PDIoT Coursework (2019-20)

BLE Step Counter

PHIPPS Robert s1601921
XIONG Jiayuan s1738850

Abstract

This project involved the prototyping and development of an IoT step counting device, working across multiple embedded development platforms and interfacing with a companion Android app allowing the user to view data collected. Communication of data was achieved by using the Bluetooth Low Energy 4.1 standard. Platforms include the Nordic Semiconductor Thingy:52 BLE sensor device and NRF52-DK MbedOS development board.

Peak detection along with various smoothing techniques were used to process data from the gyroscope data of an Inertial Measurement Unit, which allowed us to require minimal processing power and keep all data manipulation within the NRF52 for our final implementation.

Principles and Design of IoT Systems
Informatics, University of Edinburgh
January 2020

Typeset in LaTeX

1 Introduction

1.1 Project aims

At a most basic level, the project goal was to implement a simple device to track steps and relay this information to a user. It was specified that we should make use of Bluetooth Low Energy for communication of the small, low power tracking device to our Android app which displays the step count and allows the device to be controlled.

1.2 Method adopted

We opted after much consideration to use a comparatively simple peak detection algorithm, which allowed us to keep the processing requirements fairly low.

We first prototyped our algorithm using data previously collected from the "orient-android" [1] application interfacing with the Nordic Thingy:52[1] prototyping platform. Working with a Jupyter Notebook[2] we then used Pandas, SciPy and NumPy tools to visualise, analyse and implement a method for identifying steps from the data recorded. Once this algorithm was completed, we worked to port its logic across to first an Android app, which streamed data direct from the Thingy and processed it in batches before updating the user display. Finally we ported the logic one last time to run embedded on a Nordic NRF52-DK MbedOS based development board[3], which broadcast the current step count as a Generic Attribute defined within the Bluetooth Low Energy standard [2], and read by another companion Android app to display that data to the user.

The benefit of this approach meant that we could implement our algorithm entirely with Mbed OS C++, without an external dependencies on servers, and without requiring the Android device to expend battery on processing the data.

[1]Nordic Thingy:52: `https://www.nordicsemi.com/Software-and-tools/Prototyping-platforms/Nordic-Thingy-52`

[2]Jupyter Notebook project webpage: `https://jupyter.org`

[3]Nordic NRF52DK: `https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52-DK`

1.3 Summary of Results

Based on the final NRF52-DK based implementation, we managed to achieve the following levels of accuracy:

Activity type	Accuracy (%)
Walking	97%
Running	95.2%
Upstairs	87%
Downstairs	94.2%

Table 1: Summary of accuracy for NRF52-DK implementation

Based on review of other implementations and comparisons with existing mass-market step tracking devices, these are very high, especially given the limited development time, comparatively simple data processing and our success in squeezing all the logic into the NRF52-DK's embedded programming environment.

2 Literature Survey

2.1 Step tracking methodologies

The first article of relevance is one from Analog Dialogue, where it explores a 3 axis digital pedometer [3]. This (like many others) outlines a processing pipeline whereby the data coming from the sensor is first filtered, then a dynamic peak detection algorithm is applied to count the number of cycles of the pattern, from which a number of steps can be inferred. However, in this case they are making use of multiple axis of an accelerometer whereas we eventually opted to use a single axis of our IMU's gyroscope.

The next article (from Sensors) was targeting step detection using the commodity hardware within smartphones [4]. Their method involved extracting gyroscope data from the IMU and making use of what we know about the cyclic nature of walking to help improve the accuracy of their detection. Their article also covers the four common methods of step detection, being peak detection, thresholding, zero-crossing and autocorrelation. We ended up making

use of a combination of thresholding and peak detection methods in our implementation.

2.2 Data smoothing algorithms

When looking for smoothing algorithms, we came across references to the Savitzky-Golay method [5]. This somewhat vintage piece of research outlines making use of a modified version of moving average smoothing, making use of convolutional methods to fit a low degree polynomial to the dataset. It is especially beneficial in smoothing and increasing the precision of digital signals. On further investigation, it became clear that its accuracy could be improved by introducing an aspect of moving averages as a preprocessing step [6].

