

UNIVERSITÀ DI PISA

## CORDIC: Cartesian to Polar Coordinate Transformation

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Specification . . . . .	3
1.2	Circuit Applications . . . . .	3
<b>2</b>	<b>Architecture</b>	<b>5</b>
2.1	Data Representation . . . . .	5
2.2	Phase Precision and Error Analysis . . . . .	6
<b>3</b>	<b>VHDL code</b>	<b>7</b>
3.1	Cordic . . . . .	7
<b>4</b>	<b>Verification and testing</b>	<b>13</b>
4.1	Testbench . . . . .	13
4.2	Verification . . . . .	15
4.3	Error verification . . . . .	15
<b>5</b>	<b>Synthesis and Implementation</b>	<b>19</b>
5.1	Vivado Design flow . . . . .	19
5.2	RTL . . . . .	19
5.3	RTL Elaboration . . . . .	19
5.4	Synthesis and Implementation . . . . .	19
<b>6</b>	<b>Vivado results</b>	<b>20</b>
6.1	Critical Path . . . . .	20
6.2	Utilization Report . . . . .	20
6.3	Power Report . . . . .	20
<b>7</b>	<b>Final considerations</b>	<b>21</b>

# 1 Introduction

## 1.1 Specification

It is required to design a digital circuit for implementing the transformation from cartesian coordinates into polar coordinates using the CORDIC algorithm in Vectoring mode. It is implemented with these recursive equations:

$$\begin{aligned}x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \arctan(2^{-i})\end{aligned}$$

where  $d_i = -1$  if  $y_i > 0$ ,  $+1$  otherwise. After  $n$  iterations, the equations converge to:

$$\begin{aligned}x_n &= A_n \cdot \sqrt{x_0^2 + y_0^2} \\y_n &= 0 \\z_n &= z_0 + \arctan\left(\frac{y_0}{x_0}\right) \\A_n &= \prod_{i=0}^n \sqrt{1 + 2^{-2i}}\end{aligned}$$

## 1.2 Circuit Applications

The CORDIC (COordinate Rotation DIgital Computer) algorithm is an iterative method for performing vector rotations and solving mathematical functions such as trigonometric, hyperbolic, exponential, logarithmic, and square root operations. It was introduced by Jack E. Volder in 1959 to simplify the computation of these functions in hardware with limited resources.

CORDIC is widely used because it eliminates the need for multiplication and division, relying instead on shift and addition operations. This makes it highly efficient for

hardware implementations, particularly in devices with limited computational power, such as embedded systems, calculators, and digital signal processors (DSPs).

For example the Intel 8087, a floating-point coprocessor introduced in the early 1980s, utilized the CORDIC algorithm to perform efficient trigonometric and hyperbolic computations, such as sine, cosine, and arctangent, without relying on hardware multipliers. Similarly, during the Apollo Lunar Module missions, a precursor concept to CORDIC was employed in the Apollo Guidance Computer (AGC) to perform real-time navigation calculations. The AGC used iterative methods to determine angles and distances for lunar landings, efficiently converting Cartesian spacecraft coordinates to polar forms to ensure precise trajectory adjustments during descent.

The CORDIC algorithm can operate in two different modes: **rotation mode** and **vectoring mode**

- **Rotation mode:** rotates a vector by a specified angle, used for calculating trigonometric functions or vector transformations
- **Vectoring mode:** determines the magnitude and angle of a vector, useful for converting Cartesian to polar coordinates

In this context, this project will focus on developing the vectoring mode of the CORDIC algorithm to convert Cartesian coordinates into polar form.

## 2 Architecture

By default, the CORDIC algorithm converges only for input angles within the range  $(-99.7^\circ, 99.7^\circ)$ . To address this limitation and enable the algorithm to handle arbitrary input angles, we introduced an **initial correction step**. This step adjusts the input angle to bring it into the principal range of the algorithm. The adjustment ensures that the CORDIC algorithm works seamlessly for all input angles, not just those within the default convergence range.

$$x_0 = -y_{\text{input}} \cdot d_{\text{input}}$$

$$y_0 = x_{\text{input}} \cdot d_{\text{input}}$$

$$z_0 = -\frac{\pi}{2} \cdot d_{\text{input}}$$

Additionally, since the iterative process of the CORDIC algorithm introduces a scaling factor  $A_n$ , we normalize the final result by dividing  $\rho$  (the magnitude) by  $A_n$ .

