Simulink Implementation of Gaussian Blurring, Gradient/Edge Detection, and Atan2 CORDIC

Table of Contents

[Atan2 CORDIC 3](#_Toc62480344)

[Matlab 3](#_Toc62480345)

[Section One: Special Cases 3](#_Toc62480346)

[Section Two: Setup 4](#_Toc62480347)

[Section Three: Iterative Loop 4](#_Toc62480348)

[Section Four: Data Fixing 5](#_Toc62480349)

[Verification 5](#_Toc62480350)

[Simulink 6](#_Toc62480351)

[The Model 6](#_Toc62480352)

[References 9](#_Toc62480353)

# Atan2 CORDIC

When implementing this subsystem, several steps were taken to ensure that the algorithm worked long before its implementation into Simulink and the hardware. Each step had its own verification process to ensure any bugs did not propagate to the next step and were caught as early as possible. The following sections will go into detail about each step, what it accomplished, and other nuanced details to help illustrate the full process of creating the Atan2 CORDIC IP Core.

## Matlab

The first step to this process was implementing and verifying the algorithm in Matlab. The version used during this process was 2020b, Update 1. More information regarding the various versions of the toolboxes can be found /ravven-tag/matlab/README.md. The actual implementation of the Atan2 CORDIC algorithm was simple, having four distinct sections.

### Section One: Special Cases

The first section handled any special cases where the CORDIC algorithm would typically fail.



Figure : Atan2 CORDIC Special Cases

As depicted in Figure 1, the only necessary special case was when both inputs were zero. The CORDIC algorithm would incorrectly estimate the angle and magnitude, returning a number close to π for the angle. This is because atan2 is undefined when the inputs are both zero. By default, undefined behavior should return zero. Other special cases can be included and are summarized in Figure 2.

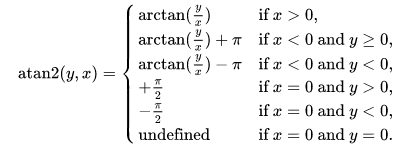


Figure : Atan2 Special Cases [1]

The special cases with defined behavior were omitted because this algorithm would eventually be implemented in hardware. Having a simpler pipeline with a few exceptions will have a smaller footprint on hardware resources.

### Section Two: Setup

The second section of the CORDIC algorithm was the setup. Figure 3 portion of the code held any constants and ran the first few starting computations. The constant *K* is introduced here but will be explained during the last section of the process. This section also introduces the function, *zsign*,



Figure : Atan2 CORDIC Setup Code



Figure : ZSign Helper Function

illustrated in Figure 4, that will treat zero as a positive number. The complexity of imitating a hardware algorithm in software is the way zero and negative numbers are treated. In binary, there is no negative bit, and multiplying by negative one can be simulated as negating the bits and adding one. In software, especially with floating point precision, negating a number requires only flipping the sign bit. Regardless, Figure 3 illustrates the four primary computations required to start the CORDIC algorithm, two of which advances the angle computation with two steps.

### Section Three: Iterative Loop

The third portion of this algorithm was the iterative part. Figure 5 illustrates the primary component of the algorithm that can be pipelined. Each of the three computations for *x*, *y*, and *z* can be computed in parallel. This code depicts the algorithm being hardcoded as ten additional iterations making the CORDIC algorithm a depth of twelve.

Figure : Atan2 Iterative Code

### Section Four: Data Fixing

The final portion of this algorithm was the optional massaging of the data so it would be ready to be used in the CPU. Figure 6 depicts the magnitude as needing to be scaled by *K* and the angle needing to be inverted.



Figure : Atan2 Data Fixing Computations

### Verification

The verification process for the algorithm as an exhaustive test, providing stimuli of all possible inputs to the algorithm. It was anticipated that the atan2 CORDIC algorithm would only be provided an input from -1 to 1. The atan2 CORDIC algorithm was verified against Matlab’s built-in atan2 function for angle comparison and the magnitude computation described in Equation 1.

Equation : Magnitude Equation

From there, the percent error as depicted in Equation 2 was computed for the magnitude analysis. This provided the relative error between the two.

Equation : Percent Error Equation

The angle comparison could not utilize any traditional relative error algorithm as the *actual* answer from Matlab’s atan2 function can be zero. Having zero as an actual answer for any of the relative error algorithms pushes them two the bounds, even if the error is small. As a result, Equation 3 was used instead.

