Hand Gesture Recognition

Technical report

Real-time interactive assistive robotics

Luis Pedro Dos Santos

Email: dossantos.luis89@gmail.com

June 17 – August 09, 2019

Technical internship

2018/2019

Acknowledgements

I would like to express my special thanks of gratitude to my internship supervisor Ahmad Lotfi, who greeted me at Nottingham Trent University and gave me important advices during my stay.

I would also like to extend my gratitude to the PhD Student Dario Ortega Anderez, who helped me all along my internship by lending me the necessary material, such as the sensors, and by helping me with my project, sharing related studies, giving me important advices and writing a conference paper with me.

I also thank David Ada Adama and Salisu Wada Yahaya for greeting me in the lab, helping me installing the setup and helping me in some parts of my project.

Finally, I want to thank Dario Ortega Anderez, David Ada Adama, Salisu Wada Yahaya, Hugo Simon and Tim Wittkor for participating in my experiments and thus allowing me to collect valuable data for my study.

Contents

Chapter 1

Introduction

This report aims to explain the work done during my internship, by describing the two algorithms: the first one being the plotting of the data from the accelerometer and the gyrometer retrieved from the MetaMotionR sensor, and the second one being the hand gesture recognition. I will further detail the important processes and techniques adopted in the algorithms and write down a critical feedback of those, explaining what works well and what needs to be improved.

Chapter 2

plot_AccGyro.py

2.1 Presentation of the algorithm

This algorithm plots the data from the accelerometer and the gyrometer from a MetaMotionR sensor, in real time, therefore providing an easy way to visualize the data.

The usage of the algorithm is as follows:

>>>   python plot_AccGyro.py [mac address of the sensor]

2.2 Description

The global variables *accX*, *accY*, *accZ* and *gyroX*, *gyroY*, *gyroZ* are arrays that will contain the data from the 2 sensors on each one of the 3 axis.

The *data_handler(self, ctx, data)* function is a callback: meaning that each time a new measure is retrieved from the sensor, this function is called.
Therefore, when this function is called, the *data* argument contains 6 values: 3 for the accelerometer and 3 for the gyrometer. Those measures are saved on the arrays cited in the precedent paragraph. Therefore, these arrays contain **all** the data, while *data* only contains the data at time t.

```
def data_handler(self, ctx, data):
    values = parse_value(data, n_elem = 2)

    global accX
    global accY
    global accZ
    global gyroX
    global gyroY
    global gyroZ

    accX = np.append(accX, values[0].x)
    accY = np.append(accY, values[0].y)
    accZ = np.append(accZ, values[0].z)
    gyroX = np.append(gyroX, values[1].x)
    gyroY = np.append(gyroY, values[1].y)
    gyroZ = np.append(gyroZ, values[1].z)
```

Fig. 1.   Callback function

The following lines of code (l.101 – 114) connect and configure the device.
At the end of the code (after l.169), the device stops taking measure and is reset.

Between these two (l.115 – 166), the data is continuously plotted. In order to do so, I used the *FuncAnimation* function from the *matplotlib.animation* library.

```python
ani = FuncAnimation(fig, animate, interval=20)
```

Fig. 2. FuncAnimation function

This function calls the *animate* function, detailed in Figure 3, every 20ms and updates the *fig* figure.
In fact, *interval* is the delay between frames, in milliseconds.

