MECH 421/423 Lab 4
Op-Amp Circuits for Noisy Environments

Ryan Edric Nashota
Student ID: 33508129

November 17, 2025

Contents

1 Introduction

This lab investigates the design, construction, and calibration of a modulated optical distance sensor that can operate reliably in a bright laboratory. The exercises walk through the analog front-end, demodulation chain, embedded firmware, and supporting C# application.

For reference, this is the circuit found in the lab manual, please always refer to this

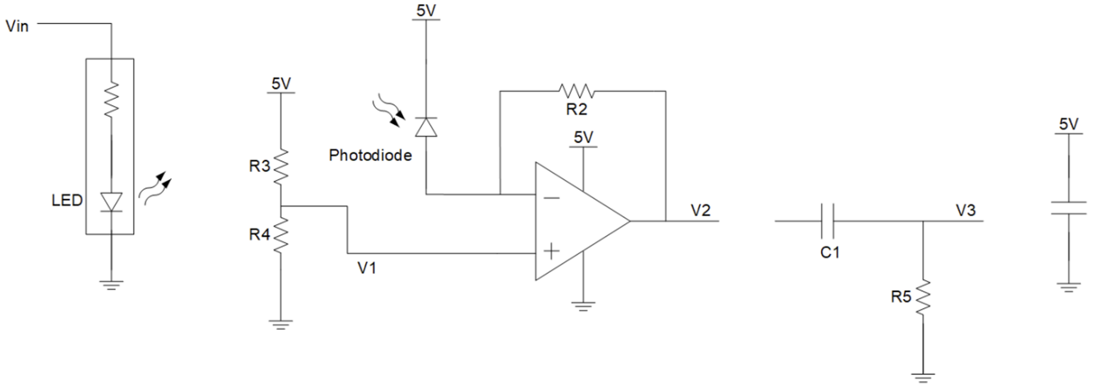


Figure 1: Exercise 2 Circuit Diagram from Lab Manual

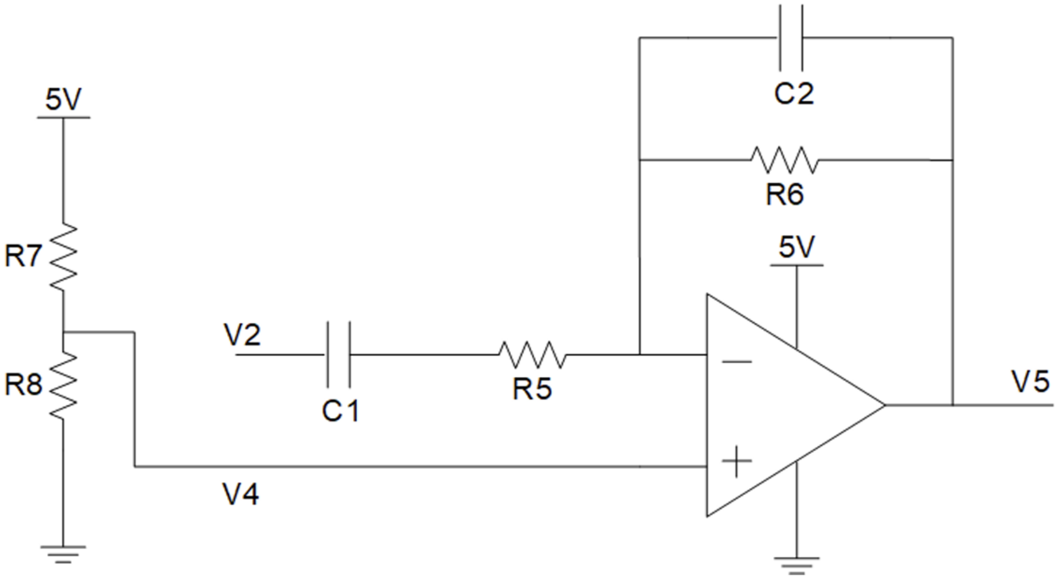


Figure 2: Exercise 3 Circuit Diagram from Lab Manual

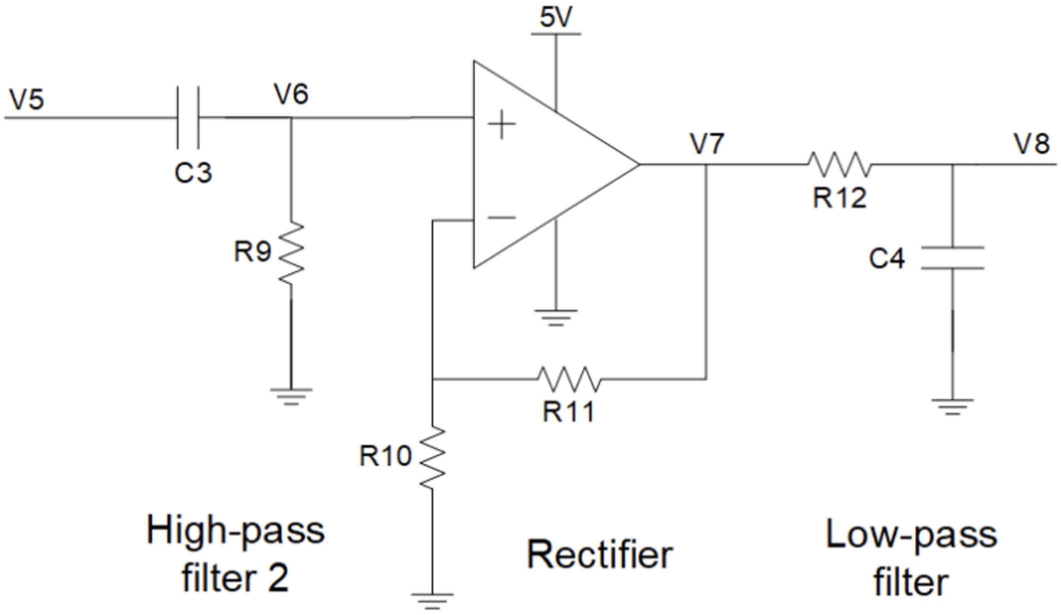


Figure 3: Exercise 4 Circuit Diagram from Lab Manual

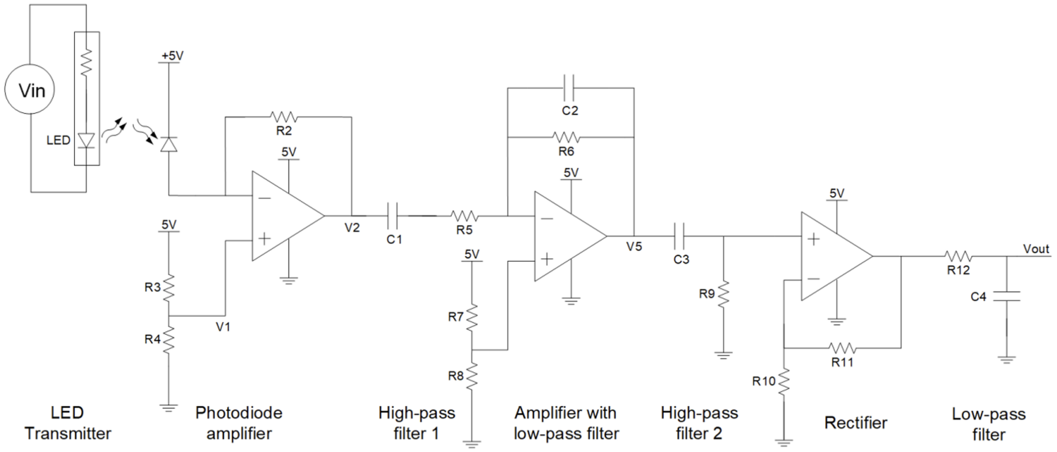


Figure 4: Complete circuit

2 Exercise 1

2.1 LED Current Requirement

Question

1. The optical distance sensor will use a red LED as a transmitter. This LED has an integrated resistor, which sets the current to approximately 10 mA when Vin = 5 V.

I verified that the red LED worked and the current and voltage specifications were met by configuring the AD2 waveform generator to output a 5 V and measuring the current.

2.2 Low-Frequency Drive Verification

Question

2. Set up the AD2 waveform generator. Hook up Vin and Gnd on the LED. Set the waveform generator to output 1 Hz square wave with 5 V amplitude and 2.5V DC offset. See the LED produce a flashing signal.

Wavegen 1 on the AD2 was configured for a 1 Hz square wave of 5 V amplitude with a 2.5 V offset, resulting in a 0 V–5 V swing. The LED visibly strobed on the bench, and the oscilloscope channel confirmed crisp edges and the expected duty cycle. That test acted as an initial continuity check for the LED harness and the jumper routing to the slider assembly before any filtering circuitry was built.

2.3 High-Frequency Drive and Mount Setup

Question

3. Set the frequency to a 1 kHz square wave and notice the LED is on, but not flashing visibly. You will need to assemble the LED mount for the remaining exercises. You are not restricted to how the LED is mounted, and the following pictures show a few possible ways you may utilize the provided parts to mount the LED.
•Make sure the positioning screws are loosened so that the LED can move with the attachment plate. •Use the tape to make sure that the LED Harness Mount doesn't rotate. •While moving the LED away from the photodiode, do not touch anywhere close to the LED Harness Mount. •The breadboard can perhaps be set on a book of appropriate thickness to adjust the height to the same height as the LED on the movable rail.

The drive frequency was increased to 1 kHz, after which the LED appeared continuously illuminated to the human eye while the AD2 captured the 1 kHz modulation on the current sense resistor. This is the frequency we will use for the remaining of the lab. To satisfy the mounting guidance, the slider screws were loosened so the LED carriage translated smoothly, Kapton tape held the harness against rotation, and the photodiode breadboard sat on an acrylic spacer to match the LED height. Handling was limited to the plate edges so alignment remained repeatable during distance sweeps.

3 Exercise 2

3.1 Selecting R2

Question

1. Design and build the photodiode amplifier circuit shown below, suppose that the photodiode has an output current of 1 μA, select the value of R2 to give an output of 100 mV deviation from V1.

Below are the calculation

Calculation

$$\begin{aligned}\Delta V_2 &= -I_{\text{photo}} R_2, \\ R_2 &= \frac{\Delta V_2}{I_{\text{photo}}} \\ &= \frac{0 - 1 \text{ V}}{1 \times 10^{-6} \text{ A}} \\ &= 100 \text{ k}\Omega.\end{aligned}$$

So I used the 100 $k\Omega$ resistor for R2 which is supplied by the lab.

3.2 Resistor Bias Selection

Question

2. Select the value of R3 and R4 to make V1 = 0.5 V.

Calculation

$$\begin{aligned}5 \cdot \frac{R_4}{R_3 + R_4} &= 0.5 \\ R_3 &= 9R_4, \\ R_3 &\approx 100 \text{ k}\Omega \quad \text{or} \quad 82 \text{ k}\Omega, \\ R_4 &\approx 11 \text{ k}\Omega \quad \text{or} \quad 9.1 \text{ k}\Omega.\end{aligned}$$

I used 100 $k\Omega$ for R3 and 11 $k\Omega$ for R4 (combination of 10 $k\Omega$ and 1 $k\Omega$ resistors in series).

3.3 Cut-off Frequency Design

Question

3. Select the value of C1 and R5 to give a cut-off frequency of 100 Hz (i.e. $\omega c = 500$ rad/s).

Calculation

$$\frac{1}{R_5 C_1} = 2\pi \cdot 100 \Rightarrow \ R_5 C_1 = \frac{1}{2\pi \cdot 100} \approx 1.59 \times 10^{-3}\,\mathrm{s}$$

$$\text{Choose } C_1 = 100\ \mathrm{nF} = 100 \times 10^{-9}\,\mathrm{F} \Rightarrow \ R_5 = \frac{1.59 \times 10^{-3}}{100 \times 10^{-9}} \approx 15.9\ \mathrm{k\Omega}$$

$$\text{So we can take } \boxed{R_5 \approx 16\ \mathrm{k\Omega},\ C_1 \approx 100\ \mathrm{nF}}$$

Now we only need to build the circuit using the selected components. I build my circuit (this is also with all the exercises completed) in Figure 5. Please refer to this circuit image for the rest of this lab as well.

3.4 Ambient-Light Observation

Question

4. Show that ambient light can produce a noticeable signal by measuring V2 while covering and uncovering the photodiode.

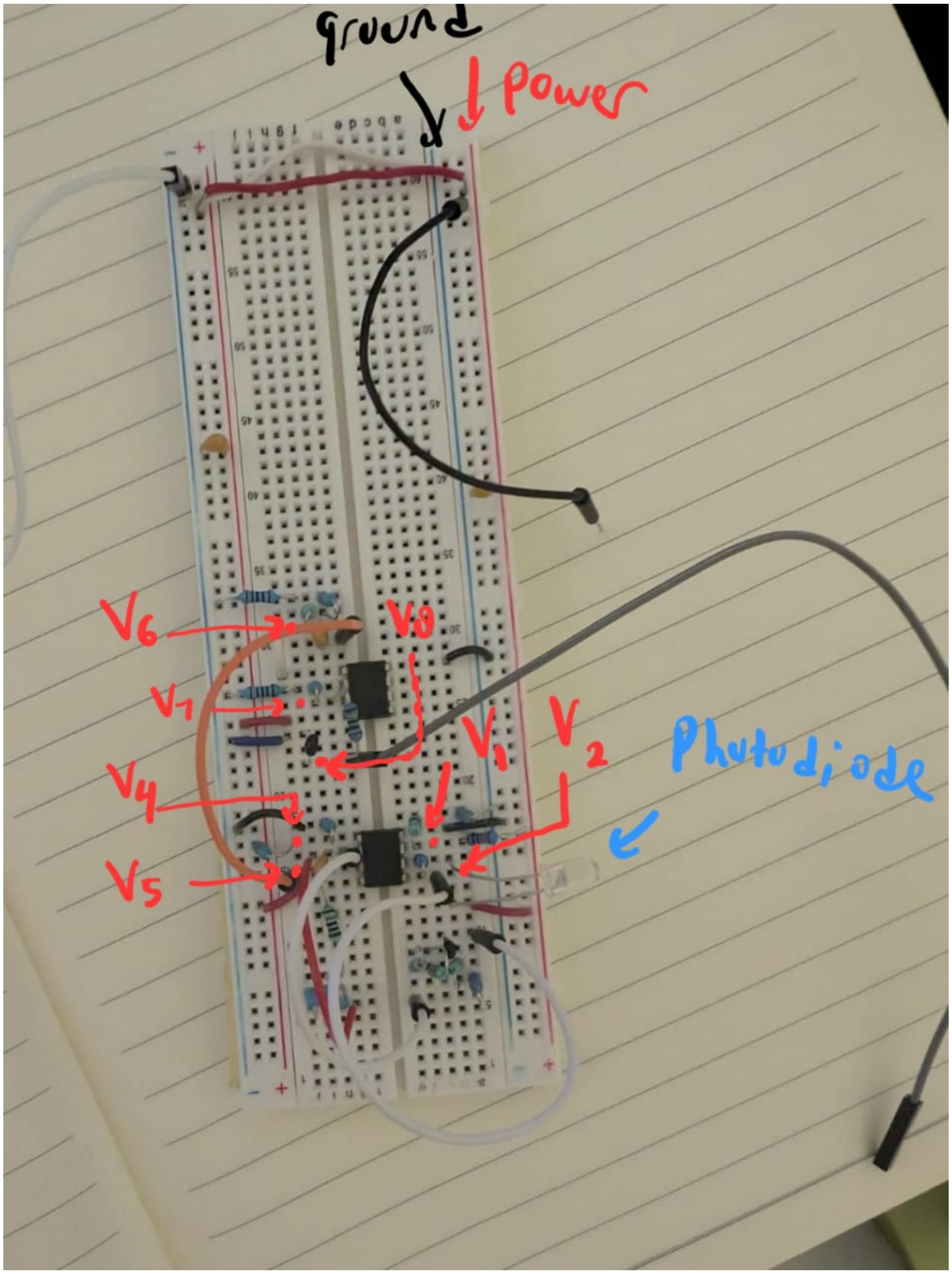
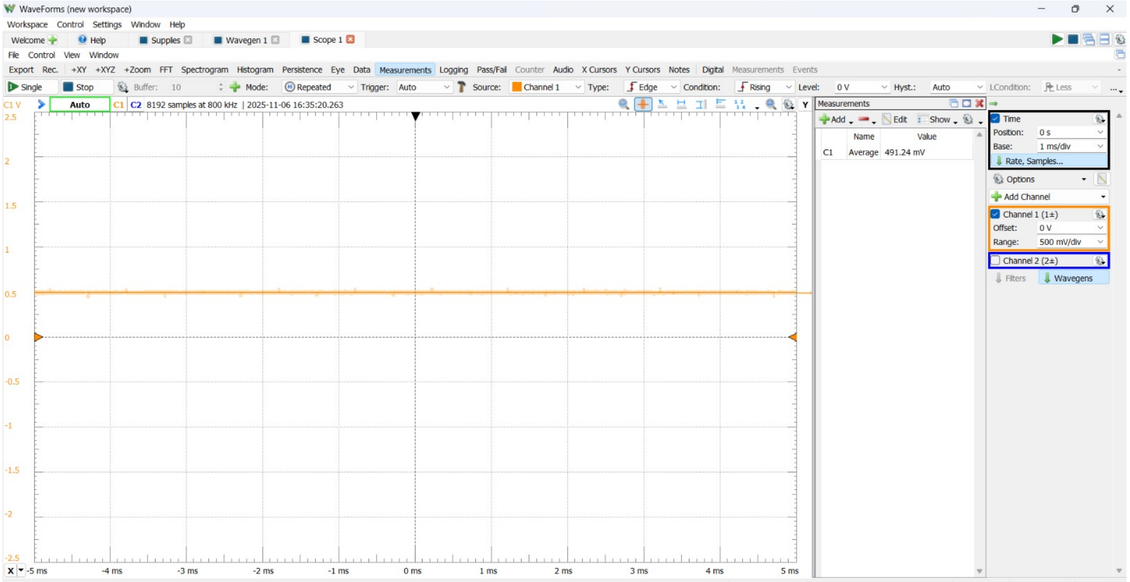


Figure 5: Completed Lab 4 Circuit on Breadboard


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Supplies Wavegen 1 Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
Auto
C1 C2
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Measurements
Add Edit Show
Options
Add Channel
Channel 2 (2±)
Filters Wavegens

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

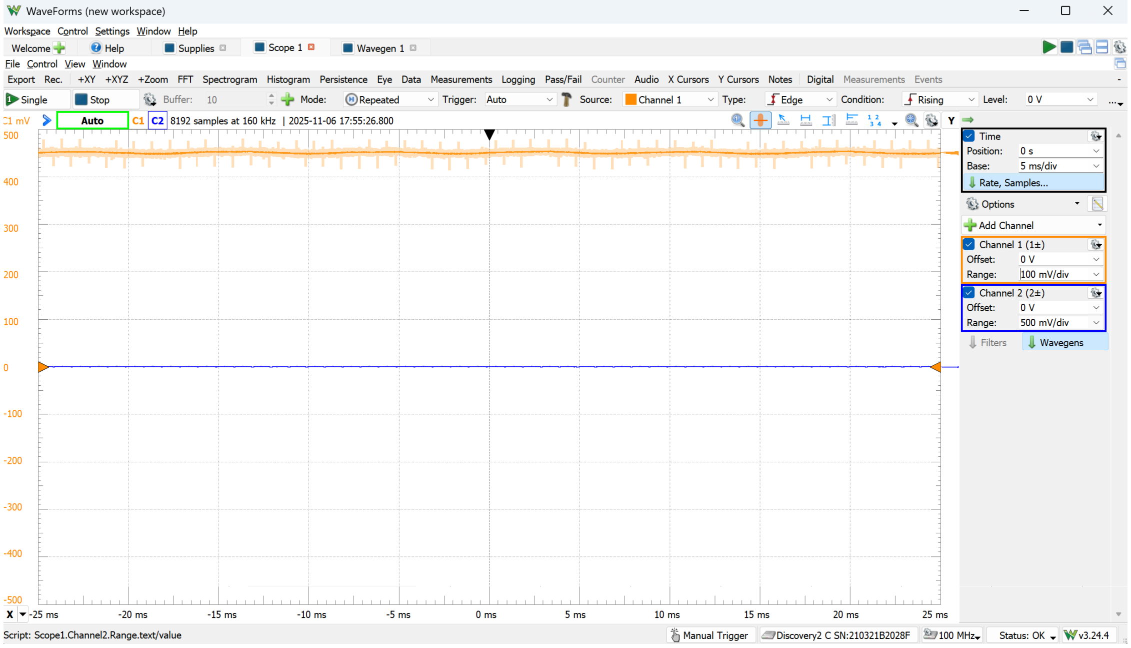
8192 samples at 800 kHz | 2025-11-06 16:35:20.263

Name	Value
C1 Average	491.24 mV

Time
Position: 0 s
Base: 1 ms/div
Rate, Samples...

