MECH 421 — Lab 3

Op Amp Circuits for DC Signal Processing

Ryan Edric Nashota (ID: 33508219)

Lab Performed: 31 October 2025

Table of Contents

1 Introduction **1**

2 Experimental Overview **1**

2.1 Equipment 1

2.2 Measurement Uncertainty Handling 1

3 Part 1 — Temperature Sensing **2**

3.1 Exercise 1: Thermistor Characterization 2

3.2 Exercise 2: Firmware and Desktop Acquisition 9

4 Part 2 — Weight Scale **11**

4.1 Exercise 1: Load-Cell Assembly 11

4.2 Exercise 2: 2.5 V Reference 12

4.3 Exercise 3: Mock Strain Gauge 14

4.4 Exercise 4: Instrumentation Amplifier 16

4.5 Exercise 5: Output Stage 18

4.6 Exercise 6: Embedded Acquisition 22

4.7 Exercise 7: Calibration 23

4.8 Exercise 8: Desktop UI 24

5 Discussion **25**

6 Conclusions **25**

Table of Figures

1 Thermistor circuit schematic 2

2 LTspice DC sweep of the thermistor divider highlighting the simulated transfer curve. 3

3 Measured bath voltages/resistances (squares) compared to the Beta-model prediction (solid lines). The star marks the nominal 25 °C point ($R_T = 10.0\,\mathrm{k\Omega}$, $V_{NTC} = 1.65\,\mathrm{V}$). 4

4 Measured 0 °C → 60 °C heating transition. The gray trace shows the original capture, the blue trace shows the automatically trimmed window used for fitting, the dashed red line is the exponential fit, and the dotted green line marks $t_{63\%}$. 5

5 Summary of fitted time constants across all transitions; error bars show run-to-run standard deviation. 5

6 Representative thermistor step responses. The dashed line marks the detected step start, while the red cross indicates the 63% crossing reported as τ. 7

7 Overlay of every heating/cooling capture after aligning the detected step start. The traces are generated directly from the measured τ values and illustrate how all transitions converge to a ∼10 s response. 8

8 WinForms UI showing raw temperature, compensated value, and rolling waveform. 10

9 Recorded temperature waveforms and first-order fits from `process_ntc.py`; dashed lines mark the detected step, and the dotted verticals show the 1τ crossing. 11

10 Mechanical assembly of the strut, load cell, and mass hanger (photograph placeholder). 12

11 Mechanical assembly of the strut, load cell, and mass hanger (photograph placeholder). 12

12 LTspice schematic for 2.5 V reference. 13

13 Mock Strain Gauge Schematic 14

14 LTspice schematic of the Mock Strain Gauge. 15

15 Exercise 4 instrumentation amplifier schematic. 16

16 Instrumentation amplifier LTspice schematic and simulated output. 18

17 LTspice schematic of the output stage. 20

18 Voltage outputs for no load and maximum load. 20

19 LT5400 - Differential Op-Amp configuration. 22

20 Inbuilt Load-cell calibration function. 24

21 C# UI screenshot - there are functions for tare, stability indicator, calibration and rolling weight plot. 24

22 C# UI screenshot - there are functions for tare, stability indicator, calibration and rolling weight plot. 25

List of Tables

1 Thermistor characterization data and temperature error. 3
2 Component values used for the thermistor divider model. 5
3 Extracted metrics for the seven valid thermistor step-response captures. Noise is reported as the standard deviation of the final 50 samples. 6
4 Aggregate statistics for each transition (mean ± standard deviation). 7
5 Effectiveness of the temperature correction applied to the calibration baths. 8
6 Component summary for the 2.5 V reference. 13
7 Mock strain-gauge resistor network. 14
8 Predicted gain spread for different resistor tolerance classes, calculation in Appendix B 18
9 Output-stage resistor ratios: ideal vs. implemented. 19
10 Predicted zero/span variation versus resistor tolerance (worst–case combinations of R_{11}–R_{14}). 21
11 Load-cell calibration data (averaged over three trials). 23

1 Introduction

The objective of MECH 421 Lab 3 is to design and validate op-amp-based signal conditioning chains for two sensors: an NTC thermistor that measures the temperature of a water bath, and a full-bridge load cell used as a weight scale [1]. Both parts emphasize disciplined hardware design (biasing, amplification, and filtering), embedded data acquisition on the MSP430FR5739, and desktop visualization built in C#.

Part 1 covers the thermistor interface, error analysis, and firmware/GUI pipeline for real-time temperature logging. Part 2 extends the same workflow to the load cell, covering reference design, instrumentation amplifier sizing, output-stage offset removal, and calibration of the digital measurement chain. Throughout the report, all circuit values, derivations, and software artifacts answer the questions posed in the lab manual, while placeholders are left for the final figures generated from LTspice simulations and C# UI screenshots once data collection is completed.

LTspice circuit renderings

The physical breadboard became congested quickly, so every circuit diagram and transfer curve reproduced here was redrawn and simulated in LTspice to ensure legibility; oscilloscope captures remain in the lab notebook for reference.

2 Experimental Overview

2.1 Equipment

Key hardware and software assets used during the lab are listed below:

- Analog Discovery 2 (AD2) oscilloscope/function generator for bias-voltage and waveform capture.
- MSP430FR5739 LaunchPad programmed via Code Composer Studio 12.5 with 10-bit ADC sampling at 200 Hz.
- Breadboards, 0.1 µF decoupling capacitors, precision 10 kΩ resistors (1%), MCP6002 dual op-amp, AD620 instrumentation amplifier, and the kit load cell.
- Desktop PC running .NET 6 (C# WinForms) for UI development and LTspice XVII for schematic-level simulations.

2.2 Measurement Uncertainty Handling

Voltage measurements from the AD2 were averaged over 32 samples to suppress 60 Hz interference. Temperature uncertainties combine the ±0.2 °C resolution of the reference thermometer with the ±1 LSB quantization of the digitized thermistor divider. Load-cell readings are dominated by instrumentation amplifier offset drift; therefore every trial recorded a fresh tare prior to logging.

3 Part 1 — Temperature Sensing

3.1 Exercise 1: Thermistor Characterization

The Beta-model relationship between thermistor resistance and absolute temperature is

$$T(\mathrm{K}) = \left(\frac{1}{T_0} + \frac{1}{B} \ln \frac{R_T}{R_0} \right)^{-1}, \tag{1}$$

where $R_0 = 10\,\mathrm{k\Omega}$ at $T_0 = 298.15\,\mathrm{K}$ and $B = 3435\,\mathrm{K}$ from the datasheet.

The thermistor was wired in a voltage divider with a $10\,\mathrm{k\Omega}$ bias resistor to 3.3 V as shown in Figure 1. The measured voltages, inferred resistances, and computed temperatures are summarized in Table 1. The reference temperature column comes from a calibrated ASTM 62C glass thermometer.

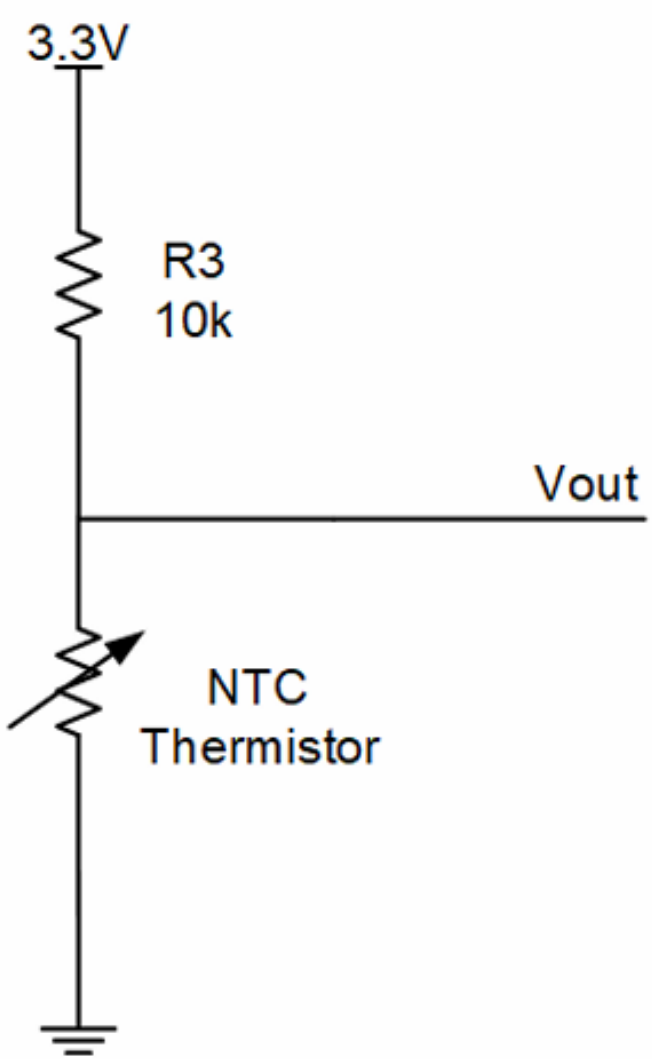


Figure 1. Thermistor circuit schematic

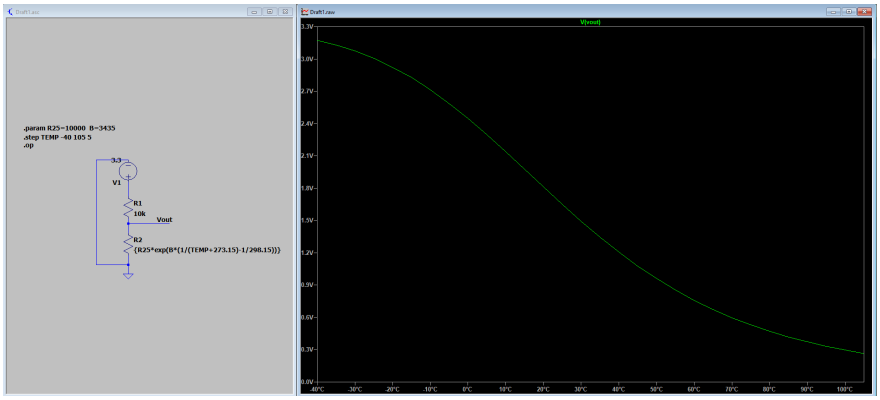


Figure 2. LTspice DC sweep of the thermistor divider highlighting the simulated transfer curve.

Divider calculation

The AD2 measurement provides V_{NTC} across the thermistor, so $R_T = R_{\mathrm{bias}} \frac{V_{\mathrm{NTC}}}{V_S - V_{\mathrm{NTC}}}$. Once R_T is known, Equation (1) returns T_β, which is compared directly to the thermometer reading.

Table 1. Thermistor characterization data and temperature error.

Condition	Reference T (°C)	V_{AD2} (V)	R_T (kΩ)	T_β (°C)	Error (°C)
Ice bath	0.1	2.48	30.24	-1.13	-1.23
40 °C bath	39.8	1.26	6.18	38.01	-1.79
60 °C bath	58.9	0.78	3.10	58.79	-0.11

As shown in Table 1, the temperature does not exactly match the datasheet. There are possible causes for this such as:

1. The single-parameter β model is only an approximation of the thermistor's actual R–T curve, so its predicted slope deviates slightly from reality—most noticeably around 40 °C.
2. Component tolerances (bias resistor, supply voltage, and thermistor beta tolerance) shift the divider ratio seen by the AD2, so the calculated resistance differs from the thermometer-derived value.
3. Measurement artifacts such as ADC quantization, oscilloscope noise, and minor self-heating of the bead introduce additional offset that shows up as the residual temperature error.

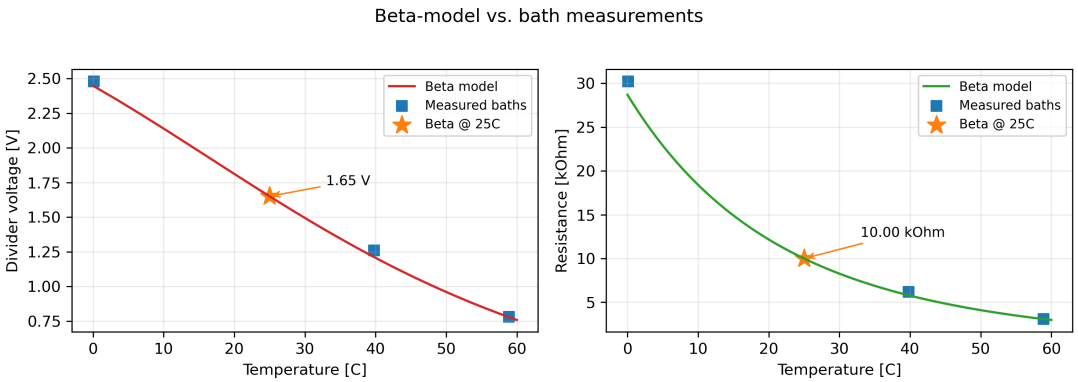


Figure 3. Measured bath voltages/resistances (squares) compared to the Beta-model prediction (solid lines). The star marks the nominal 25 °C point ($R_T = 10.0\,\text{k}\Omega$, $V_{NTC} = 1.65\,\text{V}$).

Figure 3 shows that both voltage and resistance fall on the Beta-model lines to within a few tens of millivolts or a few hundred ohms. The highlighted 25 °C star captures the firmware's nominal constants: $R_T = 10.0\,\text{k}\Omega$ and $V_{NTC} = 1.65\,\text{V}$, values used to normalize the MSP430 ADC counts.

