Electronics A + D

Gian Gentinetta

October 22, 2019

Abstract

In this experiment we aim to construct an automated cooling mechanism using the Arduino Uno platform. To achieve this we use a Grove Temperature Sensor to measure the temperature in a first attempt and later replace it with a self built negative temperature coefficient (NTC) thermistor. The final goal is to automatically activate a fan to cool the system once a certain temperature threshold is reached.This experiment was carried out in the constraints of the Physics Laboratory 3 at the Eidgenössische Technische Hochschule Zürich.

Contents

1 Introduction

1.1 Arduino and Grove Shield

The main goal of this experiment is to learn to work with the Arduino Board and its environment. Arduino Boards are simple computers including microprocessors and controllers which are used to process digital and analog inputs and outputs. The board (in our case an Arduino Uno) is equipped with multiple digital and analog ports which are used to exchange information with the circuits we build in this experiment. To simplify the signal connections (input/output pins) and the voltage supply we connected the main board with an adaptor (*Grove Base Shield*) that has user friendly Grove Connectors. Those combine the four wires (voltage common collector (VCC), ground (GND) and the input/output connections) into one cable.

1.2 Development Environment

To control the Arduino we use the Arduino IDE. This is a desktop application that allows the development of programs (in Arduino Language which is closely related to C) which tell the Arduino what it is supposed to do. It also acts as the connection to the Arduino: Once a connection (via USB cable) is established one can compile the code and load the program onto the Arduino directly from the IDE.

Every Arduino program consists of two main parts:

```
void setup() {
}
```

and

```
void loop() {
}
```

The `setup()` function is called immediately after the program has been loaded. The `loop()` function is quite self-explanatory too: It is initially called after the setup and is then looped until the program is stopped.

1.3 Temperature Sensor

The secondary goal of the experiment is to build an automatic cooling system for a circuit that we will build (and heat up). To measure the temperature we use a resistor with a temperature sensitive resistance (thermistor). In this experiment we use negative temperature coefficient (NTC) thermistors. Their resistance is given as a function of the temperature:

$$R = R_0 e^{-B(\frac{1}{T_0} - \frac{1}{T})} \tag{1}$$

, where R_0 is the resistance at $T_0 = 25\ ^\circ\mathrm{C}$ and B is a constant characterising the thermistor. To actually measure the temperature we use a voltage divider so that we can measure an output voltage V_{out} and compare it to the input V_{in}. To be able to safely read the output voltage we furthermore decouple the measurement side and the read-out side of the circuit with an Operational Amplifier (Op-amp). See figure 1 for the complete circuit.

Now the measured voltage can be converted first back into resistance and than into temperature with the following equations.

$$R = \Big(\frac{1023}{V_{meas}} - 1\Big) R_0 \tag{2}$$

$$\frac{1}{T} = \frac{\ln \frac{R}{R_0}}{B} + \frac{1}{T_0} \tag{3}$$

, where V_{meas} is the measured voltage relative to V_{in} (V_{in} would be the maximal voltage of 1023, as the Arduino measures the voltage as a 10 bit value). This will provide us the temperature in Kelvin.

1.4 Grove Components

The Grove kit that provides the adaptor shield also comes with several additional components that can be connected to the Arduino. The ones used in the experiment are a temperature sensor, a RGB LCD display and a potentiometer. The temperature sensor is in principal the same as described in the previous subsection but it has the voltage divider and Op-amp already built in so it can easily be connected to the shield to directly measure the voltage V_{meas}.

The RGB LCD display is a colour display panel on which we can show information on 2 rows of 16 digits each. The display comes with its own library[1] that should be installed in the Arduino IDE.

Finally, the potentiometer can be attached to the system to control variables while the code is running, which will be practical in multiple ways during this experiment.

1.5 Fan and Pulse Width Modulation

To cool the system we will use a fan that can be controlled through a digital Arduino pin that supports Pulse Width Modulation (PWM). PWM is a technique that uses square wave signals to transmit power or information (in this case from the Arduino to the fan). Square waves are periodical functions that send a high signal (digital 1) for a fixed proportion of the period (called duty cycle) and then a low signal (digital 0) for the remaining period. In our case, the longer the duty cycle, the faster the fan's rotation (so the highest speed is achieved with a duty cycle of 100%).

