

Electronics Final Project Report

Joshua Parr

Abstract—I designed, simulated, and implemented an audio signal analyzer that employs analog circuitry to detect and visualize a song's spectral content in real time. The system targets two distinct frequency bands—bass and treble—using second-order Sallen-Key low-pass and high-pass filters centered at 320 Hz and 640 Hz, respectively. These filters separate the audio into low- and high-frequency components, which I then process through two signal paths. Each path includes a peak detector to capture maximum amplitudes and an LED driver circuit—an analog driver for high frequencies and a pulse-width modulation (PWM) driver for low frequencies—to control brightness. I used theoretical calculations and Python code to select optimal standard component values, followed by Multisim simulations and experimental verification with the Analog Discovery 2 and WaveForms software.

I. BACKGROUND INFORMATION AND RATIONALE

I developed an audio analyzer that transforms sound into a visual display using LED indicators. The device processes an audio signal, separates low and high frequency components, and illuminates two LEDs based on their strength. The system first combines the two input channels into a single signal. Then, it filters the signal into two parts: one captures deep bass sounds below 350 Hz, and the other isolates brighter treble sounds above 640 Hz. A circuit processes the filtered signals to identify the highest sound level over a short time, ensuring I capture and use the most prominent audio parts to control LED brightness.

I selected the song "Levels" by Avicii for this project because it features strong high and low frequency components in different sections, making it ideal to showcase the analyzer's capabilities. Using Audacity software, I analyzed the song's frequency components to determine optimal low and high cutoff frequencies.

Fig. 1 displays the frequency spectra of the song's treble-heavy sections. Peaks around 650 Hz confirm that 640 Hz serves as an effective high-frequency cutoff.

Similarly, Fig. 2 shows the spectra of low-frequency sections. A peak at 320 Hz supports my choice of a low-frequency cutoff near this value.

These graphs demonstrate the song's distinct low- and high-frequency components, validating my choice of "Levels" for the audio analyzer.

II. INPUT STAGE: SUMMING AMPLIFIER

A. System Overview

I designed the summing amplifier as the front-end signal conditioning stage to combine stereo left and right audio

I prepared this final project report for ECE 2600 Electronics, Spring 2025 at the University of Virginia

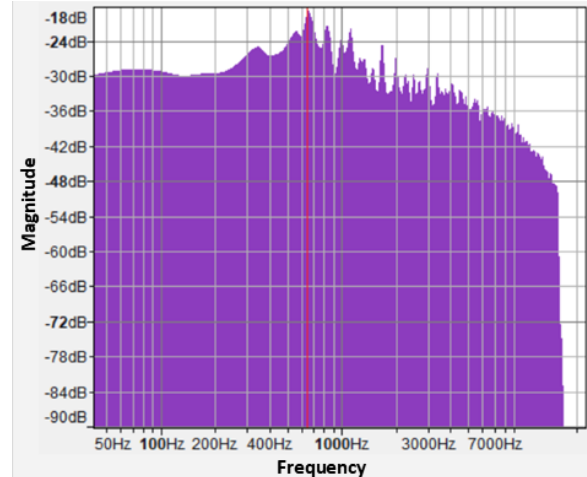


Fig. 1. I observed high frequency spectra of "Levels" with peaks between 650 and 820 Hz.

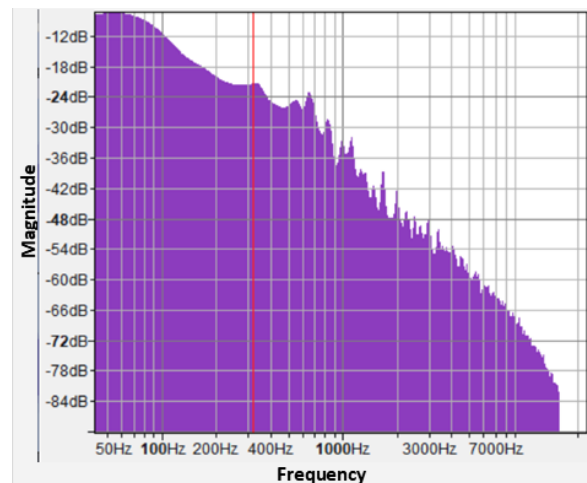


Fig. 2. I identified low frequency spectra of "Levels" with a peak around 320 Hz.

channels into a single monaural signal while filtering out DC offsets. This single-channel signal ensures uniform operation of subsequent filter and peak detection circuits, regardless of stereo separation. I set three requirements: (1) maintain unity gain with both channels present, (2) reject DC offsets below 20 Hz, and (3) produce an output signal amplitude within ± 4.5 V for op-amp rail compliance.