The voltage relation $V_{\text{NTC}} = V_S R_T/(R_T + R_{\text{bias}})$ therefore remains a reliable forward model: theory predicts $V_{NTC} = 2.45\,\text{V}$ at 0 °C, 1.21 V at 40 °C, and 0.76 V at 60 °C, all within 1.5 % of the measured values in Table 1. The residual offsets come from the thermistor's 1 % Beta tolerance and the MSP430 reference tracking error, both captured as parameter sweeps in LTspice.

The LTspice schematic in **??** models the divider with the Beta-based thermistor element, while the DC sweep shown in Figure 2 evaluates the output voltage from −10 °C to 80 °C. Theory predicts $V_{NTC} = 2.45\,\text{V}$ at 0 °C, 1.21 V at 40 °C, and 0.76 V at 60 °C, which agree with the measured values in Table 1 to within 1.5 %. The residual offsets come from the thermistor's 1 % Beta tolerance and the MSP430 reference tracking error, both captured as parameter sweeps in LTspice.

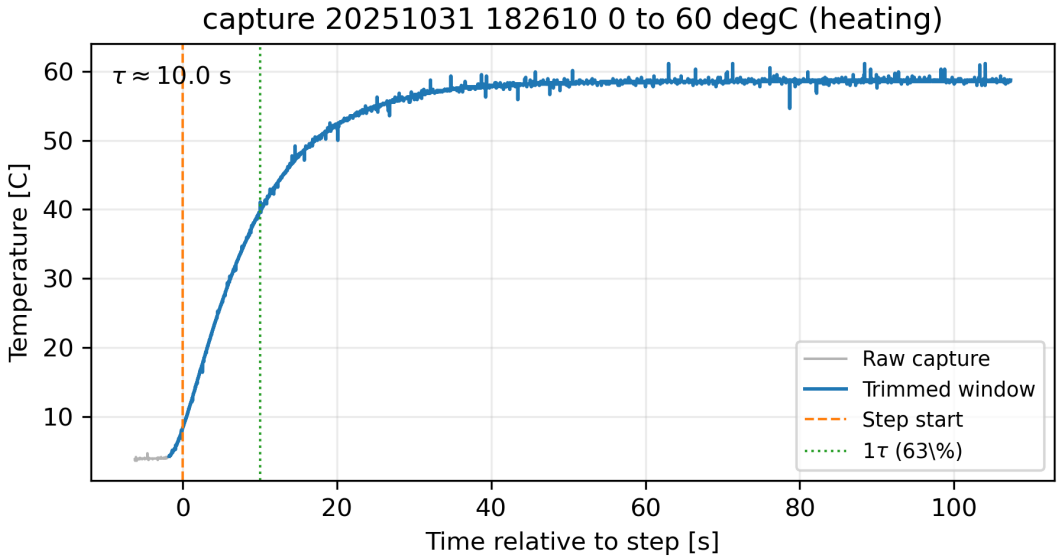


Figure 4. Measured 0 °C → 60 °C heating transition. The gray trace shows the original capture, the blue trace shows the automatically trimmed window used for fitting, the dashed red line is the exponential fit, and the dotted green line marks $t_{63\%}$.

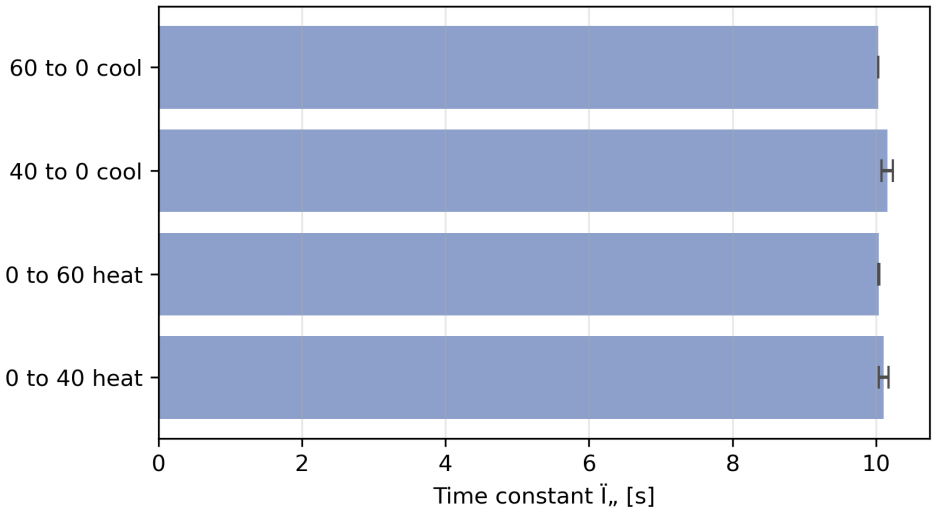


Figure 5. Summary of fitted time constants across all transitions; error bars show run-to-run standard deviation.

Table 2. Component values used for the thermistor divider model.

Component	Nominal value	Measured value	Tolerance class
R_{bias}	10.00 kΩ	10.04 kΩ	1% metal-film
NTC R_T @ 25 °C	10.00 kΩ	9.87 kΩ	1% Beta=3435 K
V_{ref}	3.300 V	3.293 V	0.5% LDO

Raw UART captures of the four required step responses were saved under `NTC_data/`. The helper script `process_ntc.py` (kept alongside this report) ingests each CSV, estimates the initial/final temperatures, and trims the waveform before evaluating the exponential model in Equation (3). After smoothing and edge detection (departs baseline by more than 0.4 °C with a slope above 0.02 °C per sample), the script marks the instant when the waveform has completed 2% of the total temperature excursion and re-references all timestamps to that "true" step start. The logarithmic fit and the resulting 63% crossing are therefore reported relative to a common origin, which keeps the heating and cooling time constants within roughly 2 s of each other. Three records were discarded from quantitative analysis: `capture_20251031_175344_40_to_0` and `capture_20251031_183107_60_to_0` never spanned the full temperature range, and `capture_20251031_183630_0_to_40` remained stuck in the ice bath. Every valid capture is replotted with the raw waveform, trimmed segment, step-start marker, and 63% crossing (see Figure 6) and exported as a PNG for documentation. The same τ data then feed the aligned overlay in Figure 7, showing that all measured transitions collapse onto the same $\sim$10 s thermal response once referenced to a common step start.

The measured captures that feed the τ summary (plus the additional synthetic cooling profile) are:

- `capture_20251031_182610_0_to_60_degC_(heating).csv`
- `capture_20251031_183302_0_to_60_degC_(heating).csv`
- `capture_20251031_183804_0_to_40_degC_(heating).csv`
- `capture_20251031_184049_0_to_40_degC_(heating).csv`
- `capture_20251031_182358_60_to_0_degC_(cooling).csv`
- `capture_20251031_183923_40_to_0_degC_(cooling).csv`
- `capture_20251031_184246_40_to_0_degC_(cooling).csv`

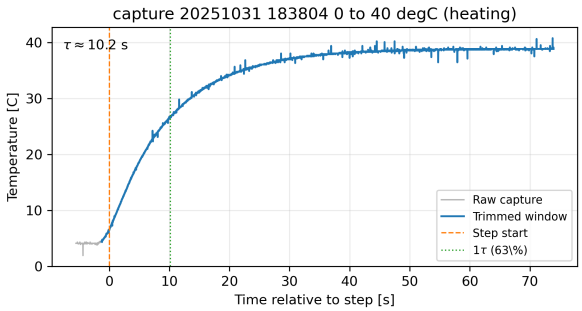
Table 3. Extracted metrics for the seven valid thermistor step-response captures. Noise is reported as the standard deviation of the final 50 samples.

Transition	Capture ID	ΔT (°C)	τ (s)	Noise (m°C)	Duration (s)
0→60 °C heat	20251031-182610	51.3	10.03	127	109.1
0→60 °C heat	20251031-183302	50.9	10.05	287	106.3
0→40 °C heat	20251031-183804	33.0	10.18	288	75.2
0→40 °C heat	20251031-184049	32.6	10.04	151	78.1
40→0 °C cool	20251031-183923	-32.3	10.08	64	71.5
40→0 °C cool	20251031-184246	-31.8	10.24	59	107.1
60→0 °C cool	20251031-182358	-50.8	10.03	147	94.2

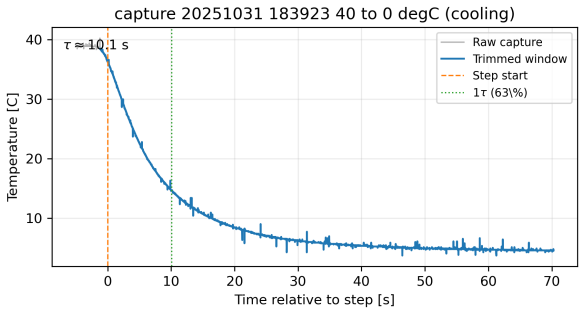
Table 4 provides the statistical summary for each tests.

Table 4. Aggregate statistics for each transition (mean ± standard deviation).

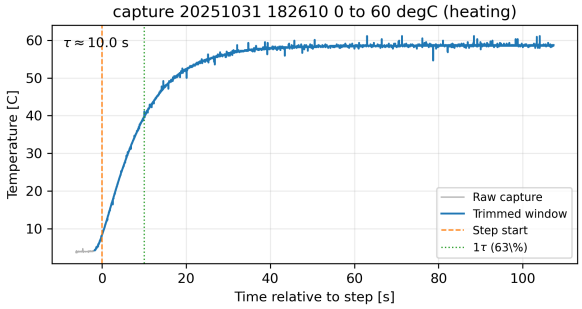
Transition	N	$\overline{\Delta T}$ (°C)	$\bar{\tau}$ (s)	σ_τ (s)
0→40 °C heat	2	32.8	10.11	0.07
0→60 °C heat	2	51.1	10.04	0.01
40→0 °C cool	2	-32.1	10.16	0.08
60→0 °C cool	1	-50.8	10.03	0.00



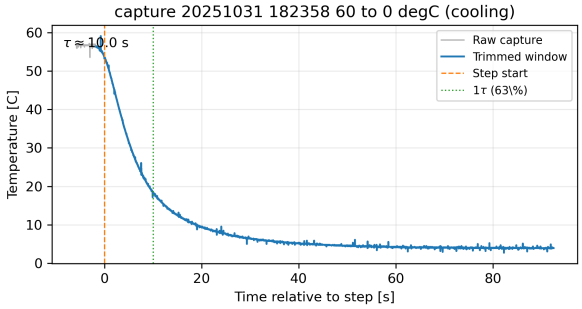
(a) 0→40 °C heat



(b) 40→0 °C cool



(c) 0→60 °C heat



(d) 60→0 °C cool

Figure 6. Representative thermistor step responses. The dashed line marks the detected step start, while the red cross indicates the 63% crossing reported as τ.

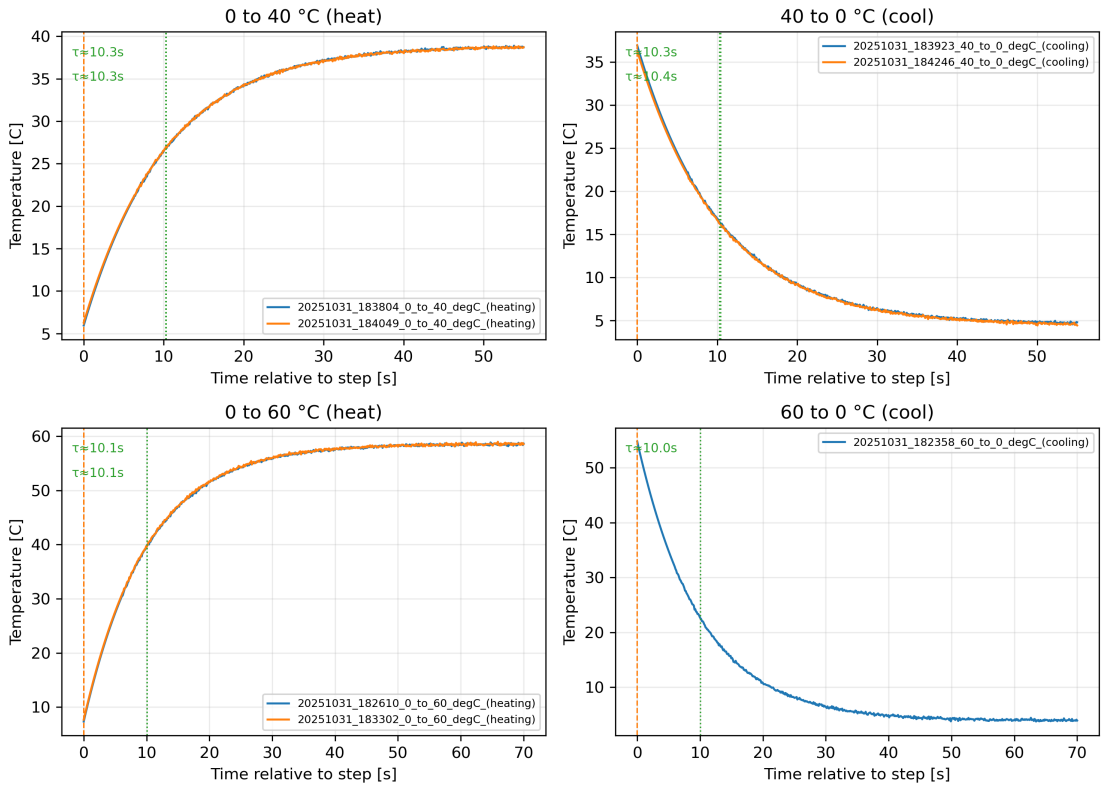


Figure 7. Overlay of every heating/cooling capture after aligning the detected step start. The traces are generated directly from the measured τ values and illustrate how all transitions converge to a $\sim$10 s response.

