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

Candidate:
Valentina Giannotti P38000189

ANNO ACCADEMICO 2023/2024

Contents

1 Introduction **2**

2 Model **3**
2.1 Unicycle Model 3
2.2 Mapping Differential Drive to Unicycle 4
2.3 Digital Twin 4

3 Control Strategy **5**
3.1 I/O Linearization 5
3.1.1 Go to point 5
3.2 Posture Regulation 10
3.2.1 Go to point 10

4 Motion Planning **16**
4.1 Gridmap 16
4.2 Motion Planning Algorithms 16
4.3 Navigation Function 17
4.3.1 Rapidly-exploring Random Trees (RRT) 18
4.3.2 Probabilistic Roadmaps (PRM) 20
4.3.3 Final observations 24
4.4 Simulation on the Real Robot 25

Chapter 1

Introduction

This project was developed with the goal of implementing advanced control strategies and planning the movement of a differential drive robot, DRIFTY. To simplify analysis and control, a unicycle model was adopted, which allows for a more manageable representation of the robot's kinematics. The project is divided into two separate papers that refers to the same Github repository [1].
In this paper, we will focus on the adopted control strategies and motion planning, considering environments rich in obstacles. In the other paper, trajectory planning strategies and a detailed evaluation of different estimators and performance comparisons will be implemented.
The methodological approach is based on kinematic equations representing the robot's linear and angular velocities. The simplification into a unicycle model allowed the use of more manageable equations for analysis and planning. During the control phase, several strategies (I/O Linearization and Posture Regulation) were implemented, evaluating their effectiveness through simulations and real hardware tests. The results obtained demonstrate how different control and planning strategies can be effectively applied to mobile robot navigation. The simulations allowed for precise calibration of the controller parameters and highlighted the robustness of the models and control strategies, while real tests confirmed the validity of the adopted approaches, with final position errors in the order of a few centimeters.
In addition to the software strategies, this project also involved significant hardware development. The description of the hardware is covered in the other paper.

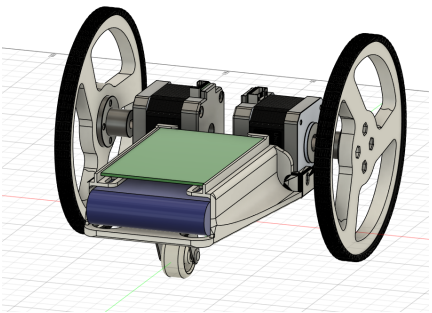

Figure 1.1: DRIFTY

Chapter 2

Model

A differential drive robot can be modeled as a unicycle for the purpose of simplifying control and planning algorithms. This abstraction is particularly useful in mobile robotics to represent the robot's kinematics in a more manageable form. A differential drive robot has two independently driven wheels mounted on the same axis. The kinematic equations for the linear velocity (v) and angular velocity (ω) are given by:

$$v = \frac{r}{2}(\omega_r + \omega_l) \tag{2.1}$$

$$\omega = \frac{r}{L}(\omega_r - \omega_l) \tag{2.2}$$

where r is the radius of the wheels, L is the distance between the wheels, (ω_r) and (ω_l) are the angular velocities of the right and left wheels, respectively.

2.1 Unicycle Model

A unicycle model simplifies the robot to a single point with a heading direction. The kinematics of a unicycle are described by its linear velocity (v) and angular velocity (ω):

$$\dot{x} = v\cos(\theta) \tag{2.3}$$

$$\dot{y} = v\sin(\theta) \tag{2.4}$$

$$\dot{\theta} = \omega \tag{2.5}$$

where (x, y) is the position of the robot and (θ) is its orientation.

2.2 Mapping Differential Drive to Unicycle

To treat a differential drive robot as a unicycle, we use the kinematic equations of the differential drive to define the linear and angular velocities of the unicycle model.
Given the differential drive velocities v and ω we can substitute these into the unicycle model equations:

$$\dot{x} = \left(\frac{r}{2}(\omega_r + \omega_l)\right)\cos(\theta) \tag{2.6}$$

$$\dot{y} = \left(\frac{r}{2}(\omega_r + \omega_l)\right)\sin(\theta) \tag{2.7}$$

$$\dot{\theta} = \frac{r}{L}(\omega_r - \omega_l) \tag{2.8}$$

These equations describe the motion of the differential drive robot in terms of the unicycle model, allowing for easier analysis and control. By modeling a differential drive robot as a unicycle, we can leverage simpler kinematic equations for control and planning. This abstraction maintains the essential characteristics of the robot's motion while simplifying the mathematical treatment.

2.3 Digital Twin

To evaluate different control strategies during the simulation phase, we developed a digital twin of our differential drive system. We roughly approximated the errors in v and w by conducting a series of repeated measurements. Our estimates yielded the following results:

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

Where:

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

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.

Chapter 3

Control Strategy

In this chapter, typical control strategies for mobile robots, such as I/O linearization and Posture Regulation, have been analyzed. Afterward, the advantages and disadvantages of each strategy will be examined and compared.

3.1 I/O Linearization

In the context of the unicycle, the system outputs, which represent the Cartesian coordinates of a point B (located along the sagittal axis of the unicycle, at a distance $|b|$ from the point of contact of the wheel with the ground), are formulated as follows:

$$y_1 = x + b\cos\theta$$
$$y_2 = y + b\sin\theta$$

This provides input-output linearization through feedback, allowing the use of a linear controller to ensure zero convergence of the tracking error:

$$u_1 = \dot{y}_{1d} + k_1(y_{1d} - y_1)$$
$$u_2 = \dot{y}_{2d} + k_2(y_{2d} - y_2)$$

3.1.1 Go to point

Several simulations were conducted and compared with the real robot behaviour to evaluate the performance of the controllers and better understand the effects of varying parameters on the real robot. The following sections will present the results and also the images of goal reached; videos of the simulations can be found in the GitHub repository [1]. After several tests on the physical hardware the value of b was chosen as $b = 0.05$, this value gave us the best performance in terms of stability.

It is important to underline that all simulations were conducted using a second-order Runge-Kutta estimator. The choice of this estimator was not arbitrary; performance comparisons between various estimators revealed that for short distances and non-complex trajectories, second-order Runge-Kutta improved performance over Euler without providing any significant advantage over fourth-order Runge-Kutta. Therefore, second-order Runge-Kutta was chosen for both real tests and simulations. This study will be further

explored in the subsequent paper.

The real hardware includes stepper motors, which have limitations on speed. Therefore, the linear velocity was saturated in the simulation ($max\_vel = 0.10$) to emulate the real hardware. A similar approach was not applied to because the contribution of velocity to the two wheels from v is 15 times greater than that from . The robot was initially positioned at (0,0) and commanded to a target position of (1,1), please note that even if the positioning was carried out trying to obtain the minimum error through the use of fixed references, obviously human intervention will ensure the presence of a non-negligible initial error.

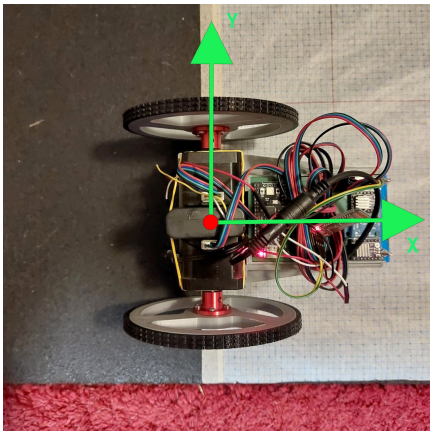

Figure 3.1: Start position

First, the simulation results were presented:
As can be seen from the images, the performance obtained in simulation using a second-order Runge-Kutta estimator, as was done for the real case, is acceptable. It is also noticeable that since theta is not controlled, it assumes an arbitrary orientation. The final position errors, reported below, are approximately four centimeter:

- Final real position error: 0.03493
- Final estimated position error: 0.01232

In the real test [1] with gains $k_1 = 0.01$ and $k_2 = 0.01$, the robot successfully reaches the desired position with an accuracy of few centimeters . The position reached by the robot is esteemed as $x = 0.99595$, $y = 0.99611$, $\theta = 0.84458$, this position is based on the robot's internal estimator system (RK2).
In the image, it can be seen that the coordinates x and y effectively reach the desired position. However, the angle θ is never controlled; it initially moves towards the direction of the goal and then is no longer regulated. Low values of k ensure low values of v and , but at the same time result in a system settling time of more than 12 seconds.
The differences between the simulated model and the real-world test are minimal.

