Lab Report: Sensors and Signals

Carina Goetz, Pascal Kindling, Pedro Serrano

December 12, 2023

1 Introduction

1.1 Learning Objectives

In this Lab work, we want to focus on applying as much as possible of the theoretical part from the Sensors and Signals class to this practical project. Since we all have a decent knowledge on C and C++ and worked with the Arduino IDE and Hardware before, we don't have to focus on this background knowledge. We also already have a sufficient knowledge about electronics to build the circuits needed for our experiments and work with the PicoScope to further examine our test results and behaviour of the components.

We want to do a decent research about our sensor to find our some practical examples of the Type A and Type B Uncertainties. Therefore we also need to measure some temperature data sets. Then we use a timer based sampling method to see, if there is any measurable jitter that might affect the measurement timing of our overall system. Last but not least we will add a sine wave distortion on top of the sensors output signal. We will then design a digital filter apply them on the analogue readings, to filter the real measurement value of the sensor.

1.2 Setup

In this project, we used a RedBoard and a KY-013 analogue temperature sensor from an Arduino sensor kit. The RedBoard can be compared to the Arduino Uno, since they both feature the same Processing Unit, an ATMega328P, and similar pins and connections. The main difference between the two modules is only the price. The RedBoard can also be programmed easily in the Arduino IDE, using a simplified version of the C++ programming language.

The KY-013 as an analogue temperature sensor designed for a temperature range of $-55°C$ to $125°C$, that is based on a Negative Temperature Coefficient (NTC) thermistor. That is a type of semi-conductive resistor, whose resistance changes according to the temperature around it. The NTC specifies, that the thermistors resistance decreases when the surrounding temperature increases. The KY-13 also consists of a $10k\Omega$ resistor that is connects to the signal output of the whole sensor. Since the NTC thermistor decreases it's resistance, it will also decrease it's Voltage output, that can then be measured on an analogue pin of the RedBoard.

According to the Documentation of the KY-013[1], the three male pins of the sensor have to be connected like in the wiring diagram 1, whereas the operating voltage of the sensor is $5V$. Because the analogue sensor pins are mapped to a $5V$ reference voltage, the measured voltage can be interpreted as an analogue value as it is.

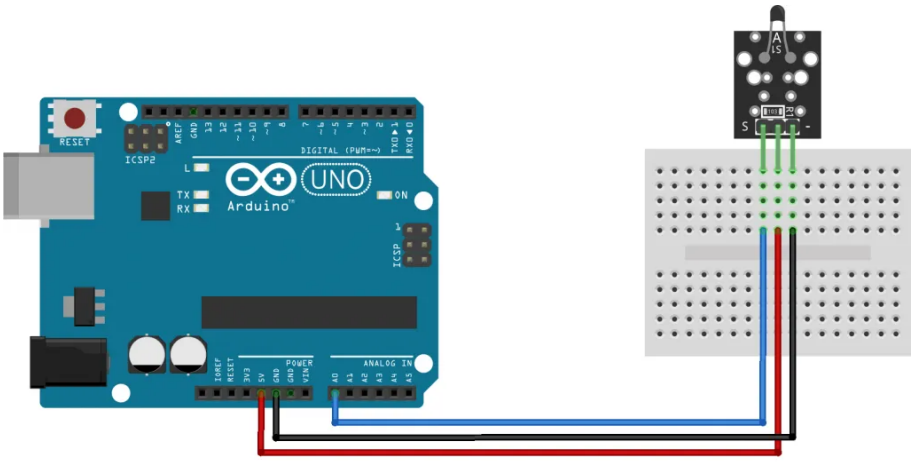


Figure 1: Basic Circuit with the Arduino Uno and the KY-013

After reading the analogue output Voltage at the connected pin of the RedBoard, we still have to apply the Steinhart-Hart equation to get a useful temperature result.

$$\frac{1}{Temperature} = A + Bln(R_{thermistor}) + C(ln(R_{thermistor}))^3 \tag{1}$$

The Steinhart-Hart coefficients A, B, C are given by the distributor/manufacturer[1], who calibrated the sensor to get these coefficients for the desired measurement range. To calculate the Resistance of the NTC thermistor from the analogue value, we first have to use the following formula to get the output Voltage:

$$V_{output} = \frac{AnalogValue \cdot V_{supply}}{Resolution} \tag{2}$$

Whereas the V_{supply} is always at a constant 5V and the maximum $Resolution$ corresponds to the 10 Bit analogue read resolution of the RedBoard. That means the analogue value could be in the range of 0 to $2^{10} - 1 = 1023$ and the maximum resolution being 1023. Using the V_{output} we can calculate the resistance of the thermistor as shown in the equation 3, where $R_{multiplier}$ corresponds to the additional $10k\Omega$ resistor before the output pin.

$$R_{thermistor} = \frac{V_{output} \cdot R_{multiplier}}{V_{supply} - V_{output}} \tag{3}$$

After we solved for all these components, we can apply them in the Steinhart-Hart Equation 1 and solve for the Temperature in Kelvin. With a quick $Temperature - 273.15$ we get the Temperature in Celsius.

The basic code we used for the first of our measurements can be found in the appendix 1.

[1] KY-013 Analog Temperature Sensor Module

2 Uncertainty

2.1 Type-A Uncertainty

A data set comprising 5000 recorded values of room temperature was analysed to ascertain the central tendency and spread of the measurements. The mean temperature calculated from this data set was $20.7°C$ with a standard deviation of 0.0793. The mean temperature serves as the central estimate of the room temperature readings. The standard deviation reflects the spread or dispersion of the collected temperature values around this mean.

