Musical Fountain using Solenoid Valves

Vibhore Jain, DESE, IISc

ABSTRACT

This aim of this mini project is to realise a musical fountain using solenoid valves. The fountain should react and actuate according to the music track that is played. Interplay of audio and visuals is an enjoyable experience as these two senses work almost in an orthogonal fashion in day to day life. Hence, experiencing correlated input signals to audio and visual sensors is joyful.

I INTRODUCTION

Music in the most abstract definition is a time series of sound waves with various amplitudes and frequencies. The distinguishing feature between music and Gaussian noise is presence of patterns in frequency and amplitude. Frequency patterns are called *Raga* in Indian classical music and their western counter parts are called chords. These patterns are usually attributed to harmonic instruments which are further classified into string and wind instruments.

The other class of instruments called percussion are responsible for providing beats to the music. Beats are repetitive sounds that define the tempo of the music. Faster repetition of beats results in higher tempo while slower repetition rate results in lower tempo. Percussion instruments usually produce sounds which are non-harmonic, viz the energy of their sound is not confined to certain frequencies rather they have a continuous distribution across the frequency spectrum. It is to be noted that instruments like *Tabla* are percussive yet capable of producing harmonic sounds.

Musical fountains are visual representation of these patterns. Commonly used topologies consist of vertically firing fountain jets whose amplitude is a function of patterns discussed above. Most commercial fountains also utilise lights to emphasize the fountain jets and add fast dynamics to the system as fountains themselves are mechanically actuated and have time constants in the order of hundreds of milliseconds. This mini-project is a realisation of similar topology of musical fountains with solenoid valves for controlling fountain jets along-side high power RGB LED emphasis lights as shown in Figure 1. It should be noted that such topology presents best visualisation of audio in low light/dark environment.

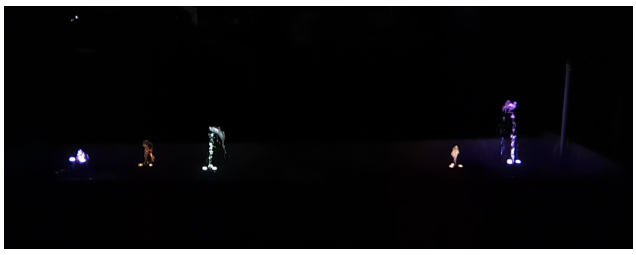

Figure 1: Musical fountain with emphasis lights

II TASKS

1. Fountain topology
2. Pneumatic design
3. Solenoid specifications
4. Solenoid drive
5. RGB LED drive
6. Driver board design
7. Embedded firmware design
8. Audio processing scripts
9. Test setup

Task-1 ***Fountain topology***

The number of independently controllable fountains used dictates unique musical patterns that can be represented visually. The implementation in this case used 8 such fountains in a linear arrangement as shown in figure 2. Each fountain was connected to the central water feed pipe with a solenoid valve in between. Since the valves used were pilot driven DC solenoid valves, the amplitude of the fountain could be either 100% or 0%. The central feeder pipe was supplied water with a single phase 230V AC water pump.

It is possible that at some point in time all the fountain solenoid valves are closed. In such condition, for the safety of pump, a fail-safe valve is put at the end of the feeder pipe which opens when all the fountain valves are closed. The whole fountain setup was placed in a reservoir which acted as source and sink of water. The end fountains visualised beats of the music track where as six other fountains visualised the notes or harmony present in the music. Each fountain is also equipped with an RGB ring light responsible for illuminating the fountain jets and adding dynamic colours to the jet.

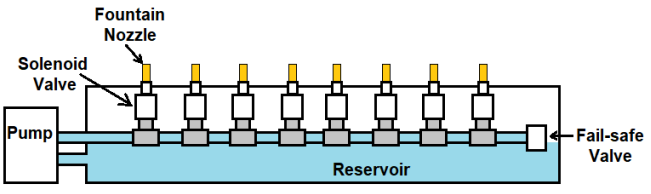

Figure 2: Physical arrangement of fountains

Task-2 ***Pneumatic design***

The water pump used to feed the central pipe is rated for a maximum water head of 30m which translates to roughly 3 bars of static pressure. The DC solenoid valves used supports pressure range of 0.2 bar to 8 bar for margin as higher pressures can be created by dynamic switching of the fountains. The choice of reducer nozzles (shown in Figure 3) for fountain and valve is to facilitate valves of 4 fountains to be open simultaneously while maintaining pressure in the feeder pipe. The feeder pipe connects to the solenoid valve using T-joints with 1/2 inch brass adaptors as shown in Figure 4. The dimensions and specifications of various parts of the pneumatic system are summarised the table 1

Parameter	Value	Unit
Motor Head	30	m
Feeder Pipe Diameter	25.4	mm
Valve Diameter	12.7	mm
Fountain Diameter	6.35	mm

Table 1: Parameters of pneumatic system

Figure 3: Reducer nozzle for fountain jet

Figure 4: T-joint to attach valves with feeder pipe

Task-3 ***Solenoid specifications***

Figure 5 shows the DC valve used for controlling the flow of water in the fountain nozzles. The actuating solenoid is rated for 12V DC and draws 500 mA of current while energised. For driving 8 solenoids independently, 8 drive circuits would be required. Also, the fail-safe valve would require its own independent drive. The current required to drive all 8 valves would thus be 4 amperes. This is dictates the power supply specifications for the electronic control board. DC valves were chosen to eliminate risk of AC shock and ease the driver design requirements.

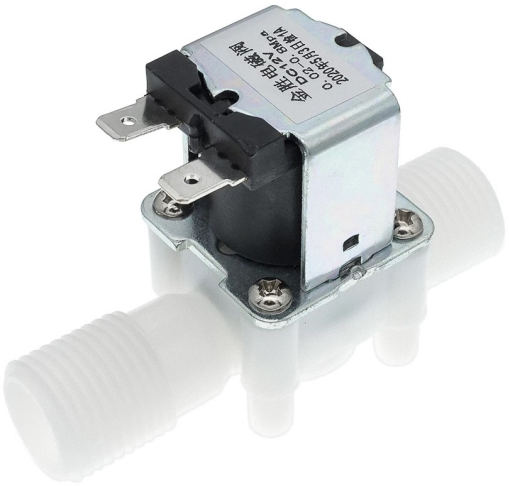

Figure 5: DC solenoid valve for controlling fountains

Task-4 ***Solenoid drive circuit***

Figure 11 shows the drive circuit for driving DC solenoid valves. It utilises two ULN2803 8 channel open drain darlington transistor array ICs for controlling up to 15 solenoid valves and one fail safe valve. The input to these drivers are 8 bit shift registers HC595 as utilising 16 GPIO for controlling ON-OFF behaviour of valves is not optimum. Instead the shift registers receive 16 bit data serially from micro-controller and drive the solenoid valves ON and OFF.

