ECS231 Project: Randomized Algorithms with Application in Low-rank Matrix Approximation

Jiaping Zhang *, Yin Zhang †
University of California, Davis

June 2016

Abstract

Randomized algorithms for large-scale matrix problems have gained a great deal of attention in recent years. With the statistical leverage, randomized methods have been studied and applied to solve large-scale linear algebra problems such as the linear least-squares problem and the low-rank matrix approximation problem. Large-scale PCA is an example of low-rank matrix approximation that plays an important role in data compression and modern data analysis. In this paper, we will give a brief overview of the theory of randomized matrix algorithms, and implement the methods for solving low-rank matrix approximation. We implement and compare two approaches:Randomized SVD and CUR Decomposition. Through our numerical experiments on testing matrices with and without decaying eigenvalue structure as well as real-life matrices, we find that both approaches have similarly good accuracy convergence in terms of relative error ratio.

1 Introduction

Randomized Numerical Linear Algebra is an interdisciplinary research area that exploits randomization as a computational resource to develop improved algorithms for large-scale linear algebra problems[1]. Randomized methods are motivated by the challenges in modern big data analysis, which root in the numerical linear algebra problems, i.e solving large-scale linear system and low-rank matrix approximation. In this project, we study the low-rank matrix approximation problem, which lays the foundation in many applications of scientific computing, including principal component analysis, face recognition, large scale data compression and fast approximate algorithms for PDEs and integral equations. There are two main approaches to solve the problem: one is randomized singular value decomposition(SVD), and the other method is CUR matrix approximation.

For the randomized SVD, it has been regonized as an efficient and reliable approach to approximate the matrix for any given fixed rank. There are different approaches to solve the low-rank matrix approximations, such as modifiled Gram-Schimidt with column pivoting, rank-revealing QR/LU factorization, Partial SVD and the Lanczos Algorithm. However, those methods have limitations in terms of reliability and computation efficiency. Alghouth truncated SVD can have the best quality, it would still fail in the context of large-scale computing since it requires $O(mn^2)$ floating point operations. In this paper, through numerical experiments, we will show how randomized algorithms with subspace iteration and random sampling techniques are powerful to solve the low-rank matrix approximation problem.

For CUR decomposition, there are lots of related research on this topic. The classic one is the deterministic CUR algorithm(SCRA)[7]. It deterministically finds the set of columns/rows by truncated pivoted QR decomposition procedure using column/rows pivoting. Another famous algorithm is two-stage randomized CUR algorithm with high probability relative-error bound[3]. It basically used the sampling probabilities

*Email: jpzhang@ucdavis.edu
†Email: yinzh@ucdavis.edu

which are proportional to the squared l" norm to get the corresponding rows and columns. So the main part for CUR algorithm is the way of matrix column selection. In the paper, we illustrate a more efficient and accurate CUR method – CUR Decomposition via Adaptive Sampling which established a more general error bound. The CUR Decomposition via Adaptive Sampling have low time complexity and can avoid maintaining the whole data matrix in RAM.

In this report, we introduce basic underlying concepts and theory of randomized algorithms for solving low-rank matrix approximation in Section 2, discuss the implementation in Section 3 and 4, explain our experiment results and compare two approaches in Section 5.

2 Basic Concepts and Theory

2.1 Low-Rank Matrix Approximation

The classical ways of matrix decomposition include the pivoted QR factorization, the eigenvalue decomposition and the singular value decomposition(SVD). Those matrix factorization methods are deterministic to represent the structures of matrices. In practice, especially in data mining, the matrices are stupendously big. It won't be ideal to afford the computational cost to compute and store large matrices. We would be interested in the truncated versions of decomposition that can still contain the main information in the large matrix with less noise. In the light of reducing the dimensionality of data, the form of truncation can be expressed as following:

$$\underset{m\times n}{A} \approx \underset{m\times k}{B}\ \underset{k\times n}{C}$$

The inner dimension k is sometimes called the numerical rank of the matrix. When the numerical rank is much smaller than either dimension m or n, it allows the matrix to be stored inexpensively and matrix multiplication to be computed rapidly. Furthermore, the low-rank matrix can be summarized as following:

$$\min_{rank(B)\leq k} ||A-B||_2$$

Note that Eckart & Young[2] proved that

$$\min_{rank(B)\leq k} ||A-B||_2 = ||A-A_k||_2 = \sigma_{k+1} \quad \min_{rank(B)\leq k} ||A-B||_F = ||A-A_k||_F = \sqrt{\sum_{j=k+1}^{n} \sigma_j^2}$$

where A_k is the rank-k truncated SVD which has the following form:

$$\underset{m\times n}{A_k} = \underset{m\times k}{U_k}\ \underset{k\times k}{\Sigma_k}\ \underset{k\times n}{V_k^T}$$

The above theory shows that the truncated SVD is the ideal rank-k approximation to A with the smallest possible 2-norm error and Frobenious-norm error. Then the question comes to how to solve the truncated SVD problem efficiently in the large-scale matrix. The goal of low-rank matrix approximations is to find the approximate matrix efficiently and as accurate and reliable as truncated SVD.

2.2 Randomized SVD

There exists two main classes of approaches to solve the truncated SVD problem: Lanczos algorithm and Randomized Algorithm with subspace iteration. The Lanczos algorithm applies the idea of subspace projection, while it is much more computational expensive due to the unefficiency for computing matrix-vector product at each iteration. Thus, it is not preferred in the context of big data applications.

With the subspace iteration technique, randomized algorithm has been proven to be highly efficient and reliable with minimum communication becuase matrix-matrix products and QR factorization have been highly optimized for maximum efficiency on modern serial and parallel architechtures[2]. Compared with

the previously-existing deterministic algorithm, as Mahoney illustrates, the randomized algorithms have worst-case running time that is asymptotically faster; their numerical implementations are faster in terms of clock-time; or they can be implemented in parallel computing environments where existing numerical algorithms fail to run at all[3].

The general idea of Randomized method with subspace iteration is to construct and operate on a randomized sketch of the input matrix. In the light of constructing k orthonormal vectors that nearly span the range of A for the approximate SVD of A, the following schema helps to understand randomized SVD:

1. Assume we have the matrix Q that has l orthognomal columns. With the orthogonal projector QQ^*, we have
$$A \approx QQ^*A$$
2. Since the number l of columns of Q is substantially less than both dimensions of A, we can efficiently compute
$$B = Q^*A$$
3. Since B has been significantly reduced in matrix size, we can then efficiently compute the rank-k truncated SVD
$$U_k\Sigma_k V_k^* = B_k \approx B$$
4. Denote $U = QU_k$, with the formula above, we can return the following as our output as approximated rank-k truncated SVD of A
$$U\Sigma_k V_k^* \approx A$$

Then it comes to the question that how we can find and define the orthogonal matrix Q. It turns out randomized methods levarage the conclusions from statistical analysis. Due to the page limit, the detailed explanation and proof can be referred in Gu's paper[2]. We show the insightful remarks as the following

- A standard Gaussian matrix is rotationally invariant and it helps to guarantee the property of converge in probability in terms of statistical analysis
- In general, it is suggested to have some over-sampling in the number of columns that can significant improve convergence in the singular value approximation; unless there is a large singular value gap at σ_l or σ_{l+1}
- Subspace iteration method approximates the leading k singular values by a good fraction on average, regardless of how the singular values are distributed

Thus, in the algorithmic implementation, before constructing the orthognomal matrix Q, we first multiply a strench matrix on A, namely, a standard Gaussian random matrix Ω(an invariant transformation) to enhance the convergence if the useful information of leading eigenvectors is missing; then, with $Y = A\Omega$, we can apply orthogonalizal factorization technique, like QR factorization, to obtain the orthognomal matrix Q.

2.2.1 Error Bound Analysis

Halko/Martinsson/Tropp proved that the time complexity of randomized SVD[8]: with $p = l - k$,

$$||A - B||_2 = O(\sigma_{k+1}) \text{ with failure probability } 5p^{-p}$$

It is also worth noting that in Gu's novel work of surveying randomized algorithm in the framework of subspace iteration[2], he derived more detailed and new error bounds for the average case and large deviation case. He concludes that the factor σ_{l-p+1}^2 plays an important role since it appears in the error bounds, which suggests that randomized algorithm can significant progress toward convergence in the case of rapidly decaying singular values.

2.3 CUR Decomposition via Adaptive Sampling

The main idea for CUR decomposition algorithm is about seeking to find a subset of c columns of A to form a matrix $C \in \mathbb{R}^{m\times c}$, a subset of r rows to from a matrix $R \in \mathbb{R}^{r\times n}$, and an intersection matrix $U \in \mathbb{R}^{c\times r}$ such that $\|A - CUR\|$ is small.

There are two main steps for the CUR Decomposition via Adaptive Sampling. First, we use a more general error bound for the adaptive column/row sampling algorithm. Then based on that, the CUR method will be more accurate with expected relative-error bounds. In section 2.3.1, we explained the error bound for the adaptive sampling algorithm. In section 2.3.2, we apply the adaptive sampling to the CUR to obtain a more effective and efficient CUR algorithms which proposed in [4].

2.3.1 Adaptive Sampling

According to the process of CUR matrix decomposition above, the decomposition has a close connection with the column selection problem. So one of a basic lemma2.1[5] for adaptive sampling for column selection is below.

Lemma 2.1 *Given a matrix $A \in \mathbb{R}^{m\times n}$, we let $C_1 \in \mathbb{R}^{m\times c_1}$ consist of c_1 columns of A, and define the residual $B = A - C_1C_1^{\dagger}A$. Additionally, for $i = 1, ...n$, we define*

$$p_i = \|b^{(i)}\|_2^2/\|B\|_F^2$$

We further sample c_2 columns i.i.d. from A, in each trial of which the i_{th} column is chosen with probability p_i. Let $C_2 \in \mathbb{R}^{m\times c_2}$ contain the c_2 sampled columns and let $C = [C_1, C_2] \in \mathbb{R}^{m\times(c_1+c_2)}$. Then, for any integer $k > 0$, the following inequality holds:

$$\mathbb{E}\|A - C_1C_1^{\dagger}A\|_F^2 \le \|A - A_K\|_F^2 + \frac{k}{c_2}\|A - C_1C_1^{\dagger}A\|_F^2$$

where the expectation is taken with respect to C_2. Here $C_1^{\dagger}$ is called the intersection matrix, which is the Moore-Penrose inverse of C_1.

