LQR as a Nav2 Controller Plugin for TurtleBot3 Path Tracking

Tommy Oh
koa18@sfu.ca

Ansh Aggarwal
aaa275@sfu.ca

Daniel Senteu
dms26@sfu.ca

JunHang Wu
jwa337@sfu.ca

***Abstract*—We implement a Linear Quadratic Regulator (LQR) path-tracking controller as a C++ Nav2 plugin for ROS2 Humble, deployed on a simulated TurtleBot3 Burger in Gazebo. The controller linearizes unicycle dynamics in body-frame coordinates, solves the Discrete Algebraic Riccati Equation (DARE) for a steady-state gain matrix, and produces smooth velocity commands at 20 Hz. We benchmark LQR against two standard Nav2 controllers – DWB (Dynamic Window Approach) and a custom C++ MPPI (Model Predictive Path Integral) controller ported from course assignment A2 – across two planners (NavFn and Smac 2D), with three repeated runs per configuration. All 18 runs reached the goal. Our custom MPPI is the outright accuracy and smoothness winner (mean CTE $0.012\,\mathrm{m}$ and lowest $\mathrm{d}v/\mathrm{d}t$ and $\mathrm{d}\omega/\mathrm{d}t$ of the three controllers with NavFn) but is the slowest to the goal. LQR occupies the compute-cheap middle ground: it matches DWB on mean cross-track error (around $0.05\,\mathrm{m}$) while producing roughly $1.6\times$ smoother linear-velocity commands, at a small time cost relative to DWB+Smac 2D.**

I. Introduction

Autonomous ground robots need a controller that converts a global path from a planner into continuous velocity commands the robot can execute in real time. The Navigation2 (Nav2) stack on ROS2 [1], [7] is the standard framework for this task and ships with several controller plugins, notably DWB and MPPI. Each exposes a different tradeoff between tracking accuracy, command smoothness, and computational cost.

Linear Quadratic Regulation (LQR) is a classical optimal-control technique that, once the dynamics have been linearized around an operating point, produces a provably stable state-feedback law with a constant gain matrix. LQR is lightweight – the expensive solve happens once, offline, when the DARE is solved – and it penalizes state deviation and control effort through two interpretable weight matrices Q and R.

This project implements LQR as a first-class Nav2 controller plugin, evaluates it against DWB and a custom C++ MPPI on the TurtleBot3 Burger platform in Gazebo, and reports quantitative performance on path-tracking, time-to-goal, and command smoothness. The primary contributions are as follows.

1) A clean two-file C++ implementation that separates pure mathematics (`lqr_solver`) from ROS2 plugin glue (`lqr_controller`).
2) A full 3×2 benchmark against DWB and custom MPPI across NavFn and Smac 2D planners, with three repeated runs per configuration.
3) An analysis of Nav2 integration challenges encountered when deploying a custom controller plugin within the Nav2 lifecycle.

The remainder of this paper is organized as follows: Section II covers background and related work, Section III describes the algorithm and system design, Section IV details the implementation, Section V presents experimental results, Section VI analyzes the performance tradeoffs between controllers and planners, and Section VII concludes.

II. Background & Related Work

Unicycle model and LQR: The TurtleBot3 Burger [10] is modelled as a unicycle (Dubins car) with state $\mathbf{x} = [p_x, p_y, \theta]^\top$ and control $\mathbf{u} = [v, \omega]^\top$. The standard body-frame error linearisation for path tracking is covered in Tedrake's *Underactuated Robotics* (Ch. 8) [2] and Abbeel's CS287 lectures [9], and in our course assignment A3, which derives the infinite-horizon LQR gain as the solution of the Discrete Algebraic Riccati Equation.

Nav2 controllers: The Nav2 stack exposes a `nav2_core::Controller` plugin interface; every controller implementation – DWB, MPPI, or our LQR – satisfies the same lifecycle contract (`configure`, `setPlan`, `computeVelocityCommands`, `setSpeedLimit`) [8] and is loaded at runtime via `pluginlib` [5], [12]. DWB uses a finite set of candidate velocity trajectories with a cost-function critic [4]. MPPI samples many stochastic rollouts and averages them, weighted by an exponential of their cost [3]; our custom C++ MPPI is a direct port of the MPPI implementation from assignment A2, now wrapped as a Nav2 plugin.

Global planners: NavFn is a Dijkstra-style wavefront planner on the costmap; Smac 2D is a more modern A*-based planner that produces shorter, more geometrically direct paths but with a non-trivial runtime cost.

While LQR has been applied to path-tracking in prior work, existing Nav2 deployments rely predominantly on DWB or MPPI. This paper integrates LQR directly into the Nav2 plugin lifecycle and provides a systematic 3×2 benchmark comparing all three controllers under identical conditions, which, to our knowledge, has not been reported previously.