### 2.1 Data Representation

To implement the CORDIC algorithm, we used fixed-point arithmetic for the input and intermediate values. Specifically:

- For  $x, y, \rho$  we used 16-bit fixed-point representation with 8 bits allocated for the fractional part.
- For  $z$  and  $\theta$  we used a 16-bit fixed-point representation with 13 bits allocated for the fractional part. This decision was made because they always lie within the range  $[-\pi, \pi]$ , and the additional fractional bits ensure high angular precision.
- For the intermediate calculations a 24-bit representation was used to minimize truncation errors during the iterative process.
- All the inputs and the outputs are limited to 16 bits due to the I/O pin constraints of the Zybo board, which restricts the width of the data buses and necessitate uniform bit-width for inputs and outputs.

The number of iterations for the CORDIC algorithm was set to 16, as this provides a high level of precision while balancing computational efficiency. Beyond 16 iterations,

the precision gained decreases significantly.

## 2.2 Phase Precision and Error Analysis

Focusing on the phase, the intermediate values are represented in Q3.21 fixed-point format, offering a maximum absolute error  $\epsilon_a$  of  $2^{-21}$ . Over the course of 16 iterations, the maximum accumulated absolute sum of these values  $\epsilon_M$  is

$$\epsilon_M = \epsilon_a * 16 = 2^{-21} * 2^4 = 2^{-17}$$

Since the final result is truncated to Q3.13, which involves discarding the least significant bits (LSBs), the accumulated error from the iterative sums becomes negligible. This ensures that the phase computation maintains high precision.

# 3 VHDL code

## 3.1 Cordic

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 ENTITY CORDIC IS
5
6     GENERIC (
7         N : POSITIVE := 20;
8         ITERATIONS : POSITIVE := 16;
9         ITER_BITS : POSITIVE := 4
10    );
11    PORT (
12        clk : IN STD_LOGIC;
13        rst : IN STD_LOGIC;
14        x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
15        y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
16        start : IN STD_LOGIC;
17        rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
18        theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
19        valid : OUT STD_LOGIC
20    );
21
22 END ENTITY;
23
24 ARCHITECTURE behavioral OF CORDIC IS
25
26     -- CONSTANT k : SIGNED(N - 1 DOWNTO 0) :=
27     --     to_signed(1304065748, N); -- 1/(Gain factor) multiplied by
28     --     2^N-1
29     CONSTANT k : SIGNED(N - 1 DOWNTO 0) :=
30         to_signed(INTEGER(0.6072529351031394 * (2 ** (N - 2))), N);
31         -- todo documentare meglio il N-2 (vivado si lamenta)
32         -- ((probabilmente per la dimensione massima degli integer))
33     CONSTANT HALF_PI : SIGNED(N - 1 DOWNTO 0) :=
34         to_signed(INTEGER(1.570796327 * (2 ** (N - 3))), N); --
35         -- todo documentare meglio il N-3
```

```

30  -- internal registers
31  SIGNAL x_t : SIGNED(N - 1 DOWNTO 0);
32  SIGNAL y_t : SIGNED(N - 1 DOWNTO 0);
33  SIGNAL z_t : SIGNED(N - 1 DOWNTO 0);
34
35  SIGNAL x_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
36  SIGNAL z_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
37
38  SIGNAL address : STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
39  SIGNAL atan_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
40
41  SIGNAL sign : STD_LOGIC;
42
43  -- atan table
44  COMPONENT ATAN_LUT IS
45    PORT (
46      address : IN STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
47      lut_out : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0)
48    );
49  END COMPONENT;
50
51  -- state type and registers
52  TYPE state_t IS (WAITING, FIX_STEP, COMPUTING, FINISHED);
53  SIGNAL current_state : state_t;
54
55  SIGNAL counter : UNSIGNED(ITER_BITS - 1 DOWNTO 0);
56
57 BEGIN
58  -----
59  -- INSTANTIATE ALL COMPONENTS
60  -----
61
62  -- output assignment
63  rho <= x_out;
64  theta <= z_out;
65
66  -- sign bit
67  sign <= '0' WHEN y_t > 0 ELSE
68    '1';
69
70  -- atan table
71  -- todo probably needs fix (next clock)
72  atan_lut_inst : ATAN_LUT
73  PORT MAP(
74    address => address,
75    lut_out => atan_out
76  );
77
78  -- atan table address
79  address <= STD_LOGIC_VECTOR(counter);
80

```

