the design of computer languages. It is part of the domain of computer science, and so the existing explanations of its meaning, implications, and the various ways around the problem tend to be abstract and rely on a specialized terminology. But we can do better. It's possible to describe all the issues involved by using an analogy to cooking.

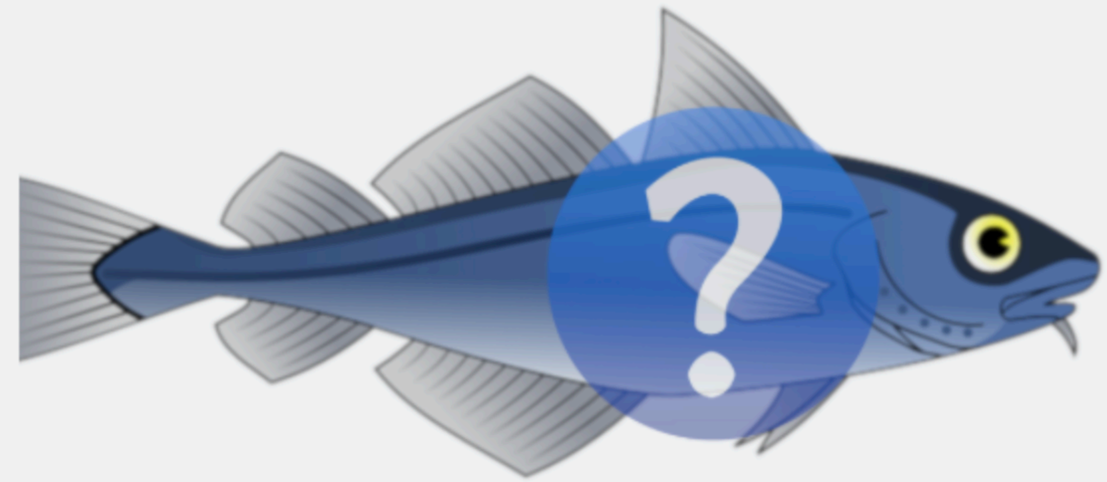
The computer science terms that we would like to analogize are *functions/programs, data types, and libraries/modules/packages*. Briefly, functions or programs are procedures for taking some input, doing something to it, and producing some output. Data types are collections of numbers or other information, which may have various kinds of structure, that the functions operate on. Libraries, etc., are collections of functions, along with descriptions of the data types that they work with, bundled together to perform a set of related tasks. An example of a library would be a set of functions for drawing graphs. The individual functions in the library might be for drawing different types of graphs, like pie charts and histograms. The data type for a pie chart, for example, would be a list of pairs of elements, with the first being a word or phrase and the second a percentage.

For anyone who has spent time in the kitchen creating dishes from recipes, this analogy will be fairly direct and natural. The library or package becomes the recipe book; imagine a somewhat focused book about making desserts, or soups, for example. The functions or programs can be thought of either as complete recipes for making a dish or as techniques or procedures, such as how to sauté. We can visualize them as gears, as they are the machinery for processing raw ingredients. The data types are the raw ingredients in this exercise.

Imagine our recipe book is organized in such a way that recipes only work with certain ingredients. For example, you can look up "how to sauté" and find the procedure, the set of steps, for sautéing onions or sautéing shrimp. All these procedures are *different,* as they use different ingredients. If recipes work like a computer language, the ingredient lists are part of, in fact enclosed within, the recipes.

