EKF-MPC Robot Navigation

Amlan Sahoo Ethan Dsouza Lorenzo Uttini

Group 05

April 10, 2025

1 Introduction

The deployment of any autonomous robotic system requires precise localization and control strategies to ensure that it navigates efficiently and perform the required task actions. In this project we explore two advanced methodologies: the Extended Kalman Filter (EKF) and Model Predictive Control (MPC) to aid in enhancing the navigation capabilities of a wheeled mobile robot.

The localization process allows the robot to accurately estimate its position and orientation within a known environment populated with identifiable landmarks. We implement an EKF-based estimation algorithm that integrates range-bearing sensor measurements with the robot's dynamic model to provide reliable pose estimates. The EKF is an extension of the standard Kalman Filter, specifically designed to address the challenges posed by nonlinear dynamics, such as in robotic applications. It employs Jacobian matrices to linearize the system around the current state, allowing it to effectively handle nonlinear relationships in the differential drive model of the robot. By considering sensor noise and process uncertainties, the EKF enhances positional accuracy, thereby improving the robot's ability to navigate in complex environments.

In parallel, trajectory control is managed using a Model Predictive Control (MPC) regulator, which anticipates the robot's future states and adjusts control inputs accordingly to facilitate smooth and efficient navigation. MPC is a widely used advanced control strategy in robotics, particularly for trajectory tracking, stabilization, and optimal control. Operating within a closed-loop feedback framework, MPC continuously solves an optimization problem to determine the control inputs that minimize a cost function over a finite prediction horizon. At each timestep, it considers the current state of the robot and predicts future states, adapting the control inputs in response.

Within the MPC framework, system dynamics are represented by matrices A and B, which characterize the relationship between the state and control input. The matrix A represents how the state transforms over time, while B describes the influence of the control inputs on the state. The cost weighting matrices Q and R are integral to the MPC cost function where Q penalizes deviations from the desired state, and R penalizes excessive control inputs. By tuning these matrices, we can prioritize either state accuracy or control smoothness, achieving an optimal balance.

This project report details the development and integration of the EKF and MPC systems within a simulated environment, followed by a comprehensive evaluation of the robot's performance.

This project is structured around four main tasks:

- **Task 1:** Implementation of an EKF-based estimation algorithm to determine the position and orientation of the robot using range and bearing measurements from known landmarks in the environment. This task explores the fundamentals of localization.
- **Task 2:** Development of an Unconstrained MPC Controller for a Differential Drive Robot, focusing on tuning matrices Q, R, and the prediction horizon N for optimal performance.
- **Task 3:** Integration of the EKF with the MPC controller to develop a robust system for trajectory tracking and stabilization in the presence of sensor noise.
- **Task 4:** Performance comparison of the MPC controller with and without EKF integration, analyzing key metrics such as tracking accuracy, response time, and robustness to noise and initial conditions.

Through these tasks, we demonstrate how the integration of EKF with MPC enhances control robustness, enabling the robot to follow its desired trajectory with minimal deviation despite noises.

2 Materials and Methods

2.1 Materials

This project was conducted within the *Roboenv* virtual Python environment, which uses *Pinocchio*, a Python wrapper for the *PyBullet* simulation framework. The Python version used was 3.11.x. The simulated robot, a four-wheeled model named "robotnik", was configured through a JSON file to define its parameters and noise settings.

Key libraries utilized in this project included:

- `NumPy ver1.26.4`: Employed to define matrices and perform various numerical computations.
- `Matplotlib ver3.9.1`: Used for generating all plots and graphical comparisons presented in this report.
- `Simulation_and_control ver0.1`: A custom Python package which provided a simple interface to the PyBullet simulator and Pinocchio for robotic simulation and control.

In Task 1, `standalone_localization_tester.py` script was used to implement the activities, which facilitated testing within a lightweight simulator. The essential classes and methods needed for the Extended Kalman Filter implementation, was done using the `RobotEstimator` class within the `robot_localization_system.py` file.

For Task 2, activities were executed through the `differential_drive.py` script, containing the core logic for Model Predictive Control (MPC). The `RegulatorModel` class within `regulator_model.py` file, contained methods which helped in system matrix computations, cost matrix definitions, and cost function minimization.

In Tasks 3 and 4, the EKF was integrated with the MPC within a new file named `differential_drive_ekf.py`, which was a copy from the Task 2 script. This separate file was created to streamline the implementation and review process for the integrated EKF-MPC functionality.

2.2 Methods/Procedure

2.2.1 Extended Kalman Filter (EKF)-based estimation algorithm

Activity 1: Evaluating the Initial EKF Localization System

In this activity, we run the initial EKF-based localization system (range-only measurements) using the default landmark configuration ([5, 10], [15, 5], [10, 15]) and analyze error plots to assess performance if the standard deviation (2σ) is less than 10 cm threshold for all timesteps.

Activity 2: Increasing Landmarks for better Localization

In this activity, we investigate various landmark configurations (grid, boundary) and shapes (square, circle, triangle) with different landmark counts to improve localization accuracy. The different configurations are defined as methods in the `Map` class. We aim to find an optimal configuration that meets the standard deviation requirement by comparing the performance of different layouts and analyzing how each one affects localization. We test a range of configurations, described in Section 3.1.2, ultimately selecting the one that keeps standard deviation (2σ) under 10 cm for all timesteps.

Activity 3: Modifying the Sensing System

To incorporate bearing measurements, we define new methods for the EKF implementation in this activity. First, we write the `landmark_range_bearing_observations` method in the `Simulator` class to generate the range and bearing measurements. The observation model used is:

$$\begin{aligned} r^i(t) &= \sqrt{(l_x^i - x(t))^2 + (l_y^i - y(t))^2} + \omega_r(t) \\ \beta^i(t) &= \arctan\left(\frac{l_y^i - y(t)}{l_x^i - x(t)}\right) - \theta(t) + \omega_\beta(t) \end{aligned} \tag{1}$$

where $r^i(t)$ represents the range to landmark i, $\beta^i(t)$ represents the bearing, and $\omega_r(t)$ and $\omega_\beta(t)$ are measurement noise terms for range and bearing, respectively.

A new method, `update_from_landmark_range_bearing_observations`, in the `RobotEstimator` processes these measurements for the EKF implementation. For each observation, we define a Jacobian matrix, C, and a measurement noise covariance matrix, $W_{\text{landmarks}}$, to capture the range and bearing noise.

The Jacobian C of the observation function with respect to the robot's state is:

$$C = \begin{bmatrix} -\frac{\Delta x^i}{r^i} & -\frac{\Delta y^i}{r^i} & 0 \\ \frac{\Delta y^i}{(r^i)^2} & -\frac{\Delta x^i}{(r^i)^2} & -1 \end{bmatrix} \tag{2}$$

where $\Delta x^i = l_x^i - x$ and $\Delta y^i = l_y^i - y$.

The measurement noise covariance matrix W for all landmark observations is constructed as a block-diagonal matrix, where each block corresponds to the noise covariance for an individual landmark. For each observed landmark, the noise covariance matrix W_{landmark} captures the measurement uncertainties in both range and bearing:

$$W_{\text{landmark}} = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\beta^2 \end{bmatrix}$$

where σ_r is the standard deviation of the range measurements, and σ_β is the standard deviation of the bearing measurements. These individual matrices are then combined into a larger block-diagonal matrix $W_{\text{landmarks}}$, which represents the measurement noise covariance for all observed landmarks.

