Circle of Fifths Euclidean Synth with synthio and CircuitPython

Created by Liz Clark


Last updated on 2023-08-29 04:41:40 PM EDT
# Table of Contents

**Overview**
- What are Euclidean Rhythms?
- Parts

**Circuit Diagram**

**3D Printing**

**Install CircuitPython**
- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

**Code the Synth**
- Upload the Code and Libraries to the Feather RP2040
- How the CircuitPython Code Works
- Tones
- I2C
- Conjunction Function
- ADSR Prep
- The Loop

**Setting I2C Addresses**

**Wiring**

**STEMMA Wiring**

**Assembly**
- STEMMA Component Mounting
- Mount the Feather and Amp
- Mount and Connect the Speaker
- USB-C Extension
- Connect to the Feather
- In Closing

**Making Beautiful Music**
Overview

This synth celebrates all things circular: the circle of fifths, Euclidean rhythms, and rotary encoders. Four synth voices play random notes in a triad to the beat of a determined Euclidean rhythm animated on the 8x8 matrix. You can scroll through the circle of fifths on each synth voice to change the triad for easy modulation between keys.

The code is written in CircuitPython with the synthio module and runs on a Feather RP2040. An I2S amp outputs audio to a speaker and the rotary encoders, alphanumeric displays, and 8x8 matrix all connect to the Feather over I2C with STEMMA QT cables.
The synth has multiple functions that can be selected using the top encoder:

Play - the synth plays and you can adjust the chords for each of the four synth voices
Euclidean - adjust the Euclidean rhythm for each voice
BPM - raise or lower the Beats Per Minute of the sequence (speed)
Beat - change the beat division of the sequence from a range of 1/16th notes to whole notes
Wave - change the waveform for each synth voice to a square wave, sine wave, sawtooth wave, or noise
ADSR - adjust the attack, decay, sustain, and release for each synth voice
Ring - adjust the percentage of ring modulation applied to each synth voice
LFO - adjust the rate of the LFO being used for ring modulation
Volume - control the overall volume of the synth

What are Euclidean Rhythms?

Euclidean rhythms are derived from an algorithm that determines how to spread the number of steps over a number of beats as equally as possible. The interplay between different time signatures can create complex and interesting polyrhythms, even something as simple as one person clapping in 4/4 time while another person claps in 6/8 can sound interesting!

Euclidean algorithms have become popular as an option in Eurorack modules that generate rhythms. C code for this algorithm was originally referenced in Godfried Toussaint's paper and Brian House ported the algorithm to Python in this repository. You'll see this function in the CircuitPython code.
The Euclidean rhythm for each synth voice is controlled with the four lower encoders in Euclidean mode. The 8x8 matrix displays the total number of steps in each rhythm. As the rhythms advance, a pixel on the matrix turns red when a note plays to denote the place in the sequence.

Parts

Adafruit Feather RP2040
A new chip means a new Feather, and the Raspberry Pi RP2040 is no exception. When we saw this chip we thought "this chip is going to be awesome when we give it the Feather..."
https://www.adafruit.com/product/4884

Adafruit I2S 3W Class D Amplifier Breakout - MAX98357A
Listen to this good news - we now have an all in one digital audio amp breakout board that works incredibly well with the
https://www.adafruit.com/product/3006
Mono Enclosed Speaker - 3W 4 Ohm
Listen up! This 2.8" x 1.2" speaker is a great addition to any audio project where you need 4 ohm impedance and 3W or less of power. We particularly like...
https://www.adafruit.com/product/3351

Adafruit ANO Rotary Navigation Encoder to I2C Stemma QT Adapter
The ANO rotary encoder wheel is a funky user interface element, reminiscent of the original...
https://www.adafruit.com/product/5740

ANO Directional Navigation and Scroll Wheel Rotary Encoder
This funky user interface element is reminiscent of the original clicking scroll wheel interface...
https://www.adafruit.com/product/5001

Quad Alphanumeric Display - Red 0.54" Digits w/ I2C Backpack
Display, elegantly, 012345678 or 9! Gaze, hypnotized, at ABCDEFGHIJKLM - well it can display the whole alphabet. You get the point. This is a nice, bright alphanumeric display that...
https://www.adafruit.com/product/1911
<table>
<thead>
<tr>
<th>Item Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x <strong>Quad Alphanumeric Display</strong> Green</td>
<td><a href="https://www.adafruit.com/product/2158">https://www.adafruit.com/product/2158</a></td>
</tr>
<tr>
<td>1 x <strong>Quad Alphanumeric Display</strong> Blue</td>
<td><a href="https://www.adafruit.com/product/2160">https://www.adafruit.com/product/2160</a></td>
</tr>
<tr>
<td>1 x <strong>Quad Alphanumeric Display</strong> White</td>
<td><a href="https://www.adafruit.com/product/2157">https://www.adafruit.com/product/2157</a></td>
</tr>
<tr>
<td>1 x <strong>Adafruit Bicolor LED Square Pixel Matrix</strong></td>
<td><a href="https://www.adafruit.com/product/902">https://www.adafruit.com/product/902</a></td>
</tr>
<tr>
<td><strong>I2C Backpack</strong></td>
<td></td>
</tr>
<tr>
<td><strong>4 x STEMMA QT cable</strong></td>
<td><a href="https://www.adafruit.com/product/4210">https://www.adafruit.com/product/4210</a></td>
</tr>
<tr>
<td><strong>100 mm</strong></td>
<td></td>
</tr>
<tr>
<td><strong>8 x STEMMA QT cable</strong></td>
<td><a href="https://www.adafruit.com/product/4399">https://www.adafruit.com/product/4399</a></td>
</tr>
<tr>
<td><strong>50 mm</strong></td>
<td></td>
</tr>
<tr>
<td>1 x <strong>USB C Round Panel Mount</strong> Extension cable</td>
<td><a href="https://www.adafruit.com/product/4218">https://www.adafruit.com/product/4218</a></td>
</tr>
<tr>
<td>1 x <strong>USB C to A cable</strong></td>
<td><a href="https://www.adafruit.com/product/5153">https://www.adafruit.com/product/5153</a></td>
</tr>
<tr>
<td><strong>1 m</strong></td>
<td></td>
</tr>
<tr>
<td>1 x <strong>M2.5 Screws and Stand-offs</strong> M2.5</td>
<td><a href="https://www.adafruit.com/product/3299">https://www.adafruit.com/product/3299</a></td>
</tr>
<tr>
<td>1 x <strong>M3 Screws and Stand-offs</strong> M3</td>
<td><a href="https://www.adafruit.com/product/4685">https://www.adafruit.com/product/4685</a></td>
</tr>
<tr>
<td>1 x <strong>Silicone Wire</strong></td>
<td><a href="https://www.adafruit.com/product/2051">https://www.adafruit.com/product/2051</a></td>
</tr>
<tr>
<td><strong>30AWG, multiple colors</strong></td>
<td></td>
</tr>
</tbody>
</table>
Most of the wiring for this project is accomplished with STEMMA QT cables. The ANO rotary encoder breakouts, alphanumeric display breakouts and bicolor LED matrix all communicate over I2C and daisy chain together to attach to the Feather RP2040.

The Fritzing diagram has the I2C addresses for each breakout listed and the [Setting I2C Addresses](#) page in this guide walks through which jumpers need to be adjusted for each breakout.

The I2S amp does require some wiring and soldering though:

- I2S amp LRC to Feather pin 11 (white wire)
- I2S amp BCLK to Feather pin 10 (yellow wire)
- I2S amp DIN to Feather pin 9 (blue wire)
- I2S amp GND to Feather GND (black wire)
- I2S amp VIN to Feather USB (red wire)
- I2S amp GAIN to I2S amp GND (green wire)

The speaker is connected to the I2S amp speaker output terminal block.