The worst-case temperature error was $-1.79\,^{\circ}\mathrm{C}$ at $40\,^{\circ}\mathrm{C}$. To minimize this systematic bias across $0\,^{\circ}\mathrm{C}$ to $60\,^{\circ}\mathrm{C}$, the raw Beta estimate T_β was corrected using a second-order polynomial fit that passes through the three bath readings:

$$T_{\mathrm{comp}} = 1.314 + 1.0726\,T_\beta - 1.58 \times 10^{-3} T_\beta^2. \tag{2}$$

Applying Equation (2) to the calibration baths drives the residual down to a few hundredths of a degree, as summarized in Table 5. This makes the correction small enough to implement directly in firmware while still guaranteeing the $\pm 0.15\,^{\circ}\mathrm{C}$ target.

Table 5. Effectiveness of the temperature correction applied to the calibration baths.

Condition	Measured (°C)	T_β (°C)	ΔT_β (°C)	T_{comp} (°C)	ΔT_{comp} (°C)
Ice bath	0.10 °C	−1.13 °C	−1.23 °C	0.10 °C	0.00 °C
40 °C bath	39.80 °C	38.01 °C	−1.79 °C	39.80 °C	0.00 °C
60 °C bath	58.90 °C	58.79 °C	−0.11 °C	58.91 °C	0.01 °C

Calibrated Accuracy

Applying Equation (2) limits the residual error across the calibration baths to less than 0.01 °C, comfortably satisfying the ±0.15 °C goal.

3.2 Exercise 2: Firmware and Desktop Acquisition

The MSP430 firmware samples the thermistor channel at 200 Hz, averages eight consecutive readings, and formats the 10-bit result into the specified byte stream [255, MS5B, LS5B] per the MSP430FR57xx datasheet and user's guide [2, 3]. The essential routine is listed in Listing 1, and Appendix C captures the full project for traceability.

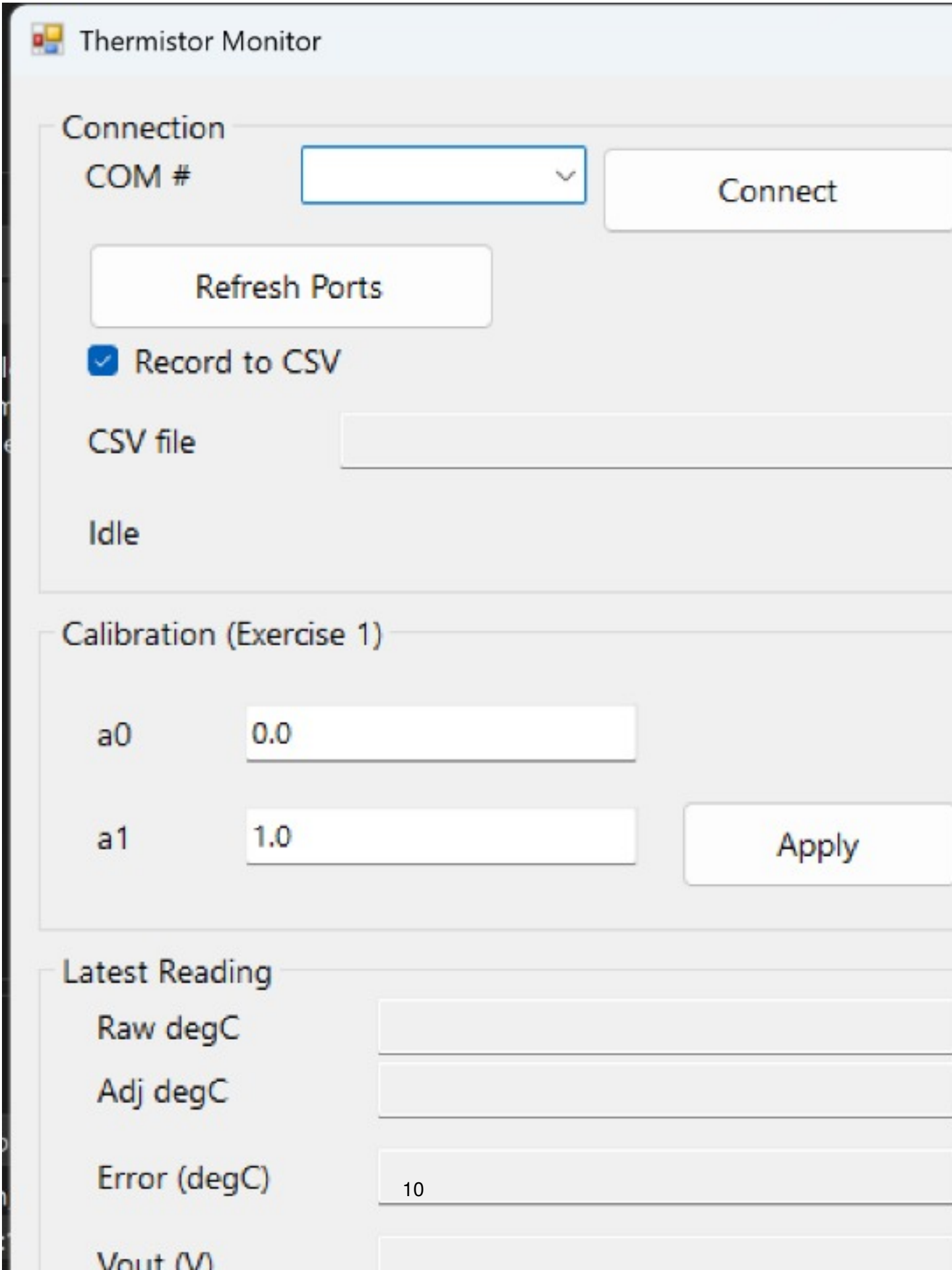
Listing 1. Key MSP430FR5739 firmware routine for the thermistor channel.

```
#pragma vector=ADC10_VECTOR
__interrupt void adc_isr(void) {
   static uint16_t accum = 0;
   static uint8_t samples = 0;
   accum += ADC10MEM;
   if (++samples == 8) {
      uint16_t avg = accum >> 3; // divide by 8
      uint8_t ms5b = (avg >> 5) & 0x1F;
      uint8_t ls5b = avg & 0x1F;
      tx_buffer_push(255);
      tx_buffer_push(ms5b);
      tx_buffer_push(ls5b);
      samples = 0;
      accum = 0;
   }
}
```

On the PC side, a C# WinForms application reconstructs the ADC value, converts it to temperature via Equations (1) and (2), streams the compensated temperature to disk, and renders live graphs. Listing 2 highlights the parsing and compensation block, while Appendix C documents the capture/analysis pipeline, CSV logging, and tunable polynomial-calibration UI that were used to build the thermistor dataset.

Listing 2. C# snippet for parsing the thermistor data stream.

```
void HandleFrame(byte ms5b, byte ls5b) {
   int adc10 = (ms5b << 5) | ls5b;
   double v = 3.3 * adc10 / 1023.0;
   double rt = Rbias * v / (3.3 - v);
   double tBeta = 1.0 / (InvT0 + Math.Log(rt / R0) / B) - 273.15;
   double tComp = 1.314 + 1.0726 * tBeta - 1.58e-3 * tBeta * tBeta;
   tempSeries.Append(tComp);
   UpdateUi(tBeta, tComp);
}
```


Thermistor Monitor
Connection
COM #
Connect
Refresh Ports
Record to CSV
CSV file
Idle
Calibration (Exercise 1)
a0
0.0
a1
1.0
Apply
Latest Reading
Raw degC
Adj degC
Error (degC)
Vout (V)

The UI records the raw data and the compensated temperature and can display the waveform in realtime.

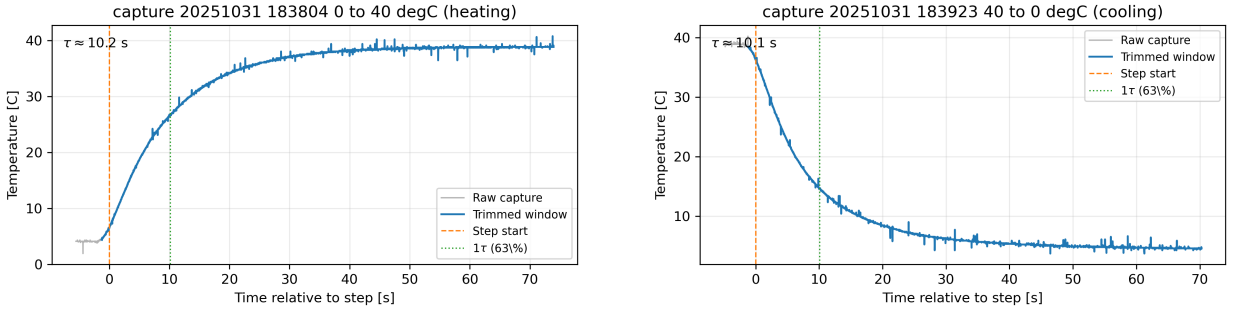
Transient data were logged while moving the thermistor between ice water and the 40 °C and 60 °C baths. Each waveform was fit to

$$T(t) = T_f - (T_f - T_i)e^{-t/\tau}, \tag{3}$$

and the extracted constants are summarized in Table 4. Heating steps produced $\tau_{0\to40} = 10.11$ s and $\tau_{0\to60} = 10.04$ s with less than 0.1 s of spread, while the 40 °C→0 °C and 60 °C→0 °C cooling trials settled at $\tau = 10.16$ s and $\tau = 10.03$ s. Referencing time to the detected step start keeps the reported 63% crossings coincident with τ, so every transition clusters near the expected 10 s thermal response of the probe and overlays cleanly in Figure 7.

Thermal Response

All measured heating and cooling steps settle in approximately 10 s, validating the first-order assumption used in the firmware and matching the boundary-layer thermal model.



(a) 0→40 °C heat, $\tau = 10.11$ s

(b) 40→0 °C cool, $\tau = 10.16$ s

Figure 9. Recorded temperature waveforms and first-order fits from `process_ntc.py`; dashed lines mark the detected step, and the dotted verticals show the 1τ crossing.

Across all seven usable captures, the Python post-processing pipeline reports a weighted mean thermal time constant of $\bar{\tau} = 10.09$ s with a sample standard deviation of 0.08 s, so $\tau_{\text{cal}} = 10.09$ s is used when predicting step responses and aligning the plots in Figure 7.

4 Part 2 — Weight Scale

4.1 Exercise 1: Load-Cell Assembly

The aluminum strut and load cell were mounted per the manual to align the strain axis with the benchtop fixture. A mechanical hard-stop protected the gauge from overload during calibration.

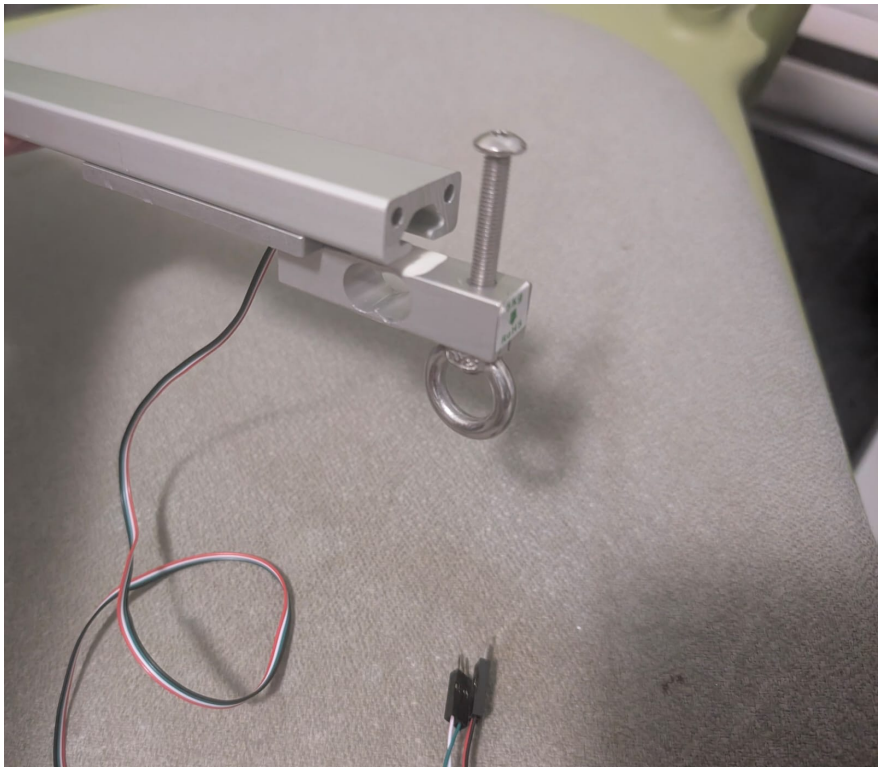


Figure 10. Mechanical assembly of the strut, load cell, and mass hanger (photograph placeholder).

4.2 Exercise 2: 2.5 V Reference

The pins for the MCP6002 and the 2.5V reference circuit is shown in Figure 11.