Activity 4: Optimizing Landmark Configuration with Range-Bearing Sensors

In this final activity, with range-bearing sensors, we revisit configurations from Activity 2 to identify a new optimal configuration. By testing various layouts, we seek to minimize the number of landmarks needed to satisfy the standard deviation (2σ) constraint under 10 cm for all timesteps.

2.2.2 Unconstrained MPC Regulator

Activity 1: Designing and implementing System Matrices for MPC Regulation

To begin this task, the necessary configuration changes were applied to `robotnik.json` file as per the instructions. In this activity, we first designed and implemented the state transformation matrix, A, and control input matrix, B, within the `updateSystemMatrices` method of the `RegulatorModel` class. The matrices were constructed using the following structures:

$$A = \begin{bmatrix} 1 & 0 & -v_0 \Delta t \sin(\theta_0) \\ 0 & 1 & v_0 \Delta t \cos(\theta_0) \\ 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} \Delta t \cos(\theta_0) & 0 \\ \Delta t \sin(\theta_0) & 0 \\ 0 & \Delta t \end{bmatrix}. \tag{3}$$

where v_0 and θ_0 represent the robot's velocity and orientation at the point of linearization, and Δt is the time step.

We evaluated two approaches to linearizing the system dynamics:

- *Static Linearization*: Linearizing the system around the goal state at the beginning of the control loop and maintaining this configuration without further updates.
- *Dynamic Linearization*: Linearizing the system around the current state-control pair, updating A and B at each time step.

To compare these approaches, we kept the initial states same and close to the origin. We used the default cost matrix configuration and prediction horizon (with $Q_{\text{coeff}} = [310, 310, 80]$, $R_{\text{coeff}} = 0.5$, and $N_{\text{mpc}} = 10$).

Activity 2: Tuning MPC Parameters: Prediction Horizon and Cost Matrices

In this activity we experimented with various values for the state cost matrix Q, control input cost matrix R, and prediction horizon N_{mpc} for the MPC to achieve effective regulation and stability. It's important to note that from this point onwards, we only apply the dynamic linearization approach, updating the system matrices around the current state-control pair at each timestep.

The state cost matrix Q prioritizes different state variables, penalizing deviations in position and orientation to ensure that the robot tracks the desired trajectory closely. The control cost matrix R penalizes large or abrupt control inputs, thus promoting smooth control actions. The prediction horizon N_{mpc} determines the number of steps into the future that the MPC considers for decision-making.

To set up the cost matrices, we first defined the coefficients Q_{coeff} and R_{coeff} in scalar or vector form. These coefficients were then passed to the `setCostMatrices` method within the `RegulatorModel` class, which constructed the full Q and R matrices. We started by testing on the default values:

$$Q_{\text{coeff}} = \begin{bmatrix} 310 & 310 & 80 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}, \quad N_{\text{mpc}} = 10$$

We then made suitable adjustments to optimize stability and control. Additionally, we found that our initial adjustments were 'overfitted' to our specific test scenario with a starting point at (2,3). To generalize, we redesigned the system matrices A and B to better handle a wider range of initial positions. This adjustment, detailed in Section 3.2.2, enabled the controller to maintain effective regulation from various starting points.

Finally, we tested the stability of the MPC by assessing its sensitivity to minor changes in the selected values of Q and R, scaling them up and down by 5%.

Activity 3: Adding the Terminal Cost Matrix P

MPCs with a large prediction horizon can be computationally very expensive and may not sustain the update rates required for smooth and stable control. In this activity we address this issue by incorporating a terminal cost matrix P.

The computation of P can be performed by solving the discrete-time algebraic Riccati equation (DARE), depending on the system's dynamics and time discretization, and it can be computed as follows:

$$P = Q + A^T PA - (A^T PB)(R + B^T PB)^{-1}(B^T PA) \tag{4}$$

where Q , R are the cost matrices, and A, B are the system matrices.

This matrix P represents the cost associated with the predicted state at the final step of the prediction horizon and helps guide the control actions towards desired regulation and stability. The terminal cost term added to the cost function can be expressed as:

$$J_{terminal} = x_{t+N}^T P x_{t+N} \tag{5}$$

This new cost term, $J_{terminal}$, is designed to approximate the long-term cost in a finite horizon by capturing the future impact of the terminal state x_{t+N}. As a result, the controller gains awareness of the state at the final step, enabling it to stabilize the system with a reduced horizon length, N.

To implement this, we created a new method, `compute_P`, inside the `RegulatorModel` class, where we followed an iterative approach for solving the discrete-time algebraic Riccati equation (DARE). Then, in Q_{bar}, which is the extended state cost matrix for the full prediction horizon N, we replaced the last block with P as the terminal cost matrix at the end. We then used this updated Q_{bar} matrix in the computation of the cost functions H and F. In this activity, we assessed the impact of P on the MPC's performance with a reduced time horizon.

2.2.3 Integration of EKF into the Full simulator

Activity 1: Integrating the EKF into simulator

The objective of this task is to integrate the Extended Kalman Filter (EKF) developed in Task 2.2.1 into the full robot simulator used in Task 2.2.2. We configured the noise parameters in `robotnik.json` by enabling the `noise_flag` and setting `base_pos_cov` and `base_ori_cov` to 1, adding noise to position and orientation measurements. This integration was implemented in a new copy of the `differential_drive.py` file labeled as `differential_drive_ekf.py`. A detailed implementation is described in the Results section 3.3

Activity 2: Validating results

In this activity, we validate the EKF's integration by comparing the estimated states with the true states provided by the simulator. For faster simulations, we temporarily disabled the terminal cost matrix P and reverted the values for Q, R, and N as follows:

$$Q_{\text{coeff}} = \begin{bmatrix} 310 & 310 & 510 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 1.1 & 0.16 \end{bmatrix}, \quad N_{\text{mpc}} = 10$$

Though the EKF worked with P enabled, we disabled it here for faster simulation to explore the results under noise.

We collected the estimated values (x, y, and θ) for path estimation and compared them to the ground truth from the simulator. We used a square boundary landmark configuration with a side length of 8 and a total of 20 landmarks.

2.2.4 Comparison of Robot's Performance with and without the Extended Kalman Filter

Activity 1 & 2: Designing Test Regime for comparing MPCT and MPCK, and Analysis of Results

In this task, we compare the two approaches: MPC with ground truth (labeled MPCT), described in Section 2.2.2, and MPC with EKF integration (labeled MPCK), described in Section 2.2.3

From the earlier tasks, we estabilished that the MPCT controller struggles to regulate and stabilize effectively in the presence of noise. To address this, the EKF-integrated MPCK controller was introduced to continuously estimate and linearize the system around the current state estimates, thereby making it a better choice for real-world applications. While MPCK has proven more effective for noisy environments, we aim to perform a fair comparison between MPCT and MPCK in this task. To achieve this, we disabled noise to create an ideal test environment in which both controllers operate on equal footing.

