Hardware Implementation of Direction of Arrival (DoA) estimation of Node Localization algorithm for Smart Antenna in Wireless Sensor Networks

Project Report

Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Engineering

in

Electronics and Tele-Communication Engineering

Faculty of Engineering, Jadavpur University

Submitted By

Hrit Mukherjee

Roll - 001710701035

and

Aveek Bhaumik

Roll – 001710701005

Declaration

We, **Aveek Bhaumik** and **Hrit Mukherjee**, hereby declare that this report on **"Hardware Implementation of Direction of Arrival (DoA) estimation of Node Localization algorithm for Smart Antenna in Wireless Sensor Networks"** submitted by us to **Jadavpur University**, **Kolkata** as a part of our final year project required for fulfilling the curriculum requirement of **Bachelor of Engineering in Electronics and Tele-Communication,** is a record of bonafide work carried out by us under the guidance of **Dr. Mrinal Kanti Naskar, Professor, Department of Electronics and Tele-Communication, Jadavpur University.**

Aveek Bhaumik

Hrit Mukherjee

Date:

Certificate

This is to certify that the report titled **"Hardware Implementation of Direction of Arrival (DoA) estimation of Node Localization algorithm for Smart Antenna in Wireless Sensor Networks"** submitted by **Aveek Bhaumik** and **Hrit Mukherjee,** students of final year of Jadavpur University, Kolkata, to fulfil the curriculum requirement of **Bachelor of Engineering in Electronics and Tele-Communication,** is a record of bonafide work carried out by them under my supervision as a part of their final year project.

Dr. Mrinal Kanti Naskar

Professor

Department of Electronics and Tele-Communication

Jadavpur University

Date:

Acknowledgement

We wish to take this opportunity to express our deepest appreciation and sincere gratitude to our project mentor **Dr. Mrinal Kanti Naskar** for his scintillating support, spontaneous guidance and continuous encouragement during the course of our project. He worked on strengthening our Digital circuit background and provided a lot of support and valuable inputs to the problems that we faced while designing the hardware. We would like to convey special thanks to **Rathindra Nath Biswas** sir and **Swarup Mitra** sir for guiding us and creating a joyful and happy atmosphere to work in.

Abstract

Direction of Arrival(DoA) estimation of radio signals is of utmost importance in many commercial and military applications. This project is aimed to achieve the hardware implementation of the ESPRIT Algorithm for DoA estimation. For the purposes of implementing the ESPRIT algorithm it is necessary to perform eigenvalue decomposition and inversion of a matrix. The conventional method of evaluating them requires the use of determinant of a matrix, which is difficult and time consuming to implement in a hardware description language. As a result certain iterative algorithms are utilized to evaluate them in a more efficient manner. These algorithms include Gauss-Jordan elimination (for Matrix Inversion), Jacobi Eigen Value Algorithm (for eigenvalue and eigenvector calculation of symmetric matrices) and the Eigensystem Computation Algorithms (for eigenvalue and eigenvector calculation of skew symmetric matrices). An approach has been proposed for evaluating eigenpairs of any general matrix as well. The simulations have been performed in Verilog, MATLAB and FORTRAN. The results and code snippets have also been provided in detail throughout the report for better understanding of the reviewer. In our future works we intend to implement a hardware to embed the entire DoA estimator in a single FPGA chip.

Contents

Introduction

Smart Antennas, also known as adaptive array antennas, are digital antenna arrays with smart signal processing algorithms used to identify spatial signal signatures such as the direction of arrival (DOA) of the signal, and use them to calculate beam forming vectors which are used to track and locate the antenna beam on the mobile/target. Smart antennas should not be confused with reconfigurable antennas, which have similar capabilities but are single element antennas and not antenna arrays. Smart antenna techniques are used notably in acoustic signal processing, track and scan radar, radio astronomy and radio telescopes, and mostly in cellular systems like W-CDMA, UMTS, and LTE. Smart antennas have many functions: DOA estimation, beamforming, interference nulling, and constant modulus preservation. Recently, smart arrays have been adopted in multiple mobile communication standards for multipath fading reduction, bit error rate (BER) boosting, co-channel interference reduction (CCI), and spectral efficiency improvement.

ESPRIT Algorithm

2.1 Block Diagram of a typical MUBTS (Multiple User Beam Tracker System)

The following figure (Fig 1) shows a typical smart antenna scheme, implemented at the receiver side (base station). MUBTS has an array antenna consisting of two rows of 8 elements and a multiple user software defined radio, supporting communication with up to four concurrent users.

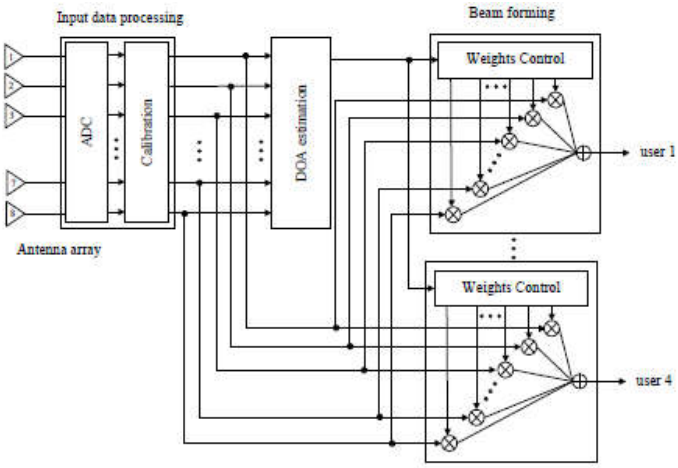


Fig 1: A 4 user beam tracking system implemented on 8 elements Uniform Linear Antenna Array

2.2 DoA Estimation using ESPRIT Algorithm

The main objective of our work is to correctly and accurately estimate the Direction of Arrival (DoA) of the signal in Smart Antenna using ESPRIT Algorithm. This method involves finding a spatial spectrum of the antenna/sensor array and calculating the DoA from the peaks of the spectrum. The goal of the ESPRIT technique is to exploit the rotational invariance in the signal subspace which is created by two arrays with a translational invariance structure.

The steps involved in the ESPRIT (Estimation of Signal Parameters via Rotational Invariance Techniques) algorithm is given below:

- Formation of Covariance Matrix from noisy data.
- Computing SVD (Singular Value Decomposition) of the Covariance Matrix.
- Obtaining the orthonormal eigenvectors corresponding to the sources.
- Splitting the matrix formed by the orthonormal eigenvectors into two subspaces.
- Computation of Rotation Operator by solving a linear system of equations.
- Obtaining the eigenvalues of the Rotation Operator.
- Finding the Direction of Arrival from the argument of eigenvalues calculated and other given parameters.

Matrix Inversion

As mentioned in the fifth step of ESPRIT algorithm, we need to compute rotation operator by solving a linear system of equations. For example, let us consider a linear system of non-homogeneous equations:

$$\mathbf{AX = B}$$

where, A = co-efficient matrix

X = input matrix or variable matrix

B = constant matrix

Therefore,

$$\mathbf{X = A^{-1}B}$$

In this case, the variable matrix, X is the rotation operator. Thus, to calculate X, we need to evaluate the inverse of the co-efficient matrix, A. Since, we wish to compute Matrix Inversion using a Hardware Description Language i.e., Verilog, it would be efficient to follow a Numerical Analysis Technique. For this project, we have chosen the **Gauss-Jordan Elimination** method for Matrix Inversion as **it is easier to implement and the whole computation is completed within one clock cycle.**

3.1 Matrix Inversion by Gauss-Jordan Elimination

Let us consider a matrix, A with order n. To obtain A^{-1}:

- Create the partioned matrix B = [A | I_n], where I_n is the identity matrix of order n.
- Perform normalisation (dividing the entire row by its leading element so as to make the first entry 1) and a set of elementary row transformations on each row of B to upper triangularise A followed by a similar set of elementary row transformations on each row of B to lower triangularise A. These steps described above are basically the Gauss-Jordan Elimination.
- If done correctly, the resulting partitioned matrix, B takes the form [I_n | A^{-1}].
- Extract A^{-1} from the second partition of B.

Let $A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$

Therefore, $B = \left[\begin{array}{ccc|ccc} a_{00} & a_{01} & a_{02} & 1 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 1 & 0 \\ a_{20} & a_{21} & a_{22} & 0 & 0 & 1 \end{array}\right]$

Now,

i. Normalise R_0 to make $a_{00} = 1$.
ii. Manipulate R_1 with the help of R0 to make $a_{10} = 0$.
iii. Normalise R_1 to make $a_{11} = 1$.
iv. Manipulate R_2 with the help of R_0 to make $a_{20} = 0$.
v. Manipulate R_2 with the help of R_1 to make $a_{21} = 0$.
vi. Normalise R_2 to make $a_{22} = 1$.
vii. Manipulate R_1 with the help of R_2 to make $a_{12} = 0$.
viii. Manipulate R_0 with the help of R_1 to make $a_{01} = 0$.
ix. Manipulate R_0 with the help of R_2 to make $a_{02} = 0$.

Eventually, we will get $B = [I_n \mid A^{-1}]$. From B, we can extract A^{-1}.

3.1.1 Implementation in Verilog

- **Input:**

$$A = \begin{bmatrix} 5 & 7 & 9 \\ 4 & 3 & 8 \\ 7 & 5 & 6 \end{bmatrix}$$

- **Code Snippets:**

```
//upper triangularisation
        for(k=0; k<order; k++)
        begin
                if(mat[k][k]!=0)
                begin
                        temp = mat[k][k];
                        for(col=0; col<(2*order); col++)
                        begin
                                mat[k][col] = mat[k][col]/temp;;
                        end
                        for(row=k+1; row<order; row++)
                        begin
                                temp = mat[row][k];
                                for(col=0; col<(2*order); col++)
                                begin
                                        mat[row][col] = mat[row][col] - temp*mat[k][col];
                                end
                        end
                end
        end
```

```
//lower triangularisation
        for(k=order-1; k>=0; k--)
        begin
                for(row=k-1; row>=0; row--)
                begin
                        temp = mat[row][k];
                        for(col=0; col<(2*order); col++)
                        begin
                                mat[row][col] = mat[row][col] - temp*mat[k][col];
                        end
                end
        end
```

- **Simulation Output:**

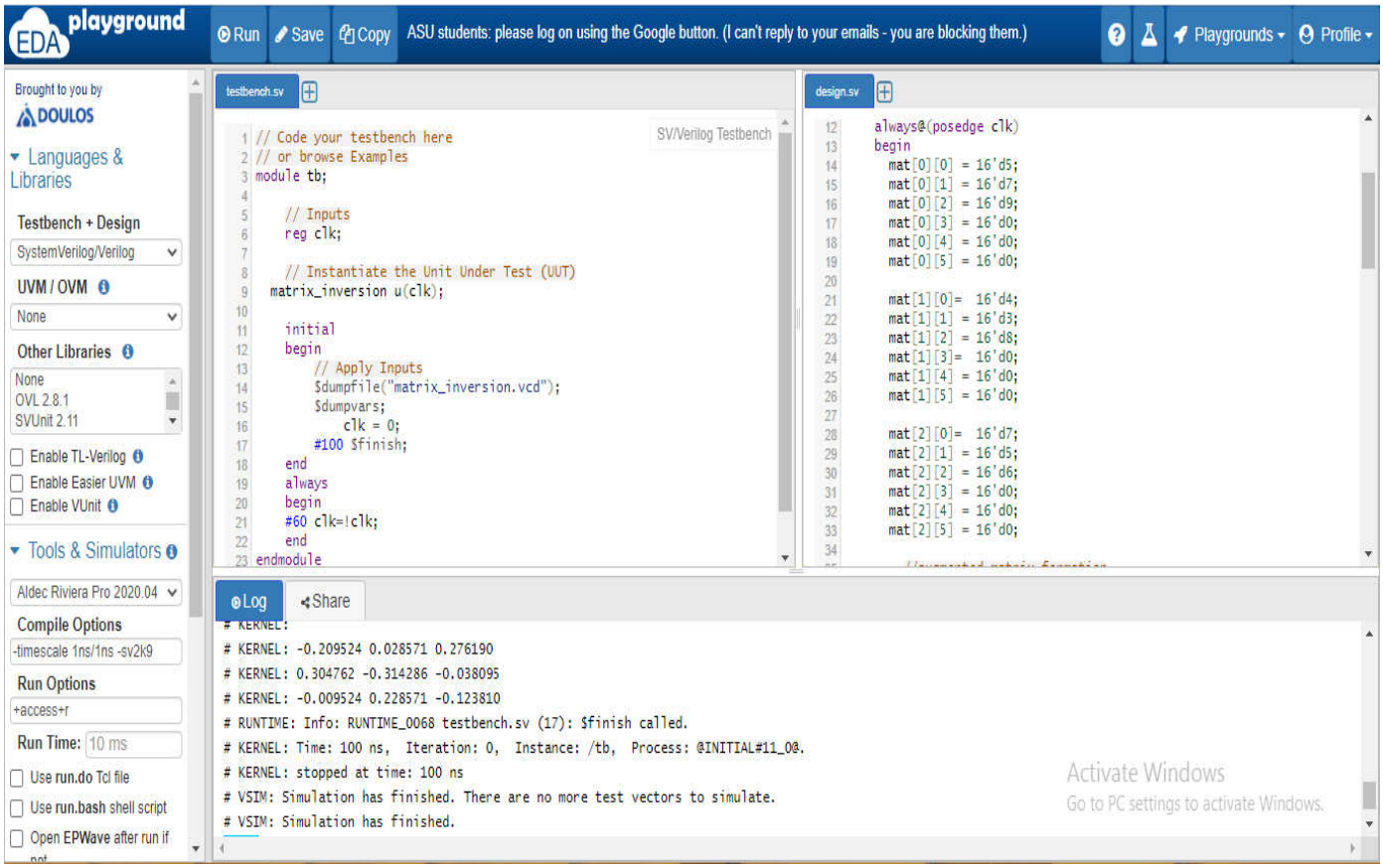


Fig 2: Simulation output for computing inverse of a matrix

Thus, we get $A^{-1} = \begin{bmatrix} -0.209524 & 0.028571 & 0.276190 \\ 0.304762 & -0.314286 & -0.038095 \\ -0.009524 & 0.228571 & -0.123810 \end{bmatrix}$

Eigen Value Decomposition

As mentioned in the third and sixth step of ESPRIT algorithm, we need to compute orthonormal eigenvectors of the corresponding sources and eigenvalues of the rotation operator respectively. So we need to perform Eigen Value Decomposition (EVD) on a given matrix. For this project, we have chosen the **Jacobi Method** method of EVD as **it involves Givens' Rotations which can be parallelized and has a lower operation count** with respect to other methods. Thus, the Jacobi Method is **fast, cost-effective, saves power and is suitable for VLSI implementation.**

4.1 Jacobi Method of Eigen Value Decomposition

Jacobi Eigen Value algorithm is an iterative method for calculating the eigenvalues and corresponding eigenvectors of a real symmetric matrix. In this method we will apply similarity transformations on the given matrix such that after a sequence of similarity transformations the matrix gets converted into a diagonal matrix. Hence, from the diagonal matrix, we can see the eigenvalues directly as the diagonal elements. Furthermore the sequence will contain the information about the eigenvectors of the matrix.

As the similarity matrices have same eigenvalues, similarity transformations of the Jacobi Method are basically rotations performed on the given matrix in each step so as to discard the off-diagonal elements in such a way that eigenvalues are preserved in the resultant diagonal matrix.

