Localization of a Mobile Robot with a GPS sensor, using Extended and Unscented Kalman Filters

A Thesis

submitted by the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Efstathios Rafailidis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN DATA SCIENCE AND ENGINEERING

University of Ioannina
School of Engineering
Ioannina 2025

Examining Committee:

- **Konstantinos Vlachos**, Assistant Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Aristidis Lykas**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Konstantinos Blekas**, Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

Special thank you to my family for supporting me through the years.

Acknowledgements

Thank you to the University and especially to my supervisor Konstantinos Vlachos

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Abstract

In a continuously moving forward world, automation is a necessity. As more and more trivial for humans tasks get assigned to robots, mobile or immobile ones, their proper function is crucial. The advantages of using a robot instead of a human are numerous and we go through some of them later in the thesis, but the main one will always be the casualties. Worst case scenario, an accident involving only robots will only result in material losses and nothing more, which makes is infinitely more preferable than an accident involving humans. After establishing why, we want to use robots, we need to make sure that they are functioning properly. One of the issues for mobile robots is the problem of their localization. Better localization means better path planning which means less accidents and more efficiency for both energy consumption and completing the task faster. Our thesis tackles the problem of robot localization and tries to find ways of improving it.

In this thesis we will compare machine learning and two Kalman filter variations, in order to find the best one for robot localization. We will start our report with the motive behind our work and what we are trying to achieve. In the second chapter we focus on the theory behind our methods and in the next one we present the simulation setup we used. In chapter four, we present our data, our methodology of processing them and our results, determining the most accurate method for robot localization. In chapter five, we discuss our conclusions from our work and suggest future work to further improve our findings.

Extended abstract

Καθώς ο κόσμος εξελίσσεται και οι τεχνολογικές εξελίξεις τρέχουν, όλο και περισσότερες δουλειές μπορούν να γίνουν από κάποιο ρομπότ. Από μια τεράστια πολυεθνική εταιρία αυτοκινήτων μέχρι μια τοπική εταιρία συσκευασίας προϊόντων η χρίση ρομπότ για συγκεκριμένες εργασίες βγάζει νόημα από πολλές πλευρές. Ένας βασικός παράγοντας είναι η μείωση του κόστους, ένα ρομπότ μπορεί να συνεχίζει να δουλεύει χωρίς να χρειάζεται να σταματήσει και φυσικά ένα ατύχημα που στο οποίο συμπεριλαμβάνονται μόνο ρομπότ θα έχει μόνο υλικές ζημιές και όχι τον τραυματισμό ή ακόμα χειρότερα την απώλεια κάποιου ανθρώπου.

Έχοντας ξεκαθαρίσει την χρησιμότητα των ρομπότ, θέλουμε να εξασφαλίσουμε και την σωστή λειτουργία τους. Στην εργασία που ακολουθεί ασχοληθήκαμε με ένα κινητό ρομπότ και ο στόχος μας ήταν να συγκρίνουμε ένα μοντέλο μηχανικής μάθησης με δύο διαφορετικές παραλλαγές των φίλτρων Kalman, με σκοπό να εντοπίσουμε εκείνη που θα μας έδινε την πιο ακριβή θέση του ρομπότ. Η γνώση της πραγματικής θέσης του ρομπότ, το καθιστά ικανό να σχεδιάζει καλύτερα την πορεία του με αποτέλεσμα να αποφεύγει πιο αποτελεσματικά τυχόν εμπόδια, άψυχα ή έμψυχα. Ο καλύτερος σχεδιασμός της πορείας του ρομπότ δεν έχει σαν μόνο θετικό την αποφυγή συγκρούσεων αλλά και την εξοικονόμηση ενέργειας καθώς δεν θα χρειαστεί κάνει περιττές κινήσεις όταν καταλάβει ότι είναι εκτός πορείας προκυμμένου να ξανά σχεδιάσει την πορεία του αλλά μια κίνηση χωρίς σφάλματα θα βοηθήσει στην περάτωση της αποστολής του συντομότερα.

Στο πρώτο κεφάλαιο της εργασίας αναφέρουμε πιο αναλυτικά τα κίνητρα μας καθώς και τους στόχους μας. Κλείνουμε το πρώτο κεφάλαιο δίνοντας μια περίληψη της δομής της εργασίας για την καλύτερη εποπτική εικόνα του αναγνώστη. Στο δεύτερο κεφάλαιο παραθέτουμε την θεωρία πίσω από την μηχανική μάθηση και το συγκεκρινοποιούμε στην παλινδρόμηση καθώς εκείνη είναι η μέθοδος που χρησιμοποιούμε στην εργασία μας. Στο δεύτερο μισό του

κεφαλαίου, παραθέτουμε την θεωρία για τις δύο παραλλαγές των φίλτρων Kalman που χρησιμοποιούμε, το unscented και το extended Kalman φίλτρο. Στο τρίτο κεφάλαιο, αναφέρουμε όλα τα κομμάτια από τα οποία αποτελείτε η διάταξη μας. Στο τρίτο κεφάλαιο ο αναγνώστης θα βρει επίσης και μια περιγραφή των χαρακτηριστικών του ρομπότ που χρησιμοποιήσαμε κατά τις προσομοιώσεις μας. Στο κεφάλαιο τέσσερα, ξεκινάμε με το πώς συλλέξαμε τα δεδομένα μας και τα επεξεργαστήκαμε ώστε να μοιάζουν με πραγματικά δεδομένα που θα λάμβανε κανείς από ένα GPS. Το κεφάλαιο συνεχίζει με τα αποτελέσματα από τις τρεις μεθόδους δίνοντας στον αναγνώστη την απάντηση για το ποια από τις τρεις μεθόδους είναι πιο ακριβής. Το τέταρτο κεφάλαιο, κλείνει με τα πειράματα που κάνουμε εκτός του προσομοιωτή, με σκοπό να βεβαιωθούμε ότι τα φίλτρα Kalman λειτουργούν σωστά. Το τελευταίο κεφάλαιο περιέχει τα συμπεράσματα μας και μελλοντική δουλεία που μπορεί να γίνει για να συνεχιστεί αυτή η εργασία.

CHAPTER 1

INTRODUCTION

1.1 Motives

The motive behind this thesis is the desire to have a robust and accurate method for better mobile robot localization. Nowadays, more and more everyday tasks can be performed by unmanned robots, especially tasks in places where no humans are expected to be, so the chances of an accident are low. Humans have been using mobile robots for various reasons instead of them manually performing these tasks.

The benefits of robots in place of human labor are numerous. First and foremost, we have safety. Robots can be sent and work in environments that are not ideal or are even dangerous for humans, for example from a vast desert or the depths of an ocean to a radioactive region like Chernobyl. Additionally, thanks to 21st-century technological advancements the possibility of life in space becomes more and more viable. With today's technology, a walk on Mars's surface for a human being would be impossible, but that is not the case for a mobile robot.

The second reason is costs. Trivial tasks like night-watching could be easily performed by a mobile robot equipped with a camera and some image recognition software. The modularity that mobile robots offer nowadays makes them a compelling option for reducing costs. Having one robot that, depending on its

equipped modules, can perform a few different tasks simultaneously is not something that big companies can ignore. Companies already use robots for heavy labor work, freeing up human talent to perform more elaborate, less intensive and less risky tasks.

Having said all that, a mobile robot needs to know its position. Localization for a human might be a mundane task, but that is not the case for a robot. Mobile robots must know their position to plan their next move. The precision needed depends heavily on the nature of the task. Night-watching in a huge empty-of-humans factory might not require millimeter-precision movements, but a maneuver on Mars's surface could cause the whole billion-dollar operation to go to waste.

1.2 Thesis goals

Our goal for this thesis is to compare different Kalman filters to a machine learning model in order to identify the better method for robot localization. Behind our motives there is the need for mobile robots to precisely know their pose; pose consists of their coordinates and their orientation. The knowledge of their precise pose will help the robot in a few different ways.

Firstly, there is the navigation. In the last few years, motor companies have been trying to implement autonomous driving for vehicles, ranging from city cars to enormous trucks carrying tens of tons of cargo daily. Localization becomes essential because an autonomous vehicle requires both its pose and its target's location so as to be able to carry out its path planning.

Next up, there are workplaces where traditional localization methods do not work. For instance, IMU localization would be unsuccessful in places where the terrain could cause the wheels to slip. In these scenarios, a robust and accurate method of localization is needed.

Lastly, the efficiency of a mobile robot, both time efficiency and energy consumption efficiency, can add up during the whole lifespan of the robot. Better path planning can help the robot reach its destination faster and consume the least amount of energy. These factors might not be of importance for every kind of work, but over a large period, these factors can play a significant role.

To summarize, our goal is to benchmark the Extended and the Unscented Kalman Filter against a machine learning model, so we can conclude which method would be better for our mobile differential drive robot (we will be talking in more detail about our robot in later chapters of this thesis).

1.3 Structure of the thesis

In this subchapter, we present the structure of the thesis. This thesis consists of five chapters. In the first chapter, we give an overview of the motives behind our study and what we tried to achieve. In the second chapter, we elaborate on the theory behind machine learning and the Kalman filters. About the theory of machine learning, we start with general information about the different machine learning methods, then we proceed with giving more information about the supervised machine learning method, and at the end we introduce the theory behind regression, which is what we used. After the theory of machine learning, we go on to discuss the theory behind Kalman filters. We follow the same pattern as we did with machine learning; we start with general information about the history of Kalman filters and how they started. Once we are done with the general information, we conclude with the theory about the extended and the unscented Kalman filters, which are the two Kalman filter variations we used for our experiments.

In the third chapter, we present our experimental setup. We start with information about the robot we used in our experiments. Then, we demonstrate the simulator we used in order to have a visual representation of our robot, which among other things helped us with making sure that the inputs we were providing our robot with were received and executed correctly. After we cover all the necessary points about the simulator, we also bring up the software called robot operation system (ROS), which provided us with the workspace which enabled us to communicate with our robot. Lastly, in this chapter, we put forward the way we were able to collect the data we needed for our experiments.

In chapter four, we show our results from our three different methods of predicting the current pose of our robot. We first specify a certain trajectory which the three methods will have to predict in order for us to distinguish the most accurate one. Then, we introduce the methodology we used in order to add noise

to our ground truth data. Next up, we go over one of our earlier failed attempts, where we used a different ROS topic to get noisy data and why that failed. At the end of the chapter, we present the results of all three methods used, by comparing them to the ground truth data.

After validating our methods in the simulator, we move on to some experiments in real life. In the first experiment, we explore the capabilities of the GPS module our robot is equipped with. In the next two experiments, we use the data from the GPS module as input for our Kalman filters in order to make sure that the filters are working correctly also outside of the simulator.

The last chapter is the conclusions chapter. In this section, we discuss our final thoughts to the reader based on the comparison of the three methods. Here is where the reader is called to ponder over the three methods, after seeing all the data and having all our notes before them. At the end of the chapter, we suggest some future work to be done.

CHAPTER 2

THEORETICAL INTRODUCTION

2.1 Machine learning theory

2.1.1 Machine learning introduction

In this chapter, I would like to give a theoretical introduction to the machine learning method that was used to predict the pose of our robot. Before we dive into the regression model we used in our experiment, I would like to give a short overview of what machine learning is.

As explained in the paper [1], machine learning is a subset of a broader term called artificial intelligence (AI). Machine learning is a process through which the computer is equipped with the ability to learn using data we provide in order to be able to make a decision as a human would. By creating and training a machine learning model, we enable it to comprehend complex problems and find solutions using the data it has already collected.

There are four methods of machine learning. In the following section, we will go through them, giving a short explanation for all of them but focusing more on the method we ended up using. The first method is the supervised. This is the method we used. In this method the machine learning model takes as input a data set and tries to recreate the label data. The label data contain the information we want our model to output. In our case, we used the position, orientation and input at a specific time interval as input data set while training our model, while

we used the position of the robot at the next time interval as label. More specific information about our data will be found in chapter 4, where we explain exactly what we did and show our results.

The next method of machine learning is called unsupervised. In this method, there is no label, meaning that we do not instruct our model on what the result should be, but rather it is left on its own to try and figure out possible patterns or connections inside the input data. In that sense, unsupervised learning can be used when there is no clear connection between the data, but we want the model to explore our dataset and check if it can find one. Also, the are no training data.

The second to last machine learning technique is the semi-supervised machine learning method. The forementioned method is a combination of the previous two. It is used when there are some data that contain label as in the supervised method but some of them do not, like in unsupervised machine learning. We might need to use the semi-supervised method, when getting more labeled data can be expensive or just difficult. In these cases, instead of just using unsupervised machine learning we use a combination of supervised and unsupervised machine learning.

The last machine learning method is called reinforced learning. In this method there is a start and an end. The agent is being given a positive reward when choosing the shortest way of reaching the goal. On the other hand, when making a wrong decision, it is given negative reward.

2.1.2 Supervised machine learning

In this section, we have an overview of the two different supervised machine learning types. The two types of supervised machine learning are called classification and regression. Here we provide an outline of both, and in the next chapter we focus more on regression, as this is the method we used in our study.

[2]Before we get into the two types, I would like to explain a term which is instrumental in reader's comprehension. The term I am referring to is called "feature". In the bibliography you might encounter the names "attribute", "variable" or "dimension", which are all synonyms. A feature is an individual

property or characteristic of the phenomenon that we are studying and has to be measurable, so it can be a part of the data set we use for our machine learning. Features can have either discrete values, like zip codes and boolean characteristics, or continuous values, such as temperature and height. Features can also be categorized as numeric or symbolic. Numeric features can be measured on a scale and can be directly used in machine learning. Symbolic features are discrete and can be grouped into categories, for example eye color or grades. Symbolic features usually need to be converted into numeric values so that they can be used in machine learning.

Both regression and classification are similar methods with just one main difference. Both methods use data sets in order to train them and then a different data set for testing their results. Both get a collection of features as input and try to predict the label of the data. Their main difference is that classification is used to predict discrete values as true or false, spam or not spam. In contrast, regression is used to predict continuous values such as price, age, position, and so on.

2.1.3 Regression

Linear regression is a statistical method used to model the relation between the features and the label. [3] The simplest form of regression would be the linear regression. In this case, we try to predict the label data using the dimensions, thus trying to find, as the name suggests, a linear relationship between the dimensions and the label. The mathematical representation would be:

$$y = \beta_1 * X + \beta_0 \tag{2.1}$$

In equation (2.1) y is the label, X is the dimension, β_1 is the slope and the β_0 is the intercept.

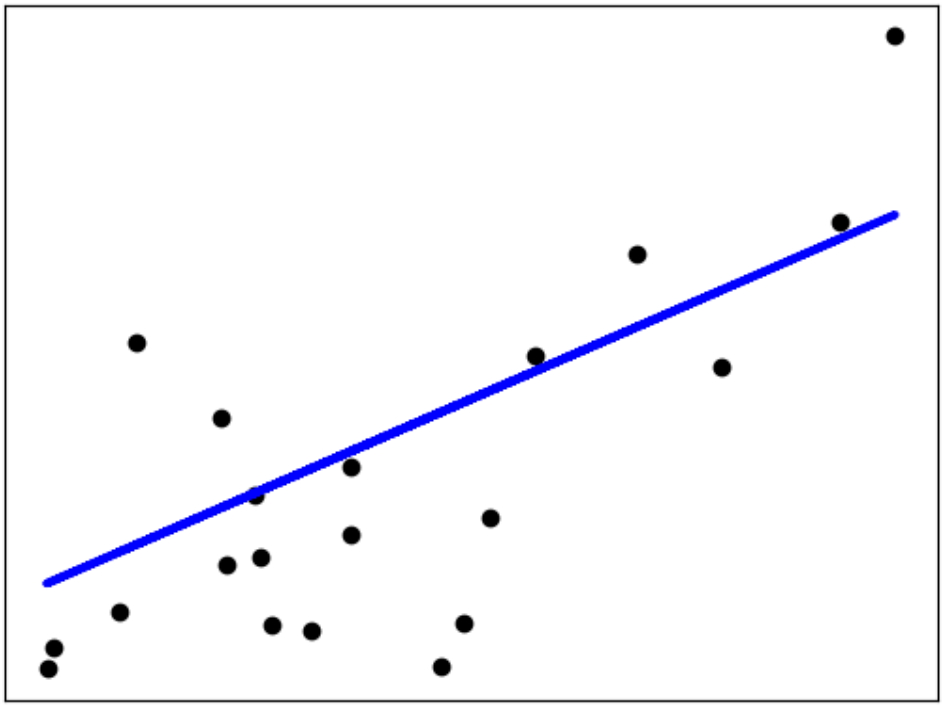

Figure 2.1: Simple example for linear regression. The black dots are the data points, and the blue line is the model's prediction.

