Pr. 1. (sol/hs024)

(a) Let $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$, where $a_n \neq 0$. The companion matrix of the monic version of $p(x)$ is:

$$\boldsymbol{P} = \begin{bmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & -\frac{a_{n-3}}{a_n} & \ldots & -\frac{a_0}{a_n} \\ 1 & 0 & 0 & \ldots & \vdots \\ 0 & 1 & 0 & \ldots & \vdots \\ & & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 \end{bmatrix}.$$

From class, the roots of $p(x)$ are the eigenvalues of $\boldsymbol{P}$. Thus, the largest (magnitude) root of $p(x)$ is the largest (magnitude) eigenvalue of $\boldsymbol{P}$. We are given that the magnitude of largest eigenvalue is unique, so the largest root of $p(x)$ is non-repeating. We find it by computing the largest (magnitude) eigenvalue of $\boldsymbol{P}$ using the **power iteration**. Starting with a random vector $\boldsymbol{x}_0$, run the algorithm

$$\boldsymbol{x}_k = \frac{\boldsymbol{P}\boldsymbol{x}_{k-1}}{\|\boldsymbol{P}\boldsymbol{x}_{k-1}\|_2}, \qquad k = 1, 2, \ldots,$$

until iteration K, when some termination criterion is reached (e.g., $\|\boldsymbol{x}_{K+1} - \boldsymbol{x}_K\|_2 \leq \varepsilon$, where ε is the required tolerance). If $\boldsymbol{v}_1$ is the eigenvector of $\boldsymbol{A}$ corresponding to λ_1, its largest eigenvalue (in magnitude),

$$\boldsymbol{A}\boldsymbol{x}_K \approx \boldsymbol{A}\boldsymbol{v}_1 = \lambda_1 \boldsymbol{v}_1 \Rightarrow \boldsymbol{x}_K' \boldsymbol{A} \boldsymbol{x}_K \approx \boldsymbol{v}_1' \boldsymbol{A} \boldsymbol{v}_1 = \lambda_1$$

which is the required largest root of $p(x)$.

If $n = 1$, then the only eigenvalue (root) is $-a_0/a_1$, so there is no need to run the power iteration.

(b) Recall from a previous problem that if $\{\lambda_i\}_{i=1}^n$ are the roots of $p(x)$, then the roots of

$$\tilde{p}(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_n$$

will be $\{1/\lambda_i\}_{i=1}^n$. If $\lambda_n \neq 0$ is the smallest magnitude root of $p(x)$, then we can apply the above algorithm to find the largest root of $\tilde{p}(x)$ via the companion matrix of its monic version:

$$\tilde{\boldsymbol{P}} = \begin{bmatrix} -\frac{a_1}{a_0} & -\frac{a_2}{a_0} & -\frac{a_3}{a_0} & \ldots & -\frac{a_n}{a_0} \\ 1 & 0 & 0 & \ldots & \vdots \\ 0 & 1 & 0 & \ldots & \vdots \\ & & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 \end{bmatrix}.$$

Let $\tilde{\lambda}_1 = \frac{1}{\lambda_n}$ be the largest eigenvalue of $\tilde{\boldsymbol{P}}$ computed via **power iteration**. The required smallest root of $p(x)$ is then computed as $\frac{1}{\tilde{\lambda}_1}$.

One subtle point is that we cannot divide by a_0 if $a_0 = 0$. In fact if $a_0 = 0$, then 0 is the smallest magnitude root of $p(\cdot)$ so no power iteration is needed in that case.

(c) A possible `Julia` implementation is

```
using LinearAlgebra: norm, I

"""
    zmax, zmin = outlying_roots(a, v0, nIters)

Use power iteration to compute the largest and smallest magnitude roots,
respectively, of the polynomial defined by the input coefficients

In:
- `a` coefficient vector of length `n + 1` defining the polynomial
`p(x) = a[n+1] x^n + a[n-1] x^n-1 + ... + a[2] x + a[1]` with `a[n+1] != 0`

Option:
- `v0` vector of length `n` with initial guess of an eigenvector; default `randn(n)`
```

```
- `nIters` number of power iterations to perform; default `100`

Out:
- `zmax` root of `p(x)` with largest magnitude
- `zmin` root of `p(x)` with smallest magnitude
"""
function outlying_roots(a ; v0::AbstractVector{<:Real} = randn(length(a)-1), nIters::Int=100)

    a[end] == 0 && throw("must have a[end] != 0")

    if length(a) == 2 # n=1 and p(x) = a[2] x + a[1]
        return (-a[1]/a[2], -a[1]/a[2])
    end

    # Compute largest magnitude root
    Cmax = compan(a)
    zmax = power_iteration(Cmax, v0, nIters)[1]

    # Compute smallest magnitude root
    p_rev = reverse(a) # reverse coefficient order
    if p_rev[end] == 0
        return (zmax, 0)
    end

    Cmin = compan(p_rev)
    zmin = 1 / power_iteration(Cmin, v0, nIters)[1]

    return (zmax, zmin)
end

"""
    lambda1, v1 = power_iteration(A, v0, nIters)

Use power iteration to approximate the largest absolute eigenpair of `A`.

In:
- `A` `n × n` matrix
- `v0` vector of length `n`
- `nIters` number of iterations to perform

Out:
- `lambda1` (estimate of) the largest absolute eigenvalue of `A`
- `v1` vector of length `n` containing (an estimate of)
  the eigenvector associated with eigenvalue `lambda1`
"""
function power_iteration(A::AbstractMatrix, v0::AbstractVector, nIters::Int)
    v1 = v0 / norm(v0)
    for _ in 1:nIters
        v1 = A * v1
        v1 = v1 / norm(v1)
    end
    lambda1 = v1' * (A * v1)
    return lambda1, v1
end

# Construct a matrix whose eigenvalues equal the roots of polynomial
function compan(a::AbstractVector)
    n = length(a)
    p = reverse(a)
    return [-transpose(vec(p[2:n] / p[1])); I(n-1)[1:n-2,:]]
end
```

