EE3510

RTL Design and Verification

ICA (Independent Component Analysis)

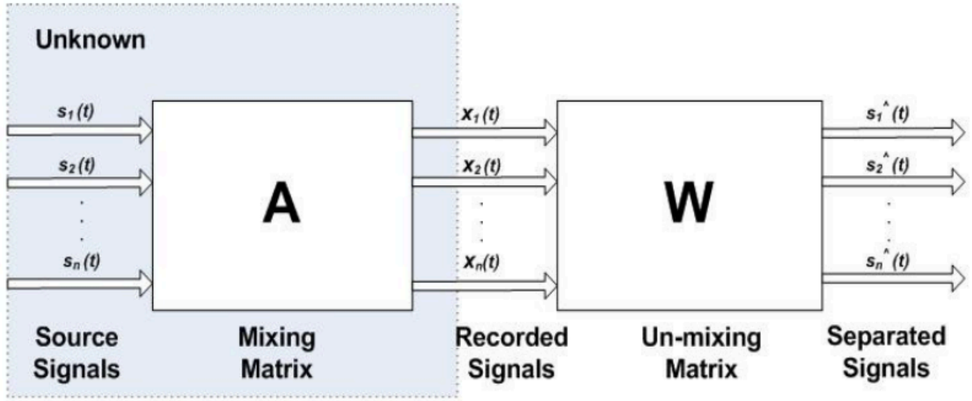
Devansh Srivatsava - EE22BTECH11207
Gargi Behera - EE22BTECH11208
Rayi Giri Varshini - EE22BTECH11215
Talasu Sowmith - EE22BTECH11218

Independent Component Analysis

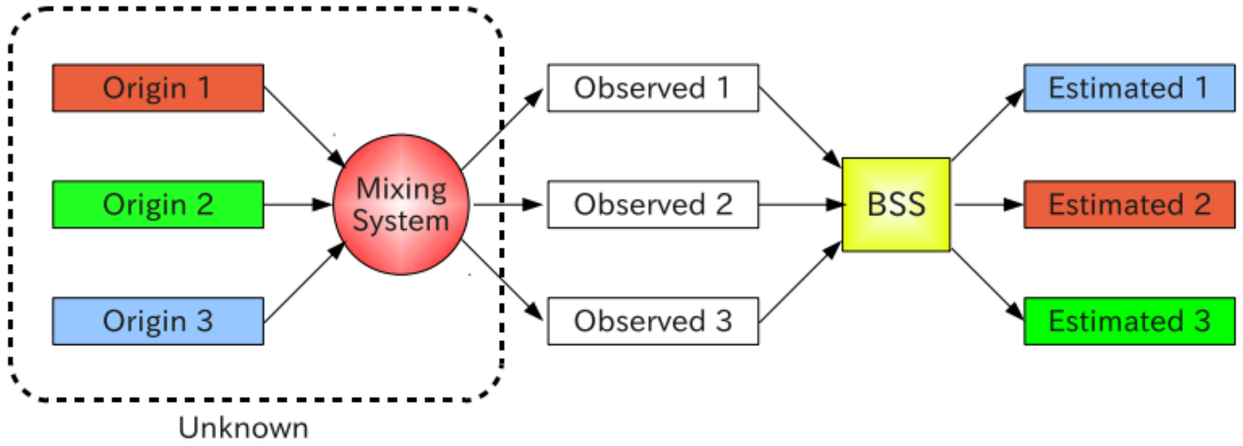
A computational technique used to separate a **multivariate signal** into additive, **independent components**. ICA, widely used for blind source separation extracting **individual signals from mixtures** by assuming different physical processes generate unrelated signals. It ranges from audio and image processing to biomedical signal analysis.

Blind Source Separation (BSS)

It is a process of separating a set of mixed signals into their original, independent sources without prior knowledge of how the signals were combined.

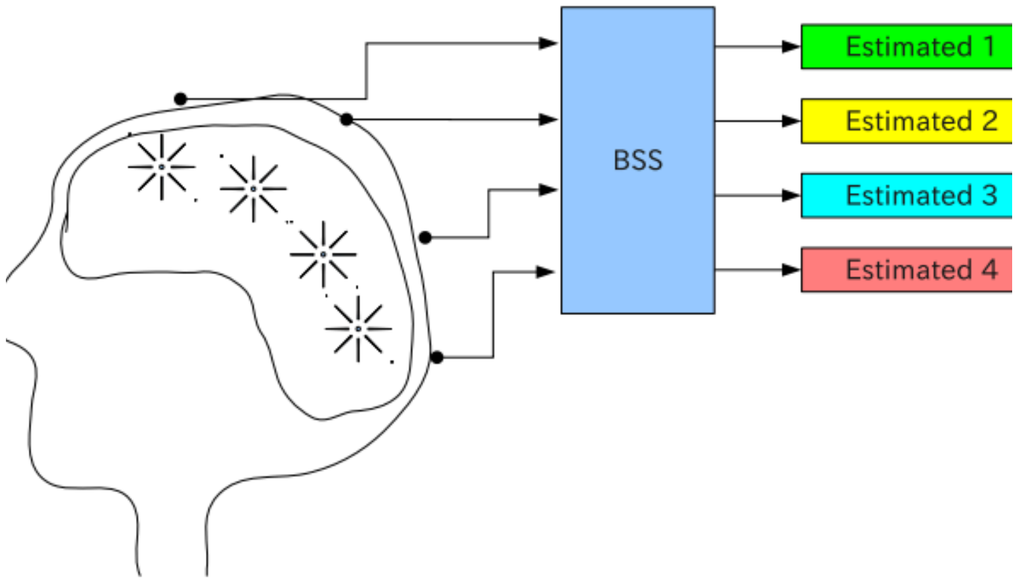


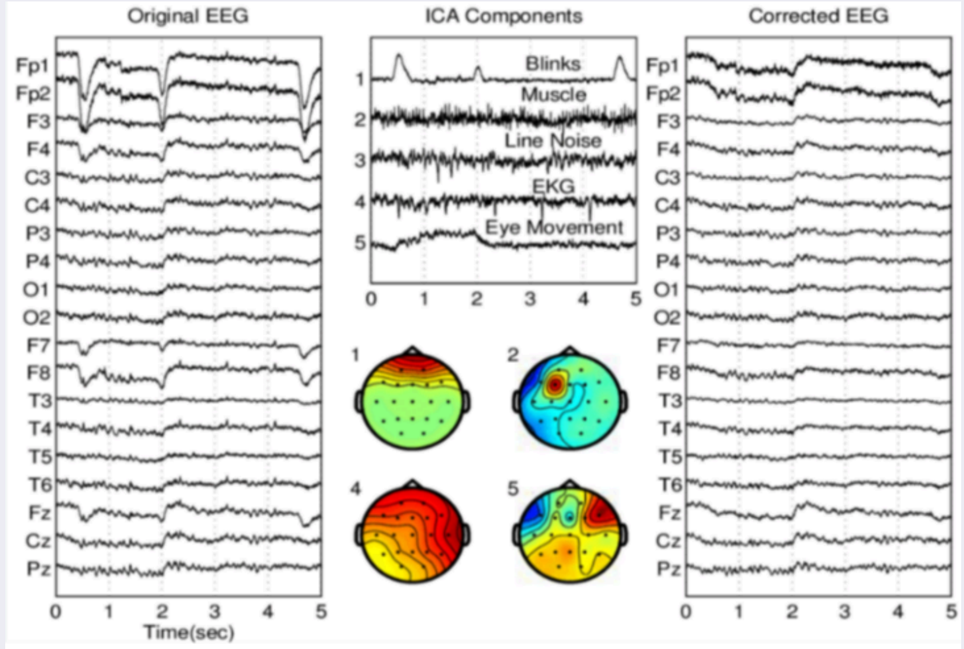
- Cocktail Party Problem
- ICA, PCA, SCA, NMF in BSS
- Indeterminacies in BSS
- Applications of BSS
- Challenges in BSS



Applications of ICA

- Cocktail Party Problem (BSS)
- EMG/MEG Analysis (Electromyography/Biomed)
- Signal Processing (Sensor, Audio & Image)
- Telecommunication & Data Analysis
- Feature Extraction

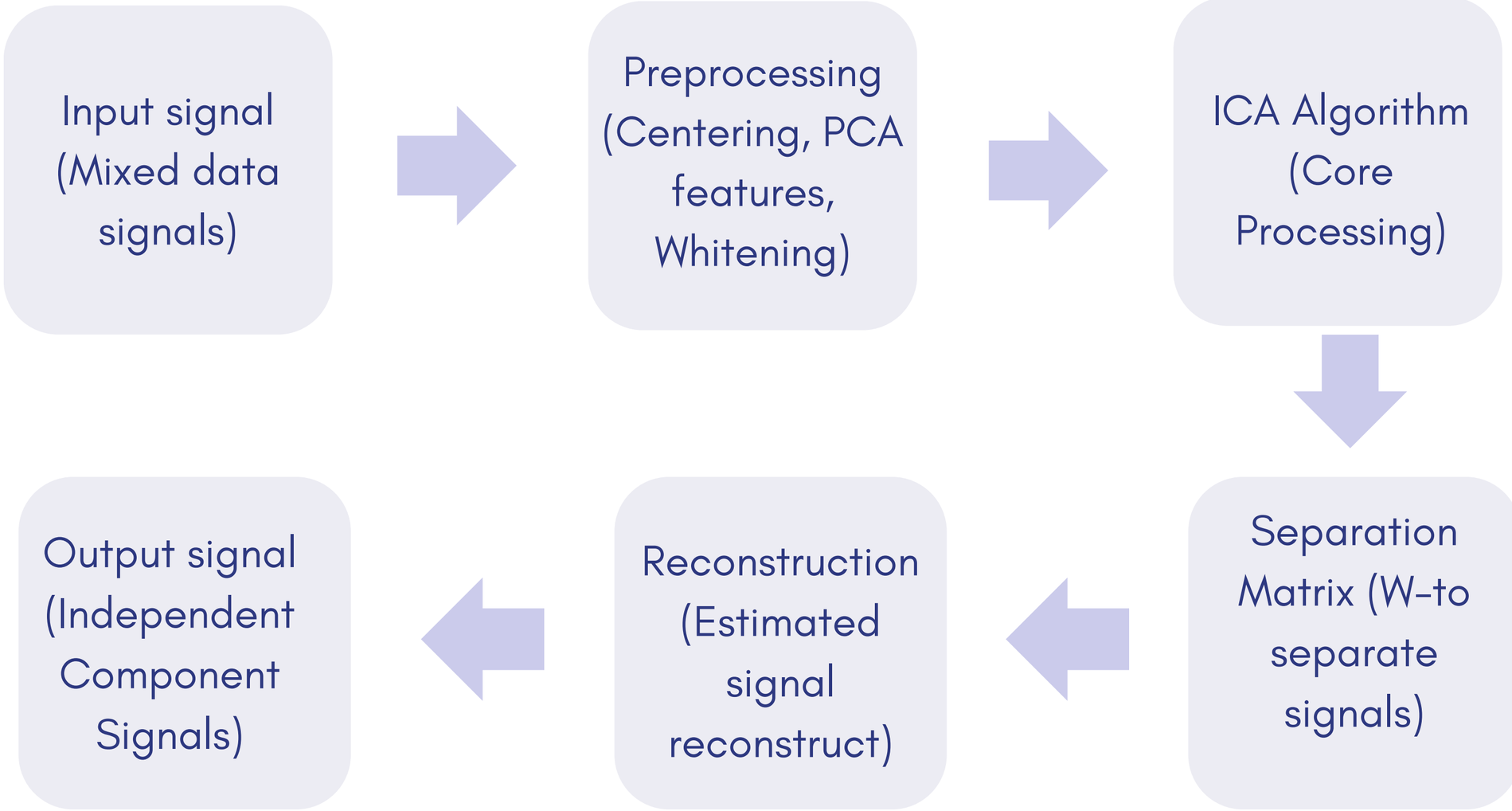




ICA used in EEG

Other applications include, data mining, machine fault detection, seismic monitoring, reflection cancelling, time series forecasting, radio communication, text document analysis etc.

ICA Block Diagram



Working of ICA

Assume that we observe n linear mixtures x1,...,xn of n independent components

xj = aj1.s1+aj2.s2+...+ajn.sn, for all j.