In Figure *2.1*, we have a simple example for linear regression. The black dots present the data points, in this case we could have the x-axes be the feature and the y-axes to be the label. The blue line is the model's attempt to draw a straight line as it tries to minimize the residual sum of squares between the data points and its own prediction. Simple linear regression only uses one feature in order to predict the value of label. With linear regression, we can determine the relation between the feature and the label.

Linear regression only works well under the specific conditions we mentioned above. If for example the relationship between features and label is not linear then, linear regression would be inappropriate to use. As seen in Figure *2.2*, on the left side we have data points with no linear relationship between them, as we see that the straight line does not fit the data points well. In this example, we are using a simple linear regression model, which is a variation of linear

regression, where the user can only have one dimension as input to predict the label. In contrast, there is multiple linear regression in which you can have more than one dimension, but you still need a linear relationship between dimensions and the label. The last constraint makes even the multiple linear regression not suitable for the dataset on the right.

As we can imagine, a machine learning model that can handle nonlinear data would be useful, as there is a plethora of use cases where the dimensions and the label are not linearly connected. For example, the pricing of a house based on the different features it could have, is a use case where a nonlinear model would be used.

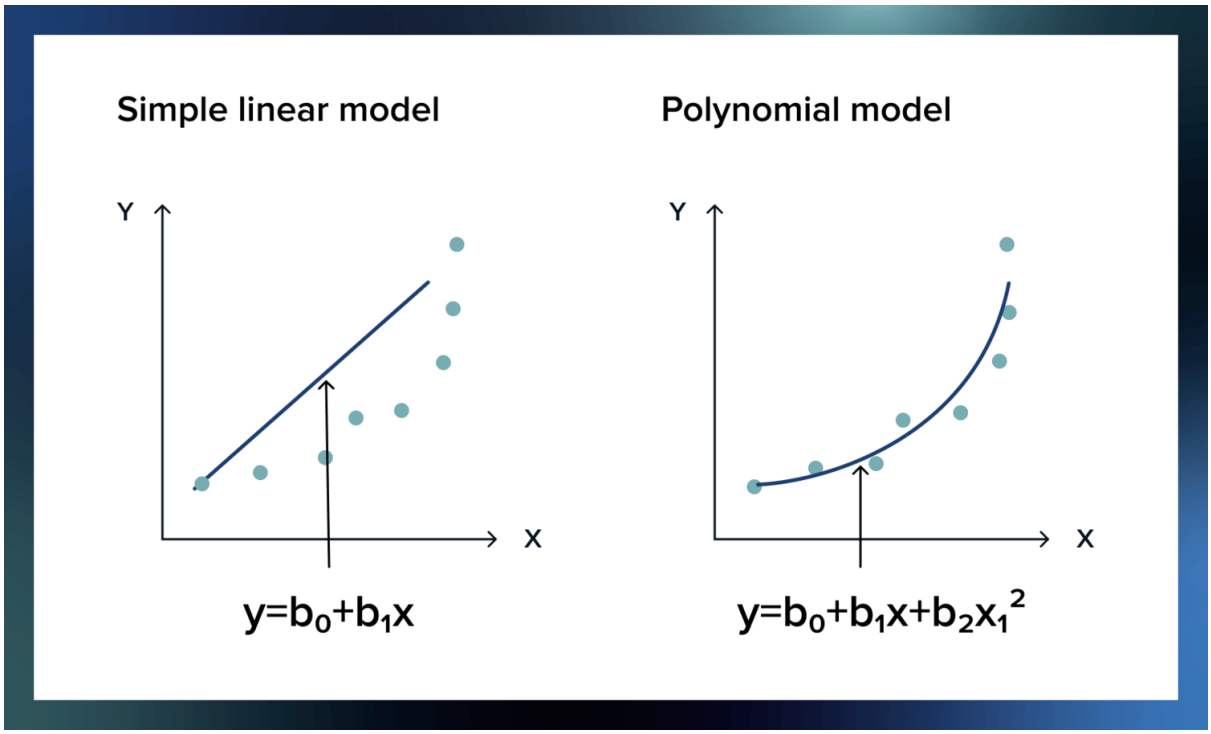

Figure 2.2: [4]Simple linear regression (left). Polynomial regression (right).

The solution to this problem comes in the form of polynomial regression. In polynomial regression the relationship between the features and the label are model based on a n^{th} degree polynomial. In our study, we were not certain about the relationship between our data, so we used this method as it is very flexible, because we can change the degree of the polynomial as we see fit. During our

experiments, we used different degrees to find the better fit. We used a mixture of data-driven decision, in which we were trying different degrees and based on the outcome we selected the one we finally used. We also used a knowledge-driven decision, in which because of the physics of the problem we could rule out high degrees for our polynomial. In the end, having combined both methods, we are certain that we used the best degree for our polynomial.

2.2 Kalman filter theory

2.2.1 Introduction to Kalman filters

In this section, I would like to introduce Kalman filters and the theory behind them. We start with the history of Kalman filters, then we offer an overview of the general mechanisms behind the Kalman filters. Lastly, we get into the specifics of the two Kalman filter variations we used in our study, the extended Kalman filter and the unscented Kalman filter.

In 1960, R.E Kalman published his paper on a new method capable of recursively computing the state of a linear dynamic system through noise in an optimal way using the mathematical formula of what would later become the Kalman filter [5]. Due to the great advancement of computational power, the Kalman filter became a subject of research and found many applications, particularly in autonomous or assisted navigation. Even in the early years of its creation, it played a crucial role in the navigation systems of the Apollo lunar mission.

One of the biggest weaknesses of the Kalman filter were the nonlinear systems. One can easily see the usefulness of a robust mathematical way for predicting the state of a nonlinear system, as there is a plethora of nonlinear systems in the real world. The breakthrough came in the 1960s in the form of the Extended Kalman filter (EKF). The EKF was able to function for nonlinear systems as it would linearize them and then apply its mechanisms. After the EKF was invented, it made the Kalman filter more applicable for real life problems. In the 1990s, the next variation of the Kalman filter would be developed, called Unscented Kalman filter. Its purpose was to overcome the limitations of the EKF, making its appeal even greater.

Before we dive into the specifics of EKF and UKF, let me give you an overview of the general mechanism of the Kalman filters. The general mechanism is the same for all Kalman filters, so it would make sense to present it here.

As a kind of high-level overview, you can think of the Kalman filter as a two-step process. Kalman filters use a form of feedback control, where the filter will predict the state of the system at a certain point in time and then it will get feedback in the form of an actual measurement. Of course, we assume that the measurement contains noise, so the filter does not consider it as the ground truth. During this study, we will be referring to these two steps as the prediction step and the update step as we see in Figure *2.3*. Every equation we will be presenting below will fall under one of the two steps. In the predict step, we estimate the state and the error covariance of the next time step, which gives us a *a priori* estimate. In the update step, where we incorporate the measurement into our estimation, we calculate a new *a posteriori* estimate[5].

This would be a rough high-level overview of what a Kalman filter does.

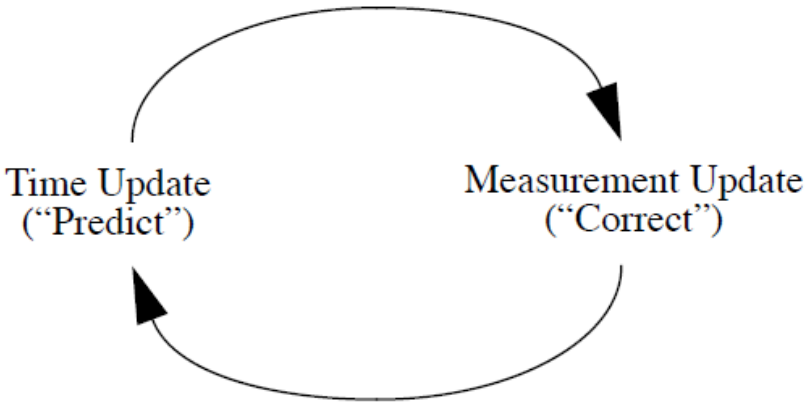

Figure 2.3: Kalman filter Cycle. The time update step predicts the state ahead in time. The measurement update adjusts the prediction by incorporating a noisy measurement at that time. The measurement update step can also be found as the update step [5].

2.2.2 Extended Kalman filter theory

In this section, we describe the steps of the extended Kalman filter. To provide a better explanation of the EKF, we present and explain the implementation of the EKF we used step by step. The implementation we used is originally from, but we had to make some modifications to the code, mostly in the inputs of the script. More information about the changes to the code will be presented in chapter 4.

First, we initialize our state vector. The state vector is a set of variables which can be used to describe a system, in our case this would be our robot moving in 2 dimensions. It is common for position and velocity variables to be a part of the state vector, same goes also for variables referring to the orientation. Lastly, system specific variables can also be a part of the state vector, like sensor biases or different types of accelerations. In our case, our state vector consists of the x and y coordinates, as we mentioned earlier, because the robot is moving in 2-dimensions, so the z axis does not come in play. As for orientation, the yaw angle will also be a part of the state vector and the last property is the velocity of the robot. So, the state vector will be a 4 by 1 vector and it will consist of x and y coordinates of the robot, its yaw angle and its velocity as show below:

$$n = \begin{bmatrix} x \\ y \\ \theta \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 2.4: The initial values of the state vector n for t = 0. x and y are the coordinates of the robot, θ is its yaw angle and v is its velocity.

After we initialize the state vector, we can proceed to predict the next state of our system. For us to be able to predict the next state we need to two things; first we need to know the current state, which we do, and the second thing we need is the motion model. The motion model describes how the state variables evolve as the time passes. We present our motion model in the figure below. We also have to mention here that the motion model depends on the robot. Our robot has differential movement, so it can be described with the equation below. A robot

that is able to steer its wheels would need different equations in order to be described.

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \\ u_{t+1} \end{bmatrix} = \begin{bmatrix} x_t + v * dt * cos(\theta_t) \\ y_t + v * dt * sin(\theta_t) \\ \theta_t + dt * \omega \\ u_t \end{bmatrix}$$

Figure 2.5: Position, yaw angle and velocity of the robot in the next time interval

In Figure *2.5*, we present the variables which refer to the next time interval with the index $t + 1$. These are the values we want to predict. The variables that refer to the current moment in time have the index t. dt is the time step we used during our experiment which was $dt = 0.1428s$. This selection was done because we could not go above ros-rate = 7, due to hardware limitations. In more powerful systems this limitation would not exist. Going back to ros-rate, this is the rate with which ros is running when operational. A higher rate would let us control our robot more times per second, so the movement would be more precise as we would be able to give instructions to our robot more frequently. Highter frequency would also be more demanding on the system and lead to higher power consumption. For our case, 7 was the higher we could go without facing any issues. Since we did not have to be extremely precise, ros-rate = 7 was satisfactory.

In the next step, we compute the Jacobian of the motion model. In order to calculate the Jacobian, we have to do partial derivates on the equations of the motion model, as seen in the equations below. Below we present only the ones which are complex and not the rest for them.

$$\frac{d_x}{d_{yaw}} = -v * dt * sin(yaw) \quad (2.1)$$

$$\frac{d_x}{d_v} = dt * cos(yaw) \quad (2.1)$$

$$\frac{d_y}{d_{yaw}} = v * dt * cos(yaw) \quad (2.2)$$

$$\frac{d_y}{d_v} = dt * sin(yaw) \quad (2.3)$$

As we can see in the above equations, we do partial derivatives on the x and y axis with respect to the variables v and yaw. Finally, after computing the rest of the partial derivatives, we end up with the Jacobian as we present it below.

$$jF = \begin{bmatrix} 1 & 0 & -dt * v * sin(yaw) & dt * cos(yaw) \\ 0 & 1 & dt * v * cos(yaw) & dt * sin(yaw) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

The last entity we need to compute in the prediction step is the covariance of the system. To begin with, we need to initialize the process noise covariance Q. The process noise covariance matrix a is $4\,x\,4$ table containing the variances of the variable from the state vector. The process noise covariance matrix contains non-zero values only in the main diagonal of the table as the variances of the variable are independent of each other. For these values we had to perform tests with different values in order to find the values that would perform the best, trying to give the EKF the best chances to perform as good as it can.

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & \sigma_v \end{bmatrix}^2 \quad (2.5)$$

Finally, we can now compute the covariance of the state using the equation (2.6)

$$P_{pred} = jF * P_{Est} * jF^T + Q \tag{2.6}$$

This is the last step for the prediction part of the EKF. Next, we move to the update part of the filter. At this point the filter uses the measurements we provided it with in order to correct its prediction. We start by initializing the Jacobian of the Observation model, which we present below.

$$jH = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.7}$$

After that, we compute the innovation or residual between the measurement and the estimated position of our robot, as we see below.

$$y = z - z_{pred} \tag{2.8}$$

where z is the table with the x and y coordinates from our measurements as we can see in equation (2.9).

$$z = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.9}$$

The z_{pred} table is the table containing the x and y coordinates we predicted earlier, and it is shown in equation (2.10).

$$z_{pred} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.10}$$

As we did with process noise covariance Q matrix, we also have to initialize the measurement noise covariance R matrix as we see below. The measurement noise covariance matrix captures the variance and the covariance of the system's measurements. For selecting values for the measurement noise covariance matrix, we did tests with different values in order to find the best performing ones. Again, as with the process noise covariance matrix, only the main diagonal has non-zero values, as the variances of the x and y coordinates are independent to each other.

$$R = \begin{bmatrix} {\sigma^2}_x & 0 \\ 0 & {\sigma^2}_y \end{bmatrix} \tag{2.11}$$

After that, we can compute the Innovation (or residual) covariance using equation (2.12). The innovation covariance quantifies the expected uncertainty the innovation which is as we said the difference between the measurement and the prediction, and we denoted it with the letter y.

$$S = jH * P_{Pred} * jH^T + R \tag{2.12}$$

After calculating the innovation covariance, we can proceed on computing the Kalman gain. The Kalman gain is chosen so that it can minimize the state covariance and it is calculated with the equation below.

$$K = P_{Pred} * H^T * S^{-1} \tag{2.13}$$

The Kalman gain determines how much weight should be put on the new measurement when updating the state of the system. If the covariance of the measurement is high, meaning that the measurement is noisy, the Kalman gain will be smaller, making the filter relay more on the prediction. In contrast, if the

measurement is less noise the Kalman gain will be bigger, making the filter rely more on the measurement instead of the prediction.

The last two steps are to update the estimated state and the estimated covariance of the system. As we see in equation

(2.14), the new estimated state is the sum of the predicted state and the Kalman gain multiplied by the residual.

$$x_{Est} = x_{Pred} + K * y \tag{2.14}$$

After the estimated state is calculated, we compute the estimated covariance using the Kalman gain, the Jacobian of the Observation model and the predicted covariance of the system. The equation we used for the estimation of the covariance is shown below.

$$P_{Est} = (I - K * jH) * P_{Pred} \tag{2.15}$$

These were the steps we used for the EKF, in chapter 4 we will present how it performed against an unscented Kalman filter, which we will present in the next chapter, and the machine learning model.

2.2.3 Unscented Kalman filter theory

Before we start diving into the equations of the unscented Kalman filter, let us point out the flows of the EKF and how the proposal of Jeffrey Uhlmann, the creator of the UKF, fixed those issues. Let us start with condition under the UKF is performing better than the EKF. UKF performs better than EKF when the system is highly non-linear, if the system does not present highly non-linear properties the two filters are expected to perform similarly. The flaw of the EKF is based on the way it calculates the optimal state vector and the optimal prediction. When the system is linear the filter can compute those values, in non-linear cases the filter has to perform a linearization and here is where the discrepancy of the two filters appears. EKF uses a linear approximation in order

to perform the linearization and then apply the rest of the steps, in contrast UKF uses third order approximation (Taylor series expansion) [6]. This is why when the system is not highly non-linear the two filters can perform similarly, as even a first order approximation does not deviate much from the result a third order approximation would give.

Now that we explained why the UKF performs better than the EKF in highly non-linear systems, we will go through the steps we followed in our implementation of the UKF, similar to what we did with the EKF.

