



MIDDLE EAST TECHNICAL UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING



SUMMER PRACTICE REPORT CENG 400

STUDENT NAME	Sait Elmas
ORGANIZATION NAME	METU Computer Engineering Department Parallel Processing Laboratory
ADRESS	Dept. of Computer Engineering Middle East Technical University Inonu Bulvari, 06531 Ankara, TURKEY
START DATE	1 August 2022
END DATE	12 September 2022
TOTAL WORKING DAYS	30 days

STUDENT'S SIGNATURE

ORGANIZATION APPROVAL

Contents

1	Introduction	2
2	Information About Project	2
2.1	Analysis Phase	3
2.1.1	Basic Linear Algebra	3
2.1.2	Matrix Factorization	4
2.1.3	Matrix Norms (Operator Norms)	5
2.1.4	Matrix Norms in Matlab and Julia	6
2.2	Design Phase	6
2.3	Implementation Phase	8
2.4	Testing Phase	11
2.4.1	Test of Algorithm 4.1	11
2.4.2	Test of Algorithm 4.2	14
2.4.3	Test of Algorithm 4.3	16
2.4.4	Algorithm 4.5	20
3	Conclusion	23

1 Introduction

In the summer practice period I worked in the **Parallel Processing Laboratory** of Middle East Technical University, Computer Engineering Department. Active research areas are listed on the web page of laboratory as:

- High Performance Computing
- Development of parallel algorithms and their applications in science and engineering
- Parallel sparse matrix computations
- Parallel application development and run-time environments

Beside these research areas there are courses that are being lectured by the members of laboratory to the undergraduate/graduate level students. Some of the courses are:

- CENG371 – Scientific Computing
- CENG478 – Introduction to Parallel Computing
- CENG487 – Introduction to Quantum Computing
- CENG571 – Numerical Analysis
- CENG577 – Parallel Computing
- CENG780 – Sparse Matrix Computations

After my application for summer practice internship and undergraduate student researcher positions, **Prof. Murat Manguoğlu** has offered the topic of "random sparse matrix factorization".

A matrix (array) that contains "a lot of" zeros is called a **sparse matrix (array)**. Although the ratio of zeros to the non-zeros is not strictly defined for a sparse matrix it will not be wrong to say that for a large matrix if the number of zeros are equal to the number of columns or rows that it is a sparse matrix.

In scientific computations the deterministic matrix algorithms are lack to solve the problems containing sparse matrices. One reason of that is these large matrices containing mostly zeros require many floating point operations that will easily overwhelming the capacity of even very powerful workstations. Another reason of this challenge is large matrices takes so much space to fit in the primary memory so there will be need more that one passes to gain the information from the disc and after the application of the relevant algorithm writing it to the disk back. In this situation the performance is measured by the amount of disk access.

For the first problem mentioned above "random methods" are in use. The idea is to sample the given matrix in such a way that the relatively simple sample contains *nearly* all information of the sparse matrix. That is it spans nearly same linear space with the original sparse matrix space. With this simple representative deterministic matrix methods of software packets can be utilized. For the second problem "parallel processing" algorithms are mainly in focus. This topic has not been studied under this summer practice period.

2 Information About Project

As mentioned in the introduction section "random methods for sparse matrix factorization" has been studied as a solution of the first major problem of sparse matrix computations. I had worked on this method by applying the algorithms that are defined in the article **"Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions"** by **N. Halko, P. G. Martinson, J. A. Tropp**. For the applications the **"Julia Language"** utilized.

The progress of this work has been supervised by Prof. Murat Manguoğlu via periodical meetings.

2.1 Analysis Phase

At this stage a review of basic linear algebra, an introductory course about numerical linear algebra and tutorials for julia-language has been studied.

2.1.1 Basic Linear Algebra

In this part some preliminary linear algebra subjects are listed. Although they are very basic it may be desirable to see all the terms that will be used at this report in one place.

This part is heavily depend on the book *Linear Algebra by Stephan H. Friedberg* and some related **wikipedia** pages.

Definition 2.1 (Vector Space). A **vector space** V over a field F (we mainly work with the fields of real and complex numbers (\mathbb{R}, \mathbb{C}) , so no need to define what a field is!) consist of two sets V (set of vectors), F (the field) and two operations $+$ (addition) and \cdot (scalar multiplication). Addition is defined on the set V that is we want to add the vectors to each other. And scalar multiplication is the relation between vectors and the field \mathbb{R} or \mathbb{C} , all together with the following properties:

For any vector v, v_1, v_2, v_3 in V and for any number a, b in \mathbb{R}

- **Commutative** vector addition: $v_1 + v_2 = v_2 + v_1$
- **Associative** vector addition: $v_1 + (v_2 + v_3) = (v_1 + v_2) + v_3$
- There must be a **0 element** satisfying: $0 + v = v + 0 = v$
- Every vector v should have an **inverse** e such that $v + e = 0$
- $1v = v1 = v$
- $(ab)v = a(bv)$
- $(a + b)v_1 = av_1 + bv_1$
- $a(v_1 + v_2) = av_1 + av_2$

\mathbb{R}^n is an example for a vector space over \mathbb{R} . $(1, 2, 3) + (2, 3, 4) = (2, 3, 4) + (1, 2, 3)$ for the vectors in \mathbb{R}^3 for instance and $(2 * 3) \cdot (1, 2, 3, 4) = 2 \cdot (3, 6, 9, 12)$ in \mathbb{R}^4 . All the properties can be easily verified for this natural vector space.

An other basic example is the set of all complex valued $m \times n$ dimensional matrices over the field of complex numbers \mathbb{C}

Definition 2.2 (Inner Product). Let V be a vector space over a field $(\mathbb{R} \text{ or } \mathbb{C})$. An **inner product** on V is a function that takes two vectors and gives a number from the field. It is usually denoted as $\langle x, y \rangle$ or just $x \cdot y$ with the following properties hold:

- (a) $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
- (b) $\langle cx, y \rangle = c\langle x, y \rangle$
- (c) $\overline{\langle x, y \rangle} = \langle y, x \rangle$. here the bar represents the complex conjugate of the number. If the field is the real numbers than it can be ignored.
- (d) $\langle x, x \rangle > 0$ if $x \neq 0$

An example of inner product is the standard Hermitian product on \mathbb{C}^n as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{j=1}^n x_j \bar{y}_j \text{ where } \mathbf{x} = (x_1, x_2, \dots, x_n), \mathbf{y} = (y_1, y_2, \dots, y_n)$$

Definition 2.3 (Inner Product Space). A vector space over a field equipped with an inner product is called an **inner product space**

Definition 2.4 (Norm). Let V be a vector space over real or complex numbers. **Norm** of a vector is a function that assigns a real number to every vector in the vector space. It is denoted as $\|\mathbf{x}\|$. The function should satisfy following three properties.

1. **Triangle inequality:** $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$
2. $\|r\mathbf{x}\| = |r| \|\mathbf{x}\|$
3. $\|\mathbf{x}\| \geq 0$ for any \mathbf{x} and $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = 0$

In different sources norms are denoted in different symbols and even with different names so it may be confusing while reading technical documents. Here is a list of basic norms with their definitions. For $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$

- **l_1 norm** (*Taxicab norm, Manhattan norm*): $\|\mathbf{x}\|_1 = \sum_{j=1}^n |x_j|$
- **l_2 norm:** $\|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^n |x_j|^2}$. This is the standart **Euclidian Distance**. With l_2 norm and standart inner product of vectors of \mathbb{R}^n has the canonical relation as $\|\mathbf{x}\|_2 = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$
- **l_p norm** (p norm): $\|\mathbf{x}\|_p = \left(\sum_{j=1}^n |x_j|^p \right)^{1/p}$
- **l_∞ norm** (*Infinity norm, maximum norm*) $\|\mathbf{x}\|_\infty = \max_i |x_i|$

Definition 2.5. For an inner product space V if the product of two vectors \mathbf{x} and \mathbf{y} , $\langle \mathbf{x}, \mathbf{y} \rangle = 0$ than we say that \mathbf{x} and \mathbf{y} are **orthogonal** (perpendicular) to each other. If length of a vector is 1 that is $\|\mathbf{x}\|_2 = 1$ than we call it **unit vector**. If a set of vectors are mutually orthogonal to each other this set is called orthogonal. Additionally an orthogonal set with every vector has length 1 is called **orthonormal** set.

