CHAOTIC OSCILLATOR AS SOUND SYNTHESIS CONTROLLER

A Design Project Report Presented to the School of Electrical and Computer Engineering of Cornell University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by Zifu Qin
MEng Field Advisor: Van Hunter Adams, Bruce R. Land
Degree Date: December 2022

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Chaotic Oscillator as Sound Synthesis Controller

Authors: Zifu Qin

Abstract: In some circumstances, human ears are better than eyes at recognizing patterns. Researchers have sonified complex datasets, including DNA sequences, to use their ears to find patterns hidden from their eyes. By sonifying the well-known chaotic system – "Lorenz System", this project qualitatively investigates the ear's response to a sonified chaotic system. To sonify the Lorenz System, the team built a microcontroller-based chaotic synthesizer which could generate sounds with chaotic patterns. In this project, the source of chaos was from modulated Lorenz Attractors which were mapped to the output frequencies of a Direct Digital Synthesis sine-wave synthesizer. The RP2040 microcontroller was used to implement the algorithm and send the generated digital frequency signals to a digital to analog converter to make sounds. This project was an exploratory experiment of sonifying a chaotic system. We found that the sonified chaotic system sounded vaguely natural and organic.

Table of Contents

1 Executive Summary

In some circumstances, human ears are better than eyes at recognizing patterns. Researchers have sonified complex datasets, including DNA sequences, to use their ears to find patterns hidden from their eyes. By sonifying the well-known chaotic system – "Lorenz System", this project qualitatively investigates the ear's response to a sonified chaotic system. To sonify the Lorenz System, the team built a microcontroller-based chaotic synthesizer which could generate sounds with chaotic patterns. In this project, the source of chaos was from modulated Lorenz Attractors which were mapped to the output frequencies of a Direct Digital Synthesis sine-wave synthesizer. The RP2040 microcontroller was used to implement the algorithm and send the generated digital frequency signals to a digital to analog converter to make sounds. This project was an exploratory experiment of sonifying a chaotic system. We found that the sonified chaotic system sounded vaguely natural and organic.

This project divides into two phases. The first phase involved implementing and testing a numeric integrator and a Direct Digital Synthesis synthesizer in both MATLAB and C Programming. I tested these algorithms independently, and then in combination. In the second phase of the project, I implemented these algorithms on a microcontroller for realtime audio synthesis.

The objective for the first phase of the project was to build a non-realtime software simulation for the chaotic synthesizer to serve as the foundation for realtime microcontroller-based prototype. The non-real time software simulation includes an Euler-integrator for the Lorenz System, a Direct Digital Synthesis (DDS) sin-wave synthesizer, and combination of the two in which the integrator sets the target frequency for the synthesizer, thus "sonifying" the Lorenz System. I first implemented these algorithms in MATLAB for easy debugging, and then in C for implementation on the microcontroller.

The objective for the second phase of the project was to build a RP2040-based chaotic sound synthesizer and explore the chaotic sounds that it generated. This prototype included a number of components, including the Raspberry Pi Pico (a breakout board for the RP2040), the MCP4822 digital to analog converter (DAC), an audio jack, and a set of speakers. The RP2040 implemented the integrator and synthesizer in realtime, and communicated voltages to the DAC via an SPI channel. The output of the DAC was then connected to both a speaker and an oscilloscope, enabling both visualization and sonification of the Lorenz System.

2 Introduction

Researchers are familiar with "Visualization" which allows us to use our eyes to discover patterns in datasets. For example, sounds can be visualized as spectrograms or voicegrams. However, in some circumstances, human ears are better than eyes at recognizing patterns. But how good are the ears at recognizing a chaotic system? This project aimed to find out by sonifying a particularly famous chaotic system: the Lorenz System. "Lorenz Butterfly" is a famous plot which describes the behavior of Chaotic Lorenz system. The patterns of chaotic behaviors can be clearly visualized by plots. But what if people want to hear the Chaotic Lorenz system, what does this "butterfly" sound like? This exploratory experiment will sonify the Chaotic Lorenz System and experience chaos with a different sense.

3 Background

3.1 Lorenz System

A chaotic system is one for which small differences in the initial state lead to huge differences in future states of the system. The Lorenz System is a famous example of Chaos theory. It is composed of three coupled ordinary differential equations (ODEs) (Eqn. 1):

$$\begin{aligned} dx/dt &= \sigma(y-x) \\ dy/dt &= x(\rho-z)-y \\ dz/dt &= xy - \beta z \end{aligned} \tag{1}$$

The *x, y, z* in equation (1) represents the coordinates of a certain state. The σ, ρ, β are the system parameters of the Lorenz System. And the left-hand side of each ordinary differential equation in equation (1) shows the slope of each coordinate under a certain state. The Lorenz Attractor represents solutions/behaviors of the Lorenz System. It can be plotted as a "Butterfly" shape shown in Figure 1, when system parameters σ, ρ, β are equal to 10, 28, and 8/3 respectively.

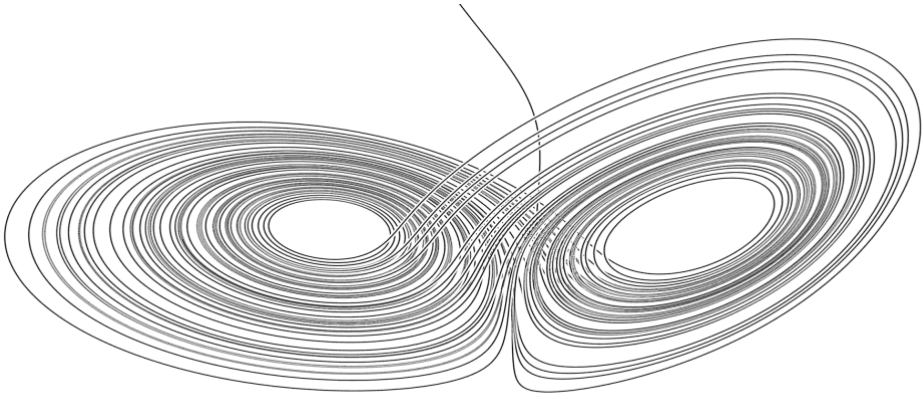


Figure 1: The Lorenz Attractor plot when system parameters σ, ρ, β are equal to 10, 28, and 8/3 respectively

3.2 Direct Digital Synthesis

Direct Digital Synthesis is an algorithm which generates sine waves of specific frequencies, Direct Digital Synthesis can give very accurate frequencies as expected. In my experiment, the accuracy could be as high as 99.8%. For sound samples with sampling frequency of 44kHz, 32-bit should be used to guarantee small enough resolutions (around $1.02 \times 10^{-5}\ Hz$) for good sound quality. The output frequency $\boldsymbol{f_{out}}$ is mathematically expressed as:

$$\boldsymbol{f_{out}} = \frac{\boldsymbol{f_s}}{\boldsymbol{2^{32}}} \cdot \boldsymbol{N}, \tag{2}$$

where *N* is the increment amount.

4. Theoretical Simulation

This project divides into two phases. The first phase involved implementing and testing a numeric (Euler) integrator and a Direct Digital Synthesis synthesizer in MATLAB and C. I tested these algorithms independently, and then in combination. In the second phase of the project, I implemented these algorithms on a microcontroller for realtime audio synthesis. The testing results in the first phase would serve as the foundation for algorithm implementations on a microcontroller in the second phase.

4.1 Lorenz System simulation in MATLAB

As mentioned in Section 3, the Lorenz System is a system of three coupled ordinary differential equations. And the behavior of the Lorenz System's state variables can be plotted as a "Butterfly" shape, as the system coefficients are $\sigma = 10, \rho = 28,$ and $\beta = 8/3$. Therefore, the general idea of this section was to implement the integrator for the Lorenz System in MATLAB and reproduce the "Butterfly" shape plot of the Lorenz Attractor. Next, the simulation and implementation will be discussed section by section in detail.

4.1.1 Create the Lorenz function

The objective of Section 4.4.1 was to reproduce the ordinary differential equation system shown in Eqn. 1. Figure 2 shows the code for this section.

```
function dx = lorenz(t,xyz,Coef)
dx = [
    Coef(1)*(xyz(2)-xyz(1));
    xyz(1)*(Coef(2) - xyz(3)) - xyz(2);
    xyz(1)*xyz(2) - Coef(3)*xyz(3);
];
end
```

Figure 2: Create a function of the Lorenz System.

A function was created in MATLAB using the syntax "*function [y1, ..., yN] = myfun(x1, ..., xM)*". The name of function I created is "*lorenz*". "*t*", "*xyz*", and "*Coef*" are all the inputs of the ordinary differential equation system. "*xyz*" is the coordinates of a certain state. It has three elements, which are *xyz(1)* = *x*, *xyz(2)* = *y*, and *xyz(3)* = *z*. Both "*t*" and "*Coef*" would be defined later. "*dx*" is the output of the ordinary differential equation system. After reproducing this ordinary differential equation system, I would define some necessary parameters for the integrator implementation.

4.1.2 Define parameters

To implement the integrator for the Lorenz System and plot the Lorenz Attractor, a few

more parameters need to be defined. Figure 3 shows the code for this section.

```
Coef = [10, 28, 8/3];
xstart = [0, 1, 20];
dt = 0.001;
tspan = dt:dt:100;
```

Figure 3: all the necessary parameters for the Lorenz System simulation.

"Coef", "xstart", "dt", and *"tspan"* were defined in this section. "*Coef*" represents the system coefficients of the Lorenz System. It has three elements, including $Coef(1) = \sigma = 10$, $Coef(2) = \rho = 28$, and $Coef(3) = \beta = 8/3$. "*xstart*" is the initial condition of the system state variables *x, y,* and *z*. So initially, $xyz(1) = x = 0$, $xyz(2) = y = 1$, and $xyz(3) = z = 20$. "*xstart*" will only affect where does the system start running, in other words, initial positions/conditions of the Lorenz System. "*dt*" is the time step between each state, and "*tspan*" shows the start time, time step, and end time for this Lorenz System simulation.

4.1.3 Implement the integrator and plot the Lorenz Attractor

After creating the function and defining all the necessary parameters for this Lorenz System simulation, I would implement the integrator for the Lorenz System and plot the Lorenz Attractor. Figure 4 shows the code for this section.

```
[t,xyz] = ode45(@(t,xyz)Lorenz(t,xyz,Coefficient),tspan,xstart);
plot3(xyz(:,1),xyz(:,2),xyz(:,3),'r','LineWidth',0.5);
grid on
box on
axis equal
```

Figure 4: Solve ODEs by ode45 and plot the Lorenz Attractor.

The entire project development started with the Lorenz System simulation in MATLAB. The objective of this Lorenz System simulation in MATLAB was to implement the integrator correctly. I implemented the integrator using the "*ode45*" function of MATLAB. "*ode45*" is a MATLAB build-in function solving ordinary differential equations. Then, to plot the Lorenz Attractor, I plotted all the state variables *x, y, z* together using the "*plot*" function of MATLAB. Figure 5 shows the plot of the Lorenz Attractor. The last three lines of the code in Figure 3 can give a clearer view of the plot by applying the grid, box, and axis.

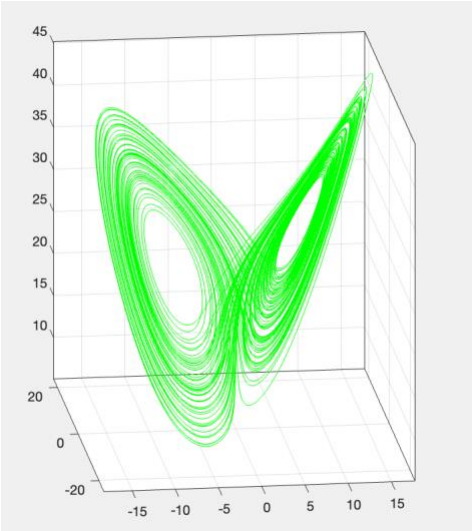


Figure 5: Plot the Lorenz Attractor in 3D.

