@110 S3.4

‘compactsvd‘ Julia Assignment

September 25, 2020

As was presented in class, Julia by default returns the “economy” SVD. Write a function in Julia that finds the compact singular value decomposition of a matrix A. The function should input an $M \times N$ matrix and return Ur, Sr, and Vr. Additionally, return the singular values as a square matrix.

```
"""
z = compactsvd(A)
Find the SVD of matrix A in compact form.
In:
* ‘A‘ M * N input matrix
Out:
* ‘Ur‘ Left singular vectors
* ‘Sr‘ Singular value matrix
* ‘Vr‘ Right singular vectors
"""
function compactsvd(A)
```

1 Solution

The comment about V_0 and U_0 was actually a misdirection. Neither is needed to write the compact SVD.

```
using LinearAlgebra
"""
z = compactsvd(A)
Find the SVD of matrix A in compact form.
In:
* `A` M * N input matrix
Out:
* `Ur` Left singular vectors
* `Sr` Singular value matrix
* `Vr` Right singular vectors
"""
function compactsvd(A)
    U, S, V = svd(A)
    M, N = size(A)
    r = rank(Diagonal(S))
    Sr = Diagonal(S)[1:r,1:r]
    Vr = (V'[1:r, 1:N])'
    Ur = U[1:M, 1:r]
    return Ur, Sr, Vr
end
```

Sr = Diagonal(S[1:r])
is less typing
and more clear
and more efficient!

@130 S3.4

Question 10

An important class of bases are orthonormal bases, since they form orthogonal (or unitary in the complex case) matrices. We'd like to write a function that takes a set of basis vectors (by definition they span F^N and are linearly independent), and creates an orthonormal basis of F^N from them.

We can create an orthonormal basis for F^N by following the Gram-Schmidt process: Take i-th basis vector, subtract from it the projection of it onto all preceding vectors, then normalize it and insert it back into the matrix

We define projection as follows:

$$\text{projection}_{\mathbf{u}}(\mathbf{z}) = \frac{\langle \mathbf{z}, \mathbf{u} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \mathbf{u}$$

We can calculate orthonormal basis vectors $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n$ from linearly independent $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ by subtracting the projection of one vector onto all the others from it to obtain a set of orthogonal vectors $\mathbf{o}_1, \mathbf{o}_2, ..., \mathbf{o}_n$, then normalizing each of them for our orthonormal basis $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n$.

$\mathbf{o}_1 = \mathbf{v}_1$
$\mathbf{o}_2 = \mathbf{v}_2 - \text{projection}_{\mathbf{o}_1}(\mathbf{v}_2)$
$\mathbf{o}_3 = \mathbf{v}_3 - \text{projection}_{\mathbf{o}_1}(\mathbf{v}_3) - \text{projection}_{\mathbf{o}_2}(\mathbf{v}_3)$
$\mathbf{o}_n = \mathbf{v}_n - \text{projection}_{\mathbf{o}_1}(\mathbf{v}_n) - \text{projection}_{\mathbf{o}_2}(\mathbf{v}_n) - ... - \text{projection}_{\mathbf{o}_{n-1}}(\mathbf{v}_n)$

This is succinctly captured by the formula:
$\mathbf{o}_i = \mathbf{v}_i - \sum_{j=1}^{i} \text{projection}_{\mathbf{o}_j}(\mathbf{v}_i)$

And finally, we normalize each $\mathbf{o}_i$ to obtain our basis vectors $\mathbf{b}_i$: $\mathbf{b}_i = \frac{\mathbf{o}_i}{||\mathbf{o}_i||_2}$

The next page has the function specification and definition:

```
"""
z = orthogonalize(V)

Create an orthnormal basis from only the columns of V
using the Gram-Schmidt process.

In:
* `V` NxN matrix with N linearly independent columns

Out:
* `U` Orthonormal basis

"""
function orthogonalize(V)
    V = float(V)
    for i = 1:size(V, 1)
        v = V[:, i]

        # Subtract projection of v onto v1, v2, ..., v(i - 1)
        # from v
        for j = 1:(i-1)
            b = V[:, j]
            proj_v_onto_b = b * ((v'b) / (b'b))
            v = v - proj_v_onto_b
        end

        # Normalize v and insert back into matrix
        v = v ./ norm(v)
        V[:, i] = v
    end
    return V
end


function test_orthogonalize(n = 1000)
    N = 10
    for i = 1:n
        A = rand(N, N)
        Q = orthogonalize(A)
    @assert isapprox(Q'Q, I)
    end
end

test_orthogonalize()
```

This can be done in one line of Julia code:
orthogonalize(A) = svd(A).U

@120 S2.2

Write a `Julia` function called `randunitary` that generates a $N \times N$ random **unitary** matrix.

```
"""
A = randunitary(N::Integer)
Generate a `N` by `N` random unitary matrix
In:
* `N` Integer
Out:
* `A` Complex Float `N` by `N` unitary matrix
"""
function randunitary(N::Integer)
```

Since there could be some round-off errors, you can use the function `isapprox(X, I)` to check if `X` is (approximately) an identity matrix.

