SMART MOVEMENT DETECTION FOR ANDROID PHONES

A Degree Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Pablo Navarro Morales

In partial fulfilment
of the requirements for the degree in
TELECOMMUNICATION ENGINEERING

Advisor: Sergio Bermejo

Barcelona, October 2016

Abstract

This project describes a decision tree based pedometer algorithm and its implementation on Android using machine learning techniques. The pedometer can count steps accurately and It can discard irrelevant motion. The overall classification accuracy is 89.4%. Accelerometer, gyroscope and magnetic field sensors are used in the device. When user puts his smartphone into the pocket, the pedometer can automatically count steps. A filter is used to map the acceleration from mobile phone's reference frame to the direction of gravity. As a result of this project, an android application has been developed that, using the built-in sensors to measure motion and orientation, implements a decision tree based algorithm for counting steps.

Resumen

Este proyecto describe un algoritmo para un podómetro basado en un árbol de decisiones y su aplicación en Android utilizando técnicas de aprendizaje automático. El podómetro puede contar los pasos con precisión y se puede descartar el movimiento irrelevante. La precisión de la clasificación general es del 89,4%. Un acelerómetro, un giroscopio y un sensor de campo magnético se utilizan en el dispositivo. Cuando el usuario pone su teléfono en el bolsillo, el podómetro puede contar automáticamente pasos. Un filtro se utiliza para asignar la aceleración del sistema de referencia de teléfono móvil a la dirección de la gravedad. Como resultado de este proyecto, la aplicación para Android que se ha desarrollado, utilizando los sensores incorporados para medir el movimiento y orientación, implementa un algoritmo basado árbol de decisión para contar los pasos.

Contents

List of figures

1 - Introduction

Commonly used pedometers are often built as separate products and their accuracy is typically affected by random motions. In this project, we present a method to count steps of walking using a mobile phone. We use several sensors to extract signal features and a decision tree to perform data classification. Gyroscopes and accelerometers are widely used to detect human motions [1]. Gyroscope sensor is used to measure the angular velocity of an object.

The work presented here uses gyroscope to measure angular velocity of user's thigh, when the phone is in the user's pocket shown in Figure 1. The accelerometer can be used as a sensor to measure the acceleration of an object [2]. The magnetic field sensor is often used in global positioning system navigation. In this work, data from this sensor are used to generate a rotation matrix. Using the matrix and the original acceleration, the vertical acceleration can be determined. Decision tree is one of the predictive modeling approaches used in statistics, data mining and machine learning. In a decision tree [3], leaves represent target values, which are also called class labels, and branches represent measurements about an item, which is also called a feature.

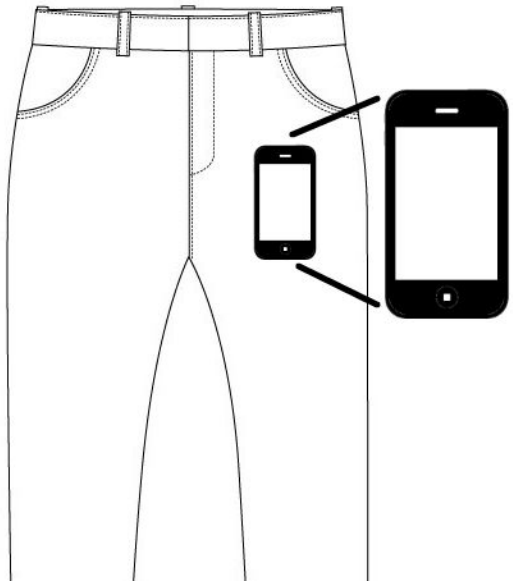

Figure 1. Mobile phone in user pocket.

2 - Background

2.1 - Introduction

Pedometers, now popular as an everyday exercise progress monitor and motivator, can encourage individuals to compete with themselves in getting fit and losing weight. Early designs used a weighted mechanical switch to detect steps, plus a simple counter. When these devices are shaken, one can hear a metal ball sliding back and forth, or a pendulum striking stops as it swings.

Today, advanced pedometers rely on microelectromechanical systems (MEMS) inertial sensors and sophisticated software to detect true steps with high probability; MEMS inertial sensors permit more accurate detection of steps and fewer false positives. Taking advantage of the low cost and minimal space- and power requirements of MEMS inertial sensors, pedometers are being integrated into an increasing number of portable consumer electronic devices such as music players and mobile phones.

2.2 - Hardware

According to the Canalys's Q4 2015 [4] global country-level smartphone market report, Google's Android has become the most popular mobile platform. Android consists of a kernel based on the Linux kernel, with middleware, libraries and APIs written in C and application software running on an application framework which includes Java-compatible libraries based on Apache Harmony [5]. Android uses the Dalvik virtual machine with just-in-time compilation to run compiled Java code, newer versions have substituted Dalvik for ART which improves the performance of the virtual machine. Besides, Android has a large community of developers writing applications that extend the functionality of the devices. One of the attractive features of Android is that Android devices have multiple different types of hardware that are built in and accessible to developers. Android can use video/still cameras, touchscreens, GPS, accelerometers, gyroscopes, magnetometers, proximity and pressure sensors, thermometers, etc. Because of additional hardware support, Android is more suitable for creating creative applications than other smartphones [6].

This project applied Android to develop an intelligent pedometer. The user's walking motion was detected via Android sensor. Pedometer application will analyze the signal, extract significant features and count steps using processing signal algorithms and machine learning techniques.

2.3 - Technology

The technology for a pedometer includes a mechanical sensor and software to count steps. Early forms used a mechanical switch to detect steps together with a simple counter. If one shakes these devices, one hears a lead ball sliding back and forth, or a pendulum striking stops as it swings. Today advanced step counters rely on MEMS inertial sensors and sophisticated software to detect steps. The use of MEMS inertial sensors permits more accurate detection of steps and fewer false positives. In this project we make use of three sensors: accelerometer, magnetometer and gyro.

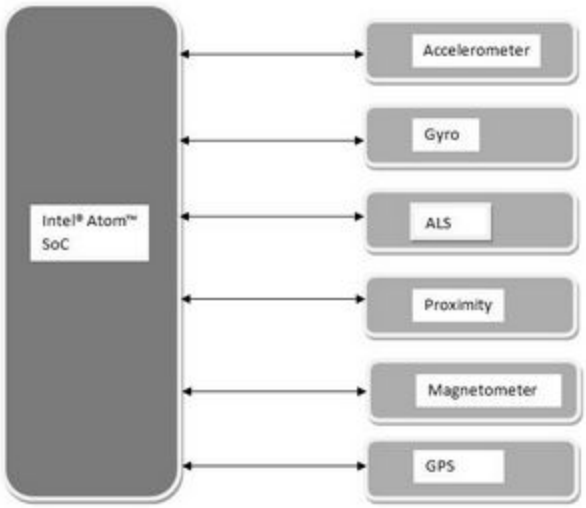

Figure 2. Typical sensors included on an Intel Atom SoC used on Android devices.

2.3.1 - Accuracy

The accuracy of step counters varies widely between devices. Typically, step counters are reasonably accurate at a walking pace on a flat surface if the device is placed in its optimal position. Although traditional step counters get affected dramatically when placed at different angles and locations, recent advances have made them more robust to those non-ideal placements. Still, most step counters falsely count steps when a user is driving a car or makes other habitual motions that the device encounters throughout the day. This error accumulates for users with moderate commutes to work. Accuracy also depends on the step-length the user enters. Best pedometers are accurate to within ± 5% error [7].

2.3.2 - Integration in personal devices

Pedometers can be found in a lot of different devices nowadays, an example:

- **Apple iPod Nano**

  The 5th and 6th generation iPod Nano by Apple features an integrated accelerometer.

- **Nike&iPod**

  Apple and Nike, offer the Nike+iPod Sports Kit, which uses a motion sensor that fits into a Nike shoe or in a pocket worn on the laces of other brands of shoes. The sensor communicates with an iPhone (3GS or higher), iPod touch (2nd generation or higher), iPod nano (4th generation or higher), or dedicated adapter to transmit workout information such as elapsed time, distance traveled, and calories burned.

- **Apple iPhone 5**

  The iPhone 5s was the first iPhone to contain an Apple Motion Coprocessor which was denoted the M7 chip paired with the first 64-bit ARM-based Apple processor, the Apple A7 SoC. The addition of the separate always on coprocessor allows the main CPU to snooze while it tracks the motion of the phone, through the use of an inertial measurement unit (IMU) consisting of an accelerometer, MEMS gyroscope and digital compass. This means that it will know when you're jogging or when you're in the car, and can take that information and store it without needing to drain the battery by having the main CPU run. It can retrofit the data to apps that you download at a later date, meaning any M7-enabled app that uses the new CoreMotion API will be able to give you information on recent training.

- **Apple iPhone 6**

  The iPhone 6 and 6 Plus contains the next generation of the Apple Motion Coprocessors with the M8 motion coprocessor, this chip was paired with the vastly improved Apple A8 SoC processor and gained the added sensor input of a Bosch Sensortech Barometer allowing the M8 to sense changes in elevation by the change in barometric pressure.

- **Apple iPhone 6s**

  The iPhone 6s and 6s Plus improved the Apple Motion Coprocessors by adding integrating it into the die of the new Apple A9 SoC processor. This saves space allowing for the reduction of the logic board size as well as reduced power usage within the

phone. This chip is also at the heart of the iPhone SE. A variant of the Apple A9, the Apple A9X also incorporates the M9 processor on-die and drives the Apple iPad Pro.

- **Fitbit**

  The Fitbit is an always-on electronic pedometer, that in addition to counting steps also displays distance traveled, altitude climbed (via a number of flights of steps count), calories burned, current intensity, and time of day. Worn in an armband at night, it also purports to measure the length and quality of a user's sleep.

- **Android**

  Android integrates a step counter with version 4.4 (KitKat) [8].

  A device already supporting this sensor is the Nexus 5. Another smartphone is the Samsung Galaxy S5, which features a built-in pedometer that uses the S Health software to display your daily step counts.

3 - System Structure

The system structure of the proposed pedometer is shown in Figure 3. Signals of original acceleration, angular velocity and magnetic field are recorded with a sampling frequency of 100 Hz. Then the signals will be cut into small segments. After that, original acceleration is mapped to the direction of the gravity. The features are extracted from each segment. Finally, all features are sent to the decision tree to classify each segment.

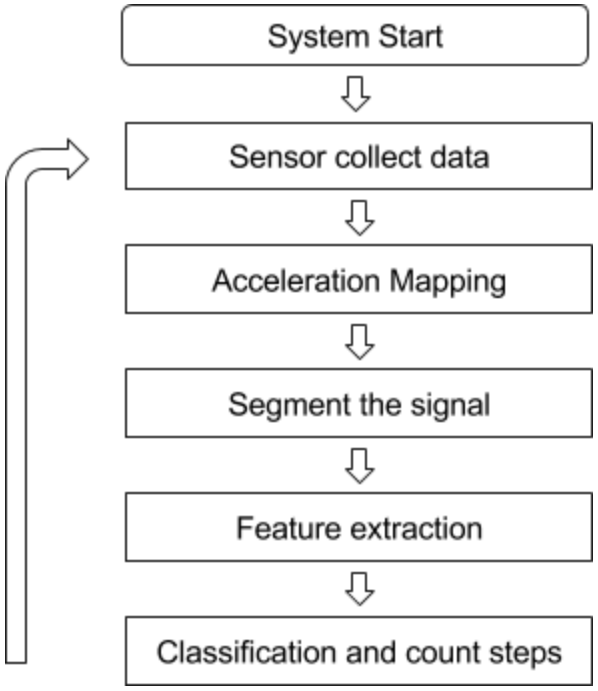

Figure 3. System chart of the pedometer

3.1 - Collecting the data

The raw data necessary for feeding the algorithms is obtained from the Android sensors systems. In our case the Android platform provides us with hardware sensors that let us monitor the walking activity.

3.2 - Acceleration mapping

Due to the inexactitudes and limitations of the hardware sensors [9], the input raw data that we receive needs to be filtered and mapped. Therefore a filter is used for smoothing the signal from inherent noise of the hardware sensors.

Together with the smoothing filter a mapping matrix is used. As vertical acceleration is one of the fundamental signals introduced by the walk and the data provided by the sensors is respected in the device reference frame, we need to map the the acceleration data to the direction of the acceleration of the gravity, to do so we use a rotation matrix mechanism whose fundamentals are the magnetic and gyroscope sensors.

