Python & MicroPython: Programmieren lernen für Einsteiger

Sensoren und Messwerterfassung

Teil 9

Bereits in Teil 5 dieser Serie wurde erläutert, wie man mit Python analoge Messwerte erfassen kann. Dazu wurde der Raspberry Pi mit einem externen Analog-Digital-Wandler ausgestattet. Dies ermöglichte die Messung analoger Strom- und Spannungswerte. Wesentlich interessanter ist jedoch der Einsatz von Sensoren, mit denen nahezu jeder physikalische Wert wie Temperatur, Feuchte, Druck, Beleuchtungsstärke, Bewegung usw. erfasst werden können. In diesem Beitrag soll daher die Auswertung und Verarbeitung von Sensorsignalen mit Python im Fokus stehen. Zudem sollen spezielle Signalverarbeitungstechniken genauer betrachtet werden.

Messwerterfassung mit Sensoren

Der Raspberry Pi ist eine der vielseitigsten Plattformen zur Messwerterfassung mit Sensoren. Durch die Kombination von Python-Programmierung und GPIO-Pins lassen sich zahlreiche physikalische Größen erfassen und verarbeiten.

Sensoren können über verschiedene Schnittstellen mit dem Raspberry Pi verbunden werden:

- **GPIO (General Purpose Input/Output):** direkte digitale Ansteuerung von Sensoren
- **I2C (Inter-Integrated Circuit):** serielles Kommunikationsprotokoll für Sensoren wie das BME280-Modul (Temperatur, Luftdruck, Feuchte)
- **SPI (Serial Peripheral Interface):** Hochgeschwindigkeitsprotokoll für Sensoren wie den MCP3008-AD-Wandler
- **UART (Universal Asynchronous Receiver-Transmitter):** Kommunikation mit seriellen Sensoren oder Modulen wie GPS

Neben diesen universellen Bussystemen haben sich auch noch einige spezielle Schnittstellen etabliert. Diese sind vor allem beim Anschluss von Sensoren an den Raspberry Pi weitverbreitet. Sie sollen daher im Folgenden genauer betrachtet werden. Die universellen Schnittstellen und ihre Anwendungen werden dann in späteren Artikeln ausführlicher erläutert.

Messwerterfassung mit Python

Eine der wichtigsten Spezialschnittstellen ist der sogenannte Eindraht oder „One-Wire-Bus“. Der One-Wire-Bus nutzt ein serielles Kommunikationsprotokoll, das mit nur einer Datenleitung (plus Masse) auskommt. Es wurde von Dallas Semiconductor (heute Maxim Integrated) entwickelt und wird häufig für den Anschluss von Sensoren und Speicherbausteinen verwendet.

Ein bekanntes Beispiel ist der DS18B20-Temperatursensor, der über den One-Wire-Bus mit Mikrocontrollern oder dem Raspberry Pi kommunizieren kann. Das Protokoll ermöglicht den Betrieb mehrerer Geräte an einer einzigen Leitung durch eine eindeutige 64-Bit-Adresse pro Gerät. Die Vorteile des One-Wire-Busses sind:

- Einfache Verdrahtung (nur eine Datenleitung)
- Energieversorgung über die Datenleitung möglich (sogenannte „parasitäre Versorgung“)
- Kostengünstig und effizient für einfache Sensoranwendungen

Eingesetzt wird der One-Wire-Bus vor allem in Temperaturmessungen, Identifikationssystemen und Datenloggern. Der DS18B20-Temperatursensor verfügt über drei Pins (Bild 1):

- **VCC** (3,3 V oder 5 V)
- **GND** (Masse)
- **DQ** (Datenleitung)

Zusätzlich zum Sensor selbst wird noch ein 4,7-kΩ-Pull-up-Widerstand zwischen VCC und DQ benötigt. Bild 2 zeigt den Aufbau.

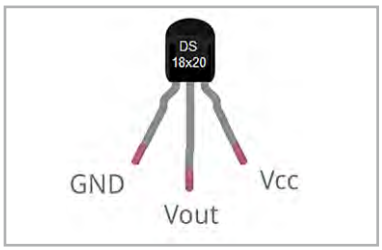

Bild 1: Temperatursensor DS18B20

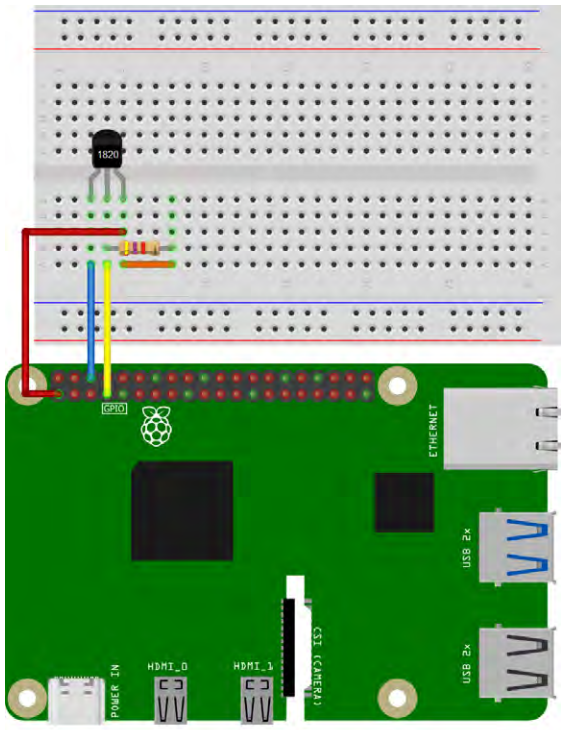

Bild 2: DS18B20-Temperatursensor am Raspberry Pi

One-Wire-Schnittstelle aktivieren

Um die One-Wire-Schnittstelle zu nutzen, muss man auf dem Raspberry Pi die Datei `/boot/config.txt` öffnen und am Ende den Eintrag `dtoverlay=w1-gpio` hinzufügen.

Danach kann das One-Wire-Interface freigeschaltet werden (Bild 3).

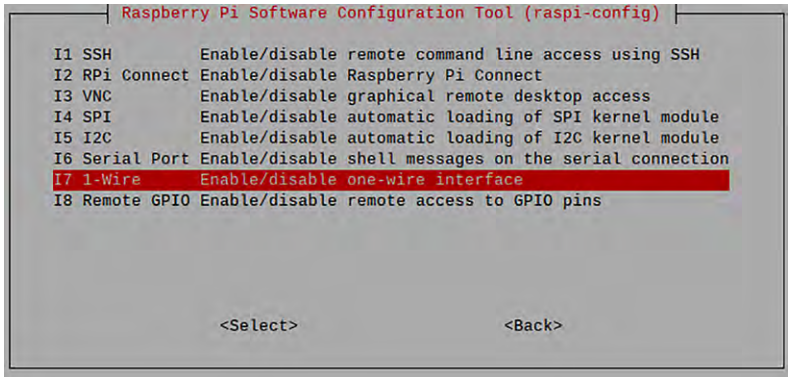

Bild 3: Freischalten des One-Wire-Interfaces

Anschließend sollte der Raspberry Pi neu gestartet werden. Nach dem Neustart sind die erforderlichen Module zu laden:

```
sudo modprobe w1-gpio
sudo modprobe w1-therm
```

Dann kann man das Verzeichnis des Sensors prüfen:

```
cd /sys/bus/w1/devices/
```

Hier sollte nach Eingeben des Befehls eine Datei zu sehen sein, z. B. 10-XXXXXXXXXXXX. Damit kann man nun die Temperatur auslesen:

```
cat /sys/bus/w1/devices/10-XXXXXXXXXXXX/w1_slave
```

In der Anweisung muss natürlich der oben ermittelte Dateiname verwendet werden. Die Ausgabe enthält eine Zeile mit `t=XXXXX`, wobei `XXXXX` die Temperatur in Milligrad Celsius (1/1000 °C) ist. So bedeutet z. B. t=21125 einen Temperaturwert von 21,125 °C. Bild 4 zeigt die vollständige Befehlsfolge in der Shell.