For the first two activities, we set up a strategy to test and analyze these two methods. In the previous tasks, we always used a fixed starting coordinate $(2, 3)$, but to effectively test our regulator's stability and adaptability we designed a new set of starting positions. We chose three starting coordinates to explore some interesting results: $(-4, 0), (1, -3)$ and $(3, 1)$. The start points are defined at appropriate places in the code file and the robot's JSON configuration.
We then analyze each of these initial conditions for both MPCT and MPCK, focusing on three key metrics:

- *Steady-state error* in position and orientation: This metric represents the final, stable deviation from the target once the robot has settled. It was calculated once any oscillations around the target ceased.
- *Settling time* for position and orientation: This is the time required for the robot to stabilize and settle down within an tolerable range of the target.
- *Overshoot* in position and orientation: Overshoot indicates how far the robot's response exceeded the target during the initial movement.

Computing these metrics analytically required substantial overhead, making the simulations slow. Therefore, we chose to visually inspect the trajectory plots to approximate the value of these metrics.

Activity 3: Trajectory and Error Comparison

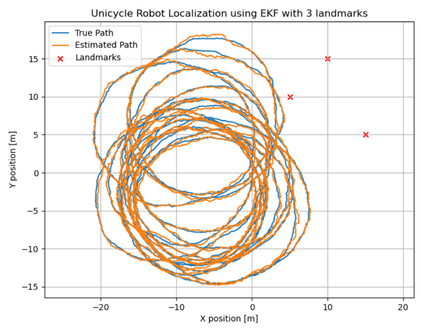
In this final activity, we plot the state trajectories for x, y, and θ and Localization Map for each start point to compare MPCT and MPCK. For the EKF implementation we define a square boundary grid with 20 landmarks for the observations. These graphs are used to evaluate the robot's ability to reach its target position from varying initial conditions and assess the different error metrics.

3 Results and Observations

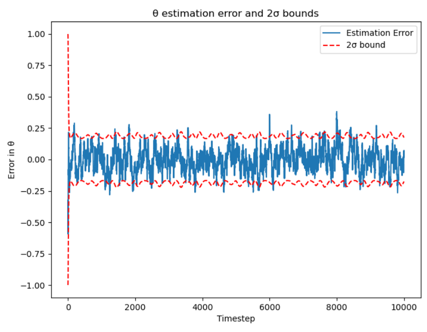
3.1 Task1: Extended Kalman Filter (EKF)-based estimation algorithm

3.1.1 Activity 1: Evaluating the Initial EKF Localization System

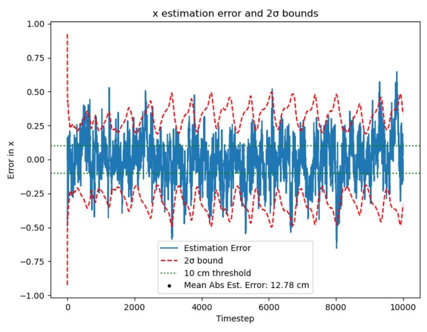
Using the initial default configuration with landmarks at [5, 10], [15, 5], and [10, 15], the resulting error plots in Figure 1 indicated that the 2σ exceeded the 10 cm threshold across all timesteps, failing to meet the performance requirement. This suggested that the 3 default landmarks led to a very weak configuration impairing the localization accuracy. Further, it could also be inferred that the range-only measurements lacked the orientation data needed for precise pose estimation, further affecting overall localization performance.



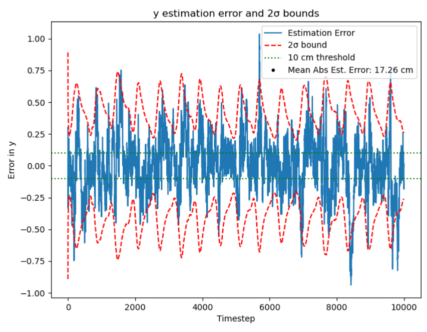
(a) Localization Map



(b) θ estimation error



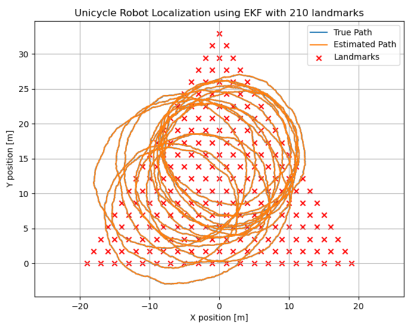
(c) x estimation error



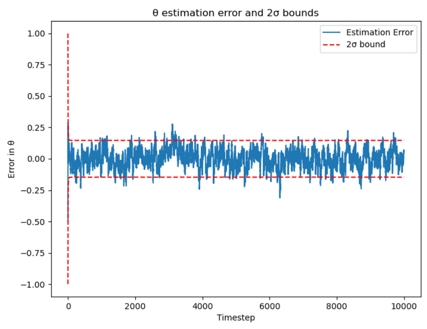
(d) y estimation error

Figure 1: Error plots for localization using the initial configuration with 3 landmarks and range-only measurements

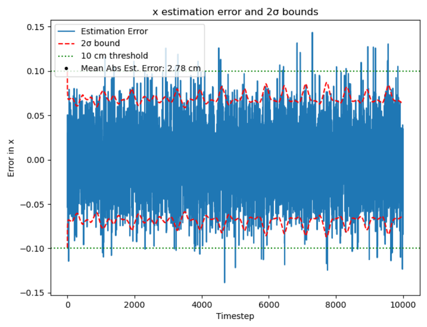
3.1.2 Activity 2: Increasing Landmarks for better Localization



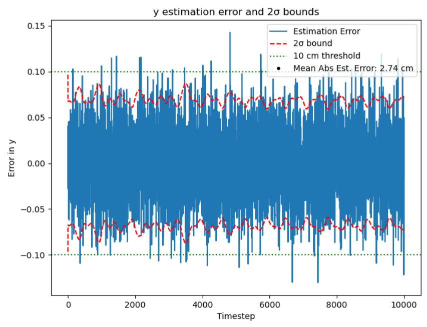
(a) Localization Map



(b) θ estimation error



(c) x estimation error



(d) y estimation error

Figure 2: Error plots for localization using a triangular grid configuration with 210 landmarks and range-only measurements

We began by testing a 3x3 square grid with 9 landmarks, spacing them at increments of 2, 10, and 20 units. While we noticed some slight improvements in the error bounds, none of these setups were able to consistently meet the 10 cm threshold. Afterward, we moved to a 7x7 grid with 49 landmarks and then a 10x10 grid with 100 landmarks, both using 2 units of spacing. These configurations showed much better results, but it was only with a 16x16 grid of 256 landmarks that we finally satisfied the standard deviation constraint. Admittedly, this configuration required a bit more computational power, but it was our first setup to truly meet the performance requirements. We also explored circular and triangular grids while increasing the number of landmarks in a similar fashion. These configurations yielded similar performance, achieving the 2σ constraint around the 200-250 landmark mark.

