

PROJECT ATLAS

AN ENCODER APPLICATION FOR APPLE WATCH

GUILLERMO ALCALÁ GAMERO

Trabajo fin de Grado

Supervisado por

**Dr. Pablo Trinidad Martín-Arroyo y Alberto Jesús
Molina Cantero**



Universidad de Sevilla

June 2018

Publicado en June 2018 por

Guillermo Alcalá Gamero

Copyright © MMXVIII

<https://github.com/guillermo-ag-95>

guialcgam@alum.us.es

Yo, D. Guillermo Alcalá Gamero con NIF número 77815829K,

DECLARO

mi autoría del trabajo que se presenta en la memoria de este trabajo fin de grado que tiene por título:

*Project Atlas,
An encoder application for Apple Watch*

Lo cual firmo,

Fdo. D. Guillermo Alcalá Gamero
en la Universidad de Sevilla
18/06/2018

To my parents.



ACKNOWLEDGES

I want to thank everyone that helped me in a way or another during this development:

Special thanks to my family. You have always supported me through this long journey. I hope for the day I will be able to give you back everything you gave me.

Thanks to my both supervisors; Pablo, for letting me carry out a project I love as my final degree work, and Alberto, who believe in me more than I did.

Thanks to my uncle Fernando, who introduced me to the world of weight lifting, and thanks to everyone from the morning crew in Germano Sport, who makes me waking up at 5 AM I bit easier.

SUMMARY

One of the reasons that led me to choose this degree was the will to solve problems by myself. I may find something wrong, or just something I thought that could be improved and I didn't want to wait for anybody to solve it.

During these past years I start lifting weights. I love it and when I love something I want to learn more about it. During this learning I found that some training systems, like the Conjugate System, use the bar speed as a variable during the execution of a lift. However, I didn't have a way to measure it. The available hardware in the market is heavy and expensive. I had the idea that this could be achieved using the built-in accelerometer of a much lighter and cheaper wearable. With this idea, I started this project: An Apple Watch application to measure the speed on a weight lifting lift.

During the development I had to learn how to deal with sensors, learn complex algorithms to filter data, ditch part of the work because the results were not what I expected, and design a real application to solve a problem I found in my everyday life.

CONTENTS

I	Introduction	1
1	Context	3
1.1	Strength training in our society	4
1.2	The use of technology in our everyday basis	4
1.3	The fitness industry and a business opportunity	5
2	Objectives	7
2.1	The problem to solve	8
2.2	The project itself	9
2.3	List of objectives	10
II	Project organization	11
3	Methodology	13
3.1	Project's organizational structure	14
3.2	Development methodology	14
3.2.1	FDD Process #1: Develop an Overall Model	14
3.2.2	FDD Process #2: Build a Features List	15
3.2.3	FDD Process #3: Plan by Feature	17
3.2.4	A word about FDD	17

4	Planning	19
4.1	Project temporal summary	20
4.2	Initial Planning	20
4.3	Project timing report	21
5	Costs	23
5.1	Project costs summary	24
5.2	Personnel costs	24
5.3	Material costs	24
5.4	Indirect costs	25
III	Project development	27
6	Iteration #1: Booting	29
6.1	Feature List	30
6.2	Architectonic Design	30
7	Iteration #2: UI Design Patterns	31
7.1	Coded views vs Storyboards	32
8	Iteration #3: Understanding the accelerometer	37
8.1	You get acceleration, not velocity	38
8.2	Raw data bias	38
8.3	Raw data vs User data	39
8.4	From acceleration to velocity	41
8.5	Velocity differs from zero with no movement	42
8.6	Threshold filter	43

- 8.7 From acceleration to velocity via Integration 44
- 9 Iteration #4: Working with dependencies 47**
 - 9.1 Cocoapods 48
- 10 Iteration #5: Understanding the data 53**
 - 10.1 Using charts to display both acceleration and velocity 54
 - 10.2 Adding Gravity chart 54
 - 10.3 Real time charts 55
 - 10.4 Dealing with library bugs 56
- 11 Iteration #6: Filter the data 59**
 - 11.1 Brainstorming of filters 60
 - 11.2 Research and study of the Kalman Filter 61
 - 11.2.1 The g-h Filter 61
 - 11.2.2 Formal terminology 67
 - 11.2.3 Discrete Bayes Filter 68
 - 11.2.4 Gaussian Probabilities 75
 - 11.2.5 One Dimensional Kalman Filters 77
 - 11.2.6 Multivariate Gaussians 83
 - 11.2.7 Multivariate Kalman Filters 88
 - 11.3 Implementing the filters 96
 - 11.4 Unit tests 96
 - 11.5 The Floating Point Problem 98
 - 11.6 Including external libraries in Unit Test files 99
- 12 Iteration #7: Parallel programming and queues 101**

12.1	iOS Concurrency	102
12.2	Concurrent code	102
12.3	Performance improvements	104
13	Iteration #8: Implementation of the filter in the app	107
13.1	From a 6x6 to a 2x2 matrix	108
13.2	Kalman filter initialization	108
13.3	Filter results	109
14	Iteration #9: Data treatment	111
14.1	You get force, not acceleration	112
14.2	The problem with Accelerometers	116
14.3	Using vector projections	117
14.4	Fixing the drift	118
14.5	Finding patterns	120
14.6	Displaying results in Speed Gauge	122
15	Iteration #10: Atlas	125
15.1	From iOS to watchOS	126
15.2	Including Healthkit	128
15.3	Turning background to foreground	129
15.4	The result	129
IV	Closing stage	131
16	User guide	133
16.1	Problem to solve and constraints	134

16.2 Instructions 134

17 Summing-up 135

17.1 Post-mortem report 136

 17.1.1 Things gone right 136

 17.1.2 Things gone wrong 136

 17.1.3 Discussion 136

17.2 Future work 137

Referencias bibliográficas 137

LIST OF FIGURES

7.1	An example of a storyboard of an watchOS project	32
7.2	The modified <i>AppDelegate.swift</i> file	33
7.3	Example of coded constraints	34
7.4	Example of unordered constraints	35
8.1	Example of raw acceleration in m/s^2	39
8.2	Bias present in both, X and Y axes when the device is resting on a surface.	40
8.3	Example of the <i>userAcceleration</i> values when the device is on rest. . . .	41
8.4	Example of the <i>userAcceleration</i> bias. The Z-axis acceleration is over zero most of the time.	42
8.5	Example of sharp steps after a movement. As you can see the velocity of the three axes (Red, Green and Blue lines) does not come back to zero.	43
8.6	Threshold filter.	43
8.7	Example of the application after the Threshold filter.	44
9.1	First version of the Pod file	49
9.2	Charts library information	50
9.3	Surge library information	51
10.1	Example of sharp steps after a movement in the gravity values.	55
10.2	Example of the application with the sensor set to 10Hz.	56
11.1	Error bars of Scale A with 70 ± 3 and Scale B with 80 ± 9	62

11.2	Error bars of Scale A with 70 ± 1 and Scale B with 80 ± 9	63
11.3	Example on how the prediction and the measurement differ.	64
11.4	Estimate calculation based on the prediction and measurement.	65
11.5	Representation of the probabilities of every position of the hallway. . . .	69
11.6	Representation of the probabilities of each door.	70
11.7	Representation of the probabilities after the new information.	70
11.8	Representation of the probabilities with a noisy sensor.	71
11.9	Representation of the probabilities where the sensor is more likely to be right than wrong.	71
11.10	Representation of the probabilities where the sensor is more likely to be right than wrong (After normalization).	72
11.11	Representation of a movement with movement uncertainty.	73
11.12	Another representation of a movement with movement uncertainty. . . .	73
11.13	Initial prior and its posterior after new data.	74
11.14	Posterior and new prior after prediction.	74
11.15	New prior and updated posterior.	75
11.16	Graphical representation of the belief as a Gaussian	78
11.17	Graphical representation of the multiplication as a Gaussian	80
11.18	Full Description of the Algorithm.	82
11.19	Graphical representation of the belief as a Gaussian.	85
11.20	Graphical representation of the belief as a Gaussian.	86
11.21	Graphical representation of the belief as a Gaussian.	86
11.22	Graphical representation of the data at time $t=1, 2$ and 3 seconds.	87
11.23	g-h filter unit tests.	98
11.24	Changes in Podfile.	100

- 12.1 Declaration of the new queue. 103
- 12.2 The closure is set to run on the new queue. 104
- 12.3 Update charts in the main queue. 104
- 12.4 Non-concurrent application performance after a minute running. 105
- 12.5 Concurrent Application performance after a minute running. 105

- 13.1 Result after the implementation of the Kalman Filter. 110

- 14.1 A representation of an object with no gravitational field. 112
- 14.2 A representation of an object moving with an acceleration of 1G. 113
- 14.3 A representation of an object under the effect of the gravity. 114
- 14.4 A representation of an object under the effect of the gravity where the force affects two axes. 115
- 14.5 A representation of the rotation of the accelerometer [13]. 116
- 14.6 Example where, after a bottom-up movement, we get the vertical velocity close to zero. 119
- 14.7 Example where, after a press, we get the vertical velocity far from zero. 119
- 14.8 Figure of the table view in the app. 123

- 15.1 Project organization in an Apple Watch app. 127
- 15.2 Code that implements how I start a workout in my app. 128
- 15.3 Code that implements how to make the audio work in the background. 129
- 15.4 Final watchOS application. 130

LIST OF TABLES

3.1	Feature planning table	18
4.1	Planning and timing summary table	20
4.2	Initial timing planning	20
4.3	Real timing planning	21
5.1	Costs summary table	24
11.1	Multiple iterations with an initial weight of 70.2 Kg, a gain rate of 1 Kg per day and a scale factor of 4/10	66
11.2	Multiple iterations with an initial weight of 70.2 Kg, an initial gain rate of 1kg per day, a measurement scale factor of 4/10 and a gain scale factor of 1/3	67
11.3	Predict step equations	88
11.4	Update step equations	89

PARTE I

INTRODUCTION

CONTEXT

Personally, I look at weight training for what it is: mathematics, biomechanics, and physics.

*Louie Simmons,
Powerlifting Coach*

This chapter makes up the idea behind the project. Section §1.1 includes a short description about its historical context, Section §1.2 explains how we use technology today and Section §1.3 explains the business opportunity I have found.

1.1 STRENGTH TRAINING IN OUR SOCIETY

Strength performance has been present in our culture for a very long time. It was present in the Ancient History with the Pankration, one of the Ancient Olympic Games events; it was present in the Modern Era as a circus event with one of the first barbell movements, the Bent Press, and it's present today with a much broader number of events.

As any other field, it has evolved over time. From a rural activity in Basque Country [23] to several world-class sports, as it is today. This process has not been casual and many factors have driven these changes.

It's been a long time since countries use sport performance as an indicator of political success. In the last century, this was done by the Soviet Union. This caused that a lot of studies about physics and mathematics applied to sport performance were done during these years. The purpose of these studies was to get better results at the Olympic Games.

Olympic Games gather many sport so, these studies should cover principles that could be applied to as many sports as possible. One of the most studied principles by Soviets was Power, thus, the relationship of force and velocity.

Power could be applied to such different sports as Javelin Thrown, Triple Jump and, the one where the relationship between strength and the sport itself is the closest, Weightlifting.

Americans took all these training principles, and they applied them to non-Olympic sports like Powerlifting.

1.2 THE USE OF TECHNOLOGY IN OUR EVERYDAY BASIS

Nowadays, we have something that Soviets didn't have back in the 60s and 70s. We have twisted those massive computers into tiny devices that fit our pockets and wrists, and we have learned to use them in ways those Soviets couldn't have imagined.

All those principles and methodologies studied by the Soviets relied on physics and mathematics and now, we have incredible calculators always with us. Everybody can apply them with much ease than before.

1.3 THE FITNESS INDUSTRY AND A BUSINESS OPPORTUNITY

All these studies were done to apply scientific principles to sports in an elite level but the fitness industry focus mainly on sporadic activity called Exercise. This causes that the majority of software that targets any kind of physical activity only covers running, abdominal work and bodyweight exercises. There is no software that takes advantage of these devices to support training based on barbell movements.

In fact, there are some devices that target this kind of training. There are called Encoders. The problem with them is that they are heavier than a wearable and definitely more expensive.

The built-in accelerometer that every wearable has, can be used to satisfy some functionalities of encoders for a fraction of the price.

OBJECTIVES

If you cannot measure it, you cannot improve it.

*Sir William Thomson (1824–1907),
Physician and Mathematician*

This chapter includes the objectives of the project. Section §2.1 exposes the problem to solve, Section §2.2 talks about the project itself and Section §2.3 defines the list of objectives of this project.

2.1 THE PROBLEM TO SOLVE

The basic unit when it comes to training is the Workout. From there, you can scale it to a Microcycle (a set of Workouts, usually, within a week), a Mesocycle (a set of Microcycles, usually, within a month), a Macrocycle (a set of Mesocycles, usually, within several months) [19] or even, a four-years Olympic cycle. We will focus on the workout itself because it is what we can measure.

Every workout is based on a set of variables that the lifter must manage to get better, thus, stronger, faster, bigger... Some of these variables are:

- The volume or total tonnage of the session (the weight per reps per sets).
- The average intensity of the session (the mean percentage of the weight lifted compared to the lifter's one repetition max or 1RM).
- The rest taken between sessions.
- The caloric intake.

One of the similarities between these four is that they are EASILY measurable:

- The volume is based on the number of sets, the number of reps per set and the weight lifted, which appears by the side of every plate (considering a 20kg barbell as the standard).
- The average intensity is a simple percentage based on the 1RM. Although it's true that the 1RM may change from one day to the other (if you are no longer a novice, you cannot lift your record every day), a lifter may take his or her last competition max as a 1RM.
- The rest is based on the work activity, stressors, time slept, etc. The only one that is not so easy to measure is the quality sleep but this won't be the subject of the project.
- Although your caloric needs can change from one day to the other, due to the changes on the Basal Metabolic Rate and the daily activity, you can guess them thanks to your body response to the calories taken.

The purpose of this project is to measure one variable that is not trivial as the previous ones. And this variable is Speed.

Speed is inherent to any strength related activity. For conceptual purposes, think of a sprinter driving into the starting blocks, a football player exploding off the line of a weight lifter squatting a maximal load. While each of these movements are markedly different from one another, they all require explosive strength [18].

Power (which results from explosive strength) can be represented as:

$$Power(P) = Force(F) * Velocity(v) \quad (2.1)$$

As the equation shows, in order to display a high level of P , one must be capable of exerting a high amount of F and V . To do so, we have two options:

- Get stronger and/or
- Get faster

In a Max Effort lift (weights at or above 90% of the 1RM) [25], as we want to keep both power and force production as high as possible (remember that force equals mass times acceleration and we are lifting near maximal weights), velocity will be very small. If bar speed gets to zero, no power is produced and the lift is considered missed in a competition.

In the sub maximal percentage area (everything below 90% of the 1RM) the scenario changes. In a Dynamic Effort lift [25], the mass of the barbell is not maximal so the force production will not be maximal. However, we want to keep the power production as high as possible so we need to keep the bar speed at a certain value. If the bar speed is very high, that means that the mass is not as high as it should be and if it's very low, we may be digging ourselves in the 90% area, or worse, we may not be producing as much power as possible.

So, how could we manage to control the barbell speed during a training session?

2.2 THE PROJECT ITSELF

In order to achieve this, this project will consist of an Apple Watch app that measures the velocity of the device thanks to the built-in accelerometer.

The purpose of this project is, solely, the instant record of the barbell acceleration. In the 1.0 version, the version submitted for this subject, it should record the bar speed flawless. As a personal project, this may evolve in the future and add more functionalities but now, it will only record the data, manage the data, and will show it to the lifter. You can create a message app with stickers and profiles and file sharing, but, if the message exchange fails, it does not matter how many fancy stickers you can add; the app is useless.

2.3 LIST OF OBJECTIVES

Objective 1. Improve my programming skills. This project will be done using the Apple's brand-new language, Swift 4. It's been a long time since I wanted to use this language and I think this is the best opportunity to learn it. The most important part is that I don't just want to learn the language but the Cocoa Framework and libraries. You can define a variable using any language; it's what you can do with it what I want to study.

Objective 2. Develop for a Wearable. This will be the first time I will develop something using a real system. I want to learn how the system works with its environment and what can I do with it.

Objective 3. Improve my UI design skills. The way Apple structure its projects is different to everything I have done. It's a design-first pattern and I would like to take this approach to analyze the pros and cons between what I have learned in college and this style of programming. Although Apple encourage to use the Storyboards as the way to develop the UI, I want to focus on designing UIs with code because I think they are more versatile.

Objective 4. Learn how to organize myself. Every project we have done in college was a group project so, this will be the first time in a long time where I judge my strengths and weaknesses when it comes to project management and coding.

PARTE II

PROJECT ORGANIZATION

METHODOLOGY

If you fail to plan, you are planning to fail!

*Benjamin Franklin (1706–1790),
Founding Father of the United States*

This chapter develops the basics about the methodology used. Section §3.1 explains how I will organize during the development and Section §3.2 explains the methodology used during the development.

3.1 PROJECT'S ORGANIZATIONAL STRUCTURE

As the sole developer of this project, I will be the one who study the technology, develop the app and test the functionalities.

As an amateur lifter, I will be able to test the app in a real environment using different lifters to test the measure under several scenarios, like different techniques and body types.

3.2 DEVELOPMENT METHODOLOGY

The first part of every project is the planning set-up. Due to my lack of experience with this kind of projects, and as my project supervisor recommended me, I will plan my schedule using an agile methodology called Feature-Driven Development (FDD). A FDD project is organized around several processes. The first three initial FDD processes can be considered a Zero Iteration, in Scrum and XP terms [4].

3.2.1 FDD Process #1: Develop an Overall Model

FDD takes a singular approach when it comes to develop a domain model. Although we develop the domain object model from the beginning, this development is an intense, highly iterative and collaborative activity.

The main purpose of this process is that everyone involved in the project gets a good understanding about the project domain, relationships and interactions.

As an individual project, the purpose is to clarify what is going to be developed and why it is going to be developed. In order to achieve this, it's a good idea to set the list of requirements of the project.

Requirement elicitation:

Objective 1. The system must record its speed using the accelerometer system. The speed only refers to the upward movement of the device. Forward and backward movements could be interesting to address any imbalance of the lifter but this feature will be delayed to a post 1.0 release due to the extensive study of the accelerometer system.

Objective 2. The system must determine where the concentric phase starts and ends. This part is crucial to know the speed of a rep. The concentric phase of a lift always starts with the lowest point of the movement and ends with its highest point. However, there are some drawbacks: in a Squat, you can lift your shoulders when you breathe and brace. This is not the way you should do the Valsalva Maneuver but some lifters do this. We will expect a correct brace so, this aspect must not be covered.

Objective 3. The system must determine when a rep starts and when it ends. As we have to determine when the concentric part of the lift starts and ends, we need to show each speed record as a different repetition. That's why we need to count how many repetitions we have done.

Objective 4. The system must show the lifter the speed of the lift in the screen. Results must be presented in a table view, where each cell contains a different speed value, representing the speed from a different repetition. The possibility of showing a graphic to find a sticking point during the rep will be delayed to a post 1.0 release due to the extensive study of the accelerometer system.

Objective 5. The lifter should be able to start and stop the measure. The way to do this is currently uncertain. A button is the simplest way to accomplish this but a lifter with his/her hands full of chalk, smelling salts and focused on breaking a PR won't stop his/her mental preparation to press a button on a wearable. Besides, although a team-mate could press it, the app is intended to be used alone. If you need someone to press the button, it won't be as useful as it should be. Another way could be using an acoustic signal, maybe a shout or similar but, in a busy gym, it may stop the measure because someone else shouts, so this aspect must be examined in a future to establish the system to use.

3.2.2 FDD Process #2: Build a Features List

Next process is about what is going to be addressed during the development. The following tasks are the ones to be performed in order to get the project done. I will set a few milestones which will help me to divide the whole list of features to develop and group them.

Milestone #1: iPhone Accelerometer Gauge app

Feature #1: Study about the accelerometer APIs. The first feature to implement will be considered as a feature zero. It's impossible to start developing anything if you don't have the proper knowledge about how the system works. During this task I will be learning about how the accelerometer actually works. Apple has some pages about the subject. I also will be looking at RayWenderlich site and Hacking With Swift books to find any information I can use.

Feature #2: Develop the view to watch in real time how the Accelerometer works.

Up next, after having an idea of how the accelerometer system works, I will build an app to record accelerometer events and study how they change, thus, what axis should I look for, what axis should I forget about, in what plane I need to work... This app should have two tabs. One showing real time accelerometer values and the other, recording the data for further study (the data recording is not the real time events of the first tab. You change the tab, start recording and then, you study them). Nowadays, I only have an iPhone, so, all measures will be done there until I get an Apple Watch. Measures with the Apple Watch will be more accurate because it will show how the system will be used. The iPhone Accelerometer Gauge app should be ported to Apple Watch in order to get better study data.

Milestone #2: Atlas.app 0.1

Feature #1: Develop an Apple Watch app which reads the data from the Accelerometer system. After the study is done, it's time to build the first version of the app. By this time, it will only record the data. No processing will be done with this data. The purpose of this version is the calibration of the accelerometer and the basic start/stop system (this may be changed if needed). This version could be considered a ported version of the iPhone Acceleration Gauge.

Milestone #3: Atlas.app 0.x

Feature #1: Implement the rep scheme system. The addition of the rep scheme determination, thus, when a rep starts and when it ends. Once the rep scheme is solid, we will jump to the next stage.

Feature #2: Implement the view to show information about every rep. We will show every rep in the app screen with no speed information. In this stage, the app will measure the entire set and will show a table with as many cells as reps.

Feature #3: Implement when the concentric phase starts and ends. Then, we will cut the rep to include only the concentric phase of the lift. Once the system know when a rep starts and when it ends, it's time to place a mark on the concentric phase starting point and ending point.

Feature #4: Implement the calculation of the rep speed. Once the app knows when the concentric phase starts and ends, I will calculate its speed and will show it to the user.

At this point, the app could be considered an Alpha Version because there may be some cases where it fails. In every previous stages, the app should be tested and retested to avoid any failure but bug are meant to be found. Then, the Alpha version will be enhanced to correct any bug and will be classified as a Beta.

Once the teacher gives me the OK to the app, it will be classified as a 1.0 version.

3.2.3 FDD Process #3: Plan by Feature

Following the Feature-Driven Development process, it's time to establish these features in the calendar, add the documentation related work and set a final schedule. In the table below you can see the initial planning for the development.

3.2.4 A word about FDD

This methodology is very useful when the developers have already worked on similar projects. They can estimate the spent time on every task with good accuracy and know what are the most problematic parts that may slow down the development.

As I have never worked on this kind of projects before, I don't know if any of the proposed tasks will take more time than expected.

To avoid this, Pablo recommended me to merge this methodology (FDD) with Lean Software Development (LSD).

LSD is a collection of principles taken from the Toyota Production System applied to the software development. With this philosophy, we will try to avoid future problems by designing experiments that we will use to check the project viability and the Minimum Viable Product (MVP) before we dig into the Apple Watch development.

We will start developing an app for the iPhone to study how the sensor works. Any

Feature planning			
Feature #1:	Study about the accelerometer APIs.	06/11/2017	10/11/2017
Feature #2:	Develop the view to watch in real time how the Accelerometer works.	11/11/2017	15/11/2017
Milestone #1	iPhone Accelerometer Gauge app	06/11/2017	19/11/2017
Feature #3	Develop Apple Watch app which reads the data from the Accelerometer system.	20/11/2017	29/11/2017
Milestone #2	Atlas.app 0.1	20/11/2017	03/12/2017
Feature #4	Implement the rep scheme system.	04/12/2017	24/12/2017
Feature #5	Implement the view to show information about every rep.	12/02/2018	25/02/2018
Feature #6	Implement when the concentric phase starts and ends.	26/02/2018	18/03/2018
Feature #7	Implement when calculation of the rep speed.	19/03/2018	08/04/2018
Milestone #3	Atlas.app Alpha	04/12/2017	08/04/2018
Milestone #4	Atlas.app Beta	09/04/2018	22/04/2018
Milestone #5	Atlas.app 1.0		

Table 3.1: Feature planning table

problem that may appear in the Apple Watch project will be addressed in the iPhone app.

PLANNING

When the facts change I alter my conclusions. What do you do, sir?

John Maynard Keynes (1883-1946),

Economist

This section includes a basic summary of the project planning and its actual execution. Section §4.1 displays the summary of the project planning, Section §4.2 sets the initial planning and Section §4.3 displays the final timing report.

4.1 PROJECT TEMPORAL SUMMARY

Project summary	
Starting date	05/10/2017
Ending date	17/06/2018
Checking periodicity	Once every 3 weeks
Weekly workload	10 hours
Total number of hours expected	300 hours
Total number of hours	275 hours

Table 4.1: Planning and timing summary table

4.2 INITIAL PLANNING

Iterations summary	
Iteration 1	06/11/17 a 19/11/17
Iteration 2	20/11/17 a 03/12/17
Iteration 3	04/12/17 a 24/12/17
Iteration 4	12/02/18 a 25/02/18
Iteration 5	26/02/18 a 18/03/18
Iteration 6	19/03/18 a 08/04/18
Iteration 7	09/04/18 a 22/04/18

Table 4.2: Initial timing planning

The length of the iteration has been chosen following the FDD principles. As FDD recommend, a feature should be developed in two weeks, as a maximum.

There is only an iteration (Iteration 3) that lasts more weeks. There are several reasons why I have chosen to break the rules there:

- It's the first iteration where I will need to develop for the Apple Watch, so I want to spend, at least, two entire weeks with the code, thus, forgetting about the memory. The extra week is when I will be writing about the progress of this iteration, so the development will still be two weeks.

- If this iteration lasts two weeks, there will be a sole week between the end of this iteration and holidays, so I prefer to lengthen the iteration to get the best of that week.
- During these three weeks, I have a very dense deliverable to present for another subject, so, I will have less time to develop the feature.

Besides, there an entire month where I will not be developing at all. That's the exams period where I want to focus on the other subjects. The day I start the Spring Semester, I will continue with the development of the iteration 4.

4.3 PROJECT TIMING REPORT

Iterations summary	
Booting	05/10/17 a 06/11/17
UI Design Patterns	06/11/17 a 16/11/17
Understanding the accelerometer	16/11/17 a 09/01/18
Working with dependencies	03/01/18 a 09/01/18
Understanding the data	09/01/18 a 22/03/18
Filter the data	03/01/17 a 05/04/18
Parallel programming and queues	29/04/18 a 02/05/18
Implementation of the filter in the app	05/04/18 a 10/05/18
Data treatment	10/05/18 a 25/05/18
Atlas	25/05/18 a 28/05/18
Closing stage	28/05/18 a 18/06/18

Table 4.3: Real timing planning

For the table above, I won't use the same iterations that appear in the initial timing report. Planning differs way too much from the real execution. The reasons will be developed in further chapter but a short summary will be delivered down here.

The original iteration 2 resulted in a 6-month development. As the system was working with real time data from a sensor, I discovered many problems and issues I couldn't have imagined. I had to study why that happened and how to solve them, so I couldn't follow the planning.

In order to create a logical project timing report, the seven iterations previously presented have been replaced by the different steps of the development. Each of these steps match a future chapter of the report.

A brief summary about these steps:

- **Booting:** This chapter includes from the project proposal to the start of the study itself.
- **UI Design Patterns:** This chapter includes the decisions taken about the use of Storyboards in my project.
- **Understanding the accelerometer:** This chapter includes my first steps with the problem to solve and with the information retrieved by the device.
- **Working with dependencies:** This chapter includes my first steps with managing the dependencies of my project and how I had to work with external libraries.
- **Understanding the data:** This chapter includes how I manage to study the strengths and flaws of the information I was getting from the sensor.
- **Filter the data:** This chapter includes the extensive study about the filter I chose to improve the data from the sensor.
- **Parallel programming and queues:** This chapter includes how I had to deal with concurrency in my app to improve its performance.
- **Implementation of the filter in the app:** This chapter includes how I chose the values to initialize the filter.
- **Data treatment:** This chapter includes how I finally treat the data to get better results.
- **Atlas:** This chapter includes the development of the Apple Watch app where I apply all the study done in the previous chapters.
- **Closing stage:** This whole part includes a discussion about what went right and wrong during the development and a few possible improvements for future versions of the project.

