

Jeffrey A. Fessler

EECS Department, BME Department, Dept. of Radiology
University of Michigan

ESMRMB MRI Together workshop

2021-12-15



Declaration: No relevant financial interests or relationships to disclose

MRI Together

A global workshop on Open Science and Reproducibility
December 2021

Speaker name:

Jeff Fessler, EECS, University of Michigan, USA

Conflicts of interest regarding this presentation:

Nothing to disclose

**The copyright of this presentation belongs to the Speaker.
This presentation is released under a [CC-BY](#) license.**



- Conflicts of interest: nothing to disclose
- The copyright of this presentation belongs to Jeff Fessler
- This presentation is released under a CC-BY license

Acknowledgment:

Thanks to many collaborators and research group members

Key URLs:

- <https://juliaimagerecon.github.io/Examples>
- <https://github.com/JeffFessler/MIRT.jl>
- <https://github.com/JeffFessler/mirt>

demos

JULIA version

Matlab version

Introduction

Gradient descent

Regularized LS

Multiple dispatch and operator overloading

Image denoising

Image super-resolution

Bibliography

- ▶ Performance of a compiled language *JIT via LLVM*
- ▶ Interactive development / prototyping *Dynamic typing / Jupyter notebooks*
- ▶ Readability
 - Syntax matching the mathematics of computational imaging *Unicode / UTF-8*
 - Built-in operations with numerical arrays *By design*
 - Namespace control `using , import`
- ▶ Relevant libraries (FFT, linear algebra, statistics, autograd, ...) *FFTW, LAPACK, ...*
- ▶ Easy use of distributed computing, GPU resources *Yes*
- ▶ Library ecosystem that facilitates reproducibility *git, Pkg, Manifest*
- ▶ Free / open source *Yes*
- ▶ IDE / debugger *Atom/Juno, VSCode, ...*
- ▶ Code reuse (e.g., “object oriented”) *multiple dispatch, subtypes*
- ▶ Memory efficient *Float16, Sparse of any type, call by ref.*
- ▶ Interoperable with other languages *ccall, pycall, ...*

- ▶ <https://julialang.org/>
- ▶ Started in 2009 around MIT (Alan Edelman, Jeff Bezanson, Stefan Karpinski, and Viral B. Shah) collaboration between computer sciences and computational sciences
- ▶ First release in 2012
- ▶ Version 1.0 in Aug. 2018
- ▶ 2017 SIAM Review paper [1]
- ▶ 2017 Forbes magazine article
- ▶ 2019 InfoWorld comparison of JULIA and Python
- ▶ 2019 Nature article about JULIA
- ▶ 2019 SIAM CSE Conference: James H. Wilkinson Prize for Numerical Software
- ▶ Sponsors of Juliacon 2018 and Juliacon 2019 include: Microsoft, Amazon, Google, Intel, NVIDIA, CapitalOne, JP Morgan, ...
- ▶ Best of Matlab, Python, LISP, ...

- ▶ Used at UM since 2017 for teaching
 - ▶ EECS 551: Matrix methods in signal processing and machine learning
 - ▶ EECS 505: Computational data science and machine learning
 - ▶ EECS 556: Image processing
 - ▶ EECS 559: Optimization methods for signal and image processing
 - ▶ ROB 101: Computational linear algebra
- ▶ Michigan Image Reconstruction Toolbox (MIRT)
 - ▶ Matlab version (inactive) <https://github.com/JeffFessler/mirt>
 - ▶ Julia version (active) <https://github.com/JeffFessler/MIRT.jl>
- ▶ Group now uses mix of Matlab, python, JULIA...
- ▶ Growing use elsewhere, e.g.,
ENGR108 at Stanford (Intro to Matrix Methods)