The last channel of the ULN2803 is used for driving the fail safe solenoid valve. The input to this channel is *wired AND* logic of other 15 channels as shown in Figure 6. The inputs to the circuit are SO1, SO2...SO15 and output is SIFS net which drives the driver channel for fail-safe solenoid. The circuit shown in Figure 6 was however changed to use only 8 outputs for the wired AND logic as only 8 solenoid valves were present in the system.

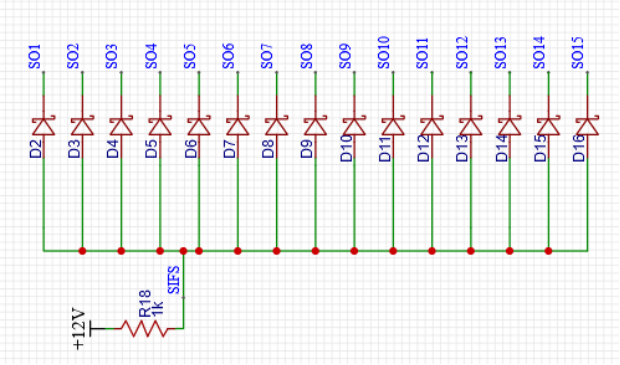

Figure 6: Wired AND logic implemented using diodes

Task-5 ***RGB LED and drive***

RGB LEDs can produce a wide gamut of colours by utilising PWM brightness control for its Red, Green and Blue channels. In this implementation, 3 Watt RGB LEDs were connected in series and powered 12V rails. The LEDs were mounted on a circular PCB with hole in the middle for fitting the fountain nozzle (refer Figure 7). Each LED was also fitted with a collimating lens to focus the light into a narrow beam (beam angle 30°). Each colour in the RGB LED can take upto 100mA of current and hence all LEDs if turned ON together would demand 300mA from the power supply. This current consumption for 8 channels when added to the solenoid drive current totals the current requirement on 12V rail to be 6.4 A.

Figure 7: Fountain module fitted with LEDs PCB

For controlling 8 RGB LED sets, 24 PWM channels would be required. To provide such large number of channels, external PWM IC LP5036RJVR was used. It is an open drain LED PWM driver and thus pull ups were added on each PWM channel to facilitate current amplification as individual channels could only handle 25mA of current. It is capable of driving upto 36 PWM channels and is controlled over I2C bus. Since the RGB LEDs required up to 300mA of current per fountain module, current amplification was required. This was achieved by using ULN2803 drive circuit. The RGB LED drive circuit is shown in Figure 12.

The IC is available only in QFN package and hand assembly is challenging due to small pitch between the pads. Also, since the outputs are buffered by ULN2803, the logic of PWM inverts in the sense that 100% PWM signal corresponds to 0 brightness of the LEDs. This inversion thus needs to be taken care in software so that colour mixing is appropriate.

Task-6 ***Driver board design***

The driver board requires a micro-controller to control the solenoid valves and RGB LEDs. For this purpose, the ubiquitous 8 bit ATmega328P micro-controller was selected. The controller runs on 5V which is also suitable for other ICs on board. The board is powered by ATX PC supply which is capable of delivering 20 A on its 12V supply rails which is crucial as the 12V supply would experience large swings in current loads. The board also features current sensor on the 12V rail that can be used to monitor current draw from the power supply. The driver board receives command from the PC over UART interface and controls the solenoid valves and RGB LEDs according to the firmware present on the ATmega328p. The driver board was designed in easyEDA, an online browser based free to use EDA tool. The final 3D render is shown in Figure 14. Some features of the driver board include:

- Active monitoring provisions for voltage supply line and 12V rail current.
- Daisy chain support for upto 4 boards and control with single micro-controller.
- Extra PWM channels to support servo solenoid valve drive (Future extension).

Task-7 ***Embedded firmware design***

The ATmega328P micro-controller takes a fixed input data frame over UART and sends relevant bytes to the RGB LED driver IC over I2C bus and solenoid drive shift registers over SPI interface. The firmware also periodically diminishes the intensity of the RGB lights of each fountain based on a timer interrupt. When the RGB colour value reaches less than a minimum threshold of 5 for all channels, the corresponding solenoid valve is de-energized. This way the PC only needs to send RGB colour values over UART and the controller takes care of actuating relevant solenoids. While the hardware is capable of driving upto 15 fountains and 12 RGB channels, the current firmware only handles only 8 channels. The complete embedded C code is documented in section VII.

Task-8 ***Audio processing scripts***

There are two audio processing scripts written in python to augment with the embedded C firmware deployed in the ATmega328P micro-controller present on the fountain controller board namely Analyse.py and Streamer.py. Analyse.py takes an audio wav file as input and does beats and notes detection on the sound track using open source music analysis library *librosa* and *audio owl*. It then generates a text file that contains the fountain actuation data in a csv format. This file contains time stamps and data to be sent to the fountain controller board over UART. The second script called Streamer.py takes

in the generated text file as input and plays the corresponding sound track on the default audio playback device. The script then synchronously sends the RGB information to the fountain control board over UART at 9600 baudrate so that the fountain visuals looks in sync with the audio track. It is to be noted that the analysis part and streaming parts are sequential and not real-time since the note detection algorithm is computationally expensive. The scripts are documented in sub-sections 8.1 and 8.2.

Task-9 ***Test setup***

The test setup consists of a reservoir with dimensions 290cm x 60cm x40cm made of scrap plywood (refer Figure 15). For water proofing, a double layer plastic sheet was spread in the reservoir before filling it with water. For testing purpose, the water pump was operated from a variable transformer to modulate the power as abruptly stopping the water flow stressed the pneumatic system. The final demo test setup is shown in Figure 16. Several sound tracks across genres were processed and visualised on the musical fountain.

III CONCLUSIONS

Following observations were made in due course of developing the musical fountain and testing it with various sound tracks:

- Beat detection was fairly easy to implement as low frequency patterns and percussion onsets are easy to detect.
- Note detection was challenging for audio tracks with vocals as well as harmonic instruments as they had significant spectral overlap.
- Machine learning algorithms were better at note detection for such audio tracks compared to traditional FFT based approaches.
- Binary nature of solenoid valves made them inappropriate for fountain applications where amplitude of audio should ideally map to the height of the fountain jet.
- Above dynamics can be achieved either by using PWM on the pump or by using servo valves instead of solenoid valves.