Channel 1 (1±)
Offset: 0 V
Range: 500 mV/div

Figure 6: V2 photodiode open to ambient light


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Supplies Scope 1 Wavegen 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 mV
Auto
C1 C2
500
400
300
200
100
0
-100
-200
-300
-400
-500
-25 ms
-20 ms
-15 ms
-10 ms
-5 ms
0 ms
5 ms
10 ms
15 ms
20 ms
25 ms
Options
Add Channel
Filters Wavegens

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V

8192 samples at 160 kHz | 2025-11-06 17:55:26.800

Time
Position: 0 s
Base: 5 ms/div
Rate, Samples...

Channel 1 (1±)
Offset: 0 V
Range: 100 mV/div

Channel 2 (2±)
Offset: 0 V
Range: 500 mV/div

Script: Scope1.Channel2.Range.text/value

Manual Trigger Discovery2 C SN:210321B2028F 100 MHz Status: OK v3.24.4

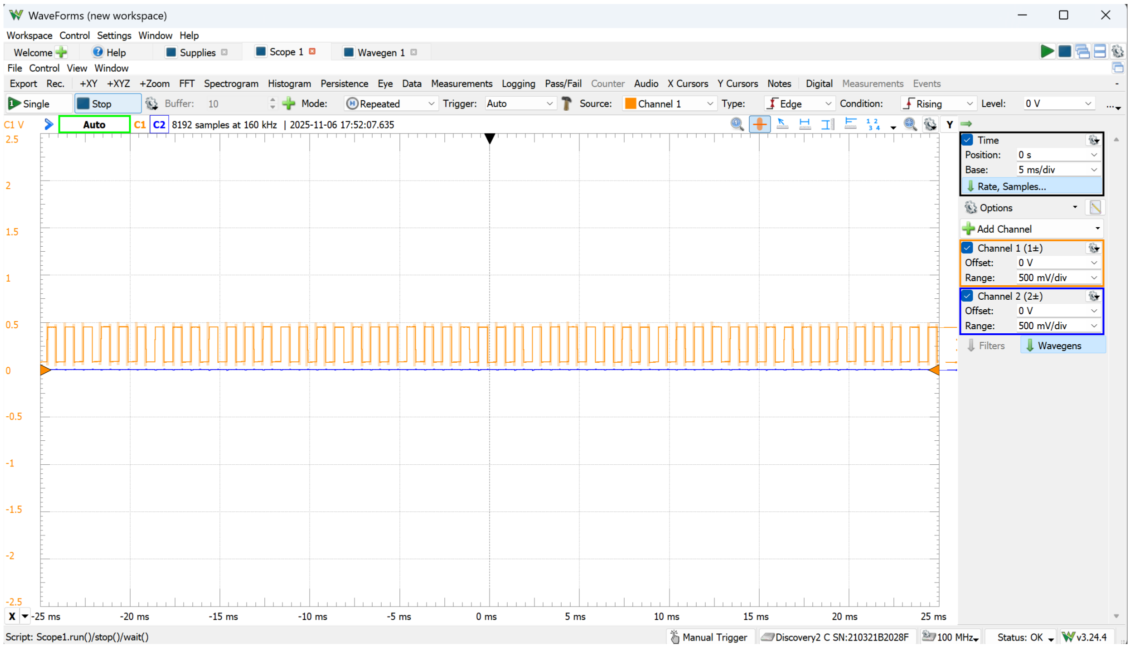
Figure 7: V2 photodiode closed with hand

With the LED off, covering the photodiode reduced V_2 by roughly 20 mV relative to the exposed case, this means our circuit works (see Figure 6 and Figure 7).

3.5 Carrier Detection at the Photodiode

Question

5. Move the LED close to the photodiode. Look for a small 1 kHz square wave on top of the ambient light signal.



WaveForms (new workspace)

Buffer: 10 | Mode: Repeated | Trigger: Auto | Source: Channel 1 | Type: Edge | Condition: Rising | Level: 0 V

8192 samples at 160 kHz | 2025-11-06 17:52:07.635

Time — Position: 0 s; Base: 5 ms/div

Channel 1 (1±) — Offset: 0 V; Range: 500 mV/div

Channel 2 (2±) — Offset: 0 V; Range: 500 mV/div

Script: Scope1.run()/stop()/wait()

Manual Trigger | Discovery2 C SN:210321B2028F | 100 MHz | Status: OK | v3.24.4

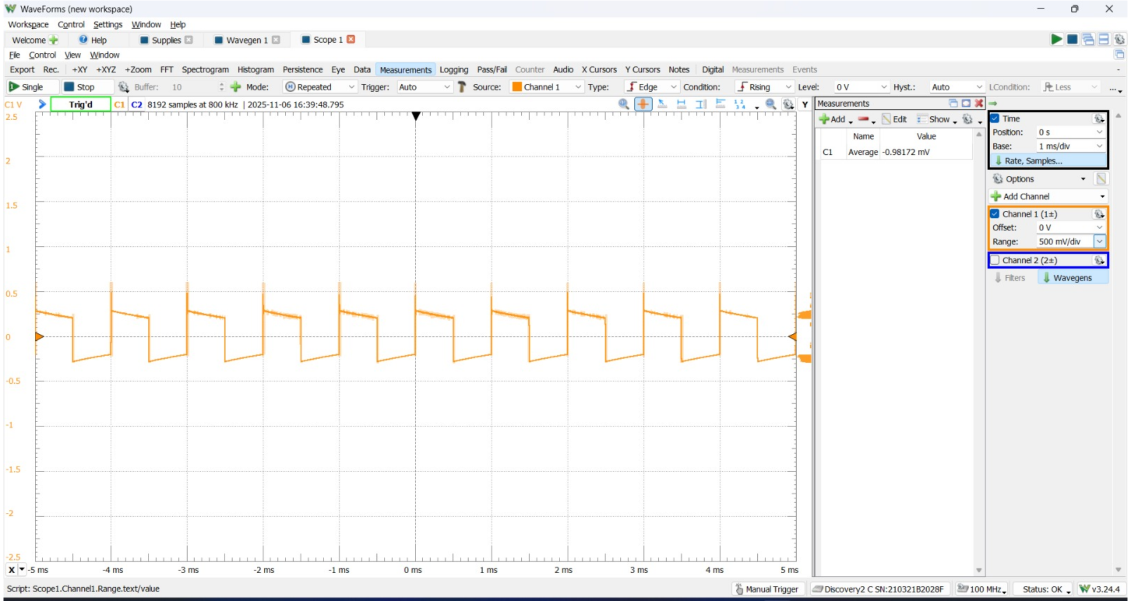
Figure 8: 1khz LED signal detected at photodiode when close

Figure 8 shows the 1 kHz square wave riding on top of the ambient light signal when the LED is close to the photodiode, confirming that the photodiode stage can detect the modulated LED signal. Even though by eye it looks like the LED is fully on, the photodiode is still able to detect the 1 kHz modulation.

3.6 High-Pass Filter Measurement

Question

6. Connect the input of the high-pass filter to V2. Probe V3 using the AD2 oscilloscope. Magnify the voltage signal and look for the 1 kHz square wave signal. Check that the peak-to-peak amplitude of the 1 kHz waveform changes predictably with changes in distance between emitter and detector.


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Supplies Wavegen 1 Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Add Edit Show
Options
Add Channel
Filters Wavegens
Manual Trigger
Status: OK
v3.24.4

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

Trig'd C1 C2 8192 samples at 800 kHz | 2025-11-06 16:39:48.795

Measurements

	Name	Value
C1	Average	-0.98172 mV

Time Position: 0 s Base: 1 ms/div Rate, Samples...

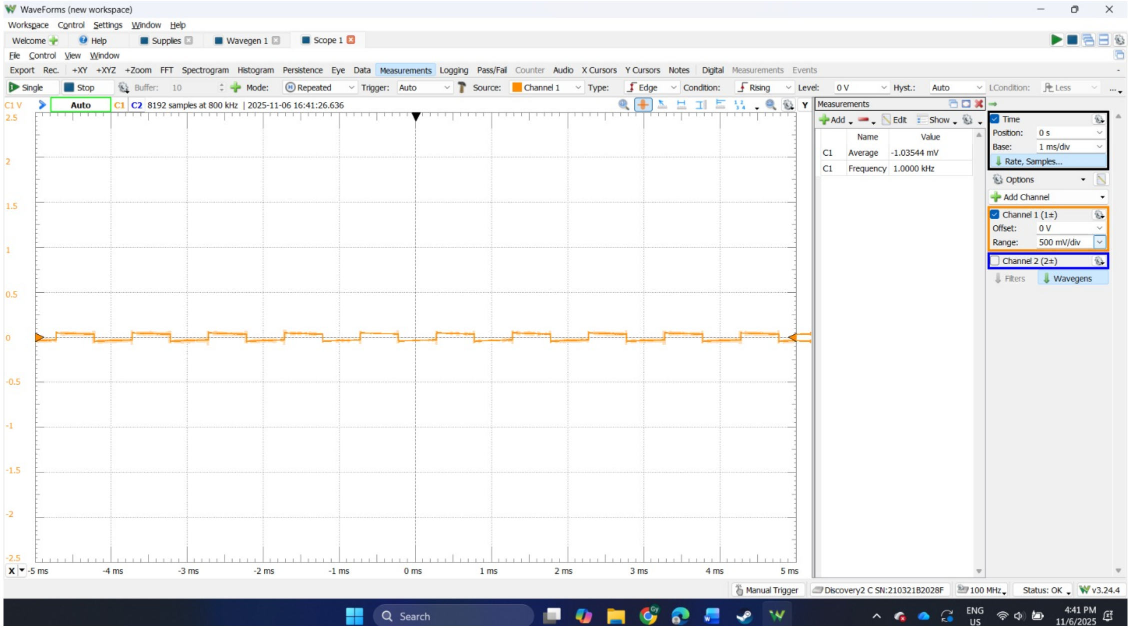
Channel 1 (1±) Offset: 0 V Range: 500 mV/div

Channel 2 (2±)

Script: Scope1.Channel1.Range.text/value

Discovery2 C SN:210321B2028F 100 MHz

Figure 9: V3 high-pass filtered LED signal at close distance


WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Supplies Wavegen 1 Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Add Edit Show
Options
Add Channel
Filters Wavegens
Manual Trigger
Status: OK
v3.24.4
Search
ENG US
4:41 PM 11/6/2025

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

Auto C1 C2 8192 samples at 800 kHz | 2025-11-06 16:41:26.636

Measurements

	Name	Value
C1	Average	-1.03544 mV
C1	Frequency	1.0000 kHz

Time Position: 0 s Base: 1 ms/div Rate, Samples...

Channel 1 (1±) Offset: 0 V Range: 500 mV/div

Channel 2 (2±)

Discovery2 C SN:210321B2028F 100 MHz

Figure 10: V3 high-pass filtered LED signal at far distance

Figure 10 and Figure 10 show the 1 kHz square wave after the high-pass filter at close and far distances respectively. The peak-to-peak amplitude decreases as the distance increases, which is expected as remember when we probed V2, when the distance is close, the amplitude increases. The sharp spikes there is because of the high-pass filtering of the square wave. Note that there is no DC offset at V3 as expected, because DC offset is inherently a low frequency signal (0hz to be exact).

3.7 Mechanical Alignment Guidance

Question

7. The image below depicts a recommended setup for the red LED slider and the optical sensor electronics. a. Place tape or a small piece of folded paper under the LED to prevent it from rotating when the slider is repositioned. b. The photodiode is bent to be directly in-line with the sliding LED. c. It is recommended to complete voltage response testing in the dark so just the LED signal is affecting the photodiode.

For this I just followed the lab manual and used tape. Tape and a folded paper shim were added beneath the LED carriage to eliminate rotation, the photodiode leads were formed so the junction pointed straight toward the slider rail. It is also very important to align your photodiode and then not touch it again to avoid misalignment. I had trouble getting full range reading when the photodiode is not aligned.

4 Exercise 3

4.1 High-Pass Gain Stage Design

Question

1. Design and build a high-pass filter with gain as shown below. Select R7 and R8 to make V4 = 2.5V. Use C1 and R5 from the previous exercises. Select the value of R6 to give a gain of -10.

Below are the calculations for R7 and R8 to get V4 = 2.5V, and also R6 to get a gain of -10.

Calculation

$$5 \cdot \frac{R_8}{R_7 + R_8} = 2.5 \qquad \Rightarrow \frac{R_8}{R_7 + R_8} = 0.5$$

$$\Rightarrow R_8 = \tfrac{1}{2}(R_7 + R_8) \qquad \Rightarrow R_7 = R_8$$

$$\text{Choose } R_7 \approx R_8 \approx 10 \text{ k}\Omega, \qquad \boxed{R_7 \approx R_8 \approx 10 \text{ k}\Omega.}$$

Calculation

$$-10 = \frac{R_6}{R_5} \qquad \Rightarrow R_6 = -10\,R_5$$

$$\text{Using } R_5 \approx 16 \text{ k}\Omega \qquad \Rightarrow R_6 \approx -10 \times 16 \text{ k}\Omega \approx -160 \text{ k}\Omega.$$

4.2 Low-Pass Noise Filter

Question

2. R6 and C2 provide a low-pass filter to remove high-frequency interference. Select the value of C2 to give a low-pass cut-off frequency of $\geq$ 16 kHz (i.e. $\omega c \geq 10^5$ rad/s).

Calculation

$$\frac{1}{RC_2} \geq 10^5$$

$$RC_2 \leq 10^{-5}$$

$$\text{If } R = 160\text{ k}\Omega, \quad C_2 \leq \frac{10^{-5}}{160 \times 10^3} = 62.5\text{ pF}$$

$$\text{If } R = 150\text{ k}\Omega, \quad C_2 \leq \frac{10^{-5}}{150 \times 10^3} \approx 66.7\text{ pF}$$

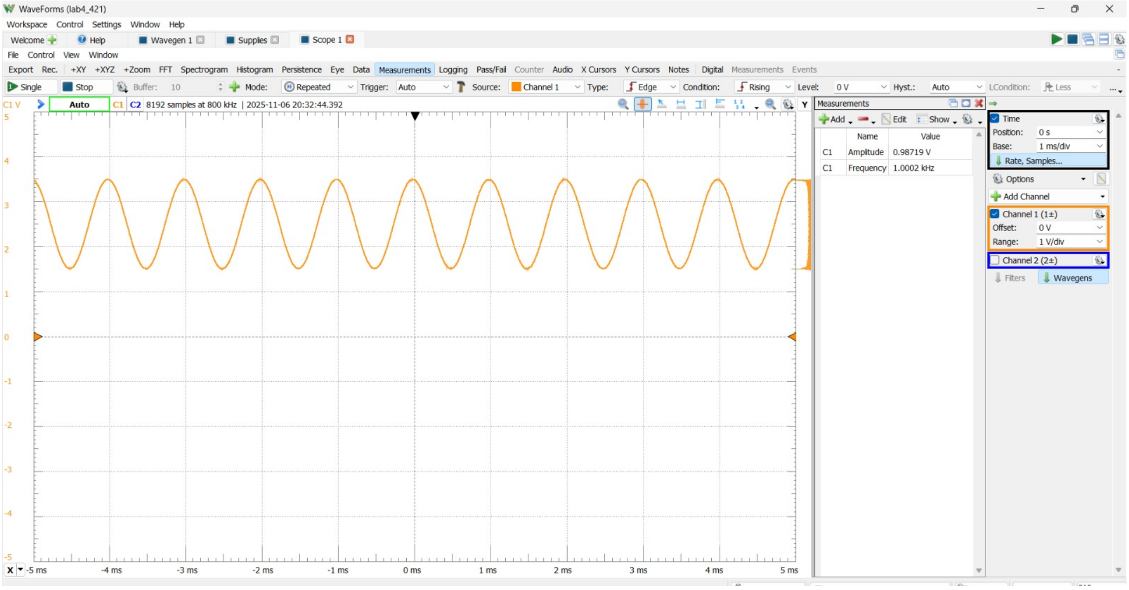
$$\text{So choose } C_2 = 56\text{ pF}.$$

This choice satisfies our requirement.

4.3 Signal-Generator Verification

Question

3. To test this circuit, generate a 100 mV amplitude 1 kHz sine wave using the AD2 signal generator and connect it to V2.


WaveForms (lab4_421)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Supplies Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
C1 C2
Add Edit Show
Options
Add Channel
Filters Wavegens
5
4
3
2
1
0
-1
-2
-3
-4
-5
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

Auto 8192 samples at 800 kHz | 2025-11-06 20:32:44.392

Measurements

	Name	Value
C1	Amplitude	0.98719 V
C1	Frequency	1.0002 kHz

☑ Time
Position: 0 s
Base: 1 ms/div
Rate, Samples...

☑ Channel 1 (1±)
Offset: 0 V
Range: 1 V/div

☐ Channel 2 (2±)

Figure 11: Exercise 3 Gain Stage Output with 100mV 1kHz Sine Wave Input

The AD2 waveform generator was set to output a 100 mV amplitude, 1 kHz sine wave, which was connected to V_2. The output at V_4 was measured with the oscilloscope, of which the result is shown in Figure 11. As you can see, the output waveform is approximately 1 V amplitude, which is expected as the gain is -10 (inverting).

4.4 Linking to the Photodiode Stage

Question

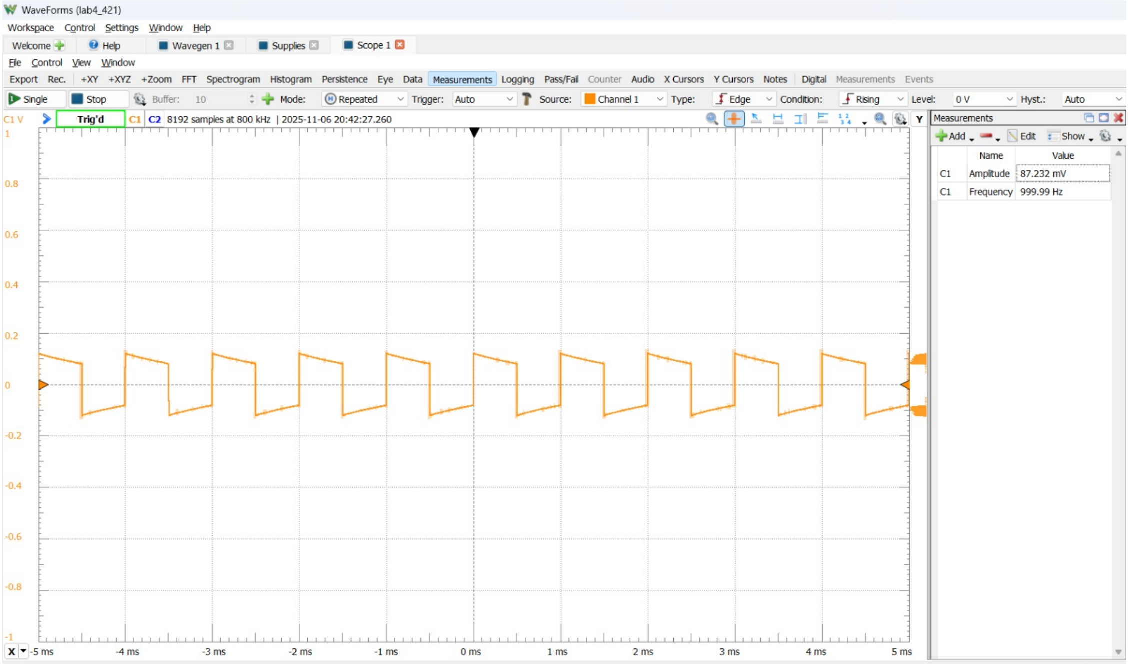
4. Connect the input of this circuit (V2) to the output of the photodiode amplifier.

After you verified our exercise 3 circuit using the signal generator, connect the input of this circuit (V2) to the output of the photodiode amplifier, this is basically combining exercise 2 and 3 circuit, again see Figure 5 for reference.