Operation	Matlab	JULIA	Python <code>import numpy as np</code>
Dot product	<code>dot(x,y)</code>	<code>dot(x,y)</code>	<code>np.dot(x,y)</code>
Matrix mult.	<code>A * B</code>	<code>A * B</code>	<code>A @ B</code>
Element-wise	<code>A .* B</code>	<code>A .* B</code>	<code>A * B</code>
Scaling	<code>3 * A</code>	<code>3A</code> or <code>3*A</code>	<code>3 * A</code>
Matrix power	<code>A^2</code>	<code>A^2</code>	<code>np.linalg.matrix_power(A,2)</code>
Element-wise	<code>A.^2</code>	<code>A.^2</code>	<code>A**2</code>
Inverse	<code>inv(A)</code>	<code>inv(A)</code>	<code>np.linalg.inv(A)</code>
Inverse	<code>A^(-1)</code>	<code>A^(-1)</code>	<code>np.linalg.inv(A)</code>
Indexing	<code>A(i,j)</code>	<code>A[i,j]</code>	<code>A[i,j]</code>
Range	<code>1:9</code>	<code>1:9</code>	<code>np.arange(1,9,1)</code>
Range	<code>linspace(0,4,9)</code>	<code>LinRange(0,4,9)</code>	<code>np.arange(0,4.01,0.5)</code>
Strings	<code>'text'</code>	<code>"text"</code>	<code>(either)</code>
Inline func.	<code>f = @(x,y) x+y</code>	<code>f(x,y) = x+y</code>	<code>f = lambda x,y : x+y</code>
Increment	<code>A = A + B</code>	<code>A += B</code>	<code>A += B</code>
Herm. transp.	<code>A'</code>	<code>A'</code>	<code>A.conj().T</code>
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	<code>[1 2; 3 4]</code>	<code>[1 2; 3 4]</code>	<code>np.array([[1, 2], [3, 4]])</code>

See <https://cheatsheets.quantecon.org/> for more.

Ways to define the function $f(x, y) = x^2 + y^3$ in JULIA:

▶ `f(x,y) = x^2 + y^3`

▶ `f = (x,y) -> x^2 + y^3`

▶ `function f(x,y)`
 `return x^2 + y^3`
`end`

▶ `function f(x,y)`
 `x^2 + y^3`
`end`

(Functions return the last value evaluated.)

- ▶ Kronecker sum definition in math, for $M \times M$ matrix \mathbf{A} : and $N \times N$ matrix \mathbf{B} :

$$\mathbf{A} \oplus \mathbf{B} \triangleq (\mathbf{I}_N \otimes \mathbf{A}) + (\mathbf{B} \otimes \mathbf{I}_M).$$

- ▶ Kronecker sum function in JULIA:

```
using LinearAlgebra: I
⊗(A,B) = kron(A,B) # Kronecker product
⊕(A,B) = I(size(B,1)) ⊗ A + B ⊗ I(size(A)[1]) # Kronecker sum
X = ones(2,2) ⊕ [1 2; 3 4] # test drive
```

- ▶ `using ...` controls name-space
- ▶ Enter symbols like \otimes with L^AT_EX codes and tab completion:
- ▶ JULIA supports argument chaining, e.g., `size(A)[1]`

```
\otimes<tab>
```

To minimize $f(\mathbf{x})$ using gradient descent (for illustration) in math:

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha \nabla f(\mathbf{x}_{k-1}), \text{ for } k = 1, 2, \dots, N_{\text{iter}}.$$

Gradient descent function code in JULIA looks remarkably similar:

```
function gd(∇f::Function, α::Real, x ; Niter::Int = 50)
    for iter=1:Niter
        x = x - α * ∇f(x)
    end
    return x
end
```

Enter symbols like α with L^AT_EX codes and tab completion:

```
\alpha<tab>
```

```
function gd( $\nabla$ f::Function,  $\alpha$ ::Real, x ; Niter::Int = 50)
```

- ▶ Type annotation (via `::`) is optional (cf. `x`)
- ▶ Functions are first-class object types in JULIA
- ▶ JULIA is dynamically typed
- ▶ `Real` is an “abstract supertype” (step size α is any type of real number)
- ▶ `subtypes(Real)` returns
`[AbstractFloat, AbstractIrrational, Integer, Rational]`
- ▶ `subtypes(AbstractFloat)` returns `[Float16, Float32, Float64, BigFloat]`
(concrete number types: stored as bits)
- ▶ Optional named arguments (after `;`) with default values