In the lemma 2.1, it only provides the algorithm of selection columns or rows for the certain relative error bounds. However, from the process of CUR decomposition, we should choose both columns and rows. So, "one can get food columns or rows separately" does not mean that one can get good columns and rows together. Instead of using lemma 2.1, they extend it to make it suitable for choose columns and rows together, the corresponding Theorem is below.

Theorem 2.2 ***(The Adaptive Sampling Algorithm)*** *Given a matrix $A \in \mathbb{R}^{m\times n}$, and a matrix $C \in \mathbb{R}^{m\times c}$ such that $rank(C) = rank(CC^{\dagger}A) = \rho$ $(\rho \le c \le n)$. We let $R_1 \in \mathbb{R}^{r_1\times n}$ consist of r_1 rows of A, and define the residual $B = A - AR_1^{+}R_1$. Additionally, for $i = 1, ..., m$, we define*

$$p_i = \|b^{(i)}\|_2^2/\|B\|_F^2$$

We further sample r_2 rows i.i.d from A, in each trial of which the i_th row is chosen with probability p_i. Let $R_2 \in \mathbb{R}^{r_2\times n}$ contain the r_2 sampled rows and let $R = [R_1^T, R_2^T]^T \in \mathbb{R}^{(r_1+r_2)\times n}$. Then we have

$$\mathbb{E}\|A - CC^{\dagger}ARR^{\dagger}\|_F^2 \le \|A - CC^{\dagger}A\|_F^2 + \frac{\rho}{r_2}\|A - AR_1R_1^{\dagger}\|_F^2$$

where the expectation is taken with respect to R_2.

The theorem2.2 bounds the error incurred by the projection onto the column space of C and row space of R simultaneously, which is correspond to CUR process. That is, it guarantees the combined effect column and row selection. So based on theorem2.2, we can use CUR based on a upper bound of relative-error. That is, given a matrix $A \in \mathbb{R}^{m\times n}$, a target rank $k(\ll m, n)$, and a column and row selection algorithm $\mathcal{A}_{col}$ which achieves relative-error upper bound by selecting $c \ge C(k, \epsilon)$ columns of A to construct C and $r_1 = c$ rows to

construct R_1,followed by selecting additional $r_2 = c/\epsilon$ rows using adaptive sampling algorithm to construct R_2, the CUR matrix decomposition achieves relative-error upper bound in expectation:

$$\mathbb{E}\|A - CUR\|_F \leq (1+\epsilon)\|A - A_k\|_F$$

where $R = [R_1^T, R_2^T]^T$ and $U = C^\dagger A R^\dagger$.

Moreover, when $C = A_k$(i.e., $c = n, \rho = k$, and $CC \dagger A = A_k$), we can see that the lemma2.1 is a direct corollary of theorem 2.2.

2.3.2 Adaptive Sampling for CUR Matrix decomposition

Based on theorem2.2, we can combined the near-optimal column selection algorithm and the adaptive sampling algorithm for solving CUR problem with much tighter theoretical relative-error upper bound.

The corresponding algorithm is described in Algorithm part. With random selects $c = \frac{2k}{\epsilon}(1 + o(1))$ columns of A to construct $C \in \mathbb{R}^{m\times c}$, and the selects $r = \frac{c}{e}(1+\epsilon)$ rows of A to construct $R \in \mathbb{R}^{r\times m}$, then we have

$$\mathbb{E}\|A - CUR\|_F = \mathbb{E}\|A - C(C^\dagger A R^\dagger)UR\|_F \leq (1+\epsilon)\|A - A_k\|_F$$

By using this equation, we can get the relative-error upper bound. The corresponding algorithm is shown in algorithm 3.

3 Algorithm

3.1 Randomized SVD

Algorithm 1 Basic Randomized SVD

procedure RANDSVD(A, k, l)
  Input: $m \times n$ matrix A with $m \geq n$, integers $k > 0$ for target rank and $n > l > k$
  Output: a rank-k approximation
  Draw a random matrix Ω with dimension $n \times l$
  Compute $Y = A\Omega$
  Compute an orthogonal column basis Q for Y with QR decomposition $Y = QR$
  Compute $B = Q^T A$
  Compute $B_k = \hat{U}_k \hat{\Sigma}_k \hat{V}_k^T$, the rank-k truncated SVD of B by solving the truncated SVD decomposition
  return QB_k as the approximate rank-k truncated SVD of A

This basic approach of randomized SVD are easy to implement. We use it as our baseline comparision.

Algorithm 2 Basic Subspace Iteration for SVD[2]

procedure RANDSVD(A, k, l, q, Ω)
  Input: $m \times n$ matrix A with $m \geq n$, integers $k > 0$ for target rank and $l \geq k$,
and $n \times l$ start matrix Ω
  Output: a rank-k approximation
  Compute $Y = (AA^T)^q A\Omega$
  Compute an orthogonal column basis Q for Y with QR decomposition $Y = QR$
  Compute $B = Q^T A$
  Compute $B_k = \hat{U}_k \hat{\Sigma}_k \hat{V}_k^T$, the rank-k truncated SVD of B by solving the truncated SVD decomposition
  return QB_k as the approximate rank-k truncated SVD of A

When we have the useful information of leading eigenvectors, it is preferred to use this method with the known orthogonal eigenvector matrix as an input to have faster convergence. We list this method here because it would be used as a subroutine in Algorithm 4 to improve the randomized subspace iteration method.

Algorithm 3 Randomized SVD with Subspace Iteration[2]

procedure RANDSVD(A, k, l, q)
  Input: $m \times n$ matrix A with $m \geq n$, integers $k > 0$ for target rank and $l \geq k$
  Output: a rank-k approximation
  Draw a random $n \times l$ start matrix Ω
  Compute $Y = (AA^T)^q A\Omega$
  Compute an orthogonal column basis Q for Y with QR decomposition $Y = QR$
  Compute $B = Q^T A$
  Compute $B_k = \hat{U}_k \hat{\Sigma}_k \hat{V}_k^T$, the rank-k truncated SVD of B by solving the truncated SVD decomposition
  return QB_k as the approximate rank-k truncated SVD of A

Note that Algorithm 1 is actually a special case of the randomized subspace iteration method when $q = 0$.The number of iteration q helps to converge faster when singular values do not decay very fast.

Algorithm 4 Improved Randomized Subspace Iteration

procedure RANDSVD(A, k, l_1, l_2, q)
  Input: $m \times n$ matrix A with $m \geq n$, integers $k > 0$ for target rank and $l_1 > l_2 \geq k$
  Output: a rank-k approximation
  Run Algorithm 1 with $l = l_1$ for a $rank - l_2$ approximation
  Set Ω to be approximate right singular vector matrix
  Run Algorithm 2 with Ω and $l = l_2$ for a $rank - k$ approximation

Algorithm 4 uses the useful information containing in the right singular vector matrix to further compute the rank-k approximated matrix.

Algorithm 5 Improved Randomized Subspace Iteration with Over-sampling Control

procedure RANDSVD(A, k, p, c, q)
  Input: $m \times n$ matrix A with $m \geq n$, integers $k > 0$ for target rank and $p, c, q \geq 0$
  Output: a rank-k approximation
  Draw a random $n \times (k + p + c)$ start matrix Ω
  Compute $Y = (AA^T)^q A\Omega$
  Compute an orthogonal column basis Q for Y with QR decomposition $Y = QR$
  Compute $B = Q^T A$
  Compute $B_k = \hat{U}_k \hat{\Sigma}_k \hat{V}_k^T$, the rank-k truncated SVD of B by solving the truncated SVD decomposition
  return QB_k as the approximate rank-k truncated SVD of A

For this algorithm, c allows a drastically different error bound and helps to control accuracy. p influences the failure chance[10].

3.2 CUR with Adaptive Sampling

The algorithm for near-optimal column selection algorithm is shown below:

Above all, the Dual Set Spectral-Frobenius Sparsification Algorithm is shown below:

So the adaptive sampling for CUR in shown in algorithm5.

The corresponding time complexity should be $\mathcal{O}((m+n)k^3\epsilon^{-2/3} + mk^2\epsilon^{-2} + nk^2\epsilon^{-4}) + T_{Multiply}(mnk\epsilon^{-1})$ to compute matrix C, U and R. With the algorithm is executed in a single-core processor, the time complexity of the CUR algorithm is linear in mn; when executed in multi-processor environment where matrix multiplication is performed in parallel, ideally the algorithm time only linear in $m+n$. The algorithm also save the storage space since it does not need to load the whole $m \times n$ data matrix A into RAM. More details please see the application part.

Algorithm 6 The Near-Optimal Column Selection Algorithm[4]

Input: a real Matrix $A \in \mathbb{R}^{m\times n}$, target k, error parameter $\epsilon \in (0,1]$, target column number $c = \frac{2k}{\epsilon}(1+o(1))$;
Compute approximate truncated SVD via random projection such that $A_k \approx \hat{U}_k\hat{\Sigma}_k\hat{V}_k^T$;
Construct $\mathcal{U} \leftarrow$ columns of $A_k \approx \hat{U}_k\hat{\Sigma}_k\hat{V}_k^T$; $\mathcal{V} \leftarrow \hat{V}_k^T$
Compute $s \leftarrow$Dual Set Spectral-Frobenius Sparsification Algorithm ($\mathcal{U}$, $\mathcal{V}$, $c-2k/\epsilon$);
Construct $C_1 \leftarrow ADiag(s)$, and then delete the all-zero columns;
Residual matrix $D \leftarrow A - C_1C_1^{\dagger}A$;
Compute sampling probabilities: $p_i = \|d_i\|_2^2/\|D\|_F^2$, $i = 1, ..., n$;
Sampling $c_2 = 2k/\epsilon$ columns from A with probability $p_1, ..., p_n$ to construct C_2;
return $C = [C_1, C_2]$.