3.3 - Processing the signal

The main purpose of this phase is to remove the information from the filtered data that is not necessary. This part makes possible to enable the next phases. In our case a segmentation algorithm is used that using pattern recognition techniques is able to make segments of the signal that include the most valuable information to extract features from.

3.4 - Feature extraction

To perform any classification process a set of features has to be extracted. They are attributes able to characterize without ambiguity each motion mode. The features selection plays a key role in the entire classification process and strongly affects the final performance of the designed classifier. In particular, to reduce the probability of miss-classification, features have been chosen in order to [10]:

- Minimize the distance among different features belonging to the same class.
- Maximize the distance among different features belonging to different classes.

In this work the feature extraction is performed after preprocessing the signal with a segmentation algorithm that is in charge of recognizing walking patterns in the sensors input data provided by the sensors.

The following features have been identified for the classification process:

- The gyroscope minimum acceleration.
- The maximum vertical acceleration.
- The minimum vertical acceleration.
- The position of the maximum acceleration.
- The variance of the acceleration.

3.5 - Decision making

In general a classification process can be considered as a mapping function [10] that given an input characterized by a set of *d* features, $f = [f_1, f_2, \ldots, f_d]$ assigns each feature vector to one of the n possible classes $c = [c_1, c_2, \ldots, c_d]$. In our cases the features are the attributes and the classes are the user's motion nodes.

Classifier algorithms are traditionally divided in two groups:

- Supervised classifier: the labelled data, whose class is known, is used to train the classifier and then to assign unlabeled data to one of the known classes.
- Unsupervised classifier: here the classes are not known a priori but are defined when the classification is completed.

For the classification algorithm used in this work that makes uses of handle MEMS signals, the classes and their characteristics are defined during the classifier design process and a supervised approach has been adopted. The classification of the user's state is performed by a decision tree classifier.

A decision tree is a non-parametric classifier with the form of a tree whose leaves consist of all the possible classes. In correspondence of each tree's internal node a test regarding or more features is specified. Traversing the decision tree from the root to the leaves any input observation can be classified.

4 - Sensors

4.1 - Sensors framework

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if we want to monitor three-dimensional device movement or positioning, or we want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dew point, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform supports three broad categories of sensors:

- Motion sensors
- These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- Environmental sensors
- These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- Position sensors
- These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

4.2 - Driver architecture

On an Android system, the sensor data is read by the Linux driver on the Kernel space, and sent to the API by the HAL driver. Therefore, the sensor data could be converted on either the Linux driver level or HAL level.

The figure below represents the Android sensor stack. Each component communicates only with the components directly above and below it, though some sensors can bypass the sensor hub when it is present. Control flows from the applications down to the sensors, and data flows from the sensors up to the applications.

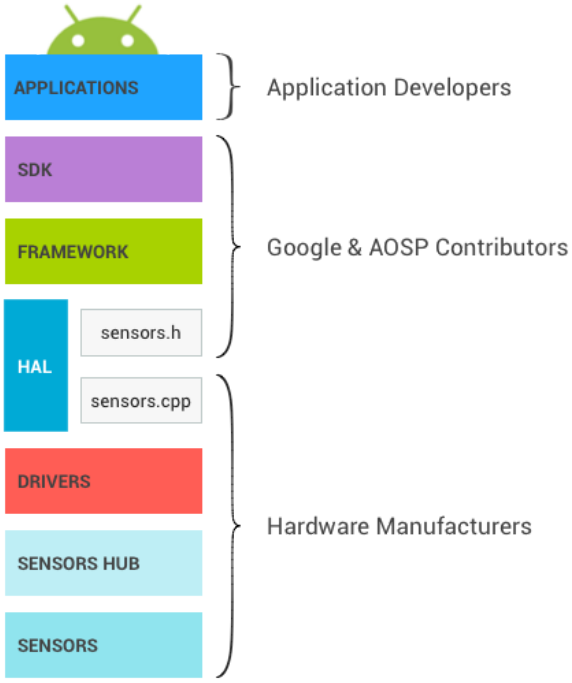

Figure 4. Layers of the Android sensor stack and their respective owner

4.2.1 - SDK

Applications access sensors through the Sensors SDK (Software Development Kit) API. The SDK contains functions to list available sensors and to register to a sensor.

When registering to a sensor, the application specifies its preferred sampling frequency and its latency requirements.

- For example, an application might register to the default accelerometer, requesting events at 100Hz, and allowing events to be reported with a 1-second latency [11].
- The application will receive events from the accelerometer at a rate of at least 100Hz, and possibly delayed up to 1 second.

4.2.2 - Framework

The framework is in charge of linking the several applications to the HAL (Hardware Abstraction Layer). The HAL itself is single-client. Without this multiplexing happening at the framework level, only a single application could access each sensor at any given time.

- When a first application registers to a sensor, the framework sends a request to the HAL to activate the sensor.

- When additional applications register to the same sensor, the framework takes into account requirements from each application and sends the updated requested parameters to the HAL.
    - The sampling frequency will be the maximum of the requested sampling frequencies, meaning some applications will receive events at a frequency higher than the one they requested.
    - The maximum reporting latency will be the minimum of the requested ones. If one application requests one sensor with a maximum reporting latency of 0, all applications will receive the events from this sensor in continuous mode even if some requested the sensor with a non-zero maximum reporting latency.
- When the last application registered to one sensor unregisters from it, the frameworks sends a request to the HAL to deactivate the sensor so power is not consumed unnecessarily.

4.2.3 - HAL

The Sensors Hardware Abstraction Layer (HAL) API is the interface between the hardware drivers and the Android framework. It consists of one HAL interface sensors.h and one HAL implementation we refer to as sensors.cpp.

The interface is defined by Android and AOSP contributors, and the implementation is provided by the manufacturer of the device.

4.2.4 - Kernel driver

The sensor drivers interact with the physical devices. In some cases, the HAL implementation and the drivers are the same software entity. In other cases, the hardware integrator requests sensor chip manufacturers to provide the drivers, but they are the ones writing the HAL implementation.

In all cases, HAL implementation and kernel drivers are the responsibility of the hardware manufacturers, and Android does not provide preferred approaches to write them.

4.2.5 - Sensor hub

The sensor stack of a device can optionally include a sensor hub, useful to perform some low-level computation at low power while the SoC can be in a suspend mode. For example, step counting or sensor fusion can be performed on those chips. It is also a good place to implement sensor batching, adding hardware FIFOs for the sensor events. How the sensor hub is materialized depends on the architecture. It is sometimes a separate chip, and sometimes included on the same chip as the SoC. Important characteristics of the sensor hub is that it should contain sufficient memory for batching and consume very little power to enable

implementation of the low power Android sensors. Some sensor hubs contain a microcontroller for generic computation, and hardware accelerators to enable very low power computation for low power sensors.

How the sensor hub is architectured and how it communicates with the sensors and the SoC (I2C bus, SPI bus, …) is not specified by Android, but it should aim at minimizing overall power use.

One option that appears to have a significant impact on implementation simplicity is having two interrupt lines going from the sensor hub to the SoC: one for wake-up interrupts (for wake-up sensors), and the other for non-wake-up interrupts (for non-wake-up sensors).

4.2.5 - Sensors

Those are the physical MEMs chips making the measurements. In many cases, several physical sensors are present on the same chip. For example, some chips include an accelerometer, a gyroscope and a magnetometer.

Some of those chips also contain some logic to perform usual computations such as motion detection, step detection and 9-axis sensor fusion.

Although the CDD power and accuracy requirements and recommendations target the Android sensor and not the physical sensors, those requirements impact the choice of physical sensors. For example, the accuracy requirement on the game rotation vector has implications on the required accuracy for the physical gyroscope. It is up to the device manufacturer to derive the requirements for physical sensors.

4.3 - Sensors coordinate system

There are a number of coordinate systems to be aware of when developing with Android devices. Is important to take into account the coordinate system that every sensor is using, because we can draw false conclusions if not taken into account.

4.3.1- World Coordinate System

The world coordinate system in Android is the ENU (east, north, up) coordinate system. This is different from the NED (north, east, down) coordinate system that is commonly used in aviation Figure 5.

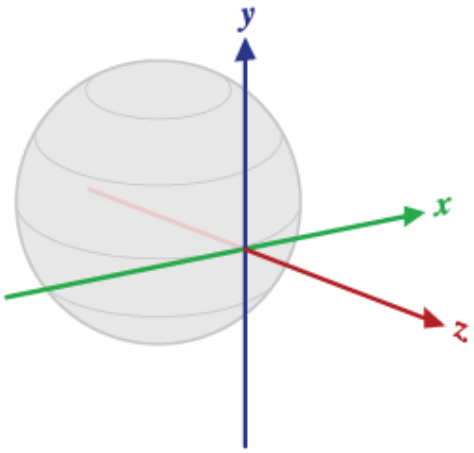

Figure 5. Universal coordinate system.

4.3.2 - Local Coordinate System

In general, the sensor framework uses a standard 3-axis coordinate system to express data values.

For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation as can be seen in Figure 6. When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values.

The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves.

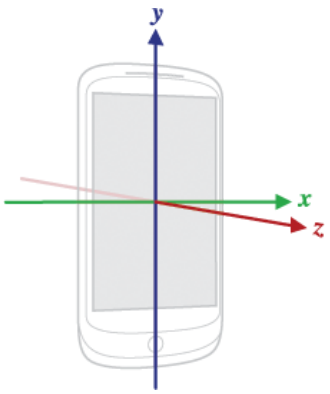

Figure 6. The axis of the gyroscope on a mobile phone.

4.3.3 - Other Coordinate System

The method SensorManager.getOrientation(), which is commonly used to get the orientation vector of the device from a rotation matrix, uses a third reference coordinate system that is not the world coordinate system. A WND (west, north, down) coordinate system is used, which is different from both the ENU and NED coordinate systems that are more common. Also worth noting is that the order of the axis returned in the method are different from those returned by the sensors.

When SensorManager.getOrientation() returns, the array values is filled with the result:

- values[0]: azimuth, rotation around the Z axis.
- Values[1]: pitch, rotation around the X axis.
- Values[2]: roll, rotation around the Y axis.

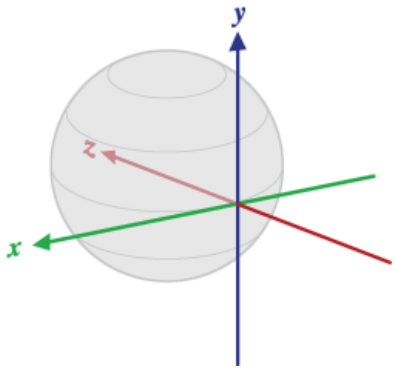

Figure 7. The axis of the gyroscope on a mobile phone.

4.4 - Sensors Offset

The accuracy of a sensor and the offset of a sensor can manifest in a similar way [12]. A reasonably accurate acceleration sensor would measure the gravity of earth when the axis of the sensor was pointed straight up towards the sky.

Determining the accuracy is a little more complicated than this because the gravity of earth actually changes slightly depending where on earth we are and because of sensor offset. If the positive axis of a an acceleration sensor overestimates the gravity of earth, and the negative axis of the same sensor under-estimates the gravity of earth, there is likely some offset occurring where the center of the axis is slightly skewed towards the positive or negative axis.

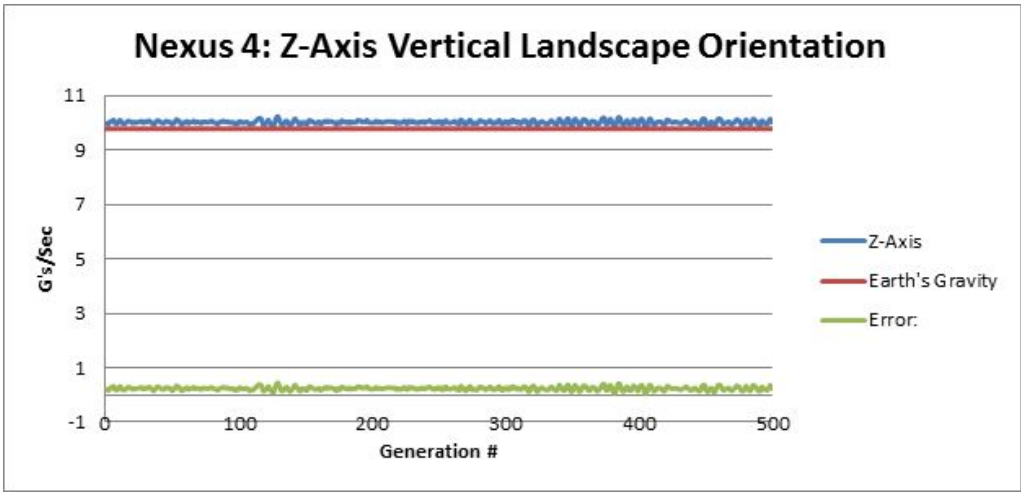

Figure 8. Positive Z-Axis slightly overestimation.