Pr. 2. (sol/hs021)
The **transition matrix** is

$$\boldsymbol{P} = \begin{array}{|c|ccc|} \hline & \text{Cheese} & \text{Grapes} & \text{Lettuce} \\ \hline \text{Cheese} & 0 & 4/10 & 6/10 \\ \text{Grapes} & 1/2 & 1/10 & 4/10 \\ \text{Lettuce} & 1/2 & 5/10 & 0 \\ \hline \end{array},$$

where P_{ij} denotes the probability of going from state j to state i.
Like any transition matrix: $\mathbf{1}'\boldsymbol{P} = \mathbf{1}'$.

(a) Let $\boldsymbol{\pi} = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix}$ denote the equilibrium distribution of the states (Cheese, Grapes, Lettuce). At equilibrium, $\boldsymbol{P}\boldsymbol{\pi} = \boldsymbol{\pi}$, so $\boldsymbol{\pi}$ is the eigenvector corresponding to the largest eigenvalue of $\boldsymbol{P}$ $(= 1)$.

Note that $\boldsymbol{P}\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is the all-ones column vector. Thus $\mathbf{1}$ is an eigenvector of $\boldsymbol{P}$ associated with the eigenvalue 1.

Upon normalizing so that $\mathbf{1}'\boldsymbol{\pi} = 1$, we see that the equilibrium distribution is $\boldsymbol{\pi} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$.

(b) $\boldsymbol{P}^2$ is a positive matrix, so $\boldsymbol{P}$ is a primitive matrix. The Perron-Frobenius theorem for a primitive matrix says that there is only one eigenvalue of $\boldsymbol{P}$ equal to one, so $\boldsymbol{\pi}$ is the unique Perron vector of $\boldsymbol{P}$ that sums to unity, i.e., the unique equilibrium distribution.

Pr. 3. (sol/hsj7t)

(a) Let $\boldsymbol{x} = \mathbf{1}_4/2$ and define $\boldsymbol{P} = \boldsymbol{x}\boldsymbol{x}'$. Clearly the elements of $\boldsymbol{P}$ are nonnegative, and $\mathbf{1}_4'\boldsymbol{P} = \mathbf{1}_4'\boldsymbol{x}\boldsymbol{x}' = 2\boldsymbol{x}' = \mathbf{1}_4'$ so $\boldsymbol{P}$ is a transition matrix.

Similarly $\boldsymbol{P}\boldsymbol{\pi} = \boldsymbol{x}\boldsymbol{x}'\mathbf{1}_4/4 = \boldsymbol{x}/2 = \mathbf{1}_4/4 = \boldsymbol{\pi}$ so $\boldsymbol{P}$ has the correct equilibrium distribution.

(b) Yes, it is unique. The rank-1 constraint limits use to matrices of the form $\boldsymbol{P} = \boldsymbol{x}\boldsymbol{y}'$. Because $\boldsymbol{\pi} = \mathbf{1}_4/4$ must be a right eigenvector of $\boldsymbol{P}$ with eigenvalue 1, we must have $\boldsymbol{x} \propto \mathbf{1}_4$.

And because $\mathbf{1}_4$ must be a left eigenvector of $\boldsymbol{P}$ with eigenvalue 1, we must have $\boldsymbol{y} \propto \mathbf{1}_4$.