Algorithm 7 Dual Set Spectral-Frobenius Sparsification Algorithm[4]

Input: Construct $\mathcal{U} = \{x_i\}_{i=1}^n \subset \mathbb{R}^l$, $(l > n)$; $\mathcal{V} = \{x_i\}_{i=1}^n \subset \mathbb{R}^k$, which $\sum_{i=1}^n v_i v_i^T = I_k$,$(k < n); k < r < n$;
Initialize: $s_0 = 0$, $A_0 = 0$;
Compute $\|x_i\|_2^2$ for $i = 1, 2, ...n$ and then computer $\delta_U = \frac{\sum_{i=1}^n \|x_i\|_2^2}{1-\sqrt{k/r}}$;
for $\tau = 0$ to $r-1$ **do**
    Compute the eigenvalue decomposition of A_τ;
    Find any index j in $1, ..., n$ and compute a weight $t > 0$ such that

$$\delta_U^{-1}\|x_j\|_2^2 \le t^{-1} \le \frac{v_j^T(A_\tau - (L_\tau+1)I_k)^{-2}v_j}{\phi(L_\tau+1, A_\tau) - \phi(L_\tau, A_\tau)} - v_j^T(A_\tau - (L_\tau+1)I_k)^{-1}v_j$$

    where $\phi(L, A) = \sum_{i=1}^k (\lambda_i(A) - L)^{-1}$ and $L_\tau = \tau - \sqrt{rk}$
    Update the j-th component of s_τ and A_τ: $s_{\tau+1}[j] = s_\tau[j] + t$ $A_{\tau+1} = A_\tau + tv_jv_j^T$;
end for
return $s = \frac{1-\sqrt{k/r}}{r}s_r$.

Algorithm 8 Adaptive Sampling for CUR[4]

Input: a real Matrix $A \in \mathbb{R}^{m\times n}$, target k, error parameter $\epsilon \in (0,1]$, target column number $c = \frac{2k}{\epsilon}(1+o(1))$, target row number $r = \frac{c}{\epsilon}(1+\epsilon)$;
Select $c = \frac{2k}{\epsilon}(1+o(1))$ columns of A to construct $C \in \mathbb{R}^{m\times c}$ using Algorithm 2;
Select $r_1 = c$ rows of A to construct $R_1 \in \mathbb{R}^{r_1\times n}$ using Algorithm 2;
Adaptively sample $r_2 = c/\epsilon$ rows from A according to the residual $A - AR_1 \dagger R_1$;
return C, $R = [R_1^T, R_2^T]^T$, and $U = C^{\dagger}AR^{\dagger}$.

4 Implementation issues

4.1 Metric of Measuring Accuracy

We define the error ratio as the follow:

$$\text{Error Ration} = \frac{||A - \tilde{A}||}{||A - A_k||}$$

where $\tilde{A}$ is the approximate truncated rank-k matrix and A_k is the rank-k matrix run by MATLAB code "svds()". Idealy, for randomized SVD, the error ratio is expected to converge to 1 quickly. If the ratio is lower than 1 or much smaller, it means the method outperforms the MATLAB routine of truncated SVD.The faster the error ratio converges to 1 or better, the better the algorithm is.

Besides, we prefer to use "2-norm"(spetral norm) as Golub and Van Loan point out that the spectral norm is comparatively robust to noise in the big data context.

4.2 Matrices in Experiments

We have used matrices we constructed for testing and also experiment on real-life matrix examples.

4.2.1 Matrices for Testing

We constructed our experiment matrices in two ways that indicated in Gu's paper[2] The first method is

$$A = (\log ||X_i - Y_j||_2)$$

where $\{X_i\}$ are n-dimensional Gaussian random variables with mean 0 and standard deviation 1 and where $\{Y_j\}$ are n-dimensional Gaussian random variable with mean μ and standard deviation 1. Note that i this way, μ is used to control the ratio of the two leading singular values of A, which means larger μ would lead to smaller ratio σ_2/σ_1 that indicates faster decay in eigenvalues.

The second approach also has the same form with the above expression while $\{X_i\}$ are equi-spaced points on the edge of the disc $||(X - \begin{pmatrix}-1\\-1\end{pmatrix})||_2 = \sqrt{2}$ and $\{Y_j\}$ are equi-spaced points on the edge of the disc $||(Y - \begin{pmatrix}2\\2\end{pmatrix})||_2 = 2\sqrt{2}$

4.2.2 Matrices for Applications

We use two matrix in the application part. The matrix are from matrix market. We are formiliar with it by ECS231. The infromation of the two matrix are shown below:

Data Set	Size	number of Nonzero	Source
west0479	479×479	1888	Matrix Market
MAHINDAS	1218×1258	7682	Matrix Market

4.3 Algorithmic Optimization

4.3.1 Randomized SVD

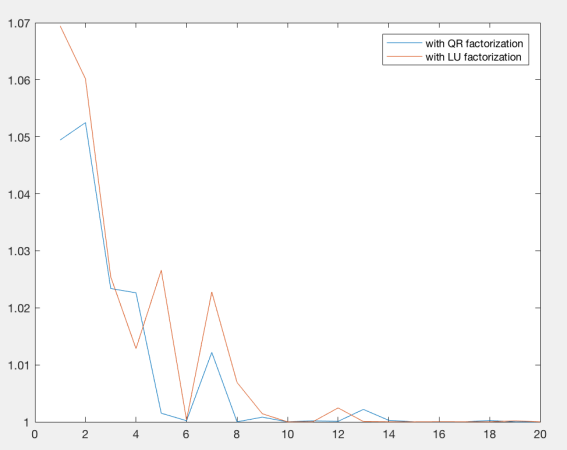
In practice, we need to be careful when implementing the subspace iteration because the computation would be prone to round-off errors. We can simply use QR factorization in each iteration as suggested in Algorithm A.1 in Gu's paper. However, Shabat et al.[8] observed that computing the LU decomposition is typically more efficient than computing the QR decomposition, and both are sufficient for most stages of the randomized algorithms for low-rank approximation[8].Thus, we also implement the LU decompositions for the most iterations except the last iteration in our MATLAB code. We would be interested to see how the trick performs comparing to applying the QR decomposition in each iteration.

4.3.2 CUR with Adaptive Sampling

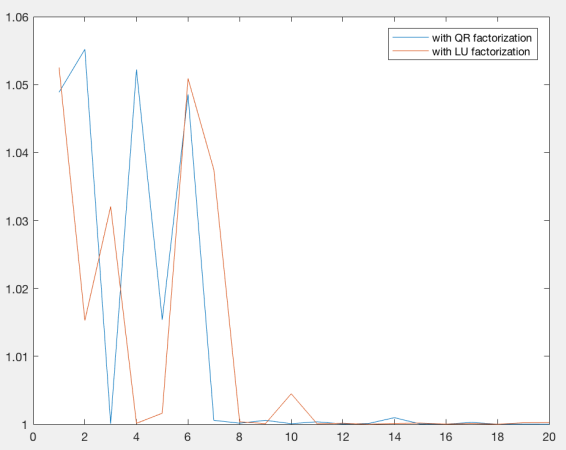
Since it is hard to compare CUR with Adaptive Sampling with randomized SVD (because we have different input and randomized SVD mainly used iteration, while CUR mainly select column and rows in two steps, so it is hard to control same variable to measure the performance of the two algorithm). So in this section, I compared the adaptive sampling based CUR algorithm(algorithm6,7,8) with the the deterministic sparse column-row approximation (SCTA) algorithm[9] and subspace sampling algorithm[7]. For SCRA, we used the code released by Stewart[9]. For each data set and each algorithm, we set k = 1, 3 or 10, and $c = ak$, $r = ac$ where a ranges in each set of experiments. We depict the error described above.

5 Experiment results

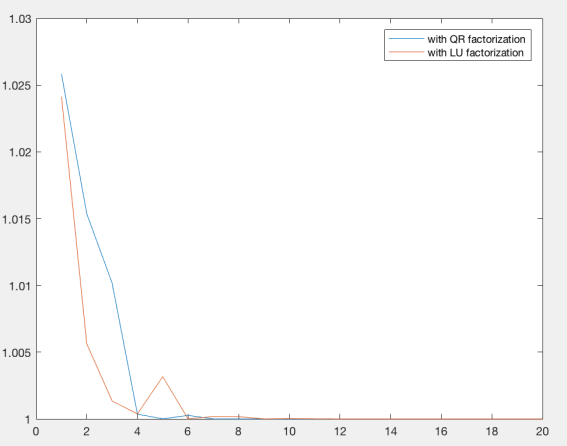
5.1 Compare Randomized Subspace Iteration on different settings



