HOMEWORK 4 SUBMISSION

Jinen Setpal

November 8, 2025

Checklist

1. Cross-checked independent work with Kunal Kapur.
2. No use of AI tools.
3. Code is included!

Problem 1

1.

$$\boldsymbol{E} = \begin{bmatrix} \boldsymbol{D} & \mathbf{u} \\ \mathbf{v}^T & \alpha \end{bmatrix}, \qquad \boldsymbol{E}^{-1} := \begin{bmatrix} \boldsymbol{X} & \mathbf{y} \\ \mathbf{z}^T & \beta \end{bmatrix}$$

$$\boldsymbol{E}\boldsymbol{E}^{-1} = \begin{bmatrix} \boldsymbol{D} & \mathbf{u} \\ \mathbf{v}^T & \alpha \end{bmatrix} \begin{bmatrix} \boldsymbol{X} & \mathbf{y} \\ \mathbf{z}^T & \beta \end{bmatrix} = \begin{bmatrix} \boldsymbol{D}\boldsymbol{X} + \mathbf{u}\mathbf{z}^T & \boldsymbol{D}\mathbf{y} + \mathbf{u}\beta \\ \mathbf{v}^T\boldsymbol{X} + \alpha\mathbf{z}^T & \mathbf{v}^T\mathbf{y} + \alpha\beta \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}$$

Unravelling the blocks, we get:

$$\begin{aligned} \boldsymbol{D}\boldsymbol{X} + \mathbf{u}\mathbf{z}^T &= \boldsymbol{I} \\ \mathbf{v}^T\boldsymbol{X} + \alpha\mathbf{z}^T &= \mathbf{0} \\ \boldsymbol{D}\mathbf{y} + \mathbf{u}\beta &= \mathbf{0} \\ \mathbf{v}^T\mathbf{y} + \alpha\beta &= 1 \end{aligned}$$

We can then solve both these systems of equations to get $\boldsymbol{E}^{-1}$ in close form:

$$\begin{aligned} \boldsymbol{X} &= \boldsymbol{D}^{-1}(\boldsymbol{I} - \mathbf{u}\mathbf{z}^T) \\ \mathbf{v}^T\boldsymbol{X} + \alpha\mathbf{z}^T &= \mathbf{0} \\ \mathbf{v}^T\boldsymbol{D}^{-1}(\boldsymbol{I} - \mathbf{u}\mathbf{z}^T) + \alpha\mathbf{z}^T &= \mathbf{0} \\ \mathbf{v}^T\boldsymbol{D}^{-1} - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}\mathbf{z}^T + \alpha\mathbf{z}^T &= \mathbf{0} \\ (\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u})\mathbf{z}^T &= -\mathbf{v}^T\boldsymbol{D}^{-1} \\ -\frac{\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} &= \mathbf{z}^T \end{aligned}$$

That sets $\mathbf{z}^T$.

$$\boldsymbol{X} = \boldsymbol{D}^{-1}\left(\boldsymbol{I} + \frac{\mathbf{u}\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}\right)$$

That sets $\boldsymbol{X}$.

$$\begin{aligned} \mathbf{y} &= -\boldsymbol{D}^{-1}\mathbf{u}\beta \\ \mathbf{v}^T\mathbf{y} + \alpha\beta &= 1 \\ -\mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}\beta + \alpha\beta &= 1 \\ (\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u})\beta &= 1 \\ \frac{1}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} &= \beta \end{aligned}$$

That sets β.

$$\mathbf{y} = -\frac{\boldsymbol{D}^{-1}\mathbf{u}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}$$

That sets $\mathbf{y}$.

$$\therefore \boldsymbol{E}^{-1} = \begin{bmatrix} \boldsymbol{D}^{-1}\left(\boldsymbol{I} + \frac{\mathbf{u}\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}\right) & -\frac{\boldsymbol{D}^{-1}\mathbf{u}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} \\ -\frac{\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} & \frac{1}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} \end{bmatrix}$$

$$\boldsymbol{E}^{-1}\boldsymbol{A} = \begin{bmatrix} \boldsymbol{D}^{-1}\left(\boldsymbol{I} + \frac{\mathbf{u}\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}\right) & -\frac{\boldsymbol{D}^{-1}\mathbf{u}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} \\ -\frac{\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} & \frac{1}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}} \end{bmatrix} \begin{bmatrix} \boldsymbol{A}_1 & \mathbf{a}_2 \\ \mathbf{a}_3^T & a_4 \end{bmatrix}$$

Decomposing the algorithm, $\mathbf{v}^T\boldsymbol{D}^{-1},\ \boldsymbol{D}^{-1}\mathbf{u} \in \mathcal{O}(n)$, this is just a series of independent multiplications and divisions, $\mathbf{v}^T(\boldsymbol{D}^{-1}\mathbf{u})$ is just the inner product $\in \mathcal{O}(n)$ taking the previous result (i.e., never compute a matrix). Thus the bottom-left block, top-right block, and bottom-right block can be computed in $\mathcal{O}(n)$. For the top-left block, we have:

$$\boldsymbol{D}^{-1}\left(\boldsymbol{I} + \frac{\mathbf{u}\mathbf{v}^T\boldsymbol{D}^{-1}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}\right)\boldsymbol{A}_1 = \boldsymbol{D}^{-1}\left(\boldsymbol{A}_1 + \frac{\mathbf{u}\mathbf{v}^T\boldsymbol{D}^{-1}\boldsymbol{A}_1}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}\right) = \underbrace{\overbrace{\boldsymbol{D}^{-1}\boldsymbol{A}_1}^{\mathcal{O}(nc)} + \frac{\overbrace{\overbrace{\boldsymbol{D}^{-1}\mathbf{u}}^{\mathcal{O}(n)}\overbrace{\mathbf{v}^T\boldsymbol{D}^{-1}\boldsymbol{A}_1}^{\mathcal{O}(n+nc)}}^{\mathcal{O}(nc)}}{\alpha - \mathbf{v}^T\boldsymbol{D}^{-1}\mathbf{u}}}_{\mathcal{O}(nc)}$$

The first summand is straightforward, given it's a diagonal matrix we are just scaling each row of $\boldsymbol{A}_1$ by some scalar, we need only one pass over the entire matrix. The first inner product has been justified prior, the next terms are an inner product follower by a vector-matrix product. Then another matrix-vector product to top-off the summand. Finally, both summands are matrices and elementwise addition is another $\mathcal{O}(nc)$ operation. Thus our final FLOP count for this computation is $\mathcal{O}(nc)$.

2. Let $\boldsymbol{A},\ \boldsymbol{B} \in \mathbb{R}^{m\times n}$. Then $\boldsymbol{A}^T\boldsymbol{B} \in \mathbb{R}^{n\times n}$. We have that $(\boldsymbol{A}^T\boldsymbol{B})_{i,j} = \langle \boldsymbol{A}_{i,:}, \boldsymbol{B}_{j,:}\rangle$ for some indices i, j. By definition, $\text{trace}(\boldsymbol{A}^T\boldsymbol{B}) = \sum_{i=1}^{n}\langle \boldsymbol{A}_{i,:}, \boldsymbol{B}_{i,:}\rangle$.

   We can speed up the computation of the inner product by first doing a single scalar multiplication, then running the fused operation until we complete the inner product.

$$\mathbf{x}^T\mathbf{y} = \underbrace{\underbrace{\mathbf{x}_1\mathbf{y}_1}_{\text{ops+=1}} + \mathbf{x}_2\mathbf{y}_2}_{\text{ops+=1}} + \cdots + \mathbf{x}_n\mathbf{y}_n$$

   So we need n ops to compute a single inner product, need n different inner products and reduce them using $n-1$ additions. Thus the total ops $= n^2 + n - 1$.

Problem 2

1. IEEE floating point comes with the following useful guarantees:

   a. $x \oplus y = (x + y)(1 + \delta)$

   b. $x \otimes y = (x + \delta x)(y + \delta y)$

   Where $\oplus$ and $\otimes$ represent computation addition and multiplication respectively.

   At an intuitive level, $\mathbf{x}^T\mathbf{y}$ the inner product is just a series of multiplications and additions. Although the overall accuracy of our computation may deviate as operations increase $\in \mathcal{O}(n)$, it is close at a per operation level, and we can bound our constant as a function of n to bound deviation. Further, the dot product function has a range of real numbers so even if we are off by a lot we can argue that the function output is the dot product of *some* $\bar{\mathbf{x}}, \bar{\mathbf{y}}$, and consequently that the naive implementation is backwards stable.

   A formal proof proceeds by induction.

   Base Case. Let $n = 1$:

   $$\mathbf{x}^T\mathbf{y} = \mathbf{x}_1 \otimes \mathbf{y}_1 = (\mathbf{x}_1 + \delta\mathbf{x}_1)(\mathbf{y}_1 + \delta\mathbf{y}_1)$$

   This is backwards stable on account of being floating point multiplication.

   Inductive Hypothesis. $\mathbf{x}^T\mathbf{y}$ where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ for $n = k$ is backwards stable.

   $$\mathbf{x}^T\mathbf{y} = (\mathbf{x}(1+\delta))^T(\mathbf{y}(1+\delta)) = \mathbf{x}^T\mathbf{y} + 2\delta\mathbf{x}^T\mathbf{y} + \delta^2\mathbf{x}^T\mathbf{y} = \mathbf{x}^T\mathbf{y}(1 + 2\delta + \delta^2)$$

   Inductive Step. For $n = k + 1$, we have:

   $$\begin{aligned}
   \mathbf{x}^T\mathbf{y} &= \sum_{i=1}^{k+1} \mathbf{x}_i \otimes \mathbf{y}_i = (\mathbf{x}_{k+1} \otimes \mathbf{y}_{k+1}) \oplus \sum_{i=1}^{k} \mathbf{x}_i \otimes \mathbf{y}_i \\
   &= [\mathbf{x}_{k+1}(1+\delta)\mathbf{y}_{k+1}(1+\delta)] \oplus (\mathbf{x}_{1:k}(1+\delta))^T(\mathbf{y}_{1:k}(1+\delta)) \\
   &= \left[[\mathbf{x}_{k+1}(1+\delta)\mathbf{y}_{k+1}(1+\delta)] + (\mathbf{x}_{1:k}(1+\delta))^T(\mathbf{y}_{1:k}(1+\delta))\right](1+\delta) \\
   &= \left[\mathbf{x}_{k+1}\mathbf{y}_{k+1} + 2\delta\mathbf{x}_{k+1}\mathbf{y}_{k+1} + \delta^2\mathbf{x}_{k+1}\mathbf{y}_{k+1} + \mathbf{x}_{:k}^T\mathbf{y}_{:k} + 2\delta\mathbf{x}_{:k}^T\mathbf{y}_{:k} + \delta^2\mathbf{x}_{:k}^T\mathbf{y}_{:k}\right](1+\delta) \\
   &= \left[\mathbf{x}^T\mathbf{y} + 2\delta\mathbf{x}^T\mathbf{y} + \delta^2\mathbf{x}^T\mathbf{y}\right](1+\delta) \\
   &= \mathbf{x}^T\mathbf{y} + 2\delta\mathbf{x}^T\mathbf{y} + \delta^2\mathbf{x}^T\mathbf{y} + \delta\mathbf{x}^T\mathbf{y} + 2\delta^2\mathbf{x}^T\mathbf{y} + \delta^3\mathbf{x}^T\mathbf{y} \\
   &= \mathbf{x}^T\mathbf{y} + 3\delta\mathbf{x}^T\mathbf{y} + 3\delta^2\mathbf{x}^T\mathbf{y} + \delta^3\mathbf{x}^T\mathbf{y} \\
   &= \mathbf{x}^T\mathbf{y}(1 + 3\delta + 3\delta^2 + \delta^3) \\
   &= \mathbf{x}^T\mathbf{y}(1 + 2\delta + \delta^2 + \delta + 2\delta^2 + \delta^3) \\
   &= \mathbf{x}^T\mathbf{y}(1 + 2\delta + \delta^2) + \delta\mathbf{x}^T\mathbf{y}(1 + 2\delta + \delta^2) \\
   &= (\mathbf{x}^T + \delta\mathbf{x}^T)(\mathbf{y} + \delta\mathbf{y}) + \delta(\mathbf{x}^T + \delta\mathbf{x}^T)(\mathbf{y} + \delta\mathbf{y}) \\
   &= (\mathbf{x}^T + \delta\mathbf{x}^T)(\mathbf{y} + \delta\mathbf{y})(1 + \delta)
   \end{aligned}$$

2. Computing $\mathbf{y} = \boldsymbol{A}\mathbf{x}$ is equivalent to computing dot products of $\boldsymbol{A}_{i,:}\mathbf{x}\ \forall i$ and stacking the results together to form a matrix. This concatenation of dot products doesn't incur any precision error (we are just moving data, no operations) therefore the backward stability guarantee from inner products above also holds for matrix-vector products.

Problem 3

1. We have the following algorithm:

```
function mysum(x::Vector{Float64})
        s = zero(Float64)
        for i=1:length(x)
                s += x[i]
        end
        return s
end
```

   This algorithm is *exactly* equivalent to:

```
function mysum(x::Vector{Float64})
        s = zero(Float64)
        for i=1:length(x)
                s += 1 .* x[i]  # line change made here
        end
        return s
end
```

   By basically adding a no-op, we can see that `mysum` is just a special case of dot product – specifically, $\texttt{mysum}(\mathbf{x}) = \mathbf{x}^T\mathbf{e}$. We know that the naive implementation of dot product is backwards stable from question 2; therefore this algorithm is also backwards stable. This naturally implies that the desired condition $\|\hat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\| \leq C_n\varepsilon$ is satisifed.

2. We can start by unpacking numerical error:

$$\begin{aligned} s &= a \oplus b \oplus c \\ &= [(a+b)(1+\delta)] \oplus c \\ &= [a+b+\delta a+\delta b] \oplus c \\ &= (a+b+\delta a+\delta b+c)(1+\delta) \\ &= a+b+\delta a+\delta b+c+\delta a+\delta b+\delta^2 a+\delta^2 b+\delta c \\ &= a+b+c+\underbrace{2\delta a+2\delta b+\delta^2 a+\delta^2 b+\delta c}_{\text{error}} \end{aligned}$$

   We want to minimize error; W.L.O.G. let $c \geq a, b$, then $2\delta a, 2\delta b$ the largest possible scaling of the error term will be minimized. So the ideal permutation is to pick the two smallest numbers and sum them up, and then sum the remaining number.

3. The answer to this is pretty much eluded in the point above; for arbitrary $\mathbf{x}$ we just sort in ascending order and then sum from left to right. We'll test on arithmetic mean, because we can compute the exact answer in closed form.

   Code

```
function mystablesum(x::Vector{Float64})
        x = sort(x)
        s = zero(Float64)
        for i=1:length(x)
                s += x[i]
        end
        return s
end

function oracle(x::Vector{Float64})  # test on arithmetic series
        return length(x) * (minimum(x) + maximum(x)) / 2
end
```

```
x = Vector{Float64}(1e9:-1e1:1)
println("Original: ", mysum(x))
println("Permuted: ", mystablesum(x))
println("Oracle: ", oracle(x))
```

Output

```
Original: 5.000000057998199e16
Permuted: 5.000000046002399e16
Oracle: 5.00000005e16
```

There is a difference, the original is indeed worse than our permuted but on a relative scale, it's not by much.

