1. Monte Carlo Simulation

When implementing an algorithm on hardware, such as CORDIC, one would like to obtain the greatest performance with as few resources possible. In the case of CORDIC, provided a desired error interval, the number of iterations and word length can be optimised. This is due to each consecutive CORDIC iteration yielding diminishing precision. Similarly, if a certain world length accommodates the error constraint, the use of a longer word will not only waste resources, it would also reduce performance due to, for instance, arithmetic operations having to be executed over more bits.

To determine the optimal word length and number of iterations, the CORDIC design detailed in section 2 was implemented in MATLAB. MATLAB's Fixed-Point Designer tool was utilised to ensure the software implementation simulated the hardware implementation accurately. For instance, floating point inputs were converted to a fixed-point representation and divisions by 2 were executed by right shifts.

A confidence level of 95%, corresponding to a Z-Score of 1.96, and mean error were used to calculate the confidence interval (CI). Furthermore, the upper bound of the CI was used when determining the optimal word length and iteration number. Due to time constraints, only a total of 20 000 input values, in range [-1, 1] and randomly generated from a uniform distribution, were used for the Monte Carlo simulation. Thus, to increase the CI's reliability and precision in future testing, a larger number of samples should be used.

1.1 Optimal Number of CORDIC Iterations

To ascertain the minimum number of CORDIC iterations to achieve the required mean error of $[-0.5 * 10^{-6}, 0.5 * 10^{-6}]$ with a 95% confidence level, the CIs for 6 to 25 CORDIC iterations were calculated. A fixed word length 32 bits, i.e. 1 sign, 1 integer and 30 fractional bits, was used across all iterations. The word length was chosen on the basis of the input and output never being more than ±1 and to allow for maximum precision in the fractional part.

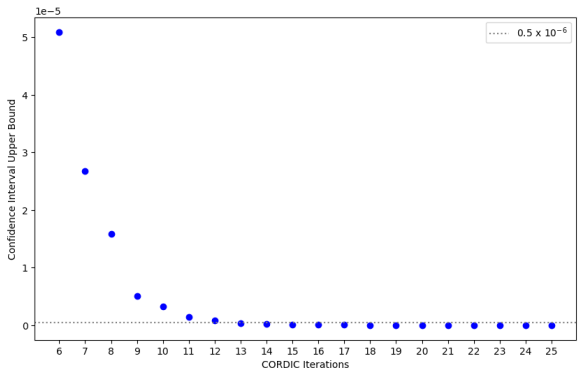

Figure 1. Confidence Interval for various numbers of CORDIC iterations.

Figure 1 shows how increasing the number of CORDIC iterations beyond 14 yields diminishing returns in terms of CI. Also, 14 iterations fulfills the required mean error. Hence, this many iterations will be used for the final design.

1.2 Optimal Word Length

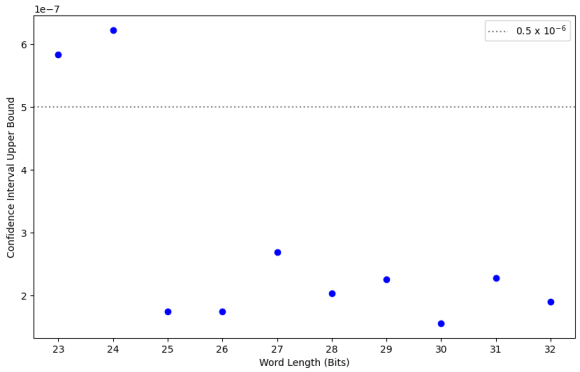

Figure 2. Confidence Interval for different word lengths. Each word had 1 sign and 1 integer bit.

According to Figure 2, the minimum word length to be within the mean error margin with 95% confidence level is 25 bits. Hence, there will be 23 bits allocated to the fractional part.

2. CORDIC Design

In order to design an efficient CORDIC algorithm, one has to understand how it works. The CORDIC algorithm approximates cos(x) and sin(x), for a given angle x, by iteratively applying the rotation matrix to a coordinate vector. By extracting cos(x) from the rotation matrix, the rotational matrix becomes:

$$\cos(x) \begin{bmatrix} 1 & -\tan(x) \\ \tan(x) & 1 \end{bmatrix}$$

If x is carefully chosen to be angles that result in tan(x) being powers of 2^{-i}, then each application of the rotation matrix simplifies to addition, subtraction and right shifts - provided fixed-point number representation is being used. The values of tan(x), where x is 45, 26.56, 14.04 and 7.13 degrees, will be 1, 0.5, 0.25 and 0.125, i.e. powers of negative 2. Since the number of iterations is known ahead of time, a LUT can be implemented to find the correspond angles of arctan(2^{-i}). The cos(x)'s that are taken out of the rotation matrix represent a scaling factor. The product of these cos(x)'s can be precomputed, the result of which is denoted as K, and multiplied by the final value of x to scale it.

