Pr. 1. (sol/hsj74)
$\alpha_k = \arg\min_\alpha f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ where $f(\boldsymbol{x}) = \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2$. Define $h(\alpha) = f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ and $\boldsymbol{g}_k = \nabla f(\boldsymbol{x}_k) = \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y})$ so $\boldsymbol{d}_k = -\boldsymbol{P}\boldsymbol{g}_k$. Then $0 = \dot{h}(\alpha_k) = \boldsymbol{d}_k' \boldsymbol{A}'(\boldsymbol{A}(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) - \boldsymbol{y})$ so solving yields $\alpha_k = \dfrac{-\boldsymbol{d}_k' \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y})}{\|\boldsymbol{A}\boldsymbol{d}_k\|_2^2} = \dfrac{-\boldsymbol{d}_k' \boldsymbol{g}_k}{\|\boldsymbol{A}\boldsymbol{d}_k\|_2^2}$.
The latter form is useful because it reuses the gradient $\boldsymbol{g}_k$ already computed.
An alternative derivation is to write $h(\alpha) = \frac{1}{2} \|\boldsymbol{A}(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) - \boldsymbol{y}\|_2^2 = \frac{1}{2} \|\boldsymbol{u}\alpha - \boldsymbol{b}\|_2^2$ where $\boldsymbol{u} = \boldsymbol{A}\boldsymbol{d}_k$ and $\boldsymbol{b} = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_k$. This is simply a LS problem in α so the minimizer is $\alpha = \boldsymbol{u}^+ \boldsymbol{b} = \frac{\boldsymbol{u}'\boldsymbol{b}}{\|\boldsymbol{u}\|_2^2}$ which simplifies to the above expressions.

Pr. 2. (sol/hsj65)
When $\boldsymbol{Y}$ has rank r and

$$\boldsymbol{Y} = \sum_{k=1}^{r} \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k',$$

the SVD soft-thresholding solution is

$$\hat{\boldsymbol{X}} = \sum_{k=1}^{r} [\sigma_k - \beta]_+ \boldsymbol{u}_k \boldsymbol{v}_k'.$$

Thus the approximation error is

$$\|\boldsymbol{Y} - \hat{\boldsymbol{X}}\|_{\mathrm{F}} = \left\| \sum_{k=1}^{r} \left(\sigma_k - [\sigma_k - \beta]_+\right) \boldsymbol{u}_k \boldsymbol{v}_k' \right\|_{\mathrm{F}} = \sqrt{\sum_{k=1}^{r} \left(\sigma_k - [\sigma_k - \beta]_+\right)^2}.$$

That answer is acceptable. Two further simplifications are:

$$\|\boldsymbol{Y} - \hat{\boldsymbol{X}}\|_{\mathrm{F}} = \sqrt{\sum_{k=1}^{r} \left(\min(\sigma_k, \beta)\right)^2} = \sqrt{\beta^2 \cdot |\{\sigma_k : \sigma_k > \beta\}| + \sum_{k\,:\,\sigma_k \le \beta} \sigma_k^2},$$

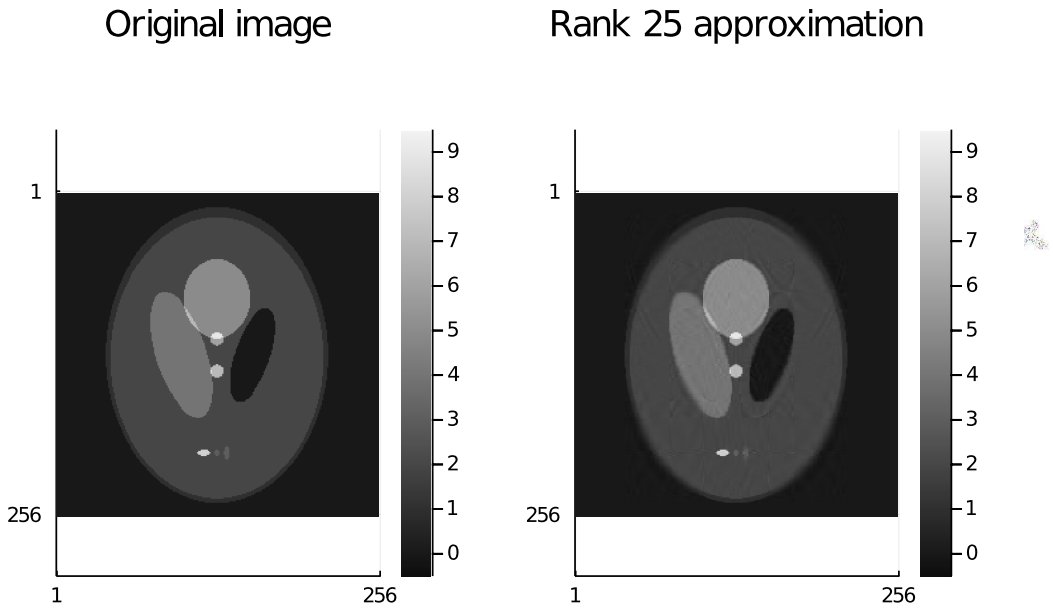
where $|\cdot|$ denotes the cardinality of the set, *i.e.*, the number of singular values above β.

