Author:	Jamiu Babatunde Mojolagbe
Department:	Electrical and Computer Engineering
Student ID:	#7804719
Email:	mojolagm@myumanitoba.ca
Course:	ECE 7650
Homework:	1
Sub. Date:	October 26, 2016

QUESTION ONE (1)

Computational complexity of an algorithm can be sub-divided into

i. Time Complexity

ii. Space/Memory Complexity

Time Complexity

The time complexity or running time of an algorithm depends on some factors which includes but not limited to the followings:

i. Number of Processors (Single or Multiple)

ii. Memory/Disk Read/Write Speed

iii. System Architecture (32-bit or 64-bit)

iv. Input to an algorithm

However, for the purpose of this homework, we are only interested in the rate of growth of time with respect to the input (that is, last item above (iv)).

Assumptions About Model Machine Used

i. It has just single processor

ii. It is made of 32-bit architecture

iii. It has sequential execution of commands

iv. It takes one (1) unit time for basic arithmetic and logical operations like addition, multiplication, division, subtraction, greater than, less than and so on.

v. It takes 1unit of time for assignment

vi. It takes 1unit of time for increment/decrement

vii. It also takes 1 unit of time for returning from a function

Programming Language Used

The algorithms presented are implemented in MATLAB. Therefore, the complexity calculation of each algorithm is based on the MATLAB implementation of the algorithms.

Most Commonly Used Operations and Calculated Complexities

In this report of complexity analysis of both Modified Gram Schmidt and Householder Reflection algorithm sub-routines or functions used within the algorithm is presented with their complexities so as to present them before they are used in the complexity calculation of each of the algorithms mentioned above. The sub-routines include: inner product (dot product), matrix vector product and norm of vector.

<u>Inner Product (Dot Product) Complexity</u>

The inner product of two vectors says $\underline{v}$ and $\underline{u}$ is implemented as predefined function in MATLAB as **dot($\underline{v}$, $\underline{u}$)**and its complexity as used throughout this report is calculate as:

Consider $\underline{v} = [v_1\ v_2\ v_3\ \ldots\ v_n]$

$\underline{u} = [u_1\ u_2\ u_3\ \ldots\ u_n]$

therefore, $\text{dot}(\underline{v}, \underline{u}) = v_1{*}u_1 + v_2{*}u_2\ v_3{*}u_3\ \ldots\ v_n{*}u_n$

Finally,

Number of Additions	$= n - 1$
Number of Multiplications	$= n$
Total complexity	$= n - 1 + n = \mathbf{2n - 1}$

<u>Matrix – Vector Product Complexity</u>

Consider a matrix A^{mxn} and a vector x^{nx1}, product Ax is given by:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \ldots & a_{1n} \\ a_{11} & a_{12} & a_{13} \ldots & a_{1n} \\ \vdots & . & . & . \\ a_{m1} & a_{m2} & a_{m3} \ldots & a_{mn} \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

From above it can be seen that inner product is required in "m" times (that is, the inner product is performed for each row of the matrix and the vector $\underline{x}$).

Therefore, complexity of Ax $= (2n - 1) * m$

$= \mathbf{2n^2 - n}$ (for a square matrix)

<u>Norm of a Vector Complexity</u>

Norm of a vector is implemented as predefined function as norm().

Consider $\underline{v} = [v_1\ v_2\ v_3\ \ldots\ v_n]$ for all $V \epsilon R^n$

$\text{norm}(\underline{v}) = \sqrt{(v_1 * v_1 + v_2 * v_2\ v_3 * v_3\ \ldots\ v_n * v_n)}$

Number of additions (+)	= n – 1
Number of square root (√)	= 1
Number of multiplication (*)	= n
Total complexity	= n – 1 + n + 1 = **2n**

<u>Modified Gram Schmidt Algorithm Complexity Analysis</u>

Modified Gram Schmidt algorithm is implemented with name modifiedGS as shown below (and its source code is contained in the attached code with the function name as filename):

Modified Gram-Schmidt Algorithm Matlab Code	**Cost**	**No of times**
`function [q, r] = modifiedGS(A)`		
`    m = length(A);`	2	1
`    r(1, 1) = norm(A(:,1));`	**2n**+1	1
`    assert(r(1, 1) ~= 0);`	1	1
`    q(:,1) = (A(:,1))./ r(1, 1);`	2	1
`    for j = 2:1:m`	2	m-1
`       q(:,j) = A(:,j);`	1	m-1
`       for i = 1:1:j-1`	2	(m-1)(m-2)
`           r(i, j) = dot(q(:,j), q(:,i));`	(**2n-1**)+1	(m-1)(m-2)
`           q(:,j) = q(:,j) - r(i, j)*q(:,i);`	3	(m-1)(m-2)
`       end`		
`       r(j, j) = norm(q(:,j));`	1+**2n**	m-1
`       assert(r(j, j) ~= 0);`	1	m-1
`       q(:,j) = q(:,j) / r(j, j);`	2	m-1
`    end`		
`end`	1	1
Note: The boldened parameters in "cost" column rep. pre-calculated values		

