LED Dynamic Control using IIR Filters – Assignment 3

Team17: Jalal Sayed(2571964s) and Tamim Abdul Maleque(2523948a)

GitHub repo: github.com/JalalSayed1/IIR-filters

Demo video: https://youtu.be/jVWNfqF8uNY

Introduction

In this assignment, we have developed an application that simulates changing the colour distribution of an LED strip using a joystick to manipulate the colour of two Red-Blue LEDs. This creates a dynamic colour-switching effect. This system makes a new way of interacting with an RGB LED strip where the controller is a joystick, making it easier to switch to a desired combination of colours.

The choice of a joystick as an input interface is particularly relevant because, in the real world, larger-sized joysticks used for different applications, such as drone control, can be susceptible to mechanical vibrations and electrical noise arising from other RLC circuit configurations for the whole system. These disturbances can cause unwanted signals that can impact control, hence, using IIR filters is ideal for processing signals in a recursive way.

Description

Set-up of Hardware

1. Connect a joystick to your Arduino board.
    - Connect the X-axis to analogue pin A0 and then connect the power and GND pins to the board's 5V and GND pins.
2. Connect two RB LED's to your breadboard.
    - Connect the red pin from one of the LEDs to digital pin 5 of the Arduino.
    - Connect the blue pin of this LED to digital pin 6.
    - Connect the other RB LED to the same pins but swapped (the red connected to the blue of the other LED and vice versa). This makes the switching effect.
    - Connect both GND pins to the board's GND.
    - No resistors are needed.

The set-up to our project is shown in figure 1, where the project was prototyped on a breadboard.

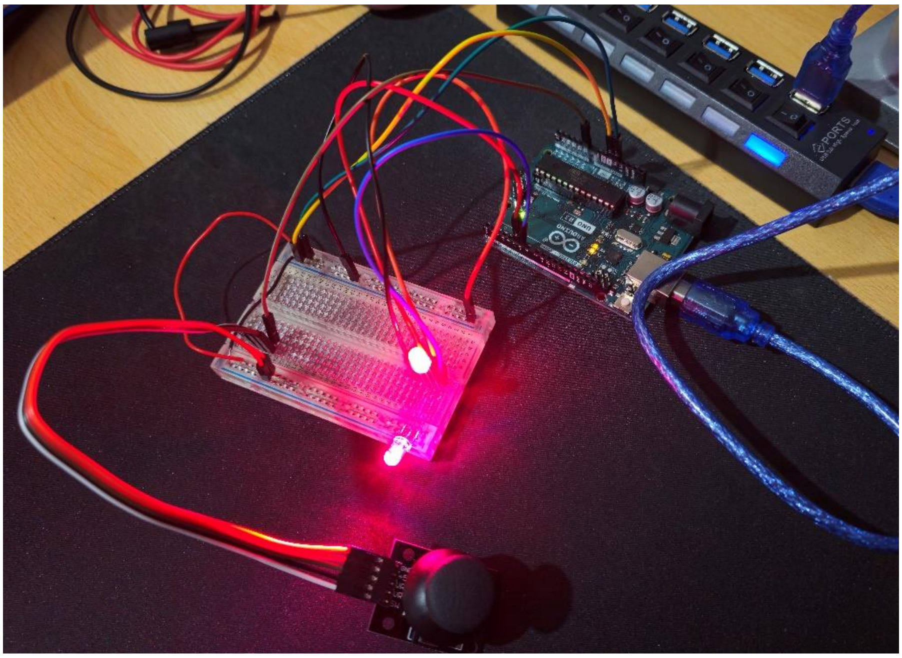

Figure 1: The set-up of the hardware for the LED dynamic control showing the joystick on the bottom, breadboard on the middle with the 2 LEDs set up in switching mode configuration and Arduino Uno R3 used for the project on top.

Dataflow Diagram

This project follows the dataflow as shown in figure 2, where 2nd Order Butterworth Lowpass filter was used. The project code as a toggle button to switch on and off the IIR filtering.