It was shown that the plot was a beautiful "butterfly" shape as expected. The system state variables *x, y,* z moved with time, and the trace of the movement formed the "butterfly" shape. So, the integrator was correctly implemented in this section. The behavior of the Lorenz System was clearly simulated and plotted.

4.2 Euler Method simulation in MATLAB

In the last section, the Lorenz System integrator was correctly implemented using the MATLAB build-in "*ode45*" function. Even though the "*ode45*" function could implement the integrator more conveniently, it is way harder to implement on the RP2040 microcontroller. Instead of using the "*ode45*" function, I would implement a numeric integrator for the Lorenz System using the Euler Method since the Euler Method is much easier to implement on the RP2040 microcontroller. Euler Method is a numeric approach to solve ordinary differential equations The Euler Method starts with an initial state, and numerically calculates the slope of each coordinate *x, y, z* under that according to its ordinary differential equations. Then, the next state can be calculated as the sum of the initial state and the displacement during the timestep. The displacement is the product of the previous calculated slope and the timestep. Then, the future state values can be calculated by this method till the time ends. The mathematical expression (Eqn. 2) of the Euler Method is shown below:

$$\frac{dy}{dt}|_{t=t_0} = f(t_0, y_0), \quad = y_n + f(t_n, y_n) \cdot (t_{n+1} - t_n), \tag{2}$$

where y_{n+1} is the value of the next step, y_n is the value of the previous, $f(t_n, y_n)$ is the previous calculated slope at the last state, and $(t_{n+1} - t_n)$ is the timestep between these two states. So, the behavior of all the state variables *x, y, z* of the Lorenz System can be numerically calculated and simulated using the Euler Method. Also, the simulation result would be shown by several plots, including *x* vs *t* plot, *y* vs *t* plot, *z* vs *t* plot, and the Lorenz System behavior plot). *x* vs *t*, *y* vs *t*, and *z* vs *t* plots could show the independent behavior of *x, y, z* versus time respectively. The Lorenz System behavior plot should be the same as the Lorenz Attractor plot made in Section 4.1.3. The objective of this section was to implement a numeric Euler-integrator for the Lorenz System. Next, the Euler Method simulation in MATLAB would be discussed section by section in detail.

4.2.1 Define parameters

To perform the Euler Method simulation in MATLAB, a few necessary parameters need to be defined. Figure 6 shows the code for this section.

```
total_state = 1000000;
x = zeros(total_state,1);
y = zeros(total_state,1);
z = zeros(total_state,1);
t = zeros(total_state,1);

Coefficient = [10, 28, 8/3];
dt = 0.0001;
x(1) = 0;
y(1) = 1;
z(1) = 20;
```

Figure 6: all the required parameters for the "Euler Method' simulation.

The first line set up the total number of states for the Lorenz System in this simulation.

1,000,000 states were enough for this simulation. The next four lines defined and created the blank spots for states variables (*x, y, z*) and time in this simulation. These blank spots were all filled with zeros. The next five lines defined the system coefficients, a single timestep, and initial state value of the Lorenz System in this simulation. In MATLAB, array indices must start with positive numbers, therefore, the indices of the initial states were 1 instead of 0. These five lines were similar with the code in Section 4.1.2.

4.2.2 Euler-integrator implementation

To implement the numeric Euler-integrator for the Lorenz System, I need to run from the first state to the 1,000,000th state (the last state), therefore, a for loop was needed for this large implementation. Figure 7 shows the code for this section.

```
for step = 1:total_state - 1
    x(step + 1) = x(step) + Coefficient(1) * (y(step) - x(step)) * dt;
    y(step + 1) = y(step) + (-x(step) * z(step) + Coefficient(2) * x(step) - y(step)) * dt;
    z(step + 1) = z(step) + (x(step) * y(step) - Coefficient(3) * z(step)) * dt;
    t(step + 1) = t(step) + dt;
end
```

Figure 7: the for loop implementing "Euler Method."

In general, the content in this for loop is similar with the function introduced in Section 4.1.2. The only difference is that the code in Figure 7 implement the integrator by the Euler Method using the equation (2). For the first three lines in the for loop, the value of next state was calculated as the sum of the previous state and the displacement which was the product of the previous state's slope and the timestep. For example, *x(step + 1)* was the next state, *x(step)* was the previous state of *x*, *Coefficient(1) * (y(step) – x(step))* represented the precious calculated slope, *dt* was the timestep, and *Coefficient(1) * (y(step) – x(step)) * dt* was the displacement between these two states. So, *x(step + 1)* was equal to *Coefficient(1) * (y(step) – x(step)) * dt*, which exactly followed the equation (2). Next, I would discuss the simulation result by multiple plots.

4.2.3 Result and discussion

I made several plots, including *x* vs *t* plot, *y* vs *t* plot, *z* vs *t* plot, and the Lorenz System behavior plot. All these plots are shown in Figure 9. Figure 8 shows the code for this section. The code in this section was used to plot the simulation result of the Euler-integrator for the Lorenz System.

```
subplot (4,1,1); plot(t,x,'black');
xlabel('time'); ylabel('x');
subplot (4,1,2); plot(t,y,'r');
xlabel('time'); ylabel('y');
subplot (4,1,3); plot(t,z);
xlabel('time'); ylabel('z');
subplot (4,1,4); plot3(x,y,z,'g');
xlabel('x'); ylabel('y'); zlabel('z');
grid on; box on; axis equal;
```

Figure 8: the code used to plot the simulation result of the Euler-integrator for the Lorenz System.

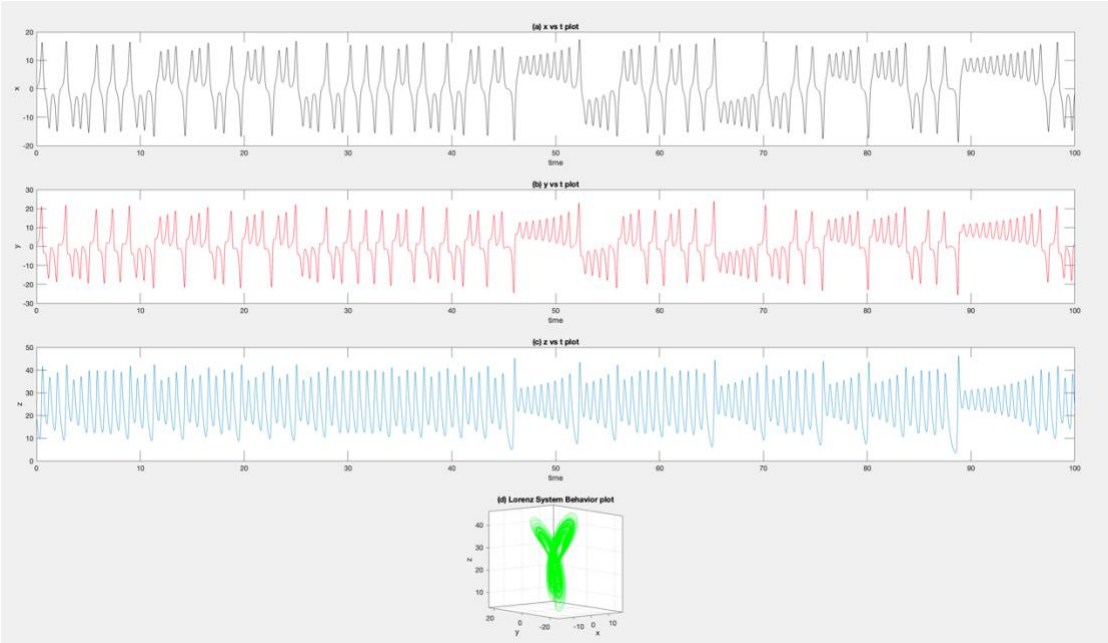


Figure 9: plots of the simulation result of the Euler-integrator for the Lorenz System*: (a) independent behavior of state variable x vs time; (b) independent behavior of state variable y vs time; (c) independent behavior of state variable z vs time; (d) the behavior of Lorenz System in 3D.*

Figure 9(d) showed the behavior of Lorenz System in 3D, which was exactly the same the simulation result in Section 4.1.3. It proved that the Euler Method could did the same job as ode45 to implement the integrator for the Lorenz System. Figure 9(a), 9(b), and 9(c) demonstrated the independent behavior of *x, y, z* versus given time respectively. From the plots, it was seen that the behavior of state variable *x* and *y* showed some similarities, but they were not exactly the same. However, the behavior of state variable *z* was quite different from the behavior of state variable *x* and *y.* Both similarities and differences were reflected in the 3D plot of Lorenz System's behaviors. Also, the similarities and differences would also be reflected in the sonification in later sections.

4.3 Direct Digital Synthesis MATLAB simulation

As mentioned in the Section 3, Direct Digital Synthesis is an algorithm which generates sine waves of specific frequencies. I could set up my own output frequencies, and a Direct Digital Synthesis synthesizer would produce very accurate frequencies as expected. In this section, I would simulate the Direct Digital Synthesis algorithm in MATLAB, the MATLAB code would be discussed section by section in detail.

4.3.1 Defining parameters

A few necessary parameters need to be defined for the Direct Digital Synthesis algorithm simulation in MATLAB. Figure 10 shows the code for this section.

```
sinetable = sin(2 * pi * (0:255)./256);
sin_array = zeros(1,256);
constant = 2^32;
fs = 44000;
f_out = 440;
increment = (f_out/fs) * constant;
accumulator = 0;
```

Figure 10: define parameters for Direct Digital Synthesis MATLAB simulation.

The first line created a sine table containing 256 values. These 256 values were calculated by equally dividing a period of sine wave into 256 pieces. The second line created a blank 256-size array used to be fill the indexed amplitudes. The rest of code in this section defined necessary parameters for the Direct Digital Synthesis algorithm simulation, including 2^{32} accumulator units (*constant*), sampling frequency (*fs*), output frequency (*f_out*), increment amount (*increment*), and the initial accumulator value (*accumulator*). The sampling frequency of 44kHz is usually used to avoid aliasing when synthesizing sounds, since the Nyquist frequency is larger than the maximum frequency (20kHz) human beings can hear. With a sampling frequency of 44kHz, I used 2^{32} accumulator units to make the accumulator be accurate enough, since 2^{32} accumulator units could give me a resolution of 1.02×10^{-5} *Hz* which ensured the accumulator accuracy. A 256-size sine table was enough for this simulation. The increment defined in the second last line followed the equation (3). The accumulator was defined to start with 0. Then, I set up the output frequency to be 440Hz, this number could be any arbitrary positive frequencies.

4.3.2 Direct Digital Synthesis algorithm implementation

I had set up a blank 256-size array used to fill the indexed amplitudes, so it means that the algorithm would run 256 times to fill every blank spot in the array, so I would need a for loop to implement the Direct Digital Synthesis algorithm simulation. Figure 11 shows the code for this section.

```
for i = 1:256
    accumulator = mod((accumulator + increment),constant);
    index = accumulator./2^24;
    sin_array(i) = sinetable(cast(index,'uint32') + 1);
end
```

Figure 11: implementation of the Direct Digital Synthesis simulation.

"i" was defined to start from 1 to 256 since the size of the blank array was 256. The first line in the for loop updated the accumulator for each implementation. The "*mod*" function was used to get the remainder as overflow happened. The "*index*" was defined to be the result of 24-bit left-shifting the accumulator, since the most significant 8 bits of the accumulator are usually recommended to be used to index into the sine table. The last line in the for loop was used to fill the indexed amplitudes into the blank array. The for loop would iterate 256 times to fill the blank array, then the Direct Digital Synthesis algorithm MATLAB simulation was done.

4.3.3 Results and discussion

To visualize the simulation result, I plotted it by using the code shown in Figure 12.

```
plot(sin_array)
xlabel("Index");
ylabel("Amplitude");
```

Figure 12: plotting the simulation result.