Let us denote by **x** the random vector whose elements are the mixtures x1,...,xn, and likewise by **s** the random vector with elements s1,...,sn. Let us denote by **A**(mixing matrix) the matrix with elements aij.

We can denote above mixing model as

x = As

By using ICA we can **compute inverse of the mixing** matrix A, say **W**(unmixing matrix), and obtain the independent component simply by:

s = Wx

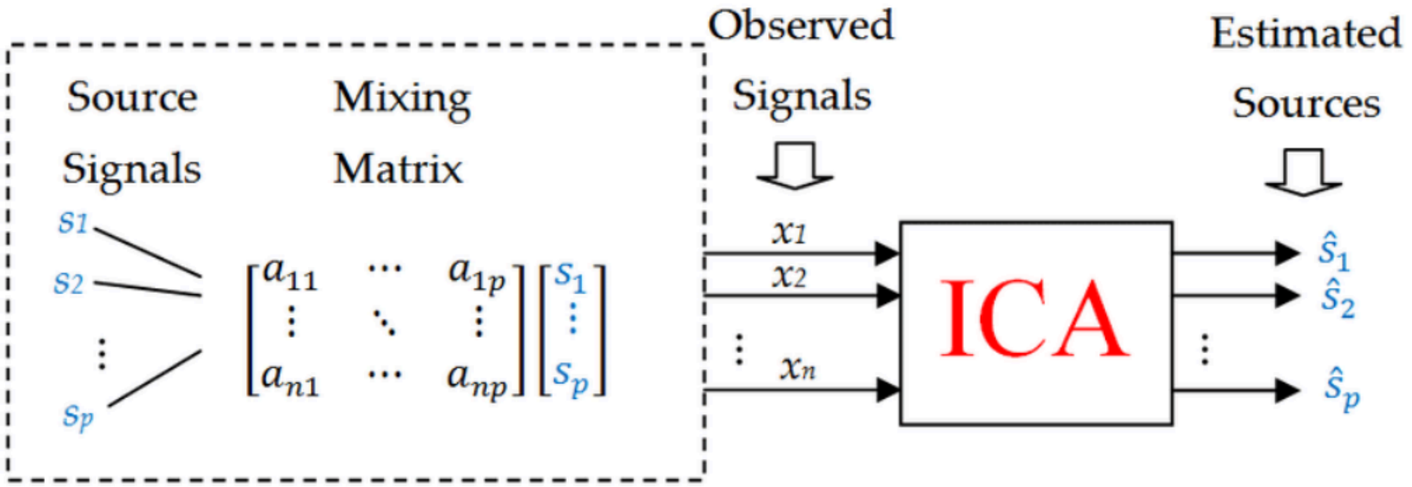
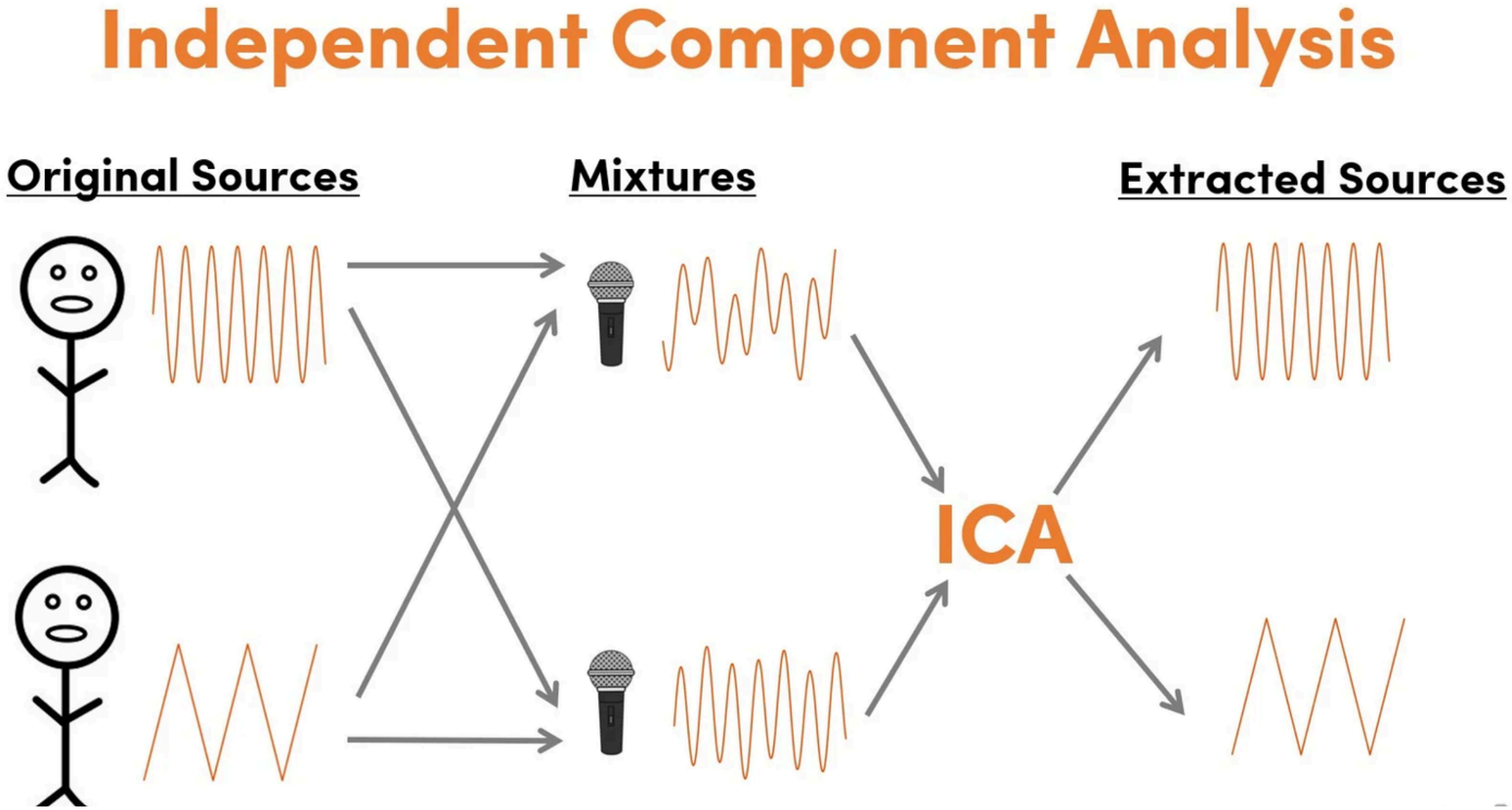


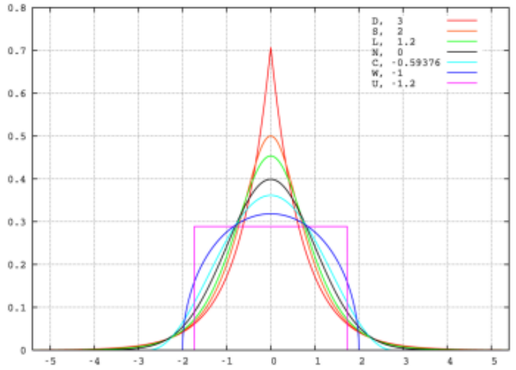
Illustration of ICA

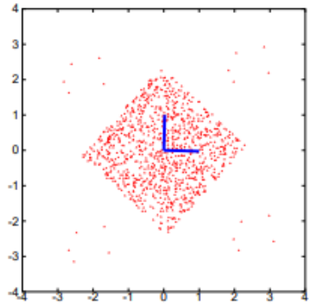


Assumptions in ICA

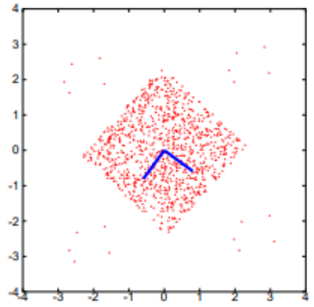
ICA assumes the source signals follow,

- Statistical Independence
- Non-Gaussian Distribution
- Linear Mixing
- Invertible Mixing Matrix
- No Noise (Ideal)





(a) Kurtosis based



(b) Neg-entropy based (using g_1)

ICA Modelling

Statistical Independence

When the joint PDF of signals equals the product of their marginal pdfs. So, knowing one signal gives no information about the other.

Non-Gaussianity Kurtosis

Kurtosis measures the spikiness of the distribution with positive values showing heavy tails (super-Gaussian) and negative values showing light tails (sub-Gaussian).

$$\text{kurt}(y) = \text{E}[y^4] - 3(\text{E}[y^2])^2$$

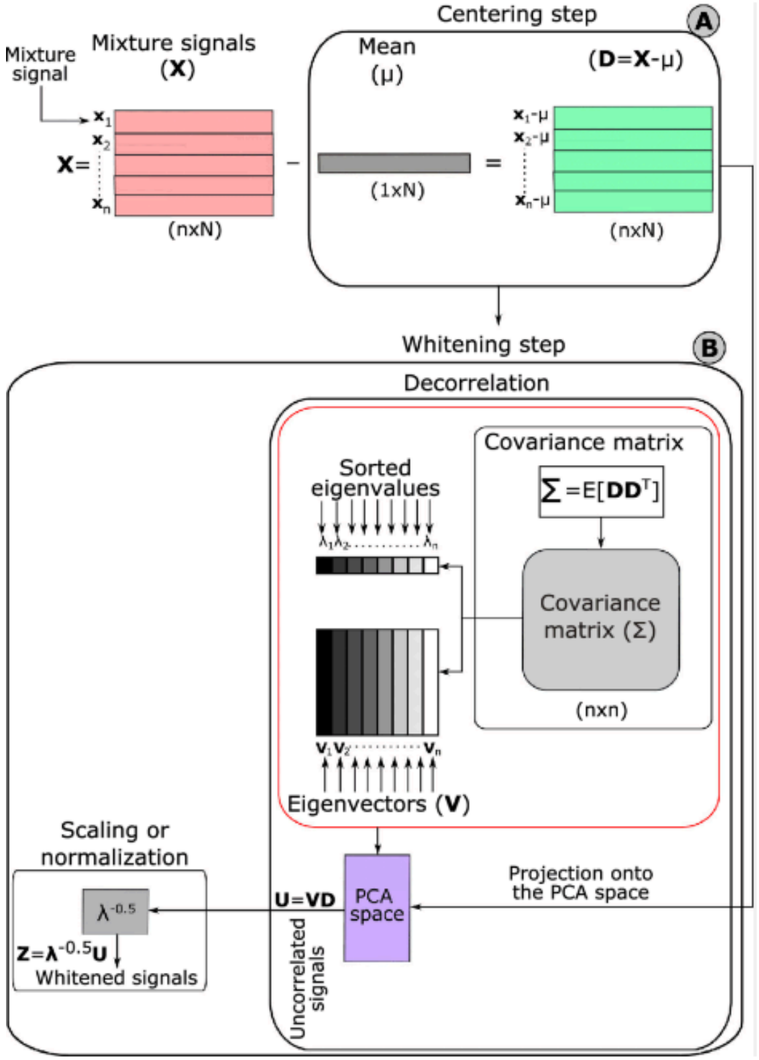
Non-Gaussinanity Neg-Entropy

It quantifies the deviation of a signal's entropy from that of a Gaussian distribution with the same covariance. As kurtosis is a fourth order function, which is weak with the outliners. Neg-entropy based ICA is robust for outliners.

Preprocessing Techniques

In ICA, preprocessing involves centering (removing the mean) and whitening (decorrelating and normalizing the data). This helps in stabilizing the algorithm to separate independent componenets efficiently.

- Whitening
- Centering



Whitening

It is the process of transforming observed signals such that their components become uncorrelated and each component has a unit variance.

White Signals satisfy E[x] = 0, and E[x.xT] = 1.

Centering

It is a process of subtracting the mean of the observed signals to make them zero-mean. This makes the ICA to focus on variance and higher order statistics of means rather than means.

ICA Algorithms

- Gradient Method
- Fast ICA
- Infomax

Gradient Method

It iteratively updates the separation matrix by minimizing a contrast function, based on non-Gaussianity. This process ensures the separation of mixed signals into independent components.