Pr. 4. (sol/hsj73)
As shown in class, $\boldsymbol{P} \succ \mathbf{0} \Rightarrow \boldsymbol{P}^{1/2} \succ \mathbf{0}$.
Using a previous HW problem for the first step and multiplying on both sides by $\boldsymbol{P}^{1/2}$:
$2\boldsymbol{P}^{-1} \succ \boldsymbol{A}'\boldsymbol{A} \iff 2\boldsymbol{I} \succ \boldsymbol{P}^{1/2}\boldsymbol{A}'\boldsymbol{A}\boldsymbol{P}^{1/2} \iff 2\boldsymbol{I} - \boldsymbol{P}^{1/2}\boldsymbol{A}'\boldsymbol{A}\boldsymbol{P}^{1/2} \succ \mathbf{0}$
$\iff \operatorname{eig}\left(2\boldsymbol{I} - \boldsymbol{P}^{1/2}\boldsymbol{A}'\boldsymbol{A}\boldsymbol{P}^{1/2}\right) > 0 \iff 2 - \operatorname{eig}\left(\boldsymbol{P}^{1/2}\boldsymbol{A}'\boldsymbol{A}\boldsymbol{P}^{1/2}\right) > 0 \iff \operatorname{eig}\left(\boldsymbol{P}^{1/2}\boldsymbol{A}'\boldsymbol{A}\boldsymbol{P}^{1/2}\right) < 2.$

Pr. 5. (sol/hsj75)

(a) Because $\boldsymbol{H}$ is Hermitian, it has real eigenvalues that, by the **Gershgorin disk theorem** satisfy $\lambda(\boldsymbol{H}) \geq h_{ii} - \sum_{j \neq i} |h_{ij}|$, and that latter quantity is nonnegative by the assumed diagonal dominance. Thus, because all of the eigenvalues of $\boldsymbol{H}$ are nonnegative, $\boldsymbol{H}$ is positive semidefinite.

(b) Let $\boldsymbol{H} \triangleq \boldsymbol{D} - \boldsymbol{B} = \operatorname{Diag}(|\boldsymbol{B}|\,\mathbf{1}) - \boldsymbol{B}$. Then $h_{ii} = \sum_j |b_{ij}| - b_{ii} = \left(\sum_{j \neq i} |b_{ij}|\right) + (|b_{ii}| - b_{ii}) \geq \sum_{j \neq i} |b_{ij}|$, because $|b| - b \geq 0$. Also for $j \neq i$: $h_{ij} = -b_{ij}$ so $\sum_{j \neq i} |h_{ij}| = \sum_{j \neq i} |b_{ij}| \leq h_{ii}$.

Thus $\boldsymbol{H}$ is diagonally dominant so $\boldsymbol{D} - \boldsymbol{B} \succeq \mathbf{0}$, *i.e.*, $\boldsymbol{D} \succeq \boldsymbol{B}$.

Pr. 6. (sol/hsj77)

(a) The gradient is $\boldsymbol{g}(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) = \boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}) + \delta^2 \boldsymbol{x}$

(b)

$$\begin{aligned} \|\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{g}(\boldsymbol{z})\|_2 &= \left\| \left(\boldsymbol{A}'(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}) + \delta^2 \boldsymbol{x}\right) - \left(\boldsymbol{A}'(\boldsymbol{A}\boldsymbol{z} - \boldsymbol{y}) + \delta^2 \boldsymbol{z}\right) \right\|_2 \\ &= \left\| (\boldsymbol{A}'\boldsymbol{A} + \delta^2 \boldsymbol{I})(\boldsymbol{x} - \boldsymbol{z}) \right\|_2 \le \left\| \boldsymbol{A}'\boldsymbol{A} + \delta^2 \boldsymbol{I} \right\|_2 \|\boldsymbol{x} - \boldsymbol{z}\|_2, \end{aligned}$$

where

$$\left\| \boldsymbol{A}'\boldsymbol{A} + \delta^2 \boldsymbol{I} \right\|_2 \le \|\boldsymbol{A}'\boldsymbol{A}\|_2 + \left\| \delta^2 \boldsymbol{I} \right\|_2 = \|\boldsymbol{A}'\boldsymbol{A}\|_2 + \delta^2 \le \|\boldsymbol{A}'\boldsymbol{A}\|_1 + \delta^2 = \|\boldsymbol{A}'\boldsymbol{A}\|_\infty + \delta^2$$

Grader: accept either of the last two answers.
For the answer $\|\boldsymbol{A}\|_1 \|\boldsymbol{A}\|_\infty + \delta^2$, deduct two points because it is a looser upper bound as shown in part (d).

(c) When $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{I}_5 \\ \boldsymbol{1}_5 \boldsymbol{1}_5' \end{bmatrix}$ and $\delta = 2$, $\boldsymbol{A}'\boldsymbol{A} = \boldsymbol{I}_5 + 5\boldsymbol{1}_5\boldsymbol{1}_5' \Rightarrow \|\boldsymbol{A}'\boldsymbol{A}\|_1 = 1 + 25 = 26 \Rightarrow \|\boldsymbol{A}'\boldsymbol{A}\|_1 + \delta^2 = 30$

(d) In this case $\|\boldsymbol{A}'\boldsymbol{A}\|_2 = \|\boldsymbol{I}_5 + 5\boldsymbol{1}_5\boldsymbol{1}_5'\|_2 = 1 + 5\|\boldsymbol{1}_5\boldsymbol{1}_5'\|_2 = 1 + 5 \cdot 5 = 26 \Rightarrow \|\boldsymbol{A}'\boldsymbol{A}\|_1 + \delta^2 = 30$.

Here the "upper bound" is conveniently exactly the same as the Lipschitz constant itself, but this equality is not true in general!