IV ACKNOWLEDGEMENTS

While the fountain was a big design challenge, it also required lot of execution efforts across different domains which is acknowledged below:

- Mechanical assembly : Arun, Srinivas, Supreeth (Mechanical Workshop)
- PCB Fabrication : Surendra (PCB Lab)
- Setup and Test : Abhishek, Divyanshu, Naresh and Vaidika (Power Electronics Lab)

V PROJECT PHOTOS

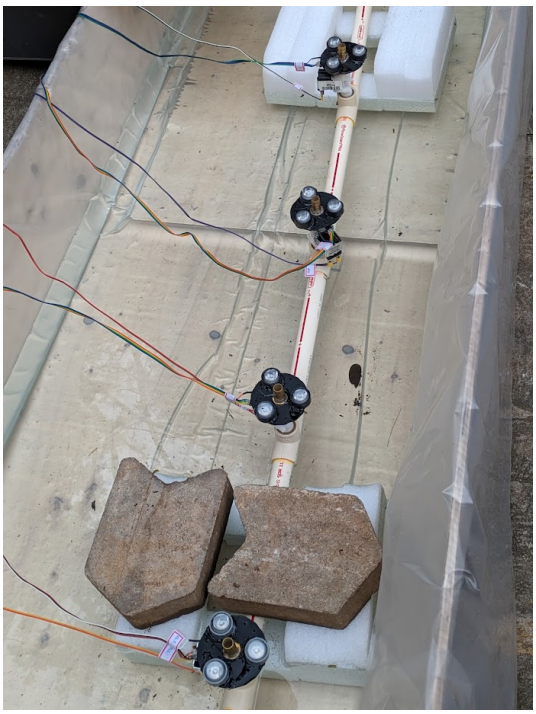

Figure 8: Fountains with LEDs setup in the reservoir

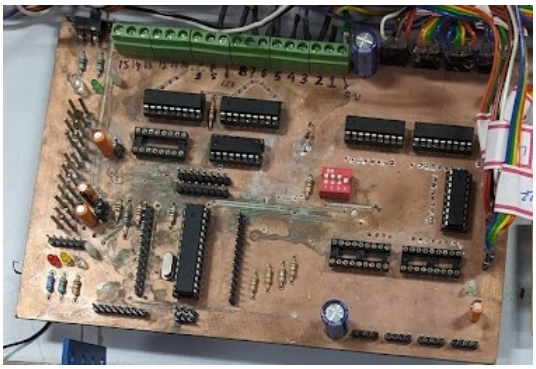

Figure 9: Fabricated and assembled fountain controller board

Figure 10: Musical Fountain show in the dark

VI SCHEMATIC & TEST SETUP

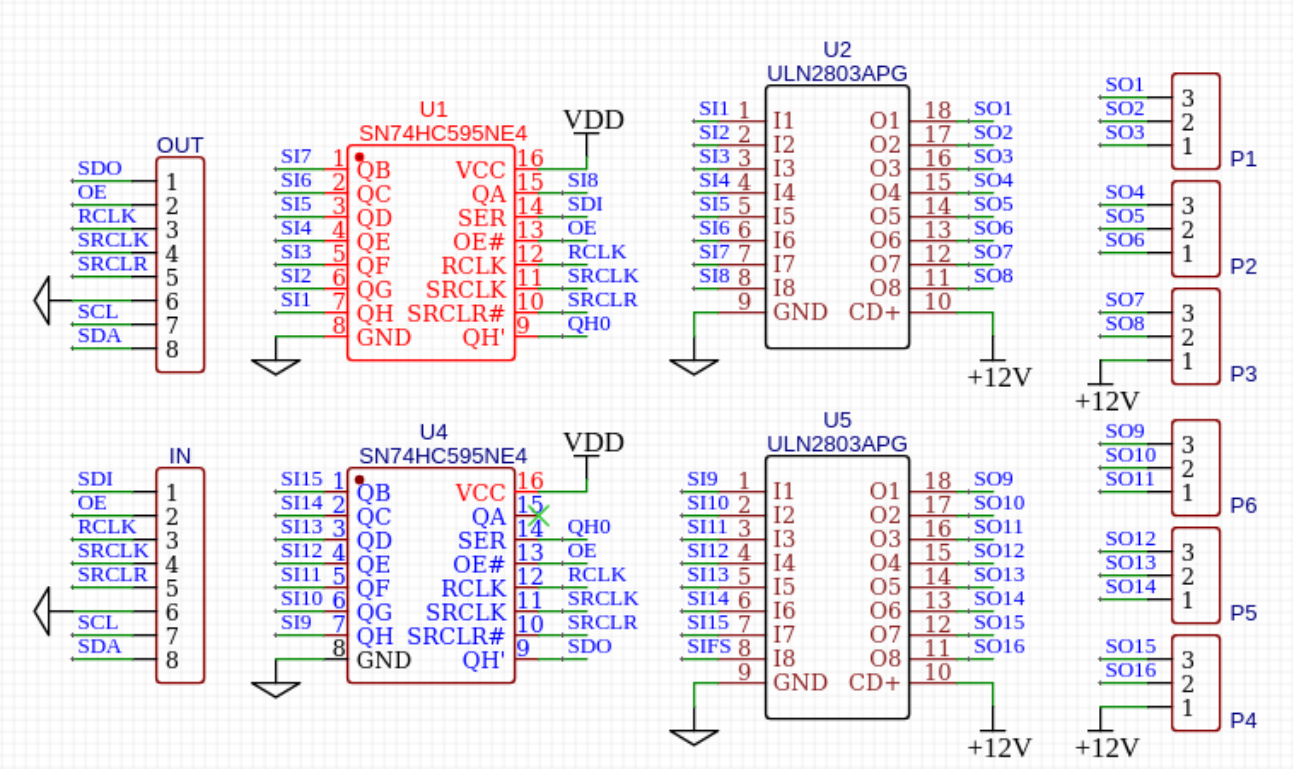

Figure 11: Drive circuit for DC solenoid valves

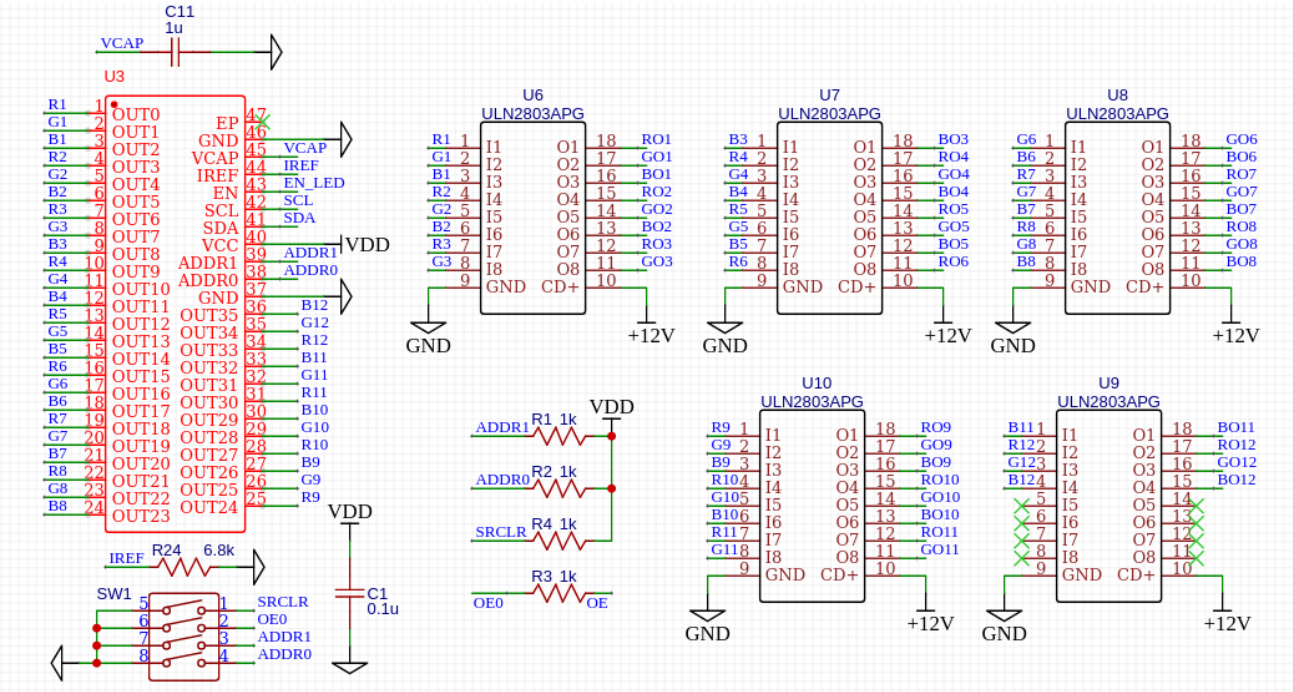

Figure 12: Drive circuit for RGB LEDs

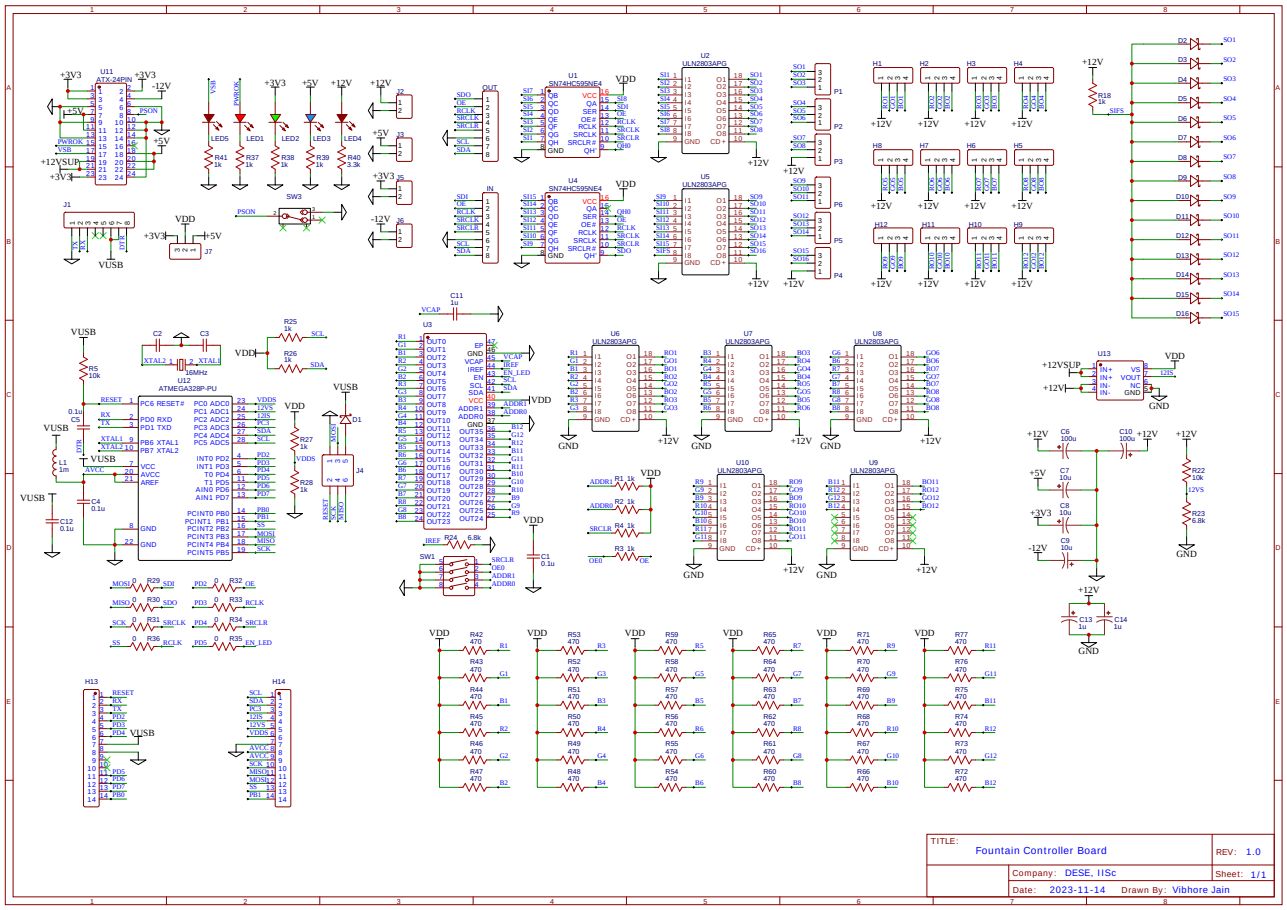

Figure 13: Complete schematic of driver board circuit

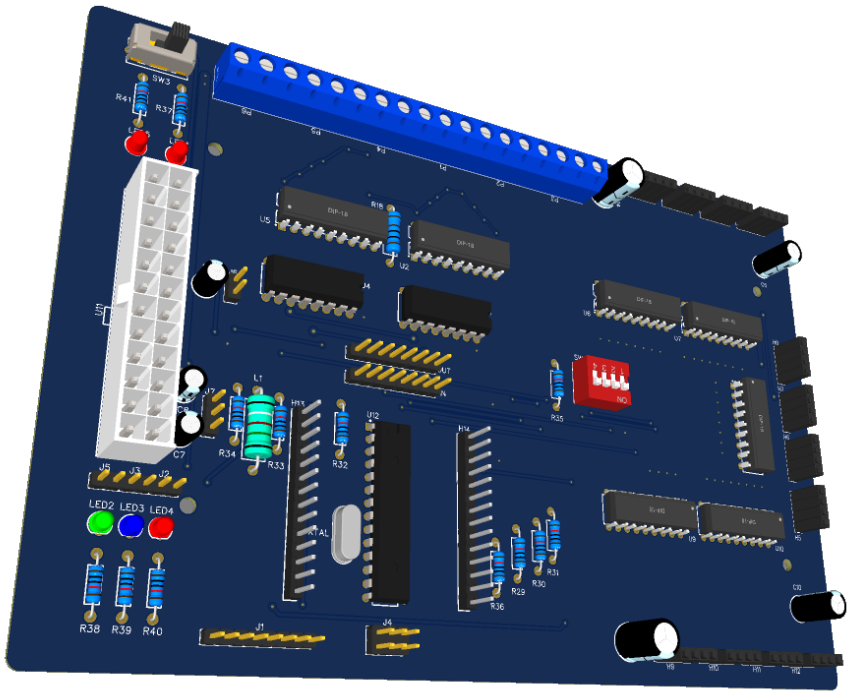

Figure 14: 3D render of the fountain control board

Figure 15: Reservoir made of scrap plywood

Figure 16: Test Setup for Demo

VII EMBEDDED C FIRMWARE

```c
/*
 * Fountain_Controller.c
 * Created: 27-11-2023 17:53:59
 *  Author: jainv
 */
// System clock speed = crystal oscillator frequency
#define F_CPU     16000000L
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <util/delay.h>

//Defines for the RGB PWM controller LP5036
#define brightness_addr      0x08
#define colour_addr        0x14
#define LP5036_addr        0x30
#define LP5036_DEVICE_CONFIG0    0x00
#define LP5036_DEVICE_CONFIG1    0x01
#define LP5036_LED_CONFIG0       0x02
#define LP5036_LED_CONFIG1       0x03
#define LP5036_RESET           0x38
#define TWI_FREQ         400000

//Defines for configuring UART
#define USART_BAUDRATE  9600
#define BAUD_PRESCALER  (((F_CPU/(USART_BAUDRATE * 16UL))) - 1)
#define ASYNCHRONOUS  (0<<UMSEL00)// USART Mode Selection
#define DISABLED    (0<<UPM00)
#define EVEN_PARITY   (2<<UPM00)
#define ODD_PARITY    (3<<UPM00)
#define PARITY_MODE   DISABLED  // USART Parity Bit Selection
#define ONE_BIT     (0<<USBS0)
#define TWO_BIT     (1<<USBS0)
#define STOP_BIT    ONE_BIT     // USART Stop Bit Selection
#define FIVE_BIT    (0<<UCSZ00)
#define SIX_BIT     (1<<UCSZ00)
#define SEVEN_BIT   (2<<UCSZ00)
#define EIGHT_BIT   (3<<UCSZ00)
#define DATA_BIT    EIGHT_BIT // USART Data Bit Selection
#define RX_INT_EN       (1<<RXCIE0)