4. Thank you Wikipedia for the wonderful reference:

Code

```
function kahansum(x::Vector{Float64})
    s = 0.0
    c = 0.0

        for i=1:length(x)
                y = x[i] - c
                t = s + y
                c = (t - s) - y
                s = t
        end

    return s
end

x = Vector{Float64}(1e9:-1e1:1)
println("Original: ", mysum(x))
println("Permuted: ", mystablesum(x))
println("Kahan: ", kahansum(x))
println("Oracle: ", oracle(x))
```

Output

```
Original: 5.000000057998199e16
Permuted: 5.000000046002399e16
Kahan: 5.00000005e16
Oracle: 5.00000005e16
```

Kahan summation wins and by some margin!

Problem 4

The overall idea is to implement Bisection, Citardauq and the approach suggested by Lumbric-Vonbrand (S.O. usernames). Since subtraction of close numbers is ill-conditioned we want can find extreme examples when $4ac \approx 0$; we set $a = c \approx 0$ to achieve this and compare the accuracy of the three approaches.

Code

```
function f(x::Float32, a::Float32, b::Float32, c::Float32)
        return a*x^2 + b*x + c
end

function roots(a::Float32, b::Float32, c::Float32)
        if a > 0
                # then it's convex, f'(x') = 0, f(x') should be negative
                if f(-b/(2a), a, b, c) > 0
                        return (NaN, NaN)
                end

                x_pos = -b/(2a) + maximum([abs(a), abs(1/a), abs(b), abs(1/b)
                    , abs(c), abs(1/c)])
                x_neg = -b/(2a)
        elseif a == 0  # linear function
                return (-c/b, NaN)
        elseif a < 0
                # then it's concave, f'(x') = 0, f(x') should be positive
                if f(-b/(2a), a, b, c) < 0
                        return (NaN, NaN)
                end

                x_neg = -b/(2a) + maximum([abs(a), abs(1/a), abs(b), abs(1/b)
                    , abs(c), abs(1/c)])
                x_pos = -b/(2a)
        end

        tol = 1e-10
        max_iter = 100000

        iter = 1
        root = [NaN, NaN]
        while iter < max_iter
                midpoint = (x_pos + x_neg) / 2
                f_mid = f(midpoint, a, b, c)

                if abs(f_mid) < tol
                        root[1] = midpoint
                elseif sign(f_mid) == 1
                        x_pos = midpoint
                elseif sign(f_mid) == -1
                        x_neg = midpoint
                end

                iter += 1
        end

        if !isnan(root[1])
                root[2] = -b/a - root[1]
        end
```

```
        return root
end

function citardauq(a::Float32, b::Float32, c::Float32)
        return (2c/(-b + sqrt(b^2 - 4a*c)), 2c/(-b - sqrt(b^2 - 4a*c)))
end

function lumbric_vonbrand(a::Float32, b::Float32, c::Float32)
        if b < 0
                r1 = (- b + sqrt(b^2 - 4a*c)) / 2a
        elseif b > 0
                r1 = (- b - sqrt(b^2 - 4a*c)) / 2a
        else
                return (sqrt(c/a), -sqrt(c/a))
        end

        r2 = - b/a - r1
        return (r1, r2)
end

# to get extreme examples, we need 4ac \approx = 0, because then b - b \
   approx 0 will incur floating point errors
a = Float32(1e-4)
b = Float32(10 * randn(1)[1])
c = Float32(1e-4)

sol = lumbric_vonbrand(a, b, c)
@show(sol)
@show(f(sol[1], a, b, c))
@show(f(sol[2], a, b, c))

sol = citardauq(a, b, c)
@show(sol)
@show(f(sol[1], a, b, c))
@show(f(sol[2], a, b, c))

sol = roots(a, b, c)
@show(sol)
@show(f(sol[1], a, b, c))
@show(f(sol[2], a, b, c))
```

Output

```
sol = (8607.079f0, 0.0f0)
f(sol[1], a, b, c) = 0.0001f0
f(sol[2], a, b, c) = 0.0001f0
sol = (0.00011618344f0, 3355.443f0)
f(sol[1], a, b, c) = 0.0f0
f(sol[2], a, b, c) = -1762.1565f0
sol = [NaN, NaN]
```

We can see that the first approach that minimizes catastrophic cancelling does well, especially compared to the other approaches where the well-conditioned computations do well (first root of Citardauq), but ill-conditioned computations get `NaN` (second root of Citardauq and Bisection).

Problem 5

1. $H(\mathbf{p}) : (0,1)^n \to \mathbb{R}_+$ is differentiable, so we can compute the condition number as:

$$\kappa(\mathbf{p}; H) = \frac{\|\nabla_{\mathbf{p}} H(\mathbf{p})\|}{H(\mathbf{p})} \|\mathbf{p}\|_1$$

Since $\mathbf{p}$ is a probability distribution, we have that $\mathbf{p}\mathbf{e}^T = 1$, so $\|\mathbf{p}\|_1 = 1$. We then have:

$$\kappa(\mathbf{p}; H) = \frac{\|\nabla_{\mathbf{p}} H(\mathbf{p})\|}{H(\mathbf{p})}$$

