Efficient Architectures for Eigen Value Decomposition

Nikhil Suryanarayanan
Dept of Electrical & Computer Engineering, University of Wisconsin - Madison

Abstract - The project presents hardware efficient architectures for Eigen Value Decomposition (EVD). Iterative and Parallel architectures are explored. The implementations have been compared in terms of speed, number of computations and number of Processing elements required. The software implementation of the architectures has been done using MATLAB.

Keywords: EVD, CORDIC, Systolic

I .INTRODUCTION

EVD is a very important matrix factorization used in several applications. The most intensive applications are source localization, principal component analysis and beam forming. EVD is a special case of Singular Value Decomposition (SVD). Major applications mentioned above use square symmetric matrices for analysis and hence focus on EVD rather than SVD is more relevant. The algorithm and implementations can definitely be extended to perform SVD with subtle changes.

The SVD/EVD of a real matrix \( A \in \mathbb{R}^{m \times n} \) be factorized using the form:

\[
\text{SVD: } A = UDV^T \\
\text{EVD: } A = UDU^T, \quad (U=V)
\]

Where, \( D \in \mathbb{R}^{m \times n} \) is diagonal matrix, \( U \in \mathbb{R}^{m \times m} \) & \( V \in \mathbb{R}^{n \times n} \) are orthogonal. This implies \( U^T U = I \) and \( V^T V = I \). \( U \) is a set of left singular vector, while \( V \) is a set of right singular vector. The elements of \( D \) are the ‘singular values’. In case of EVD, the diagonal elements are called the Eigen Values of the matrix.

In the radar signal processing, the target signal is often covered up in a clutter background. Real time filter can be used to extract the weak signals and suppress the clutter through clutter autocorrelation matrix Eigen value/eigenvector decomposition (EVD). Traditional decomposition methods cost more computational quantity and longer time delay, thus they are unable to meet real-time processing requirements. In order to find a rapid and efficient way to achieve matrix decomposition, experts conduct in-depth research. Among the results, the EVD algorithm based on two-sided Jacobi rotation attracts our attention because of the high value stability. R. P. Brent, F.T.Luk and others presented two corresponding algorithms.

There are many numerically stable algorithms for computing the SVD such as the Jacobi algorithm, the QR method and the one sided Hestenes method. For parallel implementations, the Jacobi method is far superior in terms of simplicity, regularity, and local communications. Brent, Luk and Van Loan have shown how the parallel Jacobi algorithm can compute the SVD of a square \( N \times N \) matrix in \( O(N \log N) \) time. This is to be compared with the best serial algorithms, which have a \( O(N^3) \) time complexity.

First was serial. Matrix is divided into a serial of 2 * 2 sub matrix to be computed one by one. We can acquire the Eigenvalues and the eigenvectors at the same time. Its shortcomings are involved in multiplication, division and other complex operations, difficult to achieve, and longer-lapse operations. Second was parallel. It is of parallel computation of EVD using Jacobi-like algorithms on processor arrays. Comparing to the first, the second overcomes the shortcomings of the long lapse with an array processing. But complex arithmetic is still need. The salient features of this algorithm are escape from multiplication and division, administration, and other complex operations, only involving coordinates - operation. Hence, CORDIC algorithm is particularly
implemented for Eigen value decomposition. CORDIC arithmetic is an efficient map of Jacobi-like algorithm to hardware.

The project presents arrays processing methods based on serial and parallel Jacobi EVD algorithm, to calculate the eigenvector with the features such as avoiding complex operations, saving operation lapse etc.

II. CORDIC ALGORITHM

Before introducing the algorithms, a brief introduction of CORDIC (COordinate Rotation DIgital Computer) will be useful. CORDIC is an iterative algorithm and extremely hardware friendly. One of the first steps of optimization of the project has been to unroll the iterations in the CORDIC algorithm for a high throughput. It involves a set of Shift-Add operations for computing Sine, Cosine, Arc, Hyperbolic, Coordinate Rotations. It eliminates complex computations. A single shift-add multiplier and a ROM for obtaining the constant values is sufficient. The hardware implementation also has Basic Logic Gates.

Loop Unrolling is represented in Fig 1. The iteration stage has 8 pipeline stages and the normalization stage has a latency of two clock cycles. Hence in a total delay of 10 pipelines stages, we can get the output.

