Pr. 1.
The GD iteration for the LS problem $\arg\min_{\boldsymbol{x}\in\mathbb{R}^N} f(\boldsymbol{x})$, $f(\boldsymbol{x}) \triangleq \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{y}\|_2^2$ for $M \times N$ matrix $\boldsymbol{A}$ and length-M vector $\boldsymbol{y}$ is $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y})$. Instead of using a fixed step size α, an alternative is to do a **line search** to find the best step size *at each iteration*. This variation is called **steepest descent** (or GD with a line search). For a given preconditioning matrix $\boldsymbol{P} \in \mathbb{F}^{N\times N}$, here is how preconditioned steepest descent works:

$$\begin{aligned}
\boldsymbol{g}_k &= \nabla f(\boldsymbol{x}_k) = \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y}) && \text{gradient} \\
\boldsymbol{d}_k &= -\boldsymbol{P}\boldsymbol{g}_k && \text{search direction} \\
\alpha_k &= \arg\min_{\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) && \text{line search} \\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k && \text{update.}
\end{aligned}$$

Find an expression for the step size α_k that is easily implemented, in terms of (some of) $\boldsymbol{x}_k$, $\boldsymbol{g}_k$, $\boldsymbol{d}_k$, $\boldsymbol{A}$, $\boldsymbol{P}$ and/or $\boldsymbol{y}$. Strive to find an expression that requires as little computation as possible by using quantities already computed.

Pr. 2.
Consider the following **regularized low-rank approximation** method:

$$\hat{\boldsymbol{X}} = \arg\min_{\boldsymbol{X}} \frac{1}{2}\|\boldsymbol{Y} - \boldsymbol{X}\|_{\mathrm{F}}^2 + \beta \|\boldsymbol{X}\|_* .$$

Determine an expression for the approximation error $\|\hat{\boldsymbol{X}} - \boldsymbol{Y}\|_{\mathrm{F}}$. Simplify as much as possible.

Pr. 3.
(Image compression with an SVD)
For any rank r matrix $\boldsymbol{A}$ with SVD $\boldsymbol{A} = \sum_{i=1}^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i'$, the solution to the **low-rank approximation** problem

$$\boldsymbol{A}_{\mathsf{opt}} = \arg\min_{\boldsymbol{X}} \|\boldsymbol{A} - \boldsymbol{X}\|_{\mathrm{F}} \text{ subject to } \operatorname{rank}(\boldsymbol{X}) \le K,$$

is given by $\boldsymbol{A}_{\mathsf{opt}} = \sum_{i=1}^{\min(K,r)} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i'$.
The approximation error between $\boldsymbol{A}_{\mathsf{opt}}$ and $\boldsymbol{A}$ is given by $\|\boldsymbol{A} - \boldsymbol{A}_{\mathrm{opt}}\|_{\mathrm{F}} = \sqrt{\sum_{i=\min(K,r)+1}^r \sigma_i^2}$.
This expression reveals that the approximation error decrease monotonically to zero with increasing K. When $K \ge r$, the approximation error is identically zero (in theory, ignoring finite numerical precision effects) because we have just reconstituted the entire $\boldsymbol{A}$ matrix via the sum of the pertinent r outer products.
This problem examines the visual quality of the approximation and investigates the feasibility of an SVD-based **image compression** scheme whereby, instead of storing the mn numbers in the $\boldsymbol{A}$ matrix, we instead store $K(m+n+1)$ numbers corresponding to the m, n and 1 number(s) for each of the K left and right singular vectors and the singular values, respectively.
In `Julia`, to load and view an image use:

```
using ColorTypes: Gray # you may need to add this package
using FileIO: load # ditto
using MIRTjim: jim
image_color = load("your_image.jpg") # MxN array of RGB
image = Float32.(Gray.(image_color)) # MxN array of Float32
jim(image') # you probably want the transpose here
```

If you have any problems with the `FileIO` package, then, instead of using your own image, you may use the following test image:

```
using ImagePhantoms
using MIRTjim: jim
image = shepp_logan(256, SheppLoganEmis())
jim(image)
```

Download the notebook `svd_image_compression_demo.ipynb` from the hw08 directory on Canvas, along with the data files therein. Run the notebook (no coding necessary for this step) and examine how changing the rank of the approximation changes the error and the quality of the image representation.