We can compute $\nabla_{\mathbf{p}} H(\mathbf{p})$:

$$\nabla_{\mathbf{p}} H(\mathbf{p}) = -\nabla_{\mathbf{p}} \sum_{i=1}^{n} p_i \log p_i = - \begin{bmatrix} \frac{\partial p_1 \log p_1}{\partial p_1} + 0 \\ \frac{\partial p_2 \log p_2}{\partial p_2} + 0 \\ \vdots \\ \frac{\partial p_n \log p_n}{\partial p_n} + 0 \end{bmatrix} = - \begin{bmatrix} 1 + \log p_1 \\ 1 + \log p_2 \\ \vdots \\ 1 + \log p_n \end{bmatrix} = -(1 + \log(\mathbf{p}))$$

Which we can plug into the condition number:

$$\kappa(\mathbf{p}; H) = \frac{\|1 + \log(\mathbf{p})\|}{-\sum_{i=1}^{n} p_i \log p_i}$$

Consider input $\mathbf{p} = \begin{bmatrix} \epsilon & 1-\epsilon & 0 & 0 & \cdots & 0 \end{bmatrix}$. $\epsilon \to 0 \implies H(\mathbf{p}) \to 0$, $\|1 + \log(\mathbf{p})\| \to \infty$, thus $\epsilon \to 0 \implies \kappa(\mathbf{p}; H) \to \infty$; for such input, entropy is ill-conditioned.

2. $f(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b} = \sum_{i=1}^{n} \mathbf{a}_i \mathbf{b}_i$ is the dot-product function. We note that $f(\boldsymbol{A}, \mathbf{x})$ is just a series of dot products applied for each vector partition of $\boldsymbol{A}$. So, to obtain the most ill-conditioned input for matrix-vector multiplication, we can find the worst-case dot product and just replicate one of the vectors to form worst-case matrix $\boldsymbol{A}$ and set the other vector as $\mathbf{x}$.

Case 1. $n = 1$. Then, $\boldsymbol{A} \in \mathbb{R}^{1 \times m}$, $\mathbf{x} \in \mathbb{R}^{1 \times 1}$.
This is just a set of scalar multiplications which have relative condition number 1; this case is well-conditioned.

Case 2. $n > 1$.
Now, the dot-product function has summands, which is ill-conditioned when $\mathbf{a}_i \mathbf{b}_i = -\mathbf{a}_j \mathbf{b}_j + \epsilon$ for vectors $\mathbf{a}$, $\mathbf{b}$ and indices i, j. W.L.O.G., assume summands are merged with ordering $\mathcal{I}$. Then, given arbitrary running sum $\bar{\mathbf{s}}_i$, and arbitrary (e.g., generated at random) summand $\mathbf{a}_{i+1}$, set $\mathbf{b}_{i+1} = \frac{-\bar{\mathbf{s}}_i + \epsilon}{\mathbf{a}_i}$. Thus we have constructed an ill-conditioned dot product between vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{n \times 1}$. Now:

$$\text{Let } \boldsymbol{A} := \begin{bmatrix} \mathbf{a} & \mathbf{a} & \cdots & \mathbf{a} \end{bmatrix}, \quad \mathbf{x} = \mathbf{b}$$

Matrix-vector product is ill-conditioned for such input.

3. Since orthogonal matrices are guaranteed to be invertible, given some $\boldsymbol{Q}$, we can obtain an arbitrary pre-activation output $\mathbf{h}$ by computing the corresponding input as $\mathbf{x} = \boldsymbol{Q}\mathbf{h}$. To simplify the computation, W.L.O.G., let $\boldsymbol{Q} = \boldsymbol{I}$ and $\mathbf{x} \in \mathbb{R}^n$. Thus we only really have to compute the conditioning of softplus:

$$\begin{aligned} \kappa(f; \mathbf{x}, \boldsymbol{Q}) = \kappa(f; \mathbf{h}) &= \frac{\|\nabla_{\mathbf{h}} f(\mathbf{h})\|}{\|f(\mathbf{h})\|} \|\mathbf{h}\| \\ &= \frac{\|\nabla_{\mathbf{h}} \log(1 + \exp(\mathbf{h}))\|}{\|\log(1 + \exp(\mathbf{h}))\|} \|\mathbf{h}\| \\ &= \frac{\|\frac{\exp(\mathbf{h})}{1 + \exp(\mathbf{h})}\|}{\|\log(1 + \exp(\mathbf{h}))\|} \|\mathbf{h}\| \end{aligned}$$

The Jacobian of softplus is just sigmoid, which is elementwise bounded at 1:

$$\leq \frac{\|\mathbf{e}\|}{\|\log(1 + \exp(\mathbf{h}))\|} \|\mathbf{h}\|$$

$$= \frac{n}{\|\log(1+\exp(\mathbf{h}))\|}\|\mathbf{h}\|$$

Next, we lower bound the denominator to upper bound the overall term:

$$\begin{aligned} &\leq \frac{n}{\|\log(\exp(\mathbf{h}))\|}\|\mathbf{h}\| \\ &= \frac{n}{\|\mathbf{h}\|}\|\mathbf{h}\| = n \\ \therefore \kappa(f;\mathbf{x},\boldsymbol{Q}) &\leq n \end{aligned}$$

Thus $\mathbf{y} = f(\boldsymbol{Q}^T\mathbf{x})$ is well-conditioned for small enough n.