All of these terms can be extended to define matrices with its columns are regarded as column vectors. A matrix is called orthogonal if all of its columns are mutually perpendicular and orthonormal if length of each column vector is 1 and they are orthogonal.

Definition 2.6 (Rank of a matrix). The **rank of a matrix** is the number of its columns that are linearly independent.

Definition 2.7 (Orthogonalization). Orthogonalization of a matrix A is finding a matrix say Q , that is orthonormal and that can span the same space with A . Q has $\text{rank}(A)$ number of columns.

In Matlab the **orth** command draws an orthonormal matrix for a given matrix A . As a simple function that computes the orthonormal matrix Q for a given matrix can be scripted as below. The method behind this function is called **Gram-Schmidt** orthogonalization process.

Definition 2.8 (Unitary Matrix). A square matrix A is called **unitary** if $AA^H = A^H A = I$ where A^H denotes the **conjugate transpose** of A .

Definition 2.9 (Normal Matrix). A square matrix A is called **normal** if $AA^H = A^H A$.

Definition 2.10 (Hermitian Matrix). A square matrix A is called **Hermitian** if $A = A^H$.

2.1.2 Matrix Factorization

Writing a matrix as a product of two relatively "simple" matrices may simplify complicated and expensive calculations. Here are some basic matrix factorizations.

Eigendecomposition If a square matrix A is diagonalizable than it can be written as $A = QDQ^{-1}$ where Q contains eigenvectors as its columns and D is a diagonal matrix containing the eigenvalues of A on the diagonal entries. One of the important properties of this decomposition is powers of the matrix A can be calculated as:

$$A^n = (QVQ^{-1})^n = QVQ^{-1}QVQ^{-1} \dots QVQ^{-1}QVQ^{-1} = QV^nQ^{-1}$$

Evaluating powers of a diagonal matrix is very easy for both human and computer.

QR Decomposition Writing a matrix A as a product of two matrices as $A = QR$, where Q is orthogonal and R is upper triangular.

Singular Value Decomposition For a $m \times n$ matrix A the SVD is the product of three matrices as $A = U\Sigma V^*$. Here U is $m \times m$ complex unitary matrix, V is $n \times n$ complex unitary matrix and Σ is $m \times n$ rectangular diagonal matrix.

$$A = \underset{m \times n}{Q} \cdot \underset{m \times m}{\Sigma} \cdot \underset{n \times n}{V^*}$$

The diagonal entries $\sigma_i = (\Sigma)_{ii}$ are called singular values of A . They are uniquely determined by the matrix A . They are spaced on the diagonal in descending order..

2.1.3 Matrix Norms (Operator Norms)

Definition 2.11 (Matrix Norm). If we see the set of all $m \times n$ matrices with real or complex entries as a vector space then the definition of the vector norm can be regarded as **matrix norm**. That is A its **norm** or magnitude is denoted as $\|A\|$ and satisfies the following properties.

1. $\|A\| \geq 0$ for any matrix A .
2. $\|A\| = 0$ if and only if $A = 0$
3. $\|\alpha A\| = |\alpha| \|A\|$ for and $\alpha \in \mathbb{R}$ and $A \in M^{m \times n}$
4. $\|A + B\| \leq \|A\| + \|B\|$

Definition 2.12 (Matrix p Norms). These are matrix norms induced by vector p norms. Defined as:

$$\|A\|_p = \frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p}$$

Here the vector \mathbf{x} lives in \mathbb{K}^n and $A\mathbf{x}$ lives in \mathbb{K}^m where \mathbb{K} is real or complex numbers.

- **p=1** induces maximum absolute column sum as;

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$$

- **p=2** induces the **spectral norm** as maximum **singular value** of the matrix as;

$$\|A\|_2 = \sigma_{\max}(A)$$

- **p= ∞** induces the maximum absolute row sum as;

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$$

Definition 2.13 (Frobenius Norm).

$$\|A\|_F = \left(\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 \right)^{\frac{1}{2}}$$

Definition 2.14 ($L_{p,q}$ Norms).

$$\|A\|_{p,q} = \left(\sum_{i=1}^m \left(\sum_{j=1}^n |a_{ij}|^p \right)^{\frac{q}{p}} \right)^{\frac{1}{q}}$$

In general $L_{p,q}$ norms regards matrices as collection of column vectors and calculates the norm. For example $L_{2,1}$ norm adds all l_2 norms of columns of the given matrix as:

$$\|A\|_{2,1} = \sum_{i=1}^m \left(\sum_{j=1}^n |a_{ij}|^2 \right)^{\frac{1}{2}}$$

2.1.4 Matrix Norms in Matlab and Julia

Matlab uses same command for vector and matrix norms: `norm(A,p)` calculates p norms, `norm(A,"fro")` calculates Frobenius norm and `norm(A,"inf")` calculates p_∞ norm.

```
1  >> A
2  A =
3  -2     0     1     3
4  -3    -2     5    -1
5  -3     4    -2     1
6  1      1     3    -5
7
8  >> norm(A,1)
9  ans = 11
10
11 >> norm(A,"inf")
12 ans = 11
13
14 >> norm(A,2)
15 ans = 7.7704 %as seen below p2 norm returns the max singular value
16
17 >> [U, S, V] = svd(A);
18 >> S
19 S =
20 7.7704         0         0         0
21 0    5.9055         0         0
22 0         0    4.7711         0
23 0         0         0    0.9911
24
25 >> norm(A, "fro")
26 ans = 10.9087
27
```

In Julia LinearAlgebra pack supplies the norm operator.

```
1  using LinearAlgebra
2  julia> A
3  4x4 Matrix{Int64}:
4  -2     0     1     3
5  -3    -2     5    -1
6  -3     4    -2     1
7  1      1     3    -5
8
9  julia> opnorm(A) # returns the spectral norm
10 7.770351825891583
11
12 julia> norm(A,2) # returns the Frobenius norm
13 10.908712114635714
14
15
16
```

2.2 Design Phase

At this section basics of randomized methods and algorithms defined on the reference article are explained.

As mentioned before a low rank approximation of a matrix is basically to draw a subspace of a given linear space that spans approximately the linear space itself. While drawing a subspace different approaches can be utilized and following algorithms are examples of this approaches.

Algorithm 4.1 Randomized Range Finder

Input: An $m \times n$ matrix A , an integer l .

Output: An orthonormal $m \times l$ matrix Q whose range approximates the range of A .

- 1: Draw an $m \times l$ Gaussian random matrix Ω .
- 2: Form the $m \times l$ matrix Y . $Y \leftarrow A\Omega$.

- 3: Construct an $m \times l$ matrix Q whose columns form an orthonormal basis for the range of Y . $[Q, R] \leftarrow \text{qr}(Y)$

At this first algorithm random columns are selected from a given large matrix and QR factorization of the small sample can be calculated easily. Moreover this QR factorization form a basis for more complex calculations.

Algorithm 4.2 Adaptive Randomized Range Finder

Input: An $m \times n$ matrix A , an integer r as oversampling parameter, a floating point number ϵ as approximation tolerance.