To program the fan's duty cycle we use `analogWrite(power)`, where `power` is a value between 0 and 255. So 255 corresponds to a 100% duty cycle.

1.6 PID Controller

Our final goal is to cool the circuit until it reaches a fixed equilibrium temperature T_0. As we will see it is difficult to perfectly cool the system to a given temperature without

[1] `http://wiki.seeedstudio.com/Grove-LCD_RGB_Backlight/`

over- or undershooting it – that is why we will implement a proportional-integral-derivative (PID) controller. This controller analyses the error (in our case the temperature difference $T - T_0$) and provides a suggestion to correct this error (e.g. a correction in the PWM of the fan). In this analysis three components are involved that can be differently weighted by corresponding parameters:

- The proportional part: Here we directly look at the instant error and modify the output proportional to it. Weighted by the parameter k_p.
- The integral part: Here we look at how the error accumulated over several measurements and correct the output with respect to a longer period. Weighted with k_i.
- The derivative part: Finally we look at how the error changed compared to the last measurement. In our case this is useful to detect when the system changes from being heated to being cooled. Weighted with k_d

The challenge is to tune the three parameters so that the temperature converges quickly and has minimal oscillation.

2 Experiment

2.1 Blinking LED

As a first task the Arduino's LED were programmed such that one LED is always turned on while the second LED is blinking. The main goal here was to get a better understanding of the loop function and how delays are programmed. The author finally opted to use `micros()` to receive the current time and then compare that time to a set period. This makes it possible to have different periods for different measurements (see the part about the fan duty cycle), while `delay()` pauses the whole loop. The final program can be found in the directory */3.3-BlinkingLEDmicros*.

2.2 Grove Temperature Sensor, Display and Potentiometer

From this part on we connect the Arduino Board to the Grove Shield. The first component added to the shield is the Grove Temperature Sensor. Using equations 2 and 3 we convert the measured voltage V_{meas} into the temperature.

As a second component the Grove RGB LCD display is added and programmed to display the current temperature. Additionally a maximal threshold of $30°C$ is set at which the display is programmed to turn red. This is tested by applying pressure (and thus body heat) to the temperature sensor. The code can be found in the */GroveTemp* directory.

The final part of this task is to connect a potentiometer to the shield which is used to

manually change the temperature threshold. The code is found in the */Potentiometer* directory.

2.3 Op-Amp Circuit

The next goal is to replace the Grove temperature sensor with a self-built sensor. This is done using the circuit shown in figure 1, which includes a LM358 Op-Amp and a B57164-K104-J NTC. Again we measure the voltage V_{meas} and then calculate the temperature.

However, to achieve good results we first need to calibrate the sensor by tuning the B-value in equation 2. The calibration is done with the code in the */Calibration* directory, which provides a user friendly calibration method: The B-value can be changed using the potentiometer while the two temperatures (Grove Sensor and new sensor) are shown on the display. Once the two temperatures are within reasonable range the display's background changes to a green colour. The B-value can than be read on the Serial monitor.

To test this circuit the code in directory */Op-Amp* is used.