Fast ICA

It is an efficient algorithm that separates independent components by maximizing non-Gaussianity using fixed-point iterations, making it faster than gradient-based methods. It converges quickly without requiring step-size tuning. Kurtosis & Negentropy follow this.

Infomax

It maximizes the mutual information between the input signals and their outputs, separating independent components by finding a transformation that best fits the statistical independence of the sources.

Application of ICA in EMG

- ICA methods are used to estimate **MUAP** (motor unit action potentials).
- ICA reduces the root mean squared error of the monopolar EMG signals.
- ICA can be helpful in reducing the noise activity associated with the ECG caused by upper trunk muscles.
- Facial EMG data could be affected by the crosstalk from adjacent facial muscles.

Our goals for this Project

- Isolating the distinct signals generated by different muscles from the composite EMG signal by the usage of ICA
- Reduce noise and artifacts in EMG signals, improving clarity
- Enhance relevant signal features for better classification or control
- Reducing the complexity of the algorithm
- Lowering the Power consumption.

Normalization

Normalization is a major part of whitening in the preprocessing of a signal to send it to the ICA Algorithm, which ensures the input signals follow the statistical properties that facilitate the extraction of independent components. It ensures the data has zero mean and unit variance. Whitening then decorrelates the features, making them suitable for extracting statistically independent components.

Normalization of the vector is done by diving the vector with the norm of it.

$$\mathbf{v}_{\text{norm}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

For an N-dimensional vector, the Euclidean norm is given by,

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

This scales the vector to have unit modulus.

Why use Normalization?

For a system of n-independent sources, the mixed signal vectors are related to source signal vectors as X = AS after pre-processing, for V as co-variance matrix, Z is whitened signal, it becomes,

$$\mathbf{Z} = \mathbf{V}^{-1/2}\mathbf{X}$$

1. To maximize the non-gaussanity, weight vector is updated by kurtosis iteratively as,

$$\mathbf{w}_i^{\text{new}} = E\left[\mathbf{Z}\left(\mathbf{w}_i^T\mathbf{Z}\right)^3\right] - 3\mathbf{w}_i$$

2. To make the sources independent, Orthogonalization using Gram-Schmidt Algorithm is used to prevent multiple vectors to converge to same independent component. The computation is done using CORDIC

$$\mathbf{w_i^{new}} = \mathbf{w_i^{new}} - \sum_{i=0}^{i-1}\left(\mathbf{w_i}\cdot\mathbf{w_j^c}\right)\mathbf{w_j^c}$$

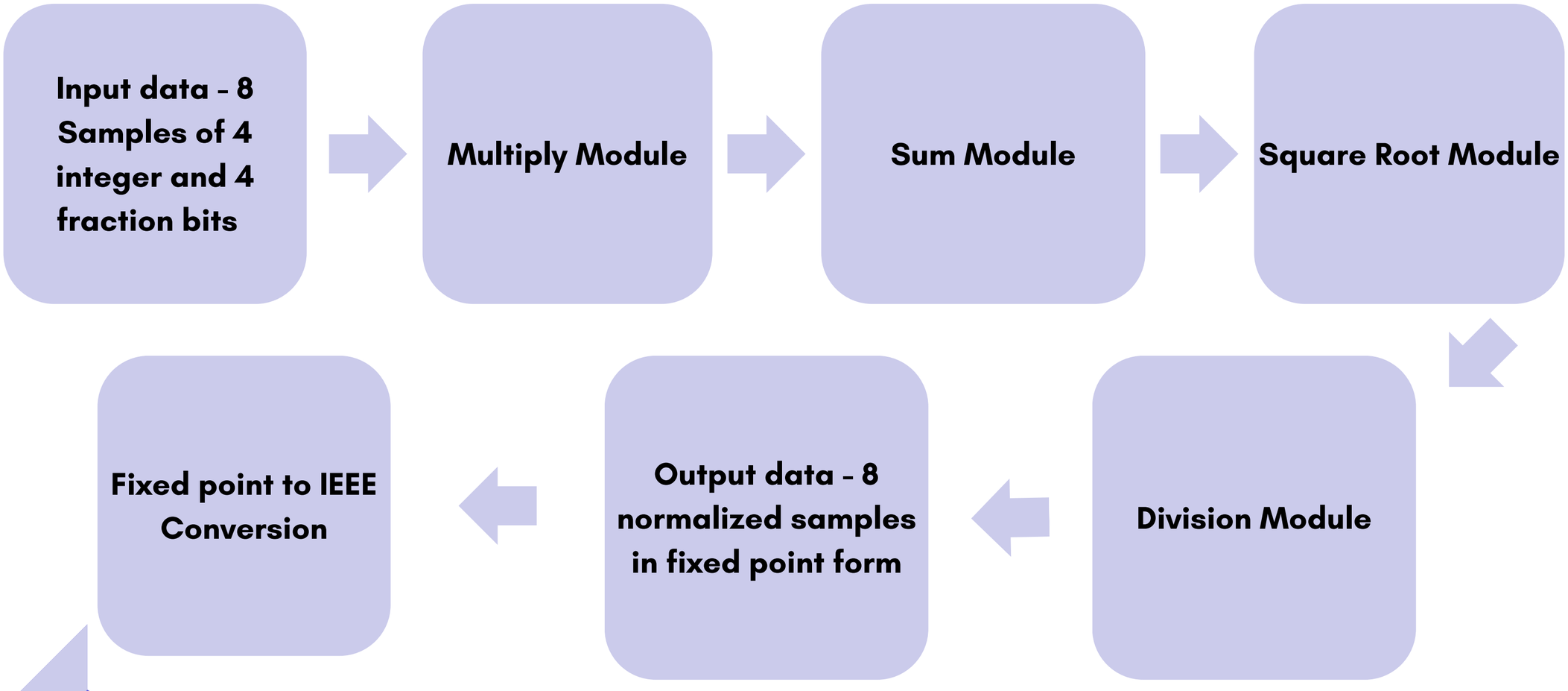
3. After orthogonalizing each vector the result is normalized using the formula

$$\mathbf{\underline{w}_i^{new}} = \frac{\mathbf{w_i^{new}}}{\left\|\mathbf{w_i^{new}}\right\|}.$$

Considerations and Assumptions

- The given sampled inputs are orthogonal i.e. each sample is along independent direction.
- The sampled data, where 8 samples are taken, in fixed-point representation of 8-bits per sample, with a split of 4 integer bits and 4 fractional bits.
- The data cannot be NULL, as it would lead to 0 norm and undefined normalized forms.
- CORDIC Algorithm for square root, multiplier and division blocks.
- If the results need to be converted into IEEE floating point format the IEEE conversion block can be used.

Block Diagram for Normalization



Process Overview

- **Input Vectors:** 8 sampled data points, each represented as 8-bit fixed-point numbers with 4 bits before the decimal point and 4 bits after. The samples are independent vector components (basically orthogonal).

- **Squaring the Samples:** Each sample is squared using a multiplication block. Squaring fixed-point numbers increases the bit-width to 16-bit outputs (8-bit integer and 8-bit fraction).

- **Summing the Squared Values:** The 8 squared samples are summed, which gives the square of norm. The sum will also be in fixed-point format with 19-bit (11-bit integer and 8-bit fraction).

- **Square Root Calculation:** The sum of squares is passed through a square root function to calculate the norm (magnitude of vector) which is, 14-bit (6-bit integer and 8-bit fraction-considering accuracy). For this CORDIC-based algorithm is used which is efficient for fixed-point arithmetic.

- **Division for Normalization:** Each original sample is divided by the calculated norm to normalize the vectors. The final result is in fixed-point format as is of 12-bit (0-bit integer and 12-bit fraction for precision, as dividing a sample with the norm is always less than 1).

- If required, **Converting** the final output to **IEEE 754 Floating-Point** can also be done by packing the result in sign, exponent and mantissa.

Multiply Module Code

testbench.sv

```
module tb_FixedPointMultiplier;
    reg [7:0] a;
    reg [7:0] b;
    wire [15:0] result;

    FixedPointMultiplier uut (
        .a(a),
        .b(b),
        .result(result)
    );

    initial begin
        // Test Case 1: 3.5 * 2.25
        a = 8'b00111000; // 3.5 in fixed-point
        b = 8'b00100010; // 2.25 in fixed-point
        #10;
        $display("3.5 * 2.25 = %b (Decimal: %f)", result, result/16.0);

        // Test Case 2: 1.5 * 4.0
        a = 8'b00011000; // 1.5 in fixed-point
        b = 8'b01000000; // 4.0 in fixed-point
        #10;
        $display("1.5 * 4.0 = %b (Decimal: %f)", result, result/16.0);
```

design.sv

```
module FixedPointMultiplier (
    input [7:0] a,
    input [7:0] b,
    output reg [15:0] result
);
    integer i;

    always @(*) begin
        result = 0;
        for (i = 0; i < 8; i = i + 1) begin
            if (b[i]) begin
                result = result + (a << i);
            end
        end
        result = result >> 4;
    end
endmodule
```

Log | Share

```
[2024-09-26 10:53:40 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
3.5 * 2.25 = 0000000001110111 (Decimal: 7.437500)
1.5 * 4.0 = 0000000001100000 (Decimal: 6.000000)
testbench.sv:25: $finish called at 20 (1s)
Done
```

For **fixed-point multiplication shift registers** are used.

Link for code, testbench and outputs: Multiplication module

Sqaure Root Module Code

```
module tb_square_root;

    reg [18:0] in;
    wire [18:0] out;

    square_root uut (
        .in(in),
        .out(out)
    );

    initial begin
        // Test case 1:
        in = 19'b00000100000101000;
        #10;
        $display("Input: %f, Output: %f", in / 256.0, out / 256.0);

        // Test case 2:
        in = 19'b00011001000100000;
        #10;
        $display("Input: %f, Output: %f", in / 256.0, out / 256.0);

        // Test case 3:
        in = 19'b00111101011000000;
        #10;
        $display("Input: %f, Output: %f", in / 256.0, out / 256.0);

        $finish;
    end
endmodule
```

```
module square_root (
    input wire [18:0] in,
    output reg [18:0] out
);

    reg [18:0] low;
    reg [18:0] high;
    reg [18:0] mid;
    reg [37:0] mid_square;
    reg [37:0] fixed_input;

    task multiply;
        input [18:0] a;
        input [18:0] b;
        output reg [37:0] result;
        integer i;
        begin
            result = 0;
            for (i = 0; i < 19; i = i + 1) begin
                if (b[i]) begin
                    result = result + (a << i);
                end
            end
        end
    endtask

    always @(*) begin
        low = 0;
        high = 19'b111111111100000000;
        out = 0;
        fixed_input = in << 8;

        while (low <= high) begin
            mid = (low + high) >> 1;
            multiply(mid, mid, mid_square);

            if (mid_square < fixed_input) begin
                low = mid + 1;
                out = mid;
            end else if (mid_square > fixed_input) begin
                high = mid - 1;
            end else begin
                out = mid;
                low = high + 1;
            end
        end
    end

endmodule
```

Log | Share

```
[2024-09-26 10:54:55 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Input: 8.156250, Output: 2.855469
Input: 50.125000, Output: 7.078125
Input: 122.750000, Output: 11.078125
testbench.sv:27: $finish called at 30 (1s)
Done
```

For **fixed-point square-root binary search algorithm** with shift registers is used.