The steps involved in the Jacobi Eigen Value algorithm is given below:

- Find the pth row and q^{th} column which correspond to the off diagonal element of the input matrix (A) having highest absolute value.
- Compute the Jacobi matrix, J(p,q,θ) after calculating the angle of Givens' rotation.
- Multiply the input matrix with the Jacobi matrix and its transpose in the following manner:

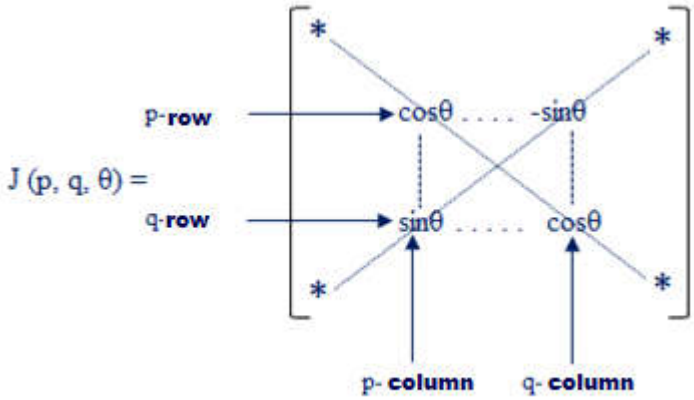
$$A_1 = J_1^T(p,q,\theta)*A*J_1(p,q,\theta)$$

- Repeat the above steps to compute (J_2, A_2), (J_3, A_3) and so on till all the non-diagonal entries become 0 and the matrix gets converted completely into a diagonal matrix. The diagonal elements will be the eigenvalues.
- The eigenvectors will be the columns of the final Jacobi matrix, J obtained by multiplying all the intermediate Jacobi Matrices.

$$J = J_1*J_2*J_3...$$

To rotate a 2X2 matrix by an angle θ in a 2 dimensional plane, we mutiply the input matrix by $\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$. This is the basis of Givens' Rotation. Here, we extend this idea to a n-dimensional plane, where n is order of the input matrix.

A n by n rotation matrix is defined as J(p,q,θ) with the form:



The pth row and qth column correspond to the largest off-diagonal element of the input matrix, A. Therefore, we get the entries a_{pq}, a_{qp}, a_{pp}, a_{qq}. Since, the matrix is a real-symmetric matrix, $a_{pq} = a_{qp}$. The angle, θ which is necessary to construct the Jacobi Matrix, J(p,q,θ) can be calculated from the following formula:

$$\theta = \frac{1}{2}\arctan\left(\frac{2a_{pq}}{a_{qq}-a_{pp}}\right)$$

For realising arctan and sin, cos using Hardware Description Language, we have utilised the CORDIC Algorithm.

4.2 CORDIC (Co-ordinate Rotation Digital Computer)

The CORDIC algorithm can compute several elementary mathematical operations such as trigonometric/hyperbolic functions, real/complex multiplications and divisions etc. efficiently. It is further modified to a generalized/unified form for different co-ordinate systems, simply introducing a new parameter (μ).

μ = 1, 0, -1 for circular, linear and hyperbolic co-ordinate systems respectively.

The key concept is laid on rotating a vector in 2-D co-ordinates for a desired angle (β) as depicted in Figure 3. The angle (β) is assumed to be a sum of several pre-defined elementary angles (β_i) that are obtained decomposing it by pseudo-micro-rotations in the following manner: $\beta = \sum_0^{p-1} \sigma_i \beta_i$, where $\beta_i = \tan^{-1}(2^{-i})$ and σ_i gives the direction of micro-rotations.

On setting values to such constituent angles as the power of 2, this iterative method turns into simple binary arithmetic with shift and add operations only. As the same hardware can be used for all applications with slight modifications at the initial conditions, it should be an ideal solution towards the flexible FPGA based design implementation.

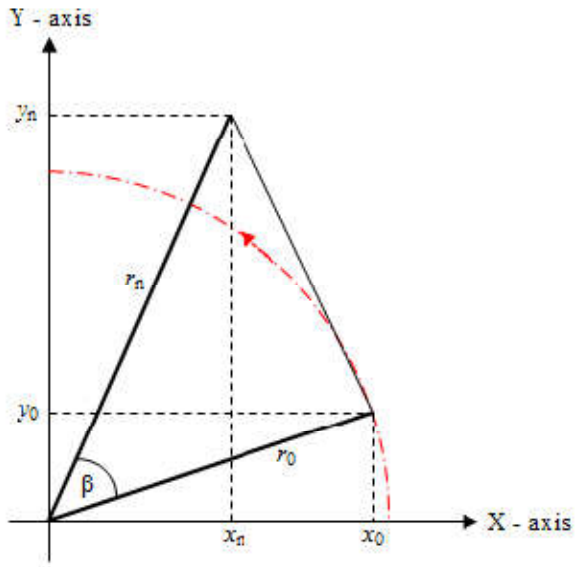


Fig 3: Rotating Vector in a 2D plane

COMPUTATIONS USING CORDIC ALGORITHM IN DIFFERENT CONFIGURATIONS

operation	configuration	initialization	output	post-processing and remarks
$\cos\theta, \sin\theta, \tan\theta$	CC-RM	$x_0 = 1$ $y_0 = 0$ and $\omega_0 = \theta$	$x_n = \cos\theta$ $y_n = \sin\theta$	$\tan\theta = (\sin\theta/\cos\theta)$
$\cosh\theta, \sinh\theta$ $\tanh\theta, \exp(\theta)$	HC-RM	$x_0 = 1$ $y_0 = 0$ and $\omega_0 = \theta$	$x_n = \cosh\theta$ $y_n = \sinh\theta$	$\tanh\theta = (\cosh\theta/\sinh\theta)$ $\exp(\theta) = (\cosh\theta + \sinh\theta)$
$\ln(a), \sqrt{a}$	HC-VM	$x_0 = a + 1$ $y_0 = a - 1$ and $\omega_0 = 0$	$x_n = \sqrt{a}$ $\omega_n = \frac{1}{2}\ln(a)$	$\ln(a) = 2\omega_n$
$\arctan(a)$	CC-VM	$x_0 = a$ $y_0 = 1$ and $\omega_0 = 0$	$\omega_n = \arctan(a)$	no pre- or post-processing
division (b/a)	LC-VM	$x_0 = a$ $y_0 = b$ and $\omega_0 = 0$	$\omega_n = b/a$	no pre- or post-processing
polar-to-rectangular	CC-RM	$x_0 = R$ $y_0 = 0$ and $\omega_0 = \theta$	$x_n = R\cos\theta$ $y_n = R\sin\theta$	no pre- or post-processing
rectangular-to-polar $\tan^{-1}(b/a)$ and $\sqrt{a^2+b^2}$	CC-VM	$x_0 = a$ $y_0 = b$ and $\omega_0 = 0$	$x_n = \sqrt{a^2+b^2}$ $\omega_n = \arctan(b/a)$	no pre- or post-processing

In our project, we have used Circular Co-ordinate System Rotating Mode (CC-RM) for computing cos and sin of a given angle, θ and Circular Co-ordinate System Vectoring Mode (CC-VM) for computing arctan of a given value, α.

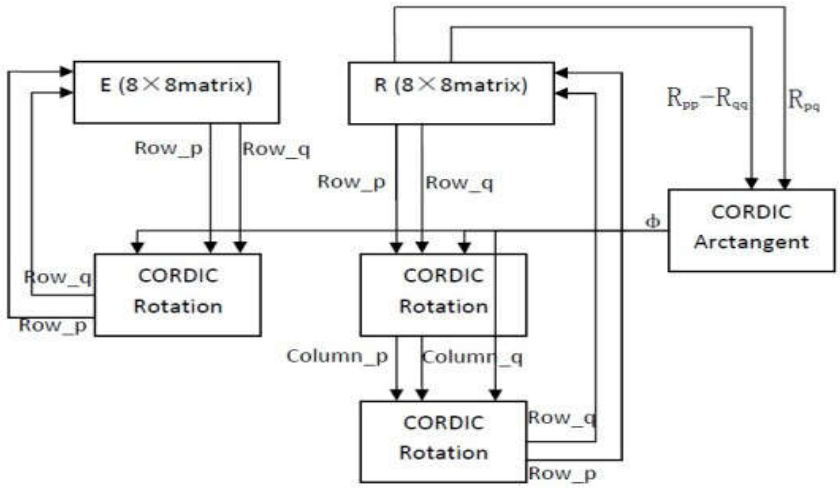


Fig 4: Architecture of CORDIC based Jacobi Algorithm

Algorithm: Pseudo-code for realizing Sine/Cosine function in CORDIC
Input: X - coordinate of rotation vector (x_1), Y - coordinate of rotation vector (y_1), Angle of rotation (z_1), Dynamic ranges of the rotation angle: {-90°, 90°} and Maximum permissible iterations (i_{max}).
Output: sine function {sin(α)}: y_{i+1} ; cosine function {cos(α)}: x_{i+1}
1 Initialize: $x_1 = 0.61$, $y_1 = 0$, $z_1 = \alpha$, i_{max}
2 for all iterations i do
3 if ($z_i \geq 0$) then
4 $\sigma_i \leftarrow 1$
5 else
6 $\sigma_i \leftarrow -1$
7 end if
8 $x_{i+1} \leftarrow x_i + \sigma_i * 2^{-i} * y_i$
9 $y_{i+1} \leftarrow y_i - \sigma_i * 2^{-i} * x_i$
10 $z_{i+1} \leftarrow z_i - \sigma_i * \tan^{-1}(2^{-i})$
11 end for
12 return y_{i+1}; x_{i+1}

4.2.1 Implementation in Verilog

- **Fixed Point Implementation:**

Here, we have followed a 17 bit fixed point format for representing all the values. The MSB i.e., bit 17 is the sign bit, bit 16 is the whole number part and the rest 15 bits represent the fractional parts. The angles are represented in radian.

- **Input:**

$x_1 = 0.603$, $y_i = 0$, $z_i = 60°$ in radians, specified in the above mentioned format.

- **Code Snippets:**

```
always @ (posedge clk)

  if (rst) begin
    x_1 <= 0;
    y_1 <= 0;
    z_1 <= 0;
  end else begin

    if (init) begin
      x_1 <= x_i;
      y_1 <= y_i;
      z_1 <= z_i;
    end else if (
    z_i[16]
    ) begin
      x_1 <= x_i + {y_i[16],y_i_shifted}; //shifter(y_1,i); //(y_1 >> i);
      y_1 <= y_i - {x_i[16],x_i_shifted}; //shifter(x_1,i); //(x_1 >> i);
      z_1 <= z_i + tangle;
```

```
    end else begin
      x_1 <= x_i - {y_i[16],y_i_shifted}; //shifter(y_1,i); //(y_1 >> i);
      y_1 <= y_i + {x_i[16],x_i_shifted}; //shifter(x_1,i); //(x_1 >> i);
      z_1 <= z_i - tangle;

    end
  end
assign x_o = x_1;
assign y_o = y_1;
assign z_o = z_1;
```

- **Simulation Output:**

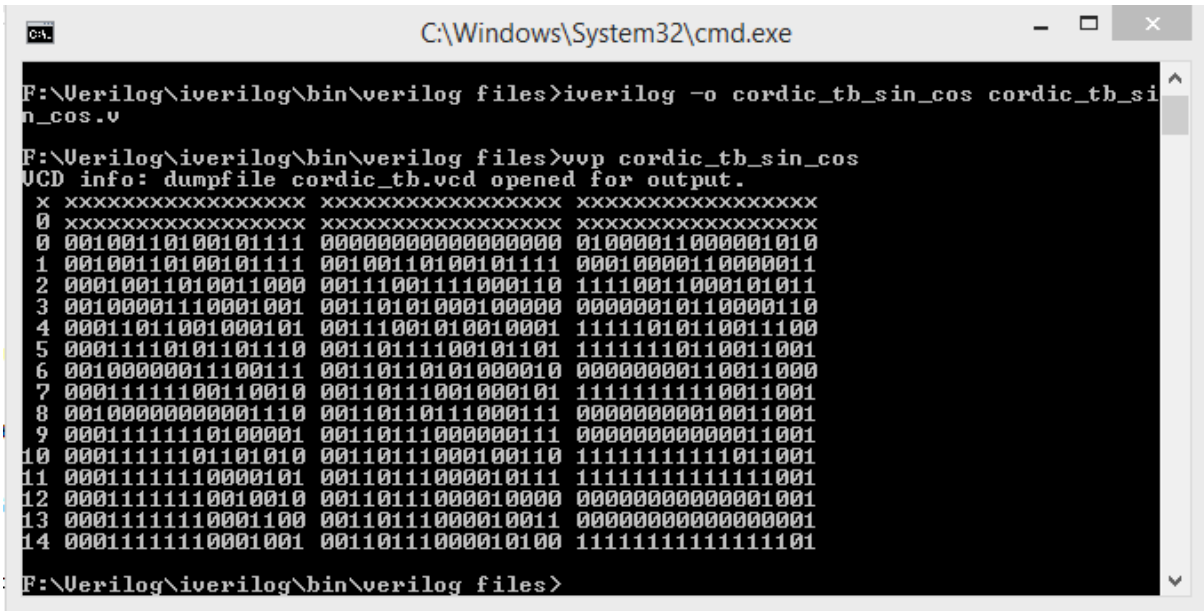


Fig 5: Simulation output for computing sin and cos using CORDIC

The values present in the 1st and 2nd column of the last (14th) iteration, represent the cosine and sine of the given input angle.

Ideally, cos60° = 0.5 and here, $x_{14} = (0.011111110001001)_2 = 0.4963$.

And, sin60° = 0.866 and here, $y_{14} = (0.110111000010100)_2 = 0.8599$.

<u>Algorithm</u>: <u>Pseudo-code for realizing arctan function in CORDIC</u>

Input: X - coordinate of rotation vector (x_1), Y - coordinate of rotation vector (y_1), Angle of rotation (z_1), Dynamic ranges of the rotation angle: {-90°, 90°} and Maximum permissible iterations (i_{max}).

Output: arctan function {$\tan^{-1}(\alpha)$}: z_{i+1}

1 Initialize: $x_1 = 1$, $y_1 = \alpha$, $z_1 = 0$, i_{max}

2 if ($y_1<0$) then

3 $x_1 \leftarrow -y_1$

4 $y_1 \leftarrow x_1$

$z_1 \leftarrow 90°$
else
$x_1 \leftarrow y_1$
$y_1 \leftarrow -x_1$
$z_1 \leftarrow -90°$
end if
for all iterations i do
if ($y_i < 0$) then
$\sigma_i \leftarrow 1$
else
$\sigma_i \leftarrow -1$
end if
$x_{i+1} \leftarrow x_i - \sigma_i * 2^{-i} * y_i$
$y_{i+1} \leftarrow y_i + \sigma_i * 2^{-i} * x_i$
$z_{i+1} \leftarrow z_i + \sigma_i * \tan^{-1}(2^{-i})$
end for
return $-z_{i+1}$

4.2.2 Implementation in Verilog

- **Fixed Point Implementation:**

Here, we have followed a 17 bit fixed point format for representing all the values. The MSB i.e., bit 17 is the sign bit, bit 16 is the whole number part and the rest 15 bits represent the fractional parts. The angles are represented in radian.

- **Input:**

$x_1 = 1$, $y_i = 0.577$, $z_i = 0°$ in radians, specified in the above mentioned format.