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{y}\|_2^2 \implies \nabla f(\mathbf{x}) = \mathbf{A}'(\mathbf{Ax} - \mathbf{y}), \quad L_{\nabla f} = \|\mathbf{A}\|_2^2, \quad \alpha = 1/L_{\nabla f}$$

```
using LinearAlgebra: opnorm # same as svdvals(A)[1]
include("gd.jl") # provides gd()
M,N = 6,4
A = randn(M,N); y = randn(M); # test data
∇f(x) = A'*(A*x - y) # LS gradient
xh = A \ y # global minimizer of f
α = 1/opnorm(A)^2 # step size
xgd = gd(∇f, α, zeros(N) ; Niter=9000)
@assert xgd ≈ xh # equivalent within precision of type
```

- ▶ `a ≈ b` equivalent to `isapprox(a,b)`

$$\hat{\mathbf{x}} = \arg \min_{\mathbf{x}} \Psi(\mathbf{x}), \quad \Psi(\mathbf{x}) \triangleq \frac{1}{2} \|\mathbf{Ax} - \mathbf{y}\|_2^2 + R(\mathbf{x})$$
$$\implies \nabla \Psi(\mathbf{x}) = \mathbf{A}'(\mathbf{Ax} - \mathbf{y}) + \nabla R(\mathbf{x})$$

```
include("gd.jl") # provides gd() function
function rls(y, A, x0, alpha::Real, gradR::Function; Niter::Int = 50)
    gradPsi(x) = A'*(A*x - y) + gradR(x) # reg. LS cost gradient
    gd(gradPsi, alpha, x0; Niter) # returns final x
end
```

- ▶ Could add type annotations like `y::AbstractVector` and `A::AbstractArray` or `A::Matrix{Float32}`
- ▶ Using “duck typing” can facilitate code reuse
- ▶ JULIA \geq 1.5 required to avoid `Niter = Niter`

$$R(\mathbf{x}) = \beta \frac{1}{2} \|\mathbf{x}\|_2^2 \implies \hat{\mathbf{x}} = \arg \min_{\mathbf{x}} \Psi(\mathbf{x}) = (\mathbf{A}'\mathbf{A} + \beta \mathbf{I})^{-1} \mathbf{A}'\mathbf{y}$$

```

using LinearAlgebra: I, opnorm # same as svdvals(A)[1]
include("rls.jl")
M,N = 6,4
A = randn(M,N); y = randn(M); # test data
β = 2; ∇R(x) = β*x # Tikhonov regularizer gradient
xh = (A'A + β*I) \ (A'*y) # global minimizer of Ψ
α = 1/(opnorm(A)^2 + β) # step size
xr = rls(y, A, zeros(N), α, ∇R ; Niter=200)
@assert xr ≈ xh # equivalent within precision of type

```

- ▶ Develop / debug code with \mathbf{A} as a (small, dense) matrix
- ▶ For most computational imaging, \mathbf{A} too big to store explicitly
- ▶ Want to avoid rewriting new versions of GD and RLS for every \mathbf{A}
- ▶ Apply code with more general linear operator \mathbf{A} that can do $\mathbf{A}\mathbf{x}$ and $\mathbf{A}'\mathbf{y}$
- ▶ Minimalist version of “operator overloading:”

```
struct LinOp # see LinearMaps.jl and LinearMapsAA.jl for pro version
    forward::Function #  $\mathbf{A} * \mathbf{x}$  will call forward(x)
    adjoint::Function #  $\mathbf{A}' * \mathbf{y}$  will call adjoint(y)
end
```

```
Base.:(*)(A::LinOp, x) = A.forward(x) # for  $\mathbf{A}\mathbf{x}$ 
Base.adjoint(A::LinOp) = LinOp(A.adjoint, A.forward) # for  $\mathbf{A}'$ 
```