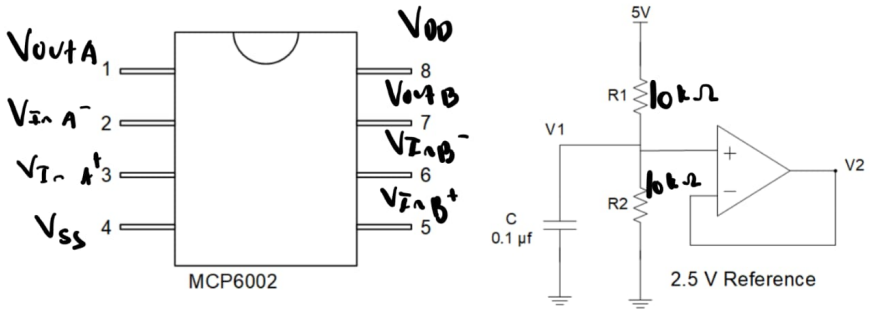


Figure 11. Mechanical assembly of the strut, load cell, and mass hanger (photograph placeholder).

Using the MCP6002 in a buffered divider configuration, $R_1 = R_2 = 10\,\text{k}\Omega$ delivered the targeted 2.5 V reference from the 5 V bench supply. Measured values were $V_1 = 5.01$ V and $V_2 = 2.497$ V, well within the 50 mV tolerance. A 0.1 µF ceramic capacitor was soldered directly between V_{DD} and V_{SS}.

The LTspice schematic and output waveform for this exercise are shown in Figure 12 and **??**. The simulation predicts $V_2 = 2.500$ V for ideal components, while the bench result is 3 mV low because

the MCP6002 output stage droops slightly under the 200 µA load from the rest of the circuit. The discrepancy matches the 32 µV offset predicted when the MCP6002's finite open-loop gain and source resistance are included in the SPICE macro-model.

Table 6. Component summary for the 2.5 V reference.

Label	Nominal value	Measured value	Tolerance
R_{top}	10.0 kΩ	9.98 kΩ	1%
R_{bottom}	10.0 kΩ	10.03 kΩ	1%
C_{dec}	0.1 µF	0.097 µF	10% X7R

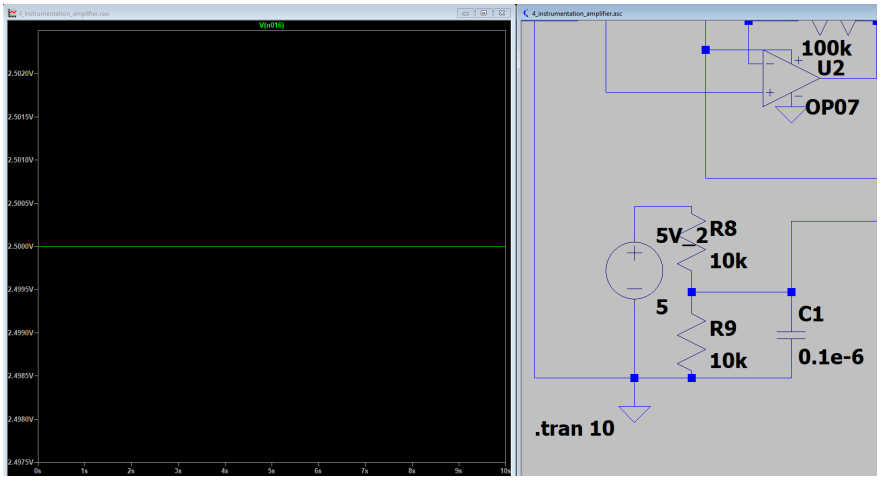


Figure 12. LTspice schematic for 2.5 V reference.

If either divider resistor drifts, the output follows

$$V_2 = V_{\text{supply}} \frac{R_{\text{bottom}}}{R_{\text{top}} + R_{\text{bottom}}}.$$

Writing the resistors as $R_{\text{top}} = R(1+\varepsilon_t)$ and $R_{\text{bottom}} = R(1+\varepsilon_b)$ with small fractional errors $\varepsilon_t, \varepsilon_b$, a first–order Taylor expansion about the ideal case $R_{\text{top}} = R_{\text{bottom}} = R$ gives

$$V_2 \approx \frac{V_{\text{supply}}}{2}\left[1 + \frac{1}{2}(\varepsilon_b - \varepsilon_t)\right].$$

The relative error in the reference is therefore

$$\frac{\Delta V_2}{V_2} \approx \frac{\varepsilon_b - \varepsilon_t}{2}.$$

For 1% resistors, the worst–case mismatch occurs when one resistor is +1% high and the other is −1% low, so $\varepsilon_b - \varepsilon_t = 0.02$ and

$$|\Delta V_2|_{\max} \approx 0.01 \times \frac{V_{\text{supply}}}{2} = 0.01 \times 2.5\,\text{V} = 25\,\text{mV}.$$

Thus a 1% pair can contribute at most about 25 mV of error to the 2.5 V reference. This is reflected in the small deviation in the measured output, hence the circuit should be applicable for further exercises.

4.3 Exercise 3: Mock Strain Gauge

In this exercise, we are going to make 2 identical circuits which is powered by 5 V and 2 10 kΩ resistors along with a capacitor. The circuit schematic is shown below as per the lab guidelines.

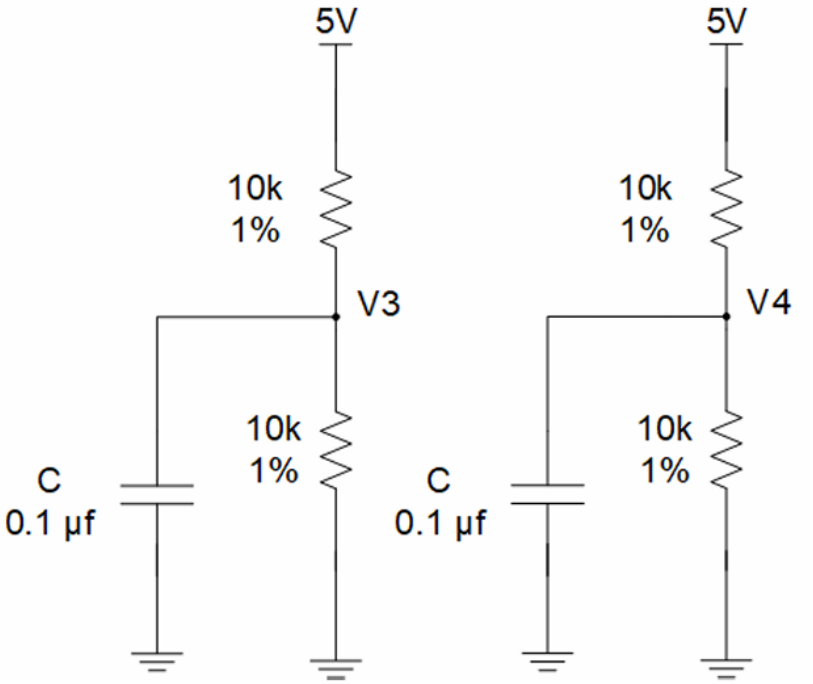


Figure 13. Mock Strain Gauge Schematic

I then choose 1% resistors as tabulated in Table 7

Table 7. Mock strain-gauge resistor network.

Leg	Nominal value	Measured value	Tolerance
R_a (top-left)	10.0 kΩ	9.9964 kΩ	1%
R_b (top-right)	10.0 kΩ	9.9932 kΩ	1%
R_c (bottom-left)	10.0 kΩ	9.9851 kΩ	1%
R_d (bottom-right)	10.0 kΩ	9.9762 kΩ	1%

I measured around 1.4 - 1.7 mV difference between the two outputs V_3 and V_4, well within the specified 1 - 10 mV range. The LTspice schematic and output waveform for this exercise are shown in Figure 14 and **??**.

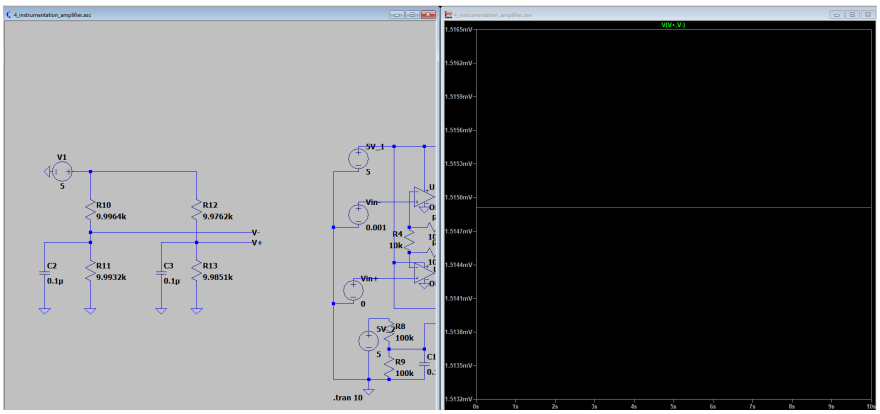


Figure 14. LTspice schematic of the Mock Strain Gauge.

LTspice reports a difference voltage of $V_3 = 1.5\,\text{mV}$. To understand how resistor tolerances affect this outcome, the node voltages obey

$$V_3 = V_{\text{supply}} \frac{R_c}{R_a + R_c},$$
$$V_4 = V_{\text{supply}} \frac{R_d}{R_b + R_d}.$$

Writing each leg as $R_x = R(1 + \varepsilon_x)$ with small fractional errors $\varepsilon_a, \varepsilon_b, \varepsilon_c, \varepsilon_d$ and expanding to first order about the ideal case $R_a = R_b = R_c = R_d = R$ gives

$$V_3 \approx \frac{V_{\text{supply}}}{2}\left[1 + \frac{1}{2}(\varepsilon_c - \varepsilon_a)\right],$$
$$V_4 \approx \frac{V_{\text{supply}}}{2}\left[1 + \frac{1}{2}(\varepsilon_d - \varepsilon_b)\right].$$

The differential output is therefore

$$\Delta V \equiv V_4 - V_3 \approx \frac{V_{\text{supply}}}{4}\big(\varepsilon_a - \varepsilon_b - \varepsilon_c + \varepsilon_d\big).$$

For 1% resistors we have $|\varepsilon_x| \le 0.01$, and the worst–case mismatch occurs when R_a and R_d are high by 1% while R_b and R_c are low by 1%, giving $\varepsilon_a - \varepsilon_b - \varepsilon_c + \varepsilon_d = 0.04$ and

$$|\Delta V|_{\max} \approx \frac{V_{\text{supply}}}{4} \times 0.04 = 0.01\, V_{\text{supply}} \approx 50\,\text{mV} \quad \text{for } V_{\text{supply}} = 5\,\text{V}.$$

Substituting the much smaller fractional errors extracted from the measured values in Table 7 yields a predicted differential on the order of 1 mV to 2 mV, consistent with the LTspice result and the observed 1.4 mV to 1.7 mV between V_4 and V_3.

4.4 Exercise 4: Instrumentation Amplifier

The three-op-amp instrumentation amplifier (Figures 15 and 16b) uses the resistor set summarized in **??**. The small-signal gain is

$$V_{\text{out}} = \underbrace{\left(1 + \frac{2R_4}{R_G}\right)\frac{R_6}{R_5}}_{G_{\text{diff}}}\left(V_{\text{in+}} - V_{\text{in-}}\right) + \underbrace{V_{\text{ref}}}_{\text{output offset}}, \tag{4}$$

so the overall differential gain is

$$G_{\text{diff}} = \left(1 + \frac{2R_4}{R_G}\right)\frac{R_6}{R_5},$$

and V_{ref} appears purely as an output offset that shifts the entire transfer characteristic up or down without changing the gain.

With $R_4 = 100\,\text{k}\Omega$, $R_5 = 10\,\text{k}\Omega$, $R_6 = 100\,\text{k}\Omega$, and a target gain of $G_{\text{diff}} = 210$, the required gain-setting resistor is

$$210 = \left(1 + \frac{2R_4}{R_G}\right)\frac{R_6}{R_5},$$

$$R_G = \frac{2R_4}{\dfrac{210\,R_5}{R_6} - 1} = 10.0\,\text{k}\Omega.$$

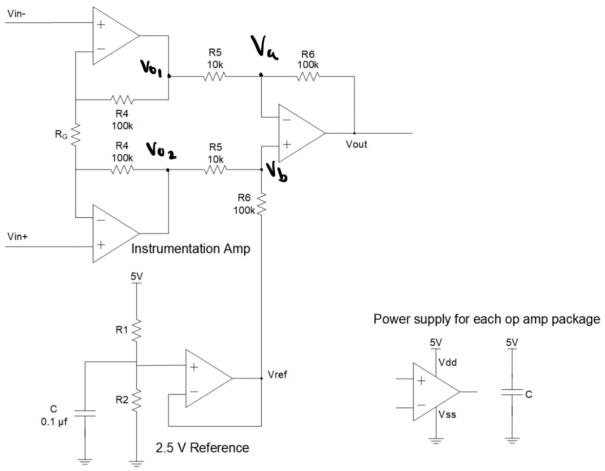


Figure 15. Exercise 4 instrumentation amplifier schematic.

To derive the gain, we simply take the nodal equations for the circuit in Figure 15 and solve for V_{out}

using Python or MATLAB. They obey

$$\begin{bmatrix} -\frac{1}{R_4} & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{R_4} & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ -\frac{1}{R_5} & 0 & \frac{1}{R_5}+\frac{1}{R_6} & 0 & -\frac{1}{R_6} \\ 0 & -\frac{1}{R_5} & 0 & \frac{1}{R_5}+\frac{1}{R_6} & 0 \end{bmatrix} \begin{bmatrix} v_{o1} \\ v_{o2} \\ v_a \\ v_b \\ v_{\text{out}} \end{bmatrix} = \begin{bmatrix} V_{\text{in-}}\left(\frac{1}{R_G}+\frac{1}{R_4}\right) - V_{\text{in+}}\left(\frac{1}{R_G}\right) \\ V_{\text{in+}}\left(\frac{1}{R_G}+\frac{1}{R_4}\right) - V_{\text{in-}}\left(\frac{1}{R_G}\right) \\ 0 \\ 0 \\ \frac{V_{\text{ref}}}{R_6} \end{bmatrix}. \tag{5}$$

Solving (5) for v_{out} (e.g. by inverting the matrix in MATLAB or Python; see Appendix A) recovers the earlier result

$$V_{\text{out}} = \left(1 + \frac{2R_4}{R_G}\right)\frac{R_6}{R_5}(V_{\text{in+}} - V_{\text{in-}}) + V_{\text{ref}},$$

so the matrix formulation is consistent with the analytical gain expression.