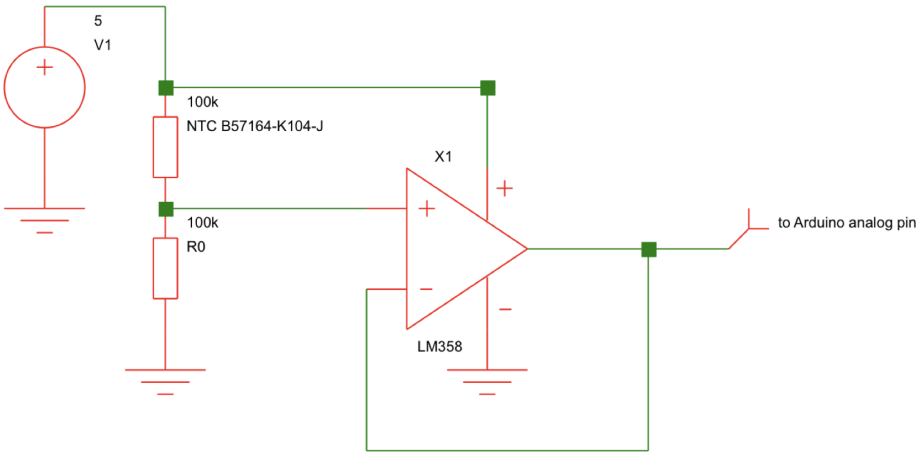

Figure 1: Circuit including the NTC and an Op-Amp used to measure the temperature.

2.4 Two-Point Controller

In this part we use the built sensor to measure the temperature of our circuit when heated (using resistors) and cooled (using a fan). To begin with two resistors ($100 \pm 1\Omega$ each) are added to the circuit so that they touch the thermistor (Figure 2). The power dissipation of the resistors directly heats the thermistor.

Now that we have a method to heat the system, we need to be able to cool it again. To achieve this a fan is added to the circuit (Figure 3) and connected to a digital (PWM

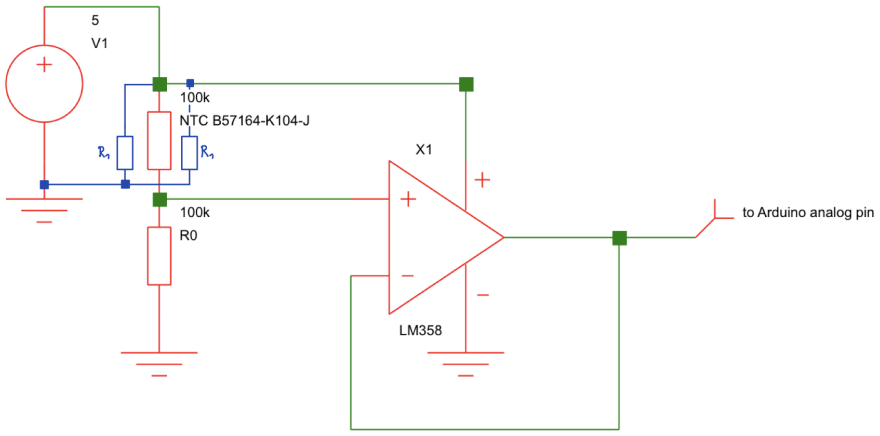

Figure 2: In this circuit the two resistors $R_1 = 100 \pm 1\Omega$ are added to heat the circuit.

enabled) Arduino pin. The code is adjusted so that there is an additional lower threshold which is set at 3° C below the threshold controlled by the potentiometer. The fan is programmed to blow at full speed if the temperature is above the higher threshold and once again turns of when the temperature is cooled down below the lower threshold.

The use of two thresholds enables the fan to actually cool the system for some degrees in contrast to a program with only one threshold where it would constantly turn on and off. The code is found in the */Two-Point* directory.

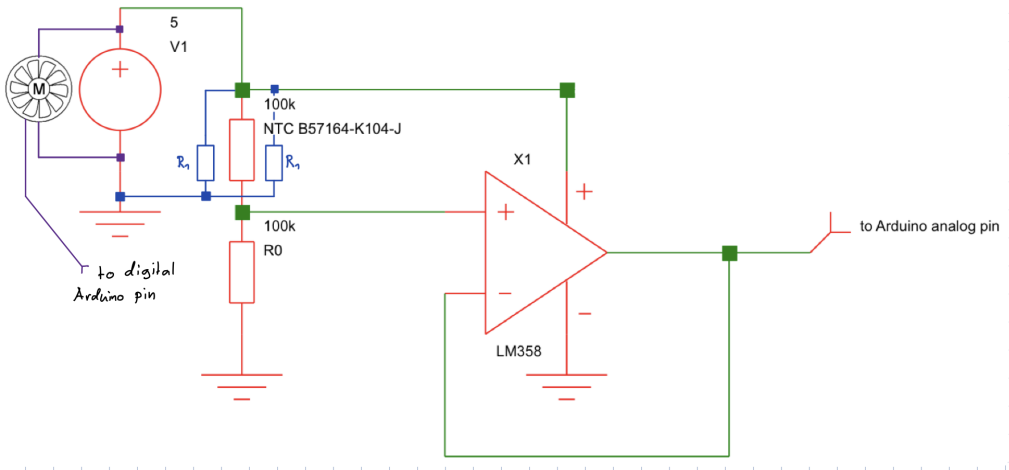

Figure 3: To cool the system a fan is added to the circuit, controllable through a digital Arduino pin.

