UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

DRIFTY

Technical project Field and service robotics

Professor:
Prof. Fabio Ruggiero

Candidates:
Andrea Morghen P38000230
Salvatore Del Peschio P38000190

ANNO ACCADEMICO 2023/2024

Contents

Chapter 1

Introduction

This document is an in-depth continuation of a previous study [2], which examined issues related to hardware, control strategies, and simulation models.The goal of this project was to design and build a differential drive robot using accessible and cost-effective components, while ensuring high performance and reliability.

The report begins by providing a detailed description of the hardware components utilized in the project. This includes an overview of the mechanical design, electronic components, and the overall structural assembly.
Following the hardware description, the report delves into the analysis of state estimation techniques implemented to enhance the robot's navigational accuracy. These techniques include Euler approximation, higher-order Runge-Kutta methods, and Kalman filters. Each estimator is explored in detail, highlighting its mathematical foundation, implementation process, and advantages. The estimators were rigorously tested through the creation of a simulated model of the differential drive robot, assessing the accuracy of predictions and behavior in the presence of errors and noise. The simulation results are presented, comparing the performance of different estimation methods under various conditions.
Furthermore, the report focuses on trajectory planning. It discusses the methodologies employed to generate efficient and smooth trajectories for the robot, considering different velocity profiles, including linear, trapezoidal, and cubic profiles. The performance of these profiles is evaluated both in obstacle-free environments and in the presence of obstacles.

Chapter 2

Hardware

In this chapter, we will provide a detailed description of the hardware components used in our project. Through our design process, we carefully selected components that offered a good balance between performance and cost. For our differential drive robot, we utilized a Raspberry Pi Pico for the main processing unit, an HC-05 Bluetooth module for wireless communication, a GY-6500 IMU sensor for motion tracking, and two stepper motors for locomotion. All these components were mounted on a custom-designed Printed Circuit Board (PCB) to ensure a compact and organized layout. The robot is powered by batteries, allowing it to operate autonomously and perform its tasks without being tethered to an external power source.

2.1 CAD

The robot's design was partially inspired by an existing online project[1], but we made significant modifications to suit our specific needs. Using Autodesk Fusion 360, we redesigned key aspects such as the battery and PCB slots, as well as the attachment for the caster wheel. These customizations ensured that our robot could accommodate the necessary components while maintaining structural integrity and functionality. The entire robot was 3D printed and assembled by us. The final design is illustrated below:

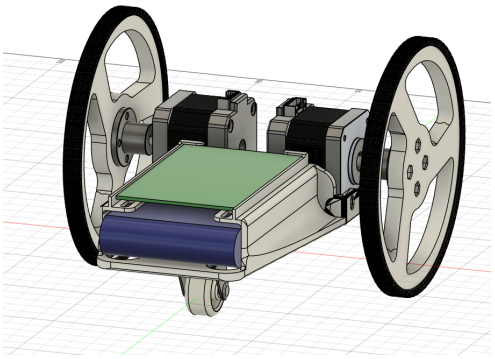

Figure 2.1: Cad Design

[1] [**link**] Remotely controlled - Arduino Self balancing robot

2.2 Components and PCB

In this section we will present all the electric components used in our system. The components constitute the subsystems for sensoring, communication, and actuation.

PCB

All components have been mounted on a Printed Circuit Board (PCB) to ensure compactness. The PCB was entirely designed by us and tailored to fit perfectly within the robot's structure, contributing to its compact and agile design. The electrical scheme for the project is as follows:

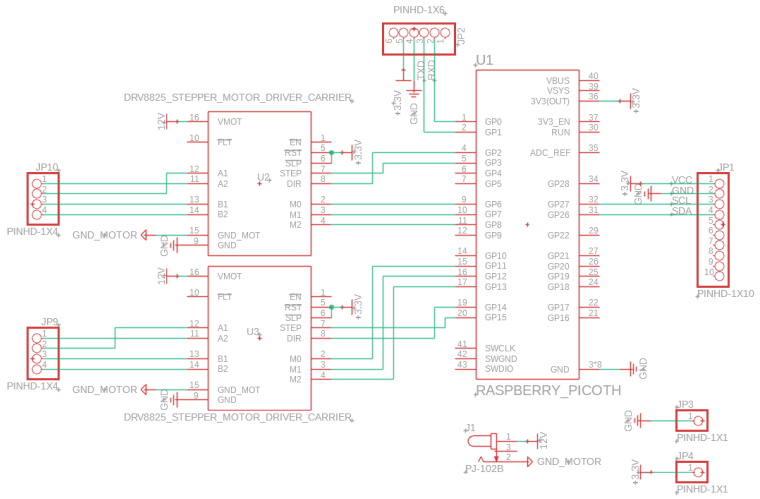

Figure 2.2: Electrical scheme

In the conceptual diagram, the different components are adequately connected via tracks. This scheme was implemented using Autodesk Fusion 360, which allowed us to create a detailed and precise design. We then transformed this scheme into a virtual PCB by adapting the paths of the various tracks into two layers, optimizing the layout for both performance and space efficiency while minimizing wiring complexity and potential points of failure. This process ensured that all connections were properly managed and that the final PCB design was compact and functional.

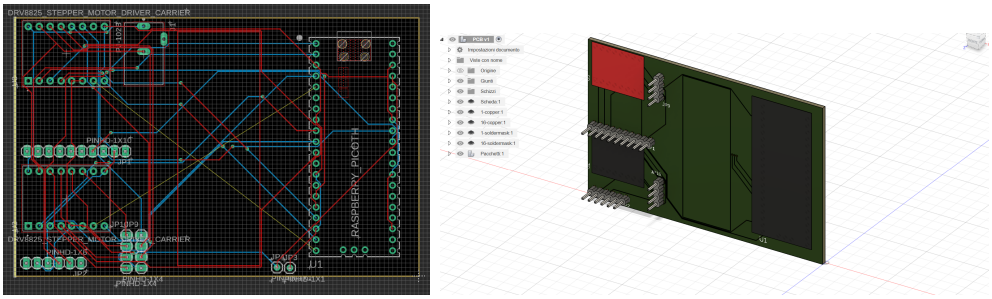

Figure 2.3: PCB routed scheme and 3d rendering

Rasberry pi Pico

The chosen microcontroller is a Raspberry Pi Pico, which is capable of handling all control operations, signal processing, and communication. It consists of an ARM Cortex-M0+ dual-core processor running at 133 MHz, 264 KB of SRAM, and 2 MB of onboard flash memory. It provides several multifunction GPIO pins and supports various interfaces, including SPI, I2C, UART, PWM, and ADC.

HC 05 Bluetooth Module

The HC-05 module is a Bluetooth device used to establish a wireless connection between the mobile robot and the GCS (ground control station). This module allows real-time data transmission, providing remote control and performance monitoring of the unicycle.

GY-6500 IMU Sensor

A GY-6500 IMU (Inertial Measurement Unit) has been used, consisting of an accelerometer and a gyroscope.The sensor uses an I2C interface to communicate with the Raspberry Pi Pico.

Stepper Motor

We used two stepper motors because they rotate in discrete steps, allowing for precise control of speed and angular position. Additionally, they are a budget-friendly choice, making the project more cost-effective. The motors provide the necessary torque for the robot movement, and their control is managed by a driver that receives control signals from the Raspberry Pi Pico.

2.3 Structure and Low level libraries

In this section, we discuss the low-level libraries and software components utilized in the project, which were implemented entirely in MicroPython using the OOP (object oriented programming) paradigm. In the development we utilized an interruption based programming to coordinate and synchronize the reading from the sensors and control operation, therefore without the use of busy waiting.

Differential Drive System

The core of the robot's movement is managed by a differential drive class. This object uses two stepper motors, each controlled by the Stepper class. The DifferentialDrive class performs the inverse kinematics calculations necessary for the actuation. By controlling the speed and direction of each motor, the robot can move forward, backward, and turn with precision.

Stepper Motor Control

The Stepper class provides a number of methods for controlling a stepper motor, allowing you to set the resolution, direction, speed, and move the motor to relative or absolute positions. The class automatically calculate the frequency of the steps for a given speed in meter per second given the resolution and the wheel diameter.

IMU

The robot's orientation and movement dynamics are tracked using an IMU. The IMU class interfaces with the MPU6500 sensor to obtain acceleration and gyroscopic data. This object is able to calculate roll, pitch, and yaw, providing real-time feedback on the robot's orientation. This class also includes methods for sensor calibration, the calibration is performed collecting the first hundred values with the robot in a still position. Then, We compute the average of these values, which represents the bias of the sensor. By adding the correct bias, the measurements will suffer from noise centered in 0 (white noise). In addition, for simulation purposes, we have obtained standard deviation to have an esteem the variability of the noise.

Localization

To determine the position and orientation of the robot, the UnicycleLocalization class employs odometry and IMU data. It supports several approximation methods: Euler, Runge-Kutta of the second and fourth order and two types of Kalman filters, which allow you to update the robot pose. This class periodically updates the estimated position using sensor readings and kinematic calculations, ensuring accurate fairly tracking of the robot's position. During simulations, the different types of estimator were compared in such a way as to undermine the best compromise between effectiveness and computational weight.