4.5 - Sensors Noise

Android devices black-box, meaning we have no idea what is actually going on under the hood, we just get the sensor outputs, their sensor implementations and they vary by the model and manufacturer. Filters may or may not be applied to sensors before providing an output, and some sensors even have filters designed into them. It is useful to have some idea of what kind of filtering is already occurring in cases where we would like to do filtering of our own. Knowing how much noise exists on the sensor outputs is a good place to start.

The actual noise seen on the sensor outputs may be larger than this reported error due to environmental noise (thermal, Vdd regulation, mechanical accelerations) on the sensor device and then referring to data sheets to determine the expected noise density of the sensor. The noise density, denoted in units of $ug/\sqrt{Hz}$, is defined as the noise per unit of square root bandwidth and can be used to determine the expected noise output from a sensor [12].

Device	Sensor	Noise Density	Maximum Output Frequency
Nexus 4	MPU-6050	$400\,ug/\sqrt{Hz}$	193Hz
Galaxy S4	Bosch Sensortec	$150\,ug/\sqrt{Hz}$	100Hz
Droid Razr	STMicro LISD2H	$220\,ug/\sqrt{Hz}$	50Hz

Figure 9. Android accelerometer specifications.

The results of some testings show that the actual noise seen on the sensor outputs may be larger than this reported error due to environmental noise (thermal, Vdd regulation, mechanical accelerations) on the sensor [12].

As shown in Figure 10 the error for the Nexus 4 is significantly higher than the GS4 or Droid Razr. It is possible that the Nexus 4 is filtering the acceleration sensor where the other two devices are not.

Device	Expected Noise	Measured Noise	Error
Nexus 4	2390 $ug/\sqrt{Hz}$	2390 $ug/\sqrt{Hz}$	48%
Galaxy S4	2390 $ug/\sqrt{Hz}$	1430 $ug/\sqrt{Hz}$	8%
Droid Razr	2390 $ug/\sqrt{Hz}$	1680 $ug/\sqrt{Hz}$	16%

Figure 10. Android testing results

4.6 - Position sensors

The Android platform provides two sensors that let us determine the position of a device: the geomagnetic field sensor and the accelerometer. The Android platform also provides a sensor that lets us determine how close the face of a device is to an object (known as the proximity sensor). The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor. Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call). For determining a device's orientation, we use the readings from the device's accelerometer and the geomagnetic field sensor.

Position sensors are useful for determining a device's physical position in the world's frame of reference. For example, we can use the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic north pole. We can also use these sensors to determine a device's orientation in our application's frame of reference. Position sensors are not typically used to monitor device movement or motion, such as shake, tilt, or thrust.

The geomagnetic field sensor and accelerometer return multidimensional arrays of sensor values for each SensorEvent. For example, the geomagnetic field sensor provides geomagnetic field strength values for each of the three coordinate axes during a single sensor event. Likewise, the accelerometer sensor measures the acceleration applied to the device during a sensor event.

4.7 - Computing the devices orientation

By computing a device's orientation, we can monitor the position of the device relative to the earth's frame of reference specifically, the magnetic north pole.

The system computes the orientation angles by using a device's geomagnetic field sensor in combination with the device's accelerometer. Using these two hardware sensors, we can provide data for the following three orientation angles:

- **Azimuth (degrees of rotation about the -z axis).** This is the angle between the device's current compass direction and magnetic north. If the top edge of the device faces magnetic north, the azimuth is 0 degrees; if the top edge faces south, the azimuth is 180 degrees. Similarly, if the top edge faces east, the azimuth is 90 degrees, and if the top edge faces west, the azimuth is 270 degrees.

- **Pitch (degrees of rotation about the x axis).** This is the angle between a plane parallel to the device's screen and a plane parallel to the ground. If we hold the device parallel to the ground with the bottom edge closest to us and tilt the top edge of the device toward the ground, the pitch angle becomes positive. Tilting in the opposite direction— moving the top edge of the device away from the ground—causes the pitch angle to become negative. The range of values is -180 degrees to 180 degrees.

- **Roll (degrees of rotation about the y axis)**. This is the angle between a plane perpendicular to the device's screen and a plane perpendicular to the ground. If we hold the device parallel to the ground with the bottom edge closest to us and tilt the left edge of the device toward the ground, the roll angle becomes positive. Tilting in the opposite direction—moving the right edge of the device toward the ground— causes the roll angle to become negative. The range of values is -90 degrees to 90 degrees.

5 - Accelerometer

5.1 - Introduction

An accelerometer is an electromechanical device that measures acceleration forces in units of m/s² or G-force (the gravity of earth) which is about 9.8 m/s².

Accelerometers measure both the static acceleration, like the gravity field of earth, and dynamic acceleration caused by the movement of the accelerometer. In fact, the accelerometer cannot differentiate between static and dynamic acceleration. This means an accelerometer can be used to determine the tilt of the device by measuring static acceleration or the linear acceleration of the device by measuring dynamic acceleration. However, an accelerometer cannot measure both static and dynamic acceleration at the same time. Accelerometers usually measure acceleration/gravity it two or three-axis, but on Android devices it is almost always three-axis Figure 11.

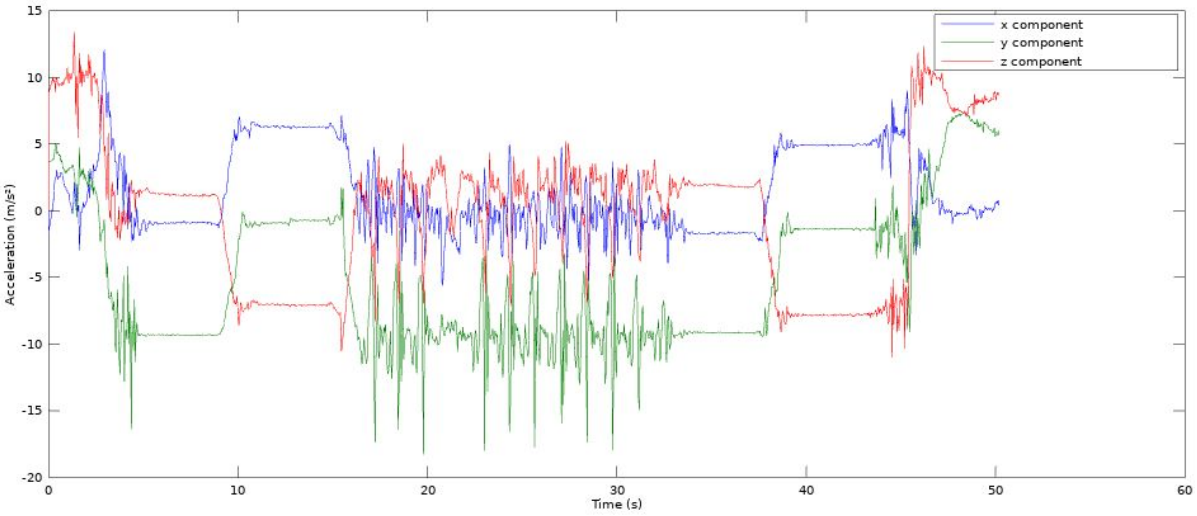

Figure 11. Acceleration data over time.

A working example of how the accelerometer works can be seen in Figure 12:

- The norm of $< x, y, z >$ should be close to 0 when in the free fall.
- When the device lies flat on the table and is pushed on its left side toward the right, the acceleration of the device is 0 m/s² minus the force of gravity -9.81 m/s²

- When the device lies flat on a table and is pushed toward the sky, the acceleration value is greater than 9.40 m/s², which corresponds to the acceleration of the device minus the force of gravity.

Mobile Position	X	Y	Z
UP	0	9.81 m/s²	0
LEFT	9.81 m/s²	0	0
DOWN	0	-9.81 m/s²	0
RIGHT	-9.81 m/s²	0	0
FRONT UP	0	0	9.81 m/s²
BACKUP	0	0	-9.81 m/s²

Figure 12. Acceleration values on each Axis for different device positions.

5.2 - Implementation

There are a number of different implementations of accelerometers available. Some rely on the capacitance between two objects. If a force from acceleration moves one of the objects, the capacitance between the objects will change. This capacitance can be converted to a voltage which can then be used to measure the force of the acceleration. Another implementation of accelerometers uses piezoelectric effect which rely on crystal structures that produce a voltage when an acceleration force is applied to them. More advanced accelerometers use lasers to measure acceleration.

Accelerometers with an analog output will produce a voltage that is directly proportional to the sensed acceleration. Digital accelerometers usually feature a serial interface be it SPI or I²C. Some digital accelerometers use pulse width modulation (PWM) for their output. This means there will be a square wave of a certain frequency, and the amount of time the voltage is high will be proportional to the amount of acceleration. Digital accelerometers are advantageous because they are less susceptible to noise than their analog counterparts.

5.3 - Caveats

All accelerometers suffer from a great deal of noise, especially inexpensive accelerometers found in mobile devices. Inexpensive accelerometers found in mobile devices are commonly referred to as "tilt-sensors" in that they are intended to measure orientation changes which are mostly static measurements and involve large shifts in axis-measurement magnitudes. In other words, a tilt-sensor is great for watching gravity go from 9.82m/s (gravity of earth) in one axis to 9.82m/s in another axis as we rotate the device, but not so great for measuring small changes in acceleration.

5.4 - Sensor Code

On Android devices, we can access the device's accelerometer sensor using the Android sensors API. In our project we will request sensor updates with 100Hz frequency. Also the sensor events are registered in *onResume()* and removed in *onPause()*. This is done so the acceleration sensor is stopped when the user navigates away from the activity to save battery life as sensors use a lot of battery life. In our project we have created a background service that collects the data independently of the activity.

This service called StepManager is fired up as soon as the MainActivity is show to the user and is stop and soon as the MainActivity calls *onDestroy()*.

6 - Gyroscope

6.1 - Introduction

The gyroscope measures the rate of rotation in rad/s around a device's x, y, and z axis. The sensor's coordinate system is the same as the one used for the acceleration sensor. Rotation is positive in the counterclockwise direction; that is, an observer looking from some positive location on the x, y or z axis at a device positioned on the origin would report positive rotation if the device appeared to be rotating counter clockwise. This is the standard mathematical definition of positive rotation and is not the same as the definition for roll that is used by the orientation sensor.

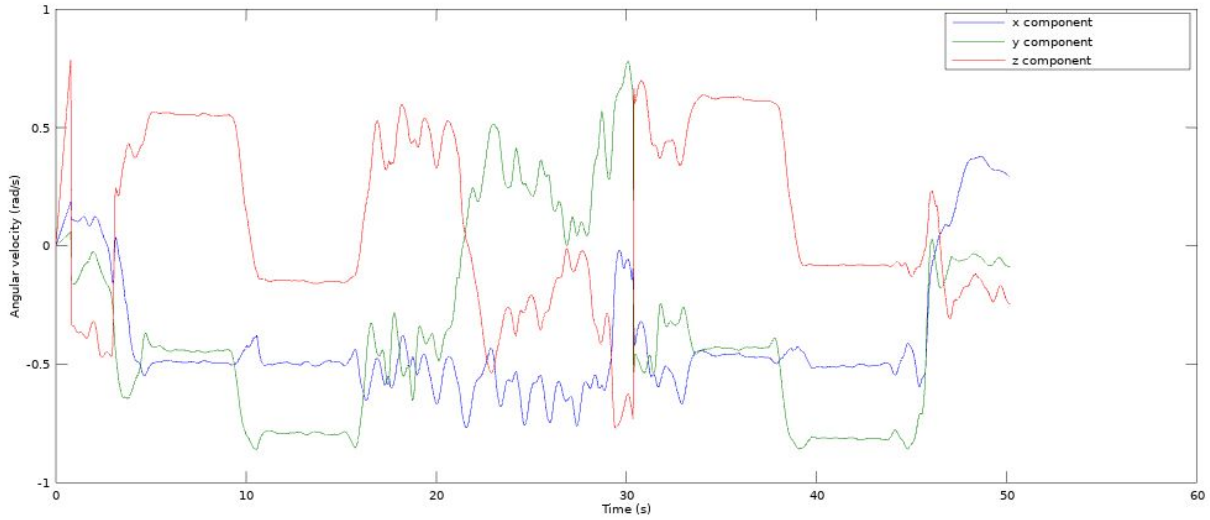

Figure 13. Angular velocity data over time.