- Speaker negative to I2S amp - output (black wire)
- Speaker positive to I2S amp + output (red wire)
3D Printing

The synth may be housed in a 3D printed enclosure described below. It consists of two parts: the case and the lid. The lid attaches to the case with M3 screws.

STL Files on Printables

circle_of_fifths_synth_stl_files.zip

The case benefits some supports for the speaker and USB-C mount cutouts. In your slicer, you can generate supports and adjust the Support Overhang Angle to 75.0 to limit the amount of supports generated.
The lid has cutouts and mounting holes for the rotary encoders, alphanumeric displays and the bicolor LED matrix.

The case has columns that act as supports for the lid that is otherwise only secured in the corners. There are also stand-offs to attach the Feather and I2S amp.

Install CircuitPython

CircuitPython () is a derivative of MicroPython () designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY drive to iterate.

CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython running on your board.

Download the latest version of CircuitPython for this board via circuitpython.org
Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.

To enter the bootloader, hold down the BOOT/BOOTSEL button (highlighted in red above), and while continuing to hold it (don't let go!), press and release the reset button (highlighted in blue above). Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!

If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

A lot of people end up using charge-only USB cables and it is very frustrating! Make sure you have a USB cable you know is good for data sync.
You will see a new disk drive appear called RPI-RP2.

Drag the adafruit_circuitpython_etc.uf2 file to RPI-RP2.

The RPI-RP2 drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)

Safe Mode

You want to edit your code.py or modify the files on your CIRCUITPY drive, but find that you can't. Perhaps your board has gotten into a state where CIRCUITPY is read-only. You may have turned off the CIRCUITPY drive altogether. Whatever the reason, safe mode can help.
Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in boot.py (where you can set CIRCUITPY read-only or turn it off completely). Second, it does not run the code in code.py. And finally, it does not automatically soft-reload when data is written to the CIRCUITPY drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the CIRCUITPY drive.

Entering Safe Mode

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.
CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.
Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the CIRCUITPY drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

Flash Resetting UF2

If your board ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 which will do a 'deep
clean' on your Flash Memory. You will lose all the files on the board, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

**Code the Synth**

Once you've finished setting up your Feather RP2040 with CircuitPython, you can access the code and necessary libraries by downloading the Project Bundle.

To do this, click on the Download Project Bundle button in the window below. It will download as a zipped folder.

```python
# SPDX-FileCopyrightText: 2023 Liz Clark for Adafruit Industries
# SPDX-License-Identifier: MIT

from random import randint
import ulab.numpy as np
import board
import audiobusio
import audiomixer
import synthio
import simpleio
from adafruit_ticks import ticks_ms, ticks_add, ticks_diff
from adafruit_ht16k33 import segments
from adafruit_ht16k33.matrix import Matrix8x8x2
from adafruit_seesaw import seesaw, rotaryio, digitalio

SAMPLE_RATE = 44100
SAMPLE_SIZE = 256
VOLUME = 5000

# waveforms, envelopes and synth setup

square = np.concatenate((np.ones(SAMPLE_SIZE//2, dtype=np.int16)*VOLUME, np.ones(SAMPLE_SIZE//2, dtype=np.int16)*-VOLUME))
sine = np.array(np.sin(np.linspace(0, 4*np.pi, SAMPLE_SIZE, endpoint=False)) * VOLUME, dtype=np.int16)
saw = np.linspace(VOLUME, -VOLUME, num=SAMPLE_SIZE, dtype=np.int16)
noise = np.array([randint(-VOLUME, VOLUME) for i in range(SAMPLE_SIZE)], dtype=np.int16)
lfo = synthio.LFO(rate = .5, waveform = sine)

amp_env0 = synthio.Envelope(attack_time=0.1, decay_time = 0.1, release_time=0.1, attack_level=1, sustain_level=0.05)
amp_env1 = synthio.Envelope(attack_time=0.05, decay_time = 0.1, release_time=0.1, attack_level=1, sustain_level=0.05)

# synth plays the notes
synth = synthio.Synthesizer(sample_rate=SAMPLE_RATE)

# these are the notes
synth0 = synthio.Note(frequency = 0.0, envelope=amp_env0, waveform=square, ring_frequency = 0, ring_bend = lfo, ring_waveform = sine)
```
synth1 = synthio.Note(frequency = 0.0, envelope=amp_env1, waveform=sine,
    ring_frequency = 0,
    ring_bend = lfo, ring_waveform = sine)
synth2 = synthio.Note(frequency = 0.0, envelope=amp_env0, waveform=square,
    ring_frequency = 0,
    ring_bend = lfo, ring_waveform = sine)
synth3 = synthio.Note(frequency = 0.0, envelope=amp_env1, waveform=sine,
    ring_frequency = 0,
    ring_bend = lfo, ring_waveform = sine)
synths = [synth0, synth1, synth2, synth3]
wave_names = ["SQUR", "SINE", "SAW ", "NOIZ"]
waveforms = [square, sine, saw, noise]
synth0_wave = 0
synth1_wave = 1
synth2_wave = 0
synth3_wave = 1

# i2s amp setup
audio = audiobusio.I2SOut(bit_clock=board.D10, word_select=board.D11, data=board.D9)
mixer = audiomixer.Mixer(voice_count=4, sample_rate=SAMPLE_RATE, channel_count=1,
    bits_per_sample=16, samples_signed=True, buffer_size=2048 )
audio.play(mixer)
vol_val = 2
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.3

# these are the triads, all major
c_tones = [130.81, 164.81, 196.00]
g_tones = [196.00, 246.94, 293.66]
d_tones = [146.83, 185.00, 220.00]
a_tones = [220.00, 277.18, 329.63]
e_tones = [164.81, 207.65, 246.94]
b_tones = [246.94, 311.13, 369.99]
fsharp_tones = [185.00, 233.08, 277.18]
csharp_tones = [138.59, 174.61, 207.65]
aflat_tones = [207.65, 261.63, 311.13]
eflat_tones = [155.56, 196.00, 233.08]
bflat_tones = [233.08, 293.66, 349.23]
f_tones = [174.61, 220.00, 261.63]

# names for the alphanumeric displays
chord_names = ["Cmaj", "Gmaj", "Dmaj", "Amaj", "Emaj", "Bmaj",
    "F#ma", "C#ma", "Abma", "Ebma", "Bbma", "Fmaj"]
chords = [c_tones, g_tones, d_tones, a_tones, e_tones, b_tones, fsharp_tones,
    csharp_tones,
    aflat_tones, eflat_tones, bflat_tones, f_tones]

# i2c setup
i2c = board.I2C()
# the encoders
seesaw0 = seesaw.Seesaw(i2c, addr=0x49)
seesaw1 = seesaw.Seesaw(i2c, addr=0x4A)
seesaw2 = seesaw.Seesaw(i2c, addr=0x4B)
seesaw3 = seesaw.Seesaw(i2c, addr=0x4C)
menu_seesaw = seesaw.Seesaw(i2c, addr=0x4D)
# the alphanumeric displays
display0 = segments.Seg14x4(i2c, address=0x70)
display1 = segments.Seg14x4(i2c, address=0x71)
display2 = segments.Seg14x4(i2c, address=0x72)
display3 = segments.Seg14x4(i2c, address=0x73)
menu_display = segments.Seg14x4(i2c, address=0x74)
# the matrix
matrix0 = Matrix8x8x2(i2c, address=0x75)

seesaws = [seesaw0, seesaw1, seesaw2, seesaw3, menu_seesaw]
buttons0 = []
buttons1 = []
buttons2 = []
buttons3 = []
menu_buttons = []
button0_states = []
button1_states = []
button2_states = []
button3_states = []
menu_states = []
button0_names = ['Select', 'Up', 'Left', 'Down', 'Right']