Time complexity, T(n) $= 2nm^2 + 5m^2 - 4nm + 4n + m + 10$

$= 2n^3 + n^2 + 5n + 10$ (for square matrix, n=m)

Big "O" Complexity

Big "O" complexity is the upper bound of rate of growth of a function. It is given by:

O(g(n)) = { f(n): there exists constants c and n_o, $f(n) \le cg(n)$, for $n \ge n_o$}

Therefore, from calculate complexity above:

$f(n) = 2n^3 + n^2 + 5n + 10$

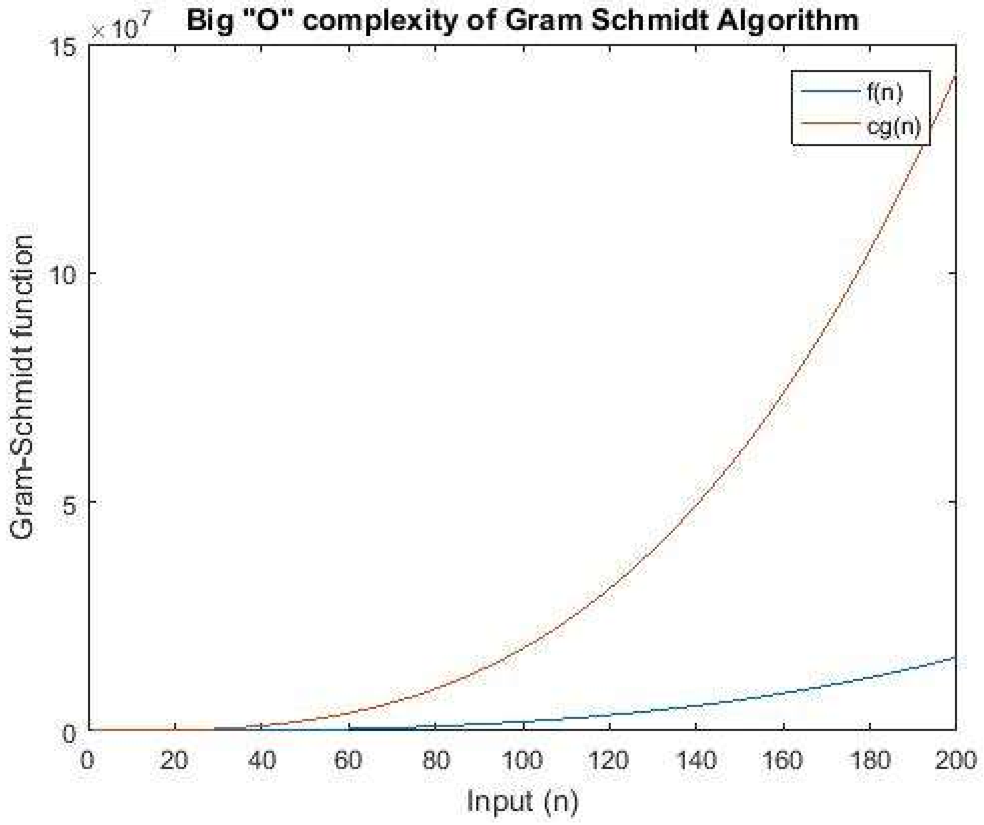
$g(n) = n^3$

$c = 2 + 1 + 5 + 10 = 18$, $n_o = 1$, $f(n) \le 18n^3$ for all $n \ge n_o$

For non – square matrix, $f(n) = 3nm^2 + 4m^2 - 6nm + 2n - 2m + 8$, and as **n** and **m** tends to infinity, then the other terms disappear leaving only $\mathbf{2nm^2}$**.**

Therefore, Gram Schmidt has big 'oh' complexity of $\mathbf{O(n^3)}$ for square matrix or $\mathbf{O(nm^2)}$ for non-square matrix.

The complexity demonstration and graph displayed below was obtained from file named `bigOgramSchmidt.m`.



