Sensor Design Project

Project Report

Omar Tanner

omsst2

18 December 2023

Word Count: 2955

EPSRC Centre for Doctoral Training in Sensor Technologies
for a Healthy and Sustainable Future

University of Cambridge

Contents

Chapter 1

Introduction

In the University of Cambridge's Sensor CDT, we develop temperature and pulse health monitoring sensors. Aimed at creating affordable monitors for resource-limited settings, our project aligns with the UN's Global Sustainability Goals [1]. We first enhance a temperature sensor for more accurate and precise measurements, then create a reliable heart-rate sensor to aid in health diagnostics and care.

Chapter 2

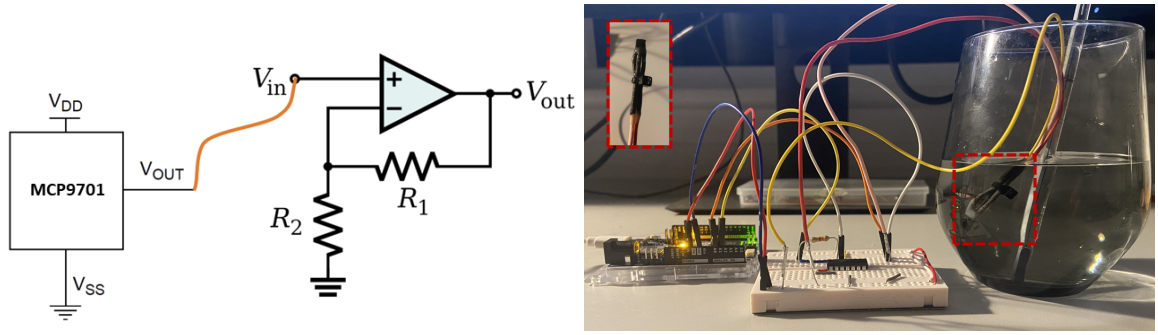
Temperature sensor

2.1 Introduction to temperature sensors

The project improves a thermistor-based sensor for accurate body temperature monitoring. Affordable and responsive thermistors, ideal for portable devices, vary resistance with temperature [2]. Enhancing accuracy and precision is crucial for reliable readings, supporting affordable health technology in resource-scarce areas.

2.2 Setup

In our project we measure temperature using an MCP9701 thermistor [3] connected to an Arduino UNO R4 WiFi microcontroller. To calibrate the thermistor and improve its resolution , we amplify its output voltage so it covers a wider range of the Arduino's Analogue to Digital Converter (ADC) steps (explained in Appendix A.1.4). Figure 2.1 shows our setup. We measure both v_{in} and v_{out} with our Arduino's $A0$ and $A1$ ports to determine the temperature both directly from the thermistor and after amplification.



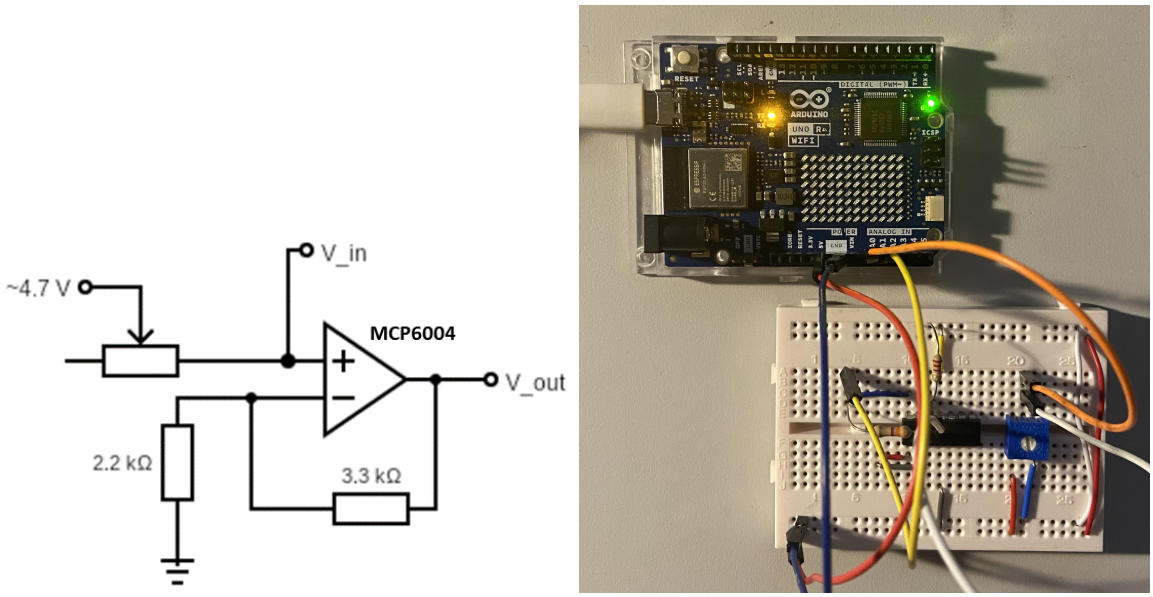
(a) (b)

Figure 2.1: Thermistor calibration setup. (a): circuit, (b): physical.

We calibrate our amplifier before the full entire circuit to determine its actual gain. From this, one can infer the higher resolution input voltage from the thermistor's amplified voltage. We choose resistances of $R_1 = 3.3k\Omega$ and $R_2 = 2.2k\Omega$ for our amplifier, making the theoretical gain $G = 1 + \frac{3.3}{2.2} = 2.5$. In practice the gain is different due to e.g. errors in the resistors' resistances and the op-amp (an MCP6004 [4]). Since the Arduino supplies $4.7V$ ($.3V$ are lost due to an internal diode), we expect the amplifier to saturate with input $> \frac{4.7}{2.5} = 1.88V$. Thus, our amplifier saturates at temperatures $> \frac{1.88-0.4}{0.0195} = 75.9°C$. This is much larger than a human's body temperature, so amplifier saturation is not a problem with our gain.

2.2.1 Amplifier calibration

Our setup for calibrating our (non-inverting operational) amplifier is shown in Figure 2.2. We vary the input voltage using a potentiometer, and measure v_{in} and v_{out} through the $A0$ and $A1$ input channels of our Arduino, respectively.



(a) (b)

Figure 2.2: Non-inverting op-amp calibration setup. (a): circuit, (b): physical.

To process the data on the Arduino (Appendix A.2.1), we run its `loop` at as high a frequency as possible, and use the `analogRead` command to read from the channels. We use a `std::deque` to store the last 100 values, compute the mean and deviation of the last 100 measurements in the queue, and output them to the Serial output.

To gather data for calibration, we vary the potentiometer's resistance so the input voltage ranges from $0.7V$ to $\sim 2V$: where it saturates. We choose $0.7V$ as a lower bound

as this corresponds to a temperature of $\frac{0.7-0.4}{0.0195} = 15.3°C$ which is far below human body temperature, and experimentally we find that the gain of the amplifier increases below this range (skewing the calibration). For each input voltage, we record the output from the Arduino. We record this for the input voltage v_{in} connected to the amplifier, the output voltage v_{out} from the amplifier, *and* the input voltage v_{in} when the amplifier is disconnected (which we denote by v^*_{in}). We measure both v_{in} and v^*_{in} because bias currents in the our op-amp [4] can offset v_{in}. We factor in the bias currents so they do not skew our temperature measurements.

We analyse our gathered data with Python using the `numpy`, `scipy` and `matplotlib` libraries (Appendix A.3.1). We calculate the mean and variance of the measured voltages v from the mean and variance of the measured ADC values n with basic statistics:

$$\bar{v} = \overline{\frac{4.7}{1024}n} = \frac{4.7}{1024}\bar{n} \tag{2.1}$$

$$\text{Var}(v) = \text{Var}(\frac{4.7}{1024}n) = (\frac{4.7}{1024})^2\text{Var}(n) \tag{2.2}$$

With the voltages, we calibrate the amplifier by fitting two lines:

1. A line to a graph of v^*_{in} vs v_{out} to determine the gain G_{out}. This line has the equation $v_{\text{out}} = G_{\text{out}}v^*_{\text{in}} + \delta_{\text{out}}$.

2. A line to a graph of v^*_{in} vs v_{in} to quantify the bias currents and obtain the actual input voltage without them. This line has the equation $v_{\text{in}} = G_{\text{in}}v^*_{\text{in}} + \delta_{\text{in}}$.

Both graphs are with respect to v^*_{in} so our calibration factors in the op-amp's bias currents. We fit these lines using Orthogonal Distance Regression [5] (`scipy.odr` in Python) to account for the variance in the values on both axes and to obtain the mean and variance of the fitted line parameters G and δ.

2.2.2 Thermistor calibration

We use our calibrated amplifier to calibrate our thermistor. Figure 2.1 shows our setup. We vary the measured temperature by placing the thermistor and an alcohol thermometer into boiling water and waiting for the water to cool to room temperature. We water-proof our thermistor by heart-shrinking its wires and covering it in epoxy resin to ensure it works in water. As with the amplifier calibration, we measure v_{in} and v_{out} through the $A0$ and $A1$ channels of our Arduino respectively (Appendix A.2.2).

We estimate the thermistor's actual voltage v^*_{in} from both v_{in} and v_{out} by using our fitted lines from amplifier calibration. With amplification:

$$v^*_{\text{in}_{\text{amp}}} \approx \frac{v_{\text{out}} - \delta_{\text{out}}}{G_{\text{out}}} \tag{2.3}$$

And without amplification:

$$v_{\text{in}}^{*} \approx \frac{v_{\text{in}} - \delta_{\text{in}}}{G_{\text{in}}} \tag{2.4}$$

Then, we compute the temperature with and without amplification. With amplification:

$$T_{\text{amp}} \approx \frac{v_{\text{in}_{\text{amp}}}^{*} - 0.4}{0.0195} \tag{2.5}$$

And without amplification:

$$T_{\text{sensor}} \approx \frac{v_{\text{in}}^{*} - 0.4}{0.0195} \tag{2.6}$$

We obtain the mean temperatures by using the above equations and the means of the variables. For the variance, we utilise the fact that out fitted amplifier line calibration parameters $\text{Var}(G)$ and $\text{Var}(\delta)$ are very small (on the order of 10^{-6}) and thus $\text{Var}(v_{\text{in}_{\text{amp}}}^{*}) \approx \text{Var}(v_{\text{out}})$ and $\text{Var}(v_{\text{in}}^{*}) \approx \text{Var}(v_{\text{in}})$. Then, with amplification:

$$\text{Var}(T_{\text{amp}}) \approx (\frac{1}{0.0195})^2 \text{Var}(v_{\text{in}_{\text{amp}}}^{*}) \approx (\frac{1}{0.0195})^2 \text{Var}(v_{\text{out}}) = (\frac{1}{0.0195})^2 (\frac{4.7}{1024})^2 \text{Var}(n_{\text{out}}) \tag{2.7}$$

And without amplification:

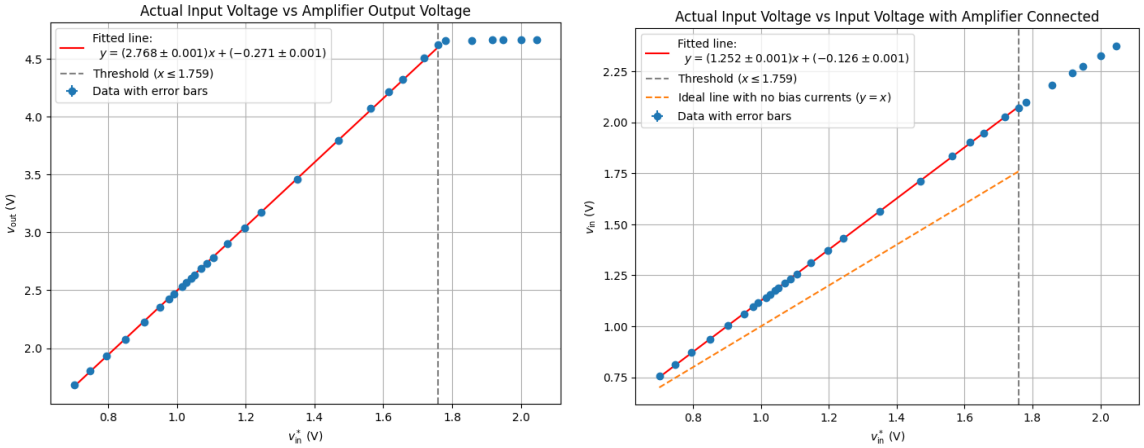
$$\text{Var}(T_{\text{sensor}}) \approx (\frac{1}{0.0195})^2 \text{Var}(v_{\text{in}}^{*}) \approx (\frac{1}{0.0195})^2 \text{Var}(v_{\text{in}}) = (\frac{1}{0.0195})^2 (\frac{4.7}{1024})^2 \text{Var}(n_{\text{in}}) \tag{2.8}$$

With the mean and variance of the measured temperatures with and without amplification, we fit two lines to graphs of the reference temperature from the thermometer T^{*} vs the temperatures from the thermistor with and without amplification, forming our 'calibration curves' (Appendix A.3.1). An 'ideal' thermistor would fit a line with gradient 1 and y-intercept 0. We assess the closeness of our fitted lines to the line of an ideal thermistor to compare the accuracy of the thermistor before and after amplification. With a fitted line $T = mT^{*} + c$, one can then compute the actual temperature from the output of the thermistor by re-arranging: $T^{*} = \frac{T-c}{m}$. This approach improves accuracy by modelling and correcting the error.