To install the `Images` package you might also need to install the `ImageMagick` package which might require "Developer Mode" on a PC, or perhaps running `Julia` as an administrator. On a Mac you might need to add the `QuartzImageIO` package.

(a) Replace the image in the notebook with a picture of your choosing. Use the SVD to find a compressed version (via the reduced rank procedure) of the image that uses 20% as many bits as the original image. For simplicity, assume that all real numbers are stored using the same number of bits, e.g., 32 bits for `Float32`.

Submit images of the original and compressed images and a scatter plot of the error as a function of the rank. Your horizontal axis should go from 0 to $\min(M, N)$.
Is the quality of the compressed image "good enough"?

(b) Repeat all of part (a) with the two MIT logo images in the hw08 directory on Canvas. Try it first for the basic "nameless" logo **without** any extra lettering.

If you have any problems loading the `.jpg` file, then instead you can load the corresponding `.fld` file as follows:

```
using FileIO
image = Float32.(load("mit_logo_nameless.fld"))
```

You might also need to add the package `AVSfldIO` to use this.

Find the smallest rank that gives decent image quality.

Now try that same rank for compressing the logo with the extra lettering.

Which logo image is well compressible with a much lower rank? Why? Hint: What is the (approximate) rank of the MIT logo?

(c) Write a function called `compress_image` that takes as input an $m \times n$ matrix $\boldsymbol{A}$ (an image) and a scalar p in $(0, 1)$ and returns an $m \times n$ matrix `Ac` containing a compressed version of $\boldsymbol{A}$ that can be represented using at most $(100 \times p)\%$ as many bits as $\boldsymbol{A}$. Your function should also return the rank, r, of the compressed matrix. Use the largest rank you can while still satisfying the compression requirement.

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

```
"""
    Ac, r = compress_image(A, p)

In:
* `A` `m × n` matrix
* `p` scalar in `(0, 1]`

Out:
* `Ac` a `m × n` matrix containing a compressed version of `A`
that can be represented using at most `(100 * p)%` as many bits
required to represent `A`
* `r` the rank of `Ac`
"""
function compress_image(A, p)
```

Email your solution as an attachment to `eecs551@autograder.eecs.umich.edu`.

Pr. 4.
(OptShrink for low-rank matrix denoising: small data size)
This problem implements and explores the **OptShrink** method of Prof. Nadakuditi as discussed in the course notes.

(a) Write a function called `optshrink1` that has two inputs. The first input $\boldsymbol{Y}$ is a noisy data array modeled to be of the form $\boldsymbol{Y} = \boldsymbol{X} + \boldsymbol{\varepsilon}$ where $\boldsymbol{X}$ is a low-rank matrix and $\boldsymbol{\varepsilon}$ is a noise matrix of the same size. The second argument is guess of the rank of $\boldsymbol{X}$. (In practice one might guess this rank by examining the scree plot for $\boldsymbol{Y}$.) The function must return the rank-r estimate $\hat{\boldsymbol{X}}$ computed by the OptShrink method.

For this problem, you are permitted to use Matlab code for OptShrink that you find on the web as inspiration, as long as you cite it in your solution code.

For this problem it is acceptable if your function works only for matrices where the dimensions are not too large.

In `Julia`, your file should be named `optshrink1.jl` and should contain the following function:

```
"""
    Xh = optshrink1(Y::AbstractMatrix, r::Int)

Perform rank-r denoising of data matrix `Y` using the OptShrink method
by Prof. Nadakuditi in this May 2014 IEEE Tr. on Info. Theory paper:
http://doi.org/10.1109/TIT.2014.2311661

In:
- `Y` 2D array where `Y = X + noise` and goal is to estimate `X`
- `r` estimated rank of `X`

Out:
- `Xh` rank-`r` estimate of `X` using OptShrink weights for SVD components

This version works only if the size of `Y` is sufficiently small,
because it performs calculations involving arrays roughly of
`size(Y'*Y)` and `size(Y*Y')`, so neither dimension of `Y` can be large.
"""
function optshrink1(Y::AbstractMatrix, r::Int)
```

Email your solution as an attachment to `eecs551@autograder.eecs.umich.edu`.