- **Code Snippets:**

```
always @ (posedge clk)

  if (rst) begin
    x_1 <= 0;
    y_1 <= 0;
    z_1 <= 0;
  end else begin

    if (init) begin
      x_1 <= x_i;
      y_1 <= y_i;
      z_1 <= z_i;
    end else if (
    y_i[16]
    ) begin
      x_1 <= x_i - {y_i[16],y_i_shifted}; //shifter(y_1,i); //(y_1 >> i);
      y_1 <= y_i + {x_i[16],x_i_shifted}; //shifter(x_1,i); //(x_1 >> i);
      z_1 <= z_i + tangle;
```

```
    end else begin
      x_1 <= x_i + {y_i[16],y_i_shifted}; //shifter(y_1,i); //(y_1 >> i);
      y_1 <= y_i - {x_i[16],x_i_shifted}; //shifter(x_1,i); //(x_1 >> i);
      z_1 <= z_i - tangle;

    end
  end
assign x_o = x_1;
assign y_o = y_1;
assign z_o = z_1;
```

➢ Simulation Output:

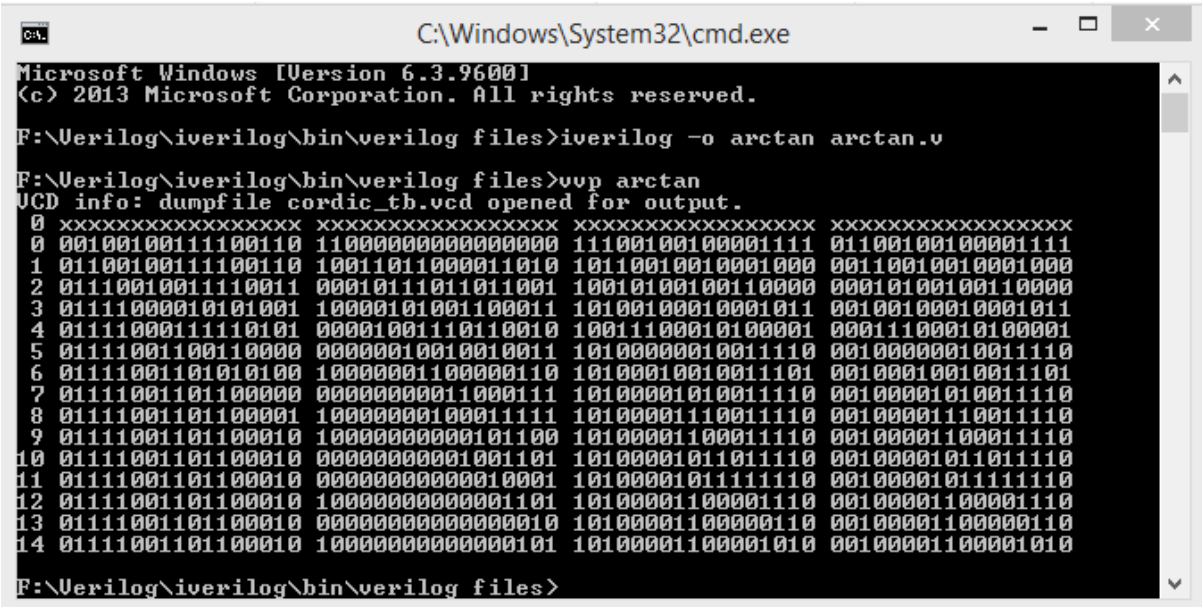


Fig 6: Simulation output for computing arctan using CORDIC

The values present in the 4th column of the last (14th) iteration, represent the arctan of the given input value.

Ideally, $\tan^{-1}(0.577) = 30° = \pi/6$ radian = 0.5236 radian.

Here, $z_{14} = (0.100001100001010)_2 = 0.52374$.

4.3 Implementation of Jacobi Eigen Value Algorithm in Verilog

- **Input:**

$$A = \begin{bmatrix} 0.33 & 0.47 & 0.67 \\ 0.47 & 1.00 & 0.47 \\ 0.67 & 0.47 & 0.33 \end{bmatrix}$$

- **Code Snippets:**

  - **Finding Largest Off-Diagonal Element**

```
always@(posedge clk)
        begin
                max = mat[0][1];
                for(i=0; i<order; i=i+1)
                begin
                        for(j=0; j<order; j=j+1)
                        begin
                                if(i!=j && mat[i][j]!= 0)
                                begin
                                        if(mat[i][j]>max || mat[i][j]<-max)
                                        begin
                                                max = mat[i][j];
                                                row = i;
                                                col = j;
                                        end
                                end
                        end
                end
        $write("%f %f %f %f %f",mat[row][row],max,mat[col][col],row,col);
        flag = 1'b0;
        R_pq = max;
        R_pp = mat[row][row];
        R_qq = mat[col][col];
        p = row;
        q = col;
        if(R_qq == R_pp)
        begin
                flag = 1'b1;
                alpha = 0;
        end
        else
        begin
                alpha = 2*R_pq/(R_qq - R_pp);
        end
        $display("\n%f",alpha);
        //instantiate CORDIC module
        end
```

- **Givens' Rotation**

```
for(i=0;i<order;i=i+1)
        begin
                for(j=0;j<order;j=j+1)
                begin
                        if(i==j)
                        begin
                                givens[i][j] = 1;
                                givens_T[i][j] = 1;
                        end
                        else
                        begin
                                givens[i][j] = 0;
                                givens_T[i][j] = 0;
                        end
                end
        end

        givens[p][p] = cos;
        givens[p][q] = sin;
        givens[q][p] = -sin;
        givens[q][q] = cos;

        givens_T[p][p] = cos;
        givens_T[p][q] = -sin;
        givens_T[q][p] = sin;
        givens_T[q][q] = cos;

        for(i=0;i<order;i=i+1)
        begin
                for(j=0;j<order;j=j+1)
                begin
                        for(k=0;k<order;k=k+1)
                        begin
    temp[i][j] = temp[i][j] + givens_T[i][k]*mat[k][j];
                        end
                end
        end

        for(i=0;i<order;i=i+1)
        begin
                for(j=0;j<order;j=j+1)
                begin
                        for(k=0;k<order;k=k+1)
                        begin
    jacobi[i][j] = jacobi[i][j] + temp[i][k]*givens[k][j];
                        end
                end
        end
end
```

➢ Simulation Results:

```
Log    Share
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4675 kB (elbread=427 elab2=4114 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: -0.330000 0.000000 0.000000
# KERNEL: 0.000000 0.329999 0.000000
# KERNEL: 0.000000 0.000000 1.669996
# RUNTIME: Info: RUNTIME_0068 testbench.sv (36): $finish called.
# KERNEL: Time: 8 ns,  Iteration: 0,  Instance: /test_bench,  Process: @INITIAL#17_0@.
# KERNEL: stopped at time: 8 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Fig 7: Simulation output for computing eigenvalues using Jacobi

As, we can see from the above figure, the resultant matrix is reduced to a diagonal matrix. The diagonal elements represent the eigenvalues of the input matrix.

Ideally, eigenvalues are $-\frac{1}{3}, \frac{1}{3}, \frac{5}{3}$.

Here, eigenvalues are -0.330000, 0.329999, 1.669996.

The eigenvectors are given by the columns of the matrix: $J = J_1 * J_2$

$$\Rightarrow J = \begin{bmatrix} 0.707106 & 0 & 0.707106 \\ 0 & 1 & 0 \\ -0.707106 & 0 & 0.707106 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.707106 & 0.707106 \\ 0 & -0.707106 & 0.707106 \end{bmatrix}$$

$$= \begin{bmatrix} 0.707106 & -0.499999 & 0.499999 \\ 0 & 0.707106 & 0.707106 \\ -0.707106 & -0.499999 & 0.499999 \end{bmatrix}$$

Ideally, eigenvalues are $\begin{bmatrix} 1/\sqrt{2} \\ 0 \\ -1/\sqrt{2} \end{bmatrix}, \begin{bmatrix} -1/2 \\ 1/\sqrt{2} \\ -1/2 \end{bmatrix}, \begin{bmatrix} 1/2 \\ 1/\sqrt{2} \\ 1/2 \end{bmatrix}$.

Here, eigenvectors are $\begin{bmatrix} 0.707106 \\ 0 \\ -0.707106 \end{bmatrix}, \begin{bmatrix} -0.499999 \\ 0.707106 \\ -0.499999 \end{bmatrix}, \begin{bmatrix} 0.499999 \\ 0.707106 \\ 0.499999 \end{bmatrix}$.

4.4 How to evaluate eigenvalues of a non-symmetric matrix?

As mentioned in the Jacobi Eigen Value algorithm, Jacobi method is only applicable for a real-symmetric matrix. However, we can convert every 2X2 non-symmetric matrix into a 2X2 symmetric matrix in the following manner.

Let us consider a 2X2 non-symmetric matrix:

A = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

We can compute J = $\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$ where θ = $\tan^{-1}(\frac{b-c}{d+a})$.

Now, multiplying A by J^T, we get,

B = J^T * A = $\cos\theta \begin{bmatrix} a - c\tan\theta & b - d\tan\theta \\ c + a\tan\theta & d + b\tan\theta \end{bmatrix}$ which is a symmetric matrix.

Let us verify the above fact by taking an example.

A = $\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$

Now, $\tan\theta = \frac{4-3}{5+2} = \frac{1}{7}$, and J = $\begin{bmatrix} \frac{7}{\sqrt{50}} & \frac{1}{\sqrt{50}} \\ -\frac{1}{\sqrt{50}} & \frac{7}{\sqrt{50}} \end{bmatrix}$.

Therefore, B = J^T * A = $\begin{bmatrix} \frac{7}{\sqrt{50}} & -\frac{1}{\sqrt{50}} \\ \frac{1}{\sqrt{50}} & \frac{7}{\sqrt{50}} \end{bmatrix} * \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} \frac{11}{\sqrt{50}} & \frac{23}{\sqrt{50}} \\ \frac{23}{\sqrt{50}} & \frac{39}{\sqrt{50}} \end{bmatrix}$, which is a symmetric matrix.

However, the eigenvalues of A and B are not identical. Thus, we intend to also solve this problem in the subsequent part of this project.

As we know, every square matrix can be expressed as a sum of symmetric and skew-symmetric matrices, therefore, by property of eigenvalues, the eigenvalue of a given non-symmetric square matrix must also be obtained from a linear combination of the eigenvalues of its corresponding symmetric and skew-symmetric matrices.

Mathematically, a given non-symmetric square matrix, A of a finite order, N can be expressed as:

$$A = B + C, \text{ where } B = \frac{1}{2}(A + A^T) \text{ and } C = \frac{1}{2}(A - A^T)$$

Now, if λ_A is an eigenvalue of A, λ_B is an eigenvalue of B and λ_C is an eigenvalue of C, then we can write $\lambda_A = f(\lambda_B, \lambda_C)$. Hence, if we can find out λ_C i.e., the eigenvalue of skew-symmetric matrix, C, then λ_A can be computed as λ_B, the eigenvalue of symmetric matrix, B, is already obtained beforehand from Jacobi method.

4.5 Eigensystem Computation for Skew-Symmetric Matrices

The eigensystem computation algorithm developed by R.C. Ward and L. J. Gray in the late 1970s is an efficient algorithm for computing eigenvalues and eigenvectors of either a skew-symmetric matrix or a symmetric tridiagonal matrix with a constant (zero) diagonal. The algorithm uses only orthogonal similarity transformations and is believed to be one of the most efficient algorithm available for computing all eigenpairs.

In this method, the given system is first transformed into a tridiagonal system. This algorithm Is designed for a matrix of dimension 2n X 2n. However, it is seen that a skew symmetric matrix whose dimension is odd has at least one 0 as an eigenvalue. Thus, a method based upon Givens Transformations is performed first on a given matrix of odd order to reduce its dimension to an even number. One of the main advantages of this approach is the observation that the eigenvector matrix is half sparse. This results in significant savings over other alternative algorithms.

The three main subroutines or steps involved in the Eigensystem Computation algorithm is given below:

- TRIZD - transforms an arbitrary real skew-symmetric matrix to skew-symmetric tridiagonal form using orthogonal similarity transformations and saving the pertinent information about these transformations.
- IMZD - computes the eigenvalues and optionally, the eigenvectors of the symmetric tridiagonal matrix with 0s on the diagonal or of a skew-symmetric tridiagonal matrix.
- TBAKZD - computes the eigenvectors of the real skew-symmetric matrix by back transforming the eigenvectors of the corresponding skew-symmetric tridiagonal matrix determined by TRIZD.

4.5.1 Implementation in FORTRAN

- **Input:**

$$A = \begin{bmatrix} 0 & 1 & 0 & -5 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 5 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & -2 & -1 & 5 & 0 \\ 5 & 0 & 0 & 0 & -1 & -2 & 0 & 0 \\ 0 & -5 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 & 0 & -5 \\ 0 & 2 & -5 & 0 & 0 & 0 & 0 & 1 \\ -2 & 0 & 0 & 0 & 0 & 5 & -1 & 0 \end{bmatrix}$$

- **Code Snippets:**

- **TRIZD subroutine:-**

```
SUBROUTINE trizd(n, a, e)

        ! *** COMPUTE ELEMENTS OF U VECTOR
```

```
      DO  k = 1, l
        a(i,k) = a(i,k) / scale
        h = h + a(i,k) * a(i,k)
      END DO

      f = a(i,l)
      g = -SIGN(SQRT(h),f)
      e(i) = scale * g
      h = h - f * g
      a(i,l) = f - g
      IF (l /= 1) THEN

! *** COMPUTE ELEMENTS OF A*U/H

        DO  j = 1, l
          g = DOT_PRODUCT( a(j,1:j-1), a(i,1:j-1) ) -  &
              DOT_PRODUCT( a(j+1:l,j), a(i,j+1:l) )

          e(j) = g / h
        END DO

! *** COMPUTE REDUCED A

        DO  j = 2, l
          f = a(i,j)
          g = e(j)

          DO  k = 1, j-1
            a(j,k) = a(j,k) + f * e(k) - g * a(i,k)
          END DO
        END DO
      END IF

      a(i,1:l) = scale * a(i,1:l)
    END IF

    a(i,i) = scale * SQRT(h)
  END DO
END IF

e(1) = 0.
RETURN
END SUBROUTINE trizd
```

- **IMZD subroutine:-**

```
SUBROUTINE imzd(n, e, matz, skew, z, ierr)

! *** FIND ZERO EIGENVALUE OF ODD ORDERED SUBMATRICES

  c = 0.
  s = -1.
```

```
  DO  i = l0, mm1, 2
    k = mm1 + l0 - i
    kp1 = k + 1
    q = -s * e(kp1)
    e(kp1) = c * e(kp1)
    IF (ABS(e(k)) <= ABS(q)) THEN
      c = e(k) / q
      r = SQRT(c*c+1.)
      e(k) = q * r
      s = 1. / r
      c = c * s
    ELSE
      s = q / e(k)
      r = SQRT(1.+s*s)
      e(k) = e(k) * r
      c = 1. / r
      s = s * c
    END IF
    IF (matz) THEN

! *** ACCUMULATE TRANSFORMATIONS FOR EIGENVECTORS

      km1 = k - 1
      z(km1,m) = -s * z(km1,km1)
      z(km1,km1) = c * z(km1,km1)
      DO  j = kp1, m, 2
        z(j,km1) = s * z(j,m)
        z(j,m) = c * z(j,m)
      END DO
    END IF

  END DO
  m = mm1
  mm1 = m - 1
  IF (l0 == m) GO TO 170

! *** ITERATION CONVERGED TO ONE ZERO EIGENVALUE

  160 e(m) = 0.
  m = mm1
  GO TO 180

! *** ITERATION CONVERGED TO EIGENVALUE PAIR

  170 e(mm1) = e(m)
  e(m) = -e(m)
  m = m - 2

  180 its = 0
  mm1 = m - 1
  IF (m > l0) GO TO 100
```

```
      IF (m == l0) GO TO 170
      IF (.NOT.matz) GO TO 30
      IF (skew) GO TO 30


    230 RETURN
END SUBROUTINE imzd
```

- **TBAKZD subroutine:-**

```
SUBROUTINE tbakzd(n, a, m, z)

    IF (m /= 0) THEN
      IF (n /= 1) THEN

        DO  i = 2, n
          l = i - 1
          h = a(i,i)
          IF (h /= 0.0_dp) THEN

            DO  j = 1, m
              s = DOT_PRODUCT( a(i,1:l), z(1:l,j) )

              s = (s/h) / h

              z(1:l,j) = z(1:l,j) - s * a(i,1:l)

            END DO
          END IF

        END DO
      END IF
    END IF

    RETURN
END SUBROUTINE tbakzd
```

Simulation Results:

The eigenvalues and the corresponding eigenvectors as obtained after execution of the program in online FORTRAN simulator are listed below in the subsequent snapshots.