//Defines for configuring SPI
#define SPI_DDR     DDRB
#define SPI_PORT    PORTB
#define CS        PINB2
#define MOSI      PINB3
#define MISO      PINB4
#define SCK       PINB5
//GPIO to enable/disable PWM LED driver
#define LED_EN      PIND5

volatile uint8_t usart_buff[28]; //Array to store UART Rx Data
volatile uint8_t update_relay;   //Flag to indicate pending update
volatile uint8_t update_leds;  //Flag to indicate pending update
char welcome[] = "Fountain Controller 2.3\n\0"; //Static test string
volatile uint8_t relay_buff[2];  //Buffer to store relay shift bytes
volatile uint8_t rgb_buff[24];   //Buffer to store RGB colour data
volatile uint8_t rgb_brightness[8]; //Brightness of 8 RGB channels
volatile uint8_t decay_fact;     //decay rate of RGB colours
volatile uint8_t idx;      //index for usart buffer
volatile uint8_t update_brightness; //Flag to update brightness

// Relay buff map
// relay_buff[0]  b7  b6  b5  b4  b3  b2  b1  b0
//
// relay_buff[1]  b7  b6  b5  b4  b3  b2  b1  b0
//
//SPI to control solenoid valves
void init_spi()
{
```

```
  SPI_DDR |= (1 << CS) | (1 << MOSI) | (1 << SCK);
  SPCR  |= (1<<SPE|1<<MSTR);  // SPI clock 1MHz
}
// Timer 1 to decay LED brightness
void init_timer1()
{
  TCCR1B |= (1<<CS12)|(1 << WGM12);   // clk = clk_io/256 = 62.5 kHz
  OCR1A  = 1250-1;            // compare match every 20ms
  TIMSK1 |= 1<<OCIE1A;          // Enable compare match interrupt on OC1A
}
// I2C to control PWM driver IC
void init_i2c()
{
  TWSR = 0; // Prescaler = 1
    TWBR = ((F_CPU / TWI_FREQ) - 16) / 2;

    // Enable TWI, generate acknowledge bit, and enable TWI interrupt
    TWCR = (1 << TWEN) | (1 << TWEA);
}
// send I2C start signal
void i2cStart(void)
{
    TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
    while ((TWCR & (1<<TWINT)) == 0);
}

// send I2C stop signal
void i2cStop(void)
{
    TWCR = (1<<TWINT)|(1<<TWSTO)|(1<<TWEN);
  _delay_us(4);
  while (TWCR & (1 << TWSTO));
}
// Multi-byte I2C data write
void i2c_send(uint8_t slave_addr, uint8_t reg_adr, volatile uint8_t *data, uint8_t size)
{
  uint8_t i = 0;
  i2cStart();
  TWDR = slave_addr<<1;
  TWCR = (1<<TWINT)|(1<<TWEN);
  while ((TWCR & (1<<TWINT)) == 0);
  TWDR = reg_adr;
  TWCR = (1<<TWINT)|(1<<TWEN);
  while ((TWCR & (1<<TWINT)) == 0);
  for(i = 0; i <size; i++)
  {
    TWDR = 255-data[i];
    TWCR = (1<<TWINT)|(1<<TWEN);
    while ((TWCR & (1<<TWINT)) == 0);