B. Design and Analysis

I configured the amplifier with an op-amp in a non-inverting T-network. For AC coupling, I used a series

capacitor and resistor pair ($C_1 = 0.1 \mu\text{F}$, $R_{in} = 82 \text{ k}\Omega$), yielding a cutoff frequency:

$$f_c = \frac{1}{2\pi R_{in} C_1} = \frac{1}{2\pi \cdot 82000 \cdot 0.1 \times 10^{-6}} \approx 19.4 \text{ Hz}.$$

I calculated the gain as:

$$\text{Gain} = \frac{R_3}{R_{in}} + R_5 \left(\frac{1}{R_1} + \frac{R_3}{R_{in} R_4} \right),$$

where $R_3 = 33 \text{ k}\Omega$, $R_4 = R_5 = 180 \text{ k}\Omega$, and $R_{in} = 82 \text{ k}\Omega$. I selected these values using a Python brute-force script from E-series standard resistors to achieve a target gain of 3.0 with minimal error. The resulting gain exactly matches 3.0, meeting the amplification needs for downstream processing.

C. Simulations

I conducted Multisim simulations using 1 kHz, 0.5 V peak sine waves on both stereo channels. Fig. 3 shows the output (orange) reaching approximately 3.0 V peak when both inputs (blue) are active, confirming the gain and frequency-domain behavior.

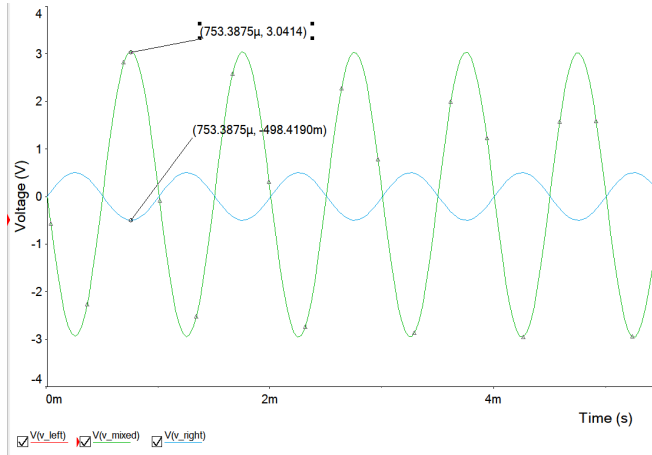


Fig. 3. I simulated the summing amplifier in Multisim, showing output (orange) and combined inputs (blue).

D. Experimental Results

I constructed and tested the summing amplifier on a breadboard with a 1 V input (from two 0.5 V sine waves). Fig. 4 displays the Analog Discovery 2 output, where the blue curve represents the combined input signal, and the orange curve shows the output. I measured a gain of -3 V, as expected for the inverting configuration. I plan to validate the final design with PCB measurements in the next iteration.

III. HIGH AND LOW PASS FILTERS

I implemented two second-order Sallen-Key filters—a low-pass filter for bass signals and a high-pass filter for treble signals—to enable real-time visualization of audio

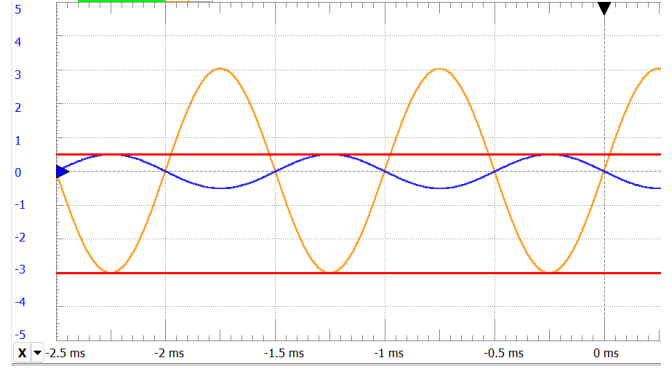


Fig. 4. I tested the summing amplifier on a breadboard. Blue = input signal; Orange = output at -3 V.

frequency bands. These filters drive two LED indicators, enhancing user feedback during music playback. I chose cutoff frequencies of approximately 320 Hz for the low-pass filter and 640 Hz for the high-pass filter to align with the spectral energy of the audio input.