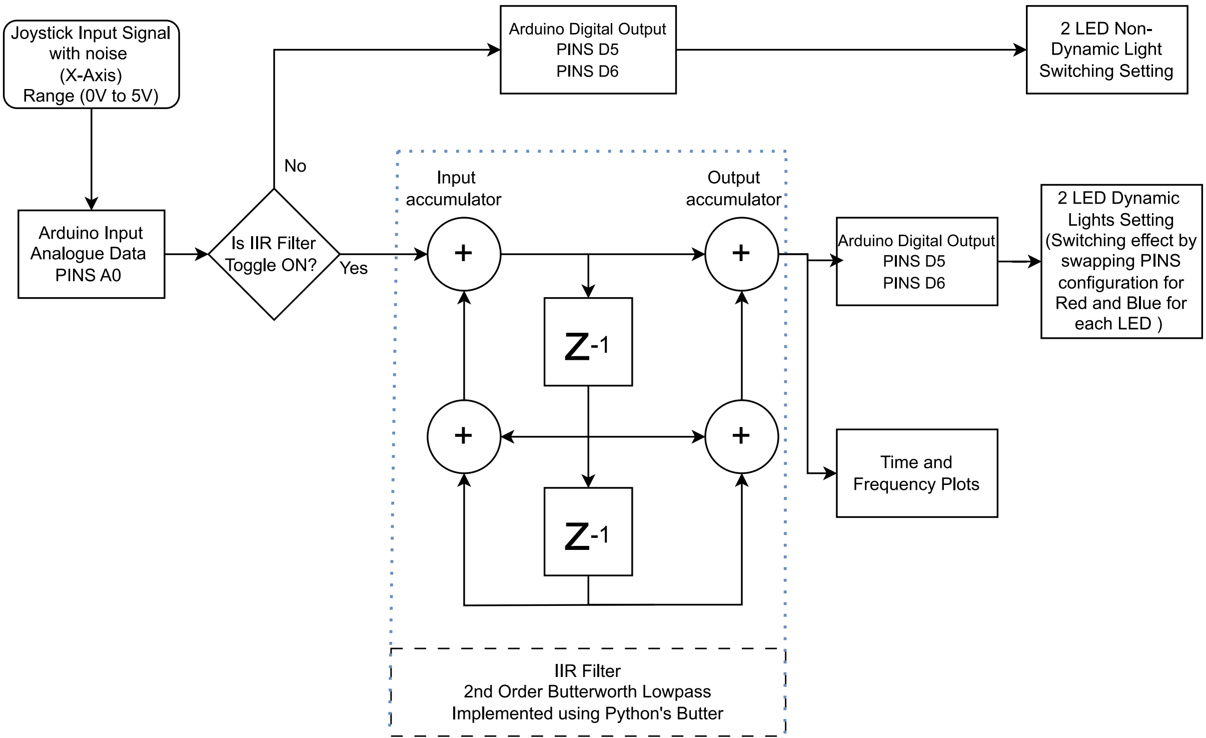

Figure 2: The dataflow diagram of the LED Dynamic Control using IIR Filters.

Filter response required

In this project, we use a low-pass filter to ensure smoother transitions. Because human-operated joystick movements usually generate lower-frequency signals (below 10Hz), the high-frequency noise arising from mechanical vibrations and/or electrical disturbances from external sources is therefore filtered out. A stable and predictable LED response to joystick movements was thus maintained through signal processing. We chose a second-order low-pass filter implemented via Python's butter function for Butterworth filtering. This specific order offered good filtering of unwanted high frequencies and maintained computational efficiency.

A bandpass response in the lower frequency was also tested. However, it was quite difficult to have a smoother transition as maintaining the certain frequency bandwidth proved difficult and hence caused flickering in the LED light transition.

A sampling frequency of 200Hz was chosen, as this frequency would effectively capture human signal movement while fulfilling Nyquist's rate of 100Hz. We attempted a higher sampling rate, about 400Hz , although we think it may have provided a more detailed data sample, this was not significant in our observation but noticed latency in the LED's response to joystick movements. Hence, a compromise of 200Hz was chosen.