# setup the buttons on all of the encoders
for i in range(1, 6):
    seesaw0.pin_mode(i, seesaw0.INPUT_PULLUP)
    seesaw1.pin_mode(i, seesaw1.INPUT_PULLUP)
    seesaw2.pin_mode(i, seesaw2.INPUT_PULLUP)
    seesaw3.pin_mode(i, seesaw3.INPUT_PULLUP)
    menu_seesaw.pin_mode(i, menu_seesaw.INPUT_PULLUP)
buttons0.append(digitalio.DigitalIO(seesaw0, i))
buttons1.append(digitalio.DigitalIO(seesaw1, i))
buttons2.append(digitalio.DigitalIO(seesaw2, i))
buttons3.append(digitalio.DigitalIO(seesaw3, i))
menu_buttons.append(digitalio.DigitalIO(menu_seesaw, i))
button0_states.append(False)
button1_states.append(False)
button2_states.append(False)
button3_states.append(False)
menu_states.append(False)

# make all of the encoders
encoder0 = rotaryio.IncrementalEncoder(seesaw0)
last_position0 = 0
encoder1 = rotaryio.IncrementalEncoder(seesaw1)
last_position1 = 0
encoder2 = rotaryio.IncrementalEncoder(seesaw2)
last_position2 = 0
encoder3 = rotaryio.IncrementalEncoder(seesaw3)
last_position3 = 0
menu_enc = rotaryio.IncrementalEncoder(menu_seesaw)
last_menuPosition = 0

# Python Implementation of Björklund's Algorithm by Brian House
# MIT License 2011
# https://github.com/brianhouse/bjorklund

def bjorklund(steps, pulses):
    steps = int(steps)
    pulses = int(pulses)
    if pulses > steps:
        raise ValueError
    pattern = []
    counts = []
    remainders = []
    divisor = steps - pulses
    remainders.append(pulses)
    while True:
        counts.append(divisor // remainders[level])
        remainders.append(divisor % remainders[level])
        divisor = remainders[level]
        level = level + 1
        if remainders[level] <= 1:
            break
    counts.append(divisor)

def build(level):
    if level == -1:
        pattern.append(0)
    elif level == -2:
        pattern.append(1)
    else:
        pass

©Adafruit Industries
for _ in range(0, counts[level]):
    build(level - 1)
if remainders[level] != 0:
    build(level - 2)

build(level)
p = pattern.index(1)
pattern = pattern[p:] + pattern[0:p]
return pattern

# using ticks for time tracking
clock = ticks_ms()

# default BPM
bpm = 120

# beat divison
beat_div = [15, 30, 60, 120, 240]
beat_index = 2
beat_names = ["1/16", "1/8", "1/4", "1/2", "HOLE"]
delay = int((beat_div[beat_index] / bpm) * 1000)

# variables for euclidean
c0 = 0
c1 = 0
c2 = 0
c3 = 0
r0 = 0
r1 = 0
r2 = 0
r3 = 0
last_r0 = 0
last_r1 = 0
last_r2 = 0
last_r3 = 0
euclid0_steps = 8
euclid0_pulses = 4
euclid1_steps = 8
euclid1_pulses = 4
euclid2_steps = 8
euclid2_pulses = 4
euclid3_steps = 8
euclid3_pulses = 4

rhythm0 = bjorklund(euclid0_steps, euclid0_pulses)
rhythm1 = bjorklund(euclid1_steps, euclid1_pulses)
rhythm2 = bjorklund(euclid2_steps, euclid2_pulses)
rhythm3 = bjorklund(euclid3_steps, euclid3_pulses)

# read buttons to update Euclidean rhythms
# pylint: disable=too-many-branches
def read_buttons(button_array, button_states, euc, e_step, e_pulse, the_step):
    for b in range(5):
        if not button_array[b].value and button_states[b] is False:
            button_states[b] = True
            if button0_names[b] == "Select":
                e_step = 8
                e_pulse = 4
                if the_step >= e_step:
                    the_step = 0
            elif button0_names[b] == "Up":
                if e_step > 16:
                    e_step = 16
                else:
                    e_step += 1
            elif button0_names[b] == "Down":
                if e_step < 1:
                    e_step = 1
else:
    e_step -= 1
if the_step >= e_step:
    the_step = 0
elif button0_names[b] == "Left":
    e_pulse -= 1
    e_pulse = max(e_pulse, 1)
else:
    e_pulse += 1
    e_pulse = min(e_pulse, e_step)
euc = bjorklund(e_step, e_pulse)
if button_array[b].value and button_states[b] is True:
    button_states[b] = False
if button0_names[b] in ("Select", "Up", "Down"):
    matrix0.fill(matrix0.LED_OFF)
    draw_steps(euclid0_steps, 0)
    draw_steps(euclid1_steps, 2)
    draw_steps(euclid2_steps, 4)
    draw_steps(euclid3_steps, 6)
return euc, e_step, e_pulse, the_step

# play euclidean rhythms and update matrix
def play_euclidean(this_synth, n, the_rhythm, rhythm_count, last_count, c, matrix_slot):
    if last_count <= 7:
        matrix0[matrix_slot, last_count] = matrix0.LED_GREEN
    else:
        c += 1
        matrix0[matrix_slot + 1, (last_count - last_count) + c] = matrix0.LED_GREEN
    c += 1
    if the_rhythm[rhythm_count] == 1:
        this_synth.frequency = n[randint(0, 2)]
        synth.press(this_synth)
        if rhythm_count <= 7:
            matrix0[matrix_slot, rhythm_count] = matrix0.LED_RED
        else:
            matrix0[matrix_slot + 1, (rhythm_count - rhythm_count) + c] =
            matrix0.LED_RED
            c += 1
        synth.release(this_synth)
    elif rhythm_count > 7:
        c += 1
    last_count = rhythm_count
    rhythm_count += 1
    if rhythm_count >= len(the_rhythm):
        rhythm_count = 0
    if rhythm_count == 1:
        c = 0
    return rhythm_count, last_count, c

# initial matrix draw
def draw_steps(euc_steps, col):
    dif = 0
    for m in range(euc_steps):
        if m <= 7:
            matrix0[col, m] = matrix0.LED_GREEN
        else:
            matrix0[col + 1, (m - m) + dif] = matrix0.LED_GREEN
        dif += 1
draw_steps(euclid0_steps, 0)
draw_steps(euclid1_steps, 2)
draw_steps(euclid2_steps, 4)
draw_steps(euclid3_steps, 6)

# clocks for playing euclidean and reading menu encoder
enc_clock = ticks_ms()
menu_clock = ticks_ms()

# the modes menu
modes = ["PLAY", "EUC ", "BPM ", "BEAT", "ADSR", "WAVE", "RING", "LFO ", "VOL "]
mode_index = 0
mode = modes[mode_index]
menu_display.print(f"   {mode}"

# default chords
chord0_sel = 0
chord1_sel = 1
chord2_sel = 0
chord3_sel = 1
display0.print(chord_names[chord0_sel])
display1.print(chord_names[chord1_sel])
display2.print(chord_names[chord2_sel])
display3.print(chord_names[chord3_sel])

