Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Android Workout

Semester Project

Raffaele Lauro

rlauro@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Klaus-Tycho Foerster, Jara Uitto
Prof. Dr. Roger Wattenhofer

June 2, 2014

Abstract

Today's smartphones have several sensors incorporated within them. The Android platform can support up to thirteen sensors, including accelerometer, gyroscope, pressure and magnetic field. For this project, we wanted to see if one of those sensors could be useful in order to count how many repetitions a user does, when he is doing some exercise at the gym or at home. After several analysis steps, we realized that, using the magnitude of the acceleration vector recorded by the accelerometer could fulfill our goal. The application we implemented can count all the repetitions a user does during a specific exercise, if it is a simple movement, and if they are good enough to be counted.

Contents

Chapter 1

Introduction

1.1 Motivation

During the last few years, going to the gym became one of the most popular hobbies. If one want to lose weight, strengthen the muscles (in order to be fit for another sport), shape the body or just spend some time with friends and train together, the reasons are many. But the purpose or final design is always the same: improve ourselves. We can compete with others, to see who can do more push ups or lift more weight. We can write in a piece of paper or a document our achievements, in order to do better next time. Or we could ask a device to do it for us. We can find several devices in the market. Let's enumerate a couple of them.

1.1.1 What already exists

One of the most popular of those devices is probably the pedometer. Simply attached on the wrist or the ankle, it can count our steps, deduce a distance, compute the speed. The sensors that can be used are the accelerometer (counting a step) or the GPS (to compute a distance directly). The pedometer is, for instance, implemented in the iPod Nano (with a Nike+ App) of Apple [1].

A more elaborated device is the Atlas Fitness Tracker financed by Indiegogo [2] and developed by Atlas Wearables [3]. It can identify the exercises, count the repetitions, deduct the calories and more. But it is also pretty expensive. Isn't there a way to do almost the same thing with something cheaper? Or maybe something that we already have, for other purposes. Like a smartphone.

1.1.2 The Convenience of a Smartphone

Our society requires that we are reachable at any time. That's why almost everybody has a cell phone with him at any time and at any place. Moreover, a lot of people have a smartphone. When we go to the gym, some of us just leave the phone in a locker during their workouts, but other already use it to listen

to music. It would be very convenient if there is an application that can count and record our workout. This way, we could simply check results of previous workouts and let the application do the counting and also verify that we do the exercises properly.

What makes the smartphone an interesting choice is the list of sensors it has. A bunch of applications use the movement of the phone as a controller. One of the most famous is Doodle Jump[1]. Some smartphones have more than just movement sensors (some can measure temperature, air pressure, etc), but we are not interested in them here. The two sensors that we will look at are the accelerometer and the gyroscope. The first measures the acceleration given to the device (gravity included) and the other measures the rotation given to the phone.

Our goal for this project was to develop such an application. Almost all smartphones have an accelerometer or a gyroscope (or both) in it, that we can use to record a motion. And the fact that we have almost every time the smartphone with use was a major motivation to start the project. We will describe in this document the development of the application.

1.2 Rest-to-Goal Movement

The application handle mostly what we called "Rest-to-Goal" movements. This name came from the description of the movement itself. We start from a rest position (stand-up for squats, plank position for push-ups, etc) and make a simple movement up to a goal position (sit position for squats, prone position for push-ups, etc) and back to the rest position again (and so on).

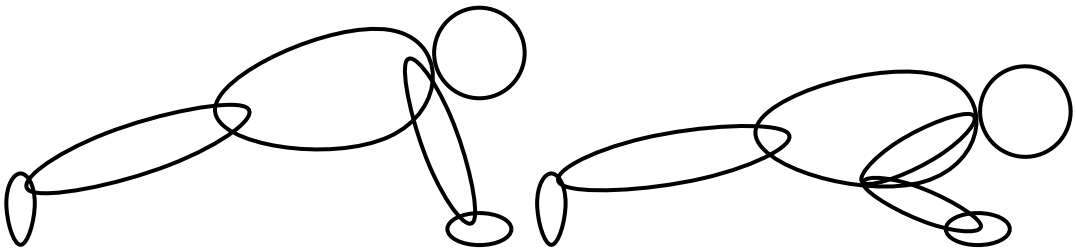

Figure 1.1: Push up on plank position (left side) and on prone position (right side).

Here is a non exhaustive list of rest-to-goal exercises:

- Push-up
- Squat

[1] Doodle Jump on AppStore or Google Play

- Pull-up
- Dips
- Biceps Curl
- Shoulder Curl
- ...

The list is pretty large. It can include almost every exercises that we can do on a machine at the gym or using dumbbells. For now on, we will refer to this kind of exercise as *rest-to-goal movement*.

1.2.1 First Intuition on Movement Analysis

The first question we may ask is the following: *Could we make an algorithm that could count rest-to-goal movements?* The answer is yes. Assuming we have an ideal accelerometer[2] within a device, placed somewhere, so that the device will sense a motion (it depends on the kind of exercise). In the rest position, the sensor does not record any movement. The user then starts and it records, assuming he will continue at a constant speed, an acceleration as a Dirac function[3] with amplitude a (the length of the acceleration vector) and shifted to the current time. As we assume the speed of the user to be constant, the acceleration will be null, until he reaches the goal position and continues backwards up to the initial position. At this time, we will have a new Dirac function with amplitude $-a$ and shifted to the current time. This intuition is depicted in Figure 1.2.

Using this intuition, the number of peaks (positive or negative) is equal to the number of repetitions done by the user. We will see in Chapter 2 how this intuition is applied in the real case.

1.3 Overview of the Thesis

The main part of the document (Chapter 2) will describe the analysis that was made from the very beginning (First experiments) to the end, including observations and conclusions that were used in the practice.

[2] Assumed to be perfectly precise and correct

[3] A Dirac function is a special kind of time function, which is equal to zero at all points of time except on $t = 0$. One can shift this function with a constant, to have all values 0 except on a specific point of time.

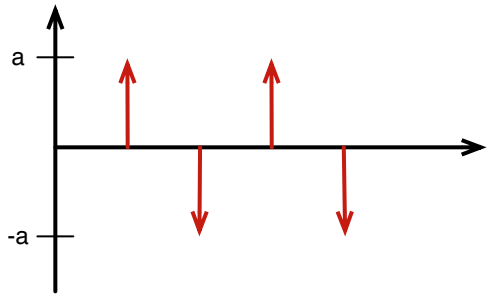

Figure 1.2: How an ideal movement recorded could look like. Each red arrow correspond to a Dirac impulse, which means that the value of the function is equal to 0 except under one of those arrows, where the value is the length of the arrow (a or $-a$, here).

In next chapter (3), we will see how all the observations made in the analysis are used and how the application is designed, in order to be as efficient as possible. A sketch of the algorithms used will be described, mostly using state machines.

We will see (in Section 3.2) that the testings gave pretty good results. We gave 0% of false negative, meaning that, if a repetition is considered as "bad" by the application, it implies that the user did it badly. The false positives are not so few. We can still say that a really bad repetition (half the repetition) is never counted. However, a slightly bad one (roughly 75% of the movement) will almost always be counted.

Chapter 2

Movement Analysis and Observations

In this chapter, we will explain the analysis we made in order to decide what kind of algorithm could or could not be useful, for recognizing a rest-to-goal movement. We started with some simple experiments on push-ups and squats and look at the accelerometer and gyroscope data recorded during the exercise. We then did the same with another subject and analyzed the recorded data. We will then come out with a first solution to count the repetitions of a given exercise. The next section will be about the principle component analysis, that we used on our data. Finally, we will describe how we handled with the noise that can be recorded during the exercises.

2.1 First Analysis on Push-ups

Our first goal was to see if there is a difference between a "proper" and a "bad" exercise. For a rest-to-goal movement, we can consider a repetition to be bad (and therefore not to be counted) if one does not reach the goal position or do not go back to the rest position. For the push-ups, for instance, a repetition is bad when one does not go deep enough or, after having reached the prone position, do not go back high enough to the plank position[1]. Figure 1.2 shows how such a push-up should look like.

Our first experiment consisted in this: record 10 "proper" push-ups, 10 push-ups, where the goal position is not reached, and 10 push-ups, where, after that the goal position is reached, the rest position is not. The data has been recorded on .csv files, for both the accelerometer and gyroscope and for each exercise (total of 6 files). Let us first take a look on the accelerometer data.

[1] This is just how we defined a push-up. But we can consider, during a workout, to make a different version of that exercise doing just half of the movement, when we go back to the rest position (after one repetition). This version is not "bad", but it is different to our definition of "proper" push-ups in this exercise.

2.1.1 Accelerometer

The accelerometer records the acceleration that is applied to the device at a fixed point of time. This acceleration is computed in the 3-dimensional axes x-y-z, like depicted in the next figure.

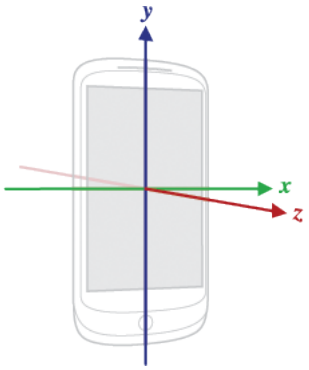

Figure 2.1: *Coordinate system (relative to a device) that is used by the Sensor API.* Source: Android API Guide, *Sensor Coordinate System* [4]

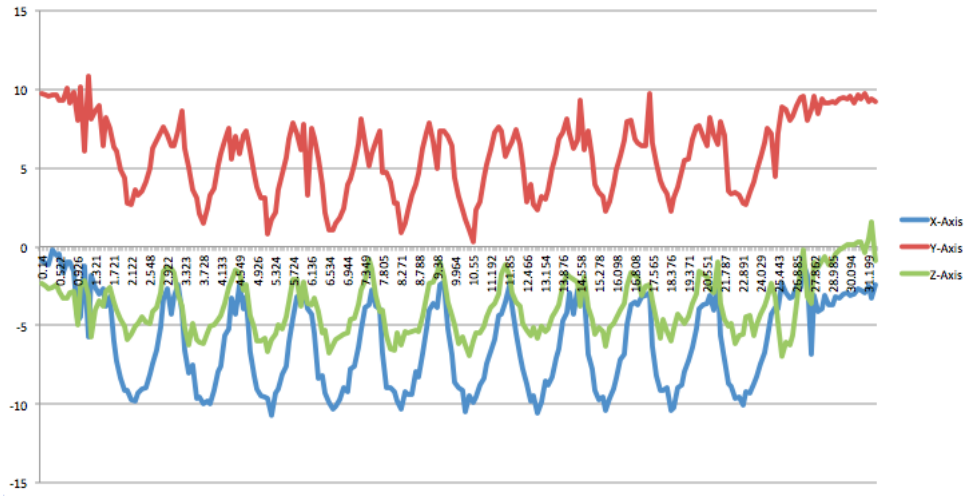

Figure 2.2: The accelerometer data of a "proper" push-up. It represents the acceleration applied to the phone in the x-, y- and z-axis against time.

