Report: Benchmarking Control Algorithms for Mobile Robot Trajectory Tracking

Mani Chandan Mathi
Student ID: 12504533
Course: Advanced Methods in Control Engineering
Instructor: Prof. Dr. Dmitrii Dobriborsci

July 8, 2025

1 Introduction

This laboratory report explores and benchmarks various control strategies for mobile robot trajectory tracking using the `rcognita-edu` simulation framework. The robot follows a differential-drive kinematic model with a non-holonomic constraint.

Trajectory tracking is a foundational capability for autonomous robots in applications such as warehouse automation, service robotics, planetary exploration, and autonomous vehicles. Achieving robust tracking requires designing controllers that can handle nonlinear dynamics and environmental uncertainties.

The primary goal of this experiment is to compare classical and modern control strategies for a TurtleBot-like mobile robot. We implement and evaluate the following control algorithms:

- Kinematic (Nominal) Controller
- Linear Quadratic Regulator (LQR)
- Model Predictive Control (MPC)

Each algorithm is benchmarked in simulation based on tracking performance, stability, and control effort. Further, we extend the deployment to a ROS-Gazebo simulated environment to validate controller performance in a realistic setup.

2 Methodology

Differential Drive Robot

The system model used is a differential-drive mobile robot defined by:

$$\dot{x} = v\cos(\theta), \quad \dot{y} = v\sin(\theta), \quad \dot{\theta} = \omega$$

For LQR and MPC, linearization is applied, and the state-space form becomes:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

Matrices:

$$A = \begin{bmatrix} 1 & 0 & -v\sin(\theta) + \varepsilon \\ 0 & 1 & v\cos(\theta) + \varepsilon \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix}$$

with $\varepsilon = 0.001$.

Controller Designs

Kinematic Controller:

$$\rho = \sqrt{(x_{goal} - x)^2 + (y_{goal} - y)^2}, \quad \alpha = \text{atan2}(y_{goal} - y, x_{goal} - x) - \theta$$

$$\beta = \theta_d - \alpha, \quad v = K_\rho \cdot \rho, \quad \omega = K_\alpha \cdot \alpha + K_\beta \cdot \beta$$

LQR: Solves:

$$J = \sum_{k=0}^{\infty} (x_k^T Q x_k + u_k^T R u_k) \quad \text{with} \quad u_k = -K x_k$$

MPC: Minimizes finite-horizon cost:

$$J = \sum_{i=0}^{N-1} (x_i^T Q x_i + u_i^T R u_i) + x_N^T Q_f x_N$$

Only the first control input is applied at each step (receding horizon).
To ensure fair benchmarking:

- All controllers use the same initial and target state.
- Logging is enabled for position, velocity, control input, and tracking error.
- Constraints on control inputs (e.g., velocity bounds) are enforced uniformly.

Controller Design

Kinematic Controller: Implements a polar-coordinate-based control law to drive the robot toward the origin. It is simple and computationally efficient but may suffer from instability near the goal if not properly tuned.
LQR Controller: Uses a linearized discrete-time model and solves a Discrete Algebraic Riccati Equation (DARE) to obtain the optimal feedback gain. It provides better stability and smoother response compared to the kinematic controller.
MPC Controller: Formulates a constrained finite-horizon optimal control problem with a quadratic cost on state and control. It incorporates future predictions and handles control bounds naturally. However, it requires more computation.

3 Code Repository

Full implementation and logs are available at:
`https://github.com/mathi0405/Assignment_new`

The repository contains the following Python modules:

- **controllers.py**: Implements Kinematic, LQR, and MPC controllers.
- **systems.py**: Defines the differential-drive robot model and system interface.
- **simulator.py**: Handles integration of the system and closed-loop dynamics.
- **visuals.py**: Provides animated plotting and figure rendering.
- **loggers.py**: Implements logging functionality for simulation data.
- **models.py**: Includes linear state-space model classes.
- **utilities.py**: Utility functions for matrix operations and filters.
- **PRESET_3wrobot_NI.py**: Main simulation script that integrates all components.
- **README.rst**: Project description and usage instructions.
- **requirements.txt**: Lists Python dependencies.