We will start with our analysis with the three constants that will help us fine tune our filter these are alpha, beta and kappa. Alpha is responsible for the spread of the sigma point, we will explain that sigma points are later, a large value would mean that the sigma points would be far from the mean, in contrast a small value would generate the sigma points near the mean. Typical values for alpha are around the $10^{-3} - 10^{-4}$ order of magnitude. In our case we went with alpha being equal to 10^{-3}. The second parameter is beta. Beta has to do with the distribution, and we set beta equal to two, which lets our sigma points match up to the fourth moment (kurtosis) of a Gaussian distribution. The last tuning parameter is kappa, kappa is also responsible for the spread of our sigma points around the mean. Typical values for kappa are $0 - (3 - n)$, where n is the number of variables in the state vector. Our selection for the values of the three tuning parameters is typical in literature [6].

In order for us to proceed we would have to explain what sigma points are, so the reader can follow along the steps of our implementation of the unscented Kalman filter. Sigma points are a minimal set of carefully selected points that capture the uncertainty of the state distribution, by having the same mean and the covariance of the distribution.

The UKF has also some similarities with the EKF, UKF is also a two-step filter with the first step being the predict step and then we have the update step. We also have a state vector which will be the same as the state vector in EKF, containing the x and y axes coordinates, the yaw angle of the robot and its velocity.

$$n = \begin{bmatrix} x \\ y \\ \theta \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.16}$$

We also have the two noise matrices as we did before, these are the process noise covariance Q and the measurement noise covariance matrix R as shown in equations (2.17) and (2.18).

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & \sigma_v \end{bmatrix}^2 \tag{2.17}$$

$$R = \begin{bmatrix} {\sigma^2}_x & 0 \\ 0 & {\sigma^2}_y \end{bmatrix} \tag{2.18}$$

After having initialized all the parameters we need, we can procced to computing. We start by generating the sigma points using the estimated values of the state and the covariance of the system. After generating them we predict their movement using the same motion model we used for the EKF. The last for update part of the filter is to calculate the predicted values for the state and the covariance of the system, using the equations (2.19) and (2.20).

$$x_{Pred} = (w_m \times sigma^T)^T \tag{2.19}$$

$$P_{Pred} = P_{Pred} + w_c * d \times d^T \tag{2.20}$$

Before we can compute the state and the covariance, we firstly need to compute the weights w_m and w_c as well as the variable d. In variable $sigma$ we

stored our sigma points. Let us start computing what we are missing. We will start with the variable d which is the difference between our sigma points and the predicted state of our system, shown in equation (2.21)

$$d = sigma - x_{Pred} \tag{2.21}$$

Next up we must calculate the weights wm and wc. The first values of the two weights get calculated using the equations below, where nx is the number of variables in the state vector and it is equal to 4. The variable λ is calculated using the equation (2.24)

$$w_m = \frac{\lambda}{\lambda+nx} \tag{2.22}$$

$$w_c = \frac{\lambda}{\lambda+nx}\,(1 - A^2 + B) \tag{2.23}$$

$$\lambda = A^2 * (nx + K) - nx \tag{2.24}$$

The rest $2 * nx - 1$ weights are calculated using the equation (2.25)

$$w_m, w_c = \frac{1}{(2*(\lambda+nx))} \tag{2.25}$$

This would be all for the prediction step for our UKF implementation. We first generated our sigma points and then we predicted their movement. After that, we calculated the weights w_c and w_m and used them in order to calculate the predicted state and covariance of the system. After the prediction step, we will move to the update state, explaining all our steps in the process.

We start the update step, by using our observation model, equation (2.26), and the predicted state we previously calculated, in order to compute our predicted measurement as shown in equation (2.27) .

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.26}$$

$$z_{Pred} = H \times x_{Pred} \tag{2.27}$$

The we compute the residual using the equation below, as we also did in the EKF.

$$y = z - z_{pred} \tag{2.28}$$

Next step would be to generate the sigma points again but this time instead of using the estimated values of the state and covariance of the state we use the predicted ones. After generating them, we calculate their covariance.

$$st = st + w_c * d \times d^T \tag{2.29}$$

$$d = z\_sigma - zb \tag{2.30}$$

$$z\_sigma = H \times sigma \tag{2.31}$$

$$z_b = (w_m \times sigma^T)^T \tag{2.32}$$

Then we will compute the Kalman gain which can be found using the equation below.

$$K = P_{xz} \times st^{-1} \tag{2.33}$$

We have already calculated the st variable, but not the variable P_{xz}. For us to do so we need the following calculations.

$$dx = sigma - x_{Pred} \tag{2.34}$$

$$dz = z_{sigma} - zb \tag{2.35}$$

$$P_{xz} = P_{xz} + w_c * dx \times dz^T \tag{2.36}$$

After that, can finally calculate the Kalman gain, and with that we can calculate the estimated state and covariance of our system, using the equations below, which is our final step in the process.

$$x_{Est} = x_{Pred} + K \times y \tag{2.37}$$

$$P_{Est} = P_{Pred} - K \times st \times K^T \tag{2.38}$$

CHAPTER 3

SIMULATION SETUP

3.1 Robot description

In this chapter, we talk about the specifics of our robot. Our robot consists of two parts. The first part is the Jackal from Clearpath robotics, and the second part is the manipulator on top of it from Kinova robotics. A picture of the robot can be found below. Everything we needed was compiled in this github repository from Sungwwoo and it can be found here: https://github.com/Sungwwoo/jackal_kinova_simulator. In this repository, we found detailed instructions on how to install all the necessary dependencies and how to launch a world in gazebo with the robot inside.

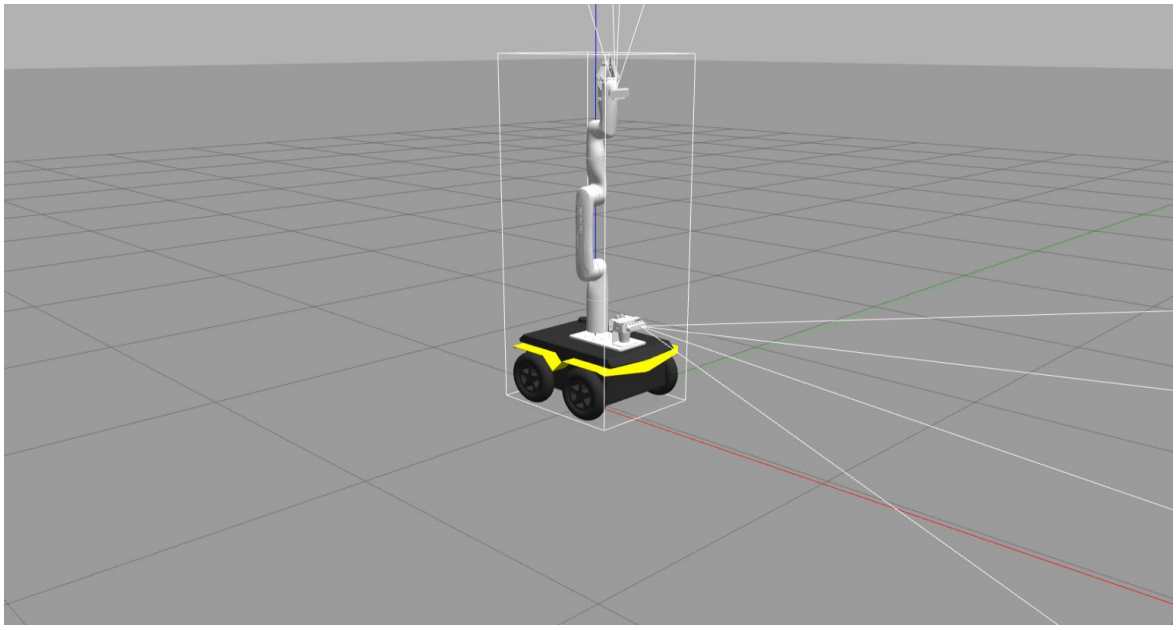

Figure 3.1: A picture of the robot used in our experiments.

The manipulator on top of Jackal was not necessary, as it was not used in any of our experiments, but it opens possibilities for further work on the matter. As for Jackal, it weights $17Kg$ and its external dimensions are $508\ x\ 430\ x\ 250mm$. The manufacturer claims $4h$ of usage and a maximum payload of $20Kg$. By default, Clearpath has open API for both ROS Melodic and ROS Kinetic. Unfortunately, we are using ROS noetic, so we are not able to use the official API on the actual robot, without migrating to a different ROS version.

The capabilities of Jackal can be extended. The robot can be equipped with a variety of sensors. Cameras can also be installed on the robot making it able to provide video feed to the user. It also provides 5V, 12V and 24V power options for any additional component to use. The robot also has an internal area which can be used for additional computing power or storage, depending on the user's needs. Satellite navigation modules can also be installed on our robot. In our case we are using a module called Duro inertial for satellite navigation, which we will explore later on.

All these technologies would be useless if they could not be properly protected and maintained. The Jackal's chassis is built from aluminum which makes it suitable for all terrain operation. Jackal's built quality gave it an IP62

rating, which validates it for use, where temperatures can vary from -20 Celsius all the way up to +45 Celsius.

Even though we did not use the manipulator as we already said, we still want to present some of its features. The kinova gen3 lite manipulator has a maximum reach of $760mm$, can hold a maximum of $0.5Kg$ of continuous payload. Its weight is $5.4Kg$ and its power consumption is $20W$.

At this point, we would like to also present the GPS module our robot is equipped with. The Duro inertial, is a multi-band, multi-constellation, GNSS and INS module, developed by Carnegie Robotics. Multi-band means that it can receive signals on multiple different frequency bands, which is useful as different navigation satellite systems use different frequency bands. Being able to use multiple navigation satellite systems increases the accuracy and reliability of localization. Carnegie Robotics claims that it can provide location data with centimetre accuracy. Multi-constellation means that the module can use different satellite constellations such as, GPS from US, GLONASS from Russia, BeiDou from Chine and Galileo from Europe. GNSS is the global navigation satellite system, which is the term coving all the satellite navigation system that provide geolocation and time data. Lastly, INS (Inertia Navigation System) is the system which uses gyroscopes and accelerometers in order to calculate the position, orientation and velocity of the object, where the module is mounted on. Combined GNSS and INS provide accurate and reliable localization.

3.2 Gazebo description

Gazebo is a 2D/3D simulator initially developed as a part of a Ph.D research project in 2002. Gazebo supports four different physics engines ODE, bullet, simbody and dart, with ODE (Open Dynamis Engine) being the default one and the one that we ended up using. These physics engines allow gazebo to simulate accurately the physical phenomena of the modeled scenarios. Essentially, the physics engines enable our simulator to simulate the laws of physics in a simulated environment.

ODE's features make is a solid option for rigid body simulation. ODE is a modern library which provides a good balance between performance and

capabilities. About the performance part, it has to be fast enough in order to be able to offer a good experience to the user. ODE manages to achieve these levels of performance also since it is rewritten in C and C++. As for its capabilities, ODE provides necessary features as joint support making it easy to simulate robots with joints, our robot also has a manipulator with joints, but we are not using them in the scope of this thesis. ODE has built-in collision detection using axis-aligned bounding boxes (AABB). We should also mention ODE's stability and the robustness.

Apart from the physics engine, gazebo also provides a graphical environment. In the gazebo graphical environment, we have two main categories, world and model. As world, gazebo considers any static object and as model all the dynamic ones. Both world and model are configurable through parameter easily accessible via the gazebo graphical environment.

At this point, we would like to present the architecture of gazebo. In Figure *3.2* we present a high-level representation of gazebo's architecture. The architecture we see below was originally created in 2004 and had little to no changes as it is simple, and it relays on third party software to enlarge its capabilities. This structure enables gazebo to be an independent platform which allows different physics engines for example. There is also a distinction between server and client. On one hand, there is the server, where the actual simulation takes place, things like the rendering, the sensors and the physics belong to the server side. The client on the other hand is responsible for the graphical user interface and the ability to interact with the simulation [7].

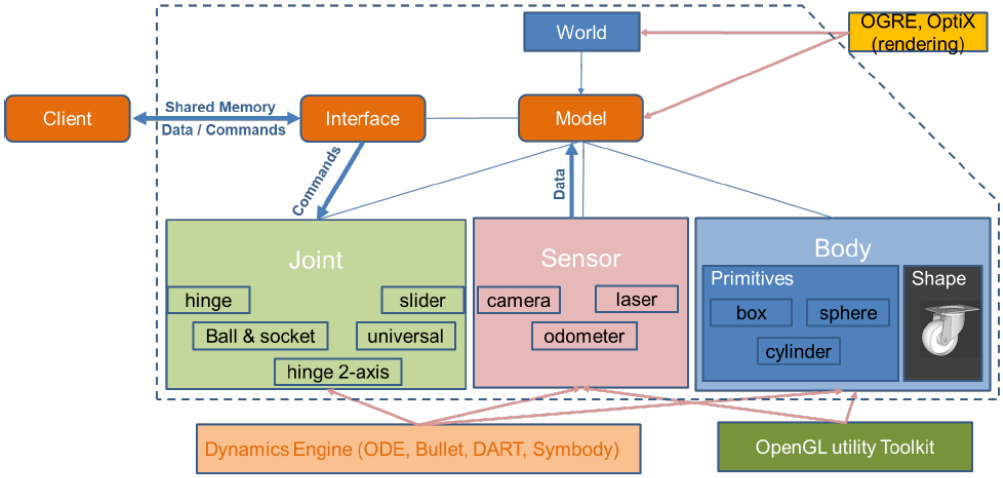

Figure 3.2: High-level gazebo architecture[7].

As we are using a wheeled mobile robot, our movement is limited to x and y axes surface. This limitation entails that we need only the x and y coordinates, regarding the position of the robot, and only the yaw angle in respect to the orientation of the robot. Below we have a picture showing our robot. In Figure *3.3*, we present our robot in an empty world, and we have noted the x axes, y axes and the yaw angle (denoted with the letter θ). These are the entities we are interested in. We have also noted the z axes for the shake of completeness.

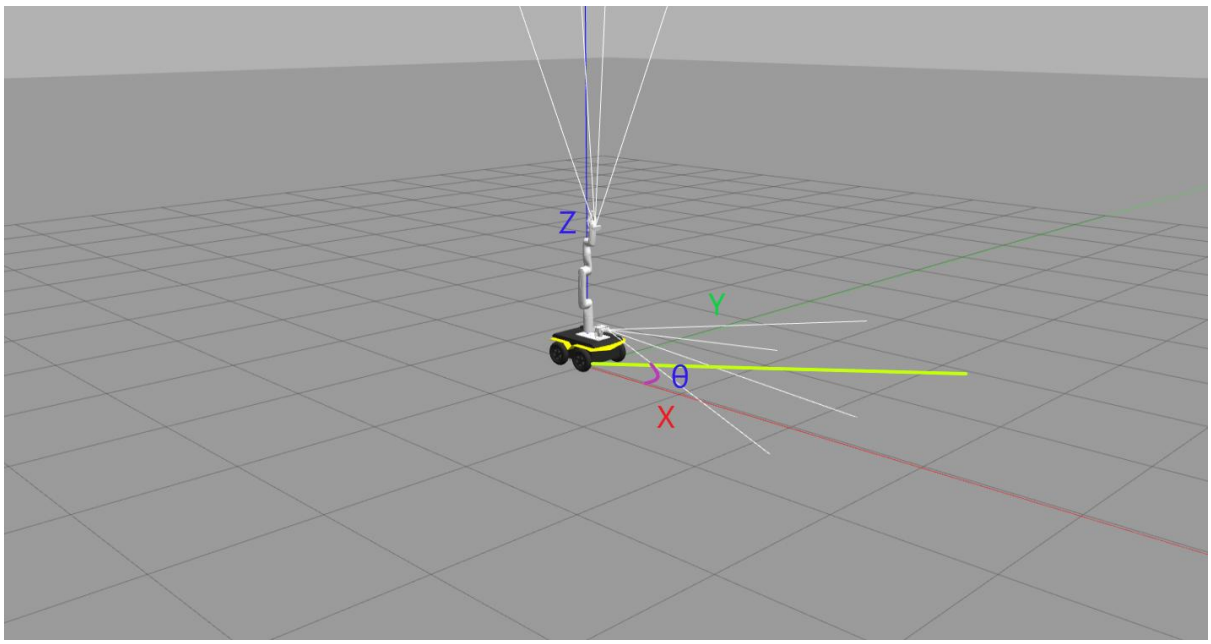

Figure 3.3: Robot's pose in Gazebo.

Having to deal with only the data for the x and y coordinates and the yaw angle for the orientation and not all six variables needed to specify the pose of a non-point abject in 3-dimensional space (see the figure below) significantly reduces the amount of data we need to compute for our experiment.