Pr. 3. (sol/hs042)

(a) The idea is to view the image as a matrix $\boldsymbol{A} = \sum_{k=1}^{\min(M,N)} \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k'$, for the $M \times N = 256 \times 256$ phantom image shown below. We know that the optimal (with respect to any unitarily invariant norm) rank-K approximation to $\boldsymbol{A}$ is $\boldsymbol{A}_K = \sum_{k=1}^{K} \sigma_k \boldsymbol{u}_k \boldsymbol{v}_k'$. This approximation requires $(1 + M + N) = 513$ real numbers for each additional term. The original matrix stores $256 \times 256 = 65536$ numbers. One-fifth of this total is 13107. Thus the number of terms that gives 48000 real numbers is $13017/513 = 25.4 \approx 25$.

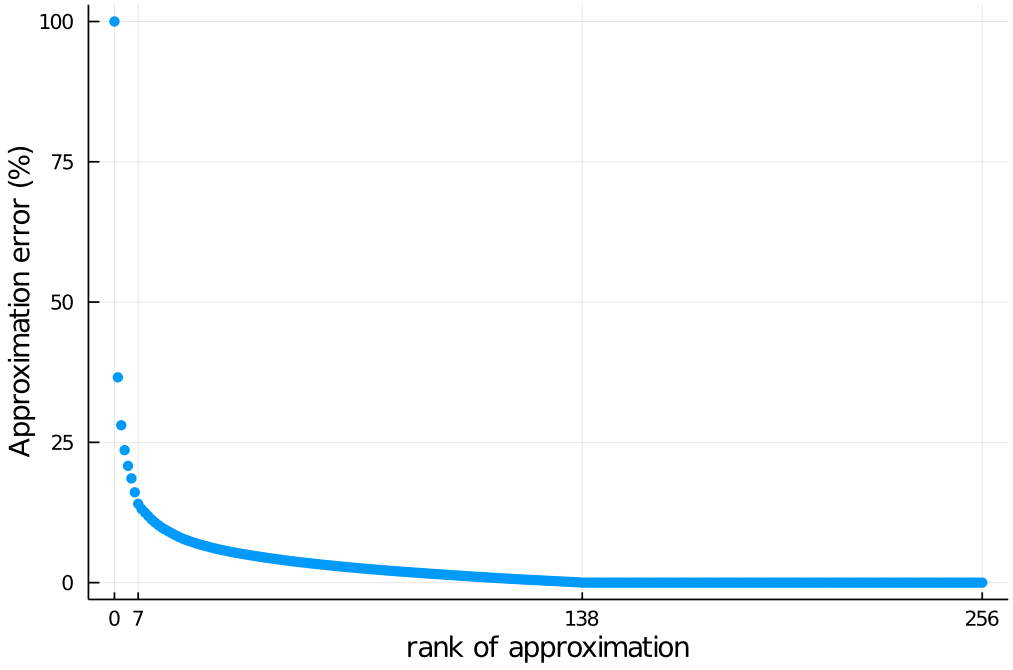
Note that if you just took one fifth of the number of terms in the full image, that would give you $N/5 \approx 51$ terms, but that would require roughly 40% memory as opposed to the 20%.