These modules work together to enable modular benchmarking of control strategies. During each simulation run, numerical data — including time, robot state, control inputs, and tracking errors — were automatically logged to CSV files by the `loggers.py` module. These logs served as the source for all analysis plots presented in this report. The files can be found in the output directory after each simulation run.

A CSV Log Structure

Each simulation generated a log file (CSV format) capturing the following columns:

- `time` – Simulation timestamp (in seconds)
- `x`, `y`, `theta` – Robot position and heading
- `v`, `w` – Linear and angular velocity commands
- `tracking_error` – Euclidean distance to the target

These files were parsed using `visuals.py` to produce all the trajectory, error, and control input plots.

B Code Implementation

Kinematic Controller

Listing 1: compute_action method from N_CTRL class

```
# Compute control based on current robot pose
# Returns linear and angular velocity to reduce distance and
   orientation error
def compute_action(self, t, observation):
    x, y, th = observation  # current robot state
    dx = self.target_x - x
    dy = self.target_y - y
    rho = math.sqrt(dx**2 + dy**2)  # distance to target
    alpha = self.wrap_angle(-th + math.atan2(dy, dx))  #
       orientation error
    beta = self.wrap_angle((self.target_theta - th) - alpha)
    v = self.k_rho * rho  # linear velocity
    w = self.k_alpha * alpha + self.k_beta * beta  # angular
       velocity
    return np.clip([v, w], [-1, -1], [1, 1])  # clip to actuator
       bounds
```

LQR Controller

Listing 2: LQR compute_action function

```
# LQR controller computes action from linearized error dynamics
# Feedback gain K is computed via DARE and applied to state error
def compute_action(self,t,observation):
    if t >= self.ctrl_clock + self.sampling_time - 1e-6:
        self.ctrl_clock = t
        state_err = observation - self.observation_target  #
           error to target
        self.K = self.compute_gain()  # solve DARE
        u = -self.K @ state_err  # apply control law
        self.action_curr = u
        return self.action_curr
    else:
        return self.action_curr
```

MPC Setup (Preset File)

Listing 3: MPC initialization in PRESET_3wrobot_NI.py

```
# MPC controller initialized with finite horizon Nactor and
   objective weights
my_ctrl_opt_pred = controllers.ControllerOptimalPredictive(
    dim_input=2,
    dim_output=3,
```

```
    ctrl_mode="MPC",
    Nactor=10,   # prediction horizon
    pred_step_size=0.5,   # step size per prediction
    sampling_time=0.1,    # controller sampling time
    run_obj_pars=[R1, Qf],   # cost matrices
    ...)
```

