System Engineering: Pedometer on pebble watch

Antoine Albertelli Eloi Benvenuti Florian Kaufmann

October 2016

1 Introduction

In the context of our System Engineering class we had to design and code a pedometer for a pebble smart watch. This document explains our design process and presents the final results

2 Requirements

The need tied to the user of pedometer are the following:

- Have a mean to count his footsteps over the day.
- Have a mean to easily read his current footsteps for the day.
- Have a mean to access an history of his footsteps over the past days.
- Must not experience discomfort due to the use of the device.

From those needs we derived the following functions:

1. Acquire the relevant data for footsteps computing.
2. Process those data to compute the number of footsteps.
3. Update the number of footsteps frequently (every 1-2 seconds).
4. Display the number of footsteps for the current day.
5. Offer a mean to reset the current number of footsteps.
6. Is able to count footsteps in the background.
7. Offer a mean to look at the history of footsteps for the past week.
8. Do not reduce the battery life of the pebble watch noticeably.
9. Do not result in an overheating of the watch.

3 Algorithms

We tried several approaches in order to find a precise enough algorithm. We implemented prototype in Matlab or Python and ran them on the sample data. We then compared the number of counted step against the actual number of steps in order to rank them.

3.1 Frequency domain

We briefly explored the idea of processing the data in the frequency domain. The approach we used was to cut the frequencies above and below a certain signal and look for peaks in the frequency spectrum. However we rejected this idea because of the complexity and computational costs compared to the time domain algorithm (see below).

3.2 Time domain

We implemented 3 time domain algorithms and compared them against each other.

Here is a quick description of their different strategy:

Algorithm 1 The 1st implementation looks at the data by blocks of 1 second. If the peak-to-peak difference was bigger than a set threshold the step counter is incremented. This algorithm had good performance on the test data but was discarded because it was too sensitive to the walking pace: If multiple steps fit in the data block only one step would be counted.

Algorithm 2 The 2nd implementation counts the steps using the data of the three axis of the accelerometer. Its strategy, applied separately to each axis, was to find the average value of the data in the block It would then count one step every time the raw data crossed the average with a negative slope. In order to reduce miscounts due to the noise, an hysteresis was applied.

This algorithm performed very badly, overshooting the number of steps by a factor 2 at least. The averaging between the 3 axis wasn't helping, since the 2 worst-performing axis were only worsening the count of the "best" axis.

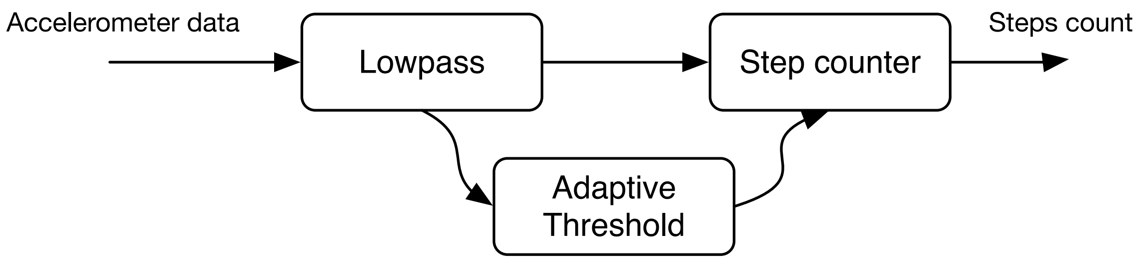

Figure 1: Final version of our algorithm

Algorithm 3 The final algorithm (fig. 1) count the steps using only 1 axis. It first applies a lowpass filter on the data to smooth out the noise. It then compares the data to

a threshold calculated using an average over the last N points. When the data crosses the threshold (with an hysteresis), one step is counted. Compared to the previous algorithm, this result in a threshold reacting more smoothly which improves the precision.

Its performance is good: on test data, it is within 10% of the correct value. Moreover, it is easy to implement correctly, doesn't consume a lot of processing power nor RAM and only has a few parameters to tune (lowpass cutoff, threshold history size and hysteresis). It is also pretty insensitive to variations in physical parameters across users.