A. Design and Analysis

I designed each filter using the unity-gain ($K = 1$) Sallen-Key topology to achieve a Butterworth-like flat response with a quality factor of $0.5 < Q \leq 0.707$. I derived the low-pass filter's transfer function as:

$$H_{LP}(s) = \frac{1}{s^2 R_6 R_7 C_7 C_8 + s(R_6 C_7 + R_7 C_7) + 1}$$

For the high-pass filter, I used:

$$H_{HP}(s) = \frac{s^2 R_8 R_9 C_9 C_{10}}{s^2 R_8 R_9 C_9 C_{10} + s(R_8 C_9 + R_9 C_9) + 1}$$

I approximated the -3 dB cutoff frequency for each as:

$$f_c = \frac{1}{2\pi \sqrt{R_6 R_7 C_7 C_8}} \quad (\text{low-pass}), \quad f_c = \frac{1}{2\pi \sqrt{R_8 R_9 C_9 C_{10}}} \quad (\text{high-pass})$$

Using a Python script to sweep standard E-series component values, I selected:

- **Low-pass filter:** $R_6 = 33 \text{ k}\Omega$, $R_7 = 680 \text{ k}\Omega$, $C_7 = 1 \text{ nF}$, $C_8 = 10 \text{ nF} \rightarrow f_c \approx 315.5 \text{ Hz}$
- **High-pass filter:** $R_8 = 270 \text{ k}\Omega$, $R_9 = 56 \text{ k}\Omega$, $C_9 = 0.68 \text{ nF}$, $C_{10} = 6.8 \text{ nF} \rightarrow f_c \approx 638.7 \text{ Hz}$

B. Multisim Simulations

I performed AC sweep simulations from 10 Hz to 5 MHz with 201 logarithmic steps to evaluate the filters' frequency response. I applied a 1 V peak sine wave to both filter inputs. Fig. 5 shows the low-pass filter (red) attenuating frequencies above 320 Hz and the high-pass filter (green) passing signals around 640 Hz, each with a -40 dB/decade roll-off, consistent with second-order filter behavior.

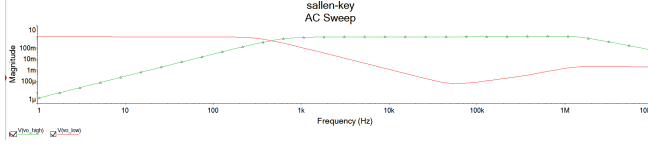


Fig. 5. I generated a Multisim Bode magnitude plot for high-pass (green) and low-pass (red) Sallen-Key filters.

C. Experimental Results

I built the filters on a breadboard and tested them using the Analog Discovery 2 and WaveForms software. I conducted an AC sweep from 10 Hz to 5 kHz with 201 frequency points and a 1 V peak input signal. The Analog Discovery's Bode analyzer provided magnitude and phase responses.

Fig. 6 shows the low-pass filter response, maintaining a flat 0 dB up to 320 Hz, then declining at nearly -40 dB/decade. Fig. 7 shows the high-pass filter with increasing gain starting at 640 Hz, reaching 0 dB by 1 kHz. These results confirm the simulated performance and verify the filters' functionality.

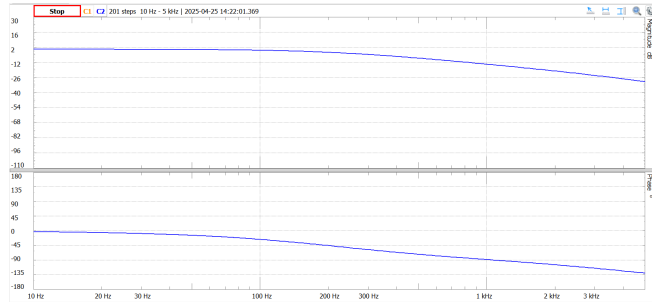


Fig. 6. I measured the low-pass filter's Bode plot using Analog Discovery 2 (1 V input, 10 Hz–5 kHz).