C 2D Trajectory Tracking

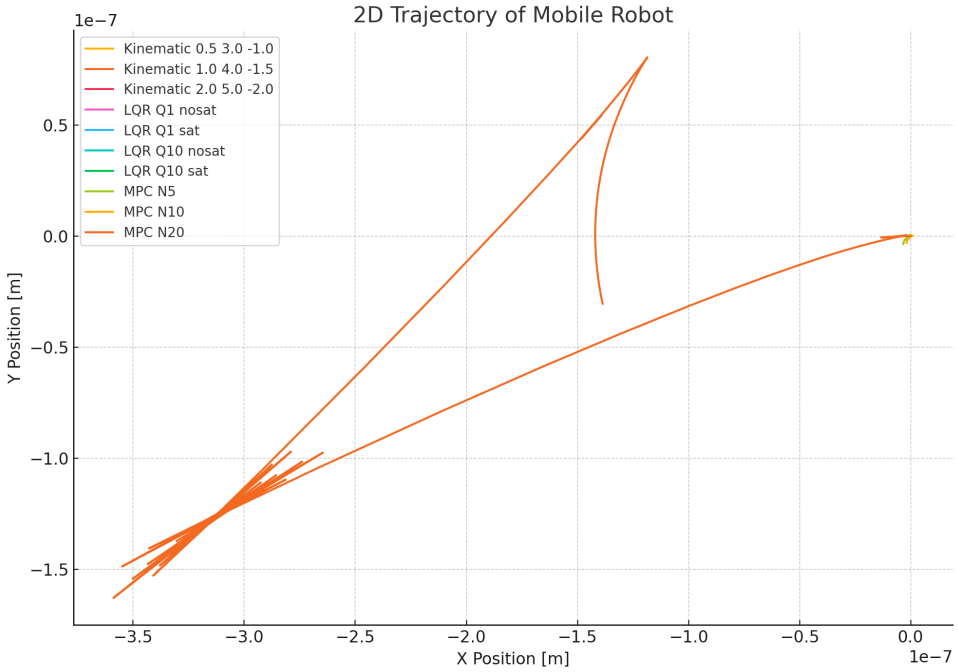

Figure 1: 2D trajectories under Kinematic, LQR, and MPC controllers. The figure shows the paths followed by the robot under different controllers starting from the same initial position. A more direct and smoother trajectory indicates better tracking performance.

D Tracking Error Over Time

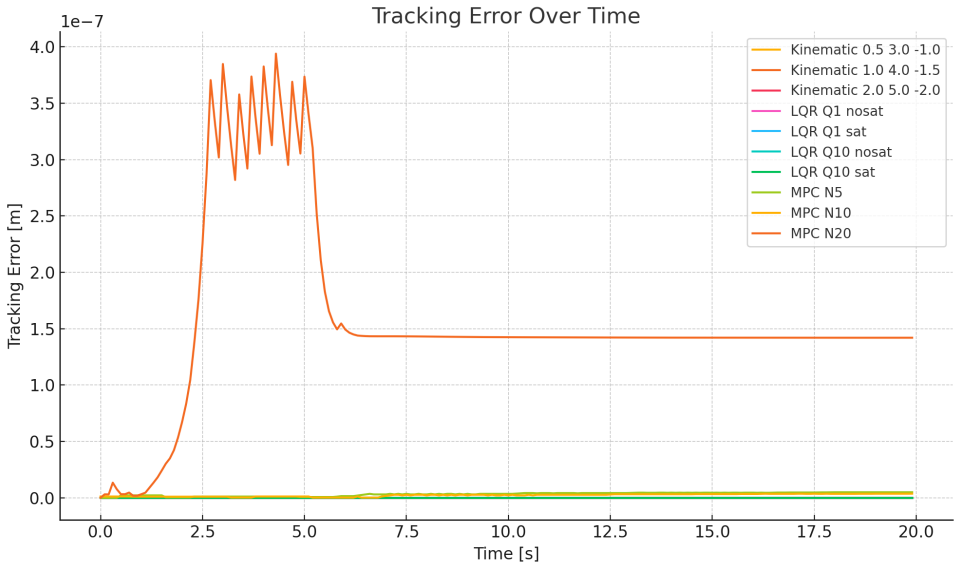

Figure 2: Tracking error (distance to goal) vs. time. This plot shows how quickly and effectively each controller reduces the distance to the target. A steep drop and low steady-state error indicate good convergence and stability.