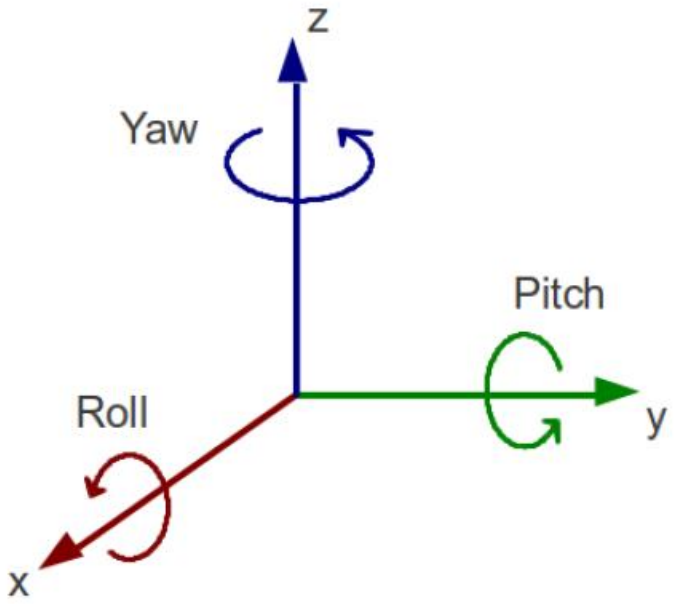

Figure 3.4: [8] Three axes and three angles, all needed in order to describe the pose of an object in the three-dimensional space.

3.3 Robot Operating System (ROS)

3.3.1 General ROS information

Robot operating system is an open-source robot software development software. ROS creates a workspace where the different components of the robot can talk to each other using nodes and topics. We would not have any use for ROS if we were not able to interact with the different components of the robot ourselves.

Let us break down the communication between components. Most robots consist of actuators that make the robot move, sensors are the components that let robots read their surroundings, and control systems which decide the robot's

actions. ROS lets us build these components with relative ease and make them interact with each other.

We should also mention the different versions ROS offers. ROS is separated in two main versions, ROS 1 and ROS 2. ROS 1 is legacy and is the one we are using, more specifically, we are using ROS 1 noetic release. ROS 1 is only available for linux and still holds the majority of the market, as it is common in the industry to delay upgrades in software for security and possibly not polished upgrades. Also, it could be that years of work in ROS 1 would need great time and effort to migrate to ROS 2, as the transition is not as direct as going from one ROS release to another of the same version. There have been tools that help with the transition from ROS 1 to ROS 2, but this is still an Issue for big project in the industry.

Having said all that, ROS 2 is still a great step forward for the robotics' community. ROS 2 offers multiplatform support for windows, linux and macOS, making broadly usable for users. One more feature of ROS 2 is the improvements in security. ROS 2 offers authentication and encryption for its communications which makes it especially appealing in a networked environment.

3.3.2 Thesis specific ROS information

After some general information about ROS, we now want to be more specific about how we used ROS. In this section we present our robot specific topics and nodes that we use and in general how we used ROS for the purpose of this thesis.

In Figure *3.5*, we have a graphic representation of all the nodes and topics as well as the relations between them that are active during our script execution. The rectangles are called topics, and the ellipses are called nodes. Nodes can subscribe to a topic; this is indicated by an arrow coming from the topic and pointing to the node. Subscribing to a topic means that the node is getting information from that topic. Nodes can subscribe to a topic but can also publish to one. Publishing is indicated by an arrow going from the node towards the topic that it publishes to. Publishing to a topic means that the node is providing information to that topic.

We are not going to examine all the topics and nodes, as there is a high number of them because of the complexity of our robot and all of its different

components. We are going to focus on the " Jackal " node. To interact with our robot, we created the aforementioned node. As we can see in Figure *3.5*, our node is subscribed to two topics "/gazebo/model_states" and "/clock" and publishing to "/jackal_velocity_controller/cmd_vel" and "/rosout".

Let us begin with the two topics that Jackal subscribes to. Starting with the simpler one which is the clock one. The clock topic creates a ground truth regarding time among the different systems of the robot, by creating a virtual clock. Having a common ground for time ensures that all the components that we talked about can work in sync. The clock topic is not relevant for us, but we are mentioning it for completion. The second topic Jackal subscribes to is the "/gazebo/model_states". This is the topic that provides us with the real pose of the robot. As we have already mentioned, our robot moves in two dimensions x and y, so its pose can be described with just the x, y coordinates and the yaw angle.

Next up, we talk about the two topics that Jackal publishes to. First up we will start with the "/jackal_velocity_controller/cmd_vel". This is the topic where we publish the speed, both linear and angular, we want our robot to have an input. The last topic we will need is the "/rosout". This topic is used for monitoring and debugging purposes. This topic is also not relevant for us but we mention it for completion.

To sum up, these aforementioned topics are the topics that our node is interacting with either by publishing, providing information to them, or by subscribing to them, retrieving information from them.

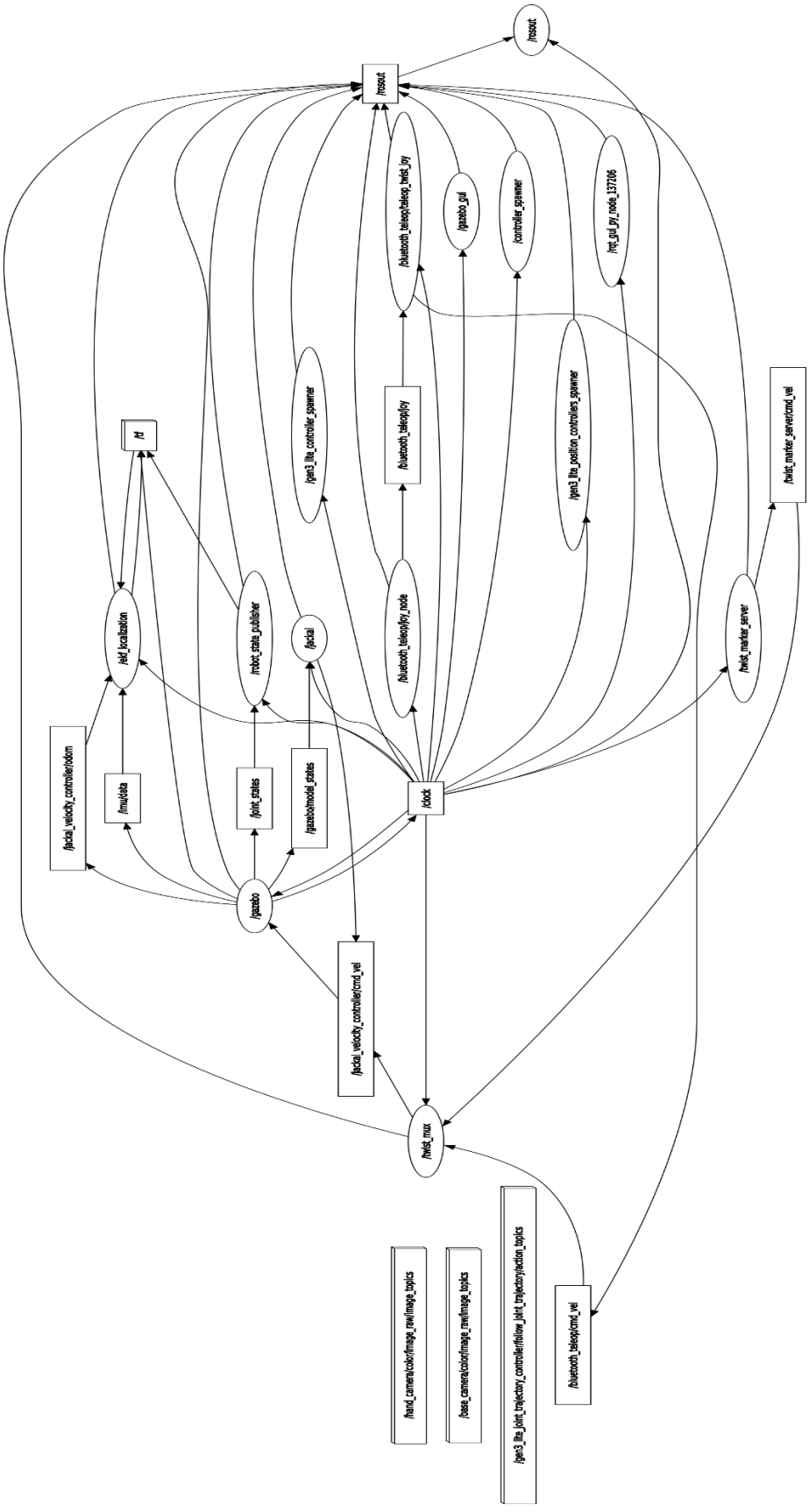

Figure 3.5: rqt_graph, all nodes and topics that run during the script execution.

3.4 Data acquisition

In this part, we explain the process of obtaining positional data from our robot. We require data about the pose of our robot for both the machine learning and the Kalman filters. This means we need to make sure that we are getting the data from the correct source. To do this, we must choose the correct topic and make sure the data get saved correctly, so no bugs get introduced during the process.

We start with the odometry data. In order to get the data from gazebo we are subscribing to the odom topic. Once we get our odometry message, we need to get into pose, so we can access the pose of our robot. Then, it is straightforward to assign the values of the x and y coordinates to some variables. For the orientation, we first need to get all three angles (roll, pitch, and yaw) as quaternions and then use the euler_from_quaternions method to convert them to Euler angles. After we acquire the x and y coordinates and the yaw angle, we can simply output our three values into a csv file, in order to make the processing of the data easy.

Now for the ground truth, we follow the same principles with the small exception that this time x, y, and z coordinates are given in one string. To solve this issue with have to resort to splitting the string into three, one for each coordinate. Once this is done, we have our x and y coordinates, meaning that we only miss the yaw angle, which can be retrieved in the same way we did for the odometry data, using the euler_from_quaternions method.

After acquiring the odometry data and the ground truth, we also need the control input. For our Kalman filters and our machine learning model to work, we have to be able to access the control input info. This is something that we could hard code into the Kalman filter and in the data for the machine learning model, but this would not only be bad practice but would also limit us to using simple movements, meaning a fixed set of linear and angular speeds. Using the time module, we are now capable of using elapsed time in order to change the control input as we wish using

the series of if statements. As a result, we can have different linear and angular speeds, meaning that our robot can now perform more complex series of movements. Once the variables for linear and angular speed get their values, we output them in a csv file as we also do for the odometry and the ground truth. Once that is also done, we publish our speeds to the cmd_vel topic.

CHAPTER 4

DATA AND RESULTS

4.1 Acquiring comparison data

As for comparing the Kalman filters and the machine learning model, we decided to use a sinus movement. To get the pose of the robot during the sinus movement we first had to find a way to make the robot move accordingly. This task was not trivial as the only inputs we could provide our robot with were its linear and angular speeds, thus we were forced to get creative.

The solution to this problem came in the form of a simple point in the 2-dimensional space. As we could not use x and y coordinates as inputs for our robot, we created a point, which would move according to a sine. Its y axis was defined with a simple sine function as we see below,

$$y = A * \sin(\omega * t) \tag{4.1}$$

where y is the y coordinate measured in meters, A is the amplitude measured in meters, t is the time measured in seconds, and ω is the angular speed measured in radians per second. In our case, we used $A = 5m$ and $\omega = 0.05\, rad/s$. These values were selected as they would give us a clear sinus movement as we see in the figure below and the actual movement needed from the robot was achievable. For example, a robot like ours would not be able to perform a movement with its angular speed being $\omega = 5\, rad/s$.

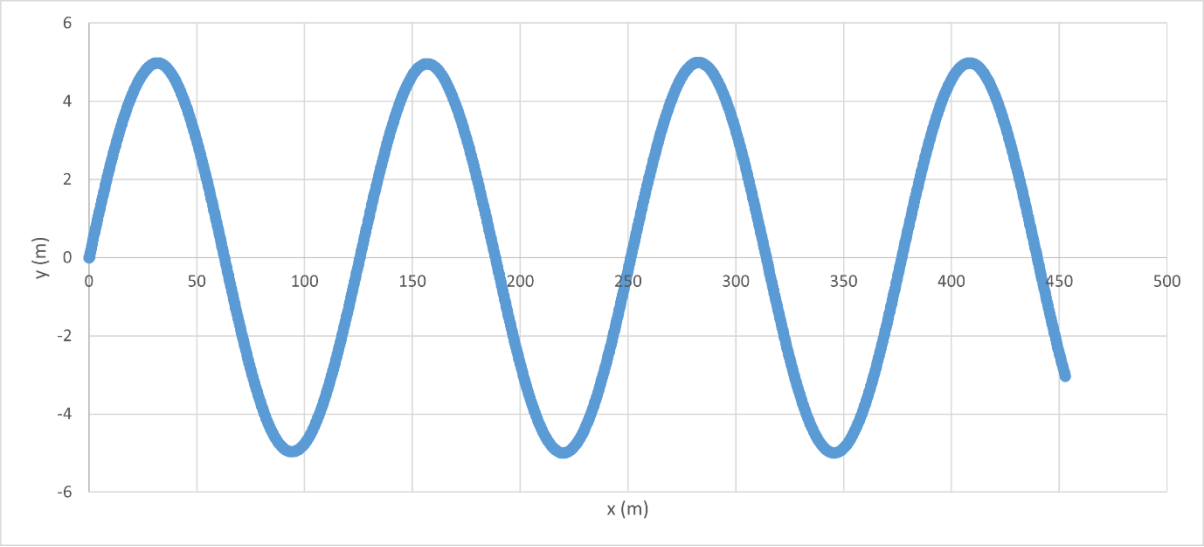

Figure 4.1: Sinus movement, A = 5m and ω = 0.05rad/s.

4.2 Adding noise

As our data came from the "/gazebo/model_states" topic, meaning that our data were error-free, making it impossible for us to correct them. The solution to this problem was given in the form of adding artificial noise to our data. Our idea was to try to mimic the data that we would get by using a GPS, so we added a random number between $-0.5m$ and $0.5m$ to every value. We used the RAND method of excel in order to add the noise. The RAND function is a uniform function meaning that each number in the range has the same probability of being selected. In the Figure *4.2* we created 10000 random numbers using the RAND function. As we can see the distribution is not perfect, but this has to do with the fact that we don't have an infinite amount of numbers in our experiment.

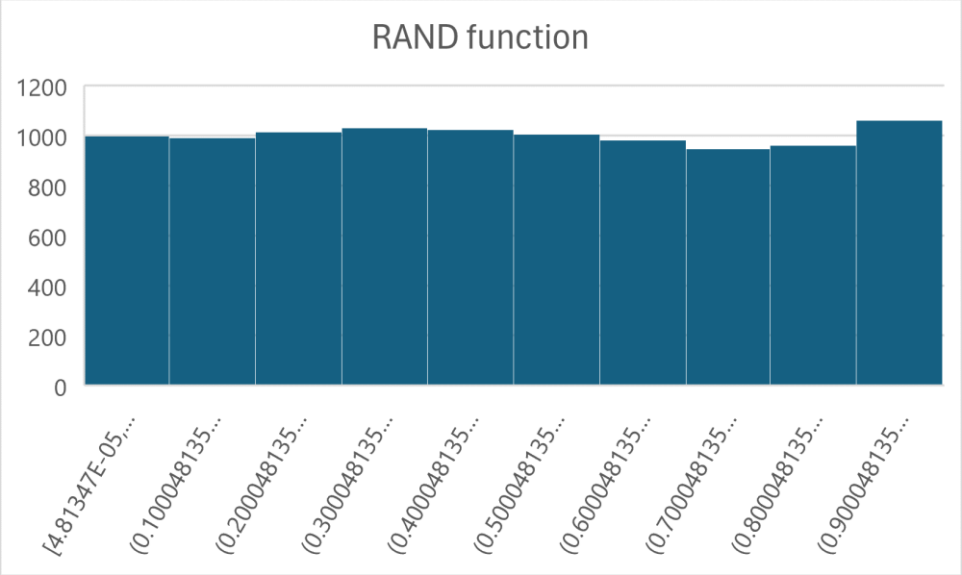

Figure 4.2: Experiment for proving the uniformity of the RAND function.

By default, the RAND function provides a random number from 0 up to 1, but in our case, we wanted the noise to be between $-0.5m$ and $0.5m$. We managed to create random numbers from -0.5 and 0.5. using the equation (4.2) in excel.

$$RAND() - 0.5 \tag{4.2}$$

This expression simply creates a random number from 0 up to 1 and by subtracting 0.5 we manage to move the results inside the range we want.

