18.335 Take-Home Midterm Solutions: Spring 2020

Problem 1: (20+5 points)

(a) We can rewrite this as:

$$\|b-Ax\|_2^2+\alpha\|x\|_2^2=\left\|\begin{pmatrix} b-Ax \\ \alpha x \end{pmatrix}\right\|_2^2=\left\|\begin{pmatrix} b \\ 0 \end{pmatrix}-\begin{pmatrix} A \\ \alpha I \end{pmatrix}x\right\|_2^2$$

where we have appended n rows of zeros to b and an $n\times n$ matrix αI to A. Therefore, we can use exactly the same analysis as class but with the condition number of $\begin{pmatrix} A \\ \alpha I \end{pmatrix}$. The singular values of this augmented matrix are the square roots of the eigenvalues of

$$\begin{pmatrix} A \\ \alpha I \end{pmatrix}^*\begin{pmatrix} A \\ \alpha I \end{pmatrix}=A^*A+\alpha^2 I$$

which simply $\sigma_k^2+\alpha^2$ where σ_k^2 are the eigenvalues of A^*A (σ_k are the singular values of A). Hence the condition number of this matrix, which is an upper bound on the condition numbrer of the regularized least-squares problem, is

$$\boxed{\sqrt{\frac{\sigma_1^2+\alpha^2}{\sigma_n^2+\alpha^2}},}$$

which goes $\to 1$ as $\alpha\to\infty$.]

(b) A larger α improves the condition number of the problem, it reduce sensitivity to errors (e.g. floating-point errors or measurement errors in b etc. On the other hand a larger α means that our minimization problem starts to favor minimizing $\|x\|$ over minimizing $\|b-Ax\|$, that is it will increase errors in the residual $\|b-Ax\|$.

Hence , a larger α therefore trades off **sensitivity to computation/measurement errors** for **larger residuals** in trying to find a "best-fit" $\hat{x}$.

Problem 2: (15+5+5 points)

(a) For small $|x|$, naively computing this function as $\tilde{f}(x)=\sqrt{1\oplus x\otimes x}\ominus 1$ will incur **cancellation errors** where the significant digits are lost, because we are **subtracting two nearly equal quantities**. Once $|x|<|\varepsilon_{\text{machine}}|$, in fact, we will get $1\oplus x\otimes x=1$ and the result will be **zero** (**all significant digits will be lost**). From Taylor expansion, one can easily see that the correct answer for small $|x|$, in contrast, is $f(x)\approx\frac{1}{2}x^2+O(x^4)$.

There are various ways to compute this more accurately. A "brute force" method would be to switch to a Taylor expansion for sufficiently small $|x|$, cancelling the -1 factor analytically, but this is awkward to implement (the cutoff to switch to a Taylor series and the required number of terms depend on the precision). Instead, a simple trick is to use the following algebraic transformation

$$f(x)=\left(\sqrt{1+x^2}-1\right)\frac{\sqrt{1+x^2}+1}{\sqrt{1+x^2}+1}=\frac{(1+x^2)-1}{\sqrt{1+x^2}+1}=\boxed{\frac{x^2}{\sqrt{1+x^2}+1}}.$$

The final expression eliminates the subtraction and cancellation error, and is accurate in floating-point arithmetic for arbitrarily small $|x|$.

(A very similar transformation was used in the lecture 2 notes, posted on the web site, for finding the roots of a quadratic equation accurately.)

(b) The condition number of f is

$$\frac{|f'(x)|}{|f(x)/x|} = \frac{x^2}{\sqrt{1+x^2}|f(x)|} = \frac{x^2}{1+x^2-\sqrt{1+x^2}} = 2+O(x^2)$$

where in the last expression we have Taylor-expanded around $x=0$. Hence, for small $|x|$, the condition number $\to 2$, which is not badly conditioned.

(c) The fact that it is well conditioned suggests that we *can* compute it with a small forward error, *not* that *all algorithms* are accurate. In particular, a small forward error is achieved for a well-conditioned problem **only if the algorithm is backwards stable**.

We can easily see that the naive algorithm $\tilde{f}$ is ***not*** **backward stable**. It yields $\tilde{f}(x) = 0$ for any sufficiently small x, hence $f(\tilde{x}) = \tilde{f}(x) \implies \tilde{x} = 0$ and $\|x-\tilde{x}\|/\|x\| = 1$, not $O(\varepsilon_{\text{machine}})$ independent of x.

Problem 3: (10+10+5 points)

(a) The nonzero pattern (the elements of $A^{(k)}$ that are *not* converging to zero) will be:

$$A^{(k)} = \underline{Q^{(k)}}^* A \underline{Q^{(k)}} \approx \begin{pmatrix} \times & \times & & & & \\ \times & \times & & & & \\ & & \times & \times & & \\ & & \times & \times & & \\ & & & & \times & \times \\ & & & & \times & \times \end{pmatrix}$$

where $\underline{Q^{(k)}} = Q^{(1)}Q^{(2)}\cdots Q^{(k)}$ as in class. As we saw in class, the QR iteration is equivalent to a simultaneous power method. For distinct $|\lambda|$, this makes $A^{(k)}$ converge to a diagonal matrix with the eigenvalues in descending order, because the columns of $\underline{Q^{(k)}}$ (equivalent to QR factorization of A^k) are the eigenvectors in descending order of $|\lambda|$. However, for *this* matrix A, the eigenvalues are in $\pm$ pairs of *equal* magnitude, so the power method will *not converge*. In particular, the first two columns of $\underline{Q^{(k)}}$ will be approximately a linear combination of the eigenvectors for ± 3, the next two columns will be in the span of the eigenvectors for ± 2, and the last two columns will be in the span of the eigenvectors for ± 1. That means that $A^{(k)}$ will (approximately) block-diagonalize into 2×2 blocks as shown.