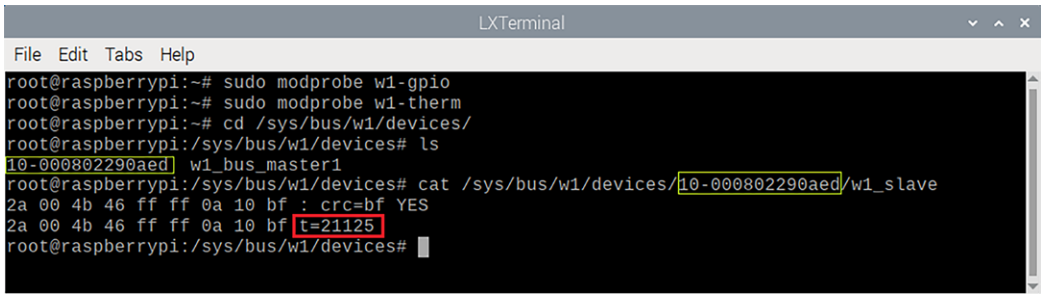

Bild 4: Abfrage des DS18x20 in der Shell

Python-Code für automatische Messung

Das Abfragen des Sensors in der Shell liefert zwar bereits einen codierten Temperaturwert, jedoch stellt dies im Allgemeinen nicht das gewünschte Endergebnis dar. Die universelle Programmiersprache Python kann natürlich auch hier Abhilfe schaffen. Ein einfaches Python-Skript zum Auslesen der Temperatur kann folgendermaßen aussehen:

```
import os
import glob
import time

# Sensorverzeichnis finden
base_dir = "/sys/bus/w1/devices/"
device_folder = glob.glob(base_dir + "10*")[0]
device_file = device_folder + "/w1_slave"

def read_temp_raw():
    with open(device_file, "r") as f:
        return f.readlines()

def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != "YES":
        time.sleep(0.2)
        lines = read_temp_raw()
    temp_output = lines[1].split("t=")
    if len(temp_output) > 1:
        temp_c = float(temp_output[1]) / 1000.0
        return temp_c

while True:
    print("Temperatur: {:.2f}°C".format(read_temp()))
    time.sleep(1)
```

Bild 5 zeigt die Ausgabe in der Thonny-Shell. Zusätzlich wird der Temperaturverlauf im Plotter grafisch dargestellt.

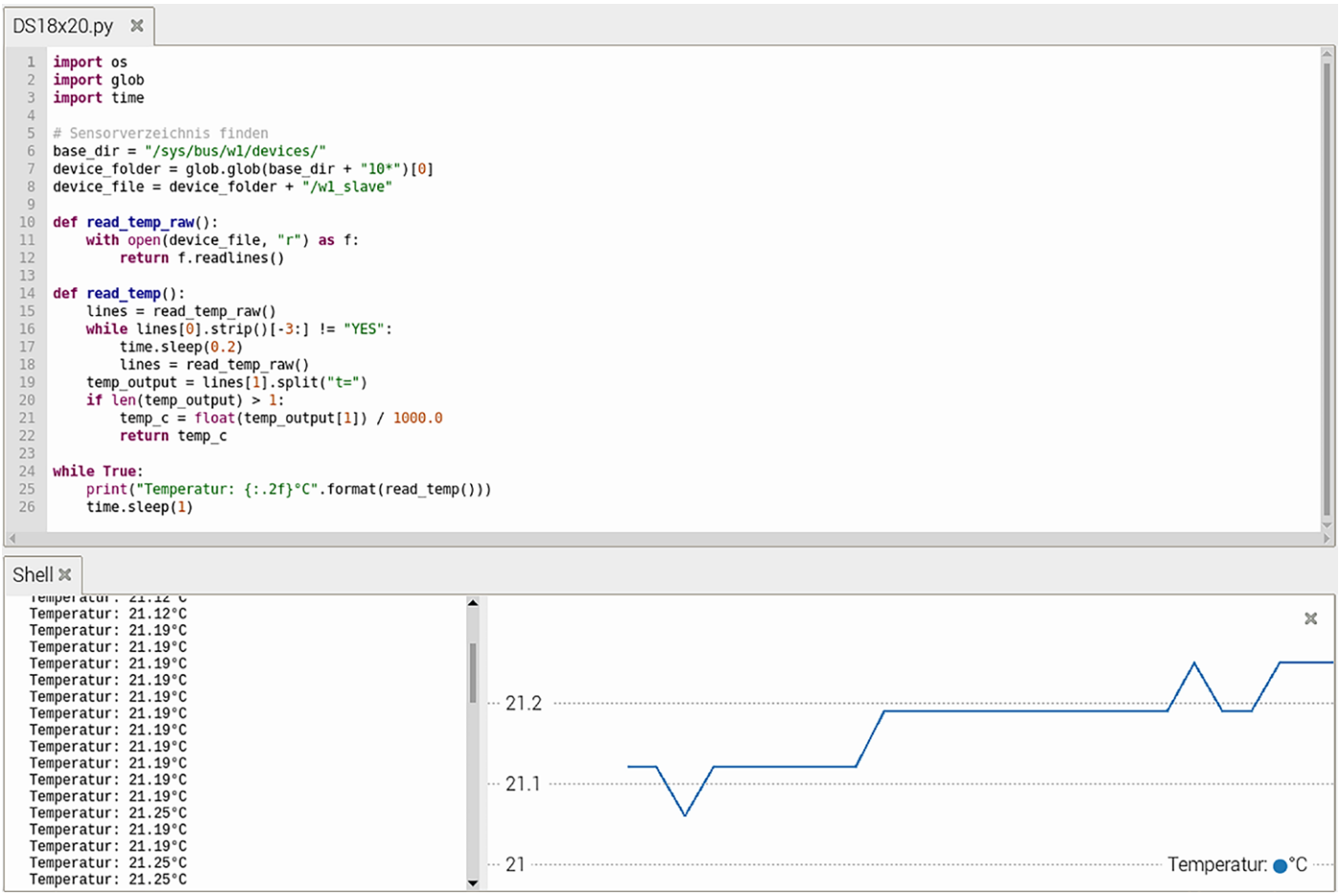

Bild 5: Auslesen der aktuellen Temperatur mit Python

Das Programm liest die Werte von einem Temperatursensor, in diesem Falle dem DS18B20, über den One-Wire-Bus aus. Die Temperaturwerte werden aus der oben vorgestellten Datei im Systemverzeichnis `/sys/bus/w1/devices/` entnommen. Zunächst werden dazu die erforderlichen Libraries eingebunden:

```
import os
import glob
import time
```

`os` ermöglicht den Zugriff auf Betriebssystemfunktionen, `glob` wird verwendet, um das Verzeichnis des Sensors automatisch zu finden. Das Basisverzeichnis selbst wird direkt angegeben. Dort wird die erforderliche Datei automatisch gesucht:

```
base_dir = "/sys/bus/w1/devices/"
device_folder = glob.glob(base_dir + "10*")[0]
device_file = device_folder + "/w1_slave"
```

Dabei ist `base_dir` das Verzeichnis, in dem sich die One-Wire-Geräte befinden. Die Anweisung

```
glob.glob(base_dir + "10*")
```

sucht nach einem Unterordner, dessen Name mit "10" beginnt (dies ist die Standardkennung für DS18B20-Sensoren).
Die Anweisung

```
device_folder[0]
```

nimmt das erste gefundene Verzeichnis. `Device_file` ist der Dateipfad zur `Datei w1_slave`, welche die Sensordaten enthält. Dann werden die Rohdaten aus der Datei gelesen:

```
def read_temp_raw():
    with open(device_file, "r") as f:
        return f.readlines()
```

„Open" öffnet die Datei `w1_slave` und liest ihren Inhalt aus. Die Datei enthält zwei Zeilen:

- Die erste Zeile zeigt, ob der Wert gültig ist ("YES" bedeutet gültig).
- Die zweite Zeile enthält den Temperaturwert in tausendstel Grad Celsius (t=XXXXX).