```
 0.80000000E+01 * I     -0.50000000E+00  +  -0.69903492E-16 * I
                        -0.27755576E-16  +  -0.50000000E+00 * I
                         0.24980018E-15  +   0.50000000E+00 * I
                        -0.16653345E-15  +   0.50000000E+00 * I
                         0.50000000E+00  +  -0.16530625E-15 * I
                         0.50000000E+00  +  -0.22207641E-16 * I
                        -0.50000000E+00  +   0.28768780E-16 * I
                         0.00000000E+00  +  -0.50000000E+00 * I


-0.80000000E+01 * I     -0.50000000E+00  +   0.69903492E-16 * I
                        -0.27755576E-16  +   0.50000000E+00 * I
                         0.24980018E-15  +  -0.50000000E+00 * I
                        -0.16653345E-15  +  -0.50000000E+00 * I
                         0.50000000E+00  +   0.16530625E-15 * I
                         0.50000000E+00  +   0.22207641E-16 * I
                        -0.50000000E+00  +  -0.28768780E-16 * I
                         0.00000000E+00  +   0.50000000E+00 * I
```

Fig 8: Simulation output showing an eigenvalue pair and its corresponding eigenvectors

```
-0.20000000E+01 * I      0.50000000E+00  +   0.20602849E-15 * I
                         0.33306691E-15  +   0.50000000E+00 * I
                         0.38857806E-15  +   0.50000000E+00 * I
                         0.00000000E+00  +   0.50000000E+00 * I
                         0.50000000E+00  +   0.72321486E-16 * I
                         0.50000000E+00  +   0.65453191E-16 * I
                         0.50000000E+00  +  -0.84791017E-16 * I
                         0.00000000E+00  +   0.50000000E+00 * I


 0.20000000E+01 * I      0.50000000E+00  +  -0.20602849E-15 * I
                         0.33306691E-15  +  -0.50000000E+00 * I
                         0.38857806E-15  +  -0.50000000E+00 * I
                         0.00000000E+00  +  -0.50000000E+00 * I
                         0.50000000E+00  +  -0.72321486E-16 * I
                         0.50000000E+00  +  -0.65453191E-16 * I
                         0.50000000E+00  +   0.84791017E-16 * I
                         0.00000000E+00  +  -0.50000000E+00 * I
```

Fig 9: Simulation output showing an eigenvalue pair and its corresponding eigenvectors

```
-0.40000000E+01 * I      0.50000000E+00  +  -0.14475210E-15 * I
                        -0.42327253E-15  +  -0.50000000E+00 * I
                        -0.34694470E-15  +  -0.50000000E+00 * I
                         0.55511151E-16  +   0.50000000E+00 * I
                        -0.50000000E+00  +   0.18892143E-15 * I
                         0.50000000E+00  +  -0.34563491E-16 * I
                        -0.50000000E+00  +   0.11668676E-15 * I
                         0.00000000E+00  +   0.50000000E+00 * I


 0.40000000E+01 * I      0.50000000E+00  +   0.14475210E-15 * I
                        -0.42327253E-15  +   0.50000000E+00 * I
                        -0.34694470E-15  +   0.50000000E+00 * I
                         0.55511151E-16  +  -0.50000000E+00 * I
                        -0.50000000E+00  +  -0.18892143E-15 * I
                         0.50000000E+00  +   0.34563491E-16 * I
                        -0.50000000E+00  +  -0.11668676E-15 * I
                         0.00000000E+00  +  -0.50000000E+00 * I
```

Fig 10: Simulation output showing an eigenvalue pair and its corresponding eigenvectors

```
-0.60000000E+01 * I      0.50000000E+00  +   0.10434969E-15 * I
                        -0.29143354E-15  +   0.50000000E+00 * I
                         0.13877788E-15  +  -0.50000000E+00 * I
                        -0.22204460E-15  +   0.50000000E+00 * I
                         0.50000000E+00  +   0.33061251E-15 * I
                        -0.50000000E+00  +   0.21728048E-16 * I
                        -0.50000000E+00  +  -0.10005914E-15 * I
                         0.00000000E+00  +  -0.50000000E+00 * I


 0.60000000E+01 * I      0.50000000E+00  +  -0.10434969E-15 * I
                        -0.29143354E-15  +  -0.50000000E+00 * I
                         0.13877788E-15  +   0.50000000E+00 * I
                        -0.22204460E-15  +  -0.50000000E+00 * I
                         0.50000000E+00  +  -0.33061251E-15 * I
                        -0.50000000E+00  +  -0.21728048E-16 * I
                        -0.50000000E+00  +   0.10005914E-15 * I
                         0.00000000E+00  +   0.50000000E+00 * I
```

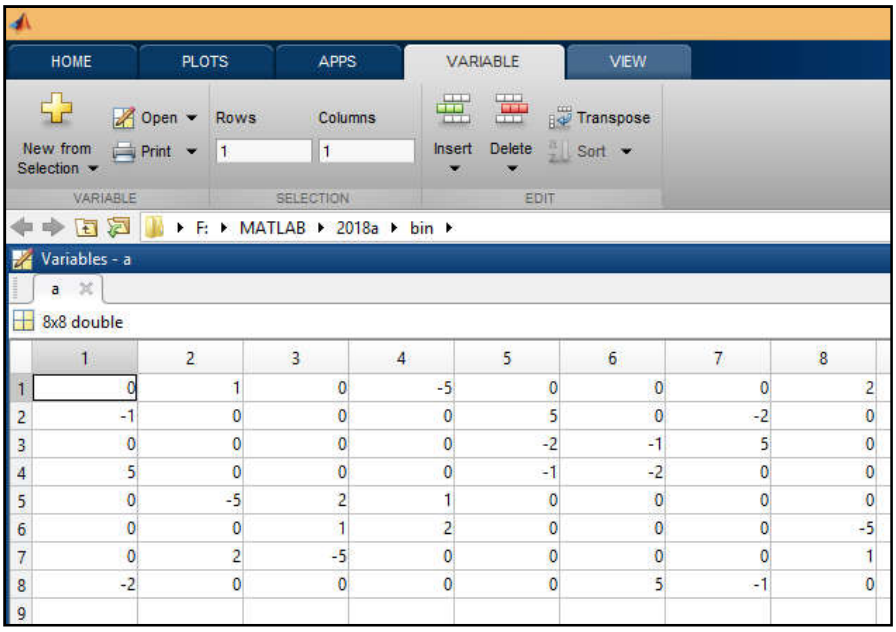
Fig 11: Simulation output showing an eigenvalue pair and its corresponding eigenvectors

Thus, $\lambda_A = \pm 8i, \pm 2i, \pm 4i, \pm 6i$ and as we can see each conjugate eigenvalue pair has an 8X1 eigenvector listed beside them.

Therefore, we can conclude that since, the algorithm executes perfectly in an online platform of a software programming language, in this case, FORTRAN, in principle, this algorithm can be implemented in Hardware Description Languages like Verilog also and hence, we can realize it by a piece of Hardware.

4.5.2 Evaluating Eigenvalues with the help of Eigensystem Computation Algorithm in MATLAB

- **Input:**



	1	2	3	4	5	6	7	8
1	0	1	0	-5	0	0	0	2
2	-1	0	0	0	5	0	-2	0
3	0	0	0	0	-2	-1	5	0
4	5	0	0	0	-1	-2	0	0
5	0	-5	2	1	0	0	0	0
6	0	0	1	2	0	0	0	-5
7	0	2	-5	0	0	0	0	1
8	-2	0	0	0	0	5	-1	0
9								

Fig 12: Input skew-symmetric matrix of order 8

- **Code Snippets:**

- **TRIZD subroutine:-**

```
function [n, a, e]=trizd(n, a, e)
   % *** COMPUTE ELEMENTS OF U VECTOR
      for  k = 1: l
         a(i,k) = a(i,k)/scale_ml;
         h = h + a(i,k)*a(i,k);
      end
      f = a(i,l);
      if(f>=0)
         g = -sqrt(h);
      else
         g = sqrt(h);
      end
      e(i) = scale_ml * g;
      h = h - f * g;
      a(i,l) = f - g;
      if(l ~= 1)
   % *** COMPUTE ELEMENTS OF A*U/H
```

```
        for  j = 1: l
          if(j==1)
            g = - dot( a([(j+1):l],j), a(i,[(j+1):l]) );
          else
            if(j==l)
              g = dot( a(j,[1:(j-1)]), a(i,[1:(j-1)]) );
            else
              g = dot( a(j,[1:(j-1)]), a(i,[1:(j-1)]) ) - dot( a([(j+1):l],j), a(i,[(j+1):l]) );
            end
          end
          e(j) = g / h;
        end
% *** COMPUTE REDUCED A
        for  j = 2: l
          f = a(i,j);
          g = e(j);
          for  k = 1: j-1
            a(j,k) = a(j,k) + f*e(k) - g*a(i,k);
          end
        end
      end
      a(i,[1:l]) = scale_ml*a(i,[1:l]);
    end
    a(i,i) = scale_ml*sqrt(h);
  end
  end
  e(1) = 0;
  end %subroutine trizd
```

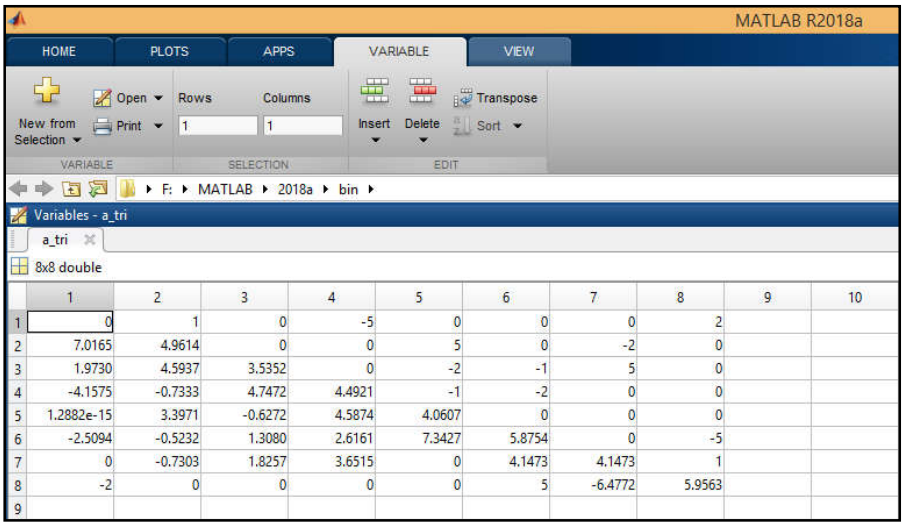
- **IMZD subroutine:-**

```
function [n, e, matz, skew, z, ierr, its, lambda]=imzd(n, e, matz, skew)

% *** FIND ZERO EIGENVALUE OF ODD ORDERED SUBMATRICES
    c = 0;
    s = -1;
    for  i = l0: 2: mm1
      k = fix(mm1 + l0 - i);
      kp1 = fix(k + 1);
      q = -s * e(kp1);
      e(kp1) = c * e(kp1);
      if (abs(e(k)) <= abs(q))
        c = e(k) / q;
        r = sqrt(c*c+1);
        e(k) = q * r;
        s = 1/r;
        c = c * s;
      else
      s = q / e(k);
      r = sqrt(1+s*s);
      e(k) = e(k) * r;
      c = 1/r;
```

```
        s = s * c;
        end
        if (matz)

    % *** ACCUMULATE TRANSFORMATIONS FOR EIGENVECTORS
        km1 = fix(k - 1);
        z(km1,m) = -s * z(km1,km1);
        z(km1,km1) = c * z(km1,km1);
        for  j = kp1: 2: m
           z(j,km1) = s * z(j,m);
           z(j,m) = c * z(j,m);
        end
        %j = fix(m+ 2);
        end
      end
      %i = fix(mm1+ 2);
      m = fix(mm1);
      mm1 = fix(m - 1);
      if (l0 == m)
        goto(371)
        return
          end

% *** ITERATION CONVERGED TO ONE ZERO EIGENVALUE
      e(m) = 0;
      m = fix(mm1);
      if (1>0)
        goto(375)
      return
      end