Observation:

It can be observed from the graph above that cg(n) is always greater than f(n); and that f(n) never grows at rate faster than cg(n) after n_o. Therefore, householder reflection algorithm has big "oh" complexity of $\mathbf{O(n^3)}$**.**

Householder Reflection Algorithm Complexity Analysis

Householder reflection algorithm is implemented with name **houseHolder** as shown below (and its source code is contained in the attached code with the function name as filename):

```
1. function [q, r] = houseHolder(r)
2.    [n, m] = size(r);
3.    I = eye(n, n);
4.    for j = 1:m
5.        for ii = 1 : j - 1
6.           r(:, j)=r(:,j)-(1+omega(ii))* w(:, ii)*dot(w(:, ii),r(:, j));
7.        end
8.        for k = 1:n
9.            if k < j; z(k, 1) = 0;
10.           elseif k == j; z(k,1)= r(k,j)+ exp(i.*angle(r(k, j))).* norm(r(k:end,j));
11.           elseif k > j; z(k, 1) = r(k, j);
12.           end
13.       end
14.
15.       w(:, j) = z/norm(z);
16.       omega(j) = dot(r(:, j), w(:, j)) / dot(w(:, j), r(:, j));
17.       r(:, j) = r(:, j) - (1 + omega(j)) * w(:, j) * dot(w(:, j), r(:, j));
18.       q(:, j) = I(:, j)-(1 + omega(j)) * w(:, j)* dot(w(:, j), I(:, j));
19.
20.       for ii = j - 1: -1: 1
21.          q(:,j)=q(:, j)-(1+omega(ii)) * w(:, ii)* dot(w(:, ii), q(:, j));
22.       end
23.   end
24. end
```

Because of the length and width of the code presented above, the code numbers shown to the left of the code are used to evaluate the cost of executing each line of code as analyzed in the table below:

Code Line	Cost	No of times
2	1	1
3	1	1
4	2	m
5	2	m(m-1)
6	4+**2n** +1	m(m-1)
8	2	mn
9,10, 11	5+**2n** (worst case chosen)	mn
15	2 + **2n**	m
16	2 +2(**2n-1**)	m
17	5 + (**2n-1**)	m
18	5 + (**2n-1**)	m
20	2	m(m-1)
21	5 + (2n-1)	m(m-1)
24	1	1

Time complexity, $T(n) = 2nm^2 + 2n^2m + 7m^2 + 4nm + 12m + 3$

$= 4n^3 + 11n^2 + 12n + 3$ (for square matrix, n=m)

$= \mathbf{\frac{4}{3}n^3 + \frac{11}{3}n^2 + 4n + 1}$

<u>Big "O" Complexity</u>

Therefore, from calculate complexity above:

$f(n) = 4n^3 + 11n^2 + 12n + 3$

$g(n) = n^3$

$c = 4 + 11 + 12 + 3 = 30$, $n_o = 1$, $f(n) \le 30n^3$ for all $n \ge n_o$

For non – square matrix,

$f(n) = 2nm^2 + 2n^2m + 7m^2 + 4nm + 12m + 3$,

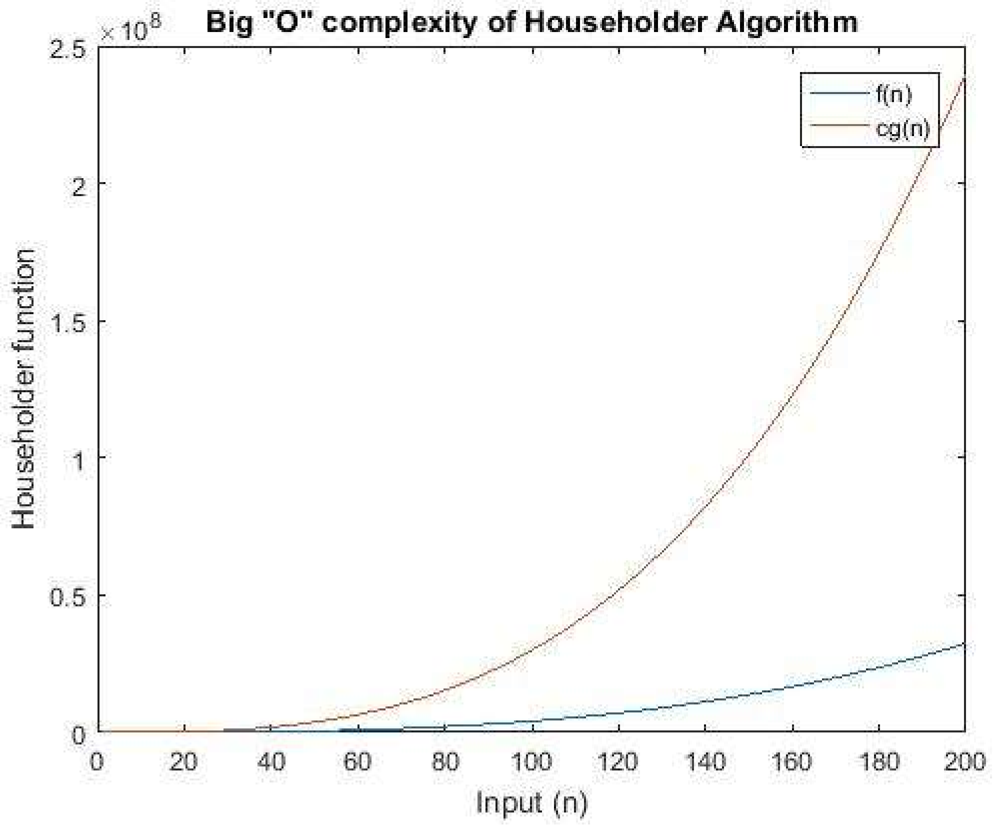
for square matrix

$f(n) = \mathbf{\frac{4}{3}n^3 + \frac{11}{3}n^2 + 4n + 1}$

and as **n** and **m** tends to infinity, then the other terms disappear leaving only $\mathbf{2nm^2 + 2n^2m}$ (for non-square) and $\mathbf{\frac{4}{3}n^3}$ (for square matrix) **.**

Therefore, Gram Schmidt has big 'oh' complexity of $\mathbf{O(n^3)}$ for square matrix or $\mathbf{O(nm^2 + n^2m)}$ for non-square matrix.

The complexity demonstration and graph displayed below was obtained from file named `bigOhouseHolder.m`.



Observation:

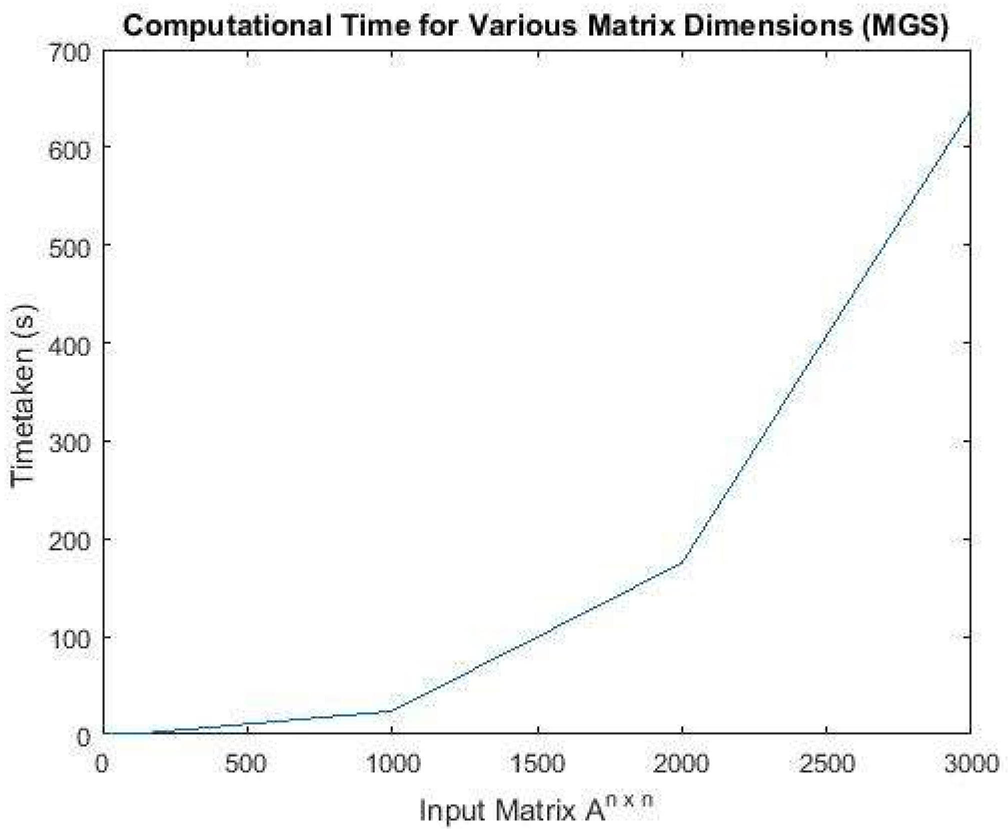
It can be observed from the graph above that cg(n) is always greater than f(n); and that f(n) never grows at rate faster than cg(n) after n_o. Therefore, householder reflection algorithm has big "oh" complexity of **O(n^3).**

QUESTION TWO (2)

