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

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 numberical 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 $v + e = 0$
- $1v = v1 = v$
- $(ab)v = a(bv)$
- $(a + b)v_1 = av_1 + av_2$
- $a(v_1 + v_2) = av_1 + av_2$

$\mathbb{R}^n$ 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 mxn dimensional matrices overt the field of complex numbers $\mathbb{C}$

Definition 2.2 (Inner Product)**.** Let V be a vector space over a field ($\mathbb{R}$ 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:

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

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 $\|\boldsymbol{x}\|$. The function should satisfy following three properties.

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

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 $\boldsymbol{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$

- $\boldsymbol{l_1}$ **norm** (*Taxicap norm, Manhattan norm*): $\|\boldsymbol{x}\|_1 = \sum_{j=1}^{n} |x_j|$
- $\boldsymbol{l_2}$ **norm**: $\|\boldsymbol{x}\|_2 = \sqrt{\sum_{j=1}^{n} |x_j|^2}$. 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 $\|\boldsymbol{x}\|_2 = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle}$
- $\mathbf{l_p}$ **norm** (p norm): $\|\boldsymbol{x}\|_p = \left(\sum_{j=1}^{n} |x_j|^2\right)^{1/p}$
- $\boldsymbol{l_\infty}$ **norm** *(Infinity norm, maximum norm)* $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$

Definition 2.5. For an inner product space V if the product of two vectors $\boldsymbol{x}$ and *bmy*, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0$ than we say that $\boldsymbol{x}$ and $\boldsymbol{y}$ 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 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 $AA^H = A^HA = I$ where $A + H$ denotes the **conjugate transpose** of A.

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

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 eigenvectors of A on the diagonal entries. One of the important properties of this decomposition is powers of the matrix A can be calculated as:

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

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.

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

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 of 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:

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

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

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

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

- **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;

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

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 ($\boldsymbol{L_{p,q}}$ 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.

```
>> A
A =
-2     0     1     3
-3    -2     5    -1
-3     4    -2     1
1     1     3    -5

>> norm(A,1)
ans = 11

>> norm(A,"inf")
ans = 11

>> norm(A,2)
ans = 7.7704 %as seen below p2 norm returns the max singular value

>> [U, S, V] = svd(A);
>> S
S =
7.7704         0         0         0
0    5.9055         0         0
0         0    4.7711         0
0         0         0    0.9911

>> norm(A, "fro")
ans = 10.9087
```

In Julia `LinearAlgebra` pack supplies the norm operator.

```
using LinearAlgebra
julia> A
4x4 Matrix{Int64}:
-2   0   1   3
-3  -2   5  -1
-3   4  -2   1
1   1   3  -5

julia> opnorm(A) # returns the spectral norm
7.770351825891583

julia> norm(A,2) # returns the Frobenius norm
10.908712114635714
```

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. $[Q, R] \leftarrow$ `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 $w^{(1)}, ..., w^{(n)}$ of length n.
2: **for** $i = 1, 2, ..., r$ **do**
3: $\quad y^{(i)} = Aw^{(i)}$
4: **end for**
5: $j \leftarrow 0$
6: $Q^{(0)} \leftarrow []$ ▷ $m \times n$ empty matrix
7: **while** $max\{\left\|y^{(j+1)}\right\|, \left\|y^{(j+2)}\right\|, ..., \left\|y^{(j+r)}\right\|\} > \epsilon/(10\sqrt{2/\pi})$ **do**
8: $\quad j \leftarrow j + 1$
9: $\quad$ Overwrite $y^{(j)}$ by $(I - Q^{(j-1)}Q^{(j-1)*})y^{(j)}$
10: $\quad q^{(j)} = y^{(j)}/\|y^{(j)}\|$
11: $\quad Q^{(j)} = [Q^{(j-1)}q^{(j)}]$
12: $\quad$ Draw a standard Gaussian Vector $w^{(j+r)}$ of length n.
13: $\quad y^{(j+r)} = (I - Q^{(j)}Q^{(j)*})A^{(j+r)}$
14: $\quad$ **for** $i = (j + 1), (j + 2), ..., (j + r - 1)$ **do**
15: $\quad\quad$ Overwrite $y^{(i)}$ by $y^{(i)} - q^{(j)}\langle q^{(j)}, y^{(j)}\rangle$
16: $\quad$ **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. $[Q, R] \leftarrow$ `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. $[Q, R] \leftarrow$ `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 $f_{pq} = n^{-1/2} e^{-2\pi(p-1)(q-1)/n}$ for $p, q = 1, 2, 3, ..., n$
- **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 $O(mnlog(l))$