    % *** ITERATION CONVERGED TO EIGENVALUE PAIR
      e(mm1) = e(m);
      e(m) = -e(m);
      m = fix(m - 2);
      its = 0;
      mm1 = fix(m - 1);
      if (m > l0)
        goto(275)
        return
      end
      if (m == l0)
        goto(371)
        return
      end
      if (~ matz)
        goto(171)
        return
      end
      if (skew)
        goto(171)
```

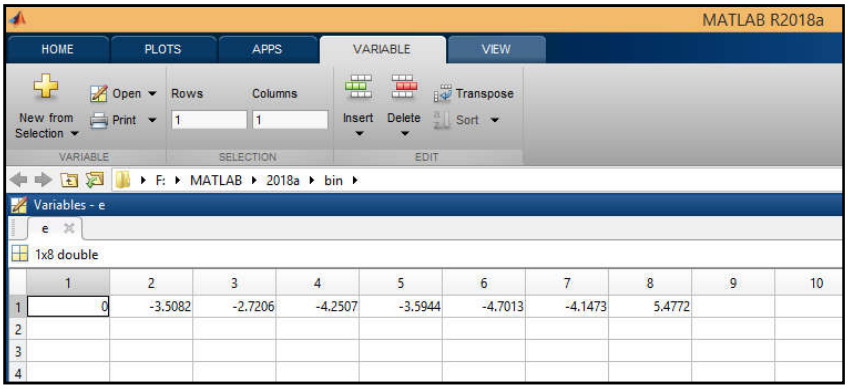
```
        return
    end
lambda = zeros(1,n); lambda(1:2:n)=abs(e(2:2:n))*sqrt(-1);lambda(2:2:n)=-abs(e(2:2:n))*sqrt(-1);
end %subroutine imzd
```

➢ Simulation Output:



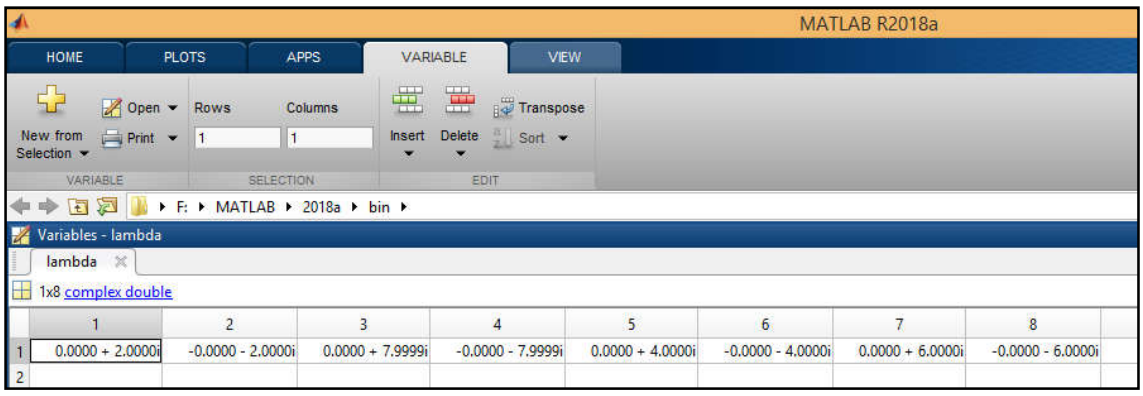
	1	2	3	4	5	6	7	8	9	10
1	0	1	0	-5	0	0	0	2		
2	7.0165	4.9614	0	0	5	0	-2	0		
3	1.9730	4.5937	3.5352	0	-2	-1	5	0		
4	-4.1575	-0.7333	4.7472	4.4921	-1	-2	0	0		
5	1.2882e-15	3.3971	-0.6272	4.5874	4.0607	0	0	0		
6	-2.5094	-0.5232	1.3080	2.6161	7.3427	5.8754	0	-5		
7	0	-0.7303	1.8257	3.6515	0	4.1473	4.1473	1		
8	-2	0	0	0	0	5	-6.4772	5.9563		
9										

Fig 13: Subroutine TRIZD output tridiagonal matrix after orthogonal transformations



	1	2	3	4	5	6	7	8	9	10
1	0	-3.5082	-2.7206	-4.2507	-3.5944	-4.7013	-4.1473	5.4772		
2										
3										
4										

Fig 14: Subroutine TRIZD output array showing the lower subdiagonal elements of the tridiagonal matrix



	1	2	3	4	5	6	7	8
1	0.0000 + 2.0000i	-0.0000 - 2.0000i	0.0000 + 7.9999i	-0.0000 - 7.9999i	0.0000 + 4.0000i	-0.0000 - 4.0000i	0.0000 + 6.0000i	-0.0000 - 6.0000i
2								

Fig 15: Subroutine IMZD output array containing the eigenvalues

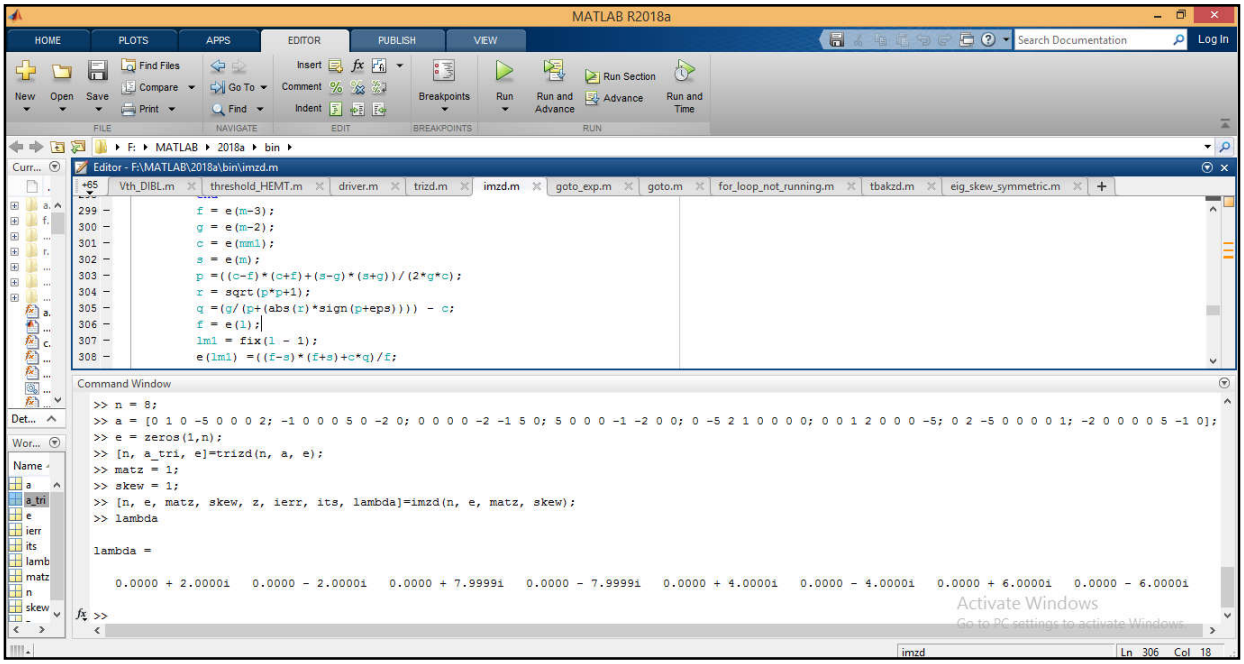


Fig 16: Simulation output for computing eigenvalues using Eigensystem Computation Algorithm

As, we can see from the above figure, the resultant array, lambda correctly holds the eigenvalues of the input skew-symmetric matrix.

Ideally, eigenvalues are $\pm 2i$, $\pm 8i$, $\pm 4i$, $\pm 6i$.

Here, eigenvalues are $\pm 2i$, $\pm 7.9999i$, $\pm 4i$, $\pm 6i$.

4.6 Eigenvalue decomposition of non-symmetric matrices

- **Input:**

$$A = \begin{bmatrix} 16 & 8 & 6 & 6 \\ 12 & 8 & -5 & 9 \\ 16 & 9 & 6 & 17 \\ 4 & -9 & 13 & 12 \end{bmatrix}$$

- **Extracting symmetric and skew-symmetric matrices from input matrix:**

$$B = \frac{1}{2}(A + A^T) = \begin{bmatrix} 16 & 10 & 11 & 5 \\ 10 & 8 & 2 & 0 \\ 11 & 2 & 6 & 15 \\ 5 & 0 & 15 & 12 \end{bmatrix}$$

$$C = \frac{1}{2}(A - A^T) = \begin{bmatrix} 0 & -2 & -5 & 1 \\ 2 & 0 & -7 & 9 \\ 5 & 7 & 0 & 2 \\ -1 & -9 & -2 & 0 \end{bmatrix}$$

- **Extracting eigenvalues of the symmetric matrix by Jacobi Method in Verilog:**

B_norm = B/16;

$$B\_norm = \begin{bmatrix} 1 & 0.625 & 0.6875 & 0.3125 \\ 0.625 & 0.5 & 0.125 & 0 \\ 0.6875 & 0.125 & 0.375 & 0.9375 \\ 0.3125 & 0 & 0.9375 & 0.75 \end{bmatrix}$$

1st iteration:

- <u>Largest off-diagonal element computation:</u>

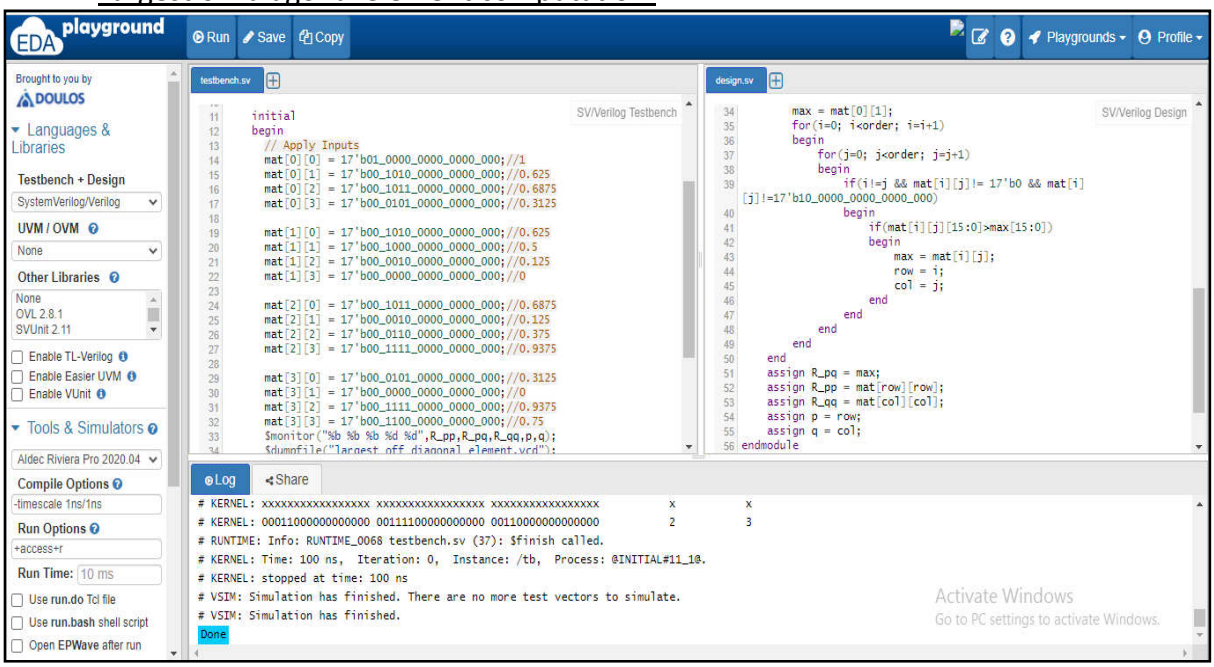


Fig 17: Simulation output for computing largest off-diagonal element

Thus, we get, p = 2, q = 3, R_pp = $(0.011)_2$ = 0.375, R_qq = $(0.11)_2$ = 0.75 and R_pq = $(0.1111)_2$ = 0.9375.

- <u>Arctan(x) computation:</u>

  $x = \frac{2R_{pq}}{R_{qq}-R_{pp}} = 5$

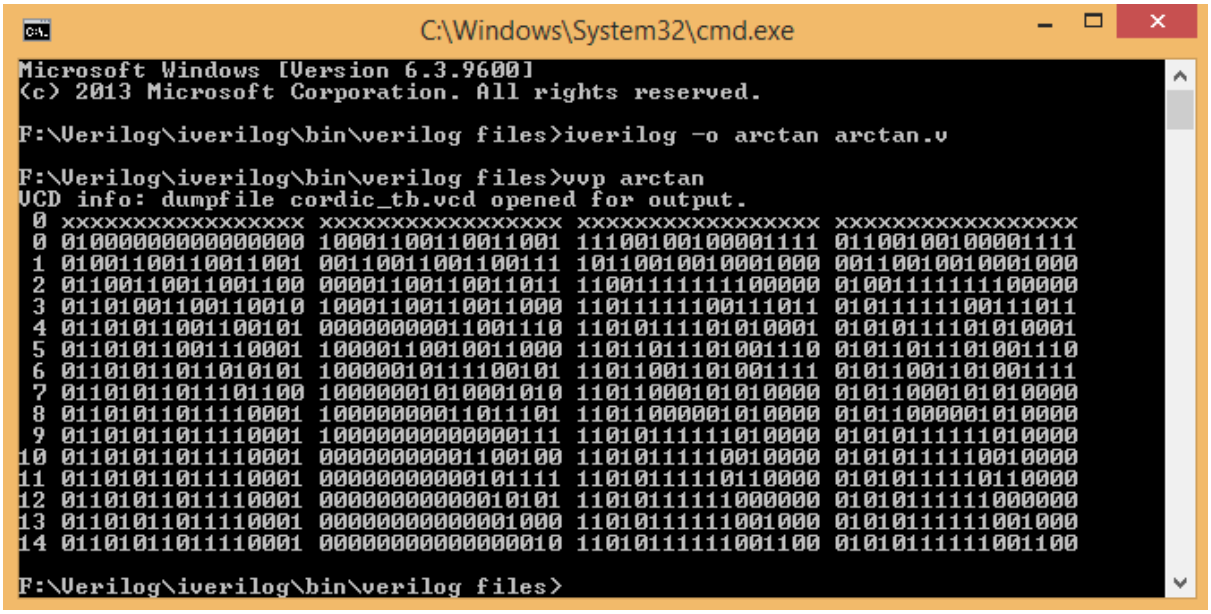
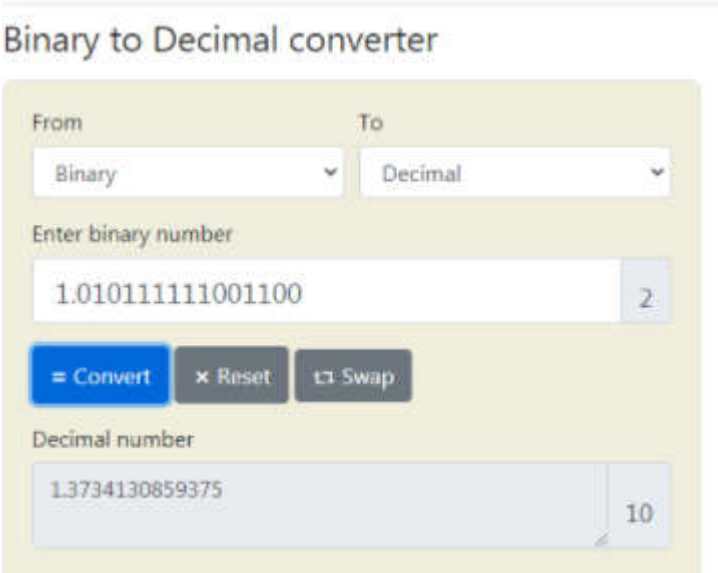


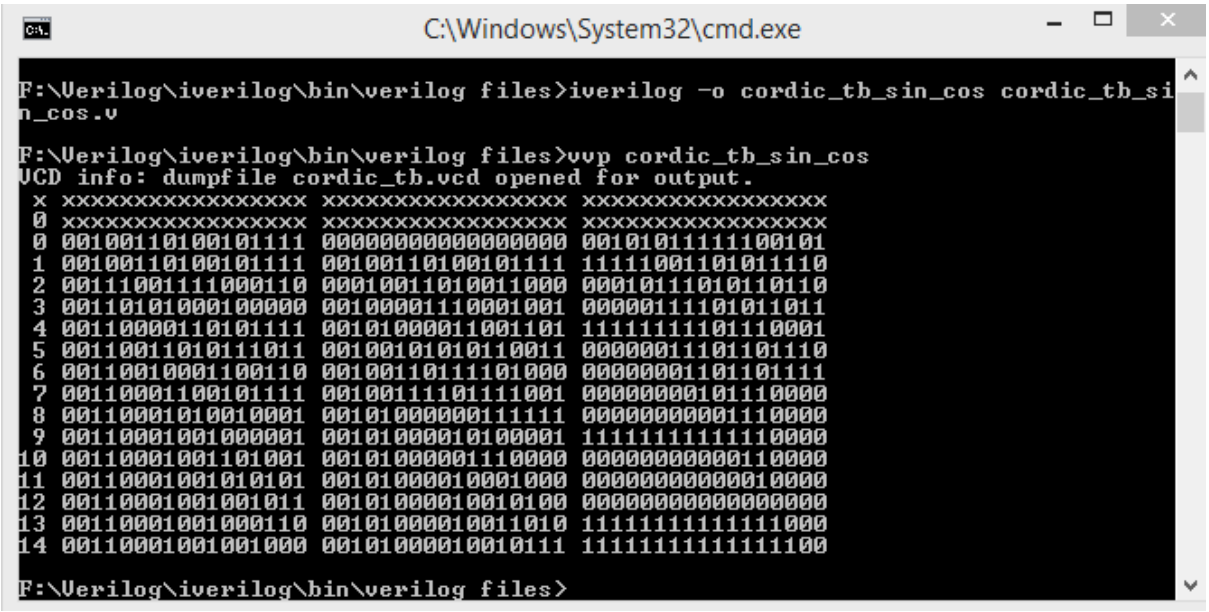
Fig 18: Simulation output for computing arctan(x) in CORDIC



Thus, we get, $\tan^{-1}(x)$ = 1.3734 radian.

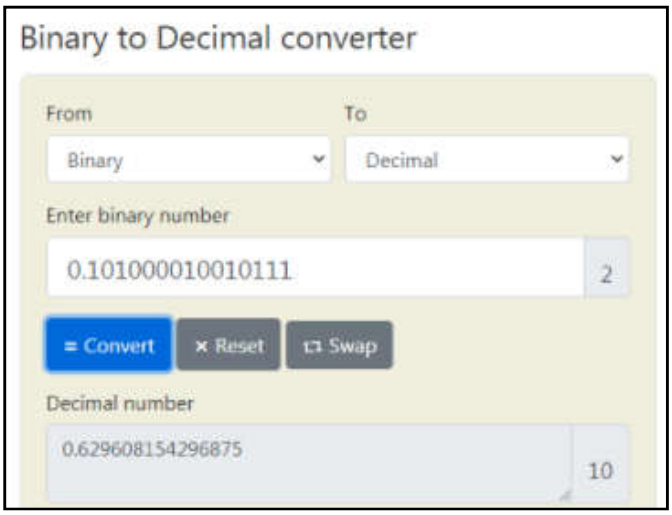
- <u>cos(θ) and sin(θ) computation:</u>

  $\theta = \frac{1}{2}\tan^{-1}(x)$ = 0.6867 radian.

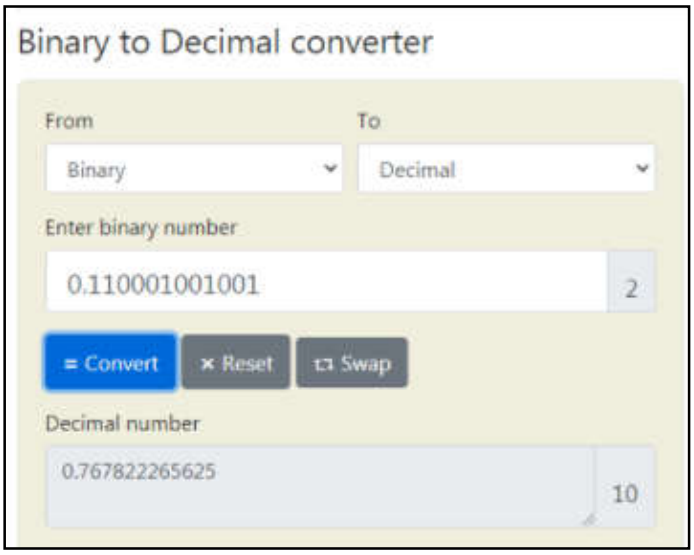