Original image (left) versus low-rank approximation image (right):



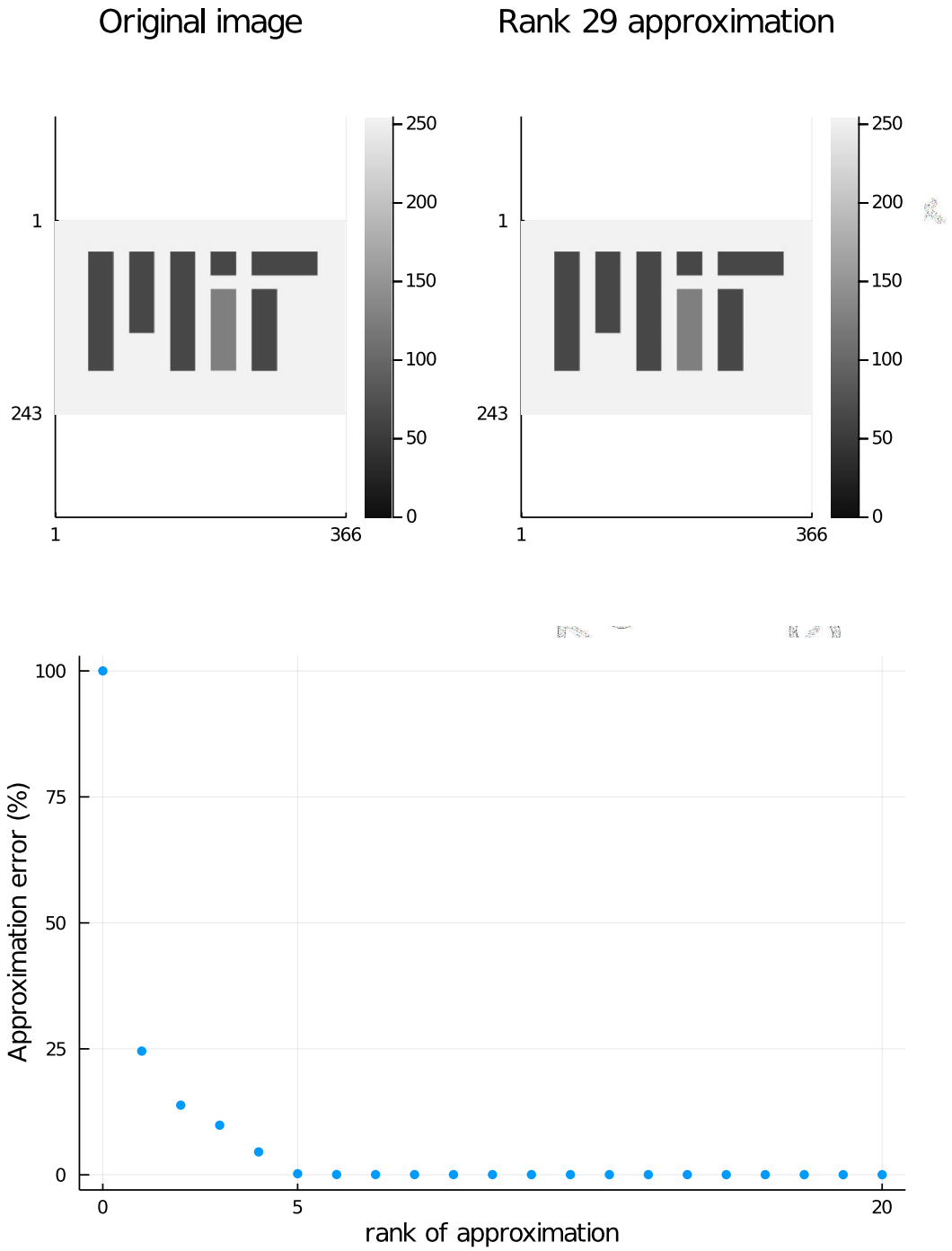
The compressed image is good enough to recognize as an ellipse phantom but its visual quality is noticeably degraded. This is an image with fine details that are not well approximated by a small number of outer products.

Relative error of low-rank approximation:



Initially the error drops rapidly as the rank increases from 0 to 10 or so, but then it decreases slowly. For $K \geq 138$ the error becomes essentially 0, because this image has many columns of 0 values so its effective rank is only $r = 138$.

(b) The MIT logo without the lettering has rank 4 if the background white space is represented as a zero in the matrix; otherwise it is rank 5. Note that the singular values σ_k for $k \geq 6$ are not numerically 0, but that they are very, very small. Either way it is nearly perfectly compressible with a low-rank approximation. In contrast, the image of the logo with the lettering is full rank, so the approximation error decreases as the approximation rank increases. Being able to spot low-rank patterns is a valuable skill!