Output: An orthonormal $m \times l$ matrix Q whose range approximates the range of A .

- 1: Draw standard Gaussian vectors $w^{(1)}, \dots, w^{(n)}$ of length n .
- 2: **for** $i = 1, 2, \dots, r$ **do**
- 3: $y^{(i)} = Aw^{(i)}$
- 4: **end for**
- 5: $j \leftarrow 0$
- 6: $Q^{(0)} \leftarrow []$ $\triangleright m \times n$ empty matrix
- 7: **while** $\max\{\|y^{(j+1)}\|, \|y^{(j+2)}\|, \dots, \|y^{(j+r)}\|\} > \epsilon/(10\sqrt{2/\pi})$ **do**
- 8: $j \leftarrow j + 1$
- 9: Overwrite $y^{(j)}$ by $(I - Q^{(j-1)}Q^{(j-1)*})y^{(j)}$
- 10: $q^{(j)} = y^{(j)}/\|y^{(j)}\|$
- 11: $Q^{(j)} = [Q^{(j-1)}q^{(j)}]$
- 12: Draw a standard Gaussian Vector $w^{(j+r)}$ of length n .
- 13: $y^{(j+r)} = (I - Q^{(j)}Q^{(j)*})Aw^{(j+r)}$
- 14: **for** $i = (j + 1), (j + 2), \dots, (j + r - 1)$ **do**
- 15: Overwrite $y^{(i)}$ by $y^{(i)} - q^{(j)}\langle q^{(j)}, y^{(i)} \rangle$
- 16: **end for**
- 17: **end while**
- 18: $Q \leftarrow Q^{(j)}$

Here a tolerance value ϵ is given to the algorithm and it returns an orthogonal matrix Q whose range approximates the given matrix A with an max error ϵ . The idea is; at each iteration the algorithm adds r many new vectors from matrix A and maintains the orthonormal matrix Q with every increment. If the desired approximation reached computation halts and returns the current matrix Q .

Algorithm 4.3 Randomized Power Iteration

Input: An $m \times n$ matrix A , integers l and q .

Output: An orthonormal $m \times l$ matrix Q whose range approximates the range of A .

- 1: Draw an $m \times l$ Gaussian random matrix Ω .
- 2: Form the $m \times l$ matrix Y . $Y \leftarrow (AA^*)^q A\Omega$.
- 3: Construct an $m \times l$ matrix Q whose columns form an orthonormal basis for the range of Y . $[Q, R] \leftarrow \text{qr}(Y)$

In this version of randomized range finder the matrix A is multiplied with its transpose and q^{th} power of this product is taken. If the singular values of the input matrix A are in some decay algorithms 4.1 and 4.2 work well. However if the singular values are in a flat structure that is do not in a well decay then power iteration reduces the weight of the small singular vectors and increase the weight of dominant singular vectors so that computation is fastened.

Algorithm 4.5 Fast Randomized Range Finder

Input: An $m \times n$ matrix A , integers l

Output: An orthonormal $m \times l$ matrix Q whose range approximates the range of A .

- 1: Draw an $m \times l$ *SRFT* test matrix Ω .
- 2: Form the $m \times l$ matrix Y . $Y \leftarrow A\Omega$.
- 3: Construct an $m \times l$ matrix Q whose columns form an orthonormal basis for the range of Y . $[Q, R] \leftarrow \text{qr}(Y)$

SRFT is *subsampled random Fourier transform* and can be computed as:

$$\Omega = \sqrt{\frac{n}{l}} DFR$$

where

- **D** is an $n \times n$ diagonal matrix whose entries are independent random variables uniformly distributed on the complex unit circle.
- **F** is the $n \times n$ unitary discrete Fourier Transform (DFT), whose entries take the values $f_{pq} = n^{-1/2} e^{-2\pi i(p-1)(q-1)/n}$ for $p, q = 1, 2, 3, \dots, n$
- **R** is an $n \times l$ matrix that samples l coordinates from n uniformly at random; i.e., its l columns are drawn randomly without replacement from the columns of the $n \times n$ identity matrix.

The idea behind this alteration is; the bottleneck of the presented algorithms are the computation of the product $A\Omega$ that requires $O(mnl)$ flops. However with an structured random matrix this can be reduced to $O(mn \log(l))$

Algorithm 5.1 Singular Value Decomposition

Input: $m \times n$ matrix A , $m \times k$ matrix Q with $\|A - QQ^*A\| < \epsilon$ for a desired ϵ .

Output: Matrices U, S, V where U, V are orthonormal and S non-negative diagonal matrices.

- 1: Form the matrix $B = Q^*A$
- 2: Compute an SVD of the small matrix $B = \bar{U}\Sigma V^*$
- 3: Form the orthonormal matrix $U = Q\bar{U}$

At this last algorithm singular value decomposition for the matrix A which is approximated by the matrix Q is being calculated.

2.3 Implementation Phase

At this stage applications of the given algorithms with Julia-language has been utilized. The complete package can be found the following repository with a Jupyter Notebook containing a demonstration.

https://github.com/SaElmas/low_rank_approximations

Julia-lang packages that has been used in this work.

```

1 using LinearAlgebra # Basic linear algebra functions such as norm matrix multp.
2 using Random        # For random number generation in various distributions
3 using Printf         # For formatted C like console output.
4 using Plots          # Basic plotting utilities
5 using Statistics     # To utilize statistical tools.
6 using Distributions  # For various probability distributions
7 using SparseArrays   # For sparse matrix methods and spare to dense conversions
8 using MatrixMarket   # To read and write Matrix Market format from/to a file.
9
```

```

1
2 # Algorithm 4.1 Randomized Range Finder
3 # Input A(mxn) matrix, integer l
4 # Output Q(mx1) matrix, approximates the range of A
5 function rrf(A, l)
6     rng = MersenneTwister()
7     _, n = size(A);
8     Om = randn(rng, Float64, (n, l))
9     Y = A*Om
10    Q_, _ = qr(Y)
11    Q = Matrix(Q_)
12    return Q
13 end
14
15 # Algorithm 4.2 Adaptive Randomized Range Finder
16 # Input A(mxn) matrix, tolerance eps, integer r as oversampling parameter
17 # Output Q(mx1) orthonormal with tolerance holds with probability 1-min{m,n}10^-r
```

```

18 function arrf(A,eps,r,plot_step)
19     # plot_step is used to generat errors and iterations vectors
20     # if plot_step is 0 then no vectors are empty
21     # for a positive value of plot_step the value is used for iteration step.
22     (m,n) = size(A);
23     W = zeros(n,r)
24     Y = zeros(m,r)
25     Q = zeros(m,1)
26     j = 0
27     max_err=0
28     for i=1:r
29         w = randn(n,1)
30         W[:,i] = w
31         Y[:,i] = A*w
32     end
33
34     for i=1:r
35         ny = norm(Y[:,i])
36         if ny > eps/(10*sqrt(2/pi))
37             max_err = ny
38         end
39     end
40     iteration_step=0;
41     iterations = []
42     errors = []
43
44     while(max_err > eps)
45         iteration_step +=1;
46         if plot_step > 0
47             if iteration_step % plot_step == 0
48                 append!(iterations, iteration_step)
49                 append!(errors,max_err)
50             end
51         end
52         j += 1
53         yj = (1.0I-Q*Q')*Y[:,j]
54         qj = yj / norm(yj);
55         Y[:,j] = yj;
56         if j==1
57             Q[:,j] = qj
58         else
59             Q = cat(Q,qj, dims=2)
60         end
61         wjr = randn(n,1)
62         yjr = (1.0I - Q*Q')*(A*wjr)
63         Y = cat(Y,yjr,dims=2)
64         Y[:,j+r] = yjr
65         for i = j+1:j+r-1
66             yi = Y[:,i]
67             Y[:,i] = yi - (qj'*yi)*qj
68         end
69         max_err = 0
70         for i= j+1:j+r-1
71             ny = norm(Y[:,i])
72             if ny > eps/(10*sqrt(2/pi))
73                 max_err = ny
74             end
75         end
76     end
77     return Q,iterations,errors
78 end
79
80 # Algorithm 4.3 Randomized Power Iteration
81 # Input A(mxn) matrix, integer l, power q
82 # Output Q(mx1) matrix, approximates the range of A
83 function rpi(A, l, q)
84     rng = MersenneTwister()