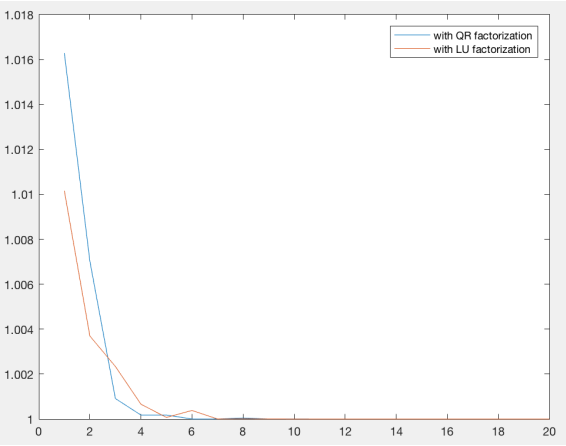
(a) $\mu = 1, l = 5$



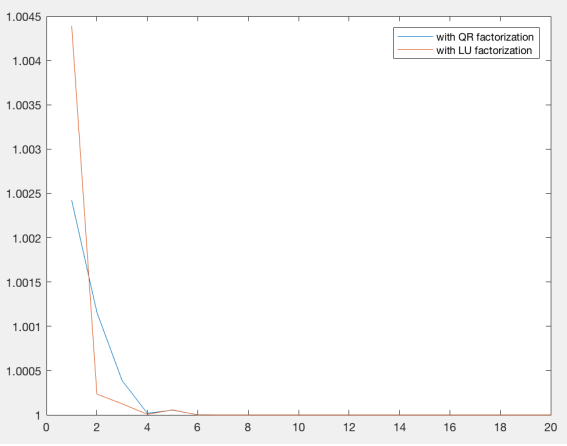
(b) $\mu = 2.5, l = 5$



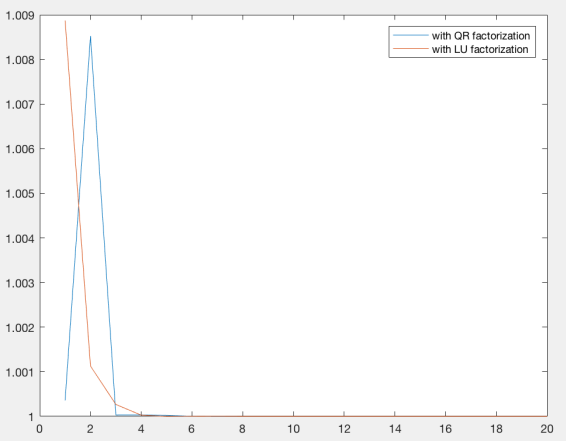
(c) $\mu = 1, l = 10$



(d) $\mu = 2.5, l = 10$



(e) $\mu = 1, l = 15$



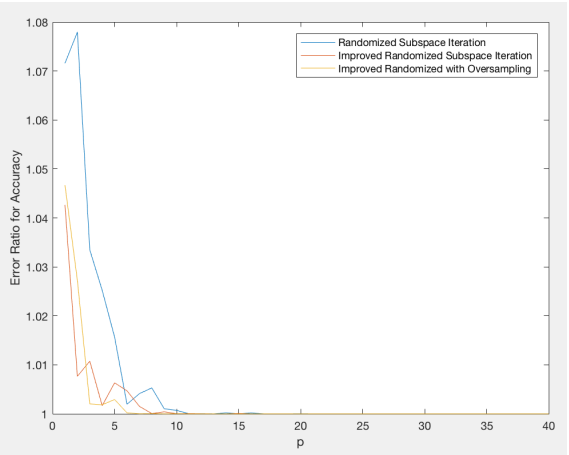
(f) $\mu = 2.5, l = 15$

Figure 1: Error Ratio For Different Eigenvalue Ratio and Columns settings

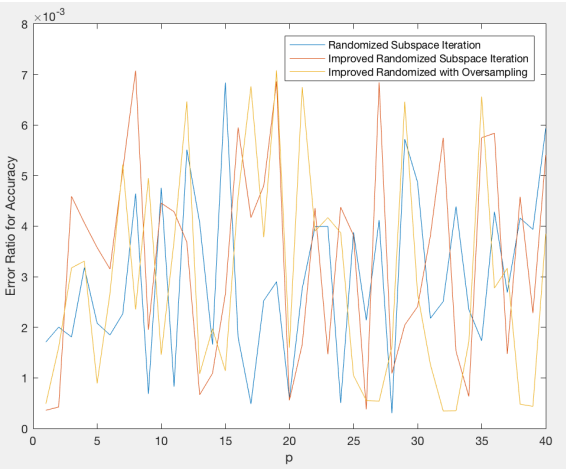
We first use Algorithm 3-randomized subspace iteration method to test on the 4000×100 matrix(generated with the first method discuss in Section 4.2.1) for different settings to get the rank-3 matrix approximation. From Figure 1, we can see that for the case with faster decaying eigenvalue as $\mu = 2.5$ indicates, the error ratio converges to 1 relatively faster than the case with slower decaying eigenvalue as $\mu = 1$, while it seems to less stable as showed in plots b) and f). This is consistent with Gu's conclusion[2] that the faster the singular values decay, the faster subspace iteration converges. On the other hand, from the plots and table of running time, we can see that using LU factorization seems to be a efficient technique that can achieve similarly good accuracy level, which is consistent to the observation from Shabat et al.[8]. Furthermore, we observe that as l increases, the pattern of convergence is more stable and faster.

5.2 Compare Improved Randomized SVD methods

In this experiment, we compare two improved randomized SVD algorithms to the baseline algorithm-randomized subspace method. We generate two matrices,the first case features slow decaying eigenvalues is generated by the first method in Section 4.2.1 with $\mu = 1$ and the second case features fast decaying eigenvalues is generated by the second method in Section 4.2.1. It is a bit unexpected at first to observe that the error ratio is much lower than 1 in the plot b). However, it makes sense as Gu explains in that case the randomized algorithm really out-perform the implmentation of "svds" in MATLAB. In the case of slow Decaying Eigenvalues, we can see that both improved randomized SVD methods have better accuracy performance than the randomized subspace iteration and converge to 1.



(a) Slow Decaying Eigenvalues



(b) Fast Decaying Eigenvalues

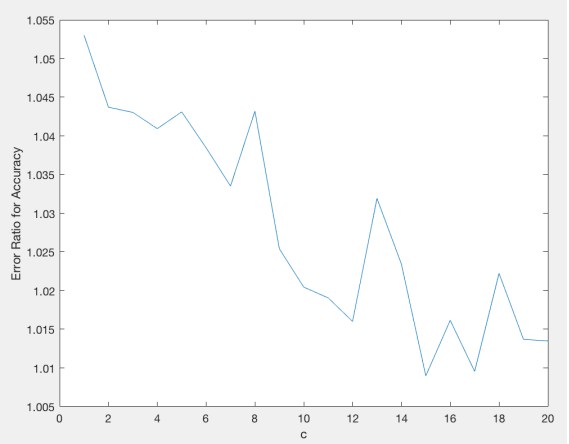
Figure 2: Error Ratio

Table 1: Running time Comparison

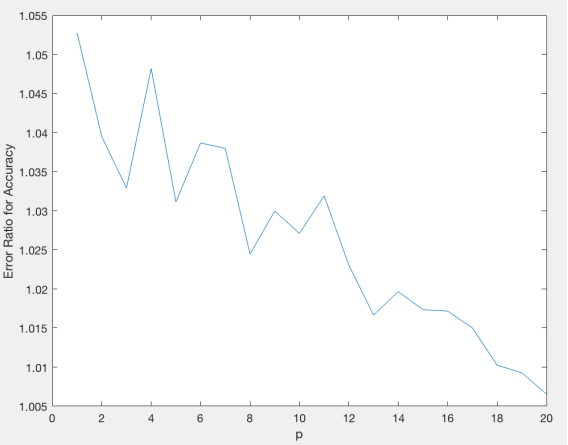
	l=5 $\mu = 1$	l=10 $\mu = 1$	l=15 $\mu = 1$
with QR factorization	0.0136	0.0221	0.0325
with LU factorization	0.0127	0212	0.0255

5.3 Compare Improved Randomized SVD method with Oversampling Control

To explore the effect of random sampling in Algorithm 5, we generate a 4000 × 4000 matrix with the first approach in Section 4.2.1 with $\mu = 1$. Then we compare the effects of the sampling parameters c and p. As we can see as c or q increases, the error ratio converges to 1. It seems with fixed c, larger p converges faster comparing to the other case.



(a) Different c with fixed p and q



(b) Different p with fixed c and q

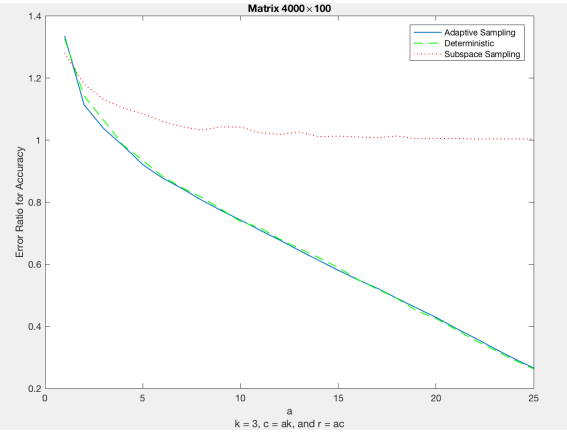
Figure 3: Error Ratio

5.4 Testing results for CUR with Adaptive Sampling

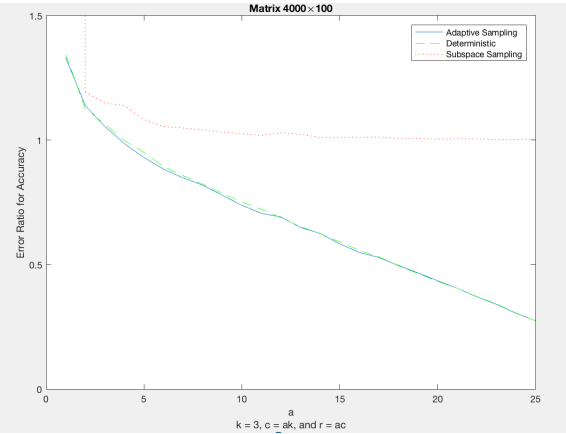
5.5 Testing for CUR

Corresponding to random SVD testing session, we test the CUR algorithm by using 4000×100, 4000×400, 4000×4000matrix(generated with the first method discuss in Section 4.2.1) for different settings to get the rank-3 matrix approximation. From Figure 4 below, we can see that all of the three algorithm has faster converge error ratio with larger decaying eigenvalue as $\mu = 2.5$ indicates, compared with slower decaying eigenvalue as $\mu = 1$. It is also consist with with Gu's conclusion that the faster the singular values decay, the faster subspace iteration converges. On the other hand, the relationship of the three algorithm results also correspond to the CUR matrix decomposition[2]. That is, compared with Deterministic algorithm and adaptive sampling, the subspace sampling has a relative high error ratio and it is not stable as the other two algorithm. However, it seems that the performance of Deterministic algorithm and adaptive sampling algorithm are pretty similar for the relative error ratio aspect.