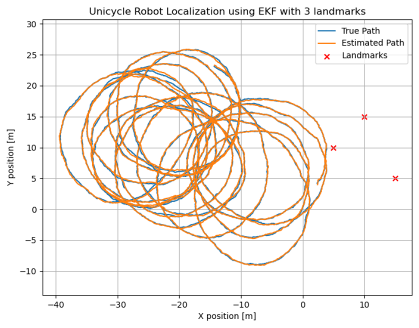
Next, we examined boundary layouts for each shape, which met the performance requirement with around the same 200 landmarks range as before. However, we noticed that along with high computational demands, these layouts were more spread out over the space and the sensors were placed quite close together along the perimeter. This made us think about whether it was practically feasible to implement such configurations.

Overall, these findings indicated how critical the number of landmarks is for achieving accurate localization. After a thoughtful trade-off between the number of landmarks, their spread, and the distance between them, we concluded that a triangular grid with 210 landmarks was the best configuration. The standard deviation (2σ) was below

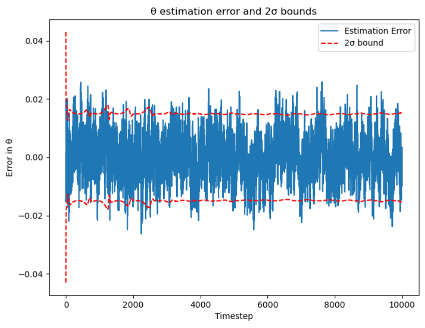
10cm threshold for all timesteps. This setup provided an effective balance between accuracy and computational efficiency, as illustrated in Figure 2.

3.1.3 Activity 3: Modifying the Sensing System

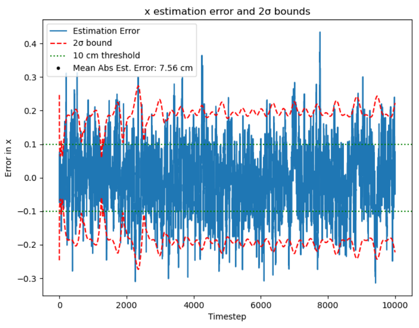
Introducing range and bearing measurements with the original three landmarks showed significant improvement in error plots as illustrated in Figure 3. While localization accuracy improved with reduced error bounds, it still fell short of the required performance standards.



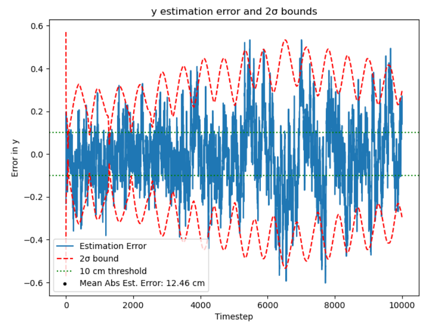
(a) Localization Map



(b) θ estimation error



(c) x estimation error



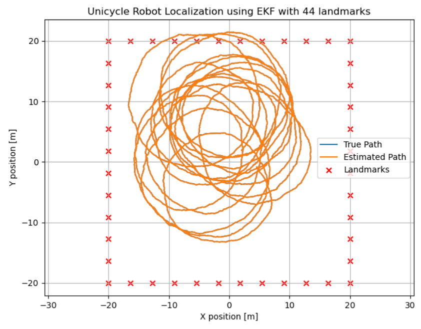
(d) y estimation error

Figure 3: Error plots for localization using the initial configuration with 3 landmarks and range-bearing measurements

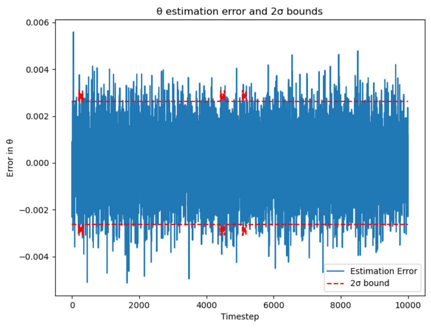
3.1.4 Activity 4: Optimizing Landmark Configuration with Range-Bearing Sensors

With both range and bearing measurements now, we revisited our grid and boundary configurations. We found that a 12x12 square grid with 144 landmarks was sufficient now to meet the standard deviation constraint consistently. We also achieved similar performance with circular and triangular grids, requiring only 120-140 landmarks which was a significant improvement compared to the previous configurations, which needed at least 200 landmarks when using range-only measurements.

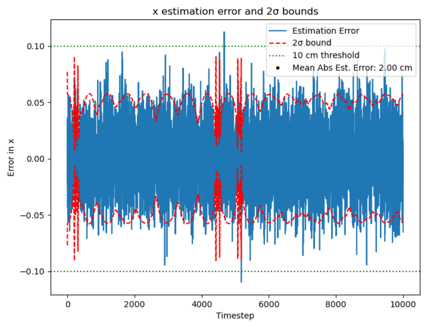
The most notable improvement, however, came from implementing boundary configurations in this case. With just 44 landmarks arranged in a square perimeter, we



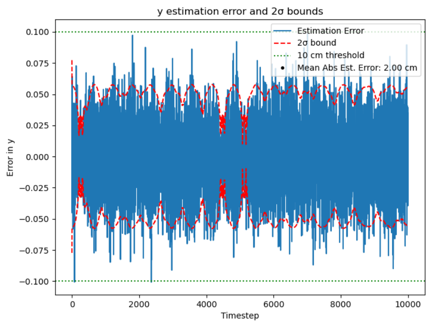
(a) Localization Map



(b) θ estimation error



(c) x estimation error



(d) y estimation error

Figure 4: Error plots for localization using a square boundary configuration with 44 landmarks and range-bearing measurements

were able to bring the standard deviation (2σ) under the 10 cm threshold consistently for all timesteps. This was also the case with the circular and triangular boundary layouts with similar number of landmarks. In addition, unlike the earlier range-only setups these configurations did not spread the landmarks extensively across the area.

Our best-performing configuration, which consistently met the performance requirements, is illustrated in Figure 4.

3.2 Task 2: Replace the Fixed Controller with an Unconstrained MPC Regulator

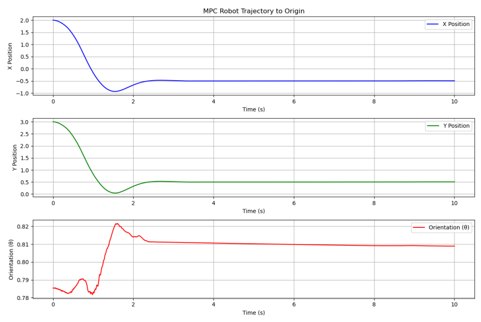
3.2.1 Activity 1: Design and Implementation of System Matrices for MPC Regulation

In this activity we first designed the A and B matrices from the structure as described in Section 2.2.2. In the static approach, where the system was linearized once around the origin, the robot ended approximately at (-0.5, 0.5) with an orientation of 0.79 rad. The trajectories are shown in Figure 5.

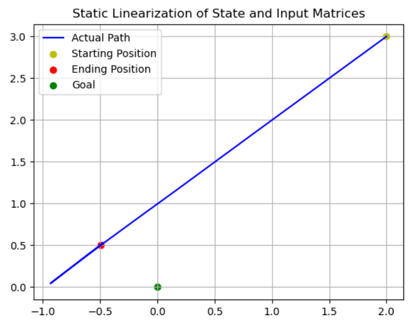
With the dynamic approach, where we updated the matrices A and B at each time step, the robot arrived closer to the desired endpoint, ending approximately around (-0.3, 0.5) with an orientation of 0.55 rad. This is illustrated in Figure 6.