# arrays of individual buttons
select_buttons = [buttons0[0], buttons1[0], buttons2[0], buttons3[0]]
left_buttons = [buttons0[2], buttons1[2], buttons2[2], buttons3[2]]
right_buttons = [buttons0[4], buttons1[4], buttons2[4], buttons3[4]]
select_states = [button0_states[0], button1_states[0], button2_states[0], button3_states[0]]
left_states = [button0_states[2], button1_states[2], button2_states[2], button3_states[2]]
right_states = [button0_states[4], button1_states[4], button2_states[4], button3_states[4]]
select_index = 0
left_index = 0
right_index = 0

# adsr mode
adsr_names = ["A", "D", "S", "R"]
synth_adsr_indexes = [0, 0, 0, 0]
adsr_properties = [0, 1, 4, 2]

adsr0_values = [amp_env0.attack_time, amp_env0.decay_time, amp_env0.sustain_level, amp_env0.release_time]
adsr1_values = [amp_env1.attack_time, amp_env1.decay_time, amp_env1.sustain_level, amp_env1.release_time]
adsr2_values = [amp_env0.attack_time, amp_env0.decay_time, amp_env0.sustain_level, amp_env0.release_time]
adsr3_values = [amp_env1.attack_time, amp_env1.decay_time, amp_env1.sustain_level, amp_env1.release_time]

all_adsr_values = [adsr0_values, adsr1_values, adsr2_values, adsr3_values]
adsr0_val = int(simpleio.map_range(amp_env0.attack_time, 0.0, 1.0, 0, 19))
adsr1_val = int(simpleio.map_range(amp_env0.decay_time, 0.0, 1.0, 0, 19))
adsr2_val = int(simpleio.map_range(amp_env0.sustain_level, 0.0, 1.0, 0, 19))
adsr3_val = int(simpleio.map_range(amp_env0.release_time, 0.0, 1.0, 0, 19))

clock_stretch = False

ring0_val = 0
ring1_val = 0
ring2_val = 0
ring3_val = 0
lfo_val = 0
# used to play/pause
play_states = [True, True, True, True, True]

while True:
    # rotary encoder reading
    if ticks_diff(ticks_ms(), enc_clock) >= 100:
        position0 = encoder0.position
        position1 = encoder1.position
        position2 = encoder2.position
        position3 = encoder3.position
        menuPosition = menu_enc.position

    # menu changes mode
    if menuPosition != last_menuPosition:
        mode_index = (mode_index + 1) % len(modes)
        if mode in ("EUC", "ADSR"):
            clock_stretch = True
        if mode in ("PLAY", "BPM", "BEAT", "WAVE") and clock_stretch:
            clock = ticks_ms()
            clock_stretch = False
        mode = modes[mode_index]
        menu_display.print(f"   {mode}"

