ECE 7650 (Advance Matrix Algorithm)

PROJECT REPORT

On

"Adaptive Cross Approximation Algorithm"

By

Jamiu Babatunde Mojolagbe

(Student ID: #7804719)

Department of Electrical and Computer Engineering

University of Manitoba

Submitted to

Ian Jeffrey, PhD

(Course Instructor)

CHAPTER ONE

1. Introduction

The sole purpose of this project is to explore the generation of low-rank approximation, in form of an outer product approximants, of rank deficient matrices by an algorithm known as "Adaptive Cross Approximation (ACA)". Perusal into merits, demerits as well as complexity of the algorithm in contrast with existing traditional method (Singular Value Decomposition) is also presented.

While as contained in the original paper proposed by Bebendorf et al in [1-3], the purpose of the algorithm is to obtain low-rank approximants for matrices obtained from solving integral equations – which are in fact, full rank, but contains rank deficient sub-blocks. However, for the sake of this presentation, the performance of this algorithm was based on randomly generated rank deficient matrices.

1.1 Merit of ACA

With ACA the memory requirement and CPU time requirement are less compare to the tradition truncated SVD. It has been shown that ACA resulted in **O(nlogn)** while it has been determined that the computational (time) complexity of SVD is said to be $\mathbf{O((nm^2 + mn^2)}$ according to [5]; where **n** is the number of rows and **m** is the number of columns. ACA requires **O(r(m+n))** memory storage for a given matrix A^{mxn}.

One more beautiful thing about this algorithm is its rank revealing nature and as such can be dubbed rank-revealing LU decomposition [1] when the input matrix is full rank - without undergoing computationally demanding Modified Gram Schmidt process.

1.2 Demerit of ACA

While the advantages of ACA are something to write home about, it as well as its setbacks, though its merits outweigh its disadvantages.

It has been observed that when a matrix is singular, ACA breaks down while the traditional SVD still works fine.

CHAPTER TWO

2. The Algorithm

The mathematical formulations of this algorithm are left out from this presentation, however those formulations can be found in the original papers [1-3]. The focus, here, would be on how the algorithm works cum its implementation.

2.1. Outline of ACA

Consider a matrix $A \epsilon \mathbb{R}^{mxn}$ of rank r which is r << min (m, n) or r < min(m, n). This matrix A can be represented as a low rank matrix as follow:

$$A = \sum_{i=1}^{r} u_i v_i^T \qquad where\ u_i \epsilon \mathbb{R}^m \qquad v_i \epsilon \mathbb{R}^n$$

As the name of the algorithm suggests, ACA algorithm aims at approximating a given matrix A^{mxn} to an approximant matrix $\tilde{A}^{mxn}$; however it does so through a product form that looks like the one presented above as follows:

$$\tilde{A}^{mxn} \quad = \quad U^{mxr} V^{rxn} \quad = \quad \sum_{i=1}^{r} u_i^{mx1} v_i^{1xn}$$

where "r" is the effective rank of original matrices A^{mxn} (for which approximation is shown above), U^{mxr} and V^{rxn}.

The goal of ACA is to achieve:

$$\|R^{mxn}\| \quad = \quad \left\|A^{mxn} \quad - \quad \tilde{A}^{mxn}\right\| \quad \leq \quad \varepsilon \|A^{mxn}\|$$

where "ε" is a given error tolerance and R is the error matrix.

Therefore, instead of storing the entire mxn entries of original matrix A, the algorithm only requires to store $(m + n)r$ entries of the approximants.

2.2. Implementation Steps of ACA

1st iteration: input matrix A, error tolerance ε

i. Randomly choose starting row index I_1 such that $I_1 \leq \min(m, n)$

ii. **Set approximant matrix $\tilde{A}$ to zero

iii. Set the first row of the error matrix, R such that $R(I_1, :) = A(I_1, :)$

iv. Obtain J_1 - the first column index – such that $absolute(R(I_1, J_1) = absolute(R(I_1, j_{max}))$

v. Calculate first approximant vector $v_1 = \frac{R(I_1, :)}{R(I_1, J_1)}$

vi. Set the first column of the error matrix, R such that $R(:, J_1) = A(:, J_1)$

vii. Calculate first approximant vector $u_1 = R(:, J_1)$

viii. Obtain approximate matrix $\tilde{A}$, $\tilde{A} = sqrt(\|u_1\|^2 \|v_1\|^2)$

ix. For the next iteration, obtain row index I_2
such that $absolute(R(I_2, J_1) = absolute(R(i_{max}, J_1))$ and $i \notin I$