Answer to the problem.

```julia
using LinearAlgebra
"""
A = randunitary(N::Integer)
Generate a `N` by `N` random unitary matrix
In:
* `N` Integer
Out:
* `A` Complex Float `N` by `N` unitary matrix
"""
function randunitary(N::Integer)
    A, _, _ = svd(rand(Complex{Float64}, N, N))
    return A
end
```

```
julia> A = randunitary(3)
3×3 Array{Complex{Float64},2}:
 -0.650929-0.0663274im   0.468988+0.0599823im  -0.0408469-0.588793im
 -0.313672-0.513416im   -0.585349-0.493557im    -0.205538-0.0976559im
 -0.395961-0.230482im    0.409696-0.149566im   0.00440019+0.774502im

julia> isapprox(A'*A, I)
true

julia> isapprox(A*A', I)
true

julia> A = randunitary(100)
100×100 Array{Complex{Float64},2}:
  -0.073769-0.0660258im   -0.026016-0.0277763im     0.0606364+0.0067512
 -0.0769283-0.0703727im  -0.0483956+0.0850524im     -0.112517+0.021191
  -0.073954-0.0675295im    0.187887+0.145126im     0.00041416-0.037919
    -0.0697-0.0697607im  -0.0494489+0.0130366im     0.0253412+0.025010
 -0.0643041-0.0641836im  -0.0163741+0.0561623im    -0.0237619-0.100003i
 -0.0750802-0.0703347im   -0.150613-0.0711526im    -0.0808881+0.017511
 -0.0712906-0.0696031im    0.183624-0.040117im      0.0531721+0.004113
 -0.0756772-0.0671894im   0.0408709-0.0340298im     0.0280556+0.093376
 -0.0721377-0.0643879im   0.0423857-0.00306784im   -0.0314801+0.030410
 -0.0692687-0.0730429im   0.0339194-0.0566075im     0.0171433+0.078172
 -0.0662876-0.0718085im   0.0282223-0.0424238im    -0.0420541-0.048197
          ⋮
 -0.0718039-0.0686449im    0.028841-0.105411im      -0.122259-0.10019im
 -0.0760828-0.0665927im   0.0464448-0.0119247im    0.00433791-0.034453
 -0.0744635-0.0639733im   0.0743639-0.108859im      0.0546004+0.010888
 -0.0697974-0.0715672im  0.00710381-0.111782im       0.118384-0.043914
 -0.0713351-0.0688471im   0.0220282+0.0191463im    -0.0318558+0.190461i
  -0.068837-0.0659168im  -0.0234896+0.0773245im   -0.00480583+0.086426
 -0.0729038-0.0661915im  -0.0269647-0.08786im        0.185855-0.011395
 -0.0666454-0.0725413im  -0.0337594-0.112663im      0.0218241+0.078717
 -0.0795917-0.0696535im  -0.0509302+0.0130188im     0.0763106-0.085448
 -0.0710539-0.0655923im   -0.103061-0.00409098im    0.0741527-0.084425
 -0.0651527-0.0679528im   0.0309778-0.0607555im     0.0621636+0.036674

julia> isapprox(A'*A, I)
true

julia> isapprox(A*A', I)
true
```

@118 3.4

Write a program ortho(v...) which, given m orthogonal nonzero vectors v1,v2,...,vm
of length n (0 < m < n) and float type,
returns a vector of vectors [v1,v2,...,v(n-m)] which serve as an orthogonal basis
for span(v1,v2,...,vm)^⊥

You may not use LinearAlgebra: nullspace for this problem

Notes on multiple returns / arguments: you can use ortho(v...) to get v, a vector of
all the inputs to the function.
So if the user called ortho([0,1,0],[1, 0,0]), you would get a vector v =
[[0,1,0],[1, 0,0]]
(of type Vector{Vector{N}}, where n is something <:AbstractFloat)

To avoid floating point errors with student answers, I would recommend that all
input vectors have a difference from eachother
and from the zero vector of something greater than 1e-5 or so.

Solution:

```
#the basis for this answer is that {e_1,...,e_n} must all be generated by these
basis vectors and if they could all be generated
#by a basis smaller than n we would have a smaller basis (which is impossible)
#so given perp vectors of size < n we must be able to find an e_i s.t. e_i -
proj(e_i,all vectors) is nonzero and get another ortho from there
#probably an even easier way to do this tbh!
#note that the easiest solution is to look at nullspace's sourcecode...
using LinearAlgebra: dot,norm
function ortho(v::Vector{<:AbstractFloat}...)
        n = length(v[1])
        basis = Vector{typeof(v[1])}(undef,n)
        for i in 1:length(v)
                basis[i] = v[i]
        end
        for i in length(v)+1:n
                basis[i] = zeros(n)
        end
        for curr in length(v)+1:n
                for i in 1:n
                        ei = zeros(typeof(v[1][1]),n)
                        ei[i] = 1
                        #ensure that this protovector is orthogonal to all prior
basis vectors
                        for j in 1:curr-1
                                ei .-= basis[j] .* dot(ei,basis[j]) ./
norm(basis[j])^2
                        end
                        if norm(ei) > 1e-10
                                basis[curr] = ei
                                break
                        end
                end
```

There is an easier way (less typing, possibly faster) using (full) SVD. Something like this that returns U_0: svd(hcat(v)).U[:,(1+length(v)):end]

```
        end
        return basis[length(v)+1:end]
end
```

@248 S2.4

In this problem, we're going to try and visualize the difference between a unitary eigendecomposition and SVD of a real square matrix.

Download the 42x42 image given to you and load into a variable **image** and perform:

a) An eigendecomposition of the matrix using **eigen(image)**
b) Compute the SVD of the matrix using **svd(image)**

Use the first 21 leading eigenvalues from step (a) to recreate the image.
Additionally, use the first 21 leading singular values obtained from step (b) to recreate another image.

Plot both side by side and compare them to the original image. Does one have better quality than the other?