(c) In general, for an $M \times N$ matrix, the rank K approximation requires $(1 + M + N)K$ real numbers. Hence, for a given compression fraction p, we want to choose the maximum value of K such that $\frac{(1+M+N)K}{MN} \leq p$, *i.e.*, $K = \text{floor}\left(\frac{pMN}{1+M+N}\right)$.

A possible `Julia` implementation is

```
using LinearAlgebra: svd, Diagonal

"""
    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)

    # Parse inputs
    m, n = size(A)

    # Compute compression factor
    r = floor(Int, (p * m * n) / (m + n + 1))
    r = min(r, m, n) # just to be safe

    # Compute compressed image
    U, s, V = svd(A)
    Ac = U[:, 1:r] * Diagonal(s[1:r]) * V[:, 1:r]'

    return Ac, r
end
```

Pr. 4. (sol/hsj63)

(a) A possible `Julia` implementation is

```
using LinearAlgebra: svd, Diagonal, I, tr

"""
    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)

    # U is [m, min(m,n)], s is min(m,n), V is [n, min(n,m)]
    (U, s, V) = svd(Y) # economy

    (m,n) = size(Y)
    r = min(r, m, n) # ensure r <= min(m,n)

    # make rectangular diagonal "S" (\hat{\Sigma}_{\hat{r}}) (m-r)x(n-r) per paper
    if m >= n # tall
        # [(n-r×)(n-r); (m-n×)(n-r)] -> [(m-r×)(n-r)]
        S = [Diagonal(s[(r+1):n]); zeros(m-n,n-r)]
    else # wide
        # [(m-r×)(m-r), (m-r×)(n-m)] -> [(m-r×)(n-r)]
        S = [Diagonal(s[(r+1):m]) zeros(m-r,n-m)]
    end
    #@show size(S) # [(m-r) × (n-r)]

    w = zeros(r)
    for k=1:r
        (D, Dder) = D_transform_from_matrix(s[k], S)
        w[k] = -2*D/Dder
    end

    Xh = U[:,1:r] * Diagonal(w) * V[:,1:r]' # (m,r) (r,r) (n,r)'
    return Xh
end


# X is n × m, where this internal n,m differ from n,m in calling routine
# this version makes "big" matrices so it is impractical for big data
function D_transform_from_matrix(z, X)
    (n,m) = size(X)
    In = I # I(n)
    Im = I # I(m)

    denom_n = z^2 * In - X * X' # denominators of the d-transform equation
    denom_m = z^2 * Im - X' * X
    inv_denom_n = inv(Array(denom_n))
    inv_denom_m = inv(Array(denom_m))

    D1 = (1/n) * tr(z * inv_denom_n)
    D2 = (1/m) * tr(z * inv_denom_m)

    D = D1*D2 # eq (16a) in Nadakuditi paper

    # derivative of D transform
    D1_der = (1/n) * tr(-2 *z^2 * inv_denom_n^2 + inv_denom_n)
```

```
    D2_der = (1/m) * tr(-2 *z^2 * inv_denom_m^2 + inv_denom_m)

    D_der = D1*D2_der + D2*D1_der # eq (16b) in Nadakuditi paper

    return (D, D_der)
end
```

(b) The modified test code is

```
using Random: seed!
using LinearAlgebra: norm, svd
include(ENV["hw551test"] * "optshrink1.jl")
seed!(0)
X = randn(30) * randn(20)' # outer product
Y = X + 40 * randn(size(X))
Xh_opt = optshrink1(Y, 1)

(U,s,V) = svd(Y) # added line
Xh_lr = U[:,1] * s[1] * V[:,1]' # finished line

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

The output values are:
`norm(Xh_opt - X) = 131.74`
`norm(Xh_lr - X) = 403.03`

Grader: if you see the (incorrect) values 146.02 and 401.19, please make a rubric and report to me.

So in this case OptShrink works *much* better than conventional low-rank approximation.
In practice the improvement depends on the noise level and other factors.

Pr. 5. (sol/hsj64)

(a) A possible `Julia` implementation is

```
using LinearAlgebra: svd, Diagonal

"""
    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)

    # U is [m, min(m,n)], s is min(m,n), V is [n, min(n,m)]
    (U, s, V) = svd(Y) # economy of course

    (m,n) = size(Y)
    r = min(r, m, n) # ensure r <= min(m,n)

    sv = s[(r+1):end] # tail singular values for noise estimation

    w = zeros(r)
    for k=1:r
```

```
        (D, Dder) = D_transform_from_vector(s[k], sv, max(m,n)-r, min(m,n)-r)
        w[k] = -2*D/Dder
    end