For iteration, k = 2 to rank (r) of matrix A do

x. Update error matrix R ($I_k th$ row), $R(I_k, :) = A(I_k, :) - \sum_{l=1}^{k-1} (u_l)_{I_k} v_l$

xi. Obtain J_k - the kth column index – such that $absolute(R(I_k, J_k) = absolute(R(I_k, j_{max}))$ and $j \notin J$

xii. Calculate kth approximant vector $v_k = \frac{R(I_k, :)}{R(I_k, J_k)}$

xiii. Update error matrix R ($J_k th$ column), $R(:, J_k) = A(:, J_k) - \sum_{l=1}^{k-1} (v_l)_{J_k} u_l$

xiv. Calculate kth approximant vector $u_k = R(:, J_k)$

xv. Obtain approximate matrix $\tilde{A}^k$

$$\tilde{A}^k = sqrt(\left\|\tilde{A}^{k-1}\right\|^2 + 2 \sum_{j=1}^{k-1} \left|u_j^T u_k\right| \cdot \left|v_j^T v_k\right| + \|u_k\|^2 \|v_k\|^2)$$

xvi. If $\|u_k\| \|v_k\| \leq \varepsilon \left\|\tilde{A}^k\right\|$ then the iteration has converged, so exit the iteration return u, v

xvii. Else continue the iteration by obtaining row index I_{k+1} for the next iteration, such that that $absolute(R(I_{k+1}, J_k) = absolute(R(i_{max}, J_k))$ and $i \notin I$

Note: This algorithm as shown decomposes a matrix into sum of low-rank approximants and error matrix – which are not computed completely or explicitly as a matrix. Again the choice of error tolerance ε, is of utmost important in ACA in order to avoid undesirable numerical error in the approximation; as used in this project, the tolerance was set to be very small $1e^{-8}$.

2.3 Implementation Note

Functions called "**aca.m**" and "**aca2.m**" were implemented in MATLAB as shown in section 2.2 above. The main differences between the two variants are the choice of row and column indices. While as shown in the algorithm presented in the previous page, the first starting row index is chosen randomly and column index is obtained by taking the index of absolute maximum of the chosen row. While for the consecutive row index I and column index J, similar procedure is followed. This is the way function "**aca2.m**" was implemented. Whereas for "**aca.m**", it follows a different approach, in that the choice of first row and column indices are set to be fixed and are chosen to be 1 in each case. Such that the consecutive row or column index is obtained by incrementing the previous index.

The implementation files are well commented and the details of the input parameters as well as return values are well documented.

The driver program for this functions is named "**drive.m**".

The input matrices supplied to these function were obtained from a function called "**createRankDefMatrix.m**". The function returns rank deficient matrix depending on the parameters supplied to it – this function also is well documented and details of the inputs and the return values are contained in the implementation file.

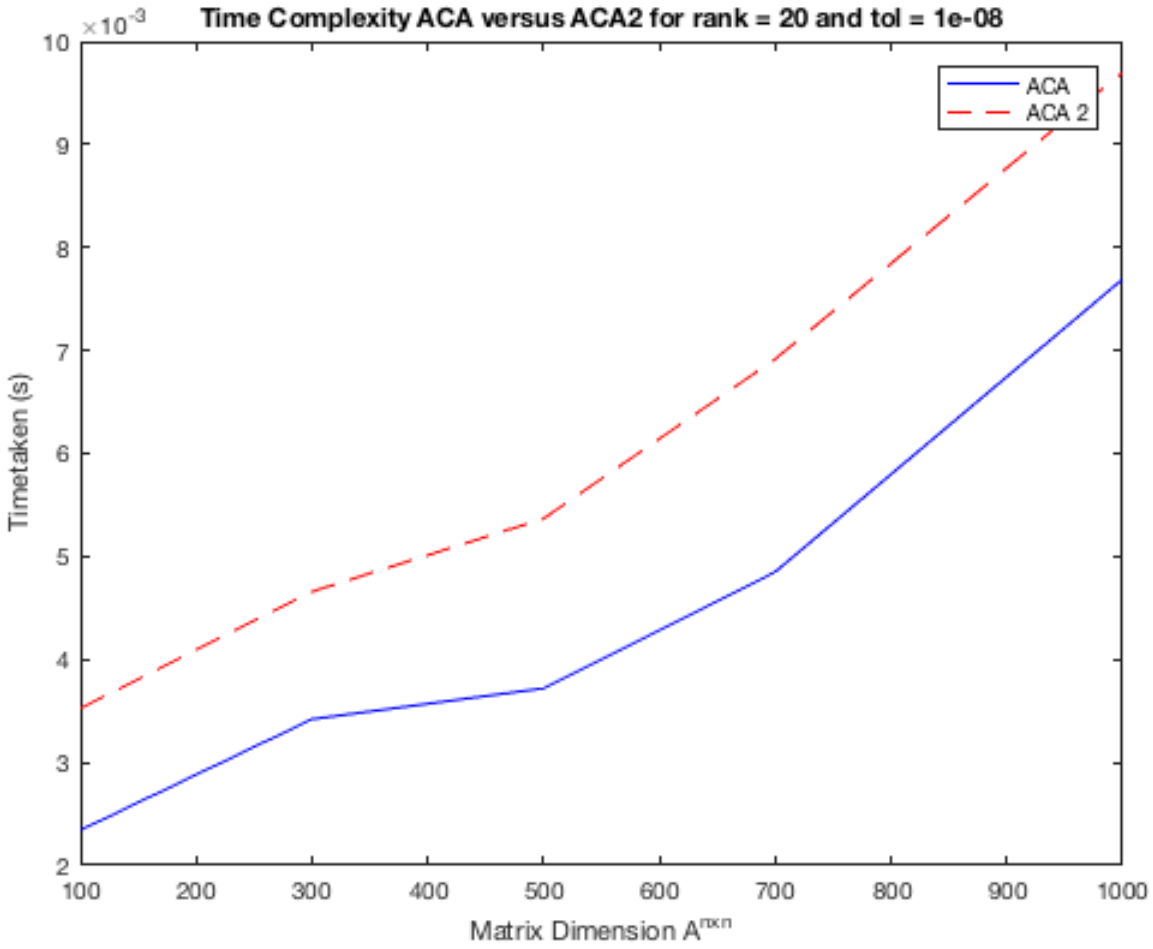
For comparison purpose, truncated SVD was implemented and its rough time complexity was taking into account.