Hint: You may need to use the package ImageIO, Plots, Images, ImageView

Solution:

```
using ImageIO, Plots, Images, ImageView

image = load("2x2square.png")
evals, evecs = eigen(image) #already sorted in Julia
U, S, Vt = svd(image)

SVD_IMG = U[:, 1:21] * diagm(S[1:21]) * Vt[:, 1:21]'
evecs2 = reverse(evecs, dims=2)
evals2 = reverse(evals)
EIG_IMG=evecs2[:,1:21] * diagm(evals2[1:21]) * evecs2[:, 1:21]'

gui = imshow_gui((100, 100), (2, 1))
canvases = gui["canvas"]
imshow(canvases[1,1], SVD_IMG)
imshow(canvases[1,2], EIG_IMG)
Gtk.showall(gui["window"])
```

1、Problem description

Write a function called basiscor that gets the coordinates of a matrix A($A \in R^{2\times2}$) in a given four basis vectors of space $R^{2\times2}$ and the coordinates is included in an output vector x

2、The function specification

"""

z = basiscor(A,B1,B2,B3,B4)

gets the coordinates of a matrix A in a given four basis vectors of Space $\mathrm{R}^{2\times2}$

In:

* `A is a matrix which needs to be operated

* B1,B2,B3,B4 are the basis vectors ,which Bi$\in \mathrm{R}^{2\times2}$ of space $\mathrm{R}^{2\times2}$

Out:

* `vector x consists of 4 coordinates of matrix A corresponding to the four basis vectors B1,B2,B3,B4 respectively.

For full credit, your solution should be computationally efficient.

"""

function basiscor(A,B1,B2,B3,B4)

3、My Julia code answer to the problem:

```
function basiscor(A,B1,B2,B3,B4)
    W=[vec(B1) vec(B2) vec(B3) vec(B4)]
    x=inv(W)*vec(A)
    return x
end
```

@118 S4.3

Problem.
S4.4: Given M by N matrix A, M length vector b, P by N matrix C, and P length vector d, consider the set of minimizing solutions $S1 = argmin_{x \in F^N} ||Ax - b||_2$. Write a julia function, minmin, which returns any length N vector y that lies in the set of minimizing solutions $S2 = argmin_{x \in S1} ||Cx - d||_2$. Note that for some returned value y, $||Ay - b||_2$ is always an absolute minimum of f(x) = $||Ax - b||_2$, but $||Cy - d||_2$ does not necessarily minimize g(x) = $||Cx - d||_2$ because the second problem is constrained.

A function header is given below.
"""
minmin(A,b,C,d) returns a vector y which minimizes $||Cy - d||_2$ given the constraint that y minimizes the function $f(x) = ||Ax - b||_2$

input:
A: M by N matrix.
b: M length vector. This pair describes the "higher priority" linear optimization problem.
C: P by N matrix.
d: P length vector. This pair describes the "lower priority" linear optimization problem.

output:
y: N length vector minimizing $||Cy - d||_2$ given the constraint that y minimizes the function $f(x) = ||Ax - b||_2$.
"""
function minmin(A::Matrix,b::Vector,C::Matrix,d::Vector)

See next page for answer.

Solution.

```
# you can use the backslash command if you want, but it will output Inf for zero-rank A or C*Vo, so be careful!
function minmin(A::Matrix,b::Vector,C::Matrix,d::Vector)
U,s,V = svd(A)
r = rank(A)
Ur = U[:,1:r]
sr = s[1:r]
Vr = V[:,1:r]
if r == 0
base_solution = zeros(size(A,2))
else
base_solution = Vr * diagm(1 ./ sr) * Ur' * b
end
if r == size(A,2) #full column rank
#base solution only solution
return base_solution
end
Vo = V[:,r+1:end]
#we can only mess around by adding the columns of Vo to x.
#so we're trying to find argmin_z(||D * (Vo*z+base_solution) - c||_2)
#which equals argmin_z(||D *Vo*z - (c - D * base_solution)||_2)
#so we can just do the same minimization again across D * Vo
U2,s2,V2 = svd(C * Vo)
r2 = rank(C * Vo)
Ur2 = U2[:,1:r2]
sr2 = s2[1:r2]
Vr2 = V2[:,1:r2]
#just return base if the matrix has no rank; nothing matters
r2 == 0 && return base_solution + zeros(size(A,2))
return base_solution + Vo * Vr2 * diagm(1 ./ sr2) * Ur2' * (d - C * base_solution)
end
```

@124 S1.6 / S7.4

Given $\boldsymbol{A} \in \boldsymbol{F}^{n\times n}$, and an initial vector $\boldsymbol{x_0} \in \boldsymbol{F}^{n}$ the power method can be used recursively to approximate the dominant eigen vector $\boldsymbol{v_1} \in \boldsymbol{F}^{n}$ associated with the largest (in magnitude) eigenvalue provided that $\boldsymbol{v_1'x_0} \neq \boldsymbol{0}$ and the largest eigenvalue(in magnitude) is unique. The sequence below then converges to $\boldsymbol{v_1}$.

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}$$

a. Write a function in Julia called **`eigvecmax`** that computes the eigen vector associated with the largest (in magnitude) eigen value of a matrix using the power method. The inputs to the function are the matrix $\boldsymbol{A} \in \boldsymbol{F}^{n\times n}$, an initial vector $\boldsymbol{x_0} \in \boldsymbol{F}^{n}$ and the number of iterations **`nIters`**. You can assume that the matrix is Hermitian and has distinct (in magnitude) eigen values and it will converge after 1000 iterations of the power method.

```
"""
x = eigvecmax(A; x0 = ones(size(A,2)), nIters = 100)
Use the power method to approximate the eigen vector associated
with the largest(in magnitude) eigen value of A
inputs:
- `A` `n × n` matrix
Option:
- `x0` initial starting vector (of length `n`) to use; default 1 vector
- `nIters` number of iterations to perform; default 100
outputs:
`x` vector of length `n` containing the approximate solution
"""
function eigvecmax(A::AbstractMatrix{<:Number};
                   x0::AbstractVector{<:Number} = ones(eltype(A), size(A,2)),
                   nIters::Int = 1000)
```