    Xh = U[:,1:r] * Diagonal(w) * V[:,1:r]'

    return Xh
end


# sn is of length n <= m
function D_transform_from_vector(z, sn, m, n)
    sm = [sn; zeros(m-n)] # m × 1

    inv_n = 1 ./ (z^2 .- sn.^2) # vector corresponding to diagonal
    inv_m = 1 ./ (z^2 .- sm.^2)

    D1 = (1/n) * sum(z * inv_n)
    D2 = (1/m) * sum(z * inv_m)

    D = D1*D2 # eq (16a) in Nadakuditi paper

    # derivative of D transform
    D1_der = (1/n) * sum(-2 * z^2 .* inv_n.^2 + inv_n)
    D2_der = (1/m) * sum(-2 * z^2 .* inv_m.^2 + inv_m)

    D_der = D1*D2_der + D2*D1_der # eq (16b) in Nadakuditi paper
    return (D, D_der)
end
```

(b) The modified test code is

```
include(ENV["hw551test"] * "optshrink2.jl")
using LinearAlgebra: svd, norm
using Random: seed!
seed!(0)
X = randn(10^5) * randn(100)' / 8
Y = X + randn(size(X))
Xh_opt = optshrink2(Y, 1)

(U,s,V) = svd(Y)
Xh_lr = U[:,1] * s[1] * V[:,1]' # finished line

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

The output values are:

`norm(Xh_opt - X) = 240.35`

`norm(Xh_lr - X) = 317.08`

Grader: if you see the incorrect values 246.32, 317.12 please report to me.

(c) OptShrink in its present form requires *all* of the singular values of $\boldsymbol{Y}$. If both dimensions are large, then computing the SVD will be impractical.

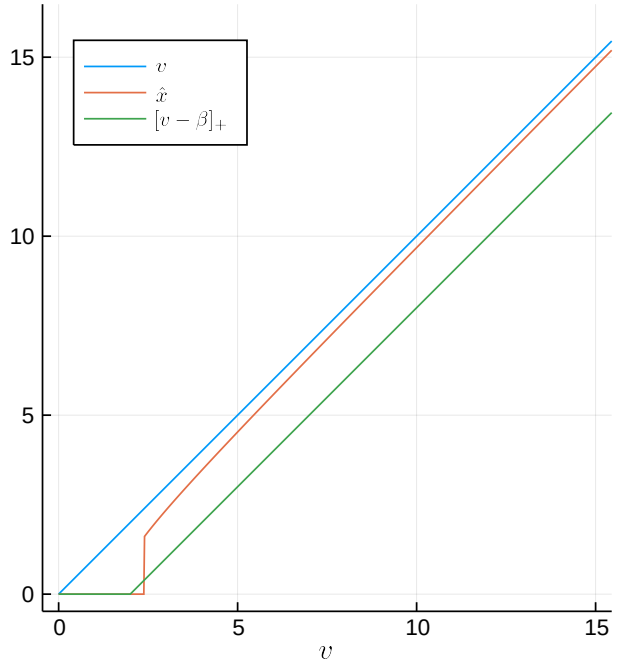
In contrast, conventional rank-K approximation can be applied to large matrices by using the `svds` function in Algebra/Arpack.jl and requesting only the first K components. It may be slow if the matrix is large, but it can be done.

Pr. 6. (sol/hsj67)

(a) A possible `Julia` implementation is

```
"""
    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)
    T = promote_type(eltype(v), typeof(reg), Float32) # optional
    fun = (v) -> (v <= 3/2 * reg^(2/3)) ? 0 :
        4/3 * v * cos(1/3 * acos(- (3^(3/2)*reg) / (4*v^(3/2))))^2
    return T.(fun.(v))
end
```

(b) Here is a plot of the shrinkage function for $p = 1/2$:



It behaves somewhat similar to using a rank regularizer except that even large input values are shrunk somewhat, though much less than when using soft thresholding.

Pr. 7. (sol/hsj6m)
$\boldsymbol{D} = \begin{bmatrix} 0 & a & 2a \\ a & 0 & a \\ 2a & a & 0 \end{bmatrix}$, so squaring each element yields $\boldsymbol{S} = a^2 \begin{bmatrix} 0 & 1 & 4 \\ 1 & 0 & 1 \\ 4 & 1 & 0 \end{bmatrix}$.