In Figure 2.2, we can see what the accelerometer recorded for the "proper" push-up. For this experiment, I was the subject[2] (1.73 meters tall). From now on, we will mention me as Subject 1. The first thing to notice in this plot is the position of the y-axis at time $t = 0$. As we can see, we have roughly $10m/s^2$ of

[2]We will see later, that the subject can be relevant for the analysis.

acceleration at time 0. We can also notice that the z-axis has an initial value of roughly $-2m/s^2$ and if we look carefully, we can notice that the x-axis has as well a very small initial value under 0. This is due to the gravity, applied to the phone. Here is a first thing we can state.

Assumption 2.1. The data recorded by the accelerometer at time $t = 0$ can be considered as the gravity vector.

So far so good, let us now take a look at the whole plots. We can notice that all the values go first down, than up again, and so on. Probably because the movement goes down, during a push-up. The most interesting thing is how the three axes make the same movement. The maxima and minima are at almost the same point of time. The only difference is the amplitude of each axis. This means that we clearly cannot take just one axis to count the repetitions, because we could pick the wrong one. If in this example we look the plot of the acceleration in z, we cannot differentiate the peaks from the noises. But, if we take a look at all the plots together, we can count the peaks. There are 10 major peaks, representing the 10 repetitions of the push-ups. The plots do not really look like the first intuition that we made in Chapter 1.2.1, but it is pretty close. And we could count the repetitions easily.

Let us now take a look at the "bad" data, depicted in Figure 2.3. First, let us take a look to the first plot (first kind of bad push-up: goal unreached). We can see that there is still an initial value, representing the gravity. Like for the proper push-up case, we cannot look at one specific axis separately from the other two. If, for instance, we look at the y-axis, we clearly cannot tell anything, but that there was a movement at the beginning and a go-back at the end. But why are the values in this axis so small? Well, if we look at the x-axis, we can see that the maximum value is a bit less than $10m/s^2$ (maybe 7 or 8)[3]. Which is smaller than the maximum values of the same axis for the proper push-up case. Excepting for the z-axis, all values are smaller in the bad push-up than in the proper one. We can pretty much say the same thing for the second kind of bad push-ups (start position not reached back). The only difference, is that the values are deeper, like shifted by 2 or $3m/s^2$ to the bottom. That make sense, because the aim of the exercise was to first reach the floor and than raise half the way from the prone position to the plank position.

We can make a first conclusion: we can recognize bad push-ups from proper push-ups. Let us now take a look at the gyroscope data.

[3]If we consider the absolute value of the maxima, this holds. But, as the values are negative, they are actually greater.

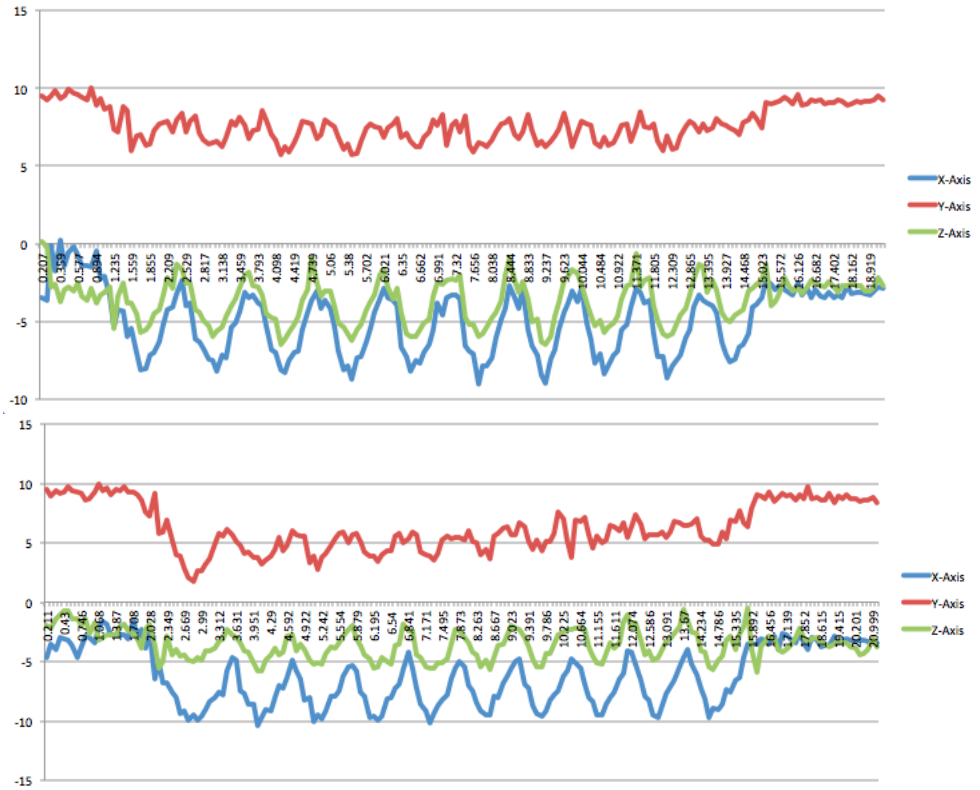

Figure 2.3: The top figure shows the plot of the acceleration (in the same 3-dimensional axes as before) of the first kind of bad push-up (goal not reached). The bottom figure shows the plot of the second kind of bad push-up (start not reached).

2.1.2 Gyroscope

The gyroscope measures the angular acceleration, rather than the tangential acceleration. Which means, the accelerometer measures some data when the device moves and the gyroscope measure some data when it rotates around one of the three axis x, y and z (Figure 2.1). This time, the data are measured in rad/s^2, rather than m/s^2.

Let us first take a look to the data for the proper push-up. It is depicted in Figure 2.4. The first thing to notice, is that the gyroscope is way more sensitive than the accelerometer, even though, in the Android application they were set to the same rate value[4]. This make it much more noisy, as we can see in the plot. Even though it makes sense, notice that there is no initial value, like it was the case for the accelerometer data. Which means, that the gravity does not affect

[4]According to the Android Sensor API Guide (*Monitoring Sensor Events* [4]), each sensor listener has a rate value, which is the smaller rate at which the sensor will listen to.

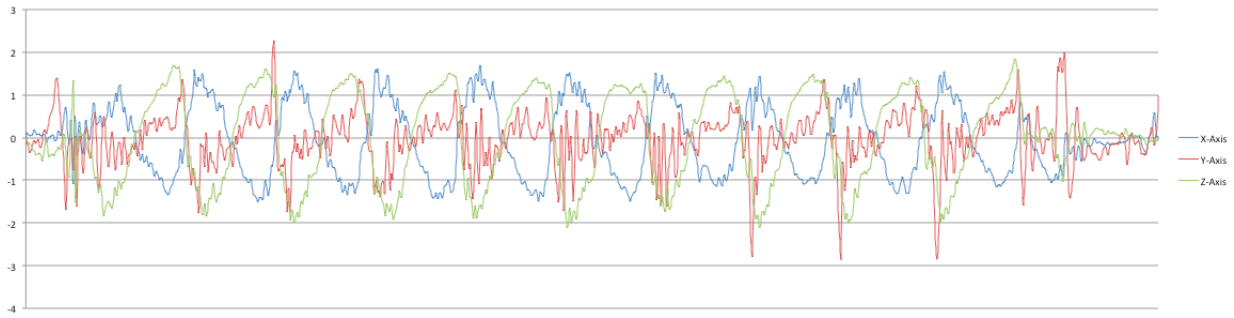

Figure 2.4: The gyroscope data of a "proper" push-up. It represents the angular acceleration applied to the phone around the x-, y- and z-axis against time.

the gyroscope sensor. Ones again, we can not really rely on the data of just one axis. Like the y-axis, which seems to follow some kind of pattern, but it is not really reliable to what happens in the x- and z-axis. If we look at those two, it looks like we have some kind of sinusoidal function, with again 10 peaks (the 10th in the x-axis is not really visible, but we can see some kind of peak at the beginning of the plot).

Let us now look at the data of the bad push-ups, depicted in Figure 2.5. We can say almost the same things as the data of the proper exercise, except that the amplitudes are smaller. For both of them, the y-axis is pretty messy and we cannot do anything with it.

If we think about it, though, the device do not rotate much, during a push-up. Analyzing the data of the gyroscope does not make a lot of sense, comparing to what the accelerometer data can show us. We will describe another experience later in this paper, where we tested the gyroscope data (Section 2.4).

2.1.3 First Conclusion

Let us sum up what we said in this section. Apparently, we can count the number of repetitions, when we do push-ups, proper or bad once. There is also a difference between proper and bad push-ups. The accelerometer seems enough to count the exercises, at least for the push-ups. For now, the gyroscope is put apart. We can, therefore, make those two assumptions:

Assumption 2.2. Using the accelerometer, we can count the number of push-ups during an exercise. As it is pretty similar to the first intuition, we can assume that this is valid for similar exercises

Assumption 2.3. There exists a difference between proper and incomplete exercises, which is the maximum acceleration reached.

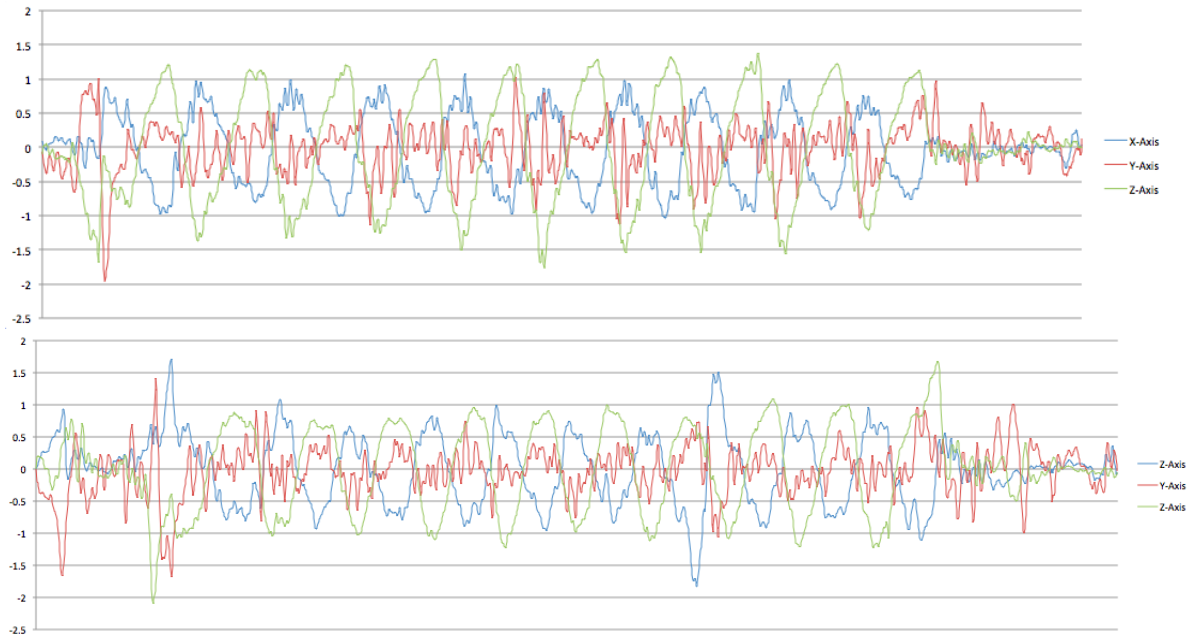

Figure 2.5: The top figure shows the plot of the angular acceleration (in the same 3-dimensional axes as before) of the first kind of bad push-up (goal not reached). The bottom figure shows the plot of the second kind of bad push-up (start not reached).