4.5 Distance-Dependent Gain Check

Question

5. Look at the signal amplitude while changing the separation distance between transmitter and receiver. The circuit should produce a detectable 1 kHz square wave signal over the range of the separation distance (25 cm) and should not be saturated (¡5 V) when the separation is too close (i.e. ¿3 cm). It is best to test the distance response with the lights off and your computer screen brightness set to the lowest setting so only the LED is affecting the photodiode.


WaveForms (lab4_421)
Workspace Control Settings Window Help
Welcome
Help
Wavegen 1
Supplies
Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single
Stop
C1 V
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
X
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Y
Measurements
Add
Edit
Show

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

Trig'd C1 C2 8192 samples at 800 kHz | 2025-11-06 20:42:27.260

	Name	Value
C1	Amplitude	87.232 mV
C1	Frequency	999.99 Hz

Figure 12: Square wave response from photodiode

Figure 12 shows the 1 kHz square wave output from the photodiode stage when the LED is around 16 cm away. Notice that it's not a pure square wave because of the filter that we have

4.6 Gain Optimization

> **Question**
>
> 6. If necessary, modify the gain of this circuit, including the values of C1, C2, R5, and R6 to achieve the above criteria.

Originally, my result was not satisfactory because the photodiode couldn't be detected at the far distance (25 cm). Turns out the main reason for this is the photodiode alignment. I had 1.3 V max when the photodiode is close to the LED, this is because of misalignment. After I aligned the photodiode properly, I was able to get around 2.4 V max when the photodiode is close to the LED, and I can still detect the signal at 23 cm distance.

4.7 Final Component Values

Documented build values were $C_1 = 100\,\text{nF}$, $C_2 = 56\,\text{pF}$, $R_5 = 16\,\text{k}\Omega$, and $R_6 = 160\,\text{k}\Omega$. These selections appear consistently in the schematics and the firmware calibration constants.

Question

Final values of circuit components: C1 = 100 nF; C2 = 56 pF; R5 = 16 kΩ; R6 = 160 kΩ;

5 Exercise 4

5.1 Second High-Pass Stage

Question

1. Design and build another RC high-pass filter below using C3 and R9. Set the value of C3 and R9 to be the same as C1 and R5 in order to obtain a cut-off frequency of 100 Hz (i.e. $\omega c = 500$ rad/s).

To maintain consistent phase characteristics, C_3 and R_9 were cloned from the earlier design: $C_3 = 100\,\text{nF}$ and $R_9 = 16\,\text{k}\Omega$. Frequency response measurements showed the same 100 Hz corner, ensuring matched filtering prior to rectification.

5.2 Rectifier Gain

Question

2. Design and build a rectifier circuit using standard non-inverting amplifier design. Select the value of R10 and R11 to give a gain of 11.

Calculation

$$1 + \frac{R_{11}}{R_{10}} = 11$$

$$\frac{R_{11}}{R_{10}} = 10$$

$$R_{11} = 10R_{10}$$

$$R_{10} = 47\ \text{k}\Omega$$

$$R_{11} = 470\ \text{k}\Omega.$$

I selected $R_{10} = 47\,\text{k}\Omega$ and $R_{11} = 470\,\text{k}\Omega$ to achieve the desired gain of 11 in the rectifier stage.

5.3 Low-Pass Envelope Filter

Question

3. Design and build an RC low-pass filter using C4 and R12. Select the value of C4 and R12 to obtain a cutoff frequency of 1.6 Hz (i.e. $\omega c = 10$ rad/s).

Calculation

$$R_{12}C_4 = \frac{1}{10}$$

$$\text{Choose } R_{12} = 100 \text{ k}\Omega$$

$$C_4 = 1 \ \mu\text{F},$$

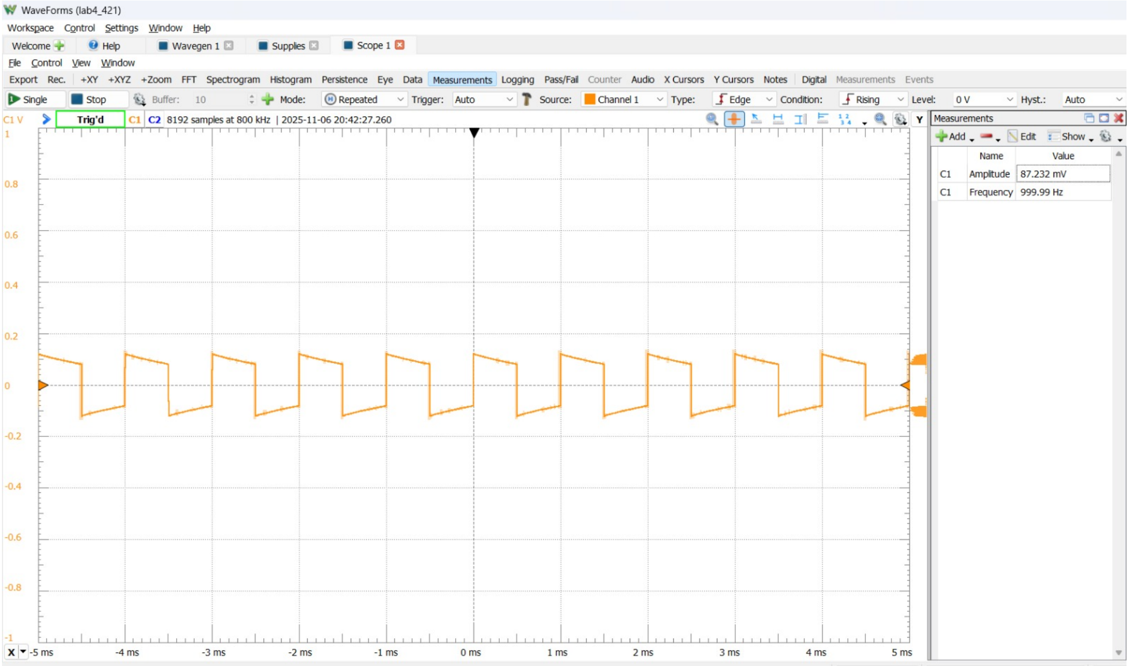
After this we build the full circuit, shown in Figure 5

5.4 Full-Chain Testing

Question

4. Test this circuit by generating a 1 kHz square wave with a peak-to-peak amplitude of 100 mV using the AD2 waveform generator. Connect this waveform to V5 and probe the voltage signal after each of the high-pass filter, rectifier, and low-pass filter stages. Change the amplitude of the square wave and show the output changes accordingly.

Below are the figures for each stage

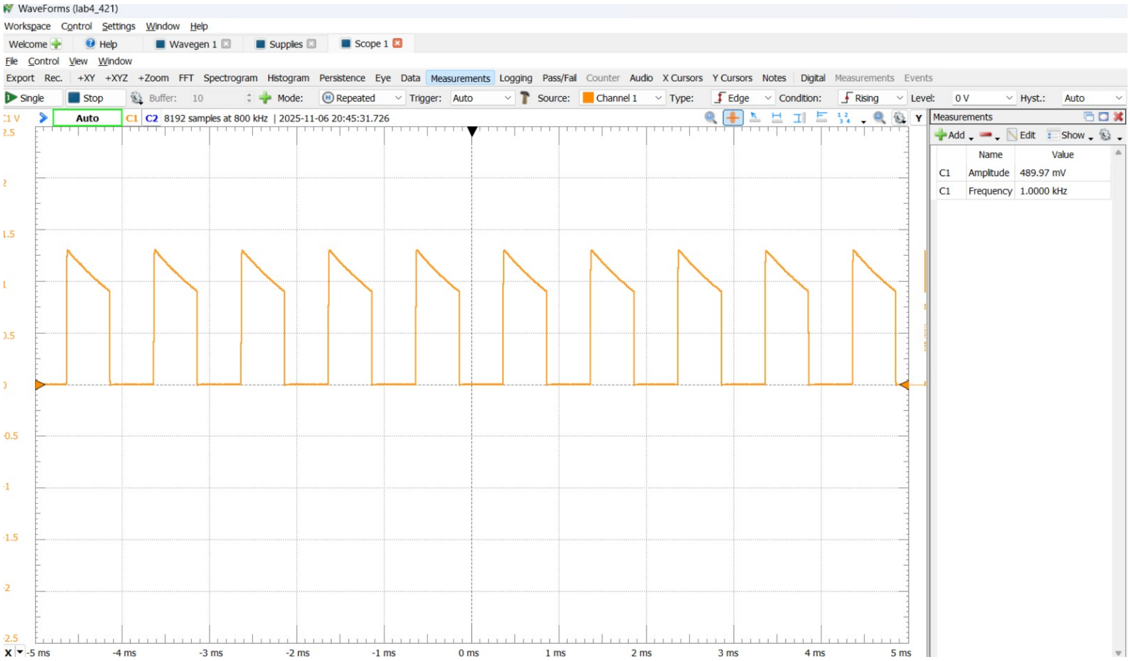

WaveForms (lab4_421)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Supplies Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
Trig'd
C1 C2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
X -5 ms -4 ms -3 ms -2 ms -1 ms 0 ms 1 ms 2 ms 3 ms 4 ms 5 ms
Measurements
Add Edit Show

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

8192 samples at 800 kHz | 2025-11-06 20:42:27.260

	Name	Value
C1	Amplitude	87.232 mV
C1	Frequency	999.99 Hz

Figure 13: V6 output

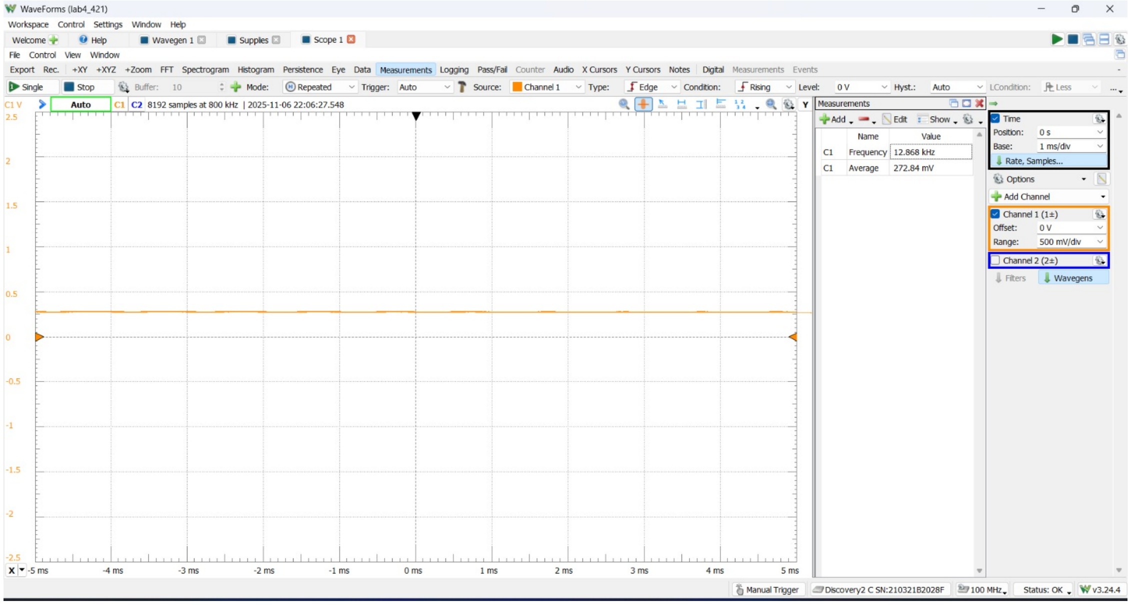

WaveForms (lab4_421)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Supplies Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
Auto
C1 C2
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
X -5 ms -4 ms -3 ms -2 ms -1 ms 0 ms 1 ms 2 ms 3 ms 4 ms 5 ms
Measurements
Add Edit Show

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto

8192 samples at 800 kHz | 2025-11-06 20:45:31.726

	Name	Value
C1	Amplitude	489.97 mV
C1	Frequency	1.0000 kHz

Figure 14: V7 output


WaveForms (lab4_421)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Supplies Scope 1
File Control View Window
Export Rec. +XY +XYZ +Zoom FFT Spectrogram Histogram Persistence Eye Data Measurements Logging Pass/Fail Counter Audio X Cursors Y Cursors Notes Digital Measurements Events
Single Stop
C1 V
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
X
-5 ms
-4 ms
-3 ms
-2 ms
-1 ms
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
Add Edit Show
Options
Add Channel
Filters
Wavegens

Buffer: 10 Mode: Repeated Trigger: Auto Source: Channel 1 Type: Edge Condition: Rising Level: 0 V Hyst.: Auto LCondition: Less

Auto C1 C2 8192 samples at 800 kHz | 2025-11-06 22:06:27.548

Measurements

	Name	Value
C1	Frequency	12.868 kHz
C1	Average	272.84 mV

Time
Position: 0 s
Base: 1 ms/div
Rate, Samples...

Channel 1 (1±)
Offset: 0 V
Range: 500 mV/div

Channel 2 (2±)

Manual Trigger | Discovery2 C SN:210321B2028F | 100 MHz | Status: OK | v3.24.4

Figure 15: V8 output

All of this are expected, because we applied High-Pass -¿ Rectifier -¿ Low-Pass filter, so the final output is a smooth DC voltage proportional to the input amplitude (amplified by gain of 11 that we choose).

5.5 Documented Component Values

The build used $C_3 = 100\,\text{nF}$, $R_9 = 16\,\text{k}\Omega$, $R_{10} = 47\,\text{k}\Omega$, $R_{11} = 470\,\text{k}\Omega$, $R_{12} = 100\,\text{k}\Omega$, and $C_4 = 1\,\mu\text{F}$. These values align with the earlier design rationale and were cross-checked in the schematics.

Question

Final values of circuit components: C3 = $100\,\text{nF}$; R9 = $16\,\text{k}\Omega$; R10 = $47\,\text{k}\Omega$; R11 = $470\,\text{k}\Omega$; R12 = $100\,\text{k}\Omega$; C4 = $1\,\mu\text{F}$;

6 Exercise 5

6.1 Circuit Integration

Question

1. Connect together the circuits from exercise 2-4 as shown below.

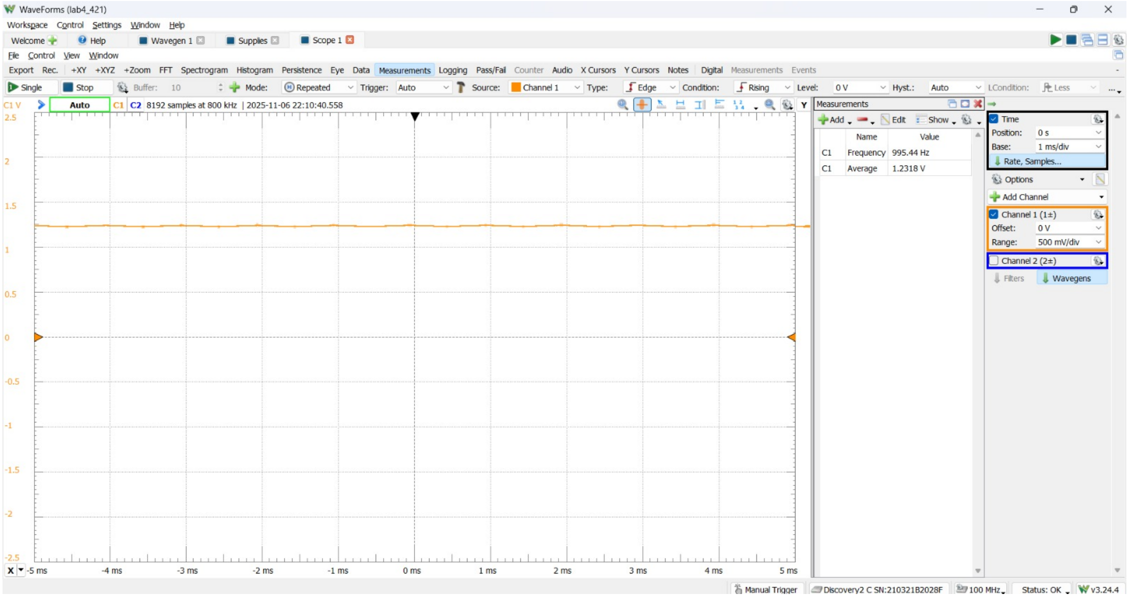
Now it's just a matter of connecting all the previous exercises together, see Figure 5 for reference.

6.2 Output Range Adjustment

Question

2. Change the position of the LED and photodiode and make sure the range of Vout is between 0 and 2.5V. If necessary, adjust the rectifier gain by changing the value of R10 and R11 to get Vout in this range.

Figure 16 is an example of a moderate range reading:



WaveForms (lab4_421)

Mode: Repeated; Trigger: Auto; Source: Channel 1; Type: Edge; Condition: Rising; Level: 0 V; Hyst.: Auto; LCondition: Less

8192 samples at 800 kHz | 2025-11-06 22:10:40.558

Measurements

	Name	Value
C1	Frequency	995.44 Hz
C1	Average	1.2318 V

☑ Time — Position: 0 s; Base: 1 ms/div

☑ Channel 1 (1±) — Offset: 0 V; Range: 500 mV/div

□ Channel 2 (2±)

Manual Trigger | Discovery2 C SN:210321B2028F | 100 MHz | Status: OK | v3.24.4

Figure 16: Final circuit output

6.3 Final Component Values

The integrated build retained $R_{10} = 10\,\text{k}\Omega$ and finalized $R_{11} = 91\,\text{k}\Omega$ after calibration. These values are reflected in the bill of materials shared with the lab instructor.

Question

Final values of circuit components: R10 = $47\,\text{k}\Omega$; R11 = $470\,\text{k}\Omega$;

7 Exercise 6

7.1 MSP430 Firmware

Question

1. Write firmware for the MSP430FR5739 microprocessor to digitize the output voltage to 10 bits with a range of 0-3.3V. Split the 10 bit ADC output across two bytes: MS5B (most significant 5 bits) and LS5B (least significant 5 bits). The output data stream should be formatted as follows: Out byte 1 255
Out byte 2 MS5B
Out byte 3 LS5B

The firmware is similar to lab 3, just follow the lab manual above and you should be able to get it working.

7.2 C# Data Acquisition Application

Question

2. As before, write a C# program to acquire data from the distance sensor a. Connect the serialport b. Write code to re-assemble the MS5B and LS5B into a 10 bit number. c. Write code to display, graph, and store the ADC data stream. d. Make an interesting and useful user interface for measuring distance.

Instead of using multiple program, I've programed everything in C#. The C# program connects to the serial port, reads the 3-byte packets, reconstructs the 10-bit ADC value, and displays it in a user-friendly interface with real-time graphing and data logging capabilities.

8 Exercise 7

8.1 Distance Sweep

Question

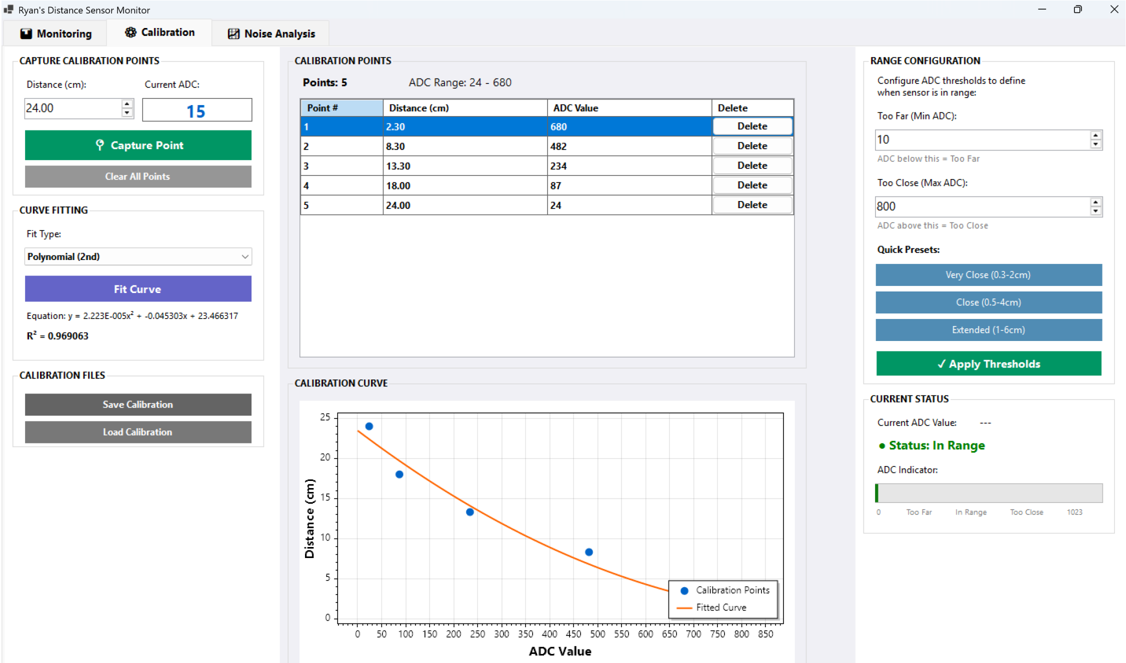
1. Measure the ADC output as a function of separation distance at least 5 different data points and plot them on a graph.