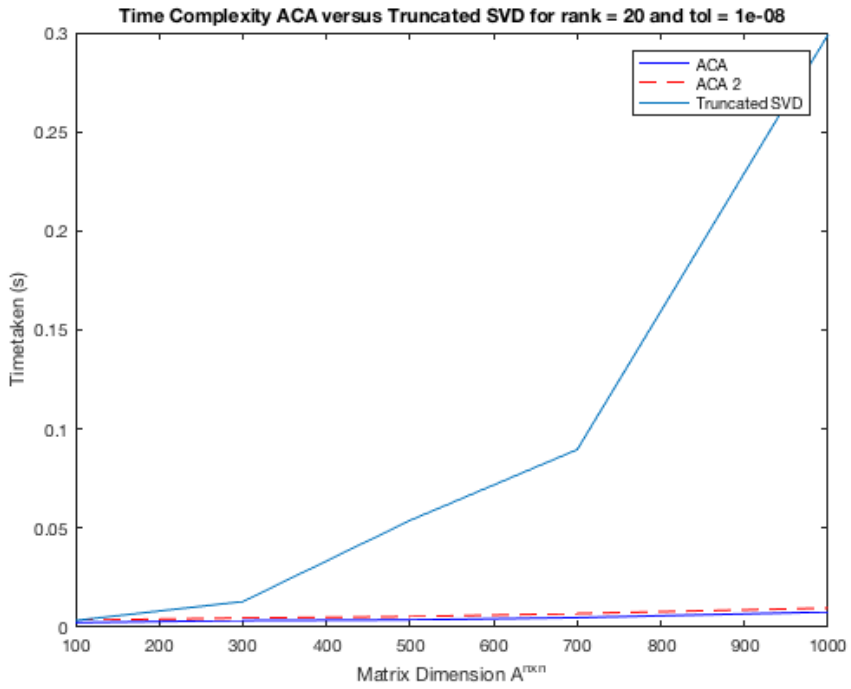
CHAPTER THREE

3.0 Results and Discussions

Case 1: For different matrices sizes for error tolerance, **tol = 1e^{-8}** and **rank (k) = 20**, the following results were obtained.

Dimension	ACA		ACA2		Truncated SVD	
	Time (s)	**Error**	**Time(s)**	**Error**	**Time (s)**	**Error**
100x100	0.002458530	1.214105e-10	0.003687960	9.559092e-11	0.002284672	2.244309e-13
300x300	0.003018485	1.137624e-10	0.004531656	5.150625e-10	0.016965700	1.031883e-12
500x500	0.005170283	4.609999e-11	0.006058068	1.082159e-10	0.052373124	1.554796e-12
700x700	0.005467378	2.525379e-10	0.007434528	2.614841e-10	0.101663492	2.825593e-12
1000x1000	0.008140703	2.469771e-09	0.009964337	4.702763e-09	0.283906691	6.255521e-12





