Jeffrey A. Fessler

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

IMSI Workshop on Computational Imaging

2024-09-04

- ▶ Performance of a compiled language
- ▶ Interactive development / prototyping
- ▶ Readability
  - ○ Syntax matching the mathematics of computational imaging
  - ○ Built-in operations with numerical arrays
  - ○ Namespace control
- ▶ Relevant libraries (FFT, linear algebra, statistics, autograd, ...)
- ▶ Easy use of distributed computing, GPU resources
- ▶ Library ecosystem that facilitates reproducibility
- ▶ Free / open source
- ▶ IDE / debugger
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

Wish list for computational imaging code

▶ Performance of a compiled language   *JIT via LLVM*

▶ Interactive development / prototyping

▶ Readability
  ○ Syntax matching the mathematics of computational imaging
  ○ Built-in operations with numerical arrays
  ○ Namespace control

▶ Relevant libraries (FFT, linear algebra, statistics, autograd, ...)

▶ Easy use of distributed computing, GPU resources

▶ Library ecosystem that facilitates reproducibility

▶ Free / open source

▶ IDE / debugger

▶ Code reuse (*e.g.*, "object oriented")

▶ Memory efficient

▶ Interoperable with other languages

- ▶ Performance of a compiled language   *JIT via LLVM*
- ▶ Interactive development / prototyping   *Dynamic typing / Jupyter notebooks*
- ▶ Readability
  - ○ Syntax matching the mathematics of computational imaging
  - ○ Built-in operations with numerical arrays
  - ○ Namespace control
- ▶ Relevant libraries (FFT, linear algebra, statistics, autograd, ...)
- ▶ Easy use of distributed computing, GPU resources
- ▶ Library ecosystem that facilitates reproducibility
- ▶ Free / open source
- ▶ IDE / debugger
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

- ▶ 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, …)
- ▶ Easy use of distributed computing, GPU resources
- ▶ Library ecosystem that facilitates reproducibility
- ▶ Free / open source
- ▶ IDE / debugger
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

- 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
- Library ecosystem that facilitates reproducibility
- Free / open source
- IDE / debugger
- Code reuse (*e.g.*, "object oriented")
- Memory efficient
- Interoperable with other languages

- ▶ 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
- ▶ Free / open source
- ▶ IDE / debugger
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

- 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
- IDE / debugger
- Code reuse (*e.g.*, "object oriented")
- Memory efficient
- Interoperable with other languages

- ▶ 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
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

- ▶ 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   *VSCode, ...*
- ▶ Code reuse (*e.g.*, "object oriented")
- ▶ Memory efficient
- ▶ Interoperable with other languages

- 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 *VSCode, ...*
- Code reuse (*e.g.*, "object oriented") *multiple dispatch, subtypes*
- Memory efficient
- Interoperable with other languages

- ▶ 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 — *VSCode, ...*
- ▶ Code reuse (*e.g.*, "object oriented") — *multiple dispatch, subtypes*
- ▶ Memory efficient — *Float16, Sparse of any type, call by ref.*
- ▶ Interoperable with other languages

Wish list for computational imaging code

- ▶ 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 *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, ... Addresses "2 (or 3?) language problem"

- 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 Julia, Matlab, python...

- Growing use elsewhere, e.g.,
  ENGR108 at Stanford (Intro to Matrix Methods)

Linear Algebra for Data Science, Machine Learning, and Signal Processing

JEFFREY A. FESSLER
RAJ RAO NADAKUDITI
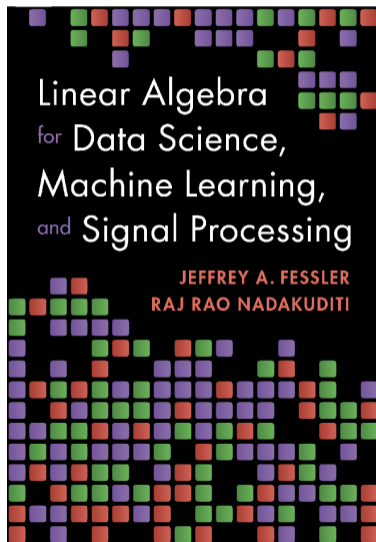
- All examples in Julia
- Online demos:
  https://github.com/JeffFessler/book-la-demo
- Topics include: low-rank matrix approximation, robust PCA, photometric stereo, matrix completion, ...
- Available in Europe/UK now; in US: July 2024 from Cambridge Univ. Press