While neither approach achieved exact regulation to the origin, the dynamic approach produced a slightly smoother trajectory and a marginally closer result to the goal. Therefore we can infer that dynamic approach is the better choice here.

To further verify our findings, we tested both methods from a starting position farther from the origin at (5, 3). In this case, the static approach resulted in a significant error, with the robot ending at approximately (30, -10), while the dynamic approach brought it to (0.8, 0.2). This helped validate our inference that the dynamic approach is the more effective approach.

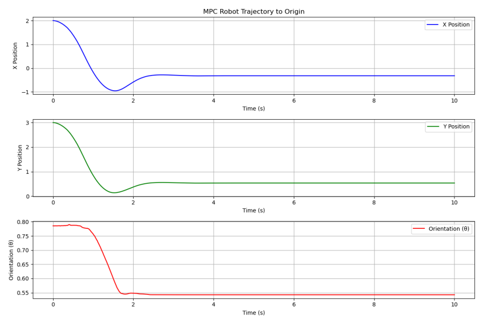


(a) Regulation of X, Y, and θ

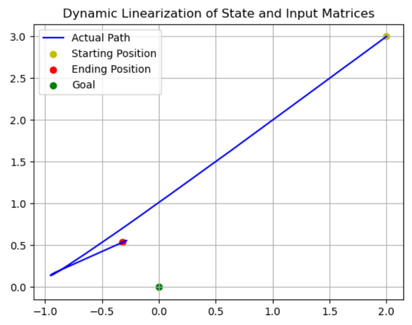


(b) 2D view of the robot's trajectory

Figure 5: Trajectories from linearizing dynamics once



(a) Regulation of X, Y, and θ



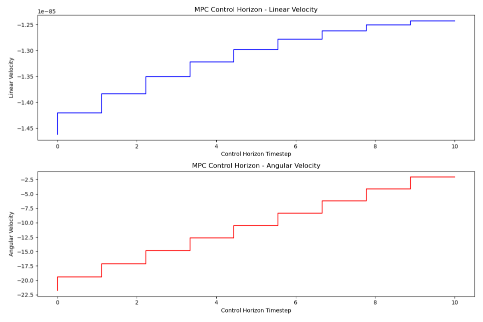
(b) 2D view of the robot's trajectory

Figure 6: Trajectories from linearizing dynamics at each time step

Additionally, we plotted Figure 7 which shows the control horizon when the system is trying to bring the orientation of the robot to 0°. We can see that at the end of the horizon, the control input is diminishing which could be a possible reason as to why the orientation wasn't able to reach 0 perfectly. Similarly, Figure (10b) shows the control horizon when the system is trying to bring the robot's X position to 0.

3.2.2 Activity 2: Tuning MPC Parameters: Prediction Horizon and Cost Matrices

Our initial configuration from Activity 1 did not achieve proper regulation, so we first increased N_{mpc} to 15 to provide a more extended look-ahead period. To further reduce the robot's deviation from the origin, we increased the weights on x, y, and θ in Q


(a) Control Horizon for regulating the robot's orientation (θ)

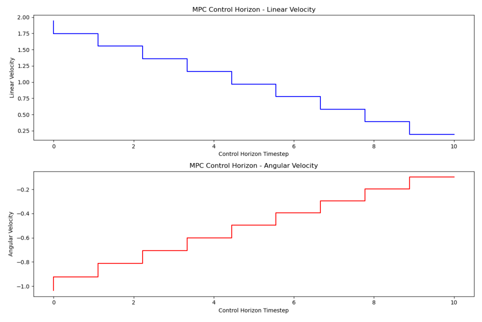
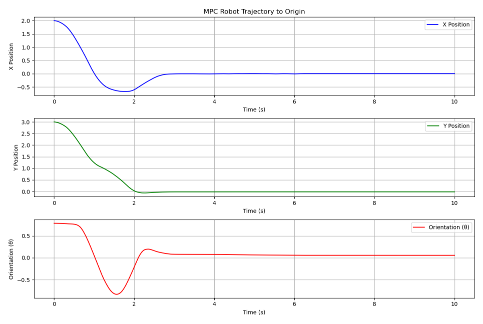

(b) Control Horizon for regulating the robot's X position

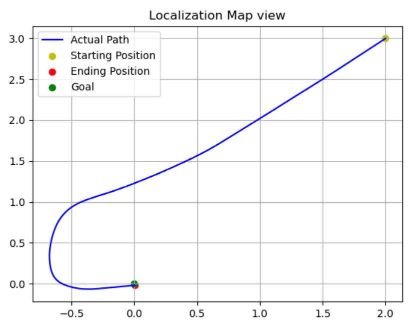
Figure 7: MPC Control Horizon for Regulating of θ and X

to penalize deviations more strongly. We also tuned R to balance this with control smoothness. At this point the configurations used were:

$$Q_{\text{coeff}} = \begin{bmatrix} 400 & 501 & 599 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 1.45 & 0.2 \end{bmatrix}, \quad N_{\text{mpc}} = 15$$

This setup resulted in a smooth trajectory with some overshooting, as shown in Figure 8, nonetheless we finally achieved precise regulation and the robot was able to end up at the origin.


(a) Regulation of X, Y, and θ

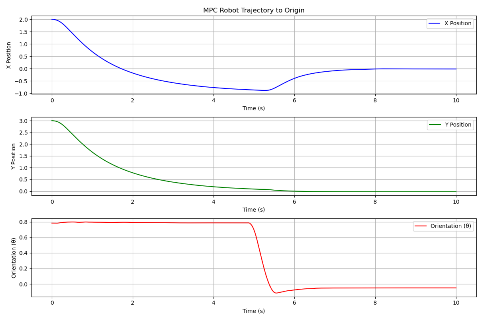


(b) 2D view of the robot's trajectory

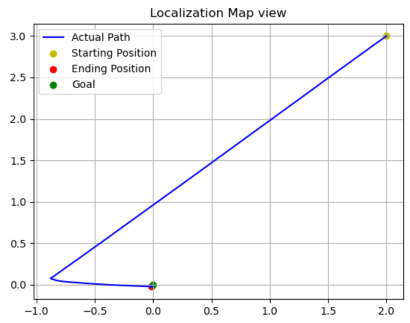
Figure 8: Trajectories for tuned MPC parameters - Case of overfitting

However at this point, this configuration overfit to the starting point (2,3) and struggled with other starting points. To create a more generalized solution, we revised the implementation of our state matrices A and B with conditional logic in `updateSystemMatrices`. In this method, we prioritized aligning along the y-axis first, continuing this alignment until the robot was within a specified tolerance of 0.1 units. Once this condition was satisfied, we then focused on regulating the orientation θ before aligning along the x-axis to reach the origin. This systematic approach helped the robot to stabilize at the desired end state from various initial positions.