Problem 6

Since SVD has reduced and full forms, I'll conveniently formalize the intuitive version first and then decompose the given matrices. We have:

$$\boldsymbol{A} \in \mathbb{R}^{n\times m}, \quad \boldsymbol{A}\boldsymbol{A}^T \in \mathbb{R}^{n\times n}, \quad \boldsymbol{A}^T\boldsymbol{A} \in \mathbb{R}^{m\times m}$$
$$\boldsymbol{A}\boldsymbol{A}^T := \boldsymbol{S}_L = \underbrace{\boldsymbol{U}}_{\in\mathbb{R}^{n\times n}} \Sigma^2 \boldsymbol{U}^T, \quad \boldsymbol{A}^T\boldsymbol{A} := \boldsymbol{S}_R = \underbrace{\boldsymbol{V}}_{\in\mathbb{R}^{m\times m}} \Sigma^2 \boldsymbol{V}^T, \quad \boldsymbol{A} = \boldsymbol{U} \underbrace{\Sigma}_{\in\mathbb{R}^{n\times m}} \boldsymbol{V}^T$$

If $n > m$, there are more rows than columns, the leftover rows contain all zero elements. Conversely, if $n < m$ the leftover columns contain all zero elements. Finally, if $n = m$, then it is exactly a diagonal matrix so we don't have any extra rows or columns to account for.

1. I notice that we have an almost diagonal matrix to start with; we only have to permute columns. Luckily permutation matrices are orthogonal. This gives us the result already and since the identitiy matrix is also orthogonal that can be $\boldsymbol{U}$ and we are done.

$$\boldsymbol{A} = \begin{bmatrix} 0 & -3 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -3 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

2. For this one, the right singular vectors are low-rank so let's start there:

$$\boldsymbol{A} = \begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix}, \quad \boldsymbol{A}^T\boldsymbol{A} = \begin{bmatrix} -5 & 2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 29 & 0 \\ 0 & 0 \end{bmatrix} = \boldsymbol{I} \begin{bmatrix} \sqrt{29} & 0 \\ 0 & 0 \end{bmatrix}^2 \boldsymbol{I}$$

So we have $\Sigma\boldsymbol{V}^T$ which is great, we can use this to figure out the remaining piece of the puzzle:

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{U}\Sigma\boldsymbol{V}^T \\
\begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix} &= \boldsymbol{U} \begin{bmatrix} \sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{I} \\
\begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} &= \boldsymbol{U} \begin{bmatrix} \sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{I} \begin{bmatrix} 1/\sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} \\
\begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} &= \boldsymbol{U} \\
\begin{bmatrix} -5/\sqrt{29} & 0 \\ 2/\sqrt{29} & 0 \end{bmatrix} &= \boldsymbol{U}
\end{aligned}$$

This is nice, but we are not done yet, because the second column is not a valid eigenvector. Since $\Sigma_{:,2}$ is also a zero vector, we are basically free to set this as anything provided it's orthogonal to our first eigenvector, and has a magnitude of one. Accordingly, we have:

$$\begin{aligned}
\begin{bmatrix} -5/\sqrt{29} & 2/\sqrt{29} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} &= 0 \\
\frac{-5}{\sqrt{29}}a + \frac{2}{\sqrt{29}}b &= 0 \\
\frac{2}{\sqrt{29}}b &= \frac{5}{\sqrt{29}}a
\end{aligned}$$

Let $a = 1$, then we have:

$$b = \frac{5}{2}$$

$\| \begin{bmatrix} 1 & 5/2 \end{bmatrix} \|_2 = \sqrt{1 + 25/4} = \sqrt{29}/2$ so we divide a and b to obtain the normalized eigenvector:

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{U}\Sigma\boldsymbol{V}^T \\
\begin{bmatrix} -5 & 0 \\ 2 & 0 \end{bmatrix} &= \begin{bmatrix} -5/\sqrt{29} & 5/\sqrt{29} \\ 2/\sqrt{29} & 2/\sqrt{29} \end{bmatrix} \begin{bmatrix} \sqrt{29} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

This gives us the SVD for $\boldsymbol{A}$.

3. This is a rank 1 matrix, so we should be able to construct this matrix as $\sigma \mathbf{u}\mathbf{v}^T$.

$$\boldsymbol{A} = \begin{bmatrix} 1 & -2 \\ 2 & -4 \\ 0 & 0 \end{bmatrix}$$

$$\text{Let } \bar{\mathbf{u}} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \quad \bar{\mathbf{v}}^T = \begin{bmatrix} 1 & -2 \end{bmatrix}$$

Basically, we just encode the fact that the second row is twice the first and the third row is 0 times the first. Now, we can normalize these two and set the scaling factor as $\sigma := \alpha\gamma$, that gets us the SVD.

$$\bar{\mathbf{u}} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} = \sqrt{5} \begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \\ 0 \end{bmatrix} = \alpha \mathbf{u}, \quad \bar{\mathbf{v}}^T = \begin{bmatrix} 1 & -2 \end{bmatrix} = \sqrt{5} \begin{bmatrix} 1/\sqrt{5} & -2/\sqrt{5} \end{bmatrix} = \gamma \mathbf{v}^T$$

$$\therefore \boldsymbol{A} = \sigma \mathbf{u}\mathbf{v}^T = 5 \begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \\ 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{5} & -2/\sqrt{5} \end{bmatrix}$$

4. Diagonal matrices are super convenient.

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Problem 7

1. $\tilde{f}(x) = f(x+\delta x)$ where $\delta x \leq C\mu x$ for scalar x is our notion of backward stability. `myf`$(x) = \sqrt{x+\mu} = \tilde{f}(x+\mu)$ for $f(x) = \sqrt{x}$ exactly satisfies this definition, so `myf` is backwards stable.
2. $\mathcal{X} = \{x \in \mathbb{R}_+ : \sqrt{x} = -10^{16}\mu\} = \emptyset$ therefore `mysqrt` cannot be backwards stable.