The convergence of the algorithm depends in the number of iteration. The unrolled pipelined implementation is extremely suitable for recent FPGAs. Most recent FPGAs have an inbuilt registers in each of the cells, hence it is extremely suitable for pipelined implementations. The iterative equations used in CORDIC are given by Eq. 3, 4 & 5. Eq. 1 and 2 represent the Vector rotations

\[ x = x \cos \Phi - y \sin \Phi \quad (1) \]
\[ y = y \cos \Phi + x \sin \Phi \quad (2) \]

The iteration equations are given as

\[ x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-l} \quad (3) \]
\[ y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-l} \quad (4) \]
\[ z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-l}) \quad (5) \]

CORDIC has been used for implementation of two modules namely, ArcTan and Sin/Cos modules. The ArcTan module is used to calculate the tangent inverse while the Sin/Cos module is used to calculate the Sine and Cosine for any given angle.

III. JACOBI SVD/EVD Algorithm

The Jacobi method generates the matrices U and V by performing a sequence of orthogonal two sided plan rotations to the input matrix with the property that each new matrix Ai is 'more diagonal' than its predecessor. The algorithm aims at annihilating the off diagonal elements using a series of orthogonal transformations. The equations can be represented as follows

\[ A(k + 1) = J_{pq}^T A(k) J_{pq} \quad (6) \]

The matrix J is called the Jacobi matrix and is of the form J (p, q, \( \theta \)).

It can be represented as

\[ J(\theta) = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \quad (7) \]

It is observed that

\[ J'(\theta) \begin{bmatrix} a & c \\ c & d \end{bmatrix} J(\theta) = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \quad (8) \]
\[ \theta = \frac{1}{2} \tan \left( \frac{2\pi c}{b-a} \right) \] (9)

The linear transformation affects only the \( p \)th, \( q \)th row and column respectively. All other columns remain unaffected. It can be pictorially represented as shown in Fig 2. After \( n \) iterations the input matrix \( A \) is converted to diagonal matrix \( A_n \). The angle \( \theta \) is chosen to annihilate the off-diagonal elements.

By repeatedly performing the iterations we can effectively diagonalize the input matrix \( A \). \( \sigma_1 \) & \( \sigma_2 \) represent the Eigen values of the Matrix.

![Figure 2](image-url)  
*Jacobi Rotations on Matrix*

But there are certain limitations associated with the Exact Jacobi method. The Jacobi iterations are highly serial in nature. There is very little scope for parallelism because of the large inter-loop Data dependency. Every iteration of the Jacobi Algorithm would require transfer of 4\( N \)-4 matrix elements to the processor. It can be that even though the operation involves matrices, there is very minimal parallelism exhibited by the Algorithm.

To reduce the number of iterations, the input matrix is initially Tri-diagonalized and then the Jacobi Algorithm is applied. This essentially reduces the number of iterations that are required to diagonalize the matrix and we can perform EVD in lesser number of iterations. But again the data transfers for each iteration remain the same and it cannot be executed in parallel in the original form.

**IV. SCOPE FOR PARALLELISM**

The above algorithm can be parallelized by dividing the problem into 2\( x \)2 EVD sub problems. Hence a matrix of size \( N \times N \) will have \( N/2 \times N/2 \) sub problems. In case \( N=6 \), then the possible sub-problems are \{ (1,2), (3,4) \}, \{ (1,3), (2,4) \}, \{ (1,4), (2,3) \}. It is also known as parallel reordering of the algorithm. All \( (p, q) \) pairs within each set are non conflicting.

Sub-problem (1,2) and (3,4) can be carried out in parallel, likewise sub-problem (1,3) and (2,4) can be executed in parallel as can sub-problem (1,4) and (2,3). This way of ordering sub-problems is called "parallel ordering" it allows the execution of \( N/2 \) sub-problems in parallel. The number of sets is \( N-1 \) and a sweep consists of \( N-1 \) steps of \( N/2 \) rotations executed in parallel.

**V. SYSTOLIC ARRAY**

Any matrix can be operated in parallel by exploiting the parallelism between individual sub-blocks of the matrix. In case of EVD also, the same applies. Eigen values can be associated as the signature for that particular matrix. Hence while devising any algorithm, we have to ensure that operations on the sub-blocks of the matrix does not alter the Eigen values of the matrix.