2.3 Existing mass-market products

Albeit far less formal, we also investigated the form factors and feature sets of existing commercial step-tracking devices, and tried to infer some idea of how they go about tracking their wearer's activity.

The most obvious example of this is the FitBit [7] series of activity trackers, which tend to be worn as a wristband. These are very popular, so it is clear that consumers are happy to wear such a device. Back in 2006, Nike released the Nike+iPod sports kit [8]. This included a small transmitter that was embedded in a specially compatible shoes, and communicated with various Apple devices as well as a Nike+ Sportwatch. With improvements in the sensors within newer devices, as well to avoid the inconvenience of having to embed a module in your shoe (which needed charging), Nike opted to release a new version of their system as an app making use of just the sensors in the new iPhones and iPods, negating the need for a dedicated tracking chip. This failure shows that shoe-mounted sensors are unlikely to be popular with consumers, and it is possible to obtain accurate enough results from sensors worn in more convenient locations.

In most of these products, the data processing and actual step tracking is carried out on-board, as they generally have some form of display to show the current steps even if the paired smartphone is disconnected, and could all gracefully handle Bluetooth disconnections for whatever reason.

3 Methodology

3.1 Development pipeline

Following the structure given to us in the tutorial schedule, we effectively developed three very similar step tracking systems, starting with convenient, high-level and high performance systems (Python and Jupyter Notebook), moving to Java on Android and finally to C++ on MbedOS. This allowed us to revise streamline our algorithm, while still allowing easy prototyping during the concept development stage.

Our first implementation made use of batch data recorded from the "Orient-android" app [1], recorded in CSV format and processed offline from within a Jupyter Notebook environment, fitted out with a full suite of data analysis tools like SciPy, NumPy and Pandas. We also contributed to the "pdiot-data" repository[4] which was used by the whole cohort to exchange data recorded from their testing.

Once we had prototyped a step detection algorithm to run offline, we modified the existing "orient-android" app used to record gyroscope data to the CSV files for analysis, and added a port of the step detection pipeline into the Java source code. This allowed us to process data and return a step count on-the-fly, and display the result in near to real time on the Android device. This still streamed all IMU data from the Thingy:52 unprocessed, and performed all analysis within the app, which would have a non-negligible additional battery draw for the end user. Due to how we had the app configured, it also required the app to be open and running in the foreground for it to continue to keep track of steps, and would ignore any steps taken for periods when the phone lost Bluetooth connection to the Thingy:52.

The final iteration was another port and revision of that initial algorithm, this time to the C++ en-

[4] "pdiot-data" - shared motion capture data from the "orient-android" app: `https://github.com/specknet/pdiot-data`

vironment provided within MbedOS[5], the embedded RTOS running on the Nordic NRF52-DK development board. To start with, we experimented with the provided MPU-9250 Inertial Measurement Unit by having the NRF dump live sensor values out over serial console to either a PC or an Android serial console app[6]. Once we had reliable gyroscope data from the IMU, we then began to port over the step detection algorithm, finding similarly functioning libraries to replace those used on the other versions of the app. Finally, once we had the step count being printed over serial, we modified an existing BLE GATT server example implementation [9] to carry a simple 32 bit payload, to be used as an integer containing the step count.

3.2 IMU location and selected data

To get started we plotted data from all of the sensors and all of the tested sensor locations (Figure 3). From looking at this data it is clear that the dataset with the most obvious cyclic pattern is the gyroscope z-axis, both on the foot and on the wrist. As the foot is a fairly awkward location to place a sensor, and we have already seen success in tracking steps with wrist-mounted devices, it was decided to make use of the gyroscope z-axis on the wrist (Figure 2. The use of the magnitude calculated from all three gyroscope axis was also investigated, as it would allow performance to be maintained even if the device were not to be mounted in the expected location, however as it is to be wrist mounted, there is very little likelyhood of the orientation being incorrect.