```

```

85     _,n = size(A)
86     Om = randn(rng, Float64, (n,l))
87     Y = (A*A')^q*A*Om
88     Q_ = qr(Y)
89     Q = Matrix(Q_)
90     return Q
91 end
92
93 # Algorithm 4.4 Randomized Subspace Iteration
94 # Input A(mxn) matrix, integer l, power q
95 # Output Q(mxl) matrix, approximates the range of A
96 function rsi(A, l, q)
97     rng = MersenneTwister()
98     _,n = size(A)
99     Om = randn(rng, Float64, (n,l))
100     Y = A*Om
101     Q = qr(Y)
102     for i = 1:q
103         @show size(A)
104         @show size(Q)
105         Y = A*Q
106         Q = qr(Y)
107     end
108     return Q
109 end
110
111 # Algorithm 4.5 Fast Randomized Range Finder
112 # Input A(mxn) matrix, integer l
113 # Output Q(mxl) matrix, approximates the range of A
114 function frrf(A,l)
115     m,n = size(A)
116     D = rucm(n);
117     F = dftg(n);
118     R = Matrix(1.0I,n,l)[: ,shuffle(1:end)]
119     Om = sqrt(n/l)*D*F*R
120     Y = A*Om
121     q,r = qr(Y)
122     Q = Matrix(q)
123     return Q
124 end
125
126 # Algorithm 5.1 Direct SVD
127 # Matrix A(mxn), Q(mxk) matrices with
128 #   with  $|A-QQ^*A| < \epsilon$ 
129 # Matrices U,S,V, U,V are orthonormal, S nonnegative diagonal matrices
130
131 function direct_svd(A,Q)
132     B = Q'*A
133     Uh,S,V = svd(B)
134     U = Q*Uh
135     return Q,S,V
136 end
137
138 # Uniform Discrete Fourier Transform Generator
139 # Input integer n
140 # Output F(nxn) complex matrix
141 function dftg(n)
142     F = ones(ComplexF64,n,n)
143     for i=1:n
144         for j=1:n
145             F[i,j] = n^(-0.5)*exp(-2*pi*(i-1)*(j-1)/n)
146         end
147     end
148     return F
149 end
150
151 # Random Complex matrix whose entries are uniformly distributed

```

```

152 # on unit circle.
153 # Random n real numbers in [-pi, pi] that is uniformly distributed chosen
154 # those will be considered as angles of complex numbers on unit circle.
155 function rucm(n)
156     rng = MersenneTwister(1234)
157     angles = rand(Uniform(-pi,pi), n,n)
158     D = ones(ComplexF64, n,n)
159     for i = 1:n
160         for j = 1:n
161             D[i,j] = exp(angles[i,j]im)
162         end
163     end
164     return D
165 end
166

```

2.4 Testing Phase

In this section various results of the applications of algorithms that have been studied are presented.

During the testing phase below listed matrices have been used in comparison:

1. A random real sparse matrix with dimension 1000x500.
2. pde2961.mtx from matrix market with following properties.
 - 2961 x 2961 with 14585 entries.
 - Rank 2961
 - Frobenius norm: $2.2 \cdot 10^2$
 - Spectral norm: 10
 - Full specification and the matrix in .mtx format is here.
3. eris1176.mtx from matrix market with following properties
 - 1176x1176 with 9864 entries
 - Rank 774
 - Frobenius norm: 1.1×10^{11}
 - Spectral norm : $\times 10^{10}$
 - Full specification and the matrix in .mtx format is here.
4. lns_511.mtx from matrix market with following properties
 - 511x511 with 2796 entries
 - Rank 391
 - Frobenius norm: 1.1×10^{11}
 - Spectral norm : $\times 10^{10}$
 - Full specification and the matrix in .mtx format is here.

2.4.1 Test of Algorithm 4.1

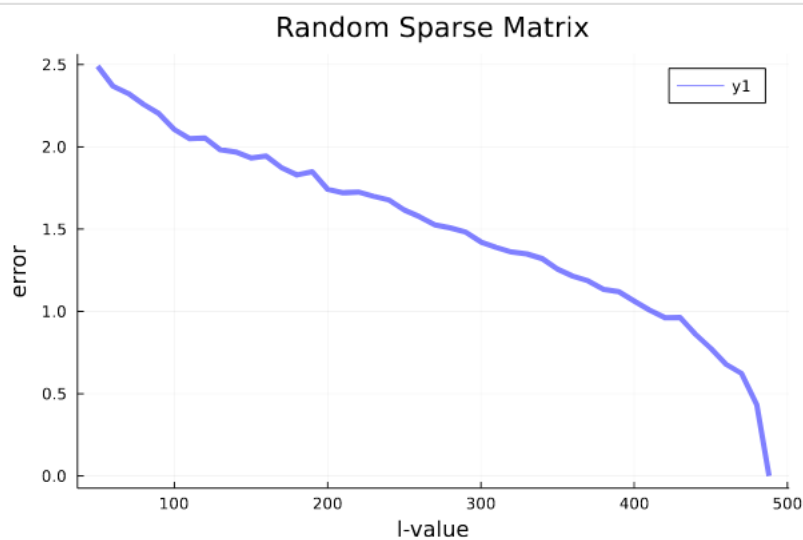
The algorithm described on design section and implemented on implementation section will be tested with the following script for all the above four matrices and relevant plots can be seen below.