6.2 - Implementation

Most gyroscopes on Android devices are vibrational and measure the rotation of a device with a pair of vibrating arms that take advantage of what is known as the Coriolis effect, which is caused by the Earth's rotation. By measuring changes in the direction of the vibrating arms caused by a rotation and the Coriolis effect, an estimation of the rotation can be produced. The gyroscope is one of three sensors that are always hardware based (the other two are the magnetic and the acceleration sensors) on Android devices. In conjunction with the acceleration sensor, the gyroscope can be used to create other sensors like gravity, linear acceleration or

rotation sensors. These sensors are all useful for detecting the movement of the device, which can either be a result of the user inputs or an external physical environment In our project we use it to indirectly determine the position of a device, like tilt-compensation on the magnetic sensor for a compass.

6.3 - Caveats

Like all sensors, a gyroscope is not perfect and has small errors in each measurement. Since the measurements from a gyroscope are integrated over time, these small errors start to add up and result in what is known a drift.

Over time, the results of the integration can become unreliable and some form of compensation is required to help compensate for the drift. This requires another sensor to provide a second measurement of the devices orientation that can then be used to augment the gyroscopes integration back towards the actual rotation of the device. This second sensor is usually a acceleration or magnetic sensor, or sometimes both. A weighted average, Kalman filter or complementary filter are common implementations of fusing other sensors to the gyroscope sensor, each with their own advantages and disadvantages. When we really get down into the implementations, we also run into real limitations with the "support" sensors as well. For instance, an acceleration sensor cannot determine the difference between the tilt of the device and linear acceleration, which makes for a vicious circular reference when trying to implement a linear acceleration sensor.

In fact, the Android Sensor TYPE_LINEAR_ACCELERATION is terrible at measuring linear acceleration under the influence of a physical environment such as the acceleration of a car because of the circular reference. The magnetic sensor is another option, but it is limited by the effects of hard and soft iron offsets and it can only measure roll and yaw, so it isn't perfect, either. It can take a lot of effort, fine tuning and possibly multiple sensor fusions and calibrations to get reliable estimations.

6.4 - Calibrated versus uncalibrated

As of Android 4.3, a new uncalibrated gyroscope sensor is available. No gyro-drift compensation has been performed to adjust the given sensor values. However, such gyro-drift bias values are returned separately in the result values so we use them for custom calibrations. This allows us to implement our own sensor fusions without having to worry about black-boxed underlying sensor fusions wrecking our carefully implemented custom calibrations and fusions. On the Nexus 4 and Nexus 5 devices, the uncalibrated gyroscope actually works fairly well on its own, but will eventually drift over long periods of time or after a lot of dynamic rotation. Since hardware implementations vary with each device and manufacturer, this may not be the case will all Android devices.

7 - Acceleration Mapping

7.1 - Introduction

An accelerometer can measure the static gravitational field of earth or it can measure measure linear acceleration but it cannot measure both at the same time. The acceleration given by the mobile phone is respected in the mobile phone reference frame shown in Figure 7. Vertical vibration is a significant signal induced by the walk. Therefore, original acceleration needs to be mapped to the direction of the gravity to generate the signal of vertical vibration Figure 14.

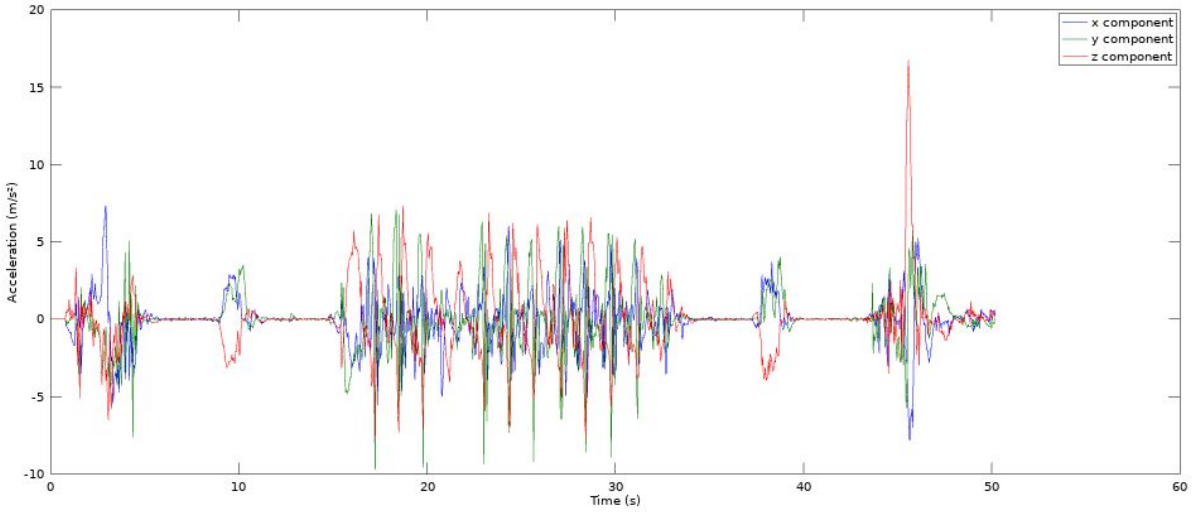

Figure 14. Linear acceleration data over time.

7.2 - Linear Acceleration

When talking about linear acceleration in reference to an acceleration sensor, what we really mean is Linear Acceleration = Measured Acceleration - Gravity so we can determine the actual acceleration of the device no matter how the device is oriented.

Android offers its own implementation of linear acceleration with Sensor.TYPE_LINEAR_ACCELERATION. Most of the time the device must have a gyroscope for this sensor type to be supported. However, some devices implement Sensor TYPE_LINEAR_ACCELERATION without a gyroscope, presumably with a low-pass filter. Regardless of the underlying implementation, Sensor TYPE_LINEAR_ACCELERATION works well for short periods of linear acceleration, but not for long periods Figure 15.

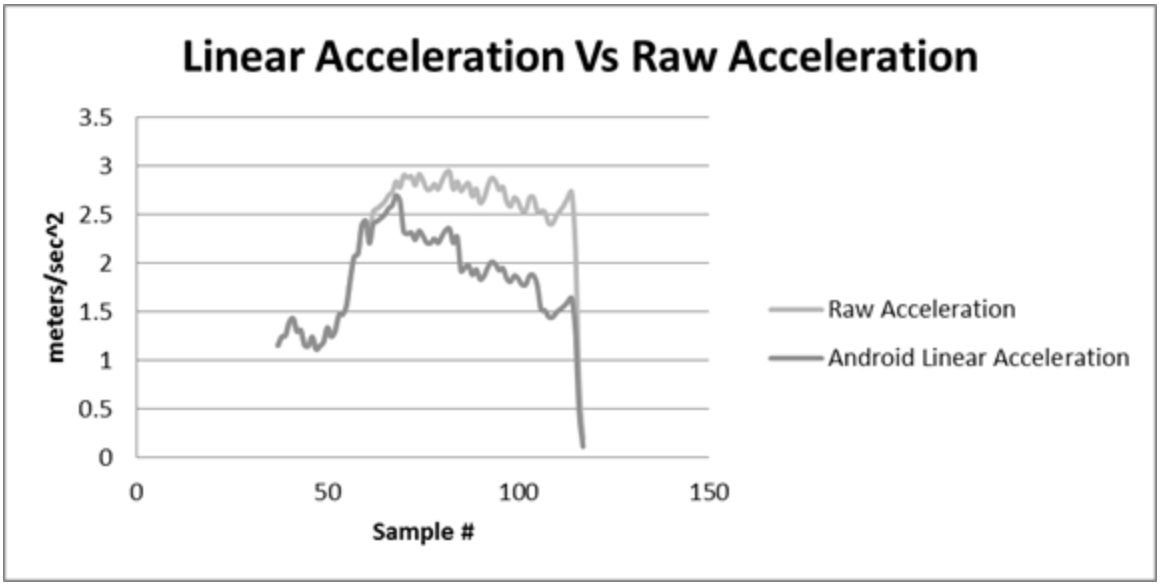

Figure 15. Linear acceleration vs Raw Acceleration over time.

In the figure above we can see how the linear acceleration linear acceleration estimation, taken Sensor TYPE_LINEAR_ACCELERATION, begins to deviate heavily from the actual acceleration, taken using Sensor TYPE_ACCELERATION, after a short period of time. This is presumably because deep under the hood of the linear acceleration algorithm, a gyroscope is used to estimate the orientation of the device, which then calculates the gravity vector which is then subtracted from the acceleration to produce linear acceleration.

The deviation occurs because the acceleration sensor is also used to compensate the drift of the gyroscope and under sustained periods of linear acceleration, the gyroscope begins to compensate for what it thinks is a long term gravity signal, but is really sustained linear acceleration. While using the API sensors is very convenient, we find the need of using a more specialized implementation.

7.3 - Vertical Acceleration

Vertical vibration is a significant signal induced by the walk [14]. Therefore, original acceleration needs to be mapped to the direction of the gravity to generate the signal of vertical vibration. There are two methods to achieve it.

In the first method, we calculate the angle between vector of linear acceleration provided by linear acceleration sensor and the vector of g provided by gravity sensor, where g denotes the

acceleration due to gravity, and |g|=9.8 m/s² . Let A_{linear} denote the linear acceleration, AGD denote the value of the acceleration in the direction of gravity, and xlinear, ylinear and zlinear denote elements of the vector of A_{linear} respectively, then they can be computed as follows:

$$\left| A_{linear} \right| = \sqrt{x^2_{\ linear} + y^2_{\ linear} + z^2_{\ linear}} \qquad (1)$$

$$cos < A_{linear} \cdot g_{linear} > \ = \frac{A_{linear} \cdot g_{linear}}{\left| A_{linear} \right| \cdot \left| g_{linear} \right|} \qquad (2)$$

$$A_{GD} = \ - cos < A_{linear} \cdot g_{linear} > \left| A_{linear} \right| \qquad (3)$$

In the second method, a rotation matrix is generated using data from the accelerometer and the magnetic field. Then the original acceleration can be mapped to the direction of gravity. Let A_{GD} denote the value of the acceleration in the direction of gravity, $M_{rotation}$ denote the rotation matrix and $A_{original}$ denote the vector of acceleration respecting to mobile phone's reference frame, then:

$$\left[0, 0, A_{GD}\right] = M_{rotation} x\, A_{original} \qquad (4)$$

In this project we use the second method which provides us an acceptable solution to obtain the data necessary for the pedometer implementation.

7.4 - Complementary Filter

The complementary filter is a frequency domain filter. In its strictest sense, the definition of a complementary filter refers to the use of two or more transfer functions [15[, which are mathematical complements of one another. Thus, if the data from one sensor is operated on by G(s), then the data from the other sensor is operated on by I-G(s), and the sum of the transfer functions is I, the identity matrix. In practice, it looks nearly identical to a low-pass filter, but uses two different sets of sensor measurements to produce what can be thought of as a weighted estimation.

In most cases, the gyroscope is used to measure the device's orientation, which can then be used to produce a gravity vector, which can then be subtracted from the acceleration vector to produce the linear acceleration vector. However, the gyroscope tends to drift due to roundoff errors and other factors.

Most gyroscopes work by measuring very small vibrations in the earth's rotation, which means they really do not like external vibrations. Because of drift and external vibrations, the gyroscope has to be compensated with a second estimation of the devices orientation, which comes from

the acceleration sensor and magnetic sensor. The acceleration sensor provides the pitch and roll estimations while the magnetic sensor provides the azimuth. A complementary filter is used to fuse the two orientations together.