COSTS

There's no such thing as a free lunch.

*Milton Friedman (1912–2006),
Nobel Prize in Economics*

This chapter includes the costs of the project. Section §5.1 displays the costs summary, Section §5.2 develops the personnel costs, Section §5.3 includes the material costs and Section §5.4 exposes the indirect costs.

5.1 PROJECT COSTS SUMMARY

Project summary	
Personnel costs	5.764 €
Gross salary	4640 €
Social Security contribution (company part)	1.438.4 €
Material costs	230.5 €
Indirect costs	630.89 €
TOTAL	6939.79 €

Table 5.1: Costs summary table

5.2 PERSONNEL COSTS

Personnel costs includes the costs of the people working for the project. During the Design and Testing (D&T) subject we were told that a developer with a C or B level in the subject usually earns 12 or 14 euros per hour (before taxes). If the developer matches an A or A+ level, it could be a bit higher. As this project is done after Design and Testing is passed, I will consider a 16 euros per hour as salary.

I have spent about 290 hours on the project so the Gross Salary is about 4.640 €. This is the gross salary.

Now that we have the gross salary, we must calculate the costs of the Social Security of the company. Following the guidelines of the government website [20], it would be 31.1% (23.6% from Common Contribution Type + 6.7% from the Part Time Unemployment + 0.2% from FOGASA + 0.6% from Professional training). With this percentage, the company would pay 1438.4 €.

The total personnel costs of the project is 6.078.4 €

5.3 MATERIAL COSTS

For this point, we must calculate the depreciation [14] of the devices we have used. I have used 3 different devices, a Mid-2013 MacBook Air, an iPhone 8 and an Apple

Watch Series 3. Following the government guidelines, computer systems are fully depreciated after 6 years, so every device has been depreciated.

- The first device, the MacBook Air cost 1800 € and the project have lasted 9 months, so we will add 225 €.
- The next device, the iPhone 8 cost 800 € and the project have lasted 9 months, so we will add 100 €.
- The last device, the Apple Watch Series 3 was also new. It had a cost of 400 €. However, as I didn't develop the WatchOS app until the last month, I will only acknowledge one month, thus, 5.55 €.

The total material costs are 230.5 €.

For more information about the environment used for the development, see §6.2

5.4 INDIRECT COSTS

For this section, as I cannot value the exact amount of money expended indirectly in the project, I will suppose a 10% of the current cost of the project, thus, 630.89 €.

PARTE III

PROJECT DEVELOPMENT

ITERATION #1: BOOTING

The beginning is the most important part of the work,

Plato (427BC–347BC),

Philosopher

This chapter plans how the project is going to be started and how are the configurations used to do so. Section §6.1 includes the feature list and Section §6.2 includes the architectonic design of the project.

6.1 FEATURE LIST

Following the structure of the features list delivered in the section 3.2.2, the project will start with an initial study of the Accelerometer API and the development of a test app called Speed Gauge (previously named iPhone Accelerometer Gauge). All the details about this iteration will be delivered in future chapters.

6.2 ARCHITECTONIC DESIGN

Development configuration As I started the project with just an iPhone, all the development and study of the behavior of the accelerometer has been done using the iPhone as the physical device. The development environment has been macOS 10.13 High Sierra with Xcode 9 as IDE. The unit tests have been run using the simulator offered by Xcode. As I'm using a sensor as data source, I couldn't use the simulator to study its behavior so, I had to deploy the app in my own device.

Pre-Production configuration As the project is intended to be an app for an Apple Watch, I had to use this device to test the behavior of the real system.

Deployment configuration As the project is an Apple Watch app, this configuration is the same compared to Pre-Production.

ITERATION #2: UI DESIGN PATTERNS

The curious task of economics is to demonstrate to men how little they really know about what they imagine the can design.

*Friedrich A. Hayek (1899-1992),
Nobel Prize in Economics*

This chapter includes the differences between the use of coded views and Storyboards. Section §7.1 explains the pros and cons of User Interfaces based on Storyboard and coded views and what type I have chosen to use in the project.

7.1 CODED VIEWS VS STORYBOARDS

One of the objectives of this project was the desire to learn how to build User Interfaces in mobile devices.

During the degree, we have learned how create web interfaces using technologies like Apache JSP or Bootstrap. This way is full coded, which means, everything you see in a screen has been coded by the programmer.

However, Apple presents two ways of doing interfaces. The former was coded as well. During Objective-C times, if you wanted to create an interface, you have to deal with views, windows and many other parameters to design your application. This way is similar with everything we have already done.

A few years ago Apple presented a new method: Storyboards. As it sounds, **Storyboard** is a **graphic representation of the interface of your app**. If you have a label, an input and a button, you will be able to see these elements as they are going to be displayed in the device.

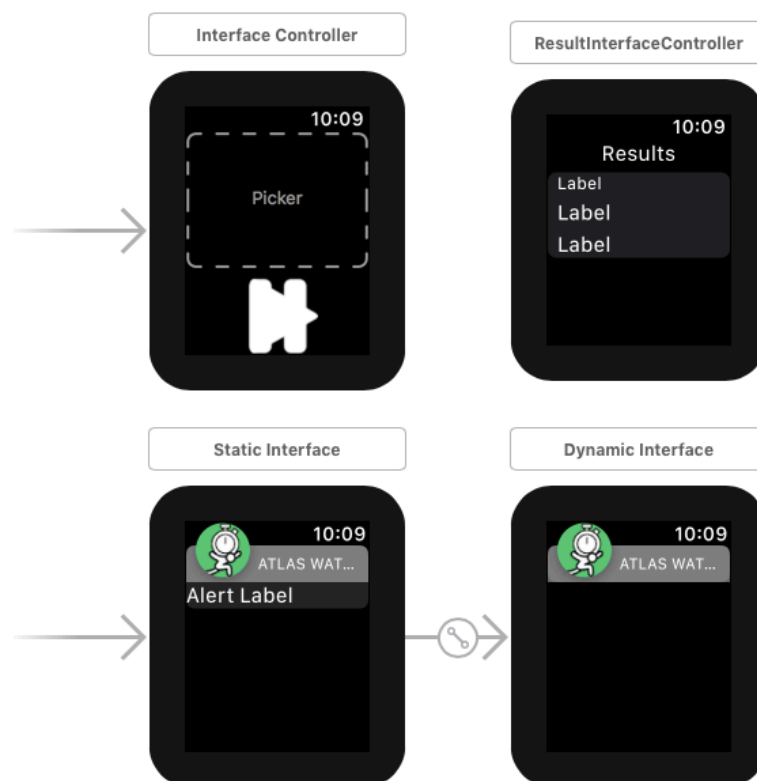
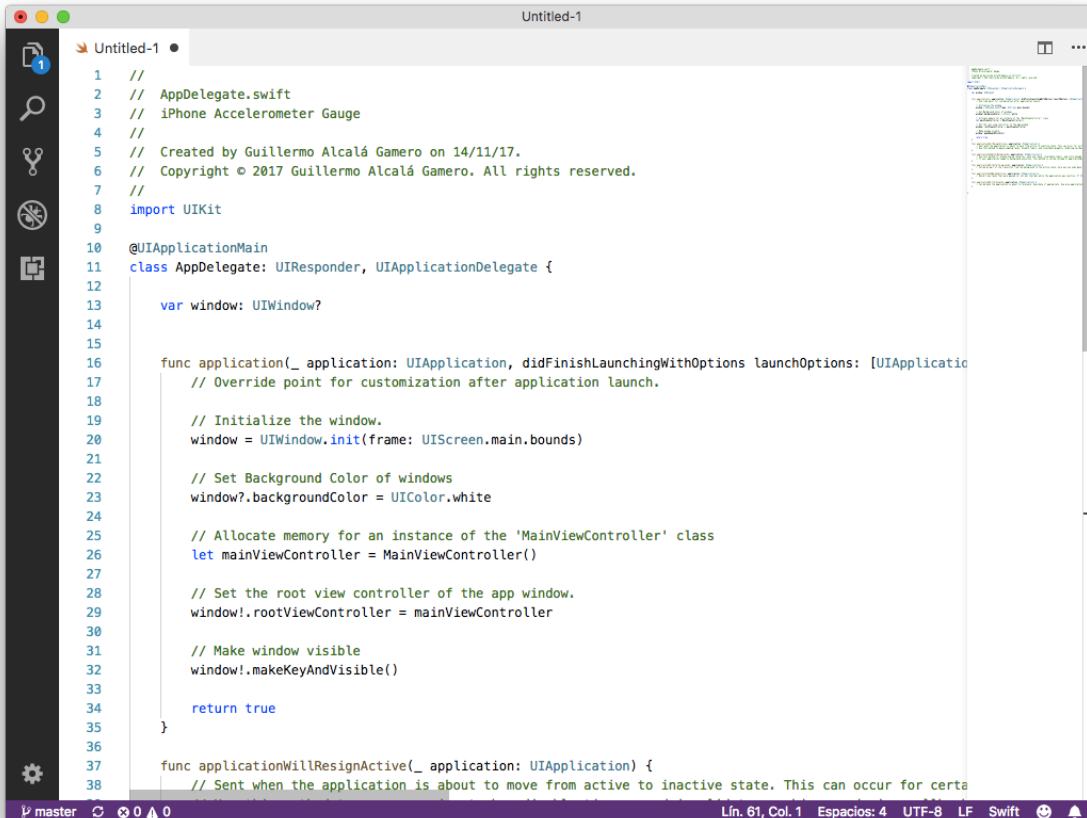


Figure 7.1: An example of a storyboard of an watchOS project

Both methods present its pros and cons. With coded views, you have to deal with how every visual element is displayed in your app. This means you have to manage the main view, windows, background color and the like. One of the first thing I had to tweak in my app was the *AppDelegate.swift* file where I had to define what view was going to be the one rendered when I launch the application.



```

1 //
2 // AppDelegate.swift
3 // iPhone Accelerometer Gauge
4 //
5 // Created by Guillermo Alcalá Gamero on 14/11/17.
6 // Copyright © 2017 Guillermo Alcalá Gamero. All rights reserved.
7 //
8 import UIKit
9
10 @UIApplicationMain
11 class AppDelegate: UIResponder, UIApplicationDelegate {
12
13     var window: UIWindow?
14
15
16     func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
17         // Override point for customization after application launch.
18
19         // Initialize the window.
20         window = UIWindow.init(frame: UIScreen.main.bounds)
21
22         // Set Background Color of windows
23         window?.backgroundColor = UIColor.white
24
25         // Allocate memory for an instance of the 'MainViewController' class
26         let mainViewController = MainViewController()
27
28         // Set the root view controller of the app window.
29         window!.rootViewController = mainViewController
30
31         // Make window visible
32         window!.makeKeyAndVisible()
33
34         return true
35     }
36
37     func applicationWillResignActive(_ application: UIApplication) {
38         // Sent when the application is about to move from active to inactive state. This can occur for certain

```

Figure 7.2: The modified *AppDelegate.swift* file

Every visual element must be declared as a variable. These may sound simple (and it is) but it's very verbose. One of my first versions of the landing view had barely 6 labels, 3 with the data from the sensor and 3 with a description of the data. Every one of these labels had a set of constraints to make sure every element was placed where it should.

Although these constraints made very clear what's going on with every element, I had a 200+ lines file with only the declarations of this basic interface. **No logic was**

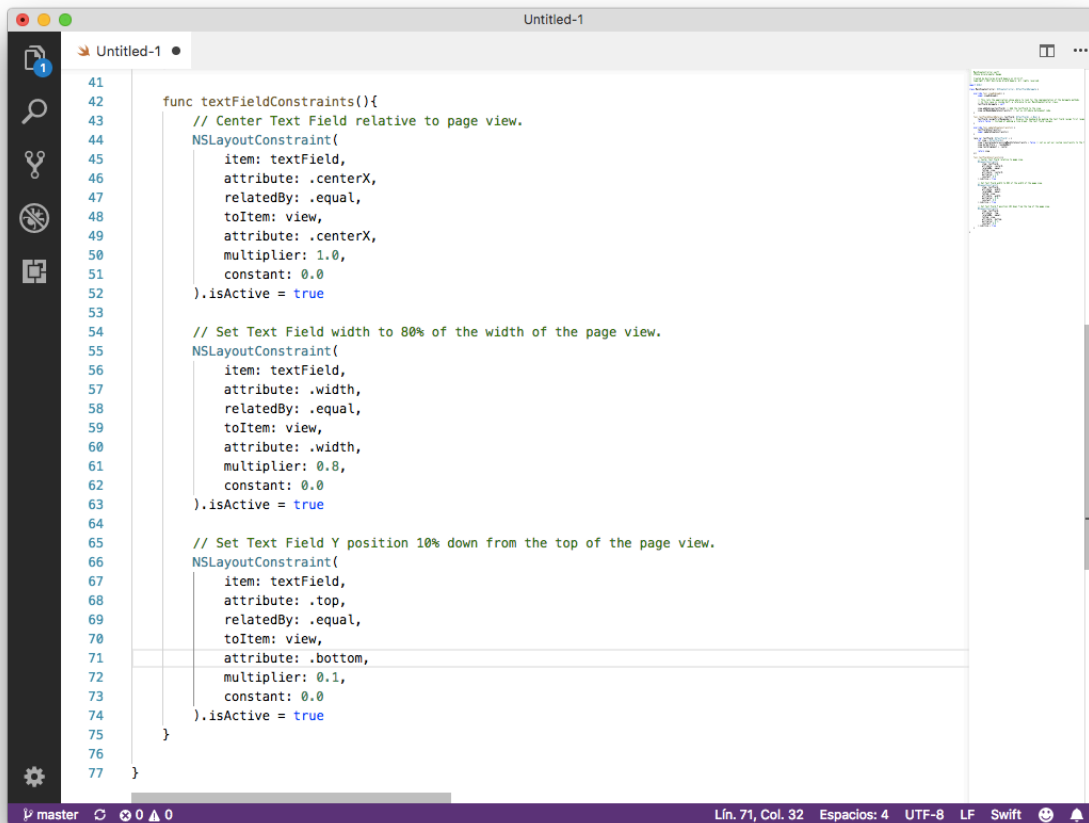


Figure 7.3: Example of coded constraints

already done I could barely organize so much code.

So, the main problem with this method is how verbose it is and the main pro is how you can reuse this code in future projects. If you have implemented a button with special features, you can just copy and paste its code in the new project.

Although I tried to refactor the code to be able to reuse everything I could, the constraints I had to use had to be defined for every element. I could have one generic label but I had to pass 6 or 7 constraints as parameters, so the size of this declaration didn't reduce.

The ability to reuse the interface elements was the main reasons I wanted to give coded views a try. However, I had to throw away the idea to avoid massive file. This is one of the reasons why some people call the MVC pattern in iOS project Massive View Controllers.

Another reason to go for Storyboard was the inability to use Segues. During the time I have studied iOS development, I had always used Segues. I thought I could use them in coded views but I couldn't. Segues are objects from the UIStoryboard class so, **if there are no storyboards, there are no segues**. With a verbose code and the inability to use the only way I knew to manage different views, I was back with the Storyboards. I tried to find the way to code the segues but I found nothing useful, so I didn't want to lose more time with a technology I don't master.

With Storyboard everything was easier... until one point. As I've said, coded constraints are verbose but very clear. You could come back after a month, read the code and see what everything does. With Storyboards, this is not that easy. Every time you set a constraint, it is stored in an XML-like file. It's not easy to read the code. You have to open the Interface Builder, look for the view (not even the element) and see the constraint. With code, you had to be very confident with what you do because you had to set 6 parameters. Here, the program set most of the constraint and stores it inside a tiny view icon, without any order. After bunch of constraints, you couldn't even see the new one.

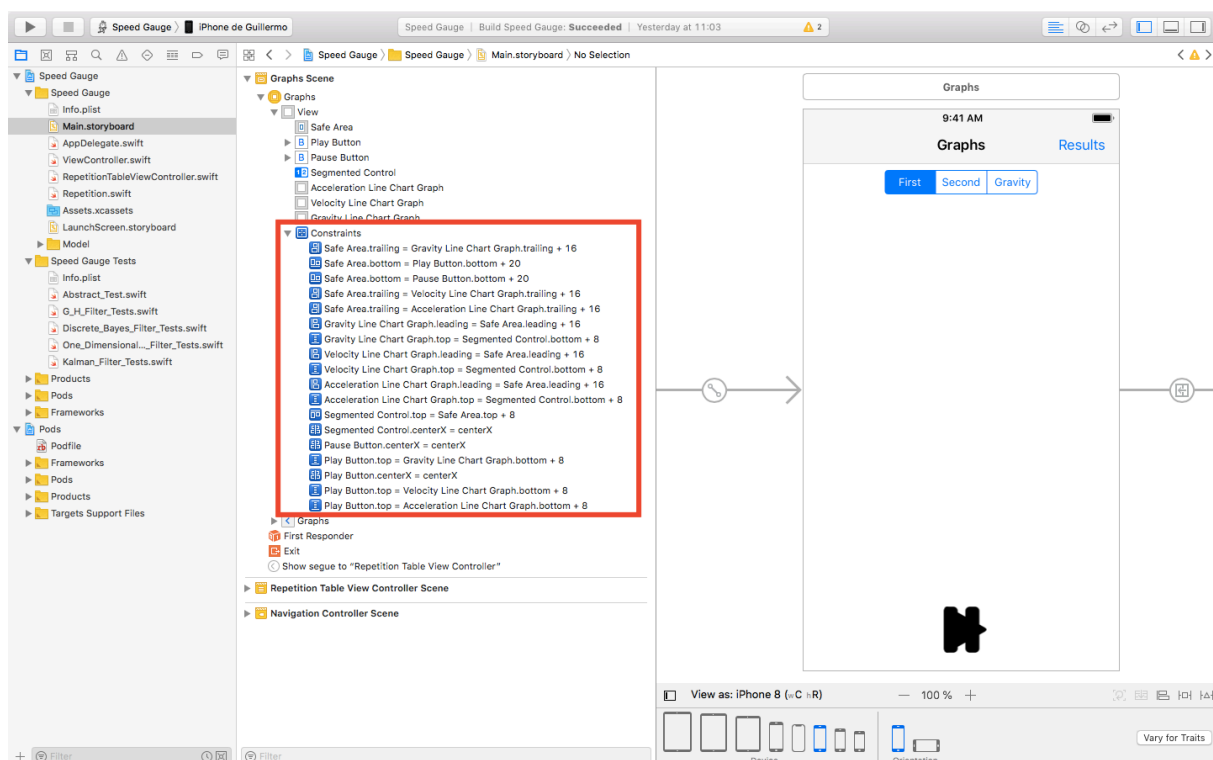


Figure 7.4: Example of unordered constraints

If you want to change a single constraint, it takes 10 seconds to figure out the change in code, but I even have to delete an entire view because I could find the constraint.

Finally, other reason to use Storyboard is because I'm using interface elements from third-party library and this libraries may not describe how to declare their elements solely with code.

ITERATION #3: UNDERSTANDING THE ACCELEROMETER

I think that little by little I'll be able to solve my problems and survive.

Frida Kahlo (1907–1954),

Painter

This chapter includes my first approach with built-in accelerometers, the problems I discovered and how I started to find solutions to these errors. Section §8.1 exposes what data I was going to get from the accelerometer, Section §8.2 explains that there was a bias in the raw data, Section §8.3 compares both raw data and user data, Section §8.4 explains how I was going to get velocities from the accelerations, Section §8.5 exposes the problems with the velocity drift, Section §8.6 includes what filter I used in first place and Section §8.7 explains how I chose a different method to calculate velocities.

8.1 YOU GET ACCELERATION, NOT VELOCITY

It may sound weird but at first I didn't think I have to deal with actual acceleration values. Acceleration is the rate of change of velocity by time.

For me, it was an abstract concept. I can figure out why you may want a sensor that gives you position values, I can figure out why you may want a sensor that gives you velocity values, but acceleration? We don't think in terms of *"I'm increasing my acceleration"*. We think *"I'm increasing my velocity"* or *"I was there and now here"* but not *"I'm keeping my acceleration at zero so, my velocity is stable"*.

I figured out what was the data I was working after developing a very simple app which displayed the data from the sensor. This app was called Speed Gauge and it has been the mock app I have used to study how the accelerometer works.

8.2 RAW DATA BIAS

This app was very simple. I was displaying the data from every axis of the sensor. I was using, what Apple calls, raw acceleration data. Raw acceleration data are the values registered by the sensor, as they are. It includes gravity, so, when you lay the device on a flat surface, you can see how the device is accelerating at 1 G, thus, 9.8 m/s^2 , as seen in Figure §8.1.

This is something I discovered with this version of the app. **The sensor was using Gs so, in order to get velocities, I had to convert the Gs to m/s^2 .** A G is 9.8 m/s^2 so, the conversion is not difficult at all. I only had to be careful with what data I received.

What I also noticed was **a bias in the axis which were not facing the gravity.** If I had the device on a flat surface, Z-Axis was accelerating almost 1G but X-Axis and Y-Axis were also accelerating a little, although there was no movement, as seen in Figure §8.2.

At this point, I had to start reading Apple documentation related to the accelerometer in order to find why this happened. What I found was something better: **the sensor delivered an already-filtered data with no gravity at all; this is the User Acceleration Data.**

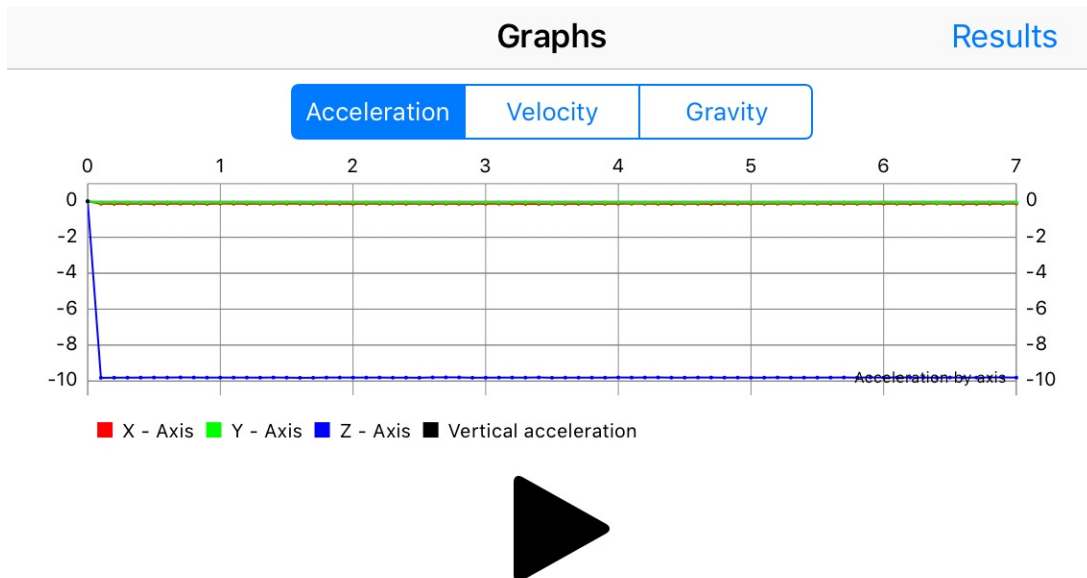


Figure 8.1: Example of raw acceleration in m/s^2

8.3 RAW DATA VS USER DATA

Apple accelerometers offer two kinds of datasets. The one I started using was the Raw data. As I have described, **Raw acceleration means you receive the data, as it is. No filter is done and the gravity acceleration is not removed.**

This acceleration is kind of useless. If I don't know the gravity that affects every axis at every moment, I cannot get proper information from the sensor. This is the reason why Apple delivers another type of acceleration, the *User Acceleration*.

The *User Acceleration* includes more information and this information is more useful. It includes both the acceleration without the effects of gravity and the gravity itself, as separate values.

This means the following: I can know the acceleration of every axis and the gravity that affects every axis.

Once I updated the Speed Gauge, I finally could see how, with the device on a flat surface, all accelerations were quasi-zero and how the gravity in the Z-Axis was -1 G or $-9.8 m/s^2$.

If you read the last paragraph carefully, you can see how I have written "quasi-zero" and there is a reason. This I'm going to write here is the main reason why the original

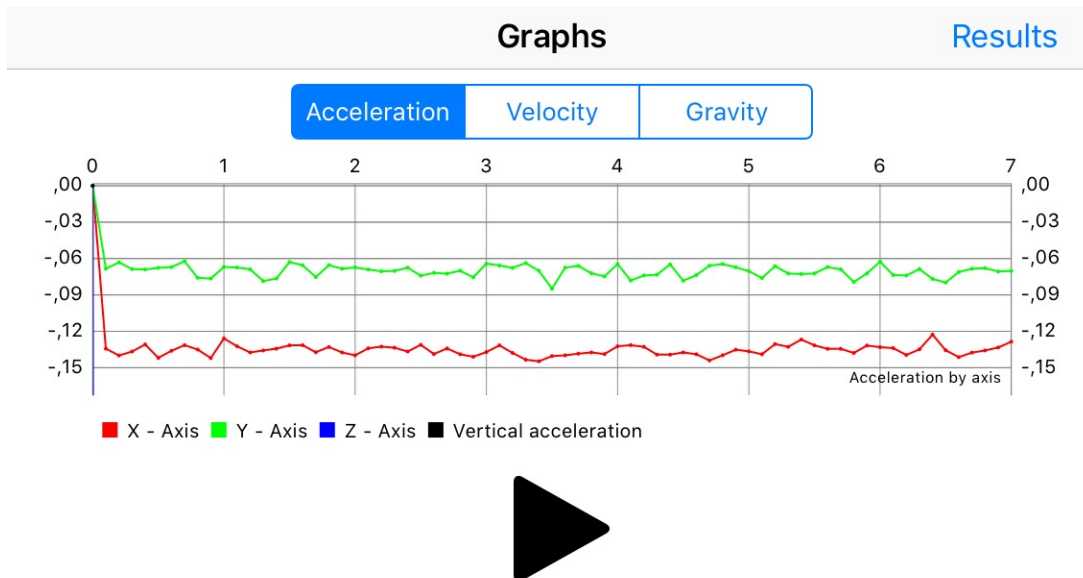


Figure 8.2: Bias present in both, X and Y axes when the device is resting on a surface.

second iteration has taken so long to finish: **All sensors, not only in this problem but in any problem, does NOT give exact measurements.**

This may sound obvious (or not) but I had to discovered it the hard way. **All sensors have a subtle variation in every measurement, they can have a bias and this bias can cause more problems than expected.**

First, **the sensor is inexact**, thus, when you read an acceleration of 1 G, it may not be exactly 1 G, it may be 0.974 G or 1.021 G as seen in Figure §8.3. This will be the foundation of the filters I have studied.

Second, **sensors may have biases**. The bias I discovered here was one related with the gravity. There was a relationship between the bias and the gravity vector. When an axis is under the gravity, the measurement are no longer zero-mean as seen in Figure §8.4. They shift from the real value, as seen in Figure §8.4. And they don't shift as expected, it can shift 0.05 m/s^2 positive at one point and then shift aggressively to -0.05 m/s^2 .

These two features of the sensors caused all the problems I've found doing my project, and they have been the one addresses by the filters I have studied.

But, before we dig ourselves in how I, almost, fixed it, let's see how I found this was a problem at all.