Code snippet for this section implementation:

```
SAMPLING_RATE = 200  # Hz

# ' IIR filter:
LP_CUTOFF = 10  # Hz
NYQUIST_RATE = SAMPLING_RATE / 2
sos = butter(2, LP_CUTOFF / NYQUIST_RATE, btype='low', output='sos')
iir_filter = IIR_filter(sos)
```

Operation: Updating the LEDs

The two LEDs should be connected in a swapped order to one another to simulate the LED strip-switching effect. When moving the joystick to one position (left or right), the red or blue colour will increase in one LED, which decreases by the same amount in the other one. When the joystick is on the furthest side, one LED will be red while the other is only blue.

Code snippet:

```
def update_led_color(pin_value):
    duty_cycle_red = interpolate(pin_value, 0, 1, 0, 1)
    duty_cycle_blue = 1 - duty_cycle_red

    board.digital[LED_RED_PIN].write(duty_cycle_red)
    board.digital[LED_BLUE_PIN].write(duty_cycle_blue)
```

When the pin value changes, the callback function gets called. The callback function stores the new value in a global variable called “current_sample” which is used by the “update” function.

Code snippet:

```
def callback(value):
    global current_sample
    current_sample = value

…

board.analog[X_AXIS_INPUT].register_callback(callback)
board.analog[X_AXIS_INPUT].enable_reporting()
```

The “update” function updates the LED with either the raw data or the filtered one and updates the plot. It is worth noting that, the filter might return a value slightly above 1 or below 0. This is because of the higher order low pass filter that we used. The PWM signal only accept values between 0 and 1 so the filtered data will not work all the time. To fix this, we overwrite the filtered data with a 0 or 1 using “np.clip()” numpy function if it happens to be below or above these values, before updating the PWM signal of the LEDs.

```
def update(frame):
    …

    # Update LED
    if use_filtered_data:
        # max value for filtered data is 1 and min is 0. Check that is true:
        filtered_data = np.clip(filtered_data, 0, 1)
        print(f"Filtered data: {filtered_data}")
        update_led_color(filtered_data)
    else:
        print(f"Raw data: {current_sample}")
        update_led_color(current_sample)

    return line_time, line_freq, filtered_line_time, filtered_line_freq

….

ani = animation.FuncAnimation(fig, update, init_func=init, blit=True, interval=1)
```

Checking sampling rate of acquisition while the main application is running

To monitor the actual rate at which the system is processing, the function calculates the current sampling rate by dividing the number of samples received in the elapsed time. This gives a real-time update of how fast the system is sampling data, measured in hertz. This continuous update helps to see if data acquisition is performing as expected. During the testing, the sampling rate acquisition was within the range of 198Hz to 200Hz, which was sufficient for our project.

Unfortunately, the actual sampling rate couldn't be shown on the plot, however it was printed to the terminal to observe and the demonstration is shown in a separate video, apologies as this was implemented later on: https://youtu.be/Fu3J9hHB_j8?feature=shared

Code snippet:

```
# used to calculate the actual sampling rate:
sample_count = 0
last_time_checked = time.time()
actual_sampling_rate = 0
current_sample = 0.0

def check_sampling_rate():
    global last_time_checked, sample_count, actual_sampling_rate
    current_time = time.time()
    elapsed_time = current_time - last_time_checked
    if elapsed_time >= 1.0:  # Check every second
        actual_sampling_rate = sample_count / elapsed_time
        print(f"Actual Sampling Rate: {actual_sampling_rate:.3f} Hz vs set sampling rate:
{SAMPLING_RATE} Hz")
        sample_count = 0
        last_time_checked = current_time
```

Comparison: (Filtered data vs raw data)

Before filtering, the response from the joystick was so fast that it made a sudden fast change in the LED colour, which is not ideal. The LED colours were simply following the blue graph below. Therefore, the transition was extremely hard to see when moving the joystick.

After filtering, the LEDs started changing colours smoothly to the desired colour combination. This is shown in the orange graph below. It's worth noting that small changes in the middle of the switching are not recorded as they don't matter, but instead we wanted to smoothly move from one colour to another.