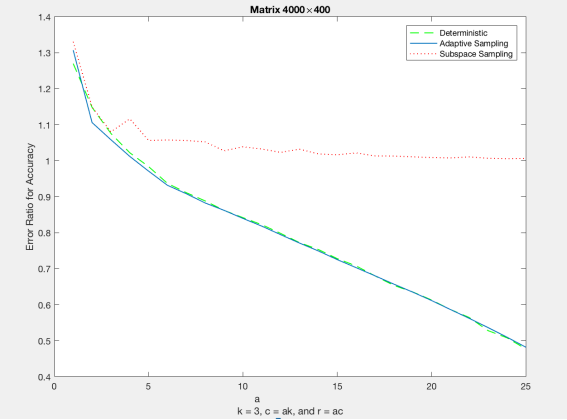
We should also notice that Deterministic algorithm and adaptive sampling can get the relative error ratio smaller than 1, which is different from random SVD. It is not hard to explain if we pay attention to the algorithm of CUR methods. It only use the results of random SVD in the middle steps, but the main part is to choose the columns and rows from the original matrix A. So if we choose enough number of columns and rows, the results of CUR methods can be very similar to the original matrix A. That is to say, CUR get a approximation of A, not A_k. So it can get a better approximation of A than truncated SVD. So the numerator part for the relative error ratio $\|A - \tilde{A}\|$ can be smaller than the denominator part $\|A - A_k\|$ of relative error ratio. it can get better error All of this indicates that the algorithm are right.



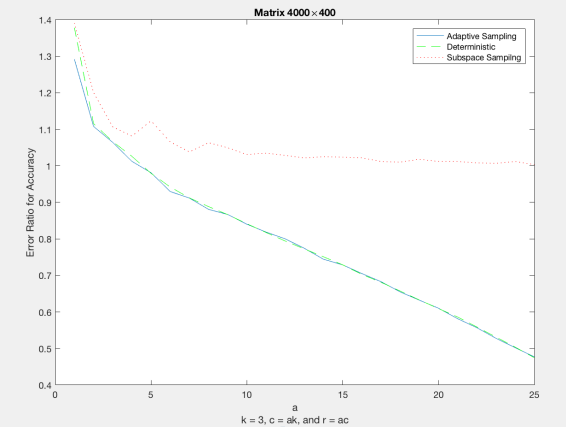
(a) $\mu = 1$



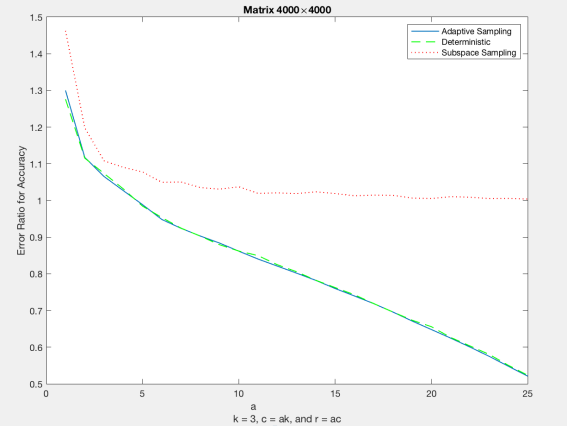
(b) $\mu = 2.5, l = 5$



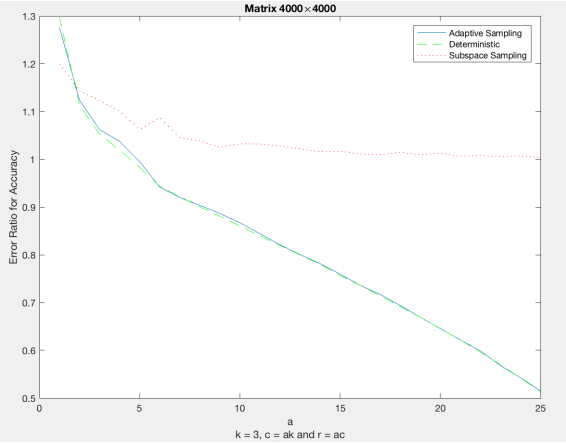
(c) $\mu = 1$



(d) $\mu = 2.5$



(e) $\mu = 1$



(f) $\mu = 2.5$

Figure 4: Error Ratio For Different Eigenvalue Ratio and Columns

5.5.1 Elapsed Time

The figure below shows the elapsed time for three algorithm with matrix 4000×4000, $k = 10$. From the plot we can see that the deterministic algorithm increase very fast with chosen more columns and rows. For adaptive algorithm and subspace algorithm, it also increases not as fast as the deterministic algorithm. The reults correspond to our theoretically analysis.

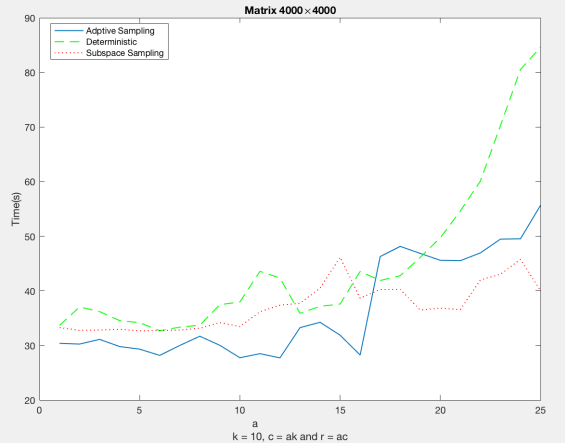
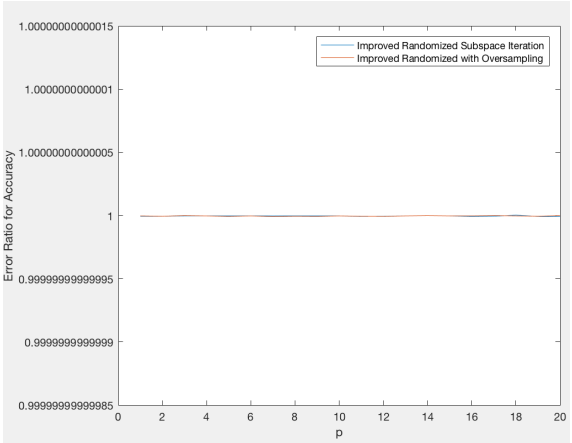


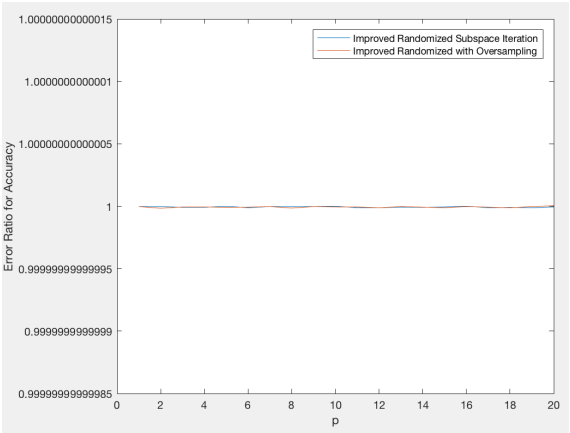
Figure 5: Time compare

5.6 Compare Randomized SVD and CUR methods on Real-life Matrix

We are also interested to see how two approaches perform and compare with each other. We used the dataset discussed in Section 4.2.2.



(a) $k = 1$

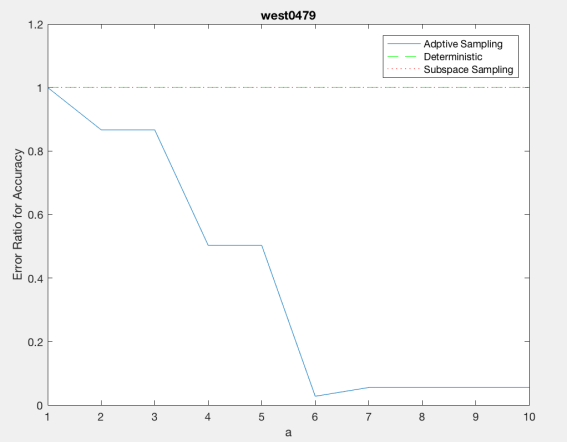


(b) $k = 3$

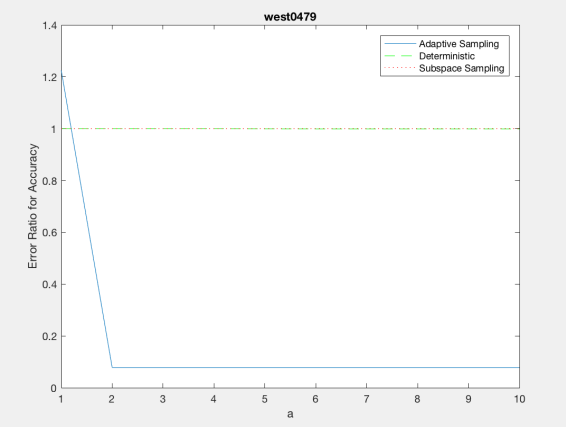
Figure 6: Results of the Randomized SVD on the west0479 data

The figure above shows that the random SVD has a great estimation for the matrix west0479. Since both of the random SVD are converge to 1 very fast with a relative error in machine precision level even with k = 1. For CUR method, the relative error ratio of deterministic algorithm and subspace algorithm are all converge to 1. One should see that the adaptive algorithm CUR has a better relative error ratio which is smaller than 1, and the relative error ratio is in the machine precision level. So it has a better estimation of A. The reason that all of the algorithm has good performance is probably because the west0479 are pretty small.