Now, a question may rise. If there is a difference when I do my push-ups deeper or when I do not, what happens if another person (taller or smaller) do the same exercise?

2.2 How about another Subject?

According to what we saw in the first experiment and the Assumption 2.3, there is a difference, when one make a deeper push-up or when he does not. What about a taller or smaller subject than Subject 1? He would stand higher in the plank position, which means that he would go deeper, if he does a proper push-up. That is why we did the same experiment as before, but with another subject (a friend of mine, 1.83 meters tall). We will call him Subject 2. We will just look at the accelerometer data. As we saw in the previous section, the gyroscope is not very useful, in this kind of exercise.

The data are plotted in Figure 2.6. Apparently, this plot and the one of the push-ups of Subject 1 (Figure 2.2) look the same. We start with an initial acceleration (the gravity, Assumption 2.1) and have 10 peaks visible at almost all axes. Let us take a closer look. The average (absolute) peak value in Figure 2.2 are 10 for the x-axis, 8 for the y-axis and 7 for the z-axis. The same values

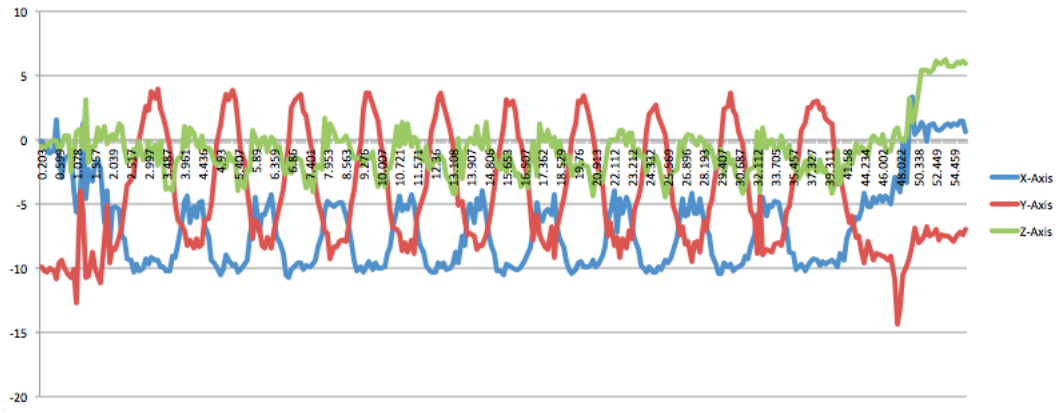

Figure 2.6: The plot shows the acceleration recorded by Subject 2 when he was doing push-ups.

for the data of Subject 2 are respectively 10, 15 and 4. In average, the values of Subject 2 are higher, but it is hard to tell. We need a way to compare the two data in a more efficient way, than looking to plots and approximate some values. We will look deeper in it in the next section.

2.2.1 Second Conclusion

The question was the following: is there a difference in the data between two different people, with different heights. We cannot make any true assumption yet, but intuitively, there may be a difference. We just need a formal way to compare the data. One thing we can tell for sure, Assumption 2.2 is confirmed, here, as we can, once again, count the peaks and get the number of repetitions. The only disturbing thing may the noise. How can we say if a peak has to be counted as a repetition or as noise? This part will be handled in the last section of this chapter.

2.3 The Acceleration Vector

In order to find a way to compare the data of two different experiments, let us first make a more formal definition of what the accelerometer records. We mentioned earlier that the accelerometer measures the acceleration applied to the device in the 3-dimensional coordinates depicted in Figure 2.1. But what is it, that we recorded, exactly? We can see the data recorded by the accelerometer as the evolution of the acceleration against time. And this is what it clearly is, an acceleration vector. This could be intuitively deduced, but we never stated

that clearly. For now on, we will consider the data of the accelerometer as a vector that evolves with the time.

2.3.1 The Magnitude

What can we say abut this acceleration vector? It is hard to imagine how this vector evolves by just looking at the data we collected so far. But let us think about the meaning of this vector. A vector is defined with its direction and its length. The first would just tell us at which direction the velocity is going to swerve, but we do not need that to count a repetition. How about the length? The magnitude of the acceleration can tell us how fast the device moves. We can suppose that the device does not move, at time $t = 0$, which means that the velocity is null, when the sensor starts. What if we could use the magnitude in order to decide how far the user goes?

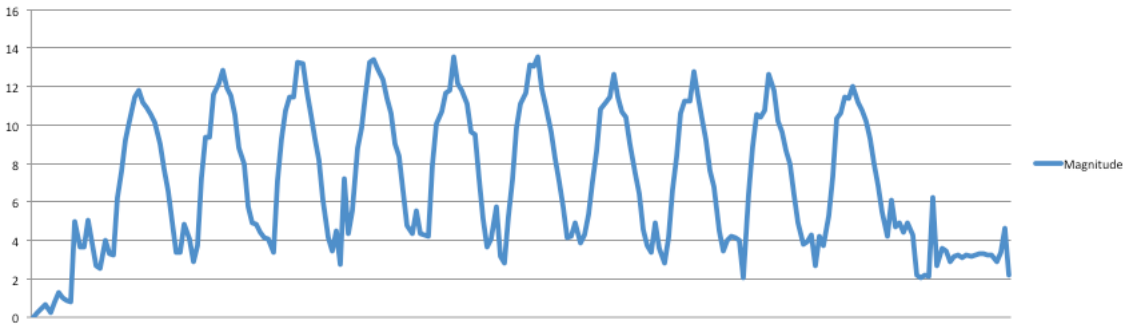

Figure 2.7: The plot shows the evolution of the magnitude of the acceleration vector against time of the first experiment (Figure 2.2). We removed here the gravity vector, defined as the acceleration recorded at time $t = 0$ (Assumption 2.1).

Figure 2.7 shows the plot of the magnitude of the acceleration vector applied to the phone during a push-up by Subject 1. The gravity vector has been subtracted, before computing the magnitude. This means that this is the magnitude of the acceleration vector produced by the push-ups. Let us now take a closer look to this plot. We can really easily recognize and count the number of repetitions. There are 10 peaks that are clearly greater than the others (which we can consider as noise). We can, therefore state a new assumption:

Assumption 2.4. Using the magnitude of the acceleration vector recorded by the accelerometer (subtracted by the gravity vector), we can count the number of repetitions.

The 10 peaks have approximately the same value, around 12 and 14 m/s^2. In the last section, we asked if we can depict a difference between the results of two different subjects. Let us see if we can answer it now.

2.3.2 Comparing Subjects

In last subsection, we have seen the magnitude of the acceleration realized by Subject 1 during push-ups. Let us see how does this magnitude looks like for Subject 2.

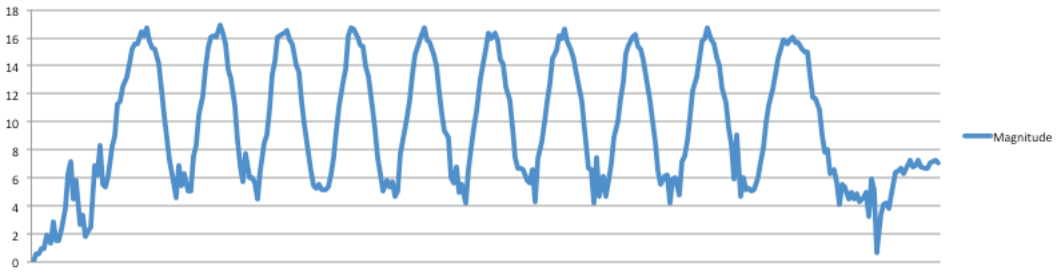

Figure 2.8: The plot shows the evolution of the magnitude of the acceleration vector against time of the first experiment, realized by Subject 2 (Figure 2.6). We removed here again the gravity vector, defined as the acceleration recorded at time $t = 0$ (Assumption 2.1).

Figure 2.8 shows the plot of the magnitude for Subject 2, doing push-ups. We can make the same remarks we did for Figure 2.7. We can count the 10 repetitions without any trouble, confirming Assumption 2.4. But let us now compare the two figures. It is pretty evident, that the peaks realized by Subject 2 are greater. And also more constant ($16m/s^2$). There is a difference of at least $2m/s^2$ between the two plots. The next two assumptions follow directly from our remarks:

Assumption 2.5. We can compare the results of two different subjects, by looking at their magnitude.

Assumption 2.6. The magnitude of the acceleration realized by different subjects during the same exercise may differ with a non negligible value.

By Assumption 2.6 we can deduce that there can be a difference, also for different exercises. In order to make an algorithm to count the repetitions in real time, we have to store in a variable the magnitude value that the user has to achieve, in order to make his repetition to be counted as one. This value has to satisfy the following restrictions:

- it can not be too small, or the algorithm will count too much repetitions;
- it can not be too large, or the algorithm will miss a few repetitions;
- by Assumption, 2.6 it will depend on the subject and the exercise (may be smaller or larger).

We will see in Chapter 3 how the app generates this value and how it deals with it. We have collected enough information, in order to count repetitions. We still have to see how to deal with noise (Section 2.6), though, and we haven't discussed much about the gyroscope.

2.4 Further Analysis on Gyroscope — Analyzing Squats

In the previous sections, we mainly discussed about the accelerometer. We decided to leave the gyroscope apart, and make more experiments later. To see how the gyroscope behaves, we need an exercise, where the device rotates. Assuming a device placed in the leg, above the knee. If the user does some squats (see Figure 2.9), it will make a rotation of 90 degrees.

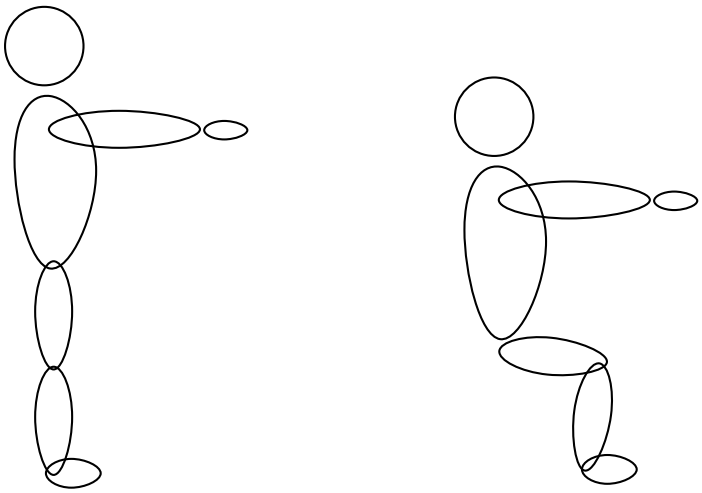

Figure 2.9: The left hand side figure shows the position of the user at start (stand-up position). The right hand side figure shows the position of the user at the end of the exercise (sit position).

The aim of this experiment is to have a more meaningful set of data of the gyroscope. For this exercise, the device should rotate 90 degrees. We want to detect this difference.

2.4.1 Gyroscope data of Squat

Like we mentioned, the device is placed at the bottom of the thigh, just above the knee. The data is recorded by Subject 1.