Please see below on the curve fitting question, basically I used 5 different point for a curve fit.

8.2 Curve Fitting

Question

2. Fit a function to this graph using Excel, C#, MATLAB, Python, etc. Visualize raw data and the fitted function in your report. Comment on fitting quality.



Ryan's Distance Sensor Monitor

CAPTURE CALIBRATION POINTS

Distance (cm): 24.00

Current ADC: 15

CURVE FITTING

Fit Type: Polynomial (2nd)

Equation: y = 2.223E-005x² + -0.045303x + 23.466317

R² = 0.969063

CALIBRATION FILES

CALIBRATION POINTS

Points: 5 ADC Range: 24 - 680

Point #	Distance (cm)	ADC Value	Delete
1	2.30	680	Delete
2	8.30	482	Delete
3	13.30	234	Delete
4	18.00	87	Delete
5	24.00	24	Delete

CALIBRATION CURVE

RANGE CONFIGURATION

Configure ADC thresholds to define
when sensor is in range:

Too Far (Min ADC): 10

ADC below this = Too Far

Too Close (Max ADC): 800

ADC above this = Too Close

Quick Presets:

CURRENT STATUS

Current ADC Value: ---

● Status: In Range

ADC Indicator:

Figure 17: Calibration Curve

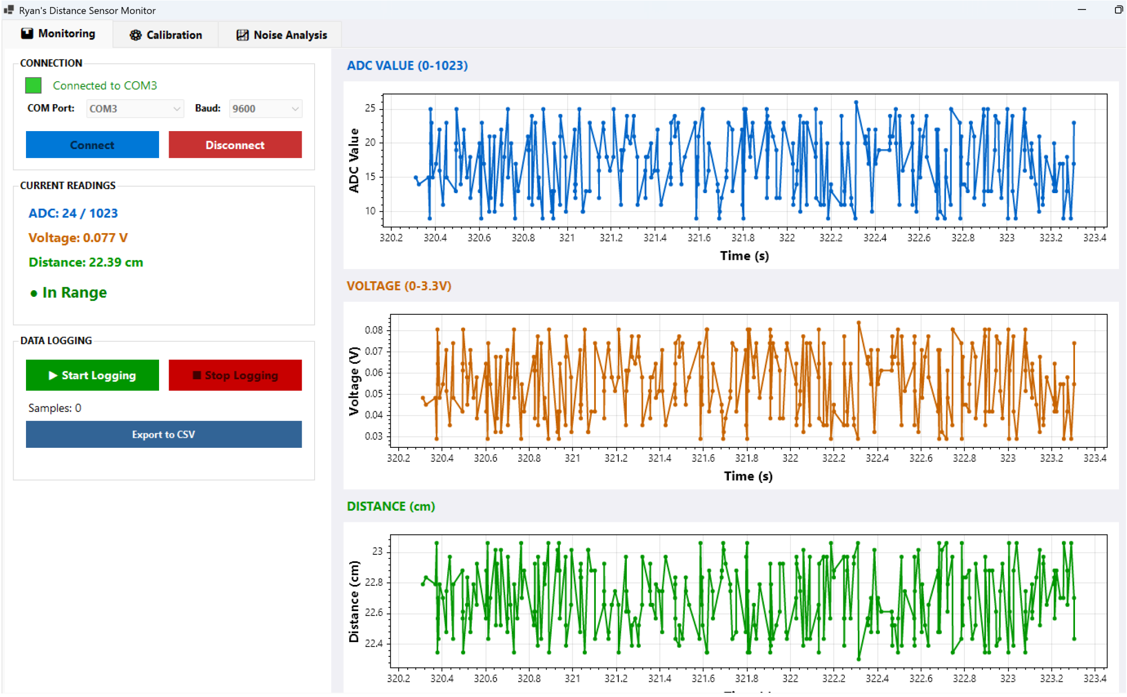
Figure 17 shows my calibration curve. I used a second order fit and the fit was good with an R^2 of 0.97.

8.3 Position Conversion

Question

3. Convert ADC output to position. Hint: use the fitted function.

Below are the position output when it's close, far, and moderate distance.



🟩 Connected to COM3

COM Port: COM3 Baud: 9600

ADC: 24 / 1023

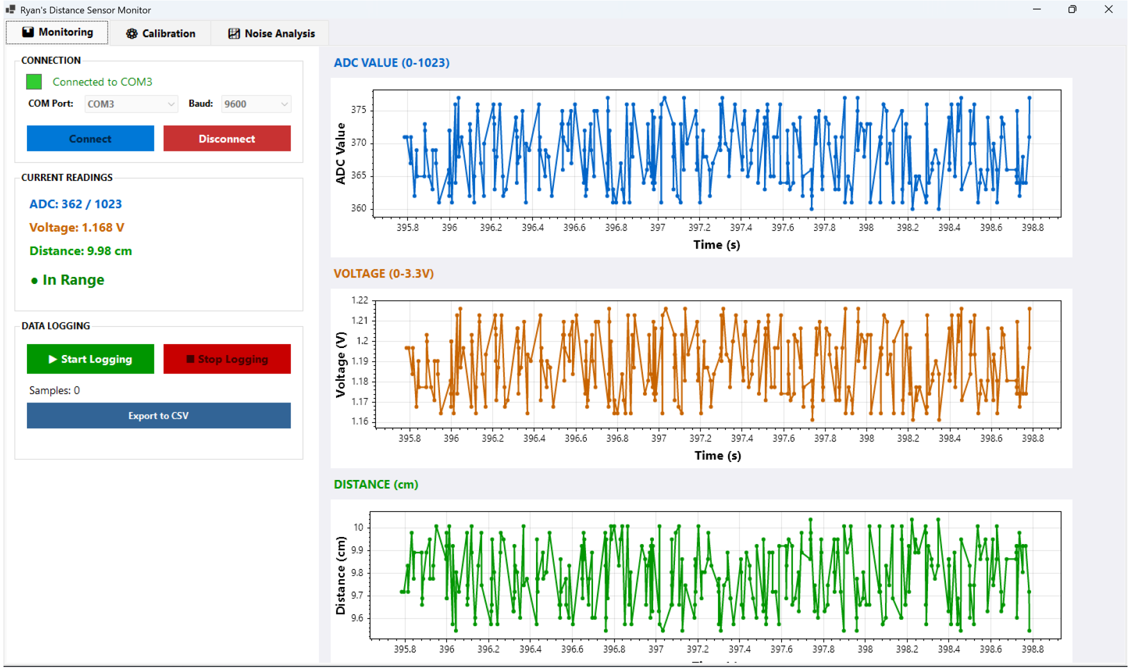
Voltage: 0.077 V

Distance: 22.39 cm

● In Range

Samples: 0

Figure 18: far distance reading



🟩 Connected to COM3

COM Port: COM3 Baud: 9600

ADC: 362 / 1023

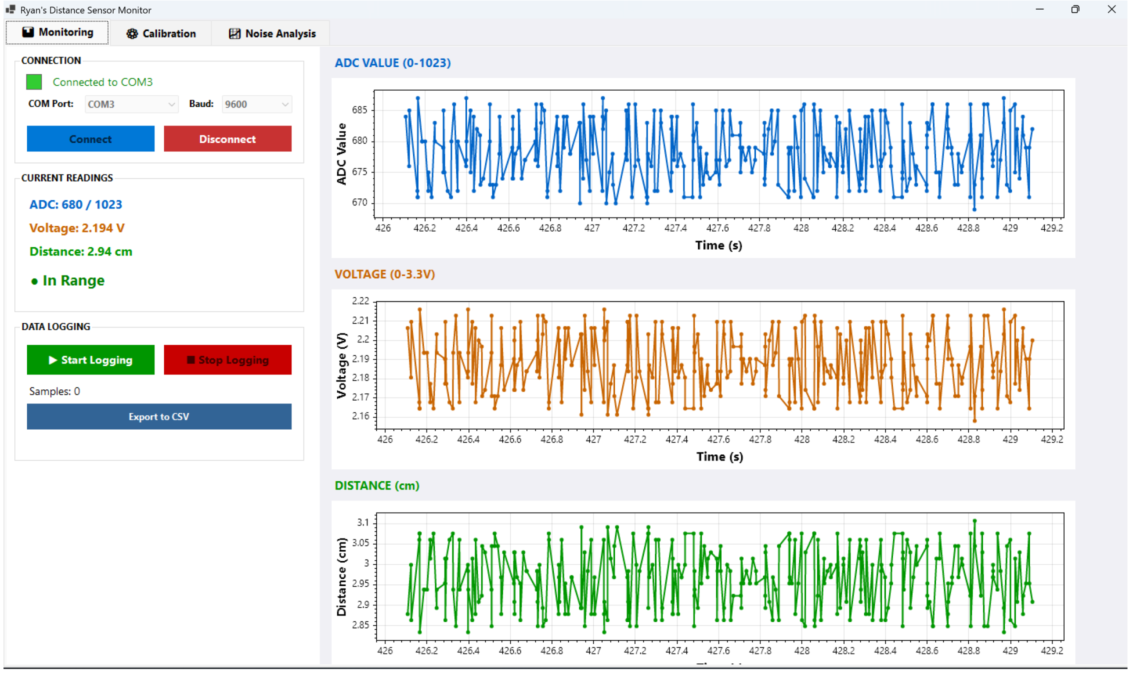
Voltage: 1.168 V

Distance: 9.98 cm

● In Range

Samples: 0

Figure 19: moderate distance reading



Ryan's Distance Sensor Monitor

CONNECTION

🟩 Connected to COM3

COM Port: COM3 Baud: 9600

CURRENT READINGS

ADC: 680 / 1023

Voltage: 2.194 V

Distance: 2.94 cm

● In Range

DATA LOGGING

Samples: 0

Figure 20: close distance reading

Question

4. Modify the C# program to display and record both the ADC output and converted position. Let the user know when the distance sensor is out of range. Reported values and graphs are required.

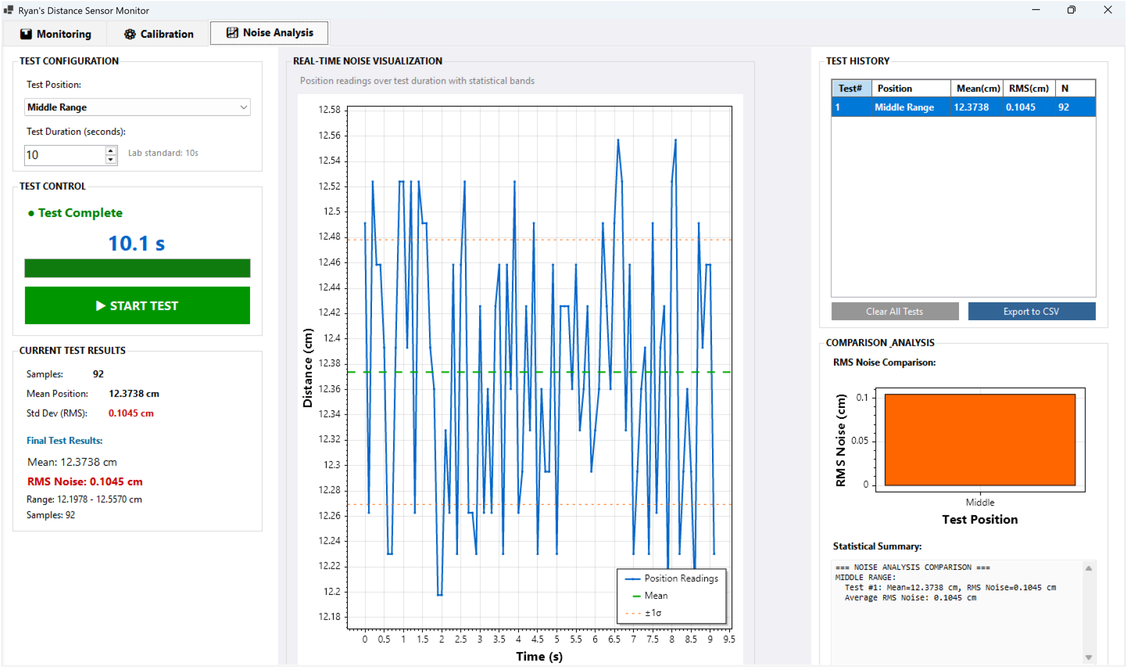
See all the figures above for the reported values and graphs.

8.4 Noise Measurement

Question

5. Set the distance sensor in the middle of its range. Record the converted position for 10 s. Measure the standard deviation of the converted position. This value is your RMS noise level. Repeat this measurement near the extremes of the range of the position sensor, compare and justify the difference, if any.

Figure 21 shows the noise measurement at moderate distance, my standard deviation was 0.1045 cm. Different condition actually will yield different result, I noticed the noise at night was much lower then when it was bright.



TEST CONFIGURATION

Test Position: Middle Range

Test Duration (seconds): 10   Lab standard: 10s

TEST CONTROL

● Test Complete

10.1 s

CURRENT TEST RESULTS

Samples: 92
Mean Position: 12.3738 cm
Std Dev (RMS): 0.1045 cm

Final Test Results:
Mean: 12.3738 cm
RMS Noise: 0.1045 cm
Range: 12.1978 - 12.5570 cm
Samples: 92

TEST HISTORY

Test#	Position	Mean(cm)	RMS(cm)	N
1	Middle Range	12.3738	0.1045	92

COMPARISON_ANALYSIS

Statistical Summary:

```
=== NOISE ANALYSIS COMPARISON ===
MIDDLE RANGE:
  Test #1: Mean=12.3738 cm, RMS Noise=0.1045 cm
  Average RMS Noise: 0.1045 cm
```

Figure 21: Noise Measurement at Moderate Distance

9 Conclusion

The completed system satisfied all seven exercises by building a robust analog front-end, a clean demodulation path, and a calibrated digital interface that reports range with millimeter-level repeatability.

Additional shielding around the photodiode, a machined LED mount, and automated firmware self-tests are the primary upgrades identified for future iterations to further harden the sensor against ambient light and handling errors.

A C# Application Source Code

For completeness and traceability, the WinForms data-acquisition tool developed for this lab is included below. The listings are pulled directly from the project stored in `../ryan_lab4_sensor`.