Bluetooth

Communication with external devices is handled by the Bluetooth class. It uses the UART protocol for sending and receiving data. This class enables remote control and monitoring, it transmits commands and receive status updates in a CSV format.

Controller

Two primary control strategies are implemented: Input-Output Linearization and Posture Regulation. Both control strategies utilize a feedback loop, periodically adjusting the robot's movements based on real-time sensor data and the desired trajectory.

Chapter 3

Estimator

3.1 Design the State Estimators

For our robot we will consider the position and orientation with their derivatives

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & \theta & \dot{\theta} \end{bmatrix}^\mathsf{T}$$

Five estimators has been implemented and compared:

- Euler integration
- Runge-kutta 2-order
- Runge-kutta 4-order
- Linear Kalman Filter
- Unscented Kalman Filter

The control input **u** is the commanded linear velocity and the commanded angular velocity or the x-velocity y-velocity and angular velocity (For the linear Kalman filter), this choice will be discussed in the following paragraphs:

$$\mathbf{u} = \begin{bmatrix} v & \omega \end{bmatrix}^\mathsf{T}$$

$$\mathbf{u} = \begin{bmatrix} \dot{x} & \dot{y} & \omega \end{bmatrix}^\mathsf{T}$$

To test the various types of estimators during the simulation phase, we created a digital twin of our differential drive. We estimated, with a rough approximation, the error in v and w through a repeated series of measurements. Our estimates led to the following result:

$$v_{\text{act}} = v_{\text{input}} + v_{\text{error\_mean}} + \text{randn}() \cdot v_{\text{std}}$$
$$w_{\text{act}} = w_{\text{input}} + w_{\text{error\_mean}} + \text{randn}() \cdot w_{\text{std}}$$

Where:

$$
\begin{aligned}
v_{\text{error\_mean}} &= -0.007 \\
v_{\text{std}} &= 0.003 \\
w_{\text{error\_mean}} &= -0.004 \\
w_{\text{std}} &= 0.0035
\end{aligned}
$$

The function randn() library that generates samples from a standard normal distribution (also known as a Gaussian distribution) with a mean of 0 and a standard deviation of 1.

3.2 Euler Approximation

Euler's approximation estimator applied to the kinematic model of a unicycle is a simple and effective method for predicting the future state of the system by discretizing the motion differential equations. Remember the kinematic model of the unicycle described before, we can define Euler's Approximation discretizing these equations over time.
Assuming that initial state (x_k, y_k, θ_k) at time t_k. We want to estimate the state $(x_{k+1}, y_{k+1}, \theta_{k+1})$ at time $t_{k+1} = t_k + T_s$, where T_s is the time step. The update equations using Euler's approximation are:

$$
\begin{cases}
x_{k+1} = x_k + v_k T_s \cos(\theta_k) \\
y_{k+1} = y_k + v_k T_s \sin(\theta_k) \\
\theta_{k+1} = \theta_k + \omega_k T_s
\end{cases}
$$

Euler's approximation is simple to implement and not computationally intensive but can introduce discretization errors, especially for systems with rapid or nonlinear dynamics. To improve accuracy, more advanced numerical integration methods, such as the Runge-Kutta method and Kalman filter, can be used. We have to notice that the choice of the time step T_s is crucial: a step that is too large can lead to significant errors, while a step that is too small increases computational cost.

3.3 Runge Kutta

Runge-Kutta methods are a family of iterative techniques for approximating solutions to ordinary differential equations (ODEs). These methods are highly valued for their balance between computational efficiency and accuracy. They provide a systematic procedure to estimate the value of an unknown function by considering multiple intermediate points within a single step.

3.3.1 Second-Order Runge-Kutta Approximation

The second-order Runge-Kutta method (RK2) improves upon the Euler method by considering the slope at the midpoint of the interval for a better approximation of the state at the next time step. This method can be expressed as follows:

$$\begin{cases} x_{k+1} = x_k + v_k T_s \cos\left(\theta_k + \frac{1}{2}\omega_k T_s\right) \\ y_{k+1} = y_k + v_k T_s \sin\left(\theta_k + \frac{1}{2}\omega_k T_s\right) \\ \theta_{k+1} = \theta_k + \omega_k T_s \end{cases}$$

where:

- x_k, y_k are the coordinates of the unicycle at time k.
- θ_k is the orientation angle at time k.
- v_k is the linear velocity.
- ω_k is the angular velocity.
- T_s is the sampling period.

The second-order Runge-Kutta approximation provides a more accurate state estimation for unicycle localization compared to simpler methods like Euler's method. Exploiting the controlled v and w values, this approach strikes a balance between computational simplicity and accuracy, making it suitable for various robotic applications. However, for more robust performance, integrating additional sensors and using a Kalman filter is recommended.

3.3.2 Fourth-Order Runge-Kutta Approximation

Additionally, the fourth-order Runge-Kutta method (RK4) has been implemented to improve the accuracy of state updates by considering function values at various points within the interval. The equations for the general case are as follows:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Applied to our case (unicycle model), the formulas become the following:

$$\begin{cases} x_{k+1} = x_k + \frac{T_s}{6}\left(v_k \cos(\theta_k) + 2v_k \cos\left(\theta_k + \frac{T_s}{2}\omega_k\right) + 2v_k \cos\left(\theta_k + \frac{T_s}{2}\omega_k\right) + v_k \cos\left(\theta_k + T_s\omega_k\right)\right) \\ y_{k+1} = y_k + \frac{T_s}{6}\left(v_k \sin(\theta_k) + 2v_k \sin\left(\theta_k + \frac{T_s}{2}\omega_k\right) + 2v_k \sin\left(\theta_k + \frac{T_s}{2}\omega_k\right) + v_k \sin\left(\theta_k + T_s\omega_k\right)\right) \\ \theta_{k+1} = \theta_k + T_s\omega_k \end{cases}$$

The RK4 method offers significant advantages in terms of accuracy compared to lower-order methods like the Euler method or the second-order Runge-Kutta method. However, it is more computationally intensive, requiring four function evaluations for each time interval. It will be shown in the simulation paragraph how the difference between RK2 and RK4 is negligible for our case. this difference depends on the non-linearity of the model that in our case has weak non-linearities.

3.4 Kalman Filters

Kalman Filters are a class of algorithms that provide efficient and recursive solutions to the problem of estimating the state of a dynamic system from knowledge of the system model and a series of noisy measurements. It combines two phases:

- **prediction step**, next state prediction using system model
- **update step**, correction of prediction using information contained in sensor measurements

next state prediction using system model update, correction of prediction using information contained in sensor measurements.

For our system, we decided to implement two types of Kalman Filters. The sensors at our disposal provided possible measurements during the update step, specifically ω (angular velocity along the z-axis of the IMU) and $\dot{v}$ (linear velocity along the x-axis of the IMU). The first measurement is already used for the prediction step via the Runge-Kutta method, so an update based on it would not bring significant improvements.

With future developments in mind, we envisioned placing a camera (an additional sensor) above the movement area. This camera would periodically provide measurements of the x and y coordinates of the differential drive robot using computer vision algorithms. We assumed the measurement error to be centered at 0, with a standard deviation of 5 cm for both coordinates.

3.4.1 Linear Kalman filter

To use the linear Kalman filter you need to make a change to the model, as it has non-linearities. Such non-linearities can be taken out of the model with a change of input signals:

$$\mathbf{u} = \begin{bmatrix} \dot{x} & \dot{y} & \omega \end{bmatrix}^\mathsf{T}$$

The measurement from the camera with a period of one second:

$$\mathbf{z} = \begin{bmatrix} x & y \end{bmatrix}^\mathsf{T}$$

Given the state vector $\mathbf{x}_k$ at time k, the state transition equation is described by $\mathbf{x}_k = \mathbf{A}_{k-1}\mathbf{x}_{k-1} + \mathbf{B}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1}$, where $\mathbf{A}$ is the state transition matrix, $\mathbf{B}$ is the control matrix, $\mathbf{u}$ is the control input, and $\mathbf{w}$ represents process noise,also ,associated with the state x there is the matrix $\mathbf{P}$ which represents the error covariance matrix. This matrix plays a crucial role in the functioning of the filter, as it quantifies the uncertainty associated with state estimation. The measurement equation is $\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k$, where $\mathbf{H}$ is the observation matrix and $\mathbf{v}$ represents measurement noise.