In the *animate* function, I plot the 6 arrays previously described that are continuously updated each time that the *callback* function is called (*accX*, *accY*, *accZ* and *gyroX*, *gyroY*, *gyroZ*).

```python
117  # animate(i) : plots the data in the 6 arrays every 'X' interval
118  def animate(i):
119      global accX
120      global accY
121      global accZ
122      global gyroX
123      global gyroY
124      global gyroZ
125  
126      #Accelerometer
127      ax1.clear()
128  
129      ax1.plot(accX, label='X-Axis')
130      ax1.plot(accY, label='Y-Axis')
131      ax1.plot(accZ, label='Z-Axis')
132  
133      ax1.title.set_text('Accelerometer')
134      ax1.grid()
135      ax1.legend(loc='upper left')
136      ax1.set_ylabel('linear acceleration (g)')
137  
138      #sliding window
139      ax1.set_xlim(left = max(0, len(accX)-400), right = len(accX)+1)
140  
141      #Gyrometer
142      ax2.clear()
143  
144      ax2.plot(gyroX, label='X-Axis')
145      ax2.plot(gyroY, label='Y-Axis')
146      ax2.plot(gyroZ, label='Z-Axis')
147  
148      ax2.title.set_text('Gyrometer')
149      ax2.grid()
150      ax2.legend(loc='upper left')
151      ax2.set_ylabel('angular velocity (degree/s)')
152  
153      #sliding window
154      ax2.set_xlim(left = max(0, len(gyroX)-400), right = len(gyroX)+1)
155  
```

Fig. 3. animate function

2.3 Feedbacks

In my study, the sensor streamed data at a frequency of 50Hz, meaning that a measure was retrieved every 20ms. That's why the *interval* value of the *animate* function is defined at 20ms. However, the plotting may not be completely smooth, since every 20ms, the computer clears the 2 subplots (just the subplots and not the figure) and plots again the 6 arrays, which can be computationally expensive.
To reduce this computation time, we can increase the number of the *interval* value and defining it at for example 100. Therefore, we are not plotting point by point anymore but plotting every 5 measures.

Chapter 3

gestureRecognition.py

3.1 Presentation of the algorithm

This is a hand gesture recognition algorithm predicting the movement performed by the user between a gesture set composed of 6 different gestures: *right*, *front*, *left*, *back*, *up* and *circle*.
A graphical user interface was developed to display the predictions.

The usage of the algorithm is as follows:

```
>>>    python gestureRecognition.py [mac address of the sensor]
```

3.2 Description

3.2.1 Training the classifier model

The very first step of the algorithm is to train the classifier.
In order to do so, a .csv file containing approximately 600 measures from 5 different subjects who performed the 6 different gestures approximately 20 times is loaded, l.306:

```
>>>    dataForTraining = pd.read_csv("Final Data set.csv")
```

A pre-processing consisting of a low-pass (l.326 – 359, Figure 4) and a high-pass (l.399 – 425, Figure 5) Butterworth filters is applied:

```python
#____________________________________________
''' LOWPASS FILTER -> Gravitational and Body acceleration '''



sample_rate = 50  # 50 Hz resolution
#signal_lenght = 50*sample_rate  # 50 seconds

Ax = dataForTraining['x-axis (g)']
Ay = dataForTraining['y-axis (g)']
Az = dataForTraining['z-axis (g)']

def butter_lowpass(cutoff, nyq_freq, order):
    normal_cutoff = float(cutoff) / nyq_freq
    b, a = signal.butter(order, normal_cutoff, btype='lowpass')
    return b, a

def butter_lowpass_filter(data, cutoff_freq, nyq_freq, order):
    b, a = butter_lowpass(cutoff_freq, nyq_freq, order)
    y = signal.filtfilt(b, a, data)
    return y


cutoff_frequency = 0.5
order = 2

Ax_grav = butter_lowpass_filter(Ax, cutoff_frequency, sample_rate/2, order)
Ay_grav = butter_lowpass_filter(Ay, cutoff_frequency, sample_rate/2, order)
Az_grav = butter_lowpass_filter(Az, cutoff_frequency, sample_rate/2, order)

# Difference acts as a special high-pass from a reversed butterworth filter.
Ax_user_beforeNoiseReduc = np.array(Ax)-np.array(Ax_grav)
Ay_user_beforeNoiseReduc = np.array(Ay)-np.array(Ay_grav)
Az_user_beforeNoiseReduc = np.array(Az)-np.array(Az_grav)
```