The above figure shows the performance of the matrix manhidasthat at the beginning of for different algorithm. For improved random SVD with oversampling, the relative error ratio are not stable at beginning when the rank k is big, but with iteration increasing, it converges fast. For improved randomized SVD with subspace iteration, it has a good performance whenever target rank k is small or big. If k is bigger, it converges faster. For CUR part, we can see that the the subspace sampling algorithm is not stable, especially when the number of chosen columns is small. But deterministic algorithm and adaptive sampling are performs better.

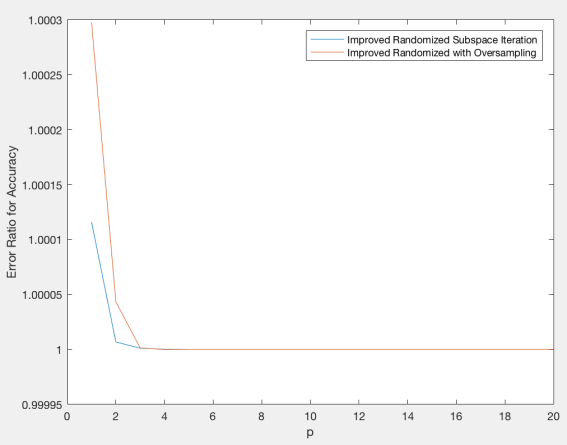


(a) $k = 1$

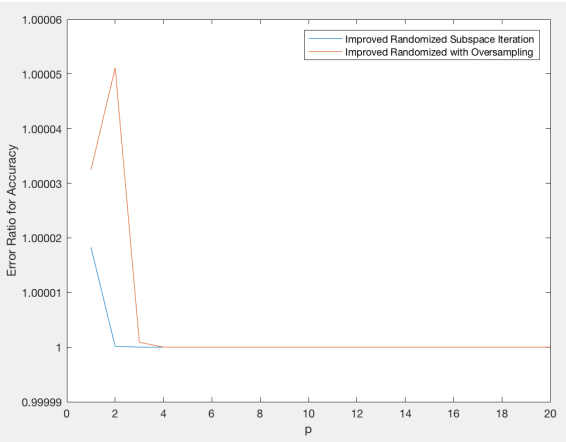


(b) $k = 3$

Figure 7: Results of the CUR algorithms on the west0479f data



(a) $k = 3$



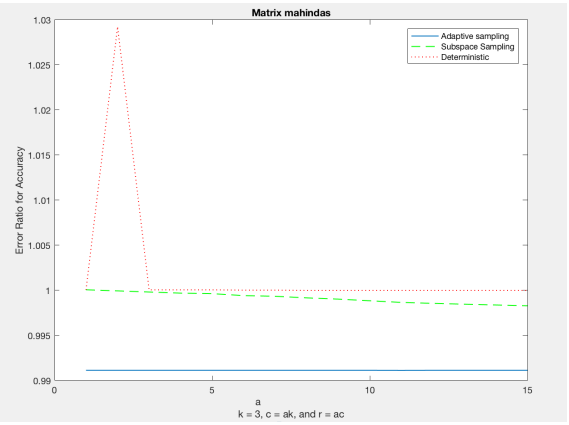
(b) $k = 10$

Figure 8: Results of the Randomized algorithms on the manhidas data

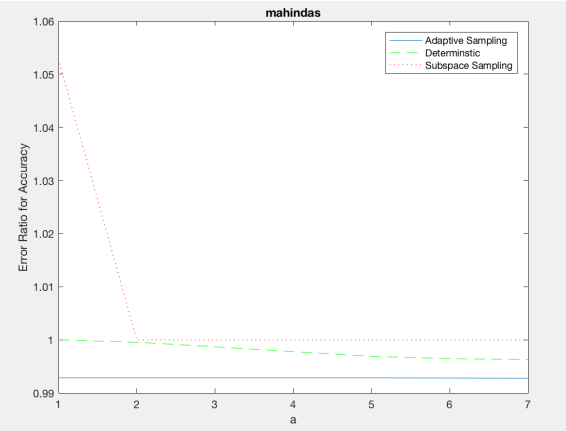
6 Conclusion

To summarize, in this project we investigated and implemented two types of randomized algorithms. Through our numerical experiments, we have showed that the randomized subspace iteration has accurate and reliable convergence results to approximate the truncated k-rank SVD matrix. The phenomenon of decaying eigenvalues can even greatly boosts the convergence performance of randomized algorithm in the subspace iteration. Furthermore, we delved into the implementation of subspace iteration to verify that the LU factorization can replace QR factorization in most iteration time with sacrificing little in convergence performance. Also we show that the subspace iteration and random sampling method are both effective to improve the accuracy and stability of randomized algorithm.

Comparing to randomized SVD algorithm, they have similarly good results. However, the best performance of random SVD is that, the estimation results will get very closer to the result of exact truncated SVD at a machine precision level. For CUR, the algorithm is to choose columns and rows from the original matrix A, so with the number of chosen columns and rows increasing, it will be more close to A. If there are enough number of columns and rows, it can have a better result then the exact truncated SVD, which means that the relative error rate will be smaller than 1. On the other hand, as pointed in the paper[2] and the session 5



(a) $k = 3$



(b) $k = 10$

Figure 9: Results of the CUR algorithms on the manhidas data

shows,it is often useful to find the low-rank matrix approximation which possess additional structures such as sparsity. Since west0479 and MAHINDAS are both sparse matrix, they apparently have a good performance than the testing matrix which is dense matrix. So CUR has a better performance in that aspect.

7 Further Discussion

There are several possible directions for us to improve our project. Due to the time constraint and computation power, we haven't delved too much into the real-world applications in large-scale matrices. Besides, we could research more about the metric and evaluation methods to better compare the randomized SVD and CUR methods.

Since SVD pr the standard QR decomposition for sparse matrix does not preserve sparsity in general, when the sparse matrix is large, computing or even storing such decompositions becomes challenging.

8 Acknowledgement

The authors would like to thank Prof.Bai for the inspiring teaching this quarter and talks on this topic. As teammates, we both appreciate each other to share insights and have discussion from different perspectives.

9 Reference

[1] Michael W. Mahoney, "Overview of RandNLA: Randomized Numerical Linear Algebra"
[2] Gu, M. (2015). Subspace Iteration Randomization and Singular Value Problems. SIAM J. Sci. Comput. SIAM Journal on Scientific Computing, 37(3). doi:10.1137/130938700
[3] Mahoney, Michael W. "Randomized algorithms for matrices and data." Foundations and Trends in Machine Learning 3.2 (2011): 123-224.
[4] Wang, Shusen, and Zhihua Zhang. "Improving CUR matrix decomposition and the Nyström approximation via adaptive sampling". The Journal of Machine Learning Research 14.1 (2013): 2729-2769.
[5]Deshpande, Amit, et al. "Matrix approximation and projective clustering via volume sampling. Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm. Society for Industrial and Applied Mathematics", 2006.
[6] Halko, Nathan, Per-Gunnar Martinsson, and Joel A. Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions." SIAM review 53.2 (2011): 217-288.
[7] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. Relativeerror CUR matrix decompositions. SIAM Journal on Matrix Analysis and Applications, 30(2):844–881, September 2008.

[8]Halko, N., Martinsson, P. G., Tropp, J. A. (2011). Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. SIAM Rev. SIAM Review, 53(2), 217-288. doi:10.1137/090771806
[8]Shabat,G., Shmueli,Y., and Averbuch.A. Randomized LU decomposition. Technical Report 1310.7202. arXiv.2013
[9]Stewart, G. W. "Four algorithms for the the efficient computation of truncated pivoted QR approximations to a sparse matrix." Numerische Mathematik 83.2 (1999): 313-323.
[10] Gu, M. Low-Rank Approximations, Random Sampling and Subspace Iteration