It's important to note that the standard deviation provides a measure of the variability within the data set. In the context of these room temperature measurements, the standard deviation suggests that the individual measurements deviate, on average, approximately 0.0793 from the mean temperature.
To account for this uncertainty, a confidence interval around the mean temperature was calculated.

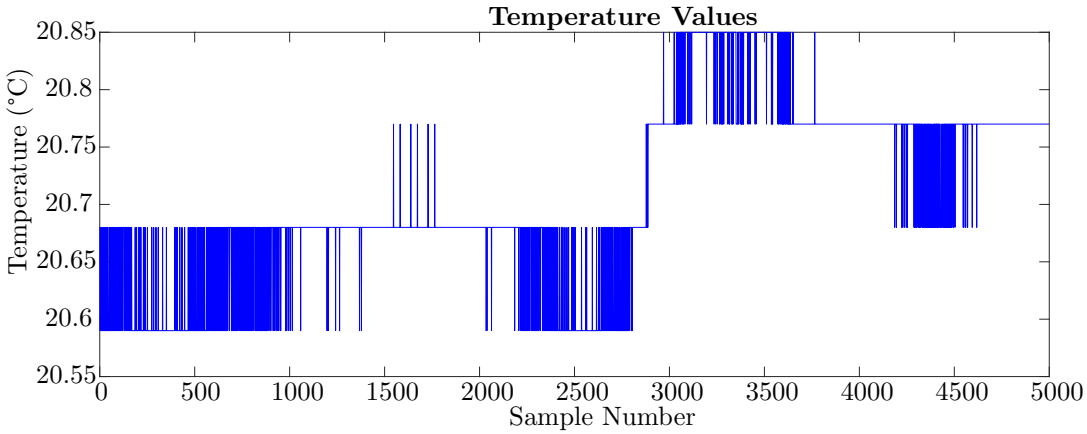
$$StandardError = \frac{\sigma}{\sqrt{n}} = \frac{0.0793}{\sqrt{5000}} \cong 0.00112 \tag{4}$$

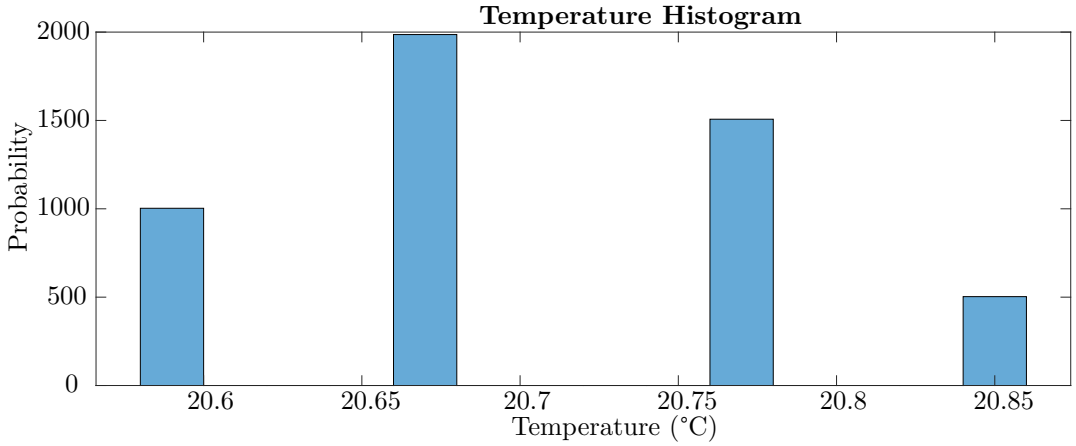
$$MarginofError = CriticalValue \cdot SE = 1.96 \cdot 0.00112 \cong 0.0022 \tag{5}$$

$$LowerLimit = Mean - MarginofError = 20.7 - 0.0022 = 20.6978 \tag{6}$$

$$UpperLimit = Mean + MarginofError = 20.7 + 0.0022 = 20.7022 \tag{7}$$

Using a confidence level of 95%, the confidence interval spans from $20.6978°C$ to $20.7022°C$. This range provides a plausible interval within which the true population mean temperature is expected to lie with a 95% confidence.





2.2 Type-B Uncertainty

We identified two main sources of uncertainty in our project: the sensor and the ADC converter from the RedBoard.
Looking at the sensor data sheet, all we can gather is that it has a measurement accuracy of $\pm 0.5°C$. We didn't find any specific tolerance of the sensors $10k\Omega$ multiplier resistor (because those could also be a minor source of uncertainty), but since most consumer electronics use resistors with a 1% tolerance, we could assume this value for the multiplier resistor.
Regarding the ADC accuracy, we looked into the data sheet and were are to find multiple sources of uncertainty that we will further describe:

- Offset Error
- Gain Error
- Integral Non-linearity (INL)
- Differential Non-linearity (DNL)
- Calibration accuracy

2.2.1 Offset Error

Offset error refers to a deviation between the actual analog input voltage and the expected or ideal input voltage that should be converted by an ADC.
This error occurs even when there's no input signal applied to the ADC. For example, if we have an ADC outputting a value of 0, meaning it is representing a zero input voltage, what happens is that, due to imperfections, such as manufacturing variations for example, there might be a small non-zero value produced.
This discrepancy between the expected and actual output at zero input is termed as the offset error.
Offset error can affect the accuracy of the ADC's readings, especially for small input signals, therefore calibration techniques are often employed to mitigate offset errors and improve the accuracy of ADC readings.
The ideal value is 0 LSB, but in our case we have a variation that can go from –3.5 LSB to +3.5 LSB.