Anschließend kann man den Temperaturwert auslesen und verarbeiten:

```
def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != "YES":
        time.sleep(0.2)
        lines = read_temp_raw()
```

Der Befehl

```
Read_temp_raw()
```

liest die Datei `w1_slave` ein.

```
While lines[0].strip()[-3:] != "YES"
```

prüft, ob die erste Zeile mit "YES" endet. Falls nicht, wartet das Programm 0,2 Sekunden und liest die Datei erneut. Dies stellt sicher, dass der Sensorwert gültig ist.

Nun kann man die Temperatur extrahieren und umrechnen:

```
temp_output = lines[1].split("t=")
if len(temp_output) > 1:
    temp_c = float(temp_output[1]) / 1000.0
    return temp_c
```

In der Zeile

```
lines[1].split("t=")
```

wird die zweite Zeile am Marker "t=" aufgetrennt. Der Temperaturwert findet sich nach diesem Marker. Über

```
float(temp_output[1]) / 1000.0
```

wird der Wert von tausendstel Grad Celsius in Grad Celsius konvertiert. Über

```
Return temp_c
```

wird anschließend der Temperaturwert zurückgegeben. Die Ausgabe der Temperatur erfolgt in der üblichen Endlosschleife.

Klima und Luftfeuchte

Allgemein versteht man unter „Feuchte" das Vorhandensein von Wasser in einem Gas oder Feststoff. Von besonderer Bedeutung ist die relative Luftfeuchtigkeit, sie wird als prozentuales Verhältnis aus dem aktuellen Wassergehalt zur maximal möglichen absoluten Feuchte, der sogenannten Sättigungsfeuchte angegeben.

Neben der Temperatur ist die relative Feuchte der wichtigste Parameter, wenn es um die Beurteilung eines Raumklimas geht. Selbst wenn in einem Wohnraum eine angenehme Temperatur von z. B. 21 °C herrscht, fühlt man sich bei zu hoher oder zu geringer Luftfeuchte nicht wohl.

Darüber hinaus spielt die Luftfeuchtigkeit auch in der Bauphysik eine wichtige Rolle. Zu hohe Luftfeuchtigkeit, z. B. in Kellerräumen, führt zu Fäulnis und Schimmelbildung und kann so ernst zu nehmende Bauschäden verursachen. Andererseits kann zu geringe Feuchte zur Schädigung von Zimmerpflanzen und zu allgemein mangelndem Wohlbefinden führen. Bild 6 zeigt Schäden in einem Kellerraum, die durch dauerhaft zu hohe Luftfeuchtewerte entstanden sind. Diese hätten durch den Einsatz einer Feuchteüberwachung mit dem Raspberry Pi zusammen mit einem DHT11-Sensor frühzeitig erkannt werden können.

Der DHT11 ist ein Temperatur- und Luftfeuchtigkeitssensor, der über eine digitale Schnittstelle mit dem Raspberry Pi 5 verbunden werden kann. Er verfügt genau wie der DS18x20 über ein einfaches digitales Interface. Der Sensor hat prinzipiell vier Pins. Wenn er auf einem Modul montiert ist, werden davon meist jedoch nur drei herausgeführt, da der vierte Pin ohne Funktion („not connected") ist:

- Pin 1: VCC → 3,3 V oder 5 V vom Raspberry Pi
- Pin 2: DATA → ein GPIO-Pin des Raspberry Pi (z. B. GPIO4, Pin 7)
- Pin 3: (NC) → unbenutzt
- Pin 4: GND → GND vom Raspberry Pi

Wird eine DHT11-Modul-Version verwendet, hat der Sensor also nur drei Pins. In diesem Fall entspricht Pin 3 dem vierten Pin (GND) in der obigen Liste.

Bild 6: Feuchteschäden in einem Kellerraum – früh erkannt mit DHT11 und Python

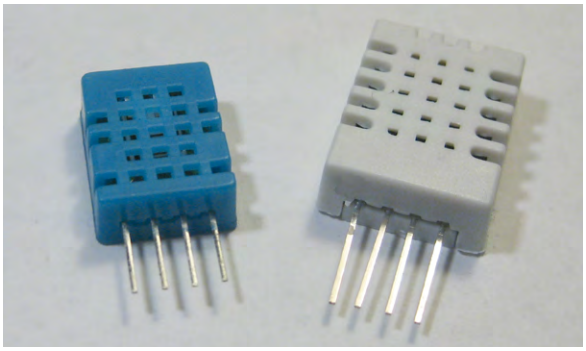

Bild 7: DHT11 und DHT22

Neben dem DHT11 existiert auch eine verbesserte Version, der DHT22 (siehe Bild 7). Dieser verfügt über eine höhere Genauigkeit und einen erweiterten Messbereich. Für die meisten Anwendungen sollte allerdings der DHT11 ausreichend sein (Bild 8).

Hinweis: Falls der Sensor direkt verwendet wird, ist ein Pull-up-Widerstand von 10 kΩ zwischen DATA und VCC empfehlenswert. Damit wird ein dauerhaft stabiles Signal gewährleistet. Bei Modulen ist dieser Widerstand meist schon intern vorhanden.

Bild 9 zeigt den Anschluss des Sensors an den Raspberry Pi.

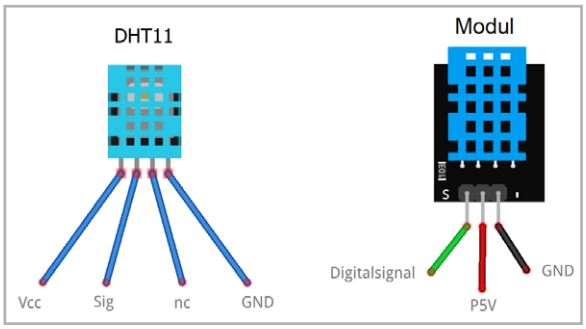

Bild 8: Pinbelegung des DHT11 und eines DHT11-Moduls

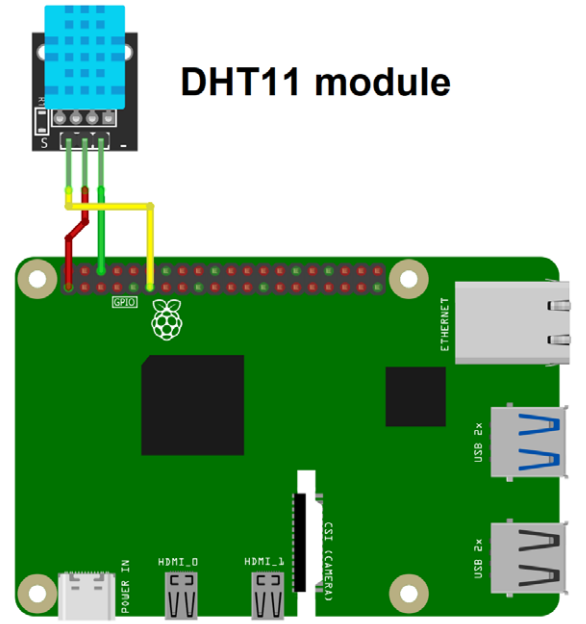

Bild 9: Anschluss des DHT11 an den Raspberry Pi

DHT11 am Raspberry Pi

Das digitale Ausgangssignal des DHT11-Sensors macht den Einsatz am Raspberry Pi besonders einfach. Man kann den Sensor sogar bis zu ca. 10 Meter entfernt von einem Raspberry Pi anschließen, ohne dass Signalstörungen auftreten. Dadurch kann die Luftfeuchtigkeit und Temperatur an Orten gemessen werden, an denen z. B. keine Stromversorgung für den Raspberry Pi verfügbar ist.

Nachdem der DHT11 mit dem Raspberry Pi verbunden ist, kann mithilfe eines Python-Skripts die aktuelle Temperatur und Luftfeuchtigkeit ausgelesen werden. Dazu kann eine DHT-Bibliothek verwendet werden, die den Umgang mit diesem Sensor stark vereinfacht.