The recursive equations are:

1. **Prediction**

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{A}_{k-1}\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_{k-1}\mathbf{u}_{k-1}$$
$$\mathbf{P}_{k|k-1} = \mathbf{A}_{k-1}\mathbf{P}_{k-1|k-1}\mathbf{A}_{k-1}^\top + \mathbf{Q}_{k-1}$$

2. **Update**

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k^\top (\mathbf{H}_k \mathbf{P}_{k|k-1}\mathbf{H}_k^\top + \mathbf{R}_k)^{-1}$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k|k-1})$$
$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1}$$

3.4.2 Unscented Kalman Filter

The Unscented Kalman Filter (UKF) is an extension of the Kalman Filter for nonlinear systems. It approximates the state distribution using a set of carefully chosen sample points called sigma points. These points capture the mean and covariance of the state distribution and are propagated through the nonlinear system equations. For a state vector $\mathbf{x}_k$, the UKF uses the unscented transform to generate sigma points $\mathbf{X}_{k-1}^{(i)}$ and their corresponding weights $W^{(i)}$.
The steps are:

1. **Sigma Point Generation**, a finite number of points are generated according to a certain criterion (Van der Merwe's Scaled Sigma Points)

2. **Prediction**, the transition function is applied to every sigma point and the state is reconstructed by a weighted sum

3. **Update**, the correction is made in the measurement space tto avoid the occurrence of measurement conversion error

These steps provide the UKF's ability to handle non-linearity more effectively than the traditional Kalman Filter.For a more complete treatment of the Kalman filter and its non-linear versions implemented in python we have taken as a reference the following MIT paper by Roger R. Labbe **Kalman-and-Bayesian-Filters-in-Python** [1]

3.4.3 Simulation comparison

Below are shown the simulation results for two different trajectories (the first simpler, the second more complex). All the simulations were carried out with an I/O Linearization controller (K1=1, K2=1) and at a frequency of 10 Hz with a cubic polynomial profile (with velocity at path point 0). Subsequently, the performance of the different estimators for both trajectories was compared.

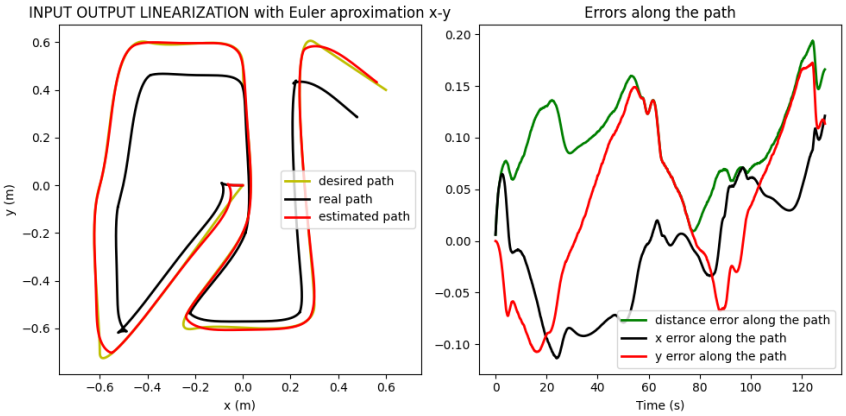

Figure 3.1: Euler

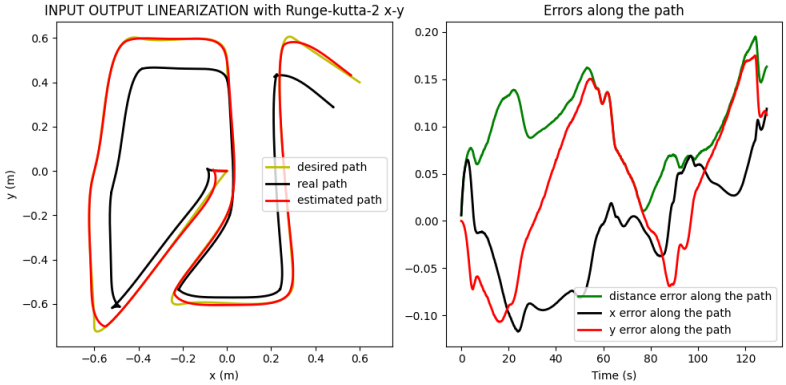

Figure 3.2: Runge Kutta 2

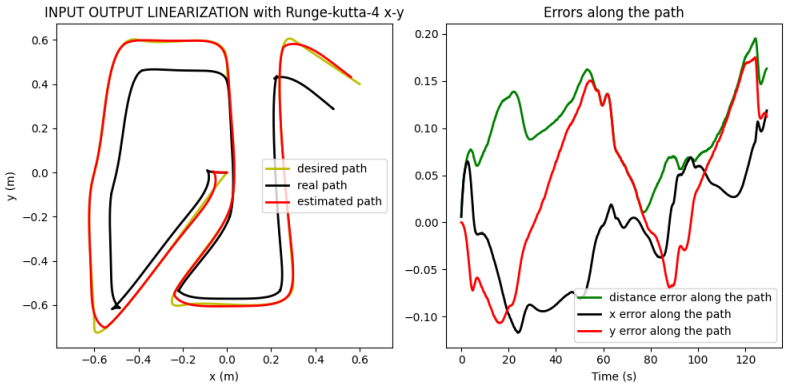

Figure 3.3: Runge Kutta 4

It is interesting to note that for our model, Euler, Runge-Kutta 2, and Runge-Kutta 4 methods provide very similar estimates. Specifically:

Maximum difference between RK-2 and Euler: 0.00353
Maximum difference between RK-4 and Euler: 0.00353
Maximum difference between RK-4 and RK-2: 1.60478 e-05

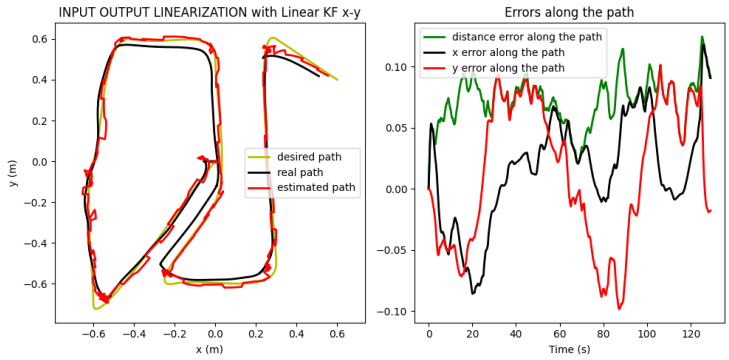

Figure 3.4: Kalman Filter

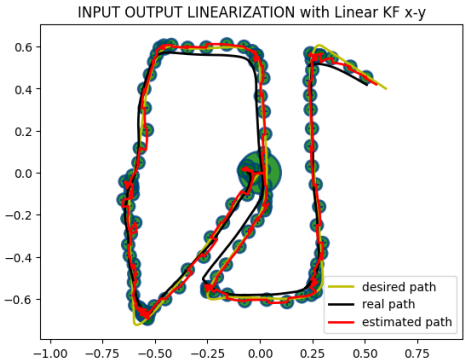

Figure 3.5: Variance Kalman Filter

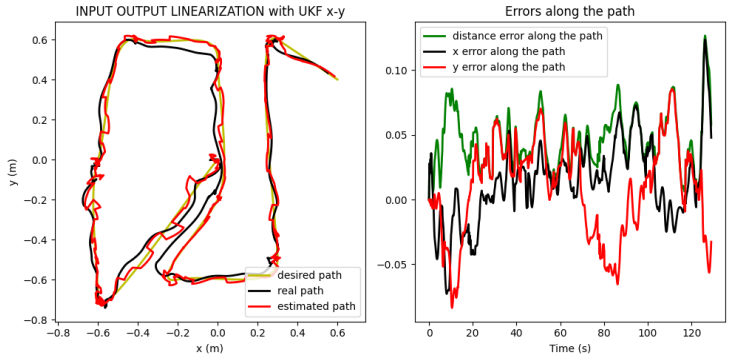

Figure 3.6: Unscented Kalman Filter

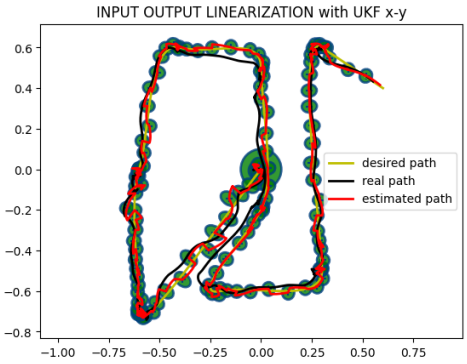

Figure 3.7: Variance of Unscented Kalman Filter

It is interesting to note how the differential drive stays very close to the desired path along its route but ends up short of the final point. To understand what is happening, we plotted the desired velocities given as inputs to the controller and those computed by the controller itself:

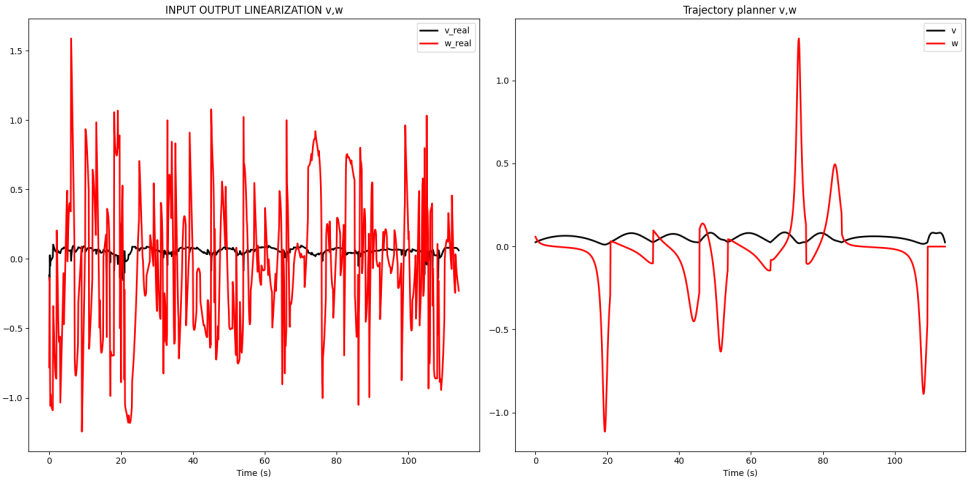

Figure 3.8: Kalman Filter

As expected, they are different. This is normal because, due to disturbances and noise in the system, the robot's controller finds itself in unexpected positions and has to make turns to get back on track. Because of these trajectory changes, the robot loses time relative to the desired path and finishes its simulation without reaching the goal.
Let's compare the euclidean errors along the path with the different estimators:

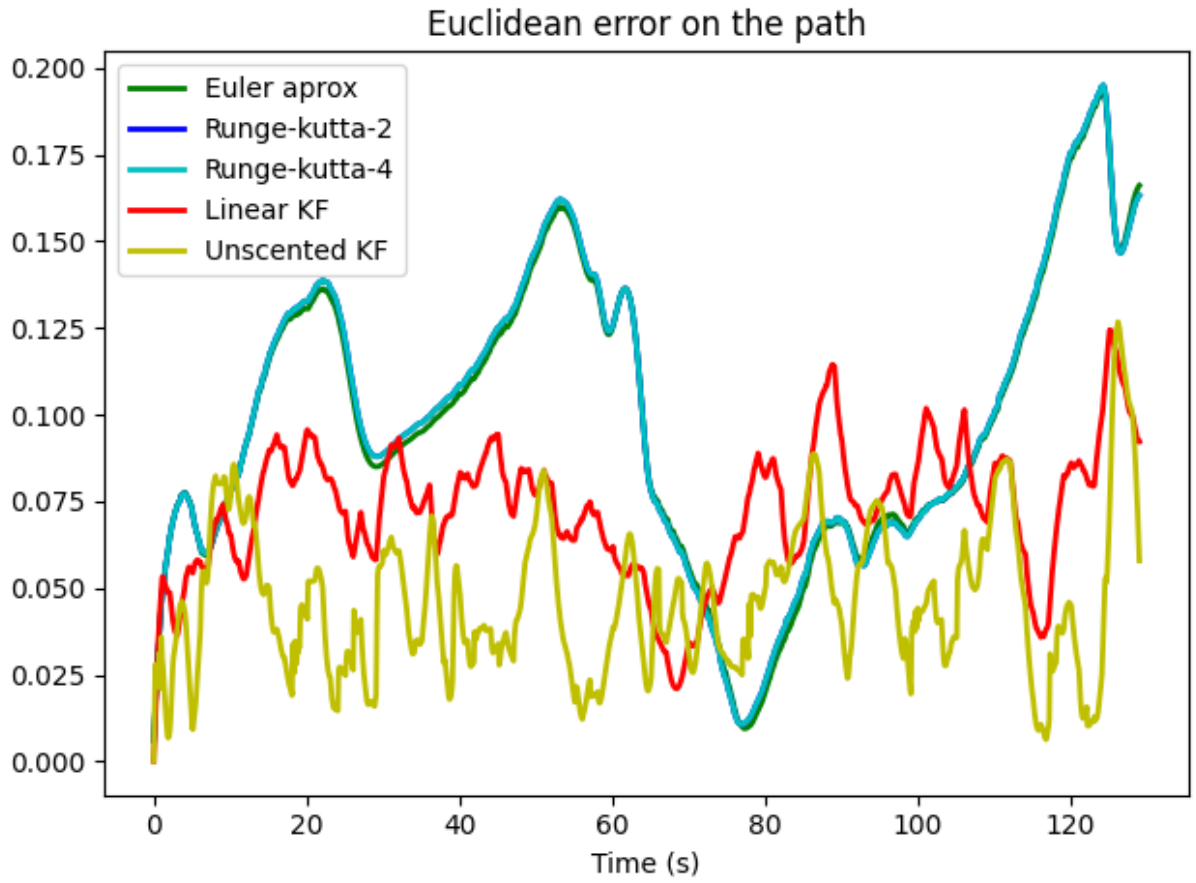

Figure 3.9: Kalman Filter

In the KF and UKF simulations the presence of an absolute measurement (such as x and y) prevents the estimator from diverging, unlike what occurs with Euler, RK2, and RK4.

Similarly to the previous case, simulations were also performed with a more complex and more longer trajectory.

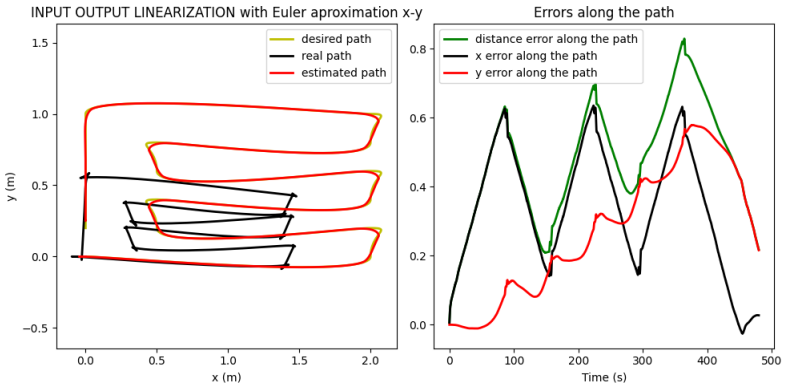

Figure 3.10: Euler

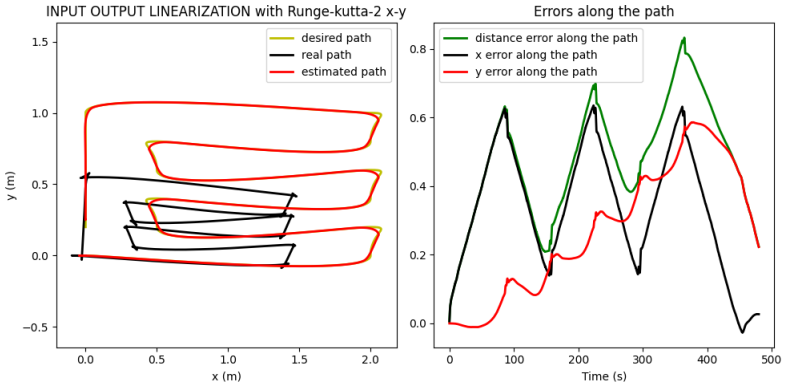

Figure 3.11: Runge Kutta 2

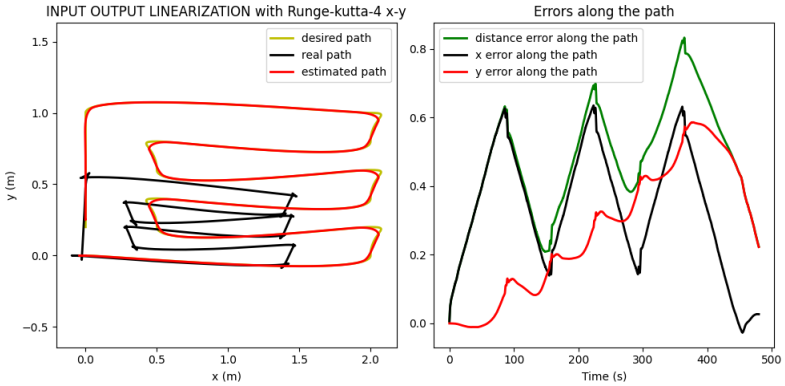

Figure 3.12: Runge Kutta 4

In this case, it is even clearer how algorithms such as Euler, Runge-Kutta 2, and Runge-Kutta 4 for prolonged trajectories are not sufficient to compensate for disturbances and errors (especially regarding errors in linear velocity). Considering the behaviour for both trajectories, we can conclude that the use of estimators such as Euler, RK2 and RK4 allows us to achieve extremely similar performance. Simulations were carried out with several f_s both smaller and larger than 10 and this result persists. For this trajectory:

Maximum difference between RK-2 and Euler: 0.00698
Maximum difference between RK-4 and Euler: 0.00695
Maximum difference between RK-4 and RK-2: 2.84131e-05