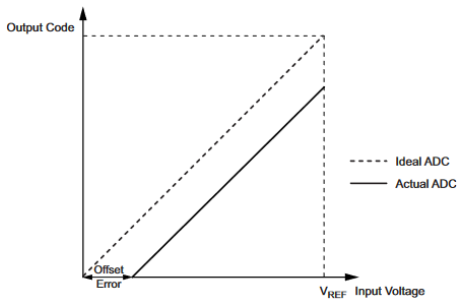


Figure 2: Offset Error

2.2.2 Gain Error

Gain error refers to the deviation in the amplification factor of an ADC. It represents the difference between the actual gain of the ADC and the ideal gain it should have for converting analog input voltages to digital values accurately.
In an ideal scenario, the gain of an ADC would precisely convert an analog voltage into the corresponding digital value according to its specified resolution and range, but due to various factors like component variations, the gain of the ADC might deviate from its intended value. This causing the ADC to output digital values that are either higher or lower than they should be based on the ideal conversion.
Like offset error, gain error can also be compensated for or calibrated to enhance the accuracy of the ADC readings.

The ideal value is 0 LSB, but, like in the offset error, we have a variation that can go from -3.5 LSB to +3.5 LSB.

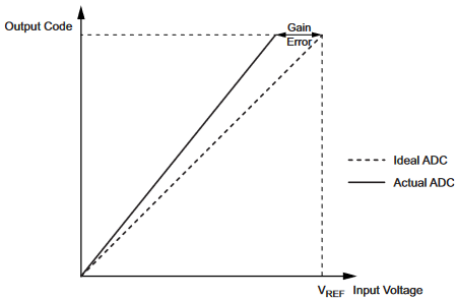


Figure 3: Gain Error

2.2.3 Integral Non-linearity (INL)

After adjusting for offset and gain error, the INL is the maximum deviation of an actual transition compared to an ideal transition for any code. It quantifies the maximum deviation between the actual output of the ADC and the ideal straight-line transfer function across its entire input voltage range.
A low INL indicates better linearity and accuracy in the conversion process, whereas a higher INL value signifies greater deviation from the ideal output. Calibration and correction techniques are often utilised to minimise INL and enhance the overall accuracy and linearity of an ADC.
The ideal value is 0 LSB. In our ADC, the maximum INL is 1.5 LSB.

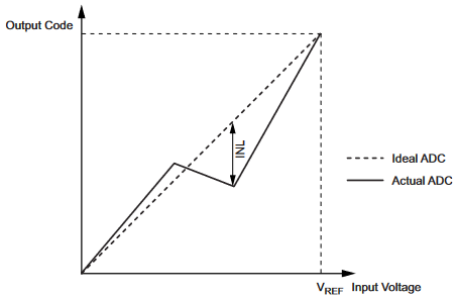


Figure 4: Integral Non-linearity

2.2.4 Differential Non-linearity (DNL)

DNL quantifies the variation in the step size between adjacent digital output codes of the ADC, compared to the ideal or expected step size. A positive DNL means that the step size is larger than expected, while a negative DNL indicates a smaller step size than anticipated.
When the ADC exhibits perfect linearity, the step size between adjacent digital codes remains consistent and matches the ideal value. However, real-world ADCs may have imperfections, leading to variations in the step sizes. These variations can be caused by mismatches in components, non-idealities in the conversion process, or other factors.
Calibration and correction techniques are employed to minimise DNL and improve the linearity and accuracy of the ADC's output.
The ideal value is 0 LSB. In our ADC, the maximum INL is 0.7 LSB.

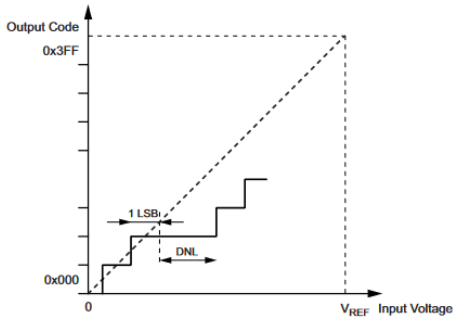


Figure 5: Integral Non-linearity

2.2.5 Calibration accuracy

The components we used are low-end components, which means they might have been calibrated via batch calibration and not individual calibration.
Batch calibration is usually applied when we have a large-scale production of devices, including ADCs, where individual calibration of each component might be impractical or time-consuming. Instead of calibrating each unit separately, batch calibration involves calibrating a group or batch of devices together, assuming that the units within a batch exhibit similar characteristics.

While batch calibration can be more efficient and cost-effective in large-scale production, it's essential to note potential limitations, one of them being potential loss of individual precision. These means that individual characteristics of units might not be fully addressed with batch calibration, potentially leading to slight discrepancies among units.
This leads to a limited precision, which means we might not achieve the same precision as individual calibration for each unit.

[1] ATmega328P Data sheet

3 Jitter analysis

In this chapter, we dive into the analysis of the jitter. An accurate constant sampling rate is one of the key aspects of a good measurement, as jitter can occur due to other interrupts or time consuming tasks that are done simultaneously. Jitter does not only affect the timing, but also influences the data values that are measured. Therefore our focus lays on examining the timing consistency of collecting our temperature data by using a digital pin to map the start to end of one measurement cycle to a high signal.