Link for code, testbench and outputs: Squareroot module

16-bit Fixed point to IEEE format Code

testbench.sv

```
// Code your testbench here
// or browse Examples
module tb_fixed16_to_ieee754;

    logic [15:0] fixed_in;
    logic [31:0] ieee_out;

    fixed16_to_ieee754 dut (
        .fixed_in(fixed_in),
        .ieee_out(ieee_out)
    );

    initial begin
        $monitor("fixed_in = %b, ieee_out = %b", fixed_in, ieee_out);

        fixed_in = 16'b0000000000000000;  //0
        #10;
        fixed_in = 16'b0000000101000000;  // 1.5
        #10;
        fixed_in = 16'b0000101000100000;  // 10.25
        #10;
        fixed_in = 16'b1111110110100000;  // -2.75
        #10;
        fixed_in = 16'b1110111101100000;  // -16.5
        #10;
        fixed_in = 16'b0000000000000001;  //(denormalized input)
        #10;
        $stop;
    end

endmodule
```

design.sv

```
// Code your design here
module fixed16_to_ieee754 (
    input logic [15:0] fixed_in,
    output logic [31:0] ieee_out
);

    logic sign;
    logic [7:0] exponent;
    logic [22:0] mantissa;

    logic [15:0] abs_value;
    integer shift_count;
    logic [31:0] norm_value;

    always_comb begin
        if (fixed_in[15] == 1'b1) begin
            sign = 1'b1; //negative number
            abs_value = ~fixed_in + 1;  // Two's complement for negative number
        end else begin
            sign = 1'b0;
            abs_value = fixed_in;
        end

        shift_count = 0; // To normalize the absolute value
        norm_value = abs_value << 16;

      while (norm_value[31] == 0 && shift_count < 31) begin
        //Shifting till we get 1. somthing and incrementing it
            norm_value = norm_value << 1;
            shift_count = shift_count + 1;
        end

      if (abs_value == 16'b0) begin //if zero
            exponent = 8'b00000000;
            mantissa = 23'b0;
      end else begin
            exponent = 8'd127 + (15 - shift_count);
        mantissa = norm_value[30:8]; // Truncation if any overflow
        end

        ieee_out = {sign, exponent, mantissa};
    end
endmodule
```

Log | Share

```
fixed_in = 0000000101000000, ieee_out = 01000011101000000000000000000000
fixed_in = 0000101000100000, ieee_out = 01000101001000100000000000000000
fixed_in = 1111110110100000, ieee_out = 11000100000110000000000000000000
fixed_in = 1110111101100000, ieee_out = 11000101100001010000000000000000
fixed_in = 0000000000000001, ieee_out = 00111111100000000000000000000000
```

Link for code, testbench and outputs: fixedtoIEEE

Division Module Code

```verilog
module tb_division;
    parameter WIDTH = 8;
    // Inputs
    reg [WIDTH-1:0] A;
    reg [WIDTH-1:0] B;
    // Outputs
    wire [WIDTH-1:0] Res;

    // Instantiate the division module (UUT)
    division #(WIDTH) uut (
        .A(A),
        .B(B),
        .Res(Res)
    );

    initial begin
      $dumpfile("tb_division.vcd");   // Dump to this file
      $dumpvars(0, tb_division);     // Dump all variables in this module (testbench
        // Initialize Inputs and wait for 100 ns
        A = 0;  B = 0;  #100;  //Undefined inputs
        //Apply each set of inputs and wait for 100 ns.
        A = 100;    B = 10; #100;
        A = 200;    B = 40; #100;
        A = 90; B = 9;  #100;
        A = 70; B = 10; #100;
        A = 16; B = 3;  #100;
        A = 255;    B = 5;  #100;
    end

endmodule
```

```verilog
module division(A,B,Res);

    parameter WIDTH = 8;
    //input and output ports.
    input [WIDTH-1:0] A;
    input [WIDTH-1:0] B;
    output [WIDTH-1:0] Res;

    //internal variables
    reg [WIDTH-1:0] Res = 0;
    reg [WIDTH-1:0] a1,b1;
    reg [WIDTH:0] p1;
    integer i;

    always@ (A or B)
    begin

        a1 = A;
        b1 = B;
        p1= 0;
        for(i=0;i < WIDTH;i=i+1)
          begin
            p1 = {p1[WIDTH-2:0],a1[WIDTH-1]};
            a1[WIDTH-1:1] = a1[WIDTH-2:0];
            p1 = p1-b1;
            if(p1[WIDTH-1] == 1)    begin
                a1[0] = 0;
                p1 = p1 + b1;   end
            else
                a1[0] = 1;
        end
        Res = a1;
    end

endmodule
```

link

Overview presentation - 3

01 Traditional Methodology

02 CORDIC in ICA

03 Proposed Methodology

04 Gram-Schmidt Orthogonalization

05 Classic and Modified GS

06 Scalar Product and Projection Computation using CORDIC

07 Cross-product

08 Generalized Algorithm for nD cross-product

09 Codes in Rotation and Vectoring Modes

Traditional Methodology

For a system of n-independent sources, the mixed signal vectors are related to source signal vectors as X = AS after pre-processing, for V as co-variance matrix, Z is whitened signal, it becomes,

$$\mathbf{Z} = \mathbf{V}^{-1/2}\mathbf{X}$$

1. To maximize the non-gaussanity, weight vector is updated by kurtosis iteratively as,

$$\mathbf{w}_i^{\text{new}} = E\left[\mathbf{Z}\left(\mathbf{w}_i^T\mathbf{Z}\right)^3\right] - 3\mathbf{w}_i$$

2. To make the sources independent, Orthogonalization using Gram-Schmidt Algorithm is used to prevent multiple vectors to converge to same independent component. The computation is done using CORDIC

$$\mathbf{w_i}^{\text{new}} = \mathbf{w_i}^{\text{new}} - \sum_{i=0}^{i-1}\left(\mathbf{w_i}\cdot\mathbf{w_j}^{c}\right)\mathbf{w_j}^{c}$$

3. After orthogonalizing each vector the result is normalized using the formula

$$\underline{\mathbf{w_i}}^{\text{new}} = \frac{\mathbf{w_i}^{\text{new}}}{\left\|\mathbf{w_i}^{\text{new}}\right\|}.$$

CORDIC Algorithm in ICA

COordinate **R**otation **DI**gital **C**omputer

CORDIC is an iterative algorithm used for performing various mathematical functions like trigonometric, hyperbolic, logarithmic, and linear operations using only shifts, additions, and subtractions. It eliminates the need for complex multiplications and relies on only bit-shifting operations. Widely used for DSP and FPGA implementations.

- No Multiplications: It only uses shift-and-add operations, which are cheaper and faster in hardware than multiplication.
- Low Complexity: Due to iterative nature and reliance on shifts, it is implemented with simple hardware, making it less complex.
- Versatility: It can compute various functions (rotation, vector magnitude, trigonometric functions) using the same hardware.

CORDIC is particularly useful in ICA, which requires efficient vector rotation and vector magnitude calculations.

CORDIC for 2D and nD

For 2D in clockwise rotation,

$$\begin{bmatrix} x_f \\ y_f \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

$$x_f = x_0 \cdot \cos(\theta) + y_0 \cdot \sin(\theta)$$

$$y_f = -x_0 \cdot \sin(\theta) + y_0 \cdot \cos(\theta)$$

For iterative rotation mode,

$$x_f = x_0 + (y_0 >> k), y_f = y_0 - (x_0 >> k)$$

$$R2D_x, j = \mathrm{Rot}_x(z_{1,j}, z_{2,j}, \mathrm{Vec}_\theta(w_{1,1}, w_{1,2}))$$

For nD in clockwise rotation,

$$RnD_x, j = \mathrm{Rot}_x(z_{1,j}, R(n-1)D_x, j, \mathrm{Vec}_\theta(w_{1,n}, V(n-1)D_x))$$

$$R_{x,j}^{2\mathrm{D}} = \mathrm{Rot}_x(z_{1,j}, z_{2,j}, \mathrm{Vec}_\theta(w_{1,1}, w_{1,2}))$$
$$V_\theta^{2\mathrm{D}} = \mathrm{Vec}_\theta(\underline{w}_{1,1}, \underline{w}_{1,2}), \quad V_x^{2\mathrm{D}} = \mathrm{Vec}_x(\underline{w}_{1,1}, \underline{w}_{1,2})$$
$$R_{x,j}^{n\mathrm{D}} = \mathrm{Rot}_x(z_{1,j}, R_{x,j}^{(n-1)\mathrm{D}}, \mathrm{Vec}_\theta(w_{1,n}, V_x^{(n-1)\mathrm{D}}))$$
$$V_\theta^{n\mathrm{D}} = \mathrm{Vec}_\theta(\underline{w}_{1,n}, V_x^{(n-1)\mathrm{D}}), \quad V_x^{n\mathrm{D}} = \mathrm{Vec}_x(\underline{w}_{1,n}, V_x^{(n-1)\mathrm{D}})$$

More on CORDIC Algorithm

Left Shift Registers (LSR) and Right Shift Registers (RSR):

- Left Shift Register (LSR): Moves bits to the left, effectively multiplying by powers of 2. This operation is used in CORDIC for scaling up values as needed during iterations.
- Right Shift Register (RSR): Moves bits to the right, effectively dividing by powers of 2.

The combination of LSR and RSR in CORDIC enables scalability and efficiency.

CORDIC Modes: Rotation and Vectoring

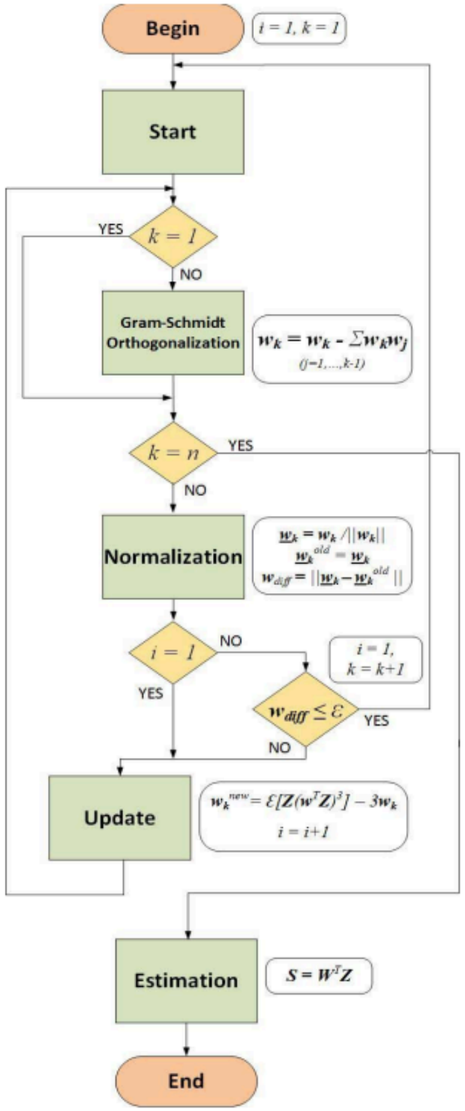
1. *Rotation Mode:*

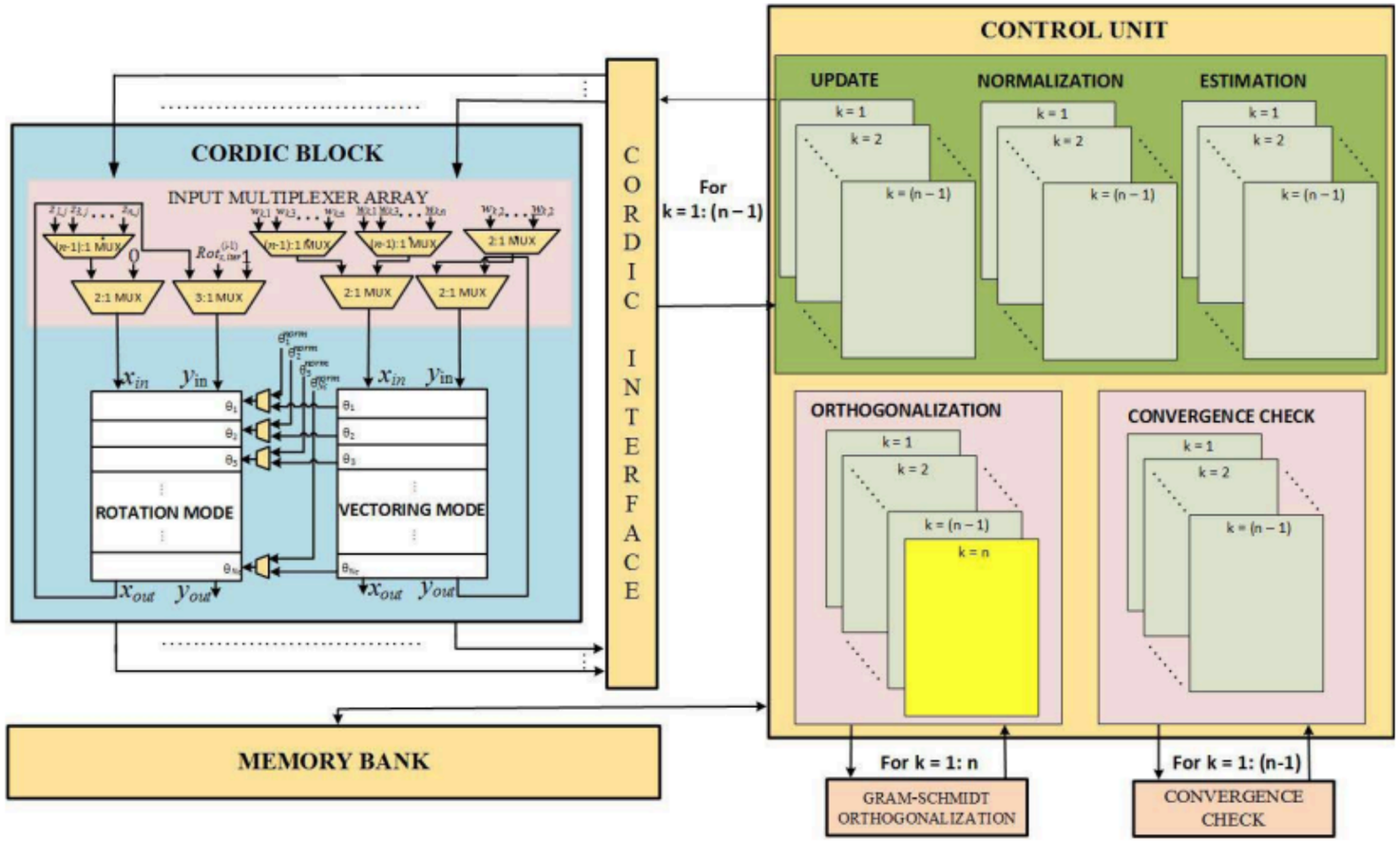
- Used to rotate a vector to achieve independent components.
- Computes the new coordinates of a vector after a specified rotation by iterating and refining the angle approximation using precomputed angles.
- Useful to achieve independent components.