  }
  i2cStop();
}
// single byte I2C data write
void i2c_send_byte(uint8_t slave_addr, uint8_t reg_adr, volatile uint8_t data)
{
  i2cStart();
  TWDR = slave_addr<<1;
  TWCR = (1<<TWINT)|(1<<TWEN);
  while ((TWCR & (1<<TWINT)) == 0);
  TWDR = reg_adr;
  TWCR = (1<<TWINT)|(1<<TWEN);
  while ((TWCR & (1<<TWINT)) == 0);
  TWDR = data;
  TWCR = (1<<TWINT)|(1<<TWEN);
  while ((TWCR & (1<<TWINT)) == 0);
  i2cStop();
}
// Uart to receive RGB data for fountains from PC
void init_uart()
{
  // Set Baud Rate
```

```
  UBRR0H = BAUD_PRESCALER >> 8;
  UBRR0L = BAUD_PRESCALER;
  // Set Frame Format
  UCSR0C = ASYNCHRONOUS | PARITY_MODE | STOP_BIT | DATA_BIT;
  // Enable Receiver and Transmitter
  UCSR0B = (1<<RXEN0) | (1<<TXEN0);
  UCSR0B |= RX_INT_EN;
}
//Function to copy buffer from UART buff to RGB buff
void cpy_buffs()
{

  uint8_t i = 0;
  //copy new values only if old values are less
  for( i = 0; i < 8; i++)
    if((usart_buff[3*i] > rgb_buff[3*i])||\
       (usart_buff[3*i+1] > rgb_buff[3*i+1])||\
       (usart_buff[3*i+2] > rgb_buff[3*i+2]))
      {
        rgb_buff[3*i] = usart_buff[3*i]-1;
        rgb_buff[3*i+1] = usart_buff[3*i+1]-1;
        rgb_buff[3*i+2] = usart_buff[3*i+2]-1;
      }
  relay_buff[1] = 0;
  //Decide if fountain is to be turned ON or OFF using colour intensity
  for( i = 0; i < 8; i++)
    if((rgb_buff[3*i] > 5)||(rgb_buff[3*i+1]>5)||(rgb_buff[3*i+2] > 5))
      relay_buff[1] |= (1<<i);
  //extract decay factor
  decay_fact = usart_buff[24];
}
// Send string over UART
void uart_str(char *ch)
{
  uint8_t i = 0;
  while(ch[i] != '\0')
  {
    while (( UCSR0A & (1<<UDRE0)) == 0); // Do nothing until UDR is ready
    UDR0 = ch[i];
    i++;
  }
}
// Send multi-byte data over SPI
void spi_send(volatile uint8_t *data, uint8_t size)
{
  uint8_t i = 0;
  while(i < size)
  {
    SPDR = data[i];
    while(!(SPSR&(1<<SPIF)));
    i++;
  }
}
// Initialise PWM LED driver
void init_LED_Drv()
{
  DDRD  |= (1<<LED_EN);
  PORTD |= (1<<LED_EN);
  i2c_send_byte(LP5036_addr,LP5036_DEVICE_CONFIG0,0x40);
  i2c_send_byte(LP5036_addr,LP5036_DEVICE_CONFIG1,0x10 | 0x08 | 0x04);
  i2c_send_byte(LP5036_addr,LP5036_LED_CONFIG0,0x00);
  i2c_send_byte(LP5036_addr,LP5036_LED_CONFIG1,0x00);
}
// Send relay data to shift registers
void relay_send()

{
  SPI_PORT &= (~(1<<CS));
  spi_send(relay_buff, 2);
  SPI_PORT |= (1<<CS);
}
```

```
// Initiliasing global variables