The data recorded by the gyroscope for proper squats (executed by Subject 1) are depicted in Figure 2.10. Excluding the crash of the application at the

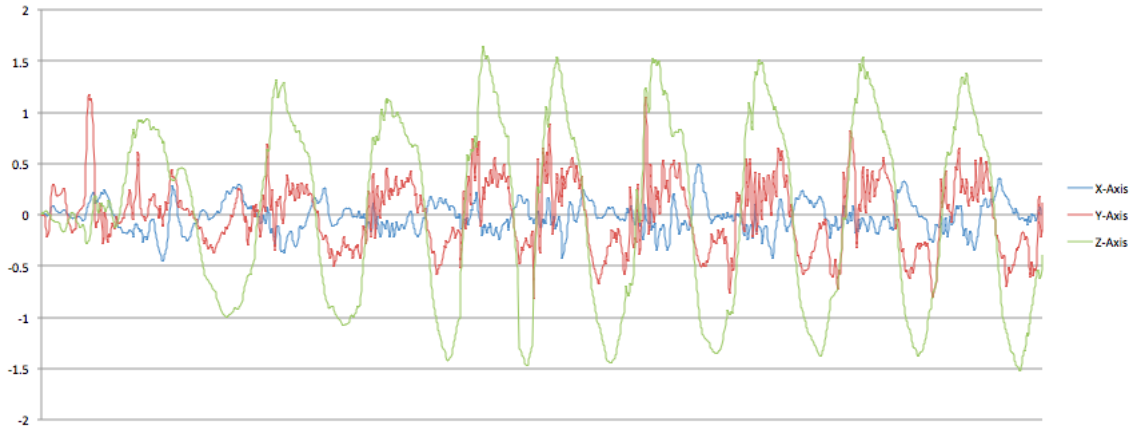

Figure 2.10: The plots describe the evolution of the angular acceleration around the 3 axes (as depicted in Figure 2.1). The aim of the exercise was to do 10 squats, but the application that records the data crashed at the end, this is why we can only see 9 peaks.

end, let us analyze the data. We can first notice that the data on the x-axis is pretty messy and noisy. The y-axis is better, but it is not very reliable either. It pretty much follows the same pattern as the data on the z-axis, but not always and not very well. So, let us look at the z-axis (green plot). We can count the 9 repetitions (the 10th is not visible because of the crash). Which is good, so far. One can count the number of repetitions, even though we have to decide wisely which axis to pick. But let us now compare this data with the incomplete squat.

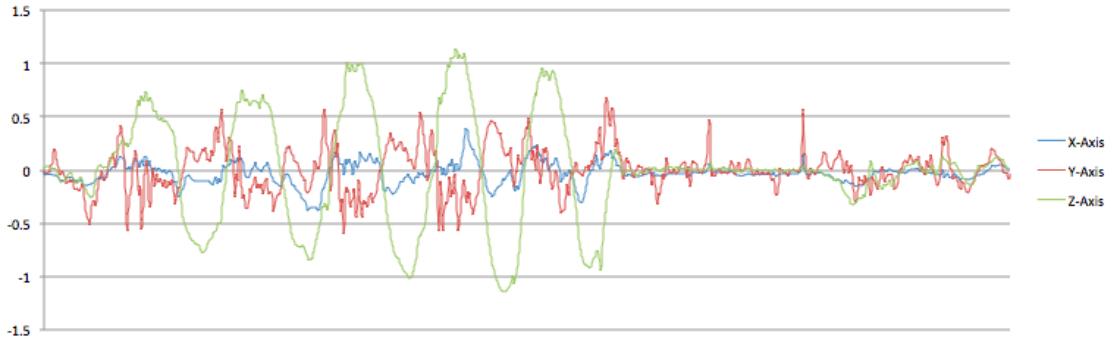

Figure 2.11: The plots describe the evolution of the angular acceleration around the 3 axes (as depicted in Figure 2.1). The aim of the exercise was to do 5 squats (to avoid the app to crash again), which are incomplete. The sit position was not reached.

Figure 2.11 shows the plots of the gyroscope data (angular acceleration) around the 3-axes x, y and z (see Figure 2.1). We can pretty much say the same

thing as last figure for the x and y axes. The z-axis is similar to the one of the complete squats, with one difference. The value reached by the incomplete squats is globally smaller. However, there is an issue, here. The smallest value of the good squats is not greater than the greatest value of an incomplete squat. So, if we record some data in real time, we cannot differentiate the two kind of squats.

2.4.2 Accelerometer data of Squat

The application used to record the data, remember, records the data of both sensors, accelerometer and gyroscope. As the gyroscope didn't get the result we expected (no real difference between incomplete and complete squats), let us look at the accelerometer data again.

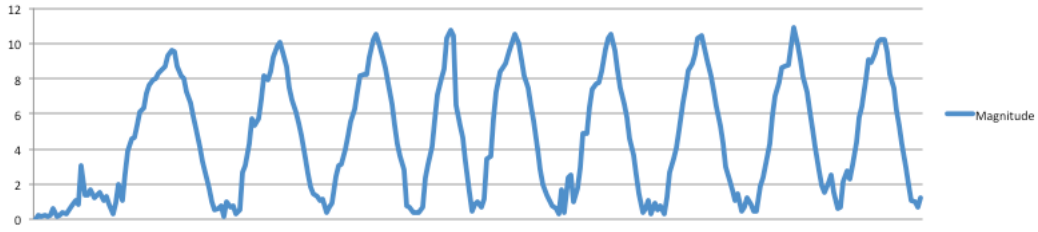

Figure 2.12: Magnitude of the acceleration vector applied to the device during complete squats. Here again, we should see 10 peaks, but the application crashed at the end.

Because of what is stated in Assumption 2.4, let us look directly to the magnitude of the acceleration (again, the gravity has been omitted from the computation). In Figure 2.12, we can see the magnitude for complete squats. Like for push-ups, we can count the repetition without any trouble. Moreover, it seems like the noise is pretty low, here. Assumption 2.4 is confirmed for squats and Assumption 2.6 is also confirmed for push-ups and squats. Remember, for Subject 1, that the value of the peaks for push-ups was between 12 and $14m/s^2$. For squats (and the same subject, of course) this value is $10m/s^2$.

In Figure 2.13 we can see the magnitude of the acceleration for incomplete squats. Just like the push-ups, the difference between good and bad squats is clearly visible in the magnitude of the acceleration applied to the device. For squats, the smaller difference of the magnitude is near $2m/s^2$. We can therefore imagine an algorithm that can count the repetitions of any rest-to-goal movement [5]. The only thing that differ from an exercise to another is this value to reach. But this algorithm should be able to count the repetitions, without knowing what kind of exercise the user is doing. It should only have to assume

[5] See Section 1.2 to see again how we defined such a movement.

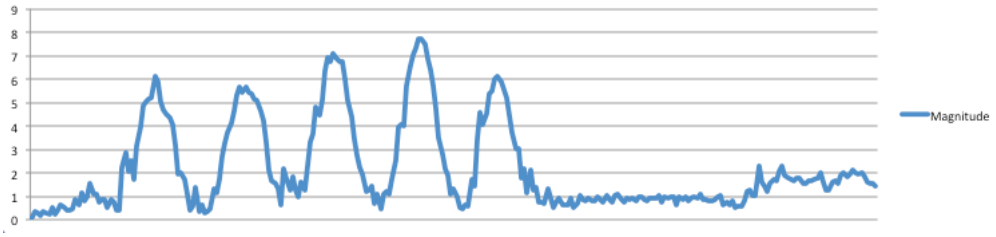

Figure 2.13: Magnitude of the acceleration vector applied to the device during 5 incomplete squats.

that the exercise is a rest-to-goal movement.

We wanted to pick an exercise where the performance of the gyroscope could be emphasized, but it seems that the magnitude of the acceleration vector, recorded by the accelerometer, is enough to make the counting. But is there a way to analyze and differentiate two different exercises? It could be very convenient, if one could just press a button and do any kind of exercise, the application would recognize the movement and count the repetitions automatically (maybe using the goal value of the exercise).

2.5 Principle Components Analysis

We already saw that we can count any kind of rest-to-goal movement using the magnitude of the acceleration vector recorded by the accelerometer. Which is enough, for the main goal of the project, but we want to extend the possibilities. So, the question now is the following: is it possible, given some data recorded by both accelerometer and gyroscope, to recognize a specific movement? Intuitively, we would say that it is. But we have to prove it, or show it, somehow. This is why we focus on the Principle Components Analysis in this section. The main idea of this concept is to generate a list of vectors (called principle components), in order to get the main behavior of a set of data. What does this mean for our problem? Assuming we have a person doing push-ups. We want to retrieve the vector that describes best this movement, which should be a vector going down.

If such a vector exists, this means that it should exist for any kind of exercises and it could be, for each exercise, different. Plus, using the same analysis on the gyroscope, we could probably differentiate every possible rest-to-goal movement. If a device moves and rotates in the same way for two different exercises, this probably means that the exercises are actually the same, or very similar.

As this concept of vector that describes the main behavior of a movement is hard to imagine, we will separate the analysis in half. We will first look at 2-dimensional data, before analyzing the 3-dimensional data that we have already collected. After that, we will compare the principle components with other data,

to see how they differ.

2.5.1 Principle Components of 2-dimensional Data

We said it many times so far, the data retrieved by the sensors are 3-dimensional. In order to have a set of 2-dimensional values, we have to omit an axis and only look at two of them. Which means, we will have three sets of data. The projections on the x-y-axes, on the x-z-axes and on the y-z-axes. Ignoring the time, we have, for each pair of axes, a set of points in a 2-dimensional plan. We can see those data as the correlation between two axes.

We now have to generate the principle components for each pair of axes. To do so, we used the algorithm on MatLab described on a paper of Jonathon Shlens, *A Tutorial on Principle Components Analysis* [5]. The idea is to first create a matrix, whose rows are the samples of each axis. We will get a $(n \times m)$-matrix, where n is the number of axis and m the number of samples. We then subtract the mean value of each row of the matrix (generating data centered on $(0, 0)$) and normalize the matrix. And we finally compute the singular value decomposition (SVD)[6] of that centered and normalized matrix, generating three matrices $\mathbf{S}$, $\mathbf{U}$ and $\mathbf{V}$, such that our new data matrix is equal to $\mathbf{U}$ times $\mathbf{S}$ times $\mathbf{V}$. The rows of the unitary $\mathbf{V}$ are the principle components of our set of data. For our 2-dimensional case, we will get two principle components, one that should describe the data and one that is orthogonal to the first.

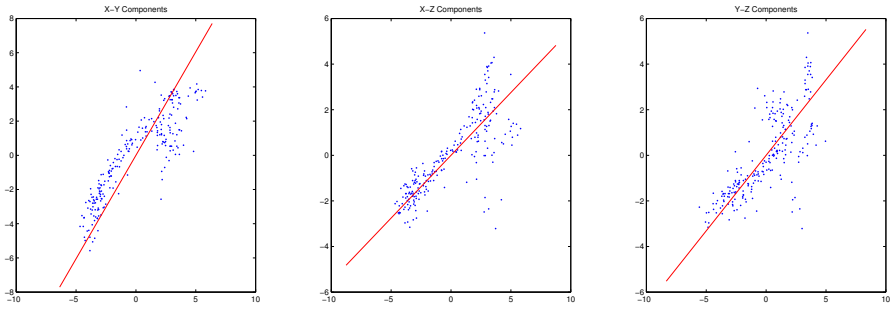

Figure 2.14: The plots show the correlation between the acceleration on two axes and their principle components. The blue dots are the data and the red line is the principle component (rearranged, so that it is visible). The order of the figures are: x-y-axes, x-z-axis and y-z-axis.