2. *Vectoring Mode:*

- Used to compute the magnitude and angle of a given vector.
- Iteratively reduces the y-component of the vector to zero, aligning the vector with the x-axis.
- Useful to normalize vectors or calculate magnitudes.

Proposed Methodology for nD Simplex FastICA





Orthogonalization

Orthogonalization is the process of converting a set of vectors into a set of orthogonal (mutually perpendicular) vectors. This is important in linear algebra, especially in signal processing, data analysis, and optimization algorithms like Independent Component Analysis (ICA).
The dot product of two orthogonal vectors v1 and v2 = 0

Grahm-Schmidt Orthogonalization

- The Gram-Schmidt orthogonalization is a mathematical process used to convert a set of linearly independent vectors into a set of orthogonal (or orthonormal) vectors that span the same subspace. It's commonly used in linear algebra, particularly in the context of inner product spaces.
- Here we are implementing GS orthogonalization using CORDIC computation. Which will save us the cost of expensive multiplier, adder, divider and square root operation.
- We consider a matrix W with dimensions n x k, where each column represents the each independent vector, and n being the dimension of the vector space.
- Our motive is to obtain matrix Q with Orthogonal Columns qi. This process can be implemented in two ways: 1) Classical GS Algorithm 2) Modified GS Algorithm

$$\mathbf{w_j} \leftarrow \mathbf{w_j} - proj_{\mathbf{w_i}}(\mathbf{w_j}) \qquad proj_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle}\mathbf{u}$$

Classic GS Algorithm

for $k = 1, 2, ..., n$ **do**
  $q_k = w_k$
  for $i = 1, 2, ..., k-1$ **do**
    $r_{ik} = q_i^t w_k$
  end for
  $q_k \leftarrow q_k - \sum_{i=1}^{k-1} r_{ik} q_i$
  $r_{kk} = \|q_k\|$
  $q_k \leftarrow q_k / r_{kk}$
end for

Modified GS Algorithm

for $k = 1, 2, ..., n$ **do**
  $q_k = w_k$
  for $i = 1, 2, ..., k-1$ **do**
    $r_{ik} = q_i^t q_k$
    $q_k \leftarrow q_k - r_{ik} q_i$
  end for
  $r_{kk} = \|q_k\|$
  $q_k \leftarrow q_k / r_{kk}$
end for

- qi are the required orthonormal vectors, wi are the given independent vectors. rik is the dot product qi and wk
- The crucial difference between the two is that in the MGS, the projections rik.qi are subtracted immediately after being calculated, whereas in the CGS, they are all subtracted in a single step.

2D GS Scalar Product Computation

- The first step for GS is the computation of the **scalar product** w2, w1, where **w2** is the **vector to be orthogonalized**, and **$\underline{\mathbf{wi}}$** indicates the **normalized value of wi**
- For that the first step is **Cartesian to Polar Conversion**

$$\underline{\mathbf{w}}_1 = [\underline{w}_{1,1}\ \underline{w}_{1,2}]^T = [\cos\theta_1\ \ \sin\theta_1]^T$$

- where **θ1** is the **Polar** calculated using the inverse tangent function:

$$\theta_1 = \tan^{-1}(\underline{w}_{1,2}/\underline{w}_{1,1})$$

- Next step is the **Scalar Product calculation** of those 2 vectors w2, w1 which is given by:

$$\langle \mathbf{w_2}, \underline{\mathbf{w}}_\mathbf{1} \rangle = w_{2,1}\cos\theta_1 + w_{2,2}\sin\theta_1$$

- that shows dot product is computed by **multiplying the components of w2** (in Cartesian coordinates) **with the cosine and sine of the polar angle θ1**

Scalar Product Computation using CORDIC

- Using **CORDIC's rotation operation**, the dot product in equation (5) can be rewritten as:

$$\langle \mathbf{w_2}, \underline{\mathbf{w}}_\mathbf{1} \rangle = \mathcal{R}ot_x(w_{2,1}, w_{2,2}, \theta_1)$$

- Here, **Rotx(w2,1, w2,2, θ1)** represents the result of **rotating the vector** (w2,1, w2,2) by an angle θ1 using **CORDIC in vectoring mode**.

Therefore we can say that,

- The angle θ1 is the polar angle of vector w1 found using **CORDIC vectoring mode**.
- After calculating θ1, CORDIC performs a **rotation of vector** w2 by this angle to compute the scalar product as a combination of cos(θ1) and sin(θ1) multiplied by the components of w2.

2D GS Projection Computation

To compute the projection of w2 onto w1

x2 0 + y2 0 = Vecx(x0, y0),

xf = x0 cos θ + y0 sin θ = Rotx (x0, y0,θ)

yf = - x0 sin θ + y0 cos θ = Roty (x0, y0,θ)

where **Rot(x/y)** denotes the x/y component of the CORDIC rotation output and V ecx denotes the magnitude of the output from CORDIC vectoring

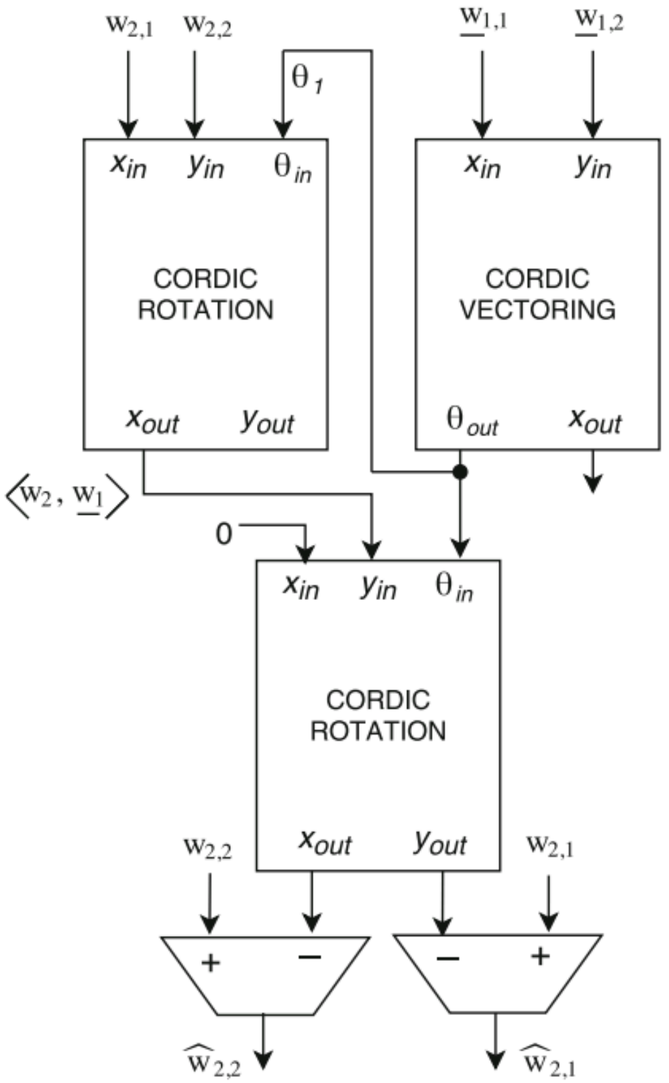
- The **orthogonalized vector $\hat{w}_2$** is calculated by subtracting the projection of w_2 onto $\underline{w}_1$ from w_2:

$$\begin{bmatrix} \hat{w}_{2,1} \\ \hat{w}_{2,2} \end{bmatrix} = \begin{bmatrix} w_{2,1} - \langle \mathbf{w_2}, \underline{\mathbf{w}}_\mathbf{1} \rangle \underline{w}_{1,1} \\ w_{2,2} - \langle \mathbf{w_2}, \underline{\mathbf{w}}_\mathbf{1} \rangle \underline{w}_{1,2} \end{bmatrix}$$

- Using CORDIC rotation the projection computation can be written as:

$$\begin{bmatrix} \hat{w}_{2,1} \\ \hat{w}_{2,2} \end{bmatrix} = \begin{bmatrix} w_{2,1} - \mathcal{R}ot_y(0, \mathcal{R}ot_x(w_{2,1}, w_{2,2}, \theta_1), \theta_1) \\ w_{2,2} - \mathcal{R}ot_x(0, \mathcal{R}ot_x(w_{2,1}, w_{2,2}, \theta_1), \theta_1) \end{bmatrix}$$

- The **orthogonalized vector components** $\hat{w}_{2,1}$ and $\hat{w}_{2,2}$ are obtained by **subtracting** the **projection** result from the **original vector w2**



Optimized 2D GS Scalar Product Computation & Projection Computation

- To decrease the complexity we use a single CORDIC Rotation and CORDIC Vectoring stage
- In the first stage we would compute of the **components of the projection of w_2 onto w_1** which is

$$\begin{bmatrix} \langle \mathbf{w_2}, \underline{\mathbf{w}}_1 \rangle \underline{w}_{1,1} \\ \langle \mathbf{w_2}, \underline{\mathbf{w}}_1 \rangle \underline{w}_{1,2} \end{bmatrix} = \begin{bmatrix} (w_{2,1}\underline{w}_{1,1} + w_{2,2}\underline{w}_{1,2})\underline{w}_{1,1} \\ (w_{2,1}\underline{w}_{1,1} + w_{2,2}\underline{w}_{1,2})\underline{w}_{1,2} \end{bmatrix}$$

- Using polar coordinates the above can be written as:

$$\begin{bmatrix} \langle \mathbf{w_2}, \underline{\mathbf{w}}_1 \rangle w_{1,1} \\ \langle \mathbf{w_2}, \underline{\mathbf{w}}_1 \rangle w_{1,2} \end{bmatrix} = \begin{bmatrix} (w_{2,1}\cos\theta_1 + w_{2,2}\sin\theta_1)\cos\theta_1 \\ (w_{2,1}\cos\theta_1 + w_{2,2}\sin\theta_1)\sin\theta_1 \end{bmatrix}$$