```

81  -- todo decidere se tenere o togliere gli assegnamenti
82  -- stupidi
83  -- todo forse z dovrebbe avere solo 2 bit interi e il resto
84  -- frazionari
85  -- aggiustare meglio spiegazione e codice per i 29 bit di
86  -- atan
87
88  -- control part
89  controllo : PROCESS (clk, rst)
90  BEGIN
91      IF (rising_edge(clk)) THEN
92
93          IF rst = '1' THEN
94              current_state <= WAITING;
95          ELSE
96
97              CASE current_state IS
98
99                  WHEN WAITING =>
100                      IF start = '1' THEN
101                          current_state <= FIX_STEP;
102                      ELSE
103                          current_state <= WAITING;
104                      END IF;
105
106                  WHEN FIX_STEP =>
107                      current_state <= COMPUTING;
108
109                  WHEN COMPUTING =>
110                      IF counter = ITERATIONS - 1 THEN
111                          current_state <= FINISHED;
112                      ELSE
113                          current_state <= COMPUTING;
114                      END IF;
115
116                  WHEN FINISHED =>
117                      current_state <= WAITING;
118                  WHEN OTHERS =>
119                      current_state <= WAITING;
120
121              END CASE;
122          END IF;
123      END IF;
124  END PROCESS;
125
126  -- operation part
127  operativa : PROCESS (clk, rst)
128  BEGIN
129      IF (rising_edge(clk)) THEN
130
131          IF rst = '1' THEN

```

```

129      valid <= '0';
130      -- Non utili
131      x_out <= (OTHERS => '0');
132      z_out <= (OTHERS => '0');
133      counter <= (OTHERS => '0');
134      x_t <= (OTHERS => '0');
135      y_t <= (OTHERS => '0');
136      z_t <= (OTHERS => '0');
137  ELSE
138
139      -- Default assignment
140      -- todo vedere se tenere o togliere
141      x_t <= (OTHERS => '-');
142      y_t <= (OTHERS => '-');
143      z_t <= (OTHERS => '-');
144      x_out <= (OTHERS => '-');
145      z_out <= (OTHERS => '-');
146      counter <= (OTHERS => '-');
147
148  CASE current_state IS
149      WHEN WAITING =>
150
151          x_t <= signed(x);
152          y_t <= signed(y);
153          z_t <= to_signed(0, N);
154          valid <= '1';
155          x_out <= x_out;
156          z_out <= z_out;
157
158      WHEN FIX_STEP =>
159          IF sign = '0' THEN
160              x_t <= y_t;
161              y_t <= - x_t;
162              z_t <= z_t + HALF_PI;
163          ELSE
164              x_t <= - y_t;
165              y_t <= x_t;
166              z_t <= z_t - HALF_PI;
167          END IF;
168
169          valid <= '0';
170          counter <= (OTHERS => '0');
171
172      WHEN COMPUTING =>
173          IF sign = '1' THEN
174              -- x_t <= x_t - y_t/(2 **
175              --      to_integer(counter));
176              x_t <= x_t - shift_right(y_t,
177              --      to_integer(counter));
178              -- y_t <= y_t + x_t/(2 **
179              --      to_integer(counter));

```