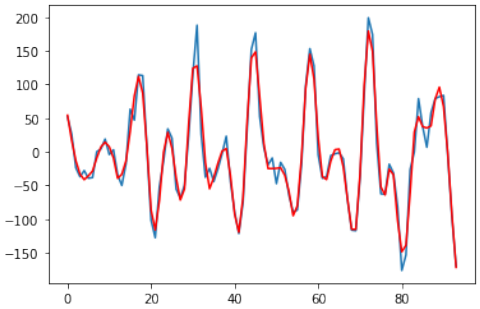

Figure 1: Gyroscope z-axis data before and after Sav-Golay filtering

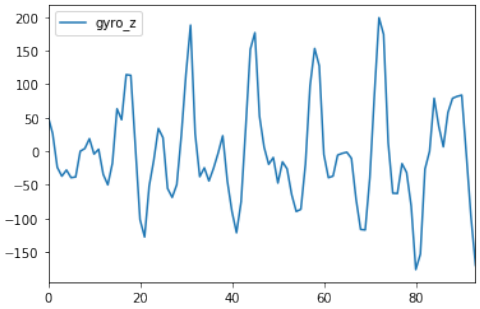

Figure 2: Raw gyroscope z-axis data from wrist mounted sensor

[5]MbedOS: `https://www.mbed.com/en/platform/mbed-os/`

[6]USB Serial Console (Google Play Store): `https://play.google.com/store/apps/details?id=jp.sugnakys.usbserialconsole`