- ▶ Full-scale version: <https://github.com/JeffFessler/LinearMapsAA.jl>


```
struct LinOp # see LinearMaps.jl and LinearMapsAA.jl for pro version
  forward::Function # A * x will call forward(x)
  adjoint::Function # A' * y will call adjoint(y)
end

Base.:(*) (A::LinOp, x) = A.forward(x) # for A*x
Base.adjoint(A::LinOp) = LinOp(A.adjoint, A.forward) # for A'
```

Works because of JULIA's **multiple dispatch** feature

- ▶ Infix `x * y` is just syntax for `*(x,y)`
- ▶ By `length(methods(*))`, there are over 360 instances of `*` in JULIA 1.5, for different combinations of argument types
- ▶ `A'` is just syntax for `adjoint(A)` that has about 40 pre-defined methods.
- ▶ Above code “overloads” `adjoint` for new argument type `LinOp`, enabling `A'`. No need for syntax like `A.adjoint()`, enabling reuse of matrix-like code!

- ▶ Many computational imaging system models involve products of linear operators
 - super-resolution forward model has blur and down-sampling
 - compressed sensing MRI has coil sensitivity, Fourier transform, sampling
- ▶ Consider $\mathbf{A} = \mathbf{BC}$ for which $\mathbf{Ax} = \mathbf{B}(\mathbf{Cx})$ and $\mathbf{A}'\mathbf{y} = \mathbf{C}'(\mathbf{B}'\mathbf{y})$

```
Base.:(*) (A::LinOp, x) = A.forward(x) # for A*x
Base.:(*) (B::LinOp, C::LinOp) = # for B*C (composition)
    LinOp(B.forward ∘ C.forward, C.adjoint ∘ B.adjoint)
```

- ▶ Multiple dispatch depends on *all* argument types (not just first argument)
- ▶ JULIA compiler chooses most specific version of function
- ▶ The ◦ above is function composition.

2D finite difference regularizer for CI with periodic boundary conditions in sum form:

$$\begin{aligned}
 R(\mathbf{x}) &= \beta \sum_{n=1}^N \sum_{m=1}^M \phi(x[n, m] - x[n - 1 \bmod N, m]) + \phi(x[n, m] - x[n, m - 1 \bmod M]) \\
 &= \sum_{k=1}^{MN} \phi([\mathbf{T}_1 \mathbf{x}]_k) + \phi([\mathbf{T}_2 \mathbf{x}]_k) = \sum_{k=1}^{2MN} \phi([\mathbf{T} \mathbf{x}]_k), \quad \mathbf{T} \triangleq \begin{bmatrix} \mathbf{T}_1 \\ \mathbf{T}_2 \end{bmatrix} \\
 &= \mathbf{1}' \phi.(\mathbf{T} \mathbf{x}),
 \end{aligned}$$

where $\phi.(\mathbf{v})$ denotes applying potential function ϕ element-wise to \mathbf{v} : $\begin{bmatrix} \phi(v_1) \\ \phi(v_2) \\ \vdots \end{bmatrix}$.

If $\phi(t) = |t|$ then $\mathbf{1}' \phi.(\mathbf{T} \mathbf{x}) = \|\mathbf{T} \mathbf{x}\|_1$, aka anisotropic TV.

```
include("linop.jl")

diff2d_forw = x -> cat(x - circshift(x, (0,1)), x - circshift(x, (1,0)), dims=3)
diff2d_back = y -> (y[:, :, 1] - circshift(y[:, :, 1], (0,-1))) +
                   (y[:, :, 2] - circshift(y[:, :, 2], (-1,0)))
T = LinOp(diff2d_forw, diff2d_back)

M,N = 6,8 # testing
x = randn(M,N)
y = randn(M,N,2)
@assert sum(y .* (T * x)) ≈ sum(x .* (T' * y)) # check adjoint <T*x,y> ≈ <x,T'y>
```

- ▶ In math: $T_1 = I - P_1 \implies T_1 x = x - P_1 x$,
where P_1 denotes the circular shift along first index
- ▶ Here the resulting linear operator `T` maps a 2D array into a 3D array.
- ▶ `T'` is the *adjoint* (not transpose) of that operator.
- ▶ `.*` does element-wise multiplication