In order to better present our method of adding noise and its results let us present you an example. In the Figure *4.3* we present in blue the actual position of our robot during a random run. These data were recorded from the "/gazebo/model_states" topic, meaning that there are not errors in our data. In the orange, we present the same data after adding noise $[-0.5..0.5]m$. As we can see, the orange dots follow the general course of the actual trajectory.

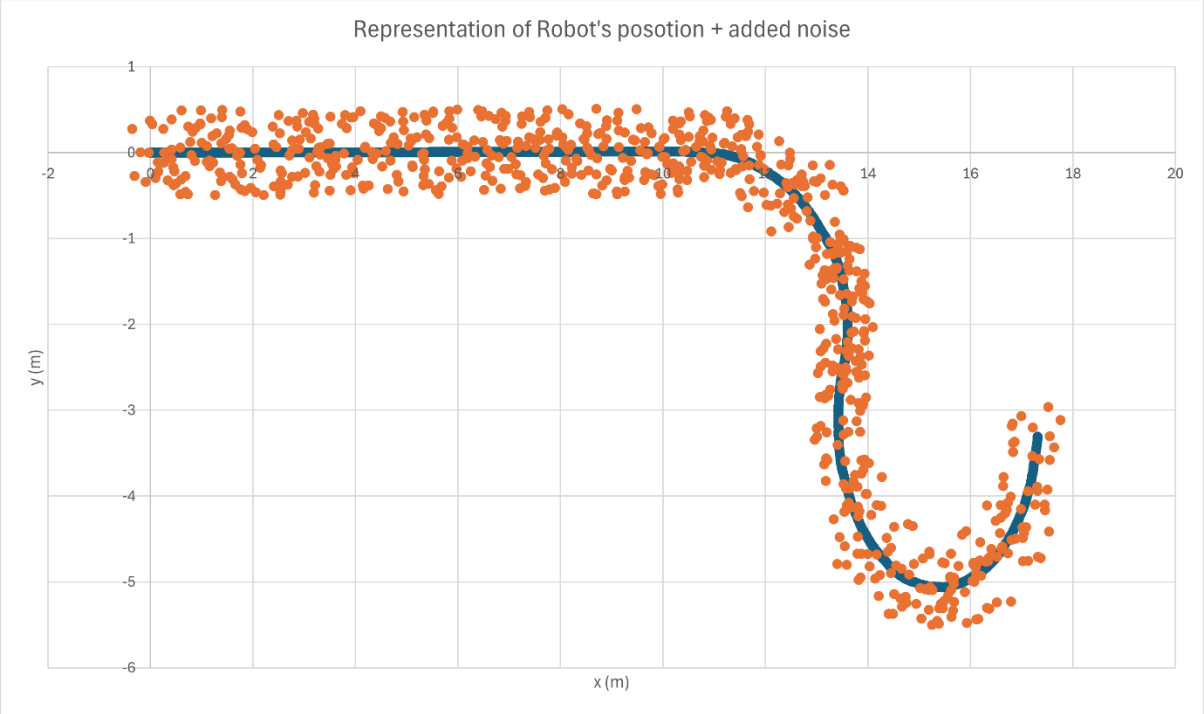

Figure 4.3: The blue dots represent the true trajectory of the robot. The orange dots represent the position of the robot after adding noise.

With the example above we have presented everything we did during the process of adding noise to our data. Hopefully this section gave you a good understanding of why and how we added noise to our data.

4.3 Odometry topic

Before we decided to add noise to our data, we tried to use the data provided by the odometry topic. The odometry topic provides the pose of the robot using data from sensors like wheel encoders or inertia measurement units (IMU). One can easily understand that the precision of odometry is quite susceptible to factors such as slippage due to differences in the terrain because the power provided to the wheels does not necessarily correspond to the distance traveled by the robot, or a terrain that is uneven may make it harder for the robot to move. In theory, none of these factors should come into play for our case as none of them was introduced in our experimental setup. But so much for theory, as our experiment showed us a different story.

In Figure *4.4*, we present a simple trajectory of our robot. In orange we present the true trajectory of the robot. The trajectory provided by the odom topic is presented by the blue line. In this movement our robot starts from the (0,0) point and finishes roughly at the (53,2) as seen from the orange line. In contrast, the blue line tells us a different story as in this case our robot starts from (0,0) as well but its finishing point is (52,11).

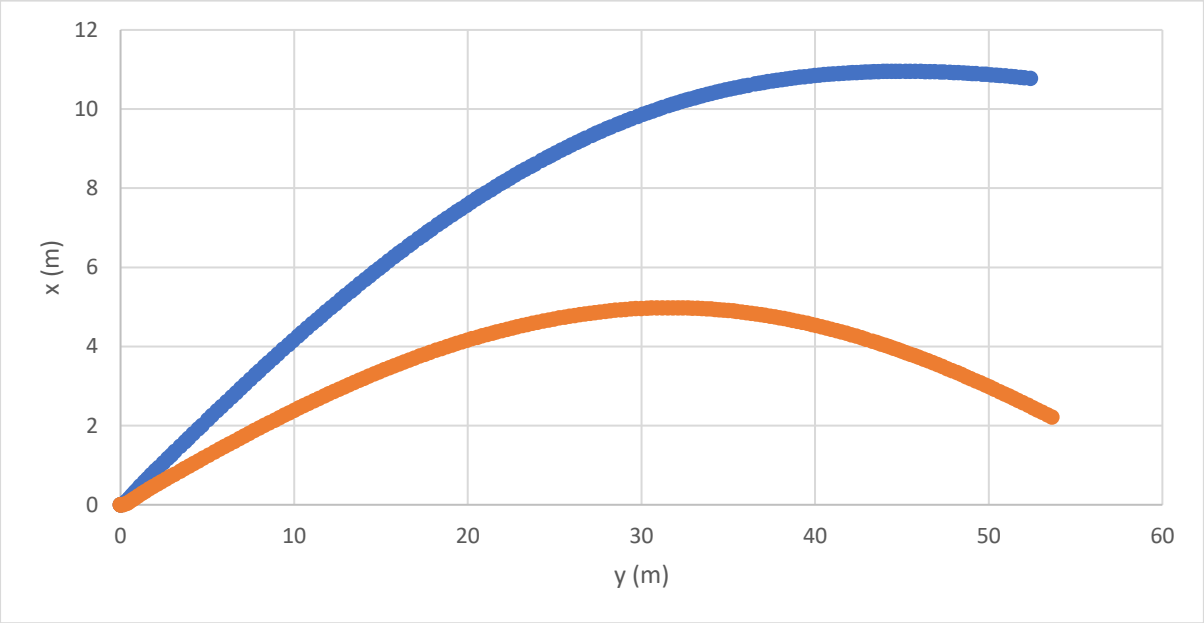

Figure 4.4: True trajectory of the robot (orange line), the trajectory given by the odom topic (blue line).

As we can see, the deviation between the true trajectory and the one from the odom topic keeps getting larger. To quantify this deviation, we present the sum of the absolute difference between the two trajectories in both axes. In the y-axes of Figure *4.5*, we calculate the sum of the absolute difference between the two trajectories for each of their points, using the equation (4.3). The difference is calculated for each one of the 200 points of the trajectories.

$$difference = |x_{i_{odom}} - x_{i_{modelstate}}| + |y_{i_{odom}} - y_{i_{modelstate}}| \qquad (4.3)$$

In the x-axes, we just present the points of the trajectories. The first point is where our robot started, and the last one presents the last point of the trajectory. As can easily be observed, the difference gets bigger as the robot travels for longer.

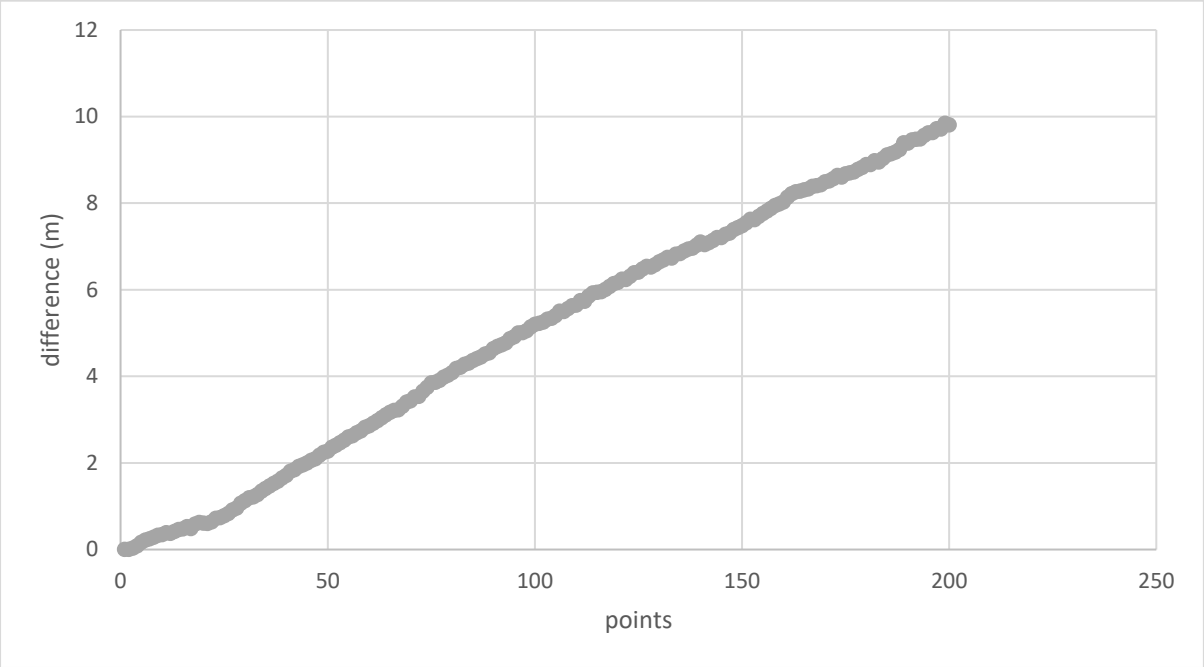

Figure 4.5: The total absolute difference between the true trajectory and the trajectory from the odom topic.

At this point, we should mention that the above experiment was done with more than one trajectories, all of them with the same result. As we needed to make sure that our observation was not a result of a random error, we used different trajectories with all of them resulting in the same conclusion: at the start of the movement the data from odom were close to the real data from the modelstate topic but as the time passed the difference between the two sources was getting bigger. In some cases, we also noticed the data from odom to completely change course and paint a completely different movement. As a result of our observations, we decided against using the odom topic.

4.4 Machine learning

In this chapter, we present the whole process of creating our machine learning model and predicting our robot's position with it. We start the process

of getting our data in order to train our model, we set up the pipeline for the code, where we have to parse our data, then we train our model. Once the training is done, we predict the position of our robot using never seen before input data. The last thing we have to do is to validate our results and evaluate them by comparing them to ground truth data, where there is no noise.

The first order of business is for us to collect the data to train our machine learning model. For this part, we have to have our robot move using random linear and angular speeds. As our robot is moving, we track its position and its orientation. Below, we present some of the trajectories our robot followed during this part.

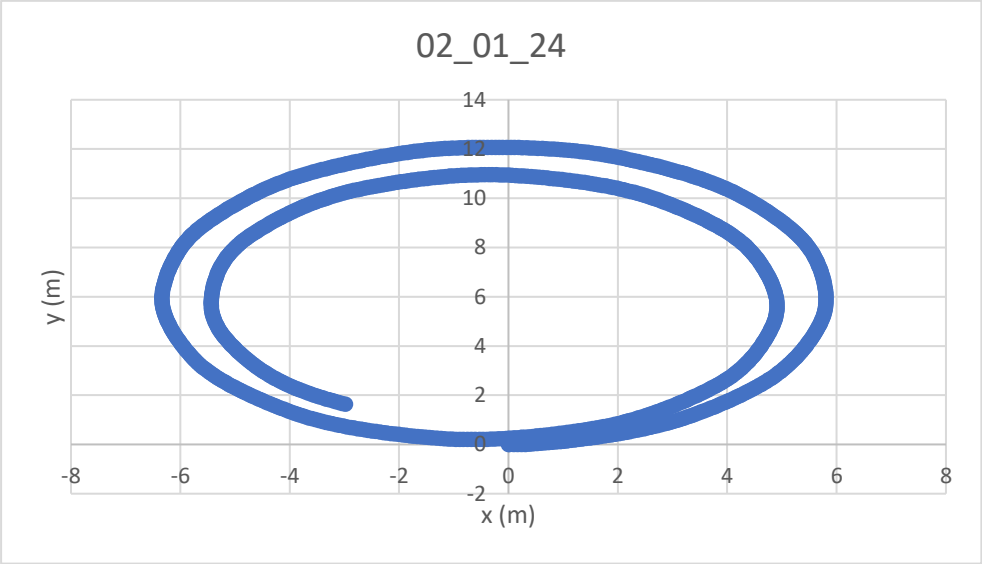

Figure 4.6: One of the trajectories used to train the machine learning model.

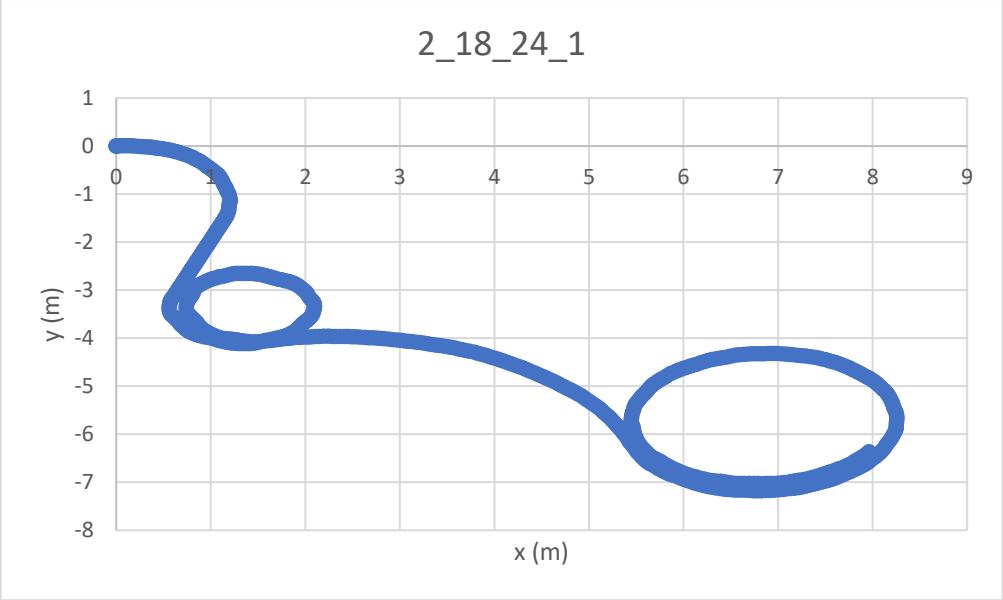

Figure 4.7: One of the trajectories used to train the machine learning model.

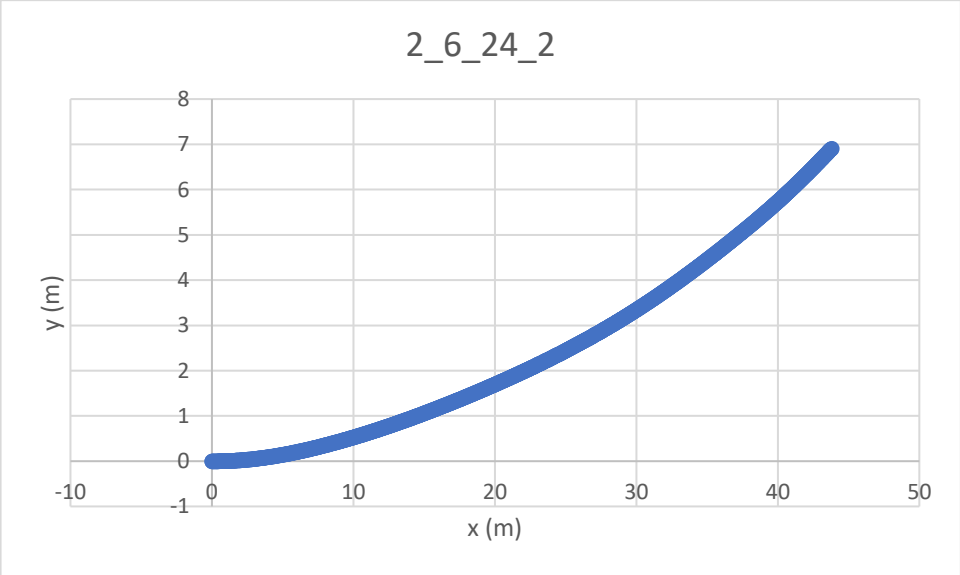

Figure 4.8: One of the trajectories used to train the machine learning model.