The return matrix of the algorithm Q approximates the given matrix A . As the sampling parameter k approaches the rank of the test matrix error goes to zero as expected. The error is in operator norm.

```

1  A = sprand(1000,500,0.004) # generates a random real sparse matrix (1000x500) and
   the dencity = 0.004
2  mmwrite("data/random_sparse.mtx", A)
3  ra = rank(A)
4  xs = collect(50:10:ra)
5  if xs[end] != ra
6      xs = [xs ; ra]
7  end
8  er = zeros(size(xs))
9  index = 1
10 for i in xs
11     Q = rrf(A,i)
12     er[index] = opnorm((1.0I - Q*Q')*A)
13     index += 1
14 end
15 plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
   Random Sparse Matrix")
16

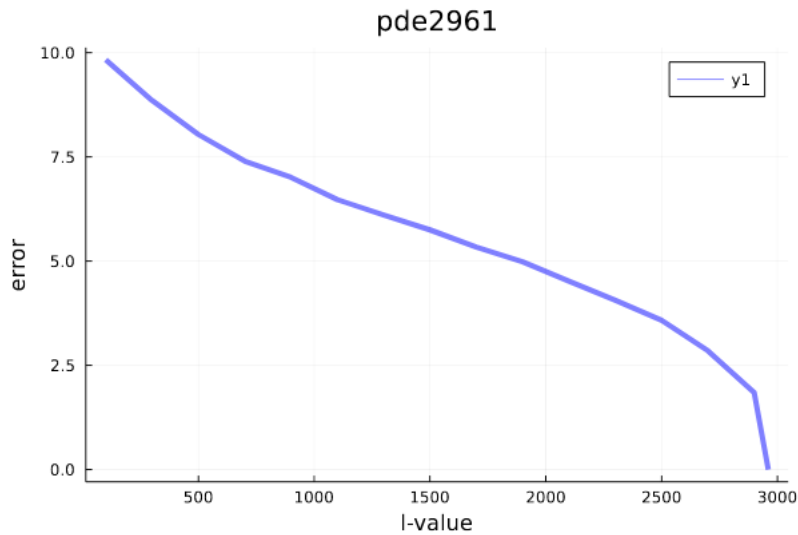
```



```

1  A = mmread("data/pde2961.mtx")
2  ra = 2961
3  xs = collect(100:200:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = rrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1
13 end
14 plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
   pde2961")
15

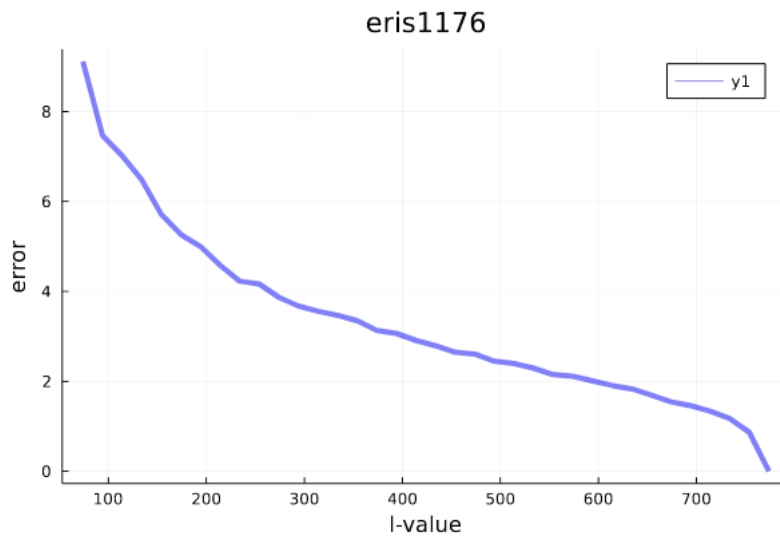
```



```

1  A = mmread("data/eris1176.mtx")
2  ra = 774
3  xs = collect(74:20:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = rrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1
13 end
14 plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
15     eris1176")

```



```

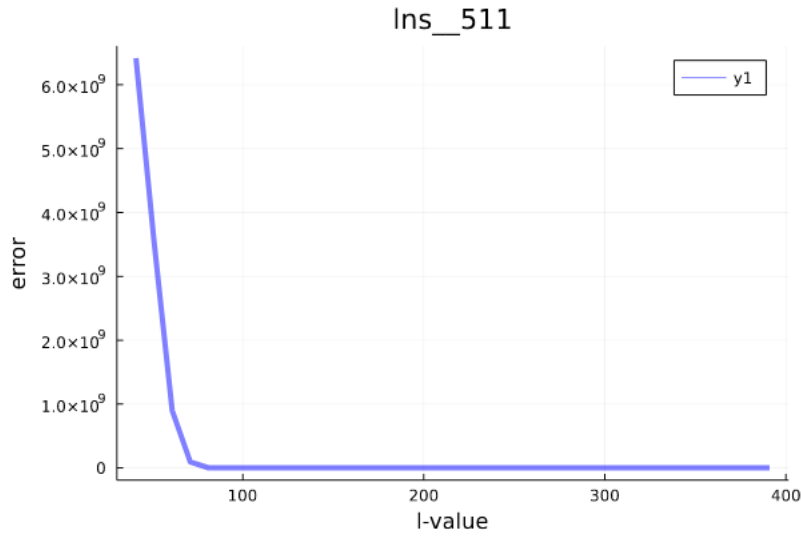
1  A = mmread("data/lms_511.mtx")
2  ra = 391
3  xs = collect(41:10:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = rrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1

```

```

13 end
14 plot(xs, er, xaxis=("l-value"), yaxis="error", line = (0.5, 4, :blue), title="
    lns_511")
15