It takes the form of:

$$gyro[0] = alpha * gyro[0] + (1 - alpha) * accel/magnetic[0] \quad (5)$$

Alpha is defined as alpha = timeConstant / (timeConstant + dt) where the time constant is the length of signals the filter should act on and dt is the sample period (1/frequency) of the sensor.

7.4.1 - Orientation Euler Angles Complementary Filter

The common way to get the attitude of an Android device is to use the *SensorManager.getOrientation()* method to get the three orientation angles. These two angles are based on the accelerometer and magnetometer output. In simple terms, the accelerometer provides the gravity vector (the vector pointing towards the centre of the earth) and the magnetometer works as a compass. The Information from both sensors suffice to calculate the device's orientation. However both sensor outputs are inaccurate, especially the output from the magnetic field sensor which includes a lot of noise.

The gyroscope in the device is far more accurate and has a very short response time. Its downside is the dreaded gyro drift. The gyro provides the angular rotation speeds for all three axes. To get the actual orientation those speed values need to be integrated over time. This is done by multiplying the angular speeds with the time interval between the last and the current sensor output. This yields a rotation increment. The sum of all rotation increments yields the absolute orientation of the device. During this process small errors are introduced in each iteration. These small errors add up over time resulting in a constant slow rotation of the calculated orientation, the gyro drift.

To avoid both, gyro drift and noisy orientation, the gyroscope output is applied only for orientation changes in short time intervals, while the magnetometer/accelerometer data is used as support information over long periods of time. This is equivalent to low-pass filtering of the accelerometer and magnetic field sensor signals and high-pass filtering of the gyroscope signals. The overall sensor fusion and filtering looks like in Figure 16.

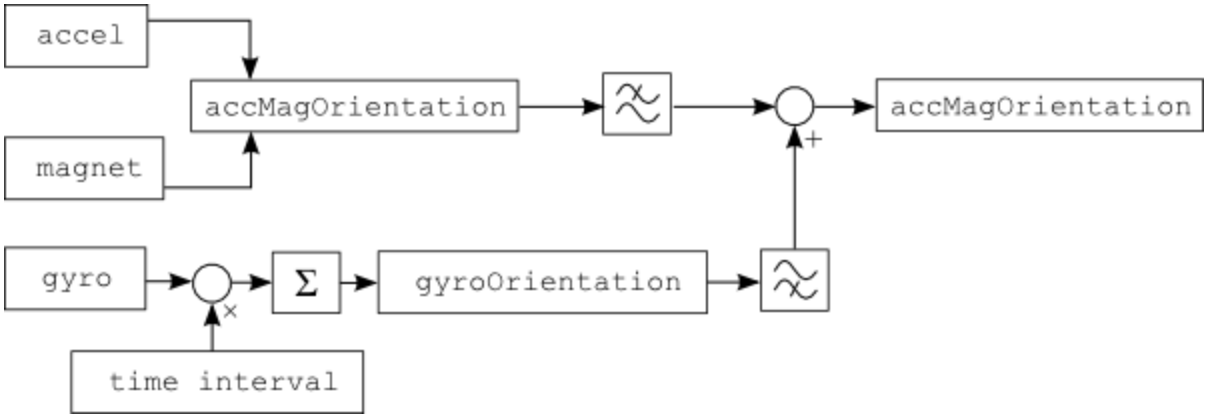

Figure 16. Filter Scheme.

The sensors provide their data at more or less regular time intervals. Their values can be shown as signals in a graph with the time as the x-axis, similar to an audio signal. The low-pass filtering of the noisy accelerometer/magnetometer signal (accMagOrientation in the Figure 16) are orientation angles averaged over time within a constant time window.

Later in the implementation, this is accomplished by slowly introducing new values from the accelerometer/magnetometer to the absolute orientation:

$$accMagOrientation = (1 - factor) * accMagOrientation + factor * newAccMagValue; \quad (6)$$

The high-pass filtering of the integrated gyroscope data is done by replacing the filtered high-frequency component from accMagOrientation with the corresponding gyroscope orientation values:

$$fusedOrientation = (1 - factor) * newGyroValue + factor * newAccMagValue; \quad (7)$$

Assuming that the device is turned 90° in one direction and after a short time turned back to its initial position, the intermediate signals in the filtering process would look something like in Figure 17.

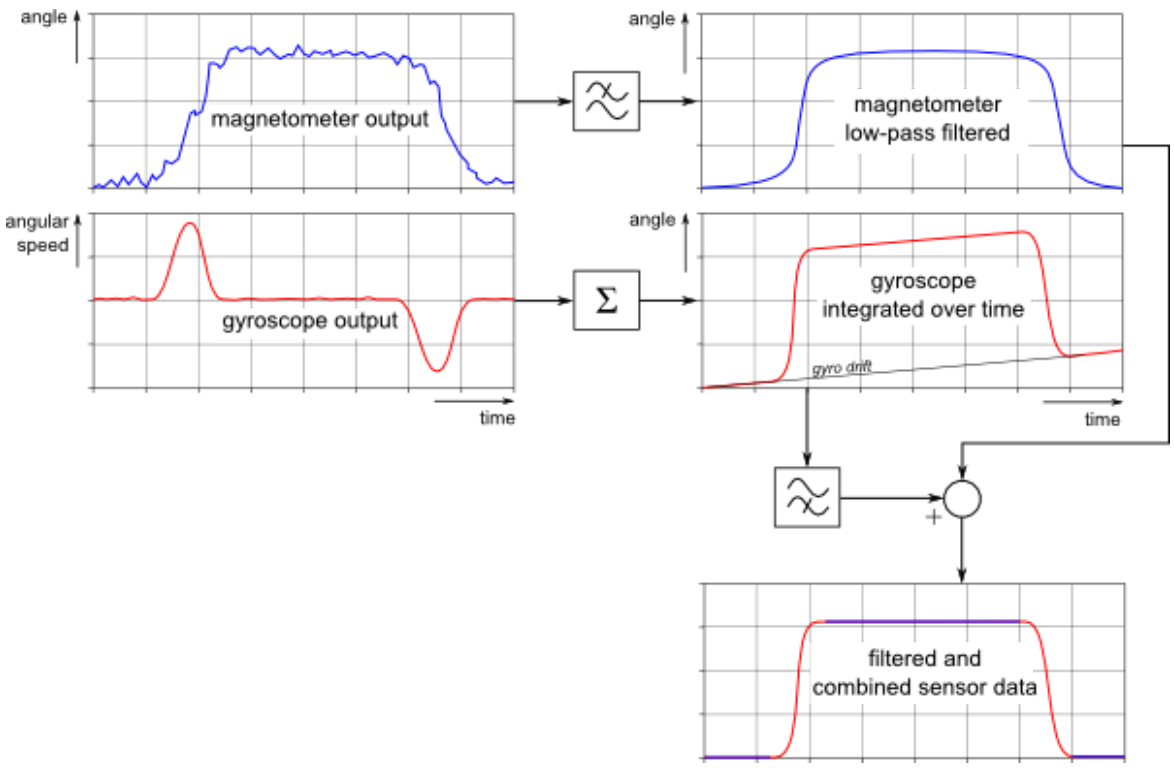

Figure 17. Intermediate filtered signals to obtain the Rotation Matrix.

The gyro drift in the integrated gyroscope signal. It results from the small irregularities in the original angular speed. Those little deviations add up during the integration and cause an additional undesirable slow rotation of the gyroscope based orientation.

The final result of applying the complementary filter to obtain the rotation matrix needed for obtaining the linear acceleration can be seen in Figure 18, where we can see acceleration, magnetic and rotation data used in the filter and linear acceleration obtained after applying the mapping function.

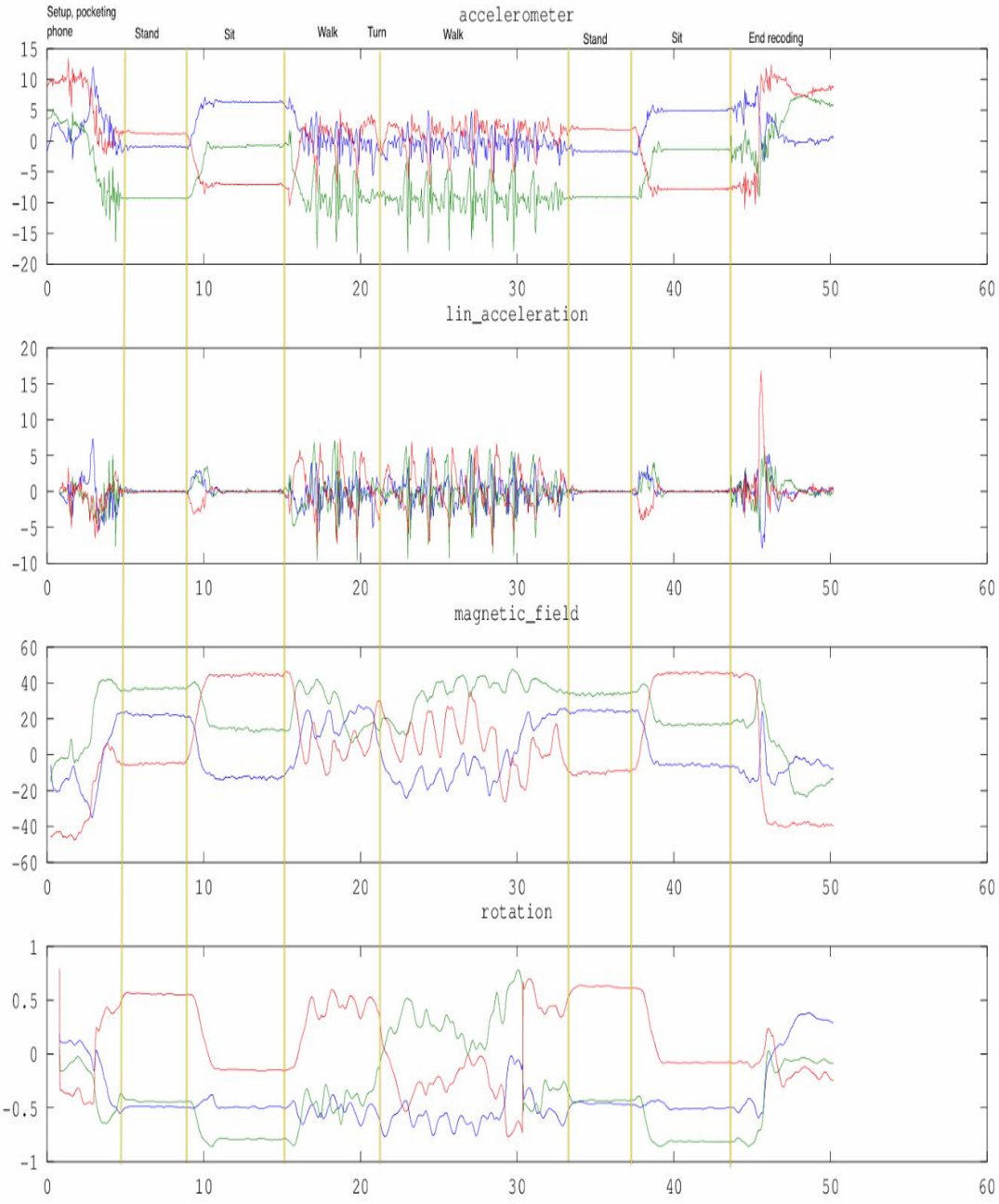

Figure 18. Annotated graph where we can see the accelerometer sensor, linear sensor, magnetic sensor and the rotation sensor over time.

8 - Gait Analysis

8.1 - Mobile Position

Most mobile phones now have a large screen and occupy most space of user's pocket. Therefore, the position of mobile phone is usually stable in the user's pocket and it is reliable to use the x-axis to detect Forward Rotation *FR* and Backward Rotation *BR*.