Zu Beginn sollte man sicherstellen, dass der Raspberry Pi auf dem neuesten Stand ist:

```
sudo apt update
sudo apt upgrade -y
```

Dann werden die benötigten Pakete (nach)installiert:

```
sudo apt install python3 python3-pip python3-venv
```

Mit der letzten Anweisung wird die Verwendung von „Virtuellen Umgebungen“ (siehe nächste Seite) auf dem Raspberry Pi ermöglicht.

Zunächst wird ein Projektverzeichnis erstellt, in dem die Python-Umgebung und das Skript gespeichert werden:

```
mkdir ~/dht11
cd ~/dht11
```

Dort wird die virtuelle Python-Umgebung erstellt

```
python3 -m venv env
```

und aktiviert

```
source env/bin/activate
```

Dieser Befehl muss bei jeder Verwendung des Skripts erneut ausgeführt werden.

Dann kann man die DHT-Bibliothek installieren:

```
python3 -m pip install adafruit-circuitpython-dht
```

Python-Skript zum Auslesen des DHT11-Sensors

Das Programm zum Auslesen des DHT11-Sensors kann dann wie folgt aussehen:

```
import time, board, adafruit_dht

dhtDevice = adafruit_dht.DHT11(board.D17)

try:
    while True:
        try:
            temperature_c = dhtDevice.temperature
            humidity = dhtDevice.humidity
            print("Temp: {:.1f}°C Humidity: {}%".
format(temperature_c, humidity))
            time.sleep(2.0)
        except RuntimeError as error:
            print(error.args[0])
            time.sleep(2.0)
            continue

except KeyboardInterrupt as error:
    print("CTRL-C pressed. Deiniting everything...")
    dhtDevice.exit()
```

Bevor das Programm in Thonny gestartet werden kann, muss dort die virtuelle Umgebung aktiviert werden (Bild 10).

Wozu braucht man beim Raspberry Pi „Virtuelle Umgebungen“?

Virtuelle Umgebungen beim Raspberry Pi (und generell in Linux und Python) sind nützlich, weil sie helfen, Abhängigkeiten sauber zu verwalten:

1. Vermeidung von Abhängigkeitskonflikten
Beim Entwickeln von Projekten mit Python benötigt man oft verschiedene Bibliotheken. Manche Projekte erfordern unterschiedliche Versionen derselben Bibliothek, was zu Konflikten führen kann. Eine virtuelle Umgebung isoliert die Abhängigkeiten jedes Projekts, sodass sie sich nicht gegenseitig beeinflussen.

2. Schutz des Systems
Wenn man Bibliotheken global installiert (sudo pip install ...), können wichtige Systemkomponenten überschrieben oder beschädigt werden. Eine virtuelle Umgebung stellt sicher, dass Änderungen innerhalb des Projekts bleiben und das Betriebssystem nicht angetastet wird.

3. Portabilität und Reproduzierbarkeit
Mit einer virtuellen Umgebung kann man eine Liste aller installierten Pakete (requirements.txt) speichern und das Set-up auf einem anderen System (z. B. einem anderen Raspberry Pi) identisch wiederherstellen. Das erleichtert die Zusammenarbeit und das erneute Einrichten eines Projekts.

4. Verschiedene Python-Versionen nutzen
Manchmal benötigt ein Projekt eine spezielle Python-Version. Eine virtuelle Umgebung ermöglicht es, unterschiedliche Versionen parallel zu nutzen, ohne das gesamte System umzustellen.

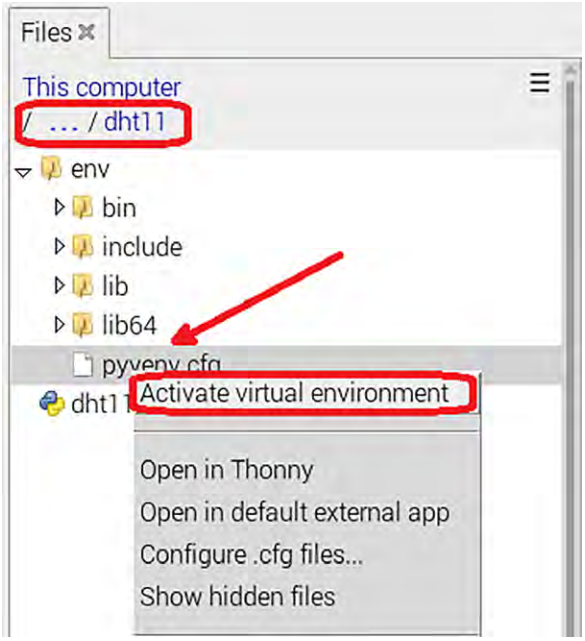

Bild 10: Aktivieren der „Virtuellen Umgebung“ in Thonny

Danach kann das Programm gestartet werden und die gewünschten Temperatur- und Luftfeuchtewerte werden angezeigt (Bild 11).

Zum Testen kann man den Sensor beispielsweise anhauchen. Dann sollten die Luftfeuchtewerte temporär deutlich ansteigen und anschließend wieder abfallen (siehe Bild 11). Die Temperatur dagegen steigt und fällt in diesem Fall nur minimal. Beim vorsichtigen Anblasen mit einem Haarföhn dagegen steigt die Temperatur stark an, während die Feuchtewerte nahezu konstant bleiben.

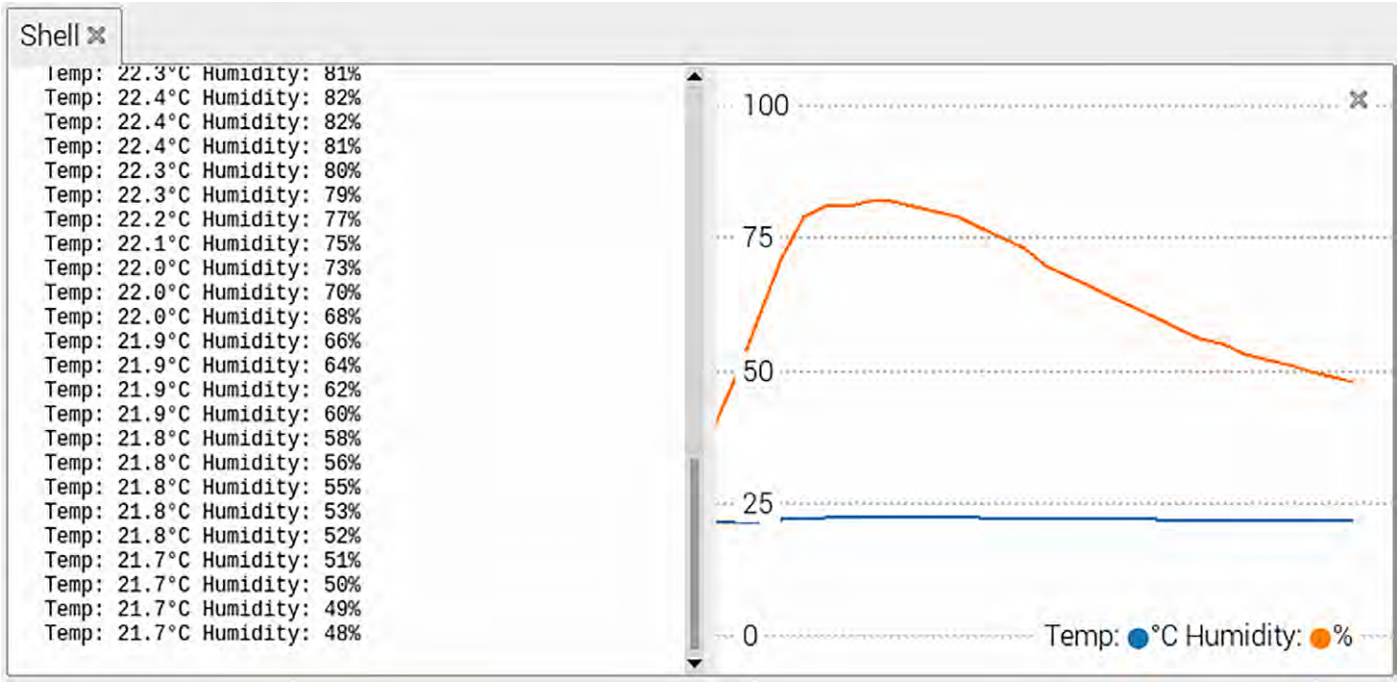

Bild 11: Messwerte in Thonny

Mittelung, Filterung und Signalkonditionierung

Bei der Arbeit mit Sensoren erhält man oft verrauschte, schlecht skalierte oder mit hochfrequenten Einstrahlungen behaftete Rohsignale. In der Sensordatenerfassung und Signalverarbeitung sind daher Mittelung, Filterung und Signalkonditionierung grundlegende Techniken, um Rauschen zu reduzieren, nützliche Informationen hervorzuheben und Signale für die weitere Analyse vorzubereiten. Mit Python ist man für diese Techniken bestens gerüstet.