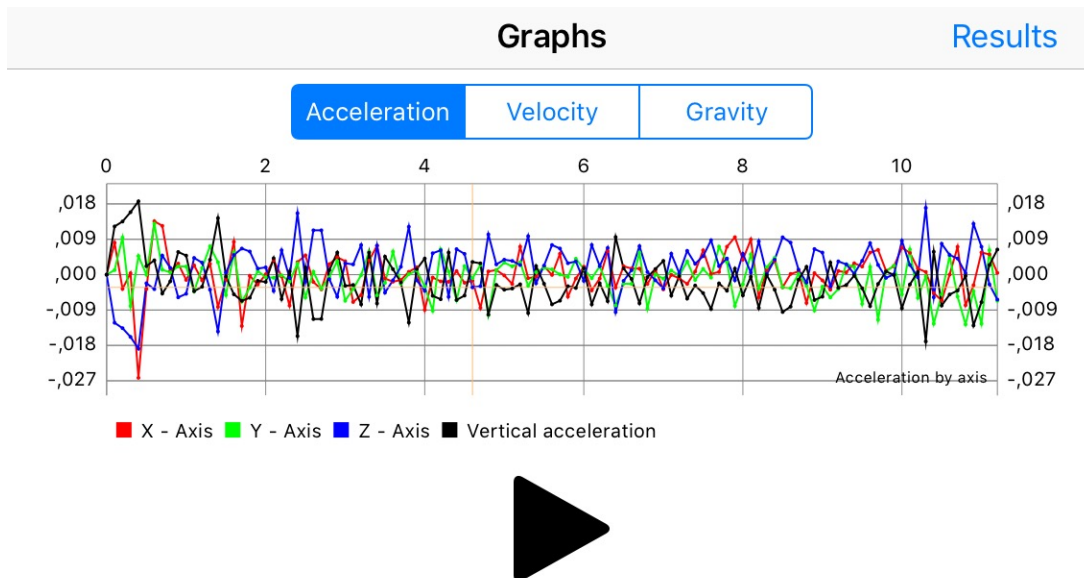


Figure 8.3: Example of the *userAcceleration* values when the device is on rest.

8.4 FROM ACCELERATION TO VELOCITY

The objective of this project is not to record accelerations, it's to record velocities. As the sensor gives acceleration data, I need to convert it to velocities. To do this, I need to understand the relationship between these two magnitudes.

Acceleration is the rate of change of velocity, thus, how velocity changes over time. This looks like a simple arithmetic equation. On the left argument, we have the velocity, the new value, the value based on the measurement of the sensor. On the right argument, we have the acceleration measurement, the current velocity (the velocity of the previous instant may describe it better).

Fortunately we already know the time interval of the measurements. Apple's accelerometer can be set from 0.1s (10Hz) to 0.01s (100Hz). I have chosen to use 100Hz for two reasons: One is that the more measurements I get, the more precise I can be. The other one is that professional encoders work at 1kHz so, 100Hz is already a low frequency.

$$velocity_{k+1} = velocity_k + acceleration_k * time_interval \quad (8.1)$$

As the relationship between velocity and acceleration suggests, you can calculate any of the variables if you know everything else. In a world of perfect sensors, a simple

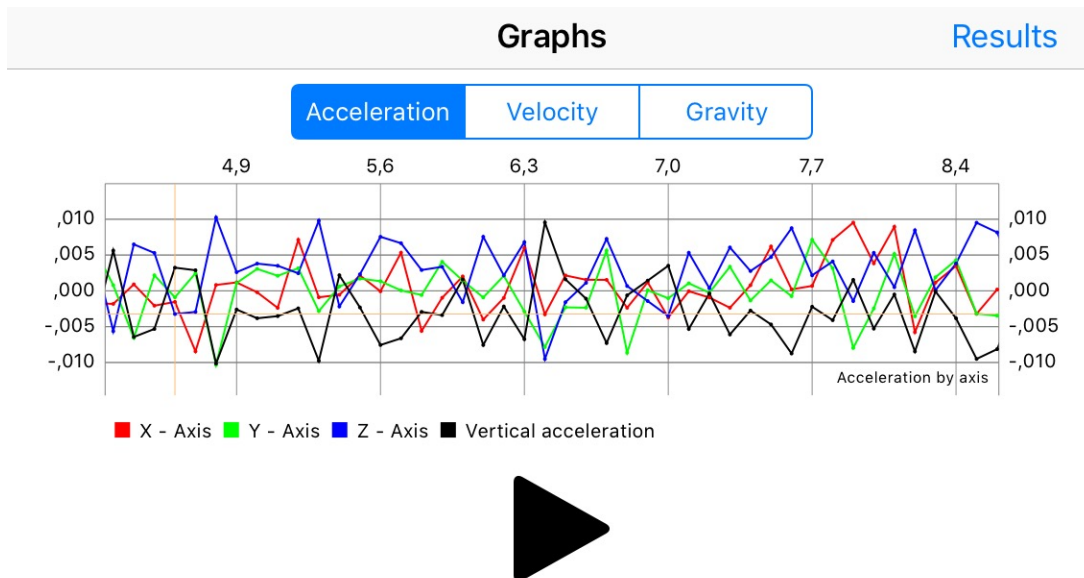


Figure 8.4: Example of the *user Acceleration* bias. The Z-axis acceleration is over zero most of the time.

equation would suffice to get the velocity. However, sensors are not perfect.

8.5 VELOCITY DIFFERS FROM ZERO WITH NO MOVEMENT

After I realized what data I got and what data I was looking for, I updated the app to calculate velocities. I don't have the proper equipment to test if the velocities were working fine. I only know a theoretical relationship that should work in the real world. The way I used to test if that worked was the following: I will move the device, from rest, up and down, side to side. Positive velocities should cancel negative velocities. As the velocity was zero at rest, it should have come back to zero after the movement. It didn't.

It didn't and it didn't by a large value as seen the Figure §8.5. I wasn't worried about the little errors added in every step. I thought positive errors would cancel negatives. The reasons were both the bias of the accelerometer due to gravity and variance of the measurement.

I realized two things: **Sensors are not perfect** and **Theory doesn't always match practice**. At this point, the proper project began.

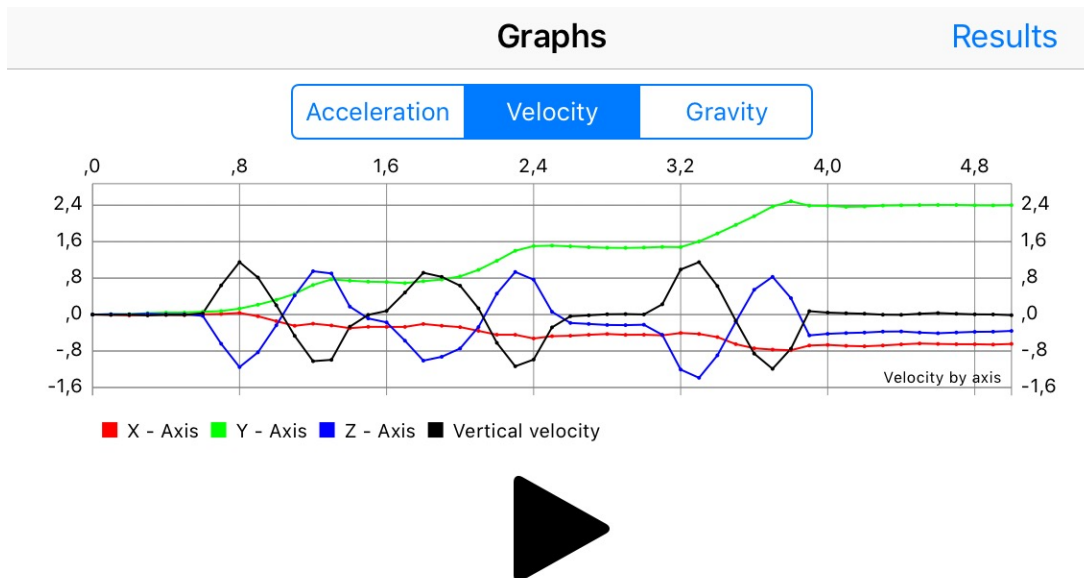


Figure 8.5: Example of sharp steps after a movement. As you can see the velocity of the three axes (Red, Green and Blue lines) does not come back to zero.

8.6 THRESHOLD FILTER

One of my first attempts was to filter the values based on a threshold. This may sound a go-to solution. If there is a little variance in the measurements and this variance in the data is not significant, we could just make everything between \pm the value of the threshold, zero. In my case I chose $\pm 0.05 \text{ m/s}^2$. The code used can be seen in Figure §8.6.

```
// Filter
if abs(newXAcceleration) < 0.05 { newXAcceleration = 0 }
if abs(newYAcceleration) < 0.05 { newYAcceleration = 0 }
if abs(newZAcceleration) < 0.05 { newZAcceleration = 0 }
```

Figure 8.6: Threshold filter.

It didn't work for some reasons, as seen in Figure §8.7.

First, although I could remove the variance from the values close to zero, a value above from 0.05 m/s^2 or below from -0.05 m/s^2 was not affected so, these errors were adding with each new measure.

Second, there was a bias due to the gravity. I couldn't control how the bias was going to appear. I only knew it was due to the gravity, it could shift from positive to negative randomly and that it wasn't greater than $\pm 0.15\text{m/s}^2$. I couldn't also just make the threshold bigger, because I would have lost relevant information. An acceleration of 0.15 is something important when the maximum velocity you are going to get is barely 3 m/s.

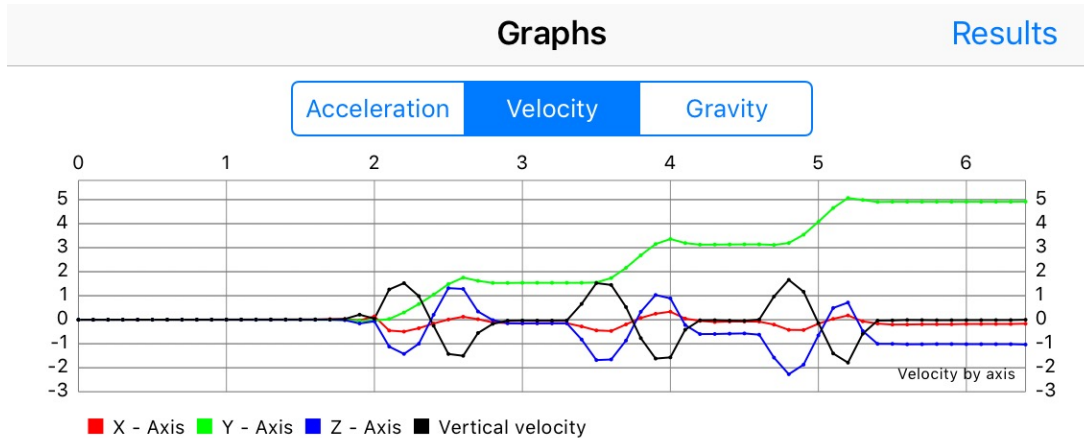


Figure 8.7: Example of the application after the Threshold filter.

After these failures, I only knew (at that point of time) one more thing I could improve: the precision of the conversion. It wouldn't solve my problem but I could improve it. I didn't want to reject the solution because the conversion was poor.

8.7 FROM ACCELERATION TO VELOCITY VIA INTEGRATION

In my last year of high school, they teach me what a change over time means in mathematics. I learn how, when you integrate a function over time, you get the area below the function (or above if negative). You could integrate velocity, get an area and the value of that area would be the position of the object. It's no different with acceleration and velocity.

The method I learned was the one I used to improve the conversion. It's called the

Newton-Cotes formula of first degree, or how they called it, the trapezoidal rule.

I'm not going to explain the trapezoid rule in a degree final project but I wanted to add this section to describe how I was starting to understand what I was working with and how **I could apply something I learned in my senior year in high school in a real system.**

Next two section will be about how I had to manage external libraries in Swift project, how I had to deal with their bugs and how I used them until I realized I need a stronger filter.

After these two, I will describe the study of different filters, the implementations and unit tests.

ITERATION #4: WORKING WITH DEPENDENCIES

The standard library saves programmers from having to reinvent the wheel.

Bjarne Stroustrup (1950),

Computer scientist

This chapter includes my first approach to manage external libraries in an iOS project. Section §9.1 explains how I needed a software to manage external dependencies and what dependencies were used.

9.1 COCOAPODS

At this point I had proper knowledge about what I was working with and what were the main problems of the sensor but my app was barely showing relevant information at all (All previous images were done using the final version of the Speed Gauge application).

I was updating a label every hundredth of a second with the new value, so I had to use an external tool like Octave to plot the measurements and study them. For me, it was time to add graphs to the app.

Apple delivers some libraries to draw graphic content but you have to manage every single aspect about it. In this case, I had to care about size of the chart, how I was going to add every measurement, how the graph was going to update to fill three hundred of measurements per second (a hundred per three axes), how I was going to draw the graph axes and update them.

That's so much work for something so "simple" as plotting a graph. There is no need to reinvent the wheel. Somebody must have faced this exact problem and must have provided a solution for it.

During the degree, we haven't dealt with dependencies in a project. In Design and Testing we used Maven. We had a pom.xml where you would add a new dependency, save the file, click in "Update Maven Project" and that's all.

We have never work with an empty project where you had to figure out how to configure these dependencies.

This section is not a "how difficult is to manage dependencies". I've just had to deal with it for the first time in an environment I don't master, so I think it worth the mention.

When you need to deal with dependencies in an Apple project, you will find three different technologies, one official and two unofficial.

Both unofficial are widely used. They are **CocoaPods** and **Carthage**. There are no such differences between the two, so I just picked CocoaPods because I remembered many tutorials where it was used. Although they are widely used, CocoaPods is more popular.

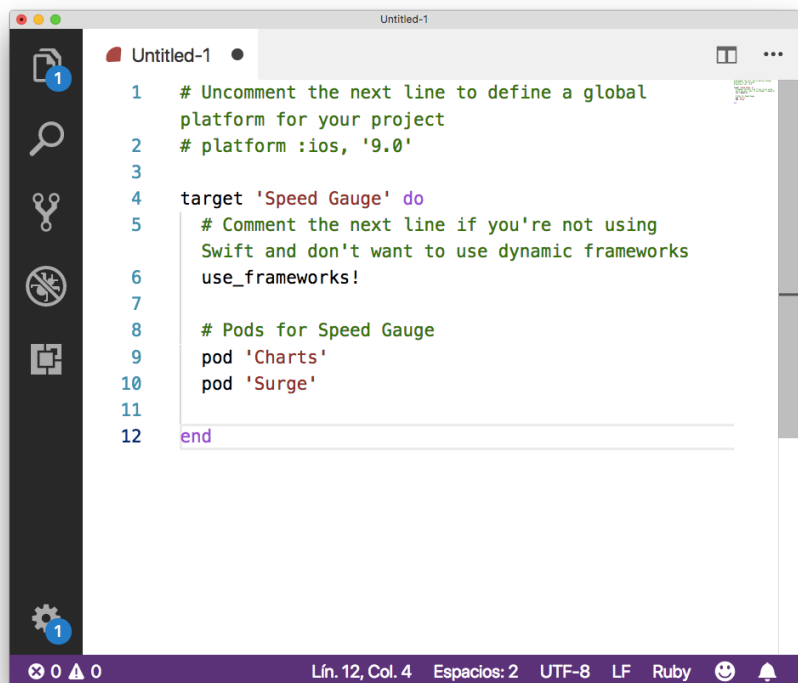
The Official one is the **Swift Package Manager**. It was presented by Apple a few

years ago. Although it the official one, it's not as stable are the unofficial ones. Many packages are not even prepared to use the official solution. In a near future, I'm sure the Swift Package Manager will be the one every developer will use, but for now, CocoaPods is a better option. I just didn't want to deal with a new package I couldn't install.

Something I didn't expect is how simple is to configure CocoaPods. You only have to do a few steps:

1. Install it (Just type "sudo gem install cocoapods" in the terminal).
2. Open the terminal and go to the project directory.
3. Type "pod init" to create a special file called Podfile.

The Podfile is the file where all dependencies will be declared. You can declare the platform and version supported, for example, iOS 11. Then you need to specify the Xcode target to link the dependencies. In my project it was "Speed Gauge". Up next, you must declare the libraries. It's as simple as write "pod '[Library_Name]'". The first version of my podfile can be seen in Figure §9.1.



```
1 # Uncomment the next line to define a global
  platform for your project
2 # platform :ios, '9.0'
3
4 target 'Speed Gauge' do
5   # Comment the next line if you're not using
   Swift and don't want to use dynamic frameworks
6   use_frameworks!
7
8   # Pods for Speed Gauge
9   pod 'Charts'
10  pod 'Surge'
11
12 end
```

Figure 9.1: First version of the Pod file

After saving the file, just type in the terminal "pod install" and all dependencies will be installed in your project. This will generate a new workspace. If you try to use the project in the former workspace it will fail. You need to use the new workspace.

If you want to maintain the same workspace for some reason, you just need to add to your Podfile this line of code, outside of the target, "workspace 'MyWorkspace'".

The first library I needed was a library that let me plot the sensor data. I was lurking GitHub and I found this one, **Charts**. It has different kinds of charts, good support by the community and a proper Readme file to help me set my own charts. A picture of the Readme file can be seen in §9.2.

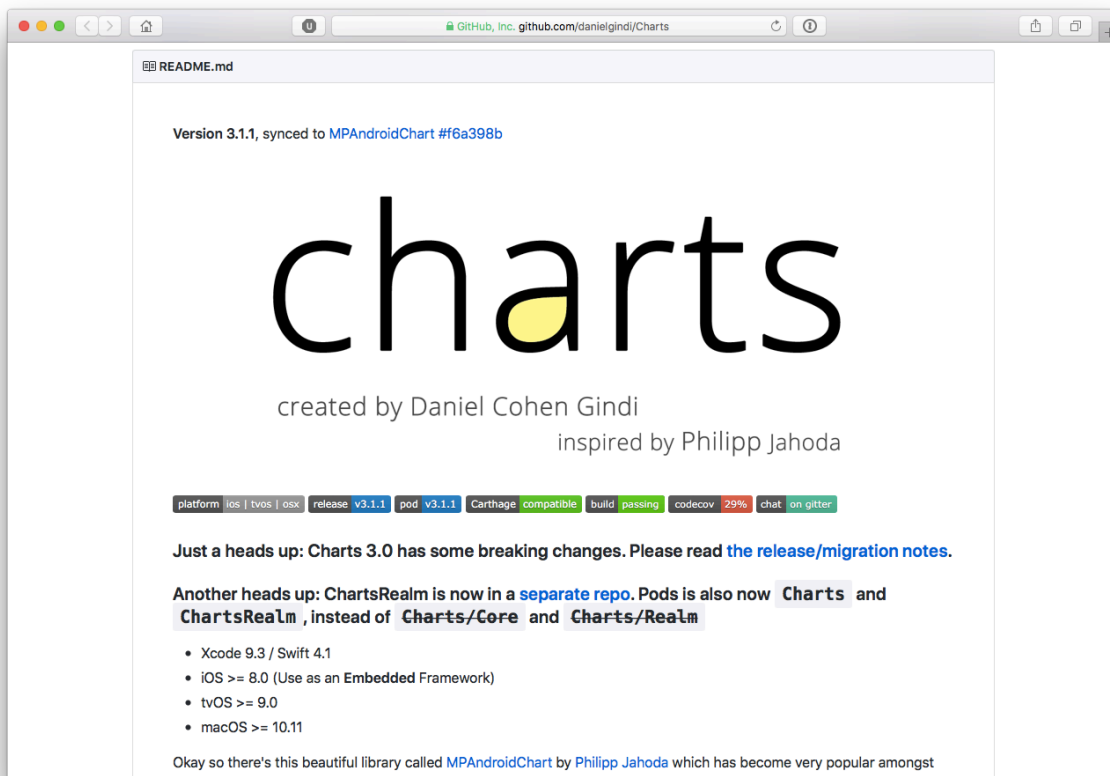


Figure 9.2: Charts library information

The second one was **Surge**. As I will present in future chapters, I will be using a lot of matrices to work with the sensor measurements. Back in the 90's, Apple, IBM and Motorola created AltiVec (a.k.a. Velocity Engine) which provided low-level instructions for mathematics calculations. When Apple made the switch to Intel CPUs,

AltiVec was ported to x86 architecture and it was renamed as Accelerate. This library can be used in a Swift project but it's coded in C, so I would need to merge C and Swift code. Someone did an abstraction layer between the C API and Swift and this layer is the Surge library I am using. Just for curiosity, Accelerate is an evolution of Velocity Engine and acceleration is the derivate of velocity so, as the library is an evolution of Accelerate, the library is named after the derivate of acceleration, the Surge. A picture of the Readme file can be seen in §9.3.

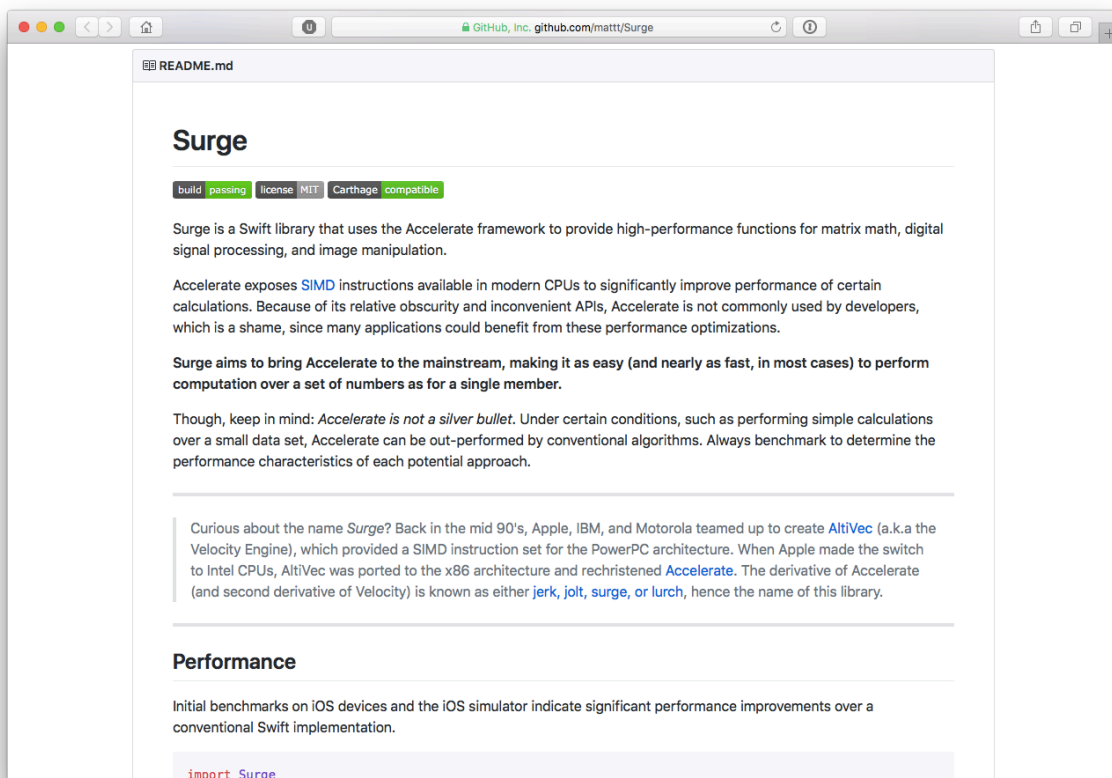


Figure 9.3: Surge library information

ITERATION #5: UNDERSTANDING THE DATA

Luck is what happens when preparation meets opportunity.

*Seneca (4 BC–65 AD),
Philosopher*

This chapter includes how I had to find the way to display the data in my device to be able to study it and find solutions to the problems I was facing. Section §10.1 explains how I used the new library to display both accelerometer and velocity data, Section §10.2 explains how I used the same library to display new relevant information about gravity, Section §10.3 develops how I updated the app to get real time feedback from the sensor and Section §10.4 exposes how I had to deal with some bugs I found in these libraries.

10.1 USING CHARTS TO DISPLAY BOTH ACCELERATION AND VELOCITY

Once I had everything configured, it was time to update the app. This time **I was going to show the sensor data and the calculated velocity in a graph**. This would improve my study of the behavior of the sensor.

After a few tests I could verify these things:

1. The accelerometer was tracking well the movement. If you moved the device, the sensor will react as expected.
2. There was a bias in the sensor that caused the measurement to be a non-zero mean at rest.
3. After a movement over one axis, the velocity of the other axis would not come back to zero.

As I described in previous chapters, there was a relationship between these bias and the gravity so, next thing I did was to study the changes of gravity in the app.

10.2 ADDING GRAVITY CHART

In order to study it, I would use the gravity data retrieve by *UserAcceleration*. **I included a new tab to show how the gravity would change over time** with the movements I made with the device.

Once I added the new chart I discovered something:

1. The amount of bias was proportional to the gravity value. That's way when the device was on a flat surface, the bias could be found in the Z-Axis.
2. During a movement over one axis, for example, the Z-Axis, a decrease in the gravity value in these axes appears. Also, an increase in the gravity value in those axes with a step in their velocity values was discovered. Sharp steps in gravity can be found in Figure §10.1.

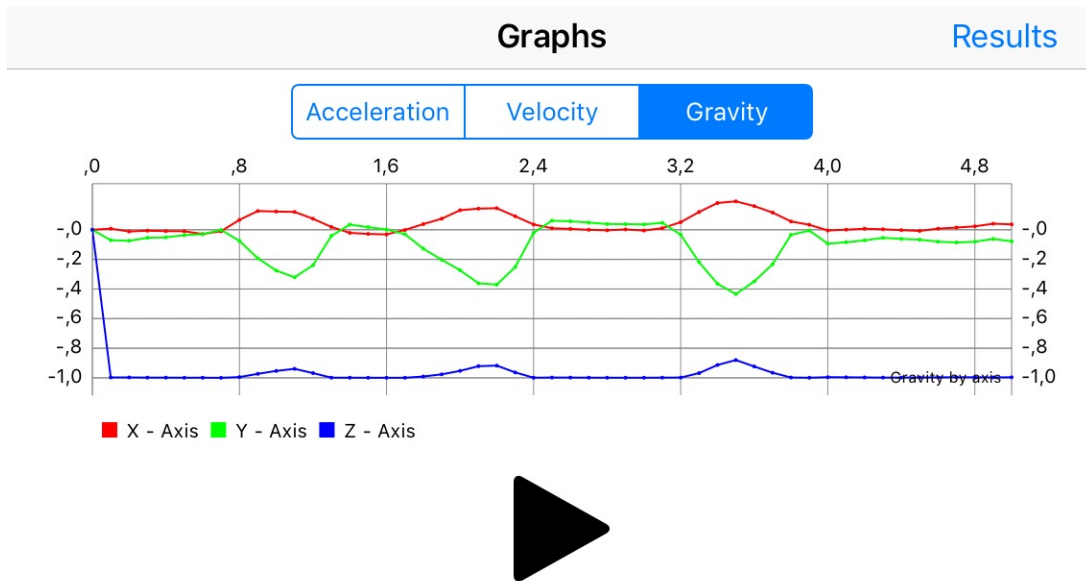


Figure 10.1: Example of sharp steps after a movement in the gravity values.

It was clear that **the gravity was affecting the correct working of the velocity calculation**. In one of the tutorials with Alberto Molina, we talked about why I was using the User Acceleration instead of the Raw one. Then, I try to use the Raw acceleration but it didn't deliver the gravity vector. In other words, in other to get the gravity, I was forced to use *User Acceleration*.

10.3 REAL TIME CHARTS

Up until this point, all my charts were drawn after I get all the data. I would press the play button, perform the movement I wanted with the device and press the pause button. Only then, the graphic was drawn.

After a little research, I found a way to draw the graphs in real time. This would make the study way easier because I could see at which point the bias shifts or if it's happened due to a specific movement.

The update interval time is 0.01 s (100 Hz) so I had update three graphs (acceleration, velocity and gravity) every 100th of a second with 3 new measures in every graph.

After I updated the app, I launched it to try how the CPU handled the workload. It

turns 120% in a matter of a second, so I had to find a way to reduce the workload.

First, I thought about reducing the update frequency, as seen in Figure §10.2, but I would lost a lot of information. It would detect some positive peaks in the acceleration but would ignore the negatives, so the velocities were worse than before.