Equation : Angle Relative Error Equation

Although Equation 3 does not provide any percentage, a minimal difference between Matlab’s atan2 function and the atan2 CORDIC algorithm implemented still needs to be small. These results are summarized in Table 1.

Table : Analysis Results of Baseline vs Atan2 CORDIC

|  |  |  |
| --- | --- | --- |
| Category | Angle | Magnitude |
| MSR | 5.594433048696700e-04 | 1.057732796575861e-06 |
| Max Error | 9.7623e-04 | 2.4354e-06 |
| Min Error | 0 | 5.2876e-07 |

As depicted in Table 1, the error between the baseline functions and the CORDIC approximations are miniscule when dealing with double floating-point precision. This algorithm was not tested within the scope of fixed point as it would be more time consuming to do it in Matlab than in Simulink. Simulink also provides the possibility of seeing the expansion of the fixed-point numbers of in-between steps for further optimization if necessary.

## Simulink

This section will go into detail about taking the Matlab Atan2 CORDIC algorithm and creating a hardware version of it. One of the key benefits of taking this algorithm and pushing it through hardware is to take advantage of parallelism between stages. It becomes especially handy when performing the same calculation over images/videos that are streamed in. Using hardware, the image/video can start to be processed well before the image/video finishes streaming in. On the CPU, the image/video must be completely saved into memory before any processing can be done.

### The Model

The four sections described in the Matlab section was broken down and illustrated for the future purpose of explaining the rationale behind the various subsystems implemented in Simulink. The entire system can be seen in Figure 7. The initial version of the model left everything as double floating-point precision. This was to ensure the structure of the model was working before complicating it with any fixed-point changes.

To break down the model, there are two distinct paths: a special cases path and the CORDIC algorithm path. As mentioned before, the special cases path covers the undefined behavior that atan2 algorithm exhibits when both inputs are zero. This path is selected when that case is true, thus setting the output to zero when necessary. Referring to Matlab, the *Setup*, *Iterative* *loop*, and *Data Fixing* portions are the standard steps. Each step is pipelined as depicted by the *z-1* blocks. Each subsystem should be considered a cloud of logic when viewing it in terms of a hardware system. Though not in its own subsystem, the *Data Fixing* portion is the final gain labeled *K* and the unary inversion prior to a mux which combines both signals into a single bus.

