Homework set 1 — MATH 393C — Spring 2019

Due on Tuesday Feb. 12. Please hand in solutions to problems 2, 3, 5, and 6.

Problem 1: Let $\mathbf{A}$ be an $m \times n$ matrix, set $p = \min(m, n)$, and suppose that the singular value decomposition of $\mathbf{A}$ takes the form

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times p}{\mathbf{U}} \; \underset{p \times p}{\mathbf{D}} \; \underset{p \times n}{\mathbf{V}^*}. \tag{1}$$

Let k be an integer such that $1 \leq k < p$ and let $\mathbf{A}_k$ denote the truncation of the SVD to the first k terms:

$$\mathbf{A}_k = \mathbf{U}(:, 1:k)\, \mathbf{D}(1:k, 1:k)\, \mathbf{V}(:, 1:k)^*.$$

Recall the definitions of the spectral and Frobenius norms:

$$\|\mathbf{A}\| = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}, \qquad \text{and} \qquad \|\mathbf{A}\|_{\mathrm{F}} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |\mathbf{A}(i,j)|^2 \right)^{1/2}.$$

Prove directly from the definitions of the norms that

$$\|\mathbf{A} - \mathbf{A}_k\| = \sigma_{k+1}$$

and that

$$\|\mathbf{A} - \mathbf{A}_k\|_{\mathrm{F}} = \left(\sum_{j=k+1}^{p} \sigma_j^2 \right)^{1/2}.$$

Problem 2: On the course webpage, download the file `hw01p2.m`. This file contains an implementation of the column pivoted QR algorithm that computes a rank-k approximation to a matrix, for any given k. Your task is now to do two modifications to the code:

(a) Starting with the function `CPQR_given_rank` write a new function with calling sequence

```
[Q,R,ind] = CPQR_given_tolerance(A,acc)
```

that takes as input an accuracy, and computes a low-rank approximation to a matrix that is accurate to precision "acc".

(b) Write a function with calling sequence

```
[U,D,V] = SVD_given_tolerance(A,acc)
```

that computes a diagonal matrix $\mathbf{D}$, and orthonormal matrices $\mathbf{U}$ and $\mathbf{V}$ such that

$$\|\mathbf{A} - \mathbf{U}\mathbf{D}\mathbf{V}^*\| \leq \varepsilon,$$

where ε is the given tolerance. The idea is to use the function `CPQR_given_tolerance(A,acc)` that you created in part (a).

Please hand in a print-out of the code that you created.

Extra problem: The file `hw01p2.m` creates a plot that shows the accuracies of two low-rank approximations: The truncated SVD on the one hand, and the truncated QR on the other. Let me encourage you to play around with this a bit, try different matrices and see how the approximations errors compare. There is no need to hand anything in!

Problem 3: In this example, we investigate the effect *blocking* has on execution time of matrix computations.

(a) Suppose that we are given two $n \times n$ matrices $\mathbf{B}$ and $\mathbf{C}$ and that we seek to compute $\mathbf{A} = \mathbf{BC}$. We could do this in Matlab either by just typing `A = B*C`, or, we could write a loop

for $i = 1 : n$
  `A(:,i) = B*C(:,i)`
end for

The code `hw01p3.m` illustrates the two techniques. It turns out that while the two methods are mathematically equivalent, doing it via a loop is much slower. In this problem, please measure the time T_n required for several different values of n. Test the hypothesis that $T_n = C\,n^3$ by plotting your measure valued of T_n versus n in a log-log-diagram. Fit a straight line through the points, and estimate C. Hand in the graph and the values of C that you estimate for the two methods.

(b) Repeat the problem in (a), but now compare three different matrix factorization algorithms:
- `[Q,R] = qr(A)`
  This factorization can be blocked. It is fast, but no good for low-rank approximation.
- `[Q,R,J] = qr(A,'vector')`
  Column pivoted QR factorization — intermediately fast, and good for low-rank approximation.
- `[U,D,V] = svd(A)`
  Singular value decomposition — slowest, but excellent for low-rank approximation.

Problem 4: Recall the *single pass* RSVD shown in Figure 1. Consider the following piece of Matlab code:

```
m=12; n=20; b=3; k=5;
A  = rand(m,n);
Gc = randn(n,k);
Gr = randn(m,k);
Yc = zeros(m,k);
Yr = zeros(n,k);

for i = 1:(n/b)
  ind       = (i-1)*b + (1:b);
  A_slice   = A(:,ind);
  Yc        = Yc + A_slice*Gc(ind,:);
  ??????????????????????????????????
end

fprintf(1,'Error in Yc = %12.3e\n',max(max(abs(Yc - A *Gc))))
fprintf(1,'Error in Yr = %12.3e\n',max(max(abs(Yr - A'*Gr))))
```

This snippet of code emulates how a streaming algorithm would interact with $\mathbf{A}$ — you read a set of b columns at a time, and use the information in $\mathbf{A}_{\text{slice}}$ to build $\mathbf{Y}_{\text{c}}$ piece by piece.

(a) Write code to replace the question marks. The result of the new code should be that after the loop completes, the matrix `Yr` has also been computed. Note that this line should reference only `A_slice`, not `A` itself. (This can be solved by a single line of code, but if you use more, then that is fine too.)

(b) Currently, the code only works if `n` is an integer multiple of `b`. Modify the code so that it works for any block size `b`.

Hand in the code you write.

ALGORITHM: SINGLE-PASS RANDOMIZED SVD FOR A GENERAL MATRIX

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 10$).