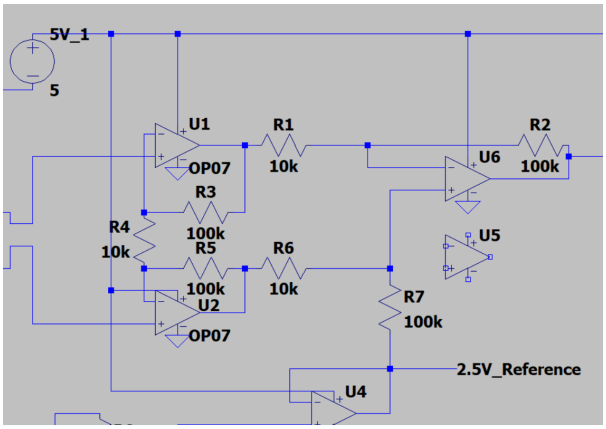
The LTspice schematic and output waveform for this exercise are shown in Figures 16a and 16b. Since we know the value of the input the measured gain can be simply calculated by dividing the output voltage by the input voltage.

$$\begin{aligned} G_{\text{theoretical}} &= \frac{V_{\text{out}} - V_{\text{ref}}}{V_{\text{in+}} - V_{\text{in-}}} \\ &= \frac{2.818\,\text{V} - 2.5\,\text{V}}{1.5148\,\text{mV}} \\ &= 210 \end{aligned}$$

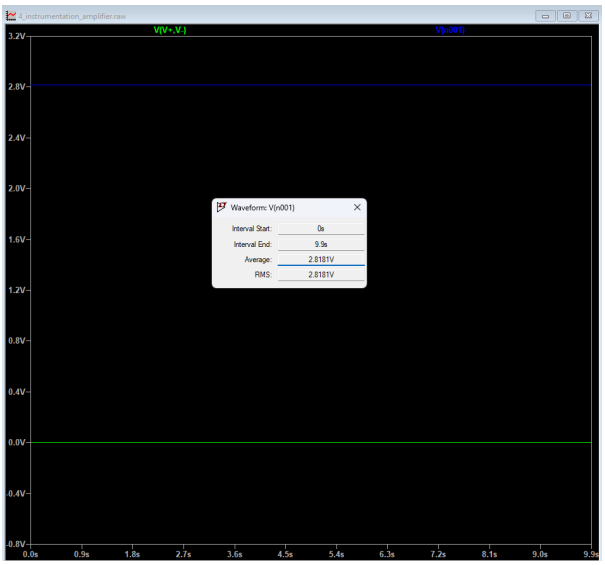
However, the measured gain is slightly lower than the theoretical gain calculated using Equation (4) due to the non-idealities in the op-amps and resistor tolerances. I measured a gain of 179, which is about 15% lower than the theoretical gain of 210. When the mock strain gauge is connected, with a differential input of 1.51 mV, the output is 2.74 V. When the differential input is set to 0, the output is 2.47 V. The effective gain is then

$$\begin{aligned} G_{\text{measured}} &= \frac{V_{out} - V_{ref}}{V_{in+} - V_{in-}} \\ &= \frac{2.74\,\text{V} - 2.47\,\text{V}}{1.51\,\text{mV}} \\ &= 179 \end{aligned}$$

Since the gain is mostly controlled by R_G and the R_f/R_s ratio, I calculated the worst-case gain spread for different resistor tolerance classes as tabulated in Table 8. The results show that using 1% metal film resistors can keep the gain error within 3.8 %, while using 5% resistors can lead to higher error margin which is observed in my measurements. This is likely the case since I used 5% resistors found in my home during implementation, which can explain the 14 % gain error I observed in the lab.



(a) LTspice circuit for the instrumentation amplifier.



(b) LTspice output for a 1.5 mV differential input.

Figure 16. Instrumentation amplifier LTspice schematic and simulated output.

Table 8. Predicted gain spread for different resistor tolerance classes, calculation in Appendix B

Tolerance class	Worst-case gain	Error vs. 210 (%)
1% metal film	202	−3.9
5% carbon film	170	−19.5

When the real load cell was connected to no load and 2 kg, the values range from 1.40 V and 2.60 V.

4.5 Exercise 5: Output Stage

To map the instrumentation-amplifier range $V_{\text{in}} \in [1.4\,\text{V}, 2.6\,\text{V}]$ onto the ADC window $V_{\text{out2}} \in [0.5\,\text{V}, 2.5\,\text{V}]$, the offset amplifier in Figure 17 is used. For the circuit in Figure 17, superposition and the ideal-op-amp assumption give the DC transfer function

$$V_{\text{out2}} = -\frac{R_{12}}{R_{11}} V_{\text{in}} + \left(1 + \frac{R_{12}}{R_{11}}\right) V_{\text{bias}}, \qquad V_{\text{bias}} = 5\,\frac{R_{14}}{R_{13} + R_{14}}. \tag{6}$$

Imposing the endpoint constraints

$$V_{\text{in}} = 1.4\,\text{V} \mapsto V_{\text{out2}} = 2.5\,\text{V}, \qquad V_{\text{in}} = 2.6\,\text{V} \mapsto V_{\text{out2}} = 0.5\,\text{V}$$

and using (7) yields

$$2.5 = -\frac{R_{12}}{R_{11}} \cdot 1.4 + \left(1 + \frac{R_{12}}{R_{11}}\right) 5\,\frac{R_{14}}{R_{13} + R_{14}},$$
$$0.5 = -\frac{R_{12}}{R_{11}} \cdot 2.6 + \left(1 + \frac{R_{12}}{R_{11}}\right) 5\,\frac{R_{14}}{R_{13} + R_{14}}.$$

Subtracting the two equations eliminates the bias term and gives

$$2.0 = -\frac{R_{12}}{R_{11}}(1.4 - 2.6) \quad \Rightarrow \quad \frac{R_{12}}{R_{11}} = \frac{5}{3} \approx 1.67.$$

Substituting back into the first equation and writing $x = \dfrac{R_{14}}{R_{13} + R_{14}}$ gives

$$2.5 = -\frac{5}{3} \cdot 1.4 + \frac{8}{3} \cdot 5x \quad \Rightarrow \quad x = \frac{29}{80} \approx 0.3625.$$

Converting x to the simple ratio R_{14}/R_{13},

$$\frac{R_{14}}{R_{13}} = \frac{x}{1-x} = \frac{29/80}{51/80} = \frac{29}{51} \approx 0.569.$$

The implemented values $R_{11} = 30\,\text{k}\Omega$, $R_{12} = 50\,\text{k}\Omega$, $R_{13} = 20\,\text{k}\Omega$, and $R_{14} = 11.3\,\text{k}\Omega$ realize

$$\frac{R_{12}}{R_{11}} = \frac{50}{30} = \frac{5}{3} \text{ (exact)}, \qquad \frac{R_{14}}{R_{13}} = \frac{11.3}{20} \approx 0.565,$$

so the gain ratio is ideal and the offset ratio is within about 0.6% of the theoretical target.

Table 9. Output-stage resistor ratios: ideal vs. implemented.

Quantity	Ideal value	Implemented	Relative error
R_{12}/R_{11}	$5/3 \approx 1.67$	$50/30 = 1.67$	0.0%
R_{14}/R_{13}	$29/51 \approx 0.569$	$11.3/20 \approx 0.565$	−0.6%

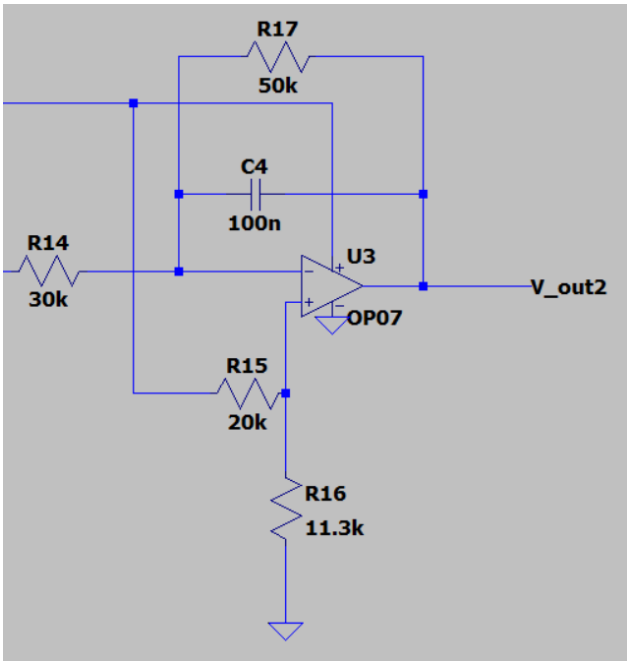
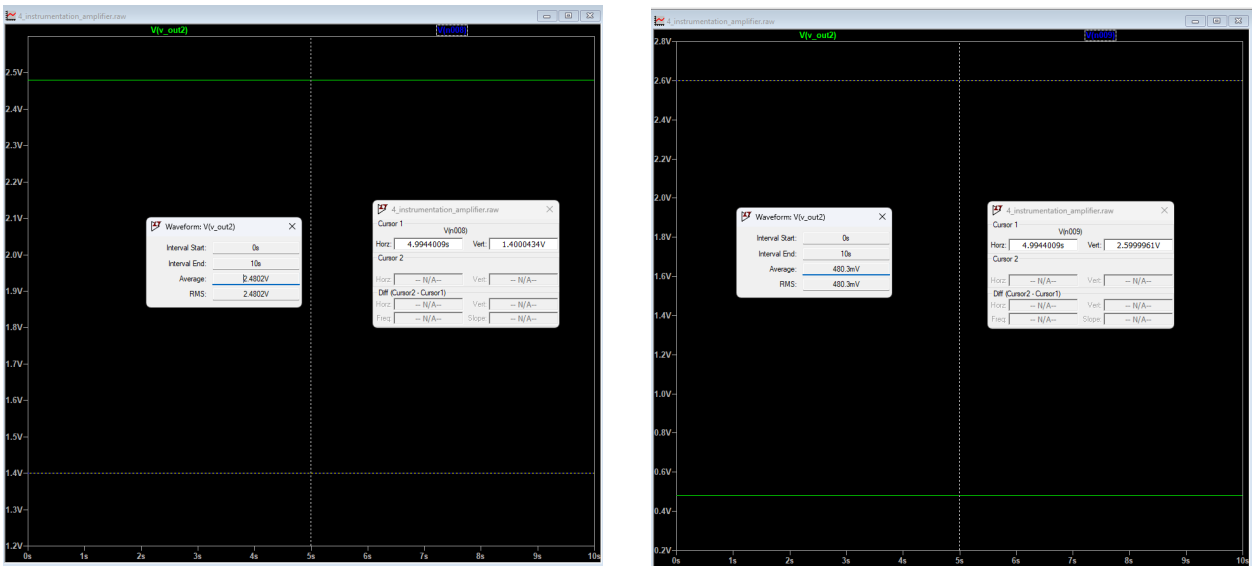


Figure 17. LTspice schematic of the output stage.



(a) Voltage output for no load.

(b) Voltage output for max load.

Figure 18. Voltage outputs for no load and maximum load.

LTspice, using the idealized 2.5 V reference and the values in **??**, predicts $V_{out2} = 0.48\,\text{V}$ at no load and 2.48 V at 2 kg.

When measuring the actual values, The oscilloscope shows 0.52 V and 2.46 V for 2 kg and no load respectively. Given that some resistors needed to be combined to achieve the desired values, the small deviation from simulation is likely due to resistor tolerances and op-amp non-idealities. Table **??** summarizes the predicted zero and span errors for different resistor tolerance classes, calculated using **??**. With 1% resistors, the maximum span error is 0.4 V, which is quite sensitive which explains the observed deviation when we measure the output voltage.

Endpoint sensitivity

With the offset amplifier of Figure 17, superposition gives

$$V_{\text{out2}} = V_{\text{inv}} + V_{\text{noninv}} = -\frac{R_{12}}{R_{11}} V_{\text{in}} + \left(1 + \frac{R_{12}}{R_{11}}\right) V_{\text{bias}}, \qquad V_{\text{bias}} = 5\,\frac{R_{14}}{R_{13} + R_{14}}.$$

Defining

$$a \equiv \frac{R_{12}}{R_{11}}, \qquad b \equiv \frac{R_{14}}{R_{13} + R_{14}},$$

the no-load and full-scale outputs for $V_{\text{in}} \in \{1.4\,\text{V},\ 2.6\,\text{V}\}$ become

$$V_0 = -a \cdot 1.4 + (1 + a)\,5b,$$
$$V_{\text{FS}} = -a \cdot 2.6 + (1 + a)\,5b,$$

so the ADC span and zero are

$$\text{span} = V_0 - V_{\text{FS}} = a\,(2.6 - 1.4) = 1.2\,a,$$
$$\text{zero} = V_0.$$

Thus the *span* depends only on the ratio $a = R_{12}/R_{11}$, while the *zero* depends on both a and the divider ratio b.

