Messen und Darstellen von Audiosignalen

Mit numpy und gnuplot

28.05.2018, Sebastian Schmidt (sebastian.seb.schmidt@fau.de)

github.com/schmidtseb/RaspberryPiUebung

Messung von Audiosignalen

Das Programm zur Temperaturmessung wird modifiziert, um nun mittels Mikrofon und ADC-Wandler Audiosignale aufzeichnen zu können.

Einzelfunktionen

Im Folgenden wird ein Teil der Komponenten des Programms im Detail besprochen - der Quellcode des kompletten Programms befindet sich im Anhang. Anders als im Programm zur Messung der Temperatur, wird hierbei `numpy` zum Speichern der Daten verwendet.

main-Funktion

Die `main`-Funktion enthält Deklarationen von Variablen, den Aufruf von wichtigen Unterfunktionen und eine Schleife, innerhalb der die eigentliche Messung stattfindet.

Wird ein Python-Skript aufgerufen, so wird es Zeile für Zeile ausgeführt. Ein Funktionsaufruf ist nur dann gültig, wenn die Funktion vorher sachgemäß definiert wurde. Dies hätte zu bedeuten, dass der Hauptteil des Programms stets am Ende stehen muss.

Dies kann jedoch durch folgende Zeilen am Programmende umgangen werden:

```
if __name__ == '__main__':
    main()
```

Wird ein Skript direkt gestartet, so wird `__name__` als `'__main__'` gesetzt. Wird ein Skript als Modul geladen (d.h. über `import`), so wird `__name__` gleich dem Namen des Moduls gesetzt. Dies erlaubt zum einen den Hauptteil des Programms in eine Funktion `main()` zu schreiben, die nun auch am Programmanfang stehen darf, aber auch den import des Programms als Modul, ohne dass die `main()`-Funktion dabei ausgeführt wird.

Initialisierung GPIO

Über `GPIO.setmode` wird die zu verwendende Nummerierung der GPIO-Pins festgelegt. Vier dieser Pins werden selektiert eine SPI-Schnittstelle zu bilden. Anschließend werden drei von ihnen als Ausgänge (SCLK, MOSI, CS) und einer von ihnen als Eingang (MISO) festgelegt.

```
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Variablen
adcChannel = 0

# Pins
SPI_SCLK = 18
SPI_MOSI = 24
SPI_MISO = 23
SPI_CS = 25

GPIO.setup(SPI_SCLK, GPIO.OUT)
GPIO.setup(SPI_MOSI, GPIO.OUT)
GPIO.setup(SPI_MISO, GPIO.IN)
GPIO.setup(SPI_CS, GPIO.OUT)
```

Schleife zur Messdatenerfassung

Zum Start der Messung wird eine Messdauer in Sekunden festgelegt und die aktuelle Zeit in der Variable `startTime` gespeichert. Diese ist in UNIX-Zeit angegeben, d.h. die Zeit in Sekunden seit dem 1.1.1970.

```
# Start Messung
messdauer = 60      # Zeit in Sekunden
startTime = time.time()
```

Bevor die Schleife durchlaufen wird, werden zwei leere Listen erstellt, um die erfassten Daten zu speichern. Diese sind in diesem Fall die Zeit des Signals, sowie dessen Amplitude.

```
timeList = []
amplitudeList = []
```

Die Daten werden in einer `while`-Schleife eingelesen, die in jeder Iteration die Differenz zwischen der Startzeit `startTime` und der aktuellen Zeit bildet und mit der Messdauer vergleicht.

Die berechnete Zeitdifferenz wird in der Liste `timeList` abgespeichert. Mittels Funktion `readADC` wird ein angegebener Kanal angesprochen und dessen aktueller ADC-Wert ausgelesen. Dieser wird anschließend in einen Amplitudenwert in Prozent konvertiert und gespeichert.

```
while startTime - time.time() < messdauer:
    timeList.append( time.time() - startTime )
    adcVal = readADC(adcChannel, SPI_SCLK, SPI_MOSI, SPI_MISO, SPI_CS)
    amplitudeList.append( adcToPercent(adcVal) )
# Ende Messung
```