[1] [**video**] I/O Linearization $k_1 = 0.01$, $k_2 = 0.01$

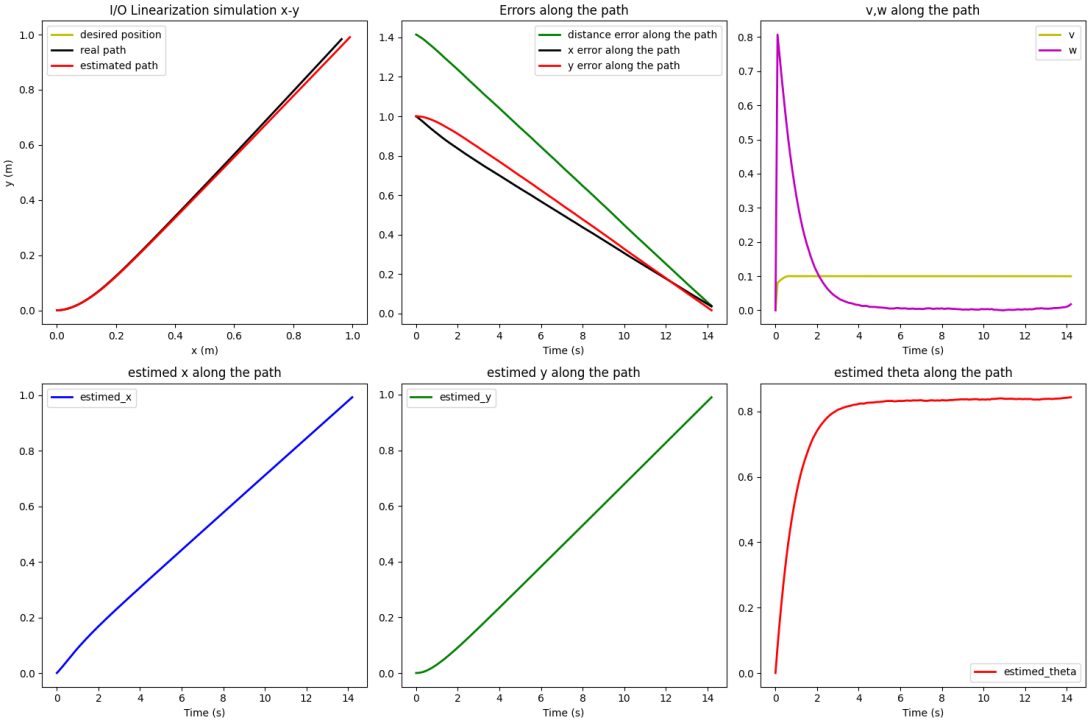

Figure 3.2: Plot I/O Linearization simulation $k_1 = 0.01$ $k_2 = 0.01$

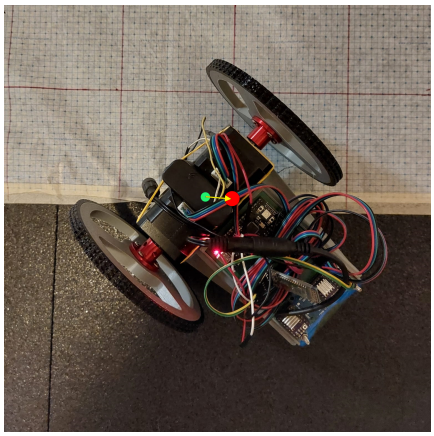

Figure 3.3: Final position with I/O Linearization $k_1 = 0.01$ $k_2 = 0.01$

Afterward, the gains were varied to $k_1 = 0.1$ and $k_2 = 0.1$. From the following simulation, it can be seen that compared to the previous one, $w = 1.75$ is significantly higher than before. Despite this, the desired position is still achieved in the simulation, and desirable results are obtained.

The final position errors, reported below, are approximately four centimeter:

- final real position error: 0.03292
- final estimated position error: 0.01094

In the real test[2] the final position reached showed a considerable error, significantly higher

[2] [**video**] I/O Linearization $k_1 = 0.1$, $k_2 = 0.1$

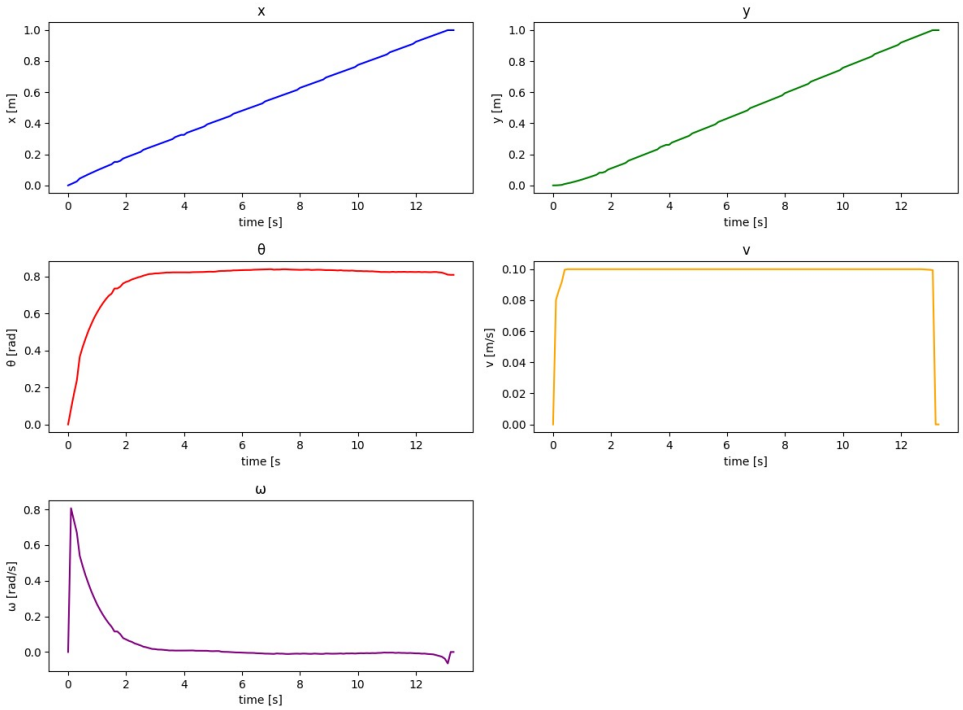

Figure 3.4: Plot I/O Linearization $k_1 = 0.01$ $k_2 = 0.01$

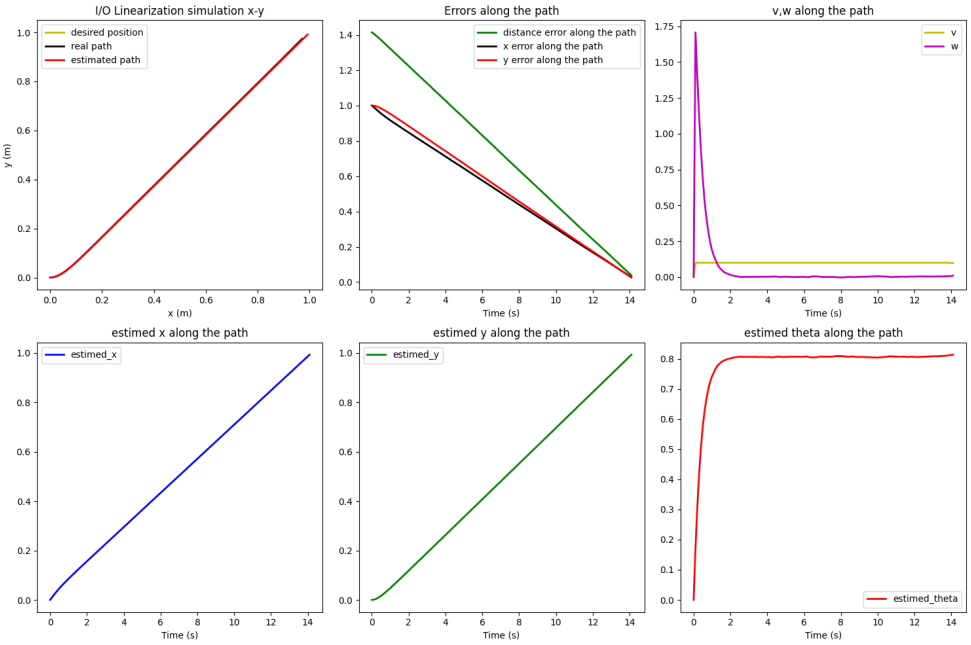

Figure 3.5: Plot I/O Linearization simulation $k_1 = 0.1$ $k_2 = 0.1$

angular speeds are calculated by the controller during the test, unlike the simulation where the slipping of the wheels is not taken into account and infinite angular speeds can be satisfied, in the real case too high angular speeds cause the robot to slip resulting in error of the estimated position and consequently on the position reached.

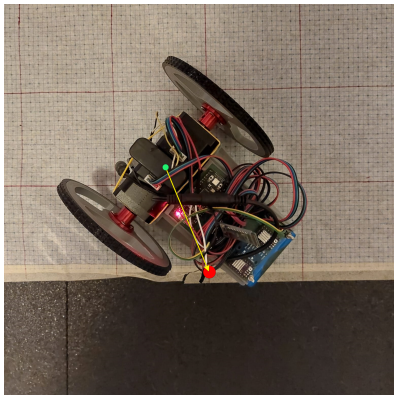

Figure 3.6: Final position with I/O Linearization $k_1 = 0.1$, $k_2 = 0.1$

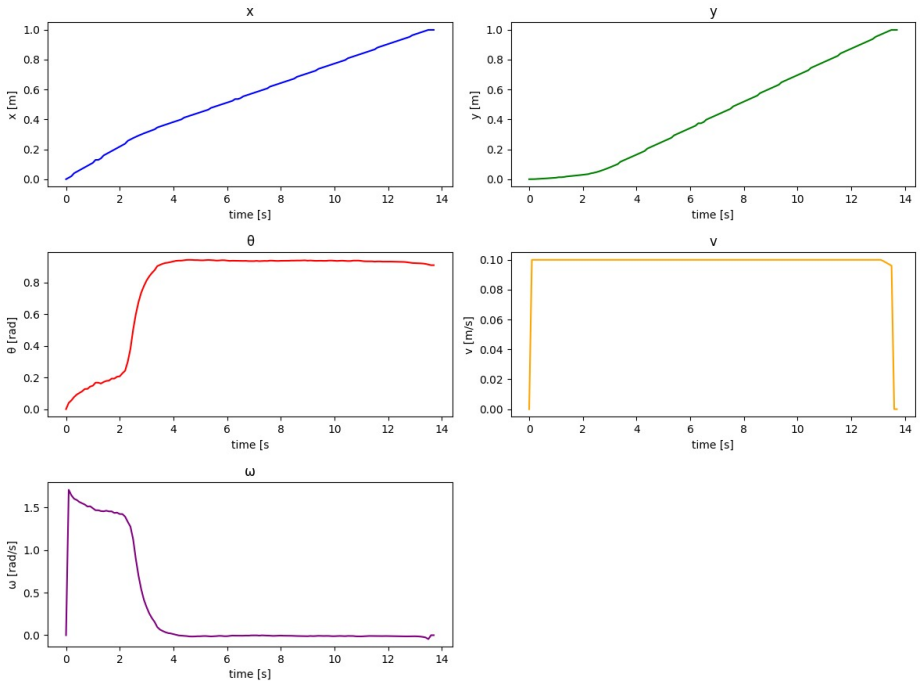

Figure 3.7: Plot I/O Linearization $k_1 = 0.1$ $k_2 = 0.1$

Unlike the previous simulation, what we can observe here is that the angular velocity reaches an initial peak of 1.5, causing the wheels to begin slipping. As can be observed from the previous graphs, in the real case, the initial peak of ω does not extinguish instantaneously. Additionally, θ varies differently compared to the simulation, as it takes into account the initial drift of the robot. By analyzing the positions of x and y, it appears that the robot reaches the desired position; however, in reality, this is not the case. The plotted positions of x and y represent where the robot believes it is, not its actual position. As seen in the photo, due to the initial drift, the robot does not actually reach the desired position. Further simulations with higher values of k are not presented due to the increase in slippage and nonlinear effects that deteriorate performance.