*a. Direct Systolic Implementation*

The systolic array (Fig. 3) is a square systolic array that implements a parallel Jacobi SVD/EVD algorithm using \((N/2)^2\) processors. It performs \( N/2 \) sub-problems in parallel, and a sweep in \( N-1 \) steps. As the number of sweeps for the SVD/EVD of \( N \times N \) matrix is \( O (\log N) \), this systolic array is capable of processing the SVD/EVD of a square matrix with \( O(N \log(N)) \) time complexity.
The angle $\theta$ is generated by the processor on the main diagonal of the array. The diagonal processors broadcast the angles, along the row (represented as $\theta_r$) and the column (represented as $\theta_i$) corresponding to their position on the off diagonal of the array. Initially a processor $P_{ij}$ holds a $2 \times 2$ sub-matrix. A step of the Jacobi algorithm consists of the solution of a $2 \times 2$ SVD/EVD problem, this is done by the diagonal processors which annihilate their two off-diagonal elements and then the application of the computed rotations to the matrix $A(3)$, which is executed by each processor applying two sided rotations to its $2 \times 2$ sub-matrix.

After the application of the rotations, the local matrices are interchanged (Figure 3) between the processors for the execution of a new step. To avoid broadcasting the rotation parameters at constant time along processor rows and columns, the rotation parameters are constant speed between adjacent processors. A processor cannot commence a rotation until data from previous rotation are available on all its lines. Thus a processor $P_{ij}$ stays idle for two time steps while waiting for $P_{i+1,j}$ to complete their (possibly delayed) steps. The price paid to avoid broadcasting is that each processor is active for one third of total computation.

Every node of the systolic array is a Processing Element (PE). Each processing element consists of a CORDIC Atan and CORDIC Rot modules. Every

The location of elements of a $2 \times 2$ sub-matrix namely $[\alpha \beta; \gamma \delta]$ is not the same in all of the PEs of the systolic array. Depending on the location of the processing element, the arrangement of element varies. For the PE located at (1,1) the arrangement is $[\alpha \beta; \gamma \delta]$. For a PE located on the first row, namely $P_{1j}$, the arrangement is $[\beta \alpha; \gamma \delta]$. For a PE located on the first column namely, $P_{i1}$, the arrangement is $[\gamma \delta; \alpha \beta]$ and for any other PE the arrangement is $[\delta \gamma; \beta \alpha]$. There is a rhythmic exchange sequence which can be better represented diagrammatically using Fig 5.

![Figure 3 Systolic Array Structure](image1)
![Figure 4 Structure of PE](image2)
![Figure 5 Timing & Exchange Sequence](image3)
Fig. 5 represents the exact direction of data exchange and times at which the exchange occurs. The sequential execution pattern can be observed by the visual representation from

![Figure 6](image6.png)

![Figure 7](image7.png)

![Figure 8](image8.png)

Figures 6, 7, 8, 9, 10, and 11 show the data exchange process at different times.
This interchange depends on the location index of the processor P\textsubscript{ij} in the array. It can be specified using the following algorithm:

\[
\begin{align*}
\text{if } i = 1 & \text{ then } [\text{out } \alpha \leftarrow \alpha \quad \text{out } \beta \leftarrow \beta] \\
\text{if } j = 1 & \text{ then } [\text{out } \alpha \leftarrow \beta \quad \text{out } \beta \leftarrow \alpha] \\
\text{else} & \quad [\text{out } \alpha \leftarrow \gamma \quad \text{out } \beta \leftarrow \delta]
\end{align*}
\]

Let \( \Delta_{ij} = |i-j| \) denote the distance of processor P\textsubscript{ij} from the diagonal. The operation of processor P\textsubscript{ij} is then delayed by \( \Delta_{ij} \) time units relative to the operation of diagonal processors. This is allow sufficient time for the rotation parameters to be propagated at unit speed along each row and column of the processor array. Also, a processor cannot commence a rotation until data from earlier rotations are available on all its input lines. Processor P\textsubscript{ij} needs data from its four neighbors \( P_{i \pm 1, j \pm 1} (1 < i, j < n/2) \). Dependencies for other processors can be seen from Fig 9 to Fig 14. The individual processors for the BLV is represented by Fig 15, 16 & 17.
b. Improved Efficiency Systolic Implementation

It is observed that every processing element remains idle for an average time period of 2 cycles per 3 cycles of execution time. The idle period for such architecture is approximately 66%. To improve the efficiency we can transmit the angle as soon as they are computed instead of waiting for the processing on the diagonal processors to be completed before commencing the transmission of the angles either along the rows or columns.

It can be represented as shown in Fig 18.