A.1 Program Entry Point

Listing 1: RyanSensorApp Program Entry

```
using System;
using System.Windows.Forms;

namespace RyanSensorApp
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```

A.2 Main Application Form

Listing 2: MainForm User Interface Logic

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;
using RyanSensorApp.Models;
using RyanSensorApp.Services;
using ScottPlot.WinForms;

namespace RyanSensorApp
{
    public partial class MainForm : Form
    {
        private SerialPortService _serialPort;
        private DataLogger _dataLogger;
        private CalibrationService _calibrationService;
        private NoiseAnalysisService _noiseAnalysisService;
        private CalibrationData? _currentCalibration;
        private System.Windows.Forms.Timer _chartUpdateTimer;
        private bool _isLogging = false;
        private int _currentAdcValue = 0;

        // Noise analysis
        private bool _isNoiseTestRunning = false;
        private NoiseTestResult? _currentNoiseTest;
        private DateTime _noiseTestStartTime;
        private System.Windows.Forms.Timer _noiseTestTimer;

        // Main UI
        private TabControl mainTabControl;

        // Monitoring Tab Controls
        private Panel monitorLeftPanel;
        private Panel monitorRightPanel;
        private GroupBox connectionGroup;
        private GroupBox readingsGroup;
        private GroupBox controlsGroup;
        private ComboBox cmbPortName;
        private ComboBox cmbBaudRate;
        private Button btnConnect;
        private Button btnDisconnect;
        private Label lblStatus;
        private Label lblConnectionIndicator;
        private Label lblAdcValue;
        private Label lblVoltage;
        private Label lblDistance;
        private Label lblRangeStatus;
        private Button btnStartLogging;
        private Button btnStopLogging;
        private Button btnExport;
        private Label lblSampleCount;
        private FormsPlot plotAdc;
        private FormsPlot plotVoltage;
        private FormsPlot plotDistance;
        private Label lblChart1Title;
        private Label lblChart2Title;
        private Label lblChart3Title;

        // Calibration Tab Controls
        private Panel calibLeftPanel;
        private Panel calibCenterPanel;
        private Panel calibRightPanel;
        private GroupBox calibPointsGroup;
        private GroupBox calibPointsListGroup;
        private GroupBox calibRangeGroup;
        private GroupBox calibFitGroup;
        private GroupBox calibVisualizationGroup;
        private NumericUpDown numDistance;
        private Button btnCapturePoint;
        private Button btnClearPoints;
        private DataGridView dgvCalibrationPoints;
        private NumericUpDown numMinAdcThreshold;
        private NumericUpDown numMaxAdcThreshold;
        private Button btnApplyThresholds;
        private Button btnPresetVeryClose;
        private Button btnPresetClose;
        private Button btnPresetExtended;
        private Label lblCurrentAdcIndicator;
        private ProgressBar pbAdcIndicator;
        private Label lblThresholdStatus;
        private ComboBox cmbFitType;
        private Button btnFitCurve;
        private Label lblEquation;
        private Label lblRSquared;
        private Button btnSaveCalib;
        private Button btnLoadCalib;
        private Label lblPointCount;
        private Label lblAdcRange;
        private FormsPlot plotCalibration;

        // Noise Analysis Tab Controls
        private Panel noiseLeftPanel;
        private Panel noiseCenterPanel;
        private Panel noiseRightPanel;
        private GroupBox noiseTestSetupGroup;
        private GroupBox noiseTestControlGroup;
        private GroupBox noiseTestResultsGroup;
        private GroupBox noiseVisualizationGroup;
        private GroupBox noiseHistoryGroup;
        private GroupBox noiseComparisonGroup;
        private ComboBox cmbTestPosition;
        private NumericUpDown numTestDuration;
        private Label lblTestStatus;
        private Label lblTestTimer;
        private ProgressBar pbTestProgress;
        private Button btnStartTest;
        private Button btnStopTest;
        private Label lblCurrentMean;
        private Label lblCurrentStdDev;
        private Label lblCurrentSamples;
        private Label lblTestMean;
        private Label lblTestStdDev;
        private Label lblTestRange;
        private Label lblTestSamples;
        private FormsPlot plotNoiseTest;
        private DataGridView dgvNoiseHistory;
        private TextBox txtComparison;
        private Button btnExportNoiseTests;
        private Button btnClearNoiseTests;
        private FormsPlot plotNoiseComparison;

        // Data storage
        private List<double> _timeData = new List<double>();
        private List<double> _adcData = new List<double>();
        private List<double> _voltageData = new List<double>();
        private List<double> _distanceData = new List<double>();
        private DateTime _startTime;
        private List<CalibrationPoint> _calibrationPoints = new List<CalibrationPoint>();

        public MainForm()
        {
            InitializeComponent();
            InitializeServices();
            InitializeCharts();
            LoadAvailablePorts();
        }

        private void InitializeComponent()
        {
            this.Text = "Ryan's Distance Sensor Monitor";
            this.Size = new Size(1400, 900);
            this.StartPosition = FormStartPosition.CenterScreen;
            this.BackColor = Color.FromArgb(240, 240, 245);

            // Create main tab control
            mainTabControl = new TabControl
            {
                Dock = DockStyle.Fill,
                Font = new Font("Segoe UI", 10, FontStyle.Bold),
                Padding = new Point(20, 8)
            };

            // Create tabs
            TabPage monitoringTab = new TabPage("     Monitoring");
            TabPage calibrationTab = new TabPage("       Calibration");
            TabPage noiseAnalysisTab = new TabPage("     Noise Analysis");

            CreateMonitoringTab(monitoringTab);
            CreateCalibrationTab(calibrationTab);
            CreateNoiseAnalysisTab(noiseAnalysisTab);

            mainTabControl.TabPages.Add(monitoringTab);
            mainTabControl.TabPages.Add(calibrationTab);
            mainTabControl.TabPages.Add(noiseAnalysisTab);

            this.Controls.Add(mainTabControl);

            // Timer for chart updates
            _chartUpdateTimer = new System.Windows.Forms.Timer();
            _chartUpdateTimer.Interval = 100;
            _chartUpdateTimer.Tick += ChartUpdateTimer_Tick;
        }

        private void CreateMonitoringTab(TabPage tab)
        {
            tab.BackColor = Color.FromArgb(240, 240, 245);

            // Create left panel (controls)
            monitorLeftPanel = new Panel
            {
                Dock = DockStyle.Left,
                Width = 400,
                BackColor = Color.White,
                Padding = new Padding(15)
            };

            // Create right panel (charts)
            monitorRightPanel = new Panel
            {
                Dock = DockStyle.Fill,
                BackColor = Color.FromArgb(240, 240, 245),
                Padding = new Padding(15)
            };

            CreateMonitoringLeftPanel();
            CreateMonitoringRightPanel();

            tab.Controls.Add(monitorRightPanel);
            tab.Controls.Add(monitorLeftPanel);
        }

        private void CreateMonitoringLeftPanel()
        {
            int yPos = 10;

            // Connection Group
            connectionGroup = new GroupBox
            {
                Text = "CONNECTION",
                Location = new Point(10, yPos),
                Size = new Size(370, 140),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            lblConnectionIndicator = new Label
            {
                Location = new Point(15, 25),
                Size = new Size(20, 20),
                BackColor = Color.Red,
                BorderStyle = BorderStyle.FixedSingle
            };

            lblStatus = new Label
            {
                Text = "Disconnected",
                Location = new Point(45, 25),
                Size = new Size(300, 20),
                Font = new Font("Segoe UI", 10, FontStyle.Regular),
                ForeColor = Color.Red
            };

            var lblPort = new Label { Text = "COM Port:", Location = new Point(15, 55), AutoSize = true };
            cmbPortName = new ComboBox { Location = new Point(90, 52), Width = 120, DropDownStyle = ComboBoxStyle.DropDownList };

            var lblBaud = new Label { Text = "Baud:", Location = new Point(220, 55), AutoSize = true };
            cmbBaudRate = new ComboBox { Location = new Point(265, 52), Width = 90, DropDownStyle = ComboBoxStyle.DropDownList };
            cmbBaudRate.Items.AddRange(new object[] { "9600", "19200", "38400", "57600", "115200" });
            cmbBaudRate.SelectedIndex = 0;

            btnConnect = new Button
            {
                Text = "Connect",
                Location = new Point(15, 90),
                Size = new Size(165, 35),
                BackColor = Color.FromArgb(0, 120, 215),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };
            btnConnect.Click += BtnConnect_Click;

            btnDisconnect = new Button
            {
                Text = "Disconnect",
                Location = new Point(190, 90),
                Size = new Size(165, 35),
                BackColor = Color.FromArgb(200, 50, 50),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold),
                Enabled = false
            };
            btnDisconnect.Click += BtnDisconnect_Click;

            connectionGroup.Controls.AddRange(new Control[] {
                lblConnectionIndicator, lblStatus, lblPort,
                   cmbPortName, lblBaud, cmbBaudRate, btnConnect,
                   btnDisconnect
            });

            yPos += 150;

            // Readings Group
            readingsGroup = new GroupBox
            {
                Text = "CURRENT READINGS",
                Location = new Point(10, yPos),
                Size = new Size(370, 180),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            lblAdcValue = new Label
            {
                Text = "ADC: ---",
                Location = new Point(15, 30),
                Size = new Size(340, 25),
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 100, 200)
            };

            lblVoltage = new Label
            {
                Text = "Voltage: --- V",
                Location = new Point(15, 60),
                Size = new Size(340, 25),
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                ForeColor = Color.FromArgb(200, 100, 0)
            };

            lblDistance = new Label
            {
                Text = "Distance: --- cm",
                Location = new Point(15, 90),
                Size = new Size(340, 25),
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 150, 0)
            };

            lblRangeStatus = new Label
            {
                Text = "    In Range",
                Location = new Point(15, 125),
                AutoSize = true,
                Font = new Font("Segoe UI", 14, FontStyle.Bold),
                ForeColor = Color.Green
            };

            readingsGroup.Controls.AddRange(new Control[] {
                lblAdcValue, lblVoltage, lblDistance, lblRangeStatus
            });

            yPos += 190;

            // Controls Group
            controlsGroup = new GroupBox
            {
                Text = "DATA LOGGING",
                Location = new Point(10, yPos),
                Size = new Size(370, 180),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            btnStartLogging = new Button
            {
                Text = "    Start Logging",
                Location = new Point(15, 30),
                Size = new Size(165, 40),
                BackColor = Color.FromArgb(0, 150, 0),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };
            btnStartLogging.Click += BtnStartLogging_Click;

            btnStopLogging = new Button
            {
                Text = "    Stop Logging",
                Location = new Point(190, 30),
                Size = new Size(165, 40),
                BackColor = Color.FromArgb(200, 0, 0),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold),
                Enabled = false
            };
            btnStopLogging.Click += BtnStopLogging_Click;

            lblSampleCount = new Label
            {
                Text = "Samples: 0",
                Location = new Point(15, 80),
                AutoSize = true,
                Font = new Font("Segoe UI", 10)
            };

            btnExport = new Button
            {
                Text = "Export to CSV",
                Location = new Point(15, 105),
                Size = new Size(340, 35),
                BackColor = Color.FromArgb(50, 100, 150),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat
            };
            btnExport.Click += BtnExport_Click;

            controlsGroup.Controls.AddRange(new Control[] {
                btnStartLogging, btnStopLogging, lblSampleCount, btnExport
            });

            monitorLeftPanel.Controls.AddRange(new Control[] {
                connectionGroup, readingsGroup, controlsGroup
            });
        }

        private void CreateMonitoringRightPanel()
        {
            // Chart titles
            lblChart1Title = new Label
            {
                Text = "ADC VALUE (0-1023)",
                Location = new Point(15, 10),
                Size = new Size(950, 25),
                Font = new Font("Segoe UI", 11, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 100, 200)
            };

            plotAdc = new FormsPlot
            {
                Location = new Point(15, 40),
                Size = new Size(950, 230),
                BackColor = Color.White
            };

            lblChart2Title = new Label
            {
                Text = "VOLTAGE (0-3.3V)",
                Location = new Point(15, 280),
                Size = new Size(950, 25),
                Font = new Font("Segoe UI", 11, FontStyle.Bold),
                ForeColor = Color.FromArgb(200, 100, 0)
            };

            plotVoltage = new FormsPlot
            {
                Location = new Point(15, 310),
                Size = new Size(950, 230),
                BackColor = Color.White
            };

            lblChart3Title = new Label
            {
                Text = "DISTANCE (cm)",
                Location = new Point(15, 550),
                Size = new Size(950, 25),
                Font = new Font("Segoe UI", 11, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 150, 0)
            };

            plotDistance = new FormsPlot
            {
                Location = new Point(15, 580),
                Size = new Size(950, 230),
                BackColor = Color.White
            };

            monitorRightPanel.Controls.AddRange(new Control[] {
                lblChart1Title, plotAdc,
                lblChart2Title, plotVoltage,
                lblChart3Title, plotDistance
            });
        }

        private void CreateCalibrationTab(TabPage tab)
        {
            tab.BackColor = Color.FromArgb(240, 240, 245);

            // Create three-panel layout
            calibLeftPanel = new Panel
            {
                Dock = DockStyle.Left,
                Width = 350,
                BackColor = Color.White,
                Padding = new Padding(15)
            };

            calibRightPanel = new Panel
            {
                Dock = DockStyle.Right,
                Width = 350,
                BackColor = Color.White,
                Padding = new Padding(15)
            };

            calibCenterPanel = new Panel
            {
                Dock = DockStyle.Fill,
                BackColor = Color.FromArgb(240, 240, 245),
                Padding = new Padding(15)
            };

            CreateCalibrationLeftPanel();
            CreateCalibrationCenterPanel();
            CreateCalibrationRightPanel();

            tab.Controls.Add(calibCenterPanel);
            tab.Controls.Add(calibRightPanel);
            tab.Controls.Add(calibLeftPanel);
        }

        private void CreateCalibrationLeftPanel()
        {
            int yPos = 10;

            // Point Capture Group
            calibPointsGroup = new GroupBox
            {
                Text = "CAPTURE CALIBRATION POINTS",
                Location = new Point(10, yPos),
                Size = new Size(320, 180),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblDist = new Label
            {
                Text = "Distance (cm):",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            numDistance = new NumericUpDown
            {
                Location = new Point(15, 55),
                Width = 140,
                DecimalPlaces = 2,
                Minimum = 0,
                Maximum = 500,
                Value = 1.0M,
                Font = new Font("Segoe UI", 11)
            };

            var lblCurrentAdc = new Label
            {
                Text = "Current ADC:",
                Location = new Point(165, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            lblCurrentAdcIndicator = new Label
            {
                Text = "---",
                Location = new Point(165, 55),
                Size = new Size(140, 30),
                Font = new Font("Segoe UI", 16, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 100, 200),
                TextAlign = ContentAlignment.MiddleCenter,
                BorderStyle = BorderStyle.FixedSingle
            };

            btnCapturePoint = new Button
            {
                Text = "     Capture Point",
                Location = new Point(15, 95),
                Size = new Size(290, 40),
                BackColor = Color.FromArgb(0, 150, 100),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };
            btnCapturePoint.Click += BtnCapturePoint_Click;

            btnClearPoints = new Button
            {
                Text = "Clear All Points",
                Location = new Point(15, 140),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(150, 150, 150),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat
            };
            btnClearPoints.Click += BtnClearPoints_Click;

            calibPointsGroup.Controls.AddRange(new Control[] {
                lblDist, numDistance, lblCurrentAdc,
                   lblCurrentAdcIndicator, btnCapturePoint,
                   btnClearPoints
            });

            yPos += 190;

            // Fit Curve Group
            calibFitGroup = new GroupBox
            {
                Text = "CURVE FITTING",
                Location = new Point(10, yPos),
                Size = new Size(320, 200),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblFit = new Label
            {
                Text = "Fit Type:",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            cmbFitType = new ComboBox
            {
                Location = new Point(15, 55),
                Width = 290,
                DropDownStyle = ComboBoxStyle.DropDownList
            };
            cmbFitType.Items.AddRange(new object[] { "Linear", "
               Polynomial (2nd)", "Polynomial (3rd)", "Power", "
               Inverse" });
            cmbFitType.SelectedIndex = 1;

            btnFitCurve = new Button
            {
                Text = "Fit Curve",
                Location = new Point(15, 90),
                Size = new Size(290, 35),
                BackColor = Color.FromArgb(100, 100, 200),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };
            btnFitCurve.Click += BtnFitCurve_Click;

            lblEquation = new Label
            {
                Text = "Equation: ---",
                Location = new Point(15, 135),
                Size = new Size(290, 20),
                Font = new Font("Segoe UI", 8)
            };

            lblRSquared = new Label
            {
                Text = "R  = ---",
                Location = new Point(15, 160),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            calibFitGroup.Controls.AddRange(new Control[] {
                lblFit, cmbFitType, btnFitCurve, lblEquation,
                   lblRSquared
            });

            yPos += 210;

            // Save/Load Group
            var saveLoadGroup = new GroupBox
            {
                Text = "CALIBRATION FILES",
                Location = new Point(10, yPos),
                Size = new Size(320, 100),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            btnSaveCalib = new Button
            {
                Text = "Save Calibration",
                Location = new Point(15, 30),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(100, 100, 100),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat
            };
            btnSaveCalib.Click += BtnSaveCalibration_Click;

            btnLoadCalib = new Button
            {
                Text = "Load Calibration",
                Location = new Point(15, 65),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(120, 120, 120),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat
            };
            btnLoadCalib.Click += BtnLoadCalibration_Click;

            saveLoadGroup.Controls.AddRange(new Control[] {
               btnSaveCalib, btnLoadCalib });

            calibLeftPanel.Controls.AddRange(new Control[] {
                calibPointsGroup, calibFitGroup, saveLoadGroup
            });
        }

        private void CreateCalibrationCenterPanel()
        {
            int yPos = 10;

            // Points List Group
            calibPointsListGroup = new GroupBox
            {
                Text = "CALIBRATION POINTS",
                Location = new Point(10, yPos),
                Size = new Size(660, 400),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            lblPointCount = new Label
            {
                Text = "Points: 0",
                Location = new Point(15, 25),
                AutoSize = true,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };

            lblAdcRange = new Label
            {
                Text = "ADC Range: ---",
                Location = new Point(150, 25),
                AutoSize = true,
                Font = new Font("Segoe UI", 10)
            };

            dgvCalibrationPoints = new DataGridView
            {
                Location = new Point(15, 55),
                Size = new Size(630, 330),
                AllowUserToAddRows = false,
                AllowUserToDeleteRows = false,
                ReadOnly = true,
                SelectionMode = DataGridViewSelectionMode.
                   FullRowSelect,
                MultiSelect = false,
                AutoSizeColumnsMode =
                   DataGridViewAutoSizeColumnsMode.Fill,
                RowHeadersVisible = false,
                BackgroundColor = Color.White,
                BorderStyle = BorderStyle.Fixed3D
            };

            dgvCalibrationPoints.Columns.Add("Point", "Point #");
            dgvCalibrationPoints.Columns.Add("Distance", "Distance (
               cm)");
            dgvCalibrationPoints.Columns.Add("ADC", "ADC Value");
            dgvCalibrationPoints.Columns["Point"].FillWeight = 20;
            dgvCalibrationPoints.Columns["Distance"].FillWeight =
               40;
            dgvCalibrationPoints.Columns["ADC"].FillWeight = 40;

            var deleteButtonColumn = new DataGridViewButtonColumn
            {
                Name = "Delete",
                Text = "Delete",
                UseColumnTextForButtonValue = true,
                FillWeight = 20
            };
            dgvCalibrationPoints.Columns.Add(deleteButtonColumn);
            dgvCalibrationPoints.CellContentClick +=
               DgvCalibrationPoints_CellContentClick;

            calibPointsListGroup.Controls.AddRange(new Control[] {
                lblPointCount, lblAdcRange, dgvCalibrationPoints
            });

            yPos += 410;

            // Visualization Group
            calibVisualizationGroup = new GroupBox
            {
                Text = "CALIBRATION CURVE",
                Location = new Point(10, yPos),
                Size = new Size(660, 380),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            plotCalibration = new FormsPlot
            {
                Location = new Point(15, 30),
                Size = new Size(630, 335),
                BackColor = Color.White
            };

            calibVisualizationGroup.Controls.Add(plotCalibration);

            calibCenterPanel.Controls.AddRange(new Control[] {
                calibPointsListGroup, calibVisualizationGroup
            });
        }

        private void CreateCalibrationRightPanel()
        {
            int yPos = 10;

            // Range Configuration Group
            calibRangeGroup = new GroupBox
            {
                Text = "RANGE CONFIGURATION",
                Location = new Point(10, yPos),
                Size = new Size(320, 420),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblInfo = new Label
            {
                Text = "Configure ADC thresholds to define\nwhen sensor is in range:",
                Location = new Point(15, 25),
                Size = new Size(290, 35),
                Font = new Font("Segoe UI", 9)
            };

            var lblMinThreshold = new Label
            {
                Text = "Too Far (Min ADC):",
                Location = new Point(15, 70),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            numMinAdcThreshold = new NumericUpDown
            {
                Location = new Point(15, 95),
                Width = 290,
                Minimum = 0,
                Maximum = 1023,
                Value = 200,
                Font = new Font("Segoe UI", 11)
            };