Computing $\|\boldsymbol{A}'\boldsymbol{A}\|_1$ is much less work than $\|\boldsymbol{A}'\boldsymbol{A}\|_2$, but still would be challening for very large problems. A further uppper bound that is even easier to compute (but is a bit looser) is $\|\boldsymbol{A}'\boldsymbol{A}\|_1 + \delta^2 \le \|\boldsymbol{A}\|_\infty \|\boldsymbol{A}\|_1 + \delta^2 = 30 + 2^2 = 34$.

Pr. 7. (sol/hs023a)
The algebraic number $x_1 = -2 + \imath\sqrt{3}$ is one root of the monic polynomial

$$p_1(x) = \left(x - (-2 + \imath\sqrt{3})\right)\left(x - (-2 - \imath\sqrt{3})\right) = x^2 + 4x + 7.$$

Similarly, $x_2 = 5 - \sqrt{7}$ is one root of

$$p_2(x) = \left(x - (5 - \sqrt{7})\right)\left(x - (5 + \sqrt{7})\right) = x^2 - 10x + 18.$$

(Of course there are more complicated polynomials with these roots.) Corresponding companion matrices are:

$$\boldsymbol{P}_1 = \begin{bmatrix} -4 & -7 \\ 1 & 0 \end{bmatrix}, \qquad \boldsymbol{P}_2 = \begin{bmatrix} 10 & -18 \\ 1 & 0 \end{bmatrix}.$$

By the properties of a Kronecker sum, a matrix having eigenvalue $x_3 = x_1 + x_2$, is given by

$$\boldsymbol{P}_3 = \boldsymbol{P}_1 \oplus \boldsymbol{P}_2 = \boldsymbol{P}_1 \otimes \boldsymbol{I}_2 + \boldsymbol{I}_2 \otimes \boldsymbol{P}_2 = \begin{bmatrix} -4 & 0 & -7 & 0 \\ 0 & -4 & 0 & -7 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 10 & -18 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 10 & -18 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 6 & -18 & -7 & 0 \\ 1 & -4 & 0 & -7 \\ 1 & 0 & 10 & -18 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

The desired polynomial can be computed using $\det(x\boldsymbol{I} - \boldsymbol{P}_3)$.
Analogously, a matrix having eigenvalue $x_4 = x_1 \cdot x_2$ is given by $\boldsymbol{P}_4 = \boldsymbol{P}_1 \otimes \boldsymbol{P}_2$. The associated polynomial is given by $\det(x\boldsymbol{I} - \boldsymbol{P}_4)$.
Since $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ have integer coefficients, their Kronecker sum $\boldsymbol{P}_3$ and Kronecker product $\boldsymbol{P}_4$ will both have integer coefficients, and so will the characteristic polynomials $\det(x\boldsymbol{I} - \boldsymbol{P}_3)$ and $\det(x\boldsymbol{I} - \boldsymbol{P}_4)$.
Now the rest is just work:

$$p_3(x) = \det(x\boldsymbol{I} - \boldsymbol{P}_3) = \left| \begin{bmatrix} x - 6 & 18 & 7 & 0 \\ -1 & x + 4 & 0 & 7 \\ -1 & 0 & x - 10 & 18 \\ 0 & -1 & -1 & x \end{bmatrix} \right| = \ldots = x^4 - 12x^3 + 46x^2 - 60x + 109.$$

To avoid mistakes, use symbolic computation to help and to check the work above:

```
using AbstractAlgebra
AbstractAlgebra.charpoly(A::Matrix) = charpoly(ZZ["x"][1], matrix(ZZ,A))
P3 = [6 -18 -7 0 ; 1 -4 0 -7 ; 1 0 10 -18 ; 0 1 1 0]
p3 = charpoly(P3) # characteristic polynomial

using LinearAlgebra: I
kronsum(A,B) = kron(A, I(size(B,1))) + kron(I(size(A,1)), B)
P1 = [-4 -7; 1 0]
P2 = [10 -18; 1 0]
@assert P3 == kronsum(P1, P2) # check hand Kron. sum work

using Polynomials
q3 = Polynomial(Int.(p3.coeffs))
r3 = Polynomials.roots(q3)
x3 = (-2 + sqrt(3)*im) + (5 - sqrt(7))
@assert any≈((x3), r3) # verify that one of the roots is ≈ x3
```

Similarly for x_4, this time doing all the work in `Julia`:

```
using AbstractAlgebra
AbstractAlgebra.charpoly(A::Matrix) = charpoly(ZZ["x"][1], matrix(ZZ,A))
P4 = kron([-4 -7; 1 0], [10 -18; 1 0])
p4 = charpoly(P4) # characteristic polynomial

using Polynomials
q4 = Polynomial(Int.(p4.coeffs))
r4 = Polynomials.roots(q4)
x4 = (-2 + sqrt(3)*im) * (5 - sqrt(7))
@assert any≈((x4), r4) # verify that one of the roots is ≈ x4
```

$$p_4(x) = \det(x\boldsymbol{I} - \boldsymbol{P}_4) = \ldots = x^4 + 40x^3 + 736x^2 + 5040x + 15876.$$

The above technique can be used to prove the closure of algebraic numbers under addition and multiplication, i.e., to prove that algebraic numbers form a **field**.

