Homework 10
APPM 5650 Fall 2021
Randomized Algorithms

Due date: Monday, Nov 1 2021 **Instructor**: Prof. Becker
Theme: Randomized SVD **Revision date**: 10/25/2021

Instructions Collaboration with your fellow students is allowed and in fact recommended, although direct copying is not allowed. Please write down the names of the students that you worked with. The internet is allowed for basic tasks only, not for directly looking for solutions.

An arbitrary subset of these questions will be graded.

Please refer to "Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions" (Halko, Martinsson, Tropp; SIAM Review 2011; we'll refer to this paper as "HMT"), and also Algo 3 of "Single-pass PCA of large high-dimensional data" (Yu, Gu, Li, Liu, Li; IJCAI 2017).

Algorithm 1 Basic 2-pass randomized SVD of HMT, i.e., Algo 4.1 + Algo 5.1. *Note: a lot of bugs are due to memory issues. If A is very large, then do not actually form $\widetilde{A}$. Another place that memory bugs arise from is when computing the `QR` factorization of Y (and B); make sure this is a thin QR, not a full QR, since we want Q to be a very tall rectangular matrix and definitely not a very big square matrix.*

---

procedure RANDOMIZED SVD(A, k, p, q) ▷ A is $M \times N$, k is target rank, p oversampling, $q \geq 0$ pwr iter
  Draw $N \times \ell$ matrix Ω, iid Normal entries ($\ell = k + p$)
  $Y = A\Omega$
  $QR = Y$ ▷ Thin `QR` factorization
  for $i = 1, \ldots, q$ **do** ▷ Optional: power iterations for increased accuracy
    $B = A^T Q$
    $QR = B$, $B \leftarrow Q$
    $Y = AB$
    $QR = Y$
  end for
  $B = A^T Q$ ▷ So $A \approx QQ^T A = QB^T$
  $\widetilde{U}\Sigma V^T = B^T$ ▷ Thin `svd`
  Keep first k columns of $\widetilde{U}$ and V, first k cols and rows of Σ
  $U = Q\widetilde{U}$
  return $\{U, \Sigma, V\}$ and/or $\widetilde{A} \stackrel{\text{def}}{=} U\Sigma V^T$
end procedure

Problem 1: [CODING] Compare the performance of

a) the randomized SVD (fix the oversampling parameter, e.g., $p = 10$)
b) the randomized SVD with a few power iterations, and
c) the Lanczos method (`svds` in Matlab, or scipy.sparse.linalg.svds in Python).

Compare the algorithms on the following two matrices (one dense, the other sparse). In Python, you don't have to do exactly this, but make similar matrices of the same size.

Matrix 1:

Algorithm 2 1-pass randomized SVD of HMT section 5.5. *Note that we include this version because it's simple, follows the basic HMT paper, and it's here to compare with the subsequent algorithm below, but to be fair, a better version of this is described in "Practical sketching algorithms for low-rank matrix approximation" (JA Tropp, A Yurtsever, M Udell, and V Cevher; SIAM Journal on Matrix Analysis and Applications 38 (4), 1454–1485).*

procedure ONE-PASS RANDOMIZED SVD(A, k, p) ▷ A is $M \times N$, k is target rank, p oversampling
  Draw $N \times \ell$ matrix Ω, iid Normal entries ($\ell = k + p$)
  Draw $M \times \ell$ matrix $\widetilde{\Omega}$, iid Normal entries
  Multiply $Y = A\Omega$ and $\widetilde{Y} = A^T\widetilde{\Omega}$ ▷ In 1-pass over data
  $QR = Y$ and $\widetilde{Q}\widetilde{R} = \widetilde{Y}$ ▷ Thin `QR` factorization
  $B = \text{argmin}_{B \in \mathbb{R}^{\ell \times \ell}} \|Q^TY - B(\widetilde{Q}^T\Omega)\|_F^2 + \|(\widetilde{Q}^T\widetilde{Y})^T - (Q^T\widetilde{\Omega})^TB\|_F^2$ ▷ Use `LSQR`
  $\widetilde{U}\Sigma\widetilde{V}^T = B$ ▷ Thin `svd`
  Keep first k columns of $\widetilde{U}$ and $\widetilde{V}$, first k cols and rows of Σ
  $U = Q\widetilde{U}$, $V = \widetilde{Q}\widetilde{V}$
  return $\{U, \Sigma, V\}$ and/or $\widetilde{A} \stackrel{\text{def}}{=} U\Sigma V^T$
end procedure

Algorithm 3 1-pass randomized SVD of Yu et al. '17

procedure BETTER ONE-PASS RANDOMIZED SVD(A, k, p) ▷ A is $M \times N$, k is target rank, p oversampling
  Draw $N \times \ell$ matrix Ω, iid Normal entries ($\ell = k + p$)
  Multiply $Y = A\Omega$ and $B = A^TY$ ▷ In 1-pass over data
  $QR = Y$ ▷ Thin `QR` factorization
  $B \leftarrow BR^{-1}$ ▷ so $B = A^TQ$; this step must be modified if R is rank deficient!
  $\widetilde{U}\Sigma V^T = B^T$ ▷ Thin `svd`
  Keep first k columns of $\widetilde{U}$ and V, first k cols and rows of Σ
  $U = Q\widetilde{U}$
  return $\{U, \Sigma, V\}$ and/or $\widetilde{A} \stackrel{\text{def}}{=} U\Sigma V^T$
end procedure

```
M   = 1e3; N   = 2*M;
A   = 1/sqrt(M*N)*randn(M,N)*diag(logspace(0,-5,N))*randn(N,N);
```

Matrix 2:

```
M   = 2e3; N   = 2*M;
A   = sprandn(M,N,0.05);
```

Deliverable: Plot the errors in spectral and Frobenius norm, and the timing, for the 3 methods as well as for a direct dense SVD (`svd` in Matlab, numpy.linalg.svd in Python), for both matrices. Overall, that is 6 plots (2 errors + timing × 2 matrices); all three methods should always be on the same plot to facilitate comparison.