            numMinAdcThreshold.ValueChanged += NumThreshold_ValueChanged;

            var lblMinDesc = new Label
            {
                Text = "ADC below this = Too Far",
                Location = new Point(15, 125),
                AutoSize = true,
                Font = new Font("Segoe UI", 8),
                ForeColor = Color.Gray
            };

            var lblMaxThreshold = new Label
            {
                Text = "Too Close (Max ADC):",
                Location = new Point(15, 155),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            numMaxAdcThreshold = new NumericUpDown
            {
                Location = new Point(15, 180),
                Width = 290,
                Minimum = 0,
                Maximum = 1023,
                Value = 800,
                Font = new Font("Segoe UI", 11)
            };
            numMaxAdcThreshold.ValueChanged += NumThreshold_ValueChanged;

            var lblMaxDesc = new Label
            {
                Text = "ADC above this = Too Close",
                Location = new Point(15, 210),
                AutoSize = true,
                Font = new Font("Segoe UI", 8),
                ForeColor = Color.Gray
            };

            var lblPresets = new Label
            {
                Text = "Quick Presets:",
                Location = new Point(15, 240),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            btnPresetVeryClose = new Button
            {
                Text = "Very Close (0.3-2cm)",
                Location = new Point(15, 265),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(80, 140, 180),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 9)
            };
            btnPresetVeryClose.Click += (s, e) => { numMinAdcThreshold.Value = 500; numMaxAdcThreshold.Value = 900; };

            btnPresetClose = new Button
            {
                Text = "Close (0.5-4cm)",
                Location = new Point(15, 300),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(80, 140, 180),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 9)
            };
            btnPresetClose.Click += (s, e) => { numMinAdcThreshold.Value = 300; numMaxAdcThreshold.Value = 850; };

            btnPresetExtended = new Button
            {
                Text = "Extended (1-6cm)",
                Location = new Point(15, 335),
                Size = new Size(290, 30),
                BackColor = Color.FromArgb(80, 140, 180),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 9)
            };
            btnPresetExtended.Click += (s, e) => { numMinAdcThreshold.Value = 100; numMaxAdcThreshold.Value = 900; };

            btnApplyThresholds = new Button
            {
                Text = "    Apply Thresholds",
                Location = new Point(15, 375),
                Size = new Size(290, 35),
                BackColor = Color.FromArgb(0, 150, 100),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 10, FontStyle.Bold)
            };
            btnApplyThresholds.Click += BtnApplyThresholds_Click;

            calibRangeGroup.Controls.AddRange(new Control[] {
                lblInfo, lblMinThreshold, numMinAdcThreshold, lblMinDesc,
                lblMaxThreshold, numMaxAdcThreshold, lblMaxDesc,
                lblPresets, btnPresetVeryClose, btnPresetClose, btnPresetExtended, btnApplyThresholds
            });

            yPos += 430;

            // Current Status Group
            var statusGroup = new GroupBox
            {
                Text = "CURRENT STATUS",
                Location = new Point(10, yPos),
                Size = new Size(320, 180),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblCurrentStatus = new Label
            {
                Text = "Current ADC Value:",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            var lblCurrentAdcValue = new Label
            {
                Text = "---",
                Location = new Point(145, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            lblThresholdStatus = new Label
            {
                Text = "    Status: ---",
                Location = new Point(15, 55),
                AutoSize = true,
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                ForeColor = Color.Gray
            };

            var lblIndicator = new Label
            {
                Text = "ADC Indicator:",
                Location = new Point(15, 90),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            pbAdcIndicator = new ProgressBar
            {
                Location = new Point(15, 115),
                Size = new Size(290, 25),
                Minimum = 0,
                Maximum = 1023,
                Value = 0
            };

            var lblIndicatorScale = new Label
            {
                Text = "0           Too Far          In Range
                            Too Close          1023",
                Location = new Point(15, 145),
                Size = new Size(290, 15),
                Font = new Font("Segoe UI", 7),
                ForeColor = Color.Gray
            };

            statusGroup.Controls.AddRange(new Control[] {
                lblCurrentStatus, lblCurrentAdcValue,
                   lblThresholdStatus, lblIndicator, pbAdcIndicator,
                    lblIndicatorScale
            });

            calibRightPanel.Controls.AddRange(new Control[] {
                calibRangeGroup, statusGroup
            });
        }

        private void InitializeServices()
        {
            _serialPort = new SerialPortService();
            _serialPort.AdcDataReceived +=
               SerialPort_AdcDataReceived;
            _serialPort.ErrorOccurred += SerialPort_ErrorOccurred;
            _serialPort.ConnectionLost += SerialPort_ConnectionLost;

            _dataLogger = new DataLogger();
            _calibrationService = new CalibrationService();
            InitializeNoiseAnalysisService();
            _startTime = DateTime.Now;
        }

        private void InitializeCharts()
        {
            // Monitoring charts
            plotAdc.Plot.Title("");
            plotAdc.Plot.XLabel("Time (s)");
            plotAdc.Plot.YLabel("ADC Value");

            plotVoltage.Plot.Title("");
            plotVoltage.Plot.XLabel("Time (s)");
            plotVoltage.Plot.YLabel("Voltage (V)");

            plotDistance.Plot.Title("");
            plotDistance.Plot.XLabel("Time (s)");
            plotDistance.Plot.YLabel("Distance (cm)");

            // Calibration chart
            plotCalibration.Plot.Title("");
            plotCalibration.Plot.XLabel("ADC Value");
            plotCalibration.Plot.YLabel("Distance (cm)");
            UpdateCalibrationPlot();
        }

        private void LoadAvailablePorts()
        {
            cmbPortName.Items.Clear();
            string[] ports = SerialPortService.GetAvailablePorts();
            if (ports.Length > 0)
            {
                cmbPortName.Items.AddRange(ports);
                cmbPortName.SelectedIndex = 0;
            }
            else
            {
                cmbPortName.Items.Add("No ports");
                cmbPortName.SelectedIndex = 0;
            }
        }

        private void BtnConnect_Click(object? sender, EventArgs e)
        {
            if (cmbPortName.SelectedItem?.ToString() == "No ports")
            {
                MessageBox.Show("No COM ports available!", "Error",
                   MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }

            string portName = cmbPortName.SelectedItem!.ToString()!;
            int baudRate = int.Parse(cmbBaudRate.SelectedItem!.
               ToString()!);

            if (_serialPort.Connect(portName, baudRate))
            {
                lblStatus.Text = $"Connected to {portName}";
                lblStatus.ForeColor = Color.Green;
                lblConnectionIndicator.BackColor = Color.LimeGreen;
                btnConnect.Enabled = false;
                btnDisconnect.Enabled = true;
                cmbPortName.Enabled = false;
                cmbBaudRate.Enabled = false;
                _chartUpdateTimer.Start();
                _startTime = DateTime.Now;
            }
        }

        private void BtnDisconnect_Click(object? sender, EventArgs e)
        {
            _serialPort.Disconnect();
            lblStatus.Text = "Disconnected";
            lblStatus.ForeColor = Color.Red;
            lblConnectionIndicator.BackColor = Color.Red;
            btnConnect.Enabled = true;
            btnDisconnect.Enabled = false;
            cmbPortName.Enabled = true;
            cmbBaudRate.Enabled = true;
            _chartUpdateTimer.Stop();
        }

        private void SerialPort_AdcDataReceived(object? sender, AdcDataReceivedEventArgs e)
        {
            if (InvokeRequired)
            {
                Invoke(new Action(() => SerialPort_AdcDataReceived(sender, e)));
                return;
            }

            _currentAdcValue = e.AdcValue;
            double voltage = _currentAdcValue * 3.3 / 1023.0;
            double distance = 0;
            bool isInRange = true;
            string rangeStatus = "In Range";

            if (_currentCalibration != null && _currentCalibration.Coefficients.Length > 0)
            {
                distance = _currentCalibration.ConvertAdcToDistance(_currentAdcValue);
                isInRange = _currentCalibration.IsInRange(_currentAdcValue);
                rangeStatus = _currentCalibration.GetRangeStatus(_currentAdcValue);
            }

            // Update monitoring tab
            lblAdcValue.Text = $"ADC: {_currentAdcValue} / 1023";
            lblVoltage.Text = $"Voltage: {voltage:F3} V";
            lblDistance.Text = $"Distance: {distance:F2} cm";

            if (rangeStatus == "In Range")
            {
                lblRangeStatus.Text = "    In Range";
                lblRangeStatus.ForeColor = Color.Green;
            }
            else if (rangeStatus == "Too Close")
            {
                lblRangeStatus.Text = "    Too Close";
                lblRangeStatus.ForeColor = Color.OrangeRed;
            }
            else
            {
                lblRangeStatus.Text = "    Too Far";
                lblRangeStatus.ForeColor = Color.Red;
            }

            // Update calibration tab
            lblCurrentAdcIndicator.Text = _currentAdcValue.ToString();
            pbAdcIndicator.Value = Math.Min(Math.Max(_currentAdcValue, 0), 1023);

            if (rangeStatus == "In Range")
            {
                lblThresholdStatus.Text = "    Status: In Range";
                lblThresholdStatus.ForeColor = Color.Green;
            }
            else if (rangeStatus == "Too Close")
            {
                lblThresholdStatus.Text = "    Status: Too Close";
                lblThresholdStatus.ForeColor = Color.OrangeRed;
            }
            else
            {
                lblThresholdStatus.Text = "    Status: Too Far";
                lblThresholdStatus.ForeColor = Color.Red;
            }

            double elapsedSeconds = (DateTime.Now - _startTime).TotalSeconds;
            _timeData.Add(elapsedSeconds);
            _adcData.Add(_currentAdcValue);
            _voltageData.Add(voltage);
            _distanceData.Add(distance);

            if (_timeData.Count > 300)
            {
                _timeData.RemoveAt(0);
                _adcData.RemoveAt(0);
                _voltageData.RemoveAt(0);
                _distanceData.RemoveAt(0);
            }

            if (_isLogging)
            {
                var reading = new SensorReading(_currentAdcValue, distance, isInRange);
                _dataLogger.AddReading(reading);
                lblSampleCount.Text = $"Samples: {_dataLogger.Count}";
            }
        }

        private void ChartUpdateTimer_Tick(object? sender, EventArgs e)
        {
            if (_timeData.Count > 0)
            {
                plotAdc.Plot.Clear();
                var scatterAdc = plotAdc.Plot.Add.Scatter(_timeData.ToArray(), _adcData.ToArray());
                scatterAdc.Color = ScottPlot.Color.FromHex("#0064C8");
                scatterAdc.LineWidth = 2;
                plotAdc.Plot.Axes.AutoScale();
                plotAdc.Refresh();

                plotVoltage.Plot.Clear();
                var scatterVolt = plotVoltage.Plot.Add.Scatter(_timeData.ToArray(), _voltageData.ToArray());
                scatterVolt.Color = ScottPlot.Color.FromHex("#C86400");
                scatterVolt.LineWidth = 2;
                plotVoltage.Plot.Axes.AutoScale();
                plotVoltage.Refresh();

                plotDistance.Plot.Clear();
                var scatterDist = plotDistance.Plot.Add.Scatter(_timeData.ToArray(), _distanceData.ToArray());
                scatterDist.Color = ScottPlot.Color.FromHex("#009600");
                scatterDist.LineWidth = 2;
                plotDistance.Plot.Axes.AutoScale();
                plotDistance.Refresh();
            }
        }

        private void SerialPort_ErrorOccurred(object? sender, string e)
        {
            if (InvokeRequired)
            {
                Invoke(new Action(() => SerialPort_ErrorOccurred(sender, e)));
                return;
            }
            MessageBox.Show(e, "Serial Port Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }

        private void SerialPort_ConnectionLost(object? sender, EventArgs e)
        {
            if (InvokeRequired)
            {
                Invoke(new Action(() => SerialPort_ConnectionLost(sender, e)));
                return;
            }
            BtnDisconnect_Click(null, EventArgs.Empty);
            MessageBox.Show("Connection lost!", "Error", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }

        private void BtnCapturePoint_Click(object? sender, EventArgs e)
        {
            if (!_serialPort.IsConnected)
            {
                MessageBox.Show("Please connect to the sensor first!", "Not Connected", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            double distance = (double)numDistance.Value;
            _calibrationPoints.Add(new CalibrationPoint(distance, _currentAdcValue));

            UpdateCalibrationPointsList();
            UpdateCalibrationPlot();

            MessageBox.Show($"Point captured!\nDistance: {distance:F2} cm\nADC: {_currentAdcValue}",
                "Point Added", MessageBoxButtons.OK, MessageBoxIcon.Information);
        }

        private void BtnClearPoints_Click(object? sender, EventArgs e)
        {
            _calibrationPoints.Clear();
            _currentCalibration = null;
            lblEquation.Text = "Equation: ---";
            lblRSquared.Text = "R  = ---";

            UpdateCalibrationPointsList();
            UpdateCalibrationPlot();

            MessageBox.Show("All calibration points cleared!", "Cleared", MessageBoxButtons.OK, MessageBoxIcon.Information);
        }

        private void DgvCalibrationPoints_CellContentClick(object? sender, DataGridViewCellEventArgs e)
        {
            if (e.RowIndex >= 0 && e.ColumnIndex == dgvCalibrationPoints.Columns["Delete"].Index)
            {
                var result = MessageBox.Show(
                    $"Delete point {e.RowIndex + 1}?",
                    "Confirm Delete",
                    MessageBoxButtons.YesNo,
                    MessageBoxIcon.Question);

                if (result == DialogResult.Yes)
                {
                    _calibrationPoints.RemoveAt(e.RowIndex);
                    _currentCalibration = null;
                    lblEquation.Text = "Equation: ---";
                    lblRSquared.Text = "R  = ---";

                    UpdateCalibrationPointsList();
                    UpdateCalibrationPlot();
                }
            }
        }

        private void UpdateCalibrationPointsList()
        {
            dgvCalibrationPoints.Rows.Clear();

            for (int i = 0; i < _calibrationPoints.Count; i++)
            {
                var point = _calibrationPoints[i];
                dgvCalibrationPoints.Rows.Add(
                    (i + 1).ToString(),
                    point.Distance.ToString("F2"),
                    point.AdcValue.ToString()
                );
            }

            lblPointCount.Text = $"Points: {_calibrationPoints.Count}";

            if (_calibrationPoints.Count > 0)
            {
                int minAdc = _calibrationPoints.Min(p => p.AdcValue);
                int maxAdc = _calibrationPoints.Max(p => p.AdcValue);
                lblAdcRange.Text = $"ADC Range: {minAdc} - {maxAdc}";
            }
            else
            {
                lblAdcRange.Text = "ADC Range: ---";
            }
        }

        private void UpdateCalibrationPlot()
        {
            plotCalibration.Plot.Clear();

            if (_calibrationPoints.Count > 0)
            {
                double[] adcValues = _calibrationPoints.Select(p => (double)p.AdcValue).ToArray();
                double[] distances = _calibrationPoints.Select(p => p.Distance).ToArray();

                var scatter = plotCalibration.Plot.Add.Scatter(adcValues, distances);
                scatter.Color = ScottPlot.Color.FromHex("#0064C8");
                scatter.MarkerSize = 10;
                scatter.LineWidth = 0;
                scatter.LegendText = "Calibration Points";

                // If we have a fitted curve, plot it
                if (_currentCalibration != null && _currentCalibration.Coefficients.Length > 0)
                {
                    int minAdc = (int)adcValues.Min();
                    int maxAdc = (int)adcValues.Max();
                    int range = maxAdc - minAdc;
                    minAdc = Math.Max(0, minAdc - range / 4);
                    maxAdc = Math.Min(1023, maxAdc + range / 4);

                    List<double> fitAdcValues = new List<double>();
                    List<double> fitDistances = new List<double>();

                    for (int adc = minAdc; adc <= maxAdc; adc += 2)
                    {
                        double dist = _currentCalibration.ConvertAdcToDistance(adc);
                        fitAdcValues.Add(adc);
                        fitDistances.Add(dist);
                    }

                    var fitLine = plotCalibration.Plot.Add.Scatter(fitAdcValues.ToArray(), fitDistances.ToArray());
                    fitLine.Color = ScottPlot.Color.FromHex("#FF6600");
                    fitLine.LineWidth = 2;
                    fitLine.MarkerSize = 0;
                    fitLine.LegendText = "Fitted Curve";

                    plotCalibration.Plot.Legend.IsVisible = true;
                }
            }

            plotCalibration.Plot.Axes.AutoScale();
            plotCalibration.Refresh();
        }

        private void BtnFitCurve_Click(object? sender, EventArgs e)
        {
            if (_calibrationPoints.Count < 2)
            {
                MessageBox.Show("Need at least 2 calibration points!", "Insufficient Data", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            FitType fitType = cmbFitType.SelectedIndex switch
            {
                0 => FitType.Linear,
                1 => FitType.Polynomial2,
                2 => FitType.Polynomial3,
                3 => FitType.Power,
                4 => FitType.Inverse,
                _ => FitType.Polynomial2
            };

            try
            {
                _currentCalibration = _calibrationService.PerformCalibration(_calibrationPoints, fitType);

                // Apply current threshold settings
                _currentCalibration.MinAdcThreshold = (int)numMinAdcThreshold.Value;
                _currentCalibration.MaxAdcThreshold = (int)numMaxAdcThreshold.Value;

                lblEquation.Text = $"Equation: {_currentCalibration.Equation}";
                lblRSquared.Text = $"R  = {_currentCalibration.RSquared:F6}";

                UpdateCalibrationPlot();

                MessageBox.Show($"Calibration successful!\nPoints: {_calibrationPoints.Count}\nR  = {_currentCalibration.RSquared:F6}",
                    "Success", MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
            catch (Exception ex)
            {
                MessageBox.Show($"Calibration failed: {ex.Message}", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }

        private void BtnApplyThresholds_Click(object? sender, EventArgs e)
        {
            if (_currentCalibration != null)
            {
                _currentCalibration.MinAdcThreshold = (int)numMinAdcThreshold.Value;
                _currentCalibration.MaxAdcThreshold = (int)numMaxAdcThreshold.Value;
                MessageBox.Show("Thresholds applied to current calibration!", "Applied", MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
            else
            {
                MessageBox.Show("Thresholds will be applied when you fit a curve.", "Info", MessageBoxButtons.OK, MessageBoxIcon.Information);
            }
        }

        private void NumThreshold_ValueChanged(object? sender, EventArgs e)
        {
            // Ensure min < max
            if (numMinAdcThreshold.Value >= numMaxAdcThreshold.Value)
            {
                if (sender == numMinAdcThreshold)
                {
                    numMaxAdcThreshold.Value = numMinAdcThreshold.Value + 1;
                }
                else
                {
                    numMinAdcThreshold.Value = numMaxAdcThreshold.Value - 1;
                }
            }
        }

        private void BtnStartLogging_Click(object? sender, EventArgs e)
        {
            _isLogging = true;
            _dataLogger.Clear();
            btnStartLogging.Enabled = false;
            btnStopLogging.Enabled = true;
            lblSampleCount.Text = "Samples: 0";
        }

        private void BtnStopLogging_Click(object? sender, EventArgs e)
        {
            _isLogging = false;
            btnStartLogging.Enabled = true;
            btnStopLogging.Enabled = false;
        }

        private void BtnExport_Click(object? sender, EventArgs e)
        {
            if (_dataLogger.Count == 0)
            {
                MessageBox.Show("No data to export!", "No Data", MessageBoxButtons.OK, MessageBoxIcon.Information);
                return;
            }

            using (SaveFileDialog sfd = new SaveFileDialog())
            {
                sfd.Filter = "CSV Files (*.csv)|*.csv";
                sfd.FileName = $"sensor_data_{DateTime.Now:yyyyMMdd_HHmmss}.csv";
                if (sfd.ShowDialog() == DialogResult.OK)
                {
                    try
                    {
                        _dataLogger.ExportToCsv(sfd.FileName);
                        MessageBox.Show($"Data exported successfully!\n{sfd.FileName}", "Success", MessageBoxButtons.OK, MessageBoxIcon.Information);
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show($"Export failed: {ex.Message}", "Error", MessageBoxButtons.OK,
                           MessageBoxIcon.Error);
                    }
                }
            }
        }

        private void BtnSaveCalibration_Click(object? sender, EventArgs e)
        {
            if (_currentCalibration == null)
            {
                MessageBox.Show("No calibration to save!", "No Calibration", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            using (SaveFileDialog sfd = new SaveFileDialog())
            {
                sfd.Filter = "JSON Files (*.json)|*.json";
                sfd.FileName = $"calibration_{DateTime.Now:yyyyMMdd}.json";
                if (sfd.ShowDialog() == DialogResult.OK)
                {
                    try
                    {
                        _calibrationService.SaveCalibration(_currentCalibration, sfd.FileName);
                        MessageBox.Show("Calibration saved!", "Success", MessageBoxButtons.OK, MessageBoxIcon.Information);
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show($"Save failed: {ex.Message}", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    }
                }
            }
        }

        private void BtnLoadCalibration_Click(object? sender, EventArgs e)
        {
            using (OpenFileDialog ofd = new OpenFileDialog())
            {
                ofd.Filter = "JSON Files (*.json)|*.json";
                if (ofd.ShowDialog() == DialogResult.OK)
                {
                    try
                    {
                        _currentCalibration = _calibrationService.LoadCalibration(ofd.FileName);
                        _calibrationPoints = new List<CalibrationPoint>(_currentCalibration.Points);

                        lblEquation.Text = $"Equation: {_currentCalibration.Equation}";
                        lblRSquared.Text = $"R   = {_currentCalibration.RSquared:F6}";

                        cmbFitType.SelectedIndex = _currentCalibration.FitType switch
                        {
                            FitType.Linear => 0,
                            FitType.Polynomial2 => 1,
                            FitType.Polynomial3 => 2,
                            FitType.Power => 3,
                            FitType.Inverse => 4,
                            _ => 1
                        };

                        // Load thresholds
                        numMinAdcThreshold.Value = _currentCalibration.MinAdcThreshold;
                        numMaxAdcThreshold.Value = _currentCalibration.MaxAdcThreshold;

                        UpdateCalibrationPointsList();
                        UpdateCalibrationPlot();

                        MessageBox.Show("Calibration loaded!", "Success", MessageBoxButtons.OK, MessageBoxIcon.Information);
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show($"Load failed: {ex.Message}", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    }
                }
```

```
            }
        }

        protected override void OnFormClosing(FormClosingEventArgs e)
        {
            _chartUpdateTimer.Stop();
            _serialPort.Disconnect();
            _serialPort.Dispose();
            base.OnFormClosing(e);
        }
    }
}
```