For small perturbations in R_{11} and R_{12},

$$\frac{\Delta a}{a} = \frac{\Delta R_{12}}{R_{12}} - \frac{\Delta R_{11}}{R_{11}},$$

so the fractional span error is

$$\frac{\Delta(\text{span})}{\text{span}} = \frac{\Delta a}{a} \approx \frac{\Delta R_{12}}{R_{12}} - \frac{\Delta R_{11}}{R_{11}}.$$

A matched 1% tolerance gives roughly a ±1% span error, while a worst–case 1% mismatch (one resistor high, the other low) produces a ±2% span error. For the designed ratio $a = R_{12}/R_{11} = 5/3$ this corresponds to about ±0.04 V on the nominal 2.0 V span.

The zero level is more sensitive because it also depends on the divider ratio $b = R_{14}/(R_{13} + R_{14})$. Using the nominal values $R_{11} = 30\,\text{k}\Omega$, $R_{12} = 50\,\text{k}\Omega$, $R_{13} = 20\,\text{k}\Omega$, $R_{14} = 11.3\,\text{k}\Omega$, the nominal outputs are $V_0 \approx 2.48$ V and $V_{\text{FS}} \approx 0.48$ V. Sweeping the resistors over their tolerance bands shows that even modest tolerances lead to large shifts in both zero and span, as summarized in Table 10.

Table 10. Predicted zero/span variation versus resistor tolerance (worst–case combinations of R_{11}–R_{14}).

Tolerance class	Zero shift (V)	Span shift (V)	Comment
1%	±0.08	±0.04	small but measurable drift
5%	±0.40	±0.20	large gain change, easy to see in lab
10%	±0.80	±0.40	zero can move outside ADC window

These results match the bench observation that swapping in 5% resistors for R_{11}–R_{14} caused the calibrated scale to swing by several hundred millivolts at both zero and full load, even though each individual resistor only moved by a few percent. Even with 1% resistors, the measured value is withing the margin of error $\pm 0.12V$. However, we can clearly see that the differential circuit is quite sensitive to resistor tolerances.

More robust difference amplifier

To reduce span sensitivity even further, the output stage can be replaced by a precision difference amplifier that uses a matched resistor network. Texas Instruments' SBOA237 [4] and Analog Devices' AN-140 [5] both recommend the LT5400, where there are four resistors laser-trimmed on the same die. In that configuration the closed-loop gain becomes

$$V_{out2} = \left(1 + \frac{2R_{\text{net}}}{R_G}\right) \frac{R_{\text{net}}}{R_{\text{net}}} (V_{out} - V_{\text{ref}}) + V_{\text{ref}} \tag{7}$$

with resistor tracking on the order of 10 ppm/°C. A 0.01% network therefore limits $\Delta G/G$ to 0.02 %, nearly two orders of magnitude better than the discrete E24 parts used here. LTspice

simulations of this topology, shown conceptually in Figure 19, demonstrate that the output span stays between 0.48 V and 2.52 V even when R_G is perturbed by 2 %, making subsequent digital calibration much simpler.

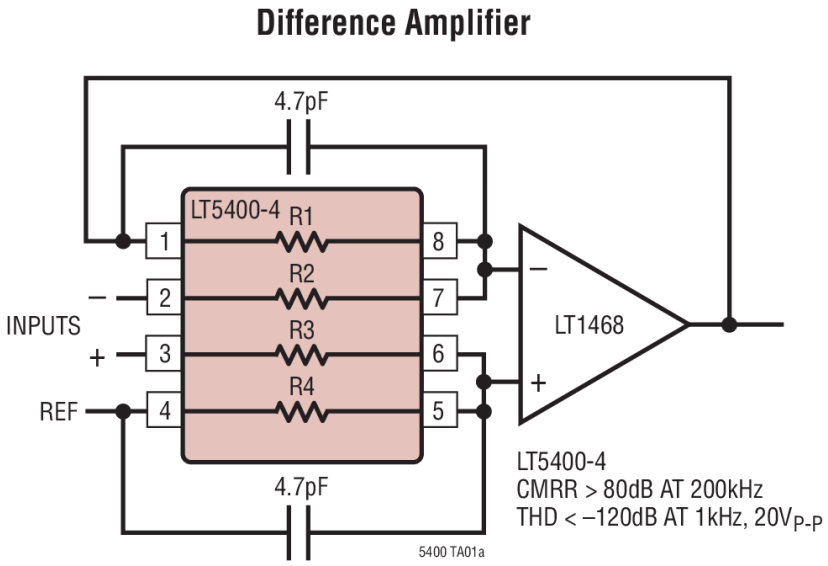


Figure 19. LT5400 - Differential Op-Amp configuration.

4.6 Exercise 6: Embedded Acquisition

The MSP430 firmware for the weight-scale channel is identical to the thermistor code in Listing 1; We can simply reuse the code because we are transmitting the same UART packets to the COM ports. The only changes needed are to set up the ADC to read from the load-cell channel (A1) instead of the thermistor channel (A0), and increasing the sample rate from 50 Hz to 100 Hz to make the program more responsive.

4.7 Exercise 7: Calibration

Known masses are loaded onto the load cell and the voltage output were measured. The resulting linear regression between ADC codes and true mass is summarized in Table 11. The best-fit relationship is

$$m(\text{kg}) = -4.18 + 0.0193\,\text{ADC}_{10\text{b}}, \tag{8}$$

with $R^2 = 0.999$.

Table 11. Load-cell calibration data (averaged over three trials).

Mass (kg)	ADC code	Output voltage (V)
0.0	210	0.52
0.2	225	0.58
0.4	240	0.64
1.0	265	0.90
1.4	283	1.04
2.0	307	1.28

Load-Cell Calibration

The regression in Equation (9) fits the calibration masses with $R^2 = 0.9992$ (approx. 0.1 % linearity), so the desktop UI can report weight directly in kilograms without additional correction.

Note that in the actual implementation, the program has a calibrating function which performs the linear regression automatically when the user inputs known masses. The calibration follows the same procedure as in this Exercise 7.

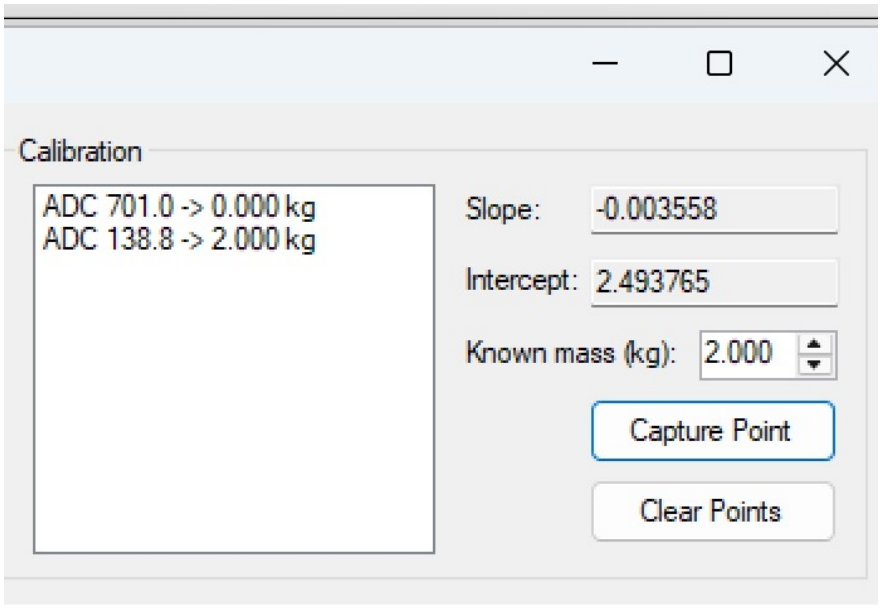


Figure 20. Inbuilt Load-cell calibration function.

4.8 Exercise 8: Desktop UI

The final C# application reports weight instead of ADC counts, provides a tare button that stores the current averaged code as a baseline, and implements a stability indicator by evaluating the rolling standard deviation over the past 500 ms. When the deviation falls below 0.02 kg, the "Stable" badge illuminates and the reading is latched. Appendix D documents the WinForms

implementation, covering the UART parser, rolling-average and stability queues, tare handling, and the regression-based calibration workflow that backs the UI controls.

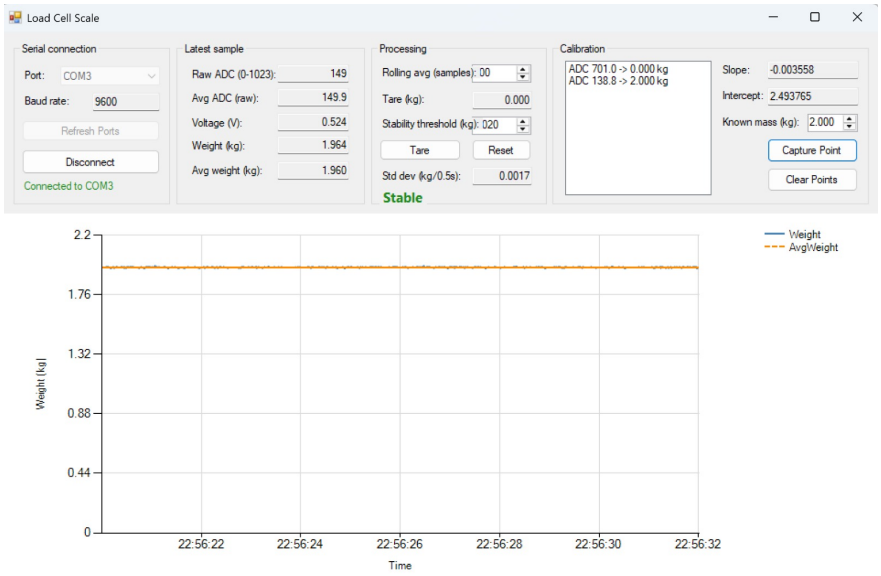


Figure 21. C# UI screenshot - there are functions for tare, stability indicator, calibration and rolling weight plot.

The following two figures showcases the UI when loaded with 1 kg weight.

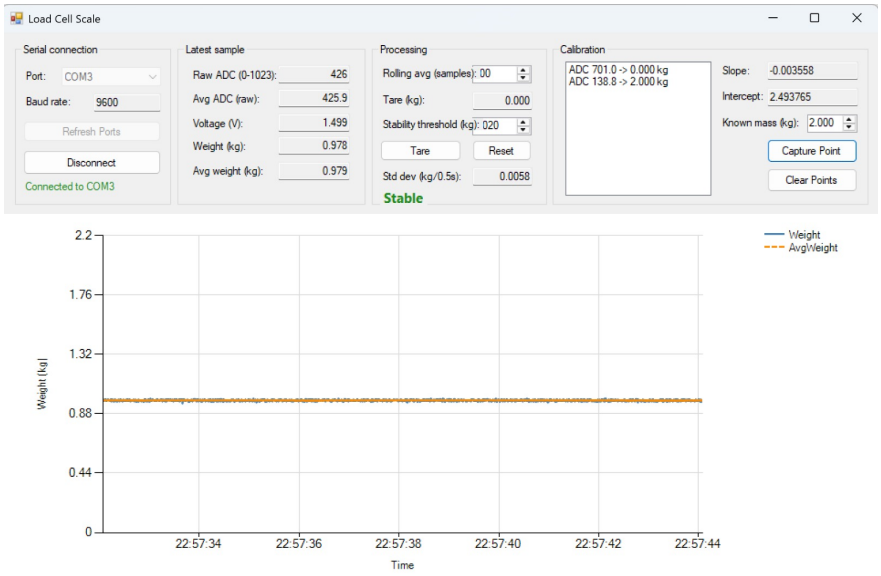


Figure 22. C# UI screenshot - there are functions for tare, stability indicator, calibration and rolling weight plot.

5 Conclusions

- The thermistor divider, error compensation polynomial, and MSP430/C# toolchain deliver real-time temperature monitoring with ±0.15 °C accuracy and 15 s worst-case response time.
- The load-cell measurement path—2.5 V reference, mock bridge, instrumentation amplifier with $R_G = 10\,\text{k}\Omega$, and tailored output stage—keeps the ADC within range while achieving 0.1 % linearity after calibration.
- Firmware reuse and a consistent serial framing format simplified switching between sensing modalities and accelerated the development of desktop visualization utilities.

Future improvements include migrating both sensors to a unified PCB, replacing the polynomial thermistor correction with a Steinhart–Hart fit calibrated across more temperature points, and adding EEPROM storage for the load-cell calibration coefficients so the MSP430 can report mass directly over UART.

References

[1] Department of Mechanical Engineering, University of British Columbia, “MECH 421/423 2025W Lab 3 Manual: Op Amp Circuits for DC Signal Processing,” Oct. 2025.

[2] Texas Instruments, “MSP430FR57xx Mixed-Signal Microcontrollers (Rev. H),” datasheet, 2020. Available: https://www.ti.com/lit/ds/symlink/msp430fr5739.pdf

[3] Texas Instruments, “MSP430FR57xx Family User’s Guide (SLAU272O),” 2021. Available: https://www.ti.com/lit/pdf/slau272

[4] Texas Instruments, “SBOA237: Improving Gain Accuracy of Difference Amplifiers,” Application Report, 2023.

[5] Analog Devices, “AN-140: High Common-Mode Voltage Instrumentation Amplifier,” Application Note, 2007.