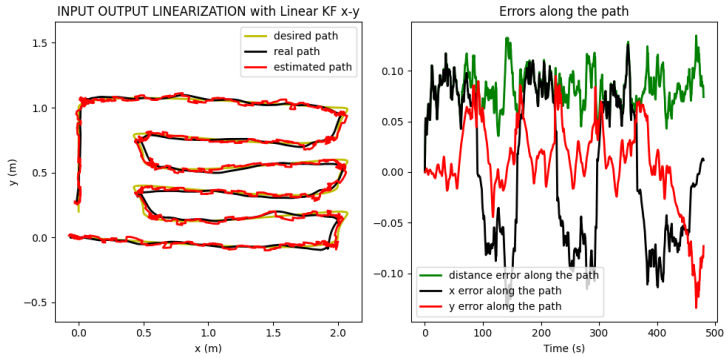

Figure 3.13: Kalman Filter

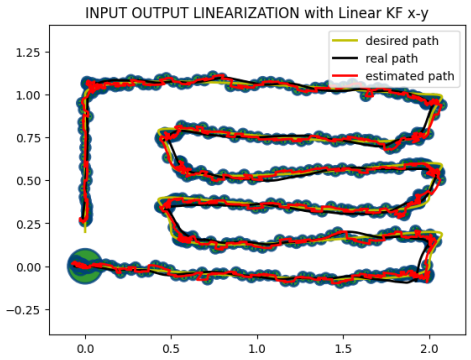

Figure 3.14: Variance Kalman Filter

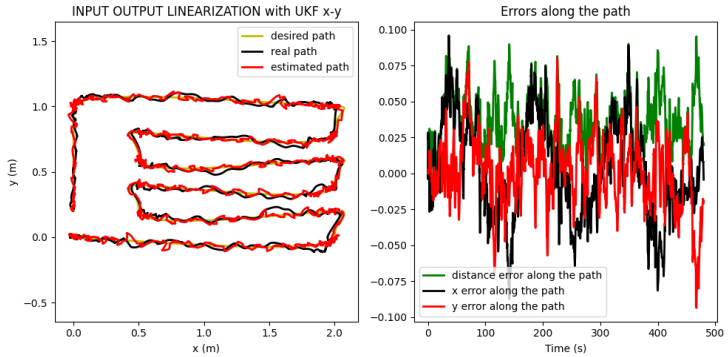

Figure 3.15: Unscented Kalman Filter

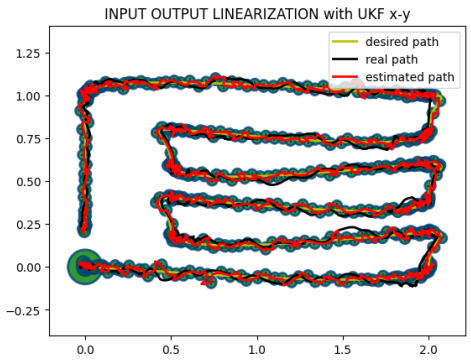

Figure 3.16: Variance of Unscented Kalman Filter

Both Kalman filters continue to perform correctly for longer trajectories despite the persistent problem of trajectory delay, which actually increases with the length of the trajectory.

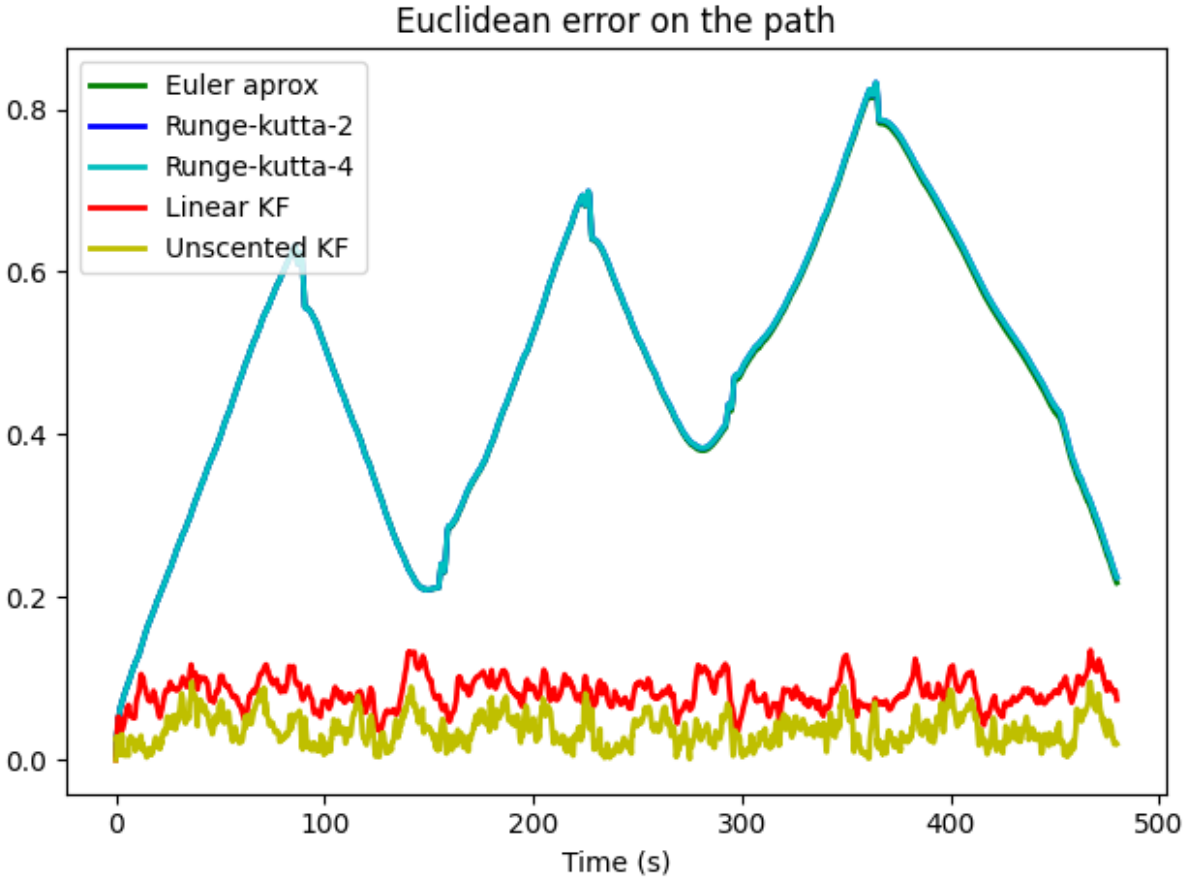

Figure 3.17: Kalman Filter

The performance of the Unscented Kalman Filter is superior at the cost of less smooth behavior. For long trajectories, it is evident that the Kalman filter allows us to achieve significantly better performance compared to Euler, Runge-Kutta 2, and Runge-Kutta 4.

Chapter 4

Trajectory Planning

Trajectory planning involves generating a series of points that connect an initial point to a final point, following a path with specific characteristics. This section will describe the process of generating Cartesian trajectories using various velocity profiles for the abscissa: linear, trapezoidal, and cubic. The section will briefly review these velocity profiles and the concept of Cartesian trajectory generation. Subsequently, it will present the results obtained by generating paths in space using the motion planning algorithms described in the first report of this work. The simulated trajectories will be described before being executed in a real test. In addition, to showcase Drifty's obstacle avoidance capabilities, several obstacles will be added to the environment.

4.1 Velocity Profiles

In trajectory planning, velocity profiles determine how the robot's speed changes over time as it moves along the path. Different velocity profiles can affect the smoothness and efficiency of the trajectory. Below are the three velocity profiles used in this study. All profiles are normalized such that $s(t)$ ranges from 0 to 1 over the trajectory segments considered.

4.1.1 Linear Profile

The linear profile assumes a constant velocity until the final time t_f:

$$s(t) = \frac{t}{t_f}$$

Since the velocity is constant, the path is predictable and smooth. However, in cases where the motion planning generates segments with small angles between them, requiring the robot to rotate quickly in a very tight space, these profiles may not be ideal due to the need to decelerate at those points.

4.1.2 Trapezoidal Profile

The trapezoidal profile includes an acceleration phase, a constant velocity phase, and a deceleration phase. It is defined as:

$$s(t) = \begin{cases} \frac{1}{2}at^2 & 0 \leq t < t_a \\ at_a(t - \frac{t_a}{2}) & t_a \leq t < t_a + t_c \\ 1 - \frac{1}{2}a(t_f - t)^2 & t_a + t_c \leq t \leq t_f \end{cases}$$

Since the duration of the acceleration phases can be controlled, this profile is versatile. It allows for adjusting the acceleration duration when the trajectory includes sharp turns.

4.1.3 Cubic Profile

The cubic profile ensures smooth acceleration and deceleration, providing a more natural motion. It is expressed as:

$$s(t) = a_3t^3 + a_2t^2 + a_1t + a_0$$

the velocity profile of the abscissa will be parabolic:

$$\dot{s}(t) = 3a_3t^2 + 2a_2t + a_1$$

and the acceleration profile linear

$$\ddot{s}(t) = 6a_3t + 2a_2$$

Four coefficients are available, it is possible to impose, besides the initial and final value also the initial and final velocity value, two cases will be analyzed: $\dot{s_i} = \dot{s_f} = 0$ and $\dot{s_i} = \dot{s_f} = 0.1$. The specific trajectory will be given by the solution to the following system of equations:

$$\begin{aligned} a_0 &= s_i \\ a_1 &= \dot{s}_i \\ a_3t_f^3 + a_2t_f^2 + a_1t_f + a_0 &= s_f \\ 3a_3t_f^2 + 2a_2t_f + a_1 &= \dot{s}_f, \end{aligned}$$

4.2 Cartesian polynomials trajectory

To generate a Cartesian trajectory between an initial point q_i and a final point q_f, third-degree polynomials are used to determine the coordinates $x(t)$ and $y(t)$. These polynomials ensure smooth transitions and can be adjusted for specific path characteristics. The equations for the coordinates $x(t)$ and $y(t)$ are:

$$x(t) = s(t)^3x_f - (s(t) - 1)^3x_i + \alpha_x s(t)^2(s(t) - 1) + \beta_x s(t)(s(t) - 1)^2$$

$$y(t) = s(t)^3y_f - (s(t) - 1)^3y_i + \alpha_y s(t)^2(s(t) - 1) + \beta_y s(t)(s(t) - 1)^2$$

where

$$\alpha_x = k\cos\theta_f - 3x_f$$
$$\alpha_y = k\sin\theta_f - 3y_f$$
$$\beta_x = k\cos\theta_i + 3x_i$$
$$\beta_y = k\sin\theta_i + 3y_i$$

These polynomials allow for flexible and precise control over the trajectory, accommodating the specific needs of different paths and scenarios.