The algorithmic formulation of the implementation can be represented as follows

If \( i=j \) (diagonal processor)
- Solve 2 x 2 SVD/EVD
- Output Rotation Parameters
- Apply rotations to sub-matrix
- Output data (2 x 2 sub-matrix)
- Wait for new data

Else (off diagonal processor)
- Wait for new rotation parameters
- Output rotation parameters
- Apply rotations to sub-matrix
- Output data (2 x 2 sub-matrix)
- Wait for new data

The timing diagram for the array is represented in Fig 19.

VI. MODIFIED SYSTOLIC IMPLEMENTATION

The previous implementation of the systolic array had lot of swaps involved and also angles had to be transmitted along the row and column. The implementation of the Systolic array can be modified to achieve the same results in more iteration but fewer swaps. The matrix \( R(\theta) \) is implemented for calculating the eigen vectors of matrix A. The modified implementation has a new matrix arrangement containing four elements

\[
U_{ij} = \begin{bmatrix} u_{2i,1j-1} & u_{2i,1j} \\ u_{2i,2j-1} & u_{2i,2j} \end{bmatrix}
\]

(10)

where, \( i = 1,2, \ldots, \frac{n}{2} \), \( j = 1,2, \ldots, \frac{n}{2} \)

The array U is operated as,

\[
[U_{i,2j-1} \quad u_{i,2j}] = [U_{i,2j-1} \quad u_{i,2j}]R^T(\theta)
\]

(11)

where, \( i = 1,2, \ldots, \frac{n}{2} \), \( j = 1,2, \ldots, \frac{n}{2} \)

The modified array must exchange its elements for synchronization. If the sub-matrix \( p \) is equal to \( [\alpha \beta; \gamma \delta] \)
δ] elements exchange, consisting of inner and exterior exchanges as depicted in the Fig 19

![Data Exchange Direction](image)

**Figure 19**

**Data Exchange Direction**

The general block diagram of the implementation is represented in Fig

![Systolic Implementation](image)

**Figure 20**

**Systolic Implementation**

The angle θ is gotten from the processor by using Eq 9, which transmits as downward by column as shown single →arrow. Double arrow represents data exchange direction. The exchange direction for data and angle is quite different from that of systolic implementation.

**VII. IMPLEMENTATION & ANALYSIS**

For the purpose of analysis of the EVD techniques, the hardware implementations were simulated on a software platform. MATLAB was used for simulation. To allow for perfect analysis, hardware considerations like accuracy of bits that can be represented and delays of each PE and its associated combinational and sequential logic delay were coded numerically. The perfect values of delays were obtained by simulating a single PE using ModelSim for precise hardware analysis.

The first implementation consisted of the conventional Exact Jacobi. Though the method was known to give slower convergence, it is extremely robust and stable method for performing Eigen Value Decomposition. Exact Jacobi method required only three sweeps over the entire matrix for annihilating all the off diagonal elements or rather for the diagonal elements to converge to the Eigen value.

The Systolic Implementation required more number of sweeps across the matrix. But the systolic implementation of the EVD had significantly lesser number of additions and multiplications. Thus saving tremendous computation time consumed at the multipliers and adders. Hence overall, Systolic array gave a much faster run-time as compared to the Exact Jacobi Method.

The Modified systolic implementation, had larger number of overall sweeps across the matrix, but it had lesser number of swaps as compared to the original Systolic implementation.

The comparison between the Exact Jacobi and Systolic implementation for a matrix of size N=8 is given in Table 1.

<table>
<thead>
<tr>
<th>Comparison Criteria</th>
<th>Exact Jacobi</th>
<th>Systolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sweeps for Convergence across the Matrix</td>
<td>3</td>
<td>22–25</td>
</tr>
<tr>
<td>Additions</td>
<td>~3500</td>
<td>~1500</td>
</tr>
<tr>
<td>Multiplications</td>
<td>~7000</td>
<td>~3000</td>
</tr>
<tr>
<td>Swaps</td>
<td>0</td>
<td>368</td>
</tr>
<tr>
<td>Speed</td>
<td>Slower</td>
<td>Faster</td>
</tr>
</tbody>
</table>

**Table 1**

The timing analysis for the improved efficiency systolic implementation can be performed. Every step in the algorithm takes $T \approx T_{add} + \max(\tau_{rd}, \tau_{ro})$, and for the original Systolic implementation $T \approx T_{add} + \tau_{rd} + \tau_{ro}$, where $T_{rd}$ is the time for the rotations in a diagonal processor & $T_{ro}$ is the time for rotations application.
in an off–diagonal processor). Which means that the time of one step in both algorithms is comparable and can be equal (example if $T_{rd} = T_{ro}$). That is why the new algorithm is about three times more efficient than the precedent one. Another consequence of the new algorithm is the division of the computation time by three: if $S$ is the number of sweeps then the computation time is $CT = (3S(N-1)+(N/2-1)+3)T$ and for the new algorithm $CT_{new} = S(N-1) + T_{2x2prob} + N/2-1$. That's because in the original algorithm a sweep takes $3(N-1)$ time steps while in our algorithm it takes $(N-1)$ time steps.