Outputs: Matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{D}$ in an approximate rank-k SVD of $\mathbf{A}$. (I.e. $\mathbf{U}$ and $\mathbf{V}$ are ON and $\mathbf{D}$ is diagonal.)

Stage A:

(1) Form two Gaussian random matrices $\mathbf{G}_{\text{c}} = \texttt{randn}(n, k+p)$ and $\mathbf{G}_{\text{r}} = \texttt{randn}(m, k+p)$.
(2) Form the sample matrices $\mathbf{Y}_{\text{c}} = \mathbf{A}\,\mathbf{G}_{\text{c}}$ and $\mathbf{Y}_{\text{r}} = \mathbf{A}^*\,\mathbf{G}_{\text{r}}$.
(3) Form ON matrices $\mathbf{Q}_{\text{c}}$ and $\mathbf{Q}_{\text{r}}$ consisting of the k dominant left singular vectors of $\mathbf{Y}_{\text{c}}$ and $\mathbf{Y}_{\text{r}}$.

Stage B:

(4) Let $\mathbf{C}$ denote the $k \times k$ least squares solution of the joint system of equations formed by the equations $\left(\mathbf{G}_{\text{r}}^*\mathbf{Q}_{\text{c}}\right)\mathbf{C} = \mathbf{Y}_{\text{r}}^*\mathbf{Q}_{\text{r}}$ and $\mathbf{C}\left(\mathbf{Q}_{\text{r}}^*\mathbf{G}_{\text{c}}\right) = \mathbf{Q}_{\text{c}}^*\mathbf{Y}_{\text{c}}$.
(5) Decompose the matrix $\mathbf{C}$ in a singular value decomposition $[\hat{\mathbf{U}},\, \mathbf{D},\, \hat{\mathbf{V}}] = \texttt{svd}(\mathbf{C})$.
(6) Form $\mathbf{U} = \mathbf{Q}_{\text{c}}\hat{\mathbf{U}}$ and $\mathbf{V} = \mathbf{Q}_{\text{r}}\hat{\mathbf{V}}$.

FIGURE 1. A basic randomized algorithm single-pass algorithm suitable for a general matrix.

Problem 5: On the course webpage, download the file `hw01p5.m`. This file contains an implementation of the basic RSVD scheme. It computes approximate matrix factorizations using the basic RSVD, and plots the approximation error versus the minimum error as produced by the truncated (full) SVD. The error is reported in both the spectral and the Frobenius norms.

For this problem, code up the single pass algorithm described in Figure 1 and include it in the comparison. Hand in a printout of your implementation of the single pass algorithm (you do not need to print out the driver code, etc, just the actual subroutine). Also hand in the error plots for three different sets of test matrices. In the code that you can download, two test cases are already included. You are welcome to use these two. For the third, come up with some matrix you find interesting yourself! It could be dense, sparse, etc. Just remember that for any of this to make sense, the singular values of the matrix you pick must show at least some degree of decay.

Note: In step (4) of the algorithm, the matrix $\mathbf{C}$ is determined by jointly solving two matrix equations. Note that this is a bit complicated to implement. For simplicity, simply pick one of the two equations and determine $\mathbf{C}$ by solving that one, ignoring the other.

Problem 6: Implement the SRFT code described in Figure 2. In your implementation, just apply the *full* FFT to the rows of the matrix $\mathbf{AD}$, and then extract $k + p$ randomly picked columns. This will in principle be slower than a properly implemented subsampled FFT that never evaluates the unneeded indices at all. But, our objective here is not to test speed, just to see how the method compares in terms of *accuracy*.

Hand in the same material that is specified for Problem 2 — printout of the subroutine, and three figures showing the errors for three different matrices. (You are welcome to plot the errors of both the SRFT and the single-pass algorithm in the same figure, and hand in one set of three plots that cover both Problem 2 and Problem 3.)

ALGORITHM: BASTARDIZED RANDOMIZED SVD BASED ON THE SRFT

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = k$).

Outputs: Matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{D}$ in an approximate rank-k SVD of $\mathbf{A}$. (I.e. $\mathbf{U}$ and $\mathbf{V}$ are ON and $\mathbf{D}$ is diagonal.)

Note: This algorithm explicitly computes $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$. This precludes $O(mn \log k)$ complexity, but lets us analyze how well the SRFT works as a random matrix. (Also, the way the application of $\mathbf{R} = \mathbf{DFS}$ is described is computationally inefficient, but is mathematically equivalent to a "proper" SRFT.)

Stage A:

(1) Draw n random numbers $\mathbf{d} = \{d_j\}_{j=1}^n$ and set $\mathbf{D} = \texttt{diag}(\mathbf{d})$. Draw at random without replacement an index vector J of length $k + p$ from the set $\{1, 2, \ldots, n\}$.
(2) Compute $\mathbf{Z} = \mathbf{ADF}$ where $\mathbf{F}$ is the discrete Fourier transform. Then form $\mathbf{Y} = \mathbf{Z}(:, J)$.
(3) Form an ON matrix $\mathbf{Q}$ by orthonormalizing the columns of $\mathbf{Y}$ so that $\mathbf{Q} = \texttt{orth}(\mathbf{Y})$.

Stage B:

(4) Compute $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.
(5) Decompose the matrix $\mathbf{B}$ in a singular value decomposition $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \texttt{svd(B,'econ')}$.
(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

FIGURE 2. A randomized algorithm based on a subsampled Fourier transform. Observe that the output matrices $\mathbf{U}$ and $\mathbf{V}$ will be complex even when $\mathbf{A}$ is real.