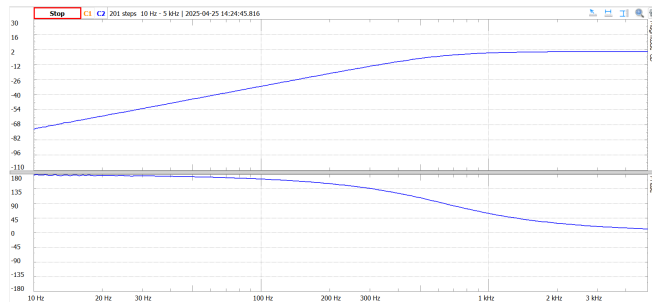


Fig. 7. I measured the high-pass filter's Bode plot using Analog Discovery 2 (1 V input, 10 Hz–5 kHz).

IV. PEAK DETECTOR CIRCUIT

A. System Overview

I designed the peak detector to capture and hold the highest instantaneous amplitude of the filtered audio signal over a short time window. This peak value controls LED brightness, visually indicating signal strength. The circuit tracks dynamic audio energy changes and drives analog

or PWM LED drivers responsively, even as the waveform oscillates. I ensured the circuit responds quickly to amplitude changes while maintaining stability to avoid flicker from brief signal dips.

B. Design and Analysis

I built the circuit with a 1N4001 diode, a $0.47 \mu\text{F}$ capacitor (C), and a $150 \text{ k}\Omega$ discharge resistor (R). The diode charges the capacitor to the input waveform's peak value (minus the diode's 0.7 V forward voltage drop). The resistor controls the capacitor's discharge rate to prevent instantaneous decay. I targeted a 60–100 ms peak hold time for smooth LED dimming.

I calculated the RC time constant τ as:

$$\tau = RC = (150 \times 10^3)(0.47 \times 10^{-6}) \approx 70.5 \text{ ms}.$$

The capacitor voltage during discharge follows:

$$V_c(t) = V_{\max} e^{-t/\tau},$$

where $V_{\max}$ represents the most recent peak voltage. I selected this duration based on empirical testing to ensure responsive yet stable LED behavior.

C. Circuit Schematic

Fig. 8 illustrates the peak detector schematic, including D_1 (1N4001 diode), C_{12} ($0.47 \mu\text{F}$ capacitor), and R_{12} ($150 \text{ k}\Omega$ resistor).

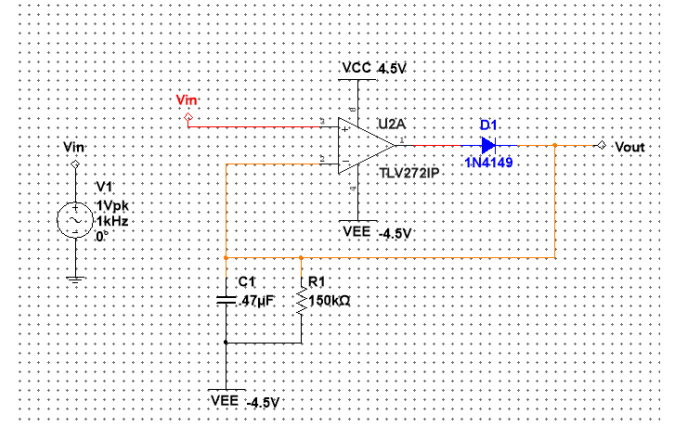


Fig. 8. I designed the peak detector circuit with D_1 , R_{12} , and C_{12} .

D. Multisim Simulation

I simulated the peak detector in Multisim using a 1 V peak, 1 kHz sine wave input with an ideal op-amp buffer and passive components. The capacitor charged to nearly the peak value and decayed with a 70.5 ms time constant. I plotted input and output waveforms to confirm the exponential decay during sine wave valleys, validating the LED's expected response time.

E. Experimental Measurements

I tested the peak detector on a PCB with a 1 V, 1 kHz sine wave using Analog Discovery 2. Fig. 9 shows the input waveform (blue) and capacitor voltage output (orange). I measured the decay over a 0.5 ms interval to calculate τ :

$$\tau = -\frac{t}{\ln(V_c/V_{\max})} \approx 71.8 \text{ ms},$$

which deviates by only 1.84% from the theoretical value, confirming the analytical model and the sub-system's performance.