void init_vars()
{
  update_relay  = 0;
  update_leds   = 0;
  idx       = 0;
}
// Routine to check all LED channels for fountains
void test_LEDs()
{
  uint8_t i = 0;
  for(i = 0; i < 8; i++)
  {
    rgb_brightness[i] = 0;  // Maximum brightness
  }
  for(i = 0; i < 24; i++)
  {
    rgb_buff[i] = 0;  // Reset all colours to 0
  }
  i2c_send(LP5036_addr, brightness_addr, rgb_brightness, 8); // Send 8 brightness values
  rgb_buff[0] = 255;
  i2c_send(LP5036_addr, colour_addr, rgb_buff, 24);  // Send 24 colour values
  _delay_ms(500);
  for(i = 0; i < 23; i++)
  {
    rgb_buff[i] = 0;  // Make current colour 0 and next one 255
    rgb_buff[i+1] = 255;
    i2c_send(LP5036_addr, colour_addr, rgb_buff, 24);  // Send 24 colour values
    _delay_ms(500);
  }
  rgb_buff[23] = 0;
  i2c_send(LP5036_addr, colour_addr, rgb_buff, 24);  // Send 24 colour values
  _delay_ms(500);
}

int main(void)
{
  cli(); // Disable interrupts while configuration
  init_vars();
  init_spi();
  init_i2c();
  init_uart();
  init_timer1();
  init_LED_Drv();
  _delay_ms(1000);
  uart_str(welcome); // Send UART message for sanity check

  uint8_t i;
  sei(); //Enable global interrupts
  test_LEDs(); //Test all RGB channels
    while(1)
    {
    if(update_relay == 1) // Check if solenoid states need to be updated
        {
      update_relay = 0;
      relay_send(); // Update solenoid data
    }
    if(update_leds == 1) // Check if LED state needs to be updated
    {

      i2c_send(LP5036_addr, colour_addr, rgb_buff, 24);  // Send 24 colour values
      update_leds = 0;
    }
    if(update_brightness == 1)
    {
      update_brightness = 0;
      i2c_send(LP5036_addr, colour_addr, rgb_buff, 24);  // Send 24 colour values
    }
    }
}
// USART RX ISR to fill UART buffer
```

```
ISR(USART_RX_vect)
{
  uint8_t ch = UDR0;
  if (ch == 0)
  {
    idx = 0;
  }
  else
  {
    usart_buff[idx] = ch;
    idx++;
    if(idx == 25)
    {
      idx = 0;
      cpy_buffs();
      update_relay = 1;
      update_leds = 1;
    }
  }
}
/// Timer1 ISR to decay RGB colours
ISR (TIMER1_COMPA_vect)
{
  if (update_leds == 1)
    return;
  update_brightness = 1;
  for( uint8_t i = 0; i < 24; i++)
    rgb_buff[i] = (rgb_buff[i]*decay_fact)>>8;

  for(uint8_t i = 0; i < 8; i++)
    if((rgb_buff[3*i]+rgb_buff[3*i+1]+rgb_buff[3*i+2]) < 15 )
      {
        relay_buff[1] &= ~(1<<i);
        update_relay = 1;
      }

}
```