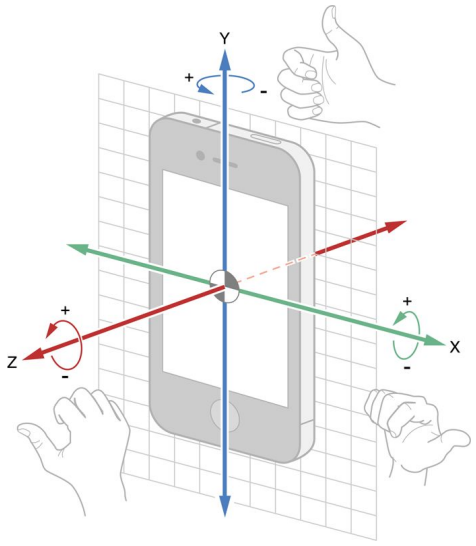

Figure 19. The three axis of the gyroscope.

The rotation movement that occurs inside the pocket allows us to identify an easily recognizable pattern that we can use to create an algorithm to segment the signal Figure 19.

8.2 - Pattern Recognition

From the characteristics that can be used to analyze running or walking, we choose acceleration as the relevant parameter. When thinking about the nature of walking we can observe a unit cycle of walking behavior, showing the relationship between each stage of the walking cycle and the change in vertical and forward acceleration Figure 20.

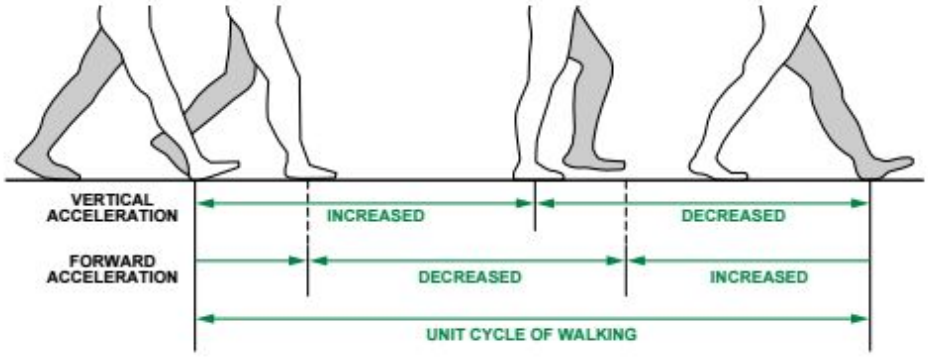

Figure 20. Walking stages and acceleration pattern [15].

From figure above we can deduct that tt least one axis will have relatively large periodic acceleration changes, no matter how the pedometer is worn, so peak detection and a dynamic threshold-decision algorithm for acceleration on all three axes are essential for detecting a unit cycle of walking or running.

8.3 - Distinguish walking and running

When observing the data from the accelerometer we can see that there is some periodicity involved. Most of the energy captured by the acceleration and angular rates associated to human movement is below 15Hz [16].

Human walk presents a particular signature due to the periodic repetition of two main phases: the stance phase, when the foot is in contact with the ground, and the swing phase, when the foot is in the air. As shown in Figure 21 and 22, the analysis in the frequency domain of inertial signals recorded with handheld devices allows capturing the periodicity of accelerometer signals due to the user's walking activity or running activity. In fact, periodicities in the time domain produce peaks in the frequency domain. Observing the presence or absence of the above peaks, for example in the accelerometer signal, it is possible to test the signal periodicity and, subsequently, understand if the inertial force sensed by the IMU is really related to the user's walking, running or to a random motion of the user's hand.

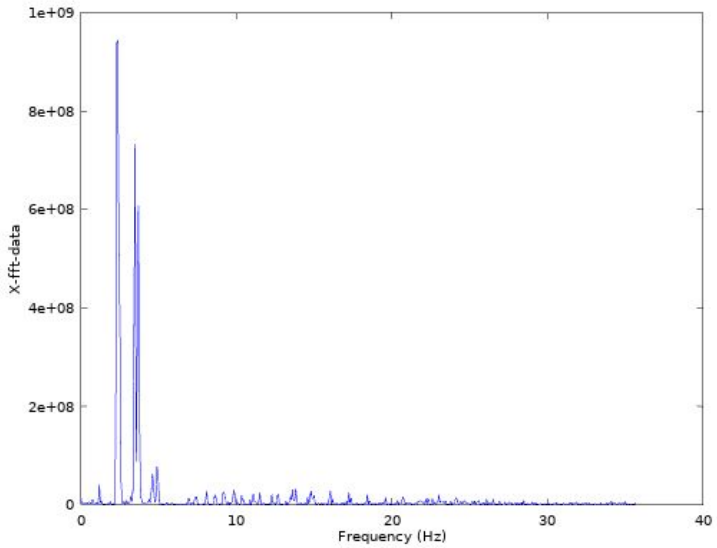

Figure 21. Fourier Transform of walking pattern.

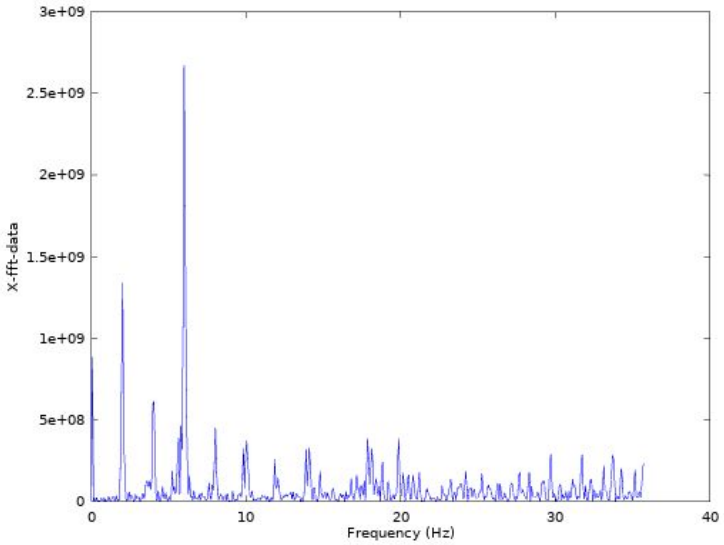

Figure 22. Fourier Transform of running pattern.

The frequency analysis of the accelerometer signal is performed using the Short Time Fourier Transform (STFT) in order to take into account the non-stationary nature of the signal. This technique assumes that a generic non stationary signal can be considered stationary for short periods of time. Then the spectrogram can be obtained by squaring the absolute value of the STFT.

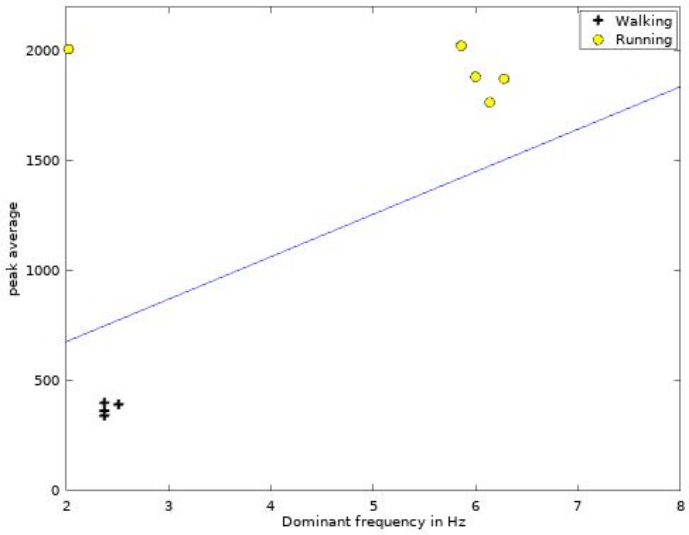

Figure 23. Decision boundary between walking and running events.

We can draw a conclusion, that is walking and running have different peak energies and different dominant frequencies. With that two features we can establish the basis for a decision tree algorithm that could distinguish between running and walking gaits as seen in Figure 23 and Figure 24.

Dominant Frequency	Peak Energy	Event
6.277902	1873.131579	Running
5.859375	2008.083333	Running
5.998884	2022.9600	Running
6.138393	1881.8064	Running
2.51116	399.54506	Walking
2.37165	391.7410	Walking
2,341.769	341.7692	Walking

Figure 24. Machine learning features and decision scheme.

8.4 - Downstairs or Upstairs

8.4.1 - Location of the maximum

Data of a segment is recorded in an array. Each data point has its own index. $IndexS$ denotes the index of the start point. $IndexE$ denotes the index of the end point. $IndexM$ denotes the index of the data point with maximum value. $LengthS$ denotes the length of a segment. $LengthM$ denotes the distance between the start point and the point with the maximum value.

$$LengthM = IndexM - IndexE \quad (8)$$

$$LengthS = IndexE - IndexS \quad (9)$$

$$Location\ Maximum = \frac{LengthM}{LengthS}\ x\ 100 \quad (10)$$

Signal in a segment represents the Backward rotation of user's thigh. In motions of walking on level ground or downstairs, when the user begins to rotate his thigh backward, his foot will soon touch the surface, due to the small distance to between the ground and the foot. Then vibration is induced by heel strike. Therefore, the maximum is located near the start point of a segment. When walking downstairs, the distance between the food and the ground is larger, due to this fact the maximum is not located near the start point.

8.4.2 - Variance of the signal

We calculate the variance of a segment as follows, where Da denotes the average of data values:

$$Da = \frac{1}{n}\sum_{i=1}^{n} Di \quad (11)$$

$$Var = \frac{1}{n}\sum_{i=1}^{n}(Di - Da)^2 \quad (12)$$

The variance of a segment of walking on level ground will be larger than that in a segment of walking up stairs. Using this feature, the motion of walking on level ground can be distinguished from the motion of walking up stairs.

9 - Processing the signal

9.1 - Data segmentation algorithm

The principle of the algorithm is to detect FR of user's thigh and use it to separate signal of each step. The system will continue to monitor the angular velocity of x-axis and detect FR. If there are 15 consecutive data points whose values are all less than -0.5 rad/s, an FR is detected. The start point of a segment is the first data point with positive value after the FR.

Figure 25. Signals of walking

The start point is located by monitoring the first positive point after detecting the 15 consecutive negative points. The end point of a segment is the last peak before the FR of the next step. A peak is located by checking whether there is a data point denoted by $x(n)$ that meets the requirement : $x(n) - x(n-1) > 0 \; and \; x(n+1) - x(n) < 0$, where n denotes the index of the data point.

After setting the start point, if no FR is detected, the end point of the segment will be set to be the 150th data point after the start point. The signals of angular velocity and vertical vibration are segmented according to start points and endpoints as shown in Figure 25. FR is an important element of a walk-like event. If no FR is detected, no segments will be created as illustrated by the signals after the segmentation. Therefore, some irrelevant motions are discarded and the reliability of the system is improved. Using this algorithm, one segment represents a walk-like event. The system can simply count the number of segments, which are considered to be true walk events by a decision tree, and obtain the number of steps of different gait patterns. The mobile phone only monitors one of user's thighs. Therefore, one segment represents 2 steps.

9.2 - Feature extraction

This phase aims at extracting and evaluating meaningful parameters able to univocally characterize each class, therefore enabling the classification process.
Six features are selected to create the decision tree:

- The gyroscope minimum acceleration.
- The maximum vertical acceleration.
- The minimum vertical acceleration.
- The position of the maximum acceleration.
- The variance of the acceleration.

9.3 - Classification

We choose the decision tree as the classification engine since it has a very low computational complexity and can be implemented on a mobile computing unit.

In order to avoid imbalanced distribution of different classes in a decision tree, the amount of each class in a training set should be balanced. If one class is the majority in a training set, the decision tree created by this training set is more likely to classify an unknown instance to that class, for that purpose three subjects with different gait patterns are asked to participate in the training set. Then C4.5 algorithm in Weka is used to identify distinct features and create a decision tree, according to the training set.

While creating the classifier Weka also evaluate the performance of this predictive model. Cross validation is a common method to evaluate the accuracy of classifiers. In Leave One-Out (LOO) cross validation, Weka result is 95.2663%.

