Python Module: Nidaqtemp

Documentation

Marc Antonio Vöhringer Carrera

September 9, 2020

Contents

1 Introduction

The nidaqtemp module is supposed to let the user read out the platinum temperature sensors connected to the *Pt1000-Messumformer* allowing to measure the temperature of up to six sensors simultaneously. It provides the class `temptask` and a few methods which allow the user to calibrate the sensors, and read out as well as record the temperatures to a file.

2 The *Pt1000-Messumformer* and the platinum temperature sensors

The sensors have a temperature dependent resistance ($1000\,\Omega$ at $0\,°\mathrm{C}$), from which the temperature can be calculated. Each sensor is connected to the input of the *Pt1000-Messumformer* as in the schematics (Fig. 1). The sensor is connected to the CH1 input port and the output X1 goes to the analog input of the *NI USB-6009*. From the resulting voltage the `nidaqtemp` module can calculate the temperature.

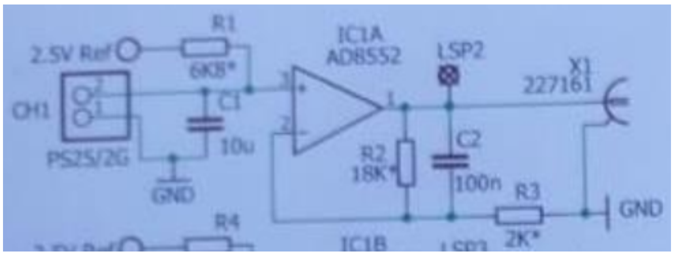


Figure 1: Schematics of the *Pt1000-Messumformer*. The sensor is connected to the CH1 input port and the output X1 goes to the analog input of the *NI USB-6009*.

The formula for calculating the voltage at the *Pt1000*s output is

$$U = \frac{R_{\mathrm{pt}}}{6800\,\Omega \cdot R_{\mathrm{pt}}} \cdot U_{\mathrm{ref}} \cdot v, \tag{1}$$

where $U_{\mathrm{ref}} = 2.5\,\mathrm{V}$, $v = 10$ is the OpAmps amplification and R_{pt} is the sensors resistance. By reshaping Eqn. 1 we arrive at

$$R_{\mathrm{pt}} = \frac{6800\,\Omega}{\frac{25\,\mathrm{V}}{U} - 1}. \tag{2}$$

The platinum sensor resistances temperature dependence is described by

$$R_{\mathrm{pt}}(T) = R_0 \cdot \left(1 + A \cdot T + B \cdot T^2\right), \tag{3}$$

where $R_0 = 1000\,\Omega$ is the sensors resistance at $0\,°\mathrm{C}$ and $A = 3.9083 \times 10^{-3}\,°\mathrm{C}^{-1}$ and $B = -5.775 \times 10^{-7}\,°\mathrm{C}^{-2}$ are constants according to IEC 6051 TK3850ppm.

Therefore the Temperature is

$$T = -\frac{A}{2B} - \sqrt{\frac{R_{\mathrm{pt}} - R_0}{R_0 \cdot B} + \left(\frac{A}{2B}\right)^2}. \tag{4}$$

The `nidaqtemp` module includes a function calculating the temperature from the *Pt1000-Messumformer*s output voltage according to Eqns. 2 and 4.

3 The `temptask` class

Initializing a `temptask` instance creates a task on the *National Instruments USB-6009*. It can be initialized with three optional arguments: **sampsize**, **samptime** and **chanlist**.

- **sampsize**: Default value is 200. This means that one single temperature measurement will be taken as the average of 200 samples to average out the electronic noise.

- **samptime**: Default is 100 ms. Specifies the time over which the samples are collected (in ms).
- **chanlist**: List containing the channels to be included in the task. Default is `[1,2,3,4,5,6]`, meaning that all sensors (channels) will be included in the task.