In the figures above, we show just three of the trajectories we used to train our model. In total, we have 17 trajectories. At this point, we should also mention that we had to make sure that none of our trajectories were similar to the trajectory we will use in order to compare our methods. Otherwise, we would have given the model an unfair advantage, as it would already know the correct results it would need to output.

Next session is about the technicalities of our process regarding the machine learning. The first thing we have to do is to import our data. For this task, we create a parser that loads our data and then stores them in a pandas dataframe. Once parsed, our data would look like in the figure below.

Table 4.4.1: Part of training data, formatted correctly so they can be used.

	A	B	C	D	E	F	G	H
1	x_t	y_t	yaw_t	u	omega	x_t+1	y_t+1	yaw_t+1
2	193.5939	-0.13907	-0.00075	1	0	193.7359	-0.13917	-0.00075
3	193.7359	-0.13917	-0.00075	1	0	193.8889	-0.13929	-0.00075
4	193.8889	-0.13929	-0.00075	1	0	194.0279	-0.13939	-0.00075
5	194.0279	-0.13939	-0.00075	1	0	194.1719	-0.1395	-0.00075
6	194.1719	-0.1395	-0.00075	1	0	194.318	-0.13961	-0.00075
7	194.318	-0.13961	-0.00075	1	0	194.4479	-0.13971	-0.00075
8	194.4479	-0.13971	-0.00075	1	0	194.596	-0.13982	-0.00075
9	194.596	-0.13982	-0.00075	1	0	194.747	-0.13993	-0.00075
10	194.747	-0.13993	-0.00075	1	0	194.9	-0.14005	-0.00075
11	194.9	-0.14005	-0.00075	1	0	195.041	-0.14015	-0.00075
12	195.041	-0.14015	-0.00075	1	0	195.175	-0.14025	-0.00075
13	195.175	-0.14025	-0.00075	1	0	195.323	-0.14036	-0.00075
14	195.323	-0.14036	-0.00075	1	0	195.462	-0.14047	-0.00075
15	195.462	-0.14047	-0.00075	1	0	195.6031	-0.14057	-0.00075

In column A, we present the current x-axes coordinate. Column B is the y-axes coordinate. In column C, there is the yaw angle of the robot. The next two columns are the control input data, with the linear and angular speed presented accordingly. With the last three columns, we present the future position and orientation of the robot in the same order as the current ones, with these being the x-axes coordinate, the y-axes coordinate and lastly the yaw angle. After explaining all the columns, we now must determine the features/dimensions and the label data. Columns A through E are the features as these depict the current state, and columns F, G, and H are the label data, meaning these are the columns our model will have to predict. After loading all the train data, we also load the test data in the same way but stored in a different dataframe.

The next step is to create our machine learning model and then train it. At the start, we separate our train data to features and label. The separation was done according to the previous paragraph. We decide upon predicting the x and y coordinates separately. For the first run, we use as features the columns A, C, D, and E and the column F as the label data. For the y coordinate, we use the columns B, C, D, and E and for the label data the column G. After the selection of features and label data, we choose the degree of our polynomial, as we already mentioned, this is done through trial and error. The best results came from a polynomial of first degree. The last step is to create our linear regression model and fit it with the train data, both features and label data. By the end of the process, we have a model capable of predicting the position and orientation of our robot, but its accuracy had still to be validated. The validation of our model comes in the form of testing it against a sinus trajectory of our robot.

The last thing we need to do is to test the predicting capabilities of our model. This time instead of separating the dataset into features and label data, we only select the features and ask our model to predict the label data. We repeat this process twice, once for the x coordinate and then for the y coordinate. When this is done, we have a dataset with the predicted values for the position of our robot.

Now, we test it against the ground truth data in order to compute the average deviation per result, for both x and y axes. In Figure *4.9*, we present the data from our machine learning model, blue line, and the ground truth, orange line. In thisFigure *4.9*, it is very difficult to distinguish the two lines, as both are close enough that they look like they overlap completely. This is the first step in our analysis, so we have an overview of how well or badly our model is performing.

The next step is to compute the actual absolute distance for each single point. The process is repeated for x and y coordinates. We use a simple absolute difference for each point in order to compute the error for each point. After that, we sum up all the errors and divide them by the number of points we have. By doing that, we calculate the average difference between the results of our model and the true position of the point. The average difference for the x coordinate is

$0.185m$ and for the y coordinate the average difference is $0.003m$. For both axes, the errors are acceptable as our model performs better than our noisy sensors that have an average of $0.25m$ for both axes. We know the exact average of the error from our sensors, as we are the ones who added the error to them.

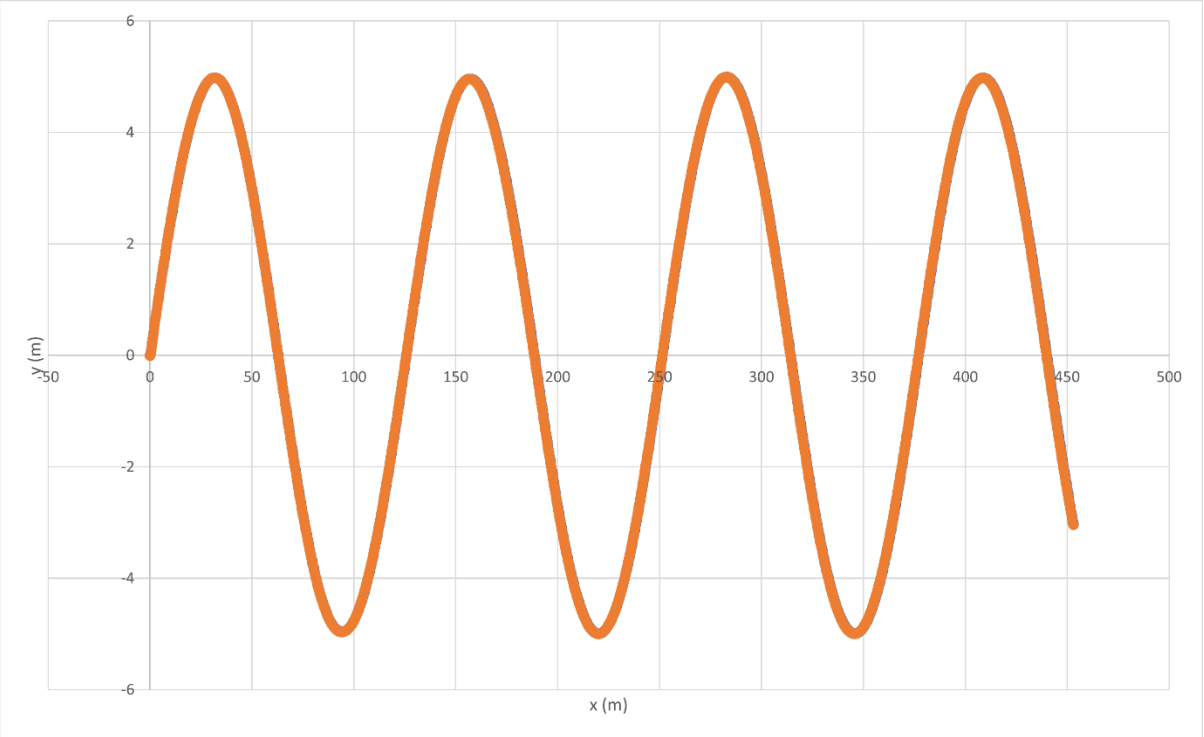

Figure 4.9: Comparison between the machine learning model and the ground truth. The orange line is the ground truth, the blue line is the result of the machine learning model.

To sum up the machine learning process, we first load our data, then we create our model and once it is trained, we calculate the average error our model produced, from the data of our comparison trajectory.

The full code for both the machine learning and the Kalman filters we used, can be found in this github repository: https://github.com/stathis-rafailidis/PythonRobotics.

4.5 Extended Kalman filter

In this chapter, we present the methodology we used for the extended Kalman filter and the results it yielded. The original code uses random numbers as inputs for the ground truth and then adds some noise in order to mimic the

measurements from the sensors. We modify the code, so it loads our data, both the data for the pose of the robot and the data for the input control to dataframes. After we make sure the filter is using our data, we have to calibrate the two noise matrixes, the process noise covariance matrix and the measurement noise covariance matrix. After the testing, we end up with the values we present below.

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & \sigma_v \end{bmatrix}^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 31 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^2 \tag{4.4}$$

The covariance of x and y are in meters, the covariance of the yaw angle is in degrees, and the covariance of the velocity is in meters per second. Below we present the matrix for the measurement noise covariance matrix, where both values are in meters.

$$R = \begin{bmatrix} {\sigma^2}_x & 0 \\ 0 & {\sigma^2}_y \end{bmatrix} = \begin{bmatrix} 3.5^2 & 0 \\ 0 & 3.5^2 \end{bmatrix} \tag{4.5}$$

As for the results, we plot them below as we did with the results of the machine learning. In Figure *4.10*, we present the comparison between the extended Kalman filter and the ground truth, ground truth in blue and EKF in orange. As with the machine learning model, the graph cannot help us much to distinguish them.

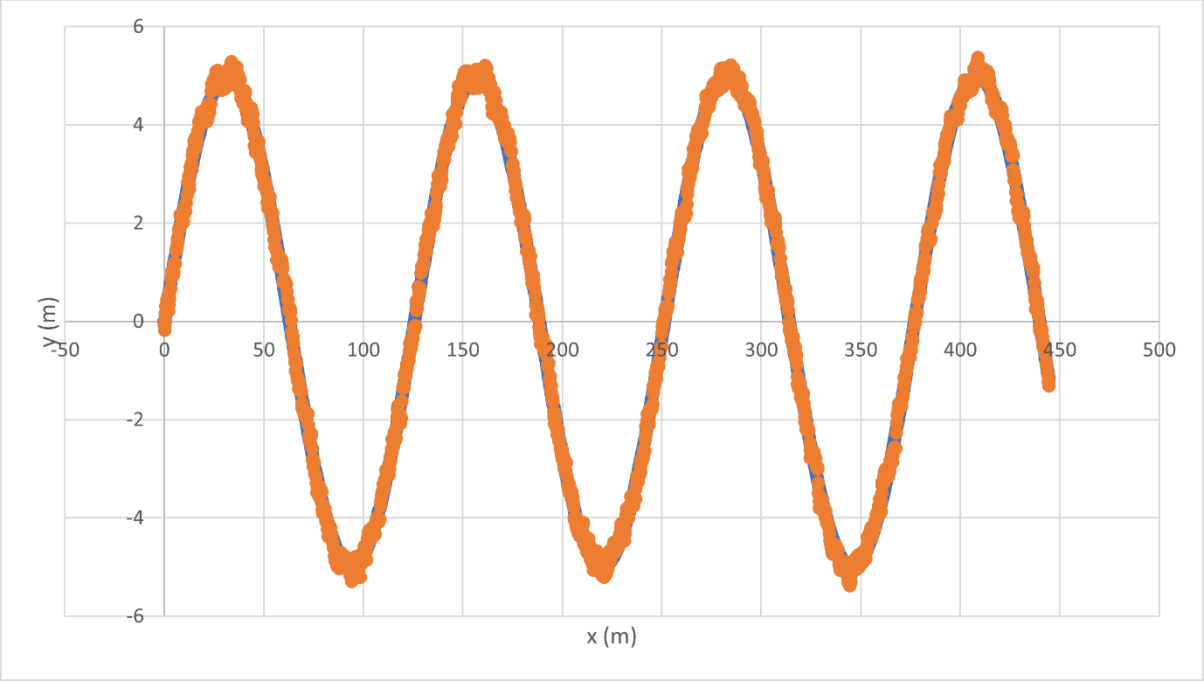

Figure 4.10: Comparison between the results of the extended Kalman filter and the ground truth. The blue line is the ground truth, the orange line is the result of the filter.

For the final analysis, we need to calculate the average error our filter produced. We follow the same strategy as we did with machine learning. After summing up all the absolute differences between ground truth and EKF, we divide by the number of points in our data set. The average difference for the x coordinate is $0.206m$, and for the y coordinate, it is $0.127m$. Again, our results are acceptable as our filter outperformed the case where we would not have any measure of reducing the errors from our sensors in place, as in that case the average error would be around $0.25m$.

4.6 Unscented Kalman filter

In this section, we demonstrate our work with the last Kalman filter variation that we studied, which is the unscented Kalman filter. The whole process is similar to the process we used for the extended Kalman filter. First, we modify the original code for the filter in order to use our data as measurements and as input control. Then through trial and error, we find the best values for the process noise covariance matrix and the measurement noise covariance matrix, shown below.

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & \sigma_v \end{bmatrix}^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}^2 \tag{4.6}$$

$$R = \begin{bmatrix} {\sigma^2}_x & 0 \\ 0 & {\sigma^2}_y \end{bmatrix} = \begin{bmatrix} 2^2 & 0 \\ 0 & 2^2 \end{bmatrix} \tag{4.7}$$

After that, we have an overview of our results with the graph shown below, which is a quick way for us to verify that our data are close to the ground truth. The orange line is the ground truth, the blue line is the EKF.

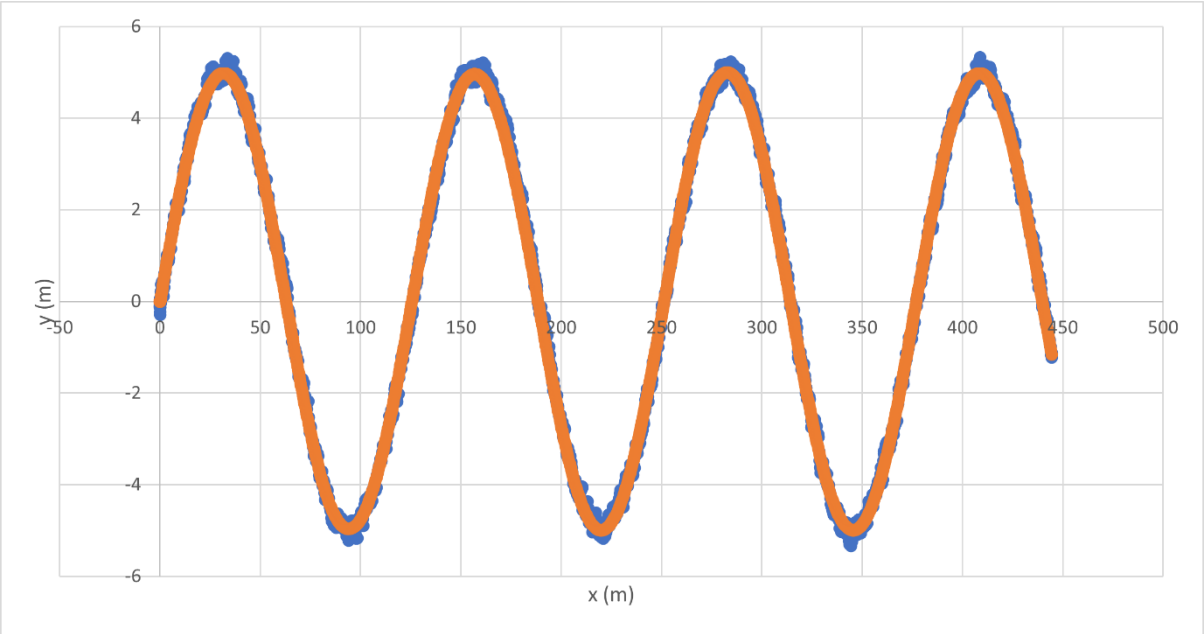

Figure 4.11: Comparison between the results of the extended Kalman filter and the ground truth. The orange line is the ground truth, the blue line is the result of the filter.

Again in Figure *4.11*, we cannot have much of an analysis, so we have to perform the same method as previously in order to determine the accuracy of our

filter. The average difference for the x coordinate is $0.184m$, and for the y coordinate, it is $0.128m$.

4.7 Real life experiments

After our experiments in the simulator, we devised some real-life experiments as well. We created three different experiments. In the first experiment we left the robot stationery for 10 minutes in order to find the average deviation for longitude and latitude. In the second experiment we left the robot stationary for 10 minutes then it moved in a straight line for 100 seconds and then stationary again for another 10 minutes. In the last experiment we had our robot moving in circles.