This chapter serves as a critical examination of the nuances surrounding jitter, offering insights that contribute to the overall reliability and validity of our laboratory results.

3.1 Setup

For this task, we added a PicoScope to the basic circuit we used to measure the data. because we wanted to measure the time one measurement cycle took, we wired the PicoScope probe to digital pin 13. The idea is, that we would set the signal of the digital pin to high when the measurement cycle would start and set it to low when it would end. Then the change of the signal could be evaluated with the PicoScope. Wiring diagram 6 shows the new setup we used to analyse the jitter.

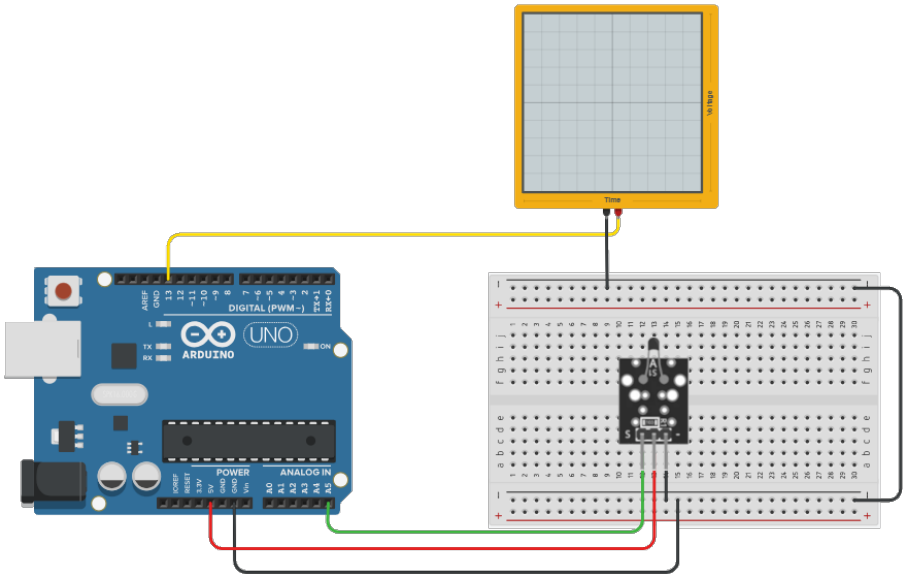


Figure 6: Circuit with PicoScope for Jitter Analysis

To insure that a constant sampling time is used, functions like delay should not be implemented, because they are interrupting the system. An alternative, that doesn't use an interrupt, is a timing based condition using millis or micros. Here, a previous and a current time is saved and if the difference between the previous and current time is greater than a specific value, for our case $200\mu s$, the condition is true. Then the current time is set to the previous round time and the measurement can be taken. The code we used for the constant sampling time can be found in the appendix 2. To minimise the amount of interruption and time consuming tasks and only focusing on the time the microcontroller takes reading the data, we stripped back the code to only reading the analogue value and not calculating or printing values. When using the PicoScope to analyse the jitter, the digital pin 13 has to be set in the code. It is set to high right after the if-clause and before the current time value is mapped to the previous time value and it is set to low after the temperature value is read.

3.2 Results

To evaluate the jitter efficiently, the PicoScope settings had to be set correctly. Because we had a signal changing between $0V$ and $5V$, the PicoScope has to visualise more than $5V$. To actually see changes over time, the PicoScope has a persistence mode, that works similarly to a memory mode on an Oscilloscope. The difference here is, that it not only visualises multiple time-lapses on top of each other, but also highlights with colours, where lines are overlapping each other. Blue lines visualise parts that are not frequently overlapped and yellow parts are more frequently used. In parts that are red, the lines overlap the most. These coloured diagrams are shown in figure 7 and 8, where on the x-axis there is the time in microseconds and on the y-axis there is the voltage.

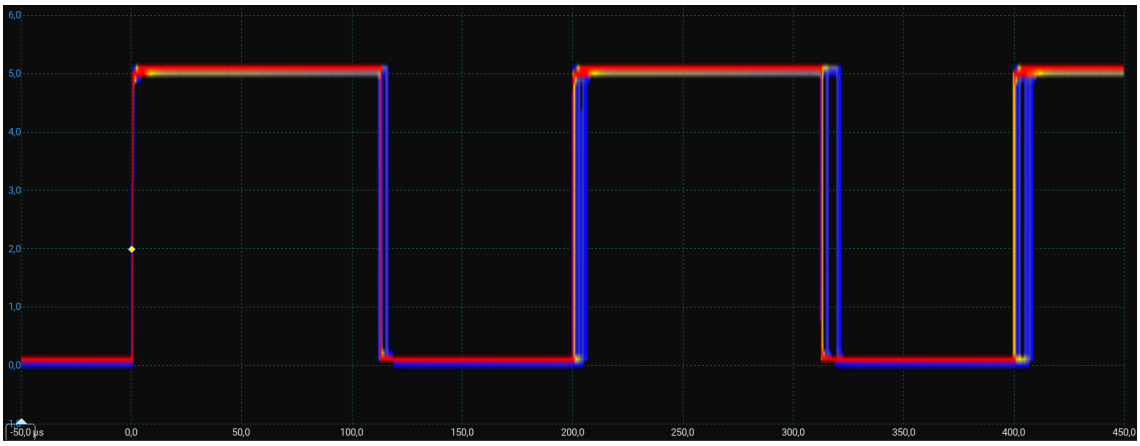


Figure 7: Sector from the PicoScope Measurement of the Jitter

In figure 7 it can easily be seen, that the measurement interval is set to $200\mu s$, as around every $200\mu s$, there is a rising edge. One important feature to notice in this figure, is that the not only does the time of a measurement cycle (until the falling edge) vary, but also the timing of the next beginning of a cycle. Even though a timing based condition is used, other interrupts could delay a code to be executed and therefore lead to such variations in time. But in most of the cases, the next cycle starts at the correct time, which is visualised by the red rising edges. Another important feature is, that the discrepancies are not standard deviations but quanta.