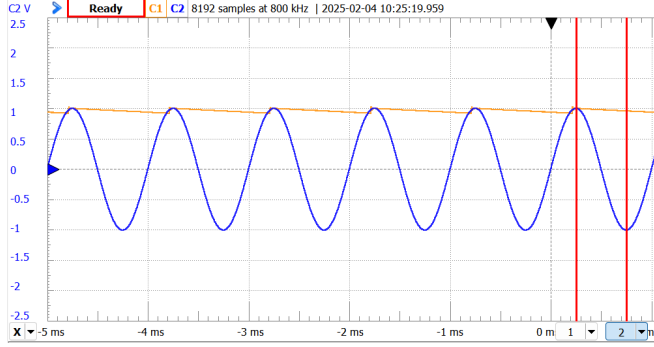


Fig. 9. I measured the peak detector's response on a PCB. Blue = input; Orange = held peak voltage.

V. ANALOG LED DRIVER

I created the analog LED driver to provide a continuous indication of high-pass frequency signal levels. When the high-pass filter output amplitude exceeds a threshold, the MOSFET activates, illuminating the LED. This enables real-time, smooth LED brightness variation corresponding to high-frequency audio content.

A. Design and Analysis

Fig. 10 depicts the analog LED driver. An op-amp buffer and diode feed the gate of an n-channel MOSFET, with the LED in the drain. A source resistor R_{13} limits the LED current, while resistors R_{10} and R_{11} set a DC bias for the gate.

I selected R_{13} to prevent exceeding the LED's maximum rated current, using the LED forward voltage $V_{LED} = 2.5 \text{ V}$, negative power supply rail $V_{EE} = -5 \text{ V}$, and peak forward current $I_{max} = 7 \text{ mA}$. I calculated:

$$R_{13} = \frac{V_{LED} - V_{EE}}{I_{max}} = 1000 \Omega,$$

and chose $R_{13} = 1.2 \text{ k}\Omega$ from available kit values. I set bias resistors R_{10} and R_{11} to ensure the MOSFET remains in cutoff (LED off) without a high-frequency signal. I determined the gate bias voltage as:

$$V_G = V_{hpf} + \frac{R_{11}}{R_{10} + R_{11}}(V_{CC} - V_{EE}) + V_{EE}.$$

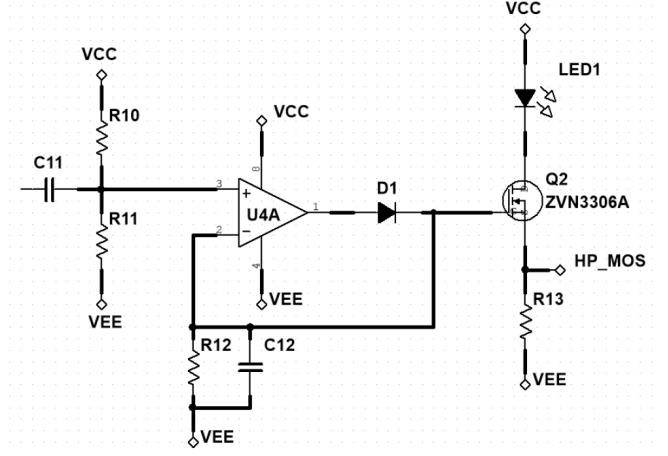


Fig. 10. I designed the analog LED driver schematic.

Using a Python script (see Appendix), I selected $R_{10} = 6.8 \text{ k}\Omega$ and $R_{11} = 1.5 \text{ k}\Omega$.

B. Experimental Results

I tested the analog LED driver with a 20 Hz AC wave as the simulated input. As I varied the input wave's amplitude, the LED brightness adjusted accordingly, confirming the driver's functionality for high-frequency signals in the song.

VI. PWM LED DRIVER

I designed the PWM LED driver to indicate low-frequency signal levels using a duty-cycle-based approach. A comparator modulates the LED by comparing the low-pass filter output with a triangle wave, generating a pulse-width-modulated signal that switches the LED on and off at a rate imperceptible to the human eye. The LED brightness reflects the low-frequency audio signal's amplitude.

A. Design and Analysis

Fig. 11 shows the PWM LED driver. A comparator sums the low-pass filter output (V_{lpf}) with a bias offset and compares it to a triangle wave.