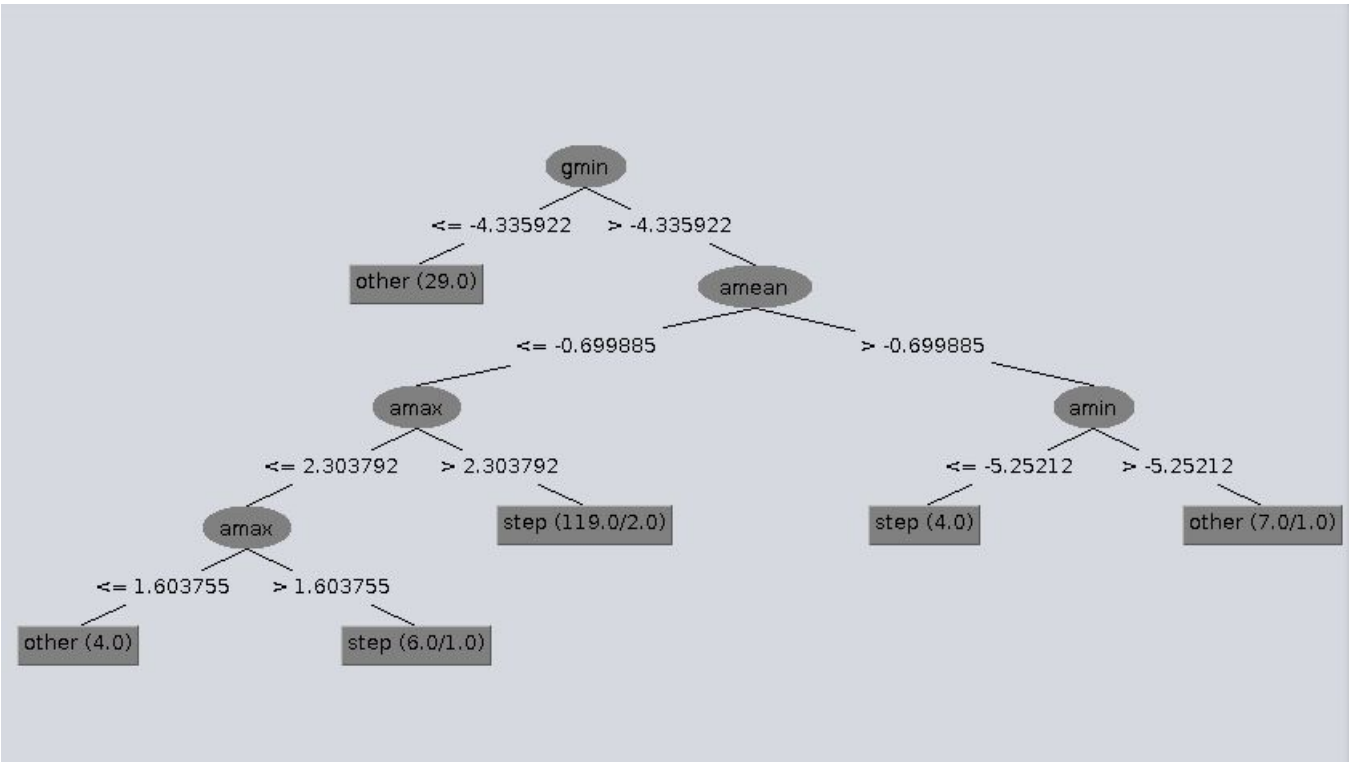

Figure 26. Weka decision tree.

10 - Results

The decision tree based pedometer is tested in a walking experiment and an anti-interference experiment. Two subjects were asked to put an Android device in their pockets using Runtastic Pedometer, in the other pocket they were asked to were the SmartPedometer app developed for this project. Then the efficiency of the proposed system can be compared with that of Runtastic Pedometer.

The subjects were asked to do two experiments. In the walking experience, each subject was asked to take steps on level ground. In the anti-interference experiment, subjects were asked to shake or swing the mobile phone and the Runtastic Pedometer, 10 times at the same time and to see whether the SmartPedometer and Runtastic Pedometer, take those motions as steps.

		SmartPedometer		**Runtastic Pedometer**	
	Total Steps	**Steps detected**	**Accuracy**	**Steps detected**	**Accuracy**
Subject 1	250	230	92%	282	87%
Subject 2	100	86	86%	122	78%
	350	316	89%	404	82.5%

Figure 27. SmartPedometer vs Runtatic Pedometer.

11 - Conclusion

A decision-based pedometer that can count steps is developed. An angular velocity based algorithm is used in this pedometer to segment signals and enable the pedometer to count steps of different gait patterns easily. The decision tree is used to improve the accuracy and reliability of the pedometer. The system has been tested in several experiments with good results. The experiment results show that the proposed pedometer produces much less false step count than a commercial product.

12 - Bibliography

[1] "A single gyroscope method for spatial gait analysis," *http://ieeexplore.ieee.org/iel5/5608545/5625939/05626397.pdf*.

[2] "Accelerometer-based fall detection for smartphones" *http://ieeexplore.ieee.org/iel7/6850263/6860015/06860110.pdf*.

[3] Rokach, L., O.Maimon,(2008) Data mining with decision trees: theory and applications, World Scientific Pub Co Inc, pp vii and 71.

[4] Canalys, "Google's Android becomes the world's leading smart phone platform," *http://www.canalys.com/pr/2011/r2011013.pdf*

[5] Android Developer, http://developer.android.com

[6] Google, Android Developer Challenge, *https://developers.google.com/fit/challenge/get-inspired*

*[7] Susan D. Vincent; Cara L. Sidman (2003). "Determining Measurement Error in Digital Pedometers". Measurement in Physical Education and Exercise Science. **7** (1): 19–24.*

[8] Low-power sensors, https://developer.android.com/about/versions/kitkat.html#44-sensors

[9] Real Time Sensing on Android http://www.cse.buffalo.edu/~lziarek/jtres14.pdf

[10] Jain, K. Statistical pattern recognition: *http://www.ccas.ru/voron/download/books/machlearn/webb02**statistical**.pdf*.

[11] Configuring Android sensors for lower latency *http://www.eetimes.com/document.asp?doc_id=1279399*

[12] Sensor Skew and Offset *http://www.kircherelectronics.com/blog/index.php/11-android/sensors/14-sensor-skew-and-offset-ellipsoidfit*

[13] Android Accelerometer Noise, Offset and Skew *http://www.kircherelectronics.com/blog/index.php/11-android/sensors/7-android-accelerometer*

[14] An accurate and adaptative perdometer, *http://ieeexplore.ieee.org/iel5/5738676/5741058/05741074.pdf*

[15] Full-Featured Pedometer Design Realized with 3-Axis Digital Accelerometer, Neil Zhao
http://www.analog.com/media/en/technical-documentation/technical-articles/pedometer.pdf

[16] Motion Mode Recognition and Step Detection Algorithms for Mobile Phone Users,
http://www.mdpi.com/1424-8220/13/2/1539/pdf

[17] Quaternion IMU Sensors with complementary filter
http://stanford.edu/class/ee267/lectures/lecture10.pdf

13 - Annex

13.1 - Android development

Environment

For the development of the SmartPedometer we have used Android Studio. Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA. Android Studio offers even more features that enhance our productivity when building Android apps, such as:

- A flexible Gradle-based build system
- A fast and feature-rich emulator
- A unified environment where we can develop for all Android devices
- Instant Run to push changes to our running app without building a new APK
- Code templates and GitHub integration to help us build common app features and import sample code
- Extensive testing tools and frameworks

Project Structure

Each project in Android Studio contains one or more modules with source code files and resource files. Types of modules include:

- Android app modules
- Library modules
- Google App Engine modules

By default, Android Studio displays our project files in the Android project view, as shown in Figure 25. This view is organized by modules to provide quick access to our project's key source files.

All the build files are visible at the top level under Gradle Scripts and each app module contains the following folders:

- manifests: Contains the AndroidManifest.xml file.
- java: Contains the Java source code files, including JUnit test code.
- res: Contains all non-code resources, such as XML layouts, UI strings, and bitmap images.

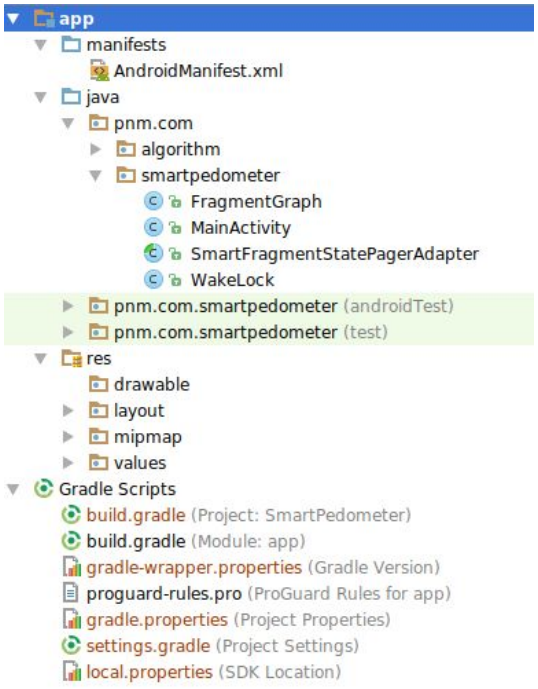

Figure 28. SmartPedometer project files.

Application fundamentals

Android apps are written in the Java programming language. The Android SDK tools compile our code along with any data and resource files into an APK: an *Android package*, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android devices use to install the app.

App Components

App components are the essential building blocks of an Android app. Each component is a different point through which the system can enter our app. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role each one is a unique building block that helps define our app's overall behavior.

The components used in our SmartPedometer are:

- Activities: An activity represents a single screen with a user interface. In SmartPedometer we have MainActivity which is the activity shown when the application is started.

- Services: A service is a component that runs in the background to perform long-running operations. A service does not provide a user interface. In SmartPedometer we use a service to collect data and to process signals in the background away from the UI thread that is in charge to update the user interface.

The Manifest File

Before the Android system can start an app component, the system must know that the component exists by reading the app's AndroidManifest.xml file. Our app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as:

- Identify any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declare the minimum API Level required by the app, based on which APIs the app uses.
- Declare hardware and software features used or required by the app, such as a camera, bluetooth services, or a multi touch screen.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="pnm.com.smartpedometer">

    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Smart Pedometer"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"
            android:screenOrientation="portrait"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service android:name="pnm.com.algorithm.StepManager"/>
    </application>

</manifest>
```

Figure 29. SmartPedometer manifest file.

Activities Lifecycle

Unlike other programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity.

During the life of an activity, the system calls a core set of lifecycle methods in a sequence similar to a step pyramid. That is, each stage of the activity lifecycle is a separate step on the pyramid. As the system creates a new activity instance, each callback method moves the activity state one step toward the top. The top of the pyramid is the point at which the activity is running in the foreground and the user can interact with it.

As the user begins to leave the activity, the system calls other methods that move the activity state back down the pyramid in order to dismantle the activity. In some cases, the activity will move only part way down the pyramid and wait, from which point the activity can move back to the top and resume where the user left off.

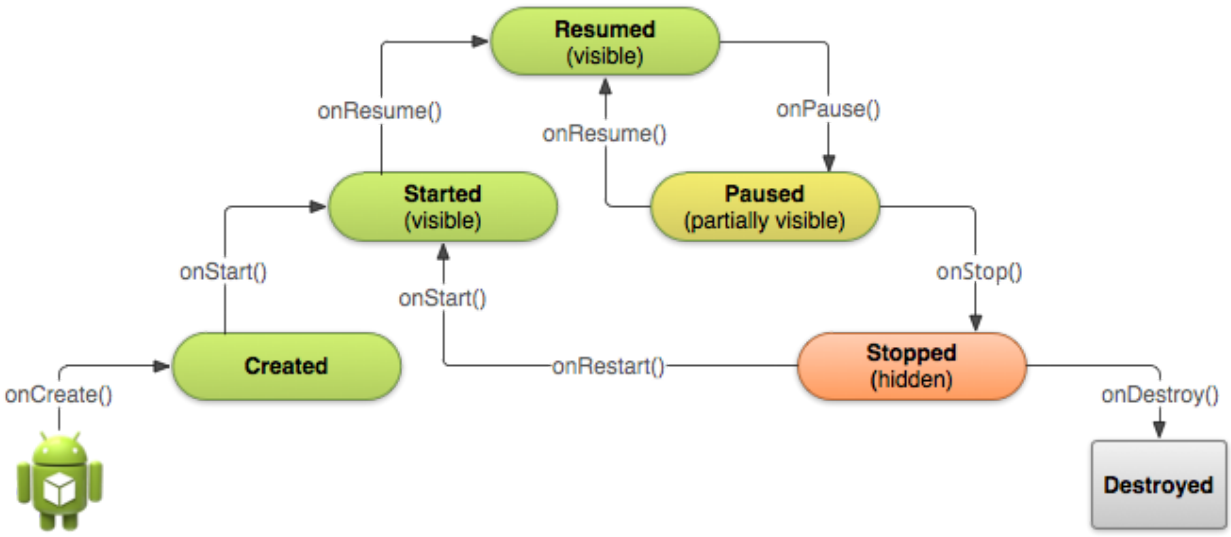

Figure 30. Android activity lifecycle.

All this aspects need to be taken into account when designing the SmartPedometer, that's why we have carefully paid attention to this aspects as can be seen in Figure 30. We can sum up some aspects as follows:

- OnCreate: Application start and we need to setup the views and the main structure. Is in this method where we bind ourselves to the StepManager's service to receive information about the steps taken so we can update the steps counter UI.
- OnResume: Right after OnCreate OnResume method is called. Is here where we create WakeLock that prevents the device from falling into battery saving mode.
- OnStop: Before calling OnDestroy method we need to inform the device that we no longer want to prevent it from going to battery saving mode so we get rid of the WakeLock.
- OnDestroy: The application is finishing so we need to tell de StepManager to stop its work.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mMessenger = new Messenger(new IncomingHandler(this));

    mWakeLock = new WakeLock();

    initializeViews();

    logAcceleration = new StringBuilder();

    mPaused = true;

    doBindService();

}

@Override
public void onResume(){
    super.onResume();
    mWakeLock.acquire(this);
}

@Override
public void onStop(){
    super.onStop();
    mWakeLock.release();
    new SaveData().execute(logAcceleration.toString());
}

@Override
public void onDestroy(){
    super.onDestroy();
    Log.d(TAG, "onDestroy");
    doUnbindService();
}
```

Figure 31. SmartPedometer lifecycle.