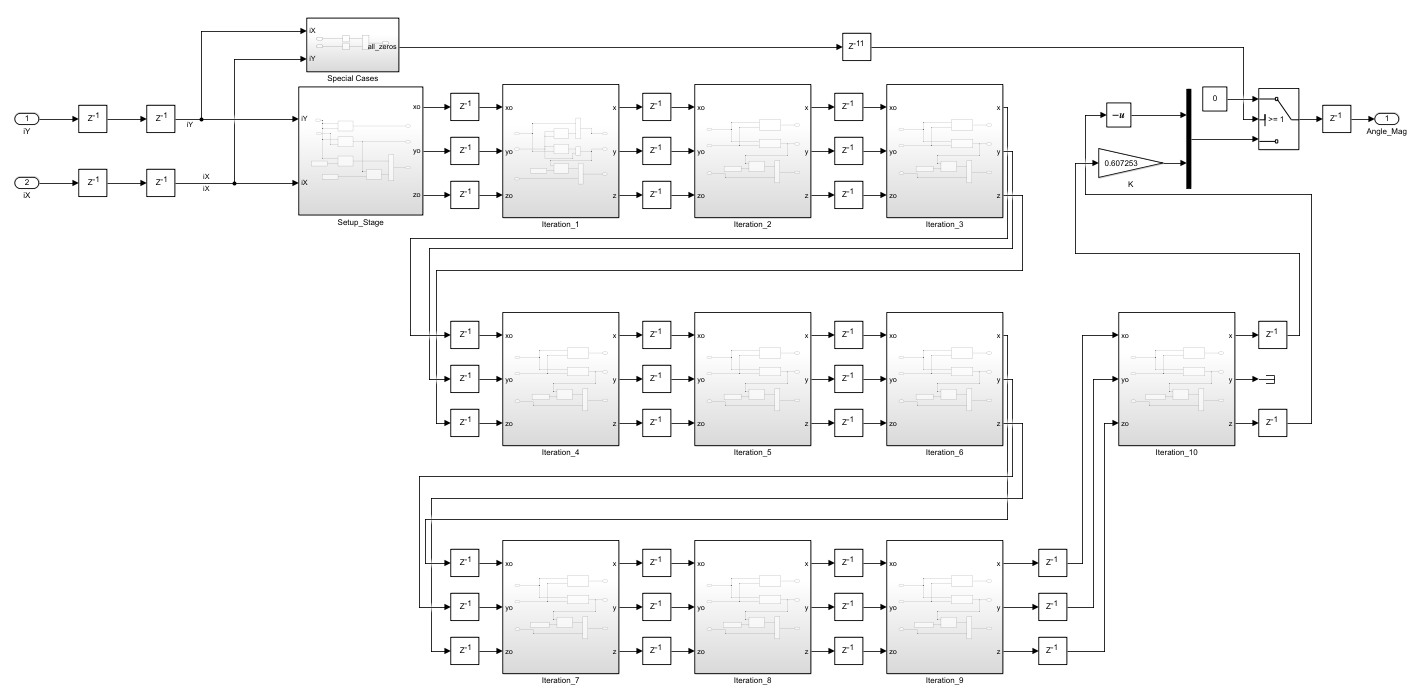


Figure : Entire Simulink Model (V1)

The inner portions of the model were simpler than their code counter parts. Most of the complexity lies within the *zsign* function as depicted in Figure 4. Due to the advantage of being in hardware, slicing the bits and inverting them no longer required a multiplication. Figure 8a and Figure 8b depict *zsign* and its inverted counterpart, respectively.

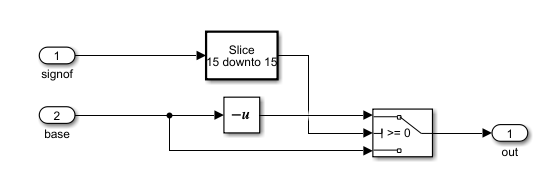
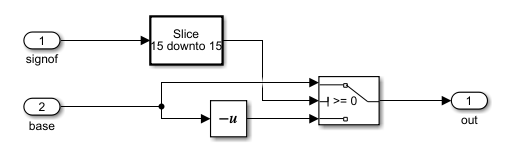


Figure : NSign (left) and Negated NSign (right) Logic Clouds

The Simulink model illustrates that the *zsign* function becomes nothing more than a multiplexer choosing the positive or negative version of the number, using the most significant bit (MSB) of the number. In this case, a slice of ‘15 downto 15’ was used since the anticipated word length was 16. The following figures breaks into each iteration block, all mimicking the code depicted in the previous section.