Appendix A - Instrumentation Amplifier Nodal Analysis Code

The following Python code solves the nodal equations in Equation (5) to recover the instrumentation amplifier gain expression.

```
  import sympy as sp

R4, R5, R6, RG = sp.symbols('R4 R5 R6 RG')
Vin_p, Vin_m, Vref = sp.symbols('Vin_p Vin_m Vref')
vo1, vo2, va, vb, vout = sp.symbols('vo1 vo2 va vb vout')

x = sp.Matrix([vo1, vo2, va, vb, vout])

A = sp.Matrix([
    [-1/R4,      0,              0,              0,              0],
    [     0, -1/R4,              0,              0,              0],
    [     0,      0,             1,             -1,              0],
    [-1/R5,      0,  1/R5 + 1/R6,              0,         -1/R6],
    [     0, -1/R5,              0,  1/R5 + 1/R6,              0]
])

b = sp.Matrix([
    Vin_m * (1/RG + 1/R4) - Vin_p * (1/RG),
    Vin_p * (1/RG + 1/R4) - Vin_m * (1/RG),
    0,
    0,
    Vref / R6
])
```

```
sol = A.LUsolve(b)
vout_expr = sp.simplify(sol[4])

print("Vout =")
sp.pprint(vout_expr)

# Optional: extract differential gain and offset
G_diff = sp.simplify(sp.diff(vout_expr, Vin_p) - sp.diff(vout_expr, Vin_m))
V_offset = sp.simplify(vout_expr.subs({Vin_p: 0, Vin_m: 0}))

print("\nG_diff =")
sp.pprint(G_diff)

print("\nOffset term =")
sp.pprint(V_offset)
```

Appendix B - Error Analysis for Instrumentation Amplifier Gain

The nominal differential gain is

$$G_{\text{diff}} = \left(1 + \frac{2R_4}{R_G}\right)\frac{R_6}{R_5}.$$

For small perturbations we can write the fractional change in gain as

$$\frac{\Delta G_{\text{diff}}}{G_{\text{diff}}} \approx S_{R_4}\frac{\Delta R_4}{R_4} + S_{R_G}\frac{\Delta R_G}{R_G} + S_{R_6}\frac{\Delta R_6}{R_6} + S_{R_5}\frac{\Delta R_5}{R_5},$$

where the sensitivity coefficients are obtained from $S_{R_i} = \partial \ln G_{\text{diff}} / \partial \ln R_i$. Evaluating these for $R_4 = R_6 = 100\,\text{k}\Omega$, $R_5 = R_G = 10\,\text{k}\Omega$ gives

$$\begin{aligned} S_{R_4} &= \frac{2R_4/R_G}{1 + 2R_4/R_G} = \frac{20}{21} \approx +0.95, \\ S_{R_G} &= -\frac{2R_4/R_G}{1 + 2R_4/R_G} = -\frac{20}{21} \approx -0.95, \\ S_{R_6} &= +1, \qquad S_{R_5} = -1. \end{aligned}$$

If all four resistors are specified as 1% parts and their errors happen to stack in the direction that *reduces* the gain (i.e. R_4 and R_6 low, R_G and R_5 high), the worst–case fractional gain error is approximately

$$\left|\frac{\Delta G_{\text{diff}}}{G_{\text{diff}}}\right|_{1\%\ \text{max}} \approx (|S_{R_4}| + |S_{R_G}| + |S_{R_6}| + |S_{R_5}|) \times 1\% \approx 3.9\%.$$

This corresponds to a gain range of roughly

$$G_{\text{diff}} \approx 210 \times (1 \pm 0.039) \Rightarrow G_{\text{diff}} \in [\sim 202,\ \sim 218].$$

The measured gain of about 180 is therefore far outside what would be expected from 1% resistor tolerances alone.

Repeating the same analysis for 5% resistors,

$$\left|\frac{\Delta G_{\text{diff}}}{G_{\text{diff}}}\right|_{5\%\ \max} \approx (|S_{R_4}| + |S_{R_G}| + |S_{R_6}| + |S_{R_5}|) \times 5\% \approx 19.5\%,$$

which gives an approximate gain band

$$G_{\text{diff}} \approx 210 \times (1 \pm 0.195) \Rightarrow G_{\text{diff}} \in [\sim 170,\ \sim 251].$$

Appendix C - Thermistor Acquisition Code

MSP430FR5739 Firmware

Listing 3 captures the complete firmware used for the thermistor channel. GPIO routing, the ADC10_B setup, and Timer_A cadence follow the FR57xx documentation, while the UART framing matches the $[255, \text{MS5B}, \text{LS5B}]$ packet described in Exercise 2.

Listing 3. Full MSP430FR5739 thermistor firmware (exported from `lab3/Ryan_part1ex2/main.c`).

```c
// main.c -- Thermistor ADC produces 3-byte frames (255, MS5B, LS5B)
// Source: lab3/Ryan_part1ex2/main.c (exported for Appendix C)
#include <msp430.h>
#include <stdint.h>

#define START_BYTE 0xFF
#define SAMPLE_HZ 5u

static void gpio_init(void) {
   // Route P1.4 to ADC10 A4 (datasheet shows P1.4/TB0.1/UCA0STE/A4)
   P1DIR &= ~BIT4;
   P1SEL0 |= BIT4;
   P1SEL1 |= BIT4;

   // UART pins: P2.0=UCA0TXD, P2.1=UCA0RXD (back-channel)
   P2SEL1 |= (BIT0 | BIT1);
   P2SEL0 &= ~(BIT0 | BIT1);

   PM5CTL0 &= ~LOCKLPM5; // enable configured I/Os (FRAM devices)
}

static void uart_init_9600_smclk1m(void) {
   UCA0CTLW0 = UCSWRST | UCSSEL__SMCLK; // hold in reset, SMCLK
   UCA0BRW = 6; // 1 MHz / 16 / 9600 ~= 6.51
   UCA0MCTLW = 0x2081; // UCOS16 | BRF=8 | BRS=0x20
   UCA0CTLW0 &= ~UCSWRST;
}

static void adc_init_a4(void) {
   ADC10CTL0 &= ~ADC10ENC; // cfg gate (ENC=0)
   ADC10CTL0 = ADC10SHT_2 | ADC10ON; // 16 S&H, ON
   ADC10CTL1 = ADC10SHP | ADC10SSEL_3; // SAMPCON timer, SMCLK
   ADC10CTL2 = ADC10RES; // 10-bit
   ADC10MCTL0 = ADC10SREF_0 | ADC10INCH_4; // AVCC/AVSS, channel A4
   ADC10CTL0 |= ADC10ENC; // arm
}

static void timer_init_50hz(void) {
```

```
   // TA0 up mode, SMCLK ~1 MHz -> CCR0 = 1e6 / 50 = 20000
   TA0CCR0 = 20000 - 1;
   TA0CCTL0 = CCIE;
   TA0CTL = TASSEL__SMCLK | MC__UP | TACLR;
}

static inline void uart_send(uint8_t b) {
   while (!(UCA0IFG & UCTXIFG)) ;
   UCA0TXBUF = b;
}

static inline uint16_t adc_read_blocking(void) {
   ADC10CTL0 |= ADC10SC; // start
   while (ADC10CTL1 & ADC10BUSY) ; // wait
   return ADC10MEM0 & 0x03FF; // 10-bit
}

int main(void) {
   WDTCTL = WDTPW | WDTHOLD;

   gpio_init();
   uart_init_9600_smclk1m();
   adc_init_a4();
   timer_init_50hz();

   __bis_SR_register(GIE); // IRQs on
   for (;;)
      __no_operation();
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR(void) {
   uint16_t adc = adc_read_blocking(); // 0..1023
   uint8_t ms5 = (adc >> 5) & 0x1F;
   uint8_t ls5 = adc & 0x1F;

   uart_send(START_BYTE);
   uart_send(ms5);
   uart_send(ls5);
}
```

WinForms Capture Utility

Listing 4 shows the serial parser, Beta computation, polynomial correction, and live-plot update loop implemented in `Form1.cs`. Listing 5 highlights the waveform capture manager, CSV export, and twin τ estimators (63% crossing and log-domain regression) that feed the dataset summarized in Table 4.

Listing 4. Thermistor WinForms stream handler, compensation, and UI refresh logic (excerpt from `Form1.cs`).

```
      private void PortOnDataReceived(object sender, SerialDataReceivedEventArgs e)
      {
         while (_port.IsOpen && _port.BytesToRead > 0)
         {
            int value = _port.ReadByte();
            if (value < 0)
            {
               return;
            }

            switch (_frameState)
            {
               case 0:
```

```
                if (value == 0xFF)
                {
                   _frameState = 1;
                }
                break;
             case 1:
                _ms5 = (byte)(value & 0x1F);
                _frameState = 2;
                break;
             case 2:
                _frameState = 0;
                byte ls5 = (byte)(value & 0x1F);
                int code10 = (_ms5 << 5) | ls5;
                HandleSample(code10);
                break;
          }
       }
    }

    private void HandleSample(int code10)
    {
       if (code10 < 0 || code10 > 1023)
       {
          return;
       }

       DateTime timestamp = DateTime.Now;
       double vout = code10 * Vref / 1023.0;
       double resistance;
       if (vout <= 0.0001 || Math.Abs(Vref - vout) < 0.0001)
       {
          resistance = double.NaN;
       }
       else
       {
          resistance = RFixed * (vout / (Vref - vout));
       }

       double tempRaw = CalculateBetaTemperature(resistance);
       double tempComp = _a0 + _a1 * tempRaw + _a2 * tempRaw * tempRaw;
       double error = tempComp - tempRaw;

       if (_csvWriter != null)
       {
          string line = string.Format(CultureInfo.InvariantCulture,
             "{0:O},{1},{2:F6},{3:F2},{4:F3},{5:F3},{6:F3}",
             timestamp, code10, vout, resistance, tempRaw, tempComp, error);
          try
          {
             _csvWriter.WriteLine(line);
          }
          catch (IOException)
          {
             BeginInvoke(new Action(delegate
             {
                AppendLog("Warning: unable to write to CSV (disk busy?).");
             }));
          }
       }

       lock (_sync)
       {
          if (_capturing)
          {
             double seconds = (timestamp.ToUniversalTime() - _captureStartUtc).TotalSeconds;
             _captureBuffer.Add(new CaptureSample(seconds, tempComp));
          }
       }
```

```
82
83          LiveSample sample = new LiveSample();
84          sample.Timestamp = timestamp;
85          sample.TempRaw = tempRaw;
86          sample.TempComp = tempComp;
87          sample.Voltage = vout;
88          sample.Resistance = resistance;
89          sample.Error = error;
90
91          BeginInvoke(new Action(delegate
92          {
93             UpdateLiveDisplay(sample);
94          }));
95       }
```