Pr. 8. (sol/hs023b)
An algebraic number $a \pm \sqrt{b}$ is one root of the monic polynomial

$$p(x) = \left(x - (a + \sqrt{b})\right)\left(x - (a - \sqrt{b})\right) = x^2 - 2ax + (a^2 - b).$$

We make a corresponding companion matrix and apply the Kronecker sum and Kronecker product to get matrices having the eigenvalues with the appropriate roots, from which we construct the desired polynomials.
A possible `Julia` implementation is

```
using LinearAlgebra: I, eigvals
using Polynomials: fromroots, coeffs

"""
    g, h = poly_coeff(a, b, c, d)

Use Kronecker sums/products to construct polynomials with integer coefficients
with the given (algebraic) numbers as zeros.

In:
* `a, b, c, d` are integers

Out:
* `g` and `h` are Int vectors of length 5 with `p[5] = 1` and `q[5] = 1`
such that, in the notation defined below, `p3(x3) = 0` and `p4(x4) = 0`

Notation:
* `x1 = a + sqrt(b)`
* `x2 = c - sqrt(d)`
* `x3 = x1 + x2`
* `x4 = x1 * x2`
* `p1(x)` quadratic monic polynomial with integer-valued coefficients with a zero at `x1`
* `p2(x)` quadratic monic polynomial with integer-valued coefficients with a zero at `x2`
```

```
* `p3(x) = g[5] x^4 + g[4] x^3 + g[3] x^2 + g[2] x + g[1]` where `g[5] = 1`
* `p4(x) = h[5] x^4 + h[4] x^3 + h[3] x^2 + h[2] x + h[1]` where `h[5] = 1`
"""
function poly_coeff(a::Int, b::Int, c::Int, d::Int)

    # Construct initial polynomials from x1 and x2
    p1 = [a*a-b, -2 * a, 1] # roots at a ± sqrt(b)
    p2 = [c*c-d, -2 * c, 1] # roots at c ± sqrt(d)

    # Compute companion matrices for those monic polynomials
    compan(p) = [transpose(reverse(-p[1:end-1])); I zeros(length(p)-2)]
    C1 = compan(p1)
    C2 = compan(p2)

    # Compute Kronecker sum/product matrices
    K3 = kron(C1, I(2)) + kron(I(2), C2) # has an eigenvalue at x3 = x1 + x2
    K4 = kron(C1, C2) # has an eigenvalue at x4 = x1 * x2

    # Compute polynomial numerically
    g = poly(K3)
    h = poly(K4)

    return g, h # Int vectors
end

# Compute coefficients for polynomial whose roots are the eigenvalues of X
function poly(X)
    d = eigvals(Matrix(X)) # eigenvalues
    p = fromroots(d) # polynomial; caution: coeffs may not exactly be integers
    return round.(Int, coeffs(p))
end
```

An alternate approach (for testing) uses the `AbstractAlgebra` package:

```
using LinearAlgebra: I
using AbstractAlgebra: ZZ, matrix, det, charpoly
charpol(A::Matrix) = charpoly(ZZ["x"][1], matrix(ZZ,A))
function poly_coeff(a::Int, b::Int, c::Int, d::Int)
    A = [2a -a^2+b ; 1 0]
    B = [2c -c^2+d ; 1 0]
    As = kron(A, I(2)) + kron(I(2), B) # sum
    Ap = kron(A, B) # product
    charpolycoef = A -> Int.(charpol(A).coeffs) # char. polynomial coefficients
    return charpolycoef.((As, Ap))
end
```

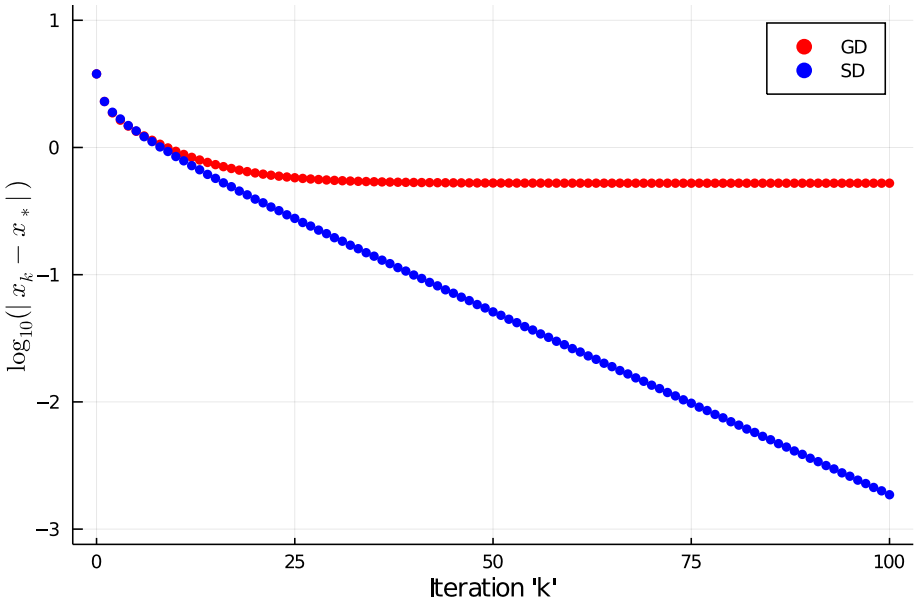
Pr. 9. (sol/hsj76)

