Technische Universität Braunschweig

Medizinische Hochschule Hannover

PLRI PETER L. REICHERTZ INSTITUT FÜR MEDIZINISCHE INFORMATIK

Documentation

Data acquisition and storage based on the sensor board “BITalino” using Python

Augusto Maneiro

Version 1

Datum: 08.03.2019

Index

0. Setting Up

- Install the Python IDE PyCharm: (https://www.jetbrains.com/pycharm/)
- Install Python 3.7: (https://www.python.org/downloads/release/python-370/)
- Install the BITalino (r)evolution Python API: (https://github.com/BITalinoWorld/revolution-python-api)
- Install the PyQtGraph library (for plotting): (http://www.pyqtgraph.org/)
- Import the Python CSV library.
- Import the Python time library.

1. Bluetooth Connection

To be able to interact with the sensor board from the computer, the first step is to establish a Bluetooth connection with it. To establish the Bluetooth connection between the board and a computer running the Windows or Linux operating system, it is needed to instantiate a new 'BITalino' object, giving as a parameter the MAC address of the board. On the other hand, it is not possible, using the BITalino (r)evolution Python development API, to connect the sensor board to a computer running MacOS using the MAC address of the board. Instead of this, the Virtual COM Port has to be used. Usually this is the format that it should have: "/dev/tty.BITalino-XX-XX-DevB", where XX-XX are the last four hexadecimal digits on the MAC address of the board.

To instantiate a new BITalino object, the "BITalino()" method included in the BITalino (r)evolution Python API is used.

```
device = BITalino(macAddress)  ## creates a new connection with the board
```

Creation of a new BITalino Bluetooth connection

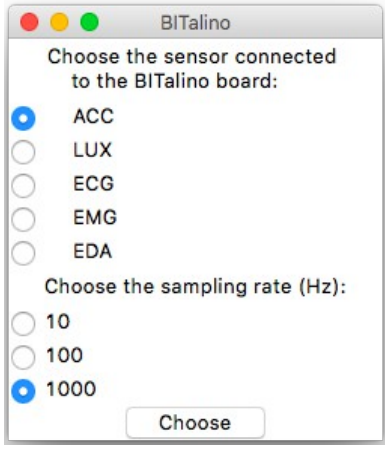
2. Configurable Parameters

The board has some parameters that can be configured by the user. This parameters are the type of sensor that is connected to the board, the sampling rate (number of samples that the sensor board takes per second, expressed in Hz) and the number of samples read

in each data acquisition.

The possible values for the sampling rate are 10 Hz, 100 Hz and 1000 Hz.

The first two parameters are set by the user through a window that shows radio buttons with the different possible values for this parameters.



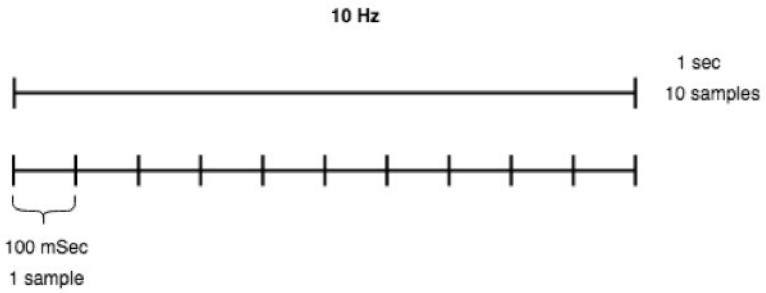
Tkinter Window with radio buttons

This window was created using the TKInter Python library. The code can be found in the *"SensorsRadioButtons.py"* Python file.

Once this two parameters are configured, we need to set the number of samples to be read in each data acquisition. The value of this parameter should be linked with the sampling rate to avoid synchronization problems.

```
## Sets the number of samples acquired in each
## data acquisition according to the chosen sample rate
if sRate == 10:
    nSamples = 1
elif sRate == 100:
    nSamples = 10
elif sRate == 1000:
    nSamples = 100
```

For example, if the sampling rate is set to 10Hz, that means that the board will read 10 samples per second. But if it performs a data acquisition every 0,1 seconds, that means that the number of read samples should be set to 1, so that every second it will get 10 samples. By default, the number of samples of the *read()* function is set to 100 samples. That is fine if the sampling rate is set to 1000 Hz and the data acquisitions are performed every 0,1 seconds. But if this is not the case, this value should be modified to prevent this kind of problems.