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 ot the small matrix $B = \bar{U}\Sigma V^*$
3: Form the orhtonormal 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

```
using LinearAlgebra
using Random
using Printf
using Plots
using Statistics
using Distributions
using SparseArrays
using MatrixMarket

function generate_sampling(n,s,p)
# n : number of columns
# s : size of sampling
# p : probabilities for each column

# generate an idendity matrix nxn
# select random s columns from this
eye = Matrix(1.0I, n,n);
total = 0;
if p isa Matrix{Float64}
  for i = 1:n
  eye[i,i] = 1 / sqrt(p[i]*s);
  end
else
  for i = 1:n
    total += norm(A[:,i])*norm(B[i,:]);
  end
  for i= 1:n
    pni = norm(A[:,i])*norm(B[i,:]) / total;
```

```
        eye[i,i] = 1 / sqrt(pni*s)
      end
    end

    rng = MersenneTwister();
    perm = randperm(rng,s);
    S = eye[:,perm];
    return S;
    end

    function basicMatrixMulitplication(A, B, c, P)
    # A mxn
    # B nxp
    # c sampling number c <= n
    # P probabilities contains n float. 0 < pi < 1 and sum(pi) = 1;
    _,n = size(A);
    S = generate_sampling(n,c,P);
    A * S , S' * B;
    end

    # Algorithm 4.1 Randomized Range Finder
    # Input A(mxn) matrix, integer l
    # Output Q(mxl) matrix, approximates the range of A
    function rrf(A, l)
    rng = MersenneTwister()
    _,n = size(A);
    Om = randn(rng, Float64, (n,l))
    Y = A*Om
    Q_,_ = qr(Y)
    Q = Matrix(Q_)
    return Q
    end

    # Algorithm 4.2 Adaptive Randomized Range Finder
    # Input A(mxn) matrix, tolerance eps, integer r as oversampling parameter
    # Output Q(mxl) orthonormal with tolerance holds with probability 1-min{m,n}10^-r
    function arrf(A,eps,r,plot_step)
    # plot_step is used to generat errors and iterations vectors
    # if plot_step is 0 then no vectors are empty
    # for a positive value of plot_step the value is used for iteration step.
    (m,n) = size(A);
    W = zeros(n,r)
    Y = zeros(m,r)
    Q = zeros(m,1)
    j = 0
    max_err=0
    for i=1:r
    w = randn(n,1)
    W[:,i] = w
    Y[:,i] = A*w
    end

    for i=1:r
    ny = norm(Y[:,i])
    if ny > eps/(10*sqrt(2/pi))
    max_err = ny
    end
    end
    iteration_step=0;
    iterations = []
    errors = []

    while(max_err > eps)
    iteration_step +=1;
    if plot_step > 0
    if iteration_step % plot_step == 0
    append!(iterations, iteration_step)
```

```
    append!(errors,max_err)
    end
    end
    j += 1
    yj = (1.0I-Q*Q')*Y[:,j]
    qj = yj / norm(yj);
    Y[:,j] = yj;
    if j==1
    Q[:,j] = qj
    else
    Q = cat(Q,qj, dims=2)
    end
    wjr = randn(n,1)
    yjr = (1.0I - Q*Q')*(A*wjr)
    Y = cat(Y,yjr,dims=2)
    Y[:,j+r] = yjr
    for i = j+1:j+r-1
    yi = Y[:,i]
    Y[:,i] = yi - (qj'*yi)*qj
    end
    max_err = 0
    for i= j+1:j+r-1
    ny = norm(Y[:,i])
    if ny > eps/(10*sqrt(2/pi))
    max_err = ny
    end
    end
    end
    return Q,iterations,errors
    end

    # Algorithm 4.3 Randomized Power Iteration
    # Input A(mxn) matrix, integer l, power q
    # Output Q(mxl) matrix, approximates the range of A
    function rpi(A, l, q)
    rng = MersenneTwister()
    _,n = size(A)
    Om = randn(rng, Float64, (n,l))
    Y = (A*A')^q*A*Om
    Q_,_ = qr(Y)
    Q = Matrix(Q_)
    return Q
    end

    # Algorithm 4.4 Randomized Subspace Iteration
    # Input A(mxn) matrix, integer l, power q
    # Output Q(mxl) matrix, approximates the range of A
    function rsi(A, l, q)
    rng = MersenneTwister()
    _,n = size(A)
    Om = randn(rng, Float64, (n,l))
    Y = A*Om
    Q = qr(Y)
    for i = 1:q
    @show size(A)
    @show size(Q)
    Y = A*Q
    Q= qr(Y)
    end
    return Q
    end

    # Algorithm 4.5 Fast Randomized Range Finder
    # Input A(mxn) matrix, integer l
    # Output Q(mxl) matrix, approximates the range of A
    function frrf(A,l)
    m,n = size(A)
```

```
    D = rucm(n);
    F = dftg(n);
    R = Matrix(1.0I,n,l)[:,shuffle(1:end)]
    Om = sqrt(n/l)*D*F*R
    Y = A*Om
    q,r = qr(Y)
    Q = Matrix(q)
    return Q
    end

    # Algorithm 5.1 Direct SVD
    # Matrix A(mxn), Q(mxk) matrices with
    # with |A-QQ*A| < epsilon
    # Matrices U,S,V, U,V are orthonormal, S nonnegative diagonal matrices

    function direct_svd(A,Q)
    B = Q'*A
    Uh,S,V = svd(B)
    U = Q*Uh
    return Q,S,V
    end

    # Uniform Discrete Fourier Transform Generator
    # Input integer n
    # Output F(nxn) complex matrix
    function dftg(n)
    F = ones(ComplexF64,n,n)
    for i=1:n
    for j=1:n
    F[i,j] = n^(-0.5)*exp(-2*pi*(i-1)*(j-1)/n)
    end
    end
    return F
    end

    # Random Complex matrix whose entries are uniformly distributed
    # on unit circle.
    # Random n real numbers in [-pi, pi] that is uniformly distributed chosen
    # those will be considered as angles of complex numbers on unit circle.
    function rucm(n)
    rng = MersenneTwister(1234)
    angles = rand(Uniform(-pi,pi), n,n)
    D = ones(ComplexF64, n,n)
    for i = 1:n
    for j = 1:n
    D[i,j] = exp(angles[i,j]im)
    end
    end
    return D
    end



    # Gram-Schmidt orthornormalization
    function gso(A)
    m,n = size(A)
    Q = zeros(m,n)
    v1 = A[:,1]
    Q[:,1] = v1 / norm(v1);
    for k = 2:n
    w = A[:,k]
    v = w
    for j = 1:k-1
    vj = Q[:,j]
    v -= (w'*vj)*vj
    end
    Q[:,k] = v / norm(v)
```

```
    end
    return Q;
    end
```