Theoretical analysis coupled with simulation of certain modules of the Systolic Array give theoretical operational frequency of 300 MHz for the Systolic implementation.

**VIII. RESULTS**

The Exact Jacobi Method is very robust but a slow implementation of EVD. It is well suited for Applications where Chip area is of primary concern. With the growing capacity on FPGAs this criterion might be more useful for ASIC implementations or low power implementations of EVD.

The Systolic implementation and the improved efficiency are very well suited for implementation on FPGA. It is optimized for speed. The direct disadvantage being it consumes more power and also requires more area for its implementation.

The Modified Systolic Array is suited for a tough combination of speed as well as power. The reduced number of swaps greatly decreases power requirements of the circuits. A rough analysis gives the ratio of consumed in the swaps to be almost half the power consumed in swaps for Improved Efficiency Systolic Implementation.

**IX. CONCLUSION**

The project addresses the need for hardware efficient architectures for Eigen Value Decomposition. Three algorithms and corresponding architectures are explored for three critical hardware constraints namely speed, area & power. Analysis of the three design allows estimation of the above three constraints.

Systolic array architectures are well suited to meet the growing demand for real time signal Processing Algorithms. The Jacobi Rotations form the base for EVD algorithms. CORDIC is used to generate the necessary trigonometric values.

**X. FUTURE WORK**

An interesting proposal that could be explored is the accumulation of the left and right singular vectors for the matrix during the computation of the Eigen Values. Also a provision to handle oversized and undersized matrices can be explored.

**XI. ACKNOWLEDGEMENT**

I would like to thank Prof Y.H.Hu for guiding me during various stages of the project. The initial stages of the project were critical as a good problem formulation was essential. I received tremendous support and advice from Prof. Hu.

**References**

- Hemkumar N, Master’s Thesis, Rice University
- Advanced Algorithmic Evaluation for Imaging, Communication and Audio Applications – Eigenvalue Decomposition using CATAPULT C Algorithmic Synthesis Methodology
- Efficient Implementation of SVD on a Reconfigurable System, Christophe Bobda, Klaus Danne and Andre Linarth, Springer-Verlag Berlin Heidelberg 2003
• Spectral Estimation using MUSIC Algorithm, Jawed Qumar, Nios II Embedded Processor Design Contest-2005

• Luo Feng, He Kun, Wu Shunjun, Research of Acquiring Eigenvector of Real Symmetric Matrix, IEEE – 2006

• Efficient Systolic Array for Singular Value and Eigen Value Decomposition, A Ahmedsaid, A. Amira, A Bouridane, IEEE 2004


• A Novel Fast Eigenvalue Decomposition based on Cyclic Jacobi Rotation and its application in eigen-beamforming, Tech Report of IEICE-Japan

• Efficient Hardware Architectures for Eigenvector and Signal Subspace Estimation, Fan Xu & Alan Wilson, IEEE Transactions on Circuits & Systems-204


• Survey of CORDIC Algorithms for FPGA Based computers, Ray Andraka, ACM-1998

• Smart Antennas for Wireless Communications, Frank B Gross, McGraw Hill, 2005 (Used for Facts & References for Comparison purposes and Specifications of Different wireless standards)