The discrepancies are also quanta at the end of one measurement cycle. One cycle takes most of the time around $112\mu s$, which is visualised in figure 8 by using rulers. Sometimes the jitter has a value of up to around $5\mu s$. In later cycles there is the possibility, that the previous jitters add up to each other and therefore increase even more, but in our analysis, only slight increase was observed. This increase can be seen when comparing all rising or all falling edges in figure 7.

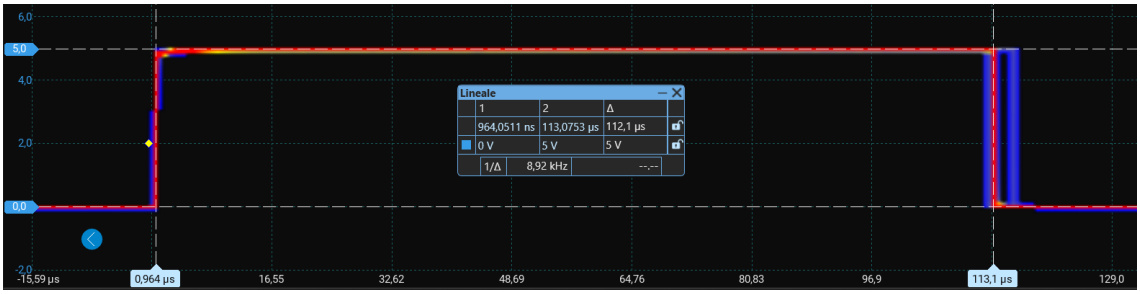


Figure 8: Median Time of one Measurement Cycle from the PicoScope Measurement

4 Testing a digital Filter

After analysing the jitter of our system, we have another real world problem we can test our system with. Measurements are often affected by additional signals, that might make the measurement signal unrecognisable without taking care about those distortions first. A popular signal distorting measurements are caused by the electricity grids, that a found basically everywhere around the globe. In Europe, those operate at a 50Hz sine wave frequency, that distorts basically all the measurements taken near the grid, so they have to be taken care of. A popular possibility to get rid of the distortion is to design a hardware or a digital filter, where only the desired signal spectrum can pass through.

first of all we had to rewire our circuit, so we could add an artificial signal distortion on top of our sensor output. Therefore we used the PicoScope as a sine wave generator and added an additional $100pF$ capacitor. This is needed because otherwise almost all of the measured voltage would be only from the sine wave generator, because of the $10k\Omega$ resistor of the Sensor is way higher than the internal resistor of the sine wave generator. The capacitor counters this problem and adds both signal on top of each other. The wiring diagram 9 shows the new layout we used, to add a sine wave with an amplitude of $1V$ and a frequency of $50Hz$ on top of the sensors output signal. Additionally, we wired the PicoScopes measuring probe to the digital pin 13, where we want to get a feedback on our systems performance.

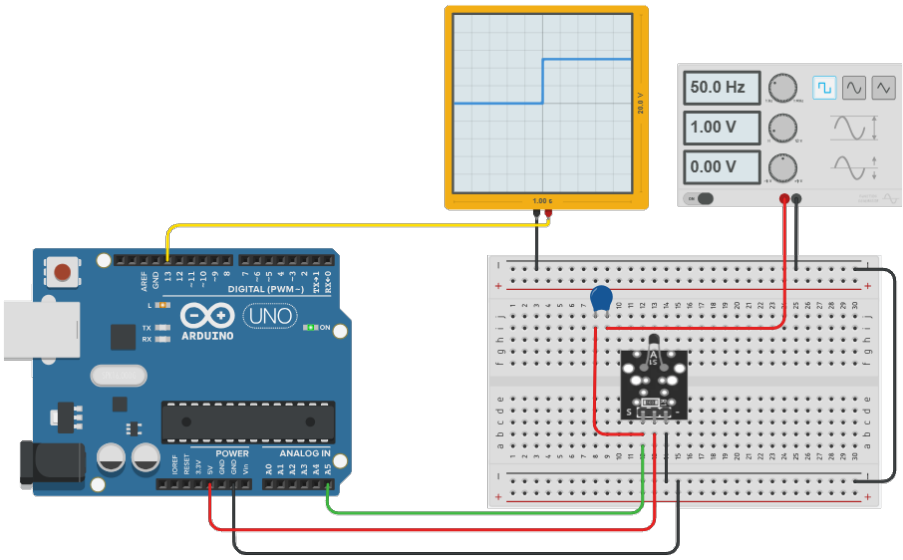


Figure 9: Circuit for using a digital filter

Now we have a distorted signal, that can be measured on the analogue pin of the RedBoard. We will now use a free online tool [1] to design a digital filter, that is able to remove the frequency of 50Hz from our measurement spectrum. We designed an Finite Impulse Response (FIR) filter, because they have a linear phase response, that means they do not distort the signal waveform. Additionally those filters need less calculating power and memory, which increases the overall performance of our system.