```
C:\Windows\System32\cmd.exe

F:\Verilog\iverilog\bin\verilog files>iverilog -o cordic_tb_sin_cos cordic_tb_si
n_cos.v

F:\Verilog\iverilog\bin\verilog files>vvp cordic_tb_sin_cos
VCD info: dumpfile cordic_tb.vcd opened for output.
 x xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 0010011010010111 1 0000000000000000 0010101111110010 1
 1 0010011010010111 1 0010011010010111 1111100110101111 0
 2 0011100111100011 0 0001001101001100 0 0001011101011011 0
 3 0011010100010000 0 0010000111000100 1 0000011110101101 1
 4 0011000011010111 1 0010100001100110 1 1111111110111000 1
 5 0011001101011101 1 0010010101011001 1 0000001110110111 0
 6 0011001000110011 0 0010011011110100 0 0000000110110111 1
 7 0011000110010111 1 0010011110111100 1 0000000010111000 0
 8 0011000101001000 1 0010100000011111 1 0000000000111000 0
 9 0011000100100000 1 0010100001010000 1 1111111111110000 0
10 0011000100110100 1 0010100000111000 0 0000000000011000 0
11 0011000100101010 1 0010100001000100 0 0000000000001000 0
12 0011000100100101 1 0010100001001010 0 0000000000000000 0
13 0011000100100011 0 0010100001001101 0 1111111111111000 0
14 0011000100100100 0 0010100001001011 1 1111111111111100 0

F:\Verilog\iverilog\bin\verilog files>
```

Fig 19: Simulation output for computing cos(θ) and sin(θ) in CORDIC



Thus, we get, sin(θ) = 0.63.



Thus, we get, cos(θ) = 0.77.

- <u>Givens Rotation:</u>

```
Log    Share
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4677 kB (elbread=427 elab2=4115 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: 1.000000 0.625000 0.331125 0.672800
# KERNEL: 0.625000 0.500000 0.095978 0.078701
# KERNEL: 0.331125 0.095978 -0.388039 -0.000212
# KERNEL: 0.672800 0.078701 -0.000212 1.497242
# RUNTIME: Info: RUNTIME_0068 testbench.sv (46): $finish called.
# KERNEL: Time: 8 ns,  Iteration: 0,  Instance: /test_bench,  Process: @INITIAL#19_0@.
# KERNEL: stopped at time: 8 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Fig 20: Simulation output for computing Givens Rotation

Thus, we get, $J_1 = \begin{bmatrix} 1 & 0.625 & 0.331125 & 0.6728 \\ 0.625 & 0.5 & 0.095978 & 0.078701 \\ 0.331125 & 0.095978 & -0.388039 & -0 \\ 0.6728 & 0.078701 & -0 & 1.497242 \end{bmatrix}$

2nd iteration:

- <u>Largest off-diagonal element computation:</u>

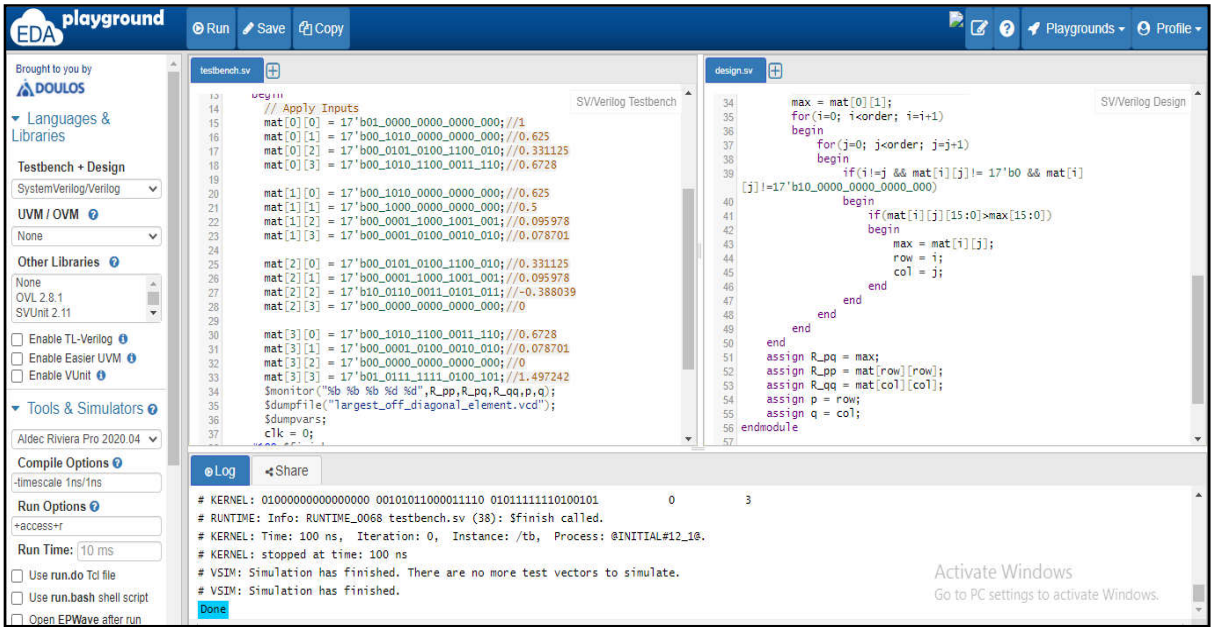
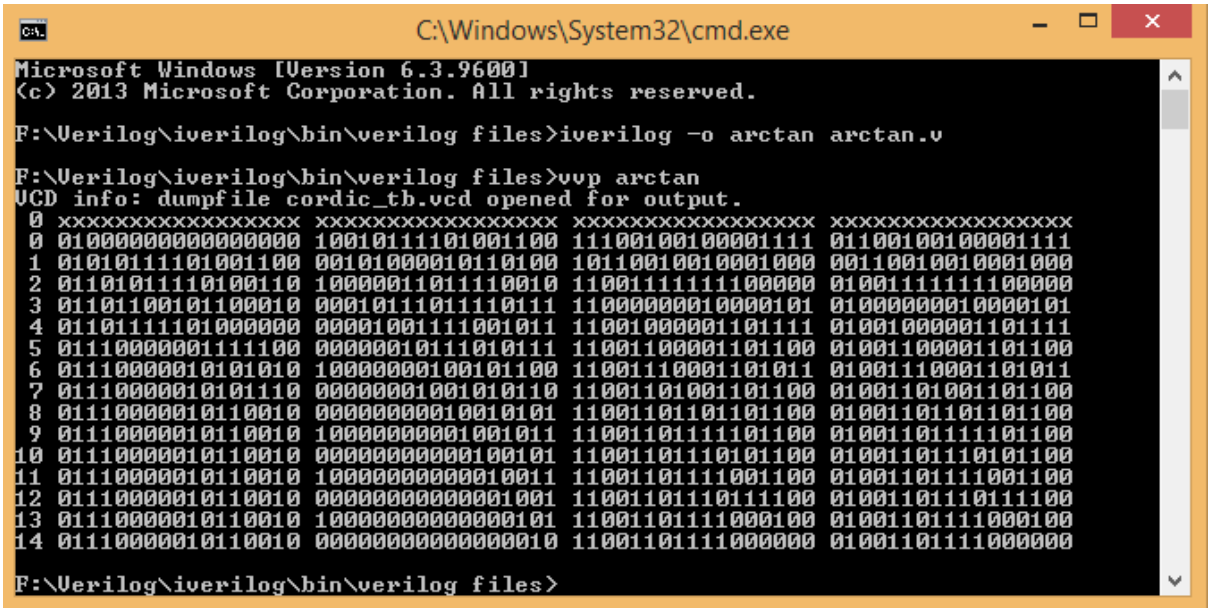


Fig 21: Simulation output for computing largest off-diagonal element

Thus, we get, p = 0, q = 3, R_pp = $(1.0)_2$ = 1.0, R_qq = $(1.011111110100101)_2$ = 1.497242 and R_pq = $(0.10101100001111)_2$ = 0.6728.

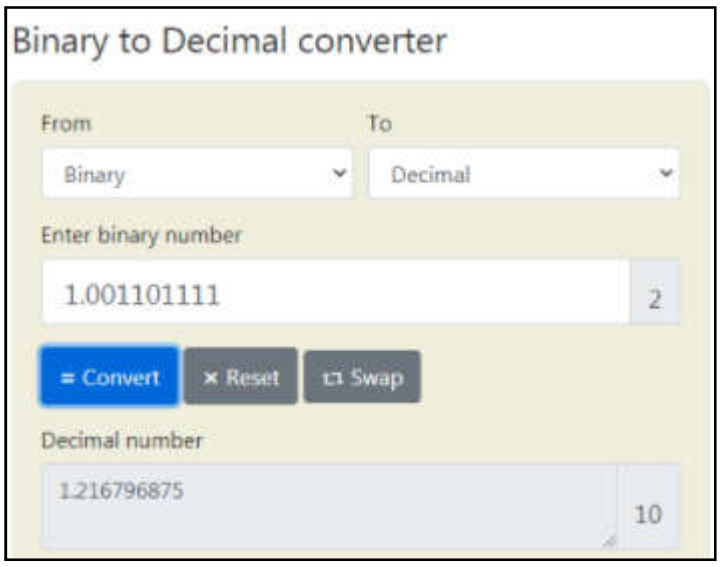
- <u>Arctan(x) computation:</u>

$x = \frac{2R_{pq}}{R_{qq}-R_{pp}} = 2.706.$



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

F:\Verilog\iverilog\bin\verilog files>iverilog -o arctan arctan.v

F:\Verilog\iverilog\bin\verilog files>vvp arctan
VCD info: dumpfile cordic_tb.vcd opened for output.
 0 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 0100000000000000 1001011110100110 0 1110010010000111 1 0110010010000111
 1 0101011110100110 0010100001011010 0 1011001001000100 0 0011001001000100
 2 0110101111010011 0 1000001101111001 0 1100111111110000 0 1001111111110000
 3 0110110010110001 0 0001011101111011 1 1000000001000010 1 0100000001000010
 4 0110111110100000 0 0000100111100101 1 1100100000110111 1 0100100000110111
 5 0111000000111110 0 0000001011101011 1 1100110000110110 0 1001100000110110
 6 0111000000101010 1 0000000100101100 1 1100111000110101 0 1001110000110101
 7 0111000000101110 0 0000000100101011 0 1100110100110110 0 1001101001101100
 8 0111000000101100 0 0000000001001010 1 1100110110110110 0 1001101101101100
 9 0111000000101100 1 0000000001001011 1 1100110111110110 0 1001101111110100
10 0111000000101100 0 0000000000100101 1 1100110111101100 0 1001101111101100
11 0111000000101100 1 0000000000010011 1 1100110111100110 0 1001101111001100
12 0111000000101100 0 0000000000001001 1 1100110111011110 0 1001101110111100
13 0111000000101100 1 0000000000000101 1 1100110111100010 0 1001101111000100
14 0111000000101100 0 0000000000000010 1 1100110111100000 0 1001101111000000

F:\Verilog\iverilog\bin\verilog files>
```

Fig 22: Simulation output for computing arctan(x) in CORDIC



Thus, we get, $\tan^{-1}(x) = 1.2168$ radian.

- <u>cos(θ) and sin(θ) computation:</u>

  $\theta = \frac{1}{2}\tan^{-1}(x) = 0.6084$ radian.

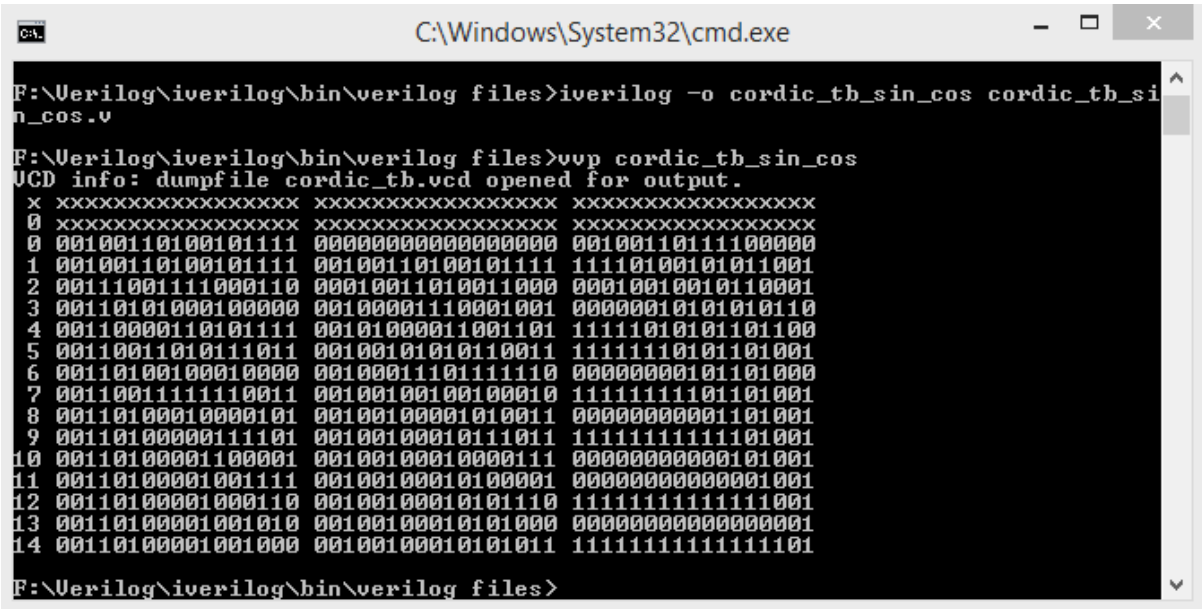
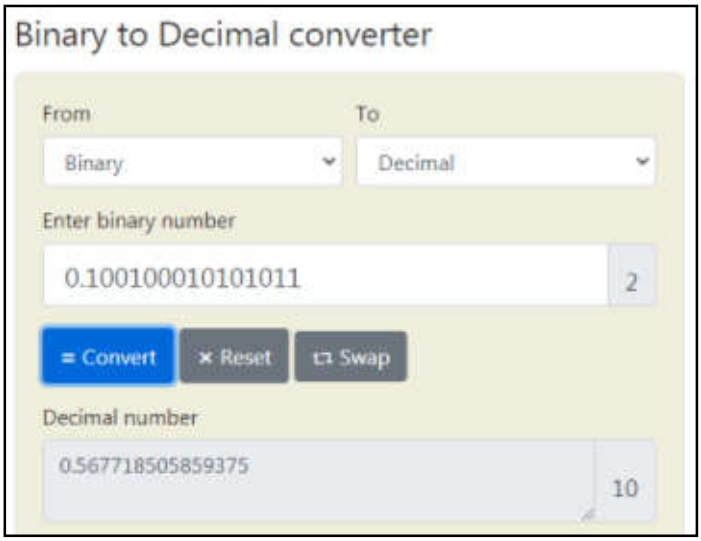
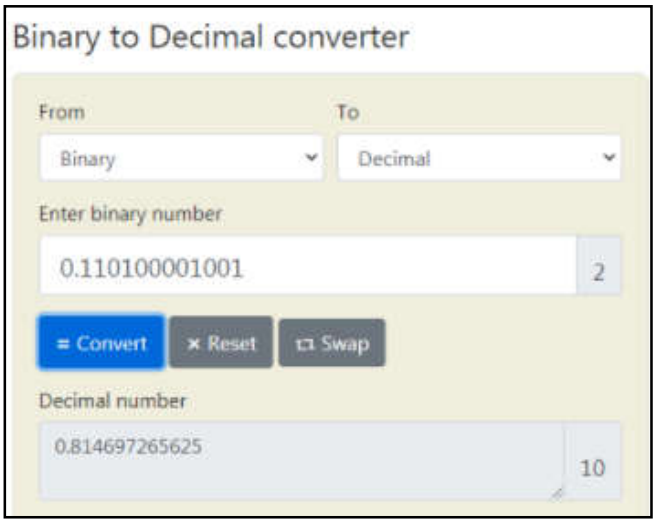


Fig 23: Simulation output for computing cos(θ) and sin(θ) in CORDIC



Thus, we get, sin(θ) = 0.5677.



Thus, we get, cos(θ) = 0.8147.

- <u>Givens Rotation:</u>