```



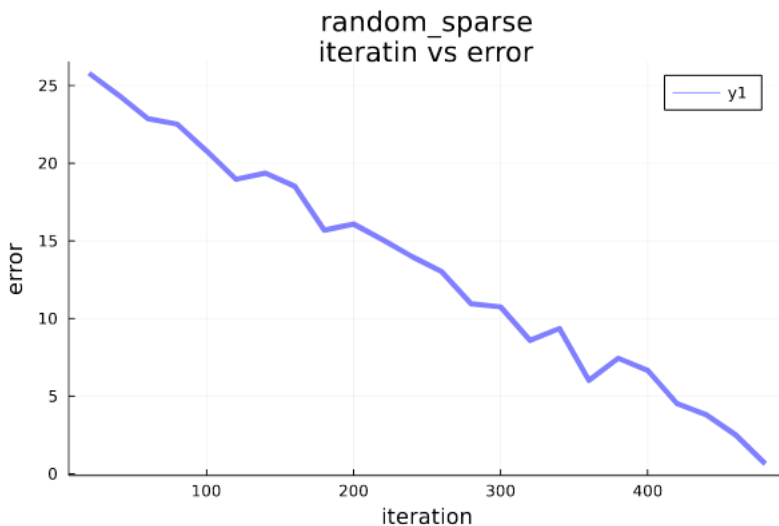
2.4.2 Test of Algorithm 4.2

As detailed in the design and implementation phase algorithm 4.2 returns a matrix Q that approximates the given matrix A up to the error value provided. Below for each test matrix a plot showing that error vs iteration value until algorithm hits the desired error.

```

1 A = mmread("data/random_sparse.mtx")
2 Q,iterations, errors = arrf(A,0.1, 10, 20);
3 mmwrite("data/4.2_random_sparse_output.mtx",sparse(Q))
4 plot(iterations, errors, xaxis=("iteration"), yaxis="error", line = (0.5, 4, :blue)
5     , title="random_sparse\niteratin vs error")

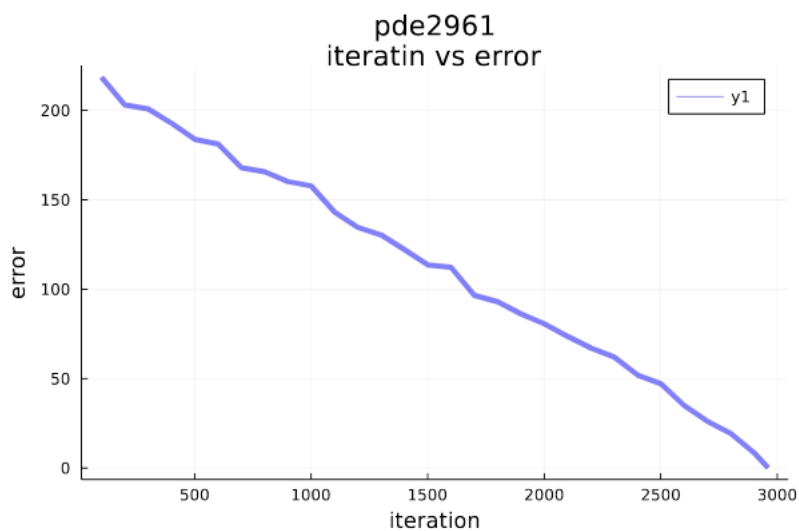
```



```

1 A = mmread("data/pde2961.mtx")
2 Q,iterations,errors= arrf(A,0.1, 10, 100);
3 mmwrite("data/4.2_pde2961_output.mtx",sparse(Q))
4 plot(iterations, errors, xaxis=("iteration"), yaxis="error", line = (0.5, 4, :blue)
5     , title="pde2961\niteratin vs error")
5 errors = readdlm("data/errors_4.2.1.txt", '\t', Float64, '\n')
6 iterations = readdlm("data/iterations_4.2.1.txt", '\t', Int, '\n')
7 plot(iterations, errors, xaxis=("iteration"), yaxis="error", line = (0.5, 4, :blue)
8     , title="pde2961\niteratin vs error")
9

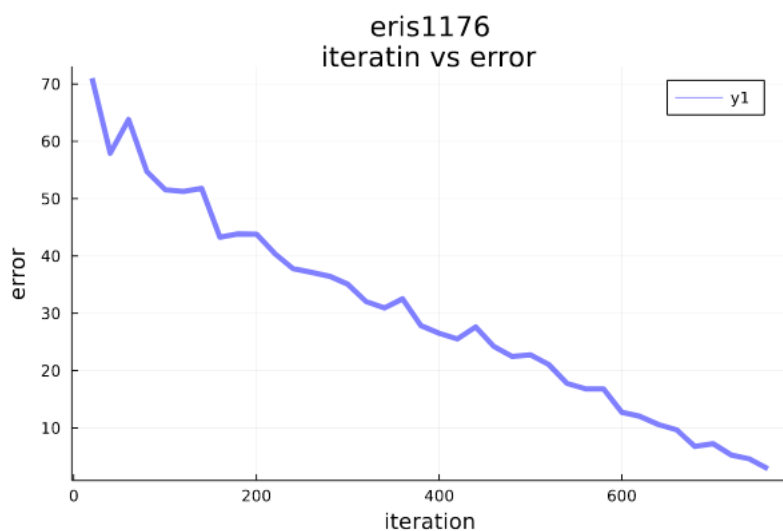
```



```

1  A = mmread("data/eris1176.mtx");
2  Q,iterations, errors = arrf(A,0.1,10,20);
3  mmwrite("data/4.2_eris1176_output.mtx", sparse(Q))
4  plot(iterations, errors, xaxis=("iteration"), yaxis="error", line = (0.5, 4, :blue)
5      , title="eris1176\niteratin vs error")
6
7

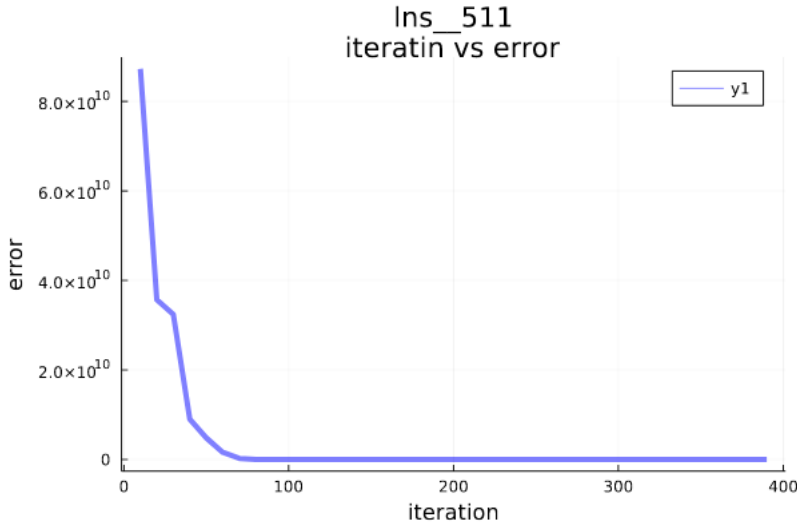
```



```

1  A = mmread("data/lms_511.mtx");
2  Q,iterations, errors = arrf(A,1,10,10);
3  mmwrite("data/4.2_lms_511_output.mtx", sparse(Q))
4  plot(iterations, errors, xaxis=("iteration"), yaxis="error", line = (0.5, 4, :blue)
5      , title="lms_511\niteratin vs error")
6
7

```

2.4.3 Test of Algorithm 4.3

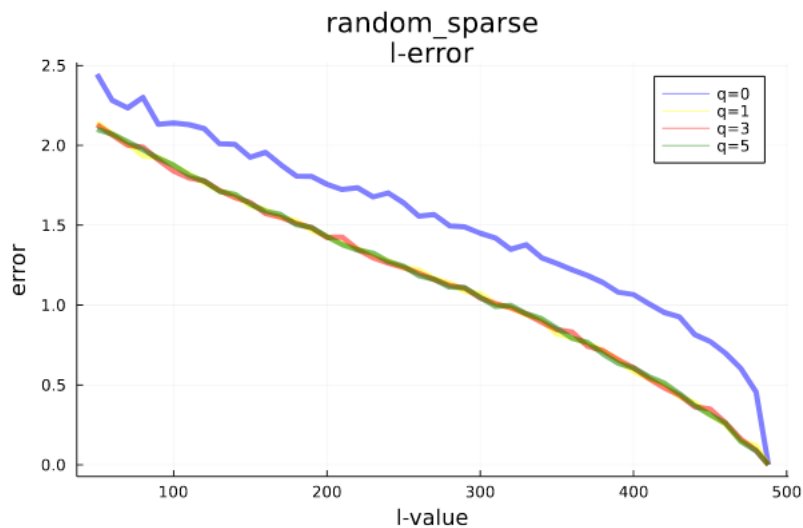
On this algorithm randomized sampling is applied to the matrix $B = (AA^*)^q A$ instead of A . B has same left and right singular vectors (U and V) with A but singular values are scaled up and down ($\sigma_j(B) = \sigma_j(A)^{2q+1}$). Hence we expect a quicker range finder algorithm by this change.