3.2 Posture Regulation

Posture regulation allows to express the desired position and orientation to be reached by the robot. To simplify the analysis and control of the robot, posture is often described using polar coordinates. In this system, the robot's position relative to a reference point is expressed by the distance ρ and the orientation angle γ, while θ represents the robot's orientation. The relationships between Cartesian and polar coordinates are:

$$\rho = \sqrt{x^2 + y^2}$$

$$\gamma = \text{atan2}(y, x) - \theta + \pi$$

$$\delta = \gamma + \theta$$

To ensure that the robot converges to the desired configuration, a feedback controller can be designed as follows:

$$v = k_1 \rho \cos(\gamma)$$

$$\omega = k_2 \gamma + k_1 \sin(\gamma) \cos(\gamma) \left(1 + \frac{k_3 \delta}{\gamma}\right)$$

with k_1, k_2, and k_3 being positive constants. Note the singularity for $\rho = 0$, where the system in polar coordinates is not defined at the target point. The one-to-one mapping between Cartesian and polar coordinates is lost at the origin. Without loss of generality, $q_d = [0\ 0\ 0]^T$, where q_d represents the desired configuration of the robot, implying that the target point is the origin with a null orientation. To adapt this controller to a real case, where our reference frame coincides with the robot's initial position, a transformation between frames was necessary. This operation has the advantage of allowing a goal to be set while maintaining consistency between the real and simulated grids. The transformation matrix used is as follows:

$$A = \begin{pmatrix} c_\theta & s_\theta & 0 & -c_\theta \cdot x_d - s_\theta \cdot y_d \\ -s_\theta & c_\theta & 0 & -c_\theta \cdot y_d + s_\theta \cdot x_d \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $c_\theta = \cos(\theta)$ and $s_\theta = \sin(\theta)$.

3.2.1 Go to point

For the Posture Regulation, different simulations and real tests were also performed by properly changing the parameters. Below, you can observe the performance obtained in simulation with a Posture Regulation control and gains of the type $k_1 = 1, k_2 = 3, k_3 = 1$. As in the previous case, the initial position was $q_i = [0, 0, 0]$, and the final position was $q_f = [1, 1, 0]$. From the simulations, it can be seen that the approach to the final position is smoother. The ω values have a higher initial peak compared to the previous simulations, and it is also noticeable that θ tends to correctly reach the desired orientation, with only a slight error (5 degrees).
The final position and orientation errors, reported below, are as follows:

- Final real position error: 0.00080

- Final estimated position error: 0.03674
- Final orientation error: -5.10297

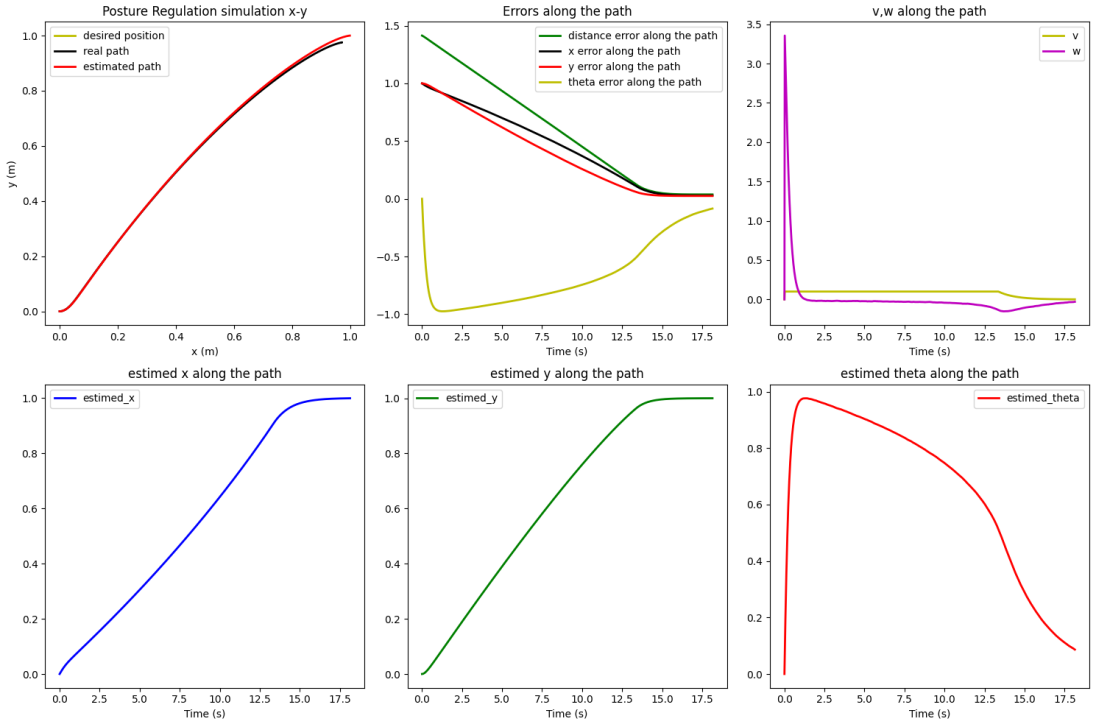

Figure 3.8: Posture Regulation simulation $k_1 = 1$, $k_2 = 3$ $k_3 = 1$

In the real test [3] with gains $k_1 = 1$, $k_2 = 3$ and $k_3 = 1$, the robot successfully reaches the desired position with an accuracy of few centimeters . The position reached by the robot is esteemed as $x = 0.99834$, $y = 0.99994$, $\theta = 0.08669$, this position is based on the robot's internal estimator system (RK2).

[3] [**video**] Posture Regulation $k_1 = 1$, $k_2 = 3$ and $k_3 = 1$

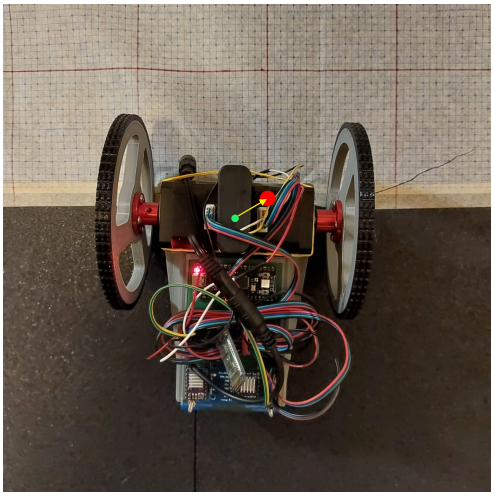

Figure 3.9: Final position with Posture Regulation $k_1 = 1$, $k_2 = 3$ $k_3 = 1$

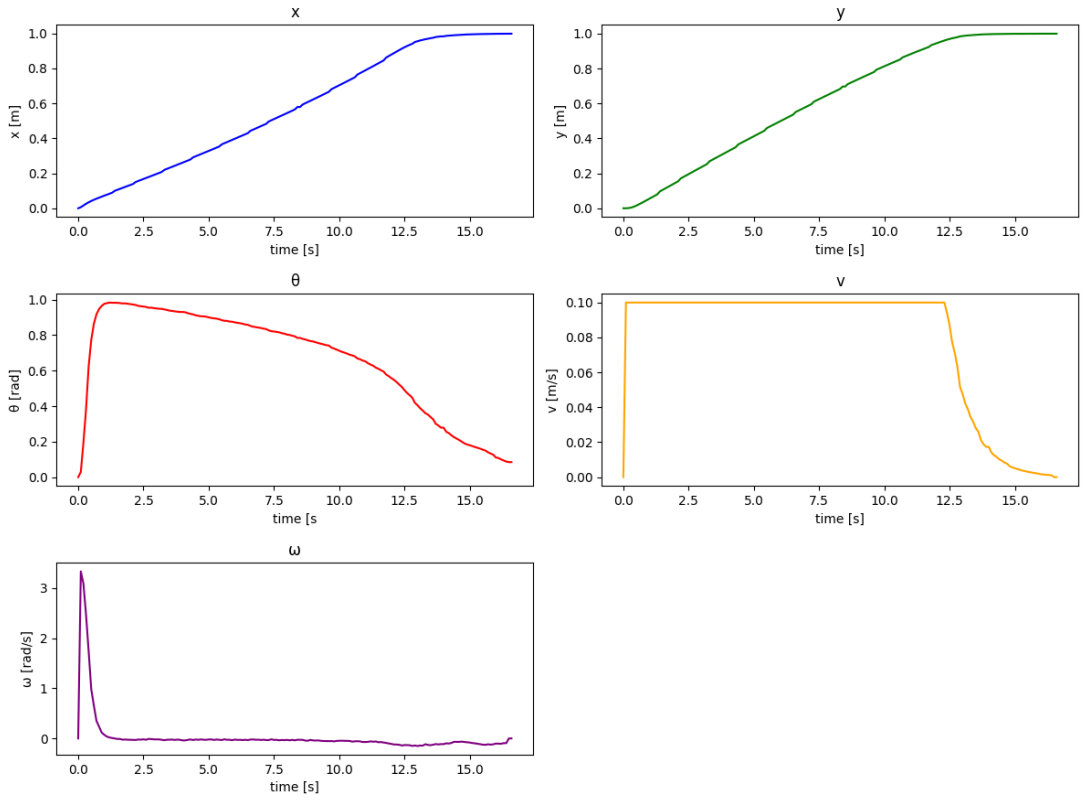

Figure 3.10: Posture Regulation simulation $k_1 = 1$, $k_2 = 3$ $k_3 = 1$

As observed in the simulation, the approach to the final position is slower. The robot first reaches the position and then adjusts its orientation. With these values of K, despite having a high ω, the system converges with minimal error and does not slip in reality. Higher values of k cause the robot to slip; therefore, simulations with greater values are not presented.