2.5 Fan Speed

Before we continue with the automatic cooling system want to learn more about the characteristics of the used fan. The fan has a built in Hall Effect Sensor that measures its rotation speed and sends two pulses per period to the analogue Arduino pin (which we connect using a 10k pull-up resistor to VCC, see Figure 4). The pull-up resistor makes sure that the signal is either HIGH (5V) or LOW (Ground) and can thus be processed by the Arduino.

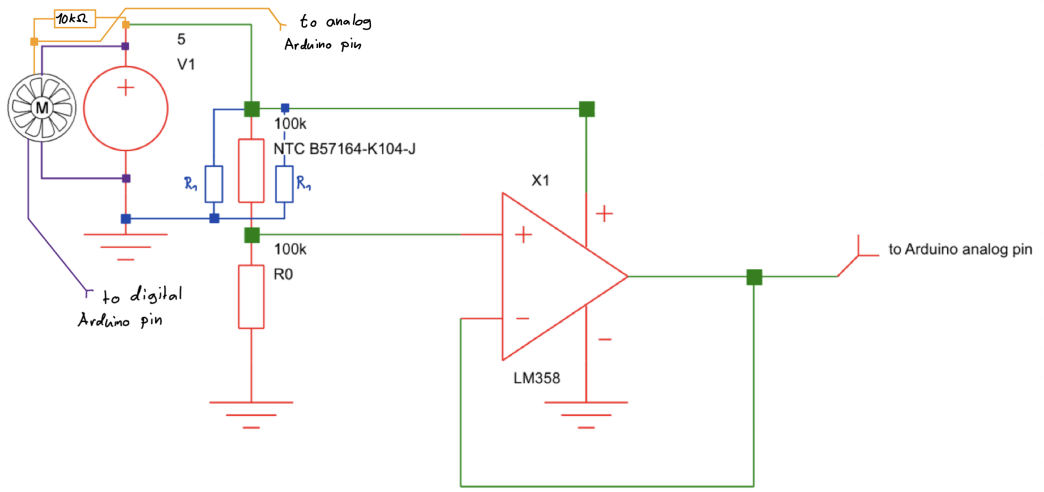

Figure 4: To measure the fan's rotation pulses a connection to a analogue Arduino pin is established with a pull-up resistor to VCC.

Using the fact that the pulse changes 4 times from LOW to HIGH or from HIGH to LOW we are able to calculate the rotations per minute (RPM) of the fan. Using the code in the */dutyCycle* directory we measure the rotations per minute for the different input powers (ranging from 0 to 255).

As we cannot measure continuously, a frequency has to be determined at which the pulse is analysed. Nyquist's theorem[2] states that the frequency can be determined correctly if there are at least two measurements in one period. The datasheet of the fan tells us that the maximal frequency is around 1900 RPM. As there are two pulses per rotation the pulse frequency is thus approximately 65 Hz which means we need to repeat our measurement at a rate of at least 130 Hz. To be safe and because the Arduino is fast enough we set the loop frequency at 10000 Hz.

Using the results from the above experiment we can now manually control the RPM of

[2] `https://en.wikipedia.org/wiki/Nyquist\T1\textendashShannon_sampling_theorem`

the fan. This is done in the code */equilibrium*, which we run at different RPMs for about one minute each until the temperature measured by the thermistor reaches roughly an equilibrium.

2.6 PID Controller

The final task of this experiment session is to implement a PID controller to maintain an (almost) constant temperature in our system. This is done in the code */PID3*, where we aim to achieve a temperature of $25° \ C$. The PID takes as input the difference between the measured temperature and the goal temperature and outputs the corresponding correction to the fan's duty cycle.