De-meaning each row yields $\boldsymbol{S}\boldsymbol{P}_{\mathbf{1}}^{\perp} = \frac{a^2}{3} \begin{bmatrix} -5 & -2 & 7 \\ 1 & -2 & 1 \\ 7 & -2 & -5 \end{bmatrix}$.

De-meaning each column yields $\boldsymbol{P}_{\mathbf{1}}^{\perp}\boldsymbol{S}\boldsymbol{P}_{\mathbf{1}}^{\perp} = \frac{a^2}{3} \begin{bmatrix} -6 & 0 & 6 \\ 0 & 0 & 0 \\ 6 & 0 & -6 \end{bmatrix}$, so the Gram matrix is

$$\boldsymbol{G} = \frac{-1}{2}\boldsymbol{P}_{\mathbf{1}}^{\perp}\boldsymbol{S}\boldsymbol{P}_{\mathbf{1}}^{\perp} = a^2 \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} = a^2 \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}' = \boldsymbol{V}_2 \boldsymbol{\Sigma}_2 \boldsymbol{V}_2', \quad \boldsymbol{\Sigma}_2 = \begin{bmatrix} 2a^2 & 0 \\ 0 & 0 \end{bmatrix}, \quad \boldsymbol{V}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ -1 & 1 \end{bmatrix}.$$

(The second column of $\boldsymbol{V}_2$ can be any vector orthogonal to the first.)
The MDS coordinate matrix estimate is

$$\boldsymbol{C} = \boldsymbol{\Sigma}_2^{1/2} \boldsymbol{V}_2' = \begin{bmatrix} \sqrt{2}a & 0 \\ 0 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & 1 \end{bmatrix} = a \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}.$$

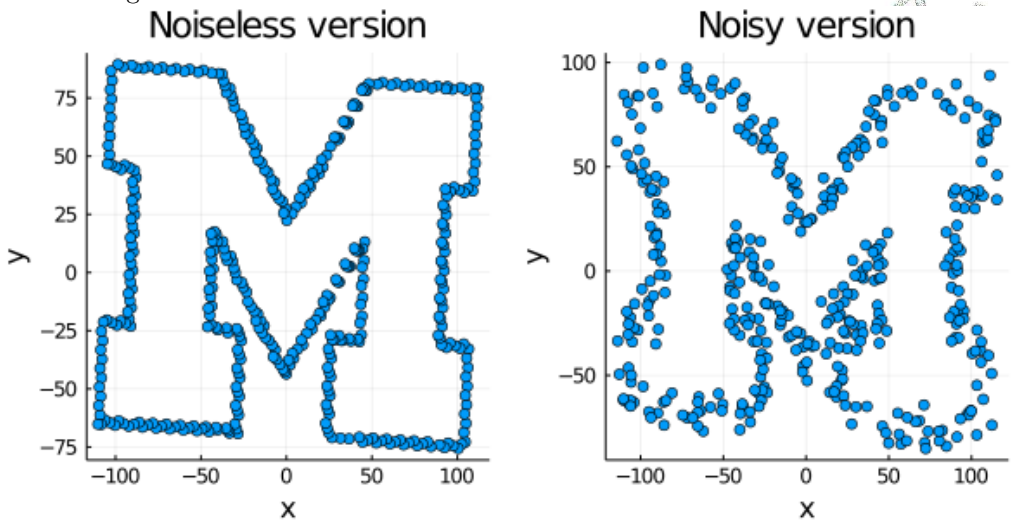
So the coordinates of the 3 points are $\{(-a, 0), (0, 0), (a, 0)\}$, which jibe with the original distance matrix $\boldsymbol{D}$.

Pr. 8. (sol/hs052task)

(a) Part 1: There are many additional ways you could test the function. Some options for the full function are to verify that the solution satisfies each of the uniqueness criteria (e.g., the centroid of each coordinate is zero and the largest magnitude element of each coordinate is positive) or to create test data with a different dimensionality or a different number of points. We could also test individual steps of the function, e.g., we could test that $\boldsymbol{S}$ is symmetric when we create noiseless test data.