Afterwards, another test was done with gains $k_1 = 1$, $k_2 = 0.3$ and $k_3 = 0.1$:

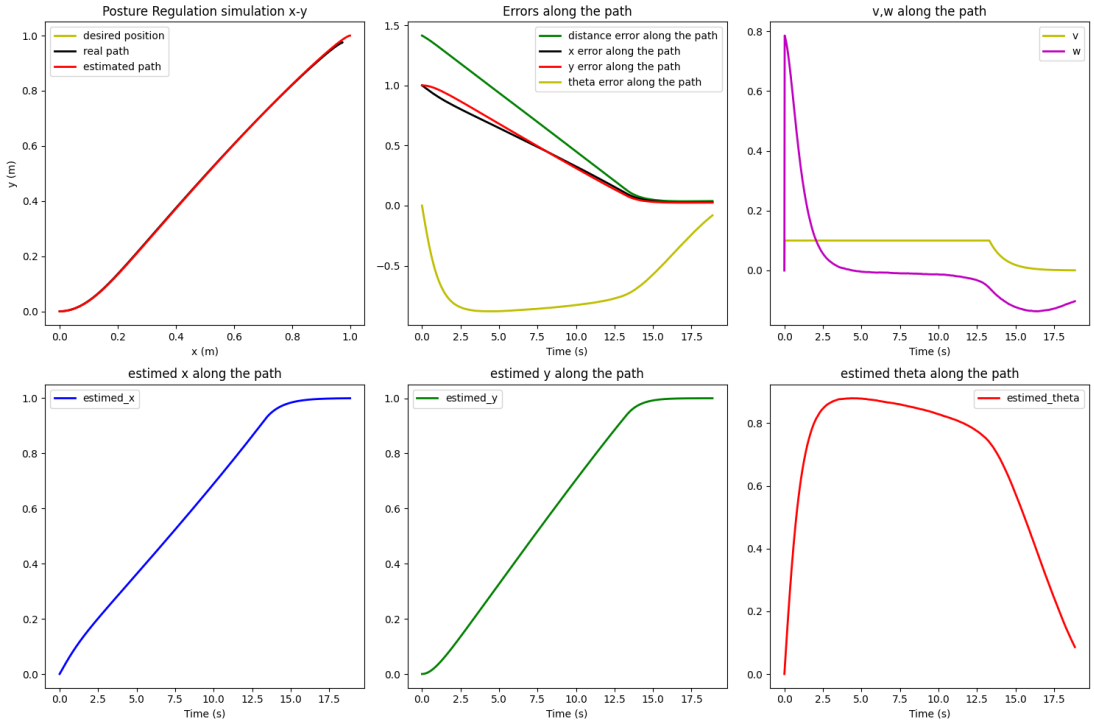

Figure 3.11: Posture Regulation simulation $k_1 = 1$ $k_2 = 0.3$ $k_3 = 0.1$

From the previous simulation, what can be seen is that by lowering the gains, the peak on ω decreases.

The real test was done [4] with gains $k_1 = 1$, $k_2 = 0.3$ and $k_3 = 0.1$, again, as in the previous one, the robot manages to reach the goal with an error of about four centimetre.

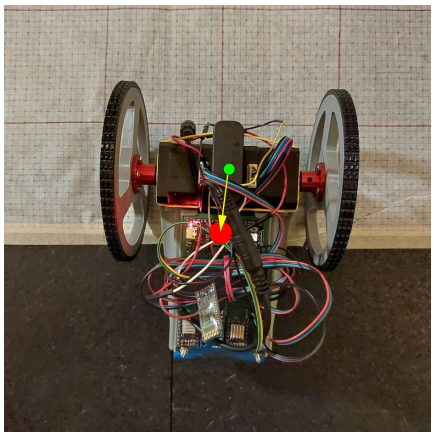

Figure 3.12: Final position with Posture Regulation $k_1 = 1$ $k_2 = 0.3$ $k_3 = 0.1$

[4] [**video**] Posture Regulation $k_1 = 1$, $k_2 = 0.3$ and $k_3 = 0.1$

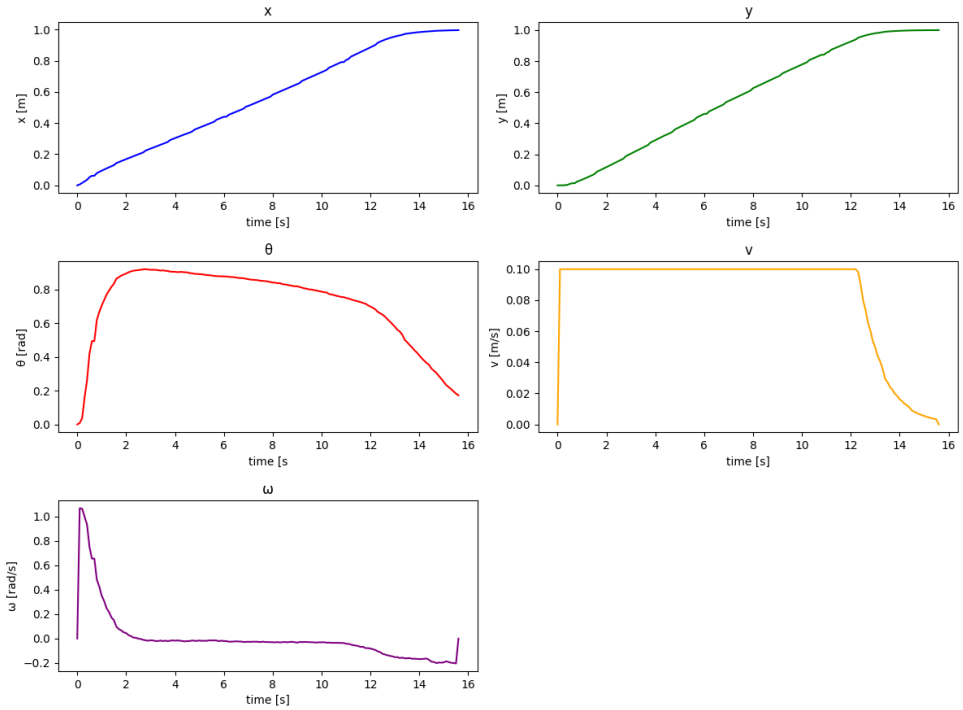

Figure 3.13: Posture Regulation test $k_1 = 1$ $k_2 = 0.3$ $k_3 = 0.1$

Lower values of k correspond to lower values of w and v, even in real tests. Additionally, the trajectory becomes longer.
The posture regulation allows us to control the angle as well. Therefore, additional simulations were conducted by modifying the angle of the final position $q_f = [1, 1, \pi]$.

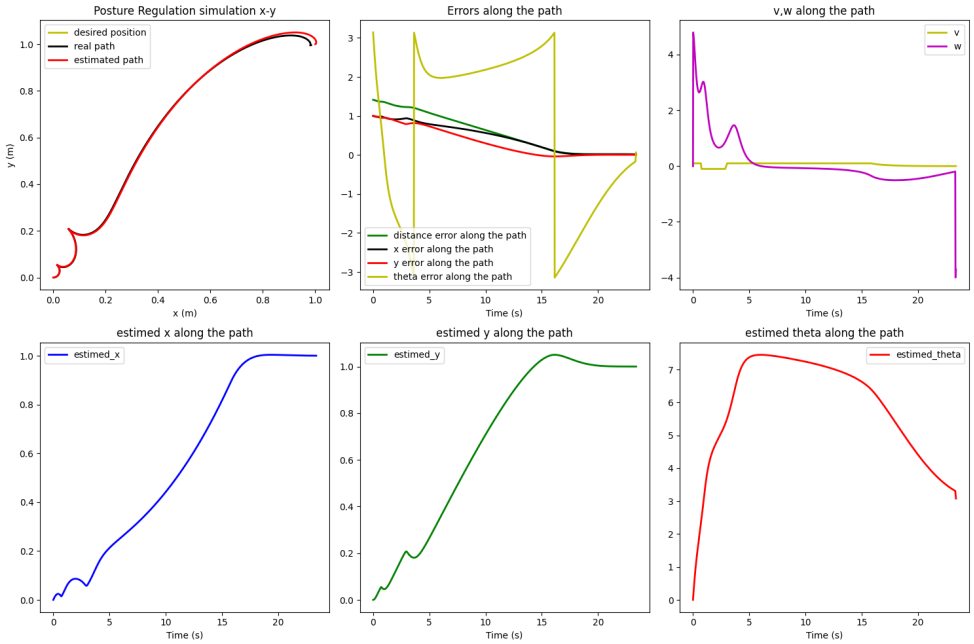

Figure 3.14: Posture Regulation test $\theta = \pi$ $k_1 = 1$ $k_2 = 0.6$ $k_3 = 0.2$

In the real test[5]

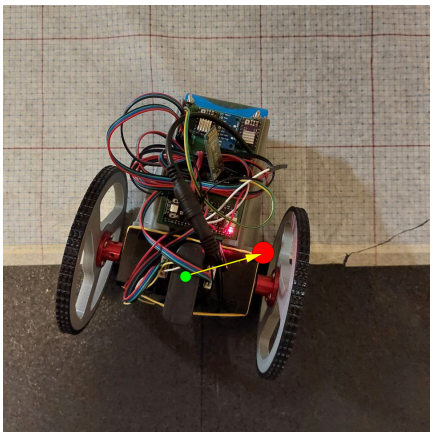

Figure 3.15: Final position with Posture Regulation $\theta = \pi$ $k_1 = 1$ $k_2 = 0.6$ $k_3 = 0.2$