2.3 Results

2.3.1 Amplifier calibration

Figure 2.3 (a) and (b) shows our graphs and fitted lines for amplifier calibration. We fit the lines only with $v_{\text{in}}^{*} \leq 1.759$ as the amplifier saturates beyond that input voltage. The error bars are not discernible because the deviation of our measured voltages are negligible when calibrating our amplifier.



(a) (b)

Figure 2.3: Graph and fitted line of (a): v_{in}^* vs v_{out}, (b): v_{in}^* vs v_{in}.

From our results we conclude:

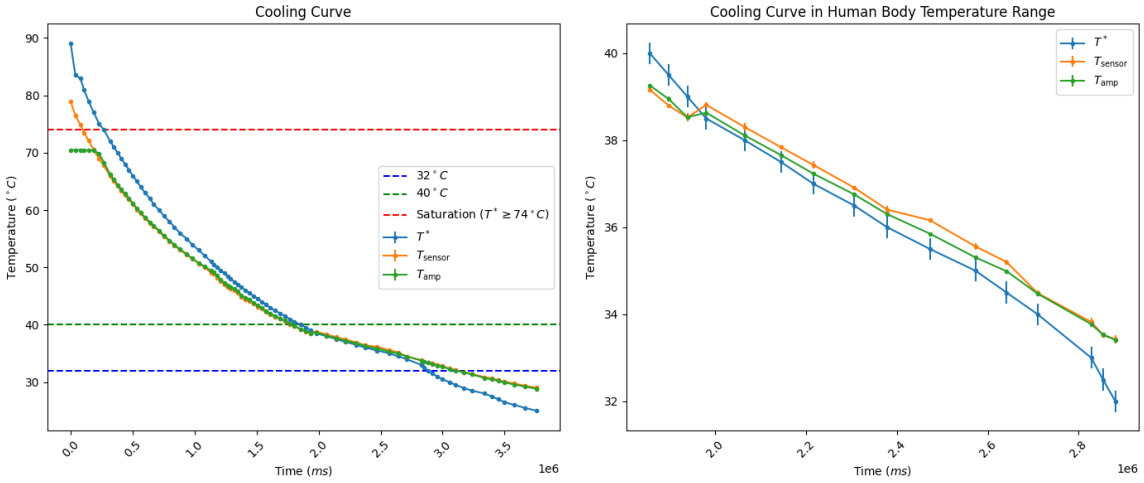
1. Our amplifier's output gain $G_{\text{out}} = 2.768 \pm 0.001$ and output offset $\delta_{\text{out}} = -0.271 \pm 0.001$.

2. Caused by bias currents, our amplifier's input gain $G_{\text{in}} = 1.252 \pm 0.001$ and input offset $\delta_{\text{in}} = -0.0126 \pm 0.001$.

2.3.2 Thermistor calibration

To calibrate our thermistor, we vary T^* from $89°C$ to $25°C$ by boiling water and waiting for it to cool, adding ice at $33°C$ to speed up the cooling. At $1°C$ increments of T^*, we sample the mean and variance of T_{sensor} and T_{amp}. When near the human body temperature range ($32°C$ to $40°C$ since 'normal' human body temperature is $35.7°C$ to $37.3°C$ [6]), we sample at $0.5°C$ increments of T^* to focus on the human body temperature range.

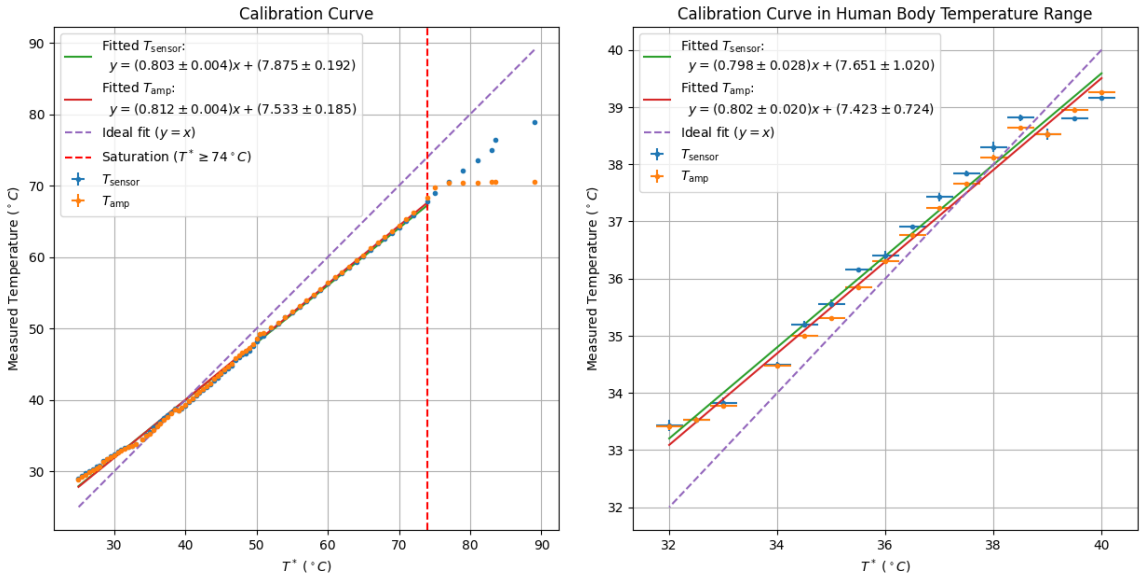
Figure 2.5 shows our cooling curve. We see the largest improvement in the residual error between the real and measured temperatures between $T^* \in [34.5°C, 38.5°C]$. Here, we calculate the residual error directly from the thermistor ($|T^* - T_{\text{sensor}}|$) as 0.458 ± 0.05, and the residual error from the amplifier ($|T^* - T_{\text{amp}}|$) as 0.260 ± 0.04: a 43.2% improvement in accuracy when measuring the temperature from our amplifier than directly from the thermistor.



(a) (b)

Figure 2.4: Cooling curve across all temperatures (a) and within human body temperature range (b).

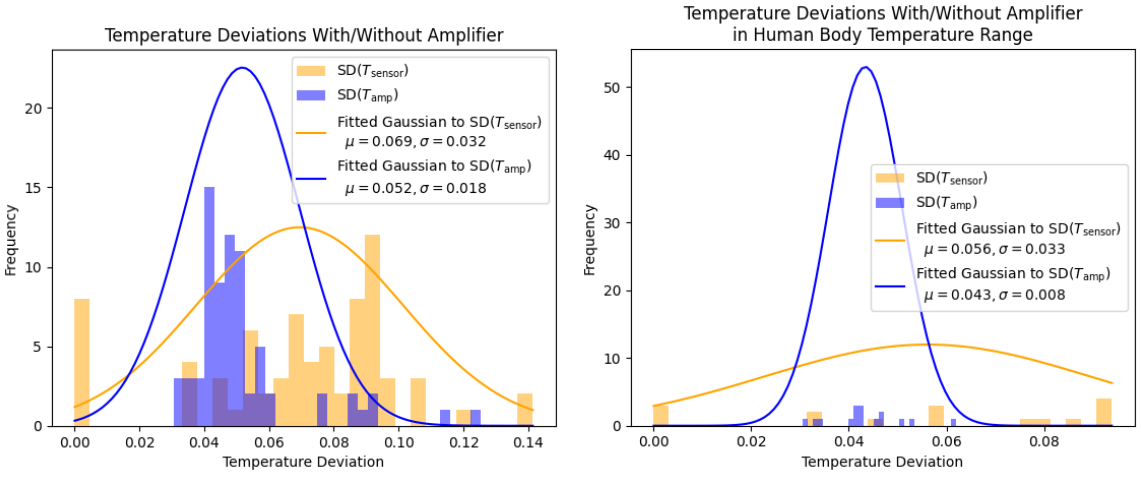
Figure 2.5 shows our calibration curves and the fitted lines outside of the saturation region ($T^* \leq 74°C$). The fitted line to T_{amp} is closer to the ideal/reference line $y = x$ than T_{sensor} across the whole range of temperatures and isolated to the human body temperature range (the gradient is closer to 1 and the intercept is closer to 0), so our temperature measurements are more accurate from the amplifier than from the thermistor directly.



(a) (b)

Figure 2.5: Calibration curve across all temperatures (a) and within human body temperature range (b).

To assess the difference in precision of the measured temperatures with and without the amplifier, we compare their variances relative to each-other, since they are so small that they are not discernible on the cooling and calibration curves. Figure 2.6 shows a histogram of the deviation of T_{sensor} and T_{amp}. We fit a Gaussian to these histograms to quantify the mean and variance of the deviations of the measured temperatures: informing us of the 'average' precision and 'spread' in precision of our measurements. In the human body temperature range, we see that the mean deviation in the measured temperature reduces from 0.056 directly from the thermistor to 0.043 with the amplifier: a 23% increase in average precision. Furthermore, the deviations are more consistent with the amplifier than with the thermistor directly. The deviation reduces from 0.033 to 0.008: a 75.8% reduction in the spread of the precisions.



(a) (b)

Figure 2.6: Histogram of temperature deviations before and after calibration, across all temperatures (a) and within the human body temperature range (b).

2.4 Discussion

Overall, we successfully enhanced thermistor resolution, increasing its accuracy by 43.2% and precision by 23%, and reducing the variability in its precision at different temperatures by 75.8%. Waterproofing extends its use to wet environments, such as in the mouth. Calibration focused on actual rather than potential device variances since we calibrate our specific fabrication of our MCP9701, which is why the measured variances are much lower than the variances across all fabrications of the MCP9701 listed in its data-sheet [3].

Our calibration has limitations which can be addressed in the future:

1. The resolution improvement is only strong within the human body temperature range. This is because we weight our amplification calibration data-set towards voltages within the human body temperature range, making our fitted model biased to that region. If we wish to expand the range of measured temperatures of our device, then we would likely need to employ non-linear modelling techniques as the data is slightly non-linear (as mentioned our op-amp's and thermistor's data-sheets [4, 3]). Alternatively, one could separately calibrate to different temperature regions.

2. We calibrate our thermistor across a much larger range than the human body temperature, limiting our gain from our amplifier. If we only calibrate our thermistor across the human body temperature range then we could use a much larger gain without saturating the amplifier at the upper limit of the temperature range, further improving the accuracy and precision. We can subtract the voltage at the lowest limit

of the temperature range with a differential amplifier to further increase the possible gain.

3. Our op-amp has strong bias currents, complicating our method. We could use a better op-amp in the future to avoid the error introduced by the op-amp and simplify our method.

Chapter 3

Pulse sensor

3.1 Introduction to pulse meters

The pulse meter employs photoplethysmography (PPG), using an LED to measure light absorption by a finger [7]. This reflects blood pressure variations, correlating observed light with cardiovascular patterns (Figures 3.1,3.2).

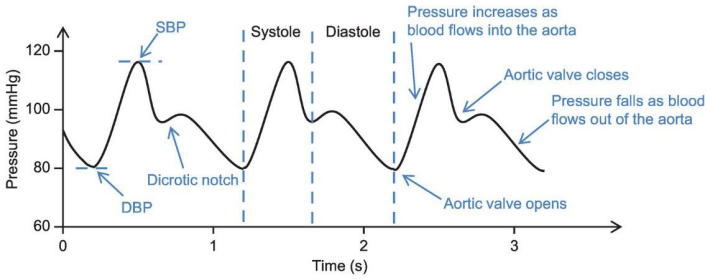


Figure 3.1: Blood pressure vs time in an aortic valve. Source: [8]

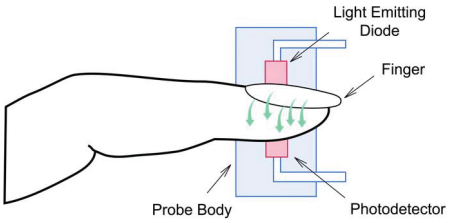


Figure 3.2: Shining light from an LED into a finger and detecting it with a Photodetector. Source: [9]

Figure 3.3 shows a graph of the received light from the LED by the photodetector. The AC component denotes the absorbed light by the arterial blood, which exhibits the same pattern as the blood pressure over time. The rest of the light forms the DC component,

which is absorbed in a consistent fashion by the non-pulsating blood, venous blood and other body tissue.