```

177          y_t <= y_t + shift_right(x_t,
178              ↳ to_integer(counter));
179          z_t <= z_t - signed(atan_out);
180      ELSE
181          -- x_t <= x_t + y_t/(2 **
182              ↳ to_integer(counter));
183          x_t <= x_t + shift_right(y_t,
184              ↳ to_integer(counter));
185          -- y_t <= y_t - x_t/(2 **
186              ↳ to_integer(counter));
187          y_t <= y_t - shift_right(x_t,
188              ↳ to_integer(counter));
189          z_t <= z_t + signed(atan_out);
190      END IF;
191
192      counter <= counter + 1;
193      valid <= '0';
194
195      WHEN FINISHED =>
196          -- x_out <=
197              STD_LOGIC_VECTOR(resize(shift_right(x_t
198                  ↳ * k , N-2),N));
199          x_out <= STD_LOGIC_VECTOR(resize(x_t * k/(2
200              ↳ ** (N - 2)), N));
201          -- x_out <= STD_LOGIC_VECTOR(x_t);
202          z_out <= STD_LOGIC_VECTOR(z_t);
203          valid <= '1';
204      END CASE;
205  END IF;
206
207  END IF;
208
209  END PROCESS;
210 END ARCHITECTURE;

```

Listing 3.1: CORDIC.vhd

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4
5 ENTITY ATAN_LUT IS
6     port (
7         address : in std_logic_vector(3 downto 0);
8         lut_out : out std_logic_vector(19 downto 0)
9     );
10    end entity;
11
12 architecture rtl of ATAN_LUT is
13
14     type LUT_t is array (natural range 0 to 15) of integer;
15

```

```
16  constant LUT: LUT_t := (
17    0 => 102943,
18    1 => 60771,
19    2 => 32109,
20    3 => 16299,
21    4 => 8181,
22    5 => 4094,
23    6 => 2047,
24    7 => 1023,
25    8 => 511,
26    9 => 255,
27    10 => 127,
28    11 => 63,
29    12 => 31,
30    13 => 15,
31    14 => 7,
32    15 => 3
33  );
34
35 begin
36
37 PROCESS (address)
38 BEGIN
39   lut_out <=
40     STD_LOGIC_VECTOR(to_signed(LUT(to_integer(unsigned(address))), 
41     20));
42 END PROCESS;
43
44 end architecture;
```

Listing 3.2: ATAN\_LUT.vhd

# 4 Verification and testing

## 4.1 Testbench

```
1 LIBRARY IEEE;
2 USE IEEE.STD_LOGIC_1164.ALL;
3 USE IEEE.NUMERIC_STD.ALL;
4 USE ieee.math_real.ALL;
5
6 ENTITY CORDIC_TB IS
7     GENERIC (
8         N : POSITIVE := 20;
9         floating : INTEGER := 10
10    );
11 END CORDIC_TB;
12
13 ARCHITECTURE Behavioral OF CORDIC_TB IS
14     -- Component declaration for CORDIC
15
16     COMPONENT CORDIC
17         PORT (
18             clk : IN STD_LOGIC;
19             rst : IN STD_LOGIC;
20             x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
21             y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
22             start : IN STD_LOGIC;
23             rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
24             theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
25             valid : OUT STD_LOGIC
26        );
27     END COMPONENT;
28
29     -- Signals for CORDIC inputs and outputs
30
31     SIGNAL clk : STD_LOGIC := '0';
32     SIGNAL reset : STD_LOGIC := '0';
33     SIGNAL x : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
34     SIGNAL y : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
35     SIGNAL start : STD_LOGIC := '0';
36     SIGNAL rho : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
```

```

37   SIGNAL theta : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
38   SIGNAL valid : STD_LOGIC := '0';
39
40   SIGNAL run_simulation : STD_LOGIC := '1';
41
42   -- Clock period definition
43   CONSTANT T_clk : TIME := 10 ns;
44
45   -- Coordinate type
46   TYPE Coordinate IS RECORD
47     x : real;
48     y : real;
49   END RECORD;
50   CONSTANT n_coordinates : NATURAL := 6;
51
52   TYPE CoordinateArray IS ARRAY (0 TO n_coordinates - 1) OF
53     Coordinate;
54
55   -- Array of coordinates to test
56   CONSTANT Coordinates : CoordinateArray := (
57     (1.0, 0.0),
58     (10.0, 10.0),
59     (0.1, 3.0),
60     (-0.1, -4.0),
61     (-1.0, 1.0),
62     (-1.0, 0.0)
63   );
64
65 BEGIN
66   -- Instantiate the CORDIC component
67   cordic_inst : CORDIC
68   PORT MAP(
69     clk => clk,
70     rst => reset,
71     x => x,
72     y => y,
73     start => start,
74     rho => rho,
75     theta => theta,
76     valid => valid
77   );
78
79   -- todo fix behavior if cordic is not ready / does not work
80
81   clk <= (NOT(clk) AND run_simulation) AFTER T_clk / 2;
82   -- Stimulus process
83   STIMULI : PROCESS
84     VARIABLE i : INTEGER := 0;
85   BEGIN
86     reset <= '1';