Syntax comparison

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

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 $\boldsymbol{A}$: and $N \times N$ matrix $\boldsymbol{B}$:

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

▶ Kronecker sum function in JULIA:

```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 ⊗ with LATEXcodes and tab completion: `\otimes<tab>`

▶ JULIA supports argument chaining, *e.g.*, `size(A)[1]`

Trotier et al., "Accelerated 3D multi-echo spin-echo sequence with a subspace constrained reconstruction for whole mouse brain T2 mapping" MRM 2024 [2]

https://github.com/CRMSB/PAPER_subspace_MESE/blob/main/Project.toml

```
name = "Subspace_MESE"
uuid = "03ee4b65-1735-82da-5df6ad43f66c"
authors = ["aTrotier <a.trotier@gmail.com> and contributors"]
version = "1.0.2"

[deps]
LinearOperatorCollection = "a4a2c56f-fead-462a-a3ab-85921a5f2575"
BartIO = "8b90e6a2-6e4d-4906-89ae-b5f72ca4a65b"
EPGsim = "b6a82cc1-40c9-4006-90b7-9176002c0410"
FFTW = "7a1cc6ca-52ef-59f5-83cd-3a7055c09341"
LinearAlgebra = "37e2e46d-f89d-539d-b4ee-838fcccc9c8e"
LsqFit = "2fda8390-95c7-5789-9bda-21331edee243"
MRICoilSensitivities = "c57eb701-aafc-44a2-a53c-128049758959"
MRIFiles = "5a6f062f-bf45-497d-b654-ad17aae2a530"
MRIReco = "bdf86e05-2d2b-5731-a332-f3fe1f9e047f"

[compat]
julia = "1.9"
LinearOperatorCollection = "=1.1.2"
BartIO = "0.3.1"
EPGsim = "0.1.1"
FFTW = "1.7.2"
LsqFit = "0.15.0"
MRICoilSensitivities = "0.1.3"
MRIFiles = "0.2.0"
MRIReco = "0.8.1"
```

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

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

Gradient descent function code in Julia looks remarkably similar:

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

Enter symbols like α with LaTeX codes and tab completion: `\alpha<tab>`

```
function gd(∇f::Function, α::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

$$f(\boldsymbol{x}) = \tfrac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 \Longrightarrow \nabla f(\boldsymbol{x}) = \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}), \quad L_{\nabla f} = \|\boldsymbol{A}\|_2^2, \quad \alpha = 1/L_{\nabla f}$$

```julia
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)`

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

$$\implies \nabla\Psi(\boldsymbol{x}) = \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}) + \nabla R(\boldsymbol{x})$$

```julia
include("gd.jl") # provides gd() function
function rls(y, A, x₀, α::Real, ∇R::Function ; Niter::Int = 50)
    ∇Ψ(x) = A'*(A*x - y) + ∇R(x) # reg. LS cost gradient
    gd(∇Ψ, α, 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

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

```julia
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 **A** as a (small, dense) matrix
- For most computational imaging, **A** too big to store explicitly
- Want to avoid rewriting new versions of GD and RLS for every **A**
- Apply code with more general linear operator **A** that can do `A*x` and `A'*y`
- Minimalist version of "operator overloading:"

```julia
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'
```

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

```julia
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 230 instances of `*()` in JULIA 1.10, for different pairs 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 $A = BC$ for which $Ax = B(Cx)$ and $A'y = C'(B'y)$