Auslesen des ADCs

Der ADC wird über die SPI-Schnittstelle angesteuert. Um ihn zu selektieren, wird ein Puls auf dem CS-Pin (Chip Select) erzeugt. Anschließend wird der SCLK-Pin (Serial Clock) auf low gesetzt, da aktuell keine Daten übertragen werden.

```
def readADC(adcChannel, SCLKPin, MOSIPin, MISOPin, CSPin):
    # Selektiere den ADC
    GPIO.output(CSPin, True)
    GPIO.output(CSPin, False)
    GPIO.output(SCLKPin, False)
```

Das zu sendende Kommando besteht aus dem anzusprechenden Kanal (die ersten drei Bits), sowie einem Startbit und der Wahl des Betriebsmodus (die anderen beiden Bits). Damit hat es eine Gesamtlänge von 5 Bits, die übetragen werden müssen.

```
    cmd = adcChannel
    cmd |= 0b11000
    cmdLength = 5
```

Das Kommando wird über eine Iteration über dessen einzelne Bits übertragen. Über `if cmd &; (1 <;<; (cmdLength - i))` wird überprüft, ob das aktuelle Bit wahr ist oder nicht. Falls dies der Fall ist, so wird der MOSI-Pin (Master Output Slave Input) auf high gesetzt, falls nicht, auf low. Daraufhin findet die Erzeugung eines einzelnen Clockpulses statt und das nächste Bit des Kommandos wird überprüft.

```
    # Senden des Kommandos
    for i in range(cmdLength):
        if cmd & (1 << (cmdLength - i)):
            GPIO.output(MOSIPin, True)
        else:
            GPIO.output(MOSIPin, False)

        # Ein Clockpuls
        GPIO.output(SCLKPin, True)
        GPIO.output(SCLKPin, False)
```

Sobald das Kommando den ADC erreicht hat, beginnt dieser mit dem Versenden der angeforderten Daten. Diese werden in der Variable `response` gespeichert, die mit dem Wert 0 initialisiert wird. Die Länge der Antwort beträgt stets 13, bestehend aus einem Nullbit und den eigentlichen Daten (12 Bit, die Auflösung des ADCs). Ein Clockpuls wird erzeugt und ein Bit über den MISO-Pin (Master Input Slave Output) eingelesen. Da das erste Bit das Nullbit ist, wird dieses nicht gespeichert. Anschließend wird über den Rest der Antwort iteriert, jeweils ein Clockpuls generiert und ein Bit eingelesen. Ist dieses wahr, wird die entsprechende Stelle in `response` gesetzt.

```
    # Empfangen von Daten
    response = 0
    responseLength = 12

    # Empfange Nullbit
    GPIO.output(SCLKPin, True)
    GPIO.output(SCLKPin, False)

    # Empfange Daten
    for i in range(responseLength):
        GPIO.output(SCLKPin, True)
        GPIO.output(SCLKPin, False)

        if GPIO.input(MISOPin):
            response |= (1 << (responseLength - i)

    time.sleep(0.5)
    return response
```

Umwandlung: ADC-Wert in Prozent

Der digitale Wert des ADCs wird in einen Amplitudenwert umgerechnet. Dieser wird in Prozent, relativ zum maximalen Wert, angegeben.

```
def adcToPercent(adcVal):
    adcResolution = float(2**12 - 1)
    voltVal = adcVal/adcResolution * 3.3

    # Spannung relativ zum halben Maximum
    voltVal = voltVal - 3.3 / 2

    # Normierung
    voltVal /= (3.3 / 2)

    return adcVal
```

Die Auflösung des ADCs beträgt 12 bit, d.h. die größte darzustellende Zahl ist $12^2 - 1 = 4095$. Der ADC arbeitet mit einer maximalen Spannung von $3.3\,\mathrm{V}$. Damit beträgt die Spannungsauflösung $3.3\,\mathrm{V}/4095$, also $805.9\,\mu\mathrm{V}$.

Das verwendete Mikrofon arbeitet lediglich im positiven Spannungsbereich. Um also eine positive und negative Amplitude des Audiosignals erfassen zu können, wird stets die Hälfte der angelegten Spannung addiert.

Der vom ADC ausgegebene Wert `adcVal` wird in eine Spannung konvertiert. Anschließend wird der Spannungsoffset von $3.3\,\mathrm{V}/2$ subtrahiert um das Signal symmetrisch um die $0\,\mathrm{V}$ zu platzieren. Abschließend wird das Signal auf das Maximum normiert und zurückgegeben.