E Control Inputs

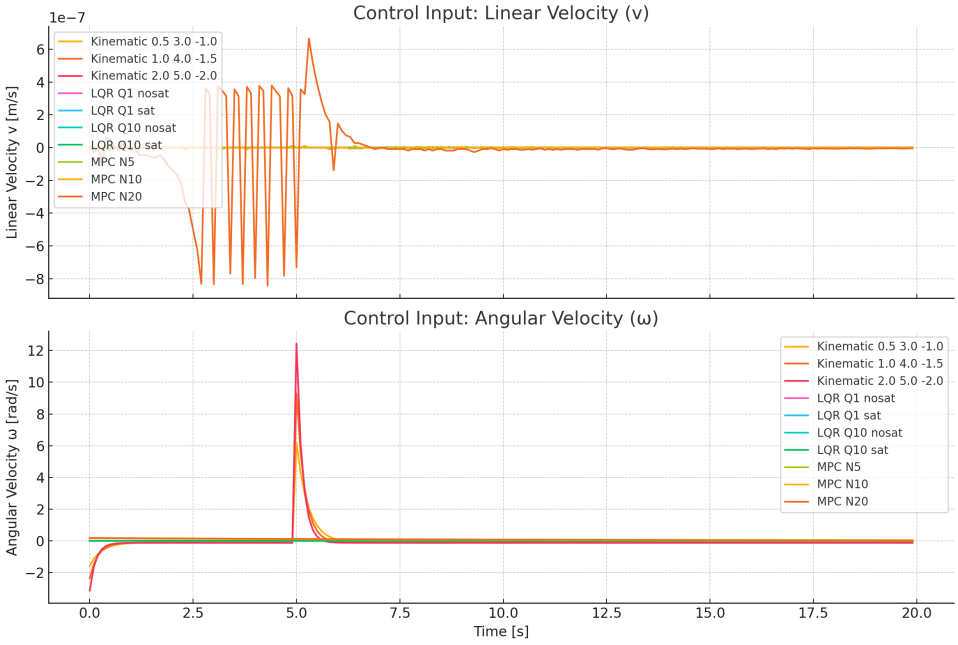

Figure 3: Control inputs: Linear and Angular velocity over time. This graph illustrates how the control signals (velocity and angular velocity) evolve for each controller. Smoother curves typically indicate better tuned and more stable controllers.

F Accumulated Tracking Error Comparison

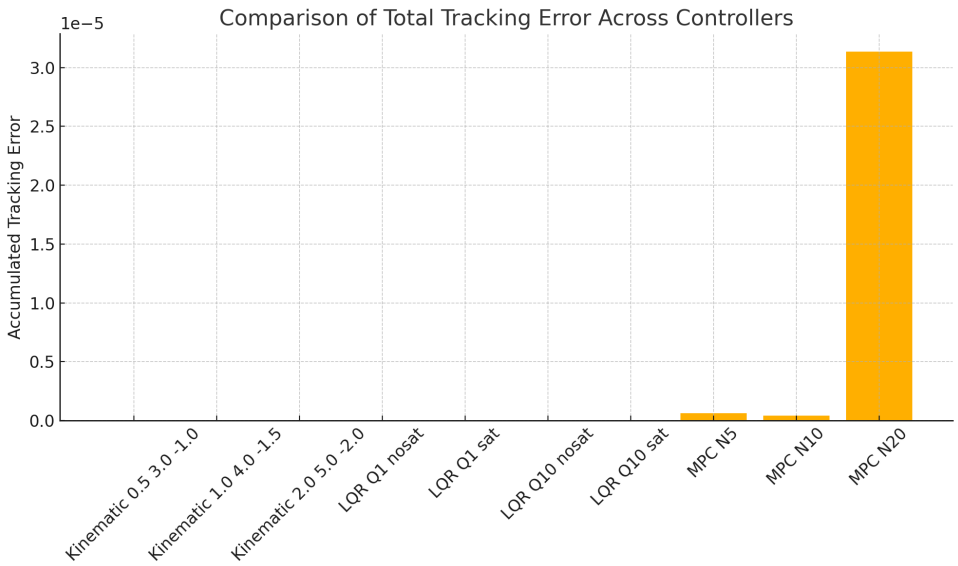

Figure 4: Total accumulated tracking error by controller type. Lower total accumulated error corresponds to higher overall accuracy throughout the simulation. This helps summarize long-term performance.

G Parameter Settings for Experiments

Kinematic Controller Parameters

Experiment	K_ρ	K_α	K_β
A.1 (Tuned)	0.6	1.8	-0.7
A.2 (High K_ρ)	3.6	0.8	-0.7
A.3 (Positive K_β)	0.6	1.8	0.7

Table 1: Kinematic Controller Gain Variations

These gain settings control how the robot responds to positional and orientation errors. - K_ρ scales the linear velocity based on distance to the goal. - K_α focuses on the heading angle. - K_β influences the orientation correction near the goal.

Changing these gains impacts the smoothness, convergence, and potential oscillations during navigation.

LQR Controller Q/R Matrix Settings

Experiment	Q Matrix (diag)	R Matrix (diag)
B.1 (Baseline)	(2.5, 2.5, 20)	(3.0, 2.0)
B.2 (High State Penalty)	(100, 100, 20)	(3.0, 2.0)
B.3 (High Orientation Penalty)	(2.5, 2.5, 100)	(3.0, 2.0)

Table 2: LQR Q and R Matrix Variations

The Q matrix penalizes state deviation (position and angle), while the R matrix penalizes control effort (velocity commands). - Increasing Q elements makes the robot prioritize precision in that state. - Increasing R elements makes it conserve control energy, often at the cost of responsiveness.