I used bias resistors R_{14} and R_{15} to offset the voltage at the comparator's non-inverting input, calculated as:

$$V_+ = V_{lpf} + \frac{R_{15}}{R_{14} + R_{15}}(V_{CC} - V_{EE}) + V_{EE}.$$

Using a Python script similar to the analog driver's (see Appendix), I selected $R_{14} = 2.2 \text{ k}\Omega$ and $R_{15} = 3.3 \text{ k}\Omega$. As V_{lpf} increases, the comparator's output duty cycle grows, brightening the LED. I calculated the source resistor R_{20} using $V_{LED} = 3 \text{ V}$, $V_{EE} = -4.5 \text{ V}$, and $I_{max} = 30 \text{ mA}$:

$$R_{20} = \frac{V_{LED} - V_{EE}}{I_{max}} = 250 \Omega,$$

and chose $R_{20} = 270 \Omega$ from kit values.

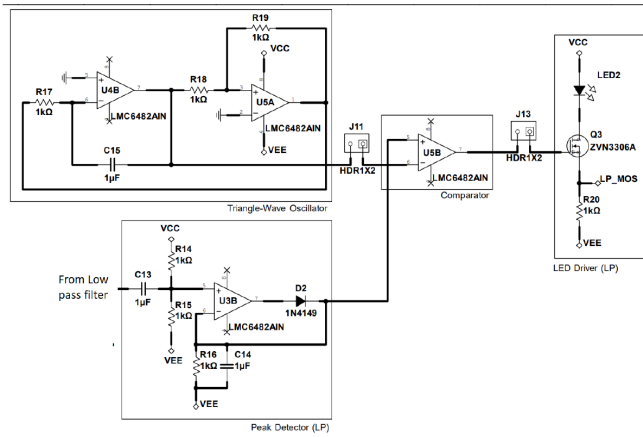


Fig. 11. I designed the PWM LED driver schematic.

B. Experimental Results

I tested the PWM LED driver by inputting a triangle wave into the comparator's non-inverting input and a 20 Hz AC wave. As I adjusted the AC signal's amplitude, the LED brightness changed correspondingly, confirming the driver's ability to control the LED for low-frequency content.

VII. SYSTEM-LEVEL DESIGN AND TESTING

I tested the complete system by inputting the song "Levels" by Avicii. The summing amplifier combines the stereo channels into a single monaural signal, removing DC offsets below 20 Hz. The low-pass and high-pass Sallen-Key filters then separate the signal into bass (below 320 Hz) and treble (above 640 Hz) components. Each filter's output feeds a peak detector, which captures the maximum amplitude over a 70.5 ms window. The low-frequency peak detector drives the PWM LED driver, modulating the LED's duty cycle to reflect bass intensity, while the high-frequency peak detector controls the analog LED driver, adjusting brightness based on treble amplitude.

When I played "Levels," the system responded dynamically. During bass-heavy sections, such as the song's iconic synth drops, the low-pass filter isolated frequencies around 320 Hz, causing the PWM-driven LED to brighten with increased duty cycle, pulsing in sync with the beat. In treble-heavy sections, like the vocal and melody-driven parts, the high-pass filter highlighted frequencies around 640 Hz, and the analog-driven LED glowed brighter, tracking the high-frequency energy. I observed smooth transitions between bass and treble dominance, with both LEDs responding in real time without flicker, thanks to the peak detectors' 70.5 ms hold time. Using the Analog Discovery 2, I verified that the filter outputs matched the expected spectral content, and the LED brightness levels aligned with the peak detector outputs, confirming the system's end-to-end functionality.

VIII. CONCLUSIONS AND THOUGHTS FOR FUTURE CLASSES

I combined theoretical design, simulation, and physical implementation to create an audio visualization system, deepening my understanding of analog filtering, op-amp configurations, peak detection, and LED driving circuits. I successfully built a system that distinguishes low and high audio frequencies in real time and dynamically drives LEDs to reflect a song's frequency components. I leveraged Python-based design tools, validated performance through Multisim simulations, and conducted hands-on testing with the Analog Discovery 2.

To future students: Thoroughly understand each sub-circuit to ensure the system functions as a whole. Test each component independently and use Multisim extensively before building on a breadboard or PCB. Simulations help identify and resolve issues quickly. Pay close attention to real component values—resistors can cause significant problems if not verified. Python proves invaluable for selecting optimal components in complex equations or iterations. Document your process as you progress to simplify writing the final report. This project enhanced my appreciation for analog design and demonstrated how computational tools and physical testing complement each other in practical circuit engineering.