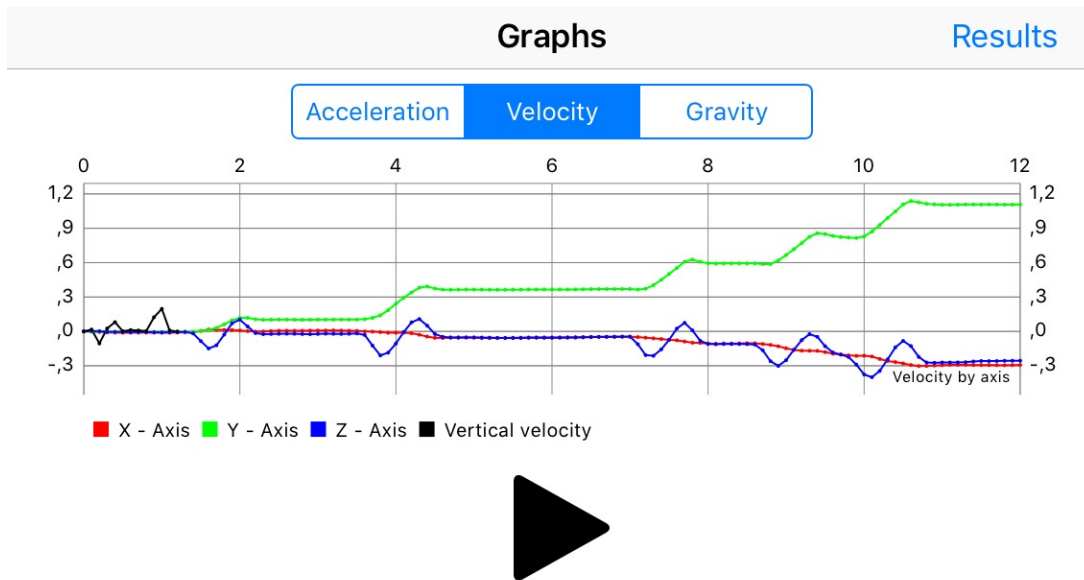


Figure 10.2: Example of the application with the sensor set to 10Hz.

My solution was to update the graph every 10th of a second. **All measurements were going to be calculated but only one out of ten would be drawn.** This reduced the workload but I found one more error.

10.4 DEALING WITH LIBRARY BUGS

Before I took the decision to draw one out of ten measures, I tried to find why this happened. I found that there were some bugs introduced in the last version of Charts.

This bug caused a performance bottleneck. Every time a new value was added to the graph, the minimum and maximum value of the chart were calculated. This calculation was done using a for each loop over the array of measurements so, adding a hundred every second caused an important drop in performance. This was addressed in a future update of the library[11].

Another bug was not a performance issue. I found that the calculation of the min-

imum and maximum value of the chart was not reset with a new chart. This was addressed in the same update as the previous bug[3].

This section is not about how I solved these bugs. Instead, **it's about how I had to learn how third-party libraries have bugs and how I had to read plenty on GitHub issues before I could realize what was happening.** This could be one of the lessons learned from the projects: Every time you use an external library, there are chances that it doesn't work as expected.

ITERATION #6: FILTER THE DATA

Without a filter, a man is just chaos walking.

Patrick Ness (1971),

Novelist

This chapter includes the study I made to find a way to filter and clean the data in order to get better measurements. This includes an extensive explanation on how the chosen filter (Kalman filter) works and what is the reasoning behind its algorithm. Section §11.1 talks about all the research to find the filter I was going to use in the app, Section §11.2 includes the extensive study done about the Kalman Filter, the chosen filter, Section §11.3 indicates how I decided to implement the filters, Section §11.4 explains how I set the Unit tests of my app, Section §11.5 develops the problem I found with Floating point numbers and Section §11.6 includes the changes I had to make in order to use the external libraries in my tests.

At the same time I was improving how the data was displayed in the application, I studied how to improve the data from the sensor. After a few tutorials with Alberto, we realized that the data must be filtered. I had to remove the bias to get reliable information.

11.1 BRAINSTORMING OF FILTERS

At this point I did research to find which filters could help me with this labor. I developed a list of possible solutions:

1. Increase the level of Newton-Cotes formula: This is not a filter. The current Newton-Cotes formula uses the first degree method. I could use a higher degree method to reduce the little errors that accumulate due to the integration.
2. Implements a Moving Average filter.
3. Implements a Kalman filter.
4. Implements a Butterworth Band-Pass filter.
5. Implements a Total-Variation denoising.
6. Implements a Wavelet Thresholding.
7. Implements a Complementary filter.

Before I developed this list, Alberto and I tried other filters like High-Pass filter, Low-Pass filter or turning the signal to a zero-mean but none of this worked.

With the little research I could do about how to improve calculation from acceleration to velocity, I found two sides of the problem. **One side said it was impossible to calculate velocities from acceleration due to integration errors.** This was addressed in Google Conference (Sensor Fusion on Android Devices: A Revolution in Motion Processing)[17].

The other side declared it was possible with a Kalman Filter. Plenty of sites defended the Kalman Filter as a way to calculate velocities from accelerations.

Before I could even show the list to Alberto, he told me it could be addressed with a Kalman Filter. I wouldn't get a perfect measurement but I could get a pretty decent one.

So, the Kalman Filter was my choice to improve the precision of the calculated velocities.

11.2 RESEARCH AND STUDY OF THE KALMAN FILTER

OK, **this section will be with no doubt the largest and most dense of the entire report.** It will include my first approach to the filter and a detailed explanation of it. I will try to make this explanation as easy as possible. Something I have learned about the filter is that it is the most dense mathematics I ever faced if you jump head first into it. After you understand the basics, it's basically common sense with matrices.

I don't want to present a very verbose writing about the filter so, I will explain it as I have learned it, from the bottom to the top, asking simple questions after simple question until you realized you have a basic functional filter. To do so, **I will follow-up the scheme present in the book "Kalman and Bayesian Filters in Python"[9]. I will reuse the examples used in the book for the sake of simplicity.**

11.2.1 The g-h Filter

Imagine you live in a world without scales (the device you stand on to weight yourself). One day, a coworker appears with a brand-new invention, the scale. You weight yourself for the first time and you get a weight of 80Kg. Then, you call another coworker to show him the new device. You weight yourself once against and you get 70 Kg. How can this happen?

Well, that's how sensors works. **They are inaccurate.** That's why filters exists in Physics. If we could get exact measurements every time, there wouldn't need to filter anything.

Is there any way we can improve the result? We could find a better sensor but that's not possible in many cases. The other way is to weight yourself in different scales. Before, you weighted yourself and you get (A) 70 Kg and (B) 80 Kg. With that information we have several options.

1. We could choose to only believe A.
2. We could choose to only believe B.
3. We could choose a number less that A and B.

4. We could choose a number greater than A and B.
5. We could choose a number between A and B.

Choosing the options 3 and 4 seems arbitrary. There is no reason to choose a number greater or lower than the measurements we have. Choosing 1 and 2 seems also arbitrary. Why do you believe one scale over the another? The remaining option is the one we want. **We believe that the real value is somewhere between A and B.**

Our first move would be to set the real value as the mean between the measurements. If we have 70 and 80 Kg, the real weight could be 75. However, **we should ask if we had any more information.** What if one scale is 3 times more accurate than the other? Now, we have a measurement with 70 ± 3 Kg and another with 80 ± 9 Kg.

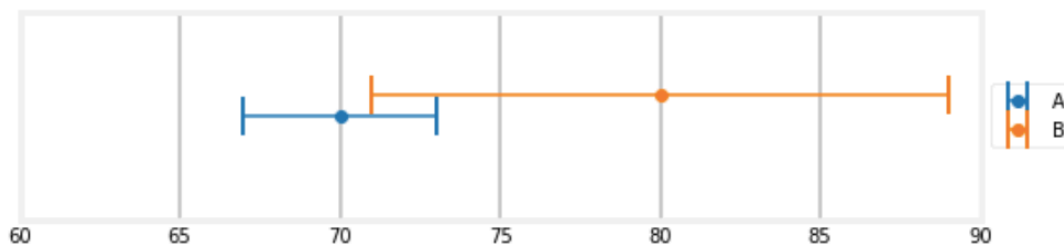


Figure 11.1: Error bars of Scale A with 70 ± 3 and Scale B with 80 ± 9 .

As you can see in the figure §11.1, neither 70 Kg nor 75 Kg are included in the overlap of the error bars. The only possible values are the ones between 71 and 73 Kg. If we only use the measurement from A because it's more accurate than B, we would give an estimate of 70 Kg. If we average A and B, we would get 75 Kg. Neither of those weights are possible given our knowledge of the accuracy of the scales. By including the measurement of B, we would give an estimate between 71 and 73 Kg, the limits of the intersections of the two error bars.

Let's take this to the extreme limits. Assume we know the scale A is accurate to 1Kg and the scale B is accurate to 9 Kg. If we plot the error bars we can see in the figure §11.2 how there is only one possible value left, 71 Kg.

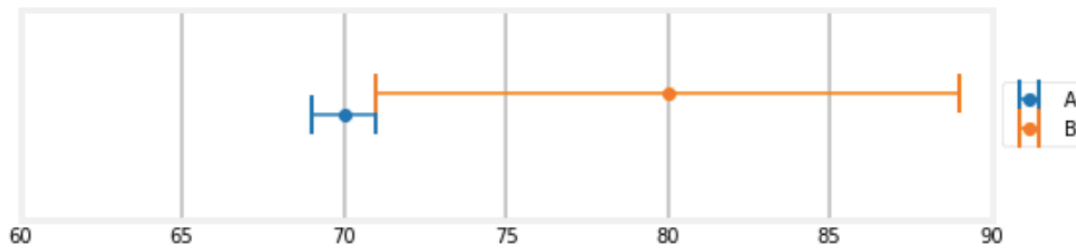


Figure 11.2: Error bars of Scale A with 70 ± 1 and Scale B with 80 ± 9 .

With two relatively inaccurate sensors we are able to deduce an extremely accurate result. So **two sensors, even if one is less accurate than the other, are better than one.** This is one of the most important lessons I have learned with this filter: **We never throw information away**, no matter how poor it is.

However, we have stayed from our problem. No customer is going to want to buy multiple scales, and besides, scales should be equally (in)accurate. What if I weight myself multiples times on the same scale? This answer is not very practical. No one has the patience to weigh themselves ten thousand times, or even a dozen times.

Let's continue with the "what's if". What's if you measured your weight once a day, and got the readings 80, 70.5 and then 79.5. Did you gain weight, loss weight or is this all just noisy measurements? We can't really say. There is an extreme range of weight changes that could be explained by these 3 measurements. However, we are measuring a human's weight so, there is no reasonable way for a human to weigh 88 Kg on the day 1, and 72 Kg the day 3 (assuming an inaccuracy of 10 Kg in the readings).

The behavior of the physical system we are measuring should influence how we interpret the measurements.

Suppose I take the following measurement with a scale: 70, 70.5, 69.5, 70.2, 69.8, 70.3, 70.5, 69.6, 70.1, 70.4. What does the intuition tell you? It is possible you gained a kilo every day and the noisy measurements happens to look like you stayed the same weight. Equally, you could have lost a kilo a day and got the same readings. But is that likely? How likely is it to flip a coin and get 10 heads in a row? Not very likely. We can't prove it but it seems pretty likely that the weight held steady.

Another what if: What if the readings were 70.2, 72.3, 71.4, 72.2, 73.2, 73.7, 74.9, 74.3, 75.2, 75.6. This data trends upward over time; not evenly but definitely upwards. We can see a trend on the weight gain.

Let's try something. Let's assume that I know I am gaining a kilo a day. It doesn't matter how I know that right now, just assume I know it is approximately correct.

The first measurement was 70.2 Kg. If our weight today is 70.2, what will it be tomorrow? Well, we think we are gaining weight at 1 kilo/day, so our prediction is 71.2.

Okay, but what good is this? Sure, we could assume the 1 kilo/day gain is accurate, and predict our weight for the next 10 days, but then why use a scale at all if we don't incorporate its readings. So let's look at the next measurement. We step on the scale again and it displays 72.3 Kg.

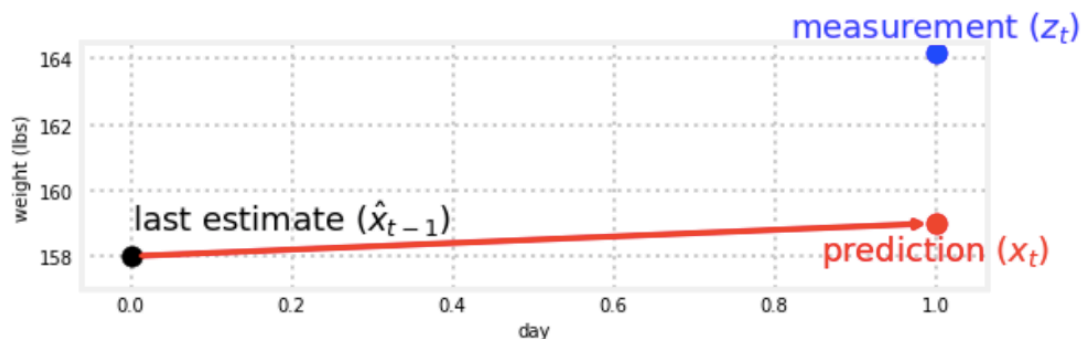


Figure 11.3: Example on how the prediction and the measurement differ.

We have a problem. **Our prediction doesn't match our measurement... and that's good.** If the prediction was always exactly the same as the measurement, it would not be capable of adding any information to the filter. And, of course, there would be no reason to ever measure since our predictions are perfect.

The key of this kind of filters is in this paragraph: **If we only form estimates from the measurement, then the prediction will not affect the result. If we only form estimates from the prediction then, the measurement will be ignored. If this is to work, we need to take some kind of blend of the prediction and the measurement.**

All I have done is replaced an inaccurate scale with an inaccurate weight prediction based of human physiology. It's still data. Math doesn't know if it came from a scale or a prediction. We have two pieces of data with a certain amount of noise, and we want to combine them.

Should the estimate be half-way between the measurement and prediction? Maybe, but in general, it seems like we might know that our prediction is more or less accurate

compared to the measurements. Probably the accuracy of our prediction differs from the accuracy of the scale.

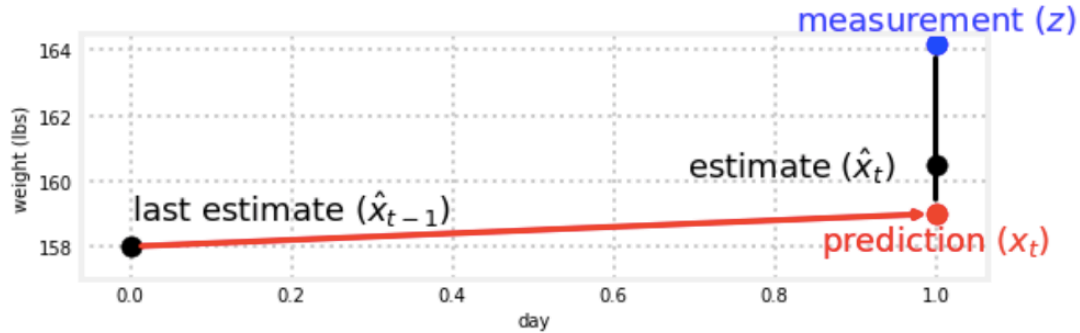


Figure 11.4: Estimate calculation based on the prediction and measurement.

Now let's try a randomly chosen number to scale our estimate: 4/10. With this, we are expressing the belief that the prediction is somewhat more likely to be correct than the measurement. We compute that as:

$$estimate = prediction + \frac{4}{10} * (measurement - prediction) \quad (11.1)$$

The difference between the measurement and the prediction is called the residual. This will become an important value to use later on. Smaller residuals imply better performance.

Let's see in the table §11.1 how this works with real data (assume an initial weight of 70.2 Kg, a gain rate of 1 Kg per day and a scale factor of 4/10):

We can see how our filter needs an initial gain rate to work as expected. If we choose a bad gain rate, the estimates would be very different from the actual data. As you can guess, a filter where you have to predict the gain rate is not very useful. If there is a change in the gain rate during the process, the filter will fail. What if we calculate the gain rate based on the existing measurements and estimates. This is very similar to the problem of the prediction and the measurement. We have the old gain and the new calculated gain. We need a way to merge these two: I will choose an arbitrary number (1/3) to weight the importance of both gains.

$$new_gain = old_gain + \frac{1}{3} * \left(\frac{residual}{time_step} \right) \quad (11.2)$$

Example with real data			
Previous	Prediction	Measurement	Estimate
70.2	71.2	72.3	71.64
71.64	72.64	71.4	72.15
72.15	73.15	72.2	72.77
72.77	73.77	73.2	73.54
73.54	74.54	73.7	74.21
74.21	75.21	74.9	75.08
75.08	75.08	74.3	75.37
75.37	76.37	75.2	75.90
75.90	76.90	75.6	76,38

Table 11.1: Multiple iterations with an initial weight of 70.2 Kg, a gain rate of 1 Kg per day and a scale factor of 4/10

Assuming an initial weight of 70.2 Kg, an initial gain rate of 1kg per day, a measurement scale factor of 4/10 and a gain scale factor of 1/3, we get the result of the table §11.2:

This algorithm is known as the g-h filter. **g and h refer to the two scaling factors that we used in our example.** g is the scaling we used for the measurements (the 4/10) and h is the scaling for the change in measurements over time (the 1/3).

This filter is the basis for a huge number of filters, including the Kalman Filter. The math may look profoundly different, but the algorithm will be exactly the same.

1. Multiple data points are more accurate than one data point, so throw nothing away no matter how inaccurate it is.
2. Always choose a number between two data points to create a more accurate estimate.
3. Predict the next measurement and the rate of change based on the current estimate and how much we think it will change.
4. The new estimate is then chosen as part away between the prediction and next measurement scaled by how accurate each is.

Now let's explore a few different problem domains to better understand this algorithm:

Example with real data				
Previous	Update Gain	Prediction	Measurement	Estimate
70.2	1.0	71.2	72.3	71.64
71.64	1.7	73.34	71.4	72.56
72.56	1.05	72.61	72.2	72.45
72.45	0.91	73.36	73.2	73.3
73.3	0.86	74.16	73.7	73.98
73.98	0.71	74.69	74.9	74.77
74.77	0.78	75.55	74.3	75.05
75.05	0.36	75.41	75.2	75.33
75.33	0.29	75.62	75.6	75.61

Table 11.2: Multiple iterations with an initial weight of 70.2 Kg, an initial gain rate of 1kg per day, a measurement scale factor of 4/10 and a gain scale factor of 1/3

Consider the problem of trying to track a train. Trains are large and slow. It takes many minutes for them to slow down or speed up significantly. So, if I know a train is at the kilometer 23 at time t and moving at 18 kph, I can be very confident in predicting its position at time $t+1$. In this example, we would set 'g' to a number close to 0 because we trust our prediction.

On the other hand, if we try to track an object in a hurricane, we cannot trust our prediction at all. We will have to rely on the measurements and basic physics; an object cannot move 10 kilometers in one second.

11.2.2 Formal terminology

Let me introduce some more formal terminology:

1. The **system** is the object we want to estimate. In previous examples, it has been the weight.
2. The **state of the system** is the current configuration or values of that system that is of interest to us. If I put a 100 Kg weight on the scale, the state is 100 Kg.
3. The **measurement is a measured value of the system**. As measurements can be inaccurate, it may differ from the value of the state.

4. The **state estimate is our filter's estimate of the state**. For example, for a 100 Kg weight, our estimate might be 99.45 Kg due to sensor errors. This is commonly abbreviated to estimate.
5. We use a **process model to mathematically model the system**. In previous examples, our process model is the assumption that my weight today is yesterday's weight plus my weight gain for the last day.
6. The system error or **process error is the error in this model**. We never know this value exactly, if we did, we could refine our model to have zero error.
7. The **predict state is known as system propagation**. It uses the process model to form a new state estimate. Due to the process error, this estimate is imperfect.

11.2.3 Discrete Bayes Filter

As we have done with the g-h Filter, let's use a simple thought experiment to see how we might reason about the use of probabilities for filtering and tracking.

Let's begin with a simple problem. Somebody invented a sonar sensor to attach to a dog's collar. It emits a signal, listen for the echo, and based on how quickly an echo comes back, we can tell whether the dog is in front of an open doorway or not. To keep the problem simple, we will assume that there are only 10 possible positions in the hallway and that the hallway is circular (if you move to the right from position 9, you are back to 0).

When I begin listening to the sensor, I have no reason to believe the dog is at any particular position in the hallway. There are 10 positions, so the probability that he is in any given position is 1/10. Figure §11.5 would be a basic representation of the probabilities of every position of the hallway.

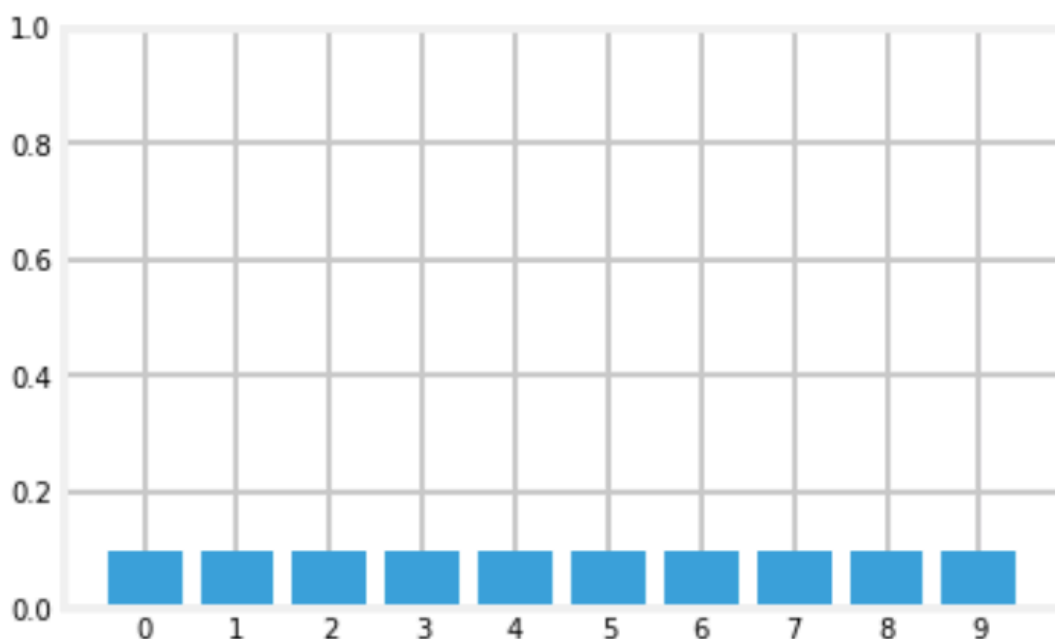


Figure 11.5: Representation of the probabilities of every position of the hallway.

In Bayesian statistics **this is called a prior**. It is the probability prior to incorporating measurements or any other information. A probability distribution is a collection of all possible probabilities for an event. Probability distributions always sum to 1 because something had to happen; the distribution lists all possible events and the probability of each.

Bayesian statistics takes past information (the prior) into account. We observe that it rains 4 times every 100 days. For this, I could state that the chance of rain tomorrow is $1/25$. This is not how weather prediction is done. If I know it is raining today and the storm front is stalled, it is likely to rain tomorrow. Weather prediction is Bayesian.

Now let's create a map of the hallway. We will use 1 for doors and 0 for walls.

I start listening to the dog's transmissions and the first data I get from the sensor is door. For now, let's assume the sensor always returns the correct answer. From this I conclude that it is in front of a door, but which one? I have no reason to believe he is in front of the first, second or third door. All doors are equally likely, so I assign, as seen in the figure §11.6, a probability of $1/3$ to each door.

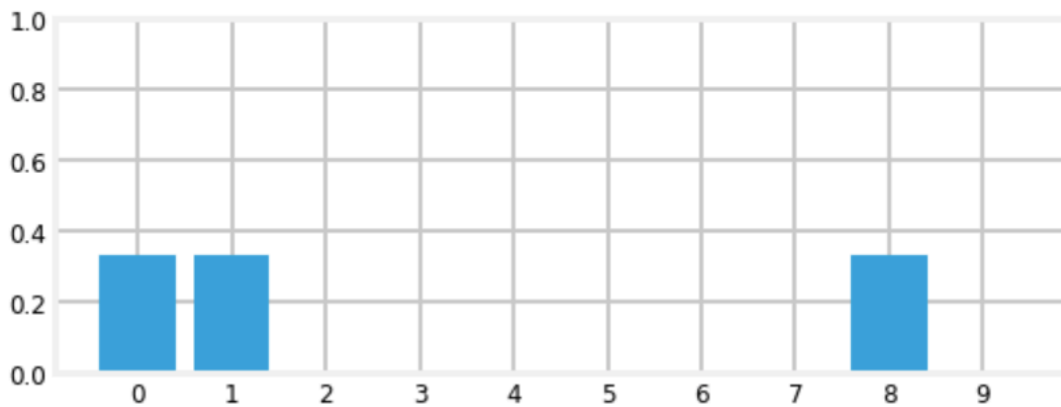


Figure 11.6: Representation of the probabilities of each door.

This is a multimodal distribution because we have multiple beliefs about the position of our dog. Of course, we are not saying that we think he is simultaneously in three different locations, merely that we have narrowed down our knowledge to one of these three locations.

This is an improvement in two ways. I've rejected a number of hallway positions as impossible, and the strength of my belief in the remaining positions has increased from 10% to 33%. This will always happen. As our knowledge improves, the probabilities will get closer to 100%.

Suppose we were to read the following from the sensor: Door, move right, door. Can we deduce the dog's location? Of course! Given the hallway's layout there is only one place from which you can get this sequence, and that is at the left end. Now, our belief is:

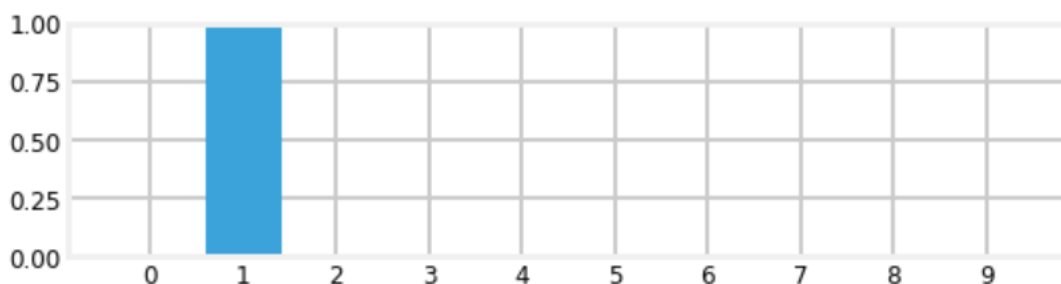


Figure 11.7: Representation of the probabilities after the new information.

This was a very simple example to explain the Bayesian Filter but first, we need to address the real world complications to the problem.

Perfect sensors are rare. Perhaps the sensor would not detect a door if the dog sat in front of it while scratching himself, or misread if he is not facing down the hallway, thus, when I get door I cannot use $1/3$ as a probability. I have to assign less than $1/3$ to each door, and assign a small probability to each blank wall position. Something like this figure §11.8:

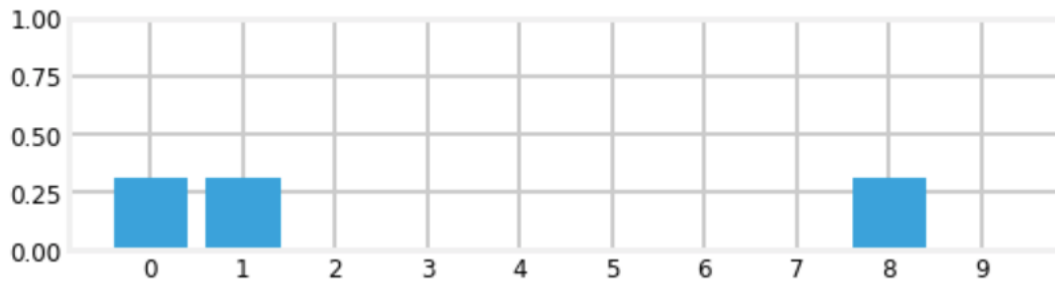


Figure 11.8: Representation of the probabilities with a noisy sensor.