b. Write a function in Julia called **`eigvalmin`** that uses the power method with the function written in **a.** i.e **`eigvecmax`** to compute the minimum eigen value of a Hermitian matrix with distinct (in magnitude) eigen values. Here the minimum eigen value is the smallest in value. In Julia your file should be named **`eigenvalmin.jl`** and should contain both function definitions.

```
"""
lambda = eigvalmin(A)
Use the power method to approximate the minimum eigen value of A

inputs:
- `A` `n × n` matrix

outputs:
`lambda` the minimum eigen value of A
"""
function eigvalmin(A::AbstractMatrix{<:Number})
```

Solution

```
using LinearAlgebra:norm, I
"""
x = eigvecmax(A; x0 = ones(size(A,2)), nIters = 100)
Use the power method to approximate the eigen vector associated
with the largest(in magnitude) eigen value of A

inputs:
- `A` `n × n` matrix
Option:
- `x0` initial starting vector (of length `n`) to use; default 1 vector
- `nIters` number of iterations to perform; default 100
outputs:
`x` vector of length `n` containing the approximate solution
"""
function eigvecmax(A::AbstractMatrix{<:Number};
                x0::AbstractVector{<:Number} = ones(eltype(A), size(A,2)),
                nIters::Int = 1000)
    x = x0
    for i = 1:nIters
        x = (A*x)/norm(A*x)
    end
    return x
end
"""
lambda = eigvalmin(A)
Use the power method to approximate the minimum eigen value of A
inputs:
- `A` `n × n` matrix
outputs:
`lambda` the minimum eigen value of A
"""
function eigvalmin(A::AbstractMatrix{<:Number})
    x = eigvecmax(A)
    lambda = (x' * A * x)/(x'*x)
    if lambda < 0
        return lambda
    end
    B = A - lambda* I(size(A,1))
    x = eigvecmax(B)
    lambda = (x' * A * x)/(x'*x)
    return lambda
end
```

@130 S4.1 / S8.1

Exam 2 Practice Problem

We've created an iterative, gradient-descent approach for least squares with traditional vectors, but we can also use a similar approach for matrix least squares, where an MxN matrix, B is a vector in $\mathbb{F}^{M\text{x}N}$.

The cost function to minimize is:

$$\hat{\mathbf{x}} = \underset{\mathbf{x}=(x_1,\ldots,x_K)\in\mathbb{F}^K}{\operatorname{argmin}} \left\| \left(\sum_{k=1}^{K} x_k \mathbf{A}_k \right) - \mathbf{B} \right\|_F$$

When $\mathbf{D}$, the matrix with columns $[\text{vec}(\mathbf{A}_1), \text{vec}(\mathbf{A}_2), ..., \text{vec}(\mathbf{A}_K)]$, has full column rank, then $\hat{\mathbf{x}}$ is the solution $\hat{\mathbf{x}} = \mathbf{D}^+\text{vec}(\mathbf{B})$. Otherwise, it is the solution with minimum norm.

The gradient descent iterative approach can be calculated by:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mu\mathbf{D}'(\mathbf{D}\mathbf{x}_i - \text{vec}(\mathbf{B}))$$

which converges to minimize the above cost function as i increases towards infinity. Note, that for convergence, our step size, μ, must lie between $0 < \mu < \frac{2}{\sigma_1^2}$ where σ_1 is the first singular value of $\mathbf{D}$.

```
using LinearAlgebra
"""
Matrix Least Squares Gradient Descent

x = mlsgd(A, B ; mu=0, x0=zeros(size(A, 1)), nIters::Int=200)

Performs gradient descent to solve the matrix least squares problem:
``\\argmin_x 0.5 \\| B  sum_{k=0}^K A_k x_k \\|_F``

In:
 `A` array of K `m × n` matrices: [A_1, A_2, ..., A_K]
 `B` `m x n` matrix

Option:
 `mu` step size to use, and must satisfy ``0 < mu < 2 / \\sigma_1(D)^2``
to guarantee convergence, where ``\\sigma_1(D)`` is the first (largest)
singular value of the `mn x K` matrix: [vec(A_1) vec(A_2) ... vec(A_K)]
 `x0` is the initial starting vector (of length `K`) to use.
Its default value is all zeros for simplicity.
 `nIters` is the number of iterations to perform (default 200)

Out:
 `x` vector of length `K` containing the approximate LS solution
"""
function mlsgd(A, B; mu::Real=0, x0=zeros(size(A, 1)), nIters::Int=200)
    D = hcat(vec.(A)...) # vec each A_k, then use `...` to feed them as args to
    ↪ hcat
    b = vec(B)

    x = x0
```

```
    if (mu == 0)
        mu = 1. / (opnorm(D, 1) * opnorm(D, Inf))
    end

    itr = 0
    for itr = 1:nIters
        x = x - mu .* (D' * (D*x - b))

        itr = itr + 1
    end

    return x
end

# Test mlsgd
M = 5; N = 4
for i = 1:100
    B = rand(M, N)
    K = 3
    A = [rand(M, N), rand(M, N), rand(M, N)]
    D = hcat(vec.(A)...)
    x = rand(K)
    b = D * x
    B = reshape(b, (M, N))

    x_gd = mlsgd(A, B, nIters=10000)
    @assert isapprox(x_gd, x)
end
```

@216 S4.1

Homography Transformation

Homography is a very important concept in computer vision. A homography matrix is a 3 by 3 matrix M which relates the transformation between two planes. Consider a point (x',y',1) in one plane and a point (x,y,1) in another plane:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The homography matrix is indeed a 3by3 matrix but with 8 degrees of freedom as it is estimated up to a scale. It is generally normalized with $h_{33} = 1$ or $h_{11}^2 + h_{12}^2 + h_{13}^2 + h_{21}^2 + h_{22}^2 + h_{23}^2 + h_{31}^2 + h_{32}^2 + h_{33}^2 = 1$. In this Julia question, the inputs are two 2byN matrixes, X and Y, corresponding to N points in two planes separately. The output should be a 3by3 homography matrix which transforms points in Y to points in X. (Hint: vec() function are useful here).