The robot we used is the for our experiments, is the same one we also used in the simulator. We have both the Jackal and the Kinova manipulator. Software wise, the robot is using the same ROS version as we used in the simulator, which is the ROS 1 noetic. The common ROS version meant that we could easily port our code from the simulator to the experiment by using a file transfer protocol (FTP) application.

Figure 4.12: Robot used for experiments.

Our experiments follow the same methodology as the one in the simulator. Using a python script we input linear and angular speed to the robot by publishing to the "/cmd_vel". In order to get the data from odometry we have to subscribe to the "odometry/filtered" topic where the position of the robot gets calculated using odometry and IMU sensors, then the data is also passed through a Kalman filter. All the data from the GPS module is taken by subscribing to the "/piksi/navsatfix_best_fix" topic.

4.7.1 Stationary robot

In this experiment the robot was left stationery for 10 minutes. The robot was given zero linear and angular speeds. Once the data was collected, we started processing. Initially we transformed the initial data from latitude and longitude to meters. Then we moved the starting point of the axes to the first point of our

data, effectively making the first point of our data the (0,0) of your cartesian coordinates. Below we present the data.

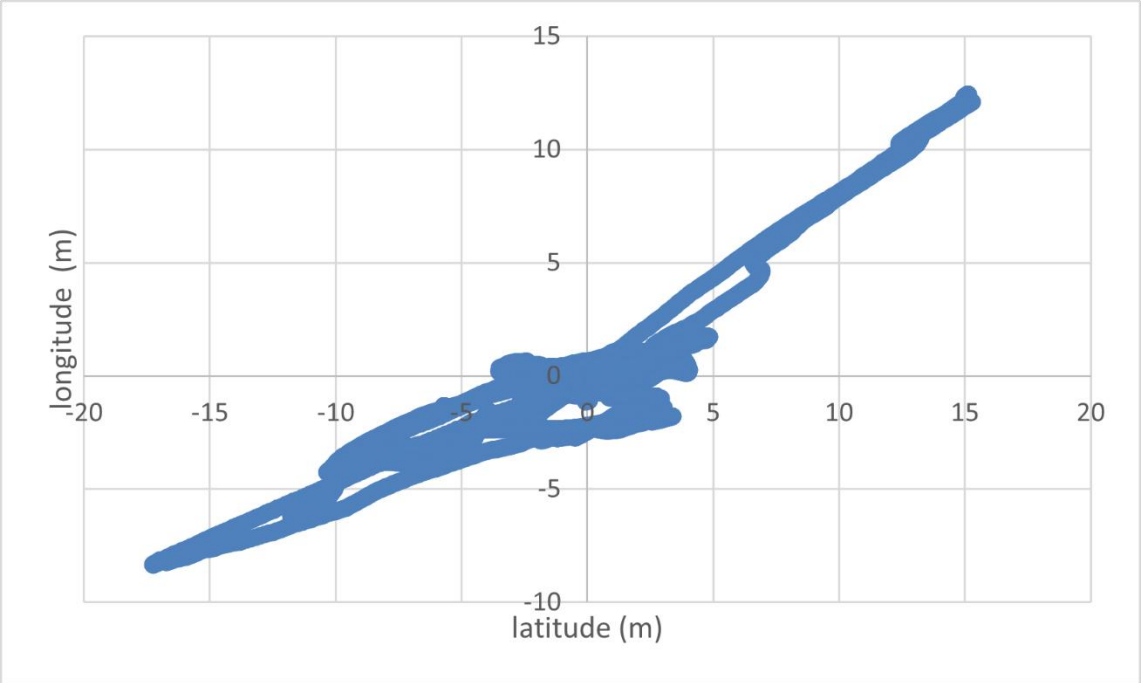

Figure 4.13: longitude and latitude of the stationary robot.

In Figure *4.13* we see that there is a range of recorded positions of the robot, even though the actual position of the robot never changed. After obtaining the data shown above, we calculated the average position of the robot at (-1.036, -0.513). After establishing the average position of the robot, we then calculated the average distance from that position. After the calculations, we found that GPS have an average of 3.713m deviation for latitude and 1.852m for longitude. With this experiment, we figured out the average error of the GPS, which was previously unknown.

4.7.2 Straight line

In this experiment, our robot was stationary for 10 minutes then in moved in a straight line and then stationary again. The goal behind this experiment is to make sure our Kalman filters work properly not only in the simulator but also in real life. The first obstacle in this case, is the fact that in contrast to the simulator we do not have the actual position of the robot in order to compare it to the results of the filter, as we did previously.

The solution to the missing ground truth data came in the form of the data collected from the "odometry_filtered" topic. In this topic odometry and IMU sensors are used in order to track the position of the robot. The final data are also passed through a Kalman filter so, the results are even closer to the actual position of the robot.

In Figure *4.14*, we present the data from the GPS. There are two areas with a lot of data, these are the cases where the robot is stationary, initial and final position, and there is a line connecting the two areas which is the part of the movement that the robot was moving in a straight line.

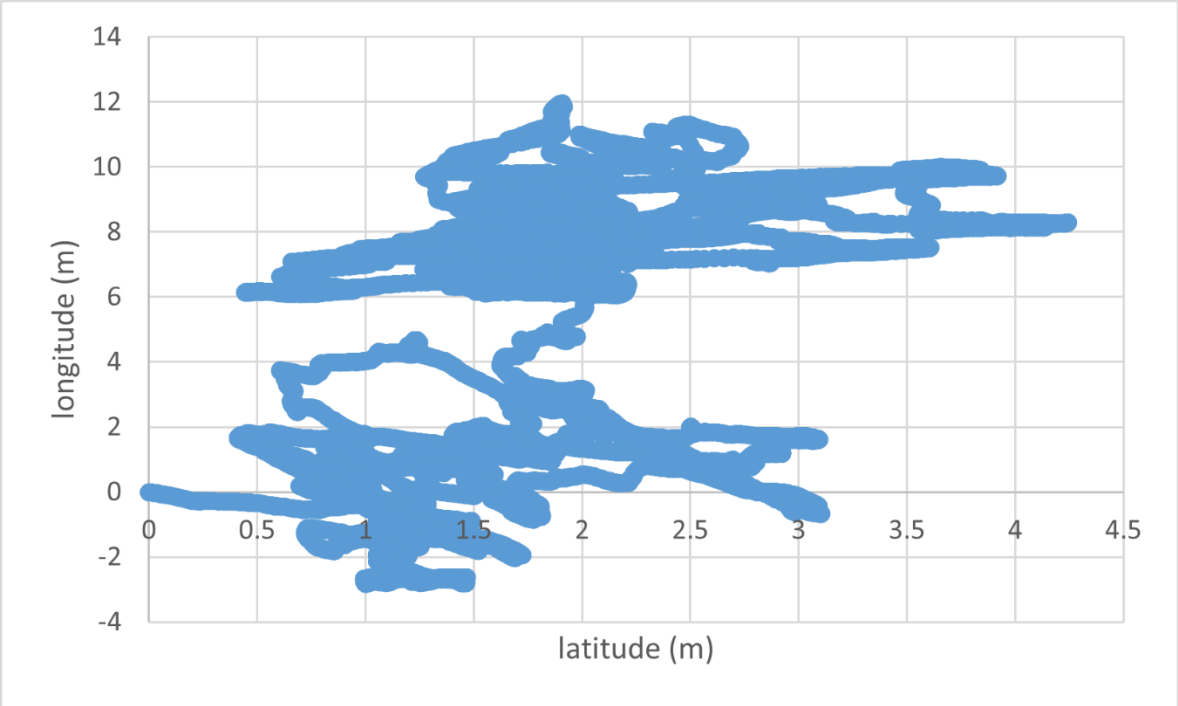

Figure 4.14: longitude and latitude of the robot for the second experiment.

In Figure *4.15* we present the data for the same movement as recorded from the odometry_filtered topic. The data from odometry presents the actual movement of the robot and this is verified from both the inputs given to the robot and what we observed during the experiment.

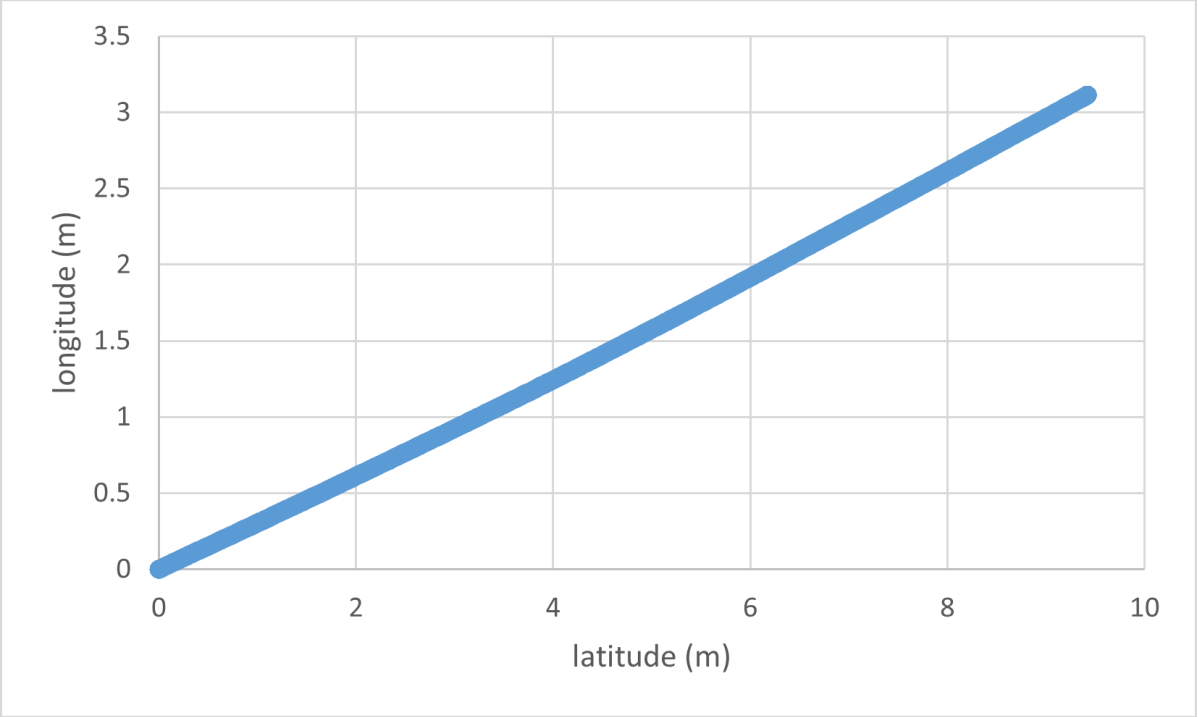

Figure 4.15: x and y coordinates of the robot recorded from the odometry_filtered topic.

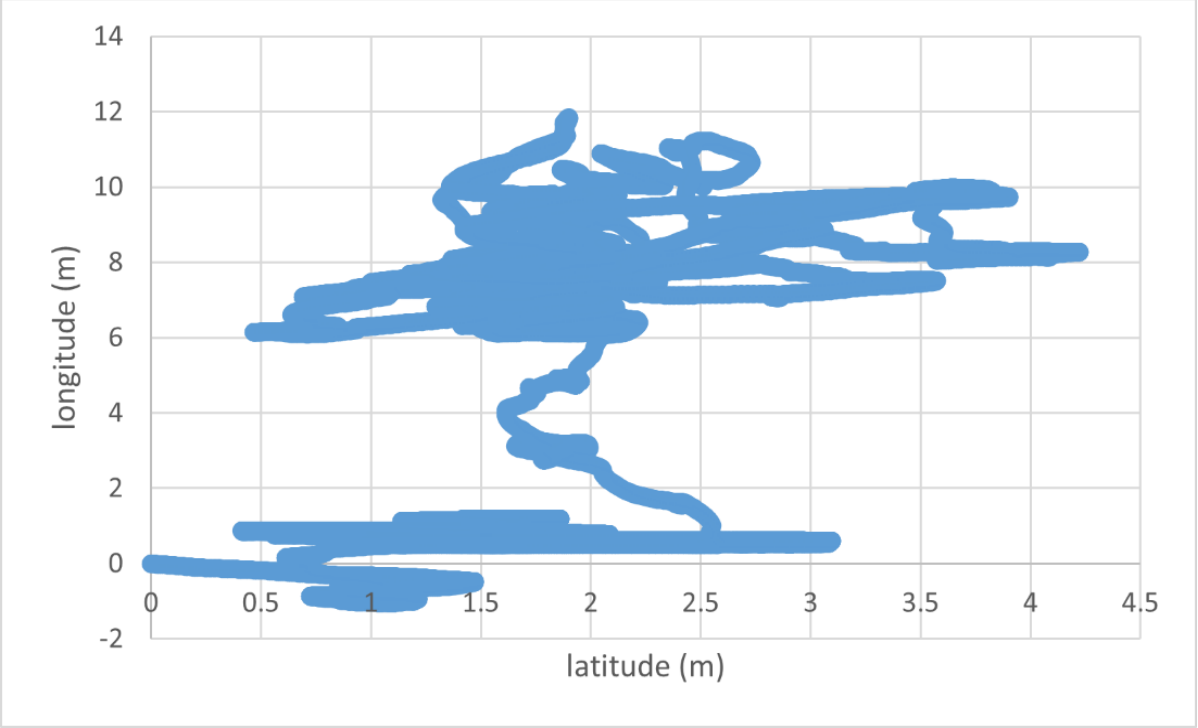

Figure 4.16: Visual representation of the EKF results

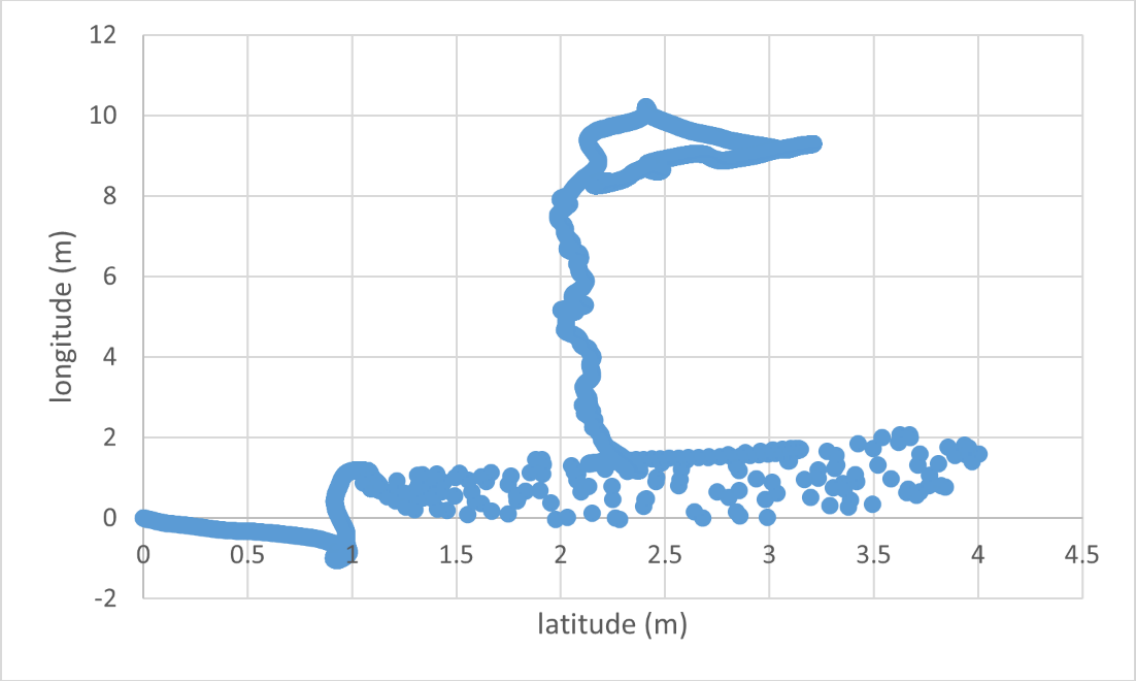

Figure 4.17: Visual representation of the UKF results

The average error for the data from the GPS were 1.3635m and 1.2083m for longitude and latitude. After we calculated the error of the raw GPS data, we used both EKF and UKF for the GPS data. The improvements we not big but were consistent as both filters yielded data with smaller errors. The EKF yielded 1.0666m and 1.2071m for longitude and latitude. Similarly, the UKF 0.7588m and 0.9131m for longitude and latitude.

4.7.3 Circles

In the last experiment, our robot is repeatedly moving in circles. The methodology is the same as in the second experiment. The first figures show the data from GPS and the second one shows the data from udometry.