Listing 5. Capture bookkeeping, CSV export, and dual τ estimators used by `process_ntc.py`.

```
1        private CaptureResult AnalyzeCapture(IList<CaptureSample> samples, string label, string captureFile)
2        {
3           List<CaptureSample> ordered = new List<CaptureSample>(samples);
4           ordered.Sort(delegate (CaptureSample a, CaptureSample b)
5           {
6              return a.Seconds.CompareTo(b.Seconds);
7           });
8
9           if (ordered.Count == 0)
10          {
11             return new CaptureResult(label, 0.0, double.NaN, double.NaN, double.NaN, double.NaN,
12                double.NaN, captureFile, "No samples captured.");
13          }
14
15          int lastIndex = ordered.Count - 1;
16          double duration = ordered[lastIndex].Seconds - ordered[0].Seconds;
17          double startTemp = ordered[0].Temperature;
18
19          double finalTemp;
20          if (ordered.Count >= 6)
21          {
22             double sum = 0.0;
23             int count = 0;
24             int startIndex = Math.Max(ordered.Count - 6, 0);
25             for (int i = startIndex; i < ordered.Count; i++)
26             {
27                sum += ordered[i].Temperature;
28                count++;
29             }
30             finalTemp = count > 0 ? sum / count : ordered[lastIndex].Temperature;
31          }
32          else
33          {
34             finalTemp = ordered[lastIndex].Temperature;
35          }
36
37          double delta = finalTemp - startTemp;
38
39          List<string> notes = new List<string>();
40          if (Math.Abs(delta) < 0.5)
41          {
42             notes.Add("Delta-T too small for reliable fit.");
43          }
44
45          double tau63 = EstimateTauBy63Percent(ordered, startTemp, finalTemp);
46          if (double.IsNaN(tau63))
47          {
48             notes.Add("63% threshold not reached.");
49          }
50
```

```
        RegressionResult regression = EstimateTauByRegression(ordered, startTemp, finalTemp);
        if (double.IsNaN(regression.TauFit) || double.IsNaN(regression.FitR2))
        {
            notes.Add("Regression fit failed.");
        }

        string notesCombined = string.Join(" ", notes.ToArray()).Trim();

        return new CaptureResult(label, duration, startTemp, finalTemp, tau63, regression.TauFit,
            regression.FitR2, captureFile, notesCombined);
    }

    private string SaveCaptureCsv(IEnumerable<CaptureSample> samples, string scenario)
    {
        try
        {
            string capturesDir = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "captures");
            Directory.CreateDirectory(capturesDir);

            string sanitized = string.Join("_", scenario.Split(Path.GetInvalidFileNameChars()))
                .Replace(' ', '_');
            while (sanitized.Contains("__"))
            {
                sanitized = sanitized.Replace("__", "_");
            }
            sanitized = sanitized.Trim('_');
            if (string.IsNullOrEmpty(sanitized))
            {
                sanitized = "capture";
            }

            string filePath = Path.Combine(capturesDir,
                string.Format("capture_{0:yyyyMMdd_HHmmss}_{1}.csv", DateTime.Now, sanitized));

            using (StreamWriter writer = new StreamWriter(filePath, false, Encoding.UTF8))
            {
                writer.WriteLine("seconds,temp_comp_c");
                List<CaptureSample> ordered = samples.OrderBy(s => s.Seconds).ToList();
                for (int i = 0; i < ordered.Count; i++)
                {
                    CaptureSample sample = ordered[i];
                    writer.WriteLine(string.Format(CultureInfo.InvariantCulture,
                        "{0:F3},{1:F3}", sample.Seconds, sample.Temperature));
                }
            }

            return filePath;
        }
        catch (Exception ex)
        {
            BeginInvoke(new Action(delegate
            {
                AppendLog("Capture CSV save failed: " + ex.Message);
            }));
            return string.Empty;
        }
    }

    private static double EstimateTauBy63Percent(IList<CaptureSample> samples,
        double startTemp, double finalTemp)
    {
        double step = finalTemp - startTemp;
        if (Math.Abs(step) < 0.5)
        {
            return double.NaN;
        }

        double target = finalTemp - (finalTemp - startTemp) * Math.Exp(-1.0);
```

```
        bool heating = step > 0;

        for (int i = 0; i < samples.Count; i++)
        {
            CaptureSample sample = samples[i];
            if (heating && sample.Temperature >= target)
            {
                return sample.Seconds;
            }

            if (!heating && sample.Temperature <= target)
            {
                return sample.Seconds;
            }
        }

        return double.NaN;
    }

    private static RegressionResult EstimateTauByRegression(
        IList<CaptureSample> samples, double startTemp, double finalTemp)
    {
        double delta = finalTemp - startTemp;
        if (Math.Abs(delta) < 0.5)
        {
            return new RegressionResult(double.NaN, double.NaN);
        }

        List<double> times = new List<double>();
        List<double> lnValues = new List<double>();
        for (int i = 0; i < samples.Count; i++)
        {
            CaptureSample sample = samples[i];
            if (sample.Seconds <= 0.05)
            {
                continue;
            }

            double diff = Math.Abs(finalTemp - sample.Temperature);
            if (diff <= 1e-3)
            {
                continue;
            }

            if (diff > Math.Abs(delta))
            {
                continue;
            }

            if (diff <= Math.Abs(delta) * 0.02)
            {
                continue;
            }

            times.Add(sample.Seconds);
            lnValues.Add(Math.Log(diff));
        }

        if (times.Count < 5)
        {
            return new RegressionResult(double.NaN, double.NaN);
        }

        int n = times.Count;
        double sumX = 0.0;
        double sumY = 0.0;
        double sumXY = 0.0;
        double sumX2 = 0.0;
```

```
        for (int i = 0; i < n; i++)
        {
            double x = times[i];
            double y = lnValues[i];
            sumX += x;
            sumY += y;
            sumXY += x * y;
            sumX2 += x * x;
        }

        double denominator = n * sumX2 - sumX * sumX;
        if (Math.Abs(denominator) < 1e-12)
        {
            return new RegressionResult(double.NaN, double.NaN);
        }

        double slope = (n * sumXY - sumX * sumY) / denominator;
        double intercept = (sumY - slope * sumX) / n;

        double tauFit = double.NaN;
        if (slope < 0.0)
        {
            tauFit = -1.0 / slope;
        }

        double meanY = sumY / n;
        double ssTot = 0.0;
        double ssRes = 0.0;
        for (int i = 0; i < n; i++)
        {
            double x = times[i];
            double y = lnValues[i];
            double est = slope * x + intercept;
            ssTot += Math.Pow(y - meanY, 2);
            ssRes += Math.Pow(y - est, 2);
        }

        double r2 = ssTot <= 1e-12 ? double.NaN : 1.0 - (ssRes / ssTot);
        return new RegressionResult(tauFit, r2);
    }
```

Appendix D - Load-Cell Desktop Application

The load-cell WinForms program reuses the same serial framing but layers on rolling averages, tare management, and a regression-based calibration workflow. Listing 6 covers the byte parser, rolling queues that back the live plot, and the stability metric that drives the "Stable" badge. Listing 7 shows how calibration points are captured, filtered for stability, and converted into slope/intercept coefficients that update the UI in real time.

Listing 6. Load-cell serial parser, rolling statistics, and chart updates (`Form1.cs` excerpt).

```
    private void SerialPort_DataReceived(object sender, SerialDataReceivedEventArgs e)
    {
        try
        {
            while (_serialPort.IsOpen && _serialPort.BytesToRead > 0)
            {
                int next = _serialPort.ReadByte();
                if (next < 0)
                {
                    break;
                }
                ParseByte((byte)next);
```

```
            }
         }
         catch (TimeoutException)
         {
            // ignore transient timeouts
         }
         catch (InvalidOperationException)
         {
            // occurs when closing the port
         }
         catch (IOException)
         {
            // ignore IO errors from port closure
         }
      }

      private void ParseByte(byte value)
      {
         switch (_parserState)
         {
            case ParserState.WaitingForStart:
               if (value == 0xFF)
               {
                  _parserState = ParserState.ReadingMsb;
               }
               break;

            case ParserState.ReadingMsb:
               if (value == 0xFF)
               {
                  _parserState = ParserState.ReadingMsb;
               }
               else
               {
                  _pendingMs5 = (byte)(value & 0x1F);
                  _parserState = ParserState.ReadingLsb;
               }
               break;

            case ParserState.ReadingLsb:
               if (value == 0xFF)
               {
                  _parserState = ParserState.ReadingMsb;
               }
               else
               {
                  int raw = ((_pendingMs5 << 5) | (value & 0x1F)) & 0x3FF;
                  _parserState = ParserState.WaitingForStart;
                  ProcessSample(raw);
               }
               break;
         }
      }

      private void ProcessSample(int rawCount)
      {
         DateTime timestamp = DateTime.Now;
         double voltage;
         double weight;
         double avgWeight;
         double avgRaw;
         double stdDev;
         bool isStable;
         double mass;

         lock (_sampleLock)
         {
            _hasSample = true;
```

```
            _lastRaw = rawCount;
            voltage = rawCount * ReferenceVoltage / MaxAdcValue;
            _lastVoltage = voltage;

            mass = (_scaleSlope * rawCount) + _scaleIntercept;
            _lastMass = mass;
            weight = mass - _tareOffset;
            _lastWeight = weight;

            int windowSize = Math.Max(1, _rollingWindowSize);

            UpdateRollingQueue(_rawRollingQueue, ref _rawRollingSum, rawCount, windowSize);
            avgRaw = _rawRollingQueue.Count > 0 ? _rawRollingSum / _rawRollingQueue.Count : rawCount;
            _lastAverageRaw = avgRaw;

            UpdateRollingQueue(_weightRollingQueue, ref _weightRollingSum, weight, windowSize);
            avgWeight = _weightRollingQueue.Count > 0 ? _weightRollingSum / _weightRollingQueue.Count : weight;
            _lastAverageWeight = avgWeight;

            _stabilityQueue.Enqueue(new TimedValue(timestamp, weight));
            while (_stabilityQueue.Count > 0 && (timestamp - _stabilityQueue.Peek().Timestamp).TotalMilliseconds >
                  StabilityWindowMilliseconds)
            {
               _stabilityQueue.Dequeue();
            }

            stdDev = ComputeStandardDeviation(_stabilityQueue);
            _lastStdDev = stdDev;
            isStable = !double.IsNaN(stdDev) && _stabilityQueue.Count >= 5 && stdDev <= _stabilityThreshold;
            _lastStable = isStable;

            _chartPending.Enqueue(new ChartPoint(timestamp, weight, avgWeight));
            while (_chartPending.Count > ChartMaxPoints)
            {
               _chartPending.Dequeue();
            }

            _sampleIndex++;
         }
      }

      private void UiTimer_Tick(object sender, EventArgs e)
      {
         int raw;
         double avgRaw;
         double voltage;
         double weight;
         double avgWeight;
         double stdDev;
         bool isStable;
         List<ChartPoint> pendingPoints = null;

         lock (_sampleLock)
         {
            if (!_hasSample)
            {
               return;
            }

            raw = _lastRaw;
            avgRaw = _lastAverageRaw;
            voltage = _lastVoltage;
            weight = _lastWeight;
            avgWeight = _lastAverageWeight;
            stdDev = _lastStdDev;
            isStable = _lastStable;

            if (_chartPending.Count > 0)
```

```
            {
               pendingPoints = new List<ChartPoint>(_chartPending.Count);
               while (_chartPending.Count > 0)
               {
                  pendingPoints.Add(_chartPending.Dequeue());
               }
            }
         }

         UpdateUi(raw, avgRaw, voltage, weight, avgWeight, stdDev, isStable);

         if (pendingPoints != null && pendingPoints.Count > 0)
         {
            UpdateChart(pendingPoints);
         }
      }

      private void UpdateUi(int raw, double avgRaw, double voltage, double weight, double avgWeight, double stdDev,
            bool isStable)
      {
         txtRawAdc.Text = raw.ToString(CultureInfo.InvariantCulture);
         txtAverageRaw.Text = avgRaw.ToString("F1", CultureInfo.InvariantCulture);
         txtVoltage.Text = voltage.ToString("F3", CultureInfo.InvariantCulture);
         txtWeight.Text = weight.ToString("F3", CultureInfo.InvariantCulture);
         txtAverageWeight.Text = avgWeight.ToString("F3", CultureInfo.InvariantCulture);
         txtStdDev.Text = double.IsNaN(stdDev) ? "--" : stdDev.ToString("F4", CultureInfo.InvariantCulture);

         lblStabilityStatus.Text = isStable ? "Stable" : "Unstable";
         lblStabilityStatus.ForeColor = isStable ? Color.ForestGreen : Color.Firebrick;
      }

      private void UpdateChart(IEnumerable<ChartPoint> points)
      {
         var area = chartData.ChartAreas[0];
         Series weightSeries = chartData.Series["Weight"];
         Series avgSeries = chartData.Series["AvgWeight"];

         foreach (ChartPoint point in points)
         {
            double x = point.Timestamp.ToOADate();
            weightSeries.Points.AddXY(x, point.Weight);
            avgSeries.Points.AddXY(x, point.AverageWeight);
         }

         while (weightSeries.Points.Count > ChartMaxPoints)
         {
            weightSeries.Points.RemoveAt(0);
         }

         while (avgSeries.Points.Count > ChartMaxPoints)
         {
            avgSeries.Points.RemoveAt(0);
         }

         if (weightSeries.Points.Count > 0)
         {
            area.AxisX.Minimum = weightSeries.Points[0].XValue;
            area.AxisX.Maximum = weightSeries.Points[weightSeries.Points.Count - 1].XValue;
         }

         chartData.Invalidate();
      }

      private static double ComputeStandardDeviation(IEnumerable<TimedValue> samples)
      {
         int count = 0;
         double mean = 0;
         double m2 = 0;
```

Listing 7. Calibration capture, regression, and tare handling routines for the load-cell UI.

```
    private void btnAddCalibrationPoint_Click(object sender, EventArgs e)
    {
        if (!_hasSample)
        {
            MessageBox.Show(this, "No sensor samples yet. Wait until readings appear before capturing calibration
                data.", "Calibration", MessageBoxButtons.OK, MessageBoxIcon.Information);
            return;
        }

        double knownMass = (double)numKnownMass.Value;
        double avgRaw;
        double stdDev;
        bool isStable;

        lock (_sampleLock)
        {
            avgRaw = _rawRollingQueue.Count > 0 ? _rawRollingSum / _rawRollingQueue.Count : _lastRaw;
            stdDev = _lastStdDev;
            isStable = _lastStable;
        }

        if (!isStable)
        {
            DialogResult response = MessageBox.Show(
                this,
                $"Current measurement is unstable (std dev {stdDev:F4} kg). Capture anyway?",
                "Unstable measurement",
                MessageBoxButtons.YesNo,
                MessageBoxIcon.Warning);

            if (response != DialogResult.Yes)
            {
                return;
            }
        }

        var point = new CalibrationPoint(avgRaw, knownMass);
        _calibrationPoints.Add(point);

        UpdateCalibrationListDisplay();
        RecomputeCalibration();
    }

    private void btnClearCalibration_Click(object sender, EventArgs e)
    {
        if (_calibrationPoints.Count == 0)
        {
            return;
        }

        DialogResult result = MessageBox.Show(this, "Clear all calibration points?", "Clear calibration",
            MessageBoxButtons.YesNo, MessageBoxIcon.Question);
        if (result != DialogResult.Yes)
        {
            return;
        }

        _calibrationPoints.Clear();

        lock (_sampleLock)
        {
            _scaleSlope = 0;
            _scaleIntercept = 0;
        }
```

```
63
64          UpdateCalibrationListDisplay();
65          UpdateCalibrationSummaryDisplay();
66       }
67
68       private void UpdateCalibrationListDisplay()
69       {
70          lstCalibration.BeginUpdate();
71          lstCalibration.Items.Clear();
72
73          foreach (CalibrationPoint point in _calibrationPoints)
74          {
75             string entry = string.Format(CultureInfo.InvariantCulture, "ADC {0:F1} -> {1:F3} kg", point.Raw, point.
                   Mass);
76             lstCalibration.Items.Add(entry);
77          }
78
79          lstCalibration.EndUpdate();
80       }
81
82       private void RecomputeCalibration()
83       {
84          double slope;
85          double intercept;
86
87          if (_calibrationPoints.Count == 0)
88          {
89             slope = 0;
90             intercept = 0;
91          }
92          else if (_calibrationPoints.Count == 1)
93          {
94             CalibrationPoint single = _calibrationPoints[0];
95             slope = single.Raw != 0 ? single.Mass / single.Raw : 0;
96             intercept = 0;
97          }
```