(a) A possible `Julia` implementation is

```
using LinearAlgebra

"""
    x = lssd(A, b ; x0=zeros(size(A,2)), nIters::Int=10)

Perform steepest descent to solve the least squares problem
`\\hat{x} = \\arg\\min_x f(x), f(x) = 1/2 \\| b - A x \\|_2`

In:
* `A` : `m × n` matrix
* `b` : vector of length `m`

Option:
* `x0` : initial vector (of length `n`); default `zeros`
* `nIters` : number of iterations to perform; default `10`

Out:
* `x` a vector of length `n` containing the approximate solution
```

```
Notes:
Because this is a quadratic cost function, there is a closed-form solution
for the step size each iteration, so no "line search" procedure is needed.

A full-credit solution uses only *one* multiply by `A` and one by `A'` per iteration.
"""
function lssd(A, b ; x0=zeros(size(A,2)), nIters::Int=10)

    # Parse inputs
    b = vec(b) # make sure it is a column
    x0 = vec(x0) # ensure it is a column

    # steepest descent
    x = x0
    Ax = A * x # initialize A * x

    for _ in 1:nIters
        g = A' * (Ax - b) # gradient
        d = - g # search direction: negative gradient
        Ad = A * d
        normAd = norm(Ad)
        if normAd == 0 # done!
            return x
        else
            step = -d' * g / normAd.^2 # from previous HW
        end
        x += step * d
        Ax += step * Ad
    end

    return x
end
```

(b) The following figure shows a plot of $\log_{10}(\|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\|)$ versus iteration k for one realization. Clearly the $\boldsymbol{x}_k$ iterates are converging to $\hat{\boldsymbol{x}} = \boldsymbol{A}^+\boldsymbol{b}$, and SD converges faster than GD. (And here GD even used the optimal step size that is impractical for large-scale problems.)



Pr. 10. (sol/hs079)

(a) (Part 2)

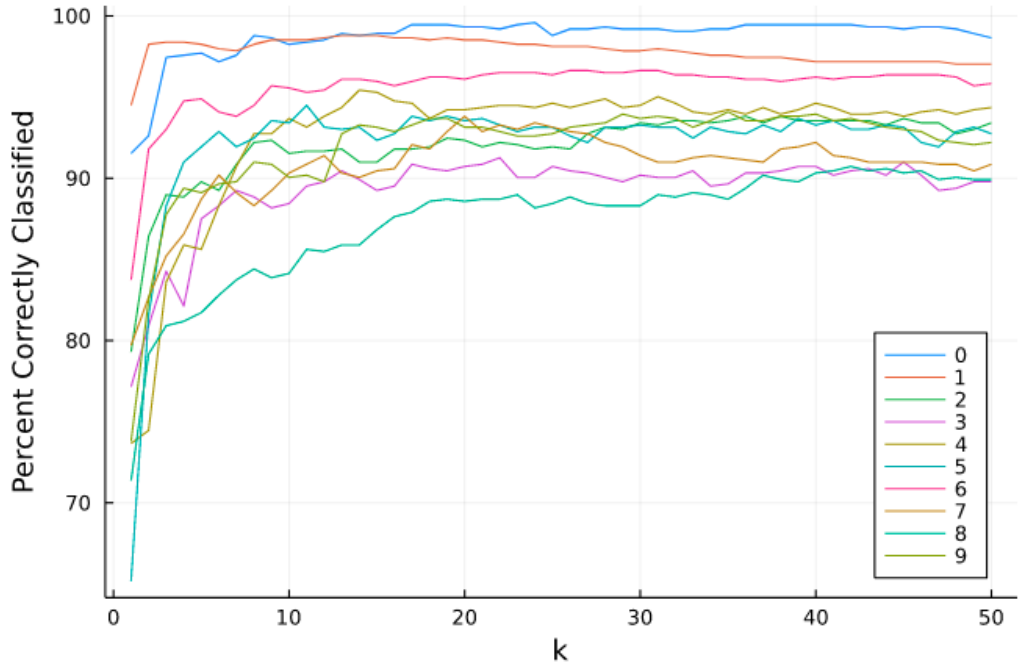
A possible `Julia` implementation is

```
using LinearAlgebra: svd

"""
    labels = classify_image(test, train, K::Int)

Classify `test` signals using `K`-dimensional subspaces
found from `train`ing data via SVD.

In:
* `test` `n x p` matrix whose columns are vectorized test images to be classified
* `train` `n x m x 10` array containing `m` training images for each digit 0-9 (in ascending order)
* `K` in `[1, min(n, m)]` is the number of singular vectors to use during classification

Out:
`labels` vector of length `p` containing the classified digits (0-9) for each test image
"""
function classify_image(test, train, K::Int)

    @assert size(test,1) == size(train,1) # check inputs
    @assert 1 ≤ K ≤ min(size(train,1),size(train,2))

    d = size(train, 3) # = 10

    # Orthonormal subspace basis of dimension K for each digit
    Us = map(i -> svd(train[:, :, i]).U[:,1:K], 1:d)

    # Calculate projection errors
    err = [vec(sum((test - U * (U' * test)).^2, dims=1)) for U in Us]
    err = hcat(err...) # p × d

    # Classify digits
    idx = findmin(err, dims=2)[2]
    labels = map(x -> x[2] - 1, vec(idx))
    return labels
end
```