In Julia, your file should be named hm.jl and should contain the following function:

```
"""
`h = hm(X, Y)
Performs a homography matrix to project points in one plane(image) to another plane(image):
In:
- `X` `2 x N` matrix
- `Y` `2 x N` matrix
Out:
`H` 3 x 3 homography matrix which projects points in Y to points in X
"""
function hm(X, Y )
```

My solution code:

```
using LinearAlgebra
function hm(X, Y )
    N=size(X)[2]
    A=zeros(2*N,8)
    O=zeros(2*N)
    h=zeros(8)
    for i in 1:N
        A[2*i-1,:]=[Y[1,i] Y[2,i] 1 0 0 0 -Y[1,i]*X[1,i] -Y[2,i]*X[1,i]]
        A[2*i,:]=[0 0 0 Y[1,i] Y[2,i] 1 -Y[1,i]*X[2,i] -Y[2,i]*X[2,i]]
        O[2*i-1]=X[1,i]
        O[2*i]=X[2,i]
    end
    h=vcat(A\O,1)
    H=reshape(h,(3,3))
    return transpose(H)
end
```

@118 S6.4

S6.42: Write a program $shattn(Y, \beta, n)$ which returns

$$\underset{X \in F^{M \text{x} N}}{\text{argmin}} \; ||Y - X||_* + \beta(||X||_{S_n})^n$$

Here $||\cdot||_*$ is the nuclear norm, and $||\cdot||_{S_n}$ is the Schatten n-norm (where σ_i is the ith singular value of the operand)

$$(\sum_{i=1}^{min\{m,n\}} \sigma_i^n)^{\frac{1}{n}}$$

You can assume that n will always be a positive integer, and that β will always be nonnegative. A function header is given below.

"""

$shattn(Y, \beta, n)$ returns a matrix X which minimizes $||Y - X||_* + \beta(||X||_{S_n})^n$, where $||\cdot||_*$ is the nuclear norm and $||\cdot||_{S_n}$ is the Schatten n-norm.

input:
Y: Matrix used in the argmin.
β: Nonnegative scalar normalization factor for the Schatten norm.
n: The normalization operator uses a Schatten n-norm. Guaranteed to be a positive integer.

output:
X: Matrix with the same size as Y minimizing $||Y - X||_* + \beta(||X||_{S_n})^n$.

"""

function shattn(Y::AbstractMatrix,β::Number,n::Integer)

See next page for answer.

Solution.

```
using LinearAlgebra
"""
we know that because the norms are unitarily invariant we have the same U,V
so consider one of Y's sigmas, s = σ_i, and the corresponding x sigma, t = σ'_i.
we need to minimize Σ_{i=1}^{min(m,n)} |σ_i − σ'_i| + β|σ'_i|^n, so
we need to minimize |s − t| + β|t|^n
it's pretty clear that increasing t past s or decreasing it below 0
increases the cost function since β is nonnegative
so minimize s − t + βt^n for t between 0 and s;
if n=1, we want either s if β < 1 or 0 otherwise
if β = 0, t=s clearly optimizes
Otherwise, we have βnt^{n−1} − 1 = 0.
so we have t = (1/(nβ))^{1/(n−1)}, which is a local min by 2nd derivative test.
remember that if the above t is greater than s, we have to use s instead
since then the function is dec from 0 to s, when it increases due to the piecewise transition.
"""
function schattn(Y::AbstractMatrix,β::Number,n::Integer)
(β == 0 || (n == 1 && β < 1)) && return Y
n == 1 && β ≥ 1 && return zeros(size(Y))
U,Σ,V = svd(Y)
r = rank(Diagonal(Σ))
Σp = zeros(r)
for i in 1:r
Σp[i] = min(Σ[i],(1/(nβ))^{1/(n−1)})
end
return U[:,1:r] * Diagonal(Σp) * V[:,1:r]'
end
```

@130 S6.1

Exam 3 Practice Problem

We've studied a Least-Squares fit of linear data of the form:

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2$$

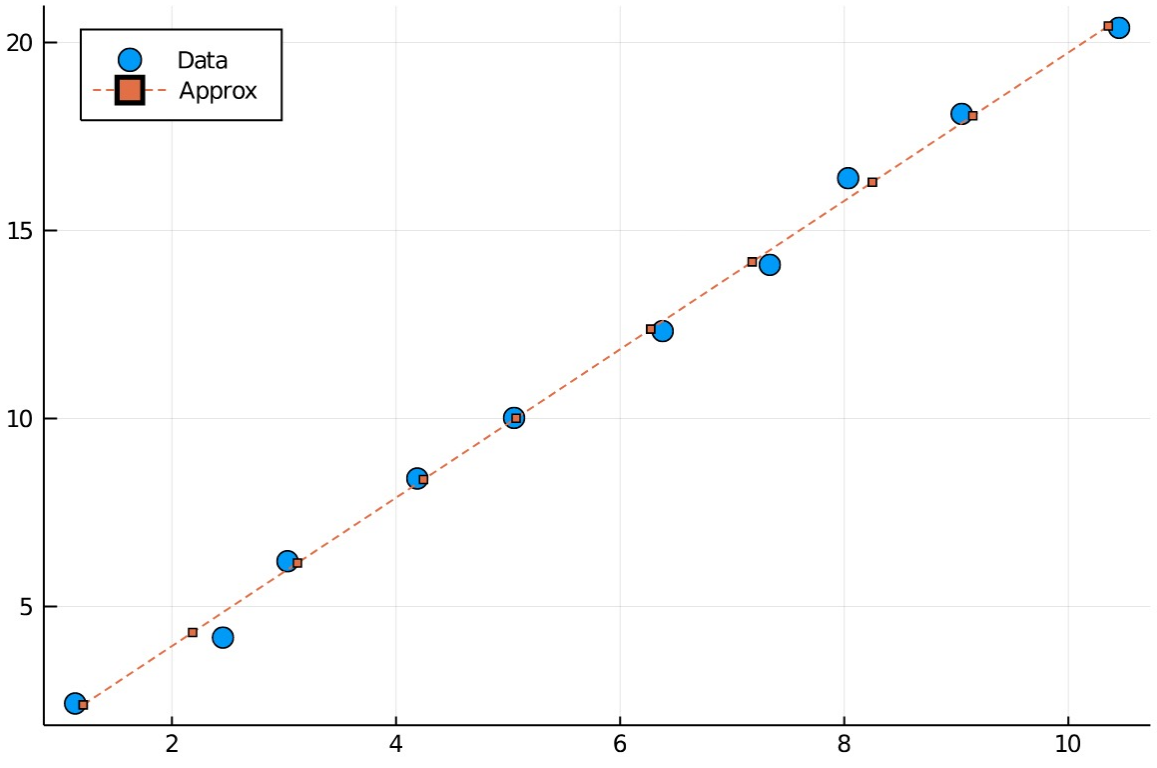
In this case, $\mathbf{A}$ is simply a scalar α. Alternatively, we could look at the data matrix $\mathbf{A} = [\mathbf{x}\mathbf{y}]^{\mathbf{T}}$ and find its rank-1 approximation to fit a different line to the data. Here, we're minimizing

$$\hat{\mathbf{B}} = \underset{\mathbf{B}:\ \mathrm{rank}(\mathbf{B})=\mathbf{1}}{\operatorname{argmin}} \|\mathbf{X} - \mathbf{B}\|_F$$

Implement the function `r1a` which projects the training data, X, onto the line fit from the rank-1 approximation of the data, as well as returns a basis vector for that line.

```
using LinearAlgebra

"""
Rank-1 Approximation of Nx2 Data Matrix

    Xh, v = r1a(X)

Use a rank-1 approximation to fit a line to 2D data, solving

    ``\\argmin_B 0.5 \\| X - B \\|_F``

In:
 `X` `n x 2` data matrix

Out:
 `Xh` `n x 2` rank-1 approximation of X
- `v` `2` dimensional basis vector
"""
function r1a(X)
    U,s,V = svd(X)
    Xh = U[:, 1] * s[1] * V[:, 1]'
    return Xh, U[:, 1]
end


using Plots
# Test r1a
M = 2; N = 10

x = [1; 2]
X = [x.*j .+ 0.5 .* rand(2, 1) for j = 1:N]
X = hcat(X...)

Xh, v = r1a(X)
U,s,V = svd(X)
@assert isapprox(v, U[:, 1])
@assert isapprox(v*(v'*X), Xh)
```

```
scatter(X[1,:], X[2,:], label="Data", markersize=6)
plot!(Xh[1,:], Xh[2,:], label="Approx", linestyle=:dash, markershape=:auto,
↪  markersize=2, legend=:topleft)
```



@216 S6.4

(Reference:6.4) In this Julia question, we just consider a kind of unconstrained cost function that penalizes high rank. The cost function's convex relaxation is the nuclear norm:

$$\hat{X} = \underset{X \in F^{MN}}{\operatorname{argmin}} \frac{1}{2}\|Y - X\|_F^2 + \beta * \|X\|_*$$

Writer a function called minimum_nuclear that takes as input an $m \times n$ matrix Y and a scalar p and return a scalar e and a $m \times n$ matrix X. X is the optimal estimator in the above formula and scalar e is the minimum value of this cost function when we input the optimal estimator X.

In Julia, your file should be named minimum_nuclear.jl and should contain the following function:

```
"""
e, X = compress_image(Y, p)
In:
* `Y` `m × n` matrix
* `p` scalar
Out:
* `X` a `m × n` matrix that is the optimal estimator to the low-rank approximation formulations with the penalization of nuclear norm.
* `e` the minimal approximation error we could achieved with the nuclear norm
"""
function minimum_nuclear(Y, p)
```

My Julia Solution:

```
using LinearAlgebra
function minimum_nuclear(Y, p)
    U,E,V=svd(Y)
    r=rank(Diagonal(E))
    E_hat=max.(E.-p,0)
    X=U*Diagonal(E_hat)*V'
    e=0.5*sum((E-E_hat).^2)+p*sum(E_hat)
    return X,e
end
```

@248 S6.5

Problem:

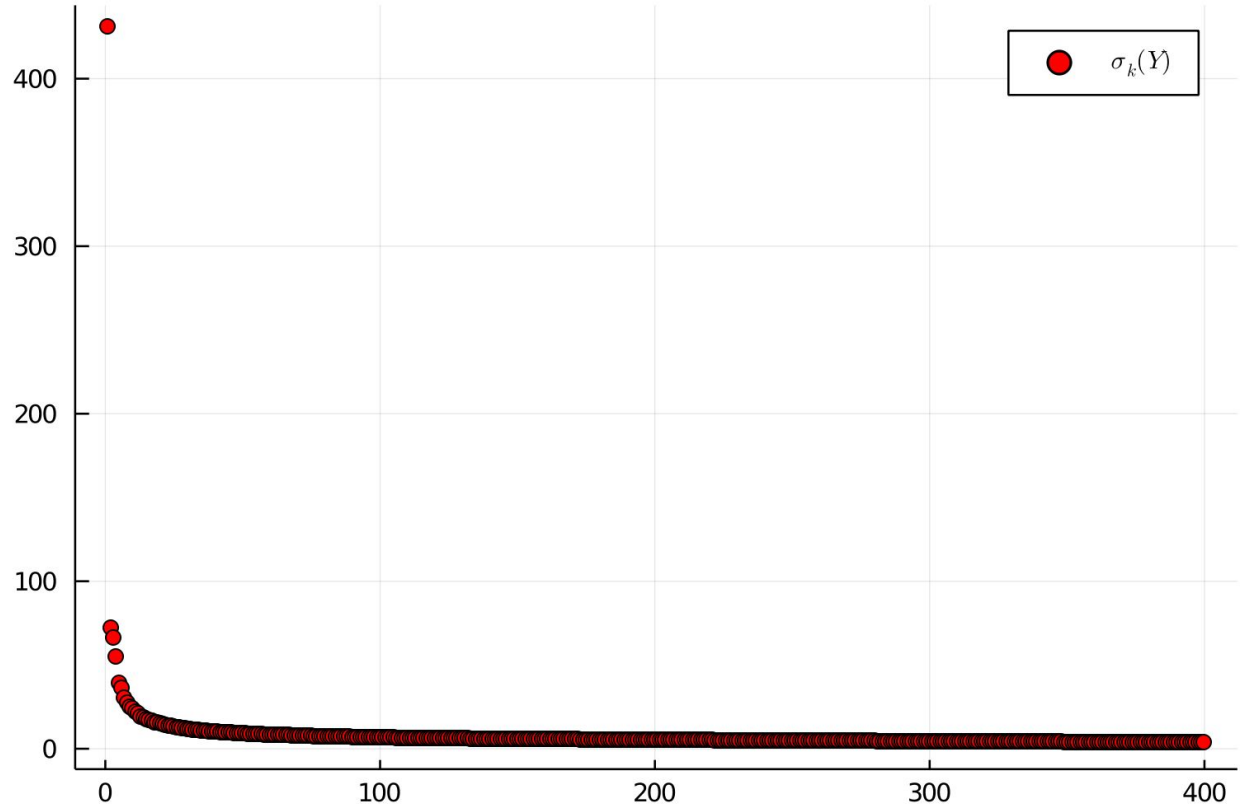
I clicked this image of a lion at a zoo, but it seems the file got corrupted! Denoise the image using any technique learned in 551.
(Optional: Compare this technique to a low-rank approximation technique in terms of image quality.)

Solution:

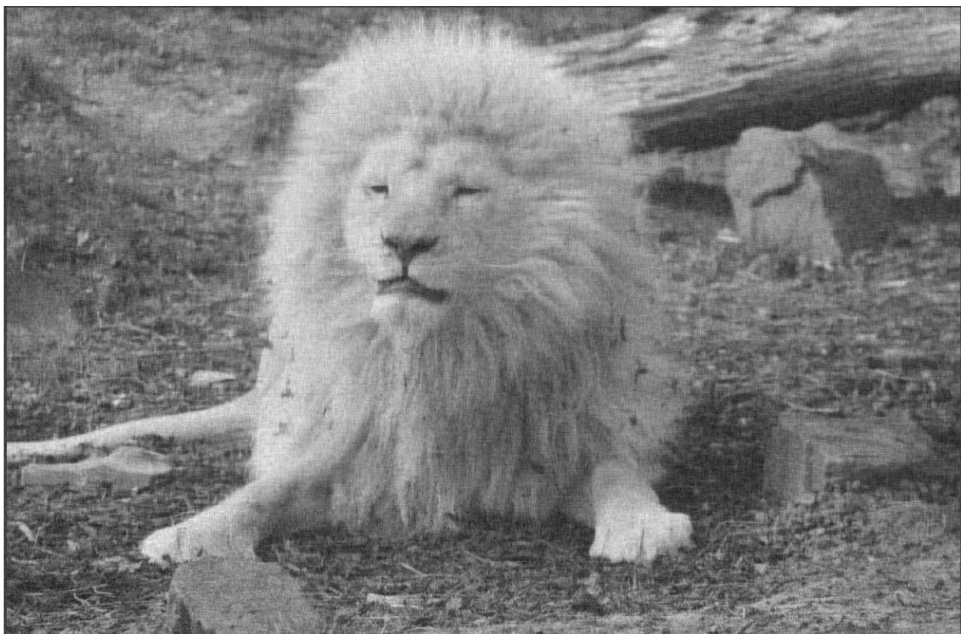
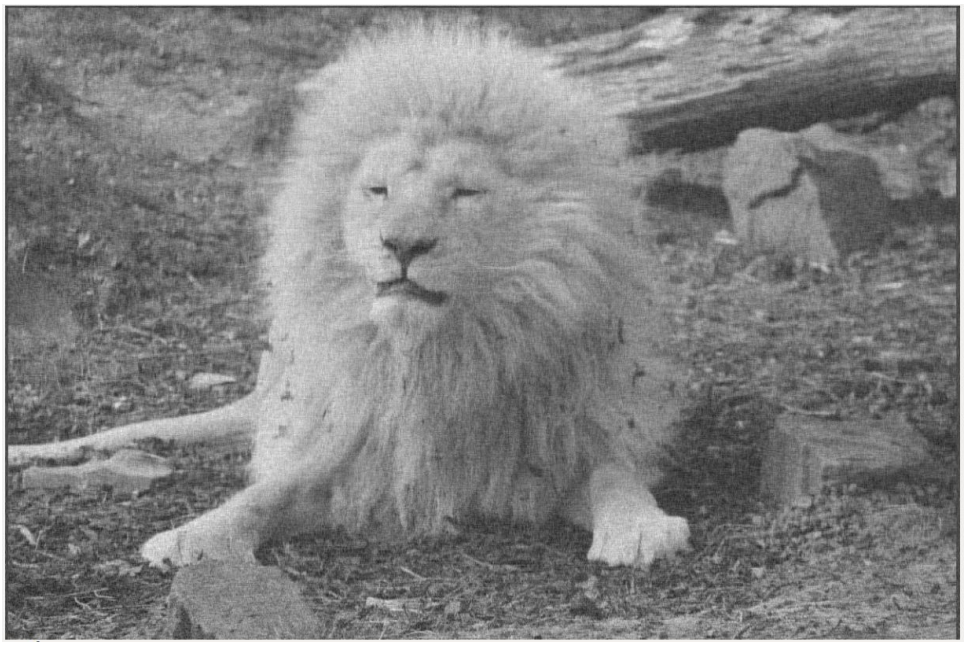
Since we only have the corrupted image, we can denoise the image using soft-thresholding, SURE, Optshrink, etc.