We applied this algorithm using MatLab on the push-up acceleration data of Subject 1. The result is depicted in Figure 2.14. As we can see, there is actually

[6] Given a matrix $\mathbf{A}$, we call SVD the decomposition generating the matrices $\mathbf{S}$ (of the same size as $\mathbf{A}$), the unitary $\mathbf{U}$ and $\mathbf{V}$ (meaning that, multiplied each of them with their complex conjugates, we get the identity matrix), such that $\mathbf{A} = \mathbf{USV}$

a vector that describes the data for each pair of axes. In theory, we should have as well a vector for the 3-dimensional case.

2.5.2 Principle Components of 3-dimensional Data

In order to get the principle components of the 3-dimensional data recorded by the accelerometer, we do the same thing as the 2-dimensional case. The idea is the same, we want a vector that describes the data. This time, as we have data on three dimensions, and therefore 3 rows on the data matrix, getting finally 3 principle components.

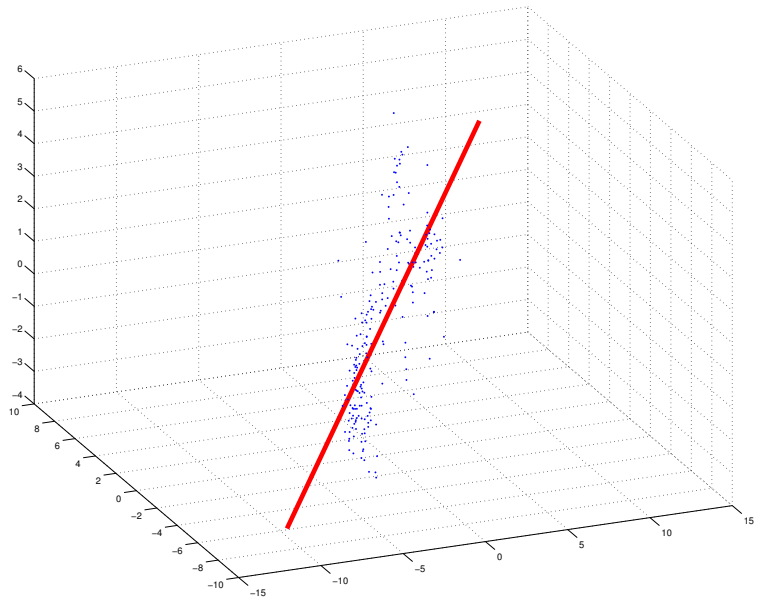

Figure 2.15: The figure shows a red vector (linear combination of two principle components) going through the push-ups data of the accelerometer. To have the main behavior of the acceleration, we omitted the time and plotted the acceleration in the x-y-z-axes.

This time, none of the three components could describe the data independently, but we had to generate a new vector, which is a linear combination of two principle components. Here, we could make some tests, to see what combination to choose. But how could we do it using an algorithm? It does not look easy. Before talking algorithms and automatization, let us first look at the component for the gyroscope data.

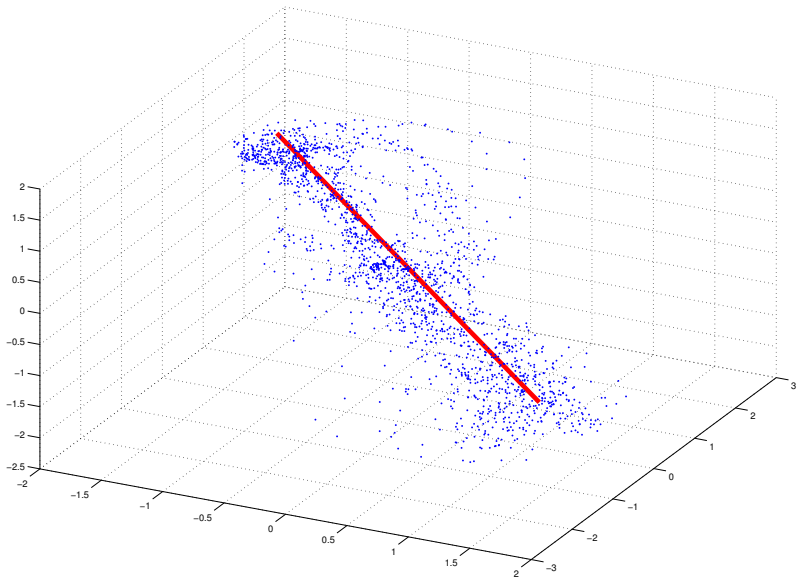

Figure 2.16: The figure shows a red vector going through the push-ups data of the gyroscope . To have the main behavior of the angular acceleration, we omitted the time and plotted the angular acceleration in the x-y-z-axes.

For the gyroscope data (Figure 2.16), one of the three components generated was enough to describe the rotation. We did exactly the same thing in order to get the principle components. The vector seems to fit pretty well in the data, but it is pretty noisy, so we cannot tell for sure.

Now that we have the vectors that describes the accelerometer and the gyroscope data, let us look how they fit with another sample of data.

2.5.3 Comparisons

Squats

We started comparing the principle components of the push-ups (for the accelerometer and the gyroscope) with the data recorded during squats. So, we plotted the data of the squats together with the vectors that describe the accelerometer and gyroscope behavior of a push-up (red vectors on figures 2.15 and 2.16). The result is depicted in Figure 2.17.

As we can see, the component of the accelerometer data for push-ups is pretty much orthogonal to the data of the squats. We cannot say the same thing for

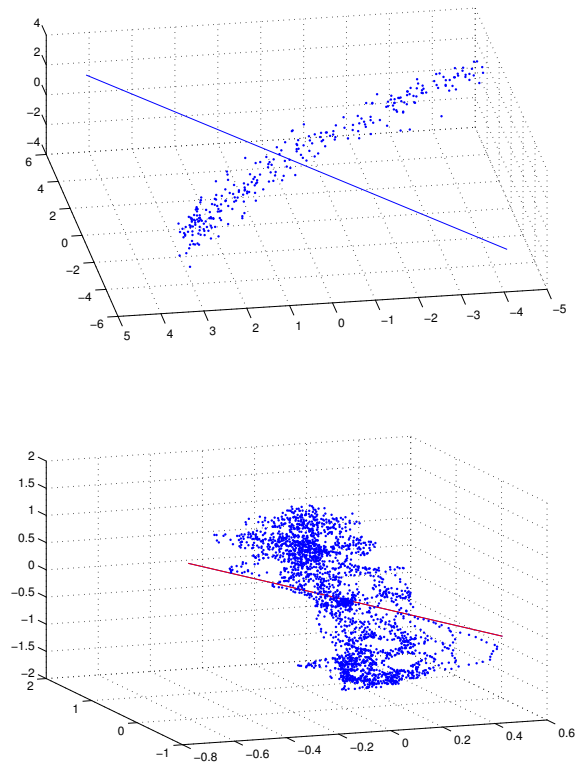

Figure 2.17: The top figure shows the plot of the accelerometer data of squats with the principle component of the push-ups. The bottom figure shows the same thing, but for the gyroscope data, this time.

the gyroscope data, where the principle component of the push-ups just cut the data. We could say that the principle component could be used to "guess" what kind of movement the user does. But, to be sure, two same exercises should look alike, in their principle components.

Push-ups of Subject 2

We did here the same thing as we did for the squats. We got the data of the push-ups of Subject 2 (for both accelerometer and gyroscope) and see if the principle components of the push-ups of Subject 1 describe the data of Subject 2. See Figure 2.18 for the plots.

Unfortunately, the results are not as we expected. Even though for the

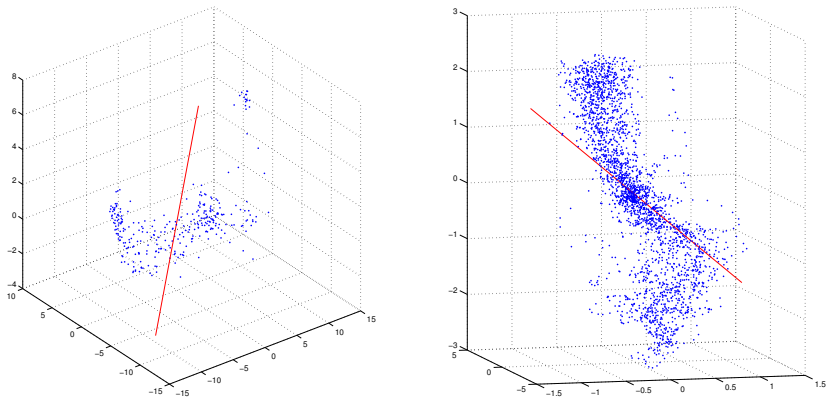

Figure 2.18: The left figure shows the plot of the accelerometer data of Subject 2's push-ups with the principle component for the same exercise did by Subject 1. The right figure shows the same thing, but for the gyroscope data.

gyroscope data, the component describes almost the data, we clearly cannot say the same thing for the accelerometer data, where the component of Subject 1's push-ups is orthogonal to the data. We didn't get enough time to deepen the subject and see what we did wrong or what other solutions are possible. It would have been nice to have an application that could recognize an exercise, before counting the repetitions.

2.6 Handling with Noise

In the previous sections, we talked about a goal value (considered as the magnitude of the acceleration), that the user has to reach, in order to complete an exercise. To do so, we need some sort of algorithm, that would get the peaks and generates the goal value using them. However, in order to count the peaks, we need to differentiate a repetition from the noise. Not all increasings (or decreasings) that we can record have to be taken in count. To do so, we need an error margin. We used two techniques, in order to handle the noise.

2.6.1 Noise Calibration

Asking the user to do some repetitions of an exercise, we can actually have an idea about this noise value. To do so, we just can ask the user to make one repetition, get the maximum value recorded and generate the noise using this value. Some tests will show that a good coefficient is $0.15 \times max$ where max

is the maximal value recorded during one repetition. This can make sense, to have a linear function that computes the noise. For small exercises, we do not want the noise to ignore the real repetitions. And for large goal values, we want a larger error margin.
This computation may be enough, but if the noise is too much present, we could need a way to reduce it, before computing this value.

2.6.2 Laplacian Smoothing

The Laplacian Smoothing is a solution for last request. The idea is to take a certain number of neighbors[7] and compute the next point, averaging it with its neighbors. This method is used for smoothing polyhedrons, but we can use it for our problem. We simply store the first n values recorded by the sensor[8], and generate the $(n+1)$th point averaging it with the n previous ones. At each iteration, we change the set of n neighbors, removing the most recently added and adding the new one. But it is important to store the actual value of the last recorded point, not its smoothed value.