Fig. 4. Lowpass filter

```python
#____________________________________________
''' HIGHPASS FILTER -> Noise reduction '''



def butter_highpass(cutoff, nyq_freq, order):
    normal_cutoff = float(cutoff) / nyq_freq
    b, a = signal.butter(order, normal_cutoff, btype='highpass')
    return b, a

def butter_highpass_filter(data, cutoff_freq, nyq_freq, order):
    b, a = butter_highpass(cutoff_freq, nyq_freq, order)
    y = signal.filtfilt(b, a, data)
    return y


cutoff_frequency = 20
order = 2

Ax_noise = butter_highpass_filter(Ax_user_beforeNoiseReduc, cutoff_frequency, sample_rate/2, order)
Ay_noise = butter_highpass_filter(Ay_user_beforeNoiseReduc, cutoff_frequency, sample_rate/2, order)
Az_noise = butter_highpass_filter(Az_user_beforeNoiseReduc, cutoff_frequency, sample_rate/2, order)

# Difference acts as a special low-pass from a reversed butterworth filter.
Ax_user = np.array(Ax_user_beforeNoiseReduc)-np.array(Ax_noise)
Ay_user = np.array(Ay_user_beforeNoiseReduc)-np.array(Ay_noise)
Az_user = np.array(Az_user_beforeNoiseReduc)-np.array(Az_noise)

```

Fig. 5. Highpass filter

After that, we employ the segmentation process on the filtered data (l.1098 – 1117), in order to extract the time segments containing the gestures we want to train our model with.
This segmentation process is based on the crossings of 2 different moving averages, defined in the function `startEndGesture2`.
At this point, the array `startEndMeasuresTrainingSet` contains the points corresponding to the start and end of all the segments detected.
The first point is deleted because it didn't correspond to a gesture. Then, segments exhibiting a duration shorter than 35 points = 0.7 seconds are filtered out, because they are too short to correspond to gestures. They are more likely to be defaults from the segmentation process.
Also, we only want to delete the segments corresponding to gestures, and so between the start (pair points in the `startEndMeasuresTrainingSet` array) and the end (odd points) of a gesture, and not between two gestures (end of a movement and start of the next one).

Now, we have our final segments and we calculate the features set, in a data frame called `dfFeaturesTrainingSet` (l.1121).

Finally, we train the classifier with this features set (l.1124).

The code corresponding to this process is shown Figure 6.

```
1088
1089  #____________________________________________
1090  ''' Training set : segmentation, features, and training the classifier '''
1091
1092
1093
1094  #Input for the segmentation process
1095  nbMovingAverage = 40
1096  nbMovingAverage2 = 300
1097
1098  #Segmentation:
1099  startEndMeasuresTrainingSet = startEndGesture2(magnitude(Ax_user, Ay_user, Az_user), nbMovingAverage, nbMovingAverage2)
1100
1101  #First point didn't correspond to a movement from the subjects
1102  del startEndMeasuresTrainingSet[0]
1103
1104  #Segments exhibiting a duration shorter than 35 points = 0.7 seconds are filtered out
1105  #(We only want to delete the segments corresponding to gestures so between the start (pair points) and end (odd points)
1106  segmentsToDelete = []
1107  for i in range (len(startEndMeasuresTrainingSet) - 1) :
1108      if (startEndMeasuresTrainingSet[i+1] - startEndMeasuresTrainingSet[i] <= 35 and (i%2) == 0 and ((i+1)%2) != 0) :
1109          segmentsToDelete.append(i)
1110          segmentsToDelete.append(i+1)
1111
1112  compteur = 0
1113  for i in segmentsToDelete :
1114      if ((i%2) == 0) :
1115          del startEndMeasuresTrainingSet[i - compteur]
1116          del startEndMeasuresTrainingSet[i - compteur]
1117          compteur += 2
1118
1119
1120  #Features:
1121  dfFeaturesTrainingSet = createDFTrainingSet(Ax_user, Ay_user, Az_user, startEndMeasuresTrainingSet)
1122
1123  #Train the classifier:
1124  clf = classificationModelTrain(dfFeaturesTrainingSet)
1125
```

segmentation

features

classifier

Fig. 6. Training the classifier model

The data frame of the features calculated for each segments is created in the `createDFTrainingSet` function (l.879) and uses the feature functions defined l.593 – 805.

The `classificationModelTrain` function is defined l.921.