4.2.1 Trajectory Generation Without Obstacles

In an environment free of obstacles, we generate various trajectories using the PRM method. These trajectories are first simulated and then tested in real conditions. These simulations were conducted without a systematic error (the errors have zero mean value). In the case of the estimators, we wanted a realistic worst-case scenario, which in this instance is not helpful. Our goal here is simply to observe the behavior as the type of generated trajectory varies. The path images are displayed as they appear on Pygame to provide a clearer understanding of the simulation videos. It is important to note that the coordinates (0,0) are located at the top left, causing the simulation graphs to be flipped vertically. Additionally, the trajectories shown below have been appropriately scaled (using a **trajectory scaling algorithm**) to comply with the imposed maximum speed.

Linear Profile

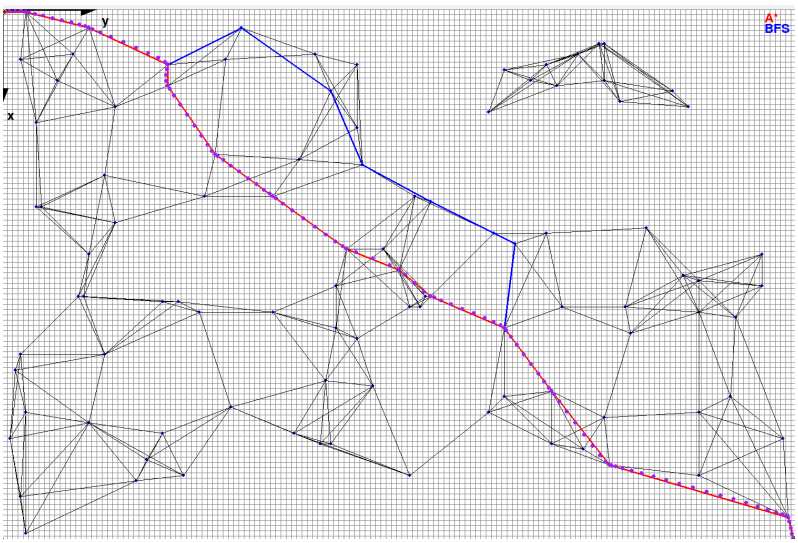

Figure 4.1: PRM trajectory with a linear profile of velocity

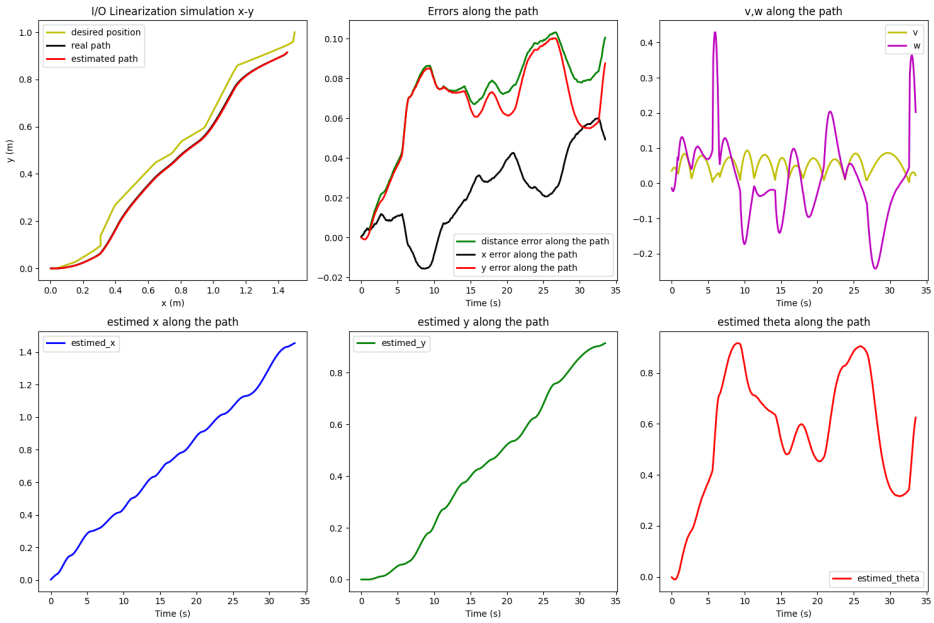

Figure 4.2: Simulation of the trajectory with a linear profile

From the simulation, it can be observed that the trajectory tends to follow the desired path quite faithfully, with a slight displacement due to very low k values (k1=k2=0.01). These k values for the controller were chosen because they performed better on the physical robot.

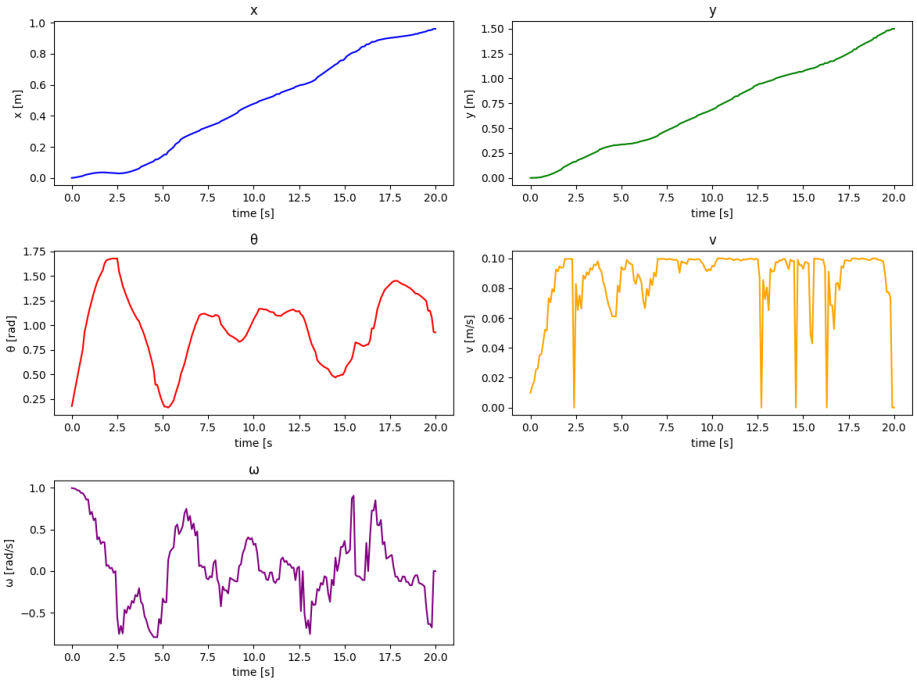

Figure 4.3: Test of the trajectory with a linear profile

The previously presented simulations differ from the real ones[1] only in having slightly lower values of ω and v. This is because the simulation model does not account for constant disturbances. The linear profile moves at a constant speed and, therefore, has better performance on the real hardware compared to the other profiles. This is because the other speed profiles, which approach near-zero speeds, tend to stall.

Trapezoidal Profile

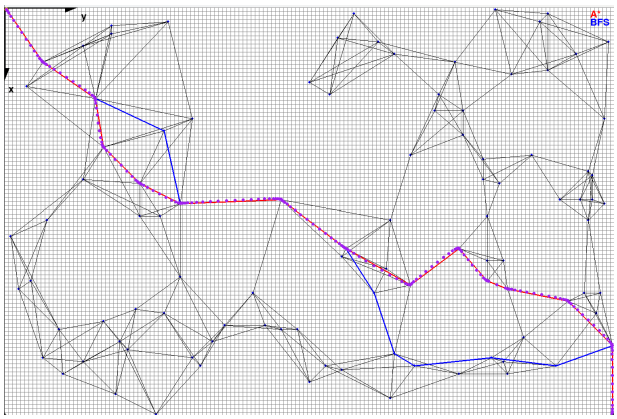

Figure 4.4: PRM trajectory with a trapezoidal profile of velocity

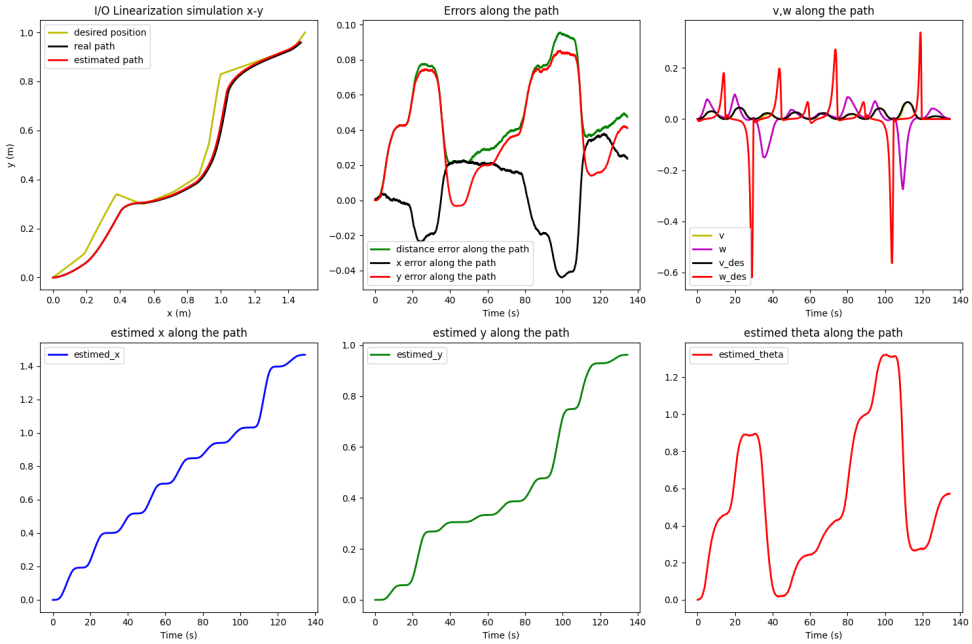

Figure 4.5: Simulation of the trajectory with a trapezoidal profile

Trajectories with a trapezoidal profile are characterized by less smooth behavior. Additionally, we can observe[2] that they exhibit a fairly recognizable pattern in their velocities.