If the sensor is noisy, it casts doubt on every piece of data. How can we conclude anything if we are always unsure? The answer, as for the problem above, is with probabilities. We are already comfortable assigning a probabilistic belief to the location of the dog; now we have to incorporate the additional uncertainty caused by the sensor noise.

Say we get a reading of door, and suppose that testing shows that the sensor is 3 times more likely to be right than wrong. We should scale the probability by 3 where there is a door, as seen in the figure §11.9.

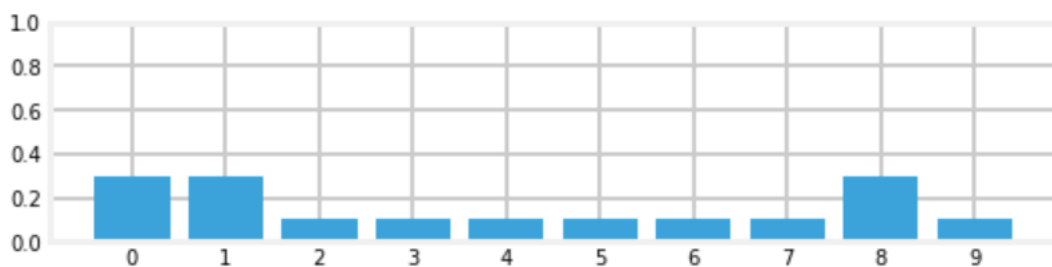


Figure 11.9: Representation of the probabilities where the sensor is more likely to be right than wrong.

As you can see, this is not a probability distribution, the sum of all these values is not 1, it's 1.6. In order to get a one, we need to do a normalization of the results, as

seen in the figure §11.10. Normalization is done dividing each element by the sum of all elements in the list.

$$\frac{\text{belief}}{\text{sum}(\text{belief})} \quad (11.3)$$

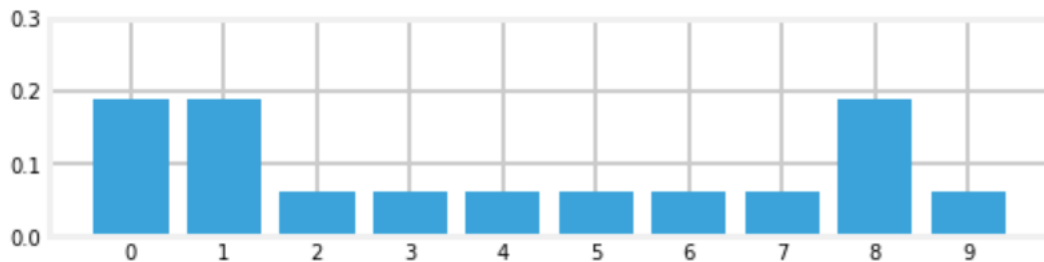


Figure 11.10: Representation of the probabilities where the sensor is more likely to be right than wrong (After normalization).

We can see that the sum is now 1.0 and that the probability of a door vs wall is still three times larger. The result also fits our intuition that the probability of a door must be less than 1/3 and that the probability of a wall must be greater than 0.0.

This result is called the posterior, which is short for posterior probability distribution. All this means is a **probability distribution after incorporation the measurement from the sensor**.

Now, it's time to incorporate movement. Assume the movement sensor is perfect, and it reports that the dog has moved one space to the right. How would we alter our belief array?

We should shift all the values one space to the right.

If we thought there was a 0.188 change of the dog being at position 1, then after it moved to the right, we should believe that there is a 0.188 change it is at position 2.

What if the sensor reported that our dog moved one space, but he actually moved to spaces, or zero? This may sound like an insurmountable problem, but let's model it and see what happens.

Assume that the sensor's movement measurement is 80% likely to be correct, 10% likely to overshoot one position to the right and 10% likely to undershoot one position to the left. That is, if the movement measurement is 4 (meaning 4 spaces to the right),

the dog is 80% likely to have moved 4 spaces to the right, 10% to have moved 3 spaces and 10% to have moved 5 spaces.

For example, consider the reported movement of 2. If we are 100% certain the dog started from position 3, then there is an 80% change it is at 5, and a 10% change for either 4 or 6. This can be seen in the figure §11.11.

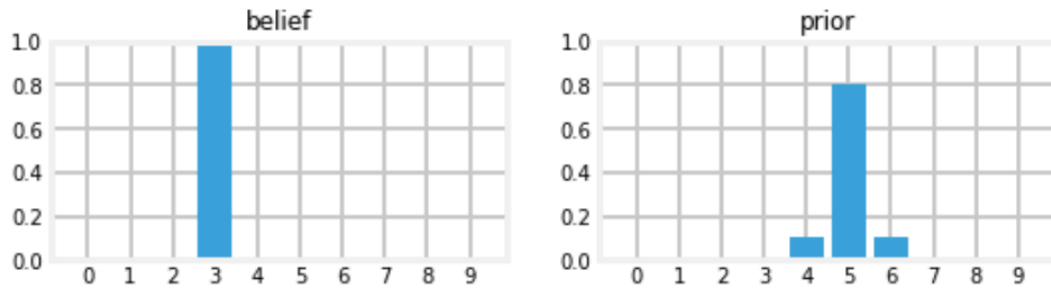


Figure 11.11: Representation of a movement with movement uncertainty.

What happens when our belief is not 100% certain, as seen in the figure §11.12?

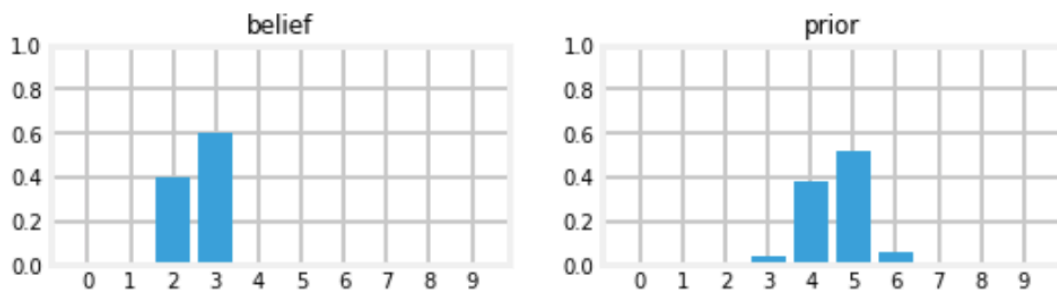


Figure 11.12: Another representation of a movement with movement uncertainty.

Here the results are more complicated, but you should still be able to work it out.

1. The 0.04 is due to the possibility that the 0.4 belief undershoot by 1.
2. The 0.38 is due to the following: the 80% change that we moved 2 positions ($0.4 * 0.8$) and the 10% change that we undershoot ($0.6 * 0.1$).
3. The 0.52 is due to the following: the 80% change that we moved 2 positions ($0.6 * 0.8$) and the 10% change that we overshoot ($0.4 * 0.1$).
4. The 0.06 is due to the possibility that the 0.6 belief overshoot by 1.

If the sensor is noisy, we lose some information on every prediction. Suppose we perform the prediction an infinite number of times. If we lose information on every step, we must eventually end up with no information at all.

The problem of losing information during the prediction may make it seem as if our system would quickly devolve into having no knowledge. However, **each prediction is followed by an update where we incorporate the measurement into the estimate. The update improves our knowledge.** The output of the update step is fed into the next prediction. Prediction degrades our certainty. That is passed into another update, where certainty is again increased.

Let's use the examples of the beginning of this section:

After the first update we have assigned a high probability to each door position, and a low probability to each wall position, as seen in the figure §11.13.

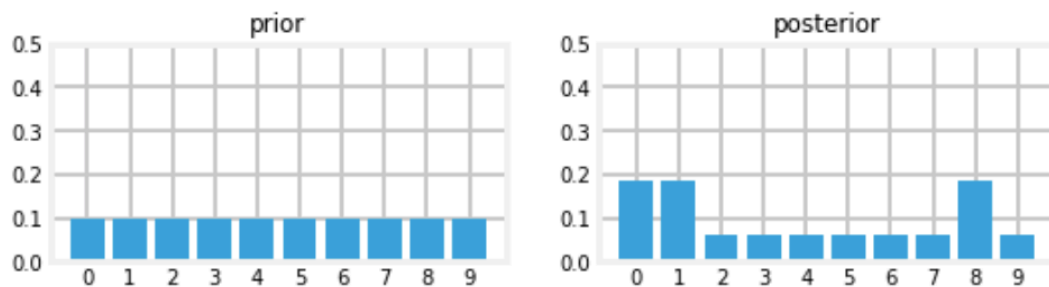


Figure 11.13: Initial prior and its posterior after new data.

The predict step shifted these probabilities to the right, as seen in the figure §11.14.

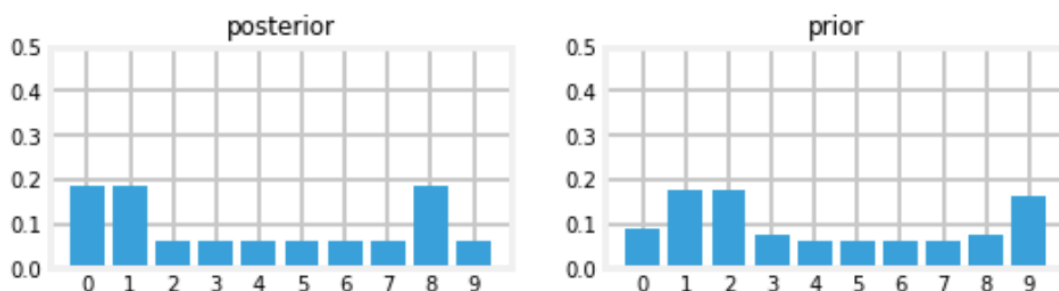


Figure 11.14: Posterior and new prior after prediction.

Note the tall bar at the position 1 in the figure §11.15. This corresponds with the (correct) case of starting at position 0, sensing the door, shifting 1 to the right, and sensing another door. No other positions make this set of observations as likely.

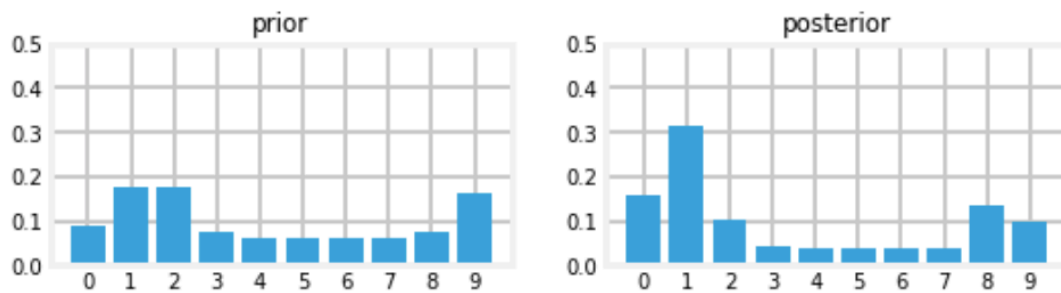


Figure 11.15: New prior and updated posterior.

This is the basis of the Discrete Bayes Filter. As with the g-h Filter, **we predict the future value based on the knowledge of the system, and then we add a measurement to compensate the lost of information during the prediction.**

This use of probabilities is the basis we will use in the Kalman Filter but the Bayesian Filter presents a serie of problems:

1. First, it's scaling. In these examples, we wanted to track one variable, the position of the dog. In real world problems, we will be tracking more variables and **the time complexity will increase with every new variable.**
2. Second, **it's the discrete aspect of the filter.** We live in a continuous world. A 100 meters hallway requires 10.000 positions to model the hallway to 1cm accuracy. So each update and predict operation would entail performing calculations for 10.000 different probabilities. It gets worse if we add dimensions. A 100x100 m² courtyard requires 100.000.000 positions.
3. Third, it's **multimodal.** It's useless to report that there is a change of 40% of being in one place and 30% in another place.

11.2.4 Gaussian Probabilities

We desire an unimodal, continuous way to represent probabilities that models how the real world works, and that is computationally efficient to calculate. Gaussian distributions provide all of these features.

Many real world observations are distributed following a gaussian distribution. It can be described with two parameters, the mean (μ) and the variance (σ^2).

The mean (μ) is what it sounds like, the average of all possible probabilities. Because of the symmetric shape of the curve, it is also the tallest part of the curve.

The square root of the variance is called the standard deviation (σ). The standard deviation is a measure of how much variation from the mean exists. For a Gaussian distribution, 68% of all the data falls within one standard deviation ($\pm 1\sigma$) of the mean, 95% falls within two standard deviations ($\pm 2\sigma$) and 99.7% within three ($\pm 3\sigma$). This is often called the 68-95-99.7 rule. If you were told that the average test score in a class was 5.9 with a standard deviation of 0.82, you could conclude that 95% of the students received a score between 4.26 and 7.54 if the distribution is normal.

A remarkable property of Gaussians is that **the sum of two independent Gaussians is another Gaussian. The product is not a Gaussian, but proportional to a Gaussian.** The discrete Bayes filter works by multiplying and adding arbitrary probability distributions. **The Kalman filter uses Gaussians instead of arbitrary distributions, but the rest of the algorithm remains the same.** This means we will need to multiply and add Gaussians.

The sum of two Gaussians is given by:

$$\mu = \mu_1 + \mu_2 \quad (11.4)$$

$$\sigma^2 = \sigma_1^2 + \sigma_2^2 \quad (11.5)$$

The product of two independent Gaussians is given by

$$\mu = \frac{\sigma_1^2 * \mu_2 + \sigma_2^2 * \mu_1}{\sigma_1^2 + \sigma_2^2} \quad (11.6)$$

$$\sigma^2 = \frac{\sigma_1^2 * \sigma_2^2}{\sigma_1^2 + \sigma_2^2} \quad (11.7)$$

The following points must be understood before we continue:

1. Normals express a continuous probability distribution.
2. They are completely described by two parameters: the mean (μ) and the variance σ^2 .

3. μ is the average of all possible values.
4. σ^2 represents how much our measurements vary from the mean.
5. The standard deviation (σ) is the square of the variance (σ^2).
6. Many aspects of nature approximate a normal distribution.

11.2.5 One Dimensional Kalman Filters

Now that we understand the discrete Bayes filter and Gaussians, we are prepared to implement a Kalman filter.

"One dimensional" means that the filter only tracks one state variable. In later sections I will explain the multidimensional form of the filter. The reason why I prefer to explain the one-dimensional version first is to avoid the heavy math present in the multidimensional filter. Here, we will learn the logic of the filter. The heavy math will be delayed until the understanding of the logic is solid.

As in the Discrete Bayes Filter section, we will be tracking a moving object in a long hallway. Here, we assume that the sensor provides a reasonably accurate position of the dog. The sensor returns the distance of the dog from the left end of the hallway in meters.

The sensor is not perfect. A reading of 23.4 could correspond to the dog being at 23.7 or 23.0. However, it is very unlikely to correspond to a position of 47.6. The errors seemed to be evenly distributed on both sides of the true position; a position of 23 meters would equally likely be measured as 22.9 or 23.1.

We predict that the dog is moving. This prediction is not perfect. Sometimes our prediction will overshoot, sometimes it will undershoot. We are more likely to undershoot or overshoot by a little than a lot.

We can express our belief in the dog's position with a Gaussian. Say we believe that our dog is at 10 meters and the variance in that belief is 1 m^2 , or $\mathcal{N}(10,1)$, as seen in Figure §11.16.

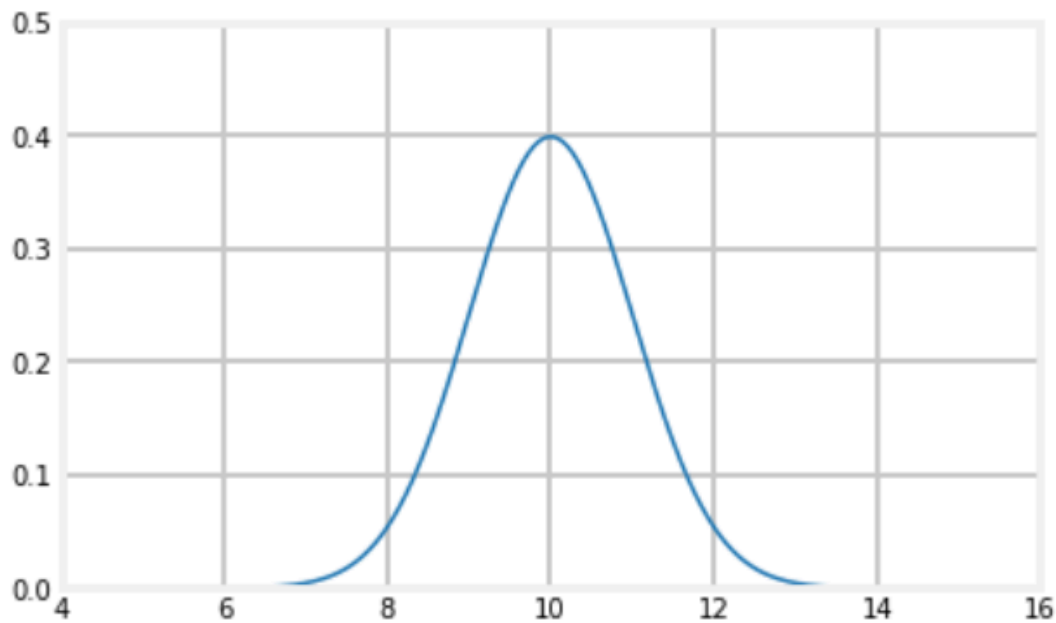


Figure 11.16: Graphical representation of the belief as a Gaussian

This plot describes our uncertainty about the dog's position. While we believe that it is most likely that the dog is at 10m, any position from 9 to 11m or so are quite likely as well.

Let's see how predictions work with gaussians.

If the dog is at 10m, his velocity is 15m/s and the epoch is 1 seconds long, we have:

$$x_k = 10 + (15 * 1) = 25m \quad (11.8)$$

We are uncertain about its current position and velocity, so this will not work. We need to express the uncertainty with a Gaussian.

If we think the dog is at 10m, and the standard deviation of our uncertainty is 0.2m, we get

$$x = \mathcal{N}(10, 0.2^2) \quad (11.9)$$

If the dog's velocity is 15 m/s, the epoch is 1 second, and the standard deviation of

our uncertainty is 0.7 m/s, we get:

$$f(x) = \mathcal{N}(15, 0.7^2) \quad (11.10)$$

The equation of the prior is

$$x_{prior} = x + f(x) \quad (11.11)$$

Let's do the math with the gaussians:

$$x_{prior} = \mu(x) + \mu(f(x)) = 10 + 15 = 25 \quad (11.12)$$

$$\sigma_{prior}^2 = \sigma^2(x) + \sigma^2(f(x)) = 0.2^2 + 0.7^2 = 0.53 \quad (11.13)$$

It makes sense that the predicted position is the previous position plus the movement. What about the variance? Recall that in the prediction in the discrete Bayes filter we always lost information. We don't really know where the dog is moving so the confidence should get smaller (variance gets larger).

Now, let's cover the updates with gaussians.

We've just shown that we can represent the prior with a Gaussian. What about the likelihood? The likelihood is the probability of the measurement given the current state. We've learned how to represent measurements as a Gaussian. For example, maybe our sensor states that the dog is at 23 m with a standard deviation of 0.4 m. Our measurement, expressed as a likelihood, is

$$z = \mathcal{N}(23, 0.4^2) \quad (11.14)$$

Both the likelihood and prior are modeled with Gaussians. We can multiply two Gaussians but the result is not a Gaussian, but proportional to one. If we normalize the result, the product is another Gaussian, as seen in Figure §11.17.

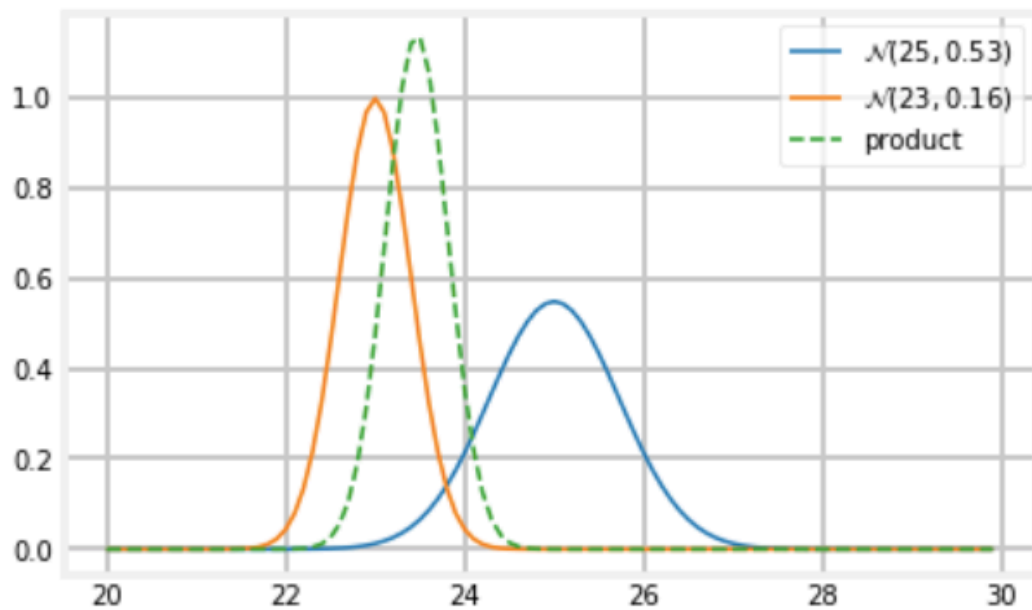


Figure 11.17: Graphical representation of the multiplication as a Gaussian

$$\mathcal{N}(\mu, \sigma^2) = \mathcal{N}(25, 0.53) * \mathcal{N}(23, 0.16) = \mathcal{N}(23.46, 0.12) \quad (11.15)$$

As in the discrete Bayes filter, after an update step the variance of the result decreases due to the incorporation of the measurement to the prediction.

This succession of predict and update steps using Gaussians is a Kalman Filter. This version is only useful if we are tracking only one variable. The multivariate version will address this problem.

The example above is useful to describe how the Kalman Filter works in a concrete case. Now that we understand the logic, we can write this version of the Kalman Filter using the standard nomenclature.

This is not the actual code. The code will be delivered using Swift. This is only a pseudocode using the "official" variable names:

```
func predict(posterior, movement) {
    let x, P = posterior // Mean and variance of posterior
    let dx, Q = movement // Mean and variance of movement

    x = x + dx
    P = P + Q
}
```

```

    let prior = Gaussian(x, P)

    return prior
}

func update(prior, measurement) {
    let x, P = prior           // Mean and variance of prior
    let z, R = measurement     // Mean and variance of measurement

    y = z - x                 // Residual
    K = P / (P + Q)           // Kalman Gain

    x = x + K * y             // Posterior mean
    P = (1 - K) * P           // Posterior variance

    let posterior = Gaussian(x, P)

    return posterior
}

```

Before we dive in the multivariate filter, let's see what some of these variables mean. In the literature R is nearly universally used for the measurement noise, Q for the process noise and P for the variance of the state.

K is the letter used to represent the Kalman Gain. To understand this concept we must study a little about the math behind the filter. The posterior x is computed as the likelihood times the prior, where both are Gaussians.

The mean of the posterior is given by:

$$\mu = \frac{\sigma_{prior}^2 * \mu_z + \sigma_z^2 * \mu_{prior}}{\sigma_{prior}^2 + \sigma_z^2} \quad (11.16)$$

This can be rewrite like this:

$$\mu = \frac{\sigma_{prior}^2}{\sigma_{prior}^2 + \sigma_z^2} * \mu_z + \frac{\sigma_z^2}{\sigma_{prior}^2 + \sigma_z^2} * \mu_{prior} \quad (11.17)$$

In this form it is easy to see that we are scaling the measurement and the prior by

weights:

$$\mu = W_1 * \mu_z + W_2 * \mu_{prior} \tag{11.18}$$

The weights sum to one because the denominator is a normalization term. We introduce a new term, $K = W_1$, giving us:

$$\mu = K * \mu_z + (1 - K) * \mu_{prior} = \mu_{prior} + K * (\mu_z - \mu_{prior}) \tag{11.19}$$

where

$$K = \frac{\sigma_{prior}^2}{\sigma_{prior}^2 + \sigma_z^2} \tag{11.20}$$

K is the Kalman Gain. It's the basis of the Kalman Filter. It is the scaling term that chooses a value partway between μ_z and μ_{prior} , thus, the residual. This is the way of reasoned about the g-h filter. It emphasized taking the residual $y = \mu_z - \mu_{prior}$ finding the Kalman gain as a ratio of our uncertainty in the prior and the measurement $K = \frac{P}{P+Q}$, and computing the posterior by adding $K * y$ to the prior.

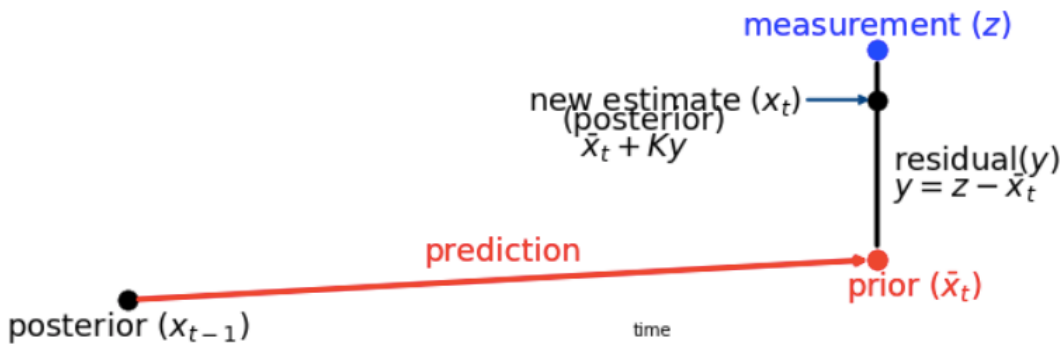


Figure 11.18: Full Description of the Algorithm.

A step by step algorithm can be described as following:

Initialization:

1. Initialize the state of the filter.
2. Initialize our belief in the state.

Predict:

1. Use the system behavior to predict the state at the next time step.
2. Adjust the belief to account for the uncertainty in prediction.

Update:

1. Get a measurement and associated belief about its accuracy.
2. Compute residual between the estimated state and the measurement.
3. Compute scaling factor based on whether the measurement or prediction is more accurate.
4. Set state between the prediction and measurement based on scaling factor.
5. Update belief in the state based on how certain we are in the measurement.

11.2.6 Multivariate Gaussians

The techniques used in the last section are very powerful, but they only work with one variable or dimension. They provide no way to represent multidimensional data, such as the position and velocity of an object in a field. Position and velocity are related to each other, and we learned in the g-h filter section that we should never throw away information. In this section I will explain how to describe these relationships probabilistically.

Our goal in this section will be to represent a normal distribution with multiples dimensions, or variables. We need to represent their means and variances, just as we do with Univariate Gaussians.

Let's believe that $x = 2$ and $y = 17$:

$$\mu = \begin{bmatrix} 2 \\ 7 \end{bmatrix} \quad (11.21)$$

We will say that the variance of x is 10 and the variance of y is 4:

$$\sigma^2 = \begin{bmatrix} 10 \\ 4 \end{bmatrix} \quad (11.22)$$

This is incomplete because it does not consider the correlations between the variables. If we were measuring the height and weight of a group of students, we can deduce that the taller students will, generally, weight more than the shorter ones. Thus, height and weight are correlated. We want to express not only what we think the variance is in the height and the weight, but also, the degree to which they are correlated.