3.2.2 GUI and connection to the device

Just before the segmentation of the training data, and after the pre-processing part, the algorithm connects to the MetaMotionR sensor (l.1076 – 1085) and the GUI is created (l.1045 – 1071), as follows:

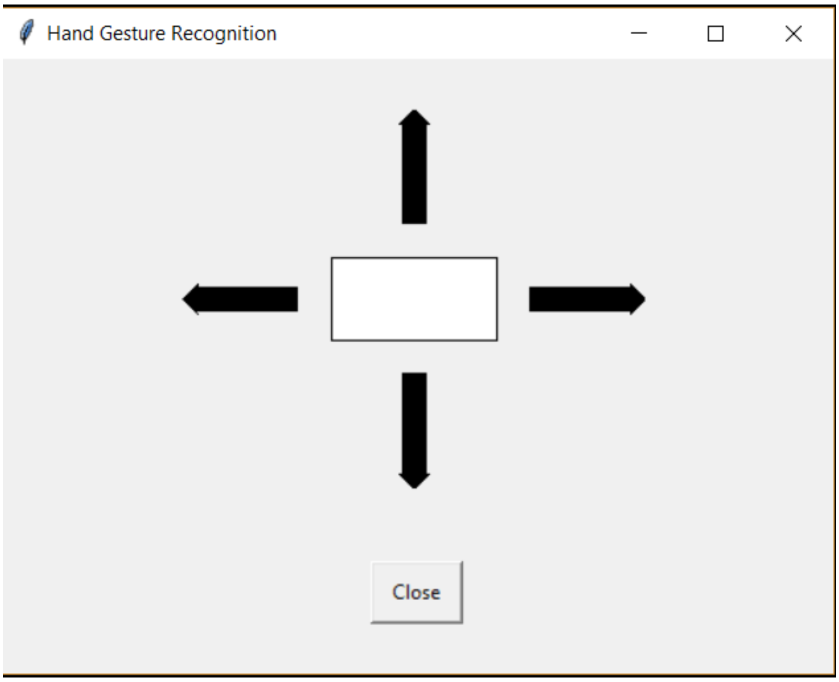

Fig. 7. Illustration of the GUI

At this point, our classifier model is trained.
We then configure the MetaMotionR sensor (l.1129 – 1168).

Right after that, the *mainloop* method from *Tkinter* is called (l. 1167). This method waits for events and continuously updates the GUI. It is important to notice that this function blocks, meaning that the execution of the python script halts here, until the GUI isn't closed by the user. The last following lines of the code stop the MetaMotionR device. Therefore, once the window of the GUI is closed by the user, everything is reset and the script ends.

The major thing of this algorithm is what follows: the execution of the script stopped at the `root.mainloop()` line, continuously updating the GUI. However, as explained for the first python script, each time that data is retrieved from the sensor, a callback function is called.
So, the script is stopped but the callback still runs.

3.3.3 Predictions on the live stream data

Now that the model is trained, that the device is connected and ready to stream data and that the GUI is ready to be updated, we can work with the live data.

The treatment of the live data doesn't occur in the *main* (at the end of the code) but in the callback function (l. 110).
This callback function called `dataAnalysisAfterCallback` runs every time that measures are retrieved from the sensor, meaning that if the sensor is streaming at a frequency of 50Hz, this function is called every 20ms.

This function is built as follows.

The new measures are saved in the `arrayDataX`, `arrayDataY` and `arrayDataZ` arrays (l.121 – 123), so these arrays contain the entire data from each one of the 3 axis.
We create a data frame composed of 3 columns containing the data from the x, y and z axis of the last '`2 * _nbMovingAverage`' points, `_nbMovingAverage` being a defined parameter.
We take the last '`2 * _nbMovingAverage`' points because this way, when calculating the moving average, the first half of the window will be composed of zeros (since the moving average can't be calculated on the `_nbMovingAverage` first points) and we will work with the second half of the window.
On those 3 signals, we apply the same pre-processing as for the training data, meaning a lowpass and a highpass filter, and then we employ the segmentation process.