In the filter designer, we defined our sampling frequency with 150Hz, because according to the Nyquist Theorem, we need at least twice out highest frequency. Since our highest frequency is at 50Hz, the defined 150Hz are sufficiently above that minimum. We than designed a filter as shown in 10, that needed 17 filter taps and had quite a nice attenuation at the 50Hz mark.
Next, we had to implement this filter to run on the Arduino. To do that we used the online converter to a C++ file and imported that into our source code. To make it simpler we removed the struct and just used the functions of the online implementation. Our implementation can be found in the appendix 3. The first of the three filter functions is the filter_init(). Here we clean the history array, where the last 17 measurements will be stored during the filtering process. Next we have the filter_put() function, that uses the analogue Value as an input. The the new value will replace the oldest measurement value

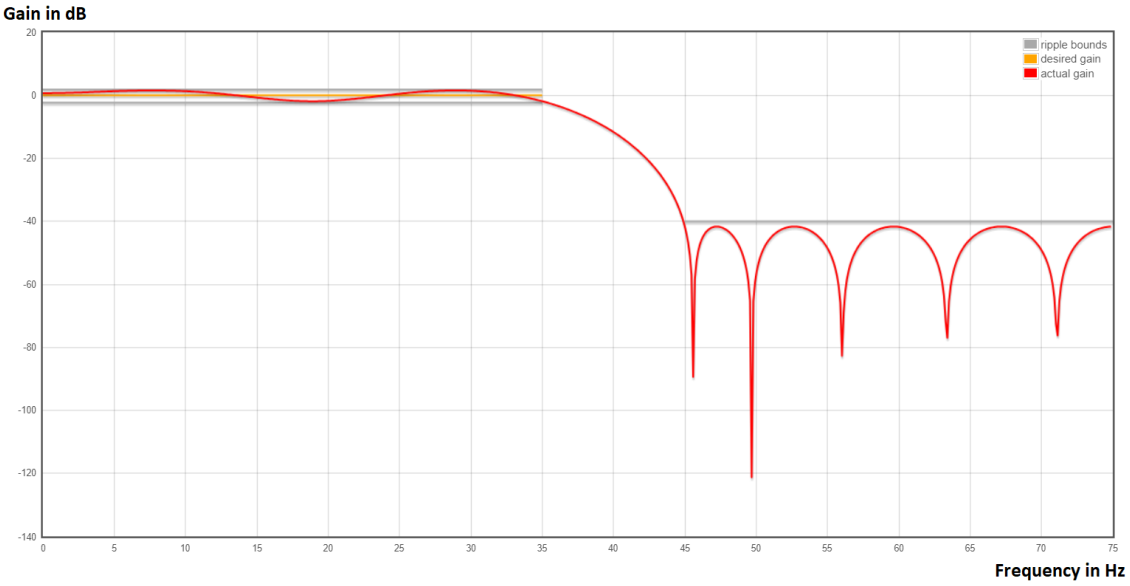


Figure 10: The Gain vs. Frequency plot of the designed filter

in the history array. The last function is the filter_get function. This does the important calculations to receive a filtered value without the interference of the sine wave distortion. Here the typical operations of the FIR filter take place. All the values in the history array get multiplied by the corresponding tap value, then those results are all added up and produce the filtered measurement value.

The plot 11 (blue) shows the measurements we have taken in our experiment before and after applying the filter. It's easy to see, that the measured signal is not more or less a constant value, but you can see the sine wave we added on that signal.

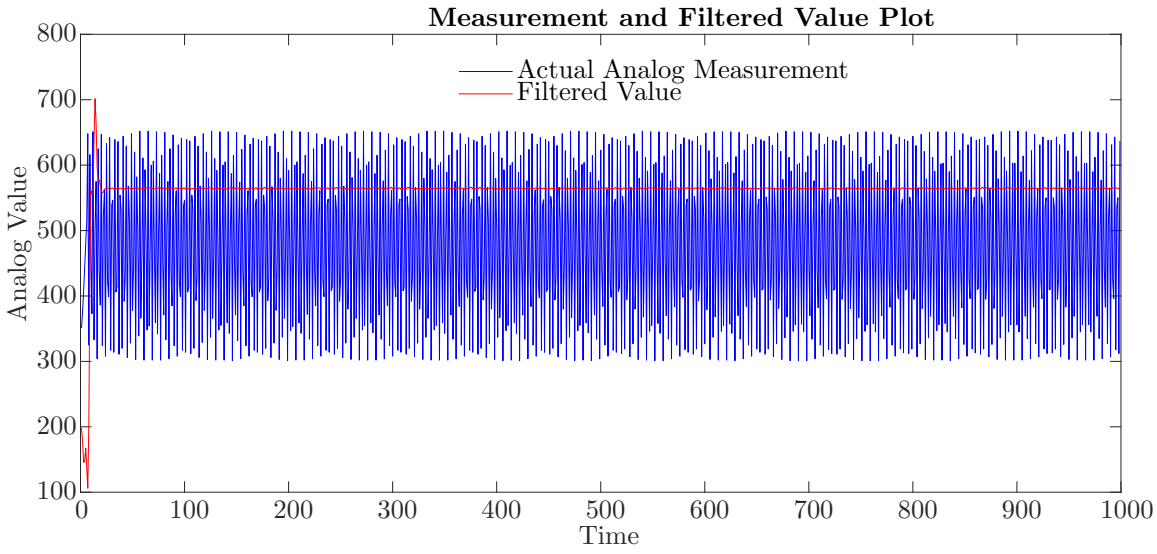


Figure 11: The analogue measurement and filtered value

The filtered signal should be an almost constant value again, that corresponds to the voltage output of the sensor. After applying the filter and it's saturation after a few measurements (because it has to fill all 17 taps in the history array with values first), we are getting quite a nice constant behaviour of our filtered value 11 (red). This shows, that our filter works quite fine, but we still have to see, if the values it's generating are plausible temperatures in the lab.