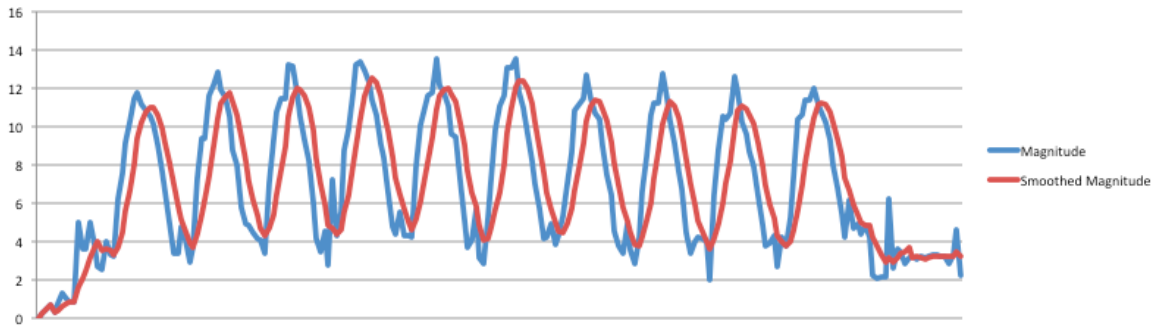

Figure 2.19: The figure shows the plots of the magnitude of the acceleration of the push-ups of Subject 1 (see Figure 2.7) and its smoothed representation, using Laplacian Smoothing.

Figure 2.19 shows the result of the Laplacian Smoothing of the acceleration of the push-ups of Subject 1 using 5 neighbor points. As we can see, the noise is pretty much reduced, but the peaks are also flattened. We have to chose the number of neighbors wisely, in order to get the best result.

[7] This number has not to be chosen slightly. Too much neighbors will smooth the data too much, reducing the value of the peaks, and just a few neighbors will not destroy the noise enough.

[8] Remember that the accelerometer is enough to count the repetitions.

Chapter 3

Implementation

Up to now, we only talked about the analysis. We concluded that:

- The accelerometer is sufficient to count repetitions of a rest-to-goal movement (Section 1.2)
- The magnitude of the acceleration vector recorded is a good way to compare different exercises
- Given the goal value that the user has to reach, we can count the proper repetitions and miss the incomplete one

From the above, we have to implement two things: some sort of calibrator, in order to measure the goal value, and the counter itself. In the next section, we will describe the two main algorithms and their state machines. After that, we will make a list of tests, describe them, show the results and explain what modifications we made to the application to correct the errors. In the last section we will briefly explain how the application looks like and how to use it.

The application is developed using an Android plugin for Eclipse[1]. The programing language is Java[2], to which are included several classes from the Android API[3].

3.1 Algorithms

There are two main algorithms, that implement what we saw in the analysis: the counter and the calibrator. We will start explaining the counter, assuming that the goal value and the noise of the given exercise are stored in some variables. We will next see how we managed to obtain those two values. We will describe those two algorithms by looking at their state machines, without going through

[1] ADT Plugin: Installing the Eclipse Plugin.
[2] Java™ Plateform, Standard 7th Edition: Java API.
[3] Android API: list of classes.

the code, neither show a part of it. We will just explain in a high level how the counter and the calibrator work.

3.1.1 The Counter

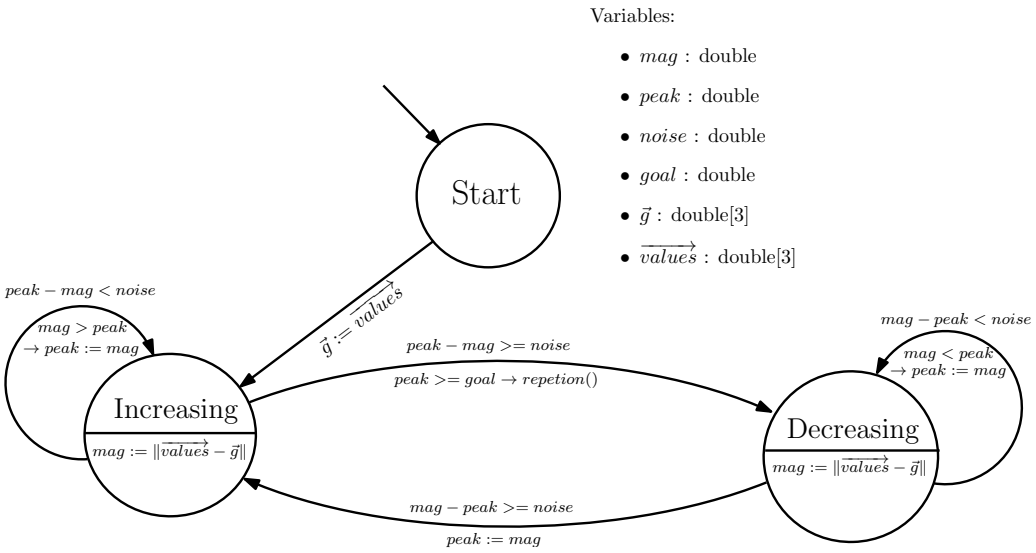

Figure 3.1: The state machine for the counter. The states are updated every time the sensor detects a movement. The variable *mag* will store the current magnitude of the acceleration vector (stored in the array *values*, depicted here as a vector); *peak* will have the maximum (or minimum) value encountered so far in the corresponding state (Increasing or Decreasing); *noise* is a constant, containing the noise value of the exercise (computed during the calibration); *goal* is the constant for the goal value that the user has to reach; and $\vec{g}$ (implemented as an array of doubles of size three in Java) will contain the gravity vector. Each arrow represents a change of state (or not), where the condition is given above it, and a possible action is written below it.

In Figure 3.1 we can see the state machine of the repetition counter for a rest-to-goal movement. We want to point first out that the goal value and the noise of the given exercise are stored in the variables *goal* and *noise*. This algorithm is incorporated into a class that implements the SensorEventListener interface[4], that we simply called `Counter`. This interface have a method called `void onSensorChanged(SensorEvent event)`, into which we implement the state machine. This method is called every time the sensor detects a movement and store the values of the acceleration vector into the attribute `values` (an array of double) of the parameter `event` of type `SensorEvent`. We depicted it in Figure 3.1 as the vector $\overrightarrow{values}$. Like we said before, the goal value and the noise are known, which means that they are constant attributes of the class `Counter`. As well as

[4] An example is presented by the Android Sensor API Guide [4]

the array of double storing the gravity vector, depicted in the state machine as $\vec{g}$. The variable *peak* in the figure is also an attribute of the class. As it may change at each call of the method `onSensorChanged()`, it cannot be local. It is not the case for the variable *mag*, which contains the current magnitude of the acceleration, without the gravity component.

Now that we have set all those clarifications, we can start explaining the state machine. Obviously, we start at state Start. It represents the first time the sensor detects something. We can say, it is the state at time $t = 0$ in the plots of Chapter 2. Like we mentioned it several times and stated in Assumption 2.1, in this state, the current acceleration recorded is the gravity. So, we store a copy of $\overrightarrow{values}$[5] into $\vec{g}$. As the user is just starting, the magnitude will increase. So, we go into the state Increasing, waiting for the next time the sensor will detect a movement. In this state, we initialize the variable *mag* with the length of the vector $\overrightarrow{values} - \vec{g}$. After that, we check if $peak - mag$ is smaller than *noise*. If it is the case, we consider that we are still increasing, so we stay in the same state and we update the *peak* value to be the current magnitude value of the vector, if $mag > peak$ (which means that the current magnitude value has to be the new peak value, because greater than it). As *peak* store the greater magnitude value, if *mag* is greater, *peak-mag* will be negative and therefore smaller than *noise*. If it is not the case, but the difference is still smaller than the noise value, we will not consider that we are decreasing yet, because it may be just some noise. However, if $peak - mag$ is greater or equal than *noise*, we go to the state Decreasing. Moreover, if *peak* is greater or equal than the goal value[6], we have to count this as a repetition. We therefore call a method (called `repetition`), which will handle it (see Section 3.3, later in the paper). In the state Decreasing we do pretty much the same thing, with the difference that the condition for staying in the state is $mag - peak < noise$ and the condition to update *peak* is $mag < peak$. Here, the current magnitude has to be greater than the previous, to have a decrease. And again, we add this error margin of *noise*. As long as we are in this state, we update the variable *mag*, the same way we do for the Increasing state. If we consider that we are increasing ($mag - peak >= noise$) we update the peak value and go to state Increasing.

The state machine does not seem to have a stopping state, because it is handled in the method `repetition`, that we will explain later. This is how the application count a repetition, basically. Let us now look at how to retrieve the variables *goal* and *noise* with the calibrator.

[5] As in Java we implemented the 3-dimensional vectors as arrays of size 3, it is important to make a copy of the elements of $\overrightarrow{values}$ into $\vec{g}$, instead of just making *g = values;*, because we would store the references of the arrays, rather than their elements, which could cause errors.

[6] We added an error margin, first of 0.1, then to $0.5 * noise$ (according to the results of the tests, that we will see later), just in case. So, the real condition is if $peak >= goal - 0.5 * noise$

3.1.2 Calibration

We splitted the calibration in half: another class counter, slightly different from the one we saw just before (called `CalibrationCounter`) that still implements the interface `SensorEventListener`, and a second class (called `Calibrator`) to handle this new counter.

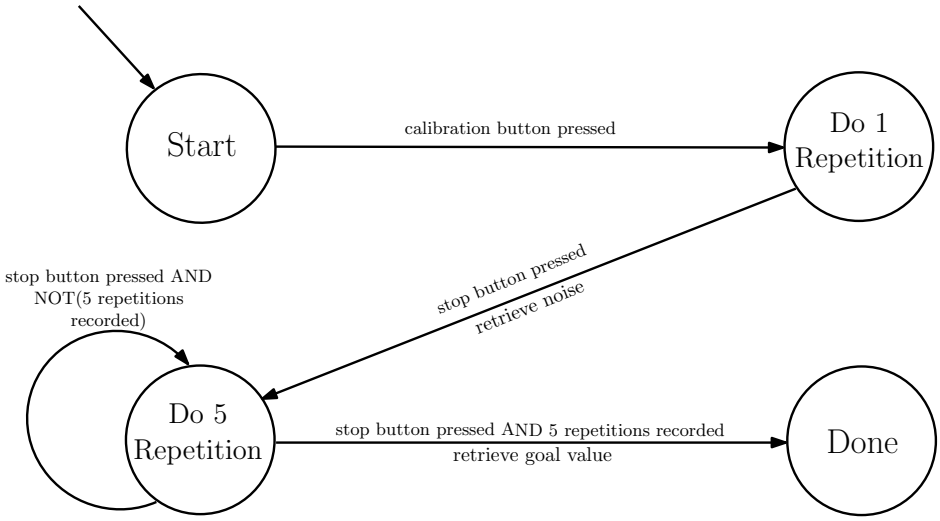

Figure 3.2: This is the state machine of the calibrator itself. At the first state (Start) we wait the user to press the calibration button, and start it. We then ask the user to do one repetition. At this state, the calibration counter is recording (the state machine is depicted in Figure 3.3). When the user press the stop button (we assume that he did at least one repetition) and we generate the noise value before going to the next state. There, the user has to complete 5 repetitions (the counter is ones again running). When he presses the stop button again, if the application counted 5 repetitions, we are done and we generate the goal value. But if he does not, we stay in the same state.