The range of angles for which the CORDIC algorithm can compute cos(x) is equal to the sum of the arctan(2^{-i}) values. For 10 iterations, this would be 99.77 degrees, such

that the range of x would be [-99.77, 99.77] degrees. However, in practise, due to the symmetry of cosine, one is only interested in the range [0, 90].

2.1 Offset Optimisation

We are required to calculate cos(x) for an angle x in the range [-1, 1] rad, which corresponds to approximately [-57.3, 57.3] degrees. However, since cos(-x) = cos(x), we can reduce our range to [0, 57.3] degrees by using $|x|$. Furthermore, a higher number of CORDIC iterations corresponds to higher precision. Thus, instead starting the algorithm at coordinates (1, 0), we start at (1, 0.5). The latter corresponds to applying a rotation matrix with an angle of 26.565 degrees. The coordinates (1, 0.5) exceed the unit circle, which is why its scaling factor cos(26.565) is included in the precomputed K. Using an initial offset is not a LUT as it is used as the starting coordinate for all inputs x.

Due to the offset, the first iteration of the CORDIC algorithm uses 14.03 degrees = arctan(2^{-2}). Starting CORDIC with 14.03 degrees results in cos(x) only being computable for x in the range of approximately [-28, 28] degrees, i.e. double the starting angle. In the context of our offset, this means the range of x for which we can compute cos(x) is [-1.435, 54.565] degrees - calculated by adding 26.655 to [-28, 28]. Although the offset offers the advantage of starting CORDIC at its third iteration, since we do not use the the large angle step of 45 degrees, it comes at the cost of being unable to compute cos(x) for x values between 54.565 and 57.3 degrees.

2.2 Exploiting Trigonometric Identities

To resolve the issue of not being able to reach x values 54.565 to 57.3, we make use of the trigonometric identity cos(x) = sin(90-x). In general, for CORDIC to be able to calculate cos(x), it has to apply rotational transformations to both the x- and y-coordinate. Thus, each time cos(x) is calculated, we also have sin(x) if we multiply the final y-coordinate by the scaling factor. As a result, for angles x greater than 45 degrees, we can calculate cos(90-x) and extract sin(90-x), which is equal to cos(x). Although this enables the calculation of cos(x) for x over the required range, it comes at the expense of an additional 'if statement' to determine whether x is greater than 45 degrees, as well as a 1 bit toggle to indicate whether we should multiply K by the x- or y-coordinate after the final iteration.

3. CORDIC Verilog Implementation

The CORDIC algorithm was implemented in Verilog using 14 iterations. Unfortunately, although the optimal word length was found to be 25 bits, I had already implemented it with 32 or 33 bits, depending on whether signed arithmetic was occurring, and ran out of time. The implementation was a pipelined design, i.e. unfolded, and had executed two CORDIC iterations per pipeline layer.

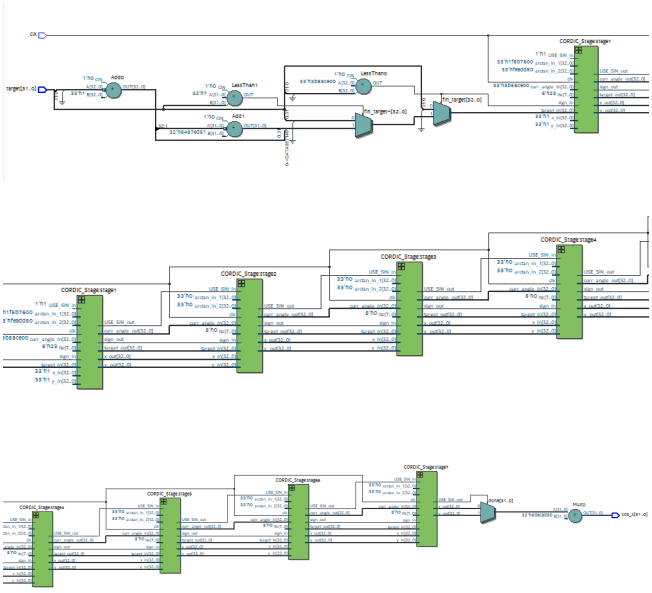

Figure 3. Top-level block diagram of the implemented CORDIC algorithm.

Figure 3 displays the 7 CORDIC stages, where each stage corresponds to two CORDIC iterations.

3.1 Hardware Optimisation

The entire design consists only of additions and right shifts, with the exception of the final multiplication by K. When the sign direction changed, the bits were inverted and 1 was added. Furthermore, when right shifting a negative number, all bits were right shifted, except the MSB - these were then concatenated back together to preserve the sign. Although there were no errors during compilation, I was unfortunately unable to do further testing due to a lack of time. The implemented design is superior to a folded design, where throughput is 1 divided by the number of iterations and latency is proportional to the number of iterations. This design is also superior to a pipelined design where only 1 iteration is completed per pipeline layer, as it take, assuming 14 iterations, double the amount of cycles to return a result.