    last_menuPosition = menuPosition

    # encoder functionality depends on mode
    # encoder 0 has most functionality
    if position0 != last_position0:
        if position0 > last_position0:
            if mode == "PLAY":
                chord0_sel = (chord0_sel + 1) % len(chords)
                display0.print(chord_names[chord0_sel])
            elif mode == "BEAT":
                beat_index = (beat_index + 1) % 5
                delay = int((beat_div[beat_index] / bpm) * 1000)
                display0.print(f"   {beat_names[beat_index]}"
            elif mode == "BPM":
                bpm += 1
                delay = int((beat_div[beat_index] / bpm) * 1000)
                display0.print(f"   {bpm}""
            elif mode == "ADSR":
                adsr0_val = (adsr0_val + 1) % 20
                mapped_val = simpleio.map_range(adsr0_val, 0, 19, 0.0, 1.0)
                all_adsr_values[0][synth_adsr_indexes[0]] = mapped_val
                the_env = synthio.Envelope(attack_time=all_adsr_values[0][0],
                                            decay_time=all_adsr_values[0][1],
                                            release_time=all_adsr_values[0][2],
                                            attack_level=1,
                                            sustain_level=all_adsr_values[0][3])
                synth0.envelope = the_env
            elif mode == "WAVE":
                synth0_wave = (synth0_wave + 1) % len(wave_names)
                synth0.waveform = waveforms[synth0_wave]
            elif mode == "RING":
                ring0_val = (ring0_val + 1) % 25
                mapped_val = simpleio.map_range(ring0_val, 0, 24, 0.0, 220.0)
                synth0.ring_frequency = mapped_val
            elif mode == "LFO":
                lfo_val = (lfo_val + 1) % 10
                mapped_val = simpleio.map_range(lfo_val, 0, 9, 0.0, 5.0)
                lfo.rate = mapped_val
            elif mode == "VOL":
                vol_val = (vol_val + 1) % 10
                mapped_val = simpleio.map_range(vol_val, 0, 9, 0.0, 1.0)
                mixer.voice[0].level = mapped_val
        else:
            if mode == "PLAY":
                chord0_sel = (chord0_sel - 1) % len(chords)
                display0.print(chord_names[chord0_sel])
elif mode == "BEAT":
    beat_index = (beat_index - 1) % 5
    delay = int((beat_div[beat_index] / bpm) * 1000)
    display0.print(f"   {beat_names[beat_index]}")
elif mode == "BPM":
    bpm -= 1
    display0.print(f"   {bpm}")
elif mode == "ADSR":
    adsr0_val = (adsr0_val - 1) % 20
    mapped_val = simpleio.map_range(adsr0_val, 0, 19, 0.0, 1.0)
    all_adsr_values[0][synth_adsr_indexes[0]] = mapped_val
    the_env = synthio.Envelope(attack_time=all_adsr_values[0][0],
                               decay_time = all_adsr_values[0][1],
                               release_time=all_adsr_values[0][3],
                               attack_level=1,
                               sustain_level=all_adsr_values[0][2])
    synth0.envelope = the_env
elif mode == "WAVE":
    synth0_wave = (synth0_wave - 1) % len(wave_names)
    synth0.waveform = waveforms[synth0_wave]
elif mode == "RING":
    ring0_val = (ring0_val - 1) % 25
    mapped_val = simpleio.map_range(ring0_val, 0, 24, 0.0, 220.0)
    synth0.ring_frequency = mapped_val
elif mode == "LFO":
    lfo_val = (lfo_val - 1) % 10
    mapped_val = simpleio.map_range(lfo_val, 0, 9, 0.0, 5.0)
    lfo.rate = mapped_val
elif mode == "VOL":
    vol_val = (vol_val - 1) % 10
    mapped_val = simpleio.map_range(vol_val, 0, 9, 0.0, 1.0)
    mixer.voice[0].level = mapped_val
last_position0 = position0
if position1 != last_position1:
    if position1 > last_position1:
        if mode == "PLAY":
            chord1_sel = (chord1_sel + 1) % len(chords)
            display1.print(chord_names[chord1_sel])
        elif mode == "ADSR":
            adsr1_val = (adsr1_val + 1) % 20
            mapped_val = simpleio.map_range(adsr1_val, 0, 19, 0.0, 1.0)
            all_adsr_values[1][synth_adsr_indexes[1]] = mapped_val
            the_env = synthio.Envelope(attack_time=all_adsr_values[1][0],
                                       decay_time = all_adsr_values[1][1],
                                       release_time=all_adsr_values[1][3],
                                       attack_level=1,
                                       sustain_level=all_adsr_values[1][2])
            synth1.envelope = the_env
elif mode == "WAVE":
    synth1_wave = (synth1_wave + 1) % len(wave_names)
    synth1.waveform = waveforms[synth1_wave]
elif mode == "RING":
    ring1_val = (ring1_val + 1) % 25
    mapped_val = simpleio.map_range(ring1_val, 0, 24, 0.0, 220.0)
    synth1.ring_frequency = mapped_val
else:
    if mode == "PLAY":
        chord1_sel = (chord1_sel - 1) % len(chords)
        display1.print(chord_names[chord1_sel])
    elif mode == "ADSR":
        adsr1_val = (adsr1_val - 1) % 20
        mapped_val = simpleio.map_range(adsr1_val, 0, 19, 0.0, 1.0)
        all_adsr_values[1][synth_adsr_indexes[1]] = mapped_val
        the_env = synthio.Envelope(attack_time=all_adsr_values[1][0],
                                   decay_time = all_adsr_values[1][1],
                                   release_time=all_adsr_values[1][3],
                                   attack_level=1,
                                   sustain_level=all_adsr_values[1][2])
        synth1.envelope = the_env
elif mode == "WAVE":
    synth1_wave = (synth1_wave - 1) % len(wave_names)
    synth1.waveform = waveforms[synth1_wave]
elif mode == "RING":
    ring1_val = (ring1_val - 1) % 25
    mapped_val = simpleio.map_range(ring1_val, 0, 24, 0.0, 220.0)
    synth1.ring_frequency = mapped_val
last_position1 = position1
if position2 != last_position2:
    if position2 > last_position2:
        if mode == "PLAY":
            chord2_sel = (chord2_sel + 1) % len(chords)
        elif mode == "ADSR":
            adsr2_val = (adsr2_val + 1) % 20
            mapped_val = simpleio.map_range(adsr2_val, 0, 19, 0.0, 1.0)
            all_adsr_values[2][synth_adsr_indexes[2]] = mapped_val
            the_env = synthio.Envelope(attack_time=all_adsr_values[2][0],
                                      decay_time = all_adsr_values[2][1],
                                      release_time=all_adsr_values[2][3],
                                      attack_level=1,
                                      sustain_level=all_adsr_values[2][2])
            synth2.envelope = the_env
elif mode == "WAVE":
    synth2_wave = (synth2_wave + 1) % len(wave_names)
    synth2.waveform = waveforms[synth2_wave]
elif mode == "RING":
    ring2_val = (ring2_val + 1) % 25
    mapped_val = simpleio.map_range(ring2_val, 0, 24, 0.0, 220.0)
    synth2.ring_frequency = mapped_val
else:
    if mode == "PLAY":
        chord2_sel = (chord2_sel - 1) % len(chords)
        display2.print(chord_names[chord2_sel])
    elif mode == "ADSR":
        adsr2_val = (adsr2_val - 1) % 20
        mapped_val = simpleio.map_range(adsr2_val, 0, 19, 0.0, 1.0)
        all_adsr_values[2][synth_adsr_indexes[2]] = mapped_val
        the_env = synthio.Envelope(attack_time=all_adsr_values[2][0],
                                   decay_time = all_adsr_values[2][1],
                                   release_time=all_adsr_values[2][3],
                                   attack_level=1,
                                   sustain_level=all_adsr_values[2][2])
        synth2.envelope = the_env
elif mode == "WAVE":
    synth2_wave = (synth2_wave - 1) % len(wave_names)
    synth2.waveform = waveforms[synth2_wave]
    elif mode == "RING":
        ring2_val = (ring2_val - 1) % 25
        mapped_val = simpleio.map_range(ring2_val, 0, 24, 0.0, 220.0)
        synth2.ring_frequency = mapped_val
last_position2 = position2
if position3 != last_position3:
    if position3 > last_position3:
        if mode == "PLAY":
            chord3_sel = (chord3_sel + 1) % len(chords)
            display3.print(chord_names[chord3_sel])
        elif mode == "ADSR":
            adsr3_val = (adsr3_val + 1) % 20
            mapped_val = simpleio.map_range(adsr3_val, 0, 19, 0.0, 1.0)
            all_adsr_values[3][synth_adsr_indexes[3]] = mapped_val
            the_env = synthio.Envelope(attack_time=all_adsr_values[3][0],
                                       decay_time = all_adsr_values[3][1],
                                       release_time=all_adsr_values[3][3],
                                       attack_level=1,
                                       sustain_level=all_adsr_values[3][2])
            synth3.envelope = the_env
elif mode == "WAVE":
    synth3_wave = (synth3_wave + 1) % len(wave_names)
    synth3.waveform = waveforms[synth3_wave]
elif mode == "RING":
    ring3_val = (ring3_val + 1) % 25
    mapped_val = simpleio.map_range(ring3_val, 0, 24, 0.0, 220.0)
    synth3.ring_frequency = mapped_val
else:
    if mode == "PLAY":
        chord3_sel = (chord3_sel - 1) % len(chords)
        display3.print(chord_names[chord3_sel])
    elif mode == "ADSR":
        adsr3_val = (adsr3_val - 1) % 20
        mapped_val = simpleio.map_range(adsr3_val, 0, 19, 0.0, 1.0)
        all_adsr_values[3][synth_adsr_indexes[3]] = mapped_val
        the_env = synthio.Envelope(attack_time=all_adsr_values[3][0],
                                                 decay_time = all_adsr_values[3][1],
                                                 release_time=all_adsr_values[3][3],
                                                 attack_level=1,
                                                 sustain_level=all_adsr_values[3][2])
        synth3.envelope = the_env
    elif mode == "WAVE":
        synth3_wave = (synth3_wave - 1) % len(wave_names)
        synth3.waveform = waveforms[synth3_wave]
    elif mode == "RING":
        ring3_val = (ring3_val - 1) % 25
        mapped_val = simpleio.map_range(ring3_val, 0, 24, 0.0, 220.0)
        synth3.ring_frequency = mapped_val
    last_position3 = position3
    enc_clock = ticks_add(enc_clock, 100)

    # synth plays based on ticks timing
    if ticks_diff(ticks_ms(), clock) >= delay:
        if play_states[0] is True:
            r0, last_r0, c0 = play_euclidean(synth0, chords[chord0_sel],
                                           rhythm0, r0, last_r0, c0, 0)
        if play_states[1] is True:
            r1, last_r1, c1 = play_euclidean(synth1, chords[chord1_sel],
                                           rhythm1, r1, last_r1, c1, 2)
        if play_states[2] is True:
            r2, last_r2, c2 = play_euclidean(synth2, chords[chord2_sel],
                                           rhythm2, r2, last_r2, c2, 4)
        if play_states[3] is True:
            r3, last_r3, c3 = play_euclidean(synth3, chords[chord3_sel],
                                           rhythm3, r3, last_r3, c3, 6)
        clock = ticks_add(clock, delay)

    # in PLAY select button controls play/pause
    if mode == "PLAY":
        for i in range(4):
            if not select_buttons[i].value and select_states[i] is False:
                select_states[i] = True
            if play_states[i] is True:
                synth.release(synths[i])
                play_states[i] = False
            else:
                play_states[i] = True
            if select_buttons[i].value and select_states[i] is True:
                select_states[i] = False
        display0.print(chord_names[chord0_sel])
        display1.print(chord_names[chord1_sel])
        display2.print(chord_names[chord2_sel])
        display3.print(chord_names[chord3_sel])