VIII PYTHON CODE

8.1 Audio Analysis Script

```python
import numpy as np
import librosa as lb
import matplotlib.pyplot as plt
from librosa.display import specshow
import math
import sys
import soundfile as sf
import audioowl
import csv
import scipy
from collections import Counter

def find_delta(a, H, P):
    """
    Find the delta value to update harmonic and percussive power spectrograms
    @ params: a: float - controls the weight of harmonic and percussive components
              H: 2D numpy array - power spectrum of harmonic component
              P: 2D numpy array - power spectrum of percussive component
    """
    H = np.c_[np.zeros((np.size(H, 0),), dtype=float), H, np.zeros((np.size(H, 0),), dtype=float)]
    P = np.r_[[np.zeros((np.size(P, 1),), dtype=float)], P, [np.zeros((np.size(P, 1),), dtype=float)]]

    delta = a*(H[:, :-2] - 2*(H[:, 1:-1]) + H[:, 2:])/4 - (1 - a)*(P[:-2, :] - 2*(P[1:-1, :]) + P[2:, :])
    /4

    return delta

def separate(filename, y=1, a_h=1, a_p=1, k_max=20):
    """
    Separate an audio file into 2 audio files, one for percussive components (drums)
    and one for harmonic components (singing & others)
    @params: filename: string - input audio file
             y: float (0 < y <= 1) - range compression coefficient, facilitates the separation
             a_h and a_p: float - control the weights of the horizontal and vertical smoothness
             k_max: number of iterations

    @return: 2 audio files "H.wav" and "P.wav" are created, which contain harmonic and
             percussive components, respoectively, and saved into the current directory.
    """
    # load the signal
    audioIn, sr = lb.load(filename, sr=None)

    # perform short-time Fourier transform
    n_fft = 2048
    n_audio = len(audioIn)
    # make sure that the signal length will not be trimmed after stft and istft
    audioIn_pad = lb.util.fix_length(audioIn, size = n_audio + n_fft // 2)
    F = lb.stft(audioIn_pad, n_fft=n_fft)

    # Calculate a range-compressed version of the power spectrogram
    W = np.abs(F)**(2*y)

    # Set initial values for harmonic and percussive power spectrogram
    H = W/2
    P = W/2
    k = 0
    a = (a_p**2)/(a_h**2 + a_p**2)

    while k < k_max - 1:
        delta = find_delta(a, H, P)

        #H = min(max(H + delta, 0), W)
        H = H + delta
        H [ H < 0 ] = 0
        np.copyto(H, W, where = H > W)

        P = W - H
```

```
        k += 1

    # binarize separation output
    H = np.multiply((H >= P).astype('int'), W)
    P = np.multiply((P > H).astype('int'), W)
    # convert separated power spectrums back to waveform signals
    # by inverse short time Fourier transform
    x_h = lb.istft(H**(1/2*y) * math.e**(1j*np.angle(F)), length=n_audio)
    x_p = lb.istft(P**(1/2*y) * math.e**(1j*np.angle(F)), length=n_audio)

    sf.write("output/H_"+filename, x_h, sr)
    sf.write("output/P_"+filename, x_p, sr)

if len(sys.argv) == 1:
    print("Please enter the audio filepath after the .py file")
    exit(0)
else:
    filename = sys.argv[1][2:]

    # Read music .wav file and plot its power spectrogram
    audioOG, srOG = lb.load(filename, sr=None)
    D = lb.amplitude_to_db(np.abs(lb.stft(audioOG)), ref=np.max)
    y1,sr1 = lb.load(lb.ex('trumpet'))

    '''plt.figure()
    specshow(D, y_axis='linear')
    plt.colorbar(format='%+2.0f dB')
    plt.title('Power spectrogram of ' + filename)
    plt.show()'''

    # Separate into harmonics & percussion
    separate(filename)
    audioH, srH = lb.load('output/H_'+filename, sr=None)
    audioP, srP = lb.load('output/P_'+filename, sr=None)

    # Get the separated (harmonics-only & percussions-only) power spectrograms
    DH = lb.amplitude_to_db(np.abs(lb.stft(audioH)), ref=np.max)
    DP = lb.amplitude_to_db(np.abs(lb.stft(audioP)), ref=np.max)
    data_h = audioowl.analyze_file(path=filename, sr=srH)
    data_p = audioowl.analyze_file(path=filename, sr=srP)

    beat_times = np.array(data_p['beat_samples'])/srP
    notes      = data_h['notes']
    notes_f  = notes
    print(len(notes_f)*512/srP)
    print(data_h['duration'])

    seq_name = filename[0:filename.index('.')]+'.txt'
    sequencer = open(seq_name, 'w',newline="")
    csvwriter = csv.writer(sequencer, delimiter=",")

    rel_byte0 = 1
    rel_byte1 = 1
    rel_byte2 = 1

    rel1 = 2
    rel2 = 3
    rel3 = 4
    rel4 = 5
    rel5 = 6
    rel6 = 7

    rel7 = 1
    rel8 = 2

    fft_N = 1024
    bin_span = srH/fft_N
    j = 0
    for i in range(0,len(beat_times)):
        item = beat_times[i]
        colour1,colour2,colour3,colour4 = [1,1,1],[1,1,1],[1,1,1],[1,1,1]
```

```
        colour5,colour6,colour7,colour8 = [1,1,1],[1,1,1],[1,1,1],[1,1,1]

        sequencer_line = [item.round(decimals=2),0]
        rel_byte1 = rel_byte1|(1<<rel1)
        rel_byte2 = rel_byte2 | (1<<rel8)
        beat_sample = int(item*srH)
        stft_spec = np.abs(np.fft.rfft(audioP[beat_sample:beat_sample+fft_N]))
        norm = stft_spec / np.linalg.norm(stft_spec)
        red_1_8 = np.sum(norm[int(10/bin_span):int(50/bin_span)])
        green_1_8 = np.sum(norm[int(50/bin_span):int(200/bin_span)])
        blue_1_8 = np.sum(norm[int(200/bin_span):int(1000/bin_span)])

        red_1_8 = int(min(max(2000*red_1_8,1),255))
        green_1_8 = int(min(max(150*green_1_8,1),255))
        blue_1_8 = int(min(max(50*blue_1_8,1),255))

        max_col = max(max(red_1_8,green_1_8),blue_1_8)
        red_1_8 = int(255*red_1_8/max_col)
        green_1_8 = int(255*green_1_8/max_col)
        blue_1_8 = int(255*blue_1_8/max_col)

        colour1   = [red_1_8,green_1_8,blue_1_8]
        colour8   = [red_1_8,green_1_8,blue_1_8]
        note_time_stamp = j*512/srH


        #,rel_byte0,rel_byte1,rel_byte2]
        #sequencer_line = sequencer_line + [rel_byte0,rel_byte1,rel_byte2]
        sequencer_line = sequencer_line + colour8+colour7+colour6+colour5+colour4+colour3+colour2+colour1
        sequencer_line = sequencer_line + [200]
        csvwriter.writerow(sequencer_line)

        if(1):
            while(note_time_stamp < beat_times[i+1]):
                colour1,colour8 = [1,1,1],[1,1,1]
                sequencer_line = [round(note_time_stamp,2),0]
                colour2,colour3,colour4 = [1,1,1],[1,1,1],[1,1,1]
                colour5,colour6,colour7 = [1,1,1],[1,1,1],[1,1,1]

                stft_spec = np.abs(np.fft.rfft(audioH[j*512:j*512+fft_N]))
                norm = stft_spec / np.linalg.norm(stft_spec)
                red = np.sum(norm[int(10/bin_span):int(50/bin_span)])
                green = np.sum(norm[int(50/bin_span):int(200/bin_span)])
                blue = np.sum(norm[int(200/bin_span):int(1000/bin_span)])

                red = int(min(max(2000*red,1),255))
                green = int(min(max(150*green,1),255))
                blue = int(min(max(50*blue,1),255))

                max_c = max(max(red,green),blue)
                red = int(255*red/max_c)
                green = int(255*green/max_c)
                blue = int(255*blue/max_c)
                data = Counter(notes_f[j:j+10])
                note_mode = data.most_common(1)[0][0]
                print("note_mode "+str(note_mode))
                if(note_mode == 0 or note_mode == 6):
                    colour2 = [red,green,blue]
                if(note_mode == 1 or note_mode == 7):
                    colour3 = [red,green,blue]
                if(note_mode == 2 or note_mode == 8):
                    colour4 = [red,green,blue]
                if(note_mode == 3 or note_mode == 9):
                    colour5 = [red,green,blue]
                if(note_mode == 4 or note_mode == 10):
                    colour6 = [red,green,blue]
                if(note_mode == 5 or note_mode == 11):
                    colour7 = [red,green,blue]

                j = j + 10
                note_time_stamp = j*512/srH
```

```
                sequencer_line = sequencer_line + colour8+colour7+colour6+colour5+colour4+colour3+colour2+
    colour1
                sequencer_line = sequencer_line + [200]
                csvwriter.writerow(sequencer_line)
        else:
            print(str(i)+" " + str(j))
    sequencer_line[0] = sequencer_line[0]+0.01
    sequencer_line[1:] = [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,200]
    csvwriter.writerow(sequencer_line)

    '''
    # Plot the 2 separated spectrograms
    plt.figure()
    specshow(DH, y_axis='linear')
    plt.colorbar(format='%+2.0f dB')
    plt.title('Power spectrogram of harmonics from ' + filename)
    plt.show()
    specshow(DP, y_axis='linear')
    plt.colorbar(format='%+2.0f dB')
    plt.title('Power spectrogram of percussions from ' + filename)
    plt.show()'''
```