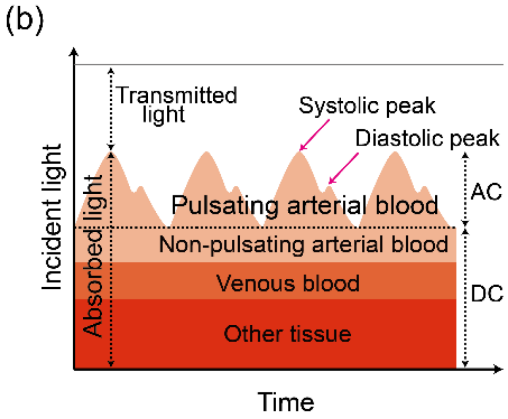


Figure 3.3: Incident light to the photodiode from the LED having passed through the finger, as a function of time. Source: [8]

Since a photodiode outputs a current proportional to its incident light intensity, it outputs a current with a small AC component a shown in Figure 3.3. By isolating this AC component, we can measure a signal of the user's heart-rate, and then use the signal to estimate their heart-rate.

3.2 Setup

The pulse meter senses heart-beats by detecting light from an LED via a photodetector (Figure 3.4). Users place their finger in the clip until their heart-rate is measured. This section details our setup components.

3.2.1 Circuit

The finger clip's photodiode creates a current based on light intensity, converted to voltage by our circuit, isolating the AC signal component. Circuit schematics are in Figure 3.5, with implementation on a breadboard (Figure 3.4).

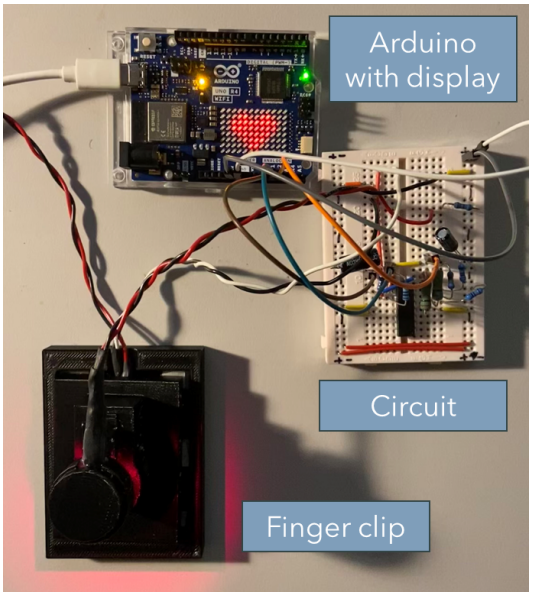
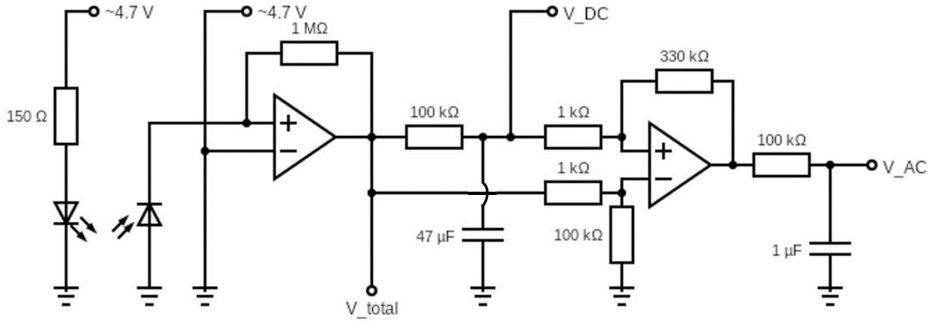
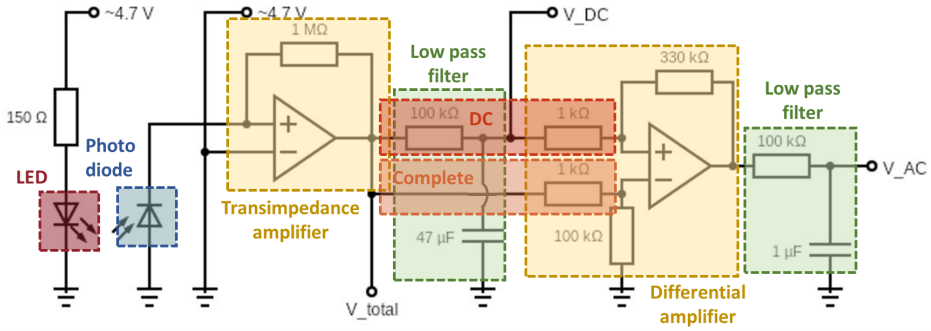


Figure 3.4: Physical setup of our pulse meter.



(a)



(b)

Figure 3.5: Pulse meter circuit (a) and annotated (b).

From left to right, the circuit operates as follows:

1. Light is emitted from the LED and received by the photodiode.
2. The photodiode generates a current proportional to the intensity of the light receieved by it.
3. The transimpedance amplifier generates a voltage proportional to the current generated by the photodiode.
4. The complete signal, $V_{\text{total}} = V'_{\text{AC}} + V_{\text{DC}}$, from the transimpedance amplifier is passed through a low-pass filter to filter out the high-frequency AC component V'_{AC} and obtain the ~ 0-frequency DC component V_{DC}.
5. The complete and DC portions of the signal are passed through a differential amplifier to obtain the AC component and amplify it. This works because $V_{\text{total}} - V_{\text{DC}} = V'_{\text{AC}}$ and the differential amplifier outputs a voltage $G(V_{\text{total}} - V_{\text{DC}}) = GV'_{\text{AC}}$, where G is the gain of amplifier.
6. The amplified AC signal from the differential amplifier is fed through another low-pass filter to remove any high-frequency noise, obtaining the isolated and 'cleaned' AC component V_{AC}.

We arrive at the shown resistances and capacitances through trial-and-error.

3.2.2 Finger clip

Our custom finger clip, modelled with Fusion360 and shown in Figure 3.6, enhances the pulse meter's optics. Features include: two LEDs for PPG in reflectance or transmission mode (which perform differently depending on the user [7]), a flexible material adapting to different finger sizes, a detachable base with Velcro for repairability, and a cover to ensure light isolation.

3.2.3 Offline data analysis

For offline Python analysis, we measure V_{total}, V_{DC} and V_{AC} via Arduino's $A0$, $A1$, and $A2$ channels (Appendix A.2.3). Each loop iteration averages the last four readings for signal smoothing and records the value with a timestamp. We also log the sampling frequency, calculated from loop iterations over time. This data is stored and imported in Python for further analysis. There, we visualise the signal using the `plotly` library and isolate it to timesteps at which V_{AC} looks like a heartbeat. This happens after the first low-pass filter's capacitor fully charges and the DC component is filtered out. With the V_{AC} signal isolated to the 'converged' region, we estimate the heart-rate from it with two algorithms: peak detection and Fourier transform (Appendix A.3.2).

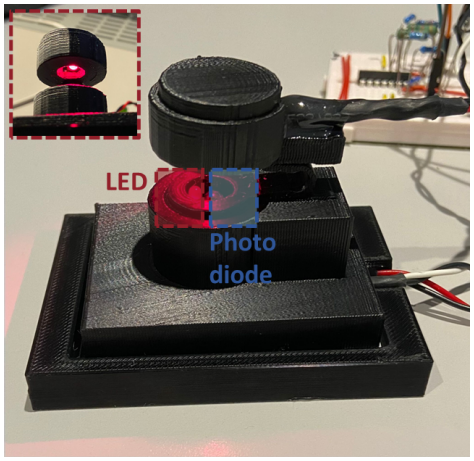


Figure 3.6: Pulse-meter finger clip.

Peak detection

Given our AC signal looking like the signal shown in Figure 3.1, we identify the peaks (SBP) using `scipy.signal.find_peaks`. Then, given the time in milliseconds of each peak, we compute mean $\bar{\Delta}$ and standard error $\text{SE}(\Delta)$ of the time in between successive peaks and the standard error of and then estimate the heart-rate as:

$$\begin{aligned} \text{BPM} &\approx \left(\frac{60 * 1000}{\bar{\Delta}}\right) \pm \left(\left|\frac{60 * 1000}{\bar{\Delta}} - \frac{60 * 1000}{\bar{\Delta} + \text{SE}(\Delta)}\right|\right) \\ &= \left(\frac{60 * 1000}{\bar{\Delta}}\right) \pm \left(60 * 1000 * \left|\frac{\text{SE}(\Delta)}{\bar{\Delta}(\bar{\Delta} + \text{SE}(\Delta))}\right|\right) \end{aligned} \tag{3.1}$$

To prevent the Diocratic notch from being detected as a peak (and then causing the heart-rate to be estimated), we supply a `distance` parameter of ~ 5 to the algorithm to only detect peaks at least 5 data-points apart. We choose 5 since the minimum time we can expect between the peaks corresponds to the upper limit of a human's heart-rate, which we estimate as 240 BPM, which corresponds to a time of $60/240 = 0.25$ seconds between each heart-beat. Given that we measure our sampling frequency as ~ 21Hz, each data-point is $\sim 1/21 = 0.0476$ seconds apart which corresponds to $0.25/0.0476 = 5.25$ data-points at our sampling frequency.

Fourier transform

Our heart-beat signal is mostly characterised by a sin wave with frequency equivalent to the heart-rate. Therefore, we estimate the heart-rate frequency in Hz by running a fast Fourier transform on the signal using `scipy.fftpack`. We use the recorded sampling

frequency it to correctly size the bins. After the Fourier transform we then obtain the frequency f^* (in Hz) with maximal amplitude (in the range of 0.5 to 4.2Hz as these frequencies correspond to feasible heart-rates) and convert it to a heart-rate:

$$\text{BPM} \approx 60 f^* \tag{3.2}$$

3.2.4 Real-time detection

Separate signal processing in Python provides slow feedback to a user of the pulse meter. To address this we display their heart-rate in real-time on the Arduino's LED display by running a simplified heart-rate estimation algorithm on it (Appendix A.2.4). We store the last 10 seconds of the AC signal, and then count the number of times N that the voltage passes from below to above the mean. Since the mean is approximately in the center of the signal, we expect one such positive crossing on every heart-beat. The Dicrotic notch typically occurs above the mean so we do not count the heart beats twice. Then, we estimate the heart-rate as:

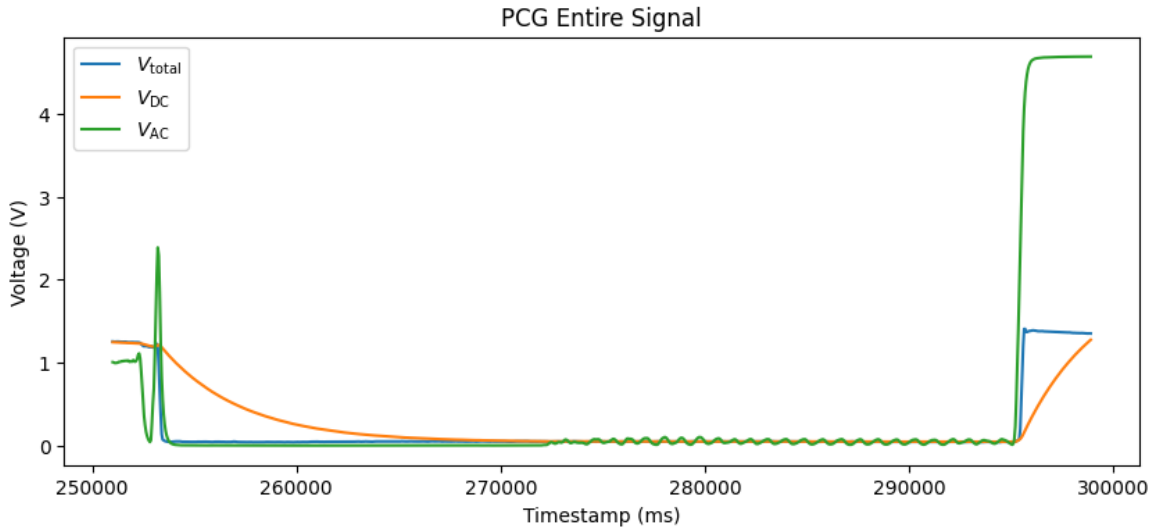
$$\text{BPM} \approx 60 * \frac{N}{10} \tag{3.3}$$

If our heart-rate estimate is within a reasonable heart-rate range (30 to 250 BPM) then we display it on the LED display one digit at a time, followed by a heart after displaying the full number. To avoid confusion before our circuit has converged, we display a static heart in this period.