    # EUC menu select resets cycle count
    elif mode == "EUC ":
        if not menu_buttons[0].value and menu_states[0] is False:
            r0 = 0
            r1 = 0
            r2 = 0
            r3 = 0
            menu_states[0] = True
        if menu_buttons[0].value and menu_states[0] is True:
            menu_states[0] = False
rhythm0, euclid0_steps, euclid0_pulses, r0 = read_buttons(buttons0, button0_states,
rhythm0, euclid0_steps, euclid0_pulses, r0)
rhythm1, euclid1_steps, euclid1_pulses, r1 = read_buttons(buttons1, button1_states,
rhythm1, euclid1_steps, euclid1_pulses, r1)
rhythm2, euclid2_steps, euclid2_pulses, r2 = read_buttons(buttons2, button2_states,
rhythm2, euclid2_steps, euclid2_pulses, r2)
rhythm3, euclid3_steps, euclid3_pulses, r3 = read_buttons(buttons3, button3_states,
rhythm3, euclid3_steps, euclid3_pulses, r3)

display0.print(f" {euclid0_pulses}"
display1.print(f" {euclid1_pulses}"
display2.print(f" {euclid2_pulses}"
display3.print(f" {euclid3_pulses}"

# BPM is adjusted
eelif mode == "BPM ":
    if not select_buttons[0].value and select_states[0] is False:
        bpm = 120
        select_states[0] = True
    if select_buttons[0].value and select_states[0] is True:
        select_states[0] = False
    display0.print(f" {bpm}"
    display1.print(" ")
    display2.print(" ")
    display3.print(" ")

# beat division is changed
eelif mode == "BEAT":
    if not select_buttons[0].value and select_states[0] is False:
        beat_names[beat_index] = 2
        select_states[0] = True
    if select_buttons[0].value and select_states[0] is True:
        select_states[0] = False
    display0.print(f" {beat_names[beat_index]}"
    display1.print(" ")
    display2.print(" ")
    display3.print(" ")

# adsr for each voice
eelif mode == "ADSR":
    for i in range(4):
        if not left_buttons[i].value and left_states[i] is False:
            synth_adsr_indexes[i] = (synth_adsr_indexes[i] - 1) % 4
            left_states[i] = True
            the_synth = synths[i]
        if left_buttons[i].value and left_states[i] is True:
            left_states[i] = False
        if not right_buttons[i].value and right_states[i] is False:
            synth_adsr_indexes[i] = (synth_adsr_indexes[i] + 1) % 4
            right_states[i] = True
        if right_buttons[i].value and right_states[i] is True:
            right_states[i] = False
        if not select_buttons[i].value and select_states[i] is False:
            the_synth = synths[i]
            all_adsr_values[i][0] = 0.1
            all_adsr_values[i][1] = 0.1
            all_adsr_values[i][3] = 0.1

©Adafruit Industries
all_adsr_values[i][2] = 0.05
the_env = synthio.Envelope(attack_time=all_adsr_values[i][0],
decay_time = all_adsr_values[i][1],
release_time=all_adsr_values[i][3],
attack_level=1,
sustain_level=all_adsr_values[i][2])
the_synth.envelope = the_env
if select_buttons[i].value and select_states[i] is True:
    select_states[i] = False
# pylint: disable=line-too-long
display0.print(f"{adsr_names[synth_adsr_indexes[0]]}
{synth0.envelope[adsr_properties[synth_adsr_indexes[0]]]:.2f}"
)
display1.print(f"{adsr_names[synth_adsr_indexes[1]]}
{synth1.envelope[adsr_properties[synth_adsr_indexes[1]]]:.2f}"
)
display2.print(f"{adsr_names[synth_adsr_indexes[2]]}
{synth2.envelope[adsr_properties[synth_adsr_indexes[2]]]:.2f}"
)
display3.print(f"{adsr_names[synth_adsr_indexes[3]]}
{synth3.envelope[adsr_properties[synth_adsr_indexes[3]]]:.2f}""
#endif change waveform
eif mode == "WAVE":
display0.print(f"{wave_names[synth0_wave]})")
display1.print(f"{wave_names[synth1_wave])")
display2.print(f"{wave_names[synth2_wave])")
display3.print(f"{wave_names[synth3_wave])")
#endif adjust ring modulation
eif mode == "RING":
display0.print(f"{synth0.ring_frequency:.1f}"
)
display1.print(f"{synth1.ring_frequency:.1f}"
)
display2.print(f"{synth2.ring_frequency:.1f}"
)
display3.print(f"{synth3.ring_frequency:.1f}"
)
#endif adjust lfo rate used for ring modulation
eif mode == "LFO ":
display0.print("RATE")
display1.print(f"{lfo.rate:.1f}""
)
display2.print(""
)
display3.print(""
)
#endif overall volume 0.0 - 1.0
eif mode == "VOL ":
display0.print(f"{mixer.voice[0].level:.1f}"
)
display1.print(""
)
display2.print(""
)
display3.print(""
)

Upload the Code and Libraries to the Feather RP2040

After downloading the Project Bundle, plug your Feather RP2040 into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called CIRCUITPY. Unzip the folder and copy the following items to the Feather RP2040's CIRCUITPY drive.

- lib folder
- code.py

Your Feather RP2040 CIRCUITPY drive should look like this after copying the lib folder and the code.py file:
How the CircuitPython Code Works

The code begins by creating some waveform and ADSR envelope objects. These objects are passed to `synthio.Note` objects. There are four `Note` objects and they will create four different voices.

The `Note` s are played by the `Synthesizer` object, which outputs through the `Mixer` object.

```python
SAMPLE_RATE = 44100
SAMPLE_SIZE = 256
VOLUME = 5000

# waveforms, envelopes and synth setup
square = np.concatenate((np.ones(SAMPLE_SIZE//2, dtype=np.int16)*VOLUME, np.ones(SAMPLE_SIZE//2, dtype=np.int16)*-VOLUME))
sine = np.array(np.sin(np.linspace(0, 4*np.pi, SAMPLE_SIZE, endpoint=False)) * VOLUME, dtype=np.int16)
saw = np.linspace(VOLUME, -VOLUME, num=SAMPLE_SIZE, dtype=np.int16)
noise = np.array([randint(-VOLUME, VOLUME) for i in range(SAMPLE_SIZE)], dtype=np.int16)
lfo = synthio.LFO(rate = .5, waveform = sine)

amp_env0 = synthio.Envelope(attack_time=0.1, decay_time = 0.1, release_time=0.1, attack_level=1, sustain_level=0.05)
amp_env1 = synthio.Envelope(attack_time=0.05, decay_time = 0.1, release_time=0.1, attack_level=1, sustain_level=0.05)

# synth plays the notes
synth = synthio.Synthesizer(sample_rate=SAMPLE_RATE)

# these are the notes
synth0 = synthio.Note(frequency = 0.0, envelope=amp_env0, waveform=square, ring_frequency = 0, ring_bend = lfo, ring_waveform = sine)
synth1 = synthio.Note(frequency = 0.0, envelope=amp_env1, waveform=sine, ring_frequency = 0, ring_bend = lfo, ring_waveform = sine)
synth2 = synthio.Note(frequency = 0.0, envelope=amp_env0, waveform=square, ring_frequency = 0, ring_bend = lfo, ring_waveform = sine)
```

©Adafruit Industries
synth3 = synthio.Note(frequency = 0.0, envelope=amp_env1, waveform=sine, ring_frequency = 0, ring_bend = lfo, ring_waveform = sine)
synths = [synth0, synth1, synth2, synth3]wave_names = ["SQUR", "SINE", "SAW ", "NOIZ"]waveforms = [square, sine, saw, noise]synth0_wave = 0synth1_wave = 1synth2_wave = 0synth3_wave = 1
# i2s amp setupaudio = audiobusio.I2SOut(bit_clock=board.D10, word_select=board.D11, data=board.D9)mixer = audiomixer.Mixer(voice_count=4, sample_rate=SAMPLE_RATE, channel_count=1, bits_per_sample=16, samples_signed=True, buffer_size=2048 )audio.play(mixer)vol_val = 2mixer.voice[0].play(synth)mixer.voice[0].level = 0.3

Tones

Arrays of tones are created for triads. They are all I (tonic) chords in the circle of fifths.