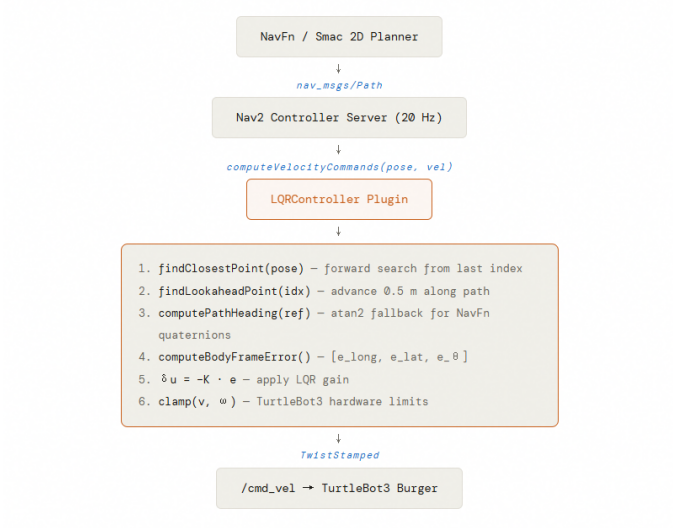

Fig. 1. System architecture. Nav2 calls the LQR plugin at 20 Hz; the plugin consumes the current pose + velocity and returns a `Twist`.

III. Algorithm & System Design

A. Architecture

The controller is loaded into the Nav2 controller server as a `pluginlib` plugin. Fig. 1 shows the dataflow from the global planner to the motor commands.

The C++ code is organized into two translation units.

- `lqr_solver.{hpp,cpp}` – pure math on Eigen, no ROS dependencies. Provides `buildLinearSystem()` (build A_d, B_d), `solveDARE()` (iterative Riccati), `computeGain()` (K), `computeBodyFrameError()`, and `normalizeAngle()`.
- `lqr_controller.{hpp,cpp}` – Nav2 plugin lifecycle. DARE is solved once inside `configure()`; `setPlan()` caches the global path; `computeVelocityCommands()` calls the search helpers (`findClosestPoint`, `findLookaheadPoint`) and returns the commanded `TwistStamped`.

B. Body-Frame Error Formulation

Rather than computing tracking error in the global frame, we project the displacement into the local frame of the reference point on the path. This body-frame projection makes the Q matrix directly interpretable, where q_{lat} penalizes cross-track error, q_{long} penalizes along-track error, and q_θ penalizes heading error.

Given the current pose (p_x, p_y, θ) and a reference $(p_{x,\text{ref}}, p_{y,\text{ref}}, \theta_{\text{ref}})$ taken from a look-ahead point on the path (default 0.5 m ahead of the closest point), let $\Delta x = p_x - p_{x,\text{ref}}$, $\Delta y = p_y - p_{y,\text{ref}}$. Then

$$e_{\text{long}} = \cos\theta_{\text{ref}}\,\Delta x + \sin\theta_{\text{ref}}\,\Delta y \tag{1}$$

$$e_{\text{lat}} = -\sin\theta_{\text{ref}}\,\Delta x + \cos\theta_{\text{ref}}\,\Delta y \tag{2}$$

$$e_\theta = \text{wrap}(\theta - \theta_{\text{ref}}) \tag{3}$$

and $\mathbf{e} = [e_{\text{long}}, e_{\text{lat}}, e_\theta]^\top \in \mathbb{R}^3$ is the state of the linearized tracking system.

C. Discrete Linearization & DARE

Linearizing the unicycle around $\mathbf{e} = \mathbf{0}$ with $v_{\text{ref}} = 0.2\,\text{m s}^{-1}$ and $\omega_{\text{ref}} = 0$, and applying forward Euler with $\Delta t = 0.05\,\text{s}$ (20 Hz), gives the constant system

$$A_d = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & v_{\text{ref}}\Delta t \\ 0 & 0 & 1 \end{bmatrix}, \qquad B_d = \begin{bmatrix} \Delta t & 0 \\ 0 & 0 \\ 0 & \Delta t \end{bmatrix}. \tag{4}$$

Because v_{ref} and ω_{ref} are constants, Eq. (4) is constant too and the DARE can be solved *once* at startup. Although the robot's instantaneous speed varies slightly during execution, the linearization error remains small for the low-speed regime of the TurtleBot3 Burger ($v \le 0.22\,\text{m s}^{-1}$), and the closed-loop gain K is reused across all cycles without recomputation. We use the fixed-point iteration

$$P_{k+1} = Q + A_d^\top P_k A_d - A_d^\top P_k B_d \left(R + B_d^\top P_k B_d\right)^{-1} B_d^\top P_k A_d, \tag{5}$$

with convergence criterion $\max|P_{k+1} - P_k| < 10^{-9}$, which typically converges in fewer than 100 iterations. The gain matrix is

$$K = \left(R + B_d^\top P B_d\right)^{-1} B_d^\top P A_d \;\in\; \mathbb{R}^{2\times 3}, \tag{6}$$

and the control law applied each cycle is

$$\boldsymbol{\delta u} = -K\mathbf{e}, \quad v = v_{\text{ref}} + \delta u_0, \quad \omega = \delta u_1, \tag{7}$$

clamped to TurtleBot3 Burger limits $v \in [0, 0.22]\,\text{m s}^{-1}$, $\omega \in [-2.84, 2.84]\,\text{rad s}^{-1}$.