To account for this new system, we set the prediction horizon N_{mpc} back to 10 to balance computation cost, and adjusted the cost matrices. The Q matrix defined below provided a suitable weight for the bearing (θ) while balancing the x and y penalties.



(a) Regulation of X, Y, and θ



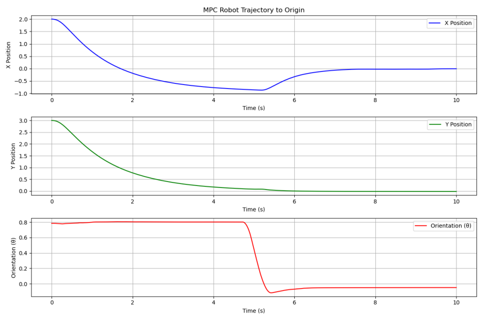
(b) 2D view of the robot's trajectory

Figure 9: Trajectories for fine-tuned MPC parameters

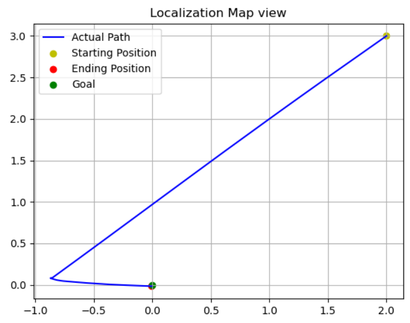
For control costs, R was also modified and the choices at this point were:

$$Q_{\text{coeff}} = \begin{bmatrix} 310 & 310 & 510 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 1.1 & 0.16 \end{bmatrix}, \quad N_{\text{mpc}} = 10$$

This setup achieved very good regulations from multiple starting positions. The trajectory graphs are illustrated as shown in Figure 9.



(a) Regulation of X, Y, and θ



(b) 2D view of the robot's trajectory

Figure 10: MPC with fine-tuned parameters scaled up

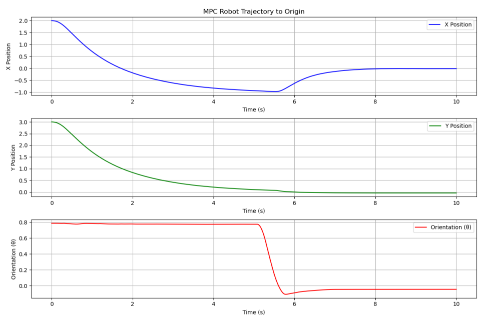
The choice of the weights in the cost matrices plays an important part in regulating the robot. For the final objective for this activity, we introduced small changes to the selected Q and R values by scaling up and down by 5% and assessed the controller's stability. The scaled-up configurations are:

$$Q_{\text{coeff}} = \begin{bmatrix} 325.5 & 325.5 & 535.5 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 1.15 & 0.17 \end{bmatrix}, \quad N_{\text{mpc}} = 10$$

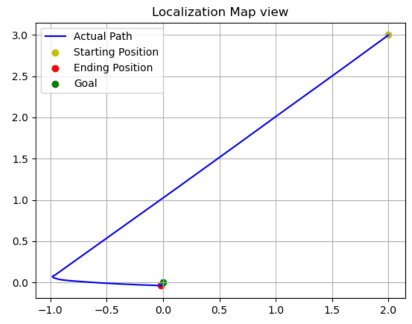
And the scaled-down configurations are:

$$Q_{\text{coeff}} = \begin{bmatrix} 294.5 & 294.5 & 484.5 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 1.05 & 0.15 \end{bmatrix}, \quad N_{\text{mpc}} = 10$$

Figures 10 and 11 illustrate the trajectory graphs for these cases respectively. Both scaled configurations maintained system stability with only minor deviations in trajectory. These results help us infer that our chosen values for Q, R, and N_{mpc} ensure stability under small changes, and that our system design is robust.



(a) Regulation of X, Y, and θ



(b) 2D view of the robot's trajectory

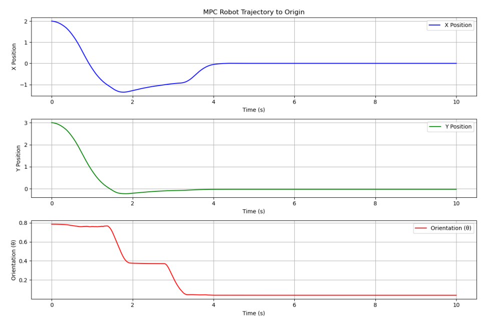
Figure 11: MPC with fine-tuned parameters scaled down

3.2.3 Activity 3: Adding the Terminal Cost Matrix P

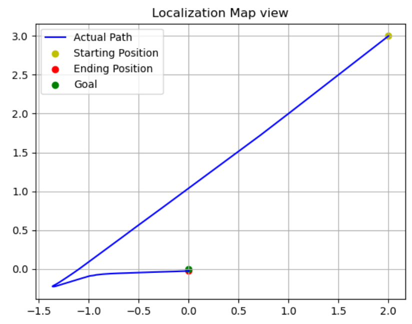
Building upon the previous activity's configuration and parameter choices, we implemented the P matrix logic. For this implementation, we observed that the robot did not regulate properly at the origin and exhibited some aggressive actions ("flying" in the simulation at the start). To address this, we reduced excessive control by increasing the weight coefficients in the cost matrix. This adjustment helped mitigate the erratic behavior and allowed the robot to regulate more effectively around the origin. The chosen values were:

$$Q_{\text{coeff}} = \begin{bmatrix} 400 & 400 & 1000 \end{bmatrix}, \quad R_{\text{coeff}} = \begin{bmatrix} 10 & 1 \end{bmatrix},$$

Next, we iteratively reduced the predictive horizon until the robot began to lose stability. Surprisingly, we observed that the controller remained stable even when $N_{\text{mpc}} = 1$, though the robot ended up slightly away from the origin. Ultimately, we selected $N_{\text{mpc}} = 5$ as it offered near-perfect regulation and stability, and it is illustrated in Figure 12.



(a) Regulation of X, Y, and θ



(b) 2D view of the robot's trajectory

Figure 12: Trajectories for MPC enabled with terminal cost matrix P and N=5

Thus, by adding the terminal cost P to the cost function, the system is able to reduce the horizon length N while maintaining almost perfect stability and control. In real-time applications, calculating a long horizon can be computationally expensive, especially in real-time settings. By using a shorter horizon length with a terminal cost, the robot can make efficient, stable decisions that approximate long-term goals, allowing it to navigate effectively without overloading its processing resources. This demonstrates the importance of the P matrix.

3.3 Task 3: Integrate the EKF into the Full Simulator for the Differential Drive Robot

3.3.1 Activity 1: Integrate EKF into the Full simulator

We first initialized the Extended Kalman Filter (EKF) by creating an instance of the `RobotEstimator` class and defined landmarks using the `Map` class. At this point, we already had an implementation of the Model Predictive Control (MPC) in our `differential_drive_ekf.py` file.