APPENDIX

Listing 1. I wrote a brute-force resistor solver for gain tuning.

```
from pandas import *
resistor_values_in_lab_kit = [10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, 82,
                              100, 120, 150, 180, 220, 270, 330, 390, 470, 560, 680, 820,
                              1e3, 1.2e3, 1.5e3, 1.8e3, 2.2e3, 2.7e3, 3.3e3, 3.9e3, 4.7e3
                              , 5.6e3, 6.8e3, 8.2e3,
                              10e3, 12e3, 15e3, 18e3, 22e3, 27e3, 33e3, 39e3, 47e3, 56e3,
                              68e3, 82e3,
                              100e3, 120e3, 150e3, 180e3, 220e3, 270e3, 330e3, 390e3, 470
                              e3, 560e3, 680e3, 820e3,
                              1e6]

R10 = resistor_values_in_lab_kit
R11 = resistor_values_in_lab_kit

gain_desired = 1.76/9

rows = [] # create an empty list to store the rows of the dataframe

for r10 in R10: #loop through all possible values of R3
    for r11 in R11: #loop through all possible values of R4
        gain_actual = r11 / (r10 + r11)

        #We want the actual gain to be less than the desired gain
        if gain_actual < gain_desired:
            difference = gain_desired - gain_actual
            rows.append({'R10': r10, 'R11': r11, 'Gain': gain_actual, 'Difference':
                        difference})

# Create DataFrame from the list of rows
dframe = DataFrame(rows)

#sort the dataframe by the gain difference, smallest to largest
dframe = dframe.sort_values(by='Difference')
print(dframe)
```

I adapted the same code structure for R_{14} and R_{15} , modifying specific values as needed.

Listing 2. I developed an iterative solver for Sallen-Key filters.

```
import numpy as np
import itertools
import pandas as pd

resistor_values_in_lab_kit = [10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, 82,
                              100, 120, 150, 180, 220, 270, 330, 390, 470, 560, 680, 820,
                              1e3, 1.2e3, 1.5e3, 1.8e3, 2.2e3, 2.7e3, 3.3e3, 3.9e3, 4.7e3
                              , 5.6e3, 6.8e3, 8.2e3,
                              10e3, 12e3, 15e3, 18e3, 22e3, 27e3, 33e3, 39e3, 47e3, 56e3,
                              68e3, 82e3,
                              100e3, 120e3, 150e3, 180e3, 220e3, 270e3, 330e3, 390e3, 470
                              e3, 560e3, 680e3, 820e3,
                              1e6]
resistor_values_in_lab_kit = [r for r in resistor_values_in_lab_kit if r >= 10e3]

#1nF, 0.1nF, 0.68nF, 0.15uF, 6.8nF, 1u, .01u, .1u, .47u in uF
```

```

capaictor_values_in_lab_kit = [1e-9, 0.1e-9, 0.68e-9, 0.15e-6, 6.8e-9, 1e-6, 0.01e-6, 0.1
    e-6, 0.47e-6]

R_values = np.array(resistor_values_in_lab_kit)
C_values = np.array(capaictor_values_in_lab_kit)

target_Q = 0.707
min_Q = 0.5
K = 1

resultsl = []
resultsh = []

#Low-pass filter
target_fc_low = 320
for Rl1, Rl2 in itertools.product(R_values, repeat=2):
    for Cl1, Cl2 in itertools.product(C_values, repeat=2):

        m = np.sqrt(Rl1 * Rl2 * Cl1 * Cl2)
        fc = 1/(2 * np.pi * m)
        Q = m / (Rl1*Cl1 + Rl2*Cl1 + Rl1*Cl2*(1-K))
        if target_fc_low < fc < target_fc_low + 20 and target_Q -.1 < Q < target_Q + .1:
            resultsl.append((Rl1, Rl2, Cl1, Cl2, fc, Q))
resultsl = sorted(resultsl, key=lambda x: x[2])

#High-pass filter
target_fc_high = 640
for Rh1, Rh2 in itertools.product(R_values, repeat=2):
    for Ch1, Ch2 in itertools.product(C_values, repeat=2):

        m = np.sqrt(Rh1 * Rh2 * Ch1 * Ch2)
        fc = 1/(2 * np.pi * m)
        Q = m / (Rh2*Ch2 + Rh2*Ch1 + Rh1*Ch2*(1-K))
        if target_fc_high - 39 < fc < target_fc_high and target_Q -.1 < Q < target_Q +
            .1:
            resultsh.append((Rh1, Rh2, Ch1, Ch2, fc, Q))

resultsh = sorted(resultsh, key=lambda x: x[2])

# Create DataFrames for low-pass and high-pass filter results
dfl = pd.DataFrame(resultsl, columns=['R1 (Ohm)', 'R2 (Ohm)', 'C1', 'C2', 'fc (Hz)', 'Q'
    ])
dfh = pd.DataFrame(resultsh, columns=['R1 (Ohm)', 'R2 (Ohm)', 'C1', 'C2', 'fc (Hz)', 'Q'
    ])

# convert to and show as 1nF, 0.1nF, 0.68nF, 0.15uF, 6.8nF, 1u, .01u, .1u, or .47u where
    applicable, so the printed values
# are the same as the ones provided in the lab kit
dfl['C1'] = dfl['C1'].apply(lambda x: f"{x*1e6:.2f}uF" if x >= 1e-6 else f"{x*1e9:.2f}nF"
    )
dfl['C2'] = dfl['C2'].apply(lambda x: f"{x*1e6:.2f}uF" if x >= 1e-6 else f"{x*1e9:.2f}nF"
    )
dfh['C1'] = dfh['C1'].apply(lambda x: f"{x*1e6:.2f}uF" if x >= 1e-6 else f"{x*1e9:.2f}nF"
    )
dfh['C2'] = dfh['C2'].apply(lambda x: f"{x*1e6:.2f}uF" if x >= 1e-6 else f"{x*1e9:.2f}nF"
    )

#Convert to kOhm where applicable
dfl['R1 (Ohm)'] = dfl['R1 (Ohm)'].apply(lambda x: f"{x/1e3:.2f}kOhm" if x >= 1e3 else f"{
    x:.2f} Ohm")
dfl['R2 (Ohm)'] = dfl['R2 (Ohm)'].apply(lambda x: f"{x/1e3:.2f}kOhm" if x >= 1e3 else f"{
    x:.2f} Ohm")
dfh['R1 (Ohm)'] = dfh['R1 (Ohm)'].apply(lambda x: f"{x/1e3:.2f}kOhm" if x >= 1e3 else f"{
    x:.2f} Ohm")
dfh['R2 (Ohm)'] = dfh['R2 (Ohm)'].apply(lambda x: f"{x/1e3:.2f}kOhm" if x >= 1e3 else f"{
    x:.2f} Ohm")