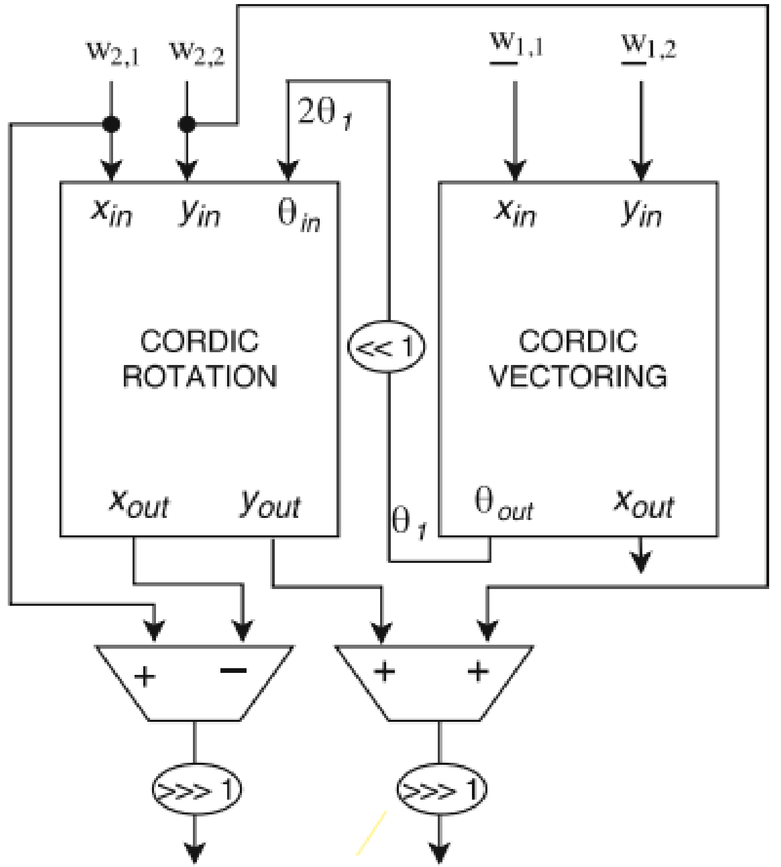
- On further simplification using CORDIC operations we get:

$$(w_{2,1}\cos\theta_1 + w_{2,2}\sin\theta_1)\cos\theta_1 = \frac{1}{2}[w_{2,2} - \mathcal{R}ot_y(w_{2,1}, w_{2,2}, 2\theta_1)]$$

- Therefore the projection computation for GS can be represented as:

$$\begin{bmatrix} \hat{w}_{2,1} \\ \hat{w}_{2,2} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} w_{2,1} - \mathcal{R}ot_x(w_{2,1}, w_{2,2}, 2\theta_1) \\ w_{2,2} + \mathcal{R}ot_y(w_{2,1}, w_{2,2}, 2\theta_1) \end{bmatrix}$$

- In this, **CORDIC vectoring mode** is used to **compute 2θ** which is fed as **input to CORDIC Rotation** mode for **projection computation**



Cross-Product Computation in ICA

Cross-product in 3D is a mathematical operation that returns a vector orthogonal to two orthogonal input vectors in 3D space. From 3D it can be generalized for nD as, the vector orthogonal to each of (n-1) input vectors in nD space.

Why use cross-product in Gram-Schmidt Orthogonalization for nD in ICA?

As in Gram-Schmidt Orthogonalization for nth dimension involves in subtracting the common component of all (n-1) unit vectors which is computationally expensive, we use nD cross product for computing the nth orthogonal vector, simplifying the process.

Application of cross-product in ICA

To extract independent components from a mixed signal, orthogonalizing the basis vectors makes the convergence to achieve faster, cross-product helps in simplifying the orthogonalization for higher dimensions reducing the computational complexity.

Mathematical formulation for 3D & nD cross-product

Let the vectors be v1 and v2 in directions e1,e2 and e3,

$$\mathbf{v}_1 = v_{1e_1}\mathbf{e}_1 + v_{1e_2}\mathbf{e}_2 + v_{1e_3}\mathbf{e}_3 = [v_{1e_1}, v_{1e_2}, v_{1e_3}]$$

$$\mathbf{v}_2 = v_{2e_1}\mathbf{e}_1 + v_{2e_2}\mathbf{e}_2 + v_{2e_3}\mathbf{e}_3 = [v_{2e_1}, v_{2e_2}, v_{2e_3}]$$

Cross-product of v1, v2 is computed as

$$\mathbf{v}_1 \times \mathbf{v}_2 = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ v_{1e_1} & v_{1e_2} & v_{1e_3} \\ v_{2e_1} & v_{2e_2} & v_{2e_3} \end{vmatrix}$$

$$\mathbf{v}_1 \times \mathbf{v}_2 = [v_{1e_2}v_{2e_3} - v_{1e_3}v_{2e_2},\ -(v_{1e_1}v_{2e_3} - v_{1e_3}v_{2e_1}),\ v_{1e_1}v_{2e_2} - v_{1e_2}v_{2e_1}]$$

Cross-product in terms of co-factors V1, V2, V3

$$\mathbf{v}_1 \times \mathbf{v}_2 = V_1\mathbf{e}_1 + V_2\mathbf{e}_2 + V_3\mathbf{e}_3$$

Cross-product in terms of minors M1, M2, M3

$$\mathbf{v}_1 \times \mathbf{v}_2 = M_1\mathbf{e}_1 - M_2\mathbf{e}_2 + M_3\mathbf{e}_3$$

For n vectors,

$$\mathbf{v_n} = \begin{vmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} & \cdots & \mathbf{e_n} \\ v_{1,1} & v_{1,2} & v_{1,3} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & v_{2,3} & \cdots & v_{2,n} \\ \hline v_{n-1,1} & v_{n-1,2} & v_{n-1,3} & \cdots & v_{n-1,n} \end{vmatrix}$$

Cross-product in terms of co-factors (Vik)

$$\mathbf{v_n} = \mathbf{e_1}V_{11} + \mathbf{e_2}V_{12} + \mathbf{e_3}V_{13} + \ldots + \mathbf{e_n}V_{1n} = \sum_{k=1}^{n} e_k V_{1k}$$

Cross-product in terms of minors

$$\mathbf{v_n} = \sum_{k=1}^{n} \mathbf{e_k}(-1)^{1+k}|M_{1k}|.$$

The direction of nth vector is perpendicular to each of (n-1) vectors

Hardware Complexity without using Cross product

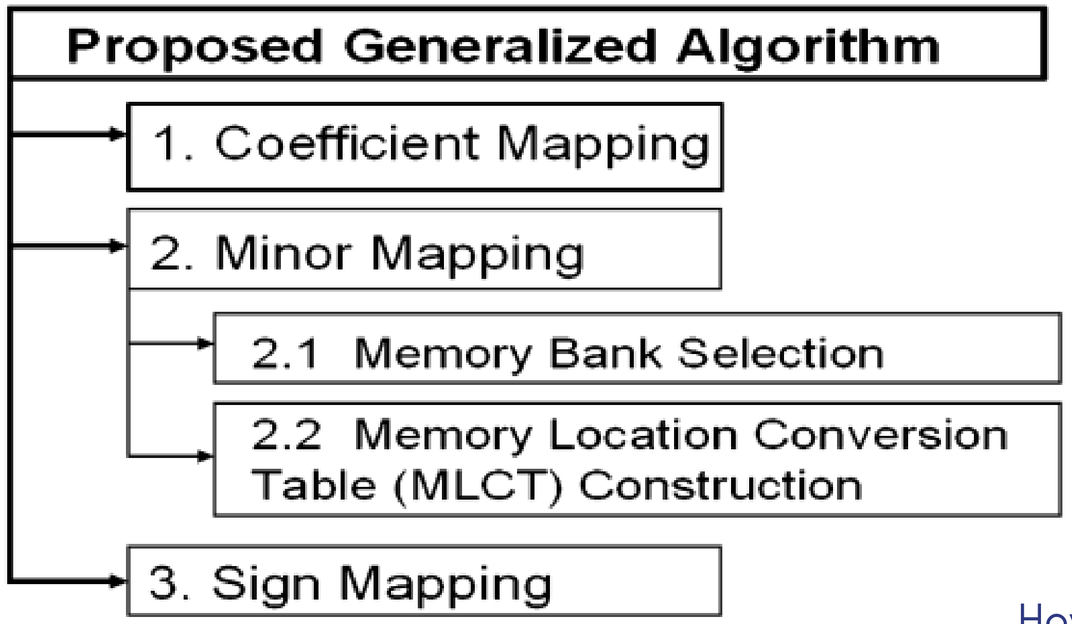
If we denote the number of multiplications, ,two operands addition, subtractions, divisions, squaring, and square rooting operation by $M,\ {}^{2}A,\ S,\ D,\ S_{qrng},$ and S_{qrt} respectively and de noting number of iterations for convergence by P and using the fore-mentioned formula, the hardware complexity of one complete nD FICA iteration stage as can be represented as:-

$$
\begin{aligned}
{}^{n}\mathbf{HW}_{fica} &= {}^{n}\mathbf{HW}_{upd} + {}^{n}\mathbf{HW}_{nrm} + {}^{n}\mathbf{HW}_{oth} \\
&= P\left[K(2n+1) + n(2n-1)\right] \times M + n^{2}P \times S \\
&\quad + P\left[K(2n-1) + n(n-1)\right] \times {}^{2}A + nP \times D \\
&\quad + P(K+n) \times S_{qrng} + P \times S_{qrt} \qquad (1)
\end{aligned}
$$

Since it involves huge computation complexity for each iteration, therefore, Reducing a FICA iteration stage decreases hardware complexity and power consumption by cutting down significant computations.

Generalized Algorithm for nD cross-product

The fundamental principle of formulating a generalized algorithm for cross-product computation hinges on the fact that there exists a structural similarity between and cross-product when expanded in their cofactor-form.



It can be observed that Vn(the cross-product in nD) can be realized using the n times cross-product operation in V(n-1) (the cross product in (n-1)D) .Similarly, cross-product in (n-1)D can be realized by using cross-product operation in (n-2)D, (n-1) times ,and so on until where this structural similarity seizes to exist.

Therefore we can conclude that:-

An nD cross-product operation can be realized by a sequence of 3-D cross-product operation

However, when representing the nD cross-product by n number of (n-1)D cross-products, three mapping schemes need to be employed:-

i) Coefficient Mapping

ii) Minor Mapping

iii) Sign mapping.

Coefficient mapping

For Decomposing an nD cross prodcut into the 3D cross product we requires d = n-3(number of decomposition levels)
Let d denotes the level of decomposition.

For coefficient mapping, we need to use the following theorem:-

Theorem1:

Coefficient Mapping—Considering i and j be the number of unit vectors in R(n) and $R^{((n-1)D}(e_j))$; i.e, 1<=i<=n ,1<=j<=(n-1), the coefficients (i.e., the unit vectors in (n-1)D cross-product
can structurally be mapped into the elements of the first row of each minor of every component of nD cross product using the following quantities:-

$$^{(n-1)D}\mathbf{e_j} \mapsto \begin{cases} ^{(nD)}v_{d,j+(d-1)}, & \text{if } j < i \\ ^{(nD)}v_{d,j+d}, & \text{otherwise} \end{cases}$$

where the number of decomposition level (d) is delimited by

$$1 \leq d \leq (n-3)$$

where, n>=4

Minor Mapping

To explain the Minor mapping scheme we use the cases of 3D and 4D cross-products as examples.