In Figure 3.2 is depicted the state machine of the calibrator itself. The calibration truly start when the user press a button (labeled "Calibrate"). At first, we ask the user to do one repetition, then to press a stop button (obviously labeled "Stop"). The `CalibrationCounter` is running, waiting the user to stop it. Its state machine is depicted in Figure 3.3. We will talk about it just later, but notice, still, that the noise value is null, for now. When we go to the next state, we first generates the noise value (we will see later how, exactly) and store it in the calibration counter. In the next state, the user has to do 5 repetitions. This time, the noise is set, so the number of repetitions recorded has to be equal to the number of repetitions that the user do. We do not talk about proper or bad repetitions, here, which means that the repetitions are supposed to be done correctly. We only count the number of repetitions. If this number is not exactly equal to 5 when the user presses the stop button, we stay in the same state. If

it is the case, we generate the goal value and we are done.

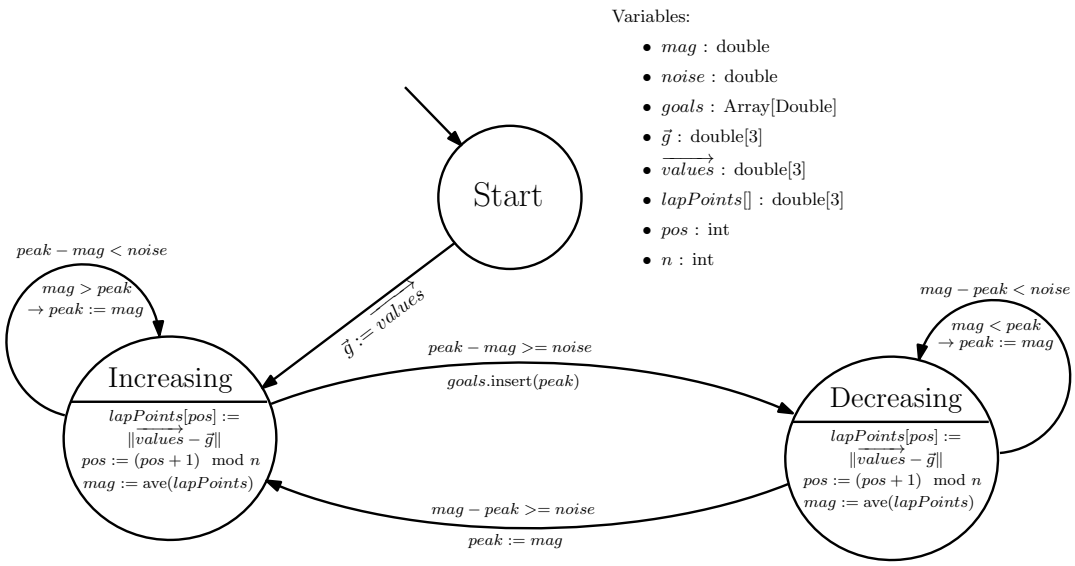

Figure 3.3: The state machine looks more complicated than the one of the counter (Figure 3.1), but just a few things are different. Instead of storing the length of the current acceleration in the variable *mag*, we store it in the array containing the n previous points, by updating the array lapPoints (containing the neighbors for the Laplacian Smoothing) at the current position (stored in the variable *pos*, whose value can change from 0 to $n-1$ at each step). In the variable *mag*, we store the average of the last n points stored so far, to have a smoothed value at each iteration in the machine. If there are not n elements in *lapPoints* yet, we just store the magnitude of the current acceleration into it. The conditions to go from a state to another are the same as those of the repetitions counter's state machine. When we go from Increasing to Decreasing, we add the current peak value in the list of goals, from which we will generate the goal value.

In Figure 3.3 is depicted the state machine of the calibration counter. It is very similar from the state machine of the counter (Figure 3.1), but there is a difference that is not trivial. In the analysis chapter, more specifically in the section where we discussed about noise and how to handle it (Secion 2.6), we saw two methods. The first is to implement a calibrator, which is exactly what we are explaining here. The second one was to smooth the data using the Laplacian Smoothing, before generating any goal or noise value. This is exactly what we implemented in this state machine. To store the n neighbors, we use an array of size n. To access the elements of this array, we use a variable *pos*, which has initially the value 0, and is incremented by 1, each time the sensor detects a movement (or each time the method `onSensorChanged` is called by the

system), and we use the modulo operator, so that, when we have more than n elements in the array, we restore the position to be zero, and start again. In this array we store the actual length of the current acceleration vector ($\overrightarrow{values} - \vec{g}$), not its smoothed value. If there are at least n values in the array, we set the current magnitude to be the average of its n neighbors (as described by the Laplacian Smoothing). If it is not the case, we just store the length of the current acceleration. The conditions to go from the Increasing and Decreasing state (and the other way around) is the same as in the counter's state machine, except for one detail. Instead of calling a method that will handle the repetition counting, we store the current peak value into a list of goal (as a Java's ArrayList of doubles). We insert the values in such a way that the greater value is at index 0.

Let us look back at the calibrator's state machine. When the user has finished the first repetition, we say that we generate the noise value. To do so, we take the first value stored in the array *goals* (which has to be the greater one, as we said before) and set the noise to be 0.15 times this value. We store this in the variable *noise* of the calibration counter, so that it can use it to count the repetitions. And after the 5 repetitions are done, we check if the size of the list of goals is equal to 5 (which will mean that we stored 5 values in it, i.e five repetitions have been done). If it is the case, we generate the goal value as the mean value of the 5 goals.

Notice that the peak value is set to be *mag*, each time we increase. Which means that we store a smoothed value in the list of goals. This values are probably smaller than their no-smoothed versions, so storing the smallest value for the counting is not a good strategy. Using a mean value, we increase the chances to have a more accurate value, without having it too large (like we stated at the end of Section 2.3).

3.2 Evaluation

Before seeing how the application looks like, we will evaluate it. We made several tests, some get good results, some didn't. From some of those results, we made some modifications to the application. We will explain all of this. Each evaluation is describe by:

- A set of exercises
- A subject that executed them
- The calibration
- A sequence of repetitions, including proper ones, bad ones (1/2 of the repetition) and slightly bad ones (3/4 of the repetition)

Those evaluations have been made in the final version of the application. But we tested the counter and the calibrator separately, before. At this point, the noise value of an exercise was computed as 0.25[7] times the (smoothed) goal value. The condition that has to be fulfilled in order to count a repetition was $peak >= goal - 0.1$[8]. We will see later, why those two things have been modified. For the calibration, the number of neighbors for the Laplacian Smoothing was 4 (which we never changed).

3.2.1 Test 1: First Time at the Gym

The user of the application was Subject 1 (1.73 meters tall). The exercises that he executed were:

1. 10 repetitions on a strength machine for pectorals.
2. 10 abs

The first step was to calibrate the device and then execute, for each exercise, this sequence of repetition : 2 proper, 1 bad, 2 proper, 1 bad, 1 slightly bad, 6 proper. Here are the results and the conclusion that we made:

1. For the pectoral machine, the calibration failed. The application counted less than 5 repetitions (3 or 4). We couldn't test the counter, though, so we stopped at this point and realized that, if the counter for the 5 repetitions failed it means that the noise value was too large.
2. For the abs, the calibration went well. The results of the repetitions sequence is the following:
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 2 proper → 0 counted
   - 1 bad → 0 counted
   - 1 slightly bad → 0 counted
   - 6 proper → 2 counted

   We got 0/3 false positives, but 4/8 false negatives.

It is pretty obvious that the first test was a fail, but this is why we changed the algorithms, in order to correct those errors. First, we set the noise value to be 0.15 times the goal value. If we look at Figure 2.19, more specifically the

[7] This coefficient was set according to the data, but it is pretty arbitrary.
[8] Arbitrary error value.

smoothed plot (the red one), we can approximate the noise to be 0.75. And the average goal value is roughly 11. Which means, that the noise value would be set to be $0.15 \times 11 = 1.65$, which a bit more than twice the actual noise deduced from the figure. If the goal value is very low, we added the following condition: if $noise < 1$ after the computation, we set it to 1. We will see later that this gave us a better result for the calibration. We also changed the condition to count a repetitions to be $peak >= goal - noise * 0.5$, instead of $peak >= goal - 0.1$.

3.2.2 Test 2: Squats with Subject 3

The user of the application is Subject 3 (girl, 1.70 meters tall). The application has been changed, using the modifications described in the first evaluation. Here is the list of exercises that she did:

1. 10 squats, with the calibration of Subject 1 (as the two subjects have the same height)
2. First calibrate, then do 10 squats with her own calibration.

The sequence of repetitions is the same for both exercises: 2 proper, 1 bad, 2 proper, 1 bad, 2 slightly bad, 5 proper.

1. With the calibration of Subject 1, we got:
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 2 slightly bad → 2 counted
   - 5 proper → 5 counted
2. The calibration of Subject 3 went well. This are her results with her own calibrated squats:
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 2 slightly bad → 0 counted
   - 5 proper → 5 counted

Just for the first exercise, we got 2/4 false positives, but they were the slightly bad ones (half way between a proper and a bad squat), plus the calibration was not hers. The second exercise went well. The rate of false negatives for both is 0%, which is great. We didn't changed any thing to the application, this time.

3.2.3 Test 3: Second Time at the Gym

The user was Subject 1 again, to test how the application works at the gym. The list of exercises is the following:

1. 10 pull-ups
2. 10 dips
3. 10 repetitions on a machine for pectoral (the same as in the first test)
4. 10 repetitions on a machine for legs

For each exercise, a calibration had to be done. Except for the pull-ups, where the first calibration failed (but the second went well), it worked for each exercise. Here are the results:

1. Pull-ups:
   - 10 proper → 10 counted
2. Dips:
   - 2 proper → 2 counted
   - 2 bad → 0 counted
   - 1 slightly bad → 1 counted
   - 2 proper → 2 counted
   - 1 bad → 0 counted
   - 1 slightly bad → 1 counted
   - 6 proper → 6 counted

   The only problem here, was that slightly bad repetitions are counted as proper ones. Once again, by "slightly bad", we mean that the repetitions was not complete, but it was pretty close to it.
3. Pectoral Machine
   - 2 proper → 3 counted (one repetition was counted twice)
   - 1 bad → 0 counted
   - 4 proper → 7 counted

   The results may look bad, if some repetitions are counted twice. But notice that, first, the goal value of the exercise was set pretty low (3.861), meaning that the noise value was as well (set to 1, because too low, otherwise). Moreover, on this kind of machine, I (Subject 1) usually stop, after I reach the goal position. This may explain why it is counted twice, sometimes.

4. Leg Machine:
    - 2 proper → 2 counted
    - 2 bad → 0 counted
    - 1 slightly bad → 1 counted
    - 2 proper → 2 counted
    - 1 bad → 0 counted
    - 1 slightly bad → 1 counted
    - 6 proper → 6 counted

We can clearly say that the app has, once again, a rate of false negatives equal to 0%. Unfortunately, 100% of "slightly bad" repetitions are counted as good repetitions. The problem is, if we change something on the algorithm, it may not work, anymore. So, we reduced the error margin of the counter to be 0.5 times the noise value. Moreover, we added a hard mode selector, where the error margin is set to be 0.2, if this mode is selected.