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

$$R(\boldsymbol{x}) = \beta \sum_{n=1}^{N} \sum_{m=1}^{M} \phi(x[n,m] - x[n-1 \mod N, m]) + \phi(x[n,m] - x[n, m-1 \mod M])$$

$$= \sum_{k=1}^{MN} \phi([\boldsymbol{T}_1\boldsymbol{x}]_k) + \phi([\boldsymbol{T}_2\boldsymbol{x}]_k) = \sum_{k=1}^{2MN} \phi([\boldsymbol{T}\boldsymbol{x}]_k), \quad \boldsymbol{T} \triangleq \begin{bmatrix} \boldsymbol{T}_1 \\ \boldsymbol{T}_2 \end{bmatrix}$$

$$= \boldsymbol{1}'\phi.(\boldsymbol{T}\boldsymbol{x}),$$

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

If $\phi(t) = |t|$ then $\boldsymbol{1}'\phi.(\boldsymbol{T}\boldsymbol{x}) = \|\boldsymbol{T}\boldsymbol{x}\|_1$, 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:

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

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

```
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?

$y = x + \varepsilon$ so $A = I$. Use $x_0 = y$

```julia
include("rls.jl"); include("diff2d.jl") # makes T
using LinearAlgebra: I
using ImagePhantoms: shepp_logan, SheppLoganEmis
using MIRTjim: jim
using Random: seed!
using Plots: plot
M,N = 120,128; xtrue = shepp_logan(M,N,SheppLoganEmis()) # 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
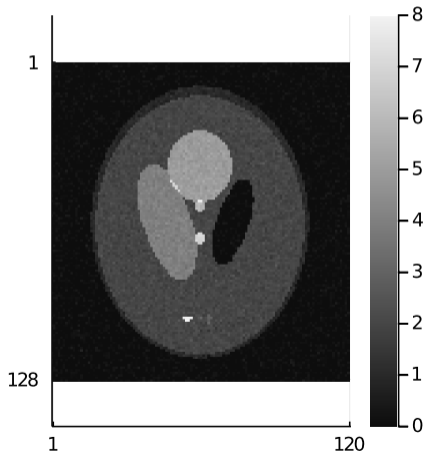
Denoised
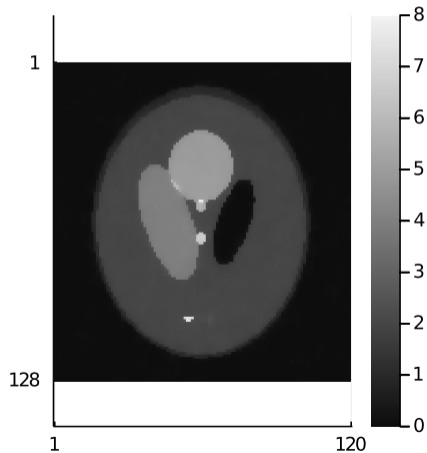
Simple 2×-downsampling model:

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

```julia
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