The way to achieve this is using a covariance matrix. This matrix is an $N \times N$ matrix (where N is the number of variables). **The diagonal of the matrix represents the variances of the variables.** The rest of spaces are left for the covariance between the variables.

In the previous example, instead of a 2×1 matrix, we have a 2×2 matrix:

$$\sigma^2 = \begin{bmatrix} 10 & ? \\ ? & 4 \end{bmatrix} \quad (11.23)$$

If we consider that there is no correlation between x and y we get this matrix:

$$\sigma^2 = \begin{bmatrix} 10 & 0 \\ 0 & 4 \end{bmatrix} \quad (11.24)$$

If x increases as y increases, we would get this:

$$\sigma^2 = \begin{bmatrix} 10 & 1.2 \\ 1.2 & 4 \end{bmatrix} \quad (11.25)$$

If x increases as y decreases, we would get this instead:

$$\sigma^2 = \begin{bmatrix} 10 & -9.7 \\ -9.7 & 4 \end{bmatrix} \quad (11.26)$$

(Note that the covariance values are arbitrary. Only the sign of the covariances are intended).

As you can see, the covariance matrix is symmetric. After all, the covariance between x and y is always equals to the covariance between y and x .

Thinking of the physical representation of this Gaussians clarifies their meaning.

For this example I will be using this values:

$$\mu = \begin{bmatrix} 2 \\ 7 \end{bmatrix} \quad (11.27)$$

$$\sigma^2 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad (11.28)$$



Figure 11.19: Graphical representation of the belief as a Gaussian.

A Bayesian way of thinking about this is that it shows the amount of error in our belief. A tiny circle would indicate that we have a very small error, and a very large circle indicates a lot of error in our belief. The shape of the ellipse shows us the relationship of the errors in x and y . Here, in Figure §11.19, the errors of x and y are equally likely. If we use this values, we would get this other plot.

$$\mu = \begin{bmatrix} 2 \\ 7 \end{bmatrix} \quad (11.29)$$

$$\sigma^2 = \begin{bmatrix} 2 & 0 \\ 0 & 6 \end{bmatrix} \quad (11.30)$$

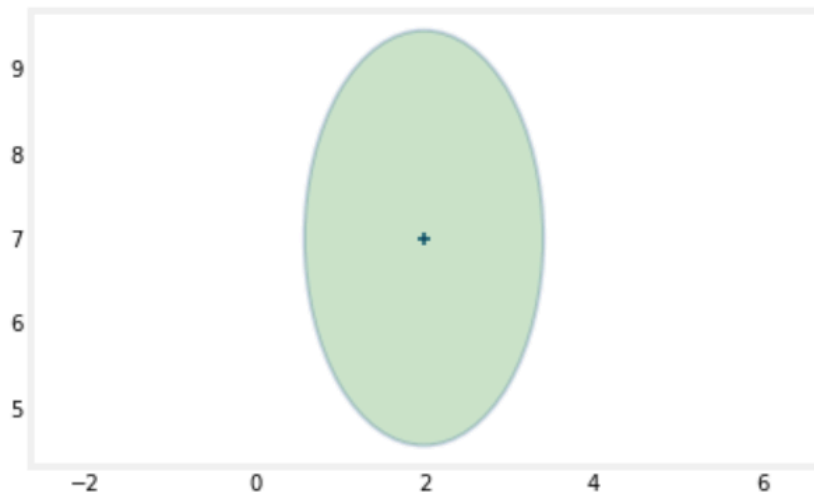


Figure 11.20: Graphical representation of the belief as a Gaussian.

Here, in Figure §11.20, there is a lot more uncertainty in y than in x . If we use matrices with non-zero values in their covariances, we get this plot:

$$\mu = \begin{bmatrix} 2 \\ 7 \end{bmatrix} \quad (11.31)$$

$$\sigma^2 = \begin{bmatrix} 2 & 1.2 \\ 1.2 & 2 \end{bmatrix} \quad (11.32)$$

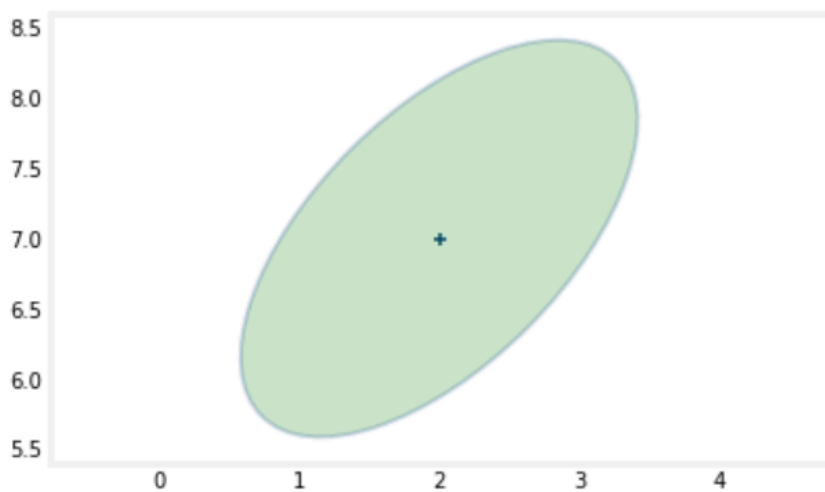


Figure 11.21: Graphical representation of the belief as a Gaussian.

Here, in Figure §11.21 the ellipse is tilted. This is due to the positive correlation between the errors of x and y .

I won't explain how to multiply two Multivariate Gaussians. It's very similar to how we did in previous sections. My goal with this chapter is to explain how to represent multiple variables with Gaussians. The sum and multiplications works similar. What I would like to explain is the concept of hidden variables.

Imagine you are tracking a plane and you get this positions at time $t=1, 2$ and 3 seconds.

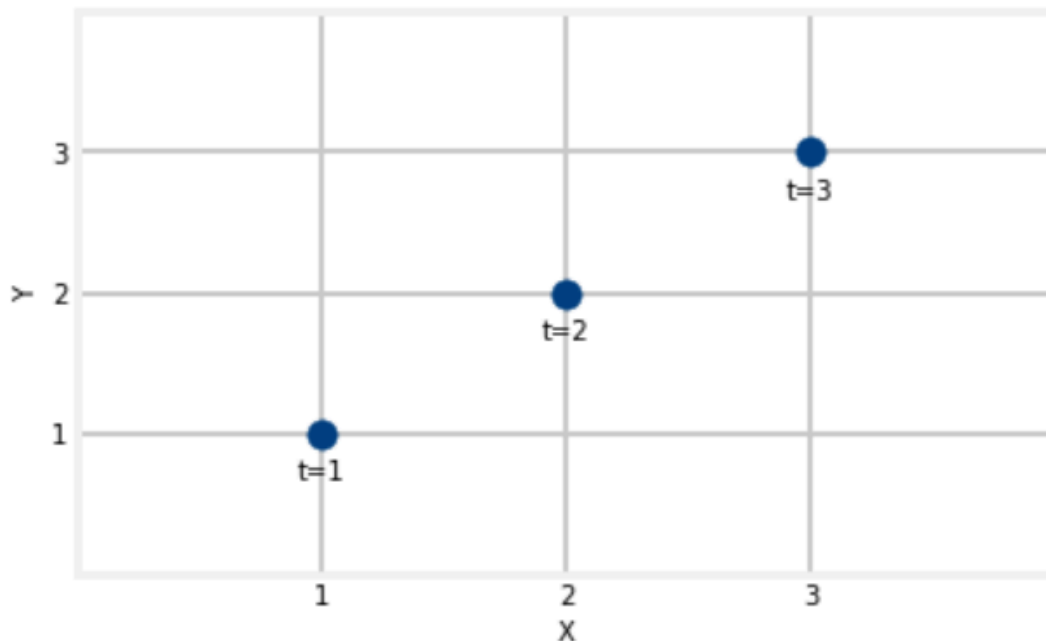


Figure 11.22: Graphical representation of the data at time $t=1, 2$ and 3 seconds.

What does your intuition tell you the value of x will be at time $t=4$ seconds? I'm sure you thought the plain would be at $(4,4)$. You inferred a constant velocity for the airplane. The reasonable assumption is that the aircraft is moving one unit each in x and y per time step. This is very similar of what we did with the g-h filter when we were trying to improve the weight prediction of the noisy scale. We incorporated weight gain into the equations because it allowed us to make a better prediction of the weight the next day. We are going to do the same with the Kalman Filter.

11.2.7 Multivariate Kalman Filters

We are now ready to study and implement the full, multivariate form of the Kalman Filter. In the last section we learned how multivariate Gaussians express the correlation between multiple random variables, such as the position and velocity of an aircraft. We also learned how correlation between variables improves the posterior.

In order to explain the Kalman Filter with more ease, I will restrict the problem to those described with Newton's equation of motion.

The univariate Kalman Filter represented the state with a univariate Gaussian. Naturally, the multivariate version will use multivariate Gaussians for the state. We learned in the last chapter that multivariate Gaussians use a vector for the mean and a matrix for the covariances. That means that the Kalman Filter needs to use linear algebra to perform the estimations.

Now, I will compare the equations of the univariate and multivariate version of the filter in table §11.3 and §11.4:

Predict step	
Univariate version	Multivariate version
$x_{prior} = x + dx$	$x_{prior} = F * x + B * u$
$P_{prior} = P + Q$	$P_{prior} = F * P * F^T + Q$

Table 11.3: Predict step equations

Predict Step:

- x and P are the state mean and covariance. They correspond to μ and σ^2
- F is the state transition function. When multiplied by x it computes the prior.
- Q is the process covariance.
- B and u are new to us. They let us model control inputs to the system.

Update Step:

- H is the measurement function. If you mentally remove H from the equations, you should be able to see that these equations are quite similar.

Update step	
Univariate version	Multivariate version
$y = z - x_{prior}$	$y = z - H * x_{prior}$
$K = \frac{P_{prior}}{P_{prior} + R}$	$K = P_{prior} * H^T * (H * P_{prior} * H^T + R)^{-1}$
$x = x_{prior} + K * y$	$x = x_{prior} + K * y$
$P = (1 - K) * P_{prior}$	$P = (I - K * H) * P_{prior}$

Table 11.4: Update step equations

- z and R are the measurement mean and noise covariance. They correspond to z and σ_z^2 in the univariate filter.
- y and K are the residual and the Kalman gain.

There are a few differences between both versions but the concepts are very similar:

1. Use a Gaussian to represent our estimate of the state and error.
2. Use a Gaussian to represent the measurement and its error.
3. Use a Gaussian to represent the process model.
4. Use the process model to predict the next state (the prior).
5. Form an estimate part way between the measurement and the prior.

The job as a designer will be to design the state (x, P) , the process (F, Q) , the measurement (z, R) , and the measurement function H . If the system has control inputs, we'll need to design B and u .

Let's go back to our problem of tracking a dog. Let's start with the predict step:

First, **we need to design the state variable**. To do so, we need to know what variables are we going to track. In this problem, they will be the position and velocity of the dog.

State variables can either be observed variables (directly measured from a sensor) or hidden variables (inferred from the observed variables). For this problem, the sensor will only read positions, so position is observed and velocity, hidden. It is important to understand that tracking these variables is a design choice. In the project, I'll be

tracking observed accelerations and hidden velocities. For now, this is a mere example to help to understand the filter.

In the univariate version, we represented the dog's position with a scalar value (e.g. $\mu = 3.27$). In the multivariate version, we will be using matrices. We use a $n \times 1$ matrix (called a vector) to store n state variables.

To represent a position of 10 meters and a velocity of 4.5 m/s, we will do it like this:

$$x = \begin{bmatrix} 10.0 \\ 4.5 \end{bmatrix} \quad (11.33)$$

Second, **we need to design the state covariance**. This is the other half of the Gaussian, the variable P . In the multivariate version, we will need a matrix of covariances; the diagonal will contain the variances of the variables and the rest of spots will represent the correlation between the variables.

To represent a position variance of 500^2 (we are quite uncertain about the initial position) and a velocity variance of 49 m^2 , we will do it like this:

$$P = \begin{bmatrix} 500.0 & 0 \\ 0 & 49 \end{bmatrix} \quad (11.34)$$

The 49 m^2 value of the velocity variance is not random at all. The top speed of a dog is 21 m/s . Following the idea that 99.7% of the values are inside the 3σ of the velocity, $3\sigma = 21$, $\sigma = 7$, $\sigma^2 = 49$.

The zeros in the covariance means doesn't mean that there is no correlation between the variables. It does mean that we have no idea about the correlation, so we initialize the covariances to zero. If we knew the covariances, we would use them.

Third, **we need to design the process model**. The process model is based on the mathematical behavior of the system. Kalman Filters implement this using this linear equation, where x_{prior} is the prior, or predicted state: $x_{prior} = F * x$

This equation can be written like this:

$$\begin{bmatrix} x_{prior} \\ x_{vel_{prior}} \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} * \begin{bmatrix} x \\ x_{vel} \end{bmatrix} \quad (11.35)$$

Our job as designers is to specify F such that $x_{prior} = F * x$ performs the prediction of our system. To do this, we need one equation for each variable. In this example, we need an equation to compute the position x and another to compute the velocity x_{vel} . We already know the equation for the position:

$$x_{prior} = x + x_{vel} * t_{increment} \quad (11.36)$$

About the velocity equation, we have no predictive model, so we assume that it remains constant between predictions. With these equations we have the following process model:

$$x_{prior} = x + x_{vel} * t_{increment} \quad (11.37)$$

$$x_{vel_{prior}} = x_{vel} \quad (11.38)$$

We can write this more similar to the linear algebra way:

$$x_{prior} = 1 * x + t_{increment} * x_{vel} \quad (11.39)$$

$$x_{vel_{prior}} = 0 * x + 1 * x_{vel} \quad (11.40)$$

We can rewrite this like:

$$\begin{bmatrix} x_{prior} \\ x_{vel_{prior}} \end{bmatrix} = \begin{bmatrix} 1 & t_{increment} \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ x_{vel} \end{bmatrix} \quad (11.41)$$

being

$$F = \begin{bmatrix} 1 & t_{increment} \\ 0 & 1 \end{bmatrix} \quad (11.42)$$

F is called the state transition matrix.

Forth, **we need to design the process noise**. A way to explain what the process noise is, could be this: A car driving along the road with the cruise control on; it should

travel at a constant speed. However, it is affected by a number of unknown factors such as how does wind affect the car, as do hills.

Its design is quite demanding. There are some methods to model it: we can consider it a continuous white noise, a piecewise white noise or even use a simplification where we set all values to zero except for a noise term in the ones that change more rapidly, thus, the higher degree derivate (velocity in the example). This design of this noise turns out to be more experimental than theoretical, so I will skip it until I design the filter for my own problem.

Fifth, **we can design the control function**. The filter does not only filter data, it allows us to incorporate the control inputs of systems like robots and airplanes. Suppose we are controlling a train. The train goes with constant velocity, but if you activate the brakes, that's information we have to incorporate to the system. We will do it with B and u .

Here u is the control input and B , the control input model function. It must compute how much x changes due to the control inputs.

With all these six variables, we have the predict step done. Now we can implement the update step of the filter. You only have to supply two more matrices so, it won't take long.

First, **we need to design the measurement function**. Up until now, we track position using a sensor that measures positions. Computing the residual is easy, just subtract the prediction from the measurement. But, what would happen if we were trying to track temperature using a thermometer that outputs a voltage corresponding to a temperature reading? We cannot subtract a temperature prediction from a voltage reading.

Both the measurement z and the state x are vectors so, we need a matrix to perform the conversion. This matrix is the variable H . We have to design H so that $H * x_{prior}$ yields a measurement. For this problem, we have a sensor that measures position, so z will be a one variable vector:

$$z = \begin{bmatrix} z \end{bmatrix} \quad (11.43)$$

The residual equation will have the form

$$y = z - H * x_{prior} \quad (11.44)$$

$$\begin{bmatrix} y \end{bmatrix} = \begin{bmatrix} z \end{bmatrix} - \begin{bmatrix} ? & ? \end{bmatrix} * \begin{bmatrix} x \\ x_{vel} \end{bmatrix} \quad (11.45)$$

We will want to multiply the position x by 1 to get the corresponding measurement of the position. We do not need to use velocity to find the corresponding measurement, so we multiply x_{vel} by 0.

$$y = z - H * x_{prior} \quad (11.46)$$

$$\begin{bmatrix} y \end{bmatrix} = \begin{bmatrix} z \end{bmatrix} - \begin{bmatrix} 1 & 0 \end{bmatrix} * \begin{bmatrix} x \\ x_{vel} \end{bmatrix} = \begin{bmatrix} z \end{bmatrix} - \begin{bmatrix} x \end{bmatrix} \quad (11.47)$$

being

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad (11.48)$$

Last, **we need to design the measurement**. The measurement is implemented by z , the measurement mean, and R , the measurement covariance.

z is easy. It contains the measurement(s) as a vector. As we only have one measurement, we have:

$$z = \begin{bmatrix} x \end{bmatrix} \quad (11.49)$$

If we had two sensors we would have:

$$z = \begin{bmatrix} z1 \\ z2 \end{bmatrix} \quad (11.50)$$

The measurement noise matrix models the noise in our sensors as a covariance matrix. In practice, this can be difficult. A complicated system may have many sensors,

the correlation between them might not be clear, and usually their noise is not a pure Gaussian.

We have only a sensor so the covariance matrix R is:

$$R = \begin{bmatrix} \sigma_z \end{bmatrix} \quad (11.51)$$

If we had two sensors, the first with a variance of 5m^2 and the second with 3m^2 , we would write:

$$R = \begin{bmatrix} 5 & 0 \\ 0 & 3 \end{bmatrix} \quad (11.52)$$

The covariances between the noise of the different sensors is set to zero because we assume there is no correlation between their noises.

With these three matrices (H , z and R) we have completed the update step.

Let's see now the final equations of the system:

These are the predict equations

Mean of the prediction:

$$x_{prior} = F * x + B * u \quad (11.53)$$

$F * x$ computes the prediction of the state for the next time step. $B * u$ computes the contribution of the controls to the state after the transition.

Covariance of the prediction:

$$P_{prior} = F * P * F^T + Q \quad (11.54)$$

We cannot simply write $P_{prior} = P + Q$. In the multivariate version, the variables are correlated, so we multiply P by F and the transpose of F to include this correlation.

and these, the update equations

System uncertainty:

$$S = H * P_{prior} * H^T + Q \quad (11.55)$$

We cannot simply write $P_{prior} + R$. We have to ensure that the P_{prior} is in measurement space. If not, we won't be able to add the sensor noise (R). If you ignore the H terms, you can see how we get the denominator of the Kalman Gain ($K = \frac{\sigma^2}{\sigma^2 + \sigma_z^2}$).

Kalman Gain:

$$K = P_{prior} * H^T + S^{-1} \quad (11.56)$$

If you ignore the H term, you can see how we get the Kalman Gain equation from the univariate version of the filter.

Residual:

$$y = z - H * x_{prior} \quad (11.57)$$

This is a easy one, we just compute the difference between the measurement and the prediction.

State update:

$$x = x_{prior} + K * y \quad (11.58)$$

Here, as we did in the g-h Filter, we add to the prediction a weighted version of the residual.

Covariance update:

$$P = (I - K * H) * P_{prior} \quad (11.59)$$

Once again, if you remove the H term, you can see we get the univariate version ($P = (1 - K) * P_{prior}$).

In these equations, the H term is primary used to ensure we are working in measurement space, thus, we are not trying to subtract temperature to voltage as we saw

earlier.

I know this have been long and tedious, there is a lot of information, definitions, theory, concepts but there are necessary to understand what's going on. I first tried to understand the Kalman Filter using a bunch of papers[6][8][2][5][12][1][7][10]. In the second to third page of those papers, I already had the equations above. It was faster, but I could not understand a thing.

As someone who went from not understanding a thing about the filter to understand the reason behind the equations, I consider worth the long and tedious reading in order to know what you are working with. The easy part of the Kalman Filter are the equations. The problem is to design the matrices in a way they are useful. To do so, you need to understand them from bottom to the top, as I have tried to show you.

Now we have the Kalman filter, it's time to start coding and see what happens.

11.3 IMPLEMENTING THE FILTERS

Once I understood the principles of this kind of filters, I started to code them. The objective of this module is to enhance the accuracy of the velocity calculations so these filters should be coded in order to be implemented in the app.

Although my plan is to use the Kalman Filter, I wanted to code the g-h Filter, the discrete Bayes Filter, the Gaussian object and the univariate version of the Kalman Filter too.

I think it's better to start with an easy filter, understand how to code it properly, test it and then, jump to next one. If I start with the most difficult one, I will make all the mistakes with the most complex, thus, the more prone to errors.

All these filters can be found in the "Model" folder of the project delivered with the report [16].

11.4 UNIT TESTS

Once the filters were coded, I needed to test if there was any error in the implementation. For this task, I have used the test suite delivered by Apple, the **XCTest Framework**.

XCTest framework is the Xcode out-of-the-box test suite. As far as I know, the Java "equivalent", JUnit, is only used for Unit Tests. However, XCTest includes Unit Tests, Performance Test, UI Testing and can be configured to be run with Continuous Integration.

At this point, I will only use the Unit Testing functionality. My goal with these tests is to ensure the lack of errors in the implementation of the tests.

These tests can be found in the "Speed Gauge Tests" folder of the project delivered with the report.

The tests are very similar to the way we have done them in previous lectures, like Design and Testing. We have a *setUp* and *tearDown* method that must be present in every test file. To avoid the repetitive code, I have created the *Abstract_Test.swift* file, where these methods are declared. All test files must be declared as sons of this *Abstract_Test* class.

The structure of a test is trivial:

1. We import the XCTest suite.
2. We import the project module as `@testable` (in this case it's called "Speed Gauge". Note the difference with the name of the Test Module "Speed Gauge Tests").
3. In every test we declare the data we want to filter, the expected result and the filter itself.
4. We use the filter to treat the data, and we compare the result of the filter with the expected data. If this comparison successes, the test is passed.

Although, the structure is simple, we need to write useful test. A test must be "right" but also useful. The way I decided to implement the tests was using the understanding I have gained about what I was testing.

Let's use the most simple filter for the example, the g-h Filter:

As we know now, in the g-h Filter, we predict the value for the next time step, then, we get the measurement from the sensor from this future (now present) time step, and we choose a weighted mean between the prediction and the measurement as the value for this time step.

So, the test for this filter must cover a variety of cases:

1. What if I trust 100% in the prediction? The measurement must be thrown away.
2. What if I trust 100% in the measurement? The entry data and the filtered data must be the same.
3. What if I use the same data as the book I've been using? Do I get the same results? If not, something must be wrong in my implementation.

This has been basically the way I have tried to focus my tests. Here, in the figure §11.23 you can see the code of my g-h Filter tests.

```

9 import XCTest
10 @testable import Speed_Gauge
11
12 class G_H_Filter_Tests: Abstract_Test {
13
14     func test_init_1(){
15         let g_h_filter = G_H_Filter.init(x0: 160.0, dx: 1, g: 0.4, h: 0.3, dt: 1)
16
17         XCTAssertEqual(g_h_filter.x0, 160.0, accuracy: 0.01)
18         XCTAssertEqual(g_h_filter.dx, 1, accuracy: 0.01)
19         XCTAssertEqual(g_h_filter.g, 0.4, accuracy: 0.01)
20         XCTAssertEqual(g_h_filter.h, 0.3, accuracy: 0.01)
21         XCTAssertEqual(g_h_filter.dt, 1, accuracy: 0.01)
22     }
23
24     func test_filter_measurement_1(){
25         let z = 158.0
26         let expected = 159.8
27
28         let g_h_filter = G_H_Filter.init(x0: 160.0, dx: 1, g: 0.4, h: 0.3, dt: 1)
29         let result = g_h_filter.filter(z)
30
31         XCTAssertEqual(expected, result, accuracy: 0.01)
32         XCTAssertEqual(g_h_filter.x0, result, accuracy: 0.01)
33     }
34
35     /// Textbook example
36     func test_filter_data_1(){
37         let data: [Double] = [158.0, 164.2, 160.3, 159.9, 162.1, 164.6, 169.6, 167.4, 166.4, 171.0, 171.2, 172.6]
38         let expected: [Double] = [159.8, 161.62, 161.92, 161.46, 161.59, 162.82, 166.09, 168.23, 168.86, 170.34, 171.50, 172.67]
39
40         let g_h_filter = G_H_Filter.init(x0: 160.0, dx: 1, g: 0.4, h: 0.3, dt: 1)
41         let result = g_h_filter.filter(data)
42
43         _ = zip(expected, result).map { XCTAssertEqual($0, $1, accuracy: 0.01) }
44         XCTAssertEqual(g_h_filter.x0, result.last!, accuracy: 0.01)
45     }
46

```

Figure 11.23: g-h filter unit tests.

11.5 THE FLOATING POINT PROBLEM

Once I read that with floating-point numbers, **if you have one problem and you add another problem, you have 1.9999999997 problems**. That's basically the Floating Point Problem.

In the implementation of the filters, I've always used the type Double when I was dealing with numbers. The data from the sensor is delivered using this type, so I found no reason to use any other basic type to represent numbers.

Once I was doing my tests I found that I was getting failed tests, even in very basic cases. The first test I did was a simple declaration of the object, and then I check that every parameter of the object was correctly set. Even in this case, I would get a failed test.

After a basic research, I realized what was happening. It may sound trivial but it was unaware of this behavior.

In this paper[21], this behavior is explained. Long story short, squeezing infinitely real numbers into a finite number of bits requires an approximate representation. Although there infinitely many integers, the result of integer computations can be stored in 32 or 64 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore, the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. **This rounding error was the reason why all my tests were failing.**

As I couldn't modify the behavior of floating-point numbers, I did another research to know how to cope with these problems when doing Unit Tests. What I found was that XCTest already address this issue.

The method used to verify if an object is equal to another is called `XCTAssertEqual(expression1 : Equatable, expression2 : Equatable)`. `Equatable` states that the object can be compared with another one.

In order to avoid this problem, I had to use this other function: `XCTAssertEqual(expression1 : FloatingPoint, expression2 : FloatingPoint, accuracy : FloatingPoint)`. The accuracy is used as a threshold. Any value between "`expression1 + accuracy`" and "`expression1 - accuracy`" will be treated as equal to `expression1`.

11.6 INCLUDING EXTERNAL LIBRARIES IN UNIT TEST FILES

Another problem I found testing my filters was the use of external libraries in the tests. I couldn't import any external library added to the project using CocoaPods. I would get compile time error that didn't let me use them. This was a mayor problem because Kalman Filter is basically matrices so, I couldn't test it.

The problem was solved this way: As I was using CocoaPods to add the libraries to the project, I would set the target to my project "Speed Gauge". What I didn't know

was that tests are considered another target, separate from the project.

I was telling CocoaPods to let use the libraries with "Speed Gauge" but it didn't know "Speed Gauge Tests" even existed. I had to restructure the Pod file to add the test target. After that, the problem was solved.

```

15 Speed Gauge/Podfile
... @@ -1,12 +1,19 @@
1 # Uncomment the next line to define a global platform for your project
2 -#platform :ios, '9.0'
3
4 -target 'Speed Gauge' do
5   - # Comment the next line if you're not using Swift and don't want to use
   dynamic frameworks
6   - use_frameworks!
7
8   # Pods for Speed Gauge
9   pod 'Charts'
10  pod 'Surge'
11
12  end
1
2  # Uncomment the next line to define a global platform for your project
3  +platform :ios, '11.0'
4
5  +# Comment the next line if you're not using Swift and don't want to use
   dynamic frameworks
6  +use_frameworks!
7
8  +def available_pods
9    # Pods for Speed Gauge
10   pod 'Charts'
11   pod 'Surge'
12  +end
13  +target 'Speed Gauge' do
14    + available_pods
15  +end
16
17  +target 'Speed Gauge Tests' do
18    + available_pods
19  end

```

Figure 11.24: Changes in Podfile.