Hints: computing the spectral norm of the error can be the most time-consuming part of your code, so you can estimate this using power iterations. Matlab does it for you if you have a sparse matrix (`normest`) but we really need it for a sparse matrix minus a low-rank matrix. I've adapted `normest` to do this and put it on Github, called `my_normest.m`, and there's also a similar version we made in Python called `power_method.py` (thanks to Will Shand).

In Matlab, to export figures, I recommend export_fig. Make sure your graphs are readable if you print them in black and white!

Problem 2: [CODING] Compare the two versions of one-pass SVD algorithms, Algorithms 2 and 3 (also include the two-pass Algorithm 1 as a benchmark). Do this on a similar matrix to problem 1, but one that has faster decaying singular values:

```
M   = 1e3; N   = 2*M;
A   = 1/sqrt(M*N)*randn(M,N)*diag(logspace(0,-5,N).^4)*randn(N,N);
```

Deliverable: for $k = 150$, what is the Frobenius norm error for all 3 randomized SVD algorithms? Comment on the accuracy of the different algorithms. You can choose how much oversampling p to use.

Hint: for Algorithm 2, you find B as the solution to a least-squares problem. However, it's not a typical least-squares problem because it is a matrix variable. It is equivalent to a standard least-squares problem in the variable $b = \text{vec}(B) \stackrel{\text{def}}{=} B(:)$ [Matlab notation], with a linear operator

$$\widetilde{\mathcal{A}}(b) = \mathcal{A}(B = \text{mat}(b)) = \begin{bmatrix} \text{vec}(BD) \\ \text{vec}(FB) \end{bmatrix}, \quad D \stackrel{\text{def}}{=} \widetilde{Q}^T\Omega, \quad F \stackrel{\text{def}}{=} (Q^T\widetilde{\Omega})^T$$

and right-hand-side $\begin{bmatrix} \text{vec}(Q^TY) \\ \text{vec}((\widetilde{Q}^T\widetilde{Y})^T) \end{bmatrix}$. Here, $\text{mat}(b)$ reshapes b into an $\ell \times \ell$ matrix, in whatever order in needs to be consistent with vec so that mat $\circ$ vec is the $\ell^2 \times \ell^2$ identity matrix.

You definitely want to use an iterative method, because making $\mathcal{A}$ explicit is a large matrix and it would be inefficient to apply ($\mathcal{O}(\ell^4)$ flops, compared to $\mathcal{O}(\ell^3)$ flops if you do it as above). In matlab, use `lsqr`, and in python use scipy.sparse.linalg.lsqr. For python, the routine will expect you to pass in a LinearOperator object.

For an iterative method, you need to know the transpose/adjoint of $\widetilde{\mathcal{A}}$. Use the fact that the order of composition of linear operators flips when you do the transpose, and that mat and vec are transposes of each other, and the transpose of $B \mapsto BD$ is $Z \mapsto ZD^T$, etc. If you're confused about this part, please come to office hours!

Note that Algorithm 3 will fail if you apply it to a matrix that has true rank less than ℓ. You can make the algorithm robust by computing the SVD of B, determining what the effective rank is, then removing columns of Y and B. You do *not need to implement this*!.

Problem 3: [CODING] Run Algorithm 3 on the 73 GB file `/rc_scratch/stbe1590/hugeSquareMatrix.mat` (on the research computing cluster; refer to HW 3 and HW 4) which contains a $10^5 \times 10^5$ matrix, for $k = 150$. You may want to check your algorithm using the smaller matrix `/rc_scratch/stbe1590/largeSquareMatrix.mat` which is only 752 MB and contains a $10^4 \times 10^4$ matrix.

Note: As in HW 4, this matrix was created by using Matlab's default save interface for v7.3, which makes a slightly modified HDF5 format, and it has some kind of 'chunking' (determined automatically), which you could in theory find and use to load the file faster.

Alternatively (also like in HW 4) load the file `/rc_scratch/stbe1590/hugeSquareMatrix.h5` (and test with `/rc_scratch/stbe1590/largeSquareMatrix.h5`) which was created using `h5create` and `h5write` in Matlab; this allows us to turn off compression, so operations are faster (between 2 and 10 times faster in my experience). This file format is pure HDF5.

Deliverable: return a plot of the estimated top 150 singular values (note that it is not easy to report the error, as this requires either a second pass through the data, or a randomized estimate), and also report the time it took to read in the data and the time it took to do the computation.

Hint: to compute something like $Y = A\Omega$ and $B = A^TY$ in 1-pass over the data, assuming the entire data matrix A is too large to store in memory, load A by rows and for simplicity, assume we can split A into 2 blocks of rows, each of which fits into memory (and it will soon be clear how to generalize to more than 2 blocks), eg., $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$, and similarly, $Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} A_1\Omega \\ A_2\Omega \end{bmatrix}$. Then $B = A^TY = \begin{bmatrix} A_1^T & A_2^T \end{bmatrix} \cdot \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = A_1^TY_1 + A_2^TY_2$. Thus the idea is as follows: load A_1 into

memory, compute $Y_1 = A_1\Omega$, then compute $B = A_1^T Y_1$. Now throw away A_1 and load A_2 into memory, compute $Y_2 = A_2\Omega$, and then update $B \leftarrow B + A_2^T Y_2$.