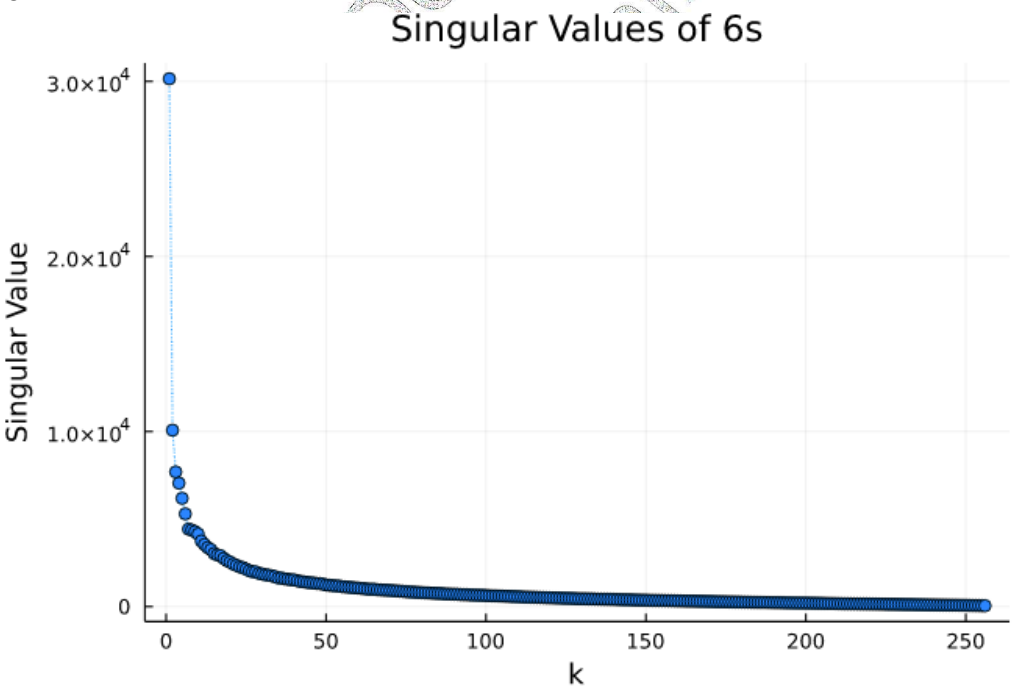
(b) (Part 3)

The following figure shows percent classification accuracy versus k for all 10 digits. Digits 0 and 1 are easy to classify and 5 seems hard. More complete analysis would generate a **confusion matrix**.

When $K = 256$, the projection matrices ideally should exactly equal the identity matrix so that in principle the residual error vector should be identically zero (for the training data). Round-off errors return numbers that are essentially random, so the classifier output becomes essentially random and performance plummets (not shown).