However, this segmentation is different from the one used for the training data, in 2 points.
The first one is that, contrary to the training data, when calculating the moving average of the last retrieved measure from the sensor, we don't know the points that come after it. So, the moving average used here is not taken from an equal number of samples on either side of a central value, but rather on the precedent points of our sample, meaning that the moving average is not aligned with the variations in the data but is shifted in time.
The second difference is that we don't take the crossings between 2 moving averages this time, but the crossings between one moving average (again, shifted in time) and a pre-defined threshold.

The number of measures for the moving average is defined at 40 and the threshold at 0.07 (l.126 – 127).

The goal after the segmentation is to know when a gesture is detected. For that, we use 2 different arrays: `intermediate` and `arrayStartEndLiveMeasures`.
`Intermediate` returns the crossing points detected by the segmentation method in the window we're working on, and is called in every callback. Therefore, it can contain the point corresponding to the beginning of the movement for example, a various number of times. And we only want to save the beginning and end of the movement once. That's why we use a 2nd array.
`arrayStartEndLiveMeasures` contains the crossings points corresponding to the beginning and the end of a gesture.
If `intermediate` contains one point and `arrayStartEndLiveMeasures` zero, that means that a crossing point between the moving average and the threshold just appeared and corresponds to the beginning of the movement. Therefore, this point is saved to the `arrayStartEndLiveMeasures` array.
Then, the window continues to 'slide' while we retrieve new data (at this step, `intermediate` can continue to return the starting point but we don't want to save it, since it's already done).
From there, each time a point is found by the segmentation, we check if the moving average of the signal follows a downwards trend or not: if so, that means that the point corresponds to the

end of the gesture. If not, that means this point is the one that we already saved and corresponds to the beginning of the gesture.

The code is shown Figure 8. However, this part is the trickiest of the algorithm (l.149 – 183), therefore it is recommended to read the comments directly written in the python script.

```
#Segmentation using the crosses between the threshold and the moving average (taking the '_nbMovingAverage
intermediate = startEndGesturePrecedentMeasures(magnitudeLiveData, _threshold, _nbMovingAverage)

#We only work with the data if the segmentation recognized something
if (len(intermediate) != 0) :

    #If one point of the segmentation has already been detected and saved to the array (so we already dete
    if (len(arrayStartEndLiveMeasures) == 1) :

        #And another crossing in the selected window appeared on the segmentation. Therefore, we need to cl
        if (len(intermediate) == 1) :

            #Only if the point just appeared on the window of the segmentation (if the point is on the last
            if (2*_nbMovingAverage - intermediate[0] < 15) :

                #Calculates the moving average, to smooth the signal
                movAvrg = movingAveragePrecedentMeasures(magnitudeLiveData, _nbMovingAverage)

                #If the curve is following a downwards trend, the point corresponds to the end of the gestu
                if (movAvrg[len(movAvrg)-1] - movAvrg[_nbMovingAverage + int(_nbMovingAverage/3)] < 0) :
                    arrayStartEndLiveMeasures = np.append(arrayStartEndLiveMeasures, s.samples-20)
                    #The moving average is late compared to the signal since we take the precedent measures
                    #So when the moving average crosses the threshold, the actual point of the signal that

        #If the segmentation detected 2 different points. That means we have the start and end of the mover
        #Since we already saved the start of the movement, we now save the point corresponding to the end c
        if (len(intermediate) == 2) :
            arrayStartEndLiveMeasures = np.append(arrayStartEndLiveMeasures, s.samples-20)

    #If any point from the segmentation has been saved before
    if (len(arrayStartEndLiveMeasures) == 0) :

        #If the segmentation just returned one point
        if (len(intermediate) == 1) :
            arrayStartEndLiveMeasures = np.append(arrayStartEndLiveMeasures, s.samples-20)
```

Fig. 8. Detection of a live gesture

Once `arrayStartEndLiveMeasures` contains 2 points (the start and end of the movement), we have a gesture performed by the user and we update the GUI depending on the prediction given by the classifier (l.189 – 297).