```
Log    Share
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4677 kB (elbread=427 elab2=4115 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: 0.523934 0.464506 0.269868 -0.000271
# KERNEL: 0.464506 0.500000 0.095978 0.418942
# KERNEL: 0.269767 0.095978 -0.388039 0.187986
# KERNEL: -0.000271 0.418942 0.188057 1.938436
# RUNTIME: Info: RUNTIME_0068 testbench.sv (45): $finish called.
# KERNEL: Time: 8 ns,  Iteration: 0,  Instance: /test_bench,  Process: @INITIAL#18_0@.
# KERNEL: stopped at time: 8 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Fig 24: Simulation output for computing Givens Rotation

Thus, we get, $J_2 = \begin{bmatrix} 0.523934 & 0.464506 & 0.269868 & -0 \\ 0.464506 & 0.5 & 0.095978 & 0.418942 \\ 0.269767 & 0.095978 & -0.388039 & 0.187986 \\ -0 & 0.418942 & 0.188057 & 1.938436 \end{bmatrix}$

3rd iteration:

- <u>Largest off-diagonal element computation:</u>

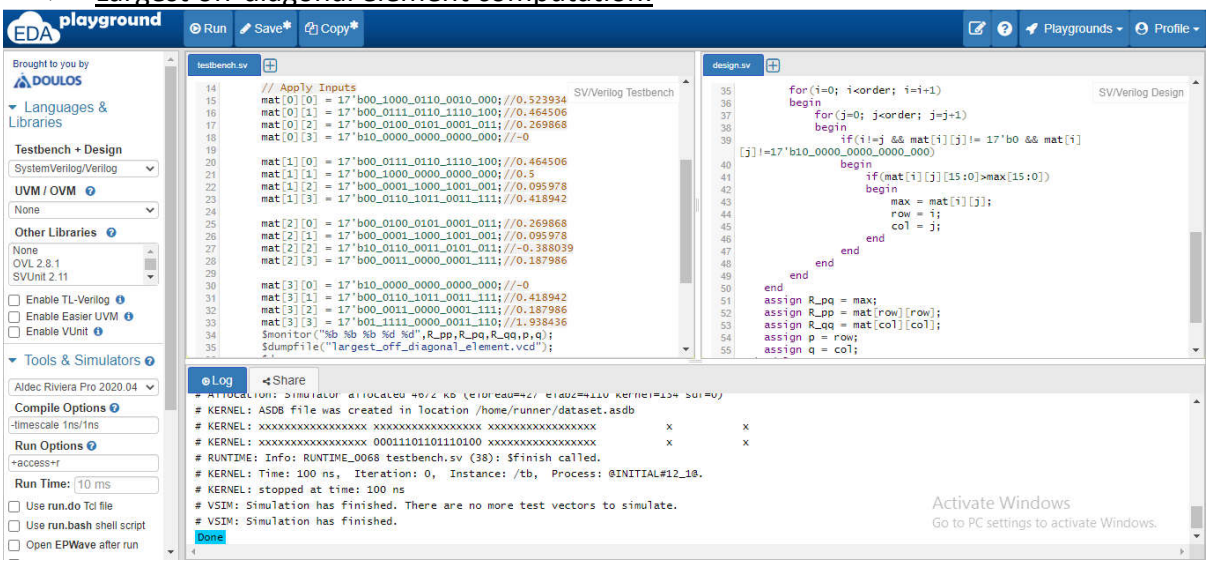
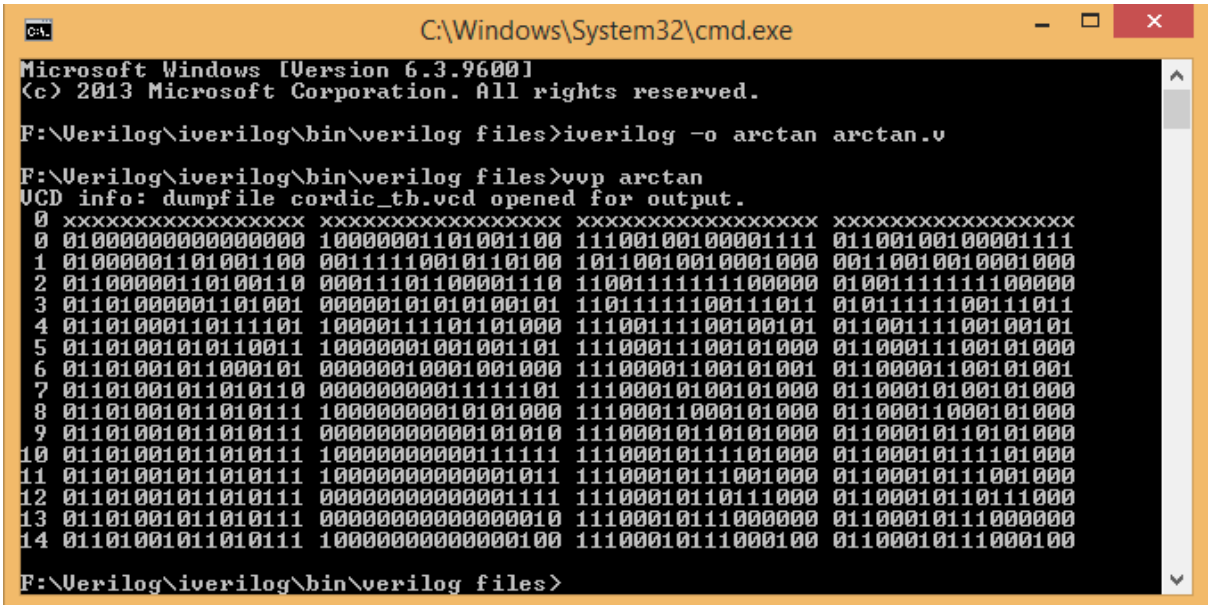


Fig 25: Simulation output for computing largest off-diagonal element

Thus, we get, p = 0, q = 1, R_pp = $(0.10000110001)_2$ = 0.523934, R_qq = $(0.1)_2$ = 0.5 and R_pq = $(0.0111011011101)_2$ = 0.464506.

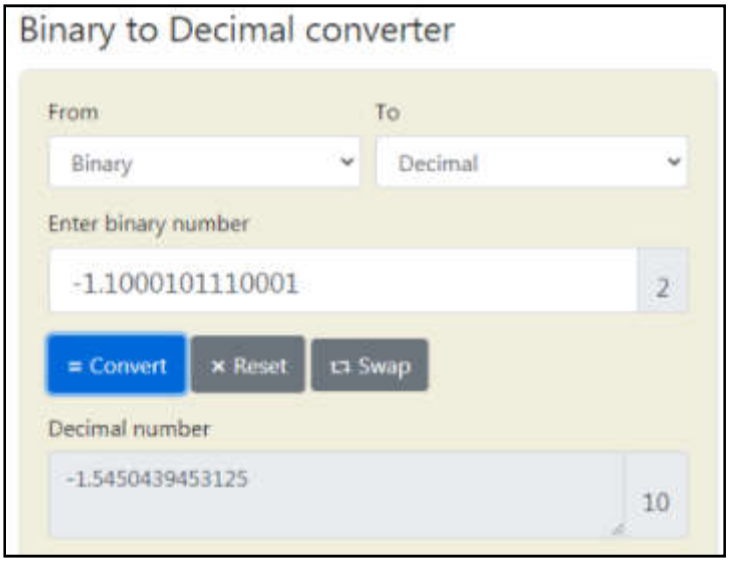
- <u>Arctan(x) computation:</u>

$x = \frac{2R_{pq}}{R_{qq}-R_{pp}}$ = -38.8156.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

F:\Verilog\iverilog\bin\verilog files>iverilog -o arctan arctan.v

F:\Verilog\iverilog\bin\verilog files>vvp arctan
VCD info: dumpfile cordic_tb.vcd opened for output.
 0 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 0100000000000000 1000000110100110 0 111001001000011 11 0110010010000111 1
 1 0100000110100110 0011111001011010 0 101100100100010 00 0011001001000100 0
 2 0110000011010011 0000110110000111 0 110011111111000 00 0100111111110000 0
 3 0110100000110100 1 0000010101010010 1 1101111110011101 1 0101111110011101 1
 4 0110100011011110 1 0000111101101000 1 1110011110010010 1 0110011110010010 1
 5 0110100101011001 1 0000000100100110 1 1110001110010100 0 0110001110010100 0
 6 0110100101100010 1 0000001000100100 0 1110000110010100 1 0110000110010100 1
 7 0110100101101011 0 0000000001111110 1 1110001010010100 0 0110001010010100 0
 8 0110100101101011 1 0000000001010100 0 1110001100010100 0 0110001100010100 0
 9 0110100101101011 0 0000000000010101 0 1110001011010100 0 0110001011010100 0
10 0110100101101011 1 0000000000011111 1 1110001011110100 0 0110001011110100 0
11 0110100101101011 1 0000000000000101 1 1110001011100100 0 0110001011100100 0
12 0110100101101011 0 0000000000000111 1 1110001011011100 0 0110001011011100 0
13 0110100101101011 0 0000000000000001 0 1110001011100000 0 0110001011100000 0
14 0110100101101011 1 0000000000000010 0 1110001011100010 0 0110001011100010 0

F:\Verilog\iverilog\bin\verilog files>
```

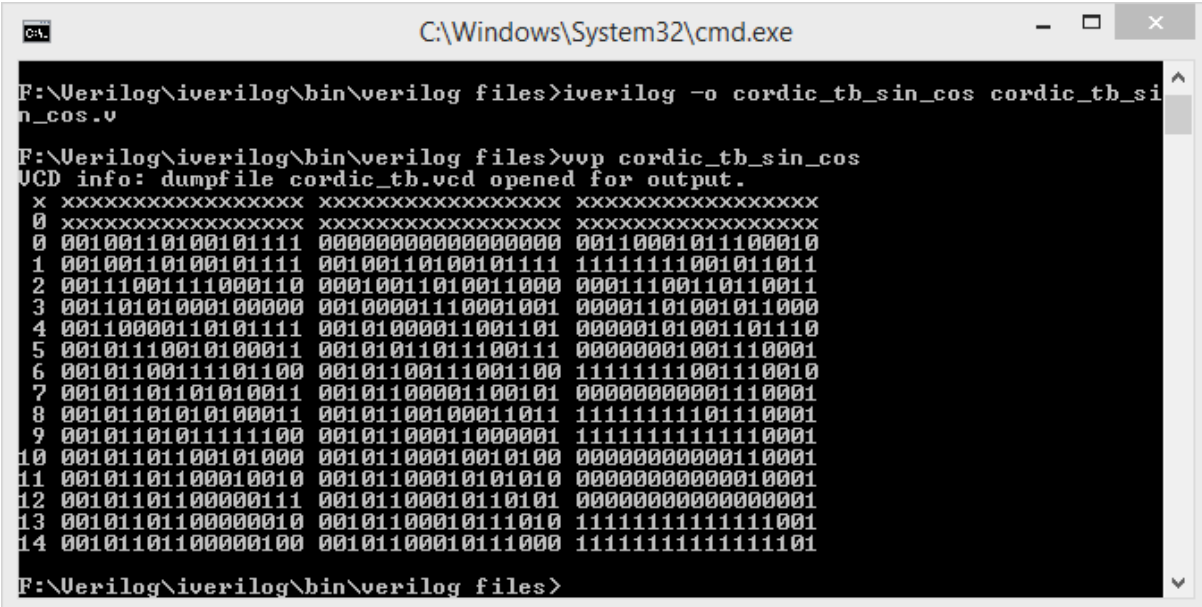
Fig 26: Simulation output for computing arctan(x) in CORDIC



Thus, we get, $\tan^{-1}(x) = -1.545$ radian.

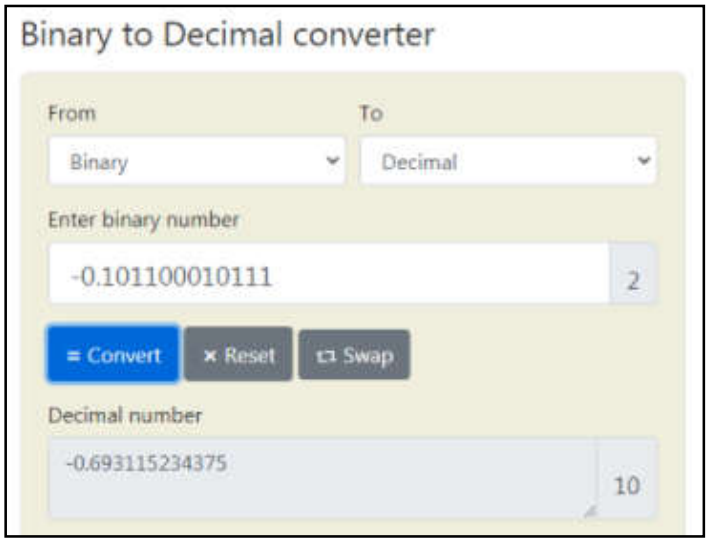
- <u>cos(θ) and sin(θ) computation:</u>

  $\theta = \frac{1}{2}\tan^{-1}(x) = -0.7725$ radian.

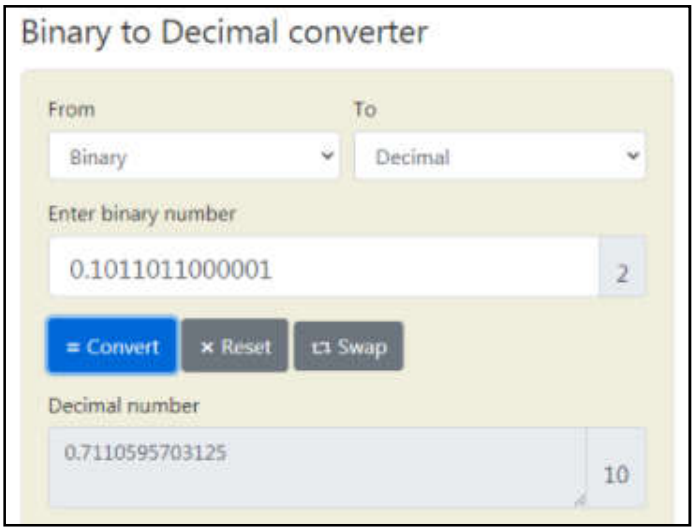


```
C:\Windows\System32\cmd.exe

F:\Verilog\iverilog\bin\verilog files>iverilog -o cordic_tb_sin_cos cordic_tb_si
n_cos.v

F:\Verilog\iverilog\bin\verilog files>vvp cordic_tb_sin_cos
VCD info: dumpfile cordic_tb.vcd opened for output.
 x xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
 0 0010011010010111 1 0000000000000000 0011000101110001 0
 1 0010011010010111 1 0010011010010111 1111111100101101 1
 2 0011100111100011 0 0001001101001100 0 0001110011011001 1
 3 0011010100010000 0 0010000111000100 1 0000110100101100 0
 4 0011000011010111 1 0010100001100110 1 0000010100110111 0
 5 0010111001010001 1 0010101101110011 1 0000000100111000 1
 6 0010110011110110 0 0010110011100110 0 1111111100111001 0
 7 0010110110101001 1 0010110000110010 1 0000000000111000 1
 8 0010110101010001 1 0010110010001101 1 1111111110111000 1
 9 0010110101111110 0 0010110001100000 1 1111111111110000 1
10 0010110110010100 0 0010110001001010 0 0000000000011000 1
11 0010110110001001 0 0010110001010101 0 0000000000001000 1
12 0010110110000011 1 0010110001011010 1 0000000000000000 1
13 0010110110000001 0 0010110001011101 0 1111111111111100 1
14 0010110110000010 0 0010110001011100 0 1111111111111110 1

F:\Verilog\iverilog\bin\verilog files>
```

Fig 27: Simulation output for computing cos(θ) and sin(θ) in CORDIC



Thus, we get, sin(θ) = -0.69312.



Thus, we get, cos(θ) = 0.71106.

- <u>Givens Rotation:</u>

```
Log    Share
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: 0.962968 -0.000092 0.258416 0.290375
# KERNEL: -0.000092 0.046645 -0.118804 0.297893
# KERNEL: 0.258416 -0.118804 -0.388039 0.187986
# KERNEL: 0.290375 0.297893 0.187986 1.938436
# RUNTIME: Info: RUNTIME_0068 testbench.sv (46): $finish called.
# KERNEL: Time: 8 ns,  Iteration: 0,  Instance: /test_bench,  Process: @INITIAL#19_0@.
# KERNEL: stopped at time: 8 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Fig 28: Simulation output for computing Givens Rotation

Thus, we get, $J_3 = \begin{bmatrix} 0.962968 & -0 & 0.258416 & 0.290375 \\ -0 & 0.046645 & -0.118804 & 0.297893 \\ 0.258416 & -0.118804 & -0.388039 & 0.187986 \\ 0.290375 & 0.297893 & 0.187986 & 1.938436 \end{bmatrix}$

Here, the principal diagonal elements are 0.962968, 0.046645, -0.388039 and 1.938436.
With further iterations, these elements converge to 0.9506, 0.0354, -0.4874 and 2.1264 respectively.
Therefore, λ_{B_norm} = 0.9506, 0.0354, -0.4874, 2.1264
⇨ $\lambda_B = 16*\lambda_{B\_norm}$ = 15.2096, 0.5669, -7.7989, 34.0224.