• MATLAB R2008a
APPENDIX

1. Code for S/W simulation of Systolic Array

```matlab
% S/W Implementation of Systolic Array for EVD
% The code is used to simulate a N x N systolic array for performing Eigen
% Value decomposition
% The program outputs the initial input matrix, the final output matrix
% and also the intermediate steps and error values
% The program also computes the total number of additions and
% multiplications required for the computation
% Written by Nikhil Suryanarayanan

clear all;
clc;

iter=30;
n=8;
A0=10*rand(n);
%A0=floor(A0);
A=A0+A0';
A0=A;
disp('ORIGINAL INPUT MATRIX');
disp(' ');
disp(A0);
T=zeros(n);
add=0;
mul=0;
tic;
for k=1:iter
    for i=1:2:n
        [J,a(i)]=getmat(A(i,i),A(i+1,i+1),A(i+1,i));
        A([i,i+1],[i,i+1])=J'*A([i,i+1],[i,i+1])*J;
        add=add+4;
        mul=mul+2;
    end
    for i=1:2:n-2
        J=formmat(a(i));
        for j=i+2:n-2
            K=formmat(a(j+2));
            A([i,i+1],[j+2,j+3])=J'*A([i,i+1],[j+2,j+3])*K;
            add=add+4;
            mul=mul+8;
        end
        K=formmat(a(i));
        for j=i+2:n-2
            J=formmat(a(j+2));
            A([j+2,j+3],[i,i+1])=J'*A([j+2,j+3],[i,i+1])*K;
        end
    end
    disp(' ');
end
toc;
```

The code above is designed to simulate a systolic array for performing Eigenvalue decomposition. It takes an initial input matrix and performs the necessary operations to simulate the systolic array's behavior, outputting the final output matrix along with intermediate steps and error values. The total number of additions and multiplications required for the computation is also computed and printed.
add = add + 4;
mul = mul + 8;
end
end

% DATA EXCHANGE

for i = 1:2:n
    for j = 1:2:n

        % The first PE element
        if ((i == 1) && (j == 1))
            T([i, i+1], [j, j+1]) = [ A(i, j)  A(i, j+3) ;
                                     A(i+3, j)  A(i+3, j+3) ];
        end

        % PEs along the first row
        elseif (i == 1)
            % PE located at the end of matrix
            if (j == n-1)
                % Special case when n = 4
                T([i, i+1], [j, j+1]) = [ A(i, j-1)  A(i, j) ;
                                         A(i+3, j-1)  A(i+3, j) ];
                else
                    T([i, i+1], [j, j+1]) = [ A(i, j-2)  A(i, j) ;
                                             A(i+3, j-2)  A(i+3, j) ];
            end

            elseif (j == 3)
                % PE located at the (1, 3) again a special case
                T([i, i+1], [j, j+1]) = [ A(i, j-1)  A(i, j+3) ;
                                         A(i+3, j-1)  A(i+3, j+3) ];
            else
                % Any other PE in the first row
                T([i, i+1], [j, j+1]) = [ A(i, j-2)  A(i, j+3) ;
                                         A(i+3, j-2)  A(i+3, j+3) ];
        end

        % PEs along the first column
        elseif (j == 1)
            % PE located at the end of matrix
            if (i == n-1)
                % Special case when n = 4
                T([i, i+1], [j, j+1]) = [ A(i-1, j)  A(i-1, j+3) ;
                                         A(i, j)  A(i, j+3) ];
                else
                    T([i, i+1], [j, j+1]) = [ A(i-2, j)  A(i-2, j+3) ;
                                             A(i, j)  A(i, j+3) ];
        end
elseif (i==3)
    T([i,i+1],[j,j+1])=[ A(i-1,j) A(i-1,j+3) ]
    %PE located at the (3,1) again special case
    A(i+3,j) A(i+3,j+3) ];
else
    T([i,i+1],[j,j+1])=[ A(i-2,j) A(i-2,j+3) ];
    %Any other PE in the first column
    A(i+3,j) A(i+3,j+3) ];
end

elseif (i==3)
    if (j==3)
        if (n==4)
            T([i,i+1],[j,j+1])=[ A(i-1,j-1) A(i-1,j) ]
            A(i,j-1) A(i,j) ];
        else
            T([i,i+1],[j,j+1])=[ A(i-1,j-1) A(i-1,j+3) ];
            A(i+3,j-1) A(i+3,j+3) ];
        end
    elseif (j==n-1)
        T([i,i+1],[j,j+1])=[ A(i-1,j-2) A(i-1,j) ];
        A(i+3,j-2) A(i+3,j) ];
    else
        T([i,i+1],[j,j+1])=[ A(i-1,j-2) A(i-1,j+3) ];
        A(i+3,j-2) A(i+3,j+3) ];
    end
elseif (j==n-1)
    if (i==n-1)
        T([i,i+1],[j,j+1])=[ A(i-2,j-1) A(i-2,j+3) ];
        A(i,j-1) A(i,j+3) ];
    else
        T([i,i+1],[j,j+1])=[ A(i-2,j-1) A(i-2,j+3) ];
        A(i+3,j-1) A(i+3,j+3) ];
    end
elseif (i==n-1 || j==n-1)
    if (i==n-1 && j==n-1)
        T([i,i+1],[j,j+1])=[ A(i-2,j-2) A(i-2,j) ];
        A(i,j-2) A(i,j) ];
    elseif (i==n-1)
        T([i,i+1],[j,j+1])=[ A(i-2,j-2) A(i-2,j+3) ];
        A(i,j-2) A(i,j+3) ];
    else
        T([i,i+1],[j,j+1])=[ A(i-2,j-2) A(i-2,j) ];
        A(i+3,j-2) A(i+3,j) ];
    end
else