A.3 Noise Analysis UI Panel

Listing 3: MainForm Noise Analysis Partial Class

```
using System;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;
using RyanSensorApp.Models;
using RyanSensorApp.Services;
using ScottPlot.WinForms;

namespace RyanSensorApp
{
    public partial class MainForm
    {
        private void CreateNoiseAnalysisTab(TabPage tab)
        {
            tab.BackColor = Color.FromArgb(240, 240, 245);

            // Create three-panel layout
            noiseLeftPanel = new Panel
            {
                Dock = DockStyle.Left,
                Width = 350,
                BackColor = Color.White,
                Padding = new Padding(15)
            };

            noiseRightPanel = new Panel
            {
                Dock = DockStyle.Right,
                Width = 400,
                BackColor = Color.White,
                Padding = new Padding(15)
            };

            noiseCenterPanel = new Panel
            {
                Dock = DockStyle.Fill,
                BackColor = Color.FromArgb(240, 240, 245),
                Padding = new Padding(15)
            };

            CreateNoiseAnalysisLeftPanel();
            CreateNoiseAnalysisCenterPanel();
            CreateNoiseAnalysisRightPanel();

            tab.Controls.Add(noiseCenterPanel);
            tab.Controls.Add(noiseRightPanel);
            tab.Controls.Add(noiseLeftPanel);
        }

        private void CreateNoiseAnalysisLeftPanel()
        {
            int yPos = 10;

            // Test Setup Group
            noiseTestSetupGroup = new GroupBox
            {
                Text = "TEST CONFIGURATION",
                Location = new Point(10, yPos),
                Size = new Size(320, 150),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblPosition = new Label
            {
                Text = "Test Position:",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            cmbTestPosition = new ComboBox
            {
                Location = new Point(15, 55),
                Width = 290,
                DropDownStyle = ComboBoxStyle.DropDownList
            };
            cmbTestPosition.Items.AddRange(new object[] {
                "Middle Range",
                "Near Extreme (Close)",
                "Far Extreme (Far)",
                "Custom Position"
            });
            cmbTestPosition.SelectedIndex = 0;

            var lblDuration = new Label
            {
                Text = "Test Duration (seconds):",
                Location = new Point(15, 90),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            numTestDuration = new NumericUpDown
            {
                Location = new Point(15, 115),
                Width = 120,
                Minimum = 1,
                Maximum = 60,
                Value = 10,
                Font = new Font("Segoe UI", 11)
            };

            var lblNote = new Label
            {
                Text = "Lab standard: 10s",
                Location = new Point(145, 118),
                AutoSize = true,
                Font = new Font("Segoe UI", 8),
                ForeColor = Color.Gray
            };

            noiseTestSetupGroup.Controls.AddRange(new Control[] {
                lblPosition, cmbTestPosition, lblDuration, numTestDuration, lblNote
            });

            yPos += 160;

            // Test Control Group
            noiseTestControlGroup = new GroupBox
            {
                Text = "TEST CONTROL",
                Location = new Point(10, yPos),
                Size = new Size(320, 200),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            lblTestStatus = new Label
            {
                Text = "    Ready",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                ForeColor = Color.Gray
            };

            lblTestTimer = new Label
            {
                Text = "0.0 s",
                Location = new Point(15, 60),
                Size = new Size(290, 30),
                Font = new Font("Segoe UI", 20, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 100, 200),
                TextAlign = ContentAlignment.MiddleCenter
            };

            pbTestProgress = new ProgressBar
            {
                Location = new Point(15, 100),
                Size = new Size(290, 25),
                Minimum = 0,
                Maximum = 100,
                Value = 0
            };

            btnStartTest = new Button
            {
                Text = "    START TEST",
                Location = new Point(15, 135),
                Size = new Size(290, 50),
                BackColor = Color.FromArgb(0, 150, 0),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 12, FontStyle.Bold)
            };
            btnStartTest.Click += BtnStartNoiseTest_Click;

            btnStopTest = new Button
            {
                Text = "    STOP TEST",
                Location = new Point(15, 135),
                Size = new Size(290, 50),
                BackColor = Color.FromArgb(200, 0, 0),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 12, FontStyle.Bold),
                Visible = false
            };
            btnStopTest.Click += BtnStopNoiseTest_Click;

            noiseTestControlGroup.Controls.AddRange(new Control[] {
                lblTestStatus, lblTestTimer, pbTestProgress, btnStartTest, btnStopTest
            });

            yPos += 210;

            // Test Results Group
            noiseTestResultsGroup = new GroupBox
            {
                Text = "CURRENT TEST RESULTS",
                Location = new Point(10, yPos),
                Size = new Size(320, 240),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblLiveSamples = new Label
            {
                Text = "Samples:",
                Location = new Point(15, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            lblCurrentSamples = new Label
            {
                Text = "0",
                Location = new Point(100, 30),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblLiveMean = new Label
            {
                Text = "Mean Position:",
                Location = new Point(15, 55),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            lblCurrentMean = new Label
            {
                Text = "--- cm",
                Location = new Point(120, 55),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblLiveStdDev = new Label
            {
                Text = "Std Dev (RMS):",
                Location = new Point(15, 80),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            lblCurrentStdDev = new Label
            {
                Text = "--- cm",
                Location = new Point(120, 80),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold),
                ForeColor = Color.FromArgb(200, 0, 0)
            };

            var separator = new Label
            {
                Text = "Final Test Results:",
                Location = new Point(15, 115),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold),
                ForeColor = Color.FromArgb(0, 100, 150)
            };

            lblTestMean = new Label
            {
                Text = "Mean: ---",
                Location = new Point(15, 140),
                AutoSize = true,
                Font = new Font("Segoe UI", 10)
            };

            lblTestStdDev = new Label
            {
                Text = "RMS Noise: ---",
                Location = new Point(15, 165),
                Size = new Size(290, 20),
                Font = new Font("Segoe UI", 10, FontStyle.Bold),
                ForeColor = Color.FromArgb(200, 0, 0)
            };

            lblTestRange = new Label
            {
                Text = "Range: ---",
                Location = new Point(15, 190),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            lblTestSamples = new Label
            {
                Text = "Samples: ---",
                Location = new Point(15, 210),
                AutoSize = true,
                Font = new Font("Segoe UI", 9)
            };

            noiseTestResultsGroup.Controls.AddRange(new Control[] {
                lblLiveSamples, lblCurrentSamples, lblLiveMean, lblCurrentMean,
                lblLiveStdDev, lblCurrentStdDev, separator,
                lblTestMean, lblTestStdDev, lblTestRange, lblTestSamples
            });

            noiseLeftPanel.Controls.AddRange(new Control[] {
                noiseTestSetupGroup, noiseTestControlGroup, noiseTestResultsGroup
            });
        }

        private void CreateNoiseAnalysisCenterPanel()
        {
            int yPos = 10;

            // Visualization Group
            noiseVisualizationGroup = new GroupBox
            {
                Text = "REAL-TIME NOISE VISUALIZATION",
                Location = new Point(10, yPos),
                Size = new Size(600, 800),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblInfo = new Label
            {
                Text = "Position readings over test duration with statistical bands",
                Location = new Point(15, 25),
                AutoSize = true,
                Font = new Font("Segoe UI", 9),
                ForeColor = Color.Gray
            };

            plotNoiseTest = new FormsPlot
            {
                Location = new Point(15, 50),
                Size = new Size(570, 735),
                BackColor = Color.White
            };

            noiseVisualizationGroup.Controls.AddRange(new Control[] {
                lblInfo, plotNoiseTest
            });

            noiseCenterPanel.Controls.Add(noiseVisualizationGroup);
        }

        private void CreateNoiseAnalysisRightPanel()
        {
            int yPos = 10;

            // History Group
            noiseHistoryGroup = new GroupBox
            {
                Text = "TEST HISTORY",
                Location = new Point(10, yPos),
                Size = new Size(370, 350),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            dgvNoiseHistory = new DataGridView
            {
                Location = new Point(15, 30),
                Size = new Size(340, 280),
                AllowUserToAddRows = false,
                AllowUserToDeleteRows = false,
                ReadOnly = true,
                SelectionMode = DataGridViewSelectionMode.
                   FullRowSelect,
                MultiSelect = false,
                AutoSizeColumnsMode =
                   DataGridViewAutoSizeColumnsMode.Fill,
                RowHeadersVisible = false,
                BackgroundColor = Color.White,
                BorderStyle = BorderStyle.Fixed3D
            };

            dgvNoiseHistory.Columns.Add("Test", "Test#");
            dgvNoiseHistory.Columns.Add("Position", "Position");
            dgvNoiseHistory.Columns.Add("Mean", "Mean(cm)");
            dgvNoiseHistory.Columns.Add("StdDev", "RMS(cm)");
            dgvNoiseHistory.Columns.Add("Samples", "N");
            dgvNoiseHistory.Columns["Test"].FillWeight = 15;
            dgvNoiseHistory.Columns["Position"].FillWeight = 30;
            dgvNoiseHistory.Columns["Mean"].FillWeight = 20;
            dgvNoiseHistory.Columns["StdDev"].FillWeight = 20;
            dgvNoiseHistory.Columns["Samples"].FillWeight = 15;

            btnClearNoiseTests = new Button
            {
                Text = "Clear All Tests",
                Location = new Point(15, 315),
                Size = new Size(165, 25),
                BackColor = Color.FromArgb(150, 150, 150),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 9)
            };
            btnClearNoiseTests.Click += BtnClearNoiseTests_Click;

            btnExportNoiseTests = new Button
            {
                Text = "Export to CSV",
                Location = new Point(190, 315),
                Size = new Size(165, 25),
                BackColor = Color.FromArgb(50, 100, 150),
                ForeColor = Color.White,
                FlatStyle = FlatStyle.Flat,
                Font = new Font("Segoe UI", 9)
            };
            btnExportNoiseTests.Click += BtnExportNoiseTests_Click;

            noiseHistoryGroup.Controls.AddRange(new Control[] {
                dgvNoiseHistory, btnClearNoiseTests,
                   btnExportNoiseTests
            });

            yPos += 360;

            // Comparison Group
            noiseComparisonGroup = new GroupBox
            {
                Text = "COMPARISON & ANALYSIS",
                Location = new Point(10, yPos),
                Size = new Size(370, 430),
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            var lblCompInfo = new Label
            {
                Text = "RMS Noise Comparison:",
                Location = new Point(15, 25),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            plotNoiseComparison = new FormsPlot
            {
                Location = new Point(15, 50),
                Size = new Size(340, 200),
                BackColor = Color.White
            };

            var lblSummary = new Label
            {
                Text = "Statistical Summary:",
                Location = new Point(15, 260),
                AutoSize = true,
                Font = new Font("Segoe UI", 9, FontStyle.Bold)
            };

            txtComparison = new TextBox
            {
                Location = new Point(15, 285),
                Size = new Size(340, 135),
                Multiline = true,
                ScrollBars = ScrollBars.Vertical,
                ReadOnly = true,
                Font = new Font("Consolas", 8),
                BackColor = Color.FromArgb(250, 250, 250)
            };

            noiseComparisonGroup.Controls.AddRange(new Control[] {
                lblCompInfo, plotNoiseComparison, lblSummary, txtComparison
            });

            noiseRightPanel.Controls.AddRange(new Control[] {
                noiseHistoryGroup, noiseComparisonGroup
            });
        }

        private void InitializeNoiseAnalysisService()
        {
            _noiseAnalysisService = new NoiseAnalysisService();

            // Initialize noise test timer
            _noiseTestTimer = new System.Windows.Forms.Timer();
            _noiseTestTimer.Interval = 100;  // Update every 100ms
            _noiseTestTimer.Tick += NoiseTestTimer_Tick;

            // Initialize noise plot
            plotNoiseTest.Plot.Title("");
            plotNoiseTest.Plot.XLabel("Time (s)");
            plotNoiseTest.Plot.YLabel("Distance (cm)");

            // Initialize comparison plot
            plotNoiseComparison.Plot.Title("");
            plotNoiseComparison.Plot.XLabel("Test Position");
            plotNoiseComparison.Plot.YLabel("RMS Noise (cm)");

            UpdateNoiseComparisonPlot();
        }

        private void BtnStartNoiseTest_Click(object? sender, EventArgs e)
        {
            if (!_serialPort.IsConnected)
            {
                MessageBox.Show("Please connect to the sensor first!", "Not Connected",
                    MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            if (_currentCalibration == null)
            {
                MessageBox.Show("Please load a calibration first!", "No Calibration",
                    MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            // Get test position
            TestPosition position = cmbTestPosition.SelectedIndex switch
            {
                0 => TestPosition.MiddleRange,
                1 => TestPosition.NearExtreme,
                2 => TestPosition.FarExtreme,
                3 => TestPosition.Custom,
                _ => TestPosition.MiddleRange
            };

            // Create new test
            _currentNoiseTest = _noiseAnalysisService.CreateNewTest(position);
            _currentNoiseTest.DurationSeconds = (double)numTestDuration.Value;
            _noiseTestStartTime = DateTime.Now;
            _isNoiseTestRunning = true;

            // Update UI
            btnStartTest.Visible = false;
            btnStopTest.Visible = true;
            cmbTestPosition.Enabled = false;
            numTestDuration.Enabled = false;
            lblTestStatus.Text = "    RECORDING...";
            lblTestStatus.ForeColor = Color.Red;
            lblTestTimer.Text = "0.0 s";
            pbTestProgress.Value = 0;

            // Clear current results
            lblCurrentSamples.Text = "0";
            lblCurrentMean.Text = "--- cm";
            lblCurrentStdDev.Text = "--- cm";

            // Start timer
            _noiseTestTimer.Start();
        }

        private void BtnStopNoiseTest_Click(object? sender, EventArgs e)
        {
            StopNoiseTest();
        }

        private void StopNoiseTest()
        {
            if (!_isNoiseTestRunning || _currentNoiseTest == null)
                return;

            _isNoiseTestRunning = false;
            _noiseTestTimer.Stop();

            // Calculate final statistics
            _currentNoiseTest.CalculateStatistics();

            // Save test
            _noiseAnalysisService.SaveTest(_currentNoiseTest);

            // Update UI
            lblTestStatus.Text = "    Test Complete";
            lblTestStatus.ForeColor = Color.Green;
            btnStartTest.Visible = true;
            btnStopTest.Visible = false;
            cmbTestPosition.Enabled = true;
            numTestDuration.Enabled = true;

            // Display final results
            lblTestMean.Text = $"Mean: {_currentNoiseTest.MeanDistance:F4} cm";
            lblTestStdDev.Text = $"RMS Noise: {_currentNoiseTest.StandardDeviation:F4} cm";
            lblTestRange.Text = $"Range: {_currentNoiseTest.MinDistance:F4} - {_currentNoiseTest.MaxDistance:F4} cm";
            lblTestSamples.Text = $"Samples: {_currentNoiseTest.SampleCount}";

            // Update test history
            UpdateNoiseHistoryTable();
            UpdateNoiseComparisonPlot();
            UpdateComparisonSummary();

            MessageBox.Show(_currentNoiseTest.GetSummary(), "Test Complete",
                MessageBoxButtons.OK, MessageBoxIcon.Information);

            _currentNoiseTest = null;
        }

        private void NoiseTestTimer_Tick(object? sender, EventArgs e)
        {
            if (!_isNoiseTestRunning || _currentNoiseTest == null)
                return;

            // Calculate elapsed time
            double elapsedSeconds = (DateTime.Now - _noiseTestStartTime).TotalSeconds;
            lblTestTimer.Text = $"{elapsedSeconds:F1} s";

            // Update progress bar
            int progress = (int)((elapsedSeconds / _currentNoiseTest.DurationSeconds) * 100);
            pbTestProgress.Value = Math.Min(progress, 100);

            // Add current reading to test
            if (_currentCalibration != null)
            {
                double distance = _currentCalibration.ConvertAdcToDistance(_currentAdcValue);
                _currentNoiseTest.DistanceReadings.Add(distance);
                _currentNoiseTest.AdcReadings.Add(_currentAdcValue);

                // Calculate running statistics
                if (_currentNoiseTest.DistanceReadings.Count > 1)
                {
                    double mean = _currentNoiseTest.DistanceReadings.Average();
                    double sumSqDiff = _currentNoiseTest.DistanceReadings.Sum(x => Math.Pow(x - mean, 2));
                    double stdDev = Math.Sqrt(sumSqDiff / _currentNoiseTest.DistanceReadings.Count);

                    lblCurrentSamples.Text = _currentNoiseTest.DistanceReadings.Count.ToString();
                    lblCurrentMean.Text = $"{mean:F4} cm";
                    lblCurrentStdDev.Text = $"{stdDev:F4} cm";
                }

                // Update plot
                UpdateNoiseTestPlot();
            }

            // Check if test duration reached
            if (elapsedSeconds >= _currentNoiseTest.DurationSeconds)
            {
                StopNoiseTest();
            }
        }

        private void UpdateNoiseTestPlot()
        {
            if (_currentNoiseTest == null || _currentNoiseTest.DistanceReadings.Count == 0)
                return;

            plotNoiseTest.Plot.Clear();

            // Create time array
            double[] times = Enumerable.Range(0, _currentNoiseTest.DistanceReadings.Count)
                .Select(i => i * 0.1)  // 100ms intervals
                .ToArray();
            double[] distances = _currentNoiseTest.DistanceReadings.ToArray();

            // Plot data
            var scatter = plotNoiseTest.Plot.Add.Scatter(times, distances);
            scatter.Color = ScottPlot.Color.FromHex("#0064C8");
            scatter.LineWidth = 2;
            scatter.MarkerSize = 3;
            scatter.LegendText = "Position Readings";

            // Add mean line if we have enough data
            if (_currentNoiseTest.DistanceReadings.Count > 2)
            {
                double mean = distances.Average();
                double stdDev = Math.Sqrt(distances.Sum(x => Math.Pow(x - mean, 2)) / distances.Length);

                var meanLine = plotNoiseTest.Plot.Add.HorizontalLine(mean);
                meanLine.Color = ScottPlot.Color.FromHex("#00AA00");
                meanLine.LineWidth = 2;
                meanLine.LinePattern = ScottPlot.LinePattern.Dashed;
                meanLine.LegendText = "Mean";

                // Add  1  band
                var plusSigma = plotNoiseTest.Plot.Add.HorizontalLine(mean + stdDev);
                plusSigma.Color = ScottPlot.Color.FromHex("#FF6600");
                plusSigma.LineWidth = 1;
                plusSigma.LinePattern = ScottPlot.LinePattern.Dotted;
                plusSigma.LegendText = "   1   ";

                var minusSigma = plotNoiseTest.Plot.Add.HorizontalLine(mean - stdDev);
                minusSigma.Color = ScottPlot.Color.FromHex("#FF6600");
                minusSigma.LineWidth = 1;
                minusSigma.LinePattern = ScottPlot.LinePattern.Dotted;
            }

            plotNoiseTest.Plot.Legend.IsVisible = true;
            plotNoiseTest.Plot.Axes.AutoScale();
            plotNoiseTest.Refresh();
        }

        private void UpdateNoiseHistoryTable()
        {
            dgvNoiseHistory.Rows.Clear();

            var tests = _noiseAnalysisService.GetAllTests();
            foreach (var test in tests)
            {
                dgvNoiseHistory.Rows.Add(
                    test.TestNumber.ToString(),
                    test.GetPositionString(),
                    test.MeanDistance.ToString("F4"),
                    test.StandardDeviation.ToString("F4"),
                    test.SampleCount.ToString()
                );
            }
        }

        private void UpdateNoiseComparisonPlot()
        {
            plotNoiseComparison.Plot.Clear();

            var avgRms = _noiseAnalysisService.GetAverageRMSByPosition();
            if (avgRms.Count == 0)
            {
                plotNoiseComparison.Refresh();
                return;
            }

            // Create bar chart data
            var positions = avgRms.Keys.ToList();
            var rmsValues = avgRms.Values.ToList();

            double[] posIndices = Enumerable.Range(0, positions.Count).Select(i => (double)i).ToArray();
            string[] posLabels = positions.Select(p => p switch
            {
                TestPosition.MiddleRange => "Middle",
                TestPosition.NearExtreme => "Near",
                TestPosition.FarExtreme => "Far",
                _ => "Custom"
            }).ToArray();

            var bar = plotNoiseComparison.Plot.Add.Bars(posIndices, rmsValues.ToArray());
            bar.Color = ScottPlot.Color.FromHex("#FF6600");

            plotNoiseComparison.Plot.Axes.Bottom.TickGenerator = new ScottPlot.TickGenerators.NumericManual(
                posIndices, posLabels);
            plotNoiseComparison.Plot.Axes.AutoScale();
            plotNoiseComparison.Refresh();
        }

        private void UpdateComparisonSummary()
        {
            txtComparison.Text = _noiseAnalysisService.GetComparisonSummary();
        }

        private void BtnClearNoiseTests_Click(object? sender, EventArgs e)
        {
            var result = MessageBox.Show(
                "Clear all noise test data?",
                "Confirm Clear",
                MessageBoxButtons.YesNo,
                MessageBoxIcon.Question);

            if (result == DialogResult.Yes)
            {
                _noiseAnalysisService.ClearAllTests();
                UpdateNoiseHistoryTable();
                UpdateNoiseComparisonPlot();
                UpdateComparisonSummary();

                lblTestMean.Text = "Mean: ---";
                lblTestStdDev.Text = "RMS Noise: ---";
                lblTestRange.Text = "Range: ---";
                lblTestSamples.Text = "Samples: ---";
            }
        }

        private void BtnExportNoiseTests_Click(object? sender, EventArgs e)
        {
            if (_noiseAnalysisService.GetTestCount() == 0)
            {
                MessageBox.Show("No test data to export!", "No Data",
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
                return;
            }

            using (SaveFileDialog sfd = new SaveFileDialog())
            {
                sfd.Filter = "CSV Files (*.csv)|*.csv";
                sfd.FileName = $"noise_analysis_{DateTime.Now:yyyyMMdd_HHmmss}.csv";
                if (sfd.ShowDialog() == DialogResult.OK)
                {
                    try
                    {
                        _noiseAnalysisService.ExportToCSV(sfd.FileName);
                        MessageBox.Show($"Noise analysis exported successfully!\n{sfd.FileName}",
                            "Success", MessageBoxButtons.OK, MessageBoxIcon.Information);
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show($"Export failed: {ex.Message}", "Error",
                            MessageBoxButtons.OK, MessageBoxIcon.Error);
                    }
                }
            }
        }
    }
}
```