The fundamental 3D vector cross product can be written as:-

$$\mathbf{v_3} = [v_{3,1}\ v_{3,2}\ v_{3,3}] = \begin{vmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} \\ v_{1,1} & v_{1,2} & v_{1,3} \\ v_{2,1} & v_{2,2} & v_{2,3} \end{vmatrix}$$

Expanding the above expression as:-

$$\mathbf{v_3} = \mathbf{e_1}\begin{vmatrix} v_{1,2} & v_{1,3} \\ v_{2,2} & v_{2,3} \end{vmatrix} - \mathbf{e_2}\begin{vmatrix} v_{1,1} & v_{1,3} \\ v_{2,1} & v_{2,3} \end{vmatrix} + \mathbf{e_3}\begin{vmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \end{vmatrix} \quad (1$$

Similarly the fundamental 4D vector cross product can be written as:-

$$\mathbf{v_4} = [v_{4,1}\ v_{4,2}\ v_{4,3}\ v_{4,4}] = \begin{vmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} & \mathbf{e_4} \\ v_{1,1} & v_{1,2} & v_{1,3} & v_{1,4} \\ v_{2,1} & v_{2,2} & v_{2,3} & v_{2,4} \\ v_{3,1} & v_{3,2} & v_{3,3} & v_{3,4} \end{vmatrix}$$

Expanding the above expression as:-

$$\mathbf{v_4} = \mathbf{e_1}\begin{vmatrix} v_{1,2} & v_{1,3} & v_{1,4} \\ v_{2,2} & v_{2,3} & v_{2,4} \\ v_{3,2} & v_{3,3} & v_{3,4} \end{vmatrix} - \mathbf{e_2}\begin{vmatrix} v_{1,1} & v_{1,3} & v_{1,4} \\ v_{2,1} & v_{2,3} & v_{2,4} \\ v_{3,1} & v_{3,3} & v_{3,4} \end{vmatrix} + \mathbf{e_3}\begin{vmatrix} v_{1,1} & v_{1,2} & v_{1,4} \\ v_{2,1} & v_{2,2} & v_{2,4} \\ v_{3,1} & v_{3,2} & v_{3,4} \end{vmatrix} - \mathbf{e_4}\begin{vmatrix} v_{1,1} & v_{1,2} & v_{1,3} \\ v_{2,1} & v_{2,2} & v_{2,3} \\ v_{3,1} & v_{3,2} & v_{3,3} \end{vmatrix}$$

Expressing the above as vector form:-

$$v_{4,1} = v_{1,2}\begin{vmatrix} v_{2,3} & v_{2,4} \\ v_{3,3} & v_{3,4} \end{vmatrix} - v_{1,3}\begin{vmatrix} v_{2,2} & v_{2,4} \\ v_{3,2} & v_{3,4} \end{vmatrix} + v_{1,4}\begin{vmatrix} v_{2,2} & v_{2,3} \\ v_{3,2} & v_{3,3} \end{vmatrix}$$

$$v_{4,2} = v_{1,1}\begin{vmatrix} v_{2,3} & v_{2,4} \\ v_{3,3} & v_{3,4} \end{vmatrix} - v_{1,3}\begin{vmatrix} v_{2,1} & v_{2,4} \\ v_{3,1} & v_{3,4} \end{vmatrix} + v_{1,4}\begin{vmatrix} v_{2,1} & v_{2,3} \\ v_{3,1} & v_{3,3} \end{vmatrix}$$

$$v_{4,3} = v_{1,1}\begin{vmatrix} v_{2,2} & v_{2,4} \\ v_{3,2} & v_{3,4} \end{vmatrix} - v_{1,2}\begin{vmatrix} v_{2,1} & v_{2,4} \\ v_{3,1} & v_{3,4} \end{vmatrix} + v_{1,4}\begin{vmatrix} v_{2,1} & v_{2,2} \\ v_{3,1} & v_{3,2} \end{vmatrix}$$

$$v_{4,4} = v_{1,1}\begin{vmatrix} v_{2,2} & v_{2,3} \\ v_{3,2} & v_{3,3} \end{vmatrix} - v_{1,2}\begin{vmatrix} v_{2,1} & v_{2,3} \\ v_{3,1} & v_{3,3} \end{vmatrix} + v_{1,3}\begin{vmatrix} v_{2,1} & v_{2,2} \\ v_{3,1} & v_{3,2} \end{vmatrix}.$$

Minor Mapping

For the Generalisation presepective, we are taking that row index corresponds to memory bank and column index corresponds to may correspond to memory location.

Now our problems simplfies to selection problem of memory bank and memory location only

Theorem 2: **Memory Bank Selection**—If ${}^{(nD)}\phi_m$ denotes m^{th} memory bank for nD cross-product, then the following relationship holds:

$$ {}^{(nD)}\phi_m = {}^{(n-1)D}\phi_m + 1 \qquad (16) $$

i.e the we have to take always the next vector to the our initial vector in (n-1)D as our memory bank in place of our initial memory bank

Minor Mapping

For the selection of memory location we have two cases

Case i)The Memory location is to be incremented with 1 with the memory location index of (n-1)D cross-product generates the memory location index of nD cross product

Case ii). Memory location index of the vectors at the same minor position in can straightway be mapped into the memory locations of the vectors without any change.

These two cases can be denoted by 0 and 1

Denote the position of the minor by r where and the memory location of an element in any minor by βi

In case of 4D to 3D where 1<=r<=3 and 1<=i<=2
We are denoting the component number by c

r → / c ↓	r = 1	r = 2	r = 3
	$\beta_1\ \beta_2$	$\beta_1\ \beta_2$	$\beta_1\ \beta_2$
c = 1	0 0	0 0	0 0
c = 2	0 0	1 0	1 0
c = 3	1 0	1 0	1 1
c = 4	1 1	1 1	1 1

Fig. 2. 3D to 4D memory location conversion table.

Minor Mapping

Theorem 3: $(n-1)D$ **to** nD **MLCT construction**—Considering ${}^{n}_{(n-1)}\beta_i$ denotes the β_i involved in the $(n-1)D$ to nD **MLCT**, $(n-2)D$ to $(n-1)D$ **MLCT** can be used to construct $(n-1)D$ to nD **MLCT** using the following formula for $n > 4$:

i) when $1 \leq c \leq (n-1)$ and $\forall c$, $1 \leq r \leq (n-2)$:

$$ {}^{n}_{(n-1)}\beta_i = \begin{cases} {}^{(n-1)}_{(n-2)}\beta_i & \text{for } 1 \leq i \leq (n-3) \\ 0 & \text{for } i = (n-2) \end{cases}; \quad (18) $$

ii) when $1 \leq c \leq (n-2)$ and $\forall c$, $r = (n-1)$:

$$ {}^{n}_{(n-1)}\beta_i|_{r=(n-1)} = {}^{n}_{(n-1)}\beta_i|_{r=(n-2)} \quad \text{for} \quad 1 \leq i \leq (n-2); \quad (19) $$

iii) when $c = (n-1)$ and $r = (n-1)$:

$$ {}^{n}_{(n-1)}\beta_i = 1 \quad \text{for} \quad 1 \leq i \leq (n-2); \quad (20) $$

iv) when $c = n$, then $\forall c$, $1 \leq r \leq (n-1)$:

$$ {}^{n}_{(n-1)}\beta_i = 1 \quad \text{for} \quad 1 \leq i \leq (n-2). \quad (21) $$

Using the adjacent theorem we can form the 4D to 5D MLCT as

r → / c ↓	r = 1	r =2	r = 3	r = 4
	β1 β2β3	β1 β2β3	β1 β2β3	β1 β2β3
c = 1	0 0 0	0 0 0	0 0 0 →	0 0 0
c = 2	0 0 0	1 0 0	1 0 0 →	1 0 0
c = 3	1 0 0	1 0 0	1 1 0 →	1 1 0
c = 4	1 1 0	1 1 0	1 1 0	1 1 1
c = 5	1 1 1	1 1 1	1 1 1	1 1 1

Fig. 3. Example of construction of $4D$ to $5D$ **MLCT** from $3D$ to $4D$ **MLCT** as shown in Fig. 2.

Minor Mapping

r → / c ↓	r = 1	r = 2	r = 3	r = 4
	$\beta_1\ \beta_2\beta_3$	$\beta_1\ \beta_2\beta_3$	$\beta_1\ \beta_2\beta_3$	$\beta_1\ \beta_2\beta_3$
c = 1	0 0 0	0 0 0	0 0 0 →	0 0 0
c = 2	0 0 0	1 0 0	1 0 0 →	1 0 0
c = 3	1 0 0	1 0 0	1 1 0 →	1 1 0
c = 4	1 1 0	1 1 0	1 1 0	1 1 1
c = 5	1 1 1	1 1 1	1 1 1	1 1 1

Fig. 3. Example of construction of $4D$ to $5D$ **MLCT** from $3D$ to $4D$ **MLCT** as shown in Fig. 2.

n

Using the above MLCT we can find the V5,3 component as

$$v_{5,3} = v_{1,1}\begin{vmatrix} v_{2,2} & v_{2,4} & v_{2,5} \\ v_{3,2} & v_{3,4} & v_{3,5} \\ v_{4,2} & v_{4,4} & v_{4,5} \end{vmatrix} - v_{1,2}\begin{vmatrix} v_{2,1} & v_{2,4} & v_{2,5} \\ v_{3,1} & v_{3,4} & v_{3,5} \\ v_{4,1} & v_{4,4} & v_{4,5} \end{vmatrix}$$
$$+v_{1,4}\begin{vmatrix} v_{2,1} & v_{2,2} & v_{2,5} \\ v_{3,1} & v_{3,2} & v_{3,5} \\ v_{4,1} & v_{4,2} & v_{4,5} \end{vmatrix} - v_{1,5}\begin{vmatrix} v_{2,1} & v_{2,2} & v_{2,4} \\ v_{3,1} & v_{3,2} & v_{3,4} \\ v_{4,1} & v_{4,2} & v_{4,4} \end{vmatrix}. \quad (22)$$

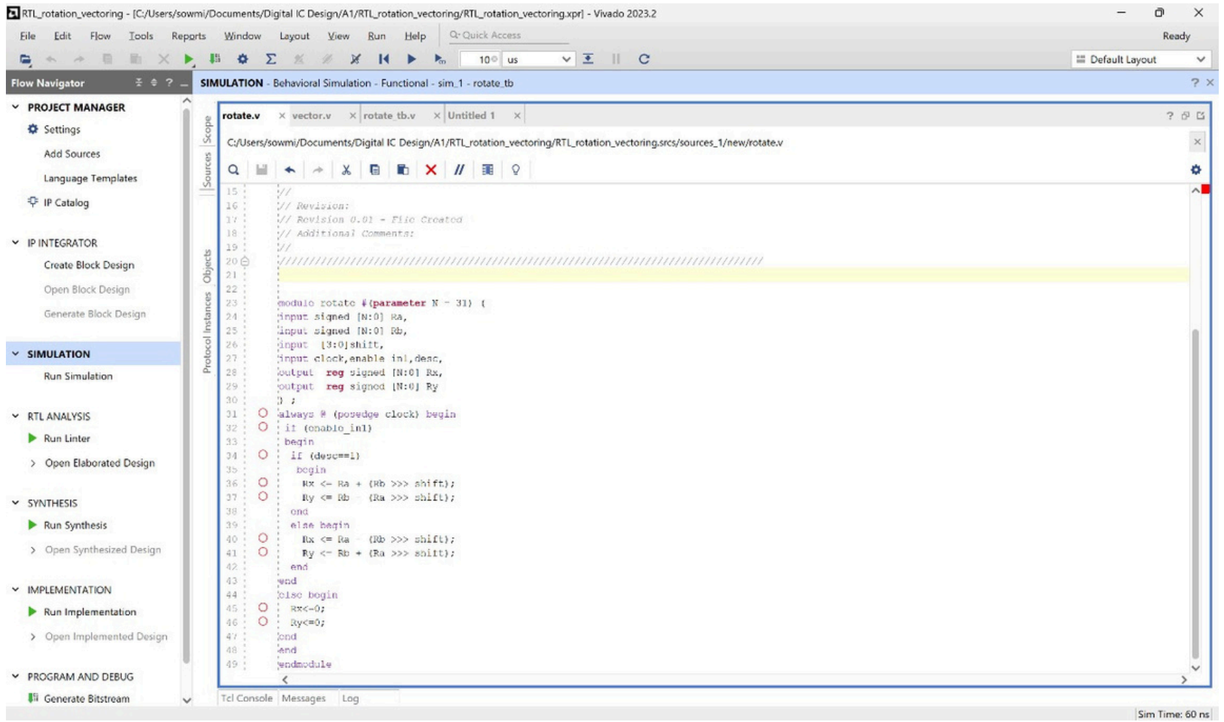
Sign Mapping

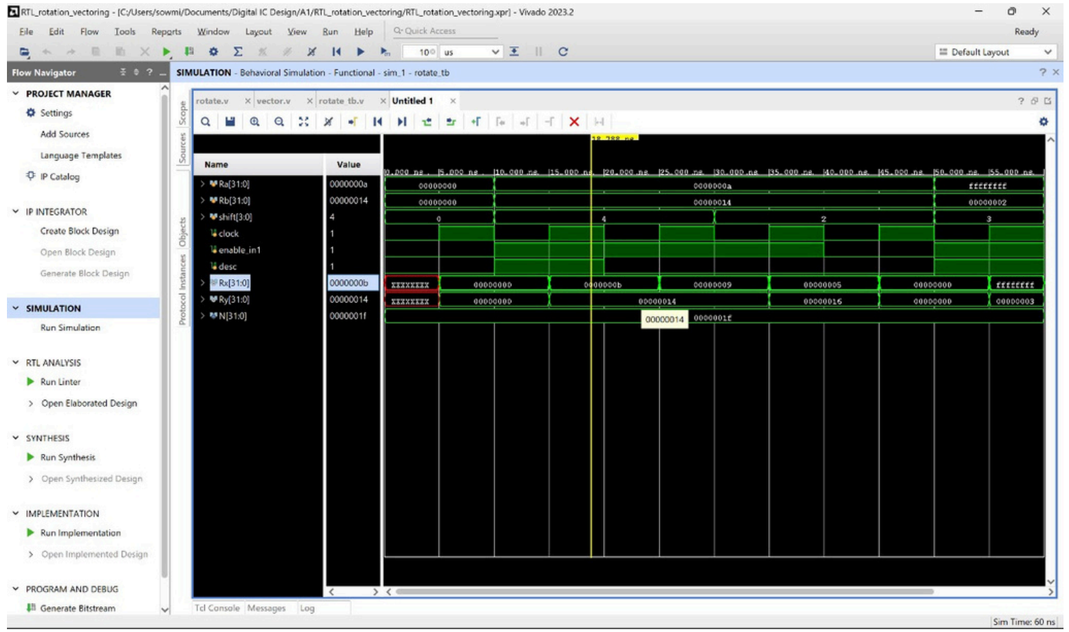
The sign of each component of the vector cross product is determined by,
For each component of the cross-product vector,
the sign is positive when the index is even
the sign is negative when the index in odd