The important part in this experiment is to fine tune the PID parameters so that the temperature converges with minimal oscillation. To do this the code prints the three parameters to the Serial interface while the temperature and RPM are shown on the LCD display so that the individual parameters can be observed and later changed in the code.

3 Results

For experiments 2.1 and 2.2 there is not much to be noted other than that the code worked as expected. See Figure 5 for plot of the changing threshold.

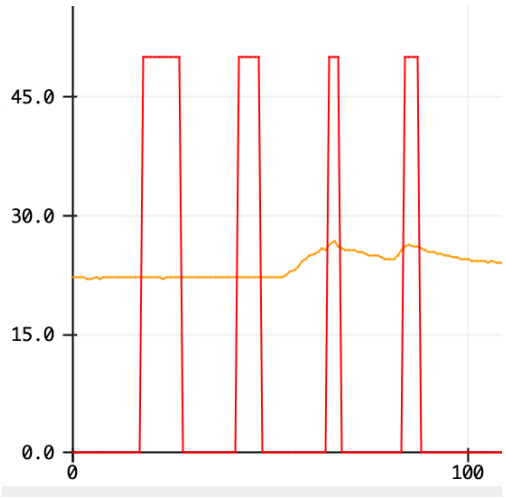

Figure 5: Screenshot of the Serial monitor for the potentiometer experiment. During the experiment the threshold was changed using the potentiometer. The orange line shows the measured temperature in °C, while the red line indicates whether the temperature is above or below the current threshold (50 if above, 0 if below). After around 50 seconds the sensor was heated (by applying pressure) while the threshold stayed constant.

In experiment 2.3 the new sensor was calibrated and a B-Value of 6700 ± 250 K was found.

The next part was to heat the sensor using resistors. The collected data is shown in figure 6. As a second step the fan was installed and programmed with the two-point controller. The higher threshold is now fixed at $T_{high} = 30°C$ and thus the lower threshold at $T_{high} = 27°C$. The data for this experiment is shown in figure 7.

Next, measured the fan's speed as a function of the duty cycle. The rotations were measured for 2 seconds at each step. The collected data is shown in figure 8. The fan started rotating for a duty cycle of 14 at 240 ± 30 RPM and reached a maximum of 1800 ± 30 RPM which is in the range of the 10% tolerance from 1900 RPM noted on the datasheet.

To measure the equilibrium temperatures we looked at the data in figure 8 to determine which points in the duty cycle are relevant to look at. We then let the temperature reach an equilibrium for several duty cycle points. The data is shown in figure 9.

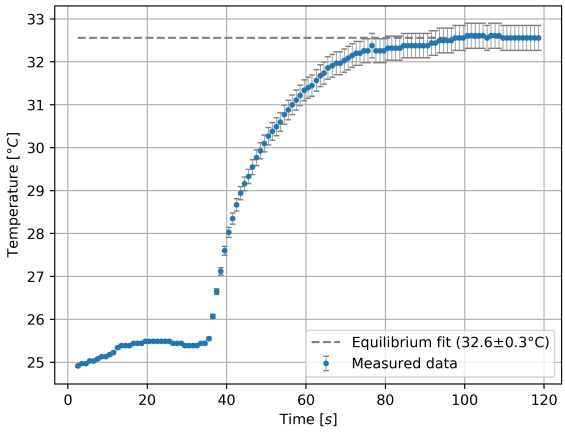

Figure 6: Plot showing how the temperature increases from room temperature to an equilibrium while the sensor is being heated by two 100 Ω resistors. The heating starts at $t = 36 \pm 2$ s.