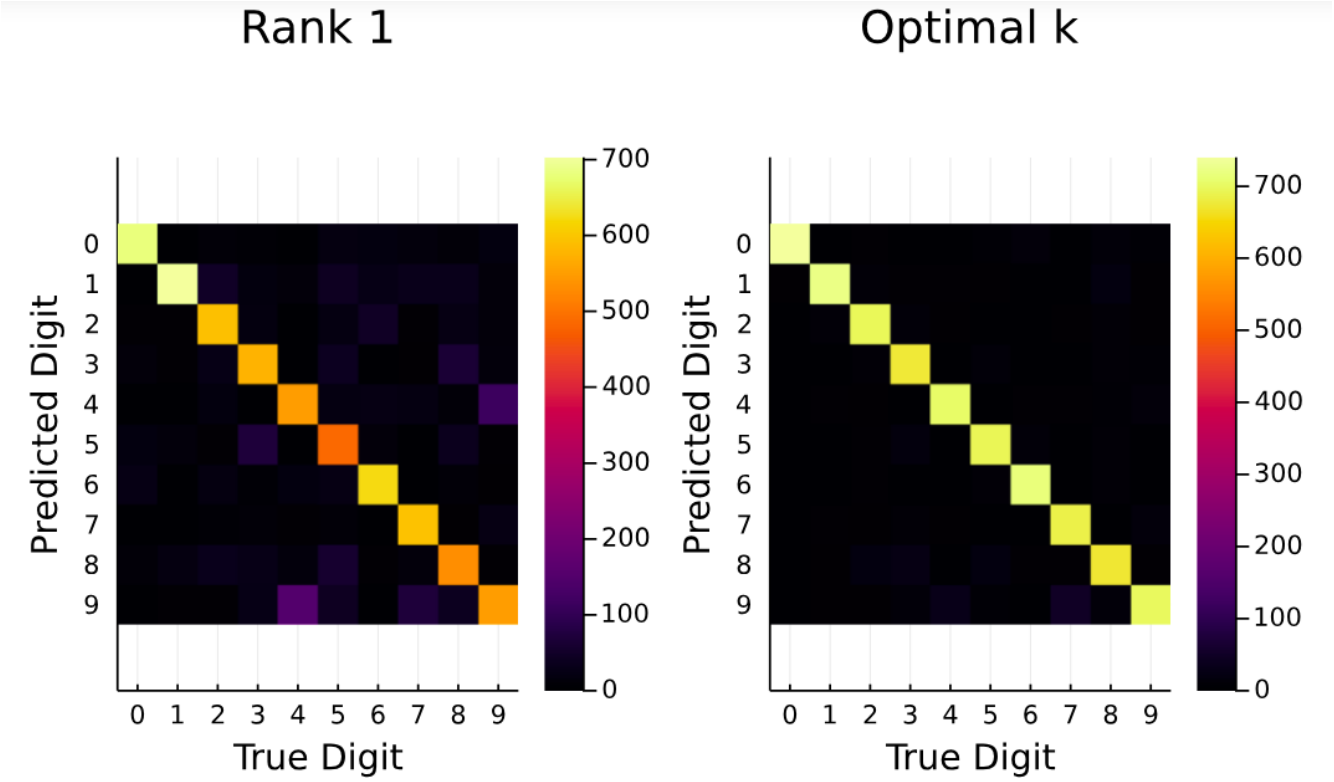


(c) (Part 4) Choosing a good value for K can greatly impact performance. The following figure shows the SVD of the training set for digit "6". It is remarkable that the optimal order (or close to optimal order) can be reasonably estimated from counting the number of singular values that look "separated". Similar observations hold for the other digits.



(d) Optional (not graded).

When using only the first left singular vector to form an orthonormal basis for a subspace of dimension = 1, re-running with the same code produces the following figure that shows the classification **confusion matrix**. Clearly, the SVD-based algorithm with rank k outperforms the rank-one classification substantially.



Non-graded problem(s) below

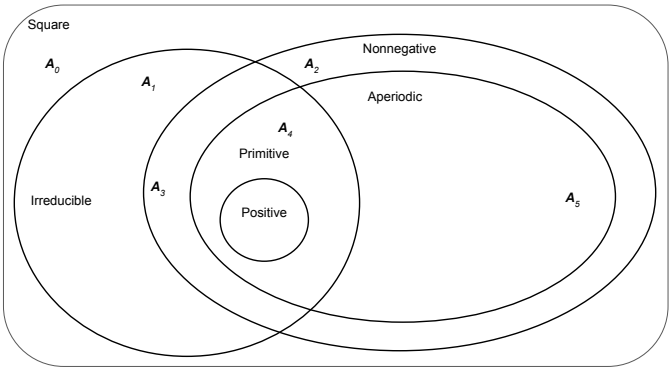
Pr. 11. (sol/hsj5p)

(a) $\boldsymbol{A} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \Rightarrow \|\boldsymbol{A}\|_2 = 1 = \|\boldsymbol{A}\|_1 .$

(b) When $\boldsymbol{A}$ is normal, $\rho(\boldsymbol{A}) = \|\boldsymbol{A}\|_2$ and $\rho(\boldsymbol{A}) \le \|\boldsymbol{A}\|$ for any induced norm so $\rho(\boldsymbol{A}) = \|\boldsymbol{A}\|_2 \le \|\boldsymbol{A}\|_1$. Thus $c = 1$.

(c) $\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \|\boldsymbol{A}\|_2 = 1 = \|\boldsymbol{A}\|_1 .$

Pr. 12. (sol/hsj7v)



Graders: you are not required to check correctness for this one.
Give credit if reasonable looking solutions are submitted.
$\boldsymbol{A}_0 = e^{\imath}$ is square, but is not nonnegative (hence not aperiodic), and is not irreducible because $e^{\imath k}$ is never real, let alone positive.

A real-valued example is $\boldsymbol{A}_0 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

$\boldsymbol{A}_1 = [-1]$ is irreducible, but not nonnegative.

$\boldsymbol{A}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ is nonnegative, its period is 2 so it is not aperiodic, and it is not irreducible.

$\boldsymbol{A}_3 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is nonnegative and irreducible, but its period is 2 so it is not aperiodic.

$\boldsymbol{A}_4 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ is primitive but it is not positive.

$\boldsymbol{A}_5 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ is aperiodic but not primitive.