I present one solution where we use the Schatten-p-norm formulation discussed in HW#9.

Step #1: Choose an appropriate regularization parameter when you have don't have a ground truth by drawing the scree plot. I think the curve starts flattening between 100 to 200, so I chose to look for regularization parameters under a certain threshold. To do so, I took the 200th singular value and calculated and "penalized" singular values greater than 1.5 * (beta^⅔). And found the best visual image at around beta = 3.




```
Using Plots, Images, ImageIO, ImageView
include("lr_schatten.jl")
Y = load("image.jpg")
for l2reg in 1:1:6
   Xs = lr_schatten(Y, 2^l2reg)
   imshow(Xs) #compare 5 images
end
```

Schatten p-norm	
Low-rank approximation	

The LR approximation is noisier than the Schatten p-norm one but when minimized both somehow produce the same results!

@308 S7.2

S7.2 If we compute a matrix-vector multiplication $\mathbf{C} \cdot \mathbf{x}$, where $\mathbf{C}$ is an N-by-N matrix, and $\mathbf{x}$ an N-by-1 vector, we'll need $O(N^3)$ number of FLOPs. However, with circulant matrices, we can improve to $O(Nlog(N))$ as shown in the slides. This problem lets you try an FFT approach to compute $\mathbf{C^{-1}} \cdot \mathbf{x}$, which reduces time similarly, without calling the expensive `inv(C)` operation. Please write the following function.

```
 """
compute_inverse_mul(C, x)

Compute C^{-1} · x in O(Nlog(N)) number of FLOPs.

Inputs:

   - C:  a circulant matrix of size N-by-N
   - x:  a vector of size N-by-1

Outputs:

   - y:  the result

"""
function compute_inverse_mul(C, x)
```

Here's the test code provided to check for correct answer and time efficiency. You should be able to achieve comparable time efficiency to the `ToeplitzMatrices` implementation and also much less time than the naive implementation `inv(C) * x`.

```
using LinearAlgebra
using ToeplitzMatrices
using BenchmarkTools

function test_compute_inver_mul()
  N = 3000
  c = randn(N)
  C = Circulant(c)
  C_mat = Matrix{eltype(C)}(C)
  x = randn(N)

  y_true = inv(C) * x
  y = compute_inverse_mul(C, x)

  @assert isapprox(y, y_true) "wrong answer"

  @btime inv($C_mat) * $x  # naive implementation
  @btime inv($C) * $x   # overriden implementation
  @btime compute_inverse_mul($C, $x)  # fft implementation
end

test_compute_inver_mul();
```

Solution

```
using FFTW

function compute_inverse_mul(C, x)
  c = C[:, 1]
  C_tr = fft(c)
  X = fft(x)
  Y = X ./ C_tr

  return ifft(Y)
end
```

Analysis
$\mathbf{C}$ is normal, so $\mathbf{C} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}'$. So $\mathbf{C}^{-1}\mathbf{x} = \frac{1}{\sqrt{N}}\mathbf{Q}\mathbf{\Lambda}^{-1}\sqrt{N}\mathbf{Q}'\mathbf{x} = \frac{1}{N}\mathbf{F}\mathbf{\Lambda}^{-1}\mathbf{F}'\mathbf{x} = \frac{1}{N}\mathbf{F}(\mathbf{\Lambda}^{-1}\mathbf{X}) = \texttt{ifft}(\mathbf{\Lambda}^{-1}\mathbf{X}) = \texttt{ifft}(1\ ./\texttt{fft}(\mathbf{C}[:,1])\ .*\ \texttt{fft}(\mathbf{x})) = \texttt{ifft}(\texttt{fft}(\mathbf{x})\ ./\ \texttt{fft}(\mathbf{C}[:,1]))$.
Here's my results of the testing code.

```
1  test_compute_inver_mul();
```

```
484.421 ms (8 allocations: 70.18 MiB)
171.290 μs (38 allocations: 189.38 KiB)
175.967 μs (116 allocations: 313.06 KiB)
```