(b) Recall from class: For any d-dimensional subspace with an orthonormal basis Q ($m \times d$), the Rayleigh–Ritz procedure finds approximate eigenvectors $x = Qz$ (Ritz vectors) in this subspace by requiring $Q^*(Ax - \nu x) = 0$, and hence $(Q^*AQ)z = \nu z$. That is, the Ritz values ν are eigenvalues of Q^*AQ.

Now, the QR iterate is $A^{(k)} = \underline{Q^{(k)}}^* A \underline{Q^{(k)}}$, and hence any $d\times d$ diagonal block of the row/col indices $i+1:i+d$ is

$$D = A^{(k)}_{i+1:i+d,i+1:i+d} = \underline{Q^{(k)}}^*_{:,i+1:i+d} A \underline{Q^{(k)}}_{:,i+1:i+d},$$

which is exactly Q^*AQ where $Q = \underline{Q^{(k)}}_{:,i+1:i+d}$, i.e. columns $i+1$ to $i+d$ of $\underline{Q^{(k)}}$. Hence the eigenvalues of D are Ritz values of this subspace.

(c) From part (a), the columns of $\underline{Q^{(k)}}$ come in pairs, each of which (for large k) is approximately in the span of the eigenvectors of ± 3, ± 2, and ± 1, respectively. That means that **if we compute the eigenvalues of the 2×2 diagonal blocks of $A^{(k)}$,** they are Ritz values for a subspace approximately spanned by pairs of eigenvectors, and hence they must converge to eigenvalues of A as $k\to\infty$.

From elementary undergraduate linear algebra, the eigenvalues of a 2×2 block D may be found by solving the quadratic equation $\det(D-\nu I)$ using the quadratic formula, which incurs a finite number of $\{\sqrt{\ }, \pm, \times, \div\}$ operations.

Problem 4: (10+15 points)

Suppose that we compute the transpose of a square matrix A in-place using obvious algorithm

```
function my_transpose!(A::Matrix)
    m, n = size(A)
    m == n || error("my_transpose! requires a square matrix")
    for i = 1:m
        for j = i+1:m
            A[i,j], A[j,i] = A[j,i], A[i,j] # swap ij and ji entries
        end
    end
end
```

(a) Because we are reading/writing both A_{ij} and A_{ji} on each loop iteration, we are accessing at least one of these discontiguously in memory. In particular, since A is column-major in Julia and j is the inner loop, A_{ji} is accessed consecutively (A_{ji} and $A_{j+1,i}$ are consecutive in memory for column-major A), but A_{ij} is non-consecutive (A_{ij} and $A_{i,j+1}$ are stored m elements apart in memory for column-major A).

For the cache complexity, we must consider the asymptotic case of large $m \gg Z > L$ and focus on the non-consecutive A_{ij} reads. At the start of each row i, at most Z elements of A are in-cache, and hence we must read in most of the row. Unfortunately, because the elements of the row are separated by $m > L$ elements, reading each element of the row is a separate cache miss. Reading the *column* elements A_{ji} incurs only $\Theta(m/L)$ misses because the column elements are consecutive, but for large m these cache lines are almost entirely disjoint from the row elements A_{ij}. Therefore, reading the row incurs $\Theta(m)$ misses, and hence there are $\boxed{\Theta(m^2)}$ misses overall—in an asymptotic big-O sense, we get no benefit from the cache in `my_transpose!`.

(b) The simplest approach is the "cache-aware" algorithm where we divide the matrix into pairs of $L \times L$ blocks, one above the diagonal and one below the diagonal, and swap them; also, there are m/L blocks of size $L \times L$ along the diagonal that we transpose in-place. (If m does not divide L, we will have some partial blocks along the edges of A.) Because the columns of each $L \times L$ block are contiguous and fit into a cache line, reading the block into cache will only incur L misses, after which the swapping/transposition can occur in-cache. There are $\Theta(m^2/L^2)$ such blocks. Hence there are $\Theta(m^2/L^2) \times L = \boxed{\Theta(m^2/L)}$ misses.

There are other possible algorithms. e.g. a cache-oblivious algorithm that divides A recursively into 4 submatrix blocks and transposes/swaps them also achieves $\Theta(m^2/L)$ cache complexity.

Technically, in order for this algorithm to work, we require $Z > L^2$, which is called the "tall-cache" assumption. As I mentioned in class, in practice this always true: cache lines are on the order of 100 bytes, whereas caches are on the order of tens of kilobytes or megabytes.

Notice that Z does not appear in the cache complexity, only L. That is because transposition is an algorithm that "touches" each element of A only once, and hence cannot benefit from temporal locality, only spatial locality (consecutive access = cache-line utilization).