Mittelung dient dazu, zufälliges Rauschen zu reduzieren, indem mehrere Messungen kombiniert werden. Ein Beispiel ist der gleitende Durchschnitt („Moving Average"), bei dem der Mittelwert über ein Fenster von Datenpunkten berechnet wird, z. B. zur

- Glättung von Temperaturschwankungen
- Mittelung von Helligkeitswerten zur Ermittlung von Sonneneinstrahlungswerten für eine Solaranlage
- Mittelung von Feuchtedaten zur Steuerung einer Bewässerungsanlage

Filter entfernen gezielt bestimmte Frequenzanteile eines Signals. Hierzu können verschiedene Filtertypen eingesetzt werden:

- Tiefpassfilter (Low-pass): Lässt niedrige Frequenzen durch, filtert hochfrequentes Rauschen heraus
- Hochpassfilter (High-pass): Entfernt langsame Trends und belässt schnelle Schwankungen
- Bandpassfilter: Lässt nur einen bestimmten Frequenzbereich passieren

Wichtige Anwendungen sind hier:

- Rauschunterdrückung in Audiosignalen
- Glättung von rauschbehafteten Sensordaten
- Entfernen unerwünschter Signalfrequenzen in drahtlosen Übertragungssystemen

Bei der Signalkonditionierung (Signal Conditioning) werden Rohsignale so angepasst, dass sie für die weitere Verarbeitung geeignet sind. Dazu gehören:

- Verstärkung zur Erhöhung der Signalstärke
- Dämpfung: Reduzierung der Signalstärke, z. B. um Übersteuerungen zu vermeiden
- Konvertierung: Änderung des Signalformats (z. B. analog zu digital, binäre Codierung etc.)
- Offset-Korrektur: Entfernt systematische Fehler im Signal

Beispiele:

- Anpassung eines Sensorsignals, bevor es von einem Mikrocontroller weiterverarbeitet wird
- Normierung von Spannungssignalen bei Analog-Digital-Wandlern

Eine der einfachsten Methoden ist das gleitende Mittel. Bild 12 zeigt eine Visualisierung von Daten zusammen mit einem gleitenden Durchschnitt.

Das zugehörige Programm (`Running _ average.py`) sieht so aus:

```
import numpy as np
import matplotlib.pyplot as plt

N=100

# Erstelle die x-Werte
x = np.linspace(0, 10, N)

# Ersetze die Sinuswerte durch Zufallszahlen
y = np.sin(x)+0.5*np.random.rand(N)  # 50 Zufallszahlen zwischen 0 und 1

# Definiere die Fenstergröße für den gleitenden Durchschnitt
window = 9

# Berechne den gleitenden Durchschnitt
average_y = []
for ind in range(len(y) - window + 1):
    average_y.append(np.mean(y[ind:ind+window]))

# Ergänze die NaN-Werte am Anfang, damit die Längen übereinstimmen
for ind in range(window - 1):
    average_y.insert(0, np.nan)

# Plotten der Daten
plt.figure(figsize=(10, 5))
plt.plot(x, y, 'y.-', label='Originaldaten')
plt.plot(x, average_y, 'r.-', label='Running average')
plt.grid(linestyle=':')
plt.legend()
plt.show()
```

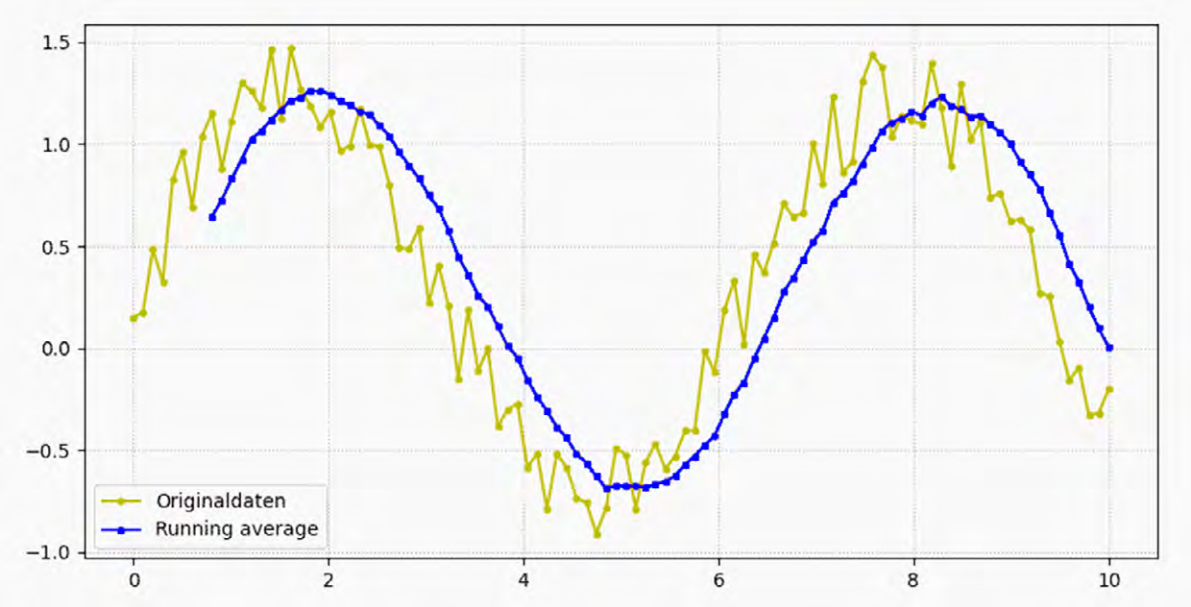

Bild 12: Daten mit einem gleitenden Durchschnitt

Das Python-Programm demonstriert die Anwendung des gleitenden Durchschnitts zur Glättung verrauschter Daten. Es beginnt mit der Erzeugung einer simulierten Datenreihe, die aus einer Sinuswelle mit hinzugefügtem Zufallsrauschen besteht:

```
y = np.sin(x)+0.5*np.random.rand(N)
```

Dieses Rauschen soll reale Messungen nachbilden, die oft von unerwünschten Schwankungen überlagert sind.

Der Kern des Programms ist die Berechnung des gleitenden Durchschnitts. Hierbei wird ein „Fenster" definierter Größe (in diesem Fall 9 Datenpunkte) über die Datenreihe geschoben. Für jede Position des Fensters wird der Mittelwert der darin enthaltenen Werte berechnet. Diese Mittelwerte bilden die geglättete Datenreihe:

```
average_y.append(np.mean(y[ind:ind+window]))
```

Der gleitende Durchschnitt dient dazu, kurzfristige Schwankungen und Ausreißer in den Daten zu reduzieren und den zugrunde liegenden Trend hervorzuheben.

Dieses Vorgehen ist neben der Auswertung von Sensordaten auch nützlich bei der Analyse von Zeitreihen, wie beispielsweise Aktienkursen oder Wetterdaten, bei denen kurzfristige Schwankungen die Erkennung langfristiger Trends erschweren können.

Da der gleitende Durchschnitt für die ersten Datenpunkte nicht berechnet werden kann (da noch nicht genügend vorhergehende Werte vorhanden sind), werden am Anfang der geglätteten Datenreihe NaN-Werte (Not a Number) eingefügt:

```
average_y.insert(0, np.nan)
```

Abschließend werden die Originaldaten und der gleitende Durchschnitt in einem Diagramm dargestellt. Dies ermöglicht einen visuellen Vergleich der verrauschten Originaldaten mit der geglätteten Version. Das Diagramm (Bild 12) zeigt, wie der gleitende Durchschnitt das Rauschen reduziert und den zugrunde liegenden Sinuswellen-Trend deutlicher sichtbar macht.