The EKF is used to estimate the robot's current state (position and orientation) based on noisy measurements and control inputs. It was configured to receive the current control input (`u_mpc`). Measurements are obtained from observed distances and bearings to known landmarks in the environment, as defined in the function `landmark_range_bearing_observations`. In each time step, the EKF first performed a prediction step using `set_control_input` and `predict_to` methods. This was followed by an update step that incorporates sensor measurements to reduce estimation error via the `update_from_landmark_range_bearing_observations(y)` function. This process produced an updated state estimate and covariance for the next control loop iteration.

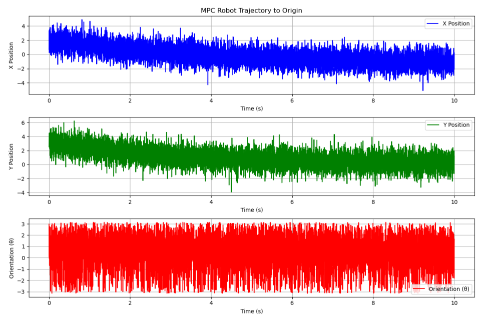
The Model Predictive Controller (MPC) linearized the system dynamics around the current EKF state estimate and generated a control sequence that minimizes the cost function over the prediction horizon N_{mpc}. Based on this cost function, the MPC computes an optimal control sequence, which we then convert into wheel velocities for the robot simulation. This integration of EKF and MPC allowed the robot to navigate accurately and efficiently even with the presence of noise, with the EKF's refined state estimates enhancing the MPC's control precision.

3.3.2 Activity 2: Validate the results

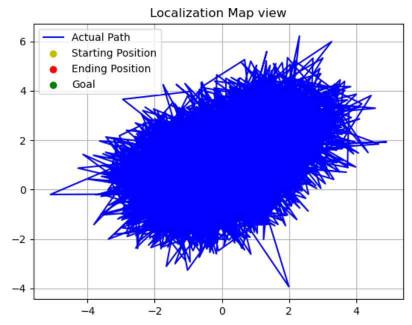
We first took a look at the robot's trajectory without the EKF, which is shown in Figure 13. As expected it demonstrates that the robot fails to localize correctly due to noise.

In contrast, Figure 14 shows the results with EKF integration, where the robot successfully localizes and regulates around the origin with high accuracy, even in the presence of noise. The EKF-enhanced MPC controller predicted the true states of the robot closely, with the x and y trajectories aligning well with minimal deviation and the θ trajectory showing minor fluctuations.

From these observations, we conclude that integrating the EKF with the MPC controller significantly improved the robot's ability to navigate and regulate around the target origin even under noisy conditions.

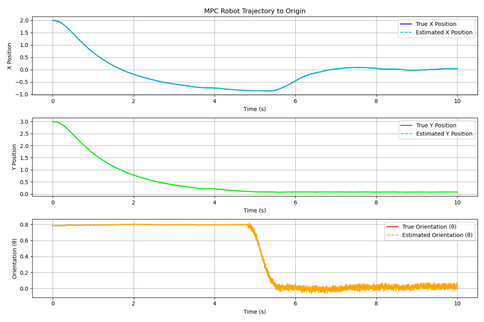


(a) Regulation of X, Y, and θ

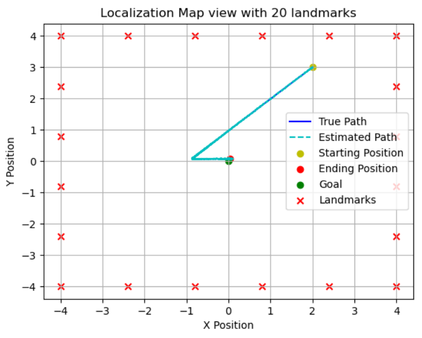


(b) 2D view of the robot's trajectory

Figure 13: Trajectories with noise enabled and without EKF implementation



(a) Regulation of X, Y, and θ



(b) 2D view of the robot's trajectory

Figure 14: Trajectories with noise enabled and EKF implementation

3.4 Task 4: Compare Robot Performance With and Without the Kalman Filter

3.4.1 Activity 1 & 2: Test Regime Design for Comparing MPCT and MPCK, and Analysis of Results

In these activities, we analyzed the results related to steady-state error, settling time and overshoot metrics for the three different initial conditions.

Tables 1 and 2 presents these metrics for both MPCT and MPCK obtained from the graphs we plotted:

Starting Position	**[-4, 0]**	**[1, -3]**	**[3, 1]**
Steady-state position error (m)	0.067	0.093	0.021
Steady-state orientation error (rad)	0.012	0.067	0.052
Overshoot for position (m)	0	0.117	0.009
Overshoot for orientation (rad)	0	0.057	0
Settling time for position (s)	2.842	7.012	5.912
Settling time for orientation (s)	0.911	6.812	4.023

Table 1: Metrics for MPCT

Starting Position	**[-4, 0]**	**[1, -3]**	**[3, 1]**
Steady-state position error (m)	0.098	0.014	0.072
Steady-state orientation error (rad)	0.006	0.022	0.045
Overshoot for position (m)	0.004	0	0.006
Overshoot for orientation (rad)	0	0.027	0
Settling time for position (s)	2.954	6.645	5.014
Settling time for orientation (s)	1.712	6.256	4.007

Table 2: Metrics for MPCK

For *steady-state errors*, we observed that the positional steady-state errors were comparable for MPCK and MPCT. However for the orientation, we could see a pattern where MPCK consistently maintained a lower steady-state orientation error across all starting positions compared to MPCT, with the largest difference at $[1, 3]$, where MPCK's error is nearly one-third of MPCT's (0.022 vs. 0.067 rad).

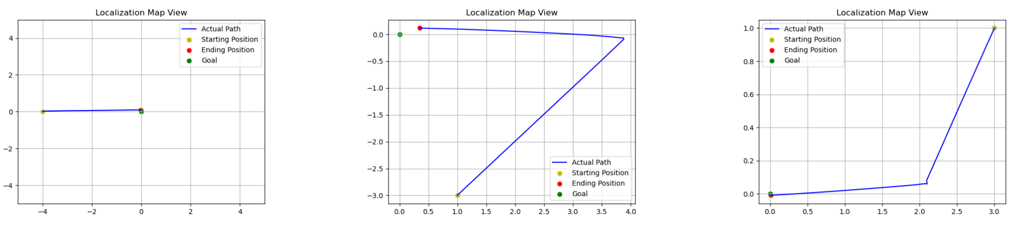
As for the *overshoot* metric, MPCK demonstrates better control over overshoot in position, with all values close to zero, compared to MPCT. This is especially evident at the $[1, 3]$ starting position, where MPCT shows 0.117 m overshoot in position versus 0 m for MPCK. For orientation overshoot, both controllers perform well, with MPCK showing minimal values. MPCT experiences slightly higher overshoot for the $[1, 3]$ position but shows no overshoot at the other starting points.

In terms of *settling time*, MPCK achieves faster settling times for both position and orientation in almost all cases. There was only one exception to this pattern at $[-4, 0]$ where MPCK settled almost twice as slowly (1.712 s) compared to MPCT (0.911 s).

The comparable and low error metrics for both these methods indicate that the MPCT and MPCK can both accurately reach and settle at the target positions and orientations. However, on marginal grounds, MPCK appears to handle varied initial conditions more adaptively than MPCT. The smaller steady-state errors and minimal overshoot across all starting points suggest that MPCK is less sensitive to the robot's initial position, making it a better choice for scenarios with unknown or dynamic starting conditions.