Erzeugung eines Frequenzspektrums

Aus den gemessenen Zeit- und Amplitudendaten des Audiosignals soll mit Unterstützung von `numpy` ein Frequenzspektrum erzeugt werden. Dazu kommt eine schnelle Fourier-Transformation (FFT) zum Einsatz. Diese erlaubt die Berechnung einer diskreten Fourier-Transformation (DFT) innerhalb einer kurzen Zeit. Wichtig für eine DFT ist, dass der zeitliche Abstand zwischen zwei Messpunkten stets einheitlich ist. Dies ist in unserem Fall nur bedingt gegeben.

Um dem entgegenzuwirken, wird der mittlere zeitliche Abstand berechnet. `np.diff()` berechnet die Differenz der benachbarten Elemente einer Liste oder eines Arrays. Anschließend wird über `np.mean()` der Mittelwert der Differenzen berechnet.

```
def calculateFFT(time, amplitude):
    # Mittlere Zeitdifferenz zwischen zwei Datenpunkten
    tMean = np.mean(np.diff(time))
```

Über `np.arange()` werden nun neue Zeitpunkte, nun mit einheitlichem Abstand, erzeugt.

```
    # Erzeuge neue Zeiten
    timeNew = np.arange(min(time), max(time), tMean)
```

Eine lineare Interpolation der alten Amplituden- und Zeitwerte erlaubt die näherungsweise Bestimmung der Amplitudenwerte zu den neuen Zeitpunkten.

```
    # Interpolation
    amplitudeNew = np.interp(timeNew, time, amplitude)
```

`numpy` besitzt einige Funktionen zur Bestimmung des Frequenzspektrums. Diese sind im Namensraum `np.fft` zu finden. `rfft` berechnet die FFT für reelle Eingangswerte. Die Hilfsfunktion `rfftfreq` berechnet die zugehörigen Frequenzwerte aus der neu bestimmten Zeitdifferenz zwischen zwei Messungen.

```
    # Calculate FFT
    freq = np.fft.rfftfreq(timeNew.size, tMean)
    freqAmplitude = abs( np.fft.rfft(amplitudeNew) )

    return freq, freqAmplitude
```

Speichern von Daten

Beispielsweise erklärt an den Amplitudendaten des Signals. `numpy` bietet mit der Funktion `savetxt` eine einfache Möglichkeit zum Speichern des Inhalts eines Arrays in einer Textdatei. Dazu wird zuerst das Array `amplitudeData` erstellt. Die Messdaten der Zeit und der Amplitude des Signals sind je in Listen gespeichert. Diese werden nun in eine sie umfassende Liste gesetzt und über `np.asarray` in ein Array konvertiert. Über den Operator `.T` wird dieses Array transponiert, wobei die Funktion analog zu einer Matrix-Transposition ist. Man erhält also ein Array, das als Einträge je den Zeitpunkt und den Amplitudenwert einer Messung enthält. Dieses wird nun über `np.savetxt` gespeichert. Über `header=` kann die erste Zeile in der Textdatei angegeben werden. Diese beginnt bereits mit einem Kommentar, also mit dem Zeichen `#`, das nicht extra mit angegeben werden muss.

```
    amplitudeData = np.asarray( [timeList, amplitudeList] ).T
    np.savetxt('amplitudeData.dat', amplitudeData, header='Time (s)\tAmplitude
(\%)')
```

Erzeugung von Beispieldaten

Bevor Messdaten aufgenommen wurden, können Beispieldaten erzeugt und anschließend ausgewertet werden. Dies erlaubt, sich mit der Funktion der FFT, aber auch dem Plotten der Daten an sich vertraut zu machen.

```
import numpy

f = 50 # Frequenz in Hz
t = np.linspace(0, 0.2, 1000) # Zeit in s
A = np.sin(2*np.pi*f*t) + np.sin(2*np.pi*2*f*t) # Amplitude in a.u.

freq = np.fft.rfftfreq(t.size, t[1] - t[0])
freqAmplitude = abs( np.fft.rfft(A) )

np.savetxt('testAmplitude.dat', np.asarray([t, A]).T, header='Time
(s)\tAmplitude (\%)')
np.savetxt('testFrequency.dat', np.asarray([freq, freqAmplitude]).T,
header='Frequency (Hz)\tAmplitude (a.u.)')
```