The template above includes type declarations for the function arguments. You are not required to use such declarations. Their benefit is that if a user tries to call the function with a wrong argument type, then the error message will be more informative. Another benefit is that one can use the same function names for different argument types, a key feature of `Julia` called **multiple dispatch**.

(b) Test your code yourself using your own $\boldsymbol{Y} = \boldsymbol{X} + \boldsymbol{\varepsilon}$ where $\boldsymbol{X}$ is a low-rank array and the noise level is small, *before* submitting to the autograder.

After your code passes the autograder, complete the following test code so that it computes both the OptShrink estimate $\hat{\boldsymbol{X}}$ and the standard rank-1 approximation to $\boldsymbol{Y}$ and see which is closer to the true $\boldsymbol{X}$.

```
# include("optshrink1.jl") # uncomment if you need this
using Random: seed!
using LinearAlgebra: norm
seed!(0)
X = randn(30) * randn(20)' # outer product
Y = X + 40 * randn(size(X))
Xh_opt = optshrink1(Y, 1)
# Xh_lr = # you finish this to make the conventional rank-1 approximation

@show norm(Xh_opt - X)
@show norm(Xh_lr - X)
```

Submit (a screen shot of) your modified test code and the `norm` output values to gradescope.

Does OptShrink provide a better estimate of $\boldsymbol{X}$ than classical low-rank approximation?

Here is a test case you can use to check your code for D, D' and the output of OptShrink.

```
using Random: seed!
seed!(0)
```

```
Y = rand(5,3)
Yh = optshrink1(Y, 2)

Y×
53 Array{Float64,2}:
 0.823648  0.203477   0.585812
 0.910357  0.0423017  0.539289
 0.164566  0.0682693  0.260036
 0.177329  0.361828   0.910047
 0.27888   0.973216   0.167036

Yh×

53 Array{Float64,2}:
 0.598335  0.236597   0.56591
 0.609996  0.154529   0.556546
 0.176241  0.0837109  0.169988
 0.448318  0.332532   0.46055
 0.237647  0.519169   0.324806

k=1: (D, D') = (0.3700091282005337, -0.4873331497276365)
k=2: (D, D') = (2.9932074647136897, -15.14496499962344)
```

Pr. 5.
(OptShrink for low-rank matrix denoising problems: big data)

(a) The OptShrink method needs to evaluate the function $D(z, \boldsymbol{S})$ and its derivative $D'(z, \boldsymbol{S})$ for a certain (rectangular) diagonal matrix $\boldsymbol{S}$. (See course notes.) If one of the two dimensions of this matrix is large, then either $\boldsymbol{S}\boldsymbol{S}'$ or $\boldsymbol{S}'\boldsymbol{S}$ may become impractical to store and manipulate. The course notes show how to compute $D(z, \boldsymbol{S})$ without forming $\boldsymbol{S}\boldsymbol{S}'$ or $\boldsymbol{S}'\boldsymbol{S}$, thereby providing a step towards making OptShrink suitable for larger problems. Similarly, for this problem you can first derive an efficient expression for $D'(z, \boldsymbol{S})$. Alternatively you can simply think about how to modify the code directly by thinking about how to avoid dealing with large matrices.

Write a function called `optshrink2` that has the same inputs and output as `optshrink1` but that is practical to use even when one of the two dimensions of $\boldsymbol{Y}$ is very large, *e.g.*, if $\boldsymbol{Y}$ is $10^6 \times 100$.
Your solution may not use `tr` (trace) or `inv` or other such functions of large matrices.

If you find any code online that solves this problem, you may use it for inspiration as long as you cite it. Please let me know by email if you do, because I am not aware of any!

In `Julia`, your file should be named `optshrink2.jl` and should contain the following function:

```
"""
    Xh = optshrink2(Y::AbstractMatrix, r::Int)

Perform rank-`r` denoising of data matrix `Y` using the OptShrink method
by Prof. Nadakuditi in this May 2014 IEEE Tr. on Info. Theory paper:
http://doi.org/10.1109/TIT.2014.2311661

In:
- `Y` 2D array where `Y = X + noise` and goal is to estimate `X`
- `r` estimated rank of `X`

Out:
- `Xh` rank-`r` estimate of `X` using OptShrink weights for SVD components

This version works even if one of the dimensions of `Y` is large,
as long as the other is sufficiently small.
"""
function optshrink2(Y::AbstractMatrix, r::Int)
```

Email your solution as an attachment to `eecs551@autograder.eecs.umich.edu`.