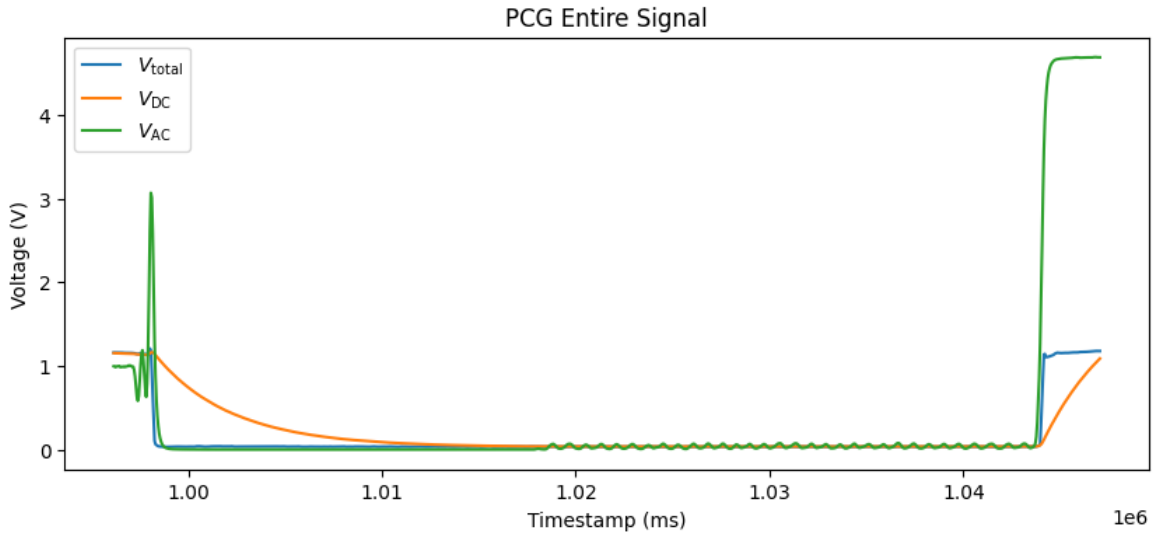
3.3 Results

3.3.1 Offline data analysis

Before and after exercise, Figure 3.7 shows our full AC signal, Figure 3.8 the detected peaks from peak detection in the converged region, and Figure 3.9 shows a graph of the Fourier transformed signal and the maximal amplitude frequency.

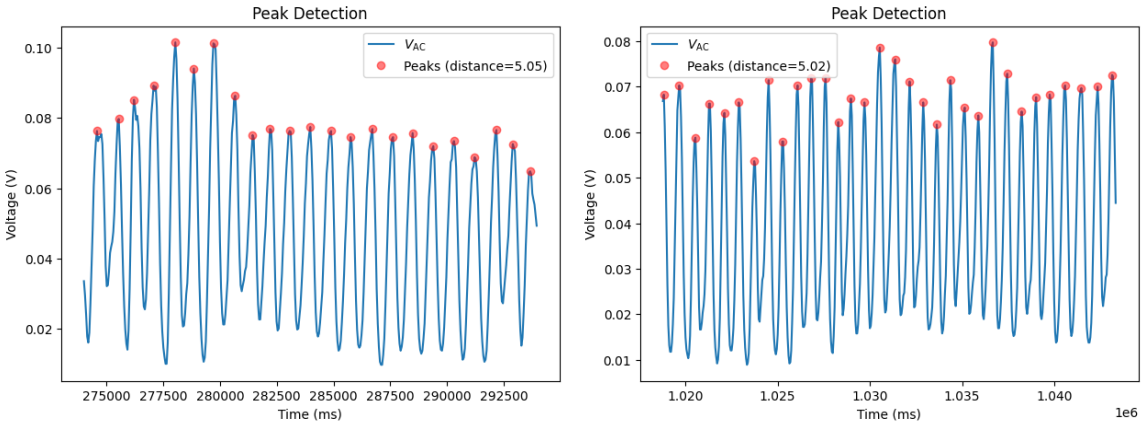


(a)



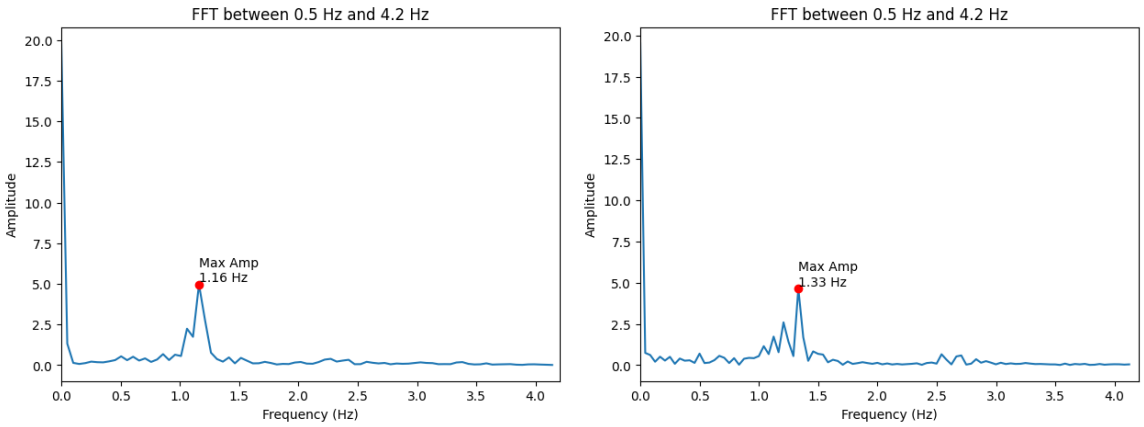
(b)

Figure 3.7: Full AC signal before (a) and after (b) exercise.



(a) (b)

Figure 3.8: Peak detection on the converged AC signal before (a) and after (b) exercise.



(a) (b)

Figure 3.9: FFT on the converged AC signal before (a) and after (b) exercise.

Before exercise, $\Delta = 852.2 \pm 16.1$ms and therefore, by Equation 3.1, BPM $\approx 70.4 \pm 1.3$ from peak detection. $f^* = 1.17$Hz and therefore, by Equation 3.2, BPM ≈ 70.0 from the Fourier transform.

After exercise, $\Delta = 783.4 \pm 9.5$ms and therefore, by Equation 3.1, BPM $\approx 76.6 \pm 0.9$ from peak detection. $f^* = 1.33$Hz and therefore, by Equation 3.2, BPM ≈ 80.0 from the Fourier transform.

Heart-rate measurements before and after exercise show expected increases, with both peak detection and Fourier transform methods providing similar results within error margins before exercise.

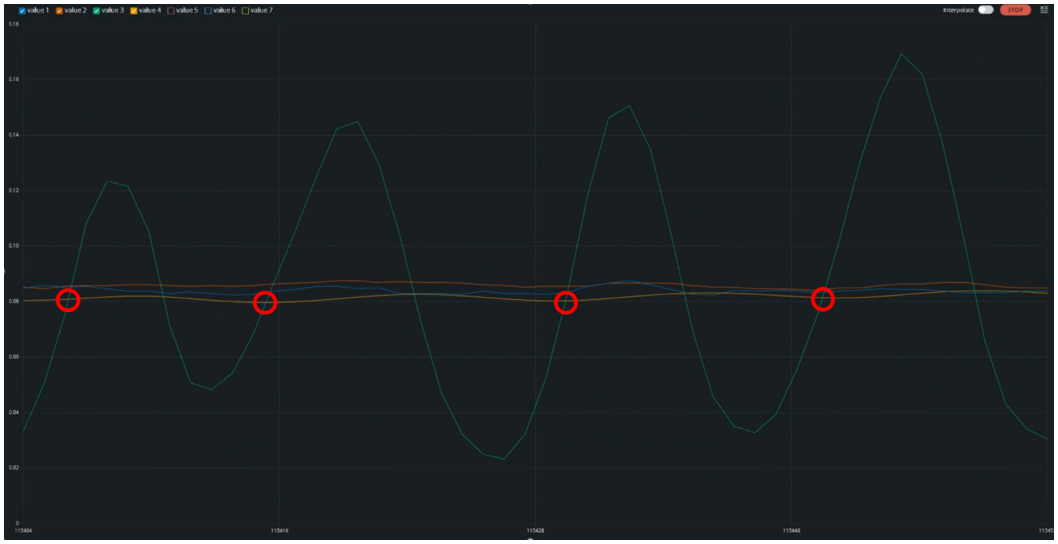


Figure 3.10: Serial plotter output after the heart-rate converges on the Arduino. The green line is V_{AC}, the yellow line is the mean, and the red circles are positive gradient crossings over the mean.

3.3.2 Real-time detection

Before exercising, after recording the data for offline analysis we measure the heart-rate directly on our Arduino. Figure 3.10 shows our obtained heart-beat signal. Since the heart-rate is based on crossings through the mean, and the mean is accurate, we obtain accurate heart-rate estimations. We read a heart-rate of 73 BPM from the LED display: close to the BPM computed offline (70) and validating our real-time detection method.

3.4 Discussion

Overall, our PPG implementation successfully detects a user's heart-beat. We obtain a smooth AC signal corresponding to a user's heart-beat from our circuit, and our online and offline methods for analysing the signal reasonably estimate their heart-rate after the capacitors in the circuit have converged.

Our setup is especially equitable because it can be battery-powered and does not require external hardware to process the signal and display it. Along with the cheap electronics and repairable finger-clip, our solution is accessible for low-income healthcare settings. Furthermore, our solution is adaptable to different body types as our finger-clip can operate in reflectance and transmission mode [7].

However, our solution has some limitations which can be addressed in the future:

1. The capacitor for filtering the DC signal takes a while to converge ($\sim 10s$), delaying heart-rate readings. We can improve the experience by fine-tuning the capacitors and resistors in the circuit for faster convergence.

2. Our simplified real-time detection algorithm is sensitive to the amplitude of the AC signal despite it only estimating the frequency. More advanced algorithms like the Fourier transform could be implemented on the Arduino to improve the accuracy and robustness of the real-time detection.

3. No measure of uncertainty is available from our offline Fourier transform or online peak detection methods. The uncertainty could be estimated and communicated to the user to gauge confidence.

Chapter 4

Conclusion

In conclusion, the project successfully developed affordable, user-friendly temperature and pulse sensors by enhancing the resolution of a thermistor and implementing a PPG-based pulse detector. The project revealed challenges and potential in healthcare sensor technology. Further, the author developed their skills in electronics and data analysis. Looking ahead, future efforts could involve enhancing the precision and ease of use of our sensor designs, and exploring their broader applications in diverse healthcare environments, furthering their impact on global health.

Bibliography

[1] United Nations. *The 17 Goals*. https://sdgs.un.org/goals. Accessed: 18 December 2023. 2015.

[2] Oliver Hadeler. *Lecture notes titled "Introduction to temperature sensors", provided for the Sensor Design Project within the Sensor CDT at the University of Cambridge*. Accessed: 18 December 2023.

[3] *Low-Power Linear Active Thermistor ICs*. MCP9701. DS20001942G. Microchip Technology Inc. 2016.

[4] *1 MHz, Low-Power Op Amp*. MCP6004. DS20001733L. Microchip Technology Inc. 2020.

[5] P. T. Boggs and J. E. Rogers. "Orthogonal Distance Regression". In: *Statistical Analysis of Measurement Error Models and Applications: Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference Held June 10-16, 1989*. Vol. 112. Contemporary Mathematics. 1990, p. 186.

[6] Ivayla I Geneva et al. "Normal Body Temperature: A Systematic Review". In: *Open Forum Infectious Diseases* 6.4 (Apr. 2019), ofz032. ISSN: 2328-8957. DOI: 10.1093/ofid/ofz032. eprint: https://academic.oup.com/ofid/article-pdf/6/4/ofz032/28311638/ofz032.pdf. URL: https://doi.org/10.1093/ofid/ofz032.

[7] Jiří Přibil, Anna Přibilová, and Ivan Frollo. "Comparative Measurement of the PPG Signal on Different Human Body Positions by Sensors Working in Reflection and Transmission Modes". In: *Engineering Proceedings* 2.1 (2020). ISSN: 2673-4591. DOI: 10.3390/ecsa-7-08204. URL: https://www.mdpi.com/2673-4591/2/1/69.

[8] Oliver Hadeler. *Lecture titled "Pulse oximetry", held at the University of Cambridge for the Sensor Design Project within the Sensor CDT*. Nov. 2023.

[9] Nuwan D Nanayakkara, S C Munasingha, and G P Ruwanpathirana. "Non-Invasive Blood Glucose Monitoring using a Hybrid Technique". In: *2018 Moratuwa Engineering Research Conference (MERCon)*. 2018, pp. 7–12. DOI: 10.1109/MERCon.2018.8421885.