```

```

87      start <= '0';
88      WAIT FOR 2 * T_clk;
89
90      reset <= '0';
91
92      FOR i IN 0 TO n_coordinates - 1 LOOP
93
94          IF valid = '0' THEN
95              WAIT UNTIL valid = '1';
96          END IF;
97
98          x <=
99              STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).x
100             * 2.0 ** floating), N));
101         y <=
102             STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).y
103             * 2.0 ** floating), N));
104         start <= '1';
105
106         WAIT FOR 5 * T_clk;
107
108         start <= '0';
109
110         IF valid = '0' THEN
111             WAIT UNTIL valid = '1';
112         END IF;
113
114         wait for 10 * T_clk;
115
116     END LOOP;
117
118     run_simulation <= '0';
119
120     WAIT;
121 END PROCESS;
122 END ARCHITECTURE;

```

Listing 4.1: CORDIC\_tb.vhd

## 4.2 Verification

### 4.3 Error verification

The implementation of the CORDIC algorithm in vectoring mode for Cartesian-to-polar conversion demonstrates exceptional accuracy in both the module ( $\rho$ ) and phase ( $\theta$ ) outputs. Using a fixed-point representation of  $Q8.8$  for  $\rho$  and  $Q3.13$  for  $\theta$ , the errors were summarized as follows:

Module Error: Max = 0.003938 units, Mean = 0.001889 units

Phase Error: Max = 0.000153 rad, Mean = 0.000061 rad

The test was performed on  $512 \times 512$  input points uniformly distributed within a circle of radius  $\frac{127}{A_n} \approx 77.1211$ . This range ensures a comprehensive evaluation of the algorithm over a high range of valid inputs while maintaining consistency with the precision of the fixed-point formats used. The fixed-point representation employed provides a precision of  $2^{-8}$  for  $Q8.8$  (module) and  $2^{-13}$  for  $Q3.13$  (phase).

Module Error:  $\log_2(\text{Max error}) \approx -7.988$

Phase Error:  $\log_2(\text{Max error}) \approx -12.674$

The observed errors are well within these precision limits, with all measured deltas smaller than the respective numerical precision of the formats. Any discrepancies between the input and output values are solely due to the conversion of these fixed-point representations into real numbers for plotting and analysis.

The 3D plots of module and phase errors visually confirm the bounded and negligible nature of the errors across all tested input combinations. Assuming no error on the input, this analysis validates that the algorithm introduces no error in the output, demonstrating the precision and robustness of CORDIC algorithm thanks to the 24 bits internal logic.

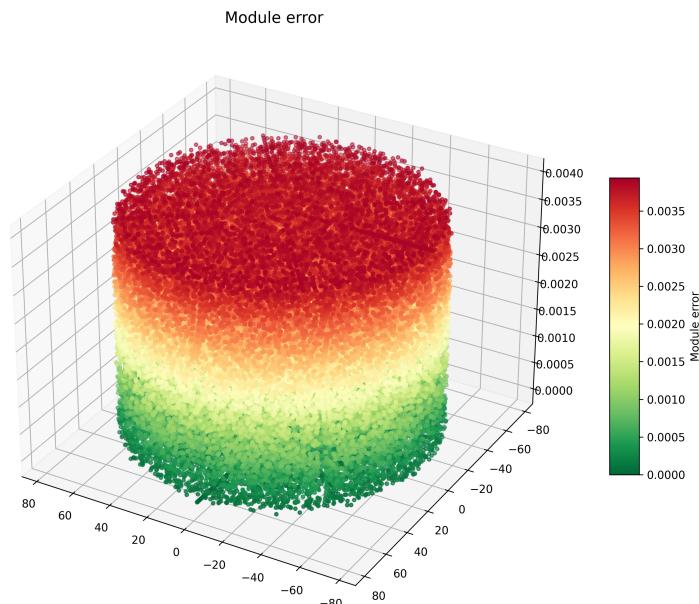


Figure 4.1: 3D plot of module error ( $\rho$ ).