[1] [**video**] PRM with a linear velocity profile test
[2] [**video**] PRM with a trapezoidal velocity profile test

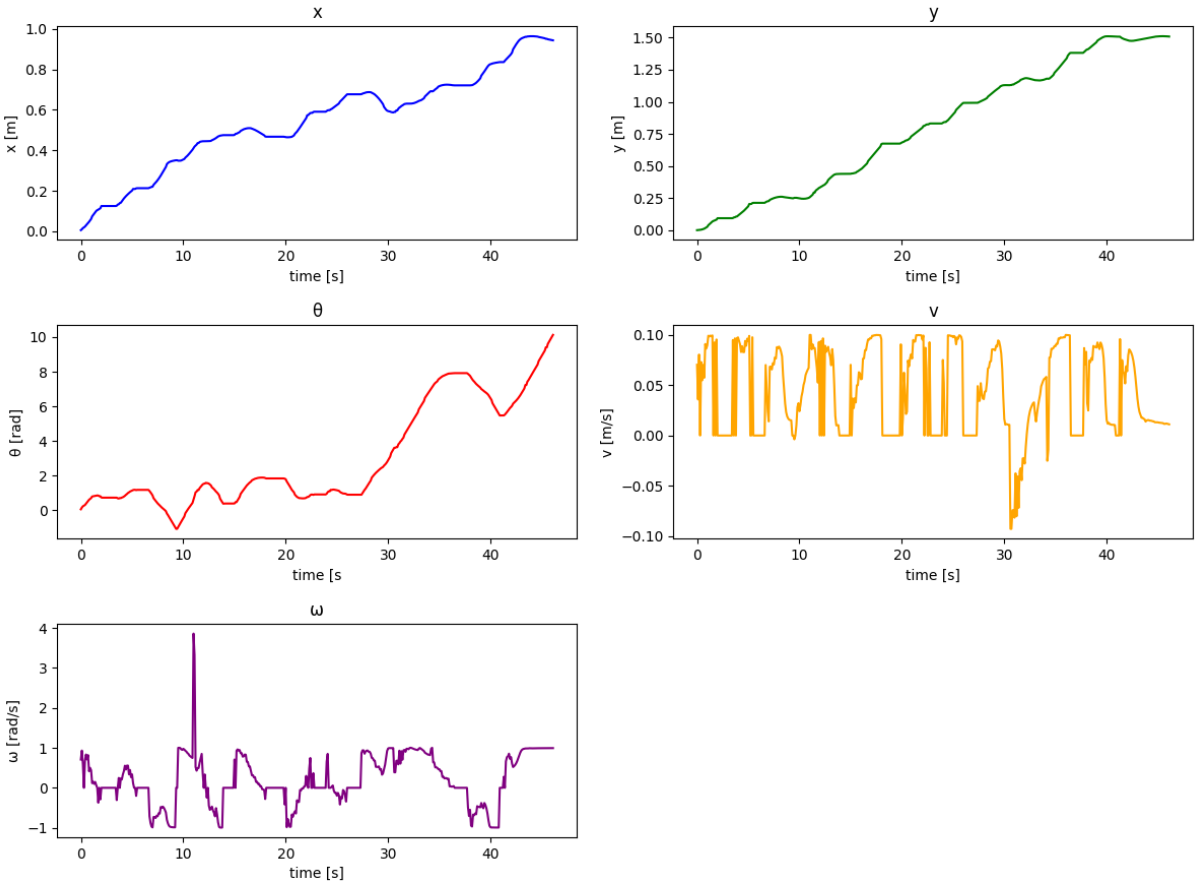

Figure 4.6: Test of the trajectory with a trapezoidal profile

Cubic Profile

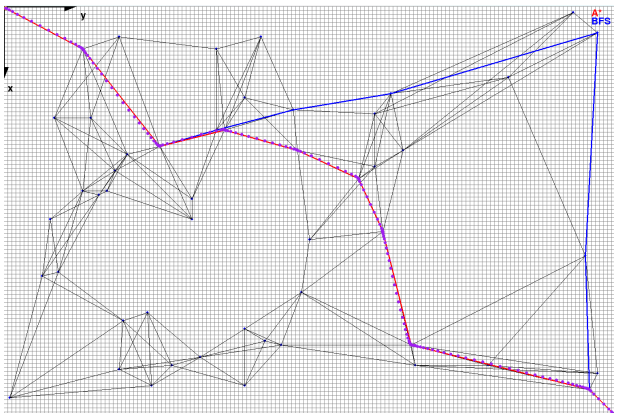

Figure 4.7: PRM trajectory with a cubic profile of velocity

The cubic trajectory works very well in simulation but does not perform as well in real-world tests[3]. As can be seen from the simulation videos, the linear profile proves to

[3] [**video**] PRM with a cubic velocity profile test

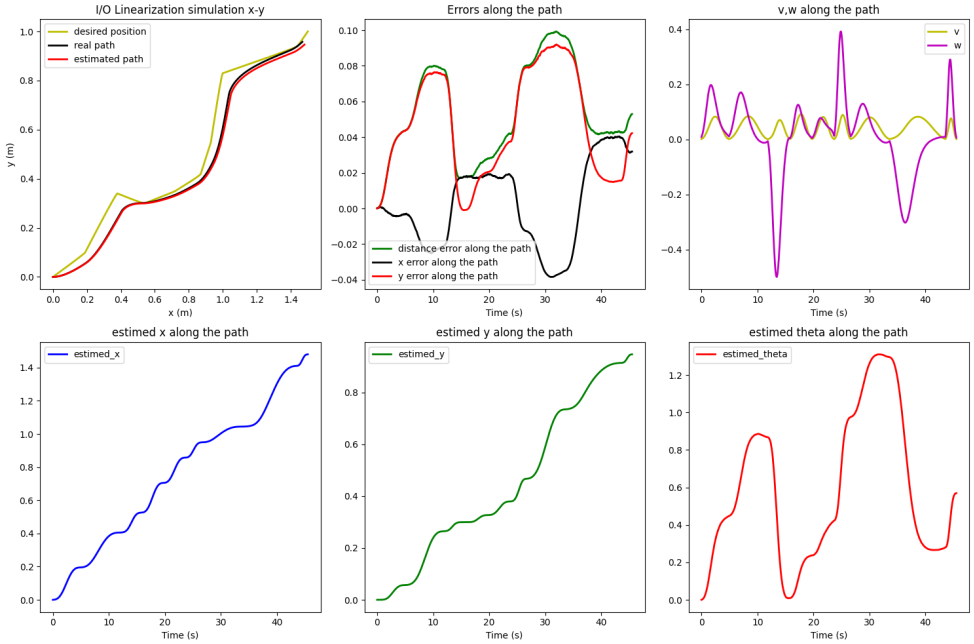

Figure 4.8: Simulation of the trajectory with a cubic profile

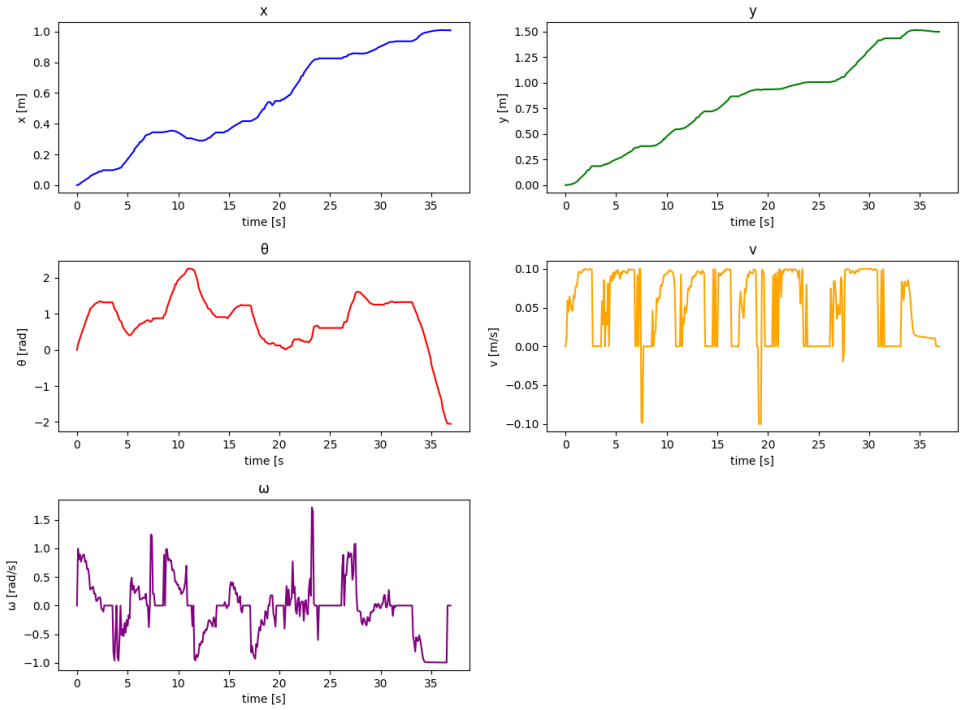

Figure 4.9: Test of the trajectory with a cubic profile

be the best in reality. Additionally, the trajectory generated with the cubic method is characterized by passing through the path points at zero velocity.