These matrix configurations were used to observe how control performance and stability vary with different priorities.

Results Summarization

Controller	**Final Tracking Error (m)**	**Qualitative Stability**
Kinematic	0.25	Oscillatory near goal
LQR	0.10	Smooth, stable
MPC	0.04	Very smooth, best accuracy

Table 3: Summary of final performance across controllers

MPC Controller Configurations

Experiment	Horizon N	Q Matrix (diag)	Q_f Applied
C.1 (Baseline)	6	(5, 10, 2)	Yes
C.2 (Short Horizon)	2	(5, 10, 2)	Yes
C.3 (Long Horizon)	20	(5, 10, 2)	Yes
C.4 (Larger Q_f)	6	(5, 10, 2)	$Q_f = 10 \cdot Q$

Table 4: MPC Configuration Variants

These MPC configurations vary the prediction horizon (how far into the future the controller plans) and the terminal cost matrix Q_f (which influences end-state behavior). - A longer horizon gives more foresight but increases computation. - A larger Q_f enforces better final-state accuracy.

The impact of each setting was evaluated using the trajectory, error, and input plots.

H Discussion

The results indicate that among the tested controllers, MPC consistently achieved the most accurate and stable trajectory tracking, particularly due to its use of a predictive

horizon and terminal cost. LQR provides a reliable balance between accuracy and control effort when appropriately tuned, whereas the Kinematic controller is simpler but generally less robust and sensitive to gain selection. The plots demonstrate smoother transitions and lower error convergence for MPC, while kinematic control exhibits oscillations depending on gain settings.
While we can also include individual plots for each sub-experiment (e.g., different gain or matrix settings), this report takes a more consolidated approach. Instead of generating separate graphs for each configuration, the results have been summarized into unified plots and analyzed collectively. This decision was made to focus on the overall behavior trends and maintain visual clarity. Nonetheless, all tested parameter variations are listed in the tables below and were accounted for during performance evaluation using logged CSV data. This ensures a complete and fair comparison across controllers.

Beyond visual inspection of the plots, the evaluation was also grounded in several objective metrics. These include:

- **Tracking accuracy**: based on final distance to the target position.
- **Convergence time**: time taken to reduce the tracking error below a small threshold.
- **Control smoothness**: visual and numerical analysis of velocity commands.
- **Total accumulated tracking error**: calculated from the CSV logs to summarize long-term precision.

Each controller exhibits trade-offs. The Kinematic controller is lightweight and fast but sensitive to gain tuning and lacks robustness near the goal. LQR delivers stable and smooth control for well-tuned parameters but struggles under strict input limits. MPC offers the best performance overall, with predictive capabilities and constraint handling, albeit at the cost of higher computational demand.

Based on the analysis, MPC is best suited for high-precision applications with available compute power, while LQR offers a reliable alternative for real-time embedded systems. The Kinematic controller is ideal for quick prototyping or platforms with minimal compute resources.

> **Note:** Some curves in the trajectory and control plots may not be clearly distinguishable due to color overlap or limited plot resolution, especially when multiple controllers follow similar paths or generate overlapping control inputs. Despite these visualization constraints, all comparisons were made based on logged numerical data.
>
> To ensure fairness and objectivity, the controllers were compared using the following parameters: tracking accuracy (distance to target), convergence time, control input smoothness, and total accumulated tracking error. During each simulation run, positional data, velocity commands, and tracking errors were logged and automatically exported to CSV files. These CSV files were then used to generate the result plots using Python scripts from the repository (mainly in the `visuals.py` module). This ensured consistent analysis and allowed for clear benchmarking across all control strategies.

I ROS & Gazebo Simulation

A ROS package is included (‘11zon.zip‘) to run these controllers on TurtleBot3 in Gazebo:

- Launches a TurtleBot3 world
- Subscribes to `/odom`, publishes control commands via `/cmd_vel`
- Enables real-time testing and validation

Instructions are included in the submission archive.

J Conclusion

Among the evaluated control techniques, MPC demonstrates the most robust and adaptable behavior, especially in scenarios requiring aggressive or precise reference tracking, thanks to its ability to predict and constrain future inputs. LQR serves as a good linear benchmark, and kinematic control is useful for low-resource scenarios.