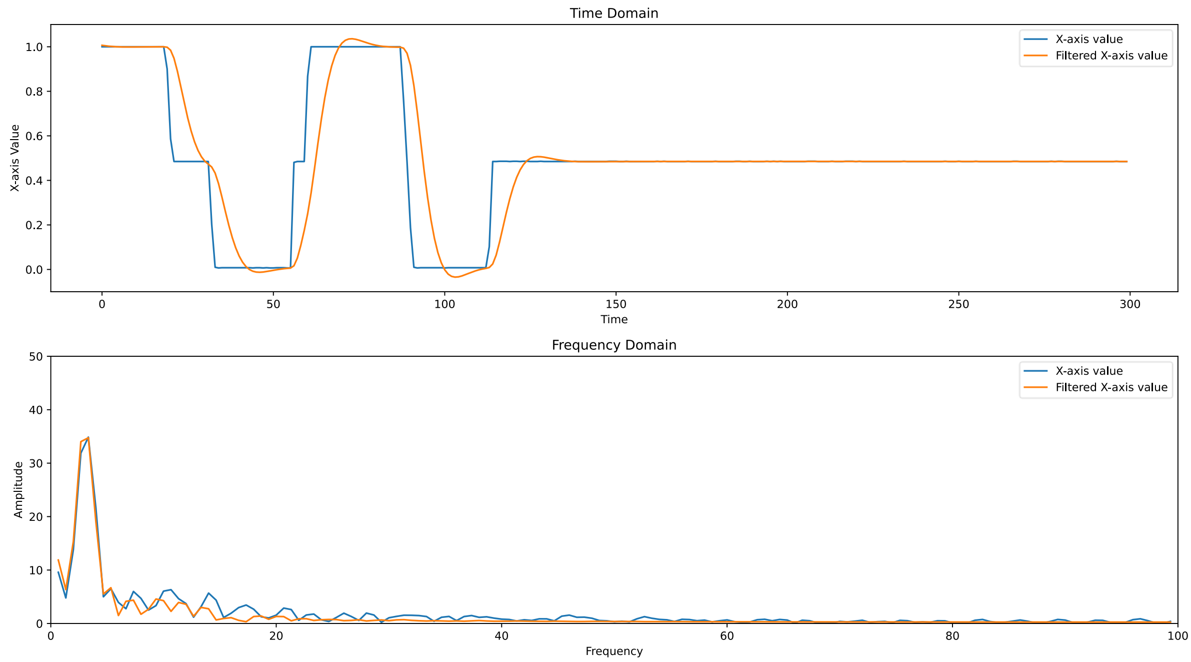

Figure 3: Time domain and frequency domain plots for joystick control of LED lights.

A demonstration video was recorded and published on YouTube, which can be accessed here.

Conclusion

In summary, our project introduced an interactive application that uses a joystick to control the colour distribution of two Red-Blue LEDs, creating a dynamic colour-switching effect. We strategically chose a joystick interface, considering potential mechanical and electrical disturbances, and addressed these challenges by implementing IIR filters for signal processing.

We have successfully filtered the data coming from the joystick in real time to smooth out the colour transition between the two LEDs in our system.

Our project idea was successfully achieved and demonstrated in a video showcasing seamless colour switching using a joystick. This project highlighted the effectiveness of real-time data filtering using IIR filters in enhancing the system's response.