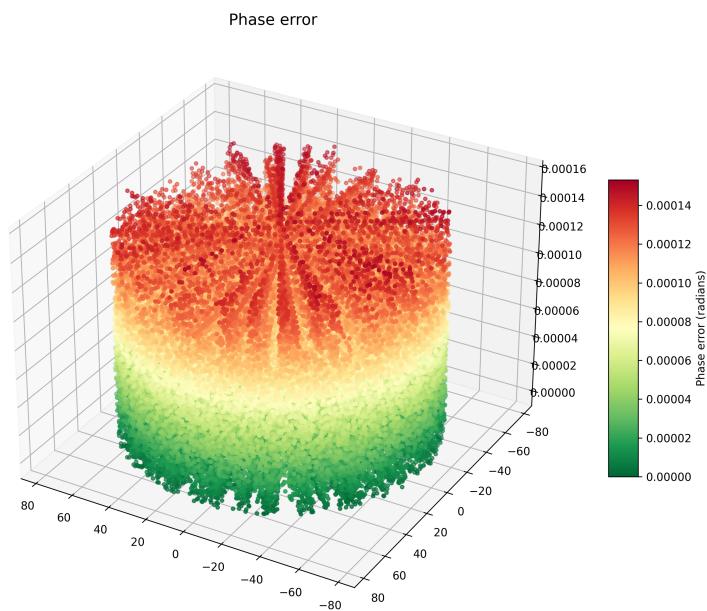


Figure 4.2: 3D plot of phase error ( $\theta$ ).

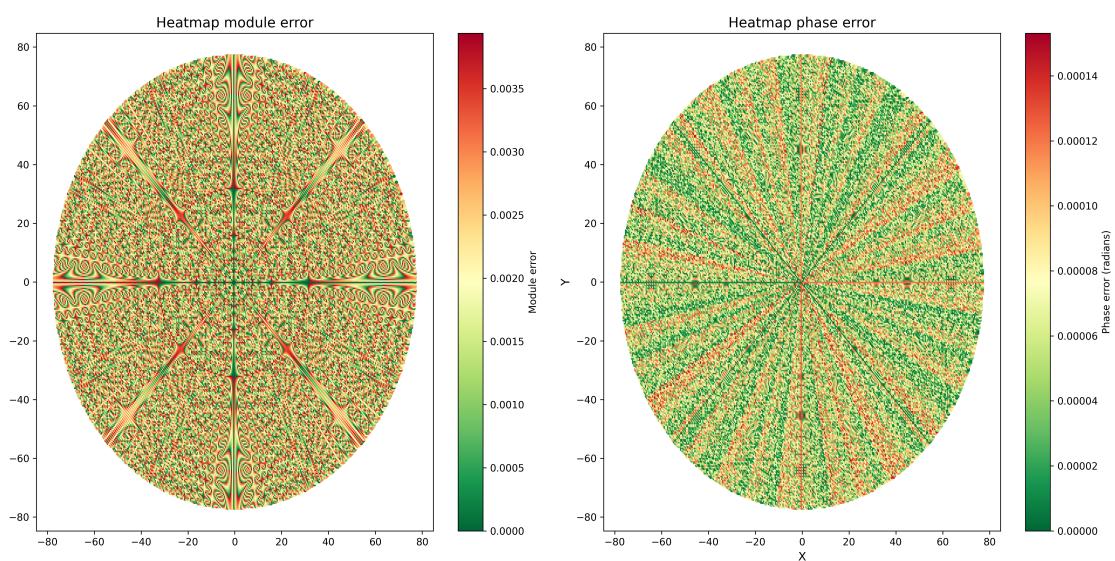


Figure 4.3: Heatmap for phase and module errors.

# **5    Synthesis and Implementation**

**5.1   Vivado Design flow**

**5.2   RTL**

**5.3   RTL Elaboration**

**5.4   Synthesis and Implementation**

# **6 Vivado results**

**6.1 Critical Path**

**6.2 Utilization Report**

**6.3 Power Report**

## **7 Final considerations**