Example with sampling rate set to 10Hz

Finally, to prepare the board to acquire data, the *start(sRate, analogChannels)* method has to be called. It receives the chosen sampling rate and the analog channels from where we want to acquire data as parameters. The analog channels have to be sent to the function as a list. In this case we are including all the channels (the 6 analog channels that the board has) even if we only acquire data from the first (A1) channel. This is how the list looks like:

```
analogChannels = [0, 1, 2, 3, 4, 5]
```

Note that the first channel (A1) is the first element on the list (0) and so on.

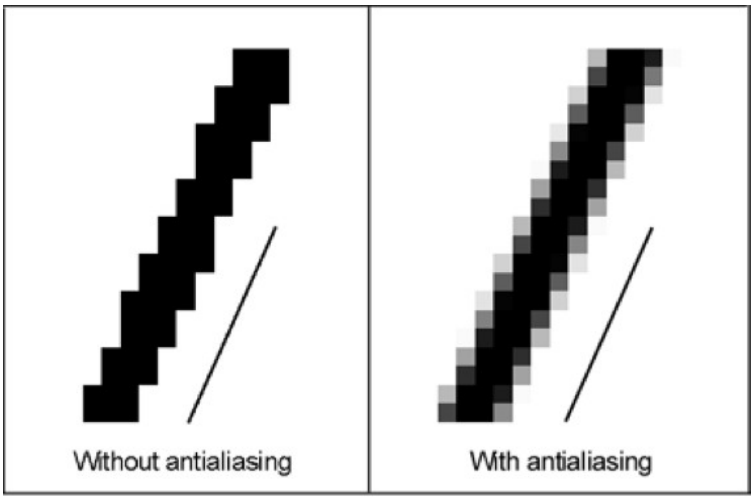
3. Plot set up

For the plotting task the PyQtGraph library was employed. It is a graphics and user interface library for Python that provides functionality commonly required in engineering and science applications.

The first step is the creation of a window that will hold our graph later. We can change the size of the window and also set a title to it.

```
## Creates the window that will hold the plot
win = pg.GraphicsWindow(title="BITalino")
win.resize(1000, 600)
win.setWindowTitle('BITalino')
```

To get prettier plots, the antialiasing parameter of the plot must be enabled. This will make the plotted signal to be smoothed.



Then we have to create the plot and add it to the window. We can choose if we want to show the grid in our plot or not, and also set labels to the axis. In this case a label is set to the X axis to show that this axis' units are expressed in seconds.

```
## Creates the plot
plot = win.addPlot(title="BITalino Sensor plot")
plot.showGrid(x=True, y=True)
plot.setLabel('bottom', 'Time', units='s')
```

The data to be plotted will be hold in two lists, one for the X axis data (time) and the other one for the Y axis data (sensor information). In a first instance we will create this two empty lists.

```
## Empty lists that will hold the plotted data
xaxis = []
yaxis = []
```

Then, depending on the type of sensor that is connected to the board, the label (that shows the name of the sensor and the data units) and the range of the Y axis have to be set so they are consistent with the plotted data and the range of values that is shown.

```
## Sets the Y-axis limits and labels depending on the connected sensor
if sensor == 'ACC':
    plot.setLimits(yMin=-4, yMax=4, xMin=0)
    plot.setLabel('left', 'ACC: g-force', units='g')
elif sensor == 'LUX':
    plot.setLabel('left', 'LUX: Luminosity', units='%')
    plot.setLimits(yMin=-1, yMax=101, xMin=0)
elif sensor == 'ECG':
    plot.setLabel('left', 'ECG:', units='mV')
    plot.setLimits(yMin=-2, yMax=2, xMin=0)
elif sensor == 'EMG':
    plot.setLabel('left', 'EMG:', units='mV')
    plot.setLimits(yMin=-2, yMax=2, xMin=0)
elif sensor == 'EDA':
    plot.setLabel('left', 'EDA:', units='')
    plot.setLimits(yMin=-1, yMax=1000, xMin=0)
```