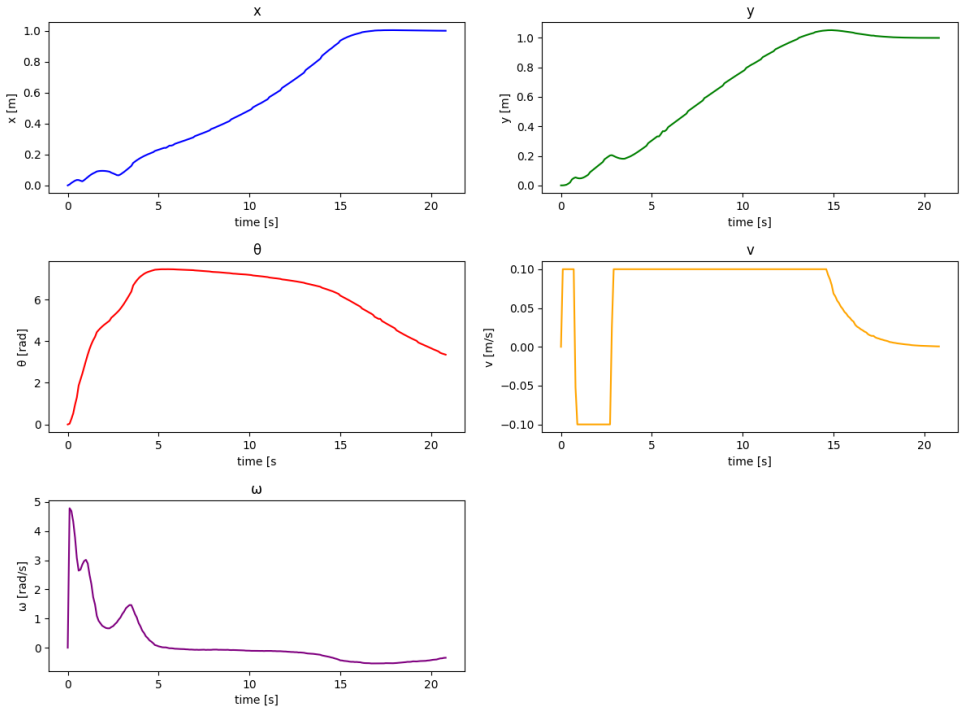

Figure 3.16: Posture Regulation test $\theta = \pi$ $k_1 = 1$ $k_2 = 0.6$ $k_3 = 0.2$

As we can see from the first two tests, the robot tends to align itself in the desired direction of θ immediately and then positions itself along the trajectory, adjusting θ by a little at the end. On the hand, in this case the desired θ is not aligned with the direction of travel so the approach to the final position is slower because the robot first reaches the position and then takes more time to achieve the desired θ value.

[5][**video**] Posture Regulation $k_1 = 1$, $k_2 = 0.6$ and $k_3 = 0.2$

Chapter 4

Motion Planning

In this chapter, we describe motion planning, i.e. the calculation of feasible paths for a robot or vehicle to move from a starting point to a destination while avoiding obstacles. The system integrates a grid-based representation of the environment and a motion planning algorithm to ensure optimal navigation.
The environment in which the robot operates is represented as a grid-map, where each cell can be free or occupied by an obstacle. This representation is essential for the motion planning algorithm, as it provides the data structure needed to calculate paths and avoid obstacles. The obstacles in the virtual environment are inflated accordingly with the dimensions of the robot to avoid collisions.

4.1 Gridmap

In the context of motion planning, the grid-map is the representation of the environment in which the robot moves. Each cell in the grid contains a value indicating whether the cell is free or occupied by an obstacle. The grid is managed by a Python class, which allows initializing the map by specifying the width, height, and resolution of the map. This flexibility enables the map to be adapted to the specific dimensions of the robot's operational area and obstacles. The class also offers methods for adding obstacles to the grid, dynamically updating the representation of the environment. A crucial aspect for obstacle avoidance is the ability to perform "inflation" of the obstacles: this process involves virtually expanding the obstacles on the map, thereby ensuring safety and preventing accidental collisions during movement. It is important to remember that on the physical robot, through a Ground Station, it is possible to insert obstacles in the desired way with an inflation set to 10 cm and generate a path by choosing which motion planning and trajectory planning methods to use.

4.2 Motion Planning Algorithms

Three motion planning methods have been implemented: Rapidly-exploring Random Roadmap Trees (RRT), Probabilistic Roadmaps (PRM) and Navigation Function. These algorithms are designed to find feasible paths in complex environments, leveraging the grid-based representation.

4.3 Navigation Function

To implement an algorithm allowing movement based on the numerical navigation function, the first step involved constructing a cost matrix starting from the endpoint q_f with a value of zero. Next, values of 1 is assigned to the four (eight in case of an 8-adiajent) orthogonal cells adjacent to q_f where no obstacle was present. This process continued iteratively, assigning a value of 2 to the orthogonal neighbors of each cell with a gain of 1, and so forth, exploring the entire matrix. The gain matrix was built by assigning incremental gain values to all 4 (8) adjacent cells, expanding the matrix progressively.

Once the gain matrix was constructed, the path from the initial point to the final point was searched using the steepest descent method, which is commonly employed for offline planning. This method ensures that the movement is directed towards the lowest cost value at each step, effectively navigating through the cost matrix towards the endpoint.

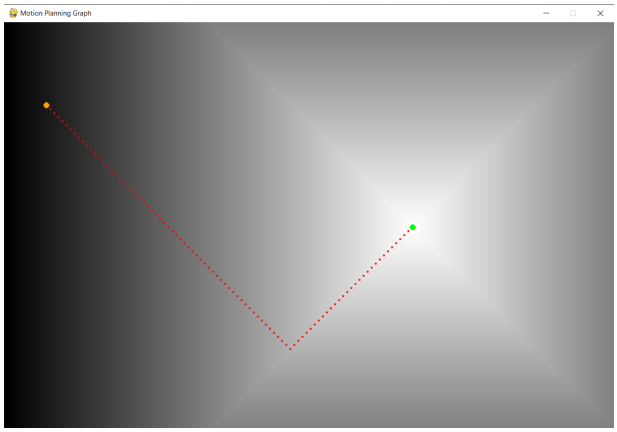

Figure 4.1: Navigation Function

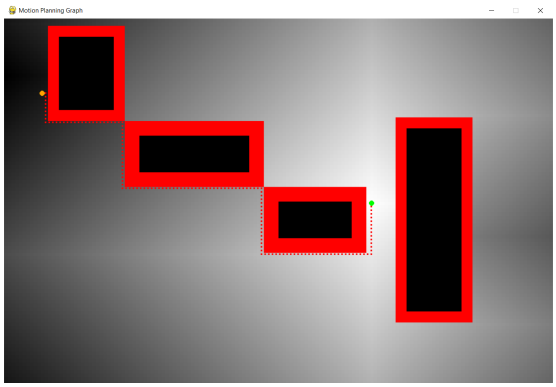

Figure 4.2: Navigation Function 4 adjacent cells

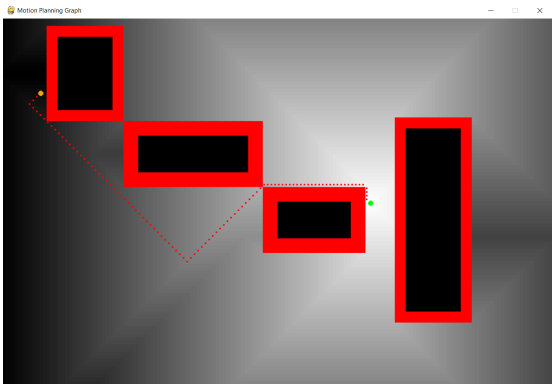

Figure 4.3: Navigation Function 8 adjacent cells

From the figure, it is noticeable that the farther one is from the goal, the darker the color becomes; conversely, the closer one is, the lighter the color becomes. Smaller values correspond to lighter shades, while larger values correspond to darker shades.

Additionally, the figure shows that, in the case of 8 cells, the path also crosses the corners. This does not pose a problem because the obstacles are considered with an inflation margin, ensuring the path is still navigable.

4.3.1 Rapidly-exploring Random Trees (RRT)

The required algorithm for path planning is the RRT method. It creates a randomized tree that iteratively branches by creating connections between random points and their nearest tree nodes. The goal is to connect the initial point q_i and a final point q_f through a path generated using the RRT algorithm, while avoiding obstacles. The algorithm consists in the generation of a random point in space and the connection of it with the nearest tree point. To do this, a δ step size is set, and a point q_{new} is generated in the direction between the nearest tree point q_{near} and the random point called q_{rand}. It is necessary to check the feasibility of the movement. If the movement is feasible, it is added to the tree; otherwise, a new random point is generated. After a certain number of iteration we have to check if the end-point can be added to the tree. If the movement from the nearest point to the goal position is possible, the graph is closed, and a feasible path is searched from the goal node iteratively back to the parent node until reaching q_i.

Afterward, various simulations were conducted using RRT graph generation algorithms with different values of δ, and the results were compared. All the simulations reported below were performed using an I/O linearized controller with gains $k_1 = k_2 = 1$ and a trajectory generated through a cubic polynomial profile with $K = 0.05$.
It is important to note that all the path images have been flipped for semplicity of visualization. In the simulations, the graphical images have the top-left cell as (0,0) due to the default settings of pygame windows. To facilitate the understanding of the graphs, we have decided to present them this way in the report.