The sampling rate is calculated (in Hz) by taking `int(sampsize/(samptime*1e-3))`. This value multiplied with the number of channels included in the task can not exceed 48 kHz as this is the maximum sampling rate for the device. An example on how to initialize a task is shown below.

```
measurement = temptask(sampsize = 100,
                       samptime = 50,
                       chanlist = [1,2,5])
measurement.closetask() #closes the task at the end of the code
```

This example initializes a task where each temperature measurement will consist of the average of 100 samples taken over 50 ms. The task only includes channels 1,2 and 5, so the temperatures from sensors 3,4 and 6 can not be measured with this task. This is why by default all six channels are selected. If one or more sensors are not connected to the *NI USB-6009* they should not be included in `chanlist` as this will return an error. After initializing the task it is closed with the `closetask` method. This has to be done as otherwise python will return an error.

Note: No temperature is measured yet! This can be done with the `gettemp` method.

3.1 `gettemp`

The `gettemp` method measures the current temperature of all sensors included in the task and saves them as a `numpy` array. The temperatures can be accessed by calling the attribute `chan_temps` and they are also returned by the `gettemp` method. Alternatively the method can also be called with a single `integer` channel number as argument, then the method will return just the temperature of the specified channel as a `float`. Regardless, the temperature of all channels is saved as `chan_temps` attribute. Example:

```
measurement = temptask(cahnlist = [1,2,3])
print(measurement.gettemp(2))
print(measurement.chan_temps)
measurement.closetask()
```

Output:

```
21.717021846626405
[21.75506446 21.71702185 21.75239309]
```

3.2 `calibrate`

It is possible that the sensors show slightly different temperatures as they have to be calibrated. This should normally best be done by exposing them to a well known temperature like boiling water and take the deviation from this temperature as global offset.

While this is in theory possible by doing the above and putting the offsets manually into a file called "`calibration_data.txt`" the `calibrate` method provides a much simpler means of calibrating the sensors. This is done by positioning the sensors next to each other and waiting some time until they are in thermal equilibrium. Calling the `calibrate method` will then average the temperature of each sensor over $N = 100$ individual measurements. The mean value of all channels is taken as "true" temperature value and the offsets from this value are saved in the "`calibration_data.txt`" file. Additionally a plot is created showing the offsets. The error bars indicate the standard deviation of the individual channel data from the average. An example of such a plot is shown in Fig. 2.

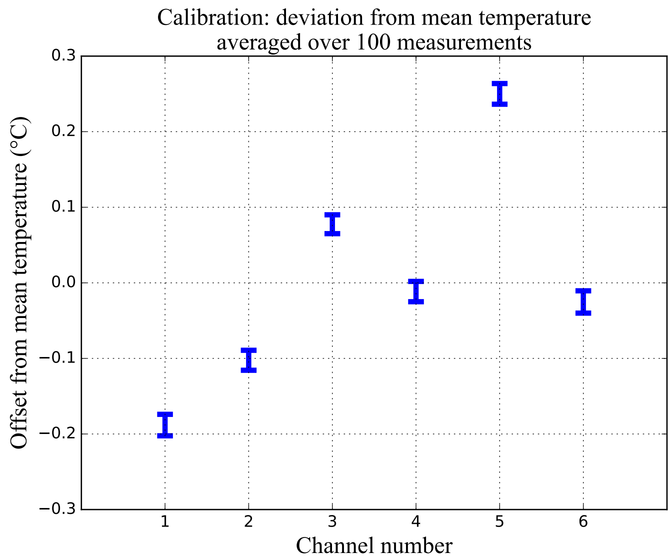


Figure 2: Example graph showing the offset values of the calibration.

The method can also be called with an `integer` as argument, specifying the number of individual measurements N over which the offset is averaged.

3.3 `savetemp`

Takes a filename as argument, which is "`temp_record.txt`" by default and appends the `chan_temps` attribute array to the file, along with the date and time at which the `savetemp` method is called. If the file does not exist a new file is created. Example:

```python
import time
measurement = temptask(chanlist=[1,2,3])
for i in range(5):
        measurement.gettemp()
        measurement.savetemp("temperature.txt")
        time.sleep(1) #waits one second
measurement.closetask()
```

This creates the file "`temperature.txt`" containing:

```
08.09.2020--19:25:00.3  21.694  nan     21.682  21.679  nan     21.696
08.09.2020--19:25:01.5  21.719  nan     21.698  21.686  nan     21.662
08.09.2020--19:25:02.6  21.711  nan     21.682  21.667  nan     21.678
08.09.2020--19:25:03.7  21.719  nan     21.671  21.684  nan     21.702
08.09.2020--19:25:04.8  21.708  nan     21.695  21.696  nan     21.683
```

It is important to note that `gettemp` has to be called before `savetemp` to update the values in the `chan_temps` array which are written to the file.