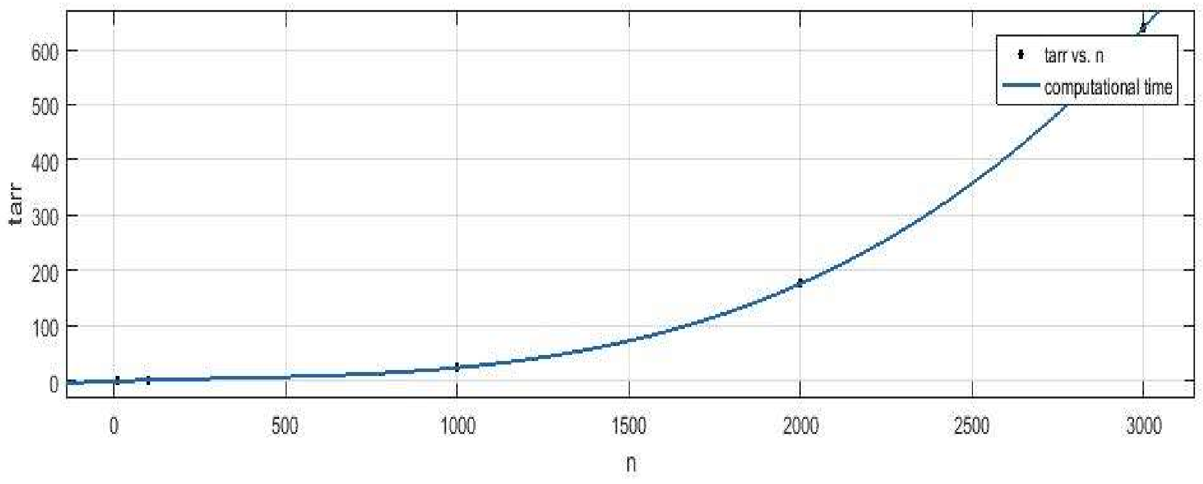
Modified Gram Schmidt algorithm for orthonormalizing the columns of a matrix was implemented in the file named `modifiedGS.m`. The driver program is named `testModifiedGS.m`. The following results were obtained:

Matrix Dimension (nxn)	Time Taken (seconds)
10x10	0.0124
100x100	0.0935
1000x1000	23.7213
2000x2000	175.4073
3000x3000	639.4735

Below is the graph, **with unfitted curve**, of computational time against input matrix of various dimensions (contained in the table above).



The fitted graph is shown below:



Observation:

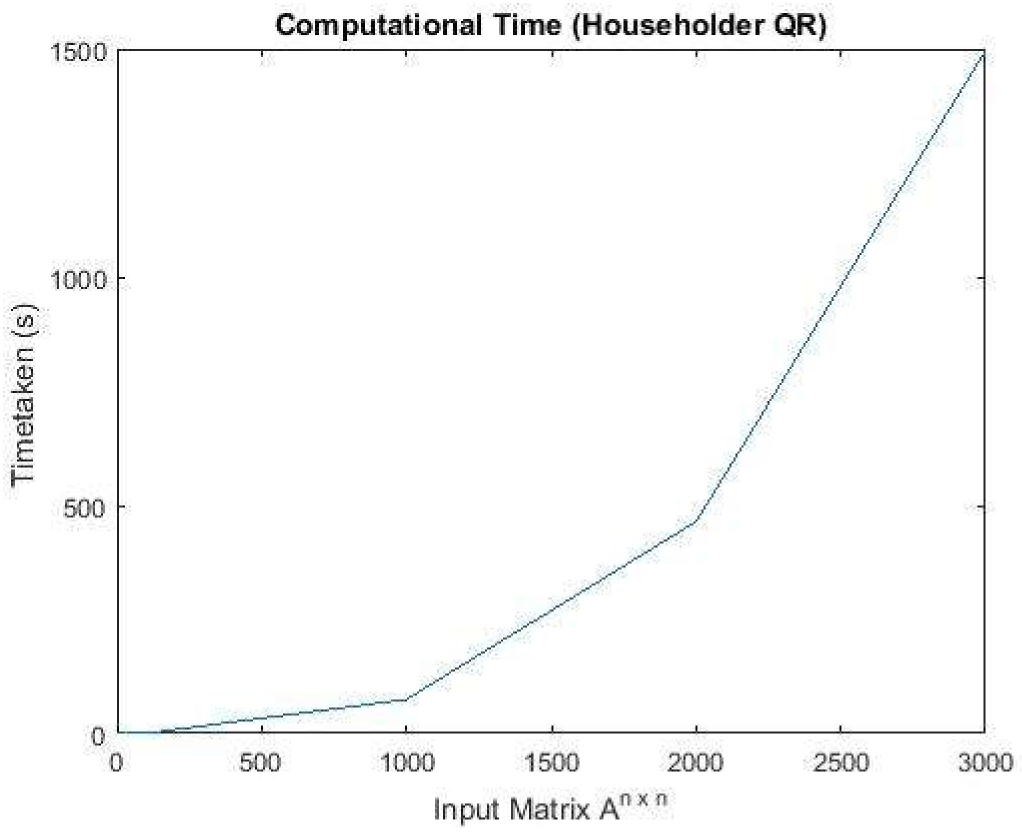
It can be observed from the table above that the time taken to complete the decomposition is roughly equal to the n^3 of the number of input matrix dimension.