Case 2: The algorithm behavior was observed when the matrix is full rank and it is of dimension 10x10. The following decomposition was obtained from the first variant named ACA for **U** and **V** respectively:

```
>> u

u =

    1.2232         0   -0.0000    0.0000    0.0000    0.0000    0.0000    0.0000   -0.0000    0.0000
   -2.0538    0.0766   -0.0000    0.0000    0.0000    0.0000    0.0000    0.0000   -0.0000    0.0000
   -0.5929    1.5698 -119.8148   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000   -0.0000
   -0.4553   -2.1868  169.1335    3.0241    0.0000         0   -0.0000   -0.0000    0.0000   -0.0000
    1.6702   -1.3297   99.4937   -1.6419    1.3981         0         0         0         0   -0.0000
   -3.8882   -1.0733   92.8400   -2.6324   -5.2956   -2.2500   -0.0000   -0.0000         0    0.0000
   -2.3481   -2.2870  179.0578    8.5541   -3.6340   -0.5903    1.3105         0   -0.0000    0.0000
   -0.9465   -1.4175  111.9382   -1.1924   -2.2682   -6.5420  -62.5018 -220.8599   -0.0000    0.0000
    0.5304   -1.4124  111.3451   -4.5563   -0.1325   -4.1629  -38.1431 -133.7237   -1.9834         0
    0.7359    3.3501 -265.7693    6.0227    1.4634   12.1165  116.7998  412.3955    5.8189    0.4394

>> v

v =

    1.0000    0.0937    1.4874    5.9266   -2.5627    1.7022   -1.7443    0.3054    3.4920   -4.1653
         0    1.0000   77.8279  196.3128  -88.6889   38.1363  -63.7162   -9.4335  198.1915 -106.8773
         0         0    1.0000    2.5331   -1.1301    0.5183   -0.8380   -0.1203    2.6013   -1.4417
         0         0   -0.0000    1.0000   -1.1783   -0.8398    0.8067    1.0104   -0.2367    1.5735
         0         0    0.0000         0    1.0000   -0.1396    3.9798    4.2647   -2.0541    5.0978
         0    0.0000    0.0000   -0.0000   -0.0000    1.0000  -11.4587  -12.6717    9.3022  -15.1212
         0         0    0.0000   -0.0000         0         0    1.0000   -2.3811   -5.3655    1.3531
         0   -0.0000   -0.0000    0.0000    0.0000    0.0000    0.0000    1.0000    1.2924   -0.0355
         0         0    0.0000         0         0         0   -0.0000    0.0000    1.0000    0.3837
         0         0   -0.0000   -0.0000         0   -0.0000    0.0000    0.0000         0    1.0000
```

It can be observed from the tables and the graphs, that the first variant of Adaptive Cross Approximation nick named ACA seems to be faster than the other variant nick named ACA2 and truncated SVD. On the same note, the error obtained tend to be very close to the truncated SVD ones thereby leading to a good approximation within certain desired tolerance while there is saving in terms of memory usage and time complexity. Another thing that was noted whose results are not included was that, when the input matrix was singular then the Adaptive Cross Approximation algorithm broke down while the truncated SVD still produced the desired result.

It was also observed that the first variant, ACA, when the matrix was full rank produced **U** as a lower triangular matrix and **V** as a unit upper triangular matrix; this indeed is LU decomposition. This was also carefully observed and noted by [4].

CHAPTER FOUR

4.0 Conclusions and Recommendations

It can be concluded from the foregoing that ACA algorithm approximates the original matrix by requiring only partial knowledge of the original matrix. The procedure requires **O(r(n+m))** memory storage which is a good result.

Again, the first variant nick named ACA is proposed as it scaled faster than the other variant – ACA2, though, the setbacks of the first variant as regard the fixed choice of indices were not investigated.

Moreover, it can be said Adaptive Cross Approximation is a rank – revealing LU decomposition as it produces LU decomposition of input matrix A, when the input matrix is full rank.

Finally, it can be concluded that for representation or approximation of low-rank matrices Adaptive Cross Approximation is recommended.

References

[1] M. Bebendorf, “Approximation of boundary element matrices,” *Numer. Math.*, vol. 86, no. 4, pp. 565-589, Jun. 2000.

[2] S. Kurz, O. Rain, and S. Rjasanow, “The adaptive cross-approximation technique for the 3-D boundary element method,” *IEEE Trans. Magn.*, vol. 38, no. 2, pp. 421–424, Mar. 2002.

[3] M. Bebendorf and S. Rjasanow, “Adaptive low-rank approximation of collocation matrices,” *Computing*, vol. 70, no. 1, pp. 1–24, Mar. 2003.

[4] K. Zhao, M. N. Vouvakis and J. Lee, “The adaptive cross approximation algorithm for accelerated method of moments computations of EMC problems,” *IEEE Trans. Magn.*, vol. 47, no. 4, pp. 763–773, Nov. 2005.

[5] Z. Liu et al, “Using adaptive cross approximation for efficient calculation of monostatic scattering with multiple incident angles,” *ACES Journal*, vol. 26, no. 4, pp. 325–333, Apr. 2011.