Receiving data from the Sensors

On Android devices, we can access the devices sensors sensor data using Android API. In our project we are requesting the data using a frequency of 100Hz. Also that the sensor events are registered in onResume() and removed in onPause(). This is done so the sensor are stopped when the user navigates away from the activity to save battery life as sensors use a lot of battery life.

In Figure 30 we can see the portion of code used to inform the Android device that we would like to receive the sensors data periodically, to do so we register a listeners using the Android's API.

```
private void initSensors() {

    if(mSensorsInited){
        return;
    }

    // Register for sensor updates.
    sensorManager.registerListener(this, sensorManager
                    .getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
            SAMPLING_RATE_US);


    // Register for sensor updates.
    sensorManager.registerListener(this, sensorManager
                    .getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD),
            SAMPLING_RATE_US);

    // Register for sensor updates.
    sensorManager.registerListener(this,
            sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE),
            SAMPLING_RATE_US);

    mSensorsInited = true;


}
```

Figure 32. Registering the sensors.

After registering the listeners a callback function will be triggered at the specified sampling rate. Each callback contains information about the sensors, so we have to

```
@Override
public synchronized void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        calculateSensorFrequency();

    }

    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        // Get a local copy of the sensor values
        System.arraycopy(event.values, 0, acceleration, 0,
                event.values.length);


        mOrientationMagneticFilter.setAcceleration(acceleration);
    }


    if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {

        // Get a local copy of the sensor values
        System.arraycopy(event.values, 0, magnetic, 0, event.values.length);

        mOrientationMagneticFilter.setMagnetic(magnetic);

    }

    if (event.sensor.getType() == Sensor.TYPE_GYROSCOPE) {
        // Get a local copy of the sensor values
        System.arraycopy(event.values, 0, rotation, 0, event.values.length);

        mOrientationMagneticFilter.setGyroscope(rotation, System.nanoTime());

        linearAcceleration = mOrientationMagneticFilter.getLinearAcceleration();


    }
}
```

Figure 33. Sensors callback listener.

ThreadPoolExecutor

In SmartPedometer app we need to run the segmentation task in the background and also we need to perform small task that run the classification algorithm while traversing the decision tree. This characteristic needs to be taken into account when designing the app.

To automatically run the task as resources become available we need to provide a managed collection of threads. To do so, we use an instanceof ThreadPoolExecutor, which run a task

from a queue when a thread in its pool becomes free. To run a task all we need to do is add it to its queue.

Once we have the overall class structure, we can start defining the thread pool. To instantiate a ThreadPoolExecutor object, we need the following values:

Initial pool size and maximum pool size:

The initial number of threads to allocate to the pool, and the maximum allowable number. The number of threads we can have in a thread pool depends primarily on the number of cores available for our device. This number is available from the system environment:

```
private static int NUMBER_OF_CORES = Runtime.getRuntime().availableProcessors();
```

Keep alive time and time unit:

The duration that a thread will remain idle before it shuts down. The duration is interpreted by the time unit value, one of the constants defined in TimeUnit.

```
private static final int KEEP_ALIVE_TIME = 1;
private static final TimeUnit KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;
```

A queue of tasks:

The incoming queue from which ThreadPoolExecutor takes Runnable objects. To start code on a thread, a thread pool manager takes a Runnable object from a first-in, first-out queue and attaches it to the thread. We provide this queue object when we create the thread pool, using any queue class that implements the BlockingQueue interface. To match the requirements of our app, we can choose from the available queue implementations:

```
mSegmentsWorkQueue = new LinkedBlockingDeque<>();
mSegmentsTaskWorkQueue = new LinkedBlockingQueue<>();
```

Create a thread of pools:

To create a pool of threads, instantiate a thread pool manager by calling ThreadPoolExecutor(). This creates and manages a constrained group of threads. Because the initial pool size and the maximum pool size are the same, ThreadPoolExecutor creates all of the thread objects when it is instantiated.

```
mSegmentsWorkQueue = new LinkedBlockingDeque<>();

mSegmentsTaskWorkQueue = new LinkedBlockingQueue<>();

mSegmentsThreadPool = new ThreadPoolExecutor(NUMBER_OF_CORES, NUMBER_OF_CORES, KEEP_ALIVE_TIME,
        KEEP_ALIVE_TIME_UNIT, mSegmentsWorkQueue);
```

The final step is to start a task, in our application we set off the Segmentation task that is in charge of doing the feature extraction and the decision tree traversal.

```
void launchTask(){

    Log.d(TAG, "Launch Classification Task");

    SegmentTask task = mSegmentsTaskWorkQueue.poll();

    if(task == null){
        task = new SegmentTask(this);
    }

    task.initTask(yAcceleration, xGyro);

    mSegmentsThreadPool.execute(task.getRunnable());
}
```

13.2 - Filter java implementation

After we initiate the SensorManager and setup the listeners, we begin to collect the data that will pe passed to the filter in order to obtain the mapped acceleration. This data is then passed to segmentation algorithm and the classifier mechanism.

The gyroscope needs additional processing, for that purpose we create a rotation vector which is similar to a quaternion. In this vector we express the rotation interval of the device between the last and the current gyroscope measurement. The rotation speed is multiplied with the time interval which passed since the last measurement, Figure 34.

```
private void getRotationVectorFromGyro(float timeFactor)
{

    omegaMagnitude = (float) Math.sqrt(Math.pow(gyroscope[0], 2)
            + Math.pow(gyroscope[1], 2) + Math.pow(gyroscope[2], 2));

    if (omegaMagnitude > EPSILON)
    {
        gyroscope[0] /= omegaMagnitude;
        gyroscope[1] /= omegaMagnitude;
        gyroscope[2] /= omegaMagnitude;
    }

    thetaOverTwo = omegaMagnitude * timeFactor / 2.0f;
    sinThetaOverTwo = (float) Math.sin(thetaOverTwo);
    cosThetaOverTwo = (float) Math.cos(thetaOverTwo);

    deltaVector[0] = sinThetaOverTwo * gyroscope[0];
    deltaVector[1] = sinThetaOverTwo * gyroscope[1];
    deltaVector[2] = sinThetaOverTwo * gyroscope[2];
    deltaVector[3] = cosThetaOverTwo;
}
```

Figure 34. Rotation vector from gyro

The gyroscope data is not processed until orientation angles from the accelerometer and magnetometer is available. This data is required as the initial orientation for the gyroscope data. Otherwise, our orientation matrix will contain undefined values. The device's current orientation and the calculated gyro rotation vector are transformed into a rotation matrix [17].

The *gyroMatrix* is the total orientation calculated from all hitherto processed gyroscope measurements. The *deltaMatrix* holds the last rotation interval which needs to be applied to the *gyroMatrix* in the next step. This is done by multiplying *gyroMatrix* with *deltaMatrix*. This is equivalent to the Rotation of *gyroMatrix* about *deltaMatrix*.

The rotation vector can be converted into a matrix by calling the conversion function *getRotationMatrixFromVector* from the *SensoManager.* In order to convert orientation angles into a rotation matrix, we use the following conversion function. Our last step is obtain the linear acceleration from the components of the filter that result from applying our matrix, Figure 35.

```
public float[] getLinearAcceleration()
{

    // Fuse the gyroscope and acceleration/magnetic sensor orientations
    // together via complementary filter to produce a new, fused
    // orientation.
    calculateFusedOrientation();

    // values[0]: azimuth, rotation around the Z axis.
    // values[1]: pitch, rotation around the X axis.
    // values[2]: roll, rotation around the Y axis.

    // Find the gravity component of the X-axis
    // = g*-cos(pitch)*sin(roll);
    components[0] = (float) (SensorManager.GRAVITY_EARTH
            * -Math.cos(gyroOrientation[1]) * Math.sin(gyroOrientation[2]));

    // Find the gravity component of the Y-axis
    // = g*-sin(pitch);
    components[1] = (float) (SensorManager.GRAVITY_EARTH * -Math
            .sin(gyroOrientation[1]));

    // Find the gravity component of the Z-axis
    // = g*cos(pitch)*cos(roll);
    components[2] = (float) (SensorManager.GRAVITY_EARTH
            * Math.cos(gyroOrientation[1]) * Math.cos(gyroOrientation[2]));

    // Subtract the gravity component of the signal
    // from the input acceleration signal to get the
    // tilt compensated output.
    linearAcceleration[0] = (this.acceleration[0] - components[0]);
    linearAcceleration[1] = (this.acceleration[1] - components[1]);
    linearAcceleration[2] = (this.acceleration[2] - components[2]);

    return linearAcceleration;
}
```

Figure 35. Linear acceleration components

13.3 - Weka training sets

Weka is a workbench that contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to these functions.

Weka supports several standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection. All of Weka's techniques are predicated on the assumption that the data is available as one flat file or relation, where each data point is described by a fixed number of attributes (normally, numeric or nominal attributes, but some other attribute types are also supported).

The data file used for our project has the form of Figure 36, where we can see the attributes declared for the classification together with its value and the decision that the algorithm should take associated with each data.

```
@relation step

@attribute amax numeric
@attribute amin numeric
@attribute amean numeric
@attribute avar numeric
@attribute amaxl numeric
@attribute gmin numeric
@attribute origin {step, other}


@data


2.5740433, -5.3999443, -1.1141856, 3.797271, 29, -2.9089508, step
2.508687, -5.119519, -1.3775461, 3.8314183, 103, -2.8552094, step
3.3826876, -6.825732, -1.3660405, 5.087219, 30, -3.0726624, step
3.199481, -5.849413, -1.3086591, 5.5527134, 2, -2.95784, step
3.8706484, -9.2880945, -1.3767126, 7.7268624, 3, -2.9395142, step
2.8181267, -9.448296, -1.5500882, 6.0138407, 100, -2.9614868, step
3.1324568, -11.328325, -1.4314953, 7.0257587, 106, -2.9065094, step
2.4250727, -11.826124, -0.75622594, 2.71643, 0, -0.21507263, step
2.8042603, -6.315309, -1.2126207, 4.4201436, 32, -2.9505005, step
6.991886, -4.092238, -0.4069272, 2.243371, 6, -6.509384, other
4.215567, -7.8840895, -1.6194341, 7.9675922, 67, -2.6255188, other
3.380331, -3.020607, -0.13239866, 1.8490987, 48, -7.1129303, other
2.5460305, -4.0644197, -0.076163396, 1.500162, 98, -12.032806, other
5.212067, -5.5713496, -0.8154704, 3.502958, 135, -2.0268707, other
12.185046, -4.0543985, 1.6049583, 12.713417, 125, -4.3359222, other
```

Figure 36. Weka training set