Finally, we have to disable the auto range property of the plot, so that the whole Y axis range of values is shown every time. And we have to create a curve object that will be in charge of plotting the data that is appended in our lists. We can also change the color of the plot with the 'pen' property (it is set to yellow).

```
plot.setRange(rect=None, xRange=None, yRange=(0, 100),
              padding=None, update=False, disableAutoRange=True)
curve = plot.plot(xaxis, yaxis, pen='y')
```

4. Data Storage

The timestamps, the raw data acquired by the board and the real values are stored in CSV files. This CSV files have also a header section with some metadata like the date and time when the data acquisition started, the device's name, the type of sensor connected to the board, the channel and the selected sampling rate. In this case the values are not separated by commas but by tabs.

The first step is to create the CSV file where the data will be written to. The names of the files are the date and time when the data acquisition started followed by the name of the sensor that is connected to the board (e.g.: “2019-02-28_11-59-29_ACC.csv”). To get the actual date and time, the 'strftime()' method of the time library was used.

```
## Creates .csv file
dt = time.strftime("%Y-%m-%d_%H-%M-%S")
with open(dt + '_' + sensor + '.csv', mode='w') as bitalino_file:
    bitalino_writer = csv.writer(bitalino_file, delimiter='\t', quotechar='"',
                                 quoting=csv.QUOTE_MINIMAL, lineterminator='\n')
    bitalino_writer.writerow(['# Date and time: ' + dt])
    bitalino_writer.writerow(['# Device name: ' + macAddress])
    bitalino_writer.writerow(['# Sensor: ' + sensor])
    bitalino_writer.writerow(['# Analog channel: ' + 'A1'])
    bitalino_writer.writerow(['# Sampling rate (Hz): ' + str(sRate)])
    bitalino_writer.writerow(['# EndOfHeader'])
    bitalino_writer.writerow(['TIMESTAMP', 'RAW VALUE', 'REAL VALUE'])
bitalino_file.close()
```

To update the CSV files with new data, a function is implemented. This function receives the timestamp, the raw data value and the real data value of each sample taken by the board, opens the CSV file and writes the new information.

```
def writeToCSVFile(timestamp, rawValue, realValue):
    global dt, sensor
    with open(dt + '_' + sensor + '.csv', mode='a') as bitalino_file:
        bitalino_writer = csv.writer(bitalino_file, delimiter='\t', quotechar='"',
                                     quoting=csv.QUOTE_MINIMAL, lineterminator='\n')
        bitalino_writer.writerow([timestamp, rawValue, realValue])
    bitalino_file.close()
```

5. Sensors

The BITalino board includes several sensors that can be plugged to it to acquire different kinds of data. Each sensor has its own transfer function. This is the function that has to be applied to the raw data acquired by the board to transform it into meaningful and significant data. This transfer functions are stored in the *"TransferFunctions.py"* Python file. It contains a function that receives the raw value acquired by the sensor board and the type of sensor used and returns the corresponding real value after applying the transfer function.

```
def transferFunction(rawData, sensor):

    cmin = 379 ## minimum calibration value for accelerometer
    cmax = 696 ## maximum calibration value for accelerometer
    value = 0

    if sensor == 'ACC':
        value = ((rawData - cmin) / (cmax - cmin)) * 2 - 1
    elif sensor == 'LUX':
        value = (rawData / 2**10) * 100
    elif sensor == 'ECG':
        value = ((((rawData / 2**10) - 1/2) * 3.3) / 1100) * 1000
    elif sensor == 'EMG':
        value = ((((rawData / 2**10) - 1/2) * 3.3) / 1000) * 1000
    elif sensor == 'EDA':
        value = 1 / (1 - (rawData / 2**10))

    return value
```