8.2 Fountain Streaming Script

```
import soundcard as sc
import sounddevice as sd
import time as tm
import numpy as np
import matplotlib.pyplot as plt
import serial
import serial.tools.list_ports
import csv
import sys
from scipy.io import wavfile


com_list = []
i=1
for item in list(serial.tools.list_ports.comports()):
    print(str(i)+". "+str(item))
    com_list.append(str(item))
    i = i+1

choice  = int(input("Input the number corresponding to Output COM Port\n"))
print("Opening "+com_list[choice-1])
ser = serial.Serial(com_list[choice-1].split(" ")[0])
ser.baudrate = 9600
tm.sleep(2)

file_name = sys.argv[1][2:]
file_stream = open(file_name,'r')
csv_read = csv.reader(file_stream, delimiter = ',')

all_rows = [[]]
time_stamp = []
byte_array = []

for item in csv_read:
    all_rows.append(item)
    time_stamp.append(float(item[0]))
    res = [int(eval(i)) for i in item[1:]]
    byte_array.append(bytearray(res))

i = 0
#print(time_stamp)
#print(byte_array)

start_time = 0

def callback(outdata, frames, time, status):
    global byte_array,i,ser,start_time
```

```
    if(start_time == 0):
        start_time = tm.time()
    if(i < len(byte_array)):
        if(tm.time() - start_time > time_stamp[i]):
            ser.write(byte_array[i])
            i = i + 1




music_file_path = file_name[0:file_name.index('.')]+'.wav'

# Load the music file using scipy
try:
    music_samplerate, music_data = wavfile.read(music_file_path)
except Exception as e:
    print(f"Error loading music file: {e}")
    exit()
#print(sd.query_devices())
#choice = int(input("Enter playback device"))
# Set the callback function
sd.default.latency = 'low'  # You can adjust the latency if needed
stream = sd.OutputStream(callback=callback, channels=music_data.shape[1])
try:
    # Start the stream
    stream.start()

    # Play the loaded music
    sd.play(music_data, music_samplerate)

    # Wait for the music to finish playing
    sd.sleep(int(len(music_data)*1000/music_samplerate))
except sd.PortAudioError as e:
    print(f"PortAudioError: {e}")
finally:
    # Stop and close the stream
    stream.stop()
    stream.close()

print("Music playback complete.")


#dser.write(bytes_to_send);
```