3.3.4 Details about the different moving averages and segmentation functions

The moving averages and segmentation function depend if we're working with the live data or the training set, i.e. if we calculate the moving average on an equal number of samples on either side of a value or not and if we segmentate the signal using 2 different moving averages or only one moving average and a threshold.

The different functions are defined between the following lines: l.527 – 588, and shown in Figure 9 – 10.

```python

#Moving average calculated from an equal number of samples on either side
def movingAverage(magnitudeSignal, n) :
    simpleMovingAverage = []
    for i in range (int(n/2)) :
        simpleMovingAverage.append(0)

    for i in range (n, len(magnitudeSignal)) :
        sum = 0
        for j in range (i-n, i) :
            sum += magnitudeSignal[j]
        simpleMovingAverage.append(sum/n)

    return (simpleMovingAverage)


#Moving average calculated from the n precedent measures of our sample (sc
def movingAveragePrecedentMeasures(magnitudeSignal, n) :
    simpleMovingAverage = []
    for i in range (n) :
        simpleMovingAverage.append(0)

    for i in range (n, len(magnitudeSignal)) :
        sum = 0
        for j in range (i-n, i) :
            sum += magnitudeSignal[j]
        simpleMovingAverage.append(sum/n)

    return (simpleMovingAverage)

```

Fig. 9. Different moving averages functions

```python

#Segmentation based on the crossings between 2 'centered' moving averages
def startEndGesture2(magnitudeSignal, nbMovingAverage1, nbMovingAverage2) :
    simpleMovingAverage1 = movingAverage(magnitudeSignal, nbMovingAverage1)
    simpleMovingAverage2 = movingAverage(magnitudeSignal, nbMovingAverage2)
    startEndMeasures = []

    #The loop starts at nbMovingAverage2/2 because the points before that are all 0's
    for i in range (150, len(simpleMovingAverage2)-1) :
        if ( (simpleMovingAverage1[i+1] >= simpleMovingAverage2[i+1] and simpleMovingAverage1[i] <= si
            startEndMeasures.append(i)

    return(startEndMeasures)


#Segmentation based on the crossings between the 'late' moving average (with the precedent points) and
def startEndGesturePrecedentMeasures(magnitudeSignal, threshold, nbMovingAverage) :
    simpleMovingAverage = movingAveragePrecedentMeasures(magnitudeSignal, nbMovingAverage)
    startEndMeasures = []
    for i in range (len(simpleMovingAverage)-1) :
        if ( (simpleMovingAverage[i+1] >= threshold and simpleMovingAverage[i] <= threshold) or (simpl
            startEndMeasures.append(i)
    return(startEndMeasures)

```

Fig. 10. Different segmentation functions

Here is a recap of their usefulness:

- `movingAverage :` employed for the training data

Moving average calculated from an equal number of samples on either side of a central value (this moving average is 'aligned' with the data).

- `movingAveragePrecedentMeasures :` employed for the live data

Moving average calculated from the n precedent measures of our sample (so this moving average is shifted in time compared to the signal).

- `startEndGesture2 :` employed for the training data

Segmentation based on the crossings between 2 'centered' moving averages.

- `startEndGesturePrecedentMeasures :` employed for the live data

Segmentation based on the crossings between the 'late' moving average (with the precedent points) and a threshold.

3.3 Feedbacks

It may be noted that the 2 filters used for the pre-processing have been inverted: to extract the acceleration due to the motion of the hand from the gravitational acceleration in the row signal, a lowpass filter is employed (resulting on a signal representing the gravitational acceleration) and then I took the difference with the row signal to obtain the user's acceleration.
The result can be obtained quicker, employing directly a highpass filter.
The same reasoning is valid for the highpass filter, employed for noise reduction purposes.

The accuracy of the system is not as good as it is expected to be after the study led for the conference paper. And this difference could come from the fact that the segmentation process for the study was the same for the data used to train the model and the data used to test it, using the 2 moving averages crossings and calculating both moving averages from an equal number of samples on either side of the value.
In this system, we train the classifier model with this precedent process, but we use a different process for the segmentation on the live stream data.
Therefore, I advise future work to be directed towards the improvement of the live segmentation process.