Problem 8

We have that:

$$\frac{1}{\kappa(\boldsymbol{A})} = \frac{1}{\|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|} = \min_{\boldsymbol{D}} \left\{ \frac{\|\boldsymbol{D}\|}{\|\boldsymbol{A}\|} : \det(\boldsymbol{A}+\boldsymbol{D}) = 0 \right\} \iff \frac{1}{\|\boldsymbol{A}^{-1}\|} = \|\boldsymbol{D}\|$$

Where $\boldsymbol{D}$ is selected such that it satisfies the required optimization problem.

First, let's look at the SVD of $\boldsymbol{A}$:

$$\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^T \implies \boldsymbol{A}^{-1} = \boldsymbol{V}\Sigma^{-1}\boldsymbol{U}^T$$

For some diagonal matrix $\boldsymbol{B}$, $\det(\boldsymbol{B}) = \prod_{i=1}^{n} \boldsymbol{B}_{i,i}$, so the shortest distance to a singular matrix is obtained simply by negating the smallest singular value of $\boldsymbol{A}$.

$$\text{Let } \boldsymbol{D} := \boldsymbol{U}\Sigma'\boldsymbol{V}^T \qquad \text{where} \qquad \Sigma' = -\sigma_{\min}\boldsymbol{E}_{n,n}$$
$$\text{Then, } \det(\boldsymbol{A}+\boldsymbol{D}) = \det(\boldsymbol{U}(\Sigma+\Sigma')\boldsymbol{V}^T) = \det(\boldsymbol{U})\det(\Sigma+\Sigma')\det(\boldsymbol{V}^T) = \pm 1 \cdot 0 \cdot \pm 1 = 0$$

Since the matrix 2 norm gives us the largest singular value, and our construction of $\boldsymbol{D}$ has just one, we have that:

$$\|\boldsymbol{D}\| = \|\boldsymbol{U}\Sigma'\boldsymbol{V}^T\| = \|\Sigma'\| = |\Sigma'_{n,n}| = \sigma_{\min}$$

Σ^{-1} just inverts each singular value, so the largest singular value of $\boldsymbol{A}^{-1}$ is the *smallest* singular value of $\boldsymbol{A}$:

$$\|\boldsymbol{A}^{-1}\| = \|\boldsymbol{V}\Sigma^{-1}\boldsymbol{U}^T\| = \|\Sigma^{-1}\| = \left|\frac{1}{\Sigma_{n,n}}\right| = \frac{1}{\sigma_{\min}} = \frac{1}{\|\boldsymbol{D}\|}$$

This proves the desired statement.

Problem 9

1. It was interesting to see that $\mathbf{s_{text},}_i > \mathbf{s_{random},}_i \; \forall i \in \{0, 1, \ldots, d\}$. I tried this multiple times to confirm it wasn't a fluke. In each case, the graph looked about the same, particularly in terms of the double-dropoff. The first drop looks smoother, but is in fact quite a bit more pronounced given it is log-scale. The second one is just the rank of the system, given the singular values in both instances fall to about zero towards the end. $\mathbf{s_{random}}$ appears to have lower rank than $\mathbf{s_{text}}$, and accordingly the singular values reach zero earlier. At a broader level though, I am not quite sure what comparing the singular values of the transformation induced by the resultant matrix is indicative of.