Appendix

1. LED controller code:

```
from pyfirmata2 import Arduino, PWM
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from py_iir_filter.iir_filter import IIR_filter
from scipy.signal import butter
import tkinter as tk
import time

# ' Constants
X_AXIS_INPUT = 0  # Analog pin A0 for X axis
LED_RED_PIN = 5  # Digital pin D5 for red LED
LED_BLUE_PIN = 6  # Digital pin D6 for blue LED

PORT = Arduino.AUTODETECT
BUFFER_SIZE = 300  # samples
SAMPLING_RATE = 200  # Hz
print(f"Sampling rate: {SAMPLING_RATE} Hz")

# Plotting figure size:
WIDTH = 15  # inches
HEIGHT = WIDTH * (9 / 16)  # 16:9 aspect ratio

# ' Arduino Setup
board = Arduino(PORT)
board.samplingOn(1000 / SAMPLING_RATE)

# ' Set the pins as PWM output
board.digital[LED_RED_PIN].mode = PWM
board.digital[LED_BLUE_PIN].mode = PWM

# ' IIR filter:
LP_CUTOFF = 10  # Hz
NYQUIST_RATE = SAMPLING_RATE / 2
sos = butter(2, LP_CUTOFF / NYQUIST_RATE, btype='low', output='sos')
iir_filter = IIR_filter(sos)

# ' Plotting:
# initialized to none so they will be created in setup_plotting()
fig, (ax_time, ax_freq) = None, (None, None)
line_time = None
filtered_line_time = None
line_freq = None
filtered_line_freq = None
time_domain_data = np.zeros(BUFFER_SIZE)
```

```
filtered_time_domain_data = np.zeros(BUFFER_SIZE)
freq_domain_data = np.zeros(BUFFER_SIZE // 2)
filtered_freq_domain_data = np.zeros(BUFFER_SIZE // 2)

# current sample from pin:
current_sample = 0.0
# used to calculate the actual sampling rate:
sample_count = 0
last_time_checked = time.time()
actual_sampling_rate = 0
current_sample = 0.0

# Function to check actual sampling rate for data acquisition
def check_sampling_rate():
    global last_time_checked, sample_count, actual_sampling_rate
    current_time = time.time()
    elapsed_time = current_time - last_time_checked
    if elapsed_time >= 1.0:  # Check every second
        actual_sampling_rate = sample_count / elapsed_time
        print(f"Actual Sampling Rate: {actual_sampling_rate:.3f} Hz vs set sampling rate:
{SAMPLING_RATE} Hz")
        sample_count = 0
        last_time_checked = current_time

# Make a button to toggle between using filtered data or not in real time:
use_filtered_data = False
status_text = None


# Use the filtered data to update the LED or raw data (True or False):
def toggle_filtered_data():
    global use_filtered_data, status_text
    use_filtered_data = not use_filtered_data
    text = "Using filtered data to update LED.." if use_filtered_data else "NOT using filtered data to
update LED.."
    print(text)
    status_text.config(text=text)


# Create a function to setup the GUI elements
def setup_gui():
    global use_filtered_data, status_text
    root = tk.Tk()
    root.title("Toggle Filtered Data")

    # Set the initial size of the window (width x height)
    window_width = 300
    window_height = 80
    screen_width = root.winfo_screenwidth()
```

```
    screen_height = root.winfo_screenheight()
    x_coordinate = (screen_width / 2) - (window_width / 2)
    y_coordinate = (screen_height / 2) - (window_height / 2)
    root.geometry("%dx%d+%d+%d" %
                  (window_width, window_height, x_coordinate, y_coordinate))

    # Create" a button to toggle filtered data
    toggle_button = tk.Button(
        root, text="Toggle LED response", command=toggle_filtered_data)

    text = "Using filtered data to update LED.." if use_filtered_data else "NOT using filtered data to
update LED.."
    status_text = tk.Label(root, text=text)

    status_text.pack()
    toggle_button.pack()

    return root


# Function to update LED color based on joystick values
def update_led_color(pin_value):
    # pin_value is already between 0 and 1:
    duty_cycle_red = pin_value
    duty_cycle_blue = 1 - duty_cycle_red

    board.digital[LED_RED_PIN].write(duty_cycle_red)
    board.digital[LED_BLUE_PIN].write(duty_cycle_blue)


def setup_plotting(fig, ax_time, ax_freq, time_domain_data, line_time, line_freq,
filtered_time_domain_data, filtered_line_time):
    # Initialize Figures
    fig, (ax_time, ax_freq) = plt.subplots(2, 1, figsize=(WIDTH, HEIGHT))

    # Time Domain Plot
    ax_time.set_xlabel('Time')
    ax_time.set_ylabel('X-axis Value')
    ax_time.set_title('Time Domain')
    ax_time.set_ylim(-0.1, 1.1)