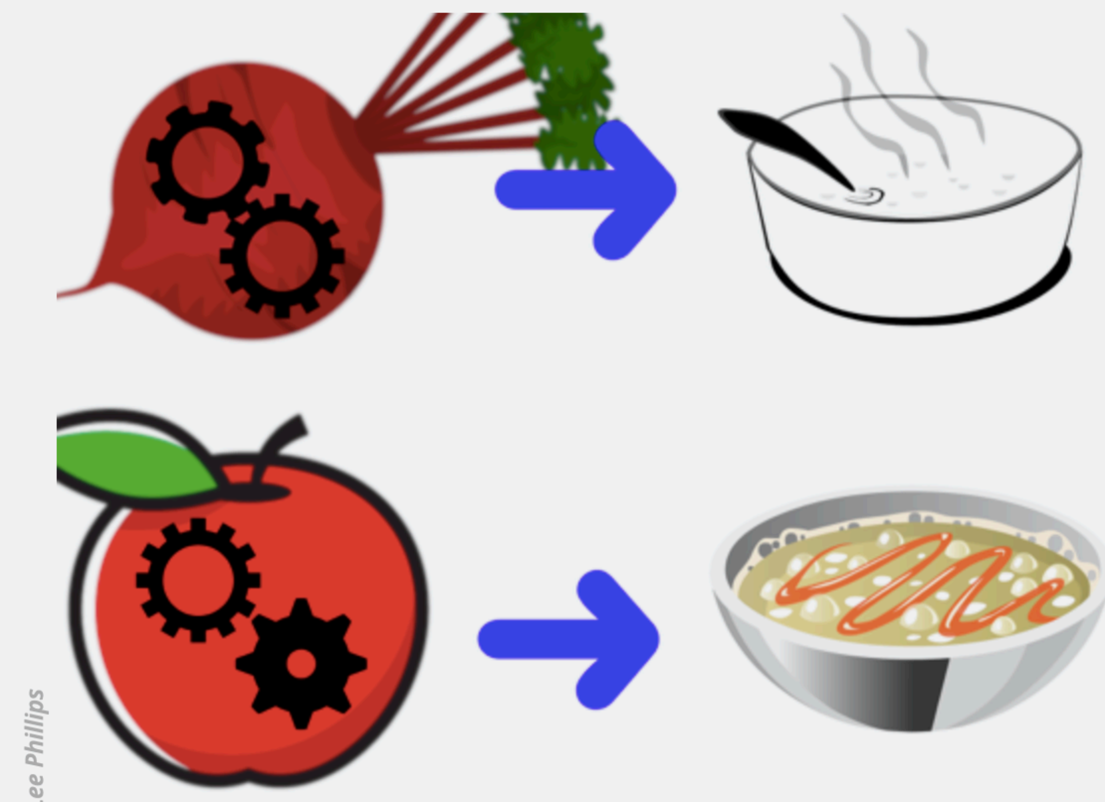


Recipes that only work with specified ingredients.



A new ingredient

There is more than one way to organize a recipe book, however. What if it were organized arou ingredients, rather than around methods of cooking? For each ingredient, there would be a se techniques or methods that go with it. Continuing with our iconography, this could be represented with this picture:



```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

development, teaching, other

- call python (using PyPlot)
- call C

https://github.com/Wikunia/Javis.jl

```julia
function draw_latex(video, action, frame)
    black_red = blend(O, Point(0, 150), "black", "red")
    setblend(black_red)
    fontsize(50)
    latex(
        L"""\begin{equation}
        \left[\begin{array}{cc}
        2 & 3 \\  4 & \sqrt{5} \\
        \end{array} \right]
        \end{equation}""",
        O,
        valign = :middle,
        halign = :center
    )
end
```

The biggest change is that we added the `blend` and `setblend` functions. `blend` creates a linear blend between two points using two given colors - in this case, black and red. `setblend` applies the blend to the drawn object. We also use the `translate` function this time as it makes writing the `blend` function easier.

Can you guess what happens when we execute the code with this newly updated `draw_latex` function? Here is what the output looks like:

$$\left[\begin{array}{cc} 2 & 3 \\ 4 & \sqrt{5} \end{array}\right]$$

- Jupyter = **ju**lia + **py**thon + **R**
- **Pluto**
  - live ("reactive")
  - no hidden state. "**At any instant**, the program state is **completely described** by the code you see."
  - notebooks are julia source (vs ipynb)

https://youtu.be/lAF8DjrQSSk?t=1327

Pluto for education

For students:

- + Easy to install
- + **Live feedback**
- + Fewer confusing bugs

For teachers:

- + Write **engaging** course material
- + Autograding (`.jl`)

Fall 2020

Trial runs with **TU Berlin & MIT**

Feedback from students and teachers

Spring 2021

Guides for writing course material (*template repositories, video tutorials, etc*)

`PlutoEducation.jl` with useful widgets and autograding tools

```
golden_ratio = missing
· golden_ratio = missing
```

Exercise 8: `golden_ratio`
Keep working on it!

tools › Autograder.jl

Autograder.jl
Project ID: 13560

19 Commits **1** Branch **0** Tags **154 KB** Files **154 KB** Storage

Julia client for the Michigan Autograder

master Autograder.jl

Spring 2021 | MIT 18.S191/6.S083/22.S092

Introduction to Computational Thinking

Math from computation, math with computation

by Alan Edelman, David P. Sanders & Charles E. Leiserson

https://computationalthinking.mit.edu/Spring21/newton_method/

https://computationalthinking.mit.edu/Spring21/transforming_images/

https://computationalthinking.mit.edu/Spring21/2d_advection_diffusion/

finally Done