10 Appendix

```
##############main.m
function [A, X, Y] = creatmatrix(n,mu,r,c)
% e.g creatmatrix(8,2,10,8) ouputs 10,8 matrix
X = randn(n,r);
Y = mu+randn(n,c);
A = [];
for i = 1:r
    for j = 1:c
        A(i,j) = log(norm(X(:,i) - Y(:,j)));
    end
end

##############randSVD_SI.m
function [U, S, V]= randSVD_SI(A,k,l,q,useLU)
%inputs: A is a m,n matrix
%        k the target rank
%        l the interger to specify the number of column for a random
%         start matrix , l>=k
%        q number of iteration for power methods
[m, n] = size(A);
if m < n,
    disp('The input matrix should have number of rows m larger than or equal to number of c
    return
end
if l < k,
    disp('l should be larger than k');
    return
end
X = randn(n, l);
Y = A*X;
if(~useLU)
    [Q,R] = qr(Y, 0 ) ;
    for j = 1 : q
        Y = A'*Q;
        [Q,R] = qr(Y, 0 ) ;
        Y = A*Q;
        [Q,R] = qr(Y, 0 ) ;
    end
else
    [Q,R] = lu(Y);
    for it = 1:q
      Q = (Q'*A)';
```

```
        [Q,R] = lu(Q);
        Q = A*Q;
        if(it < q)
          [Q,R] = lu(Q);
        end
        if(it == q)
          [Q,R] = qr(Q,0);
        end
      end
end
clear R;

B = Q'*A;
[W, S ,V] = svd(B,'econ');
U = Q*W;

U = U(:,1:k);
S = S(1:k,1:k);
V=V(:,1:k);

###############

###############randSVD_impOS.m
function [U, S, V]= randSVD_impOS(A,k,q,p,c,useLU)
%inputs: A is a m,n matrix
%        k the target rank
%        p the interger that remains in control of failure chance, k+p=l
%        c the integer that allows a drastically different error bound,
%           controls accuracy,
%        q number of iteration for power methods
[m, n] = size(A);
if m < n,
    disp('The input matrix should have number of rows m larger than or equal to number of c
    return
end

l = k+p+c;
X = randn(n, l);
Y = A*X;
if(~useLU)
    [Q,R] = qr(Y, 0 ) ;
    for j = 1 : q
        Y = A'*Q;
        [Q,R] = qr(Y, 0 ) ;
        Y = A*Q;
        [Q,R] = qr(Y, 0 ) ;
    end
else
    [Q,R] = lu(Y);
    for it = 1:q
      Q = (Q'*A)';
      [Q,R] = lu(Q);
      Q = A*Q;
      if(it < q)
```

```
        [Q,R] = lu(Q);
      end
      if(it == q)
        [Q,R] = qr(Q,0);
      end
    end
end
clear R;

B = Q'*A;
[W, S ,V] = svd(B,'econ');
U = Q*W;

U = U(:,1:k);
S = S(1:k,1:k);
V=V(:,1:k);


#############randSVD_impSI.m
function [U, S, V]= randSVD_impSI(A,k,l1,l2,q,useLU)
%inputs: A is a m,n matrix
%        k the target rank
%        l1 the interger to specify the number of column for the first random
%          start matrix , l1>l2>=k
%        l2 the interger to specify the number of column for the second random
%          start matrix , l1>l2>=k
%        q  number of iteration for power methods
%        useLU boolean value to specify to use LU factorization to do
%        reorthogonalization

[m, n] = size(A);
if m < n,
    disp('The input matrix should have number of rows m larger than or equal to number of c
    return
end
if l2 < k||l1<l2
    disp('l2 should be larger than k and l1 should be larger than l2');
    return
end
[U1,S1,V1] = randSVD(A,l2,l1);
[U, S, V] = truncatedSVD_SI(A,k,l2,q,V1,useLU);

############creatematrix2.m
function [A, X, Y] = creatmatrix2(n,r,c)
% e.g creatmatrix2(200,10,10) ouputs 10,10 matrix
X = [];
Y = [];
A = [];

th = 0:pi/n:2*pi;
x1 = sqrt(2) * cos(th) - 1;
y1 = sqrt(2) * sin(th) - 1;
X(1,:) = x1;
X(2,:) = y1;
```

```
x2 = 2*sqrt(2) * cos(th) + 2;
y2 = 2*sqrt(2) * sin(th) + 2;
Y(1,:) = x2;
Y(2,:) = y2;

###############
function [idx] = AdaptiveSampling(prob, r)

n = length(prob);

iter = 0;
flag = false;
while true
    iter = iter + 1;
    % Remark: here the sampling strategy is not the same as that in the paper
    %           According to the paper, one should sample one index at a time using the fu
    %           However, the MATLAB function "mnrnd" appears to have bug, so here the imple
    selected = binornd(1, min(1, r * prob));
    selected = (selected == 1);
    if sum(selected) >= r
        break;
    end
    if iter > 20
        flag = true;
        break;
    end
end

index = 1:n;
idx = index(~~selected);

if flag == false
    idx = idx(1:r); % more than r columns are selected
end

end


###################
function [C,U,R]= ASCUR(A, k , a)
n = size(A,2);
%c = 2*k/epsi;
c = a*k;
%r = c/epsi*(1+epsi);
r = a*c;
C= NOCS(A, k, c);
r1 = c;
R1t = NOCS(A',k, r1);
R1 = R1t';
r2 = r - r1;
D = A - A*pinv(R1)*R1;
p = [];
normd2 = norm(D)^2;
```

```
for i = 1:n
    p(i) = D(:,i)'*D(:,i)/normd2;
end
r2 = round(r2);
index = AdaptiveSampling(p, r2);
R2 = A(index,:);
R = [R1',R2'];
U = pinv(C)*A * pinv(R);


#######################
function [A, X, Y] = creatmatrix(n,mu,r,c)
% e.g creatmatrix(8,2,10,8) ouputs 10,8 matrix
X = randn(n,r);
Y = mu+randn(n,c);
A = [];
for i = 1:r
    for j = 1:c
        A(i,j) = log(norm(X(:,i) - Y(:,j)));
    end
end
#######################
function out = deterministic(in)

c = in.c;
r = in.r;
p = in.p;

cidx = zeros(c,1);
ridx = zeros(r,1);

%tic;
[~,~,Va]=svds(in.A,p);
cidx(:,1) = MSelect(Va(:,1:p),p,c);
C = in.A(:,cidx);

[~,~,Va]=svds(in.A',p);
idx21 = MSelect(Va(:,1:p),p,r);
ridx(:,1) = idx21;

R = in.A(ridx,:);
%out.construct_time(1,1) = toc;

%tic
[Qc,~] = qr(C,0);
[Qr,~] = qr(R',0);

B = Qc'*in.A*Qr;
CUR = Qc*B*Qr';
[Ub,Sb,Vb] = svds(B,in.k);
Bk = Ub*Sb*Vb';
CUR_k = Qc*Bk*Qr';

%residual = in.A-CUR;
```

```
%residual_k = in.A - CUR_k;
out.C = Qc;
out.U = Bk;
out.R = Qr';

%out.metric_computing_time(1,1) = toc;


end


##########################
function s=DSS(V,U,r);
%Copyright Malik Magdon-Ismail


[k,n]=size(V);
[l,m]=size(U);
if(m~=n)
        s='dimensions do not match'
        return
end
%if(r<=k)
%       s='k<r please'
%       return
%end

s=zeros(n,1);
Atau=zeros(k,k);
sumx = 0;
for i = 1:n
    sumx = sumx + U(:,i)'*U(:,i);
end
%itaU = sumx/(1 - sqrt(k/r))

%Btau=zeros(l,l);
delL=1;
delU= sumx/(1-sqrt(k/r));
Ik=eye(k);
Il=eye(l);

for tau=0:r-1;
        Ltau=tau-sqrt(r*k);
        Utau=delU*(tau+sqrt(r*l));
        ALinv=inv(Atau-(Ltau+delL)*Ik);
        ALinv2=ALinv*ALinv;
        lamAtau=svd(Atau);
        phiL1=sum(1./(lamAtau-Ltau-delL));
        phiL2=sum(1./(lamAtau-Ltau));
        %BUinv=inv((Utau+delU)*Il-Btau);
        %BUinv2=BUinv*BUinv;
        %lamBtau=svd(Btau);
        %phiU1=sum(1./(Utau-lamBtau));
        %phiU2=sum(1./(Utau+delU-lamBtau));
```

```
        for i=1:n
                vi=V(:,i);
                ui=U(:,i);
                Lower=vi'*ALinv2*vi/(phiL1-phiL2)-vi'*ALinv*vi;
                Upper= 1/delU * (U(:,i)'*U(:,i));
                if(Upper<=Lower)
                        t=2/(Upper+Lower);
                        s(i)=s(i)+t;
                        Atau=Atau+t*vi*vi';
                        %Btau=Btau+t*ui*ui';
                        break;
                end
        end
end

s=s*(1-sqrt(k/r))/r;
#################################
function C= NOCS(A, k, c)
n = size(A,2);
%c = 2*k/epsi*(1+0.01);
epsi = 2*k/c*(1+0.01);
[U, S, V]= randProjSVD(A, k , 3);
U = A - U*S*V;
V = V';
s = DSS(V,U,round(c - 2*k/epsi));
C1 = A*diag(s);
C1( :, all( ~any( C1 ), 1 )) = [];
D = A - C1*pinv(C1)*A;
p = [];
normd2 = norm(D)^2;
for i = 1:n
    p(i) = D(:,i)'*D(:,i)/normd2;
end
c2 = 2*k/epsi;
c2 = round(c2);
index = AdaptiveSampling(p, c2);
C2 = A(:,index);
C = [C1,C2];

end

#################################
function [U, S, V]= randProjSVD(A, k , q )
[m n ] = size(A);
O = randn( n , k );
Y = A*O;
[Q R] = qr(Y, 0 ) ;
for j = 1 : q
Y = A'*Q;
[Q R] = qr(Y, 0 ) ;
Y = A*Q;
[Q R] = qr(Y, 0 ) ;
end
B = Q'*A;
```

```
[U, S ,V] = svd(B);
U = Q*U;
%Ak = U*S*V;

###################################
function [C, G, R] = SkeletonApproximation(A, k, c,r)

[U, S, V] = svds(A,k);

% Construct the matrix C containing r columns from A.
[qV, rV, pV] = qr(V', 0);