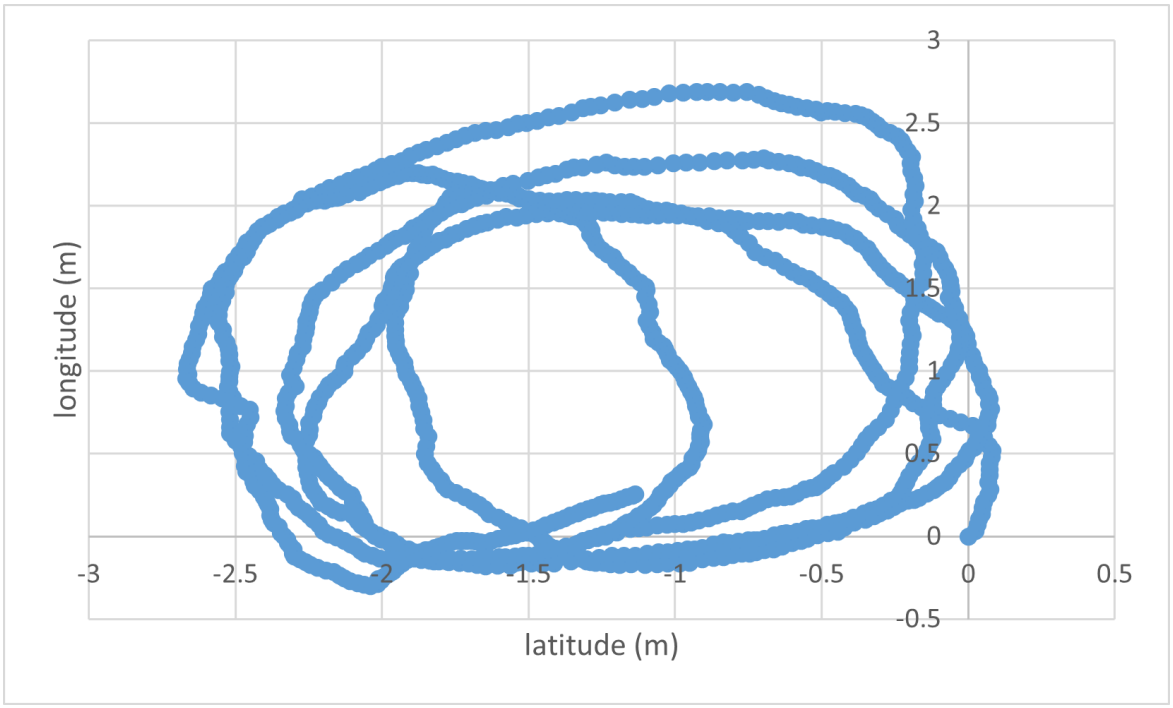

Figure 4.18: Longitude and latitude of the robot for the third experiment.

As seen in Figure *4.18*, the circular motion is clearly shown. If we were to compare it with the data from odometry, it is clear that there GPS data is noisy compared to the odometry ones.

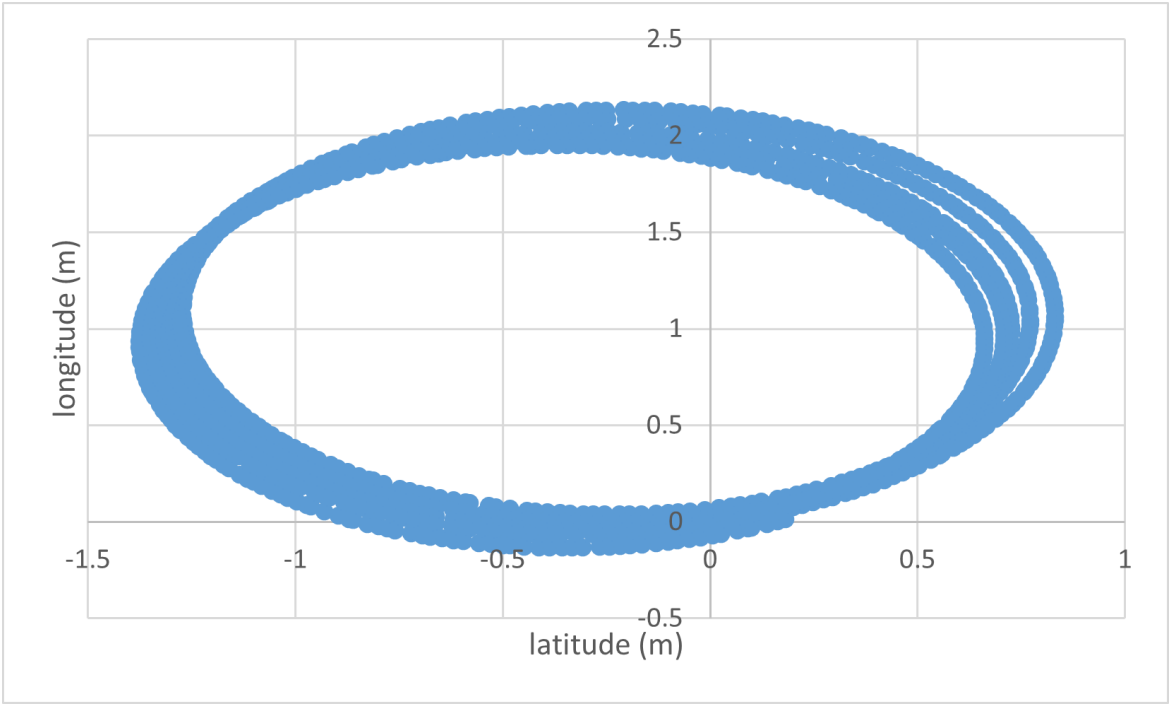

Figure 4.19: x and y coordinates of the robot recorded from the odometry_filtered topic.

In the two figures below, we have a visual representation of the Kalman filters. We start with the EKF and then with the UKF. In Both case we have trajectories closer to the one from the odomotry topic.

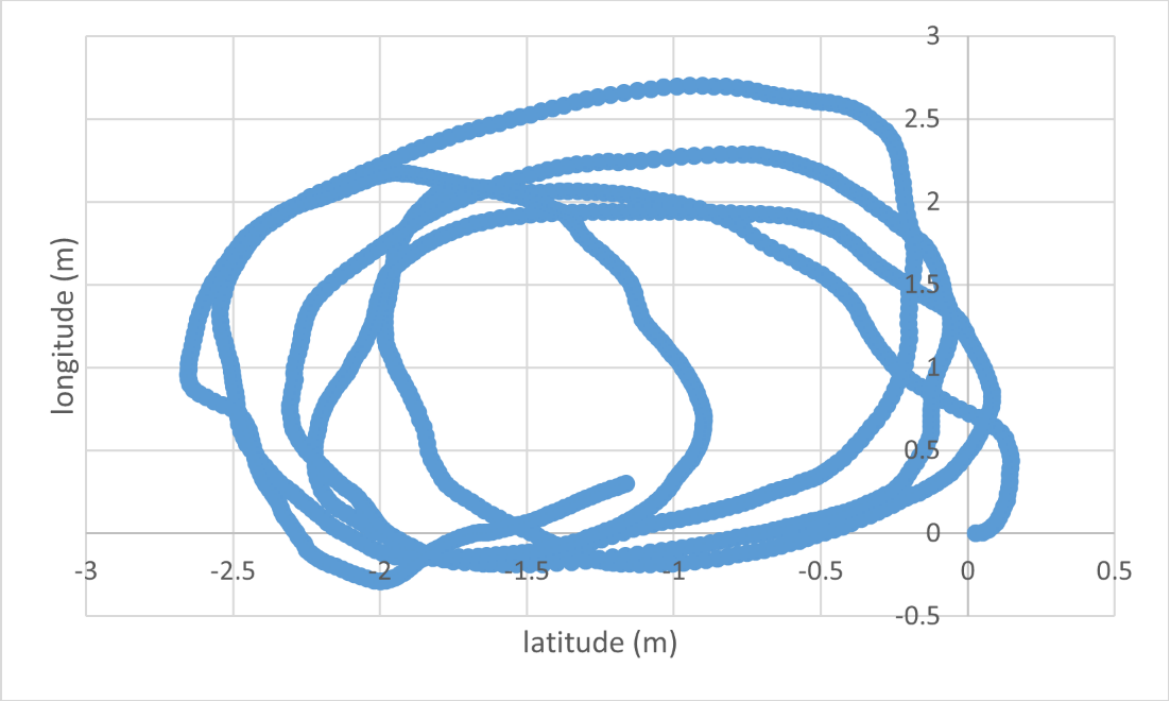

Figure 4.20: Visual representation of the EKF results.

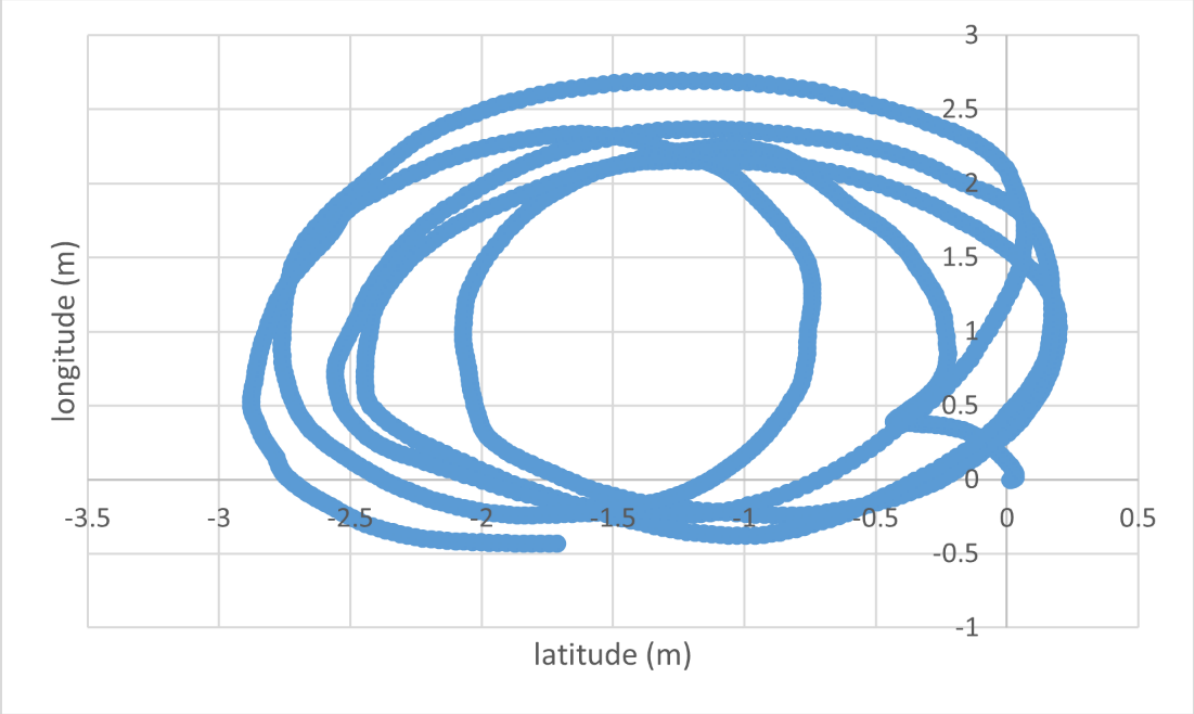

Figure 4.21: Visual representation of the UKF results.

As we did also with the second experiment, in order to evaluate our filters, we will be comparing the average errors compared to the error of just the raw GPS data. The average errors for the data from the GPS were 1.0879m and 0.6780m for longitude and latitude. After we calculated the error of the raw GPS data, we used both EKF and UKF for the GPS data. The EKF yielded 1.0831m and 0.6723m for longitude and latitude. Similarly, the UKF 1.0293m and 0.2742m for longitude and latitude. As noticed, the UKF for the circular movement had much better results as the UKF is specially designed for nonlinear cases.

CHAPTER 5

CONCLUSION

5.1 Conclusions

After everything is set and done, we want to sum up all our findings and share our conclusions with the reader. We started this paper presenting the methods we used for this thesis. After having talked about our simulation setup and our methodology, we went on and shared our results for all of them. The final numbers of the average deviation from the ground truth our methods ended up having are shown again below.

Table 5.1: Results of all the methods.

	x (m)	y (m)
ML	0.185	0.003
EKF	0.206	0.127
UKF	0.184	0.128

As we can see, the machine learning was two orders of magnitude closer to the ground truth for the y axis and performed similarly to the UKF in the x axes. This outcome could be better with more training, but this does not

necessarily belong to the scope of this thesis as we have already proved that the machine learning is more accurate than the two filters.

The results show that machine learning is a clearly better method for robot localization, but this is not the full story. It might be that the ML is more accurate, but one should also consider the effort required to achieve these kinds of results. In our case, we were using simulations in order to obtain the ground truth data, which make the process of obtaining them easy, but if we were to use an actual robot in real life, the process of obtaining the true pose of the robot would be far from trivial; An array of accurate sensors would be required, and the synchronization of them would also pose a big obstacle.

About the real-life experiments, the results that were yielded were positive. Both our filters performed better than just using the raw GPS data. Also, we fact that the average GPS error could be calculated, gave us an insight of how close the GPS data is to the truth. One more benefit of calculating the GPS error is that after this thesis this knowledge can be transferred to all the future projects with the same robot we used.

Bottom line, ML performed better than the Kalman filters, but there is no such thing as a free lunch. ML would need a great amount of effort in order to gather the ground truth data, and then also computer resources would be needed for training the model. On the other hand, both filters consist of a few hundred lines of code which once in place there is no more work to be done. Filters will contain errors inherently, but if the order of magnitude of the errors is known and fits the application, one would be incentivized to use them, as they are the more effortless and cheaper solution.

To conclude this thesis, machine learning performs better than Kalman filters. If accuracy is of most importance, one should utilize machine learning. If an estimate of the robot's position will suffice, both filters can perform well enough, making them a great solution for robot localization.

5.2 Future work

In the last section of this thesis, we want to share with the reader our ideas for future work. The are two main topics, the first one would be to use more machine

learning methods and Kalman filter variations to further enlarge the pool of methos tested, the second one would be to also establish a key performance indicator (KPI) for the effort needed for each method.

Adding new methods and comparing them to the existing ones would be truly interesting, especially for the machine learning methods. Using different ML methods could yield some impressively accurate data but then we should also consider the effort and the carbon footprint such a method would have. It is not a secret that ML has developed significantly in the last decades with the improvement in efficiency of computers, but the energy costs remain an issue. It is significantly cheaper than it used to be to run a computer, but a cluster of computers running a deep neural network is still something that creates a significant carbon footprint.

In order to have the full picture, further KPIs should be developed. For example, a KPI about the effort invested or the amount of funds invested would give the reader a more complete image and even more so, if they are interested in using some of these methods themselves.

BIBLIOGRAPHY

[1] Ö. ÇELİK, "A Research on Machine Learning Methods and Its Applications," *J. Educ. Technol. Online Learn.*, vol. 1, no. 3, pp. 25–40, 2018, doi: 10.31681/jetol.457046.

[2] A. Lykas and K. Blekas, "preprocessing." University of Ioannina, Ioannina, 2022.

[3] D. Maulud and A. M. Abdulazeez, "A Review on Linear Regression Comprehensive in Machine Learning," *J. Appl. Sci. Technol. Trends*, vol. 1, no. 2, pp. 140–147, 2020, doi: 10.38094/jastt1457.

[4] Developers Scikit-learn, "Linear Regression Example." https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py.

[5] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," *Proc. Siggraph Course*, vol. 8, Jan. 2006.

[6] Z. Cai and D. Zhao, "Unscented Kalman filter for non-linear estimation," *Geomatics Inf. Sci. Wuhan Univ.*, vol. 31, no. 2, pp. 180–183, 2006.

[7] Z. B. Rivera, M. C. De Simone, and D. Guida, "Unmanned ground vehicle modelling in Gazebo/ROS-based environments," *Machines*, vol. 7, no. 2, pp. 1–21, 2019, doi: 10.3390/machines7020042.

[8] I. Bagheri, S. Alizadeh, and E. Irankhah, "Design and Implementation of Wireless IMU-based Posture Correcting Biofeedback System," no. June, 2020.

Short Biography

Efstathios Rafailidis was born in 1997 in Katerini, Greece. He completed his undergraduate studies in physics at the university of Thessaloniki, in June of 2021. After his undergraduate studies, where he focused on nuclear physics, he found an interest in computer science. His interest in computer science translated to him enrolling for a master's degree in computer science and engineering from the university of Ioannina. At the same time as his master's degree, he had also started working as a software engineer.