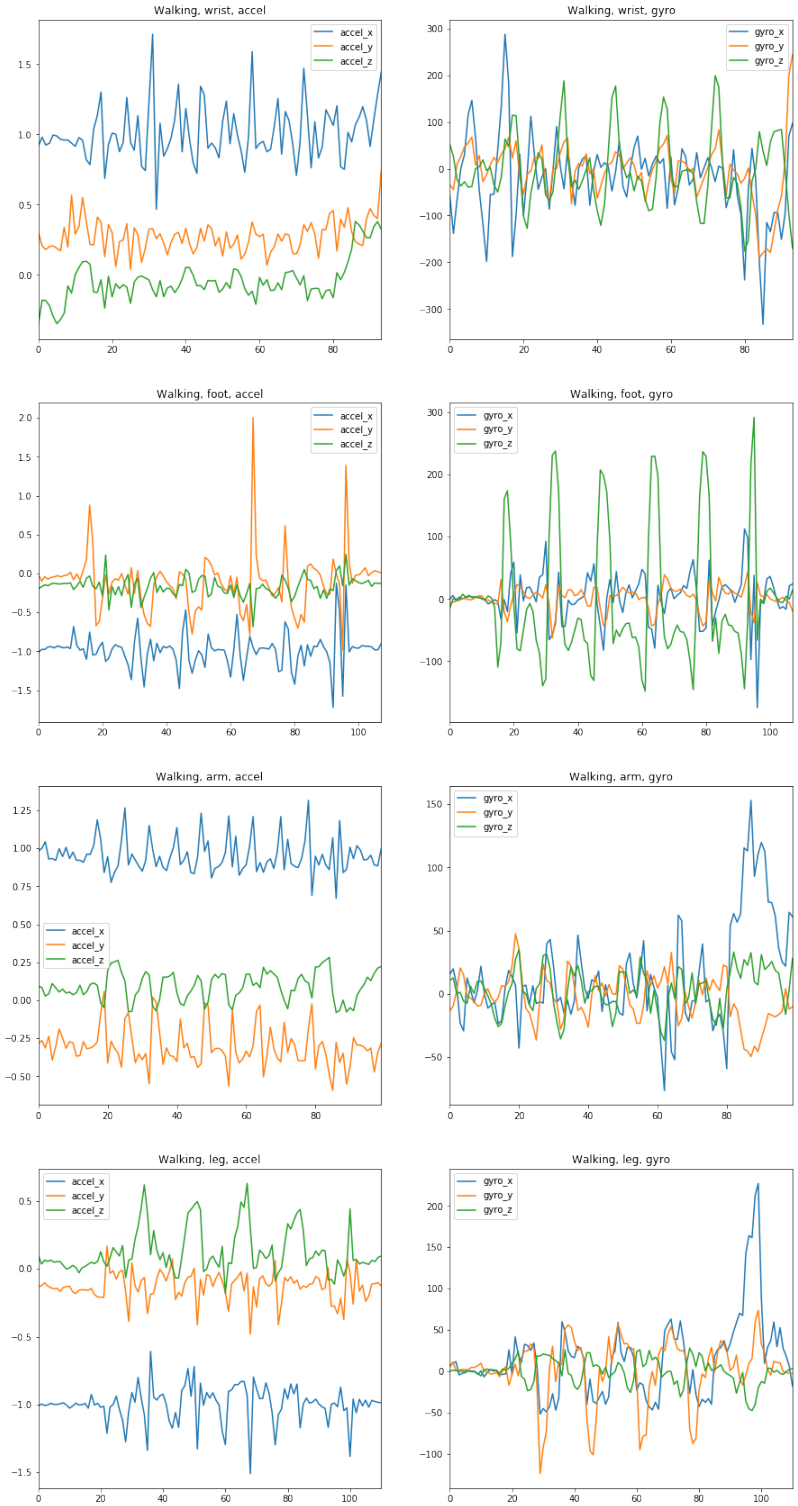

Figure 3: Comparison of sensor locations and axes

3.3 Step detection algorithm

Multiple options were investigated to allow us to achieve this goal, including the use of cloud compute resources running machine learning frameworks such as TensorFlow[7], however we were very keen to ensure we didn't over-engineer our solution. After experimenting with methods, it became clear that it should be possible to achieve fairly respectable accuracies using comparatively very straightforward processing. This would also allow us to keep as much of the processing as possible local, and eventually contained within the tracker itself.

The system processes sensor data in chunks of 100 samples, this minimises complexity of implementation, however does occasionally result in artefacts if there is a sudden change in activity on the boundary between two sampling windows. Based on our testing however, this doesn't seem to adversely affect the accuracy.

First, to optimise the batch of samples for processing, we use simple thresholds to trigger different smoothing profiles, making use of Savitzky-Golay filtering. At this stage we also identify whether there is enough movement in the data to be processed, otherwise the batch is ignored.

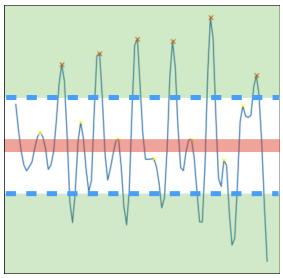

Figure 4: Illustration of peak detection with thresholding

Once the data has been filtered (Figure 1), we make use of one of two signal detection algorithms. For the Android-based Thingy:52 implementation we make use of a smoothed Z score with moving average and standard deviation based thresholds. This is effectively a peak detection algorithm, but with dynamically calculated thresholds. It is a slightly modified version of the algorithm set out by Jean-Paul in a StackOverflow response titled "Robust peak detection algorithm (using z-scores)" [10].

For the NRF52-DK implementation, a simple thresholded peak detection was used, with fixed thresholds calculated ahead of time when each block of 100 sensor samples is passed to the subroutine. This seemed to have a far lower tendency to overfit, and had very similar accuracy with the benefit of running far quicker on the embedded hardware.

It also works to ensure that there is a zero-cross between peaks by post-processing the signals output from the algorithm. Finally, there is a hard-coded minimum deviation on both implementations to ensure that periods of no activity do not result in noise being amplified and counted as steps.

In the spirit of IoT and SpeckNet, we decided it was far more worthwhile to target our implementation to replicate this on-board processing as much as possible, making use of the processing power we have at the "edge" as opposed to centralising this onto a cloud hosted backend. This also greatly reduced the amount of data being transferred over the Bluetooth connection, and allowed our implementations to run offline without an internet connection.

[7]Google TensorFlow Project: `https://www.tensorflow.org`

```
def mainloop():
  gyrodata.append(getgyrodata())
  if len(gyrodata) > COUNT_WINDOW:
  # process data
  (min, max, range) = getminmaxrange(gyrodata)
  if range < minthresh:
    gyrodata = []
    return

  if range < lowthresh:
    filtered = savgolay(gyros, walklag, walkdegree)
    signals = simplepeakdetection(filtered, walkparams)
    _globalstepcount += transitions

  else:
    filtered = savgolay(gyros, runlag, rundegree)
    signals = simplepeakdetection(filtered, runparams)
    _globalstepcount += transitions

def simplepeakdetection(data, avg, range, threshmult = 0.80)
  threshold = range*0.5*threshmult
  threshold = 1.0 if threshold < 1.0
  iqmin = avg - threshold
  iqmax = avg + threshold

  signals = [];
  for sample in data:
    if (iqm < sample < iqx) then signals.append(0) # within threshold
    else if (sample < iqm) then signals.append(-1) # below threshold
    else if (p > iqx) then signals.append(1) # above threshold

  # find transitions:
  lastsignal = 0
  count = 0
  for signal in signals:
    if signal is not lastsignal then count += 1
    lastsignal = signal

  return count;
```