Plotten von Daten

Das Plotten von Daten soll nun mittels gnuplot erfolgen. Erstelle einen Plot deiner Messdaten, d.h. des Audiosignals und dessen Frequenzspektrums, der unter anderem die folgenden Voraussetzungen erfüllt:

- Korrekte Beschriftung der Achsen, d.h. inklusive Einheiten
- Skalierung der Achsen: Im Falle von Frequenzspektren bietet es sich oftmals an, diese logarithmisch darzustellen
- Darstellung von Gitternetzlinien

```
import RPi.GPIO as GPIO
import time
import numpy as np

def main():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)

    # Variablen
    adcChannel = 0

    # Pins
    SPI_SCLK = 18
    SPI_MOSI = 24
    SPI_MISO = 23
    SPI_CS = 25

    GPIO.setup(SPI_SCLK, GPIO.OUT)
    GPIO.setup(SPI_MOSI, GPIO.OUT)
    GPIO.setup(SPI_MISO, GPIO.IN)
    GPIO.setup(SPI_CS, GPIO.OUT)

    # Initialiserung der Messdateien
    fAmplitude = initFile('amplitudeData.dat', ['Zeit', 'Amplitude'])
    fFrequenz = initFile('freuqencyData.dat', ['Frequenz', 'FFT'])

    # Start Messung
    messdauer = 60      # Zeit in Sekunden
    startTime = time.time()

    timeList = []
    amplitudeList = []
    while startTime - time.time() < messdauer:
        timeList.append( time.time() - startTime )
        adcVal = readADC(adcChannel, SPI_SCLK, SPI_MOSI, SPI_MISO, SPI_CS)
        amplitudeList.append( adcToPercent(adcVal) )
    # Ende Messung

    # Speicherung Amplitudendaten
    amplitudeData = np.asarray( [timeList, amplitudeList] ).T
    np.savetxt('amplitudeData.dat', amplitudeData, header='Time (s)\tAmplitude
(\%)')

    # Erzeugung und Speicherung der Freuqenzdaten
    freq, freqAmplitude = calculateFFT(timeList, amplitudeList)
    frequencyData = np.asarray( [freq, freqAmplitude] ).T
    np.savetxt('frequencyData.dat', frequencyData, header='Frequency
(Hz)\tAmplitude (a.u.)')

def readADC(adcChannel, SCLKPin, MOSIPin, MISOPin, CSPin):
```

```
    # Selektiere den ADC
    GPIO.output(CSPin, True)
    GPIO.output(CSPin, False)
    GPIO.output(SCLKPin, False)

    cmd = adcChannel
    cmd |= 0b11000
    cmdLength = 5

    # Senden des Kommandos
    for i in range(cmdLength):
        if cmd & (1 << (cmdLength - i)):
            GPIO.output(MOSIPin, True)
        else:
            GPIO.output(MOSIPin, False)

        # Ein Clockpuls
        GPIO.output(SCLKPin, True)
        GPIO.output(SCLKPin, False)

    # Empfangen von Daten
    response = 0
    responseLength = 12

    # Empfange Nullbit
    GPIO.output(SCLKPin, True)
    GPIO.output(SCLKPin, False)

    # Empfange Daten
    for i in range(responseLength):
        GPIO.output(SCLKPin, True)
        GPIO.output(SCLKPin, False)

        if GPIO.input(MISOPin):
            response |= (1 << (responseLength - i)

    time.sleep(0.5)
    return response

def adcToPercent(adcVal):
    adcResolution = float(2**12 - 1)
    voltVal = adcVal/adcResolution * 3.3

    # Spannung relativ zum halben Maximum
    voltVal = voltVal - 3.3 / 2

    # Normierung
    voltVal /= (3.3 / 2)

    return adcVal
```

```
def calculateFFT(time, amplitude):
    # Mittlere Zeitdifferenz zwischen zwei Datenpunkten
    tMean = np.mean(np.diff(time))

    # Erzeuge neue Zeiten
    timeNew = np.arange(min(time), max(time), tMean)

    # Interpolation
    amplitudeNew = np.interp(timeNew, time, amplitude)

    # Calculate FFT
    freq = np.fft.rfftfreq(timeNew.size, tMean)
    freqAmplitude = abs( np.fft.rfft(amplitudeNew) )

    return freq, freqAmplitude

if __name__ == '__main__':
    main()
```