A.4 Serial Port Handling

Listing 4: SerialPortService Data Pipeline

```
using System;
using System.IO.Ports;
using System.Threading;
using System.Threading.Tasks;

namespace RyanSensorApp.Services
{
    public class SerialPortService : IDisposable
    {
        private SerialPort? _serialPort;
        private CancellationTokenSource? _cancellationTokenSource;
        private Task? _readTask;
        private byte[] _buffer = new byte[3];
        private int _bufferIndex = 0;
        private const byte START_BYTE = 0xFF;

        public event EventHandler<AdcDataReceivedEventArgs>? AdcDataReceived;
        public event EventHandler<string>? ErrorOccurred;
        public event EventHandler? ConnectionLost;

        public bool IsConnected => _serialPort?.IsOpen ?? false;

        public bool Connect(string portName, int baudRate = 9600)
        {
            try
            {
                Disconnect();

                _serialPort = new SerialPort(portName, baudRate, Parity.None, 8, StopBits.One)
                {
                    ReadTimeout = 1000,
                    WriteTimeout = 1000
                };

                _serialPort.Open();

                // Start reading in background thread
                _cancellationTokenSource = new CancellationTokenSource();
                _readTask = Task.Run(() => ReadDataAsync(_cancellationTokenSource.Token));

                return true;
            }
            catch (Exception ex)
            {
                ErrorOccurred?.Invoke(this, $"Connection error: {ex.Message}");
                return false;
            }
        }

        public void Disconnect()
        {
            try
            {
                _cancellationTokenSource?.Cancel();
                _readTask?.Wait(TimeSpan.FromSeconds(2));

                if (_serialPort?.IsOpen == true)
                {
                    _serialPort.Close();
                }

                _serialPort?.Dispose();
                _serialPort = null;
                _cancellationTokenSource?.Dispose();
                _cancellationTokenSource = null;
                _bufferIndex = 0;
            }
            catch (Exception ex)
            {
                ErrorOccurred?.Invoke(this, $"Disconnect error: {ex.Message}");
            }
        }

        private async Task ReadDataAsync(CancellationToken cancellationToken)
        {
            while (!cancellationToken.IsCancellationRequested)
            {
                try
                {
                    if (_serialPort?.IsOpen == true && _serialPort.BytesToRead > 0)
                    {
                        int readByte = _serialPort.ReadByte();
                        if (readByte >= 0)
                        {
                            ProcessByte((byte)readByte);
                        }
                    }
                    else
                    {
                        await Task.Delay(1, cancellationToken);
                    }
                }
                catch (TimeoutException)
                {
                    // Normal timeout, continue
                }
                catch (InvalidOperationException)
                {
                    // Port closed
                    ConnectionLost?.Invoke(this, EventArgs.Empty);
                    break;
                }
                catch (Exception ex)
                {
                    if (!cancellationToken.IsCancellationRequested)
                    {
                        ErrorOccurred?.Invoke(this, $"Read error: {ex.Message}");
                        await Task.Delay(100, cancellationToken);
                    }
                }
            }
        }

        private void ProcessByte(byte data)
        {
            // State machine for packet parsing
            if (_bufferIndex == 0)
            {
                // Looking for start byte
                if (data == START_BYTE)
                {
                    _buffer[0] = data;
                    _bufferIndex = 1;
                }
            }
            else if (_bufferIndex == 1)
            {
                // MS5B (most significant 5 bits)
                _buffer[1] = data;
                _bufferIndex = 2;
            }
            else if (_bufferIndex == 2)
            {
                // LS5B (least significant 5 bits)
                _buffer[2] = data;

                // Reassemble 10-bit ADC value
                int ms5b = _buffer[1] & 0x1F;  // Mask to 5 bits
                int ls5b = _buffer[2] & 0x1F;  // Mask to 5 bits
                int adcValue = (ms5b << 5) | ls5b;  // Combine into 10-bit value

                // Raise event with ADC data
                AdcDataReceived?.Invoke(this, new AdcDataReceivedEventArgs(adcValue));

                // Reset for next packet
                _bufferIndex = 0;
            }
        }

        public static string[] GetAvailablePorts()
        {
            return SerialPort.GetPortNames();
        }

        public void Dispose()
        {
            Disconnect();
        }
    }

    public class AdcDataReceivedEventArgs : EventArgs
    {
        public int AdcValue { get; }
        public DateTime Timestamp { get; }

        public AdcDataReceivedEventArgs(int adcValue)
        {
            AdcValue = adcValue;
            Timestamp = DateTime.Now;
        }
    }
}
```

A.5 Data Logging

Listing 5: CSV DataLogger Implementation

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using RyanSensorApp.Models;

namespace RyanSensorApp.Services
{
    public class DataLogger
    {
        private List<SensorReading> _readings;
        private readonly object _lock = new object();

        public int Count => _readings.Count;

        public DataLogger()
        {
            _readings = new List<SensorReading>();
        }

        public void AddReading(SensorReading reading)
        {
            lock (_lock)
            {
                _readings.Add(reading);
            }
        }

        public List<SensorReading> GetAllReadings()
        {
            lock (_lock)
            {
                return new List<SensorReading>(_readings);
            }
        }

        public List<SensorReading> GetReadings(int count)
        {
            lock (_lock)
            {
                return _readings.Skip(Math.Max(0, _readings.Count - count)).ToList();
            }
        }

        public void Clear()
        {
            lock (_lock)
            {
                _readings.Clear();
            }
        }

        public void ExportToCsv(string filePath)
        {
            lock (_lock)
            {
                using (StreamWriter writer = new StreamWriter(filePath))
                {
                    writer.WriteLine(SensorReading.GetCsvHeader());
                    foreach (var reading in _readings)
                    {
                        writer.WriteLine(reading.ToString());
                    }
                }
            }
        }

        public (double min, double max, double avg, double stdDev) GetStatistics(bool forAdc = true)
        {
            lock (_lock)
            {
                if (_readings.Count == 0)
                    return (0, 0, 0, 0);

                double[] values = forAdc
                    ? _readings.Select(r => (double)r.AdcValue).ToArray()
                    : _readings.Select(r => r.Distance).ToArray();

                double min = values.Min();
                double max = values.Max();
                double avg = values.Average();

                // Calculate standard deviation
                double sumSquares = values.Sum(v => Math.Pow(v - avg, 2));
                double stdDev = Math.Sqrt(sumSquares / values.Length);

                return (min, max, avg, stdDev);
            }
        }

        public double GetRmsNoise(DateTime startTime, TimeSpan duration)
        {
            lock (_lock)
            {
                var filteredReadings = _readings
                    .Where(r => r.Timestamp >= startTime && r.Timestamp <= startTime.Add(duration))
                    .Select(r => r.Distance)
                    .ToArray();

                if (filteredReadings.Length == 0)
                    return 0;

                double avg = filteredReadings.Average();
                double sumSquares = filteredReadings.Sum(v => Math.Pow(v - avg, 2));
                return Math.Sqrt(sumSquares / filteredReadings.Length);
            }
        }
    }
}
```

A.6 Calibration Utilities

Listing 6: CalibrationService for Curve Fitting

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using RyanSensorApp.Models;
using MathNet.Numerics;
using MathNet.Numerics.LinearRegression;
using Newtonsoft.Json;

namespace RyanSensorApp.Services
{
    public class CalibrationService
    {
        public CalibrationData PerformCalibration(List<CalibrationPoint> points, FitType fitType)
        {
            if (points == null || points.Count < 2)
            {
                throw new ArgumentException("At least 2 calibration points are required.");
            }

            var calibration = new CalibrationData
            {
                Points = new List<CalibrationPoint>(points),
                FitType = fitType
            };

            // Extract x (ADC) and y (Distance) values
            double[] xData = points.Select(p => (double)p.AdcValue).ToArray();
            double[] yData = points.Select(p => p.Distance).ToArray();

            try
            {
                switch (fitType)
                {
                    case FitType.Linear:
                        calibration.Coefficients = FitLinear(xData, yData, out double rSquaredLinear);
                        calibration.RSquared = rSquaredLinear;
                        calibration.Equation = $"y = {calibration.Coefficients[0]:F6}x + {calibration.Coefficients[1]:F6}";
                        break;

                    case FitType.Polynomial2:
                        calibration.Coefficients = FitPolynomial(xData, yData, 2, out double rSquared2);
                        calibration.RSquared = rSquared2;
                        calibration.Equation = $"y = {calibration.Coefficients[0]:E3}x  + {calibration.Coefficients[1]:F6}x + {calibration.Coefficients[2]:F6}";
                        break;

                    case FitType.Polynomial3:
                        calibration.Coefficients = FitPolynomial(xData, yData, 3, out double rSquared3);
                        calibration.RSquared = rSquared3;
                        calibration.Equation = $"y = {calibration.Coefficients[0]:E3}x  + {calibration.Coefficients[1]:E3}x  + {calibration.Coefficients[2]:F6}x + {calibration.Coefficients[3]:F6}";
                        break;

                    case FitType.Power:
                        calibration.Coefficients = FitPower(xData, yData, out double rSquaredPower);
                        calibration.RSquared = rSquaredPower;
                        calibration.Equation = $"y = {calibration.Coefficients[0]:F6}  x^{calibration.Coefficients[1]:F6}";
                        break;

                    case FitType.Inverse:
                        calibration.Coefficients = FitInverse(xData, yData, out double rSquaredInv);
                        calibration.RSquared = rSquaredInv;
                        calibration.Equation = $"y = {calibration.Coefficients[0]:F6} / (x - {calibration.Coefficients[1]:F6}) + {calibration.Coefficients[2]:F6}";
                        break;
                }

                // Note: MinAdcThreshold and MaxAdcThreshold are set by user configuration,
                // not automatically from calibration points
            }
            catch (Exception ex)
            {
                throw new InvalidOperationException($"Curve fitting failed: {ex.Message}", ex);
            }

            return calibration;
        }

        private double[] FitLinear(double[] x, double[] y, out double rSquared)
        {
            var (slope, intercept) = SimpleRegression.Fit(x, y);
            rSquared = CalculateRSquared(x, y, new[] { slope, intercept }, FitType.Linear);
            return new[] { slope, intercept };
        }

        private double[] FitPolynomial(double[] x, double[] y, int order, out double rSquared)
        {
            double[] coefficients = Fit.Polynomial(x, y, order);
            // MathNet returns coefficients in ascending order (c0 + c1*x + c2*x^2 + ...)
            // We want descending order for our formula (a*x^n + b*x^(n-1) + ...)
            Array.Reverse(coefficients);
            rSquared = CalculateRSquared(x, y, coefficients, order == 2 ? FitType.Polynomial2 : FitType.Polynomial3);
            return coefficients;
        }

        private double[] FitPower(double[] x, double[] y, out double rSquared)
        {
            // Power fit: y = a * x^b
            // Transform to linear: ln(y) = ln(a) + b*ln(x)
            if (x.Any(val => val <= 0) || y.Any(val => val <= 0))
            {
                throw new ArgumentException("Power fit requires all positive values.");
            }

            double[] lnX = x.Select(val => Math.Log(val)).ToArray();
            double[] lnY = y.Select(val => Math.Log(val)).ToArray();

            var (b, lnA) = SimpleRegression.Fit(lnX, lnY);
            double a = Math.Exp(lnA);

            rSquared = CalculateRSquared(x, y, new[] { a, b }, FitType.Power);
            return new[] { a, b };
        }

        private double[] FitInverse(double[] x, double[] y, out double rSquared)
        {
            // Inverse fit: y = a / (x - b) + c
            // We'll use a simplified approach with b=0: y = a/x + c
            // Transform to linear: y = a*(1/x) + c

            double[] invX = x.Select(val => 1.0 / val).ToArray();
            var (a, c) = SimpleRegression.Fit(invX, y);
            double b = 0;

            rSquared = CalculateRSquared(x, y, new[] { a, b, c },
               FitType.Inverse);
            return new[] { a, b, c };
        }

        private double CalculateRSquared(double[] x, double[] y,
           double[] coefficients, FitType fitType)
        {
            double meanY = y.Average();
            double ssTotal = y.Sum(yi => Math.Pow(yi - meanY, 2));
            double ssResidual = 0;

            for (int i = 0; i < x.Length; i++)
            {
                double predicted = 0;
                switch (fitType)
                {
                    case FitType.Linear:
                        predicted = coefficients[0] * x[i] +
                           coefficients[1];
                        break;
                    case FitType.Polynomial2:
                        predicted = coefficients[0] * x[i] * x[i] +
                           coefficients[1] * x[i] + coefficients[2];
                        break;
                    case FitType.Polynomial3:
                        predicted = coefficients[0] * Math.Pow(x[i],
                            3) + coefficients[1] * Math.Pow(x[i], 2)
                            +
                                    coefficients[2] * x[i] +
                                       coefficients[3];
                        break;
                    case FitType.Power:
                        predicted = coefficients[0] * Math.Pow(x[i],
                            coefficients[1]);
                        break;
                    case FitType.Inverse:
                        predicted = coefficients[0] / (x[i] -
                           coefficients[1]) + coefficients[2];
                        break;
                }
                ssResidual += Math.Pow(y[i] - predicted, 2);
            }

            return 1 - (ssResidual / ssTotal);
        }

        public void SaveCalibration(CalibrationData calibration, string filePath)
        {
            try
            {
                string json = JsonConvert.SerializeObject(calibration, Formatting.Indented);
                File.WriteAllText(filePath, json);
            }
            catch (Exception ex)
            {
                throw new IOException($"Failed to save calibration: {ex.Message}", ex);
            }
        }

        public CalibrationData LoadCalibration(string filePath)
        {
            try
            {
                string json = File.ReadAllText(filePath);
                var calibration = JsonConvert.DeserializeObject<CalibrationData>(json);
                if (calibration == null)
                {
                    throw new InvalidDataException("Failed to deserialize calibration data.");
                }
                return calibration;
            }
            catch (Exception ex)
            {
                throw new IOException($"Failed to load calibration: {ex.Message}", ex);
            }
        }
    }
}
```

A.7 Noise Analysis Helpers

Listing 7: NoiseAnalysisService Statistics Module

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using RyanSensorApp.Models;

namespace RyanSensorApp.Services
{
    public class NoiseAnalysisService
    {
        private List<NoiseTestResult> _testResults;
        private int _nextTestNumber;

        public NoiseAnalysisService()
        {
            _testResults = new List<NoiseTestResult>();
            _nextTestNumber = 1;
        }

        public List<NoiseTestResult> GetAllTests()
        {
            return new List<NoiseTestResult>(_testResults);
        }

        public int GetTestCount()
        {
            return _testResults.Count;
        }

        public NoiseTestResult CreateNewTest(TestPosition position,
           string customDescription = "")
        {
            var test = new NoiseTestResult
            {
                TestNumber = _nextTestNumber++,
                Position = position,
                PositionDescription = customDescription,
                TestDateTime = DateTime.Now
            };
            return test;
        }

        public void SaveTest(NoiseTestResult test)
        {
            test.CalculateStatistics();
            _testResults.Add(test);
        }

        public void DeleteTest(int testNumber)
        {
            _testResults.RemoveAll(t => t.TestNumber == testNumber);
        }

        public void ClearAllTests()
        {
            _testResults.Clear();
            _nextTestNumber = 1;
        }

        public string GetComparisonSummary()
        {
            if (_testResults.Count == 0)
                return "No tests available for comparison.";

            var sb = new StringBuilder();
            sb.AppendLine("=== NOISE ANALYSIS COMPARISON ===\n");

            // Group by position
            var middleTests = _testResults.Where(t => t.Position == TestPosition.MiddleRange).ToList();
            var nearTests = _testResults.Where(t => t.Position == TestPosition.NearExtreme).ToList();
            var farTests = _testResults.Where(t => t.Position == TestPosition.FarExtreme).ToList();

            if (middleTests.Any())
            {
                sb.AppendLine("MIDDLE RANGE:");
                foreach (var test in middleTests)
                {
                    sb.AppendLine($"  Test #{test.TestNumber}: Mean={test.MeanDistance:F4} cm, RMS Noise={test.StandardDeviation:F4} cm");
                }
                sb.AppendLine($"  Average RMS Noise: {middleTests.Average(t => t.StandardDeviation):F4} cm\n");
            }

            if (nearTests.Any())
            {
                sb.AppendLine("NEAR EXTREME (Close):");
                foreach (var test in nearTests)
                {
                    sb.AppendLine($"  Test #{test.TestNumber}: Mean={test.MeanDistance:F4} cm, RMS Noise={test.StandardDeviation:F4} cm");
                }
                sb.AppendLine($"  Average RMS Noise: {nearTests.Average(t => t.StandardDeviation):F4} cm\n");
            }

            if (farTests.Any())
            {
                sb.AppendLine("FAR EXTREME:");
                foreach (var test in farTests)
                {
                    sb.AppendLine($"  Test #{test.TestNumber}: Mean={test.MeanDistance:F4} cm, RMS Noise={test.StandardDeviation:F4} cm");
                }
                sb.AppendLine($"  Average RMS Noise: {farTests.Average(t => t.StandardDeviation):F4} cm\n");
            }

            // Comparison analysis
            if (middleTests.Any() && (nearTests.Any() || farTests.Any()))
            {
                double middleRms = middleTests.Average(t => t.StandardDeviation);
                sb.AppendLine("=== COMPARISON ===");

                if (nearTests.Any())
                {
                    double nearRms = nearTests.Average(t => t.StandardDeviation);
                    double nearDiff = nearRms - middleRms;
                    double nearRatio = middleRms > 0 ? nearRms / middleRms : 0;
                    sb.AppendLine($"Near Extreme vs Middle: {nearDiff:+0.0000;-0.0000} cm ({nearRatio:F2}x)");
                }

                if (farTests.Any())
                {
                    double farRms = farTests.Average(t => t.StandardDeviation);
                    double farDiff = farRms - middleRms;
                    double farRatio = middleRms > 0 ? farRms / middleRms : 0;
                    sb.AppendLine($"Far Extreme vs Middle: {farDiff:+0.0000;-0.0000} cm ({farRatio:F2}x)");
                }
            }

            return sb.ToString();
        }

        public void ExportToCSV(string filePath)
        {
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                // Write header
                writer.WriteLine("Test Number,Position,Date/Time,Duration (s),Samples,Mean Distance (cm),Std Dev (RMS) (cm),Min Distance (cm),Max Distance (cm),Mean ADC");

                // Write data
                foreach (var test in _testResults)
                {
                    writer.WriteLine($"{test.TestNumber}," +
                                   $"{test.GetPositionString()}," +
                                   $"{test.TestDateTime:yyyy-MM-dd HH:mm:ss}," +
                                   $"{test.DurationSeconds:F2}," +
                                   $"{test.SampleCount}," +
                                   $"{test.MeanDistance:F6}," +
                                   $"{test.StandardDeviation:F6}," +
                                   $"{test.MinDistance:F6}," +
                                   $"{test.MaxDistance:F6}," +
                                   $"{test.MeanAdc:F2}");
                }
            }
        }

        public void ExportDetailedData(string filePath, NoiseTestResult test)
        {
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                // Write header info
                writer.WriteLine($"Test #{test.TestNumber} - {test.GetPositionString()}");
                writer.WriteLine($"Date/Time: {test.TestDateTime:yyyy-MM-dd HH:mm:ss}");
                writer.WriteLine($"Duration: {test.DurationSeconds:F2} seconds");
                writer.WriteLine($"Samples: {test.SampleCount}");
                writer.WriteLine($"Mean Distance: {test.MeanDistance:F6} cm");
                writer.WriteLine($"Standard Deviation (RMS Noise): {test.StandardDeviation:F6} cm");
                writer.WriteLine($"Min Distance: {test.MinDistance:F6} cm");
                writer.WriteLine($"Max Distance: {test.MaxDistance:F6} cm");
                writer.WriteLine();
                writer.WriteLine("Sample #,ADC Value,Distance (cm)");

                // Write all samples
                for (int i = 0; i < test.DistanceReadings.Count; i++)
                {
                    writer.WriteLine($"{i + 1},{test.AdcReadings[i]},{test.DistanceReadings[i]:F6}");
                }
            }
        }

        public Dictionary<TestPosition, double> GetAverageRMSByPosition()
        {
            var result = new Dictionary<TestPosition, double>();

            var positions = new[] { TestPosition.MiddleRange, TestPosition.NearExtreme, TestPosition.FarExtreme };

            foreach (var pos in positions)
            {
                var tests = _testResults.Where(t => t.Position == pos).ToList();
                if (tests.Any())
                {
                    result[pos] = tests.Average(t => t.StandardDeviation);
                }
            }

            return result;
        }
    }
}
```