To create a 2D plot, there are usually two elements in the "*plot*" function in MATLAB. But I only put one element ("*sin_array*") in the "*plot*" function since the x-axis was index in default. After implementing the code in Figure 12, I would get the simulation result shown in Figure 13.

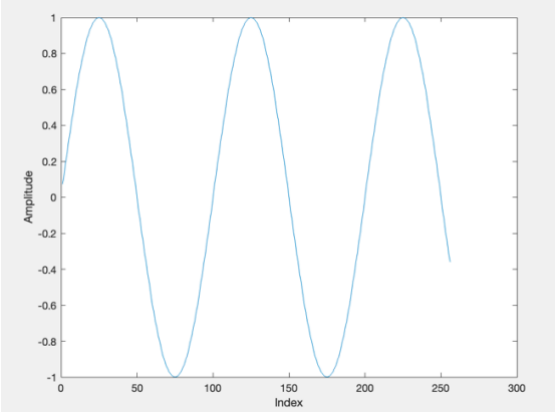


Figure 13: the Direct Digital Synthesis algorithm simulation result.

It was noted that the x-axis was the index number, and the y-axis was the amplitudes in the array. These amplitudes in the array were mapped from the 256-size sine table. The sine wave stopped at 256 since the size of the array was only 256. It was reasonable that the simulation result gave me a 440Hz sine wave which was equal to the output frequency I initially assigned.

4.4 Rewrite the MATLAB simulation in C

So far, I had done all algorithm implementations/simulations in MATLAB. Both the Euler-integrator for the Lorenz System and the Direct Digital Synthesis synthesizer were implemented correctly. However, the objective of this project was to sonify the Lorenz System by implementing algorithms on the RP2040 microcontroller in realtime. It means that the algorithm had to be written in C. Therefore, I would rewrite all the previous MATLAB implementations in C.

4.4.1 Header files

The start of the C program should be header files needed for this program. In this C program, I would include two header files – "*stdio.h*" and "*math.h*". "*stdio.h*" defined variable types and various function I would need. And "*math.h*" would provide various mathematical functions I would need for this C program.

4.4.2 Sine table setup

This entire C program could be written in a single main. Then, the first section of the main was to create a sine table containing all the 256 amplitude values for the Direct Digital Synthesis synthesizer. I could not develop the Direct Digital Synthesis synthesizer without creating a sine table first. Figure 14 shows the code for this section.

```
 9   int main(int argc, const char * argv[]) {
10       double sintable[256] = {0};
11       for(int i = 0; i <256; i++){
12           sintable[i] = sin(2*PI*i/256);
13       }
```

Figure 14: creating the sine table for DIRECT DIGITAL SYNTHESIS.

The sine table here had the same function as the sine table created in Section 4.3.1. The sine table provided indexed amplitudes for the Direct Digital Synthesis synthesizer, the only difference between the sine table here and the sine table in Section 4.3.1 was the syntax

difference between MATLAB and C. The first line in the main created a blank 256-size array used to contain all the 256 amplitude values. Then, the next for loop generated 256 values by equally dividing a period of sine wave into 256 pieces. Then, the for loop filled all these 256 values into the blank 256-size array that I just created to get the sine table ready for the Direct Digital Synthesis synthesizer.

4.4.3 Defining parameters

The objective of Section 4.4 was to combine the numeric integrator and the Direct Digital Synthesis synthesizer and rewrite the combination in C. So, I need to define parameters for both the numeric integrator and the Direct Digital Synthesis synthesizer in this C program. Figure 15 shows the C codes for this section.

```
15      int constant = (int)pow(2, 32);
16      double fs = 44000;
17      unsigned int index;
18      unsigned int accumulator = 0;
19      double sin1[100000] = {0};
20
21      FILE* file = fopen("/Users/zifuqin/Desktop/final_report_test_1/output.txt", "w");
22      if(file){
23
24          double x = 0, y = 1, z = 20;
25          double sigma = 10.0;
26          double rho = 28.0;
27          double beta = 8.0 / 3.0;
28          double timestep = 0.001;
29          int m = 100;
```

Figure 15: defining parameters for both Lorenz System and DIRECT DIGITAL SYNTHESIS.

I had defined the parameters of the Lorenz System and the parameters of the Direct Digital Synthesis synthesizer in MATLAB in Section 4.2.1 and Section 4.3.1 respectively. Most of the parameters were the same except for some syntax differences between MATLAB and C. So, I would not repeat them again. They could be reviewed in Section 4.2.1 and 4.3.1. However, there were some major differences I would like to discuss more. In the second line of the code in Section 4.3.1, I created a blank 256-size array used to store the simulation results which were filled by the indexed amplitudes from the sine table. In this Section 4.4.3, I created a blank 100,000-size array for the same usage, but it owned a larger size which allowed me to observe the simulation results more clearly. Line 21 and 22 of the C code in this section were used to write all the simulation results into a text file. This would be introduced later in Section 4.4.5. There was a new parameter "*m*" in this section. The parameter "*m*" did not exist in previous sections. The parameter "*m*" here was used to adjust the relative implementation speed between the numeric integrator and the Direct Digital Synthesis synthesizer. This new parameter "*m*" would be discussed later in Section 4.4.4 in more detail. Also, the output frequencies "*f_out*" was not defined in this section. The reason would also be discussed in Section 4.4.4.

4.4.4 Combination of Lorenz System and Direct Digital Synthesis

Previously, I implemented the Euler-integrator and the Direct Digital Synthesis synthesizer in two separate MATLAB files which could be reviewed in Section 4.2 and 4.3. But I would combine them and rewrite the combination in C in this section. Figure 16 shows the C codes for this section.

```
31          for(int i = 0; i < 100000; ++i){
32              if(i % m == 0){
33                  double xt = x + timestep * sigma * (y - x);
34                  double yt = y + timestep * (x * (rho - z) - y);
35                  double zt = z + timestep * (x * y - beta * z);
36                  x = xt;
37                  y = yt;
38                  z = zt;
39              }
40
41              double F_out = 450+(5*y);
42              unsigned int increment = (unsigned int)((F_out/fs) *constant);
43              accumulator = accumulator + increment;
44              index = accumulator>>24;
45              sin1[i] = sintable[(unsigned int)index];
46              fprintf(file, "%u %lf\n", (unsigned int)index, sin1[i]);
```

Figure 16: combination of the Euler-integrator and the Direct Digital Synthesis in C

In Section 4.2 and 4.3, I wrote one for loop for the Euler-integration and one for loop for the Direct Digital Synthesis synthesizer to implement them. However, to combine the Euler-integrator and the Direct Digital Synthesis synthesizer, I would write them into a single for loop in this C program. The contents of this C program were similar with the previous MATLAB programs except for some syntax differences. So, I would be focusing on the major differences. The first difference was the for-loop iterations. Previously, the for loop for the Euler-integrator would iterate 1,000,000 times, and the for loop for the Direct Digital Synthesis synthesizer would iterate 256 times. But I wrote them in this single for loop shown in Figure 16 and let the for loop iterate 100,000 times in this C program. 100,000 times were enough for me to observe the chaotic patterns from the visualized simulation results.

Previously, the output frequency "*f_out*" defined in the Direct Digital Synthesis synthesizer implementation in Section 4.3.1 could be any arbitrary positive frequencies. However, to build a chaotic synthesizer, the Euler-integrator would set the target output frequency for the the Direct Digital Synthesis synthesizer. It means that the output frequency "*f_out*" should depend on the state variables *x, y, z* of the Lorenz System. The general idea of the chaotic sound synthesizer was that the frequencies of chaotic sounds would change with the behaviors of the Lorenz System. I would define the output frequency "*f_out*" after defining state variables *x, y, z* in the for loop. So, that was the reason why I did not define "*f_out*" in Section 4.4.3. Even though I knew that "*f_out*" depended on state variables *x, y, z,* I could not directly let "*f_out*" be equal to *x, y, z*. I had to perform frequencies modulations to the state variable *x, y, z*. The reason was that the range of values of *x, y, z* was from -25 to 40. The negative values in this range were not eligible to be non-negative output frequencies. ($x, y \in (-25,25), z \in (0,40).$) Even though the positive *x, y, z* values could be as large as 40, they were still too low to be heard by human ears. Also, the change of *x, y, z* values from one state to the next state was too small, it means that the chaotic patterns were not apparent if I directly defined "*f_out*" as *x, y, or z.* So, it was necessary to perform frequency modulations to *x, y, z* to allow human ears to hear the frequency with obvious chaotic patterns. For example, I would times "*x*" with 5 to increase the change of "*x*" value from one state to the next state, in order words, "amplifying" the chaotic patterns. Now, the range of "*x*" value would be from -125 to 125. The frequencies of sounds which could be heard by human ears were usually from 100Hz to 1000Hz. So, I need to add an offset (base frequency) to "$(5 \cdot x)$" to avoid the negative frequencies and get into the range of 100Hz – 1000Hz. In this C program, I added 440Hz to "$(5 \cdot x)$", so the final output frequency would be around 300Hz to 400Hz which was appropriate as output frequencies. I just gave an example here. I could also do the same thing to *y* and *z*. The numbers I used could be modified

as long as the frequency modulation could give appropriate output frequencies which could be heard by human ears and had apparent chaotic patterns. Also, it means that different frequency modulations could give different sound synthesis effects. The different sound synthesis effects would be heard in later experiments as I changed the ways of frequency modulations.

It was mentioned that there was also a new defined parameter "*m*" used to adjust the relative implementation speed between the Euler-integrator and the Direct Digital Synthesis synthesizer. In this C program, I implemented both the Euler-integrator and the Direct Digital Synthesis synthesizer in the same for loop, and the output frequency of the Direct Digital Synthesis synthesizer depended on the state variables *x, y, z* of Lorenz System. It means that the Euler integrator and the Direct Digital Synthesis synthesizer would be implemented with the same speed as the for loop was iterating, and the output frequencies of the Direct Digital Synthesis synthesizer would change as fast as the state variable *x, y, z.* The problem was that state variables *x, y,* z (Lorenz System) was changing too fast to allow the Direct Digital Synthesis synthesizer to generate accurate output frequencies as expected. So, the chaotic patterns would not be heard if they were implemented with the same rate. To solve the problem, I implemented the Euler-integrator relatively more slowly than the Direct Digital Synthesis synthesizer by defining a new parameter "*m*" and implementing the Euler-integrator in an if statement within the for loop. Figure 15 shows the code which can achieve this. The if statement is "*if(i % m == 0){}*", and the Euler-integrator was implemented in this if statement. The if state statement allowed the Euler-integrator to be implemented *m* times more slowly than the Direct Digital Synthesis synthesizer. As the Euler-integrator was implemented once to generate a set of *x, y, z* values, the values of *x, y, z* would hold for the next *m* iterations of the for loop. The values of *x, y, z* could not be updated until the *(m + 1)th* iteration. However, the Direct Digital Synthesis synthesizer would still be implemented and generate output frequency based on the held *x, y, z* values before the *(m + 1)th* iteration. It would let the Euler-integrator be implemented relatively *m* times more slowly than the Direct Digital Synthesis synthesizer to allow the Direct Digital Synthesis synthesizer to generate accurate output frequencies with apparent chaotic patterns. Therefore, the parameter "*m*" could change the relative implementation speed between the Euler-integrator and the Direct Digital Synthesis synthesizer, in other words, the parameter "*m*" could change the oscillating speed of the chaos in the sound synthesis. As "*m*" was higher, the Euler-integrator would be implemented more slowly than the Direct Digital Synthesis synthesizer, and the chaos in the sound synthesis would oscillate more slowly. So, the parameter "*m*" could also be tuned to generate different sound synthesis effects.