ITERATION #7: PARALLEL PROGRAMMING AND QUEUES

A single-threaded program has only one finger. But a multi-threaded program has multiple fingers. Each finger still moves to the next line of code as defined by the flow control statements, but the fingers can be at different places in the program, executing different lines of code at the same time.

*Albert Sweigart,
Software developer*

This chapter includes the study and use of parallel programming to improve the performance of the app. Section §12.1 talks about concurrency in the iOS platform, Section §12.2 displays the code necessary to make the code concurrent and Section §12.3 displays the performance improvements after the changes.

12.1 IOS CONCURRENCY

One of the most important advances in modern computing is the ability to perform more than one task at the time. Previous processors could not execute more than one instruction simultaneously. With multi-core processors that is not a problem anymore. You can write different parts of your code to be executed in multiple cores. This is important. You have to specify that a chunk of code is meant to be run concurrently. If not, you will have a single-core application running in a multi-core device.

There are two ways to achieve this in Apple operating systems: Using the Grand-Central-Dispatch or using Operation Queues[15].

The Grand-Central-Dispatch (GCD) is the most commonly used API to manage concurrent code and execute operations asynchronously at the UNIX level of the system. It's a low-level C API that enables developers to execute tasks concurrently.

Operation Queues, on the other hand, are a high level abstraction of the queue model and is build in top of GCD. That means you can execute tasks concurrently just like GCD, but in an object-oriented fashion.

In this project I have used the latter. I first tried with GCD but I found that Operation Queues led to a clearer code.

12.2 CONCURRENT CODE

In order to implement concurrent code I had to understand a few rules about concurrency.

First, **operation queues have different priorities.** These priorities are set in an enumerate called *QualityOfService*. There are 5 different values:

1. *userInteractive*: Used for work directly involved in providing an interactive UI. For example, processing control events or drawing to the screen.
2. *userInitiated*: Used for performing work that has been explicitly requested by the user, and for which results must be immediately presented in order to allow for further user interaction. For example, loading an email after a user has selected it in a message list.

3. *utility*: Used for performing work which the user is unlikely to be immediately waiting for the results. This work may have been requested by the user or initiated automatically, and often operates at user-visible timescales using a non-modal progress indicator. For example, periodic content updates or bulk file operations, such as media import.
4. *background*: Used for work that is not user initiated or visible. In general, a user is unaware that this work is even happening. For example, pre-fetching content, search indexing, backups, or syncing of data with external systems.
5. *default*: Indicates no explicit quality of service information. Whenever possible, an appropriate quality of service is determined from available sources. Otherwise, some quality of service level between *userInteractive* and *utility* is used.

Second, **the *main* queue is the one where interface updates take place.** If any heavy task is added to the *main* queue, UI updates will look slower. If this task is very heavy, the UI could freeze and the screen won't respond to any user interaction.

Last, **closures receive the Operation Queue where they are going to be executed as a parameter.** Heavy math will be placed in a background queue and charts updates will be placed in the *main* queue.

Now that we understand the basics of concurrent programming, we need to update the app.

1. We declare an *OperationQueue* object, as in Figure §12.1.

```
let queue: OperationQueue = OperationQueue()
```

Figure 12.1: Declaration of the new queue.

2. We pass the new queue to the closure as a parameter. All the code inside will be executed in this queue, as in §12.2.

```

func startRecordData(){
    guard motionManager.isDeviceMotionAvailable else { return }

    initializeStoredData()

    motionManager.startDeviceMotionUpdates(to: queue) { (data, error) in
        if let data = data {
            self.updateStoredData(data)
        }
    }
}

```

Figure 12.2: The closure is set to run on the new queue.

3. We add the UI updates methods to the *main* queue. This means heavy math will be executed in the background but charts updates, as in §12.3.

```

OperationQueue.main.addOperation {
    self.reloadGraphs()
}

```

Figure 12.3: Update charts in the main queue.

With this simple changes, we are setting our code to work concurrently.

12.3 PERFORMANCE IMPROVEMENTS

These changes made an impact in performance. Previously, I was doing both math and charts updates in the main queue. This caused the *main* queue to be full of workload, as in §12.4.

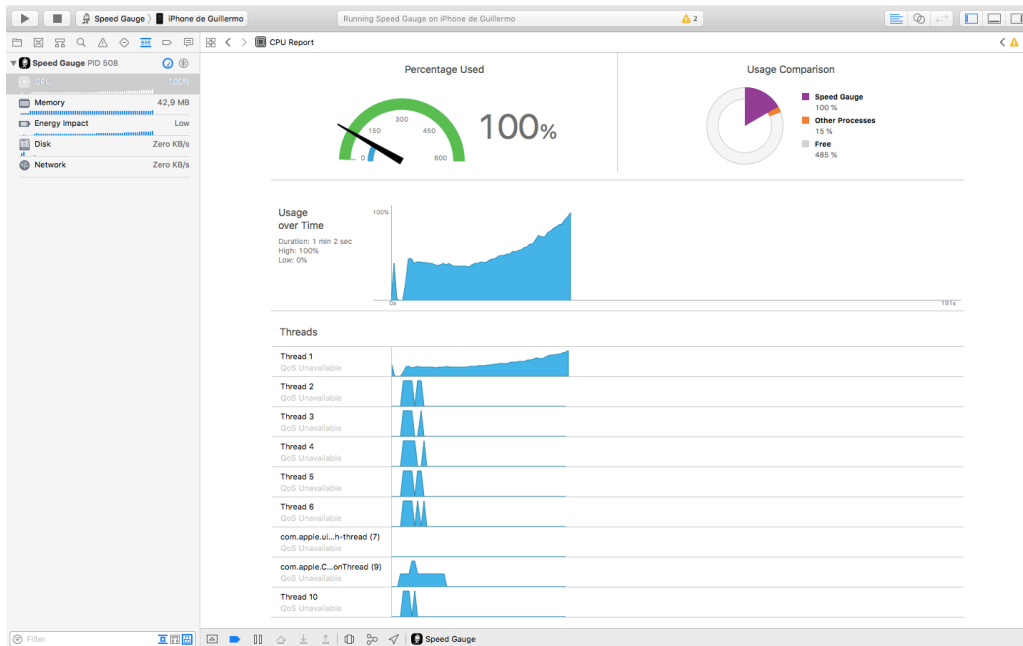


Figure 12.4: Non-concurrent application performance after a minute running.

However, I found that I was updating all 9 charts every tenth of a second. To gain a bit of performance, I decided to update only the graphs I was displaying. With these changes, I could reduce the CPU workload, as in §12.5.

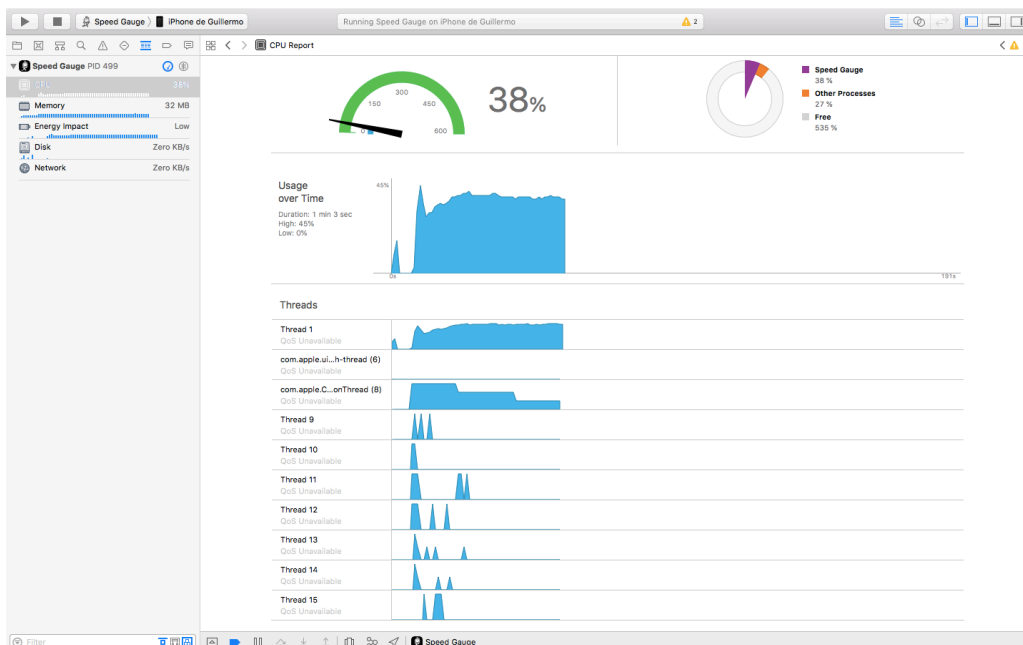


Figure 12.5: Concurrent Application performance after a minute running.

ITERATION #8: IMPLEMENTATION OF THE FILTER IN THE APP

Life's under no obligation to give us what we expect.

Margaret Mitchell (1900–1949),

Novelist

This chapter includes how I initialize the Kalman filter matrices and the reason why it didn't work out. Section §13.1 explains how I could reduce the complexity of the problem, Section §13.2 displays the initialization of the Kalman Filter used in that version of the project and Section §13.3 explains the results after the filter was implemented in my app.

13.1 FROM A 6X6 TO A 2X2 MATRIX

Although I could improve the performance of my code, the Kalman Filter is full of matrices, thus, we need to multiply, transpose and inverse a lot of data. Both multiplication and inversion are mathematical operation with a time complexity of $O(n^3)$, so reducing the size of the matrix would improve the performance drastically.

In this project, I need to get velocities from accelerations. Both acceleration and velocity are defined by 3 axes; X, Y and Z. In the Kalman Filter implementation, I would need to work with 6x6 matrices. However, **I don't need to compute the six component (3 axes per magnitude), only the acceleration and velocity in the gravity axis.** With these changes, I can reduce the workload of the app.

13.2 KALMAN FILTER INITIALIZATION

The most difficult part of the Kalman Filter is the initialization of the different matrices. Next, I will explain the reason behind the values of every matrix:

$$x = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (13.1)$$

The state variables represent the value of the magnitudes we measure. We consider that the device is at rest, so both acceleration and velocity are zero.

$$P = \begin{bmatrix} 16 & 0 \\ 0 & 16 \end{bmatrix} \quad (13.2)$$

The state covariance represents the covariance of the variables we measure. For the first iteration we define it as an upper bound. I consider that the maximum velocity and acceleration we can measure will be roughly 12 meters per second. This is 3σ . As the diagonal of the matrix is σ^2 , we get that $\sigma^2 = 16$. The rest of the matrix is set to zero as we don't know what is the covariance between the acceleration and the velocity.

$$F = \begin{bmatrix} 1 & t_increment \\ 0 & 1 \end{bmatrix} \quad (13.3)$$

The transition matrix is the one used in the examples. We have a magnitude and its derivative so, it remains the same.

$$Q = \begin{bmatrix} 0 & 0 \\ 0 & \sigma_{sensor} \end{bmatrix} \quad (13.4)$$

The process covariance is set with zeros except the lower right element. That element corresponds with the variance of the sensor, as there is no datasheet of the Apple processor, we don't know the exact value to choose.

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad (13.5)$$

$$u = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad (13.6)$$

Both B and U are set to zero because there are no control inputs. A control input would be a lever that modifies the acceleration but in our problem, there is no such thing.

$$H = \begin{bmatrix} 0 & 1 \end{bmatrix} \quad (13.7)$$

H is the measurement matrix, thus, the matrix that we use to compute the residual. As we get accelerations from the sensor and they are placed in the second row of the matrices, the first row should be a zero and the second one, a one.

$$R = \begin{bmatrix} 0.01 \end{bmatrix} \quad (13.8)$$

Finally, R is the measurement noise. As the variance of the sensor at rest always lays between ± 0.05 , I considered 0.01 a good start.

13.3 FILTER RESULTS

After I set the matrices, I run a few tests. Unfortunately, the results were not what I expected. The graphs in §13.1 showed almost the same results as was previously getting.

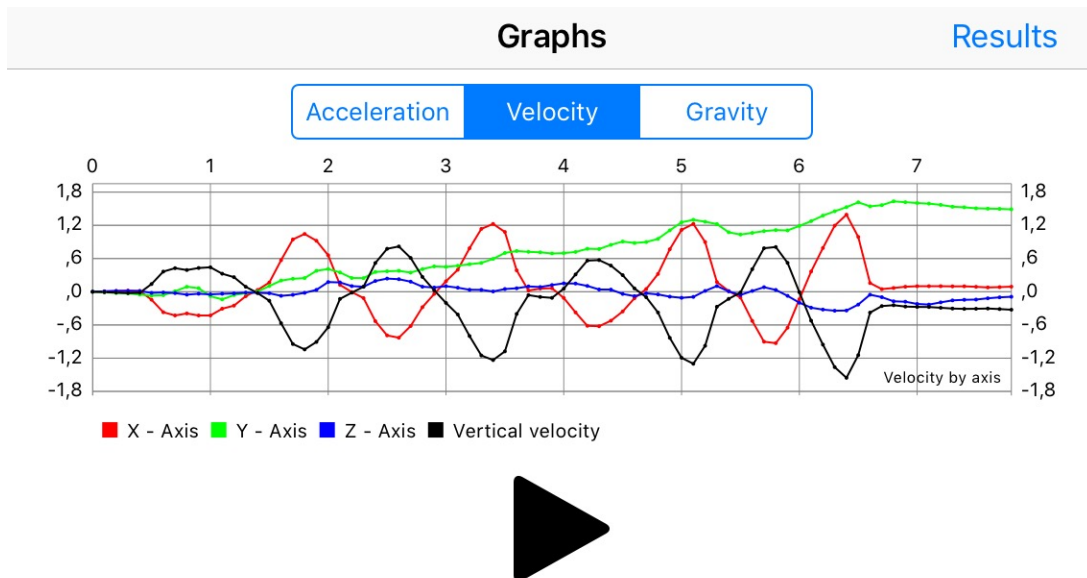


Figure 13.1: Result after the implementation of the Kalman Filter.

After further study I can understand why this happens:

1. **I don't have enough information about the sensor.** Apple does not deliver any information about its sensors, so I have no way to initialize the filter with the right values.
2. **The *userAcceleration* we get from the sensor is an already filtered using a Kalman-like filter** so, the data cannot be further improved with this kind of filters.
3. **I don't have any control input that could improve the filter**, so I have to rely on the means and covariances I consider to be closest to the real ones.

After these results, I needed to reconsider how I was going to treat my data. In the next chapter you will find how I get better estimates without using the Kalman filter but applying the knowledge I have gain with its study.

ITERATION #9: DATA TREATMENT

Give me a place to stand, and a lever long enough, and I will move the world.

Archimedes (287 BC–212 BC),

Mathematician

This chapter includes how I found a way to constraint the problem to a point where I could get useful data. It also includes the treatment applied to the data. Section §14.1 explains how a real accelerometer works, Section §14.2 explains what's the problem with accelerometers, Section §14.3 develops the use of vector projections to calculate the vertical velocity, Section §14.4 explains how I finally fixed the drift of the integrated velocity, Section §14.5 explains how I studied the fixed data in order to get the information I wanted and Section §14.6 talks about the update of the app where I finally the display the results.

14.1 YOU GET FORCE, NOT ACCELERATION

If you recall from the chapter 8, there was a section called "You get acceleration, not velocity". There, I explain that I thought I was going to be working with velocity values and not with accelerations. I argued that thinking in terms of acceleration was something anybody does in their day-by-day basis.

After getting those bad results from the implementation of the Kalman Filter and having a tutorial with Alberto, he suggested **I should search information about IMUs**. At that point I finally understood how an accelerometer works and how they measure "acceleration".

IMUs (Inertial Measurement Units) are electronic devices that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surrounding the body, using a combination of accelerometers and gyroscopes, sometimes also magnetometers. We could agree that I was working with an IMU.

As I was using the accelerometer, I studied how they work in real life. As I did with the Kalman Filter, I will explain how they work the same way I learned it. This information can be found using this link [24].

When thinking about accelerometers it is often useful to image a box in shape of a cube with a ball inside it, as seen in figure §14.1.

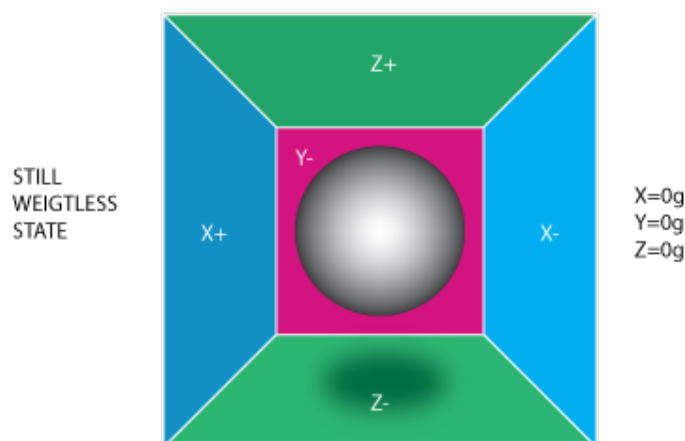


Figure 14.1: A representation of an object with no gravitational field.

If we take this box in a place with no gravitation fields or with no other fields that might affect the ball's position, **the ball will simply float in the middle of the box.**

From the picture above you can see that we assign to each axis a pair of walls (we removed the wall Y+ so we can look inside the box). Imagine that each wall is pressure sensitive. If we move suddenly the box to the left (we accelerate it with acceleration $1G = 9.8m/s^2$), the ball will hit the wall X-. **We then measure the pressure force that the ball applies to the wall** and output a value of $-1G$ on the X-Axis, as seen in the figure §14.2.

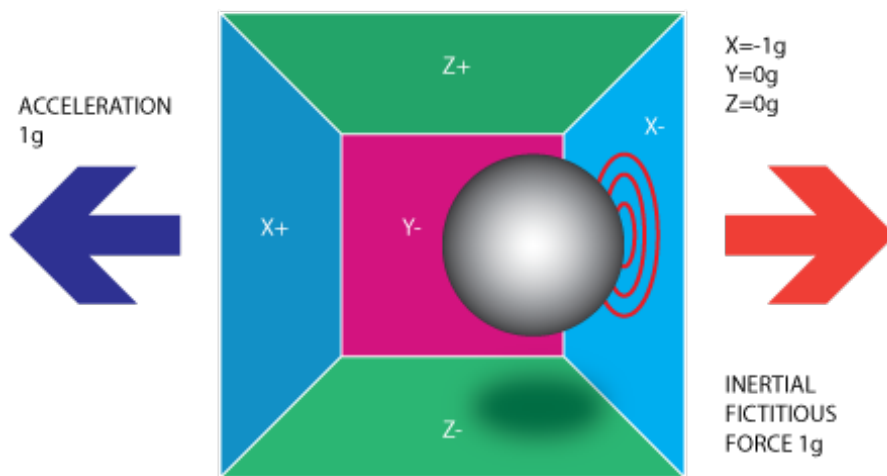


Figure 14.2: A representation of an object moving with an acceleration of $1G$.

Please note that **the accelerometer will actually detect a force that is directed in the opposite direction from the acceleration vector.** This force is often called Inertial Force or Fictitious Force. An accelerometer measures acceleration indirectly through a force that is applied to one of its walls (according to our model, it might be a spring or something else in real life accelerometers). This force can be caused by the acceleration, but as we'll see in the next example it is not always caused by acceleration. This is the reason why I was getting my charts inverted. When I was doing a movement in straight up, the velocity used to be negative. I wasn't getting wrong values, I was just getting the inertial force that the sensor was detecting.

If we take our model and put it on Earth the ball will fall on the Z- wall and will apply a force of $1G$ on the bottom wall, as shown in the picture §14.3:

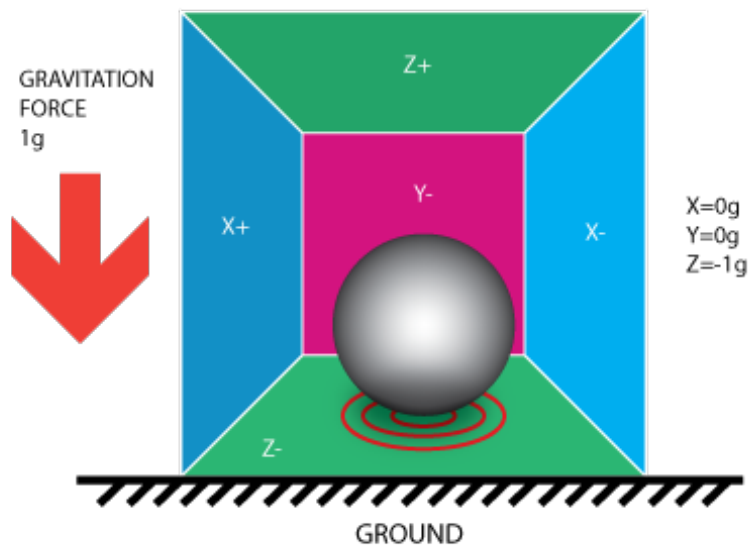


Figure 14.3: A representation of an object under the effect of the gravity.

In this case the box isn't moving, but we still get a reading of $-1G$ on the Z-Axis. The pressure that the ball has applied on the wall was caused by a gravitation force. In theory, it could be a different type of force – for example, if you imagine that our ball is metallic, placing a magnet next to the box could move the ball so it hits another wall. This was said just to prove that in essence **accelerometer measures force not acceleration**. It just happens that acceleration causes an inertial force that is captured by the force detection mechanism of the accelerometer.

While this model is not exactly how an accelerometer sensor is constructed, it is often useful in solving accelerometer related problems.

So far we have analyzed the accelerometer output on a single axis and this is all you'll get with a single axis accelerometers. The real value of three-axis accelerometers comes from the fact that they can detect inertial forces on all three axes. Let's go back to our box model, and let's rotate the box 45 degrees to the right. The ball will touch 2 walls now: Z- and X- as shown in the picture §14.4:

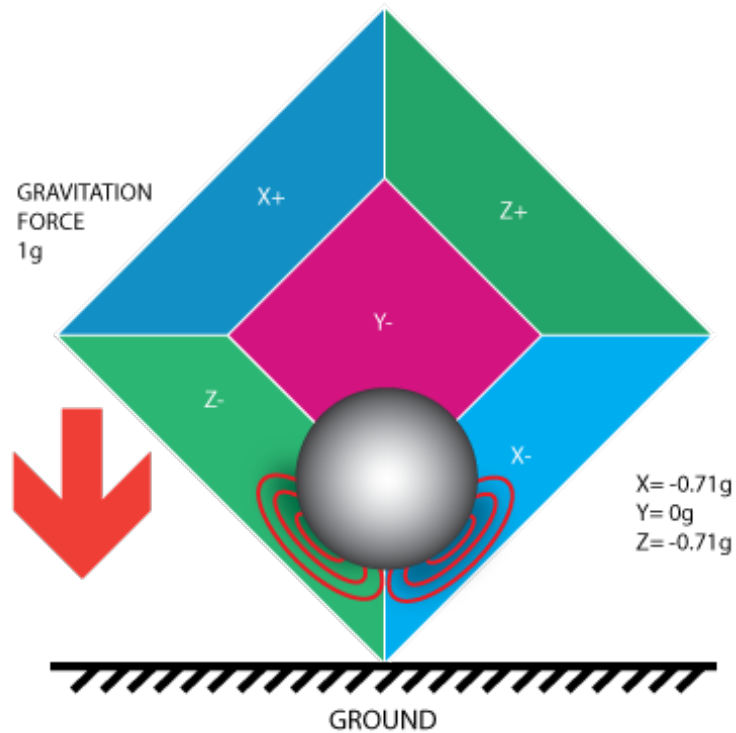


Figure 14.4: A representation of an object under the effect of the gravity where the force affects two axes.

The values of 0.71 are not arbitrary, they are actually an approximation for the square root of $1/2$. Every force applied follows the Pythagorean theorem, thus, if we name our vector R , and the projection of R is every axis, R_x , R_y , R_z we have that:

$$R^2 = \sqrt{R_x^2 + R_y^2 + R_z^2} \quad (14.1)$$

If we use this equation with the data of the last image, we get this:

$$1 = \sqrt{\sqrt{1/2^2} + \sqrt{0^2} + \sqrt{1/2^2}} \quad (14.2)$$

We will use this theorem in future sections to obtain the vertical acceleration and velocity of the device.

14.2 THE PROBLEM WITH ACCELEROMETERS

IMUs gather many sensors for a reason. **Accelerometers are good measuring inertial force, but they are not that good with the rotation of the device.**

As you have the force vectors, you can measure the rotation of the device computing the angle between t and $t+1$ vector with the gravity vector. If we do this we see that, during a rotation, the graph presents a lot of noise.

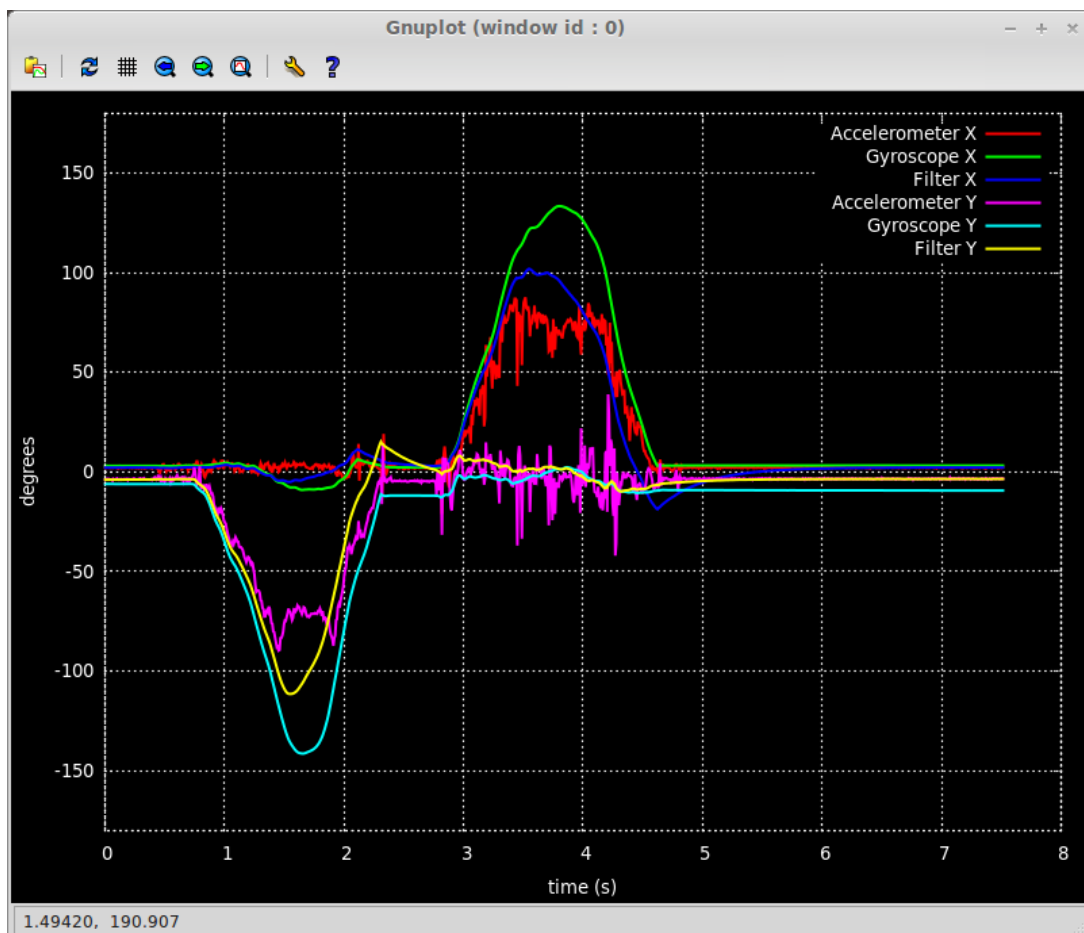


Figure 14.5: A representation of the rotation of the accelerometer [13].