Gradient of regularizer uses derivative $\dot{\phi}$ of potential function element-wise:

$$R(\mathbf{x}) = \sum_k \phi([\mathbf{T}\mathbf{x}]_k) = \mathbf{1}'\phi.(\mathbf{T}\mathbf{x}) \implies \nabla R(\mathbf{x}) = \mathbf{T}'\dot{\phi}.(\mathbf{T}\mathbf{x}).$$

Example: Fair potential function $\dot{\phi}(z) = \beta \frac{z}{1 + |z|/\delta}$

```

δ = 0.1 # parameter for Fair potential function
β = 5 # regularization parameter
dφ(z) = β * z / (1 + abs(z)/δ) # derivative of potential function
∇R(x) = T' * dφ.(T * x) # regularizer gradient
  
```

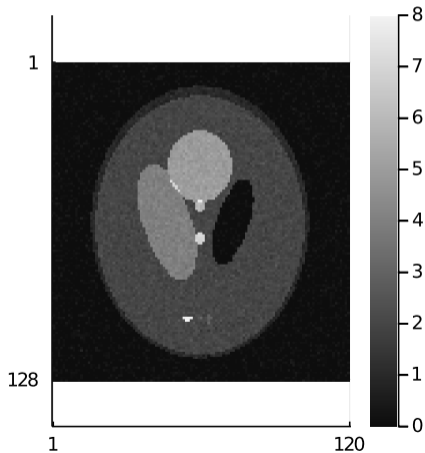
- ▶ `f.(v)` applies `f` element-wise in JULIA
- ▶ JULIA matches math or vice-versa?

$\mathbf{y} = \mathbf{x} + \varepsilon$ so $\mathbf{A} = \mathbf{I}$. Use $\mathbf{x}_0 = \mathbf{y}$

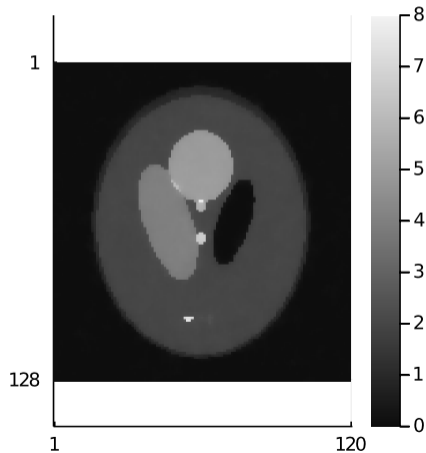
```
include("rls.jl"); include("diff2d.jl") # makes T
using LinearAlgebra: I
using MIRT: jim, ellipse_im
using Random: seed!
using Plots: plot
M,N = 120,128; xtrue = ellipse_im(M,N) # shepp-logan digital phantom
σ = 0.2; seed!(0); y = xtrue + σ * randn(size(xtrue)) # noisy data

δ = 0.1 # parameter for Fair potential function
β = 5; dφ(z) = β * z / (1 + abs(z)/δ) # derivative of potential function
∇R(x) = T' * dφ.(T * x) # regularizer gradient
α = 1 / (1 + 8β) # step size = 1 / Lipschitz constant
xh = rls(y, I, y, α, ∇R ; Niter=60)
plot(jim(y, title="Noisy", clim=[0,8]), # jiffy image display
     jim(xh, title="Denoised", clim=[0,8])) # savefig("denoise-y-xh.pdf")
```

Noisy



Denoised



Simple 2×-downsampling model:

$y[m, n] = x[2m, 2n] + \varepsilon[m, n]$ corresponds to $\mathbf{y} = \mathbf{A}\mathbf{x} + \varepsilon$ for $\mathbf{A} : \mathbb{R}^{2M \times 2N} \mapsto \mathbb{R}^{M \times N}$

```
include("linop.jl") # defines LinOp
down2_forw = (x) -> x[1:2:end, 1:2:end] # down-sampling in 1 line
function down2_adjoint(y) # up-sampling (adjoint) takes 2 lines:
    x = zeros(eltype(y), size(y).*2)
    x[1:2:end, 1:2:end] .= y # key step for up-sampling
    return x
end
A = LinOp(down2_forw, down2_adjoint)

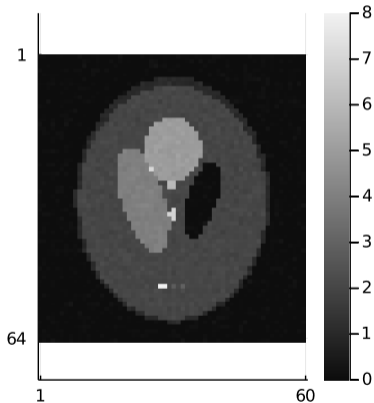
M, N = 60, 64; x = randn(2M, 2N); y = randn(M, N) # test data
@assert sum(y .* (A * x)) ≈ sum(x .* (A' * y)) # check adjoint
```

► `.=` does element-wise (in-place) assignment

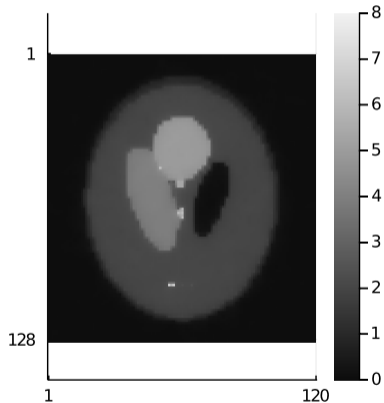

```
include("rls.jl"); include("diff2d.jl") # makes T
include("down2-linop.jl") # makes A
using MIRT: jim, ellipse_im
using Random: seed!
using Plots: plot, savefig
using ImageFiltering: imfilter
M,N = 120,128; xtrue = ellipse_im(M,N) # shepp-logan digital phantom
σ = 0.1; seed!(0); y = A*xtrue; y += σ * randn(size(y)) # noisy data

x0 = imfilter(A'*y, ones(2,2)) # initial guess
δ = 0.1 # parameter for Fair potential function
β = 2; dφ(z) = β * z / (1 + abs(z)/δ) # derivative of potential function
∇R(x) = T' * dφ.(T * x) # regularizer gradient
α = 1 / (1 + 8β) # 1 / Lipschitz constant (||A||2 = 1)
xh = rls(y, A, x0, α, ∇R; Niter=600) # same RLS as before!
plot(jim(y, title="Noisy Lo-Res data", clim=[0,8]),
     jim(xh, title="Recovered", clim=[0,8])) # savefig("superres-y-xh.pdf")
```

Noisy Lo-Res data



Recovered



- ▶ Many plotting back-ends including `pyplot`
 - `Interact` package provides GUI elements (sliders, buttons) in Jupyter
 - Time-to-first-plot needs improvement (better in JULIA 1.6)
- ▶ No `clear` function
 - `x = nothing` frees memory used by `x`
- ▶ JULIA default indexing is 1-based, not 0-based
 - `for x in list`
 - `for (i,x) in enumerate(list)`
 - Can create variables with other indexing conventions
 - `FFTVIEWS` package enables $[-N/2, \dots, N/2 - 1]$ indexing for DSP
 - Array comprehension largely eliminates need for `ndgrid`
 - `[function(j,k,l) for j=0:J-1, k=0:0.5:K-1, l=1:L]`
- ▶ **Encapsulation** can be circumvented
- ▶ Default is 1 thread (config file option to multithread)
- ▶ Obsolete Q/A on stackexchange etc.