There were several important syntax differences I would like to discuss. Previously, to, I used the "*mod*" function in MATLAB to update the overflowing accumulator. But in C, I could directly write "*accumulator = accumulator + increment*" which would automatically update the overflowing accumulator. Also, I added 1 to the sine table index in MATLAB to avoid non-positive index since indexes in MATLAB started at 1. However, the indexes in C started at 0, so I could direct use *"(unsigned int)index*" without adding 1 to it.

4.4.5 Writing the result to files

I had discussed the core contents of the C program in previous sections. So, the next step would be generating all the implementation results. It was not intuitive to discuss the implementation results without plotting and hearing it. So, in this C program, all the implementation results would not be printed on the console since we could not plot it or hear it from the console. Instead, all the implementation results would be written to a text file since it was more convenient to plot and hear the implementation results in the text file using MATLAB.

Figure 17 shows the C codes for this section.

```
21      FILE* file = fopen("/Users/zifuqin/Desktop/final_report_test_1/output.txt", "w");
22      if(file){
46              fprintf(file, "%u %lf\n", (unsigned int)index, sin1[i]);
47          }
48          fclose(file);
49      }
```

Figure 17: writing all the simulation results in a text file.

As shown in Figure 16, I could write all the implementation results into a text file by the function "*FILE* file = fopen("directory/name.txt", "w"); if(file){ fprintf(file, simulation result); fclose(file)}*". Then, all the implementation results would be stored in the *name.txt* within the directory I defined. In the next section, I would discuss the implementation results.

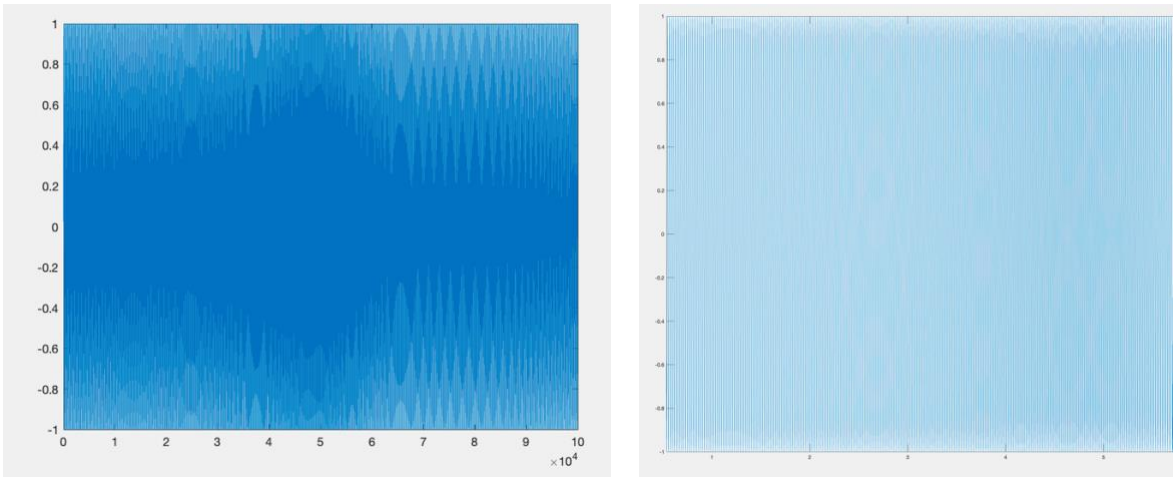
4.4.6 Results and discussion

Now, I had all the implementation results stored in "*output.txt*", which was shown in Figure 18.

```
1 0.024541
2 0.049068
3 0.073565
5 0.122411
6 0.146730
7 0.170962
9 0.219101
10 0.242980
11 0.266713
13 0.313682
14 0.336890
```

Figure 18: all the simulation results stored in the text file.

As shown in the line 46 of codes in Figure 17, the simulation results should contain two columns, one for the index number, and another one for the corresponding indexed amplitudes. However, as shown in the first column in Figure 18, the index was not listed continuously. For example, the index "*3*" was directly jumping to "*5*". This was because the index number for the next state might be rounded to a larger number as being calculated by equation (). So, it was reasonable to have non-continuous index numbers shown in Figure 19. Then, I would plot and hear the implementation results in the text file by MATLAB. Figure 19 shows the implementation results plot, (a) was the regular version, and (b) was the zoom-in version.



(a) *(b)*

Figure 19. the MATLAB plot of the C program implementation *results: (a) the regular version; (b) the zoom-in version.*

It could be seen that the plot was a pure sine wave, but the frequency kept changing with increasing index numbers as expected. Also, the overall frequency of the sine wave plot was around the range of 300Hz to 400Hz, which followed my previous calculation in Section 4.4.4. Using the MATLAB function "*soundsc(A(:,2),44000);*", where "(*A(:,2)*)" was the second column of the text file containing all the indexed amplitude, I heard the output frequencies with apparent chaotic patterns. The first thing it reminds me was mosquito. The sound kept oscillating with strong structural patterns. This was the first time I sonified the Lorenz System by the non-realtime software simulation, and the sound was quite reasonable with interesting chaotic patterns. Overall, the software simulation was successful to get reasonable implementation results. And the Lorenz System was successfully sonified by the non-realtime software simulation.

5. Hardware Prototyping

With all the software simulation results, I had a basic idea how the chaotic sound synthesizer would sound like, then the realtime microcontroller-based prototype would start. The objective of this phase was to reproduce the sounds in the software simulations through a RP2040 microcontroller and a DAC. With the advice from Prof. Hunter Adams and Prof. Bruce Land, the RP2040 microcontroller and the MCP4822 digital to analog converter (DAC) –were used for the realtime microcontroller-based prototype.

5.1 Raspberry Pi Pico C/C++ Building Environment Setup

The first step would be setting up the building environment for Raspberry Pi Pico. Users could choose either C/C++ building environment or Python build environment for Raspberry Pi Pico. With the advice from Prof. Hunter Adams and Prof. Bruce Land, the C/C++ build environment was chosen since C/C++ could give faster realtime hardware implementation than Python.

Initially, I was trying to set up the C/C++ Building Environment for Raspberry Pi Pico on MAC OS. Following Raspberry Pi Pico datasheet, I implemented the setup on the MAC OS with the latest M1 Chip, however, an unsolvable problem occurred due to the M1 Chip. Alternatively, to solve the problem, a Window Virtual Machine from "Parallels" was installed on my M1 MAC computer so that the Raspberry Pi Pico C/C++ Building Environment could be set up on

Windows OS. With the detailed guide from Prof. Hunter Adams' website, the C/C++ Building Environment for Raspberry Pi Pico was successfully set up. The detailed setup steps could be found on Prof. Hunter Adams' website. Now, I could let Raspberry Pi Pico work by writing C programs and implementing the C programs on RP2040.

5.2 Procedures of building a project on RP2040: blinking an LED

To get used to developing on Raspberry Pi Pico in C/C++, several test projects were developed on the Raspberry Pi Pico building environment. The first test project was blinking an LED. It was a simple project, but it allowed me to understand the flow of building a complete project on Raspberry Pi Pico.

5.2.1 Example C program of blinking an LED

The first step was writing a C program used to blink the LED pin on Raspberry Pi Pico. An example C program of blinking the LED was given on Prof. Hunter Adams' website. I first tried to understand every single line of the given example C program, then I was trying to rewrite it by myself. Figure 20 shows the entire example C program of blinking the LED. The

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"

const uint LED_PIN = 25 ;

int main() {

    stdio_init_all() ;

    gpio_init(LED_PIN) ;
    gpio_set_dir(LED_PIN, GPIO_OUT) ;

    while(1) {

        gpio_put(LED_PIN, 0) ;
        sleep_ms(250) ;
        gpio_put(LED_PIN, 1);
        puts("Hello world\n") ;
        sleep_ms(1000) ;
    }
}
```

Figure 20: The entire example C program of blinking the LED pin on Raspberry Pi Pico

first three lines listed all the required libraries to develop this project on Raspberry Pi Pico. The fourth line was used to define the blinking LED pin which was the GPIO 25 on Raspberry Pi Pico. The next several lines were the main which was implemented to blink the LED pin on Raspberry Pi Pico. The first line in the main was used to initialize all the stdio types supporting the project. (A detailed official explanation could be found in the Raspberry Pi Pico SDK Documentation). The next two lines were used to initialize the LED pin defined above. The following while loop made the LED pin high first, delayed 250ms, made the LED pin low, printed "Hello World", delayed 1000ms, and repeated. So, the example C program could let the LED pin on Raspberry keep blinking. Then, the C program above could be renamed as test.c for the future development.

5.2.2 Required files

After writing the C program for the project of blinking an LED, the next step was to create the "CMakeLists.txt". Figure 21 shows the example "CMakeLists.txt".

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project)

pico_sdk_init()

add_executable(test test.c)

pico_enable_stdio_usb(test 1)
pico_enable_stdio_uart(test 0)

pico_add_extra_outputs(test)

target_link_libraries(test pico_stdlib)
```

Figure 21: The "CMakeLists.txt" used in the project of blinking the LED on Raspberry Pi Pico

Here is an important note. All the names in the *"CMakeLists.txt"* should follow the names of the project file and C file. Also, the Pico libraries used in the C program should be listed on the last line of *"CMakeLists.txt"*. Next, the "pico_sdk_import.cmake" file should be ready. The "pico_sdk_import.cmake" file could be found in the external folder from the pico-sdk installation.

5.2.3 Procedures

As these three files ("test.c", *"CMakeLists.txt", and "*pico_sdk_import.cmake") were ready, I could navigate to the directory that I had installed "pico-sdk" on "Developer Command Prompt for VS 2019". The current directory should be "C:\Users\zifuqin\Pico", then make a directory named "test" by typing "mkdir test". Now, those three files mentioned above should be dragged into this "test" directory. Get back to the "Developer Command Prompt for VS 2019", in the "test" directory, a "build" directory should be made by typing "mkdir build". Then, by typing following lines on "Developer Command Prompt for VS 2019":

- C:\Users\zifuqin\Pico\test> cd build
- C:\Users\zifuqin\Pico\test\build> cmake -G "NMake Makefiles" ..
- C:\Users\zifuqin\Pico\test\build> nmake

The project should be successfully built on Raspberry Pi Pico if no errors were shown on the display. A test.uf2 file should be found in the "build" directory. Next, I pressed & held the "BOOTSEL" button on Raspberry Pi Pico, connected Raspberry Pi Pico to the computer by a microUSB-to-USB-C cable, and released the "BOOTSEL" button. Now, the Raspberry Pi Pico drive should appear on the computer. Then, as the test.uf2 was dragged into the Raspberry Pi Pico drive, the Raspberry Pi Pico drive would disappear, and the LED pin on Raspberry Pi Pico should start blinking. The first project of blinking an LED on Raspberry Pi Pico was done.

5.2.4 Results and discussion

In general, the system just complied the C file that humans could read to the uf2 file that machines could read by compliers. So, alternatively, to blink an LED on Raspberry Pi Pico, a uf2 file downloaded from Internet could be directly dragged into the Raspberry Pi Pico drive since it means that Raspberry Pi Pico just received a uf2 file that it could directly read. To get a uf2 file,

a C file was needed to be complied by the complier.

5.3 Building the Direct Digital Synthesis project and COM PORT serial connection

After successfully developing a LED blinking project on RP2040, I understood the basic procedures of build a project on RP2040. The next step would be building another project on RP2040 to implement the previous C program in Section 4.4. The previous C program was the combination of the Euler-integrator and the Direct Digital Synthesis synthesizer. To make the simulation results less complicated, I would delete the Euler-integration implementation in this project and only implement the Direct Digital Synthesis synthesizer. So, as I assigned an output frequency, the implementation result should give me the same output frequency as I previously assigned. Since I had not built up the system circuit, the way I showed the implementation result was by a COM PORT serial connection with PuTTY. Instead of writing the implementation result to a text file in Section 4.4.5, all the implementation results should be printed on PuTTY by the COM PORT serial connection this time. If the implementation result shown on PuTTY gave me the same output frequency as I previously assigned, it means that RP2040 implemented the Direct Digital Synthesis synthesizer correctly.