Appendix A

Appendix

Here we provide additional explanations of the fundamentals behind our electronics, and our Arduino and Python code used for data acquisition and analysis.

A.1 Electronics explanations

A.1.1 Analogue to digital converters

When reading voltages from an Arduino, the resolution of the measured voltage is limited by the Arduino's Analogue to Digital Converter (ADC). The ADC converts the continuous analogue voltage into a discrete digital value with N bits, which can then be processed with code. With a supply/reference voltage v_{ref}, the ADC splits v_{ref} into 2^N steps. For an input voltage v_{in} to the Arduino, the digital output value from the ADC n_{ADC} is given by

$$n_{\text{ADC}} = \lfloor \frac{2^N}{v_{\text{ref}}} v_{\text{in}} \rfloor \tag{A.1}$$

Then, to obtain the approximate value of v_{in} from n_{ADC}, denoted as v_{ADC}, we reverse the discretisation:

$$v_{\text{ADC}} = \frac{v_{\text{ref}}}{2^N} n_{\text{ADC}} \tag{A.2}$$

Now, $v_{\text{ADC}} \approx v_{\text{in}}$ because:

$$v_{\text{ADC}} = \frac{v_{\text{ref}}}{2^N} n_{\text{ADC}} = \frac{\cancel{v_{\text{ref}}}}{\cancel{2^N}} \lfloor \frac{\cancel{2^N}}{\cancel{v_{\text{ref}}}} v_{\text{in}} \rfloor \approx v_{\text{in}} \tag{A.3}$$

Note that, since the ADC approximates the input voltage as an integer, we apply the floor and the terms do not exactly cancel, producing some error.

Our Arduino supplies voltages of $5V$ or $3.3V$, and its default ADC resolution is $N = 10$ bits. Therefore, with $v_{\text{ref}} = 5 - 0.3 = 4.7V$ (accounting for the $0.3V$ lost to an internal diode), the input voltage from the digital output value from the ADC is given by:

$$v_{\text{ADC}} = \frac{4.7}{1024} n_{\text{ADC}} \tag{A.4}$$

Our Arduino uses a Successive Approximation (SAR) ADC, which uses a single comparator and converges to v_{digital} over some time. SAR ADCs have the advantage of being cheap, however their converging nature means they can be too slow in certain domains. For our purposes, it converges fast enough.

A.1.2 Amplifiers

Amplifiers are electrical components that can be used to improve the resolution of measured voltages by an Arduino. A common kind is a non-inverting operational amplifier (op amp): an op amp with added resistors and feedback to tame the gain to a reasonable value. For a normal op amp with input voltage v_{in}, reference voltage v_- (usually ground), and gain G, the output voltage is given by $v_{\text{out}} = G(v_{\text{in}} - v_-)$ with $G \approx 10^5$ to 10^9. This huge gain is unusable in practice. To tame it to more reasonable value, we use a non-inverting op amp, as shown in Figure A.1. It contains a regular op amp (the triangular component), two extra resistors R_1 and R_2, and feedback from the output to the reference voltage. The gain of a non-inverting op amp is given by $G = \frac{v_{\text{out}}}{v_{\text{in}}} = 1 + \frac{R_1}{R_2}$. For $R_1 = 3.3k\Omega$ and $R_2 = 2.2k\Omega$, $G = 1 + \frac{3.3}{2.2} = 2.5$: much more reasonable than the normal op amp.

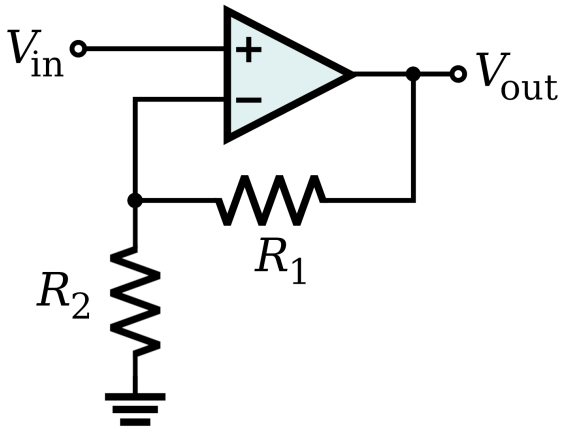


Figure A.1: Non-inverting op amp. Source: Wikimedia.

A.1.3 Voltage to temperature conversion of a thermistor

To calculate the measured temperature by the thermistor using our estimate of its output voltage (v_{ADC}), we refer to its voltage characteristics in its data-sheet. According to the MCP9701 data-sheet, the thermistor has a bias voltage ($V_{0°C}$) of $400mV$ at $0°C$ and a temperature coefficient (T_C) of $19.5mV/°C$. Therefore, the thermistor's output temperature T is given by:

$$T \approx \frac{v_{\text{ADC}} - V_{0°C}}{T_C} = \frac{v_{\text{ADC}} - 0.4}{0.0195} \tag{A.5}$$

Note $\approx$ because we are calculating the temperature using our estimated voltage from the ADC. Now, substituting in Equation A.4, we have

$$T \approx \frac{\frac{4.7}{1024} n_{\text{ADC}} - 0.4}{0.0195} \tag{A.6}$$

Equation A.6 expresses the full mapping from $n_{\text{ADC}} \to T$, which one would implement on their Arduino to obtain the measured temperature from the digital value returned by `analogRead`.

A.1.4 Increasing the resolution of a thermistor

For a thermistor connected to an Arduino, there are several ways to increase its temperature resolution:

1. Use an ADC with more bits or a supply a lower reference voltage. These approaches increase N or decrease v_{ref} respectively in Equation A.1, increasing the number of ADC steps n_{ADC} and hence decreasing the voltage step of the ADC given by $\frac{v_{\text{ref}}}{2^N}$ in Equation A.2.

2. Amplify the thermistor's output voltage so it covers a larger range of ADC steps. According to Equation A.1, for $v_{\text{in}} \in [v_-, v_+]$, the ADC outputs values:

$$n_{\text{ADC}} \in \left[\lfloor \frac{2^N}{v_{\text{ref}}} v_- \rfloor, \lfloor \frac{2^N}{v_{\text{ref}}} v_+ \rfloor \right] \tag{A.7}$$

   Therefore, the number of values output by the ADC, $\Sigma^{\text{in}}_{\text{ADC}}$ is given by:

$$\Sigma^{\text{in}}_{\text{ADC}} = \lfloor \frac{2^N}{v_{\text{ref}}} v_+ \rfloor - \lfloor \frac{2^N}{v_{\text{ref}}} v_- \rfloor \approx \frac{2^N}{v_{\text{ref}}} (v_+ - v_-) \tag{A.8}$$

   Now, if we amplify v_{in} with a gain of G, $v_{\text{out}} \in [Gv_-, Gv_+]$ and by the same logic we have:

$$\Sigma^{\text{out}}_{ADC} \approx \frac{2^N}{v_{\text{ref}}} (Gv_+ - Gv_-) = G \frac{2^N}{v_{\text{ref}}} (v_+ - v_-) = G\Sigma^{\text{in}}_{\text{ADC}} \tag{A.9}$$

   Therefore, we map the input voltage from the thermistor to G times as many ADC steps after amplifying the input voltage by a factor of G.

These methods aim to account for the error introduced by the ADC. To put this error into perspective, our temperature step is given by Equation A.6 as $4.7/1024/0.0195 = 0.24^\circ C$ (the coefficient of n_{ADC}). Since our ADC outputs only the integer part of n_{ADC}, the maximal error from the ADC on the measured temperature (assuming the ADC has converged to the correct digital value) is $0.24^\circ C$. Therefore, any efforts to improve the resolution from the ADC may reduce the error in the output temperature by at most $0.24^\circ C$.

A.2 Arduino code

A.2.1 Amplifier calibration data acqusition

```
#include <queue>
#include <limits>

int inputPin = A0; // voltage being input to the amplifier
int amplifiedInputPin = A1; // amplified voltage

int numReadings = 100; // number of measurements to average over

std::deque<float> inputs;
std::deque<float> amplifiedInputs;

int adcRes = 10;
int adcMax;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println("I␣belong␣to␣Omar.␣This␣program␣is␣amplifier␣calibration.");
  analogReadResolution(adcRes);
  adcMax = pow(2, adcRes);
}

void loop() {
  int inputValue = analogRead(inputPin);
  int amplifiedInputValue = analogRead(amplifiedInputPin);

  float inputVoltage = (float)inputValue * 4.7 / (float)adcMax;
  float amplifiedInputVoltage = (float)amplifiedInputValue * 4.7 / (float)adcMax;

  Serial.println(amplifiedInputVoltage / inputVoltage);

  if (inputs.size() == numReadings) {
    inputs.pop_front();
    amplifiedInputs.pop_front();
  }
  inputs.push_back(inputVoltage);
  amplifiedInputs.push_back(amplifiedInputVoltage);

  // compute mean and std dev. and output
  unsigned long currentTime = millis();
  Serial.print(currentTime);
  Serial.print(":");
```

```
  outputStatistics(inputs);
  outputStatistics(amplifiedInputs);
  Serial.println();
}

void outputStatistics(std::deque<float>& in_values) {
  if (in_values.size() == 0) {
    return;
  }
  float sum = 0;
  for (float value : in_values) {
    sum += value;
  }
  float mean = sum / in_values.size();
  float sum_sq_delta = 0;
  for (float value : in_values) {
    sum_sq_delta += pow(value - mean, 2);
  }
  float deviation = sqrt(sum_sq_delta / in_values.size());
  Serial.print(mean, 10);
  Serial.print(",");
  Serial.print(deviation, 10);
  Serial.print(",");
}
```

A.2.2 Thermistor calibration data acqusition

```
#include <queue>
#include <limits>

int sensorPin = A0; // original temp in
int amplifiedSensorPin = A1; // amplified temp in

int numReadings = 30; // number of measurements to average over

std::deque<int> inputsSensor;
std::deque<int> inputsAmplified;

int adcRes = 10; // number of bits resolution of arduino ADC
int adcMax;

/*
 output schema:
   <timestamp>:<mean of adc value of input to amplifier>,<deviation of adc value of input to amplifier>,<
       ↪ mean of adc value of output from amplifier>,<deviation of adc value of output from amplifier>
```

```
*/
void setup() {
  // put your setup code here, to run once:

  Serial.begin(9600);
  Serial.println("I␣belong␣to␣Omar.␣This␣program␣is␣temperature␣sensor␣calibration.");
  analogReadResolution(adcRes);
  adcMax = pow(2, adcRes);
}

void loop() {
  int sensorValue = analogRead(sensorPin);
  int amplifiedSensorValue = analogRead(amplifiedSensorPin);

  if (inputsSensor.size() == numReadings) {
    inputsSensor.pop_front();
    inputsAmplified.pop_front();
  }
  inputsSensor.push_back(sensorValue);
  inputsAmplified.push_back(amplifiedSensorValue);

  unsigned long currentTime = millis();
  Serial.print(currentTime);
  Serial.print(":");

  float meanInputSensor = 0;
  float varianceInputSensor = 0;
  computeMeanAndVariance(inputsSensor, meanInputSensor, varianceInputSensor);
  float meanInputAmplified = 0;
  float varianceInputAmplified = 0;
  computeMeanAndVariance(inputsAmplified, meanInputAmplified, varianceInputAmplified);

  // Output just the raw amplifier input and output ADC values
  Serial.print(meanInputSensor, 10);
  Serial.print(",");
  Serial.print(sqrt(varianceInputSensor), 10);
  Serial.print(",");
  Serial.print(meanInputAmplified, 10);
  Serial.print(",");
  Serial.println(sqrt(varianceInputAmplified), 10);

}

void computeMeanAndVariance(std::deque<int>& in_values, float& out_mean, float&
    ↪ out_variance) {
  if (in_values.size() == 0) {
    return;
```

```
  }
  float sum = 0;
  for (int value : in_values) {
    sum += (float)value;
  }
  out_mean = sum / in_values.size();
  float sum_sq_delta = 0;
  for (int value : in_values) {
    sum_sq_delta += pow(value − out_mean, 2);
  }
  out_variance = sum_sq_delta / in_values.size();
}
```