```

```

print(df1.to_string(index=False))
print(dfh.to_string(index=False))

df1.to_html('low_pass_filter_results.html', index=False)
dfh.to_html('high_pass_filter_results.html', index=False)

```

Listing 3. I created a resistor solver for the summing amplifier sub-circuit.

```

from pandas import *
resistor_values_in_lab_kit = [10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, 82,
                              100, 120, 150, 180, 220, 270, 330, 390, 470, 560, 680, 820,
                              1e3, 1.2e3, 1.5e3, 1.8e3, 2.2e3, 2.7e3, 3.3e3, 3.9e3, 4.7e3
                              , 5.6e3, 6.8e3, 8.2e3,
                              10e3, 12e3, 15e3, 18e3, 22e3, 27e3, 33e3, 39e3, 47e3, 56e3,
                              68e3, 82e3,
                              100e3, 120e3, 150e3, 180e3, 220e3, 270e3, 330e3, 390e3, 470
                              e3, 560e3, 680e3, 820e3,
                              1e6]

R1 = 82e3 # units equal Ohms
R3 = resistor_values_in_lab_kit
R4 = resistor_values_in_lab_kit
R5 = resistor_values_in_lab_kit

# declared desired gain and tolerance
gain_desired = 3
tolerance = 5e-3

rows = [] # create an empty list to store the rows of the dataframe

for r3 in R3: #loop through all possible values of R2
    for r4 in R4: #loop through all possible values of R3
        for r5 in R5: #loop through all possible values of R4
            gain_actual = (r3/R1) + (r5*(1/R1 + r3/(R1*r4))) #calculate the actual gain

            if gain_desired - tolerance <= abs(gain_actual) <= gain_desired + tolerance:

# Create DataFrame from the list of rows
dframe = DataFrame(rows)

#sort the dataframe by the gain difference, smallest to largest
dframe = dframe.sort_values(by='Error')
print(dframe)
#grab the first row of the dataframe, which will be the best
print(dframe.head(1))

```