Praktische Anwendung des gleitenden Durchschnitts

Neben dem Herausfiltern von Rauschen aus Zeitreihendaten kann die Mittelung dabei helfen, saisonale Zyklen in den Daten sichtbar zu machen. Diese Methode wird in vielen Bereichen eingesetzt – von der Physik über Umweltwissenschaften bis hin zur Finanzwelt. Um einen gleitenden Durchschnitt zu berechnen, ist das folgende Vorgehen empfehlenswert:

1. Definieren einer Fenstergröße (z. B. 2 bis n–1, wobei n>2 die Anzahl der Datenpunkte im Fenster ist)
2. Berechnen des Durchschnitts innerhalb dieses Fensters
3. Verschieben des Fensters um einen Datenpunkt und Wiederholen des Vorgangs, bis das Ende der Datenreihe erreicht ist

Das folgende Programm zeigt eine Anwendung für den DHT11-Sensor (`DHT11 _ averaging.py`):

```
import time
import adafruit_dht
import board
import numpy as np
import matplotlib.pyplot as plt

# Initialisierung des DHT22 Sensors
sensor = adafruit_dht.DHT11(board.D17)  # D4 ist der GPIO-Pin

# Listen für die Daten
temperature_readings = []
window_size = 5  # Fenstergröße für die Mittelung
smoothed_readings = []

try:
  # 50 Messungen durchführen

  for i in range(50):
    try:
      # Temperaturdaten auslesen
      temp_c = sensor.temperature
      temperature_readings.append(temp_c)

      # Berechne den gleitenden Durchschnitt, wenn genug Daten vorhanden sind
      if len(temperature_readings) >= window_size:
        window = temperature_readings[-window_size:]
        average = np.mean(window)
        smoothed_readings.append(average)
      else:
        smoothed_readings.append(np.nan)  # Nicht genug Daten für Mittelung

     print(f"{temp_c:.2f}°C, Geglättet: {smoothed_readings[-1]:.2f}°C")

     time.sleep(2)  # 2 Sekunden warten zwischen den Messungen

    except RuntimeError as error:
      # Fehler beim Lesen ignorieren
      print(f"Lesefehler: {error}")
```

```
    time.sleep(2)  # 2 Sekunden warten zwischen den Messungen

    # Plot der Rohdaten und geglätteten Daten
    plt.figure(figsize=(10, 5))
    plt.plot(temperature_readings, 'k.-', label='Rohdaten')
    plt.plot(smoothed_readings, 'r.-', label='Gleitender Durchschnitt')
    plt.xlabel('Messung')
    plt.ylabel('Temperatur (°C)')
    plt.title('Temperaturmessung mit gleitendem Durchschnitt')
    plt.legend()
    plt.grid(linestyle=':')
    plt.show()

except KeyboardInterrupt as error:
    print("CTRL-C pressed - Ending...")
    sensor.exit()
```

In Bild 13 sind Originaldaten und der geglättete Messwertverlauf zu sehen.

Das Programm liest zunächst Temperaturdaten vom DHT11-Sensor und glättet sie dann mit einem gleitenden Durchschnitts-Verfahren. Dazu wird zunächst der Sensor am GPIO-Pin D17 initialisiert, anschließend werden die Messwerte in einer Liste abgespeichert. Nach jeder Messung wird geprüft, ob genügend Daten für den gleitenden Durchschnitt vorhanden sind. Ist dies der Fall, wird der Mittelwert der letzten 5 Messungen berechnet und in einer weiteren Liste gespeichert. Die gemessenen und geglätteten Temperaturen werden in der Konsole ausgegeben und nach 50 Messungen in einem Diagramm dargestellt. Das Diagramm zeigt die Rohdaten und den geglätteten Durchschnitt, um den Effekt der Glättung zu verdeutlichen. Fehler beim Lesen des Sensors werden dabei abgefangen und ignoriert:

```
except RuntimeError as error:
      # Fehler beim Lesen ignorieren
      print(f"Lesefehler: {error}")
```

Beim Drücken von Strg+C wird das Programm beendet.

Bei der praktischen Anwendung sollte man beachten, dass mit zunehmender Fenstergröße die Kurve immer „glatter" wird. Allerdings treten dann jedoch Verzögerungen auf (Peaks und Täler verschieben sich nach hinten, siehe Bild 13). Es ist daher häufig sinnvoll, verschiedene Fenstergrößen zu testen, um die beste Balance für gegebene Daten zu finden.

Filterung von Messwerten

In der Signalverarbeitung werden sowohl Hoch-, Tief- als auch Bandpassfilter eingesetzt, um bestimmte Frequenzbereiche eines Signals gezielt durchzulassen oder zu dämpfen. Diese Methoden helfen dabei, Störungen zu entfernen oder relevante Informationen aus einem Signal herauszufiltern.

Ein Tiefpassfilter lässt niedrige Frequenzen passieren und dämpft hohe Frequenzen. Er wird häufig verwendet, um Rauschen (oft hochfrequent) zu reduzieren oder Signale zu glätten.
Beispiele:

- Glättung von Messwerten bei einem Optosensor, um schnelle, unerwünschte Schwankungen zu eliminieren
- Reduktion von Störungen bei Audiosignalen

Ein Hochpassfilter dagegen lässt hohe Frequenzen passieren und dämpft niedrige Frequenzen. Er wird verwendet, um langsame Trends oder Driften aus einem Signal zu entfernen.
Beispiele:

- Erkennung schneller Signaländerungen, z. B. bei Bewegungssensoren
- Entfernen von Gleichstromanteilen in Audiosignalen
- Reduktion von langsamen Signaldriften, z. B. bei thermischen Einflüssen

Da Tiefpässe ein ähnliches Verhalten wie Mittelungen zeigen, soll hier exemplarisch eine Hochpassfilterung vorgestellt werden. Ein Python-Programm zur Hochpassfilterung kann wie folgt aussehen:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, lfilter
```

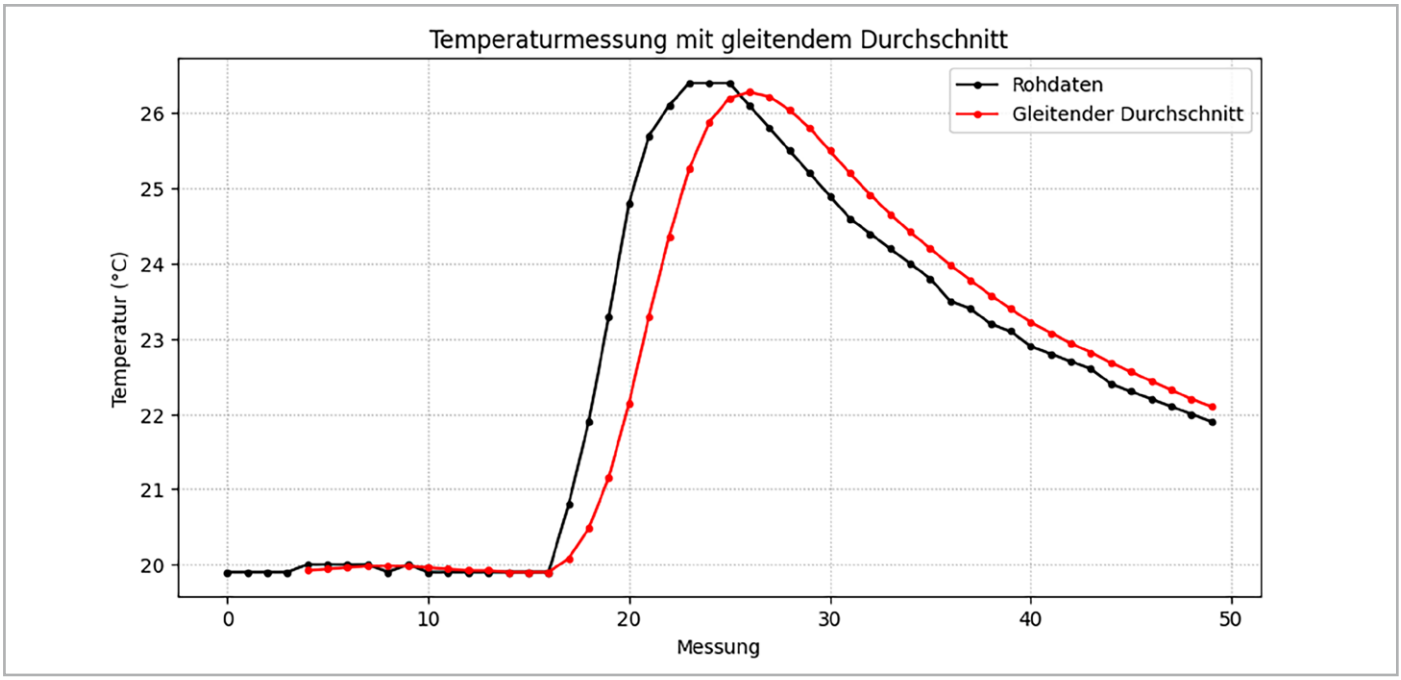

Bild 13: Originaldaten und geglätteter Messwertverlauf