To check this, we used a short Matlab script, that just imports the analogue data and converts it using the formulas described in 1.2 into temperature readings. The plot 12 shows, that our Temperature values are slightly above $20°C$ and that is a very comfortable temperature in our Lab.

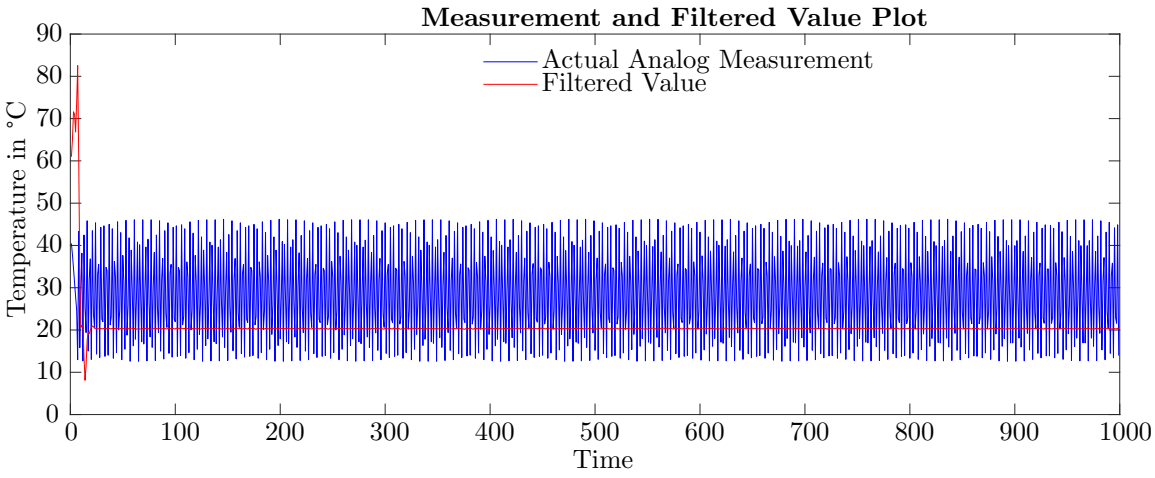


Figure 12: The analogue measurement and filtered value converted to temperature

This filtered Temperature can be evaluated as correct, because it is in the same range as we measured in the tasks before, like in 2.1. Therefore our filter does the job it was designed for quite well. The last thing we can evaluate is it's performance, because the calculations of the FIR need more calculating power than just reading the analogue values. To do that we take a look at the PicoScope wired to the digital pin 13, as shown in 9. As in 3.2, we use the PicoScopes Persistence mode to take a look at the time it takes for applying the filter. The image 13 shows the Persistence reading of the whole filtering process.

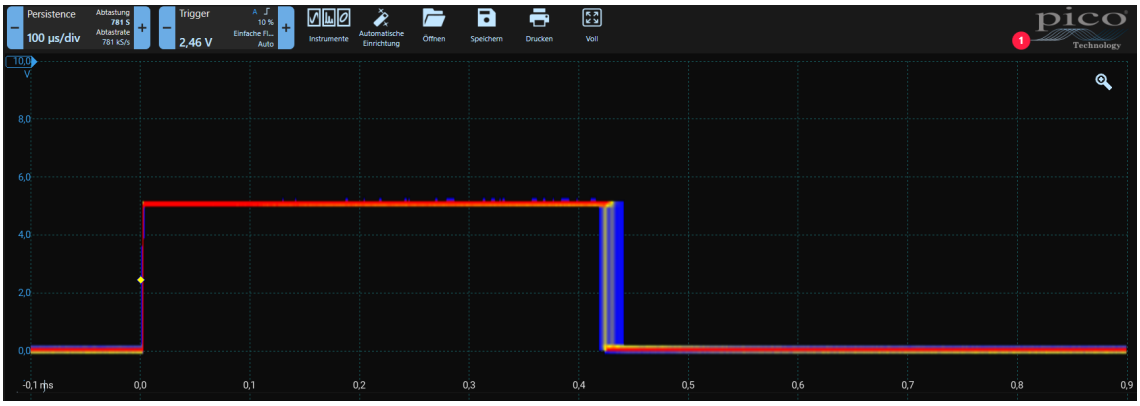


Figure 13: PicoScope Persistence readings of a filtering cycle, Time in ms

It is easy to see, that one cycle of filtering needs slightly more than $0.4ms = 400\mu s$. Compared to the measurements we took in the jitter analysis 7, the filter needs about $300\mu s$ more calculation time. This is because now we have to perform operations on an array, and do the filters calculations for every newly measured value. Because our filter has 17 taps, we need to perform 17 multiplications and 16 additions in every run. Since we sampled at a rate of $150Hz$, we are taking one sample every $6.7ms$, so there is still enough time to complete the filtering calculations before the next run starts.