Figure 5: Pseudocode for basic step detection

3.4 BLE GATT communication profile and wireless

Originally, data was obtained from the Thingy:52 using its standard BLE attributes, as this board was running factory firmware. This meant that we were reading in raw sensor information.

Once the NRF52-DK was successfully counting steps and outputting the data over serial, we then needed to find a way to pass this data to the companion Android app. Unfortunately there is no standard attribute defined within the GATT specification [2], so for the purposes of our application the `RUNNING_SPEED_AND_CADENCE` device UUID was repurposed. It is not an ideal fit, as we are not broadcasting the rate, but the cumulative step count value, however it seemed the closest option within the specification. The Bluetooth stack is based around the Heart Rate service example provided within the MbedOS documentation [9].

This initially broadcast data using the Heart Rate Monitor characteristic, however this used a single 8 bit word for the data. This would only have offered a maximum value of 256 (`2^8`), which while perfectly appropriate for heart rates in BPM, would be very quickly reached in our use case. It was modified to use two 16 bit words, and the generic "Analog" characteristic with ID `0x2A58`, this gives us a theoretical maximum value of 65,536 (`2^16`) for our characteristic, which seems more appropriate, although could still be rather low in some use cases. The characteristic is defined in Figure 7.

The "runner" service for the sensor is scheduled to run the update routine every 80ms, which triggers the sensors to be polled, and the BLE characteristic is updated every time the data buffer contains 100 entries. This means that the step count is updated approximately every 8 seconds.

3.5 Software and firmware architecture

For the final implementation running on the NRF52-DK, all processing of data is carried out within the Mbed OS on the development board. Only the current step count is broadcast over BLE. This means the companion Android application simply reads the characteristic's value, and displays it to the user.

The NRF-52 firmware is structured around the following files:

- `main.cpp`: Entrypoint to the firmware
  - `main()`: Creates instance of the BLE framework, initialises the IMU and attaches the runner instance to the BLE service.
  - `StepCountRunner`: defines UUID for the GATT service and creates event queue/scheduler.
  - `updateSensor()`: reads data from IMU sensor and triggers `processBuffer()` once it has read 100 (or `COUNT_WINDOW`) samples. Also flashes LED2.
  - `processBuffer()`: performs smoothing and signal detection on current buffer of gyro values.
  - `simplepeakdetection()`: performs smoothing and signal detection on current buffer of gyro values.
- `SGSmooth.cpp`: library to implement Savitzy-Golay smoothing
  - `sg_smooth(vector, width, degree)`: performs smoothing on array provided as a vector of floats with moving average of window width and fitting to a polynomial of the degree given
  - This library was sourced from the Linear Algebra Toolkit created by GitHub user `thatcristoph` [11]
- `stepcountservice.cpp`: definitions for the BLE service

  Based off of the Heart Rate Monitor service from the MbedOS examples repository and modified for use in this project [9]

Figure 6: NRF52-DK firmware file structure summary

```
/* representation for the bytes of the step count characteristic. */
struct StepCountValueBytes {
    static const unsigned MAX_VALUE_BYTES  = 2;
    /* no Flags, and up to two bytes for step count. */

    StepCountValueBytes(uint16_t stepCount) : valueBytes() {
        updateStepCount(stepCount);
    }

    void updateStepCount(uint16_t stepCount) {
        // split 16bit unsigned int into two 8bit unsigned ints
        valueBytes[0] = (uint8_t)(stepCount & 0xFF);
        valueBytes[1] = (uint8_t)(stepCount >> 8);
    }
}
```

Figure 7: Definition of step count value BLE characteristic bytes

4 Results

4.1 Real-world testing methodology

To test the performance of the trackers, we wore them and walked, ran, climbed and descended stairs around Appleton Tower and in various other buildings, keeping track of the count ourselves and comparing that result to the measured steps by each implementation. With the testing for stairs, we included any small landings and turns in the count.