5.3.1 Modifying the C program in Section 4.4

In this section, I would delete the Euler-integrator implementation from the original C program in Section 4.4 to get less complicated implementation results. So, the new C program would only contain the Direct Digital Synthesis synthesizer, and the output frequency was assigned by myself. ("*f_out = 100;*") The new C program was also modified so that it could print the implementation results on PuTTY instead of writing the results into a text file. Figure 22 shows the code for this section.

```c
#include <stdio.h>
#include <math.h>
#include "pico/stdlib.h"
#define PI 3.1415926535897

int main(int argc, const char * argv[]) {

    stdio_init_all();
    double sintable[256] = {0};
    for(int i = 0; i <256; i++){
        sintable[i] = sin(2*PI*i/256);
    }

    double fs = 44000;
    double F_out = 100;
    int constant = (int)pow(2, 32);
    unsigned int index;
    unsigned int accumulator = 0;
    double sin1[10000] = {0};
    for(int i = 0; i < 10000; ++i){

        unsigned int increment = (unsigned int)((F_out/fs) *constant);
        accumulator = accumulator + increment;
        index = accumulator>>24;
        sin1[i] = sintable[(unsigned int)index];
        printf("Result: %d %lf\n", (unsigned int)index, sin1[i]);
        sleep_ms(1000) ;

        }
    return 0;
}
```

Figure 22: the new modified DIRECT DIGITAL SYNTHESIS simulation C program.

Since I deleted the Euler-integrator implementation, "*f_out*" could be any appropriate positive frequencies. "*f_out*" was 100 in this case. Line 3 ("*#include "pico/stdlib.h""*") and line 8 ("*stdio_init_all()"*) in Figure 21 were necessary "*pico*" configurations for developing projects on RP2040 as introduced in Section 5.3. In line 26, instead of using the previous "*fprintf*" function to write the simulation result into a text file, I just simply used "*printf("name options", (unsigned int)index, sin1[i])*" to directly print out the implementation result. I also added "*sleep_ms(1000);*" to let the program to print one line per second.

5.3.2 Procedures

In general, I just built another project on RP2040. So, most of steps were similar with what I introduced previously in Section 5.2. The slight difference was that there was an extra step of making a COM PORT serial connection, the printed implementation result would be shown on PuTTY. The specific procedures are listed below:

- Drag the generated uf2 file to the Raspberry Pi Pico drive, then the Raspberry Pi Pico drive will disappear.
- Open the "Device Manager" and find the specific connect COM PORT number in "*Ports(COM & LPT)*", it should be COM PORT3.
- Open PuTTY, click on serial, type in the specific COM PORT#, changed the speed to 11520, then run it.

Now, they should make a successful COM PORT serial connection, and the implementation result should be printed on PuTTY. Figure 23 shows the simulation result printed on PuTTY.

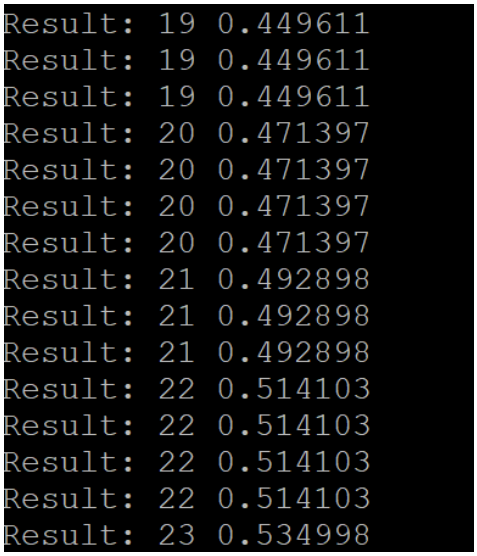


Figure 23: the output result printed on PuTTY.

It was seen that the output result printed on PuTTY was the same as the simulation result in Section 4.4.6. It means that RP2040 implemented the Direct Digital Synthesis synthesizer correctly.

5.4 Building the system circuit.

Now, RP2040 could implement the Direct Digital Synthesis synthesizer correctly and give me digital frequency signals by serial connection. But the objective of this project was to hear the physical sounds, so a digital to analog converter (DAC) was needed to connect to the RP2040 to convert the generated digital frequency signals to analog frequency signals. The MCP4822 was chosen to be the DAC for this project. After the RP2040 microcontroller and the DAC MCP4822 were ready, the next step would be connecting these two components together on a breadboard to let them communicate. Before connecting the RP2040 to the DAC, the Raspberry Pi Pico should be soldered. When doing the soldering, you should be careful to align all the pins on one side and avoid shorting. To connect the RP2040 to the DAC, several datasheets including "MCP4822 Datasheet", "Raspberry Pi Pico Datasheet", and "RP2040 Datasheet" were quite helpful as guides. As building up the circuit, "MCP4822 Datasheet" provided the detailed pinout of MCP4822 and the functions of each pin; "Raspberry Pi Pico Datasheet" provided the detailed pinout of Raspberry Pi Pico; "RP2040 Datasheet" provided the detailed function explanations of each pin on the microcontroller.

5.4.1 MCP4822 DAC and Raspberry Pi Pico pinout

This section would discuss the pins of both Raspberry Pi Pico and MCP4822 DAC in detail. Not all the pins of Raspberry Pi Pico would be introduced in detail since there were over 40 pins on Raspberry Pi Pico. So, only pins used in this project would be discussed in great detail. Details of all the other pins could be found in the datasheet mentioned above. MCP4822 is a dual 12-bit voltage output DAC with 8 available pins and SPI compatible Serial Peripheral Interface, operating with a single power supply from 2.7V to 5.5V [3]. Figure 24 shows the pinout of MCP4822 DAC. Table 1 shows the detailed information of each pin.

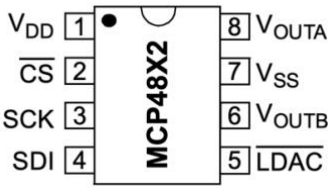


Figure 24: The pinout of MCP4822 DAC.

Table 1: Detailed information of each pin on MCP4822.

Pin Number	Pin Name	Pin Function
1	V_{DD} (Positive Supply Voltage)	V_{DD} pin is the positive supply voltage pin. The supply voltage input should be from 2.7V to 5.5V relative to the V_{SS} ground.
2	$\overline{CS}$ (Chip Select)	$\overline{CS}$ is the chip select pin used to enable/close the serial clock input and serial data input.
3	SCK (Serial Clock Input)	SCK is the SPI compatible serial clock input.
4	SDI (Serial Data Input)	SDI is the SPI compatible serial data input.
5	$\overline{LDAC}$ (Latch DAC Input)	$\overline{LDAC}$ is the synchronization pin which allows both analog output A and analog output B to update simultaneously. This pin would not be used in this project, always connecting to the ground.
6	V_{OUTB} (Analog Output B)	V_{OUTB} is the DAC output B pin which outputs the transferred signals in channel B.
7	V_{SS} (Analog Ground Pin)	V_{SS} is the relative ground pin.
8	V_{OUTA} (Analog Output A)	V_{OUTB} is the DAC output A pin which outputs the transferred signals in channel A.

Raspberry Pi Pico is a microcontroller board built using the microcontroller chip RP2040. It features two ARM Cortex-M0+ cores run up to 133MHz; 264kB on-chip SRAM supporting for up to 16MB of off-chip Flash memory with over 40 pins available. There are 30 GPIO pins with 2 UARTs, 2 SPI controllers, 2 I2C controller, 16 PMW channels, USB 1.1 controller and PHY with host and device support, and 8 PIO state machines for peripherals [2]. Figure 25 shows the pinout of Raspberry Pi Pico. According to the pinout of Raspberry Pi Pico, there are multiple functions on the same GPIO pin. For example, for GPIO 0, it has several functions including UART0 TX, I2C0 SDA, and SPI0 RX. Therefore, the specific function should be selected when developing the software. In this project, the SPI controller was mainly used. There are two SPI controllers in total, which are SPI 0 and SPI 1 respectively. It was found that SPI 0 controller is assigned from GPIO 0 to GPIO 7, and SPI1 controller is assigned from GPIO 8 to GPIO 15. Also, the default SPI 0 is assigned from GPIO 16 to GPIO 19. If you only need one SPI controller, only one SPI controller could be chosen (either SPI0 or SPI 1). It means that SPI0 and SPI1 could not

be mixedly used as you would like to use only one SPI controller. In this project, only one SPI controller was needed, so I chose SPI 0 (GPIO 0 to GPIO 7) to work on.

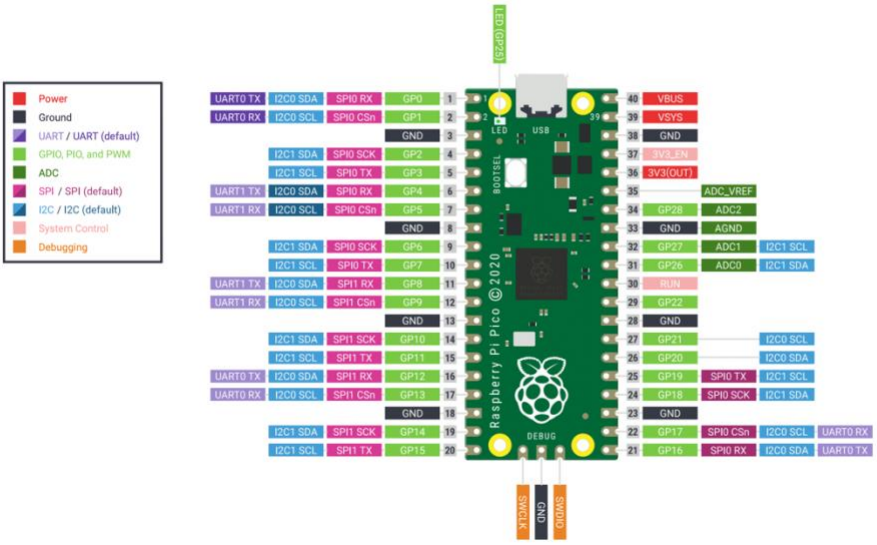


Figure 25: The pinout of Raspberry Pi Pico.

As shown on the pinout, SPI 0 controller has different functions for each pin of SPI 0. These functions would be briefly discussed in Table 2.

Table 2: Information about SPI0 controller.

Physical Pin Number	GPIO Pin Number	Pin Name (SPI Controller)	Pin Function (SPI Controller)
1,6	0,4	SPI0 RX/MISO (**R**eceived **M**aster **I**nput, **S**lave **O**ut)	SPI0 RX/MISO was used to receive data from the subnode.
2,7	1,5	SPI0 CSn (Chip Select)	SPI0 CSn is the chip select pin used to enable/close the serial clock input and serial data input.
4,9	2,6	SPI0 SCK (Serial Clock)	SCK is the Serial Clock pin, which would be attached to the SCK pin of MCP4822.
5,10	3,7	SPI0 TX/MOSI (**T**ransmitted, **M**aster **O**utput, **S**lave **I**nput)	SPI0 TX/MOSI was used to transmit data to the DAC.

5.4.2 Connecting MCP4822 to Raspberry Pi Pico

With all these information, I could start building the system circuit by connecting the Raspberry Pi Pico to the MCP4822 DAC. Pin 1 (V_{DD}) on DAC should be connecting to the pin 36 on Raspberry Pi Pico to get the power supply; Pin 2 ($\overline{CS}$) on DAC should be connecting to the pin 7 (GPIO 5) on Raspberry Pi Pico to get the SPI0 chip select control; Pin 3 (SCK) on DAC should

be connecting to the pin 9 (GPIO 6) to get the SPI0 serial clock control; Pin 4 (SDI) on DAC should be connecting to the pin 10 (GPIO 7) to transmit the data from RP2040 to the DAC. Pin 5 ($\overline{LDAC}$) on DAC would not be used in this project, so it would always connect to the ground. Pin 7 (V_{SS}) on DAC would always be connecting to the ground. Both pin 6 (V_{OUTB}) and pin 8 (V_{OUTA}) were left unconnected for now. Till now, the circuit was successfully built, I could start work on the software development.