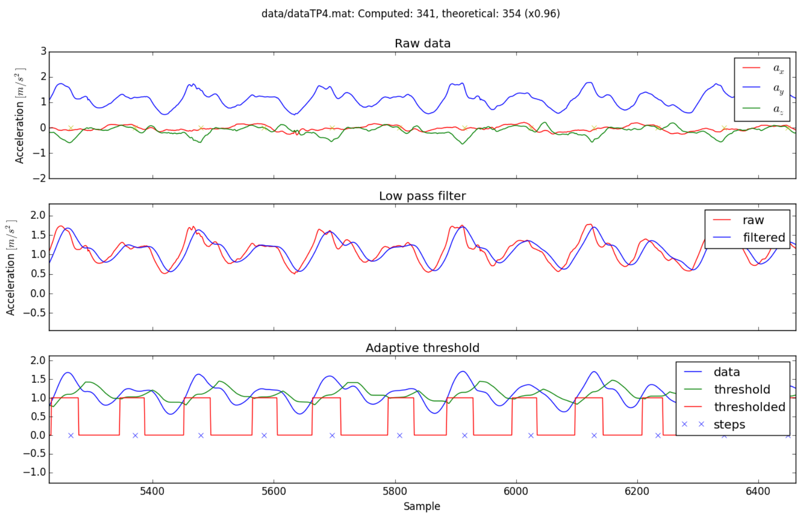

Figure 2: Processing steps of our final algorithm. First, the raw data (top) pass through a low pass filter to reduce noise (middle). Finally the threshold is compared to the filtered data and a step is counted on every crossing (bottom).

4 Implementation notes

The processing code was first written in Python, then in C. The code was written to be very portable by separating platform-dependent code from processing functions. This allows the code to be tested on a development machine, which increases development velocity. We also wrote unit tests for the processing code (under the "tests" folder). This prevents bugs and reduces time spent debugging on the Pebble.

Being able to count the steps while using other functions of the watch was a critical function (see section 2). To do so, the processing algorithm is running in a background worker, while the pedometer user interface runs in an application. This separation allows the user to go on the main window of the application only to check his current number of steps and then leave the app to use other functions.

To communicate between the worker and the application a message queue is used[1]. We implemented three messages for our use cases:

1. The worker sends the step count and the current state (turned on or off) to the application. This is used to update the screen.

[1]`https://developer.pebble.com/guides/events-and-services/background-worker/`

2. The application sends one message to toggle the state between on and off to the worker.

3. The last message type is sent from the application to the worker to reset the step counter to zero.

5 Interface design

Figure 3 shows a screenshot of the interface when the pedometer is running. We wanted to display the minimum amount of infos while still providing the user with the necessary ones in order to instantly know how much steps he had already walked and the functions of the different buttons. In order to communicate them we used commonly known symbols such as small "play" and "pause" icons for the top button and circling arrows for the bottom one, used to reset the count.

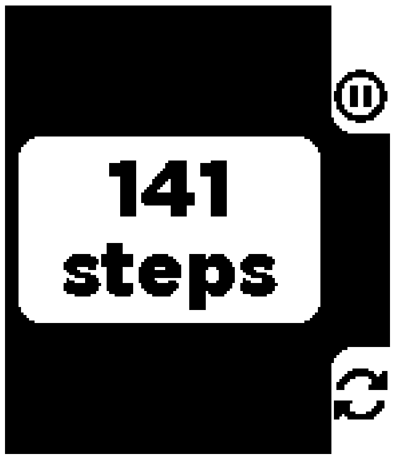

Figure 3: Screenshot of our interface showcasing the reset and play/pause button

6 Possible improvements

Looking back at our requirements (section 2) we can see that our current implementation could be improved. For example, data logging could be implemented using Pebble's Storage API[2], but we did not have enough time to design a user interface around it.

The battery consumption of the device should also be measured and optimized. According to the Pebble Developer guide[3], lowering the display refresh rate as well as the accelerometer batch size could help here.

[2]https://developer.pebble.com/guides/events-and-services/persistent-storage/
[3]https://developer.pebble.com/guides/best-practices/conserving-battery-life/