As expected while q is equal zero algorithm works as like algorithm 4.1.

```

1  A = mmread("data/random_sparse.mtx")
2  ra = rank(A)
3  xs = collect(50:10:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er0 = []
8  er1 = []
9  er3 = []
10 er5 = []
11 for i in xs
12     Q0 = rpi(A,i,0)
13     append!(er0,opnorm((1.0I - Q0*Q0')*A))
14 end
15 plot(xs, er0, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    random_sparse\nl-error", label="q=0")
16
17 for i in xs
18     Q1 = rpi(A,i,1)
19     append!(er1,opnorm((1.0I - Q1*Q1')*A))
20 end
21 plot!(xs, er1, xaxis="l-value", yaxis="error", line = (0.5, 4, :yellow), label="q
    =1")
22
23 for i in xs
24     Q3 = rpi(A,i,1)
25     append!(er3,opnorm((1.0I - Q3*Q3')*A))
26 end
27 plot!(xs, er3, xaxis="l-value", yaxis="error", line = (0.5, 4, :red),label="q=3")
28
29 for i in xs
30     Q5 = rpi(A,i,1)
31     append!(er5,opnorm((1.0I - Q5*Q5')*A))
32 end
33 plot!(xs, er5, xaxis="l-value", yaxis="error", line = (0.5, 4, :green), label="q
    =5")
34

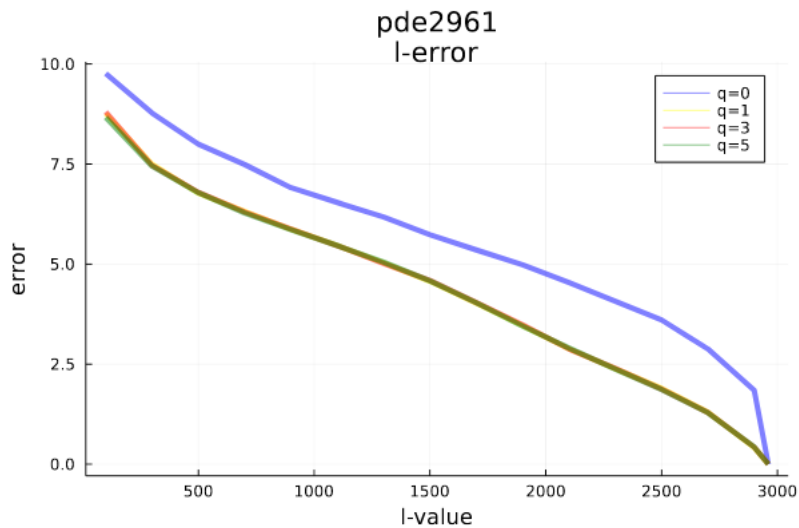
```



```

1  A = mmread("data/pde2961.mtx")
2  ra = 2961
3  xs = collect(100:200:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er0 = []
8  er1 = []
9  er3 = []
10 er5 = []
11 for i in xs
12     Q0 = rpi(A,i,0)
13     append!(er0,opnorm((1.0I - Q0*Q0')*A))
14 end
15 plot(xs, er0, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
16     pde2961\nl-error", label="q=0")
17
18 for i in xs
19     Q1 = rpi(A,i,1)
20     append!(er1,opnorm((1.0I - Q1*Q1')*A))
21 end
22 plot!(xs, er1, xaxis="l-value", yaxis="error", line = (0.5, 4, :yellow), label="q
23     =1")
24
25 for i in xs
26     Q3 = rpi(A,i,1)
27     append!(er3,opnorm((1.0I - Q3*Q3')*A))
28 end
29 plot!(xs, er3, xaxis="l-value", yaxis="error", line = (0.5, 4, :red),label="q=3")
30
31 for i in xs
32     Q5 = rpi(A,i,1)
33     append!(er5,opnorm((1.0I - Q5*Q5')*A))
34 end
35 plot!(xs, er5, xaxis="l-value", yaxis="error", line = (0.5, 4, :green), label="q
36     =5")

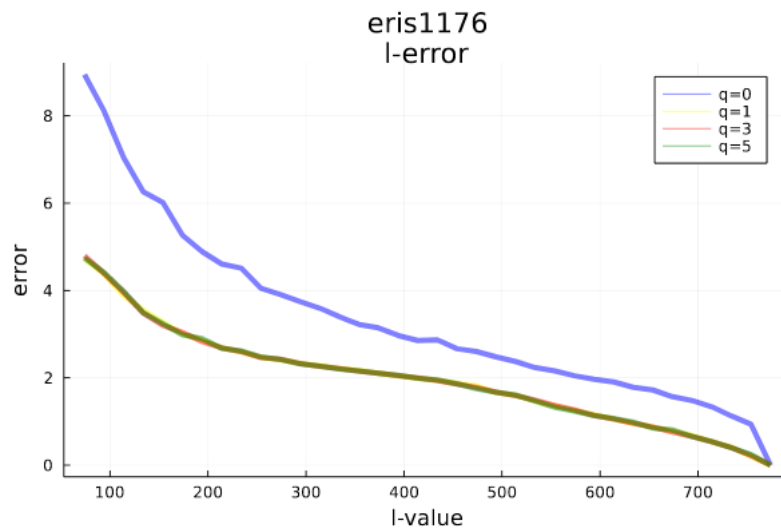
```



```

1  A = mmread("data/eris1176.mtx")
2  ra = 774
3  xs = collect(74:20:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er0 = []
8  er1 = []
9  er3 = []
10 er5 = []
11 for i in xs
12     Q0 = rpi(A,i,0)
13     append!(er0,opnorm((1.0I - Q0*Q0')*A))
14 end
15 plot(xs, er0, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    eris1176\nl-error", label="q=0")
16
17 for i in xs
18     Q1 = rpi(A,i,1)
19     append!(er1,opnorm((1.0I - Q1*Q1')*A))
20 end
21 plot!(xs, er1, xaxis="l-value", yaxis="error", line = (0.5, 4, :yellow), label="q
    =1")
22
23 for i in xs
24     Q3 = rpi(A,i,1)
25     append!(er3,opnorm((1.0I - Q3*Q3')*A))
26 end
27 plot!(xs, er3, xaxis="l-value", yaxis="error", line = (0.5, 4, :red),label="q=3")
28
29 for i in xs
30     Q5 = rpi(A,i,1)
31     append!(er5,opnorm((1.0I - Q5*Q5')*A))
32 end
33 plot!(xs, er5, xaxis="l-value", yaxis="error", line = (0.5, 4, :green), label="q
    =5")
34

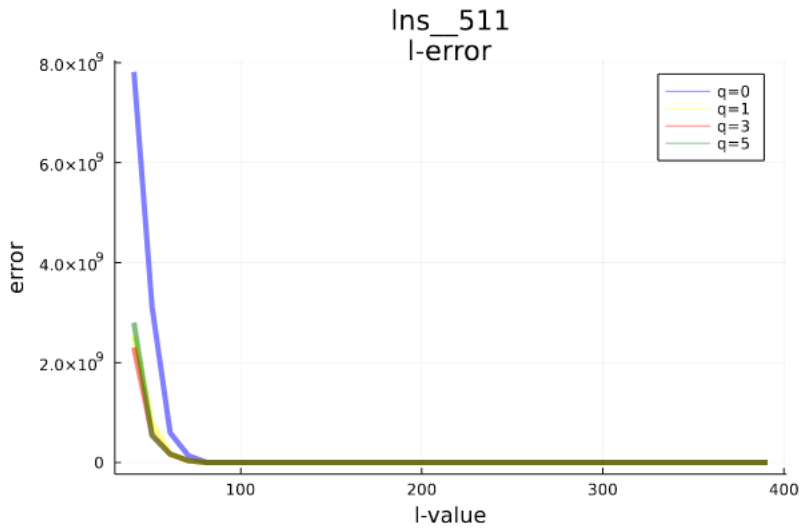
```



```

1  A = mmread("data/lms_511.mtx")
2  ra = 391
3  xs = collect(41:10:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er0 = []
8  er1 = []
9  er3 = []
10 er5 = []
11 for i in xs
12     Q0 = rpi(A,i,0)
13     append!(er0,opnorm((1.0I - Q0*Q0')*A))
14 end
15 plot(xs, er0, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    lms_511\nl-error", label="q=0")
16
17 for i in xs
18     Q1 = rpi(A,i,1)
19     append!(er1,opnorm((1.0I - Q1*Q1')*A))
20 end
21 plot!(xs, er1, xaxis="l-value", yaxis="error", line = (0.5, 4, :yellow), label="q
    =1")
22
23 for i in xs
24     Q3 = rpi(A,i,1)
25     append!(er3,opnorm((1.0I - Q3*Q3')*A))
26 end
27 plot!(xs, er3, xaxis="l-value", yaxis="error", line = (0.5, 4, :red),label="q=3")
28
29 for i in xs
30     Q5 = rpi(A,i,1)
31     append!(er5,opnorm((1.0I - Q5*Q5')*A))
32 end
33 plot!(xs, er5, xaxis="l-value", yaxis="error", line = (0.5, 4, :green), label="q
    =5")
34
35