Gyroscopes, in the other hand, are very efficient detecting rotation changes but, they drift over time due to the integration of the angular velocity to get angular positions.

Some authors use a filter called the Complementary Filter that mixes the accuracy of the accelerometer tracking forces and the accuracy of the gyroscope tracking rotations.

Unfortunately, **due to the extensive study of the Kalman Filter, there was little time left so Alberto and I decided to keep the gyroscope study aside** and focus on how I was going to get the vertical velocity from the sensor and treat the data after we got the entire sample and not in real time.

As this is a personal project that I decided to implement as my final project, the study of the gyroscope would be an interesting improvement because I could get better estimates and, if they are good enough, I could do the data treatment in real time. As for now, I will only use the accelerometer.

14.3 USING VECTOR PROJECTIONS

As I wrote in previous sections, in this project I don't need to compute the velocity of the device itself, **I only need the vertical velocity**. By vertical, I mean the velocity of the device parallel to the gravity vector.

With this constraint, I need to define the amount of *userAcceleration* (remember that *userAcceleration* was the name of the acceleration that the user is giving to the device) that is parallel to the gravity.

For example, **if a move the device straight up, the full inertial force will be parallel to the gravity**. If I move it from left to right, the force is perpendicular it. However, if I move the device from the bottom left to the top right, part of the force is used to lift the device, to face the gravity. **This force is the one we want to take account, not the one used to move it from left to right**.

How do we get the force in the gravity vector? Using the projection of the *userAcceleration* onto the gravity vector. For this we will use this formula:

$$proj_{\vec{a}}\vec{b} = \frac{\vec{a} * \vec{b}}{|\vec{a}|^2} * \vec{a} \quad (14.3)$$

With this formula, we obtain the projection onto the gravity. Although the gravity vector is always perpendicular to the floor, the system of reference is not the Earth but the device, so this delivers a 3 component vector. Now, we need to compute the module of the vector to know the amount of force applied. The only problem left is the fact that modules only deliver positive values (remember that we are using the Pythagorean theorem so, it will be positive). What happens if we are moving the device

straight down? We would receive the same value as if it goes straight up. The thing is that we already have the answer.

If you recall the previous equation, we multiply both vectors. This multiplication is called the Dot product [22]. This product delivers a sole value as we are multiplying every component on each vector with its counterpart in the other one, thus, we multiply the value of the X-Axis force in both vectors plus the product in the Y-Axis plus the product of the Z-Axis. **The important part of this is the sign of the Dot Product.** If the sign is positive, that means that both vectors are in the same direction; if not, they are in opposite direction.

In our project, this could be resumed this way: **If the Dot product is positive, the projection points to the floor. If it's negative, it points to the ceiling** as gravity always points to the floor.

This way I could get the vertical acceleration (and vertical velocity via integration) of the device thanks to the gravity as a point of reference. As I said in previous chapters, accelerometers are noisy and with the integration, some drift is expected up to this point. Next section will explain how this was reduce thanks to further constraints added to the system.

14.4 FIXING THE DRIFT

Drift is the main problem I've found in this project. As I don't have a way to measure the velocity but the sensor itself, the only way I've found to verify it's working right is moving the device, stop it and verify that the final velocity comes back to zero.

That's doesn't happen in normal conditions. As I have said earlier, accelerometers does not track rotations with much precision and values are biases when they are in the axis facing gravity. When you move the device in a straight line, the values are more likely to come back to zero, as we can see in Figure §14.6.

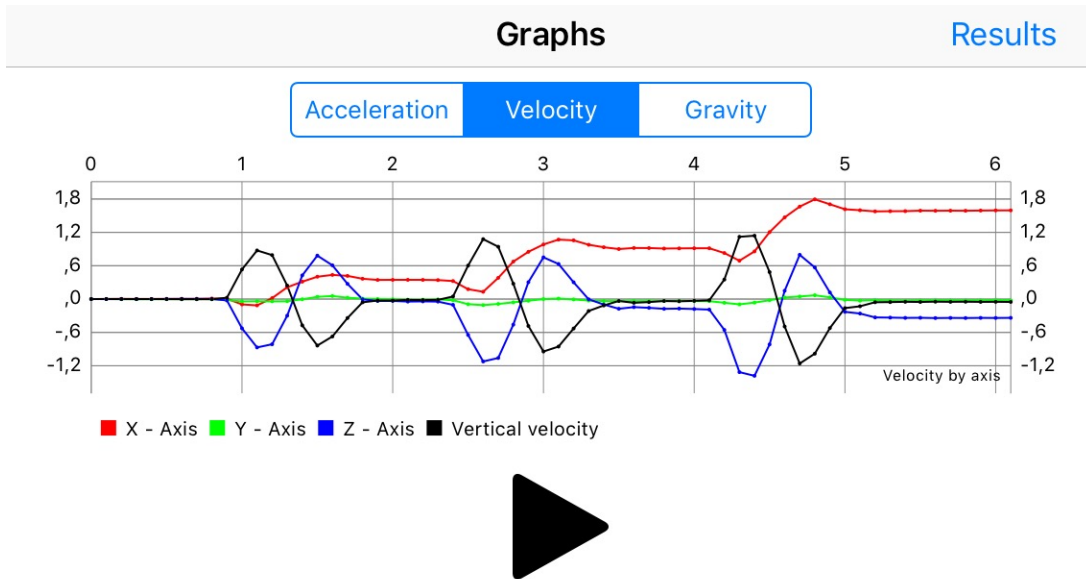


Figure 14.6: Example where, after a bottom-up movement, we get the vertical velocity close to zero.

However, when there is a movement involved, the drift is way bigger as the bias is present in every axis, as seen in §14.7. Remember, in this picture, vertical max velocities are around 1 m/s . A drift of a 10% from the real final value is considered far from being right.

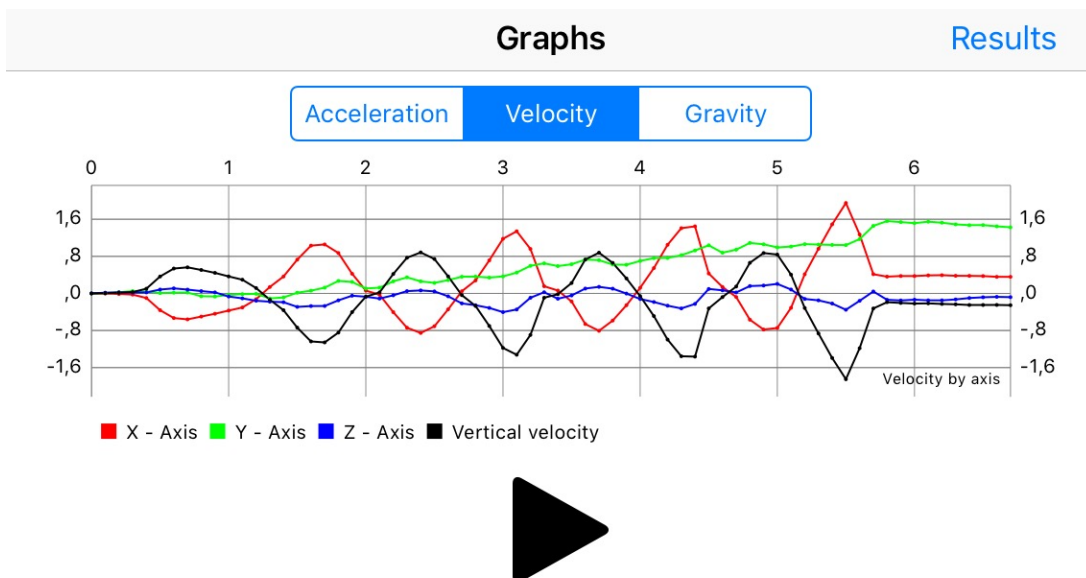


Figure 14.7: Example where, after a press, we get the vertical velocity far from zero.

Moreover, thanks to the Kalman Filter study, **we have learned we should use all the information we have about the system.** As my project is about lifting weights, we know that the final velocity must be zero.

We know that there is a bias in the sensor where the axis that face gravity has a non-zero mean noise. This non-zero mean noise causes the integration to drift lineally, so, we could try to remove this drift lineally too.

The method I have used is done after we have all the measurements. **As I need the final velocity to correct the sample, this cannot be achieved in real time.**

- First, **we suppose the final velocity should be zero.** Then, we compute the final drift of the sample. As the drift is mostly lineal, we can remove this drift lineally.
- Second, **we calculate the slope of the drift.** With the slope, we know the amount of drift per time unit.
- Last, **we lineally subtract (or add if we have negative values) the drift from the sample.** With this method, we get pretty good results. We can see that not only the end of the sample but the stops between reps are now very close to zero.

This is the first time we get results that matches the real life behavior of the device, so our next step is to treat the data to extract the useful information.

14.5 FINDING PATTERNS

As I explained in the first chapters, this is all about measuring the velocities of every repetition during a barbell movement. Now, we have a system that tracks velocity pretty accurate but, we don't have a way to reduce thousands of values to a few that represents the velocity of every rep.

Up until this point, I have been studying the system displaying the graphs in the device but the goal is not to draw a graph in the Apple Watch but to display a few values that summarize all the information of the graph.

Now, **we must study how we are going to distinguish what values belong to every rep.** To do so, we have to think about what we have right now:

1. We have a filtered sample that, at rest, shows values very close to zero, but not zero. That means that **we don't get an array of zeros at rest, we get very different values very close to zero**, some positives and some negatives.
2. We know that **the concentric phase of the lift happens when the velocity is bigger than zero for a time frame**. If we get a positive value and a negative value right next, with a sample frequency of 100Hz, that is not a rep.
3. We know that the **concentric phase of the lift ends when**, after a significant time frame, **the values come back to zero**.
4. Elite lifters, working with max weights display a bar speed close to 0.16 m/s and more novice lifters, 0.3 m/s . That means that **we are not likely to move the bar at 0.005 m/s** .
5. As the human body is a complex set of levers, there are sticking points where the bar speed will go down. That means that **max velocity is not useful by its own. We need the mean velocity to contextualize the information**.

With this information, I have designed the next algorithm to detect repetitions:

1. We declare a few variables that will store the starting and ending time spots of the reps, the maximum velocity of the current rep and the max and mean velocities of every rep.
2. As we won't get zeros at rest, **we will assume that every value between -0.1 m/s and 0.1 m/s is considered to be zero**. This can be assumed due to the fact that we are not going to move the barbell slower. This can be understood as a lower bound.
3. If the current value is bigger than zero and the max velocity of the rep is also zero, that means that we are in the starting point of the rep. Also, if the current value is bigger than the maximum value, we update the latter. With this we will get the peak velocity of the rep.
4. If the current value is zero and the maximum velocity of the rep is not zero, we have the ending point of the rep. Before we assume this is a valid rep, we need to check if the time frame is bigger than a threshold. We assume that no repetition is going to last less than 0.3 seconds. With this decision, the system won't track either the drift data if any or little movements of the lifter like the rebound after

ending a rep (if you are squatting at 0.8 m/s and you stop suddenly, it's very likely that you won't brake perfectly, the bar might bound a little and this bounce could be registered as a rep)

5. If the rep is not considered valid, the last starting and ending points are removed and the maximum velocity reset. If it's valid, we compute the mean velocity in this time frame.

14.6 DISPLAYING RESULTS IN SPEED GAUGE

Now that we have useful information, we need to display it. As my sample app is full of graphics in its main view, I will need to create **a separate view where this information will be displayed.**

The way we will present the information is using a *TableViewController*. This is a class that inherit from *ViewController* (the one used in the main view) that displays the information as a list. Every iOS app that display any kind of information in a table uses this class.

By using this class we can access to another view, but we need to pass data from one view to the other:

1. We need to declare a prepare function where we are going to get data from one view and pass it to another.
2. We need to declare the variables that will contain the past information from the original view in the destination view.
3. Once we have both, we just need to assign the information from the source to the destination variables. Once we access the new view, the information of the source view will be available.

Now, we need to display the data in the table. Thanks to the *TableViewController* class there is some boilerplate code that let us to display it. The way I will be going to display this information will be with one section per rep and a row per section to display the information.

This would be way the information will be given to the user.

← Graphs	Results
Repetition 1	
Max (Mean) velocity	1.18 (0.73) m/s
Repetition 2	
Max (Mean) velocity	1.29 (0.80) m/s
Repetition 3	
Max (Mean) velocity	1.19 (0.76) m/s

Figure 14.8: Figure of the table view in the app.

Finally, we have a full app that computes and display the information we want. Next chapter will be about the development of the Apple Watch app as the final step of the project.

ITERATION #10: ATLAS

If you don't know, the thing to do is not to get scared, but to learn.

Ayn Rand (1905–1982),

Novelist

This chapter includes the final development of the Apple Watch app, the problems I have found and how I fixed them. Section §15.1 develops the changes I needed to make to port the app from iOS to watchOS, Section §15.2 explains how I had to include Healthkit in my app, Section §15.3 explains how I had to deal with the watchOS life cycle of the app and Section §15.4 displays the final features of the app.

This will be the last iteration of the development. We have understood how the accelerometer works, how to deal with external libraries, we've learned how to work with real-time data and how to filter it (unless I couldn't use it in the final version) and we have understood that we should implement all the information we have about a system in order to improve our dataset. Now it's time to apply all the study and research in the final product: the Apple Watch app that tracks velocities.

15.1 FROM IOS TO WATCHOS

All previous study of the accelerometer has been done using an iPhone. This presents two main problems:

1. I'm not using the device as it will be used in the final version. I cannot tie the phone to my forearm as I will do with the Apple Watch, so **I could face that my app does work as expected when I tested it in a real environment.**
2. As I'm using a completely different device, I must port my code from iOS to watchOS. The language used is the same (Swift) but **the project organization and life cycle of the app are different**, so I will need to make some changes to my app.

In order to follow the development, we will treat the second problem, first. I cannot test the app as I will use it if I haven't developed it yet.

watchOS was initially a collection of companion apps, thus, you develop an app for iOS and then, you add an extension for the Apple Watch. This means that if don't have an iPhone nearby, the watch was kind of useless.

Fortunately, that changed a couple versions back. **Now, you can develop fully operational standalone apps.** Thanks to this change, the app will be used without the need of any other device but the watch.

However, this initial decision caused that an watchOS project still includes an iOS bundle. Moreover, the way you install the app in the watch is; first, you install the iOS bundle and then, the bundle installs the app in the watch. The watch is not completely independent but, for the purpose of the project, that's enough independence.

Let's talk a little more about the "watchOs bundle". In fact, **the bundle is made of two independent targets: the watchOS app and the watchOS app extension**, as seen

in §15.1.

The watchOS Watchkit app includes the interface of the app, thus, the storyboard we will use. One of the purposes of the development was the will to learn how to program views programatically but I later decided to use storyboards because external library rarely offer any support on how to code the interface. Unexpectedly, this decision paid dividends. watchOS apps does not use coded interfaces, so Storyboards are obligatory.

The watchOS Watchkit app extension included the logic of the app. The default *ViewController.swift* file is called here *InterfaceController.swift* and the iOS *viewDidLoad* method is known as the *awake* method.

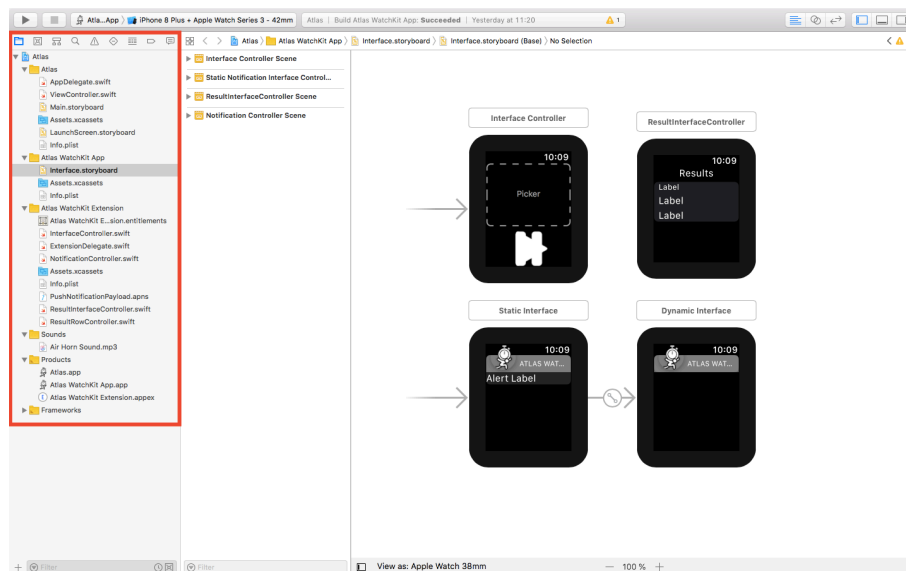


Figure 15.1: Project organization in an Apple Watch app.

This was one of the first changes I faced during the port. I wasn't sure how much different an iOS project and a watchOS project were, but I was sure I would need to do a few changes. Next problem is something I never thought about and caused several problems during the last stage of the development.

watchOS was thought to be a companion, thus, I look at it for a few seconds to read a message or pick up a call but it was not designed to be used for longer periods of time. That's the reason why the foreground app goes to background when the screen turns off. At first, it does not seem much problem but **what happens when neither the sensors nor the audio work by default in background mode?** To address this issue I had to include the Healthkit framework.

15.2 INCLUDING HEALTHKIT

I don't want to sound repetitive but the Apple Watch was designed as a companion for the iPhone, however, I want to emphasize the "was". From watchOS 3.0, the watch supports standalone apps. There must be a way to keep the app awake even if the display goes dark.

For this purpose, Apple, with the Healthkit framework, introduce the concept of workouts. **During a workouts sensors wouldn't turn off with the screen and more sensors would become active**, such as calorimeter or the heart rate sensor. We won't use these sensors as we only need the accelerometer, but we will take advantage of this feature.

For this to work, we need to define what is a workout and when do they start and end. To define a workout we will need to define the kind of activity and its location, in this case, we use *functionalStrengthTraining* and *indoor*. Then, we will create a workout and we tell the framework when it will start and end. The code used can be found in the figure §15.2.

```
func initializeWorkout(_ start: Bool){
    // Initialize Health Store object
    guard HKHealthStore.isHealthDataAvailable() else { return }
    healthStore = HKHealthStore()

    // Initialize configuration
    let configuration = HKWorkoutConfiguration()
    configuration.activityType = .functionalStrengthTraining
    configuration.locationType = .indoor

    do {
        let session = try HKWorkoutSession(configuration: configuration)
        session.delegate = self
        if start == true {
            healthStore?.start(session)
        } else {
            healthStore?.end(session)
        }
    } catch {
        // Handle errors.
    }
}
```

Figure 15.2: Code that implements how I start a workout in my app.

As workouts can be more energy demanding for the device, I have chosen to **start**

a workout when we press the start button and stop when we press the pause button. That's not really a workout in Sport Science terms but as I only need the sensors to be awake during the set of repetitions, this seemed a good choice.

15.3 TURNING BACKGROUND TO FOREGROUND

Unfortunately, this wasn't the only problem. **Healthkit takes care of the sensors but not the audio.** In my app I have implemented a timer. As you will have your hands on the bar, there is no way to press a button without altering the data. This timer let you choose from a default set of delays to add and the app itself will notify you with a sound.

This sound comes from the library *AVFoundation* (also used in iOS) and it presents the same problem as the sensors. **To address this problem it's necessary to set the current audio session to active.** Just with that change, the app will play audios even if it's in the background. The code used can be found in the figure §15.3.

```
@objc func playSound(){
    let audioSession = AVAudioSession.sharedInstance()
    do {
        try audioSession.setActive(true)
        try audioSession.setCategory("AVAudioSessionCategoryPlayback")
        let path = Bundle.main.path(forResource: "Air Horn Sound.mp3", ofType: nil)!
        let url = URL(fileURLWithPath: path)
        audioPlayer = try AVAudioPlayer(contentsOf: url)
        audioPlayer?.play()
    } catch {
        // Couldn't load file
        print("Couldn't load file")
    }
}
```

Figure 15.3: Code that implements how to make the audio work in the background.

15.4 THE RESULT

At this point we finally have a working version of the project where we can:

1. Set a timer to add a specific delay to the start of the set.
2. Play an audio to notify when the set starts.
3. Keep track of the vertical acceleration and treat the data afterward.

4. Stop the data to trigger the treatment.
5. Display the results in a table just after the data treatment. The user no longer have to press a button to access the results.

The interface of the app can be seen in the figure §15.4.



Figure 15.4: Final watchOS application.

If you want to see a video of the application running, you can access the Github repository [16] and download the two available videos.

PARTE IV

CLOSING STAGE

USER GUIDE

It just works.

Steve Jobs (1955–2011),

Businessman

This chapter explains how to use the app. Section §16.1 explains the problem to solve and the constraints I have supposed and Section §16.2 gives the instructions where I explain how to use the app.

16.1 PROBLEM TO SOLVE AND CONSTRAINTS

The purpose of this app is to record the vertical velocity of the barbell during a lift like the Squat, Press or Deadlift.

As the lift is supposed to be done in a vertical straight line, no forward or backward movement will be recorded.

This app does not require of any other hardware other than the Apple Watch itself.

16.2 INSTRUCTIONS

1. The Apple Watch must be securely tied to the wrist. If the Apple Watch moves from the wrist, the recorded velocity will not match the reality.
2. The landing view includes a timer selector and a play button.
3. The timer selector includes several preset delays, from no delay at all to 30 seconds. The user is able to choose the value of the timer using the fingertip or the Apple Watch Digital Crown to scroll through the list.
4. The play button triggers the recording of the vertical velocity. This begins after the delay. If you have chosen a delay of 5 seconds; the system will start the measurement in 5 seconds from the moment the user pressed the button.
5. As this app is meant to be used with both hands on the bar, a notification sound will be played after the delay, at the same time as the measure starts, to notify the lifter that the app has started to record the velocities.
6. The measurement can only be paused if the user presses a pause button that will replace the play button while the app is recording data.
7. Just after the measurement is paused, the system will display the results to the user in a table. Each row includes the number of the repetition and the max and mean velocity of every repetition.
8. In order to repeat the cycle, the user must press in the top left corner of the screen to return to the landing view.

SUMMING-UP

The only kind of writing is rewriting.

Ernest Hemingway (1899–1961),

Novelist

This chapter is a retrospective about the development of the project. Section §17.1 includes a post-mortem discussion about the project and Section §17.2 gives some possible improvements for future updated of the project.

17.1 POST-MORTEM REPORT

Now that we have finished the project, it's time for the lessons learned. This report is a retrospective about what things went right, what others went wrong and discuss the reasons why.

17.1.1 Things gone right

- The development of the Apple Watch app was a success.
- Improved skills and knowledge about the Apple framework.
- Improved knowledge about statistics, gaussians and bayesian probabilities.
- Positive first experience with real time data and sensors.

17.1.2 Things gone wrong

- Too much time spent in a filter I don't use in the final version.
- Lack of real time feedback of the app.
- The initial planning didn't match the real planning at all.

17.1.3 Discussion

Once I have finished the project, it's a good idea to meditate about the flaws of the development in order to avoid these in the future.

The main problem I have faced was how I was going to improve the data I was getting from the sensor. To do so, I studied during 4 months how the Kalman Filter works. I consider this was too much time spent on a filter but as I didn't know if that filter was going to be a lifesaver for my project, I needed to study it. It was way too much time, but I couldn't ditch a possible solution because "it would take too long". I think the problem is more about the time than the fact that I had to study it. In a future project, I would spend a month or two with it, but not four.

Another problem was the bad estimations of the initial planning. The initial planning was set to finish the project by April, but the project development didn't move

forward until mid-May. It too easy to say that I would try to plan better if I had to start the project again but let be honest, with no previous experience in the platform nor the problem, the initial planning should be very flexible to avoid a bad planning. I would need to study more about project management to be able to plan projects like this. Now, I lack this knowledge.

17.2 FUTURE WORK

Although the app finally worked, there are many improvements that could be done:

1. Add gyroscope data to improve the measurement. Use of the Complementary filter to mix both accelerometer and gyroscope data. In the Complementary filter we mix the long time reliability of the accelerometer with the short time reliability of the gyroscope to help the device to track better the orientation changes that affects the impact of the gravity on every axis.
2. Real-time results. Nowadays, the data correction is done after the measurement ends. If the data is good enough, this correction may not be necessary and the results could be display in real time.
3. Develop the iOS target. Now, the project is only an Apple Watch app but the iOS app could let the coach to start and stop the measurement and give feedback at real-time.

BIBLIOGRAPHY

- [1] A high-accuracy step counting algorithm for iphones using accelerometer. <https://pdfs.semanticscholar.org/0a3a/c47dff1853631ba9fed73c996c57dfdcede0.pdf>. (page 96).
- [2] An introduction to the kalman filter. https://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf. (page 96).
- [3] Chartdataset min and max values not recalculated when calling clear(). <https://github.com/danielgindi/Charts/issues/3260>. (page 57).
- [4] An introduction to feature-driven development. <https://dzone.com/articles/introduction-feature-driven>. Accessed: 2019-11-20. (page 14).
- [5] How a kalman filter works, in pictures. <http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/>. (page 96).
- [6] Implementing positioning algorithms using accelerometers. <https://www.nxp.com/docs/en/application-note/AN3397.pdf>. (page 96).
- [7] Introduction to the kalman filter and tuning its statistics for near optimal estimates and cramer rao bound. <https://arxiv.org/pdf/1503.04313.pdf>. (page 96).
- [8] Kalman filtering. <https://c.mql5.com/forextd/forum/29/kalman.pdf>, . (page 96).
- [9] Kalman and bayesian filters in python. <https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python>, . (page 61).
- [10] Kalman filtering for drift correction in ir detectors. <http://www.wseas.us/e-library/conferences/2006madrid/papers/512-452.pdf>, . (page 96).
- [11] Realtime linechart poor performance due to calcminmax. <https://github.com/danielgindi/Charts/issues/3166>. (page 56).

- [12] The extended kalman filter: An interactive tutorial for non-experts. https://home.wlu.edu/~levys/kalman_tutorial/. (page 96).
- [13] Using a complementary filter to combine accelerometer and gyroscopic data. <http://blog.bitify.co.uk/2013/11/using-complementary-filter-to-combine.html>. (pages xiii, 116).
- [14] Agencia tributaria. Tabla de coeficientes de amortización lineal. http://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml. (page 24).
- [15] AppCoda. ios concurrency: Getting started with nsoperation and dispatch queues. <https://www.appcoda.com/ios-concurrency>, December 2015. (page 102).
- [16] G. A. Gamero. Project atlas github repository. <https://github.com/guillermo-ag-95/Project-Atlas>. (pages 96, 130).
- [17] Google. Sensor fusion on android devices: A revolution in motion processing. <https://www.youtube.com/watch?v=C7JQ7Rpwn2k>, August 2010. (page 60).
- [18] Jordan Syatt. Developing explosive strength and power for athletic performance. <https://www.syattfitness.com/westside-barbell/developing-explosive-strength-and-power-for-athletic-performance/>. (page 9).
- [19] M. Israetel, J. Hoffman, and C.W. Smith. *Scientific Principles of Strength Training*. Juggernaut Training Systems, 2015. (page 8).
- [20] Ministerio de Empleo y Seguridad Social. Bases y tipos de cotización 2018. http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm. (page 24).
- [21] Oracle. What every computer scientist should know about floating-point arithmetic. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html. (page 99).

- [22] Oregon State. Dot products and projections. <http://math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/dotprod/dotprod.html>, 1996. (page 118).
- [23] Rogue Fitness. Levantadores - the basque strongman - a documentary film. <https://www.youtube.com/watch?v=vck32S27RmM>, December 2015. (page 4).
- [24] Starlino. A guide to using imu (accelerometer and gyroscope devices) in embedded applications. http://www.starlino.com/imu_guide.html, December 2009. (page 112).
- [25] V. Zatsiorsky and W. Kraemer. *Science and Practice of Strength Training*. Human Kinetics, 1995. (page 9).