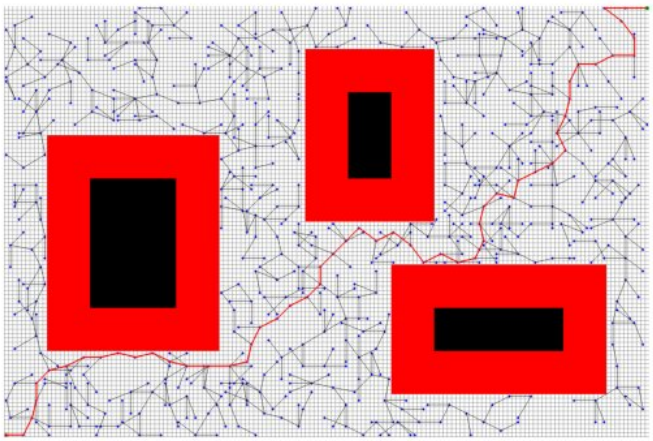

Figure 4.4: RRT δ=5

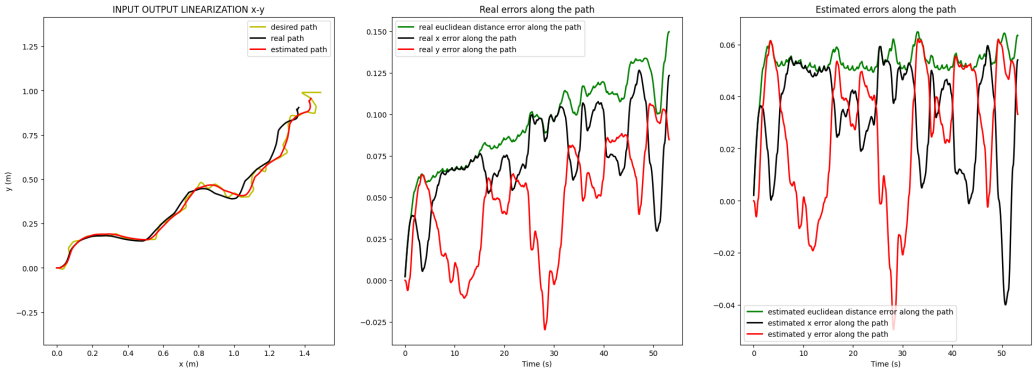

Figure 4.5: RRT δ=5

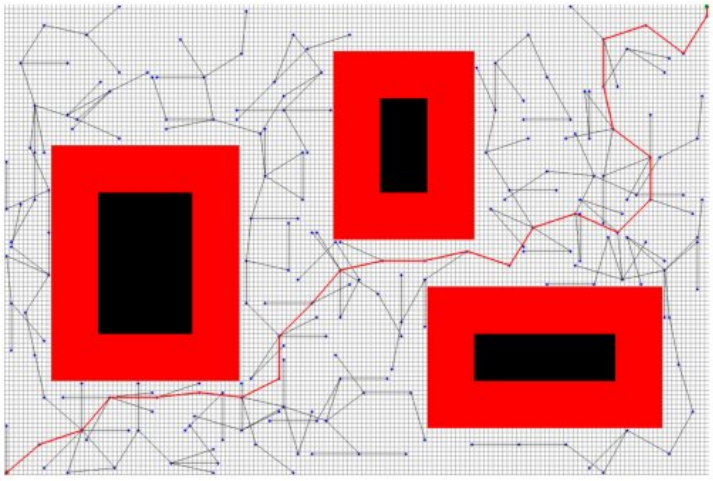

Figure 4.6: RRT δ=10

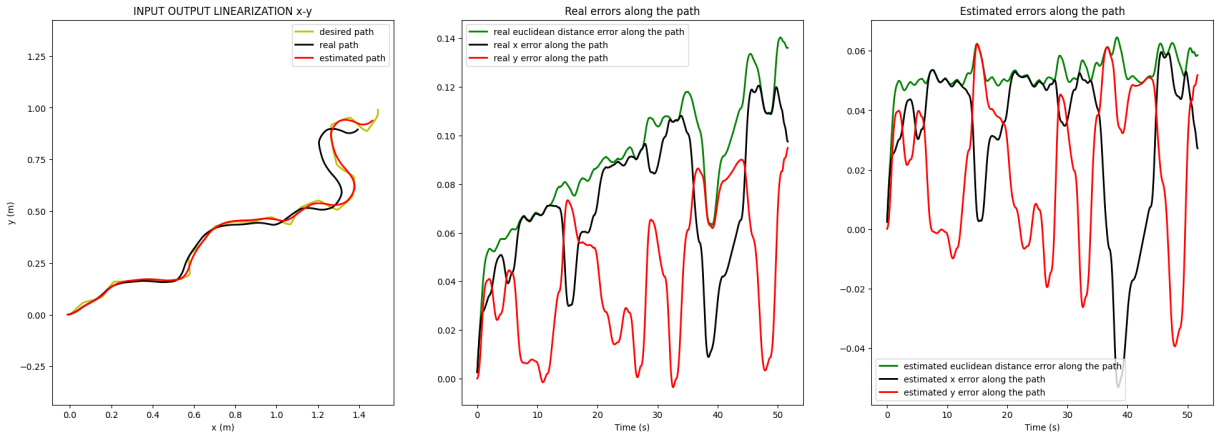

Figure 4.7: RRT δ=10

We can observe that by decreasing δ, the graph becomes denser. This is because modifying δ allows us to adjust the distance between q_{near} and q_{new}. As δ decreases, the spacing between points reduces, resulting in a more densely connected graph. On the other hand, when δ is greater than 30, it becomes difficult to ensure closure, since the minimum

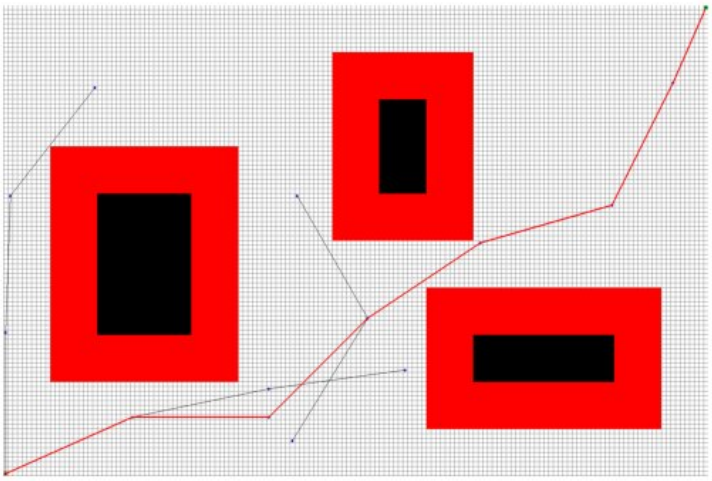

Figure 4.8: RRT δ=30

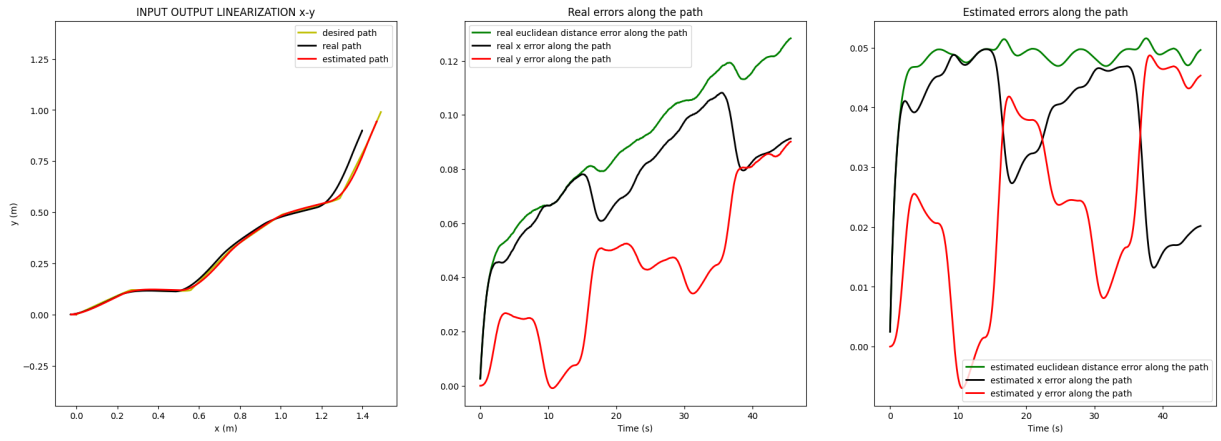

Figure 4.9: RRT δ=30

distance between the lengths of the obstacles is 30.
Also, there is no substantial difference in the errors, as the error reduces with the decrease in the path length. For this algorithm, the δ parameter does not significantly indicate the final path length. Consequently, the choice of δ depends on the type of obstacles present.

4.3.2 Probabilistic Roadmaps (PRM)

PRM is another widely used algorithm for path planning, particularly suitable for multi-query problems. It constructs a roadmap by randomly sampling points in the free space and connecting them with feasible paths. The process begins with sampling, where points are randomly generated in feasibles positions to form the nodes of the roadmap. Next, in the connection phase, each node is connected to its nearest k neighbors, ensuring the connections are collision-free. During the query phase, the constructed roadmap is used to find paths between different start and goal positions. Finally, in the path extraction step, the path is extracted by finding a sequence of connected nodes from the start to the goal utilizing graph-search algorithms.

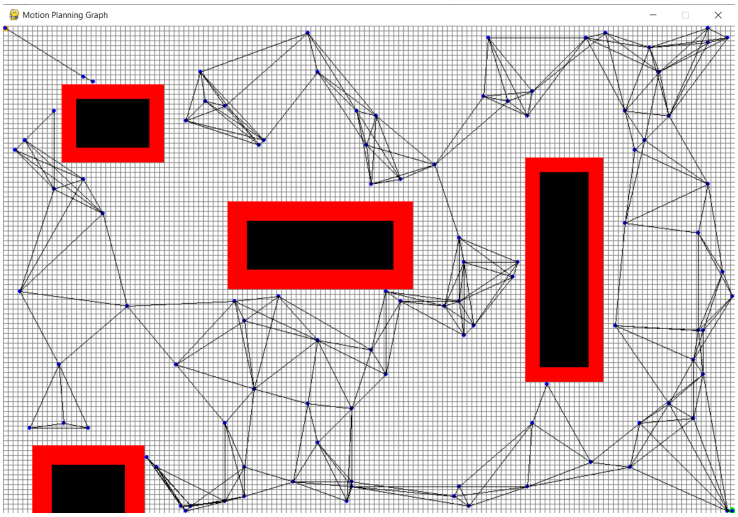

Figure 4.10: PRM graph $k = 5$

Implementation of Search Algorithms

BFS Algorithm

In the project, we implemented the Breadth-First Search (BFS) algorithm to explore and find optimal paths within a graph. BFS explores one level completely before moving on to the next, ensuring that the path found is the shortest possible in terms of the number of nodes traversed. The algorithm was implemented using a queue to manage the nodes to be explored. Initially, the starting node is inserted into the queue and marked as visited. Then, iteratively, the node at the front of the queue is extracted, and for each of its adjacent nodes that have not yet been visited, they are added to the queue and marked as visited. This process continues until the queue is empty or the destination node is found. The implementation of BFS in our project was fundamental in ensuring that the robot could navigate through its environment efficiently, finding the shortest path in terms of the number of steps (nodes) required.