A.2.3 Pulse data acqusition

```
#include <queue>
#include <limits>

bool print_time = true;

std::deque<float> inputs_A0;
std::deque<float> inputs_A1;
std::deque<float> inputs_A2;

int adcRes = 12;
int adcMax;

int numReadings = 4;

long numIterations = 0;
long startTime;

void setup() {
    Serial.begin(9600);
    analogReadResolution(adcRes);
    adcMax = pow(2, adcRes);
    Serial.println(adcMax);

    delay(1500);
    startTime = millis();
}

void loop() {
  if (print_time) {
    unsigned long currentTime = millis();
```

```
    Serial.print(currentTime);
    Serial.print(":");
  }

  numIterations++;
  float timePerIteration = (abs(millis() - (float)startTime) / (float)numIterations);
  float hz = 1000 / timePerIteration;
  Serial.print(hz, 10);
  Serial.print(",");

  float v_0 = analogRead(A0) * (4.7 / (float)adcMax);
  float v_1 = analogRead(A1) * (4.7 / (float)adcMax);
  float v_2 = analogRead(A2) * (4.7 / (float)adcMax);

  if (inputs_A0.size() == numReadings) {
    inputs_A0.pop_front();
    inputs_A1.pop_front();
    inputs_A2.pop_front();
  }
  inputs_A0.push_back(v_0);
  inputs_A1.push_back(v_1);
  inputs_A2.push_back(v_2);

  float mean_v0 = mean(inputs_A0);
  float mean_v1 = mean(inputs_A1);
  float mean_v2 = mean(inputs_A2);

  Serial.print(mean_v0, 5);
  Serial.print(",");
  Serial.print(mean_v1, 5);
  Serial.print(",");
  Serial.println(mean_v2, 5);
}

float mean(std::deque<float>& in_values) {
  if (in_values.size() == 0) {
    return -1;
  }
  float sum = 0;
  for (float value : in_values) {
    sum += value;
  }
  return sum / in_values.size();
}
```

A.2.4 Online pulse detection

Note that we use the open-source `fonts.h` file from the following website: `https://arduinogetstarted.com/tutorials/arduino-uno-r4-led-matrix-displays-number-character`.

```
#include <queue>
#include <limits>
#include "Arduino_LED_Matrix.h"
#include "fonts.h"

bool print_time = false;

std::deque<float> inputs_A0;
std::deque<float> inputs_A1;
std::deque<float> inputs_A2;

std::deque<float> q_signal;
std::deque<long> q_timestamps;
int signalSize = 200; // 200 readings

int adcRes = 12;
int adcMax;

int numReadings = 4;
// long delay_ms = 20;

long numIterations = 0;
long startTime;

float currentHeartRate = 0;

// digit display stuff
ArduinoLEDMatrix matrix;
uint8_t frame[8][12] = {
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
  { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
};
uint8_t heart[8][12] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0 },
    { 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0 },
```

```
    { 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0 },
    { 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0 },
    { 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
};
long previousHeartRateDisplayTime = 0;
long heartRateUpdateInterval = 500;
std::queue<int> displayHeartRate;
char digits[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
bool displayingHeart = false;

void setup() {
    Serial.begin(9600);
    analogReadResolution(adcRes);
    adcMax = pow(2, adcRes);
    Serial.println(adcMax);

    delay(1500);
    matrix.begin();
    startTime = millis();
}

void loop() {
  if (print_time) {
    unsigned long currentTime = millis();
    Serial.print(currentTime);
    Serial.print(":");
  }

  numIterations++;
  float timePerIteration = (abs(millis() - (float)startTime) / (float)numIterations);
  float hz = 1000 / timePerIteration;

  float v_0 = analogRead(A0) * (4.7 / (float)adcMax);
  float v_1 = analogRead(A1) * (4.7 / (float)adcMax);
  float v_2 = analogRead(A2) * (4.7 / (float)adcMax);

  if (inputs_A0.size() == numReadings) {
    inputs_A0.pop_front();
    inputs_A1.pop_front();
    inputs_A2.pop_front();
  }
  inputs_A0.push_back(v_0);
  inputs_A1.push_back(v_1);
  inputs_A2.push_back(v_2);
```

```
float mean_v0 = mean(inputs_A0);
float mean_v1 = mean(inputs_A1);
float mean_v2 = mean(inputs_A2);

Serial.print(mean_v0, 6);
Serial.print(",");
Serial.print(mean_v1, 6);
Serial.print(",");
Serial.print(mean_v2, 6);
Serial.print(",");

if (q_signal.size() == signalSize) {
  q_signal.pop_front();
  q_timestamps.pop_front();
}
q_signal.push_back(mean_v2);
long now = millis();
q_timestamps.push_back(now);

currentHeartRate = computeHeartRate();
Serial.println(currentHeartRate);

// update heart rate display
if (currentHeartRate < 30 || currentHeartRate > 250) {
  if (!displayingHeart) {
    // display heart
    clear_frame();
    matrix.renderBitmap(heart, 8, 12);
    displayingHeart = true;
  }
  heartRateUpdateInterval = 0;
}
else {
  // should display heart rate.
  displayingHeart = false;
  if (now >= previousHeartRateDisplayTime + heartRateUpdateInterval) {
    clear_frame();
    previousHeartRateDisplayTime = now;
    if (displayHeartRate.size() == 0) {
      // We have displayed each digit.
      // Populate the digit queue.
      populateHeartRateDigits(currentHeartRate);
      // make the digits show at a rate corresponding to the heart rate!
      heartRateUpdateInterval = (long)((1000.0 * (60.0 / currentHeartRate)) / (float)(
          ↪ displayHeartRate.size() + 1));
      // Display heart <3
      matrix.renderBitmap(heart, 8, 12);
```

```
        }
        else {
          // Pop a digit from the heart rate queue and update.
          int heartRateDigit = displayHeartRate.front();
          displayHeartRate.pop();
          add_to_frame(digits[heartRateDigit], 4);
          display_frame();
        }
      }
    }
}

void populateHeartRateDigits(int heartRate) {
  if(heartRate >= 10)
        populateHeartRateDigits(heartRate / 10);

    int digit = heartRate % 10;

    displayHeartRate.push(digit);
}

float mean(std::deque<float>& in_values) {
  if (in_values.size() == 0) {
    return -1;
  }
  float sum = 0;
  for (float value : in_values) {
    sum += value;
  }
  return sum / in_values.size();
}

float computeHeartRate() {
  float signalDuration = (float)abs(q_timestamps.back() - q_timestamps.front());

  // compute mean of signal
  float sum = 0;
  for (float value : q_signal) {
    sum += value;
  }
  float meanSignal = sum / (float)signalSize;

  // count number of times signal passes the mean in a positive direction
  int numTimes = 0;
  int i = 0;
  float previousValue;
  for (float value : q_signal) {
```

```
    if (i != 0 && previousValue <= meanSignal && value > meanSignal) {
      numTimes++;
    }
    i++;
    previousValue = value;
  }

  Serial.print(meanSignal, 6);
  Serial.print(",");
  Serial.print(numTimes);
  Serial.print(",");
  Serial.print(signalDuration);
  Serial.print(",");
  if (numTimes == 0 || signalDuration == 0) {
    return 0;
  }
  return 1000 * 60.0 * ((float)numTimes / signalDuration);
}



// digit display stuff

void clear_frame() {
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 12; col++) {
      frame[row][col] = 0;
    }
  }
}

void display_frame() {
  matrix.renderBitmap(frame, 8, 12);
}


void add_to_frame(char c, int pos) {
  int index = -1;
  if (c >= '0' && c <= '9')
    index = c - '0';
  else if (c >= 'A' && c <= 'Z')
    index = c - 'A' + 10;
  else {
    Serial.println("WARNING: unsupported character");
    return;
  }
```

```
  for (int row = 0; row < 8; row++) {
    uint32_t temp = fonts[index][row] << (7 − pos);
    for (int col = 0; col < 12; col++) {
      frame[row][col] |= (temp >> (11 − col)) & 1;
    }
  }
}
```