```
# Hochpassfilter-Definition
def butter_highpass(cutoff, fs, order=5):
    nyquist = 0.5 * fs  # Nyquist-Frequenz
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='high', analog=False)
    return b, a

def highpass_filter(data, cutoff, fs, order=5):
    b, a = butter_highpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y

# Filtereinstellungen
fs = 100.0  # Abtastrate (Hz)
cutoff = 10.0  # Grenzfrequenz (50 Hz)
order = 2  # Filterordnung

# Simuliere Optosensordaten mit Störungen im niedrigen Frequenzbereich
t = np.linspace(0, 5, int(fs * 5))  # Zeitachse für 5 Sekunden
low_freq_noise = np.sin(2 * np.pi * 5 * t)  # Niederfrequentes Rauschen (5 Hz)
high_freq_signal = np.sin(2 * np.pi * 100 * t)  # Hochfrequentes Signal (100 Hz)
sensor_data = low_freq_noise + high_freq_signal  # Signal + niederfrequentes Rauschen

# Hochpassfilter anwenden
filtered_data = highpass_filter(sensor_data, cutoff, fs, order)

# Daten visualisieren
plt.figure(figsize=(12, 6))
plt.plot(t, sensor_data, label='Rohdaten (mit niederfrequentem Rauschen)', color='red', alpha=0.6)
plt.plot(t, 3*filtered_data, label='Gefilterte Daten (Hochpass 50 Hz)', color='blue', linewidth=2)
plt.xlabel('Zeit (s)')
plt.ylabel('Signalwert')
plt.title('50-Hz-Hochpassfilter für Sensor-Daten')
plt.legend()
plt.grid(True)
plt.show()
```

In Bild 14 erkennt man, dass niederfrequente Signalanteile praktisch vollständig herausgefiltert werden, während die höherfrequenten Signale erhalten bleiben.

Das Programm simuliert und filtert Optosensordaten, die durch niederfrequente Störungen beeinflusst sind. Es erzeugt ein synthetisches Signal, das aus einem hochfrequenten Nutzsignal und einem niederfrequenten „Rauschen" besteht.

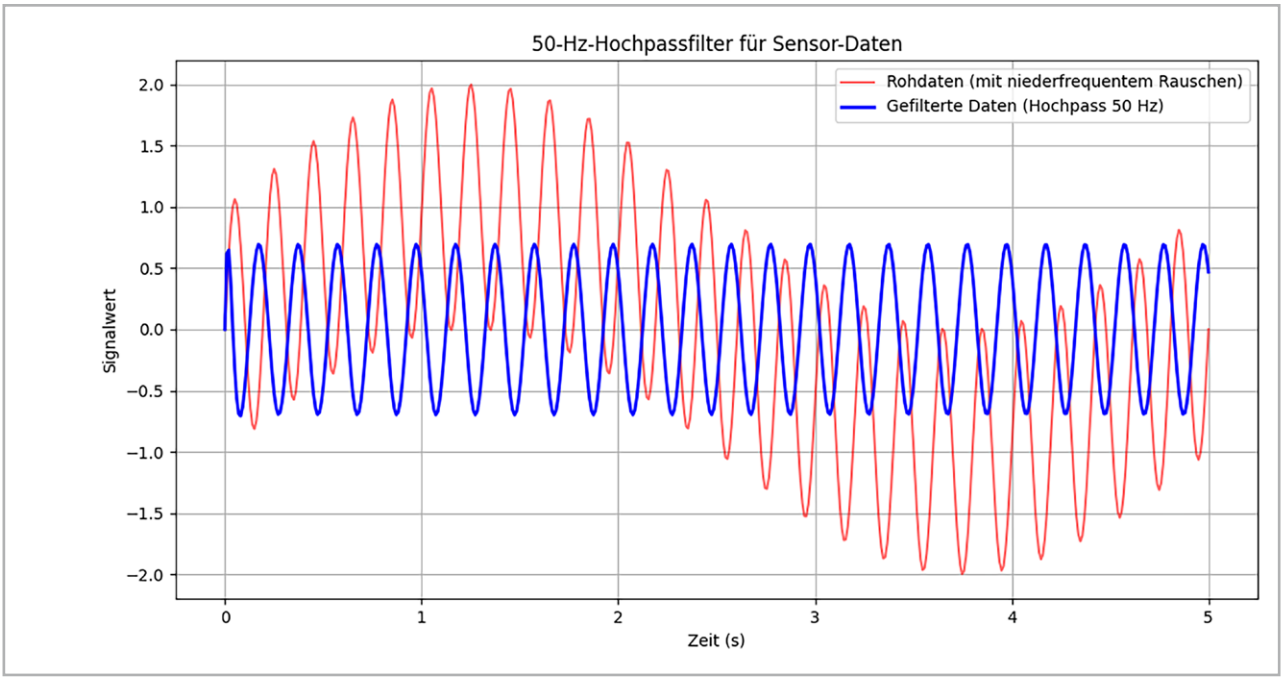

Bild 14: Originaldaten und geglätteter Messwertverlauf

Anschließend wird ein Hochpassfilter angewendet, um das niederfrequente Rauschen zu entfernen. Die Funktionsweise basiert auf einem Filter, dessen Grenzfrequenz und Ordnung einstellbar sind:
- fs = 100.0 # Abtastrate (Hz)
- cutoff = 10.0 # Grenzfrequenz (50 Hz)
- order = 2 # Filterordnung

Das gefilterte Signal wird anschließend zusammen mit den Rohdaten in einem Diagramm dargestellt, um die Wirksamkeit des Filters zu demonstrieren.

Signalkonditionierung

Signalkonditionierung bezeichnet die Verarbeitung und Anpassung von Signalen, um sie für die weitere Analyse, Messung oder Steuerung nutzbar zu machen. Dabei werden die rohen Ausgangssignale eines Sensors so verändert, dass sie den Anforderungen des Messsystems entsprechen – zum Beispiel eines Raspberry Pi 5 oder eines Mikrocontrollers, wie dem Pi Pico.

Die Ziele der Signalkonditionierung sind:
- Anpassung der Signalstärke durch Verstärken oder Abschwächen des Signals
- Skalierung von Messwerten auf bestimme Wertebereiche
- Linearisierung zur Umwandlung eines nicht-linearen Signals in ein lineares Verhältnis
- Trennung des Signals, um Störungen oder Rückkopplungen zu vermeiden („Isolierung")
- Anpassung des Signaltyps, z. B. von A/D-Wandlerwerten zu realen Temperaturen oder Helligkeiten etc.

Häufig auftretende Anwendung sind z. B.:
- Bei einem Optosensor am Raspberry Pi kann die Signalkonditionierung notwendig sein, um das Lichtsignal in ein klares, digitales Signal umzuwandeln.
- In Audiosystemen wird die Signalkonditionierung genutzt, um Störgeräusche herauszufiltern und die Lautstärke anzupassen.
- In industriellen Anwendungen werden Messsignale von Temperatur- oder Drucksensoren verstärkt und gefiltert, bevor sie verarbeitet werden.

Das folgende Programm führt eine Offset-Korrektur, Filterung, Zentrierung auf die Nulllinie, Verstärkung und Normalisierung durch. Dabei werden lediglich „matplotlib" und „NumPy" als Bibliotheken verwendet.

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Simuliertes Sensorsignal (Sinuswelle + Rauschen)
np.random.seed(0)  # Für reproduzierbare Ergebnisse
time = np.linspace(0, 1, 500)  # 1 Sekunde mit 500 Messpunkten
signal = 2 * np.sin(2 * np.pi * 5 * time)  # Reines Sinussignal mit 5 Hz
noise = 0.5 * np.random.randn(500)  # Zufälliges Rauschen
raw_signal = signal + noise

# 2. Offset zum ursprünglichen Signal hinzufügen
offset = 3.0  # Konstanter Offset
offset_signal = raw_signal + offset

# 3. Verstärkung des Signals
gain = 2.0
amplified_signal = offset_signal * gain

# 4. Tiefpassfilterung (gleitender Mittelwert)
window_size = 30
filtered_signal = np.convolve(amplified_signal, np.ones(window_size)/window_size, mode='valid')

# 5. Normalisierung des Signals (Skalierung zwischen 0 und 1)
normalized_signal = (filtered_signal - np.min(filtered_signal)) / (np.max(filtered_signal) - np.min(filtered_signal))

# 6. Zentrierung auf die Nulllinie (Mittelwert abziehen)
centered_signal = normalized_signal - np.mean(normalized_signal)

# 7. Weitere Verstärkung des konditionierten Signals (20-fach)
final_gain = 20.0
processed_signal = centered_signal * final_gain

# 8. Visualisierung
plt.figure(figsize=(12, 6))
# plt.plot(time, raw_signal, label="Rohsignal mit Rauschen", alpha=0.5)
```

```
plt.plot(time, offset_signal, color='red', alpha=0.7)
plt.plot(time[:len(processed_signal)], processed_signal, color='blue', linewidth=2)
plt.xlabel('Zeit (s)')
plt.ylabel('Signalstärke')
plt.title('Signalkonditionierung: Offset, Zentrierung, Verstärkung und Normalisierung')
plt.legend()
plt.grid(True)
plt.show()
```