C = A(:, pV(1:c));

% Construct the matrix R containing r rows from A.
[qU, rU, pU] = qr(U', 0);
R = A(pU(1:r),:);

% Construct the matrix G. We set G as the solution of the following problem
%                          min_G || A - CGR ||_F.
% This choice of G is in general different from the one in [1]. We chose
% this G since we find its construction simpler than the one in [1] and we
% believe that - in practice - is as ''good'' as the one in [1].
G = pinv(C, .05) * A * pinv(R, .05);

###################################
function [C, U, R] = subspace_expected(A,k,c,r)

[Ua,~,Va]=svds(A,k);


cidx = CX_SubspaceExpected(Va, k, c);
C = A(:,cidx);

ridx = CX_SubspaceExpected(Ua, k, c);
R = A(ridx,:);

[Qc,~] = qr(C,0);
[Qr,~] = qr(R',0);

B = Qc'*A*Qr;
CUR = Qc*B*Qr';
[Ub,Sb,Vb] = svds(B,k);
U = Ub*Sb*Vb';
C = Qc;
R = Qr';

end


############################
#############main
clear
load mahindas.mat
%load west0479.mat
```

```
A = Problem.A;
A = full(A);
%for kk = 1:100

%[A, X, Y] = creatmatrix(100,2.5,4000,4000);
k = 10;
[Uk,Sk,Vk] = svds(A,k);
Ak = Uk*Sk*Vk';
[C,U,R]= ASCUR(A, k, 10);
Acur = C*U*R;

ErrorRatio = [];
TimeCost = [];
for a = 1:15
    t0=clock;
    [C,U,R]= ASCUR(A, k, a);
    TimeCost(a)=etime(clock,t0);
     Acur = C*U*R;
     ErrorRatio(a) = norm(A - Acur)/norm(A - Ak);
end

ErrorRatioD = [];
TimeCostD = [];
for a = 1:10
c = a*k;
r = a*a*k;
t0=clock;
[C, G, R] = SkeletonApproximation(A, k, c,r);
TimeCostD(a)=etime(clock,t0);
Acur = C*G*R;
ErrorRatioD(a) = norm(A - Acur)/norm(A - Ak);
end

ErrorRatioS = [];
TimeCostS = [];
for a = 1:10
c = a*k;
r = a*a*k;
t0=clock;
[C, U, R] = subspace_expected(A,k,c,r);
TimeCostS(a)=etime(clock,t0);
Acur = C*U*R;
ErrorRatioS(a) = norm(A - Acur)/norm(A - Ak);
%end

end

plot(1:7,ErrorRatio(1:7)-0.4,1:7,ErrorRatioD(1:7),'--g',1:7,ErrorRatioS(1:7),':r')
title('mahindas')
xlabel('a')
ylabel('Error Ratio for Accuracy')

######################main testing
% First experiment:
```

```
% create a 4000,4000 matrix with mu=1 such that the ratio for the first
% two eigenvalues is large
%[A, X, Y] = creatmatrix(100,1,4000,100);

%k=3

% compute the exact truncated SVD
%[Uk,Sk,Vk] = svds(A,k);
%Ak = Uk*Sk*Vk';
% compute the approximate rank-k SVD
%[C,U,R]= ASCUR(A, k , 1);
%Acur = C*U*R;
% use Error ratio defined as follows to access the accuracy for our
% algorithms
%ErrorRatio = norm(A - Acur,'fro')/norm(A - Ak, 'fro');

%For Randomized SVD algorithms
%with mu = 1 for large eigenvalues ratio; and mu = 2.5 for small eigenvalue
%ratio
%Case 1:
%For fixed k=1, plot the error ratios convergence varying on p, randSVD_SI
%with LU reorthogonalization or QR reorthognalization
% k = 3;
% l = 15; %we tested on l=5,10,15
% [Uk,Sk,Vk] = svds(A,k);
% Ak = Uk*Sk*Vk';
% t1 = [];
% t2 = [];
% eps_ls1 = [];
% eps_ls2 = [];
% for q = 1:20
%       tStart1=tic;
%       [U,S,V]= randSVD_SI(A,k,l,q,false);
%       tElapsed1=toc(tStart1);
%       t1 = [t1 tElapsed1];
%       A_approx1 = U*S*V';
%       tStart2=tic;
%       [U,S,V]= randSVD_SI(A,k,l,q,true);
%       tElapsed2=toc(tStart2);
%       t2 = [t2 tElapsed2];
%       A_approx2 = U*S*V';
%       ErrorRatio1 = norm(A - A_approx1)/norm(A - Ak);
%       ErrorRatio2 = norm(A - A_approx2)/norm(A - Ak);
%       eps_ls1 = [eps_ls1 ErrorRatio1];
%       eps_ls2 = [eps_ls2 ErrorRatio2];
% end
%
% plot(1:20,eps_ls1,1:20,eps_ls2)
% legend('with QR factorization','with LU factorization')

%calculate average running time
% mean(t1) %for using QR factorization  0.0136, 0.0221, 0.0325 s|0.0131 0.0229 0.0355
% mean(t2) %for using LU factorization  0.0127, 0.0212, 0.0255 s|0.0142 0.0204 0.0303
```

```
%Case 2:
% a)
[A, X, Y] = creatmatrix2(4000,4000,100);
k= 10;
tStart=tic;
[Uk,Sk,Vk] = svds(A,k);
tElapsed=toc(tStart1);
Ak = Uk*Sk*Vk';

%b)
[A, X, Y] = creatmatrix(100,1,4000,100);
k= 10;
tStart=tic;
[Uk,Sk,Vk] = svds(A,k);
tElapsed=toc(tStart1);
Ak = Uk*Sk*Vk';

%c)
[A, X, Y] = creatmatrix(100,4,4000,100);
k= 10;
tStart=tic;
[Uk,Sk,Vk] = svds(A,k);
tElapsed=toc(tStart1);
Ak = Uk*Sk*Vk';
%For fixed k=10, plot the error ratios convergence varying on iteration q, comparing
%randSVD_SI(l=15), randSVD_impSI(l1=20,l2=15,useLU=false), and randomSVD_impOS(p=5,c=5)
eps_ls1 = [];
eps_ls2 = [];
eps_ls3 = [];
t1 = [];
t2 = [];
t3 = [];
l1 = 20;
l2 =15;
p = 5;
c = 5;
tStart=tic;
[Uk,Sk,Vk] = svds(A,k);
tElapsed=toc(tStart1);
Ak = Uk*Sk*Vk';
for q = 0:40
      tStart1=tic;
      [U1,S1,V1]= randSVD_SI(A,k,l,q,true);
      tElapsed1=toc(tStart1);
      t1 = [t1 tElapsed1];
      tStart2=tic;
      [U2,S2,V2]= randSVD_impSI(A,k,l1,l2,q,true);
      tElapsed2=toc(tStart2);
      t2 = [t2 tElapsed2];
      tStart3 = tic;
      [U3,S3,V3]= randSVD_impOS(A,k,q,p,c,true);
      tElapsed3=toc(tStart3);
      t3 = [t3 tElapsed3];
```

```
    A1_approx = U1*S1*V1';
    A2_approx = U2*S2*V2';
    A3_approx = U3*S3*V3';
    ErrorRatio1 = norm(A - A1_approx)/norm(A - Ak);
    ErrorRatio2 = norm(A - A2_approx)/norm(A - Ak);
    ErrorRatio3 = norm(A - A3_approx)/norm(A - Ak);
    eps_ls1 = [eps_ls1 ErrorRatio1];
    eps_ls2 = [eps_ls2 ErrorRatio2];
    eps_ls3 = [eps_ls3 ErrorRatio3];
end

plot(1:40,eps_ls1(2:41),1:40,eps_ls2(2:41),1:40,eps_ls3(2:41))
legend('Randomized Subspace Iteration','Improved Randomized Subspace Iteration','Improved
xlabel('p') % x-axis label
ylabel('Error Ratio for Accuracy') % y-axis label

%
% %calculate average running time  svds: 187.9236 | 224.1788 | 365.8224
mean(t1) %for using QR factorization  0.0421 | 0.0401 | 0.0442
mean(t2) %for using LU factorization  0.0446 | 0.0435 | 0.0473
mean(t3) %for using LU factorization  0.0485 | 0.0471 | 0.0509


%Case 3:
%For randomSVD_impOS method, plot the error ratios convergence varying on c
%to see how the accuracy changes
%a) Fixed p and q, different c
A = creatmatrix(100,2,4000,4000);
k = 50;
[Uk,Sk,Vk] = svds(A,k);
Ak = Uk*Sk*Vk';
p = 5;
q = 5;
t1 = [];
eps_ls1 = [];

for c = 1:20
    tStart1=tic;
    [U,S,V]= randSVD_impOS(A,k,q,p,c,true);
    tElapsed1=toc(tStart1);
    t1 = [t1 tElapsed1];
    A_approx1 = U*S*V';

    ErrorRatio1 = norm(A - A_approx1)/norm(A - Ak);

    eps_ls1 = [eps_ls1 ErrorRatio1];
end

plot(1:20,eps_ls1)
%legend('with QR factorization','with LU factorization')
xlabel('c') % x-axis label
ylabel('Error Ratio for Accuracy') % y-axis label

%b) Fixed q and c, different p
```

```
c = 5;
q = 5;
t1 = [];
eps_ls1 = [];

for p = 1:20
      tStart1=tic;
      [U,S,V]= randSVD_impOS(A,k,q,p,c,true);
      tElapsed1=toc(tStart1);
      t1 = [t1 tElapsed1];
      A_approx1 = U*S*V';

      ErrorRatio1 = norm(A - A_approx1)/norm(A - Ak);

      eps_ls1 = [eps_ls1 ErrorRatio1];
end

plot(1:20,eps_ls1)
%legend('with QR factorization','with LU factorization')
xlabel('p') % x-axis label
ylabel('Error Ratio for Accuracy') % y-axis label

%Case 4 real application   work
A = Problem.A;
A = full(A);
k =10;
tStart=tic;
[Uk,Sk,Vk] = svds(A,k);
tElapsed=toc(tStart);
Ak = Uk*Sk*Vk';
eps_ls1 = [];
eps_ls2 = [];
t1 = [];
t2 = [];
l1 = k+20;
l2 =k+10;
p = 10;
c = 10;
for q = 0:20
      tStart1=tic;
      [U1,S1,V1]= randSVD_impSI(A,k,l1,l2,q,true);
      tElapsed1=toc(tStart1);
      t1 = [t1 tElapsed1];
      tStart2 = tic;
      [U2,S2,V2]= randSVD_impOS(A,k,q,p,c,true);
      tElapsed2=toc(tStart2);
      t2 = [t2 tElapsed2];

      A1_approx = U1*S1*V1';
      A2_approx = U2*S2*V2';

      ErrorRatio1 = norm(A - A1_approx)/norm(A - Ak);
      ErrorRatio2 = norm(A - A2_approx)/norm(A - Ak);
```

```
    eps_ls1 = [eps_ls1 ErrorRatio1];
    eps_ls2 = [eps_ls2 ErrorRatio2];

end

plot(1:20,eps_ls1(2:21),1:20,eps_ls2(2:21))
legend('Improved Randomized Subspace Iteration','Improved Randomized with Oversampling')
xlabel('p') % x-axis label
ylabel('Error Ratio for Accuracy') % y-axis label
```