end
\[
T([i, i+1], [j, j+1]) = \begin{bmatrix}
A(i-2, j-2) & A(i-2, j+3) \\
A(i+3, j-2) & A(i+3, j+3)
\end{bmatrix};
\]

\begin{verbatim}
end
end
end
A=T;
if (mod(k,1)==0)
    Fr_norm=norm(sort(diag(A))-eig(A0));
    disp(Fr_norm);
    if (Fr_norm<0.001)
        disp('Iteration Converged at :');
        disp(k);
        disp('NORM :');
        break;
    end
end
end
toc;
disp(' ');
disp(' ');
disp('MATRIX AFTER EIGEN DECOMPOSITION');
disp(A);
disp('THE DIAGONAL ELEMENTS OF MATRIX & ACTUAL EIGEN VALUES');
disp([sort(diag(A)),eig(A0)]);
disp('ERROR');
disp(sort(diag(A))-eig(A0));
disp('ADDITIONS');
disp(add);
disp('MULTIPLICATIONS');
disp(mul);
ORIGINAL INPUT MATRIX
18.1336
8.3901    6.8183    4.7473   12.5321    9.5317    0.3081    3.3934    2.5012
8.2284    6.9676    4.1814    7.3682   17.3636    2.5012   11.1834   15.5143
51.6085
51.4244
40.7173
40.6771
27.6427
7.8321
\end{verbatim}
7.3995
6.1485
6.5147
6.4765
6.4763
2.5229
2.5417
1.2743
1.2987
1.1798
0.2215
0.0909
0.0322
0.0317
0.0052
0.0041
0.0018
2.7663e-005

Iteration Converged at:
  24

NORM:
Elapsed time is 0.051621 seconds.

<table>
<thead>
<tr>
<th>Matrix After Eigen Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-12.9996 -0.0003 -0.0000 0.0019 0.0000 -0.0067 -0.0001 0.0000</td>
</tr>
<tr>
<td>-0.0003 74.5039 -0.0000 -0.0000 -0.0000 -0.0001 0.0000 0.0000</td>
</tr>
<tr>
<td>0 -0.0000 -16.2938 0.0002 -0.0001 0.0000 0.0000 0.0015</td>
</tr>
<tr>
<td>0.0019 -0.0000 0.0002 7.3373 -0.0003 -0.0001 0 0.0057</td>
</tr>
<tr>
<td>0.0000 0.0000 -0.0001 -0.0003 16.4035 -0.0193 0.0049 0.0002</td>
</tr>
<tr>
<td>-0.0067 -0.0001 0.0000 -0.0001 -0.0193 -2.0844 -0.0007 -0.0000</td>
</tr>
<tr>
<td>-0.0001 0.0000 0.0000 0.0000 0.0049 -0.0007 1.7618 -0.0012</td>
</tr>
<tr>
<td>0.0000 0.0000 0.0015 -0.0057 0.0002 -0.0000 -0.0012 -4.6086</td>
</tr>
</tbody>
</table>

The Diagonal Elements of Matrix & Actual Eigen Values

| -16.2938 -16.2938 |
| -12.9996 -12.9996 |
| -4.6086 -4.6086 |
| -2.0844 -2.0844 |
| 1.7618 1.7618 |
| 7.3373 7.3373 |
| 16.4035 16.4035 |
| 74.5039 74.5039 |
ERROR
1.0e-004 *

0.0020
0.0430
0.0272
0.1605
0.0131
-0.0287
-0.2172
-0.0000

ADDITIONS
1536

MULTIPLICATIONS
2496

Published with MATLAB® 7.6
% Program to compute the EIGEN VALUES of a Matrix
% The program uses Jacobi's rotation
% Written by Nikhil Suryanarayanan

clear all;
clc;

format short;

% Matrix Size
n=8;
A0=randn(n,n);
iterations=10;