5.1. Accelerometer (ACC)

By default, only the Z axis is connected to the accelerometer sensor, but the X-axis and Y-axis can be also connected (this procedure will not be covered in this document). This is the accelerometer's transfer function:

TRANSFER FUNCTION
[-3g, 3g]

$$ACC(g) = \frac{ADC - C_{min}}{C_{max} - C_{min}} . 2 - 1$$

$ACC(g)$ – ACC value in g-force (g)
ADC – Value sampled from the channel
C_{min} – Minimum calibration value[1] (typically $C_{min} \approx 208$)
C_{max} – Maximum calibration value[1] (typically $C_{max} \approx 312$)

The values that it obtains goes from -3g to 3g. To obtain the calibration values (C_{min} and C_{max}) we have to perform a very slow 360º rotation around the Z-axis of the sensor and assign the minimum raw value obtained to C_{min} and the maximum raw value to C_{max}.
More information about this sensor can be found in the *"ACC_Sensor_Datasheet.pdf"* document inside the *"Sensor_Data_Sheets"* folder.

5.2. Light Sensor (LUX)

TRANSFER FUNCTION
[0%, 100%]

$$LUX(\%) = \frac{ADC}{2^n} . 100\%$$

$LUX(\%)$ – LUX value in percentage (%)
ADC – Value sampled from the channel
n – Number of bits of the channel[1]

The values that this sensor obtains are expressed in percentage. 0% represents the complete absence of light. The value of the n parameter will depend on which channel the LUX sensor is plugged to. In BITalino the first four channels (A1, A2, A3 and A4) are sampled using 10-bit resolution (n = 10), while the last two (A5 and A6) are sampled using 6-bit (n = 6).
More information about this sensor can be found in the *"LUX_Sensor_Datasheet.pdf"* document inside the *"Sensor_Data_Sheets"* folder.

5.3. Electrocardiography (ECG)

TRANSFER FUNCTION
[-1.5mV, 1.5mV]

$$ECG(V) = \frac{\left(\frac{ADC}{2^n} - \frac{1}{2}\right).VCC}{G_{ECG}}$$

$$ECG(mV) = ECG(V).1000$$

VCC = 3.3V (operating voltage)
G_{ECG} = 1100 (sensor gain)

$ECG(V)$ – ECG value in Volt (V)
$ECG(mV)$ – ECG value in millivolt (mV)
ADC – Value sampled from the channel
n – Number of bits of the channel[1]

The values that this sensor acquires goes from -1.5mV to 1.5mV.
The value of the n parameter will depend on which channel the ECG sensor is plugged to. In BITalino the first four channels (A1, A2, A3 and A4) are sampled using 10-bit resolution (n = 10), while the last two (A5 and A6) are sampled using 6-bit (n = 6).
More information about this sensor can be found in the *"ECG_Sensor_Datasheet.pdf"* document inside the *"Sensor_Data_Sheets"* folder.

5.4. Electromyography (EMG)

This sensor measures the muscle activity.

TRANSFER FUNCTION
[-1.65mV, 1.65mV]

$$EMG(V) = \frac{\left(\frac{ADC}{2^n} - \frac{1}{2}\right).VCC}{G_{EMG}}$$

$$EMG(mV) = EMG(V).1000$$

VCC = 3.3V (operating voltage)
G_{EMG} = 1000 (sensor gain)

$EMG(V)$ – EMG value in Volt (V)
$EMG(mV)$ – EMG value in millivolt (mV)
ADC – Value sampled from the channel
n – Number of bits of the channel[1]

The values that this sensor acquires goes from -1.65mV to 1.65mV.
The value of the n parameter will depend on which channel the EMG sensor is plugged to. In BITalino the first four channels (A1, A2, A3 and A4) are sampled using 10-bit resolution

(n = 10), while the last two (A5 and A6) are sampled using 6-bit (n = 6).
More information about this sensor can be found in the *"EMG_Sensor_Datasheet.pdf"* document inside the *"Sensor_Data_Sheets"* folder.

5.5. Electrodermal Activity (EDA)

This sensor is used to measure the skin activity.