4.2.2 Trajectory Generation With Obstacles

When obstacles were introduced, the trajectories will be generated to avoid collisions. The obstacle introduce in the gridmap an occupied area that have to be avoided to prevent collision, the surface occupied from the obstacles will be represented in black. To ensure a satisfying behavior, taking into consideration that the robot is represented as a point on the map an inflation of the obstacle had to be make. The results highlighted DRIFTY's ability to generate feasible trajectories with the motion planning and trajectory planning algorithms to reach the target location avoiding obstacles.

Obstacles Inflation

To ensure effective obstacle avoidance, we inflated the obstacles by 10 cm. This inflation allowed the robot to approach the obstacles closely enough to navigate through narrow corridors without colliding, so providing a small safety buffer.

PRM Trajectory

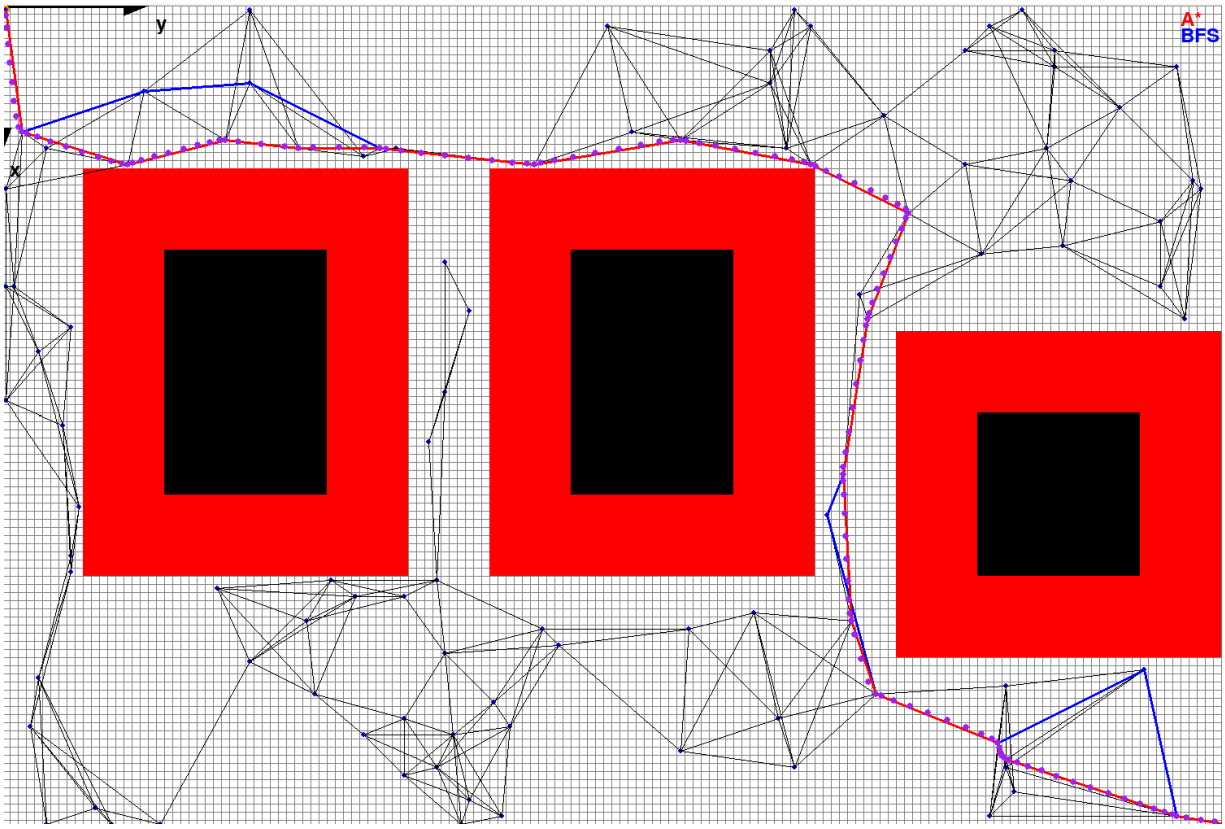

Figure 4.10: PRM linear velocity profile trajectory for obstacle avoidance

RRT Trajectory

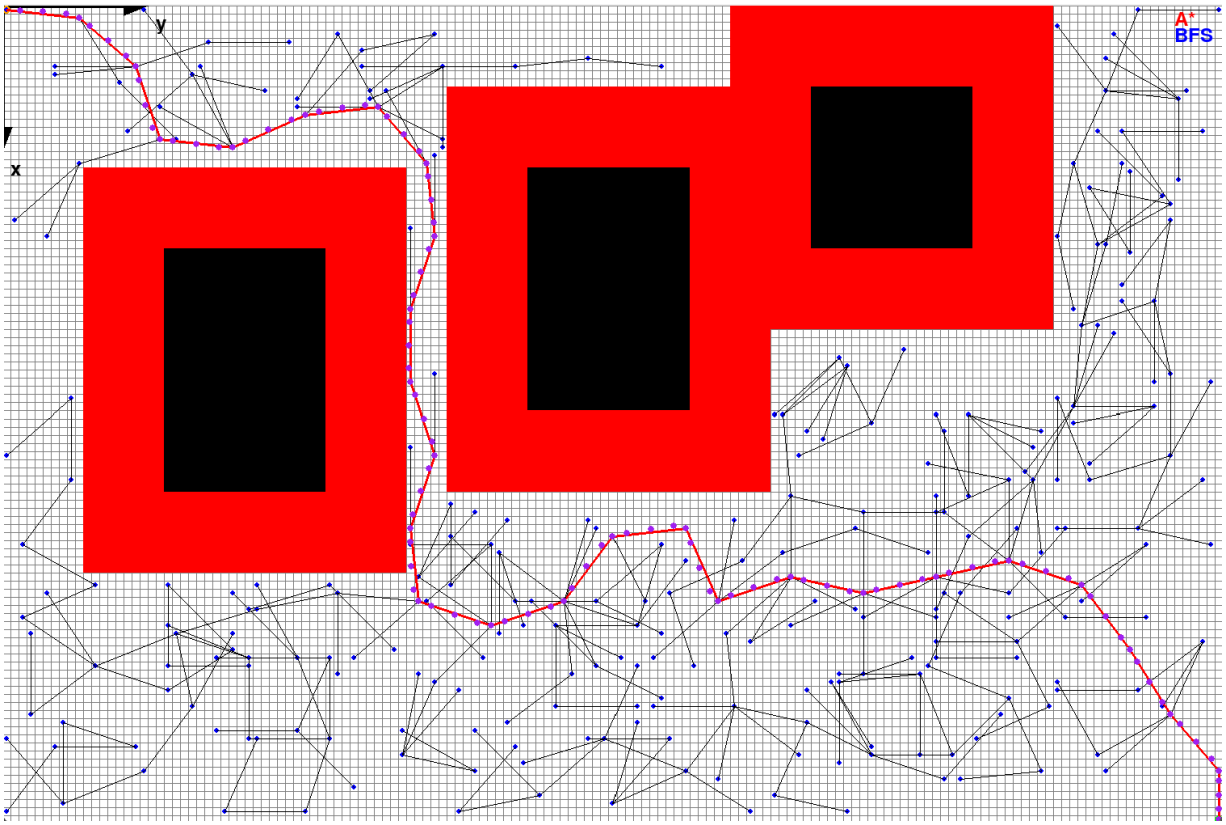

Figure 4.11: RRT linear velocity profile trajectory for obstacle avoidance with a narrow corridor

The results[4][5] confirm that the motion and trajectory planning algorithms used are effective for guiding the robot along a specified path to reach a goal location while also enabling robust obstacle avoidance.

[4] [**video**]Obstacle avoidance with a RRT cartesian trajectory and linear profile of velocity
[5] [**video**] Obstacle avoidance with a PRM cartesian trajectory and linear profile of velocity

Bibliography

[1] Roger R. Labbe. *Kalman and Bayesian Filters in Python*. 2015. URL: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python.

[2] Andrea Morghen Savio Del Peschio Valentina Giannotti. *Drifty, Technical Project FSR*. 2024. URL: https://github.com/saviodp7/TechnicalProjectFSR.