3.2.4 Test 4: Hard Mode

Once again, the user is Subject 1. The aim of this test is to see if the hard mode reduces the number of false positives. We tested on abs and push-ups. Here are the results:

1. Abs:
    - 2 proper → 2 counted
    - 2 bad → 0 counted
    - 1 slightly bad → 0 counted
    - 2 proper → 2 counted
    - 1 bad → 0 counted
    - 1 slightly bad → 0 counted
    - 6 proper → 6 counted
2. Push-ups:
    - 2 proper → 2 counted
    - 2 bad → 0 counted
    - 1 slightly bad → 1 counted
    - 2 proper → 2 counted
    - 1 bad → 0 counted

- 1 slightly bad → 1 counted
- 6 proper → 6 counted

For the abs we have a perfect result, with neither false positives nor false negatives. For the push-ups, we have still the same result. Maybe, the error margin has to depend on the kind of exercise. Unfortunately, we did not have enough time to check this assumption or to test it.

3.2.5 Conclusion

We only get false negatives in the first test, but given the results of the following ones, we can say that this error is corrected. At the end, we have a false negative rate of 0%. The false positives are harder to define, as we have have two types of bad repetitions. The real "bad" repetitions are never counted. However, if we do a repetition that is close to be consider as "proper", it is always counted. The hard mode seems to reduce this rate, but it still depends on the exercise.

The fact that we do not have any false positives, could come from this imprecision of the algorithm, that counts more repetitions that it should. But we prefer this result from the opposite, where good repetitions could not been counted. A user would probably prefer that the application counts the repetitions that are not perfect, rather than a program that would not count the good ones.

3.3 The Application

This is a very small section, where we will see how to use the application and how it looks like. Let us first take a look to the home activity (Figure 3.4). We have three buttons. Each of them open a new activity. The first one, open a page where we can pick and add exercises, for the workout. The second button leads us in the calibration page, where we can add or update the calibration of an exercise. The last one open an activity, where all the results of the workouts and their exercises are listed. The application is called SimpleExerciseWorkout, because the only exercises that the workout can have, in order to count the repetitions, have to be a rest-to-goal movement, which can be defined as "simple".

3.3.1 Workout Selection

Figure 3.5 shows how the activity looks like, when the user presses on the first button, in the home view. The user can pick an exercise and set how many repetitions he wants to do. He can add another exercise and set the rest time (in seconds) between two repetitions. As a result of the third test, we added the toggle button, where one can set the workout to be "hard" or "easy". The easy mode will set an error margin to be 0.5 times the noise value of the exercise.

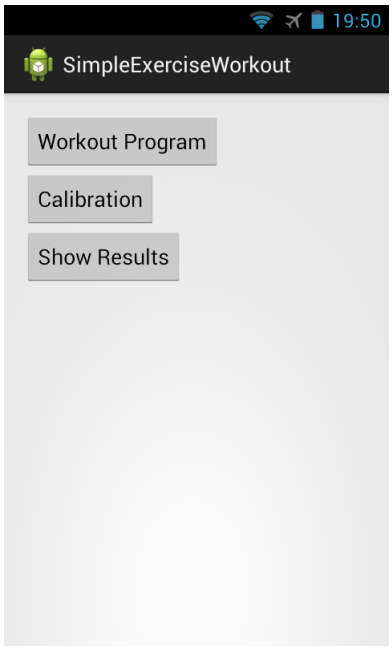

Figure 3.4: Home activity of the application.

The hard mode, however, set it to 0.2. We can also decide if we want the sounds notification during the counting or not. The Start button will start the counting.

The view of the counting is depicted in Figure 3.6. The list of exercises and their goals, picked in the workout selection view, are stored in arrays. For each exercise, we generate a counter (whose state machine is depicted in Figure 3.1), whose noise and goal value are retrieved in the database (stored during the calibration). Each repetition that has to be counted (call of the method `repetition`, that we already mentioned) will decrease the counter . When this counter is null, we start the rest timer and then start the next exercise (if there is one left to do) or the app will show the result of the user and store it in the database. It will be visible in the result view, that we will describe later.

3.3.2 Calibration

The Calibration view is depicted in Figure 3.7. The main state machine of the calibration is implemented here. After the first click on "Calibrate", the user will start the calibration and has to do one repetition. After that, he has to press the "Stop" button and the noise value will be generated. Second click on Calibrate and he will start the second part of the calibration, doing 5 repetitions. When he presses the Stop button, if the app counted 5 repetitions, the goal value will be generated and the given exercise will be added or updated in the database,

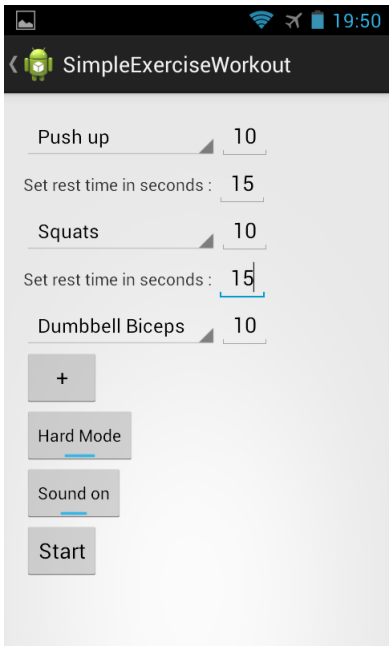

Figure 3.5: We can pick the exercises using the spinner (where all calibrated exercises in the database are listed) and set he number of repetitions to do. If there are more than one exercise, the user has to chose a rest time in seconds between two exercises. There is an add button (labeled with "+"), that add a field for the rest time, a spinner to pick the exercise and a field to set the number of repetitions. We have two toggle buttons, one to set the hard or easy mode, another to set the sounds during the counting to on or off. Finally, the Start button open a new activity, where the counting is done.

with the computed noise and goal value.

3.3.3 Results

The view that appears after pressing the Result button on the home page is depicted in Figure 3.8. For each exercise in the database, we show:

- The date and time at which the exercise has been inserted in the database
- The workout number, in which it was executed
- The name of the exercise
- The number of repetitions the user had to do
- The time the user needed to do those repetitions

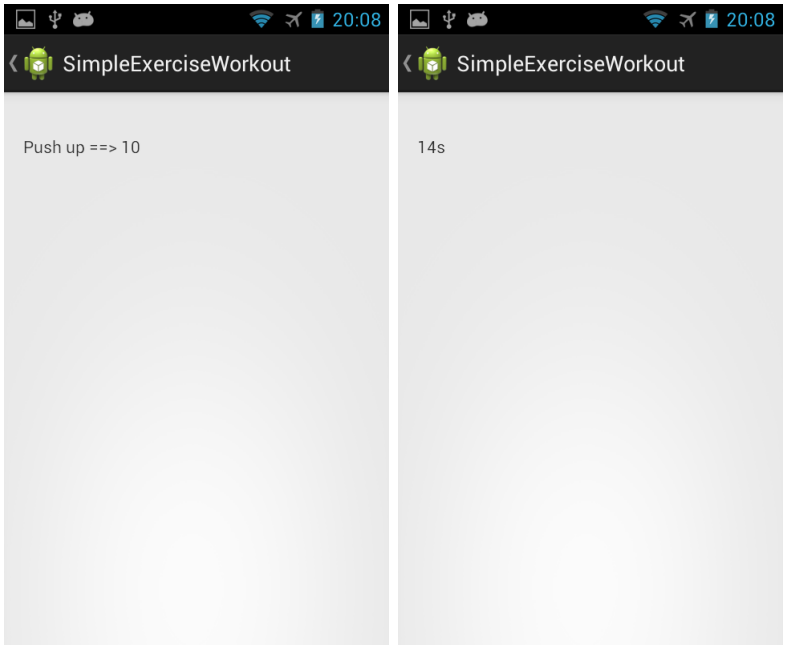

Figure 3.6: The left figure shows how the counting view look like. We can see the name of the exercise (here Push-up) and the number of repetitions left to do. In the right figure, we can see how the view looks like, between two exercises. We have a counter showing the time left, before the next exercise counting will start.

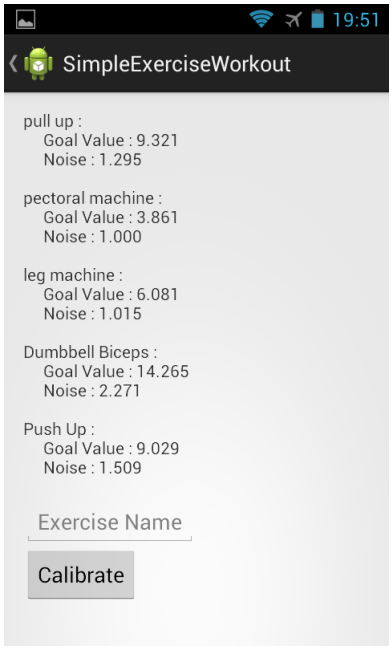

Figure 3.7: We have first, the list of exercises already calibrated. In the field, we can write an exercise name. The button Calibrate starts the calibration, as described in the state machine on Figure 3.2. If the exercise name already exists, it is updated in the database, otherwise it is simply added when the calibration is done, with its goal and noise value.

Figure 3.8: This activity lists the results of the workouts.

Chapter 4

Further Development

4.1 The Application

Due to time restrictions, we were not able to develop a perfect application. We saw that there are a few things that could be improved. For instance, we saw in Section 3.2 that the repetitions that are slightly bad are still counted. This could be avoided, probably by being more precise on the definition of the error margin, when the algorithm count the repetitions. We set it to 0.2 for the hard mode. But we saw that this margin could depend to the exercise. On the last test, the counter on abs generated no false positives, but on push-ups it did.

Another possible improvement could be on the result page. We just listed the results, without any further analysis. But we could think of a feature, where the user could see if he improved himself, between two workout sessions. A search option could be useful. For instance, one could want to see what were his results for the push-ups.

Another nice feature, but that requests more attention and work, could be a network implementation, where a user could compare himself with his friends or with other users. We could also imagine some kind of list of goals or achievements, that any user could try himself to do. Or simply, someplace where there are some lists of workout, that the user could import and try by himself on his own app.

4.2 Further Research

We did not have enough time to deepen the principle component analysis, for this project. We just could collect some data and retrieve their principle components, but no other analysis could have been done with it.If one could define a typical behavior for a given exercise, it should be possible to build an algorithm that would recognize any kind of movement and differentiate an exercise with some random movement. For instance, one could just press a button, then do some push-ups, stand-up afterwards, walk a little bit, pick up a dumbbell, do some

biceps curl, press a stop button, and the app will tell him how many push-ups he did and how many biceps curls. Intuitively, we could imagine that this kind of feature could be implemented using the principle component analysis.

Our application can only handle simple movements, like push-ups or abs. We can imagine a more complex exercise, which could, for instance, be compound of several rest-to-goal movements.

Bibliography

[1] Apple Inc., iPod Nano.

[2] Indiegogo: official website.

[3] Atlas Wearables, Atlas Fitness Tracker

[4] Android API Guide *Sensors Overview*.

[5] Jonathon Shlens, *A Tutorial on Principal Component Analysis*, Center for Neural Science, New York University and Systems Neurobiology Laboratory, Salk Insitute for Biological Studies La Jolla, April 22, 2009.