A.3 Python code

A.3.1 Amplifier and thermistor calibration

```
# Amplifier calibration
# schema: each line is: actual vin mean, actual vin deviation, amplifier vin mean, amplifier vin deviation,
    ↪ amplifier vout mean, amplifier vout deviation
raw_data = """
<paste amplifier calibration data here>
"""

import numpy as np
from scipy.odr import Model, RealData, ODR
import matplotlib.pyplot as plt

# Parsing raw data
lines = raw_data.strip().split('\n')
actual_vin_mean = []
actual_vin_deviation = []
amp_vin_mean = []
amp_vin_deviation = []
amp_vout_mean = []
amp_vout_deviation = []

for line in lines:
    values = line.split(',')
    actual_vin_mean.append(float(values[0]))
    actual_vin_deviation.append(float(values[1]))
    amp_vin_mean.append(float(values[2]))
    amp_vin_deviation.append(float(values[3]))
    amp_vout_mean.append(float(values[4]))
    amp_vout_deviation.append(float(values[5]))

offset = 1e−8 # A small offset to avoid zero deviations
actual_vin_deviation = [max(dev, offset) for dev in actual_vin_deviation]
amp_vin_deviation = [max(dev, offset) for dev in amp_vin_deviation]
amp_vout_deviation = [max(dev, offset) for dev in amp_vout_deviation]
```

```
# Define linear model
def linear_model(B, x):
    return B[0] * x + B[1]
    # return B[0] * x

# Function to plot graph with error bars and fitted line
def plot_graph(x_data, y_data, x_error, y_error, x_label, y_label, title, threshold_max, plot_ref =
    ↪ False):
    filtered_x = []
    filtered_y = []
    filtered_x_err = []
    filtered_y_err = []

    for i, x in enumerate(x_data):
        if x <= threshold_max:
            filtered_x.append(x)
            filtered_y.append(y_data[i])
            filtered_x_err.append(x_error[i])
            filtered_y_err.append(y_error[i])

    linear = Model(linear_model)
    data = RealData(filtered_x, filtered_y, sx=filtered_x_err, sy=filtered_y_err)
    odr = ODR(data, linear, beta0=[1.0, 0])
    output = odr.run()

    # Fitted line
    fitted_slope = output.beta[0]
    fitted_intercept = output.beta[1]
    fitted_slope_variance = output.cov_beta[0, 0]
    fitted_intercept_variance = output.cov_beta[1, 1]

    plt.figure(figsize=(8, 6))

    line_label = f'Fitted␣line:\n␣␣$y␣=␣({fitted_slope:.3f}␣\pm␣{np.sqrt(fitted_slope_variance):.3
        ↪ f})x␣+␣({fitted_intercept:.3f}\pm␣{np.sqrt(fitted_intercept_variance):.3f})$'

    plt.errorbar(x_data, y_data, xerr=x_error, yerr=y_error, fmt='o', label='Data␣with␣error␣bars'
        ↪ )
    plt.plot(filtered_x, fitted_slope * np.array(filtered_x) + fitted_intercept, color='red', label=
        ↪ line_label)
    plt.axvline(x=threshold_max, color='gray', linestyle='--', label=f'Threshold␣($x␣\leq␣{
        ↪ threshold_max:.3f}$)')

    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)
```

```
    plt.grid(True)

    if plot_ref:
      plt.plot(filtered_x, filtered_x, linestyle = '--', label = "Ideal line with no bias currents (
          ↪ $y = x$)")

    plt.legend()
    plt.show()

    print(f"Fitted Line Equation: y = {fitted_slope:.3f}x + {fitted_intercept:.3f}")
    print(f"Gradient - Mean: {fitted_slope}, Variance: {fitted_slope_variance}")
    print(f"Y-Intercept - Mean: {fitted_intercept}, Variance: {fitted_intercept_variance}")
    print(f"Parameterr covariance: {output.cov_beta[0,1]} {output.cov_beta[1,0]}")

print(actual_vin_mean, amp_vin_mean)

low_limit = .26
up_limit = 1.6

# Plot graph: Actual Vin vs Amplifier Vin
plot_graph(actual_vin_mean, amp_vin_mean, actual_vin_deviation, amp_vin_deviation,
           '$v_{\mathrm{in}}^*$ (V)', '$v_{\mathrm{in}}$ (V)', 'Actual Input Voltage vs
               ↪ Input Voltage with Amplifier Connected', 1.7592859268, True)

# Plot graph: Actual Vin vs Amplifier Vout
plot_graph(actual_vin_mean, amp_vout_mean, actual_vin_deviation, amp_vout_deviation,
           '$v_{\mathrm{in}}^*$ (V)', '$v_{\mathrm{out}}$ (V)', 'Actual Input Voltage vs
               ↪ Amplifier Output Voltage', 1.7592859268)


# Temperature calibration

data_temp_cal = """
<paste temperature sensor calibration data here>
"""

timestamps = []
input_adc_mean = []
input_adc_deviation = []
output_adc_mean = []
output_adc_deviation = []
real_temperature = []
real_temperature_deviation = []

# Splitting the data by line
lines = data_temp_cal.split('\n')
```

```
for line in lines:
    if line:
        parts = line.split(':')
        timestamps.append(int(parts[0]))
        values = list(map(float, parts[1].split(',')))
        input_adc_mean.append(float(values[0]))
        input_adc_deviation.append(float(values[1]))
        output_adc_mean.append(float(values[2]))
        output_adc_deviation.append(float(values[3]))
        real_temperature.append(float(values[16]))
        real_temperature_deviation.append(0.25)


# Normalise all the timestamps
timestamps = timestamps - np.min(timestamps)

# Sort timestamps and reorder other arrays accordingly
sorted_data = sorted(zip(timestamps, input_adc_mean, input_adc_deviation, output_adc_mean,
    ↪ output_adc_deviation, real_temperature, real_temperature_deviation))

# Unpack the sorted data into separate arrays
timestamps_sorted, input_adc_mean_sorted, input_adc_deviation_sorted,
    ↪ output_adc_mean_sorted, output_adc_deviation_sorted, real_temperature_sorted,
    ↪ real_temperature_deviation_sorted = zip(*sorted_data)

# Assign the sorted values back to the original arrays
timestamps = list(timestamps_sorted)
input_adc_mean = list(input_adc_mean_sorted)
input_adc_deviation = list(input_adc_deviation_sorted)
output_adc_mean = list(output_adc_mean_sorted)
output_adc_deviation = list(output_adc_deviation_sorted)
real_temperature = list(real_temperature_sorted)
real_temperature_deviation = list(real_temperature_deviation_sorted)


# Applying our model

ampInputOffsetMean = -0.126
ampInputGainMean = 1.252

in_voltages = 4.7 * np.array(input_adc_mean) / 1024.0
in_voltages_variance = pow(4.7 / 1024.0, 2) * np.power(input_adc_deviation, 2)
actual_in_voltages = (in_voltages - ampInputOffsetMean) / ampInputGainMean
actual_in_voltages_variance = (1.0 / np.power(ampInputGainMean, 2)) * in_voltages_variance
actual_in_temperatures = (actual_in_voltages - 0.4) / 0.0195
actual_in_temperatures_variance = (1.0 / pow(0.0195, 2)) * actual_in_voltages_variance
```

```
ampOutputOffsetMean = −0.271
ampOutputGainMean = 2.768

out_voltages = 4.7 * np.array(output_adc_mean) / 1024.0
out_voltages_variance = pow(4.7 / 1024.0, 2) * np.power(output_adc_deviation, 2)
actual_out_voltages = (out_voltages − ampOutputOffsetMean) / ampOutputGainMean
actual_out_voltages_variance = (1.0 / np.power(ampOutputGainMean, 2)) *
    ↪ out_voltages_variance
actual_out_temperatures = (actual_out_voltages − 0.4) / 0.0195
actual_out_temperatures_variance = (1.0 / pow(0.0195, 2)) * actual_out_voltages_variance

import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
import seaborn as sns
from scipy import stats

def plot(limit):
  real_temperature_np = np.array(real_temperature)
  real_temperature_deviation_np = np.array(real_temperature_deviation) + offset
  actual_in_temperatures_variance_np = np.array(actual_in_temperatures_variance) + offset
  actual_out_temperatures_variance_np = np.array(actual_out_temperatures_variance) + offset
  actual_in_temperatures_np = np.array(actual_in_temperatures)
  actual_out_temperatures_np = np.array(actual_out_temperatures)
  timestamps_np = np.array(timestamps)

  indices_within_range = [i for i in range(len(real_temperature))] if (not limit) else [i for i, temp
      ↪ in enumerate(real_temperature) if 32 <= temp <= 40]
  indices_within_range_calibration_curve = [i for i in indices_within_range if
      ↪ real_temperature_np[i] <= 74]

  indices_in_lowest_error_range = [i for i, temp in enumerate(real_temperature) if 34.5 <= temp
      ↪ <= 38.5]
  errors_in = np.abs(actual_in_temperatures_np[indices_in_lowest_error_range] −
      ↪ real_temperature_np[indices_in_lowest_error_range])
  errors_out = np.abs(actual_out_temperatures_np[indices_in_lowest_error_range] −
      ↪ real_temperature_np[indices_in_lowest_error_range])
  print(f'Mean␣error␣in␣{np.mean(errors_in):.3f}␣se␣{stats.sem(errors_in):.3f}')
  print(f'Mean␣error␣out␣{np.mean(errors_out):.3f}␣se␣{stats.sem(errors_out):.3f}')

  plt.figure()
  plt.title('Temperature␣Deviations␣With/Without␣Amplifier' + ('\nin␣Human␣Body␣
      ↪ Temperature␣Range' if limit else ''))

  # Plot histograms
  in_temperatures = np.sqrt(actual_in_temperatures_variance_np[indices_within_range])
```

```
out_temperatures = np.sqrt(actual_out_temperatures_variance_np[indices_within_range])

in_temperatures_not_saturated = np.sqrt(actual_in_temperatures_variance_np[
    ↪ indices_within_range_calibration_curve])
out_temperatures_not_saturated = np.sqrt(actual_out_temperatures_variance_np[
    ↪ indices_within_range_calibration_curve])

# Calculate histograms
n_bins = 30
plt.hist(in_temperatures_not_saturated, bins=n_bins, alpha=0.5, label='$\mathrm{SD}(T_{\
    ↪ mathrm{sensor}})$', density=False, color='orange')
plt.hist(out_temperatures_not_saturated, bins=n_bins, alpha=0.5, label='$\mathrm{SD}(T_{\
    ↪ mathrm{amp}})$', density=False, color='blue')

# Fit Gaussian distributions to the histograms
mu_in, sigma_in = np.mean(in_temperatures_not_saturated), np.sqrt(np.var(
    ↪ in_temperatures_not_saturated))
mu_out, sigma_out = np.mean(out_temperatures_not_saturated), np.sqrt(np.var(
    ↪ out_temperatures_not_saturated))

x = np.linspace(min(min(in_temperatures_not_saturated), min(out_temperatures_not_saturated
    ↪ )),
                max(max(in_temperatures_not_saturated), max(
                    ↪ out_temperatures_not_saturated)), 100)

fit_line_in = norm.pdf(x, mu_in, sigma_in)
fit_line_out = norm.pdf(x, mu_out, sigma_out)

fit_line_in_desc = f'$\mu␣=␣{mu_in:.3f},␣\sigma␣=␣{sigma_in:.3f}$'
fit_line_out_desc = f'$\mu␣=␣{mu_out:.3f},␣\sigma␣=␣{sigma_out:.3f}$'

label_line_in = 'Fitted␣Gaussian␣to␣$\mathrm{SD}(T_{\mathrm{sensor}})$\n␣␣' +
    ↪ fit_line_in_desc
label_line_out = 'Fitted␣Gaussian␣to␣$\mathrm{SD}(T_{\mathrm{amp}})$\n␣␣' +
    ↪ fit_line_out_desc

plt.plot(x, fit_line_in, label=label_line_in, color='orange')
plt.plot(x, fit_line_out, label=label_line_out, color='blue')

plt.xlabel('Temperature␣Deviation')
plt.ylabel('Frequency')
plt.legend(loc='best')

# Assuming ratio is defined as in your code snippet
ratio = np.sqrt(actual_in_temperatures_variance_np[indices_within_range]) / np.sqrt(
    ↪ actual_out_temperatures_variance_np[indices_within_range])
```

```
plt.figure()
plt.title('Temp␣Deviation␣from␣Sensor␣/␣Temp␣Deviation␣from␣Amplifier')
plt.hist(ratio, label="Ratio␣of␣Deviation␣of␣Temperature␣from␣Sensor␣vs␣Amplifier", bins
    ↪ =30, density=True)

# Fit a Gaussian distribution to the data
mu = np.mean(ratio)
sigma = np.sqrt(np.var(ratio))
x = np.linspace(mu − 3 ∗ sigma, mu + 3 ∗ sigma, 100)
fit_line = norm.pdf(x, mu, sigma)
plt.plot(x, fit_line, label='Fitted␣Gaussian', color='red')

plt.axhline(y=1, color='r', linestyle='−−', label='Ref')

plt.axvline(mu, label=f'Mean␣=␣{mu}', linestyle='−−')

plt.xlabel('Ratio')
plt.ylabel('Frequency')
plt.legend()

mean_dev_temp_from_sensor = np.mean(np.sqrt(actual_in_temperatures_variance_np[
    ↪ indices_within_range]))
se_temp_from_sensor = np.std(np.sqrt(actual_in_temperatures_variance_np[
    ↪ indices_within_range])) / len(actual_in_temperatures_variance_np[
    ↪ indices_within_range])

mean_dev_temp_from_amplifier = np.mean(np.sqrt(actual_out_temperatures_variance_np[
    ↪ indices_within_range]))
se_temp_from_amplifier = np.std(np.sqrt(actual_out_temperatures_variance_np[
    ↪ indices_within_range])) / len(actual_out_temperatures_variance_np[
    ↪ indices_within_range])

print(f"error␣temperature␣from␣sensor:␣{mean_dev_temp_from_sensor}␣+/−␣{
    ↪ se_temp_from_sensor}\nerror␣temperature␣from␣amplifier:␣{
    ↪ mean_dev_temp_from_amplifier}␣+/−␣{se_temp_from_amplifier}")
print(f"ratio␣of␣deviation␣of␣temp.␣from␣sensor␣vs␣amplifier:␣{np.mean(ratio)}␣+/−␣{np.
    ↪ std(ratio)␣/␣len(ratio)}")

plt.figure()
plt.title('abs␣delta␣temperature␣to␣real␣temperature␣with␣2␣methods')
plt.plot(real_temperature_np[indices_within_range], np.abs(actual_in_temperatures_np[
    ↪ indices_within_range] − real_temperature_np[indices_within_range]), marker = ".",
    ↪ label = "delta␣sensor␣temperature␣to␣real")
plt.plot(real_temperature_np[indices_within_range], np.abs(actual_out_temperatures_np[
    ↪ indices_within_range] − real_temperature_np[indices_within_range]), marker = ".",
    ↪ label = "delta␣amplified␣temperature␣to␣real")
```

```
plt.figure()
plt.title('abs␣delta␣temperature␣between␣two␣methods')
plt.plot(real_temperature_np[indices_within_range], np.abs(actual_in_temperatures_np[
    ↪ indices_within_range] − actual_out_temperatures_np[indices_within_range]), marker =
    ↪ ".", label = "delta␣sensor␣temperature␣to␣amplified␣temperature")

fig = plt.figure(figsize=(7,6))
ax = fig.add_subplot(1, 1, 1)
plt.errorbar(timestamps_np[indices_within_range], real_temperature_np[indices_within_range],
    ↪  yerr=real_temperature_deviation_np[indices_within_range], label='$T^*$', marker='.')
plt.errorbar(timestamps_np[indices_within_range], actual_in_temperatures_np[
    ↪ indices_within_range], yerr=np.sqrt(actual_in_temperatures_variance_np[
    ↪ indices_within_range]), label='$T_{\mathrm{sensor}}$', marker='.')
plt.errorbar(timestamps_np[indices_within_range], actual_out_temperatures_np[
    ↪ indices_within_range], yerr=np.sqrt(actual_out_temperatures_variance_np[
    ↪ indices_within_range]), label='$T_{\mathrm{amp}}$', marker='.')

# Adding horizontal dashed lines at temperatures 30 and 40 degrees
if not limit:
  plt.axhline(y=32, color='blue', linestyle='−−', label='$32^\circ␣C$')
  plt.axhline(y=40, color='green', linestyle='−−', label='$40^\circ␣C$')
  plt.axhline(y=74, color='red', linestyle='−−', label='Saturation␣($T^*␣\geq␣74^\circ␣C$)')

plt.xlabel('Time␣($ms$)')
plt.ylabel('Temperature␣($^\circ␣C$)')
plt.title('Cooling␣Curve' + ('␣in␣Human␣Body␣Temperature␣Range' if limit else ''))

plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()

# CALIBRATION CURVES

# Model function for a straight line (y = mx + c)
def linear_func(B, x):
    return B[0] * x + B[1]

# Define data for the ODR fitting
data_in = RealData(real_temperature_np[indices_within_range_calibration_curve],
                   actual_in_temperatures_np[indices_within_range_calibration_curve],
                   sx=real_temperature_deviation_np[indices_within_range_calibration_curve],
                   sy=np.sqrt(actual_in_temperatures_variance_np[
                       ↪ indices_within_range_calibration_curve]))

data_out = RealData(real_temperature_np[indices_within_range_calibration_curve],
                    actual_out_temperatures_np[indices_within_range_calibration_curve],
```

```
                    sx=real_temperature_deviation_np[indices_within_range_calibration_curve
                        ↪ ],
                    sy=np.sqrt(actual_out_temperatures_variance_np[
                        ↪ indices_within_range_calibration_curve]))

# Perform ODR fitting
model = Model(linear_func)
odr_in = ODR(data_in, model, beta0=[1.0, 0.0])
odr_out = ODR(data_out, model, beta0=[1.0, 0.0])
fit_in = odr_in.run()
fit_out = odr_out.run()

# Extract parameters (gradient and y-intercept) and their uncertainties
gradient_in, intercept_in = fit_in.beta
gradient_out, intercept_out = fit_out.beta

gradient_err_in, intercept_err_in = fit_in.sd_beta
gradient_err_out, intercept_err_out = fit_out.sd_beta

# Plotting the calibration curves with original points and fitted lines
plt.figure(figsize=(7,7))
plt.title('Calibration␣Curve' + ('␣in␣Human␣Body␣Temperature␣Range' if limit else ''))
plt.errorbar(real_temperature_np[indices_within_range], actual_in_temperatures_np[
    ↪ indices_within_range],
            yerr=np.sqrt(actual_in_temperatures_variance_np[indices_within_range]),
            xerr=real_temperature_deviation_np[indices_within_range], fmt='.',
            label='$T_{\mathrm{sensor}}$')

plt.errorbar(real_temperature_np[indices_within_range], actual_out_temperatures_np[
    ↪ indices_within_range],
            yerr=np.sqrt(actual_out_temperatures_variance_np[indices_within_range]),
            xerr=real_temperature_deviation_np[indices_within_range], fmt='.',
            label='$T_{\mathrm{amp}}$')



plt.plot(real_temperature_np[indices_within_range_calibration_curve], linear_func([gradient_in,
    ↪  intercept_in], real_temperature_np[indices_within_range_calibration_curve]),
        label=('Fitted␣$T_{\mathrm{sensor}}$:␣\n␣␣' + f'$y␣=␣({gradient_in:.3f}␣\pm␣{
            ↪ gradient_err_in:.3f})x␣+␣({intercept_in:.3f}␣\pm␣{intercept_err_in:.3f})$'))

plt.plot(real_temperature_np[indices_within_range_calibration_curve], linear_func([
    ↪ gradient_out, intercept_out], real_temperature_np[
    ↪ indices_within_range_calibration_curve]),
        label=('Fitted␣$T_{\mathrm{amp}}$:␣\n␣␣' + f'$y␣=␣({gradient_out:.3f}␣\pm␣{
            ↪ gradient_err_out:.3f})x␣+␣({intercept_out:.3f}␣\pm␣{intercept_err_out:.3f})$'))
```

```
  plt.plot(real_temperature_np[indices_within_range], real_temperature_np[indices_within_range
      ↪ ], '--', label='Ideal fit ($y = x$)') # Reference line

  plt.xlabel('$T^*$ ($^\circ C$)')
  plt.ylabel('Measured Temperature ($^\circ C$)')
  plt.legend()
  plt.grid(True)

  if not limit:
    plt.axvline(x=74, color='r', linestyle='--', label='Saturation ($T^* \geq 74^\circ C$)')

  plt.legend()
  plt.show()

  # Display mean and variance of gradients and y-intercepts
  print(f"Sensor Temp Fit: Gradient = {gradient_in:.4f} +- {gradient_err_in:.4f}, Intercept =
      ↪ {intercept_in:.4f} +- {intercept_err_in:.4f}")
  print(f"Amp Temp Fit: Gradient = {gradient_out:.4f} +- {gradient_err_out:.4f}, Intercept =
      ↪  {intercept_out:.4f} +- {intercept_err_out:.4f}")

plot(False)
plot(True)
```