QUESTION THREE (3)

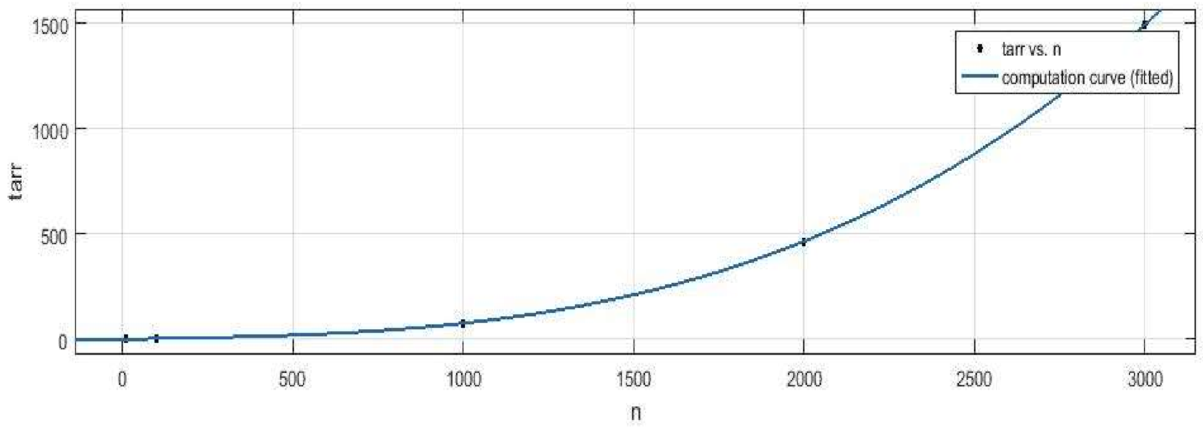
Householder QR decomposition of a matrix was implemented in the file named `houseHolder.m`. The driver program is named `testHouseHolder.m`. The following results were obtained:

Matrix Dimension (nxn)	Time Taken (seconds)
10x10	0.0250
100x100	0.1896
1000x1000	74.6380
2000x2000	465.0581
3000x3000	1495.8253

Below is the graph, **with unfitted curve**, of computational time against input matrix of various dimensions (contained in the table above).



The fitted graph is shown below:



Observation:

It can be observed from the table above that the time taken to complete the decomposition is roughly equal to the n^3 of the number of input matrix dimension. However, it can also be observed from the table above when compared to the one under Modified Gram Schmidt that, Modified Gram Schmidt is relatively faster than Householder QR decomposition.

QUESTION FOUR (4)

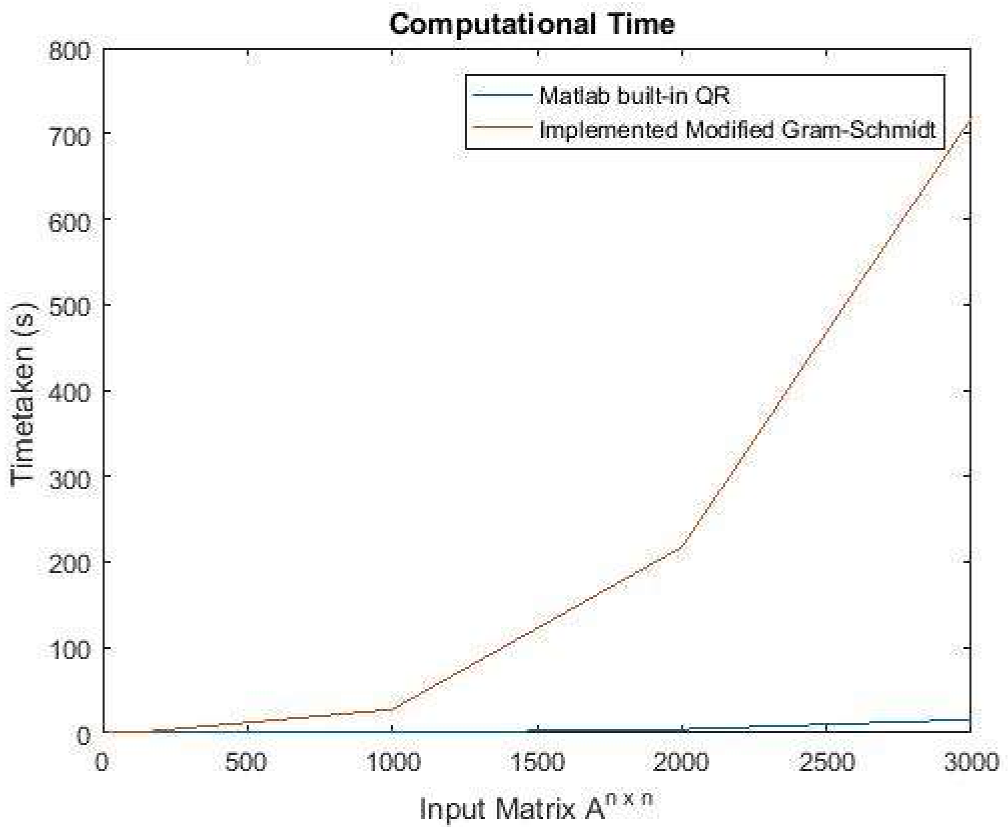
For the solution to this question, MATLAB built-in routine for performing QR decomposition is used, and it is implemented in MATLAB as:

"[Q,R] = qr(A), where A is m-by-n, produces an m-by-n upper triangular matrix R and an m-by-m unitary matrix Q so that A = Q*R.".

(source https://www.mathworks.com/help/matlab/ref/qr.html)

This built-in routine is tested against implemented Modified Gram Schmidt in the file named matlabQR_vs_myQR.m and the following results were obtained:

Matrix Dimension (nxn)	Time (s) Modified Gram S. QR	Time (s) MATLAB QR
10x10	0.0024	0.8797
100x100	0.0800	0.8828
1000x1000	27.5020	0.5042
2000x2000	217.3469	4.4196
3000x3000	717.8990	15.6419



Observation:

It can be deduced from the above table and graph that, built-in MATLAB routine is faster in overall than implemented Gram Schmidt approach to obtaining QR decomposition of a matrix. However, it can also be noted that, when the dimension of the input matrix is 10 x 10 and 100 x 100, the implemented Gram Schmidt method appear to be faster but deteriorates as the input dimension increase geometrically. Therefore, the MATLAB built-in function qr() is faster than the implemented Modified Gram Schmidt based on the obtained data above.

QUESTION FIVE (5)

For the sake of this homework only two algorithms of obtaining the QR decomposition of a given matrix A shall be considered, that is, Modified Gram-Schmidt and Householder Reflection QR decomposition algorithms. Again more emphasis will be laid on the most expensive operations in terms of memory usage and data storage in each of the algorithms. These expensive operation includes but not limited to:

i. Matrix-Vector Product
ii. Vector Cross Product
iii. Inner Product/Dot Product
iv. Norm of a Vector
v. Storage of matrices (both intermediate/temporary and resulting matrices)

Generally speaking, one of the major setbacks, in terms of memory requirement, of QR decomposition as compare to other algorithms is the need to store and operate on new matrices Q and R as there is no way to get that into the physical memory or RAM of machine with input matrix of relatively very large dimension; as conventional RAM has limited size.

However, comparing Modified Gram Schmidt to Householder Reflection QR decomposition, a number of conclusions/deductions can be made based on the implementation of both algorithms shown below:

Modified Gram Schmidt Code

```
function [q, r] = modifiedGS(A)
   m = length(A);
   r(1, 1) = norm(A(:,1));
   assert(r(1, 1) ~= 0);
   q(:,1) = (A(:,1))./ r(1, 1);

   for j = 2:1:m
      q(:,j) = A(:,j);
      for i = 1:1:j-1
          r(i, j) = dot(q(:,j), q(:,i));
          q(:,j) = q(:,j) - r(i, j)*q(:,i);
      end
      r(j, j) = norm(q(:,j));
      assert(r(j, j) ~= 0);
      q(:,j) = q(:,j) / r(j, j);
   end
end
```

Householder Reflection QR Decomposition

```
function [q, r] = houseHolder(r)
    [n, m] = size(r);
    I = eye(n, n);
    for j = 1:m
        for ii = 1 : j - 1
            r(:, j)=r(:,j)-(1+omega(ii))* w(:, ii)*dot(w(:, ii),r(:, j));
        end
        for k = 1:n
            if k < j; z(k, 1) = 0;
            elseif k == j; z(k,1)= r(k,j)+ exp(i.*angle(r(k, j))).* norm(r(k:end,j));
            elseif k > j; z(k, 1) = r(k, j);
            end
        end

        w(:, j) = z/norm(z);
        omega(j) = dot(r(:, j), w(:, j)) / dot(w(:, j), r(:, j));
        r(:, j) = r(:, j) - (1 + omega(j)) * w(:, j) * dot(w(:, j), r(:, j));
        q(:, j) = I(:, j)-(1 + omega(j)) * w(:, j)* dot(w(:, j), I(:, j));

        for ii = j - 1: -1: 1
            q(:,j)=q(:, j)-(1+omega(ii)) * w(:, ii)* dot(w(:, ii), q(:, j));
        end
    end
end
```