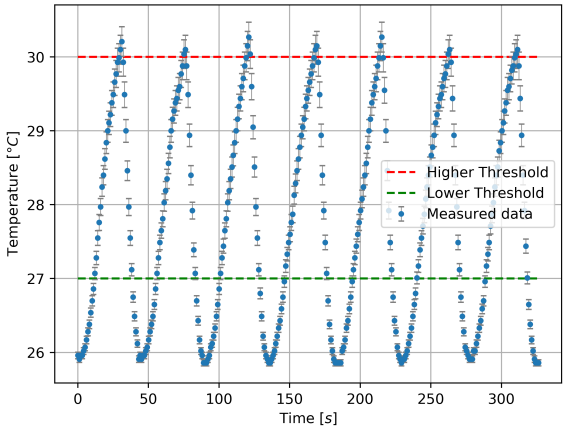

Figure 7: Plot showing how the temperature behaves under the two-point cooling method. The fan is turned on once the temperature reaches 30°C and turns off again after the sensor cooled down to 27°C.

The final experiment was all about finding nice PID parameters to have fast convergence with minimal oscillations. After tuning the parameters in the code we found that

- $k_p = 20$ (proportional)
- $k_i = 30$ (integral)
- $k_d = 10$ (derivative)

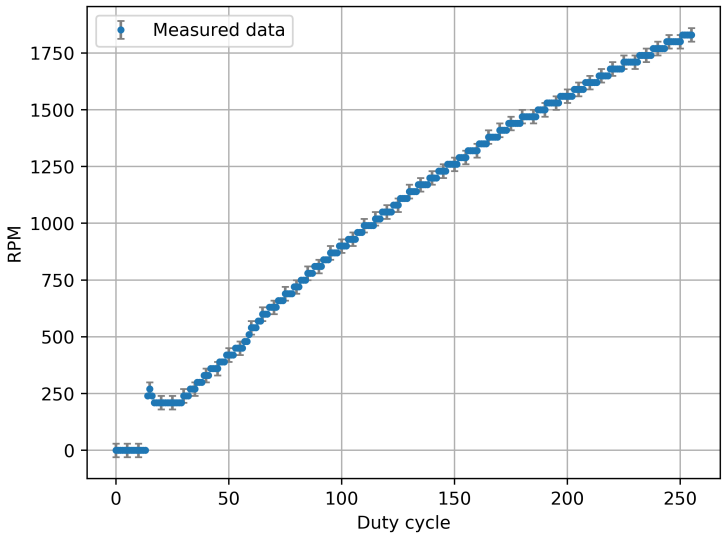

Figure 8: Plot showing how fast the speed turns (RPM) for different power inputs. The uncertainties are shown every 5 measurements to allow better readability.

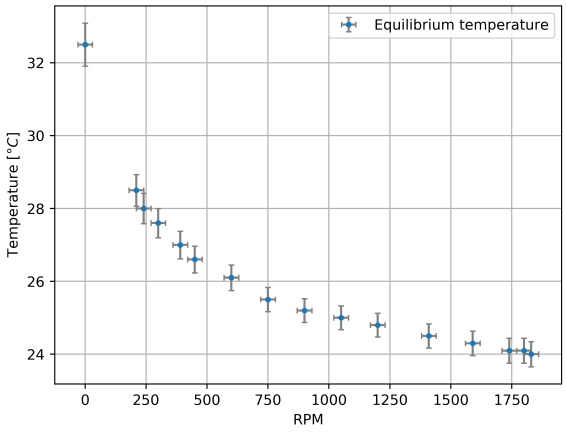

Figure 9: Plot showing the equilibrium temperatures of the system as a function of the fan's RPM.

provide satisfying results. The collected data is shown together with a set of data collected with *bad* parameters ($k_p = 100, k_i = 30, k_d = 50$) in figure 10. For both sets of parameters the goal was to reach an equilibrium temperature of 25 °C.