```



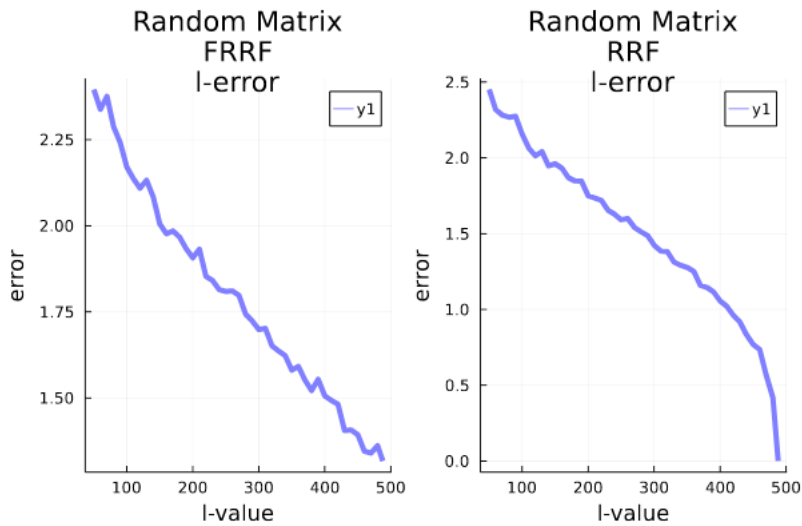
2.4.4 Algorithm 4.5

In this section randomized range finder algorithm 4.1 is presented in comparison with fast randomized range finding algorithm for all the four test matrices.

```

1  A = mmread("data/random_sparse.mtx")
2  ra = rank(A)
3  xs = collect(50:10:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = frrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1
13 end
14
15 p1 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    Random Matrix\nFRRF\nl-error")
16
17 er = zeros(size(xs))
18 index = 1
19 for i in xs
20     Q = rrf(A,i)
21     er[index] = opnorm((1.0I - Q*Q')*A)
22     index += 1
23 end
24 p2 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    Random Matrix\nRRF\nl-error")
25
26 plot!(p1,p2)
27

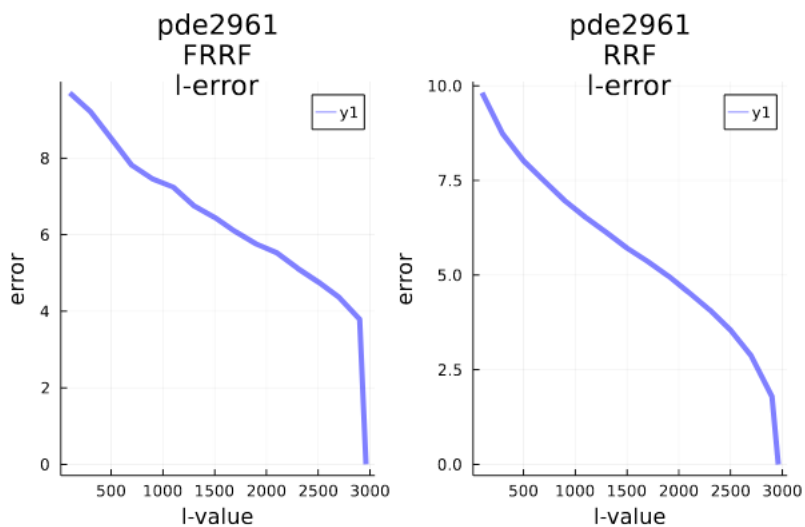
```



```

1  A = mmread("data/pde2961.mtx")
2  # @show rank(A) yields rank of A is 2961
3  ra = 2961
4  xs = collect(100:200:ra)
5  if xs[end] != ra
6      xs = [xs ; ra]
7  end
8  er = zeros(size(xs))
9  index = 1
10 for i in xs
11     Q = frrf(A,i)
12     er[index] = opnorm((1.0I - Q*Q')*A)
13     index += 1
14 end
15 p1 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    pde2961\nFRRF\nl-error")
16
17 er = zeros(size(xs))
18 index = 1
19 for i in xs
20     Q = rrf(A,i)
21     er[index] = opnorm((1.0I - Q*Q')*A)
22     index += 1
23 end
24 p2 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    pde2961\nRRF\nl-error")
25
26 plot!(p1,p2)
27

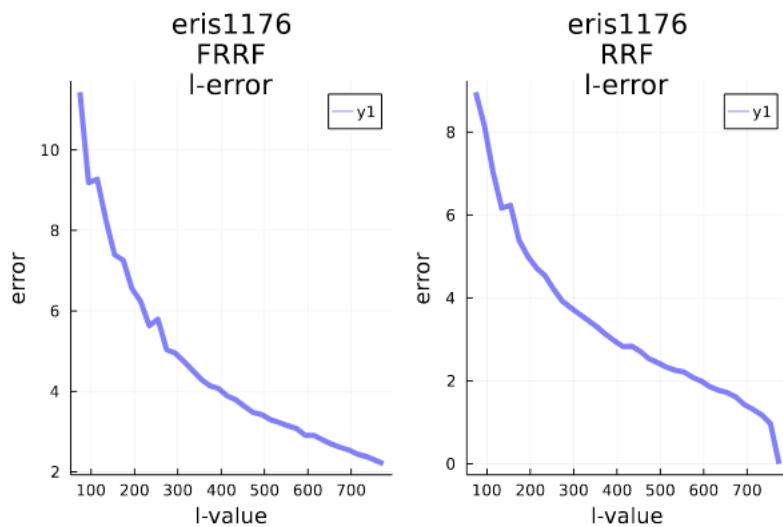
```



```

1  A = mmread("data/eris1176.mtx")
2  ra = 774
3  xs = collect(74:20:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = frrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1
13 end
14 p1 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    eris1176\nFRRF\nl-error")
15
16 er = zeros(size(xs))
17 index = 1
18 for i in xs
19     Q = rrf(A,i)
20     er[index] = opnorm((1.0I - Q*Q')*A)
21     index += 1
22 end
23 p2 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    eris1176\nRRF\nl-error")
24
25 plot!(p1,p2)
26

```



```

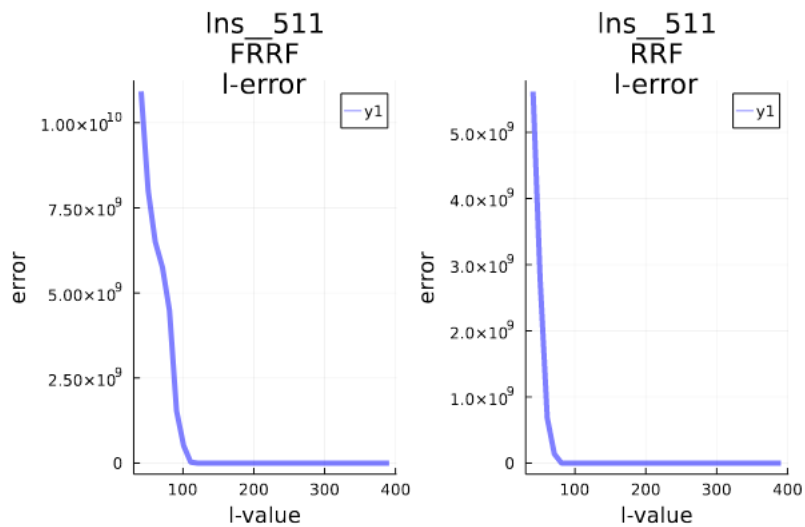
1  A = mmread("data/lms_511.mtx")
2  ra = 391
3  xs = collect(41:10:ra)
4  if xs[end] != ra
5      xs = [xs ; ra]
6  end
7  er = zeros(size(xs))
8  index = 1
9  for i in xs
10     Q = frrf(A,i)
11     er[index] = opnorm((1.0I - Q*Q')*A)
12     index += 1
13 end
14 p1 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    lms_511\nFRRF\nl-error")
15
16 er = zeros(size(xs))
17 index = 1
18 for i in xs

```

```

19     Q = rrf(A,i)
20     er[index] = opnorm((1.0I - Q*Q')*A)
21     index += 1
22 end
23 p2 = plot(xs, er, xaxis="l-value", yaxis="error", line = (0.5, 4, :blue), title="
    lns_511\nRRF\nl-error")
24
25 plot!(p1,p2)
26

```



3 Conclusion

In this practice period...