```julia
include("rls.jl"); include("diff2d.jl") # makes T
include("down2-linop.jl") # makes A
using ImagePhantoms: shepp_logan, SheppLoganEmis
using MIRTjim: jim
using Random: seed!
using Plots: plot, savefig
using ImageFiltering: imfilter
M,N = 120,128; xtrue = shepp_logan(M,N,SheppLoganEmis()) # 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")
```
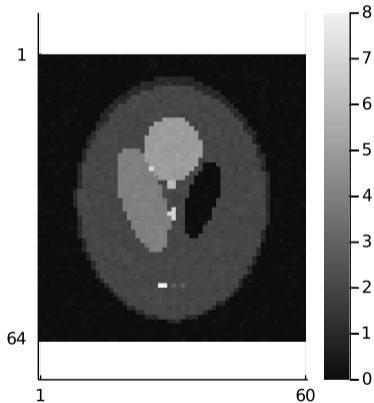
Noisy Lo-Res data   Recovered
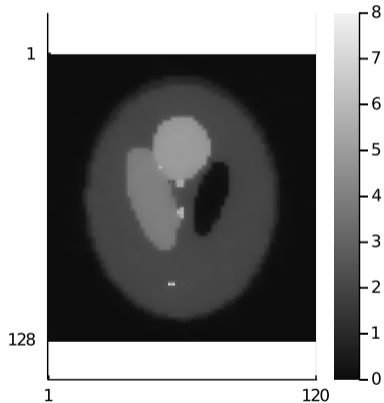
▶ GitHub organization: `https://github.com/JuliaImageRecon`
- SPECTrecon.jl
- Sinograms.jl
- ImagePhantoms.jl
- TrainingPhantoms.jl
- NeuralImagingNetworks.jl

▶ MRI `https://github.com/MagneticResonanceImaging`
- BlochSim.jl
- MRIFieldmaps.jl
- MRIReco.jl
- ...

- https://github.com/JuliaMath/FFTW.jl
- https://github.com/JuliaNLSolvers/Optim.jl
- https://github.com/FluxML
- https://github.com/JuliaDiff/ForwardDiff.jl

Ala carte style...

- ▶ Many plotting back-ends including `pyplot`
  - ○ `Interact` package provides GUI elements (sliders, buttons) in Jupyter
  - ○ Time-to-first-plot continually improving (much better in JULIA 1.10)
- ▶ 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 $[-N/2, \ldots, N/2 - 1]$ indexing for DSP
  - ○ Array comprehension largely eliminates need for `ndgrid`
    `[function(j,k,l) for j in 0:J-1, k in 0:0.5:K-1, l in 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
GIRFReco.jl [3]

```julia
"""
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, α::Real, ∇R::Function ; Niter::Int = 50)
    ∇Ψ(x) = A'*(A*x - y) + ∇R(x) # reg. LS cost gradient
    gd(∇Ψ, α, 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	$f(x_1, x_2, ...)$	$\{\}$	$O(1)$	constant
single	$x_1.f(x_2, ...)$	$\{x_1\}$	$O(|X_1|)$	linear
multiple	$f(x_1, x_2, ...)$	$\{x_1, x_2, ...\}$	$O(|X_1| |X_2| \cdots)$	exponential

From 2019 JuliaCon talk by Stefan Karpinski https://youtu.be/kc9HwsxE1OY?t=679

- Python integration
- @
- Debugger feedback

From 2019 JuliaCon talk
by Mike Innes
https://youtu.be/OcUXjk7DFvU?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 [4] with `ForwardDiff.jl`
  - Reverse-mode automatic differentiation [5] with `Zygote.jl`
- Compilation to cloud TPUs [6]
- Optimization package(s) [7]
- GPU integration [8]
- 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 `jlcode.sty` from
https://github.com/wg030/jlcode

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] A. J. Trotier, Nadege Corbin, S. Miraux, and E. J. Ribot. "Accelerated 3D multi-echo spin-echo sequence with a subspace constrained reconstruction for whole mouse brain T2 mapping." In: *Mag. Res. Med.* (2024).

[3] A. Jaffray, Z. Wu, S. J. Vannesjo, K. Uludağ, and L. Kasper. "GIRFReco.jl: An open-source pipeline for spiral magnetic resonance image (MRI) reconstruction in Julia." In: *J. Open Source Soft.* 9.97 (2024), p. 5877.

[4] J. Revels, M. Lubin, and T. Papamarkou. *Forward-mode automatic differentiation in Julia.* 2016.

[5] M. Innes. *Don't unroll adjoint: Differentiating SSA-form programs.* 2019.

[6] K. Fischer and E. Saba. *Automatic full compilation of Julia programs and ML models to cloud TPUs.* 2018.

[7] 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.

[8] 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.