Comparing these two algorithms based the above mentioned criteria, we have

	Modified Gram Schmidt	Householder Reflection
Matrix-Vector Product	Nil	Nil
Vector Cross Product	Nil	Nil
Inner Product/Dot Product	1	6
Norm of a Vector	2	2
Entry-wise Arithmetic operation a vector	4	11
Storage of matrices	3 (q, r, A)	3 (q, r, I)

Observation:

From the table above it is obvious that Householder QR decomposition use more memory in terms operations on the vectors than Modified Gram Schmidt algorithm. However, despite the fact that Householder method has been carefully implemented to store the new upper triangular matrix R into the input matrix A but the presence of identity matrix I in the Householder QR has also worsened the case and made it to still come in the same level as Modified Gram Schmidt with total number of stored matrices to be 3.

QUESTION SIX (6)

The QR algorithm is an algorithm used as an alternative method for calculating eigenvalues and eigenvectors of a matrix. It is sometimes referred to as QR iteration (because of its iterative approach) or Francis Algorithm (named after the its developer).

QR algorithm is one of the simplest algorithm to implement and understand because of its very simple approach.

Drawback of QR Algorithm

Unlike LU and QR factorization which have finite number of steps, QR algorithm has infinite process based on its iterative approach and basically when to stop the iteration is a question yet to be answered. Its iteration never closes.

The QR Algorithm

What are we look for in QR algorithm?

We are interested in finding a sequence of unitary transformations such that:

$(U_k^* \dots U_1) A (U_1 \dots U_k) = T$ (upper triangular matrix)

The diagonal entries of the upper triangular matrix T, are the eigenvalues of the matrix A.

How do we get to that?

Consider a matrix A which can be re-written as given below:

$A = U T U^*$ (where U is a unitary matrix)

$T = U^* A U$

Using the above we relationship we can then

$A_1 = U_1^* A U_1$

$A_2 = U_2^* A_1 U_2$ (this continues up to k, where $k \rightarrow \infty$)

Substitute A_1 into A_2

$A_2 = U_2^* U_1^* A U_1 U_2$

From above it can be derived for a generic case or as algorithm as follow:

$(U_k^* \dots U_1) A (U_1 \dots U_k) = T$

Where A_1, A_2 used above represent upper triangular matrix (previously used as T), again they represent each level of iteration. That is, A_1 is the resulting matrix at first iteration and A_2 is the resulting matrix at the second iteration … iteration continues until the desired iteration (k) value or practically speaking, iteration can be stopped when the diagonal entries of the resulting upper triangular matrix A_k (which is also the eigenvalue of matrix A) approaches the exact eigenvalue of matrix A.

Implementation of QR Algorithm

This algorithm is implemented in the file name `QRAlgorithm.m`. The results obtained for A 5 by 5 matrix for different iteration values (k) are shown below:

	k=5	k=10	k=15	k=20	k=25	k=30	k=35	K=40
λ_1	37.4296	46.9200	45.8666	45.9697	45.9595	45.9605	45.9604	45.9604
λ_2	-20.2689	-29.9034	-28.8499	-28.9530	-28.9428	-28.9438	-28.9437	-28.9437
λ_3	6.6506	6.7955	6.7954	6.7954	6.7954	6.7954	6.7954	6.7954
λ_4	-0.8487	-0.8496	-0.8496	-0.8496	-0.8496	-0.8496	-0.8496	-0.8496
λ_5	0.0375	0.0375	0.0375	0.0375	0.0375	0.0375	0.0375	0.0375

The obtained eigenvalues from the same matrix by MATLAB built-in function **eig(A)** are:

Λ = 45.9604

-28.9437

6.7954

-0.8496

0.0375

Observation:

From the table and exact results obtained by built-in MATLAB function for calculating eigenvalues of a matrix, it can be observed that at iterations $k < 30$ the obtained eigenvalues are not equal to the exact values but close to it, however from iteration $k >= 30$, the obtained eigenvalues are the same with the one calculated using MATLAB built-in routine.

References

(Strang, Linear Algebra and its Application, 2006, p. 364)

https://en.wikipedia.org/wiki/QR_algorithm