This process of saving the temperature every x seconds can also be done automatically using the `record` method.

3.4 `record`

This method takes one obligatory argument `t_total` which is the total time over which the temperature should be recorded. The second argument `t_interval` specifies the time interval in seconds between the individual measurements, which is 1 s by default. The last argument `filename` is the name of the file to which the temperature values are going to be appended. It is also "`temp_record`" by default. Example:

```
        measurement = temptask()
        measurement.record(5, 1, "temperature.txt")
        measurement.close()
```

This creates a similar output to the text file than in the previous example, except that the timestamps are more equally spaced, as the method takes the time that is needed to execute the code into account. Also all channels are included in the task and saved to the file in this example, as the task is called without arguments.

3.5 `printtemp`

Provides a quick way to print the values saved in `chan_temps` to the console. Has to be called after `gettemp` to show upgraded temperature values. Example:

```
measurement = temptask(chanlist=[1,3,4])
measurement.gettemp()
measurement.printtemp()
measurement.closetask()
```

Output:

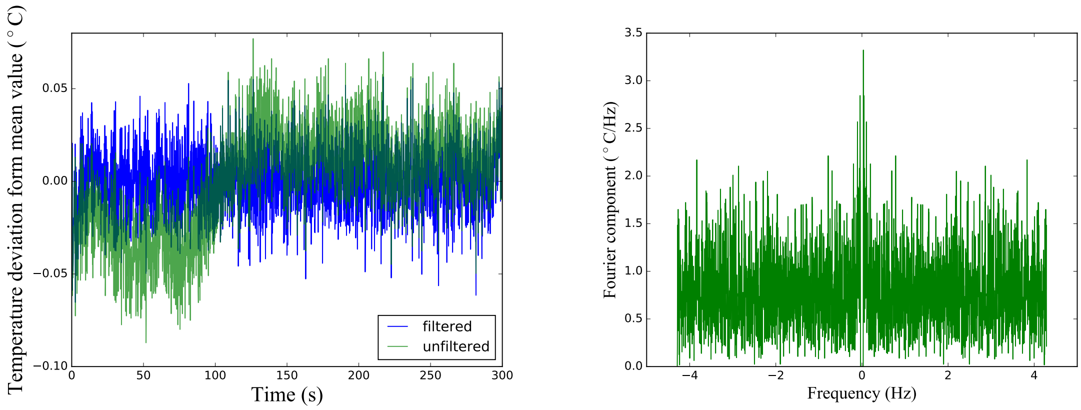
```
ch1:  21.67 °C  ch3:  21.66 °C  ch4:  21.61 °C
```

3.6 `closetask`

Closes the task on the *NI USB-6009* and has to be called at the end of the python script to prevent an error.

4 Precision of the temperature sensors

The sensors precision should be below 0.1 °C. This was tested by continuously taking the temperature over a few minutes and plotting the data as a histogram. To eliminate the slow temperature drifts, the data was fourier transformed and all frequency components below 3 mHz were cut. This allowed to fit a well matching gaussian curve to the histogram, for which an example can be found in Fig. 4. A plot of the corresponding temperature data plotted over time can also be found in Fig. 3a and its fourier transform in Fig. 3b. The data was taken with a `sampsize` of 200 and a `samptime` of 100 ms.



(a) Recording (b) Fourier transform

Figure 3: Sensor temperature recorded over 5 minutes and its fourier transform. The green line is the unfiltered data, while the blue line represents the data where low frequencies were cut. The fourier transform of the unfiltered data shows a peak around zero which is cut out. This data was taken with a `sampsize` of 200 and a `samptime` of 100 ms.