Additionally, we tested scenarios such as leaving the mobile device away from where we were walking to cause a Bluetooth connection failure, allowing us to test how the implementations would handle such an event. In these tests, the NRF52-DK based implementation obviously was superior, as it continued to count steps even with a failed connection, as all processing was on-board. The Thingy:52 implementation required an app restart after losing connection, which caused it to lose the current step count as well as any data from the period when the connection was lost.

4.2 Tracking evaluation

A summary of the results from our testing is available in Table 2, with the raw data used for those calculations displayed in Table 3.

We achieved a consistent accuracy in the region of 90%, with up to 97% for walking with the NRF52-DK implementation. Both implementations tended to under-count steps, which implies that with further tuning of detection parameters we should be able to bias the tracking such that the average accuracy is centered around 100%.

For the Thingy:52 implementation, it particularly had trouble with running, this is likely due to the lower sampling rate, caused by the bandwidth limit on the BLE connection being used to stream sensor data to the companion app. This lower sample rate effectively acted as a low-pass filter, which with higher-frequency activity (running) caused some steps to be ignored.

Stairs were also a challenge, due to the less cyclic nature of motion, as well as heavier steps causing high-frequency noise in the data. Thankfully, the magnitude based activity detection and automatic calibration of point detection thresholds in the NRF52-DK implementation managed to improve accuracy considerably for these activities.

Activity	**Actual steps**	**Number of tests**	**Median counted**	**Accuracy** (%)
Thingy:52 and streaming gyro data to Android				
Walking	100	5	92	92%
Running	100	5	84	84.8%
Ascending stairs	100	5	106	94.4%
Descending stairs	100	5	92	90.8%
NRF52-DK with all on-board processing				
Walking	100	5	97	97%
Running	100	5	101	95.2%
Ascending stairs	100	5	90	87%
Descending stairs	100	5	98	94.2%

Table 2: Accuracy of both implementations of the step tracker

Activity	**Actual steps**	**Runs**	**Counted steps per test run**
Thingy:52 and streaming gyro data to Android			
Walking	100	5	98, 96, 84, 90, 92
Running	100	5	80, 84, 82, 90, 88
Ascending stairs	100	5	106, 104, 106, 102, 110
Descending stairs	100	5	92, 92, 94, 88, 88
NRF52-DK with all on-board processing			
Walking	100	5	96, 101, 97, 96, 102
Running	100	5	103, 93, 101, 92, 105
Ascending stairs	100	5	85, 120, 92, 90, 88
Descending stairs	100	5	105, 91, 98, 93, 106

Table 3: Raw test results

4.3 User experience (UX)

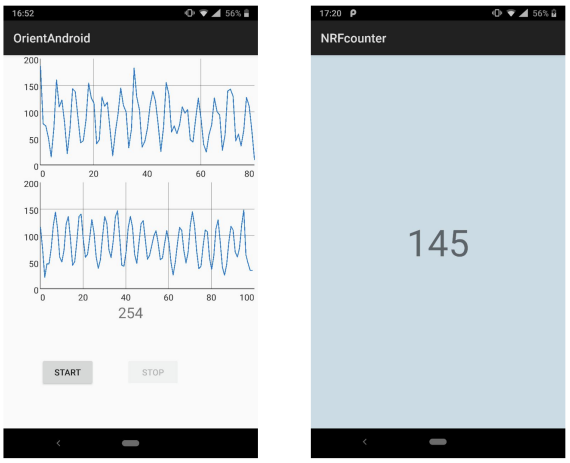

Figure 8: Thingy:52 (left) and NRF52-DK (right) companion applications in normal use

For the purposes of this project, the apps developed are incredibly bare-bones (Figure 8), however thanks to that, maintaining a simple UI was easy. In the case of the NRF52-DK implementation, we simply have a counter displayed, along with colour coded backgrounds to show connection status: orange for connecting, blue for connected and red for disconnected. The Thingy:52 implementation included live graphs of the gyroscope data, which were helpful during testing and development. These were excluded from the later NRF52-DK app, as this information was not broadcast over BLE.