- **Extracting eigenvalues of the skew-symmetric matrix by Eigensystem Computation Method in MATLAB:**

$$C = \begin{bmatrix} 0 & -2 & -5 & 1 \\ 2 & 0 & -7 & 9 \\ 5 & 7 & 0 & 2 \\ -1 & -9 & -2 & 0 \end{bmatrix}$$

- <u>Triadiagonal Matrix Computation by TRIZD subroutine:</u>

Variables - c_tri

c_tri

4x4 double

	1	2	3	4	5
1	0	-2	-5	1	
2	7.3326	5.1850	-7	9	
3	-3.6697	-15.1826	11.0449	2	
4	-1	-9	-11.2736	10.2248	
5					

Fig 29: TRIZD subroutine simulation output for computing tridiagonal matrix

➢ <u>Eigenvalue Computation by IMZD subroutine:</u>

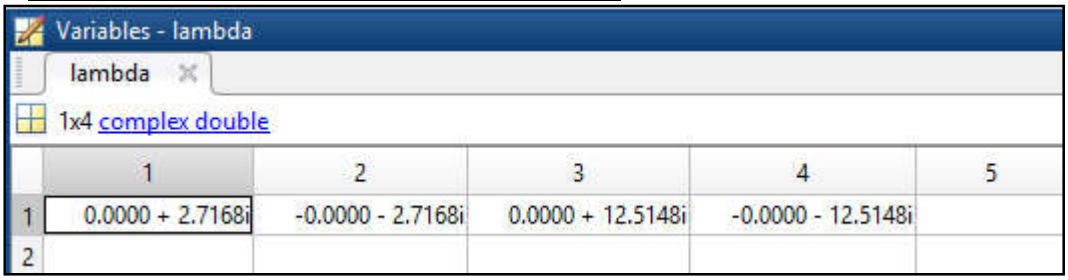


Fig 30: IMZD subroutine simulation output for computing eigenvalues

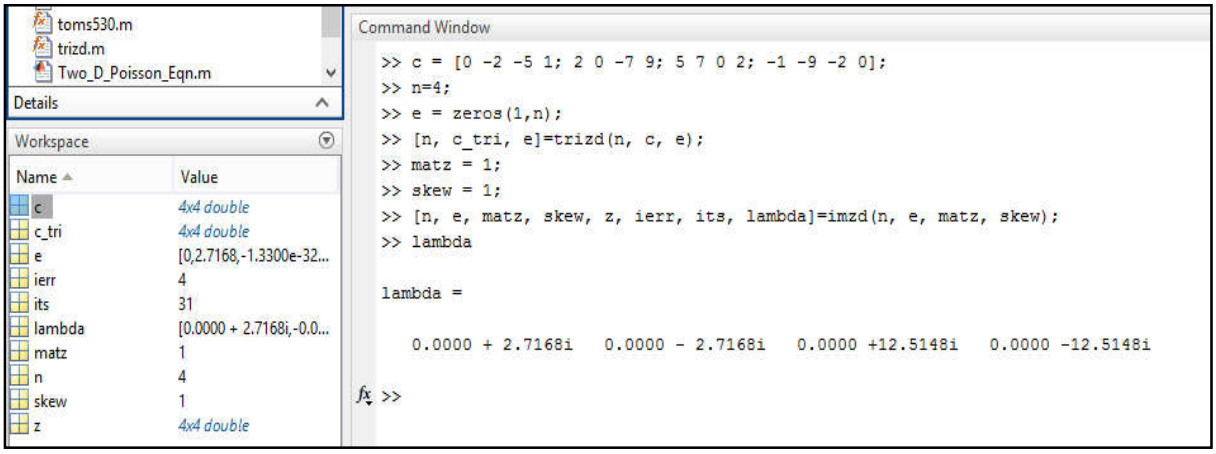


Fig 31: Simulation output for Eigensystem Computation Algorithm in MATLAB

Therefore, $\lambda_C = \pm 2.7168i,\ \pm 12.5148i$.

➢ Eigenvalue computation for the input matrix from the eigenvalues of the corresponding symmetric and skew-symmetric matrices:

$\lambda_A = f(\lambda_B, \lambda_C) = a_1 \lambda_B + a_2 \lambda_C$ (let)

a_1	λ_B	a_2	λ_C	λ_A
-2.2485	0.5669	0.5127	12.5148i	-1.2747 +6.4168i
0.1634	-7.7989	0.5127	-12.5148i	-1.2747 -6.4168i
0.7928	15.2096	0	2.7168i	12.0575
0.955	34.0224	0	-2.7168i	32.4918

Table 1: Computation of eigenvalues of non-symmetric matrix from that of symmetric and skew-symmetric matrices

Thus, $\lambda_A = -1.2747 \pm 6.4168i$, 12.0575, 32.4918.

Direction of Angle of Arrival (DoA) Estimation

5.1 Problem Statement: ESPRIT algorithm to predict the angles of arrival for

an M = 4 element array where noise variance σ^2_n = .1. Approximate the correlation matrices by time averaging over K = 300 data points.

5.2 Solution:

Solution The signal subspace for the entire ideal ULA array correlation matrix is given by

$$\overline{E}_x = \begin{bmatrix} .78 & .41+.1i \\ .12-.02i & .56 \\ -.22+.07i & .54-.05i \\ -.51+.25i & .46-.1i \end{bmatrix}$$

The two subarray signal subspaces can now be found by taking the first three rows of $\overline{E}_x$ to define $\overline{E}_1$ and the last three rows of $\overline{E}_x$ to define $\overline{E}_2$

$$\overline{E}_1 = \begin{bmatrix} .78 & .41+.1i \\ .12-.02i & .56 \\ -.22+.07i & .54-.05i \end{bmatrix} \qquad \overline{E}_2 = \begin{bmatrix} .12-.02i & .56 \\ -.22+.07i & .54-.05i \\ -.51+.25i & .46-.1i \end{bmatrix}$$

Constructing the matrix of signal subspaces we get

$$\overline{C} = \begin{bmatrix} \overline{E}_1^H \\ \overline{E}_2^H \end{bmatrix} [\overline{E}_1 \ \overline{E}_2] = \begin{bmatrix} .67 & .26+.06i & .2-.03i & .39 \\ .26-.06i & .78 & -.37+.12i & .78-.11i \\ .2+.03i & -.37-.12i & .4 & -.32-.08i \\ .39 & .78+.11i & -.32+.08i & .82 \end{bmatrix}$$

Performing the eigendecomposition we can construct the matrix $\overline{E}_C$ such that

$$\overline{E}_C = \begin{bmatrix} \overline{E}_{11} & \overline{E}_{12} \\ \overline{E}_{21} & \overline{E}_{22} \end{bmatrix} = \begin{bmatrix} .31 & .8 & -.44+.1i & -.22+.11i \\ .63-.09i & -.16 & .45 & -.61-.03i \\ -26-.05i & .57+.1i & .75 & .15-.11i \\ .66 & .01+.02i & .07-.17i & .73 \end{bmatrix}$$

We can now calculate the rotation operator $\overline{\Psi} = -\overline{E}_{12}\overline{E}_{22}^{-1}$ given the rotation matrix

$$\overline{\Psi} = -\overline{E}_{12}\overline{E}_{22}^{-1} = \begin{bmatrix} .58-.079i & .2-.05i \\ -.67+.23i & .94-.11i \end{bmatrix}$$

Next we can calculate the eigenvalues of $\overline{\Psi}$ and solve for the angles of arrival using

$$\theta_1 = \sin^{-1}\left(\frac{\arg(\lambda_1)}{kd}\right) = -4.82^\circ$$

$$\theta_2 = \sin^{-1}\left(\frac{\arg(\lambda_2)}{kd}\right) = 9.85^\circ$$

Future Scope of Improvement

➢ ***6.1 Computation of complex eigenvalues of a complex matrix***

As mentioned in the Jacobi Eigen Value algorithm, Jacobi method is only applicable for a real-symmetric matrix. However, we can convert every n by n matrix with complex entries into a 2n by 2n matrix with all real entries in the following manner.

Let us consider a 2X2 complex matrix:

$$A = \begin{bmatrix} a_{11} + jb_{11} & a_{12} + jb_{12} \\ a_{21} + jb_{21} & a_{22} + jb_{22} \end{bmatrix}$$

Now, this matrix can be represented as a 4X4 real matrix as shown below:

$$A = \begin{bmatrix} a_{11} & -b_{11} & a_{12} & -b_{12} \\ b_{11} & a_{11} & b_{12} & a_{12} \\ a_{21} & -b_{21} & a_{22} & -b_{22} \\ b_{21} & a_{21} & b_{22} & a_{22} \end{bmatrix}$$

Let us verify that the above two representations have the same eigenvalues by taking an example.

$$P = \begin{bmatrix} 1 & j \\ j & 1 \end{bmatrix} \text{ and } Q = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Solving, $|P - \lambda_P I_2| = 0$

we get, $\lambda_P = 1 + j, 1 - j$.

Again, $|Q - \lambda_Q I_4| = 0$

we get, $\lambda_Q = 1+ j, 1 + j, 1 - j, 1 - j$.

So, both the matrices have the same eigenvalues.

However, in our Verilog Implementation, we can only extract the real eigenvales of the complex matrix and not the complex ones. Thus, we intend to work towards resolving this issue in future.

➢ ***6.2 Computation of all eigenpairs of a general matrix***

As the Jacobi Eigen Value algorithm is only applicable for symmetric matrices and the Eigensystem Computation algorithm proposed by R.C. Ward and L. J. Gray is only developed for skew-symmetric matrices and symmetric matrices with constant (zero) diagonal, we aspire to develop a technique that would perform eigenvalue decomposition for any general matrix.

Since the eigenvalues of a triangular matrix are its diagonal elements, for general matrices there is no finite method like gaussian elimination to convert a matrix to triangular form while preserving eigenvalues. But it is possible to reach something close to triangular. An upper Hessenberg matrix is a square matrix for which all entries below the subdiagonal are zero. A lower Hessenberg matrix is one for which all entries above the superdiagonal are zero. Matrices that are both upper and lower Hessenberg are tridiagonal. Several methods are commonly used to convert a general matrix into a Hessenberg matrix with the same eigenvalues. **Givens Rotation** is one such efficient technique to convert any general matrix into an Hessenberg matrix, having time complexity $4n^3/3 + O(n^2)$. Hessenberg and tridiagonal matrices are the starting points for many eigenvalue algorithms because the zero entries reduce the complexity of the problem. Iterative algorithms solve the eigenvalue problem by producing sequences that converge to the eigenvalues. Some algorithms also produce sequences of vectors that converge to the eigenvectors. Mostly, the eigenvalue sequences are expressed as sequences of similar matrices which converge to a triangular or diagonal form, allowing the eigenvalues to be read easily. The eigenvector sequences are expressed as the corresponding similarity matrices. **QR Algorithm** is one such efficient iterative algorithm which is applicable for Hessenberg matrices and correctly computes all eigenvalues and eigenvector pairs, having time complexity $O(n^2)$ and $4n^3/3 + O(n^2)$ respectively. Thus, we intend to implement the combination of the above mentioned algorithms in order to compute all eigenpairs of any general matrix.

References

1. Jing Yan, Yangqing Huang, Hantao Xu and Guy A. E Vandenbosch, “Hardware Acceleration of MUSIC Based DoA Estimator in MUBTS”, EuCAP 2014.
2. Abdulrahman Alhamed, Nizar Tayem, Tariq Alshawi, Saleh Alshebeili, Abdullah Alsuwailem, Ahmed Hussain, “FPGA-based Real Time Implementation for Direction-of-Arrival Estimation”, Submitted, The Journal of Engineering, IET.
3. Ahmed A. Hussain, Nizar Tayem, Abdel-Hamid Soliman, Redha M. Radaydeh, “FPGA-based Hardware Implementation of Computationally Efficient Multi-Source DOA Estimation Algorithms”, Submitted, IEEE Access.
4. Frank B. Gross, “Smart Antennas for Wireless Communications with MATLAB”, Mc-Graw Hill Publications.
5. Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, Koushik Maharatna, “50 Years of CORDIC: Algorithms, Architectures, and Applications”, IEEE Transactions on Circuits and Systems, vol. 56, no. 9, Sept. 2009.
6. Huijie Zhu, Yizhou Ge, Bin Jiang, “Modified CORDIC Algorithm for Computation of Arctangent with Variable Iterations”, ICSP, 2016.
7. P. Venkata Rao, K. R. K. Sastry, “Implementation of Complex Matrix Inversion using Gauss-Jordan Elimination Method in Verilog”, International Journal of Computer Applications, vol. 122, no. 3, July, 2015.
8. M. H. C. Pardekooper, “An eigenvalue algorithm for skew-symmetric matrices“, Springer-Verlag, 1971.
9. R. C. Ward, L. C. Gray, “Eigensystem Computation for skew-symmetric matrices and a class of symmetric matrices“, Union Carbide Corporation 32, Computer Sciences Division, May, 1976.
10. Minseok Kim, Koichi Ichige, Hiroyuki Ami, “DESIGN OF JACOB1 EVD PROCESSOR BASED ON CORDIC FOR DOA ESTIMATION WITH MUSIC ALGORITHM”, PIMRC, 2002.
11. Marius Pesavento, Alex B. Gershman and Martin Haardt, “Unitary Root-MUSIC with a Real-Valued Eigendecomposition: A Theoretical and Experimental Performance Study”, IEEE Transactions on Signal Processing, vol. 48, no. 5, May, 2000.
12. J. Gotze, S. Paul, and M. Sauer, “An efficient Jacobi-like algorithm for parallel eigenvalue computation,” *IEEE Transactions on Computers*, vol. 42, no. 9, Sept. 1993.
13. Yang Liu, Christos-Savvas Bouganis, Peter Y. K. Cheung, “Hardware Efficient Architectures for Eigenvalue Computation”, IET Computers and Digital Techniques, vol. 3, no. 1, Feb. 2009.
14. Karthikeyan N, Sathyanarayanan S, Vamsi Krishna S, Shankar Balachandran, “FPGA implementation of Singular Value Decomposition“, 2012.