Test your code yourself using your own $\boldsymbol{Y} = \boldsymbol{X} + \boldsymbol{\varepsilon}$ where $\boldsymbol{X}$ is a low-rank array and the noise level is small, *before* submitting to the autograder. The results of `optshrink1` and `optshrink2` should be nearly identical for small problems. Then test your code using a very tall or very wide $\boldsymbol{Y}$ to make sure it can handle it, *before* submitting to the autograder.

(b) After your code passes the autograder, complete the following test code so that it computes both the OptShrink estimate $\hat{\boldsymbol{X}}$ and the standard rank-1 approximation to $\boldsymbol{Y}$ and see which is closer to the true $\boldsymbol{X}$.

```
# include("optshrink2.jl") # uncomment if needed
using LinearAlgebra: norm
using Random: seed!
seed!(0)
X = randn(10^5) * randn(100)' / 8 # test large case now
Y = X + randn(size(X))
Xh_opt = optshrink2(Y, 1)
# Xh_lr = # you finish this part

@show norm(Xh_opt - X)
@show norm(Xh_lr - X)
```

Submit (a screen shot of) your modified test code and the `norm` output values to gradescope.

Does OptShrink provide a better estimate of $\boldsymbol{X}$ than classical low-rank approximation?

(c) (Not graded) Will your solution work if both dimensions of $\boldsymbol{Y}$ are large? Why or why not?

Pr. 6.
(Shrinkage using $\|\cdot\|_p$ with $p = 1/2$ non-convex) When v is real and nonnegative, the course notes derive the following minimization problem solution:

$$\arg\min_{x \geq 0} \frac{1}{2}(v - x)^2 + \beta |x| = [v - \beta]_+, \qquad [t]_+ \triangleq \max(t, 0) = \begin{cases} t, & t > 0 \\ 0, & \text{otherwise,} \end{cases}$$

for $\beta > 0$. A drawback of this **soft thresholding** formulation in the context of **low-rank matrix approximation** (and in other sparsity problems) is that even the large singular values are shrunk by β. An alternative is to replace the **regularizer** $|x|$ with a function that increases less rapidly, such as $|x|^{1/2}$.
So now consider this minimization problem for $\beta > 0$:

$$\hat{x} = \arg\min_{x \geq 0} f(x; v, \beta), \quad f(x; v, \beta) \triangleq \frac{1}{2}(v - x)^2 + \beta |x|^{1/2}.$$

Letting $t = \sqrt{x}$, we want to minimize $\frac{1}{2}|v - t^2|^2 + \beta t$. Differentiating yields the (depressed) cubic equation $t^3 - vt + \beta/2 = 0$ that turns out to have 3 real roots when $v \geq 0$ is sufficiently large. One can solve it using a **trigonometric method** leading to the solution $\hat{t} = 2\sqrt{\frac{v}{3}} \cos\left(\frac{1}{3} \arccos\left(-\frac{3\beta}{4v}\sqrt{\frac{3}{v}}\right)\right)$. By considering the second derivative, one can (optionally) verify that this solution is correct when $v > \frac{3}{2}\beta^{2/3}$. Thus the final solution is

$$\hat{x} = \begin{cases} \frac{4}{3} v \cos^2\left(\frac{1}{3} \arccos\left(-\frac{3^{3/2}\beta}{4v^{3/2}}\right)\right), & v > \frac{3}{2}\beta^{2/3} \\ 0, & \text{otherwise.} \end{cases}$$

(a) Write a `Julia` function that evaluates this solution for a given regularization parameter β and for an array of nonnegative input v values.

In `Julia`, your file should be named `shrink_p_1_2.jl` and should contain the following function:

```
"""
    out = shrink_p_1_2(v, reg::Real)

Compute minimizer of ``1/2 |v - x|^2 + reg |x|^p``
for `p=1/2` when `v` is real and nonnegative.

In:
* `v` scalar, vector, or array of (real, nonnegative) input values
* `reg` regularization parameter

Out:
* `xh`  solution to minimization problem for each element of `v`
(same size as `v`)

"""
function shrink_p_1_2(v, reg::Real)
```

Email your solution as an attachment to `eecs551@autograder.eecs.umich.edu`.