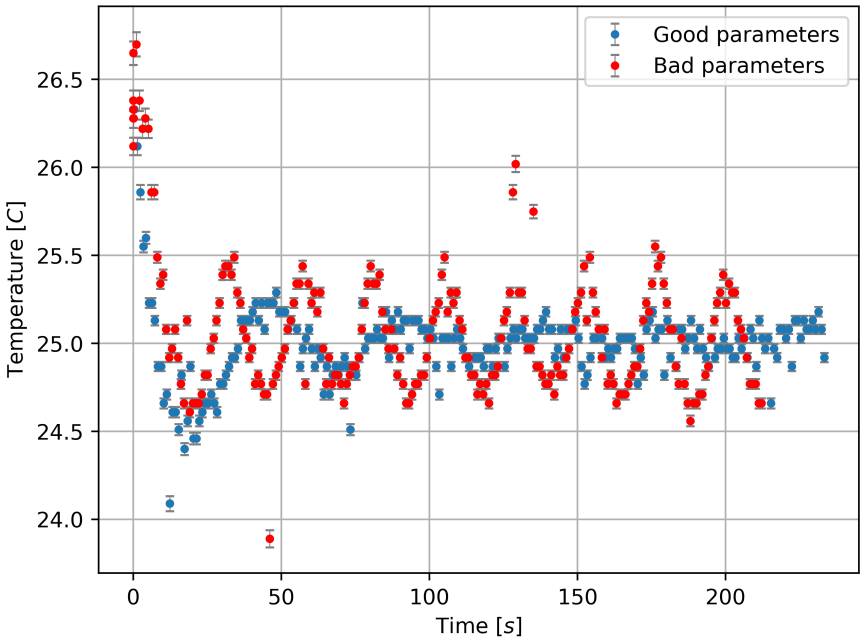

Figure 10: Plot showing how the temperature behaves when cooled with a PID controlled fan, once with *good* parameters ($k_p = 20, k_i = 30, k_d = 10$) and once with *bad* parameters ($k_p = 100, k_i = 30, k_d = 50$).

4 Data analysis

This experiment differs from most other experiments as most of the data analysis happens already during the measurement period. The Arduino (and the code written) already converts its input voltages into the data we want to analyse. Thus, the description of said calculations have been moved to the experiment section in this report. We set our focus in this section on the error analysis.

4.1 Error Analysis of Temperature Measurements

The advantage of this experiment was that the code directly converted the input voltages into a temperature value using the equations mentioned above. This simplified the work process as it enabled an instant feedback (e.g. via the LCD monitor). The downside to this is that all the data we collected comes without error propagation and we have to reverse engineer the accumulated errors. To do this we first analyse the error sources:

- Zero power resistance $R_0 = (100 \pm 1)\ k\Omega$.
- Nominal B-constant 4275 ± 25 K (Grove) and 6700 ± 250 K (circuit).

- We only measure integers, thus V_{meas} has an error of ± 0.5.

Now we can use python to receive the temperature including correctly propagated errors for a given input Voltage V_{meas}. If this is done for $V_{meas} = 1, 2, \ldots, 1023$, one can plot the uncertainty of T as a function of T (see figure 11). This graph is then used to add uncertainty values to all of the collected data, shown as error bars in the plots in the result section.

Note that for the equilibrium experiment an additional error of ± 0.3 °C appears as the temperature continued to fluctuate in this range even after some minutes of waiting.

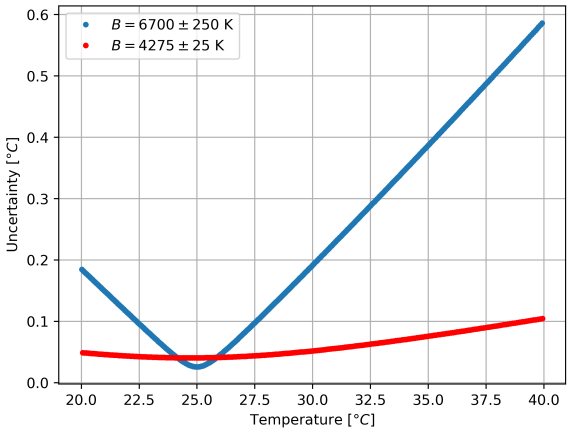

Figure 11: This plot shows how the errors propagate for the different sensors. We used $T_0 = 25$ °C in equation 3 so the error is minimal at that temperature. We see that the absolute error of the B-value has a significant impact on the total error propagation.