# these are the triads, all majorc_tones = [130.81, 164.81, 196.00]g_tones = [196.00, 246.94, 293.66]d_tones = [146.83, 185.00, 220.00]a_tones = [220.00, 277.18, 329.63]e_tones = [164.81, 207.65, 246.94]b_tones = [246.94, 311.13, 369.99]fsharp_tones = [185.00, 233.08, 277.18]csharp_tones = [138.59, 174.61, 207.65]aflat_tones = [207.65, 261.63, 311.13]eflat_tones = [233.08, 293.66, 349.23]f_tones = [174.61, 220.00, 261.63]

# names for the alphanumeric displayschord_names = ["Cmaj", "Gmaj", "Dmaj", "Amaj", "Emaj", "Bmaj", "F#ma", "C#ma", "Abma", "Ebmaj", "Bbmaj", "Fmaj"]chords = [c_tones, g_tones, d_tones, a_tones, e_tones, b_tones, fsharp_tones, csharp_tones, aflat_tones, eflat_tones, bflat_tones, f_tones]

I2C

Next are the I2C peripherals. There are five ANO rotary encoders, five alphanumeric displays and one 8x8 matrix.

# i2c setupi2c = board.I2C()# the encodersseesaw0 = seesaw.Seesaw(i2c, addr=0x49)seesaw1 = seesaw.Seesaw(i2c, addr=0x4A)seesaw2 = seesaw.Seesaw(i2c, addr=0x4B)seesaw3 = seesaw.Seesaw(i2c, addr=0x4C)menu_seesaw = seesaw.Seesaw(i2c, addr=0x4D)
# the alphanumeric displays

display0 = segments.Seg14x4(i2c, address=0x70)
display1 = segments.Seg14x4(i2c, address=0x71)
display2 = segments.Seg14x4(i2c, address=0x72)
display3 = segments.Seg14x4(i2c, address=0x73)
menu_display = segments.Seg14x4(i2c, address=0x74)

# the matrix
matrix0 = Matrix8x8x2(i2c, address=0x75)

seesaws = [seesaw0, seesaw1, seesaw2, seesaw3, menu_seesaw]

buttons0 = []
buts = [i, 5]
buts[3] = False

# make all of the encoders
encoder0 = rotaryio.IncrementalEncoder(seesaw0)
last_position0 = 0
encoder1 = rotaryio.IncrementalEncoder(seesaw1)
last_position1 = 0
encoder2 = rotaryio.IncrementalEncoder(seesaw2)
last_position2 = 0
encoder3 = rotaryio.IncrementalEncoder(seesaw3)
last_position3 = 0

menu_enc = rotaryio.IncrementalEncoder(menu_seesaw)
last_menuPosition = 0

Conjunction Function

There are two functions that are used in the loop. The first reads all of the selector encoder buttons to adjust the Euclidean rhythm parameters.

```python
def read_buttons(button_array, button_states, euc, e_step, e_pulse, the_step):
    for b in range(5):
        if not button_array[b].value and button_states[b] is False:
            button_states[b] = True
        if button0_names[b] == "Select":
            e_step = 8
            e_pulse = 4
```

© Adafruit Industries
if the_step >= e_step:
    the_step = 0
elif button0_names[b] == "Up":
    if e_step &gt; 16:
        e_step = 16
    else:
        e_step += 1
elif button0_names[b] == "Down":
    if e_step &lt; 1:
        e_step = 1
    else:
        e_step -= 1
if the_step >= e_step:
    the_step = 0
elif button0_names[b] == "Left":
    e_pulse -= 1
    e_pulse = max(e_pulse, 1)
else:
    e_pulse += 1
    e_pulse = min(e_pulse, e_step)
euc = bjorklund(e_step, e_pulse)
if button_array[b].value and button_states[b] is True:
    button_states[b] = False
if button0_names[b] in ("Select", "Up", "Down"):
    matrix0.fill(matrix0.LED_OFF)
    draw_steps(euclid0_steps, 0)
    draw_steps(euclid1_steps, 2)
    draw_steps(euclid2_steps, 4)
    draw_steps(euclid3_steps, 6)
return euc, e_step, e_pulse, the_step

The second actually plays the Euclidean rhythms with the passed in chord to the
designated synth voice.

def play_euclidean(this_synth, n, the_rhythm, rhythm_count, last_count, c,
matrix_slot):
    if last_count &lt;= 7:
        matrix0[matrix_slot, last_count] = matrix0.LED_GREEN
    else:
        c += 1
        matrix0[matrix_slot + 1, (last_count - last_count) + c] = matrix0.LED_GREEN
    c += 1
    if the_rhythm[rhythm_count] == 1:
        this_synth.frequency = n[randint(0, 2)]
        synth.press(this_synth)
        if rhythm_count &lt;= 7:
            matrix0[matrix_slot, rhythm_count] = matrix0.LED_RED
        else:
            matrix0[matrix_slot + 1, (rhythm_count - rhythm_count) + c] =
            matrix0.LED_RED
            c += 1
    else:
        synth.release(this_synth)
        if rhythm_count &gt; 7:
            c += 1
        last_count = rhythm_count
    rhythm_count += 1
    if rhythm_count &gt;= len(the_rhythm):
        rhythm_count = 0
    if rhythm_count == 1:
        c = 0
    return rhythm_count, last_count, c
ADSR Prep

A few arrays and variables are prepared in order to affect the ADSR envelope for each synth voice. When a change is made to an envelope, a new envelope must be instantiated. These arrays allow you to store the previous ADSR values to pass to this new envelope. As a result, if you change the attack value, the previous decay, sustain and release values are retained and passed to the new envelope.

```python
# adsr mode
adsr_names = ["A", "D", "S", "R"]
synth_adsr_indexes = [0, 0, 0, 0]
adsr_properties = [0, 1, 4, 2]
adsr0_values = [amp_env0.attack_time, amp_env0.decay_time,
                amp_env0.sustain_level, amp_env0.release_time]
adsr1_values = [amp_env1.attack_time, amp_env1.decay_time,
                amp_env1.sustain_level, amp_env1.release_time]
adsr2_values = [amp_env0.attack_time, amp_env0.decay_time,
                amp_env0.sustain_level, amp_env0.release_time]
adsr3_values = [amp_env1.attack_time, amp_env1.decay_time,
                amp_env1.sustain_level, amp_env1.release_time]

all_adsr_values = [adsr0_values, adsr1_values, adsr2_values, adsr3_values]
adsr0_val = int(simpleio.map_range(amp_env0.attack_time, 0.0, 1.0, 0, 19))
adsr1_val = int(simpleio.map_range(amp_env0.decay_time, 0.0, 1.0, 0, 19))
adsr2_val = int(simpleio.map_range(amp_env0.sustain_level, 0.0, 1.0, 0, 19))
adsr3_val = int(simpleio.map_range(amp_env0.release_time, 0.0, 1.0, 0, 19))
```

The Loop

The loop has four tasks happening: reading the rotary encoders, playing the Euclidean rhythms, updating the alphanumeric displays, and reading the encoder buttons.