Bild 15 zeigt die Programmausgabe. Man erkennt, dass die Signalamplitude entsprechend der Verstärkung zugenommen hat. Der Offset des Signals wurde entfernt, d. h. das konditionierte Signal (blau) ist nun symmetrisch zur Nulllinie. Zudem wurde das Signal auf Werte zwischen -10 und +10 normalisiert.

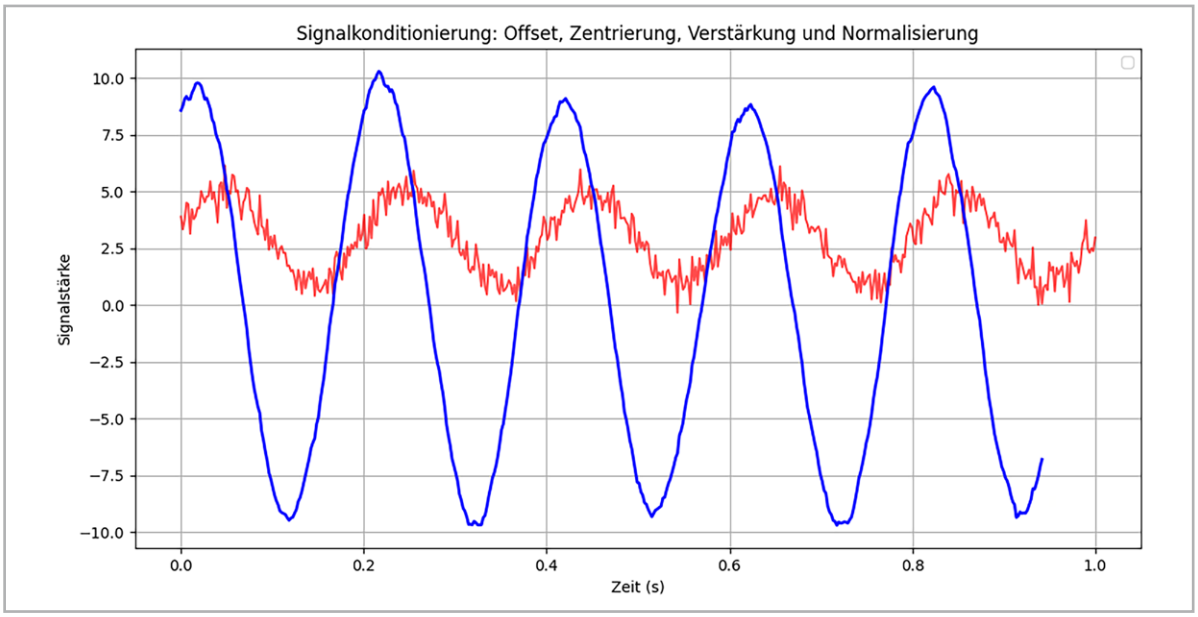

Bild 15: Signalkonditionierung

Zusammenfassung und Ausblick

In der modernen Mess- und Sensortechnik spielt Python eine zentrale Rolle, insbesondere in Verbindung mit Plattformen wie dem Raspberry Pi. Die Programmiersprache Python ermöglicht eine einfache Ansteuerung verschiedenster Sensoren, wie dem DS18x20 oder dem Feuchtigkeitssensor DHT11. Dank zahlreicher Bibliotheken können Daten effizient erfasst, verarbeitet und visualisiert werden. Typische Anwendungen umfassen Temperaturmessungen, Luftfeuchtigkeitsüberwachung und die Analyse von Sensordaten in Echtzeit. Zudem wurde in diesem Artikel die Datenaufbereitung umfassend betrachtet. Dazu gehören die Mittelung von Messwerten, die Rauschreduzierung und die Anpassung der Daten an bestimmte Wertebereiche usw.

Alle Grafiken und Messwerte wurden dazu mithilfe der MatPlotLibrary angezeigt und dargestellt. Neben dieser Bibliothek existieren allerdings noch weitere Anwendungen, die interessante und vielseitige Grafikdarstellungen erlauben. Eine der wichtigsten Varianten hierzu ist tkinter, die Standard-GUI-Bibliothek (Graphical User Interface) für Python. Sie bietet einfache Möglichkeiten zur Erstellung von Desktop-Anwendungen. So können z. B. eigene Fenster für die Ausgabe von Messwerten oder selbst definierte „Displays“ aufgebaut werden. Tkinter ist dabei einfach zu erlernen und eignet sich auch für Anfänger, die mit der GUI-Programmierung beginnen möchten. Zudem ermöglicht tkinter die Verarbeitung von Ereignissen wie z. B. Mausklicks und Tastatureingaben, um auf Benutzerinteraktionen zu reagieren. ELV

Ergänzungen und Anregungen

- Kann man beide Sensoren (DS18x20 und DHT11) gemeinsam an einem Raspberry Pi betreiben?
  - Wie müsste das zugehörige Programm aussehen?
  - Wie viele Sensoren wären maximal möglich?

- Wie sähe ein Programm zur Signalkonditionierung für den DS18x20 aus, mit den folgenden Eigenschaften:
  - Mittelung von jeweils 100 Messwerten
  - Zentrierung der Temperatur auf 25 °C (Raumtemperatur)
  - Rauschreduzierung auf ±0,2 °C

Material

Raspberry Pi mit Netzteil	
z. B. Raspberry Pi 4 Model B, 8 GB RAM	Artikel-Nr. 250567
z. B. Raspberry Pi 4 USB-Netzteil Typ C	Artikel-Nr. 250962
Breadboard und Jumper-Kabel	
DS18x20-Sensor	
DHT11-Sensor oder -Modul	

Zum Download-Paket