4.2 Error Analysis of the RPM Calculations

In this part we tried to minimize the measurement uncertainty by measuring the pulse signal at a high frequency. There are, however, two significant error sources for this experiment:

- To save memory the rotation count was measured as an integer variable. This leads to a high error quote as we later divide the count by 4 and multiply by 30 (see code) without converting the intermediate results to floating point variables. As `C` rounds all results of the division down to the next smallest integer, we lose up to 1 rotation in the division. The upscaling to one minute thus gives an error of ± 30 RPM.
- The rotation count is measured for 2 seconds and then a "prediction" is made for the rotations per minute. This is accurate for short periods but for periods lasting longer than 2 seconds we could actually miss a rotation. This error would, however,

too be rounded away in the division described above and thus does not effectively increase the absolute error. It is also to be noted that the fan did not spin (not even very slowly) for the points in the duty cycle where the RPM was measured to be 0, thus this is more a theoretical error source.

In conclusion the possible error sources are smaller than the rounding error in the code so the total error is ±30 RPM.

5 Discussion

All our measurements proceeded as expected. There are, however, some results that should be discussed in further detail.

When calibrating the NTC sensor in experiment 2.3 we found the B-value of 6700 ± 250 K which lies far off the value provided on the data sheet, which predicted a B-value of 4256 ± 25 K. It is hard to reason on the source of this disagreement as we have no information on how the sensor was originally calibrated. The result is, however, less surprising when compared to the results of colleagues who performed the same experiments in this laboratory. After short consultations it was found that most of them received B values in the range between 6000 and 9000 K.

The next result to be discussed are the measurements of the two-point cooling method shown in figure 7. We can clearly see that this method is, as expected, not optimal to achieve a stable temperature equilibrium. Even though the temperature hardly surpasses the higher limit, it undershoots the lower limit by more than one degree. The problem is that using this method the fan is either turned on on a full duty cycle or turned off completely: once the fan is turned off the blades continue to rotate for a moment until they completely stop, which explains the undershoot. One could argue that the large oscillation could be reduced by decreasing the difference between the limits, but then the problem would be that the fan does not turn on instantly and as mentioned needs some time to stop, too and thus would not be fast enough to handle smaller threshold differences. This method could be improved by controlling the fan's duty cycle proportional to the temperature difference which would enable a controlled slow down. This is, however, part of the last experiment, where we use the PID controller.

The PID controller worked remarkably better as it converged quickly and the oscillation stayed in a range of ±0.3 K. As mentioned above it comes with the advantage of the fan being fine tuned, which prevents extreme over- and undershoots as in the two-point method. Despite that, looking at the graph in figure 10 we see that using a PID controller alone does not provide good convergence. It is crucial to use fitting parameters k_p, k_i and k_d to achieve satisfying results. This is particularly clear regarding the measurements using

bad parameters. The parameters $k_p = 100$ and $k_d = 50$ are set too high (by a factor 5) which results in the fan reacting to strongly on both the error and it's change which then causes the temperature to oscillate. While the initial undershoot is roughly equivalent for both choices of parameters, the *good* parameters already have a smaller first overshoot and then nearly converge after around 100 s.

Finally, we will briefly discuss the fan's RPM behaviour. From figure 8 it can be seen that fan only starts rotating at a duty cycle of 14/255. This is probably due to the power being to weak to set the fan in motion. Once the fan starts spinning it behaves irregularly until the power is strong enough such that the inner resistance of the fan's mechanism is negligible. At this point the RPM increases in a predictable way as can be seen in the plot. However, the increase is not continuous: The RPM values always stay constant for 3 to 10 duty cycle values, which indicates that the real resolution of the signal is probably less than 255.

6 Conclusion

We consider the experimental session a success. All our results are within the range of our expectations with exclusion of the NTC's B-constant, where the measured result has, however, been confirmed by colleagues performing the same experiment. Furthermore, we gained a better understanding of the Arduino board and it's development environment when performing the experiments. The final task of building an automated cooling system for the circuit was achieved as well: Using the PID controller we were able to achieve a steady temperature equilibrium with oscillations in the range of ± 3 degrees Celsius.

7 Source Code

The Arduino code used in this experiment is available on GitHub: `https://github.com/gentinettagian/Arduino-ElectronicsAD.git`