% Create Symmetric Matrix
A0=A0+A0';
add=0;
mul=0;

% Applying Jacobi's rotation
A=A0;
tic;
for k=1:iterations
    for j=1:n-1
        for i=n:-1:j+1
            J=jacrot(A(i,i),A(j,j),A(i,j));
            A([i,j],:)=J'*A([i,j],:);
            A(:,[i,j])=A(:,[i,j])*J;
            add=add+(n)*2*2;
            mul=2*2*n*2+mul;
        end
    end
    if (mod(k,1)==0)
        Fr_norm=norm(sort(diag(A))-eig(A0));
        disp(Fr_norm);
        if(Fr_norm<0.001)
            disp('Iteration Converged at :');
            disp(k);
            disp('NORM :');
            break;
        end
    end
end
end
toc;

% Compare with true eigenvalues
% Diagonal  Eigen Values  Error
disp('ADDITIONS');
disp(sort(diag(A)),eig(A0),sort(diag(A))-eig(A0));
disp('MULTIPLICATIONS');
disp(mul);
1.5005

0.2517
Iteration Converged at: 4

NORM:
Elapsed time is 0.020339 seconds.

<table>
<thead>
<tr>
<th>Diagonal</th>
<th>Eigen Values</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6927</td>
<td>-7.6927</td>
<td>0.0000</td>
</tr>
<tr>
<td>-2.2481</td>
<td>-2.2481</td>
<td>0.0000</td>
</tr>
<tr>
<td>-1.8364</td>
<td>-1.8364</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.3643</td>
<td>0.3643</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.0449</td>
<td>1.0449</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.3777</td>
<td>1.3777</td>
<td>0.0000</td>
</tr>
<tr>
<td>3.9051</td>
<td>3.9051</td>
<td>0.0000</td>
</tr>
<tr>
<td>5.6233</td>
<td>5.6233</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

ADDITIONS
3584

MULTIPLICATIONS
7168

Published with MATLAB® 7.6
%TO COMPUTE THE ANGLE FROM THE GIVEN VECTORS

clear all;
clc;
format short;
n=100;
w=0:(n-1);
W=0.5.^w;

%W=[45 26.565 14.036 7.125 1.786 0.8939 0.4469 0.4469/2 .4469/4 .4469/8];
x=0;
y=10;
%theta=90;
angle=0;
d=-1;

for i=1:n
    x1=x-y*d*(2^-(i-1));
    y1=y+x*d*(2^-(i-1));
    angle=angle-d*atan(W(i))*180/pi;
    if (y1==0)
        d=0;
    elseif (y1<0)
        d=1;
    else
        d=-1;
    end
    x=x1;
    y=y1;
end

x1=x1;
y1=y1;
disp(angle);
end
clear all;
clc;
format short;
n=10;
w=0:(n-1);
W=0.5.^w;

%W=[45 26.565 14.036 7.125 3.576 1.786 0.8939 0.4469 0.4469/2 .4469/4 .4469/8];
x=1/1.647;
y=0;
%theta=90;
angle=72;
d=1;
if (angle==45)
x=1/1.414;
end
for i=1:n
    x1=x-y*d*(2^-(i-1));
    y1=y+x*d*(2^-(i-1));
    angle=angle-d*atan(W(i))*180/pi;
    if(angle==0)
        d=0;
    elseif(angle>0)
        d=1;
    else
        d=-1;
    end
    x=x1;
    y=y1;
    disp(angle);
end
x1=x1;
y1=y1;
disp(angle);
disp('cos');
disp(x1);
disp('sin');
disp(y1);
AUXILIARY FUNCTIONS

function \( M = \text{formmat}(\text{angle}) \)
\[ M = [\cos(\text{angle}) \ \sin(\text{angle}); -\sin(\text{angle}) \ \cos(\text{angle})] \];

function \([M, \text{theta}] = \text{getmat}(a, b, c)\)
\[
\text{theta} = 0.5 \cdot \arctan(2c/(b-a));
M = [\cos(\text{theta}) \ \sin(\text{theta}); -\sin(\text{theta}) \ \cos(\text{theta})];
\]

function \( J = \text{jacrot}(a, b, \text{din}) \)
\[
\text{theta} = 0.5 \cdot \arctan(2 \cdot \text{din}/(b-a));
J = [\cos(\text{theta}) \ \sin(\text{theta}); -\sin(\text{theta}) \ \cos(\text{theta})];
\]