- ▶ Detailed documentation at <https://docs.julialang.org>
Online tutorials, stackexchange, github conversations, wikibook:
https://en.wikibooks.org/wiki/Introducing_Julia
- ▶ `7:0.1:100000` stored as `Range` type, not as a huge array
- ▶ JULIA does not force you to indent any particular way
- ▶ Functions can return a tuple of variables

```
rotate = (x,y,θ) -> (x*cos(θ) + y*sin(θ), -x*sin(θ) + y*cos(θ)) # 3 in, 2 out  
xnew, ynew = rotate(x, y, π/4) # use both outputs
```

- ▶ Some of many useful packages for CI:
`JuliaImages`: image processing. `NFFT.jl`: Nonuniform FFT. `MRIReco.jl`

```

"""
This function applies `Niter` iterations of GD
to compute the minimizer ``\hat{x}`` of the cost function
``\Psi(x) = \frac{1}{2} \| A x - y \|_2^2 + R(x)``
given the gradient ``\nabla R``
"""
function rls(y, A, x, alpha::Real, gradR::Function ; Niter::Int = 50)
    gradPsi(x) = A'*(A*x - y) + gradR(x) # reg. LS cost gradient
    gd(gradPsi, alpha, x ; Niter) # returns updated x
end

```

In [8]: 1 ?rls

Out[8]: This function applies `Niter` iterations of GD to compute the minimizer \hat{x} of the cost function $\Psi(x) = \frac{1}{2} \|Ax - y\|_2^2 + R(x)$ given the gradient ∇R

Levels of Dispatch



dispatch degree	syntax	dispatch arguments	expressive order	expressive power
none	$f(x_1, x_2, \dots)$	$\{ \}$	$O(1)$	constant
single	$x_1.f(x_2, \dots)$	$\{ x_1 \}$	$O(X_1)$	linear
multiple	$f(x_1, x_2, \dots)$	$\{ x_1, x_2, \dots \}$	$O(X_1 X_2 \dots)$	exponential

From 2019 JuliaCon talk by Stefan Karpinski <https://youtu.be/kc9HwsxE10Y?t=679>

- Python integration
- @
- Debugger feedback

From 2019 JuliaCon talk

by Mike Innes

<https://youtu.be/0cUXjk7DFvU?t=1055>

```
49 py"""
50     import torch.nn.functional as F
51     def foo(W, b, x):
52         return F.sigmoid(W*x + b)
53     """ ✓
54
55 W = randn(2, 5) |> 2×5 Array{Float64,2}:
56 b = randn(2) |> Float64[2]
57 x = rand(5) |> Float64[5]
58
59 foo(W, b, x) |> Float64[2]
60
61
62
63 dW, db = gradient(W, b) do W, b
64     sum((foo(W, b, x) .- [0, 1]).^2)
65 end
```

- ▶ **Oct. 2019 SIAM News article about differentiable programming in JULIA**
 - ▶ Forward-mode automatic differentiation [2] with `ForwardDiff.jl`
 - ▶ Reverse-mode automatic differentiation [3] with `Zygote.jl`
- ▶ Compilation to cloud TPUs [4]
- ▶ Optimization package(s) [5]
- ▶ GPU integration [6]
- ▶ Intro to machine learning with JULIA
<https://tinyurl.com/ml2-18-jf>
- ▶ Help wanted!
<https://github.com/JeffFessler/MIRT.jl/blob/master/doc/matlab-to-julia.md>

Acknowledgment: Beamer slides made with modified `j1code.sty` from
<https://github.com/wg030/j1code>

Longer talk and code available online at

<https://www.ima.umn.edu/2019-2020/SW10.14-18.19/28302>

<http://web.eecs.umich.edu/~fessler/papers/files/talk/19/ima-julia>



- [1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. “Julia: A fresh approach to numerical computing.” In: *SIAM Review* 59.1 (2017), 65–98.
- [2] J. Revels, M. Lubin, and T. Papamarkou. *Forward-mode automatic differentiation in Julia*. 2016.
- [3] M. Innes. *Don’t unroll adjoint: Differentiating SSA-form programs*. 2019.
- [4] K. Fischer and E. Saba. *Automatic full compilation of Julia programs and ML models to cloud TPUs*. 2018.
- [5] P. K. Mogensen and Asbjørn Nilsen Riseth. “Optim: A mathematical optimization package for Julia.” In: *J. of Open Source Software* 3.24 (2018), p. 615.
- [6] T. Besard, C. Foket, and B. De Sutter. “Effective extensible programming: unleashing Julia on GPUs.” In: *IEEE Trans. Parallel. Dist. Sys.* 30.4 (Apr. 2019), 827–41.