The plot with the temperature recording shows the deviation from the mean value for the unfiltered data in green, and the filtered data where low frequencies were cut out in blue. It can be seen, that slow temperature drifts are filtered out, while the fast noise remains, indicating the precision of the sensor. In the histogram plot the fit suggests a gaussian noise with a width of $\sigma = 17.38\,\mathrm{mK}$, meaning that the 5σ interval is smaller than $\pm 0.1\,°\mathrm{C}$.

This can now be done for different `sampsize`s to see how the precision behaves. In Fig. 5 the width/standard deviation σ of the gaussian curve fitted to the histogram is plotted against the `sampsize`.

To investigate the behavior, a function of the form

$$f(x) = a \cdot x^b + c \tag{5}$$

is fitted to the data, while the last data point at a `sampsize` of 500 was not considered

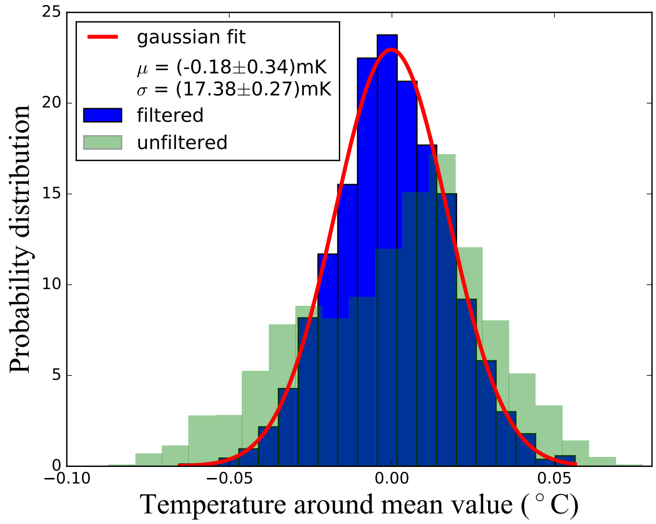


Figure 4: Historgram of the temperature recording. The green bars in the background show the unfiltered data and the blue bars represent the data were low frequencies were cut out. A gaussian curve was fitted to the filtered data.

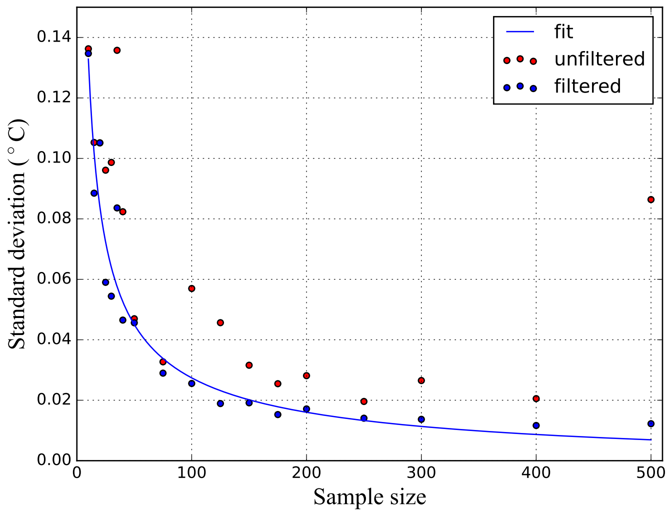


Figure 5: Standard deviation of temperature vs. `sampsize` for a `samptime` of 100 ms.

in the fit. The resulting fit parameters are:

$$\begin{aligned} a &= 0.59(20)\,^{\circ}\mathrm{C} \\ b &= -0.63 \pm 0.16 \\ c &= -0.050(13)\,^{\circ}\mathrm{C}. \end{aligned}$$

This suggests a one over square root of the sample size behavior of the noise, for sample sizes well below 500.

There could not be found any significant dependence of the sensors precision on the `samptime`. It seems that the *NI USB-6009*s sampling rate does not have a large influence on the electronic noise, which means that the `samptime` does not matter as much. It should just be chosen short enough such that temperature drifts do not affect a single measurement, which for a `samptime` below a second should not be the case. A default value of `sampsize` $= 200$ and `samptime` $= 100$ results in a total sampling rate of 12 kHz which still lies well below the maximum and therefore seems adequate.