3D to nD computation

- The given sampled inputs are orthogonal i.e. each sample is along independent direction.
- The sampled data, where 8 samples are taken, in fixed-point representation of 8-bits per sample, with a split of 4 integer bits and 4 fractional bits.
- The data cannot be NULL, as it would lead to 0 norm and undefined normalized forms.
- CORDIC Algorithm for square root, multiplier and division blocks.
- If the results need to be converted into IEEE floating point format the IEEE conversion block can be used.

Code for CORDIC Rotating and Vectoring





References

An Overview of Independent Component Analysis and Its Applications

Ganesh R. Naik and Dinesh K Kumar

Independent Component Analysis: Algorithms and Applications

Aapo Hyvärinen and Erkki Oja

Simplex FastICA: An Accelerated and Low Complex Architecture Design Methodology for nDFastICA

Swati Bhardwaj and Amit Acharyya

References

Algorithm and Architecture for N-D Vector Cross-Product Computation

Amit Acharyya, Koushik Maharatnaand Bashir M. Al-Hashimi

Low-Complex and Low-Power n-dimensional Gram-Schmidt Orthogonalization Architecture Design Methodology

Swati Bhardwaj1 Shashank Raghuraman1 Jayesh B. Yerrapragada Koushik Agathya Jagirdar Maharatna Amit Acharyya1

Grahm-Schmidt Orthogonalization

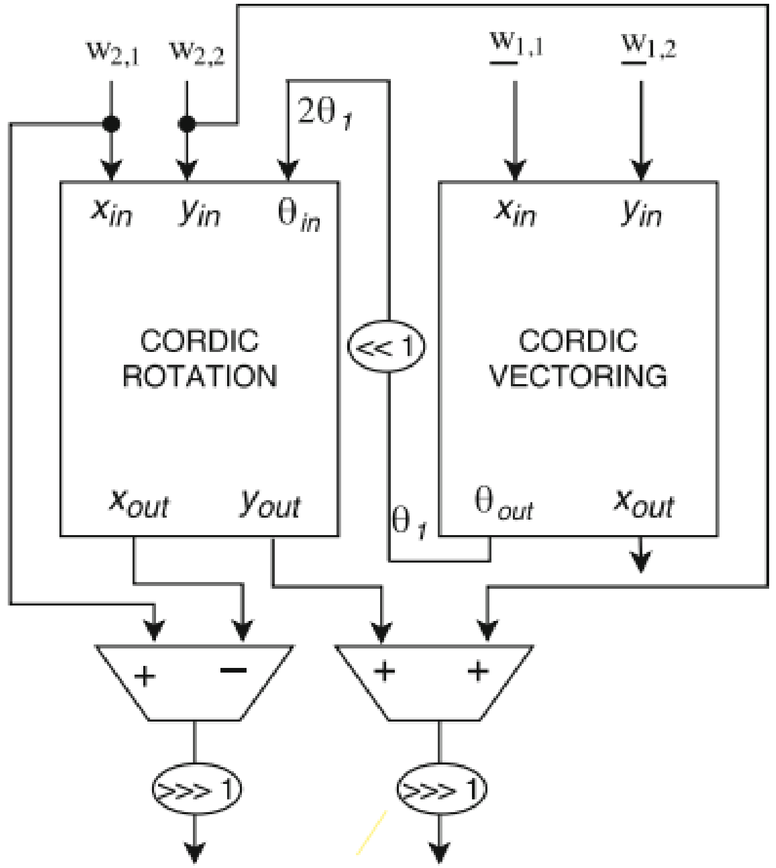
without CORDIC

for $k = 1, 2, ..., n$ **do**
  $q_k = w_k$
  for $i = 1, 2, ..., k-1$ **do**
    $r_{ik} = q_i^t q_k$
    $q_k \leftarrow q_k - r_{ik} q_i$
  end for
  $r_{kk} = \|q_k\|$
  $q_k \leftarrow q_k / r_{kk}$
end for

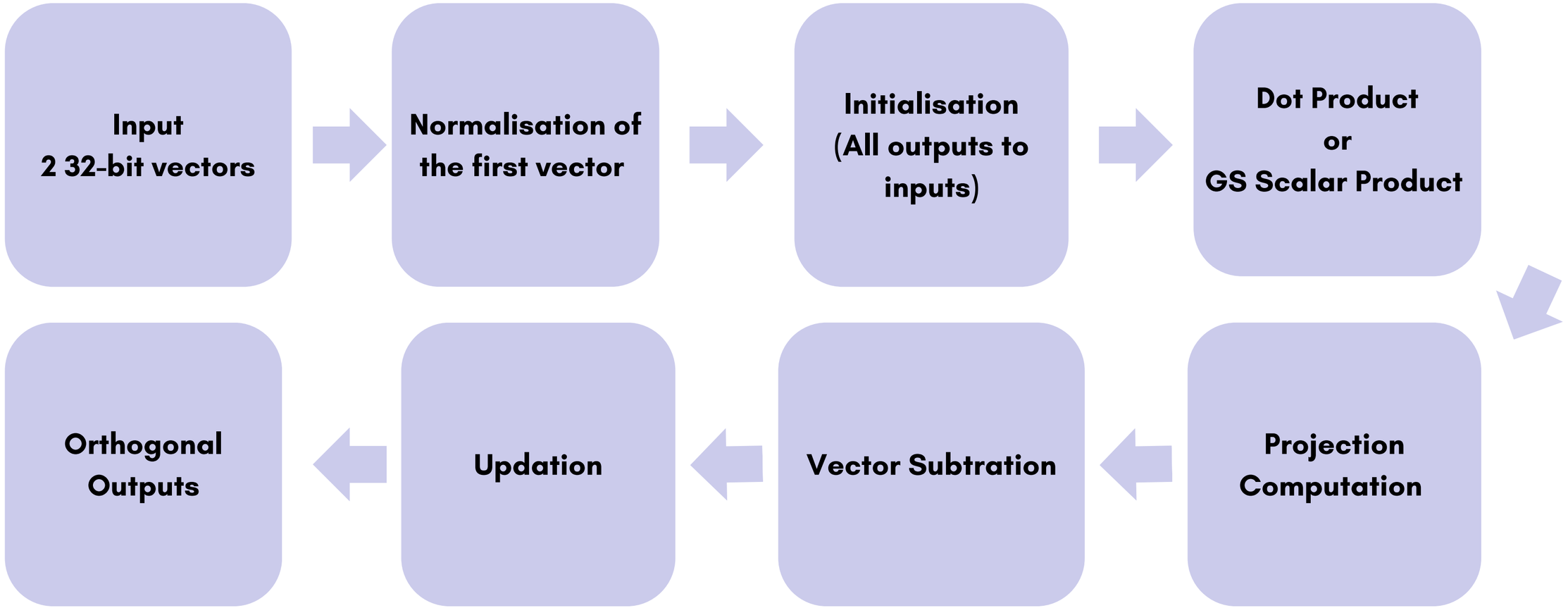
- A code and a test bench have been written to implement Gram-Schmidt orthogonalization using the basic algorithm outlined here.
- The input consists of 8-bit, 8-sample vectors, and the output provides 8-bit, 8-sample orthogonal vectors.
- The design is divided into the following blocks:
  - Normal Square
  - Dot Product
  - Projection

Grahm-Schmidt Orthogonalization Using CORDIC

- A code and test bench have been implemented, utilizing the following blocks to obtain the desired orthogonal output.
- Two 32-bit vectors are provided as input.
- The design is divided into the following blocks:
- Modules being used:
  - Ratate_1 (for first iteration tanθ = 45)
  - Rotate (for CORDIC Rotation)
  - Vector_1
  - Vector
  - Mux_21 (Multiplexer 2:1)
  - Dot_product (For Dot product calculation)



Block Diagram for Normalization



Grahm-Schmidt Orthogonalization

Code for GSO without CORDIC

Code for rotate 1, rotate 16

Code for vector1 and vector16 combined

Code for Dot Product using CORDIC

Code for 2D CORDIC GSO

ICA (Independent Component Analysis)

We have initially worked on Normalization, Gram-Schmidt Orthogonalization in 2D using Verilog, understood how GSO and normalization work.
In normalization, we checked on square root module.
In GSO, we checked on dot product calculation, cross product calculation, division blocks using shift registers instead of multiplication. Then to improve the hardware, we switched to CORDIC vectoring and rotating to implement GSO, Normalization, Update Blocks.

Modules implemented:

- Square root Module
- Division Module
- Dot Product Module
- Cross Product Module

CORDIC using parallel logic

We have worked on vectoring and rotation blocks in 2D and also extended it to 3D by instantiating. We have worked on GSO in 2D and normalization in 2D using parallel logic.

Modules Implemented:

- Vectoring Code
- Rotating Code
- GSO_2D Code
- Normalization_2D Code
- Update_2D Code

CORDIC using Doubly pipelining 2D

We have worked on vectoring and rotation blocks in 2D and also extended it to 3D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic.

Modules Implemented:

- Vectoring Code for 2D input vector
- Rotating Code for 2D input vector
- GSO_2D Code
- Normalization_2D Code
- Update_2D Code (not implemented)

CORDIC using Doubly pipelining 7D

We have worked on vectoring and rotation blocks in 2D and also extended it to 7D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic. (working)

Modules Implemented:

- Vectoring Code for 7D input vector
- Rotating Code for 7D input vector
- GSO_7D Code
- Normalization_7D Code
- Update_7D Code (not implemented)

Normalization using CORDIC

We have worked on vectoring and rotation blocks in 2D and also extended it to 7D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic. (working)

Modules Implemented:

- Vectoring Code for 7D input vector
- Rotating Code for 7D input vector
- GSO_7D Code
- Normalization_7D Code
- Update_7D Code (not implemented)

CORDIC using Doubly pipelining 7D

We have worked on vectoring and rotation blocks in 2D and also extended it to 7D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic. (working)

Modules Implemented:

- Vectoring Code for 7D input vector
- Rotating Code for 7D input vector
- GSO_7D Code
- Normalization_7D Code
- Update_7D Code (not implemented)

CORDIC using Doubly pipelining 7D

We have worked on vectoring and rotation blocks in 2D and also extended it to 7D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic. (working)

Modules Implemented:

- Vectoring Code for 7D input vector
- Rotating Code for 7D input vector
- GSO_7D Code
- Normalization_7D Code
- Update_7D Code (not implemented)

CORDIC using Doubly pipelining 7D

We have worked on vectoring and rotation blocks in 2D and also extended it to 7D by instantiating. We have worked on GSO in 2D and normalization in 2D using doubly pipelining logic. (working)

Modules Implemented:

- Vectoring Code for 7D input vector
- Rotating Code for 7D input vector
- GSO_7D Code
- Normalization_7D Code
- Update_7D Code (not implemented)

Thank You