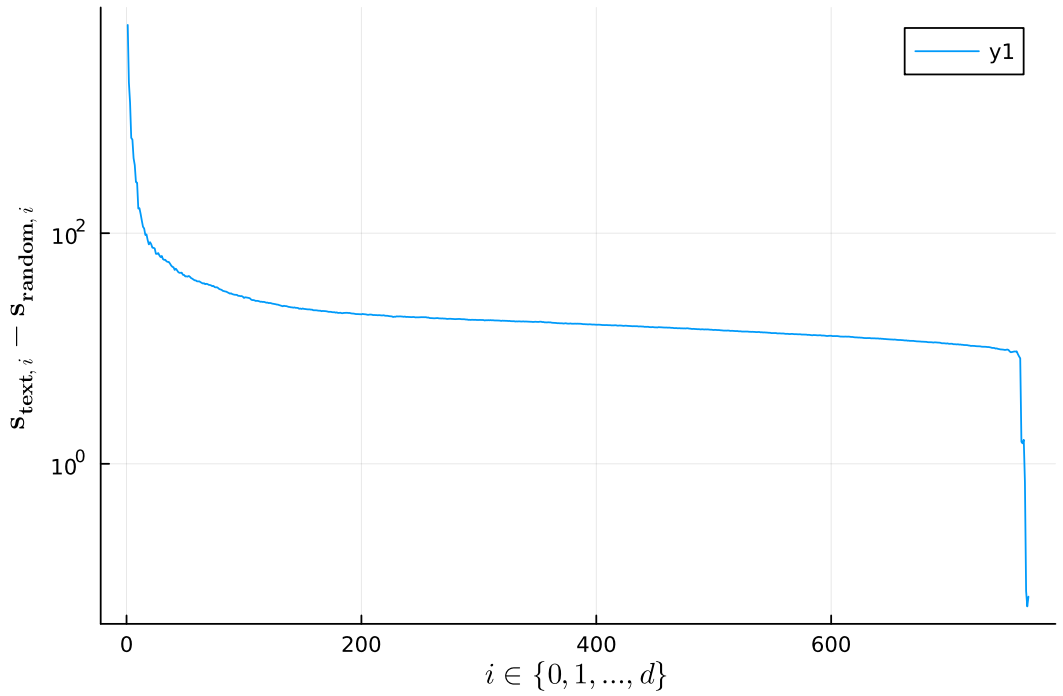
   Code

```
tokens = read_token_sequence()
Y = reduce(vcat, [gpt2func(tokens[i], gpt2model) for i in 1:100])
s_Y = svdvals(Y)

n = size(Y)[1]
Z = reduce(vcat, [gpt2func(rand(0:50256, length(tokens[i])), gpt2model)
    for i in 1:100])
s_Z = svdvals(Z)

p = Plots.plot(s_Y - s_Z, yaxis=:log, ylabel=L"\mathbf{s}_{\mathbf{text},
    i} - \mathbf{s}_{\mathbf{random}, i}", xlabel=L"i \in \{0, 1, \ldots,
    d\}")
```

   Output



2. The first challenges arises when actually attempting to obtain $\boldsymbol{Y}$ for all 10,000 sequences. Since the GPT-2 implementation is CPU-only, it takes a long time for the forward pass to complete. We can improve the speed using the `CUDA.jl` library.

   The next challenge arises when actually attempting to compute the SVD. For $\boldsymbol{Y} \in \mathbb{R}^{N \times d}$, $N \gg d$ and it is computationally expensive to compute SVD of such large matrices. We can simplify this by computing the

thin QR decomposition, where $\boldsymbol{R} \in \mathbb{R}^{d \times d}$, and then compute the SVD over this smaller matrix. We can then compose $\boldsymbol{QU}$ to obtain the new SVD.

3. For this problem, I'll start by proving a useful property of orthogonal matrices:
Lemma 1. *The matrix multiplication of two orthogonal matrices results in a new orthogonal matrix.*

Proof. Let $\boldsymbol{A}, \boldsymbol{B}$ be two orthogonal matrices. Then, T.P.T:

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} = \boldsymbol{B}\boldsymbol{B}^T = \boldsymbol{B}^T\boldsymbol{B},\ \det(\boldsymbol{A}) = 1 = \det(\boldsymbol{B}) \implies (\boldsymbol{AB})^T\boldsymbol{AB} = \boldsymbol{AB}(\boldsymbol{AB})^T = \boldsymbol{I},\ \det(\boldsymbol{AB}) = 1$$

First, we prove the transpose being inverse:

$$\boldsymbol{AB}(\boldsymbol{AB})^T = \boldsymbol{ABB}^T\boldsymbol{A}^T = \boldsymbol{AI}\boldsymbol{A}^T = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I}$$
$$(\boldsymbol{AB})^T\boldsymbol{AB} = \boldsymbol{B}^T\boldsymbol{A}^T\boldsymbol{AB} = \boldsymbol{B}^T\boldsymbol{IB} = \boldsymbol{B}^T\boldsymbol{B} = \boldsymbol{I}$$

And next the determinant:

$$\det(\boldsymbol{AB}) = \det(\boldsymbol{A})\det(\boldsymbol{B}) = 1 \cdot 1 = 1$$

□

Going back to the problem, we have:

$$\boldsymbol{E} \in \mathbb{R}^{50257 \times 768}, \qquad \boldsymbol{Y} \in \mathbb{R}^{13226 \times 768}$$

We can compute the QR decomposition of both these matrices:

$$\boldsymbol{A} = \boldsymbol{YE}^T = (\boldsymbol{Q}_Y\boldsymbol{R}_Y)(\boldsymbol{Q}_E\boldsymbol{R}_E)^T = \boldsymbol{Q}_Y\boldsymbol{R}_Y\boldsymbol{R}_E^T\boldsymbol{Q}_E^T$$

Then compute the SVD of the middle two matrices:

$$\boldsymbol{A} = \underbrace{\boldsymbol{Q}_Y\boldsymbol{U}_R}_{\boldsymbol{Q}_A}\underbrace{\Sigma_R}_{\Sigma_A}\underbrace{\boldsymbol{V}_R^T\boldsymbol{Q}_E^T}_{\boldsymbol{V}_A^T}$$

This gives us the SVD of $\boldsymbol{A}$.

Code

```
Q_E, R_E = qr(Float64.(gpt2model.E))
Q_Y, R_Y = qr(Float64.(Y))

U_R, S_R, Vt_R = svd(R_Y * R_E')

# SVD of A
Q_A = Q_Y * U_R
S_A = diagm(S_R)
Vt_A = Vt_R * Q_E'
```