A* Algorithm

Next, we implemented the A* search algorithm to find optimal paths within a graph. A* was chosen for its ability to combine the efficiency of breadth-first search with the optimisation of the search heuristic, allowing for fast and accurate navigation. This algorithm uses both the path cost from the initial node to a current node and a heuristic estimate of the cost from the current node to the destination node. This approach ensures that the path found is not only the shortest in terms of distance, but also the most computationally efficient. By using a priority queue to expand the nodes with the lowest estimated total cost, A* is able to find the optimal path in a systematic and efficient manner (optimal computationally).
In the image above PRM with $K = 10$ was used, the following results were obtained:

- For the A* algorithm, the length of the path was 1.876 meters, and it consisted of 14 points.

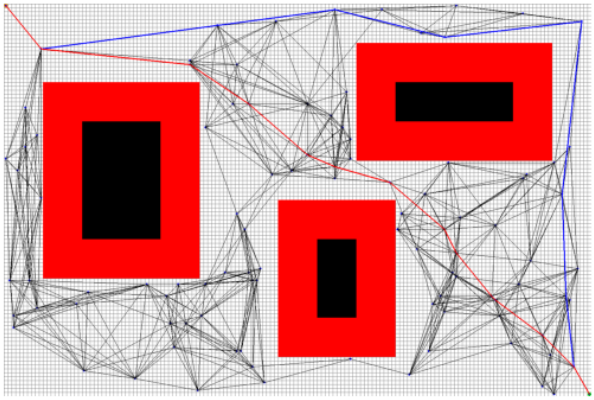

Figure 4.11: A* (red) and BFS (blue) paths

- On the other hand, for the BFS algorithm, the path length was 2.504 meters with a total of 10 points.

Both algorithms are valid, BFS tends to minimise the nodes while A* the distance between the start and end point by choosing the minimum path in terms of distance.

Simulation with varing k

Taking into account the observations made before, all the simulations were performed on the A* path using an I/O linearized controller with gains $k_1 = k_2 = 1$, a cubic polynomial profile with $K = 0.05$. In the same way as before all the path images have been flipped for ease of visualization.

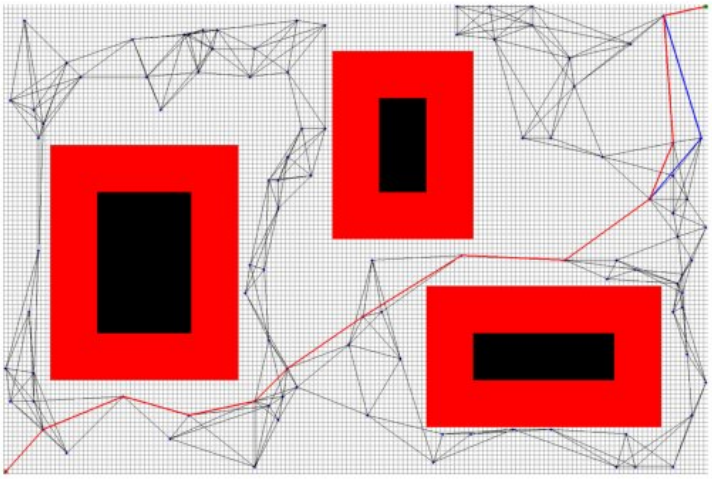

Figure 4.12: PRM =5

As k increases, there is a significant improvement in terms of error reduction, as the increase in connections allows for more direct paths with fewer course changes and lower path length. This significantly reduces the error, as can be seen in the figures above.

On the other hand, for very low values of k (less than 5), it might be impossible to find paths since the generated graphs could be disconnected.