3.4.2 Activity 3: Compare the trajectories and errors

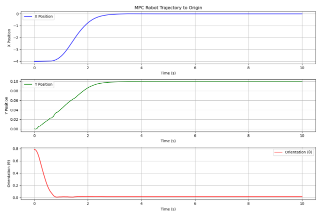
In this activity, the goal was to compare the performance in trajectories for MPCT and MPCK. Figure 15 shows a 2D view of 3 different initial base position trajectories with MPCT. These experiments were conducted with no noise in the system.



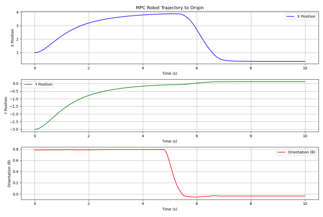
(a) [-4, 0] (b) [1, -3] (c) [3, 1]

Figure 15: MPCT - 2D view with different starting positions

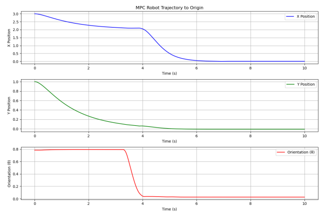
Similarly, Figure 16 shows the MPCT trajectories for x, y and θ from the same set of initial positions mentioned above.



(a) [-4, 0]



(b) [1, -3]

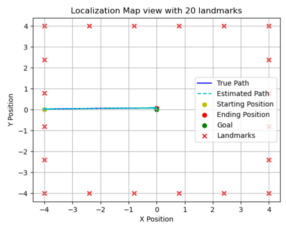


(c) [3, 1]

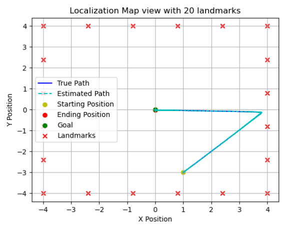
Figure 16: MPCT - Individual trajectories with different starting positions

From the results above, we can see that MPCT performs very well in reaching the target $[0, 0]$. In particular, from Figure 15, we observe that for the initial coordinate $[3, 1]$, the robot reaches the target perfectly, while for $[-4, 0]$ and $[1, -3]$, it comes very close to the target with minimal error. These small deviations are also confirmed in Figure 17a and Figure 16c, respectively.

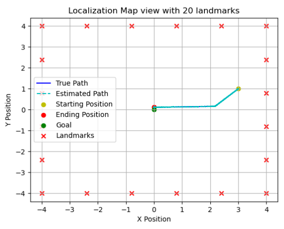
Figures 17 and 18 illustrate MPCK trajectories from identical initial positions.



(a) [-4, 0]

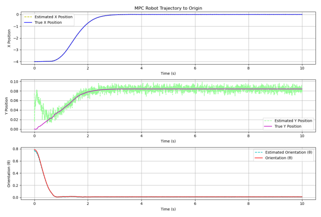


(b) [1, -3]

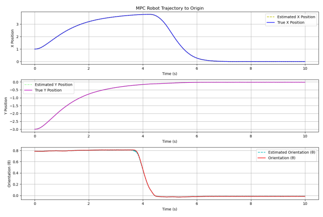


(c) [3, 1]

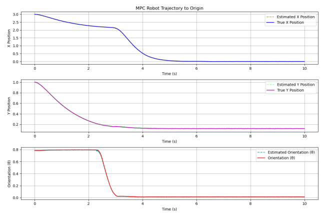
Figure 17: MPCK - 2D view with different starting positions



(a) [-4, 0]



(b) [1, -3]



(c) [3, 1]

Figure 18: MPCK - Individual trajectories view with different starting positions

Similar to MPCT, the MPCK trajectories shown in Figure 18 consistently reach the target with minimal deviations. In this case, the robot approaches the final goal with greater precision when starting from $[1, -3]$ compared to the other initial coordinates, $[-4, 0]$ and $[3, 1]$.

Regarding the metrics we described in the previous section, the values in Table 2 and Table 1 align well with the state trajectories in Figure 18 and Figure 16. With these figures, we can make deeper visual inference about the steady-state error and overshooting.

In Figure 18a for instance, an overshoot is evident around 2.25 seconds, which could be reduced by increasing the control input cost for the x position. Similarly, in Figure 18c another overshoot appears around 2 seconds. Conversely, with MPCT we don't see much overshooting, but we do observe signs of weak control. For example, in Figure 16c we see the robot took a very long time to settle along the y position. This could be mitigated by increasing the y position cost weight in the Q matrix or reducing the control input cost.

Overall, both MPCT and MPCK show relatively low steady-state error, consistent with the values in Table 2 and Table 1. In some cases, however, steady-state error is not zero, particularly for the system's orientation, θ, as shown in Figure 18b.

Future work could involve fine-tuning the system parameters to further minimize steady-state error, overshooting, and weak control behavior, as well as to reduce the settling time. Additionally, exploring advanced control techniques, such as model predictive control with constraints or robust control methods, could enhance stability and performance under varying operating conditions. These improvements would aim to ensure a more responsive and accurate control system across different scenarios.

4 Conclusion

This project explored the integration of Extended Kalman Filter (EKF) and Model Predictive Control (MPC) for enhancing the navigation capabilities of a wheeled mobile robot. Through systematic experimentation and analysis across four major tasks, we gained valuable insights into optimal localization strategies and control methodologies.

In our investigation of EKF-based localization, we discovered that the initial three-landmark configuration was insufficient for accurate state estimation. Through extensive testing of various landmark configurations, we found that range-only measurements required approximately 200 landmarks to meet the (2σ) threshold requirement of under 10 cm. However, by incorporating bearing measurements alongside range data, we achieved comparable accuracy with just 44 landmarks in a square boundary configuration, demonstrating the significant advantage of multi-modal sensing in robotic localization.

The implementation of the MPC regulator revealed several key findings. Our comparison of static and dynamic linearization approaches showed that dynamic updating of system matrices at each timestep provided superior regulation performance. Through careful tuning of the cost matrices Q and R, along with the prediction horizon N, we achieved stable control across various initial conditions. The introduction of the terminal cost matrix P proved particularly valuable, allowing us to reduce the prediction horizon from N=10 to N=5 while maintaining control stability, thus potentially reducing computational overhead in real-world applications.

The integration of EKF with MPC (MPCK) demonstrated notable improvements over the standard MPC with ground truth (MPCT), particularly in terms of settling time and noise handling. Our comparative analysis across three different starting positions ([-4,0,0], [1,-3,0], and [3,1,0]) revealed that while both approaches achieved similar steady-state errors (below 0.1m for position and under 0.07 rad for orientation), MPCK consistently demonstrated faster settling times. This suggests that the EKF integration enhances the controller's responsiveness and stability under varying initial conditions.

Overall, this project demonstrates the effectiveness of combining EKF-based state estimation with MPC for robust robot navigation, while also highlighting the important trade-offs between sensor configurations, computational complexity, and control performance. The findings provide valuable insights for the practical implementation of autonomous navigation systems in real-world applications.