QUESTION ONE (1)

Computational complexity of an algorithm can be sub-divided into

1. Time Complexity
2. 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:

1. Number of Processors (Single or Multiple)
2. Memory/Disk Read/Write Speed
3. System Architecture (32-bit or 64-bit)
4. 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

1. It has just single processor
2. It is made of 32-bit architecture
3. It has sequential execution of commands
4. It takes one (1) unit time for basic arithmetic and logical operations like addition, multiplication, division, subtraction, greater than, less than and so on.
5. It takes 1unit of time for assignment
6. It takes 1unit of time for increment/decrement
7. 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.

Inner Product (Dot Product) Complexity

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

Consider v = [v1 v2 v3 ... vn]

u = [u1 u2 u3 ... un]

therefore, dot(v, u) = v1\*u1 + v2\*u2 v3\*u3 ... vn\*un

Finally,

Number of Additions = n – 1

Number of Multiplications = n

Total complexity = n – 1 + n = **2n - 1**

Matrix – Vector Product Complexity

Consider a matrix Amxn and a vector xnx1, product Ax is given by:

a11 a12 a13 ...a1n  x1

a11 a12 a13 ...a1n x2

. .

. .

. .  . . .

am1 am2 am3 ...amn xn

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 x).

Therefore, complexity of Ax = (2n – 1) \* m

= **2n2 – n** (for a square matrix)

Norm of a Vector Complexity

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

Consider v = [v1 v2 v3 ... vn] for all VϵRn

norm(v) = √(v1\*v1 + v2\*v2 v3\*v3 ... vn\*vn)

Number of additions (+) = n – 1

Number of square root (√) = 1

Number of multiplication (\*) = n

Total complexity = n – 1 + n + 1 = **2n**

Modified Gram Schmidt Algorithm Complexity Analysis

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):

|  |  |  |
| --- | --- | --- |
| **Gram-Schmidt Algorithm Matlab Code** | **Cost** | **No of times** |
| 1. function [q, r] = modifiedGS(A)  2. m = length(A);  3. r(1, 1) = norm(A(:,1));  4. assert(r(1, 1) ~= 0);  5. q(:,1) = (A(:,1))./ r(1, 1);  6.  7. for j = 2:1:m  8. q(:,j) = A(:,j);  9. for i = 1:1:j-1  10. r(i, j) = dot(q(:,j), q(:,i));  11. q(:,j) = q(:,j) - r(i, j)\*q(:,i);  12. end  13. r(j, j) = norm(q(:,j));  14 assert(r(j, j) ~= 0);  15. q(:,j) = q(:,j) / r(j, j);  16. end  17. end | 2  **2n**+1  1  *n*+1  2  1  2  (**2n-1**)+1  2+*n*  1+**2n**  1  1+*n*  1 | 1  1  1  1  m-1  m-1  (m-1)(m-2)  (m-1)(m-2)  (m-1)(m-2)  m-1  m-1  m-1  1 |
| **Note:** The boldened parameters in “cost” column rep. pre-calculated values and italized ones are arithmetic operation on a vector; and its complexity is of T(n) = n |  |  |

Time complexity, T(n) = 2 + 2n + 1 + 1 + n + 1 + 2(m-1) + m – 1+ 2(m-1)(m-2) +

2n(m-1)(m-2) + (2+n)(m-1)(m-2) + (1+2n)(m-1) + m – 1 + (1+n)(m-1) + 1

= 3nm2 + 4m2 – 6nm + 2n – 2m + 8

= 3n3 – 2n2 + 8 (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 no, f(n) ≤ cg(n), for n≥ no}

Therefore, from calculate complexity above:

f(n) = 3n3 – 2n2 + 8

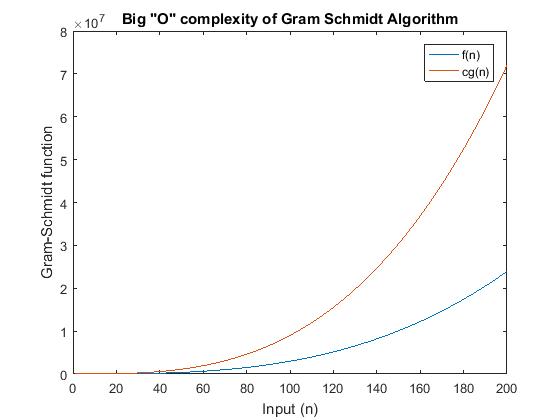
g(n) = n3

c = 3 – 2 + 8 = 9, no = 1, f(n) ≤ 9n3 for all n≥ no

For non – square matrix, f(n) = 3nm2 + 4m2 – 6nm + 2n – 2m + 8, and as **n** and **m** tends to infinity, then the other terms disappear leaving only **3nm2.**

Therefore, Gram Schmidt has big ‘oh’ complexity of **O(n3)** for square matrix or **O(nm2)** 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 no.

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) = 2nm2 + 2n2m + 7m2 + 4nm +12m + 3

= 4 n3 + 11 n2 + 12n + 3 (for square matrix, n=m)

=  **n3 + n2 + 4n + 1**

Big “O” Complexity

Therefore, from calculate complexity above:

f(n) = 4 n3 + 11 n2 + 12n + 3

g(n) = n3

c = 4 + 11 + 12 +3 = 30, no = 1, f(n) ≤ 30n3 for all n≥ no

For non – square matrix,

f(n) = 2nm2 + 2n2m + 7m2 + 4nm +12m + 3,

for square matrix

f(n) =  **n3 + n2 + 4n + 1**

and as **n** and **m** tends to infinity, then the other terms disappear leaving only **2nm2 + 2n2m** (for non-square) and  **n3** (for square matrix) **.**

Therefore, Gram Schmidt has big ‘oh’ complexity of **O(n3)** for square matrix or **O(nm2 + n2m)** for non-square matrix.

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