The rotary encoders are read with a small delay. The menu encoder controls which mode is active. Each of the four other encoders' functionality changes depending on the mode.

```python
if ticks_diff(ticks_ms(), enc_clock) &gt;= 100:
    position0 = encoder0.position
    position1 = encoder1.position
    position2 = encoder2.position
    position3 = encoder3.position
    menuPosition = menu_enc.position
    # menu changes mode
    if menuPosition != last_menuPosition:
        if menuPosition &gt; last_menuPosition:
            mode_index = (mode_index + 1) % len(modes)
```
else:
    mode_index = (mode_index - 1) % len(modes)
if mode in ("EUC", "ADSR"):
    clock_stretch = True
if mode in ("PLAY", "BPM", "BEAT", "WAVE") and clock_stretch:
    clock = ticks_ms()
    clock_stretch = False
mode = modes[mode_index]
menu_display.print(f"   {mode}"
last_menuPosition = menuPosition
...

if position1 != last_position1:
    if position1 > last_position1:
        if mode == "PLAY":
            chord1_sel = (chord1_sel + 1) % len(chords)
            display1.print(chord_names[chord1_sel])
        elif mode == "ADSR":
            adsr1_val = (adsr1_val + 1) % 20
            mapped_val = simpleio.map_range(adsr1_val, 0, 19, 0.0, 1.0)
            all_adsr_values[1][synth_adsr_indexes[1]] = mapped_val
            the_env = synthio.Envelope(attack_time=all_adsr_values[1][0],
                                      decay_time = all_adsr_values[1][1],
                                      release_time=all_adsr_values[1][3],
                                      attack_level=1,
                                      sustain_level=all_adsr_values[1][2])
            synth1.envelope = the_env
        elif mode == "WAVE":
            synth1_wave = (synth1_wave + 1) % len(wave_names)
            synth1.waveform = waveforms[synth1_wave]
    ...

No matter which mode, the synth plays on; using ticks to keep time. **play_states[]** keeps track of whether or not a synth voice is paused.

```python
# synth plays based on ticks timing
if ticks_diff(ticks_ms(), clock) &gt;= delay:
    if play_states[0] is True:
        r0, last_r0, c0 = play_euclidean(synth0, chords[chord0_sel],
                                          rhythm0, r0, last_r0, c0, 0)
    if play_states[1] is True:
        r1, last_r1, c1 = play_euclidean(synth1, chords[chord1_sel],
                                          rhythm1, r1, last_r1, c1, 2)
    if play_states[2] is True:
        r2, last_r2, c2 = play_euclidean(synth2, chords[chord2_sel],
                                          rhythm2, r2, last_r2, c2, 4)
    if play_states[3] is True:
        r3, last_r3, c3 = play_euclidean(synth3, chords[chord3_sel],
                                          rhythm3, r3, last_r3, c3, 6)
    clock = ticks_add(clock, delay)
```

Just like the rotary encoders, the buttons and alphanumeric displays have different functionality depending on the mode. The alphanumeric displays will update to show different values. For example, in Wave mode, the waveform for each voice is displayed. As the waveform is changed with the rotary encoder, the display updates.

```python
# in PLAY select button controls play/pause
if mode == "PLAY":
    for i in range(4):
        if not select_buttons[i].value and select_states[i] is False:
            select_states[i] = True
```
if play_states[i] is True:
    synth.release(synths[i])
    play_states[i] = False
else:
    play_states[i] = True
if select_buttons[i].value and select_states[i] is True:
    select_states[i] = False
display0.print(chord_names[chord0_sel])
display1.print(chord_names[chord1_sel])
display2.print(chord_names[chord2_sel])
display3.print(chord_names[chord3_sel])

# change waveform
elif mode == "WAVE":
    display0.print(f"   {wave_names[synth0_wave]}")
    display1.print(f"   {wave_names[synth1_wave]}")
    display2.print(f"   {wave_names[synth2_wave]}")
    display3.print(f"   {wave_names[synth3_wave]}")

Setting I2C Addresses

This project uses multiple I2C breakouts. As a result, their addresses need to be adjusted by either soldering or cutting the address jumpers.

There are five ANO rotary encoder breakouts. These will be on addresses 0x49-0x4D.

0x49 - default address, leave jumpers as-is
0x4A - cut jumper A0
0x4B - cut jumper A1
0x4C - cut jumpers A0 and A1
0x4D - cut jumper A2

There are five alphanumeric display breakouts. These will be on addresses 0x70-0x74.

0x70 - default address, leave jumpers as-is
0x71 - solder jumper A0
0x72 - solder jumper A1
0x73 - solder jumpers A0 and A1
0x74 - solder jumper A2
The bicolor LED matrix is has the same default I2C address (0x70) as the alphanumeric displays. It will be set to 0x75 by soldering jumpers A0 and A2.

Wiring

Cut, splice, and tin five pieces of different colored wire (yellow, white, blue, red and black) that are approximately 3 inches in length. Cut, splice, and tin one piece of wire that is approximately 1 inch in length.
Insert the 1 inch piece of wire into the GND pad on the I2S amp from the front. Then, insert the black piece of wire into the same GND pad from the back. Solder both pieces in place.

Solder the other end of the 1 inch piece of wire into the GAIN pad on the I2S amp. This sets the gain to 12dB since it is tied to GND.

Solder the remaining four pieces of wire into the LRC, BCLK, DIN and VIN pads on the I2S amp.
Solder the pieces of wire from the I2S amp to the Feather RP2040.

- LRC to pin 11
- BCLK to pin 10
- DIN to pin 9
- GND to GND
- VIN to USB

Solder the terminal block to the speaker output on the I2S amp.

STEMMA Wiring
Connect the ANO rotary encoders on addresses 0x49-0x4C left to right with STEMMA QT cables.

Plug the alphanumeric display on address 0x70 into the rotary encoder on 0x49 (farthest to the left).

Plug in the alphanumeric displays on addresses 0x71-0x73 left to right into the alphanumeric display on address 0x70.
Plug in the alphanumeric display on address 0x74 into the alphanumeric display on address 0x73.

Plug in the rotary encoder on address 0x4D into the alphanumeric display on address 0x74. Then, plug in the bicolor LED matrix into the remaining port on the 0x4D rotary encoder.

That completes the STEMMA wiring.
Assembly

STEMMA Component Mounting

Attach the four rotary encoders with M2.5 screws and nuts to the cutouts at the bottom of the lid.
Attach the four alphanumeric displays with M2 screws and nuts to the center cutouts in the lid.

Attach the menu rotary encoder to the top encoder cutout on the lid with M2.5 screws and nuts. Attach the 8x8 matrix and menu alphanumeric display to the top cutouts on the lids with M2 screws and nuts.

Mount the Feather and Amp

Secure the Feather and I2S amp to the standoffs in the case with M2.5 screws.
Mount and Connect the Speaker

Cut and splice the speaker wires to prepare them for the I2S amp terminal block.

Mount the speaker with M2.5 screws and nuts into the mounting hole in the back of the case.

Secure the black speaker wire into the negative (-) terminal in the terminal block on the I2S amp. Secure the red speaker wire into the positive (+) terminal block on the I2S amp.
USB-C Extension

Insert and mount the USB-C extension into the mounting hole on the side of the case. Plug the cable into the USB-C port on the Feather.

Connect to the Feather

Plug a STEMMA QT cable into the STEMMA port on the Feather.

Plug the other end of the Feather STEMMA QT cable into the open port on the lower right rotary encoder.
In Closing

Attach the lid to the case with M3 screws in the corner mounting holes.

Making Beautiful Music
You can turn the encoder for each of the synth voices to change the chord being played. The chords are programmed in the order of the circle of fifths. If you advance one by one, it will sound very natural since you will be modulating between the keys.

You can change the BPM (the speed) and beat division in the menus. The beat can be divided into sixteenth notes, eighth notes, quarter notes, half notes or whole notes.

There are four waveforms to choose from: square, sine, sawtooth or noise. Each synth voice can have its waveform changed.