5.5 Software Development for Direct Digital Synthesis

Previously, I had done the combination (The Euler-integrator and the Direct Digital Synthesis synthesizer) implementation in C in Section 4.4. The program could create a text file printing all implementation results. Next, I modified the original C program by deleting the the Euler-integrator implementation and adding some "*pico*" configurations to create a new C program for the COM PORT serial connection. After correctly implementing the new C program on RP2040, it printed all the implementation results on PuTTY by a COM PORT serial connection. The simulation results shown on PuTTY gave me exactly the same output frequency as I previously assigned. Now, after building up the circuit, I would create another project on RP2040 As I implement the algorithm on RP2040, "*f_out*" should be detected from the V_{OUT} pin of the DAC in the form of analog frequency signals. The V_{OUT} pin of the DAC should give me the same output frequency as I assigned previously assigned in the Direct Digital Synthesis synthesizer implementation, so it would be obvious to see if the RP2040 microcontroller correctly communicated with the DAC. Therefore, for the software development, I would build a new project on RP2040 to implementing only the Direct Digital Synthesis synthesizer as a start. Most steps of building a project on RP2040 were the same as the previous sections. The major differences and difficulties were to do the software development which let RP2040 communicate with DAC to generate the analog output frequency I previously assigned. The RP2040 SDK documentation and Prof. Van Hunter Adams' example codes were the guide for me to complete this task.

The general idea of this task was to build a timer interrupt and an interrupt service routine in the program. The timer interrupt would be set to overflow at 44kHz which was the sampling frequency, so the algorithm would enter the interrupt service routine 44,000 times per second. Each time it entered the interrupt service routine, the algorithm would do following things:

- Pull the chip select low to enable the data transaction.
- Update the Direct Digital Accumulator for channel A.
- Use the updated accumulator to index the sine table for channel A.
- Perform the SPI data transaction for channel A.
- Pull the chip select high to stop the data transaction.
- Leave the interrupt service routine.

So, the algorithm would repeat this 44,000 times per second and keep transmitting data, then V_{OUTA} pin on DAC would keep outputting analog frequency signals.

5.5.1 Header Files

The start of the algorithm in a C file should be the header files. In this project, the standard C header files (stdio.h and math.h were included), and the "pico/stdlib.h" and hardware/spi.h" from the "C SDK for the Raspberry Pi Pico" were also included to provide the

required library resources for this project. Figure 26 shows the header files (includes) for this algorithm.

```
2   #include <stdio.h>
3   #include <string.h>
4   #include <math.h>
5   #include "pico/stdlib.h"
6   #include "pico/binary_info.h"
7   #include "hardware/spi.h"
```

Figure 26: the required header files for this C program.

5.5.2 Define DIRECT DIGITAL SYNTHESIS Parameters

In this section, all the necessary parameters for the Direct Digital Synthesis synthesizer implementation were defined. Figure 27 shows the code for this section.

```
12  //Defining Direct Digital Synthesis parameters
13  typedef signed int fix15;
14  #define fix2int15(a) ((int)(a >> 15))
15  #define PI 3.1415926535897
16  #define TWO32 4294967296.0
17  #define FS 44000
18  #define sine_table_size 256
19  fix15 sin_table[sine_table_size];
20  double Fout_1 = 100.0;
21  volatile unsigned int accumulator_1;
22  volatile unsigned int increment_1 = (Fout_1*TWO32)/FS;
```

Figure 27: defining necessary DIRECT DIGITAL SYNTHESIS parameter for the software development.

Most of the parameter were similar with the parameters of the Direct Digital Synthesis synthesizer defined in previous sections. I would mainly discuss the differences. In this section, most of the parameters were defined by "*#define*" for convenience. Also, I used the fixed point athematic to define "*typedef signed int fix15*". Fixed-point could help me speed up the interrupt service routine as modulating the amplitude of sine waves. So, as defining the sine table, I used "*fix15 sin_table[sine_table_size]*" to perform the amplitude modulation of sine waves to achieve faster interrupt service routine. This idea was from Prof. Van Hunter Adams' website. Also, another way to speed up the interrupt service routine was performing all the necessary numerical calculations outside the interrupt service routine. For example, line 22 of Figure 27 could be written in the interrupt service routine, but this numerical calculation would slow down the interrupt service routine. So, I wrote this line here outside the interrupt service routine to make the interrupt service routine faster. This advice was from Prof. Van Hunter Adams and Prof. Bruce R. Land.

5.5.3 Defining SPI and DAC parameters

After defining the parameters of the Direct Digital Synthesis synthesizer, SPI and DAC parameters should be defined as well for the SPI channel communication between RP2040 and DAC. The SPI and DAC parameters should be consistent with the circuit schematic. Figure 28 shows the code for this section.

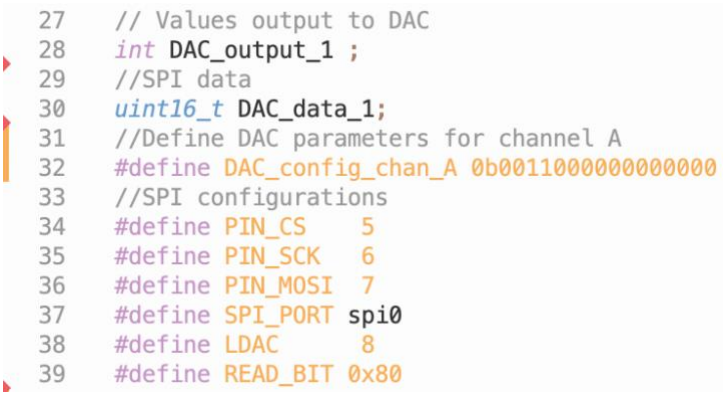


Figure 28: defining all the SPI parameters.

Based on the circuit I built in Section 5.4. I defined all the pins in line 34 – 39 of Figure 27 as their GPIO pin numbers. All the pins and their (*GPIO*) numbers should be consistent with the circuit design. Also, the "*SPI_PORT*" was defined as "*spi0*" since I used the SPI0 channel in this project. "*int DAC_output_1*" was defined as the digital simulation results. "*unit16_t DAC_data_1*" was defined as a 16 bits integer SPI data since the writing command for MCP4822 DAC was 16 bits. The left four bits were configuration bits, and the rest 12 bits were the data bits. The first four configuration bits of MCP4822 DAC were from bit 15 to bit 12. "*Bit 15" was the* "*DAC Channel A or DAC Channel B Selection bit*" (*1 = write to channel A; 0 = write to channel B*). "*Bit 14*" was a don't care bit. "*Bit 13*" was the "*output gain selection bit*" which was always 1 in the project, representing "*1x (V_{OUT} = V_{REF} * D/4096)*". And "*Bit 12*" was the "*Output shutdown control bit*" which was always 1 in this project to keep the output data always active. From "*Bit 11*" to "*Bit 0*", they were all data bits. More details about the MCP4822 DAC bits configuration could be found in the MCP4822 Datasheet. So, "*DAC_config_chan_A*" was defined as "*0b0011000000000000*" based on the bit configurations mentioned above. "*0b0011000000000000*" represented that the data would be written to channel A (*V_{OUTA} Pin*). The output was always active, and the 12 data bits were left blank for masking.

5.5.4 Interrupt Service Routine

After defining all the necessary parameters, I could develop the interrupt service routine. As mentioned in the introduction of Section 5.5. The organization of the interrupt service routine should be:

1. Pull the chip select low to enable the data transaction.
2. Update the Direct Digital Synthesis Accumulator for channel A.
3. Use the updated accumulator to index the sine table for channel A.
4. Masking with DAC control bits.
5. Perform the SPI data transaction for channel A.
6. Pull the chip select high to stop the data transaction.
7. Leave the interrupt service routine.

The software design should also follow this organization. Figure 29 shows the code for this section.

```
43  // Interrupt Service Routine
44  bool repeating_timer_callback(struct repeating_timer *t) {
45
46      // Pull Chip Select Low
47      gpio_put(PIN_CS, 0);
48      //Update DDS Accumulator and Index the sinetable
49      accumulator_1 += increment_1;
50      DAC_output_1 = sin_table[accumulator_1>>24];
51
52      // Do the SPI transmission
53      // Mask with DAC control bits
54      DAC_data_1 = (DAC_config_chan_A | (DAC_output_1 & 0xfff))  ;
55      // SPI write (no spinlock b/c of SPI buffer)
56      spi_write16_blocking(SPI_PORT, &DAC_data_1, 1);
57      gpio_put(PIN_CS, 1);
58
59  }
```

Figure 29: interrupt service routine.

In general, the interrupt service routine was a repeating timer callback. It should be called 44,000 times per second based on the sampling frequency 44,000Hz in this project. So, the interrupt service routine should start with the function of "*bool repeating_timer_callback(struct repeating_timer *t)*" as the callback of the repeating timer, and the data structure "*struct repeating_timer*" would be defined in main. The first step in the interrupt service routine was to enable the data transaction by pulling chip select low using the function "*gpio_put(PIN_CS, 0)*", where "*PIN_CS*" was the pin I previously defined. "*0*" was the bool value representing "*low*" position, and "*gpio_put(unit gpio, bool value)*" was used to drive a single GPIO pin high/low. I would also use the same function at the end of the interrupt service routine to disable the data transaction by pulling the chip select pin high, but I would use the bool value "*1*" instead of "*0*" to drive the chip select pin high and stop the data transaction.

Next two steps would be updating the Direct Digital Synthesis accumulator for channel A and using the updated accumulator to index the sine table values as the values output to DAC ("*DAC_output_1*"). These were something I had done multiple times in previous section. These contents could be reviewed in Section 4.3.2 and 4.4.4. Now, "*DAC_output_1*" had been converted to a 12-bit number, the conversion was made in main which would be discussed in Section 5.5.5. Therefore, I need to mask the 12-bit data "*DAC_output_1*" with the configuration bits I defined in Section 5.5.3. the configuration bits I defined in Section 5.5.3. was "*DAC_config_chan_A = 0b0011000000000000*". By using the function "*DAC_data_1 = (DAC_config_chan_A | (DAC_output_1 & 0xfff));*", I masked the 12-bit data with the 4-bit configuration and assign them as "*DAC_data_1*". Now, "*DAC_data_1*" should be a 16-bit data containing the configuration and data. Therefore, I would transmit the 16-bit data "*DAC_data_1*" to the DAC. By using the function "*spi_write16_blocking(spi_inst_t*spi, const unit16_t *src, size_t len)*", I could write the 16-bit data to an SPI device which was the MCP4822 DAC in this project. In this case, to transmitting the 16-bit data to the DAC via an SPI channel, I used the command of "*spi_write16_blocking(SPI_PORT, &DAC_data_1, 1);*". Now, the data transaction should be done, then I would pull the chip select pin high to stop the data transaction and leave the interrupt service routine. After completing all the steps mentioned in this section, the interrupt service routine was implemented once. And the entire process would be repeated 44,000 times per second to keep transmitting data from RP2040 to the MCP4822 DAC. All the functions

discussed in this section could be found in both Raspberry Pi Pico SDK Documentation v1.3.0 and Prof. Van Hunter Adams' website.