(b) Part 2: (autograder)

(c) Part 3: image results



Non-graded problem(s) below

Pr. 9. (sol/hsj61)
As described in the course notes, because $\|\cdot\|_2$ and $\|\cdot\|_*$ are untarily invariant:

$$\hat{\boldsymbol{X}} = \boldsymbol{U}_r \hat{\boldsymbol{\Sigma}}_r \boldsymbol{V}_r', \quad \hat{\boldsymbol{\Sigma}}_r = \arg\min_{\boldsymbol{S} \succeq \boldsymbol{0}} \frac{1}{2} \|\boldsymbol{\Sigma}_r - \boldsymbol{S}\|_2^2 + \beta \|\boldsymbol{S}\|_*, \quad \boldsymbol{S} = \mathrm{Diag}(s_1, \dots, s_r)$$

So here we must solve

$$\arg\min_{s_1,\dots,s_r \geq 0} \left\{ \left(\frac{1}{2} \max_k |\sigma_k - s_k|^2 \right) + \beta \sum_{k=1}^r s_k \right\}.$$

Having $s_k > \sigma_k$ would only increase the cost, so we must solve

$$\arg\min_{s_1,\dots,s_r} \left\{ \left(\frac{1}{2} \max_k (\sigma_k - s_k)^2 \right) + \beta \sum_{k=1}^r s_k \right\}, \text{ s.t. } 0 \leq s_k \leq \sigma_k, \forall k.$$

Consider first the case where $r = 1$, then we have simply

$$\hat{\sigma}_1 = \arg\min_{0 \leq s_1 \leq \sigma_1} f(s_1, \sigma_1, \beta) = \max(\sigma_1 - \beta, 0), \qquad f(s_1, \sigma_1, \beta) \triangleq \frac{1}{2} (\sigma_1 - s_1)^2 + \beta s_1$$

by differentiating and setting to zero and minding the constraints. So the rank-1 minimizer here is

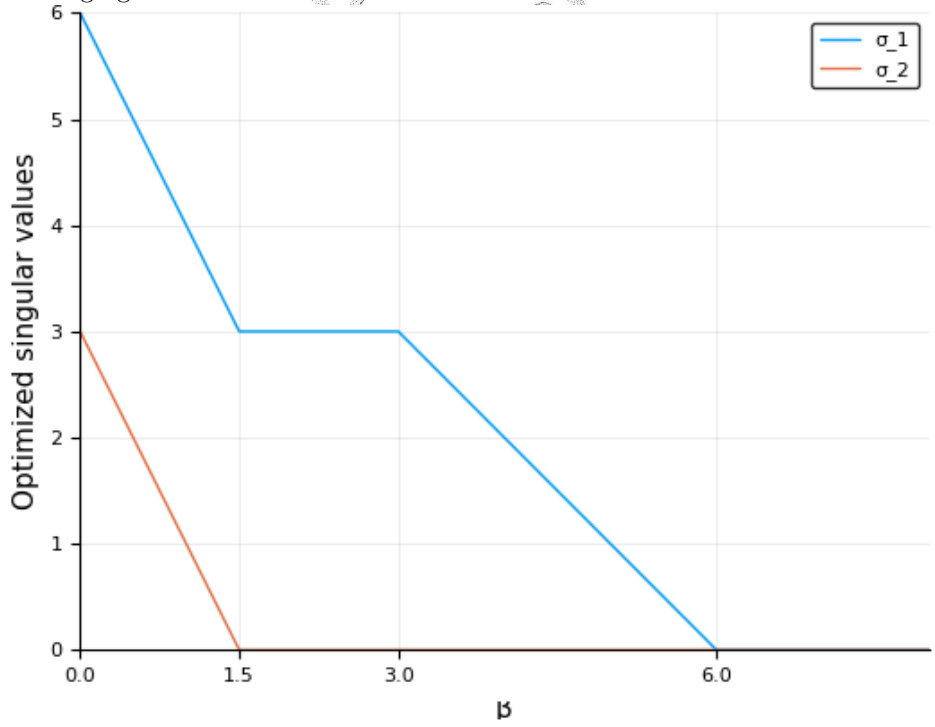
$$\hat{\boldsymbol{X}} = \max(\sigma_1 - \beta, 0) \boldsymbol{u}_1 \boldsymbol{v}_1',$$

which is the same solution as when using the Frobenius norm.
Grader: give full credit for correct solutions to the rank-1 case.
However, for $r = 2$, the solution (found numerically) is different because of the $\max_k$ term:

$$\hat{\sigma}_2 = [\sigma_2 - 2\beta]_+$$

$$\hat{\sigma}_1 = \begin{cases} [\sigma_1 - 2\beta]_+, & 0 \leq \beta \leq \sigma_2/2 \\ \sigma_1 - \sigma_2, & \sigma_2/2 \leq \beta \leq \sigma_2 \\ [\sigma_1 - \beta]_+, & \sigma_2 \leq \beta \leq \sigma_1 \\ 0, & \sigma_1 \leq \beta \end{cases}$$

The following figure illustrates.