[1] Free online FIR filter designer

5 Summary of Learning Experience

Out intentions for this Lab works were to apply as much as possible of the theoretical background we learned in the Sensors and Signals class. We hoped to further understand the contents by using them in a practical experiments. As explained in the report, we managed to test all the required and the optional experiments and achieved quite good measurements with our analogue temperature sensor. It was very helpful, that all of us already had experience with the Arduino IDE and working with the hardware we needed, therefore we had enough time to focus on our tests.

At first, we had a bit of a problem by getting good measurements on an ESP32. Those problems were related to the fact, that the ESP maps the incoming analogue signals to a $1.1V$ Reference Voltage, not $5V$ as the RedBoard. To counter this effect, we would have to use additional resistors to get the signals voltage range from 0 to $5V$ down to 0 to $1.1V$. To simplify the setup in this report, we mostly sticked to the RedBoard when measuring, but still sometimes worked around with the ESP, just for the experience.

In the first part gained a good amount of experience at working with the Hardware Documentation and identifying uncertainty sources. Afterwards we mainly used the PicoScope to see the effects of the jitter in real time. It was very interesting, that despite our simple and efficient code, we were still able to detect some jitter in our system when performing the ADC operations. Last but not least we simulated a distorted sensor signal and tried a digital filter to get rid of our artificial distortions. This was the first time we were really able to see the effect of a digital filter in real-time on our incoming data. We also were very happy, that our simple FIR filter was already able to remove our distortions quite well.

All in all we achieved all of our desired learning achievements and were able to experiment with many practical issues of using analogue sensors.

6 Code Appendix

```
//constants
#define ThermistorPin A5
#define V_Supply 5.0
#define RMultiplier 10000
#define Resolution 1023
#define delayTimeMilliseconds 500

//variables
int analogVal;
float V_Out,ln_RTherm, RTherm, Temp;
float A = 0.001129148, B = 0.000234125, C = 0.0000000876741;

void setup() {
  Serial.begin(9600);
}

void loop() {
    analogVal = analogRead(ThermistorPin);

    V_Out = analogVal*V_Supply/Resolution;
    RTherm = (V_Out*RMultiplier)/(V_Supply-V_Out);
    ln_RTherm = log(RTherm);
    Temp = (1.0 / (A + B*ln_RTherm + C*pow(ln_RTherm,3)));

    Temp = Temp - 273.15;
    Serial.println(Temp);
    delay(delayTimeMilliseconds);
}
```

Listing 1: Code for the usual delayed measurement method

```
//constants
#define ThermistorPin A5
#define V_Supply 5.0
#define RMultiplier 10000
#define Resolution 1023
#define measurementIntervalMicros 200

//variables
int analogVal;
float V_Out,ln_RTherm, RTherm, Temp;
float A = 0.001129148, B = 0.000234125, C = 0.0000000876741;
//for a timed measurement
unsigned long previousMicros, currentMicros;

void setup() {
  Serial.begin(9600);
  previousMicros = micros();
}

void loop() {
  currentMicros = micros();

  // Check if it's time to take a measurement
  if (currentMicros - previousMicros >= measurementIntervalMicros) {
    previousMicros = currentMicros;
    analogVal = analogRead(ThermistorPin);

    V_Out = analogVal*V_Supply/Resolution;
    RTherm = (V_Out*RMultiplier)/(V_Supply-V_Out);
    ln_RTherm = log(RTherm);
    Temp = (1.0 / (A + B*ln_RTherm + C*pow(ln_RTherm,3)));

    Temp = Temp - 273.15;
    Serial.println(Temp);
  }
}
```

Listing 2: Code for the timed measurements using micros

```
//constants
#define analogInPin A5
#define digitalPin 13
#define FILTER_TAP_NUM 17

//variables
int lastSampleTime, last_index = 0;
float filteredValue;
double history[FILTER_TAP_NUM];
String k = ",";

static double filter_taps[FILTER_TAP_NUM] = {
  -0.0053716533078543875,
  -0.06169932676967435,
  -0.08288645925622837,
  0.018846581918095025,
  0.06791228614536901,
  -0.0758787741210704,
  -0.0754728884948242,
  0.30757107436393344,
  0.5792669239203247,
  0.30757107436393344,
  -0.0754728884948242,
  -0.0758787741210704,
  0.06791228614536901,
  0.018846581918095025,
  -0.08288645925622837,
  -0.06169932676967435,
  -0.0053716533078543875
};

void filter_init() {
  for(int i = 0; i < FILTER_TAP_NUM; ++i) {history[i] = 0;}
}

void filter_put(double input) {
  history[last_index++] = input;
  if(last_index == FILTER_TAP_NUM) {last_index = 0;}
}

double filter_get() {
  double acc = 0;
  int index = last_index, i;
  for(i = 0; i < FILTER_TAP_NUM; ++i) {
    index = index != 0 ? index-1 : FILTER_TAP_NUM-1;
    acc += history[index] * filter_taps[i];
  };
  return acc;
}

void setup() {
  Serial.begin(9600);
  filter_init();
  pinMode(digitalPin, OUTPUT);
  lastSampleTime = micros();
}

void loop() {
    if (micros() - lastSampleTime > ((1/150) * 1000000)){ //sample with 150Hz
      lastSampleTime = micros();
      digitalWrite(digitalPin,HIGH);

      filter_put(analogRead(analogInPin));
      filteredValue = filter_get();

      digitalWrite(digitalPin,LOW);
      Serial.println(filteredValue);
    }
}
```

Listing 3: The code for applying a digital filter to the measurements