5.5.5 Main

All the program started being implemented from the main. The main of the C program was consisted of 3 parts, which were "*Initialization*", "*Sine table set up*", and "*Adding a repeating timer*". Figure 30 shows the code for this section.

```
61  //Main
62  int main(int argc, const char * argv[]){
63
64      stdio_init_all();
65      spi_init(SPI_PORT, 2000000);
66      spi_set_format(SPI_PORT,16,0,0,0);
67      gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
68      gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
69      gpio_set_function(PIN_CS, GPIO_FUNC_SPI);
70      gpio_init(PIN_CS);
71      gpio_set_dir(PIN_CS, GPIO_OUT);
72      gpio_init(LDAC);
73      gpio_set_dir(LDAC, GPIO_OUT);
74      gpio_put(LDAC, 0);
75
76      //Set up the sinetable
77      for(int i = 0; i < 256; i++){
78          sin_table[i] = (int) (2047*sin(2*PI*(float)i/(float)sine_table_size) + 2047);
79      }
80
81      struct repeating_timer timer_0;
82      add_repeating_timer_us(-23, repeating_timer_callback, NULL, &timer_0);
83  }
```

Figure 30: the main of the whole C program in Section 5.5.

The first line of the "*Initialization*" part was "*stdio_init_all*" which was also mentioned in Section 5.2.1. "*stdio_init_all*" was used to initialize all the stdio types supporting the project. Next, the SPI0 channel used in this project should be defined. Using the function "*spi_init(spi_inst_t *spi, uint baudrate)*", the specified SPI0 or SPI1 could be defined. In this project, only SPI0 would be used. The second element "*uint baudrate*" in the function "*spi_init*" represents the Baudrate requested in Hz. 2,000,000Hz would be enough for this project. Then, the initialized SPI0 could be configured by the function "*spi_set_form*". In the function "*spi_set_form*", there were 5 elements in total, including SPI channel specifier (SPI0 or SPI1), the number of data bits per transfer, SSPCLKOUT polarity, SSPCLKOUT phase, and the order (must be SPI_MSB_FIRST). In this project, the SPI channel would be configured as "*spi_set_format(SPI_PORT,16,0,0,0);*". After initializing and configuring the SPI channel, I would start working on the GPIO initialization and configuration. First, I would initialize all the GPIO pins that I had defined and assign the SPI function to these pins using the functions "*gpio_set_function*" and "*gpio_init*". The detailed syntax I used could be found in Figure 30. As mentioned in Section 5.4.1, the $\overline{\text{LDAC}}$ should be always low using the functions "*gpio_set_dir(LDAC, GPIO_OUT);*" and "*gpio_put(LDAC, 0);*" Also, the chip select pin should be set to the direction "*GPIO_OUT*" using the function "*gpio_set_dir(PIN_CS, GPIO_OUT);*" to be prepared for the interrupt service routine. Therefore, all the used pins were initialized and configured.

The next part of the main was the sine table set up. The general idea of this sine up was same as sine table setup in previous sections (Section 4.3.1 and 4.4.3). But the difference was that I should perform amplitude modulation to the sine table here to allow the values in the sine table to be 12 bit long. So, these 12-bit values could be filled into the "DAC_output_1" and masked

with the 4-bit configuration to form the 16-bit data for the DAC. I would perform the amplitude modulation to the original sine table by "2047*sin(2*PI*(float)i/(float)sine_table_size) + 2047" since 2047 was $2^{12} - 1$ which was also the capacity of 12-bit data.

The final part of the main was adding the repeating timer. I had developed the interrupt service routine which should be called. So, in the main, I need to set it up and called the timer as the rate of sampling frequency. First, the repeating_timer was defined as timer_0 by "*struct repeating_timer timer_0;*". Next, I would add this timer and call it repeatedly at the time interval of $\frac{1}{f_s}\ ms$ using the function "*add_repeating_timer_us(int64_t delay_us, repeating_timer_callback_t callback, void * user_data, repeating_timer_t * out);*". To make the interrupt service routine be repeatedly called as the interval of $\frac{1}{f_s = 44{,}000 Hz} = 23\ ms$, I wrote the command as "*add_repeating_timer_us(-23, repeating_timer_callback, NULL, &timer_0);*" which was consistent with the interrupt service routine and the sampling frequency. After setting up the repeating timer, the main of the C program was done, and all the software development for this section was done. All the functions I discussed in this section could be found in both Raspberry Pi Pico SDK Documentation and Prof. Van Hunter Adams' website.

5.6 Software development of Lorenz System

From the results in Section 5.5.6, it was seen that the oscilloscope gave exactly the same 100Hz frequency sine wave as I defined previously. It means that RP2040 successfully implemented the DIRECT DIGITAL SYNTHESIS C program, and the C program was working properly to give me accurate output frequencies. Therefore, the next step would be adding the Lorenz System to the C program in Section 5.5, then the output pins of the DAC should give me oscillating output frequencies with chaotic patterns like what we have seen in Section 4.4.6. After introducing the general idea of Section 5.6, the code for this section would be discussed section by section in great details.

5.6.1 Defining Lorenz System parameters

In Section 4.4.4, I combined the Euler-integrator and the Direct Digital Synthesis synthesizer and rewrote the combination in C. Section 5.6.1 would define the parameters of the Euler-integrator for the Lorenz System. Section 5.6.1 would share some same contents as Section 4.4.4. For more details, please refer to the Section 4.4.4. The major difference was made in the interrupt service routine which would be discussed in Section 5.6.2.

5.6.2 Modification of interrupt service routine

Previously in Section 5.5.4, I had developed an interrupt service routine for the Direct Digital Synthesis synthesizer, and "*f_out*" could be defined by my own choice outside the service routine. However, as mentioned in Section 4.4.4, as adding the Euler-integrator implementation to the Direct Digital Synthesis synthesizer, "*f_out*" would be based on the frequency modulation of state variables *x, y, z.* It means that both "*f_out*" and the Euler-integrator should be developed within the interrupt service routine since "*f_out*" and state variables *x, y, z* should keep being updated with repeatedly calling the interrupt service routine to keep generating oscillating "*f_out*" with chaotic patterns. So, the difference between the interrupt service routine here and the interrupt service routine in Section 5.5.4 was the new added Euler-integrator and modulated "*f_out*". Figure 31 shows the code for this section.

```
61  // Interrupt Service Routine
62  bool repeating_timer_callback(struct repeating_timer *t) {
63      counter += 1;
64      // Pull Chip Select Low
65      gpio_put(PIN_CS, 0);
66      //for(int o = 0; o < 100000; ++o){
67      if(counter % m == 0){
68          double xt = x + tt * a * (y - x);
69          double yt = y + tt * (x * (b - z) - y);
70          double zt = z + tt * (x * y - c * z);
71          x = xt;
72          y = yt;
73          z = zt;
74      }
75      //}
76      //Update DDS Accumulator and Index the sinetable
77
78
79      Fout_1 = 450+(5*y);
80      phase_incr_main_1 = (Fout_1*TWO32)/FS;
81      phase_accum_main_1 += phase_incr_main_1;
82      ii = phase_accum_main_1>>24;
83      DAC_output_1 = sin_table[ii];
84
85      // Mask with DAC control bits
86      DAC_data_1 = (DAC_config_chan_A | (DAC_output_1 & 0xfff))  ;
87
88      // SPI write (no spinlock b/c of SPI buffer)
89      spi_write16_blocking(SPI_PORT, &DAC_data_1, 1);
90
91      gpio_put(PIN_CS, 1);
92
93  }
```

Figure 31: the modified interrupt service routine.

As adding the Euler-integrator implementation to the interrupt service routine, I still expected the Euler-integrator to be implemented relatively *m* times more slowly than the Direct Digital Synthesis synthesizer to allow the Direct Digital Synthesis synthesizer to generate accurate output frequency. To achieve this, I used the same if statement as Section 4.4.4. Then, "*f_out*" could be frequency-modulated in the same way as Section 4.4.4 as well. Therefore, the organization of the interrupt service routine now became to:

1. Pull the chip select low to enable the data transaction.
2. Enter the if statement to update the values of state variable *x, y, z* and hold the values for the next *m* interrupt service routine calls.
3. Calculate the "*f_out*" based on the values of state variable *x, y, z* and hold the values of "*f_out*" for the next *m* interrupt service routine calls.
4. Update the Direct Digital Synthesis Accumulator for channel A.
5. Use the updated accumulator to index the sine table for channel A.
6. Masking with DAC control bits.
7. Perform the SPI data transaction for channel A.
8. Pull the chip select high to stop the data transaction.
9. Leave the interrupt service routine.

In general, the organization of interrupt service routine in this section was same as Section 5.5.4. The only differences were the Euler-integrator and "*f_out*". There were no other changes need to be made for now. As implementing the C program on RP2040 in this section, the V_{OUTA} pin of the DAC would give me oscillating analog output frequencies signals with chaotic patterns, which should be same as the implementation results in Section 4.4.6. Now, the device

worked correctly as a RP2040 based-chaotic sound synthesizer for only one channel (channel A; V_{OUTA} pin). The next step was to expand the device to both channels.

5.6.3 Expending to two channels

One channel of the DAC worked properly as expected, therefore, adding another worked channel would not be quite complicated. I would repeat what I did for the first channel and modify the interrupt service routine. Figure 32 shows the code for this section.

```
53  volatile unsigned int phase_accum_main_1;
54  volatile unsigned int phase_incr_main_1;
55  volatile unsigned int phase_accum_main_2;
56  volatile unsigned int phase_incr_main_2;
57  double Fout_1;
58  double Fout_2;
59
60  // Interrupt Service Routine
61  bool repeating_timer_callback(struct repeating_timer *t) {
62
63      // Pull Chip Select Low
64      gpio_put(PIN_CS, 0);
65
66      if(counter % m == 0){
67          double xt = x + tt * a * (y - x);
68          double yt = y + tt * (x * (b - z) - y);
69          double zt = z + tt * (x * y - c * z);
70          x = xt;
71          y = yt;
72          z = zt;
73      }
74
75          Fout_1 = 450+(5*y);
76          Fout_2 = 450+(5*x);
77          phase_incr_main_1 = (Fout_1*TWO32)/FS;
78          phase_accum_main_1 += phase_incr_main_1;
79          phase_incr_main_2 = (Fout_2*TWO32)/FS;
80          phase_accum_main_2 += phase_incr_main_2;
81
82          ii = phase_accum_main_1>>24;
83          iii = phase_accum_main_2>>24;
84          DAC_output_1 = sin_table[ii] + 2047;
85          DAC_output_2 = sin_table[iii] + 2047;
86
87      // Do the SPI transmission
88      // Mask with DAC control bits
89      DAC_data_1 = (DAC_config_chan_A | (DAC_output_1 & 0xfff))  ;
90      DAC_data_2 = (DAC_config_chan_B | (DAC_output_2 & 0xfff))  ;
91
92      // SPI write (no spinlock b/c of SPI buffer)
93      spi_write16_blocking(SPI_PORT, &DAC_data_1, 1);
94      gpio_put(PIN_CS, 1);
95      gpio_put(PIN_CS, 0);
96      spi_write16_blocking(SPI_PORT, &DAC_data_2, 1);
97
98
99      gpio_put(PIN_CS, 1);
100
101 }
```

Figure 32: expending to two channels.

The first step should be done was to define some parameters again with different names to serve for the second channel. These parameters include “DAC_output”, “DAC_data”,

"DAC_config_chan", "accumulator", "increment", and "f_out". Therefore, there would be two sets of parameters working for two different channels. Figure 32 shows how I defined these two sets of parameters. Then, another major difference was the interrupt service routine. The objective of the interrupt service routine here was to achieve data transaction for both channels as calling interrupt service routine once. So, the interrupt service routine needed to be rearranged to achieve the goal. The organization of the new rearranged interrupt service routine became:

1. Pull the chip select low to enable the data transaction.
2. Enter the if statement to update the values of state variable *x, y, z* and hold the values for the next *m* interrupt service routine calls.
3. Calculate both "*f_out_1*" and. "*f_out_2*" based on the values of state variable *x, y, z* and hold the values of "*f_out_1*" and. "*f_out_2*" for the next *m* interrupt service routine calls.
4. Update the Direct Digital Synthesis Accumulator for both channel A and B.
5. Use the updated accumulator to index the sine table for channel A and B.
6. Masking with DAC control bits for both channel A and B.
7. Perform the SPI data transaction for channel A.
8. Pull the chip select high to stop the data transaction.
9. Pull the chip select low again to enable the data transaction.
10. Perform the SPI data transaction for channel B.
11. Pull the chip select high again to stop the data transaction.
12. Leave the interrupt service routine.