A major area of UX that would need to be improved is the Bluetooth pairing process, as both apps require that the device's MAC address is hard-coded into the application at compile time. Implementing some form of simple discovery flow for new users would be a very high priority were these to be developed further.

Additionally, neither of the apps are able to maintain a connection unless the app is running and in the foreground of the Android companion device. In the case of the NRF52-DK implementation, this is less of an issue as it is able to continue tracking independently, but for the Thingy:52 based app, this is a major issue as it is only able to track while the app is running and connected.

5 Conclusion

Based on our testing, it appears that there is not a huge amount to be gained from too much additional processing on the data, as thanks to our chosen IMU positioning, we receive a very clean and obviously cyclic dataform. This has the added advantage of minimising processing power requirements on both the companion smartphone and within the embedded processor. This should contribute to a higher battery life.

An interesting advantage of the NRF52-DK implementation was that we were able to achieve a higher level of accuracy simply due to the fact we were able to poll the gyroscope at a higher rate, as we were not constrained by the bandwidth of the Bluetooth LE 4.0 connection to the companion app.

5.1 Caveats and potential improvements

Given the development process, the device has only received intensive testing from us, and there is a fairly high chance that we have overfit the detection algorithms to our specific walking styles. A potential extension would be to make use of cloud based machine learning to automatically tune the detection parameters to best fit the individual's gait, and to allow us to improve the robustness of our activity detection.

5.2 Reflection

This project was an interesting exploration of a different side of IoT, as in most cases the focus is on the "internet" part of the implementation, with very little thought given to the hardware actually gathering the data and reacting to the world. This was our first time working with the Mbed development environment, and experience of working with such limited resources was motivation to ensure that our solution was not over-engineered, as opposed to the often used

method of throwing more computation power, data and lots of machine learning at the problem, which could probably have been solved just as well with some well-thought-out statistics, as in this case.

The use of low bandwidth BLE as opposed to TCP/IP for communication encouraged us to embrace "edge computing", by trying to perform as much (or all) of the data processing as close to the sensors as possible. Again, this was using far lower-level programming languages, but by progressively working through from a high-level Python based toolchain, down to the C++ running on the NRF52-DK, it was surprisingly easy to maintain functionality across platforms.

References

[1] A. Bates. Orient-android. [Online]. Available: https://github.com/specknet/orient-android

[2] (2019, December) Gatt characteristics. Bluetooth SIG Inc. [Online]. Available: https://www.bluetooth.com/specifications/gatt/characteristics/

[3] N. Zhao, "Full-featured pedometer design realized with 3-axis digital accelerometer," *Analog Dialogue*, vol. 44, no. 2, June 2010. [Online]. Available: https://www.analog.com/media/en/analog-dialogue/volume-44/number-2/articles/pedometer-design-3-axis-digital-acceler.pdf

[4] X. Kang, B. Huang, and G. Qi, "A novel walking detection and step counting algorithm using unconstrained smartphones," *Sensors (Basel)*, vol. 18, no. 297, January 2018. [Online]. Available: http://dx.doi.org/10.3390/s18010297

[5] A. Savitzky and M. J. E. Golay, "Smoothing and differentiation of data by simplified least squares procedures," *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1638, July 1964.

[6] H. Azami, K. Mohammadi, and B. Bozorgtabar, "An improved signal segmentation using moving average and savitzky-golay filter," *Journal of Signal and Information Processing*, vol. 3, pp. 39–44, 2012. [Online]. Available: http://dx.doi.org/10.4236/jsip.2012.31006

[7] (December, 2019) Fitbit activity trackers. [Online]. Available: https://www.fitbit.com/uk/home

[8] Wikipedia. Nike+. [Online]. Available: https://en.wikipedia.org/wiki/Nike%2B

[9] ARMmbed, "mbed-os-example-ble: Ble demos using mbed os and mbed cli." [Online]. Available: https://github.com/ARMmbed/mbed-os-example-ble

[10] J.-P. van Brakel, "Robust peak detection algorithm (using z-scores)," *StackOverflow*, March 2014. [Online]. Available: https://stackoverflow.com/a/43512887

[11] thatcristoph. (2012, December) Linear algebra "toolkit". [Online]. Available: https://github.com/thatchristoph/vmd-cvs-github/tree/master/plugins/signalproc