Before submitting, test your code yourself by making the plot needed for the next part and seeing whether that plot makes sense.

You will use this function for a low-rank matrix completion problem in a later HW. Like $\|\cdot\|_0$, the $|x|^{1/2}$ function is **non-convex**, but at least it is continuous, whereas $\|\cdot\|_0$ is discontinuous.

(b) Plot the solution for `v = LinRange(0, 8*reg, 801)` when $\beta = 2$. Also plot the soft thresholding function on this same graph, and show the line of identity.

Discuss what advantages this shrinkage function appears to have over soft thresholding.

Submit your plot and comments to gradescope.

Pr. 7.

Apply the **multi-dimensional scaling** (MDS) algorithm to the following distance matrix: $\boldsymbol{D} = \begin{bmatrix} 0 & a & 2a \\ a & 0 & a \\ 2a & a & 0 \end{bmatrix}$.
Do the work by hand - no `Julia`. (You may use `Julia` to check your answer.) Hint: look for a rank-1 outer product.
Plot (by hand) the resulting coordinates in 2D. Use $d = 2$ throughout the problem.
Verify that your answer makes sense in light of the original distance matrix $\boldsymbol{D}$.

Pr. 8.
Multidimensional scaling: discussion task
Download the `task-4-mds.ipynb` jupyter notebook file from Canvas under this week's discussion folder and follow all instructions to complete the task. You may work individually, but we recommend that you work in pairs or groups of three.
When you are finished, upload your solutions to gradescope. Note that the submission for the task is separate from the rest of the homework because the task allows you to submit as a group. Only upload one submission per group! Whoever uploads the group submission should add all group members in gradescope, using the "View or edit group" option on the right-hand sidebar after uploading a PDF and matching pages. Make sure to add all group members, because this is how they will receive credit.

Non-graded problem(s) below
(Solutions will be provided for self check; do not submit to gradescope.)

Pr. 9.
The course notes discuss the (convex) regularized **low-rank approximation** problem

$$\hat{\boldsymbol{X}} = \underset{\boldsymbol{X} \in \mathbb{C}^{M \times N}}{\arg\min} \frac{1}{2} \|\boldsymbol{Y} - \boldsymbol{X}\|_{\mathrm{F}}^2 + \beta \|\boldsymbol{X}\|_* .$$

Determine the solution when we replace the Frobenius norm with the **spectral norm**:

$$\hat{\boldsymbol{X}} = \underset{\boldsymbol{X} \in \mathbb{C}^{M \times N}}{\arg\min} \frac{1}{2} \|\boldsymbol{Y} - \boldsymbol{X}\|_{2}^2 + \beta \|\boldsymbol{X}\|_* .$$

It is sufficient to solve it for the case where $\mathrm{rank}(\boldsymbol{Y}) = 1$.

Pr. 10.
Prove that $\sigma_1^2(\boldsymbol{A})\boldsymbol{I} \succeq \boldsymbol{A}'\boldsymbol{A}$. This inequality is important for establishing convergence of certain iterative algorithms for **least-squares** and related problems.

Pr. 11.
For $\boldsymbol{Y} \in \mathbb{C}^{M \times N}$, determine (analytically) the solution of the **regularized low-rank approximation** method:

$$\hat{\boldsymbol{X}} = \underset{\boldsymbol{X} \in \mathbb{C}^{M \times N}}{\arg\min} \|\boldsymbol{Y} - \boldsymbol{X}\|_* + \beta \frac{1}{2} \|\boldsymbol{X}\|_{\mathrm{F}}^2 .$$

(This problem was suggested by a student in f17 for Exam2.)
Think about your solution and consider whether it seems to be a good method for low-rank approximation.

Pr. 12.
Prove that if $\boldsymbol{P}$ is a projection matrix, and, for a square matrix $\boldsymbol{A}$, we define $e^{\boldsymbol{A}} = \sum_{k=0}^{\infty} \frac{1}{k!} \boldsymbol{A}^k$, where $k!$ denotes factorial of k, then

$$e^{\boldsymbol{P}} = \boldsymbol{I} + (e-1)\boldsymbol{P}.$$

Hint: what is $\boldsymbol{P}^k$ when $\boldsymbol{P}$ is a projection matrix?