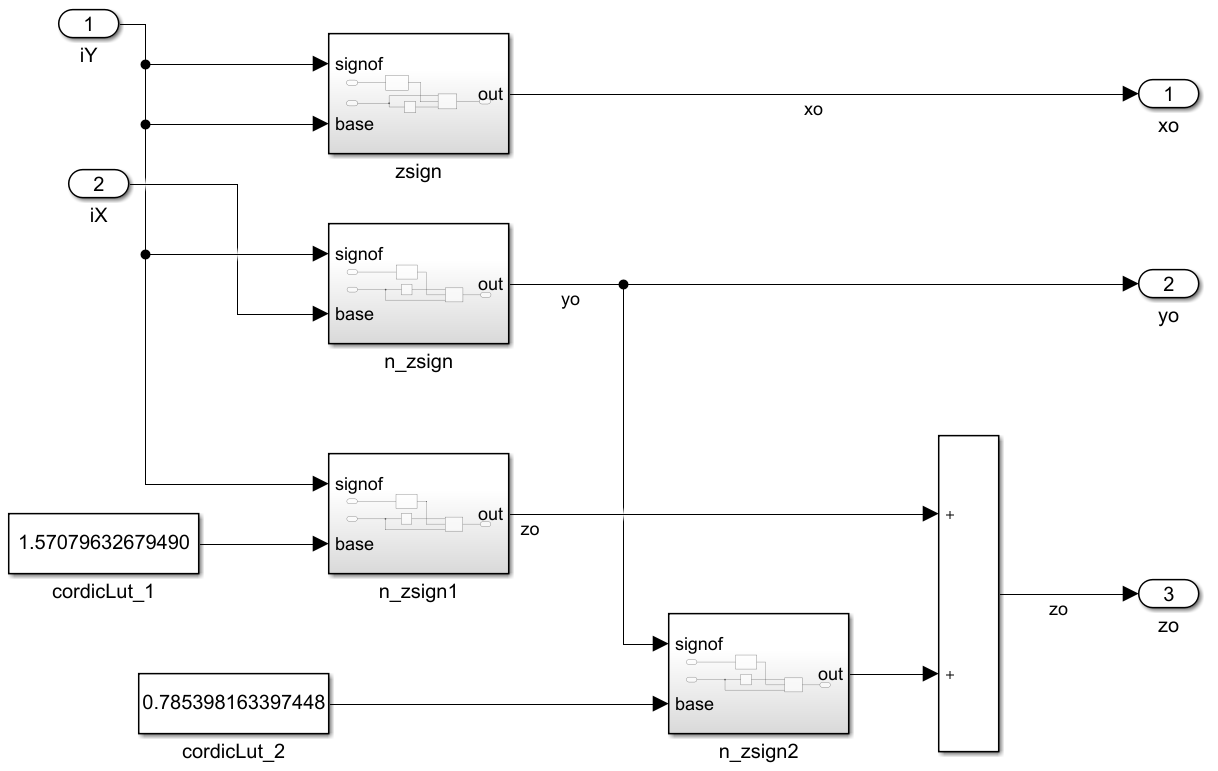


Figure : Simulink Setup Block

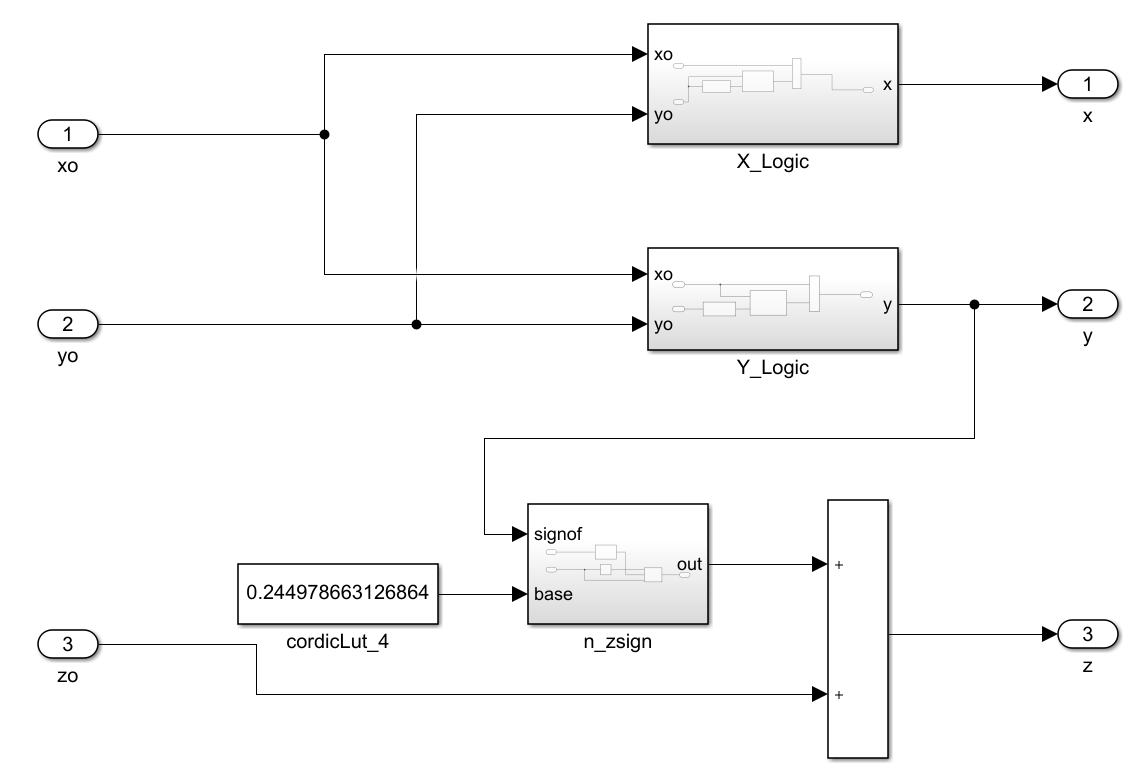


Figure : Simulink Iteration Block

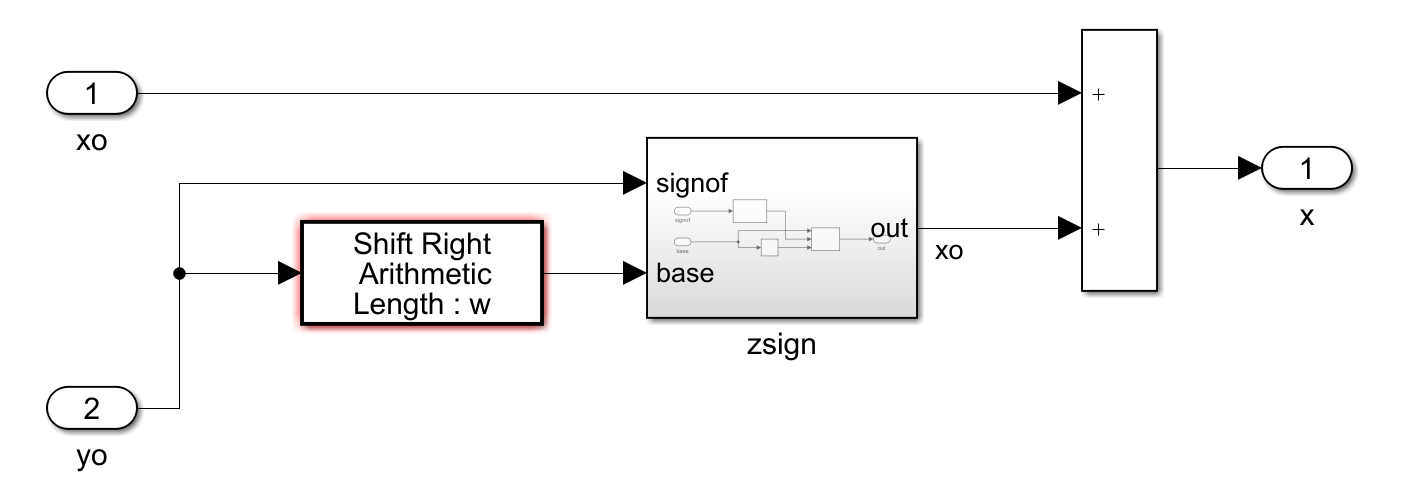
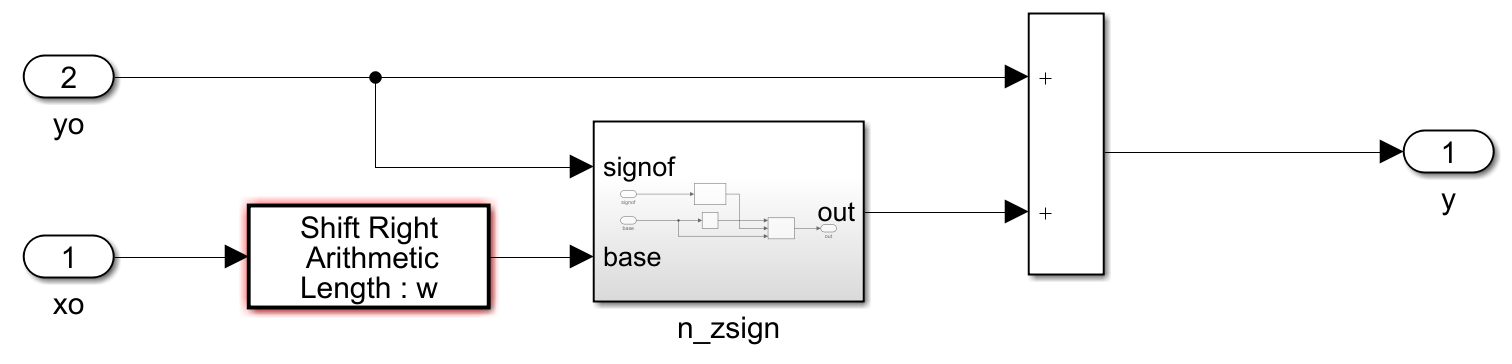
 

Figure : X (left) and Y (right) Logic Clouds

Figure 10 illustrates typically how each iteration block is organized. Of course, with each iteration, there will be a different *cordicLut* constant and a different arithmetic shift right length (w is left there to indicate the iteration block minus one). For example, the first iteration block omits the arithmetic shift entirely as it would be a shift of zero, indicating to do nothing. The primary idea behind these logic clouds is to have as little logic between each pipelined stage as possible. This will maximize the frequency the FPGA can move through each pipelined stage, and evidently reduce the potential chance for error between stages.

# References

|  |  |
| --- | --- |
| [1] | Wikipedia, "Atan2," Wikipedia, 14 Decemeber 2020. [Online]. Available: https://en.wikipedia.org/wiki/Atan2. [Accessed 24 January 2021]. |