TRANSFER FUNCTION
$[1\mu S, \infty\mu S]$

$$R(MOhm) = 1 - \frac{ADC}{2^n}$$

$$EDA(\mu S) = \frac{1}{R(MOhm)}$$

$R(MOhm)$ – Sensor resistance value mega-Ohm $(MOhm)$
$EDA(\mu S)$ – EDA value in micro-Siemens (μS)
ADC – Value sampled from the channel
n – Number of bits of the channel[1]

The values that this sensor acquires goes from 1 µS to ∞ µS.
The value of the n parameter will depend on which channel the EDA sensor is plugged to. In BITalino the first four channels (A1, A2, A3 and A4) are sampled using 10-bit resolution (n = 10), while the last two (A5 and A6) are sampled using 6-bit (n = 6).
More information about this sensor can be found in the *"EDA_Sensor_Datasheet.pdf"* document inside the *"Sensor_Data_Sheets"* folder.

6. The *update* loop

This function is the one in charge of getting the data that the sensor boards sends, transforming the raw values into real ones applying the transfer functions, writing new data to the CSV files and updating the plot to show the real time data.
The *update()* function is called at regular periods of time. To do this a QTimer object is implemented, which is in charge of emitting a timeout signal at equally spaced periods of time (this time parameter can be modified). The QTimer calls the *update()* function each time this timeout signal is emitted.

```
timer = QtCore.QTimer()
timer.timeout.connect(update)  ## the update function is called every time the timeout signal is emitted
timer.start(10)  ## emits a timeout signal every 10 mSec
```

The first action that this function performs is the reading of samples. To do this the *read(nSamples)* method of the BITalino (r)evolution Python API is used. The data acquired is organized in a matrix whose lines correspond to samples and the columns are as follows:

- Sequence number (used to check for lost data, goes from 0 to 15 and then starts again at 0);
- 4 digital channels (are always present);
- 1-6 analog channels (as defined in the *start()* method).

Once we have the data matrix, we have to iterate through it to transform all the raw values with the transfer function. Then we will add this values and the corresponding timestamps to the 'xaxis' and 'yaxis' lists that we created before. Finally we call the *writeToCSVFile(mytimer, rawValue, yValue)* to add the new data to our CSV file, and then the *myTimer* variable, that contains the timestamps, is updated.

```
data = device.read(nSamples) ## reads samples

for sample in data:
    rawValue = sample[5]
    yValue = tf.transferFunction(rawValue, sensor) ## data after applying transfer function
    yaxis.append(yValue)
    xaxis.append(mytimer)

    writeToCSVFile(mytimer, rawValue, yValue)
    mytimer += 1 / sRate
```

Note that *sample[5]* is the value assigned to the *rawValue* variable. The index 5 is used because it is the one that corresponds to the data acquired in the A1 channel in the data matrix.

Once the *for* loop is complete, the data contained in the 'xaxis' and 'yaxis' lists is set to the curve to be plotted using the *setData()* method. To enable the plot to move through the X-axis as the time goes by, we need to continuously set the X-axis range of the plot. In this case, we show the data acquired up to 10 seconds before the actual time and 10 seconds after.

```
curve.setData(xaxis, yaxis)
plot.setRange(rect=None, xRange=(mytimer-10, mytimer+10),
              padding=None, update=False, disableAutoRange=True)
```

The last part of the *update()* function consist in erasing old data from the 'xaxis' and 'yaxis' lists to prevent them from growing without limits and avoid performance issues.

```
## after some time the first values of the xaxis and yaxis
## lists are eliminated to prevent them from growing with no limits
if len(xaxis) >= sRate * 30:
    del xaxis[:sRate*10]
    del yaxis[:sRate*10]
```

7. Disconnection

Every time the plot window is closed or when a problem with the acquisition occurs, the BITalino device has to be stopped and the connection with the board has to be closed. To do this two functions are implemented:

```
def closeConnection(device):
    device.close()
    print('CONNECTION CLOSED')

def stopDevice(device):
    device.stop()
    print('DEVICE STOPPED')
```

Both *close()* and *stop()* methods are included in the BITalino (r)evolution Python API.