A.3.2 Offline pulse detection

```
import matplotlib.pyplot as plt
import numpy as np
from google.colab import drive

drive.mount('/content/drive')

file_path = '<enter path>'

def read_pcg_data(file_path):
    timestamps = []
    hz = []
    channel_1 = []
    channel_2 = []
    channel_3 = []

    with open(file_path, 'r') as file:
        for line in file:
            parts = line.strip().split(':')
            timestamps.append(int(parts[0]))
            channels = parts[1].split(',')
```

```
            hz.append(float(channels[0]))
            channel_1.append(float(channels[1]))
            channel_2.append(float(channels[2]))
            channel_3.append(float(channels[3]))

    signal_frequency = np.mean(hz)
    print(signal_frequency)
    return signal_frequency, timestamps, channel_1, channel_2, channel_3

signal_frequency, timestamps, channel_1, channel_2, channel_3 = read_pcg_data(file_path)

def plot_selected_intervals(timestamps, channel_1, channel_2, channel_3, interval_duration,
    ↪ selected_intervals, selected_channels):
    if interval_duration == 0: # If no interval duration is given, plot the entire signal
        start_index = 0
        end_index = len(timestamps)
        selected_intervals = [0] # Default interval for entire signal
    else:
        time_range = interval_duration
        total_duration = timestamps[-1] - timestamps[0]
        num_intervals = total_duration // time_range

    if not selected_intervals: # If no intervals specified, take all intervals
        selected_intervals = range(num_intervals) if interval_duration != 0 else [0]

    if not selected_channels: # If no channels specified, take all channels
        selected_channels = [1, 2, 3]

    for i in selected_intervals:
        if interval_duration != 0:
            start_index = i * (time_range // (timestamps[1] - timestamps[0]))
            end_index = start_index + (time_range // (timestamps[1] - timestamps[0]))

        plt.figure(figsize=(10, 4))

        channel_str = ['$V_{\mathrm{total}}$', '$V_{\mathrm{DC}}$', '$V_{\mathrm{AC}}$']

        for channel in selected_channels:
            channel_data = None

            if channel == 1:
                channel_data = channel_1[start_index:end_index]
            elif channel == 2:
                channel_data = channel_2[start_index:end_index]
            elif channel == 3:
                channel_data = channel_3[start_index:end_index]
```

```
            plt.plot(timestamps[start_index:end_index], channel_data, label=channel_str[channel
                ↪ - 1])

        plt.xlabel('Timestamp (ms)')
        plt.ylabel('Voltage (V)')
        if interval_duration != 0:
            plt.title(f'ECG Signal - Interval {i + 1}')
        else:
            plt.title(f'PCG Entire Signal')
        plt.legend()
        plt.show()

# View entire signal
plot_selected_intervals(timestamps, channel_1, channel_2, channel_3, 0, [], [])

# Interactive plot

import plotly.graph_objs as go
import plotly.io as pio

def plot_interactive(timestamps, channel_1, channel_2, channel_3):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x=timestamps, y=channel_1, mode='lines', name='$V_{\mathrm{total
        ↪ }}$'))
    fig.add_trace(go.Scatter(x=timestamps, y=channel_2, mode='lines', name='$V_{\mathrm{DC
        ↪ }}$'))
    fig.add_trace(go.Scatter(x=timestamps, y=channel_3, mode='lines', name='$V_{\mathrm{AC
        ↪ }}$'))

    fig.update_layout(
        title='PCG Signal',
        xaxis=dict(title='Timestamp (ms)'),
        yaxis=dict(title='Voltage (V)'),
        width=1100,
        height=500
    )

    pio.show(fig)

# Display an interactive plot for all channels
plot_interactive(timestamps, channel_1, channel_2, channel_3)

import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import find_peaks
from scipy import fftpack
```

```
def estimate_bpm_from_peaks(timestamps, signal, peak_indices):
    peak_times = [timestamps[i] for i in peak_indices]
    differences = [peak_times[i + 1] − peak_times[i] for i in range(len(peak_times) − 1)]

    mean_difference = np.mean(differences)
    std_difference = np.std(differences)

    mean_bpm = 60 ∗ 1000 ∗ (1 / mean_difference)
    print(f"std␣{std_difference}␣len␣{len(differences)}␣error_difference␣{(std_difference␣/␣np.
        ↪ sqrt(len(differences)))}")
    error_difference = std_difference / np.sqrt(len(differences))

    print(f"Mean␣peak␣difference␣=␣{mean_difference}+−{error_difference}ms␣=>␣mean␣BPM␣
        ↪ from␣peaks␣=␣{mean_bpm}␣beats␣per␣minute.")

    return mean_bpm

def find_nearest_timestamp(timestamps, target_time):
    nearest_time = min(timestamps, key=lambda x: abs(x − target_time))
    return nearest_time

def apply_methods_to_channel_3(timestamps, channel_3, start_time, end_time, sampling_rate):
    # Find the nearest timestamps
    start_nearest = find_nearest_timestamp(timestamps, start_time)
    end_nearest = find_nearest_timestamp(timestamps, end_time)

    # Find indices corresponding to the nearest timestamps
    start_index = timestamps.index(start_nearest)
    end_index = timestamps.index(end_nearest)

    # Select signal within the specified time interval
    signal = channel_3[start_index:end_index]
    selected_timestamps = timestamps[start_index:end_index] # Corresponding timestamps

    time_step = 1 / sampling_rate

    # Peak Detection and Signal Plot
    # Annotate peaks on the signal plot
    def plot_peaks(signal, distance, height, timestamps):
      # Peak Detection
      plot_time = np.arange(len(signal)) # Using indices for x−axis
      plt.figure(figsize=(7, 5)) # Adjust figure size
      plt.plot(timestamps, signal, label='$V_{\mathrm{AC}}$') # Plot the signal over time

      peak_indices, _ = find_peaks(signal, distance=distance, height=height)
```

```
    plt.plot(np.array(timestamps)[peak_indices], np.array(signal)[peak_indices], 'ro', label=f'
        ↪ Peaks (distance={distance:.2f})', alpha=.5)

    plt.title('Peak Detection')
    plt.xlabel('Time (ms)') # Set x-axis label to Time in milliseconds
    plt.ylabel('Voltage (V)')
    plt.legend()
    plt.show()

    return estimate_bpm_from_peaks(timestamps, signal, peak_indices)

# distance for peak detection is set based on max heartrate of 250 BPM. such a heartrate has a number
    ↪ of seconds between heartbeats of 60 / 250 = 0.24 seconds. we have a timestep of time_step,
    ↪ thus take the number of multiples to that time_step: 0.24 / time_step.
mean_bpm = plot_peaks(signal, 0.24 / time_step, None, selected_timestamps)

# FFT
def plot_fft_and_compute_max_freq(signal, sampling_rate):
    n = len(signal)
    T = 1.0 / sampling_rate
    freqs = np.fft.fftfreq(n, T)
    fft_vals = np.fft.fft(signal)
    fft_range = fft_vals[:n//2]
    positive_freqs = freqs[:n//2]

    # Define frequency range in terms of bins
    start_freq_index = int(n * 0.0) # Corresponding index for 0 Hz
    end_freq_index = int(n * 4.2 / sampling_rate) # Corresponding index for 4.2 Hz

    # Take FFT within the specified frequency range
    specific_fft_range = fft_range[start_freq_index:end_freq_index]
    specific_freqs = positive_freqs[start_freq_index:end_freq_index]

    plt.figure(figsize=(7, 5))
    plt.plot(specific_freqs, np.abs(specific_fft_range))
    plt.title('FFT between 0.5 Hz and 4.2 Hz')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Amplitude')

    # Find the maximum amplitude frequency within the range of 0.5 Hz to 4.2 Hz
    start_index_05 = int(n * 0.5 / (sampling_rate)) # Corresponding index for 0.5 Hz
    end_index_42 = int(n * 4.2 / (sampling_rate)) # Corresponding index for 4.2 Hz
    specific_fft_range_05_42 = fft_range[start_index_05:end_index_42]
    specific_freqs_05_42 = positive_freqs[start_index_05:end_index_42]

    max_amp_index = np.argmax(np.abs(specific_fft_range_05_42))
    max_amp_freq = specific_freqs_05_42[max_amp_index]
```

```
        max_amp = np.abs(specific_fft_range_05_42[max_amp_index])

        plt.plot(max_amp_freq, max_amp, 'ro') # Plot the maximum amplitude frequency point
        plt.text(max_amp_freq, max_amp, f"Max Amp\n{max_amp_freq:.2f} Hz",
            ↪ verticalalignment='bottom', horizontalalignment='left', color='black')

        plt.xlim(0, 4.2) # Set x-axis limits to display frequencies between 0 Hz and 4.2 Hz
        plt.show()

        return max_amp_freq

    max_amp_freq = plot_fft_and_compute_max_freq(signal, sampling_rate)

    # Estimate BPM from FFT highest frequency
    best_bpm = 60 * max_amp_freq
    print(f"Highest amplitude FFT frequency = {max_amp_freq}Hz => BPM estimate from
        ↪ FFT = {best_bpm} beats per minute.")

    print(f"Peak detection BPM estimate = {mean_bpm}, FFT BPM estimate = {best_bpm}")


# Obtain specific region of signal where PCG has converged.

start_time = 274000
end_time = 294000

# Needs to be correct, important
sampling_rate = signal_frequency

start_nearest = find_nearest_timestamp(timestamps, start_time)
end_nearest = find_nearest_timestamp(timestamps, end_time)
start_index = timestamps.index(start_nearest)
end_index = timestamps.index(end_nearest)

signal = channel_3[start_index:end_index]

print("Applying methods to original signal:")
apply_methods_to_channel_3(timestamps, channel_3, start_time, end_time, sampling_rate)
```