    line_time, = ax_time.plot(np.arange(BUFFER_SIZE),
                              time_domain_data, label='X-axis value')
    filtered_line_time, = ax_time.plot(np.arange(
        BUFFER_SIZE), filtered_time_domain_data, label='Filtered X-axis value')
    ax_time.legend()  # Add legend for time domain plot

    # Frequency Domain Plot
    ax_freq.set_xlabel('Frequency')
    ax_freq.set_ylabel('Amplitude')
```

```
    ax_freq.set_title('Frequency Domain')
    ax_freq.set_xlim(0, SAMPLING_RATE / 2)
    ax_freq.set_ylim(0, 50)
    line_freq, = ax_freq.plot([], [], label='X-axis value')
    filtered_line_freq, = ax_freq.plot([], [], label='Filtered X-axis value')
    ax_freq.legend()  # Add legend for frequency domain plot

    return fig, (ax_time, ax_freq), time_domain_data, line_time, line_freq,
filtered_time_domain_data, filtered_line_time, filtered_line_freq


# Function to Update Plots
def update(frame):
    global time_domain_data, filtered_time_domain_data, current_sample, use_filtered_data
    check_sampling_rate()
    if current_sample is not None:
        # Update Time Domain Data
        # store raw data:
        time_domain_data = np.roll(time_domain_data, -1)
        time_domain_data[-1] = current_sample

        # Apply filter:
        filtered_data = iir_filter.filter(current_sample)
        filtered_time_domain_data = np.roll(filtered_time_domain_data, -1)
        filtered_time_domain_data[-1] = filtered_data

        # plot time domain data:
        line_time.set_ydata(time_domain_data)
        filtered_line_time.set_ydata(filtered_time_domain_data)

        # Update Frequency Domain Data
        fft_data = np.fft.fft(time_domain_data)
        filtered_fft_data = np.fft.fft(filtered_time_domain_data)
        # get the frequency bins:
        fft_freq = np.fft.fftfreq(BUFFER_SIZE, 1 / SAMPLING_RATE)

        # plot the data:
        mask = fft_freq > 0  # Only plot the positive frequencies
        line_freq.set_data(fft_freq[mask], np.abs(fft_data[mask]))
        filtered_line_freq.set_data(
            fft_freq[mask], np.abs(filtered_fft_data[mask]))

    # Update LED
    if use_filtered_data:
        # max value for filtered data is 1 and min is 0. Check that is true:
        filtered_data = np.clip(filtered_data, 0, 1)
        #print(f"Filtered data: {filtered_data}")
        update_led_color(filtered_data)
    else:
```

```
        #print(f"Raw data: {current_sample}")
        update_led_color(current_sample)

    return line_time, line_freq, filtered_line_time, filtered_line_freq


# Function to Initialize the Plot
def init():
    line_time.set_ydata(np.zeros(BUFFER_SIZE))
    filtered_line_time.set_ydata(np.zeros(BUFFER_SIZE))
    line_freq.set_data([], [])
    filtered_line_freq.set_data([], [])
    return line_time, line_freq, filtered_line_time, filtered_line_freq


# Arduino Callback for LED Update
def callback(value):
    global current_sample, sample_count
    current_sample = value
    sample_count += 1

# Setup everything related to plotting:
fig, (ax_time, ax_freq), time_domain_data, line_time, line_freq, filtered_time_domain_data,
filtered_line_time, filtered_line_freq = setup_plotting(
    fig, ax_time, ax_freq, time_domain_data, line_time, line_freq, filtered_time_domain_data,
filtered_line_time)

# Create Animation
ani = animation.FuncAnimation(
    fig, update, init_func=init, blit=True, interval=1, cache_frame_data=False)

board.analog[X_AXIS_INPUT].register_callback(callback)
board.analog[X_AXIS_INPUT].enable_reporting()

try:
    root = setup_gui()
    plt.tight_layout()
    plt.show()

except KeyboardInterrupt:
    print("Interrupted by user")

finally:
    # Cleanup:
    board.digital[LED_BLUE_PIN].write(0)
    board.digital[LED_RED_PIN].write(0)
    board.exit()
    print("Program terminated")
```

2. IIR filter code:

```
class IIR_filter:
  """IIR filter"""
  def __init__(self,sos):
    """Instantiates an IIR filter of any order
    sos -- array of 2nd order IIR filter coefficients
    """
    self.cascade = []
    for s in sos:
      self.cascade.append(IIR2_filter(s))

  def filter(self,v):
    """Sample by sample filtering
    v -- scalar sample
    returns filtered sample
    """
    for f in self.cascade:
      v = f.filter(v)
    return v
```