I am unsure how to solve it for a general rank. If you think you have a solution, first have a friend in the class check it, and if you both agree it looks correct, then please tell me about it!

Pr. 10. (sol/hsj80)
For all $\boldsymbol{x}$: $\|\boldsymbol{A}\boldsymbol{x}\|_2 \le \|\boldsymbol{A}\|_2 \|\boldsymbol{x}\|_2 \Rightarrow \|\boldsymbol{A}\|_2^2 \|\boldsymbol{x}\|_2^2 \ge \|\boldsymbol{A}\boldsymbol{x}\|_2^2 \Rightarrow \boldsymbol{x}' \|\boldsymbol{A}\|_2^2 \boldsymbol{I}\boldsymbol{x} \ge \boldsymbol{x}'\boldsymbol{A}'\boldsymbol{A}\boldsymbol{x} \Rightarrow \|\boldsymbol{A}\|_2^2 \boldsymbol{I} = \sigma_1^2(\boldsymbol{A})\boldsymbol{I} \succeq \boldsymbol{A}'\boldsymbol{A}$

Pr. 11. (sol/hsj66)

$$\hat{\boldsymbol{X}} = \boldsymbol{U}_r \hat{\boldsymbol{\Sigma}}_r \boldsymbol{V}_r', \quad \hat{\boldsymbol{\Sigma}}_r = \arg\min_{\boldsymbol{S} \succeq \boldsymbol{0}} \|\boldsymbol{\Sigma}_r - \boldsymbol{S}\|_* + \beta \frac{1}{2} \|\boldsymbol{S}\|_{\mathrm{F}}^2, \quad \boldsymbol{S} = \mathrm{Diag}(s_1, \dots, s_r),$$

as described in the course notes. So here we must solve

$$\arg\min_{s_1,\dots,s_r \ge 0} \left\{ \sum_{k=1}^{r} \left(|\sigma_k - s_k| + \beta \frac{1}{2} s_k^2 \right) \right\}, \ \textit{i.e.,} \ \hat{\sigma}_k = \arg\min_{s \ge 0} |\sigma_k - s| + \beta \frac{1}{2} s^2.$$

If $0 < \hat{\sigma}_k < \sigma_k$ then we can treat $|\sigma_k - s|$ as just $\sigma_k - s$ and differentiate to get $0 = -1 + \beta\hat{\sigma}_k$ so $\hat{\sigma}_k = 1/\beta$. But this solution is correct only if $1/\beta \le \sigma_k$. If $1/\beta > \sigma_k$, then one can check that $|\sigma_k - s| + \beta\frac{1}{2}s^2$ is descending on $[0, \sigma_k]$ and ascending for $s > \sigma_k$ so the minimizer is at σ_k. Thus

$$\hat{\sigma}_k = \begin{cases} \sigma_k, & \sigma_k \le 1/\beta \\ 1/\beta, & \sigma_k \ge 1/\beta \end{cases} = \min(1/\beta, \sigma_k).$$

This solution has the undesirable effect of keeping the small singular values untouched but reducing the large singular values to $1/\beta$. This is the opposite of what we want typically, so once again we see that proper choice of norm is crucial for low-rank approximation.

Pr. 12. (sol/hs105)
Since $\boldsymbol{P}$ is a projection matrix, $\boldsymbol{P}^k = \boldsymbol{P}$. Therefore

$$\begin{aligned} e^{\boldsymbol{P}} &= \boldsymbol{P}^0 + \frac{\boldsymbol{P}^1}{1!} + \frac{\boldsymbol{P}^2}{2!} + \ldots + \frac{\boldsymbol{P}^k}{k!} + \ldots = \boldsymbol{I} + \frac{\boldsymbol{P}}{1!} + \frac{\boldsymbol{P}}{2!} + \ldots + \frac{\boldsymbol{P}}{k!} + \ldots \\ &= \boldsymbol{I} + \left(\frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{k!} + \ldots \right) \boldsymbol{P} = \boldsymbol{I} + (e-1)\boldsymbol{P}. \end{aligned}$$