It was seen that the chip select pin should be pulled low again to allow another SPI data transaction for another channel in one interrupt service routine. I would build another project on RP2040 based on the C program in this section. As dragging the uf2 file to the Raspberry Pi Pico drive, both output channels of the DAC should output the oscillating analog frequency signals based on "*f_out_1*" or "*f_out_2*". As attaching the probes of the oscilloscope to either the V_{OUTA} pin or V_{OUTB} pin of the DAC, oscilloscope would show the oscillating "*f_out_1*" or "*f_out_2*". I would discuss the results shown on the oscilloscope in the Section 5.8. The core part of the RP2040 based chaotic sound synthesizer was finished. Then, the next step would be connecting the V_{OUTA} pin and V_{OUTB} pin of the DAC to an audio jack and hearing the chaotic sounds from the speaker.

5.7 Completing the circuit

Now, I could visualize the output result from the oscilloscope. However, the objective of this project was to sonify the Lorenz System instead of visualizing it. So, to convert the analog output frequency signals, I would connect an audio jack to the V_{OUTA} pin & V_{OUTB} pin and ground. As the power supply was on and the speaker was connected to the audio jack, I could sonify the Lorenz System, in other words, hear the chaos of the Lorenz System.

5.7.1 Audio Jack

Now, I had the analog sound signals generated from the V_{OUTA} pin and V_{OUTB} pin of the DAC. Therefore, I could hear the sonified Lorenz System from the device by an audio jack and a speaker. Figure 33 shows the audio jack I used in this project.

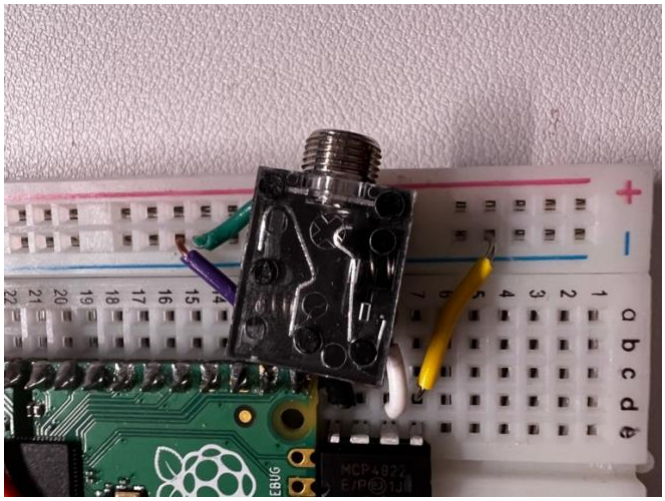


Figure 33: the audio jack used in this project.

There were three pins on the audio jack. The front pin was the ground, the left pin was the left channel, and the right pin was the right channel. The audio jack could be directly connected to the V_{OUTA} pin & V_{OUTB} pin of the DAC and the ground. So, the audio jack should be soldered with wires to connect to the V_{OUTA} pin, V_{OUTB} pin, and ground conveniently. As the audio jack was well connected and the uf2 was in the RP2040, the RP2040-based chaotic sound synthesizer was ready to be used. As I plugged the power supply into the MicroUSB of the Raspberry Pi Pico and connected a speaker to the audio jack, I would hear the chaotic sounds from the speaker. It means that the Lorenz System was successfully sonified by a RP2040 based chaotic sound synthesizer in realtime. The sound result would be discussed in Section 5.8.

5.8 Final Result and discussion

Now, the device -- RP2040-based chaotic sound synthesizer was finalized. You could always sonify Lorenz System from the speaker of the device as long as the power supply was on. In Section 5.6 and 5.7, the output result of the RP2040 based chaotic sound synthesizer could be either visualized by the oscilloscope or sonified by the audio jack and speaker. Figure 34 shows the oscillating output frequencies of the device shown on the oscilloscope. And Figure 35 shows the finalized RP2040 based chaotic sound synthesizer.

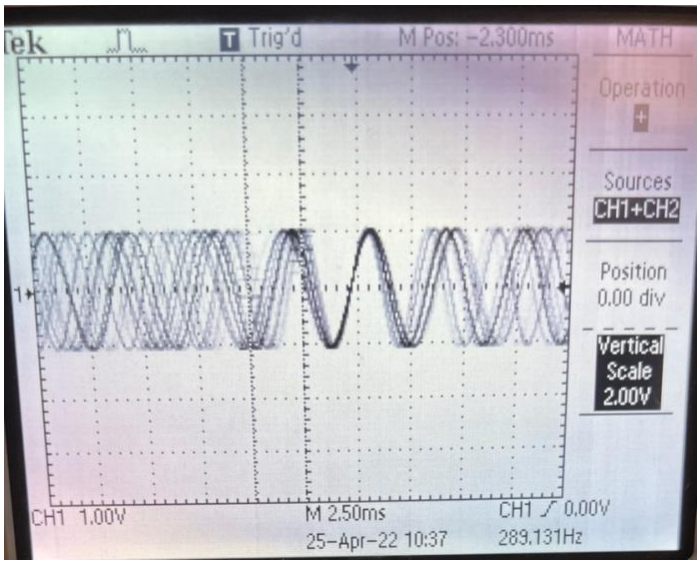


Figure 34: the oscillating output frequencies of the device shown on the oscilloscope

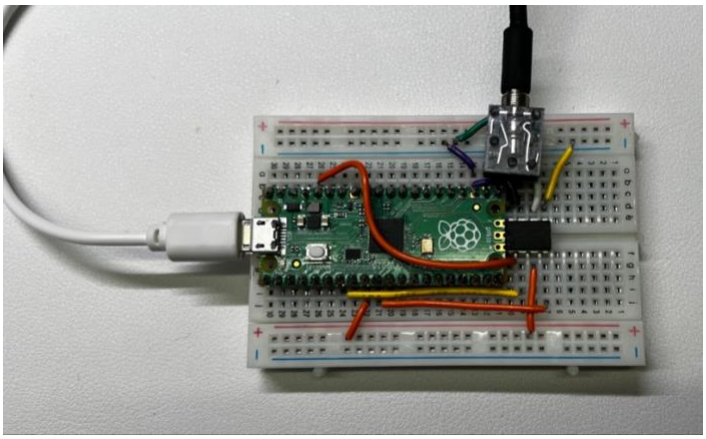


Figure 35: the finalized RP2040-based chaotic sound synthesizer

In Section 5.6, I used the same frequency modulation as Section 4.4. I had generated the plot of the oscillating output frequencies in Section 4.4. And the oscillating output frequencies of the device shown on the oscilloscope was the same as the plot in Section 4.4. Therefore, the device successfully reproduced the simulation results. The sound result in Section 4.4 was clearer since it was generated by the computer simulation. There were some noises in the sound I heard from the device speaker in Section 5.7 due to the hardware prototyping, however, the sound result from the device speaker showed the similar chaotic patterns with the sound results from the computer simulation. It was also reasonable.

As mentioned in previous sections, there were a number of parameters could be tuned to change the chaotic sound synthesis effects. These parameters include "*m (the relative implementation speed between the Euler-integrator and the Direct Digital Synthesis synthesizer)*", the base frequency, frequency modulations, state variables *x, y, z*, and Lorenz System parameters σ, ρ, β. I had done several experiments in the lab, as I tuned these parameters, the device would give me different feedback though the output sounds.

There was an intuitive relationship between the parameter "*m*" and the output sound. As the parameter "*m*" was increased, the chaos in the output sound were clearly oscillating more slowly. The change of the base frequency could also lead to apparently different feedbacks. Also, the different combination of state variables in both channels could give different sound synthesis effects. Therefore, there were various relationships between these parameters and the output sound synthesis effects. Exploring these relationships would be a major task in the future work.

6 Timeline

Date	Contents
Fall 2021 (Done) & Spring 2022 (Done)	Design Project: Chaotic Oscillator as a Sound Synthesizer Controller
Fall 2021 (Done)	Software Simulation
Oct 4 – Oct 19	Design Project Intro
Oct 19 – Oct 26	Lorenz System simulation on MATLAB

Oct 26 – Nov 9	Lorenz System simulation using “Euler Method” on MATLAB
Nov 9 – Nov 16	Lorenz System simulation with using “Euler Method in C Programming
Nov 16 – Nov 23	Direct Digital Synthesis simulation on MATLAB
Nov 23 – Nov 30	Thanksgiving Break
Nov 30 – Dec 8	First Draft of Project Progress Report
Dec 8 – Dec 14	Final Draft of Project Progress Report
Spring 2022 (Done)	Design Project: Hardware Prototyping
Jan 18 – Jan 21	Direct Digital Synthesis simulation in C
Jan 21 – Feb 28	Combination of Lorenz System and Direct Digital Synthesis in C
Feb 28 – Feb 4	Raspberry Pi Pico C/C++ development environment setup on Mac OS
Feb 4 – Feb 11	Raspberry Pi Pico C/C++ development environment setup on Windows
Feb 11 – Feb 18	Blinking an LED
Feb 18 – Feb 25	COM PORT Serial Connection
Feb 25 – Mar 11	Building the system circuit
Mar 11 – Mar 25	Software Development of Direct Digital Synthesis
Mar 25 – Apr 8	Software Development of Lorenz System
Apr 8 – Apr 15	Adding an Audio Jack
Apr 15 – Apr 22	Sonify the Lorenz System, Finalize the Device, and Demo to Advisors
Apr 22 – May 9	Innovation: Exploring the relationship between parameters and sound synthesis effects. Poster Session Preparation.
May 10	Poster Session
May 10 – May 16	First Draft of Final Report
May 16 – May 21	Second Draft of Final Report
May 21 – May 23	Final Draft of Final Report
May 23	Deadline of the Final Draft of Final Report

7 Acknowledgement

I would like to thank my advisors Prof. Van Hunter Adams and Prof. Bruce R. Land, who gave me this opportunity to do this special project and guided me throughout this MEng design project. Also, I would like to thank my academic advisor Scott E. Coldren, who helped me with the instructions of Design Project Selection & Poster Session.

8 Reference

[1] “Raspberry Pi Pico C/C++ SDK”. [Online] Available: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

[2] “Getting Started with Raspberry Pi Pico”. [Online] Available: https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf

[3] “MCP4802/4812/4822 Data Sheet”. [Online] Available: https://ww1.microchip.com/downloads/en/DeviceDoc/20002249B.pdf

[4] V.H. Adams, “Dual-core Direct Digital Synthesis (DDS) on RP2040 (Raspberry Pi Pico)”. [Online] Available: https://vha3.github.io/Pico/Multi/MultiCore.html

[5] V.H. Adams, “Setting up the Raspberry Pi Pico for C/C++ Development on Windows”. [Online] Available: https://vanhunteradams.com/Pico/Setup/PicoSetup.html

[6] V.H. Adams, “Creating a new C/C++ Raspberry Pi Pico Project on Windows”. [Online] Available: https://vanhunteradams.com/Pico/Setup/NewProjectWindows.html

[7] V.H. Adams, “Understanding the C/C++ SDK architecture for the Raspberry Pi Pico”. [Online] Available: https://vanhunteradams.com/Pico/Setup/SDKArchitecture.html

[8] V.H. Adams, “Direct Digital Synthesis”. [Online] Available: https://vanhunteradams.com/DDS/DDS.html

[9] Dugnolle, Thierry. “Streamline in a Lorenz system.” [Online] Available: https://commons.wikim edia.org/wiki/File:Streamline_in_a_Lorenz_system.gif#file.