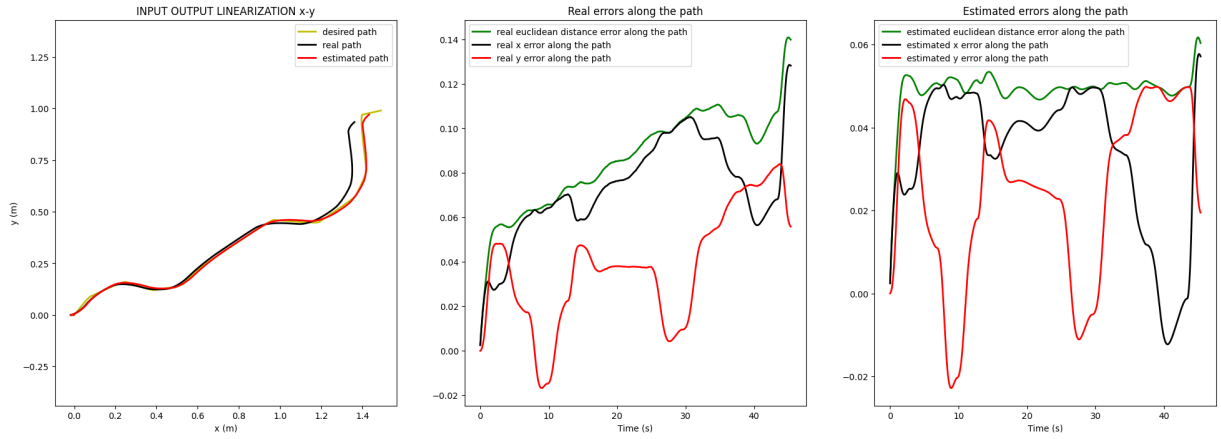

Figure 4.13: PRM with A* algorithm =5

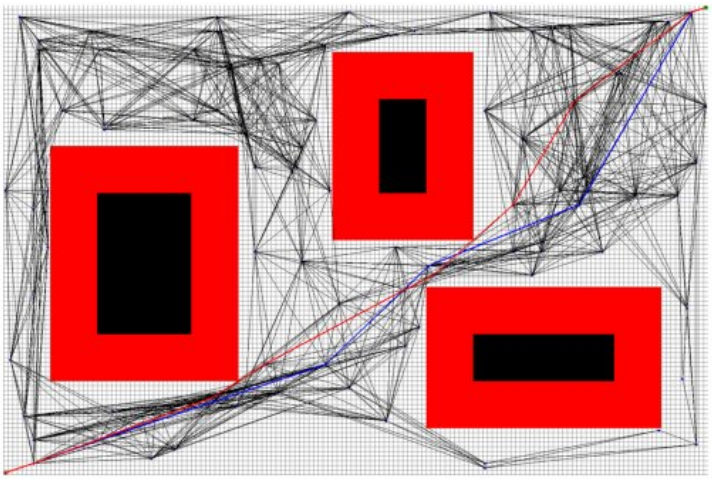

Figure 4.14: PRM =20

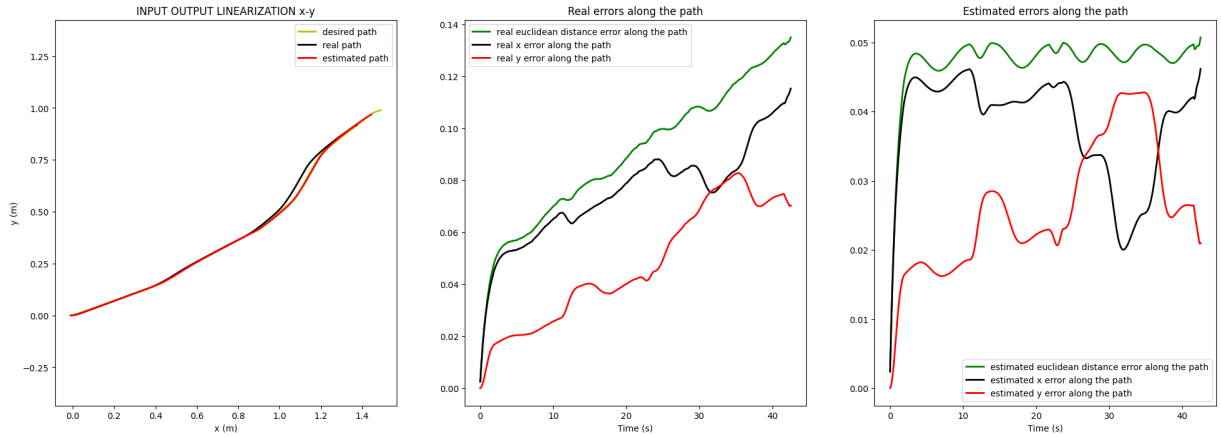

Figure 4.15: PRM with A* algorithm =20

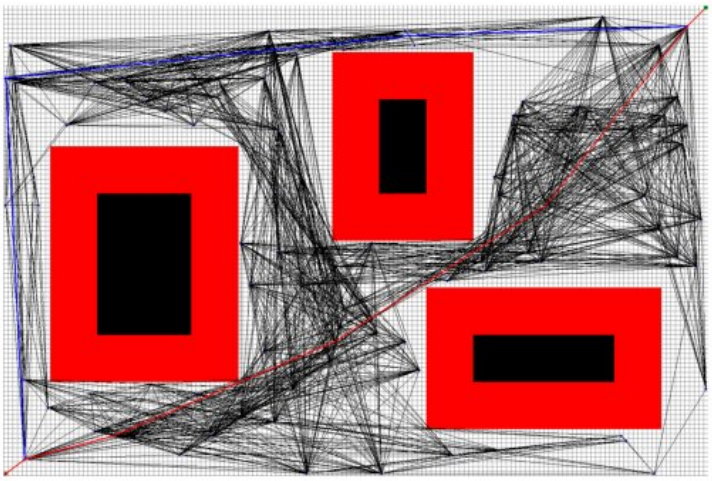

Figure 4.16: PRM =50

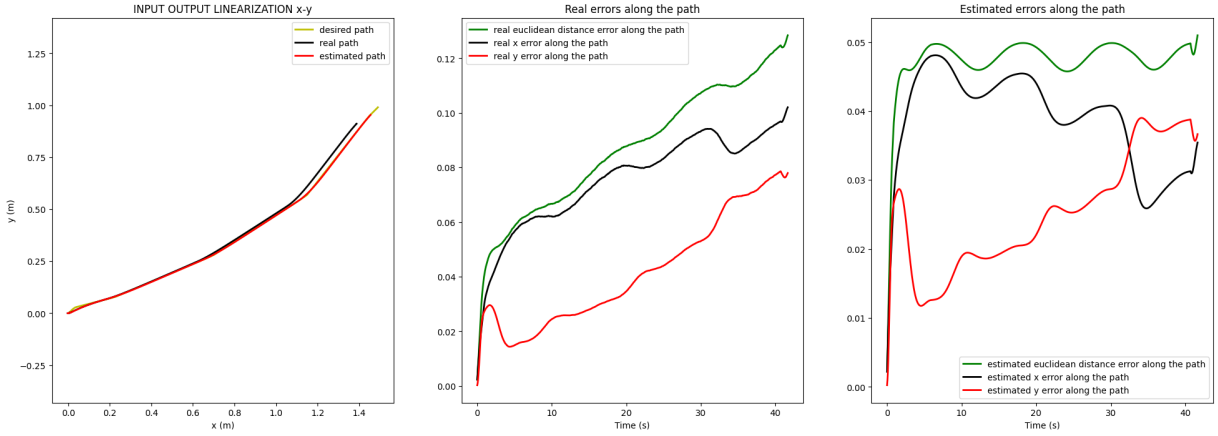

Figure 4.17: PRM with A* algorithm =50

4.3.3 Final observations

This is a final consideration made after noticing the systematic error present in the estimated error along the path.It is indeed possible to notice that in all RRT and PRM simulations, the estimated error oscillates around a value different from zero (between 0.04 and 0.06). We hypothesized that by computing the desired speeds at runtime with the appropriate function (compute_desired_speed), it is possible to account for the cumulative delay present in all simulations conducted so far. However, by making this change, the trajectory planning as a whole is not fully utilized; only the path sequence is considered. Despite this, the simulations improve significantly, bringing the average error of the estimated path close to zero, and consequently, drastically reducing the error on the real path as well. Below are two simulations on the same path in the two described modes:

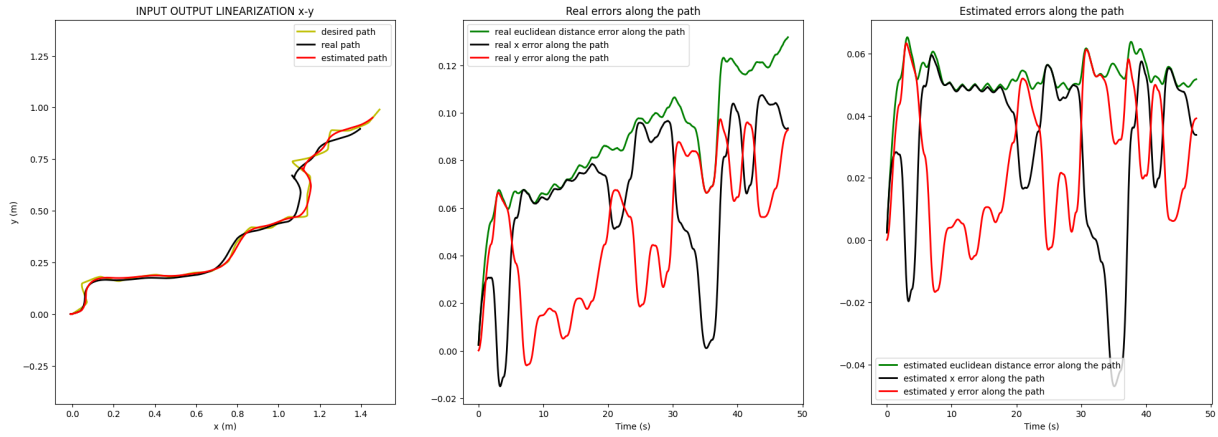

Figure 4.18: With desire speed

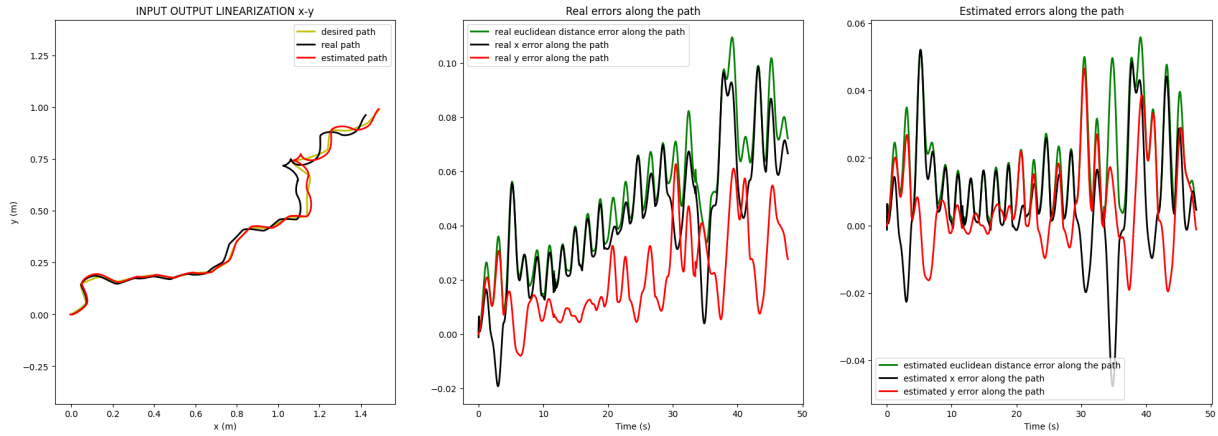

Figure 4.19: With computed speed

4.4 Simulation on the Real Robot

Simulations were also conducted with different trajectory generation methods on the physical robot. It was observed that using RRT for trajectory generation allowed the robot to navigate through very narrow spaces, while using PRM did not produce a trajectory capable of passing through such tight spaces unless extremely high values of k were chosen. Below are the generated paths, and attached are the videos of the simulations.

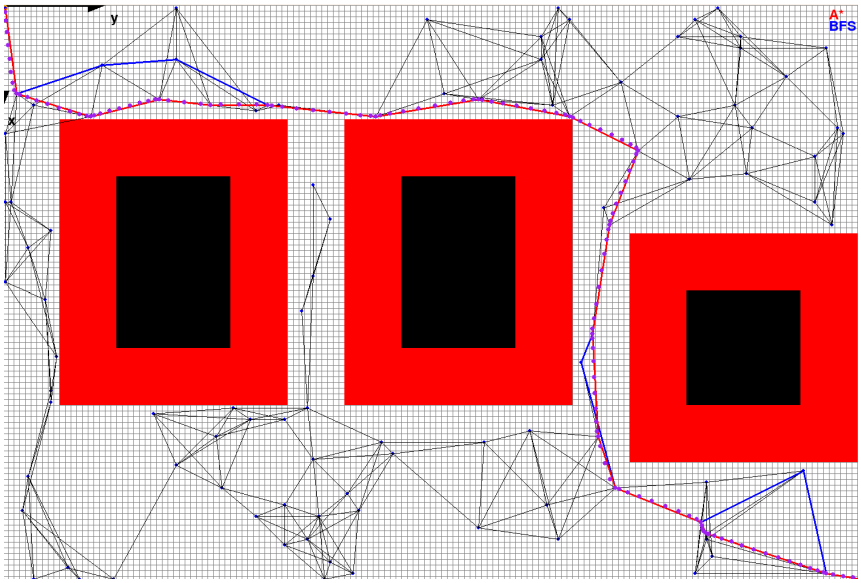

Figure 4.20: Obstacle avoidance PRM

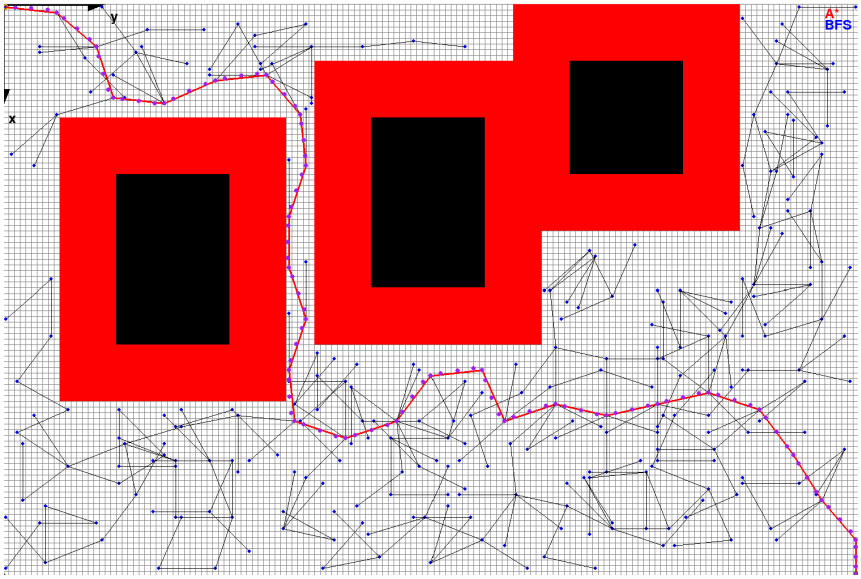

Figure 4.21: Obstacle avoidance RRT

Bibliography

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