The final cost matrices are the specification defaults

$$Q = \text{diag}(1.0, 3.0, 1.0), \qquad R = \text{diag}(1.0, 0.5). \tag{8}$$

The weighting follows standard LQR intuition: $q_{\text{lat}} = 3.0$ is elevated over $q_{\text{long}} = q_\theta = 1.0$ because lateral deviation dominates path-tracking error for a unicycle on a smooth path, while the lower angular control cost $r_\omega = 0.5 < r_v = 1.0$ reflects the TurtleBot3's higher angular limit ($2.84\,\text{rad s}^{-1}$ vs. $0.22\,\text{m s}^{-1}$). The A3-specification defaults produced stable tracking without retuning.

IV. Implementation

The plugin structure follows the standard Nav2 plugin creation walkthrough.[1]

Controller integration details were informed by an existing Nav2 controller tutorial.[2] The plugin integrates with

[1] "Create Custom Plugins for ROS2 Navigation," YouTube. [Online]. Available: https://www.youtube.com/watch?v=X128gB2lVF0

[2] "Create Custom Controller Integration with Nav2," YouTube. [Online]. Available: https://www.youtube.com/watch?v=OEkIWiLqCsk&start=445

the Nav2 stack via `pluginlib` and implements the `nav2_core::Controller` interface; the Nav2 Pure Pursuit tutorial [6] served as a reference plugin. The DARE fixed-point iteration is solved once at `configure()` time with `Eigen` [11], producing a 2×3 gain matrix that is cached for the lifetime of the controller. Each call to `computeVelocityCommands()` proceeds through the following steps.

1) Query `goal_checker->isGoalReached()`. If the goal has been reached, return zero velocity and short-circuit.
2) Run `findClosestPoint()` – a forward linear search starting from the index of the previous cycle, with a small backward tolerance to absorb localization jitter.
3) Run `findLookaheadPoint()` to advance $0.5\,\mathrm{m}$ along the path from the closest index.
4) Compute the reference heading θ_{ref} with `computePathHeading()`, which falls back to $\operatorname{atan2}(\Delta y, \Delta x)$ between adjacent waypoints when the pose quaternion is identity (as with NavFn output).
5) Compute the body-frame error $\mathbf{e}$.
6) If the remaining path length is below `goal_slowdown_radius`, linearly attenuate v_{ref} toward zero with a 5% floor so the robot eases into the goal tolerance instead of overshooting.
7) Apply $\boldsymbol{\delta u} = -K\mathbf{e}$, clamp to hardware limits, and publish as a `TwistStamped`.

Reference parameters such as `desired_linear_vel`, `goal_slowdown_radius`, and `lookahead_dist` are exposed through the standard Nav2 YAML parameter mechanism.

The implementation is publicly available and includes the full benchmark script and parameter configurations.[3]

V. Experiment / Testing

A. Setup

All runs use the same Gazebo world (`turtlebot3_world`), the starting point $(x = -2.0, y = -0.5)$, and the goal point $(x = 2.0, y = 0.5)$. Metrics are computed from rosbag recordings at 20 Hz odometry. The three controllers (LQR, DWB, custom MPPI) are crossed with two global planners (NavFn, Smac 2D) giving a 3×2 benchmark matrix, with three repeated runs per configuration.

An automated Python benchmark script was developed to ensure reproducibility and eliminate timing noise introduced by human intervention. The script initializes the robot node, launches Gazebo, RViz2, and Nav2 in a headless backend configuration, initiates `ros2 bag record`, dispatches the navigation goal via `nav2_simple_commander`, and terminates the recording upon goal completion. As the script operates entirely in the backend, it is platform-independent and compatible with all major operating systems.

[3]Source code repository: https://github.com/TommyOh0428/LQR-Controller

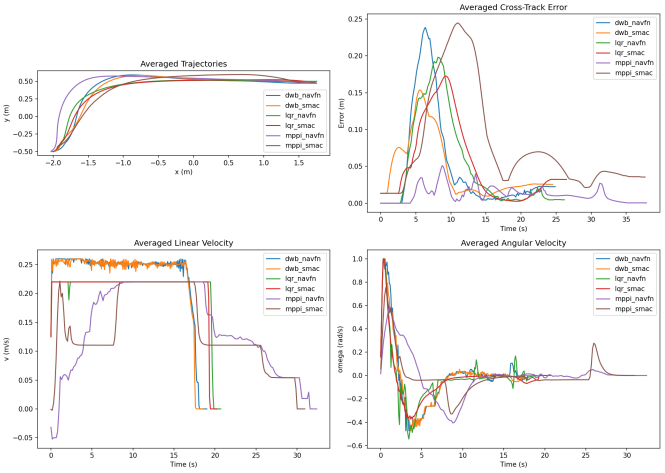

Fig. 2. Six-configuration benchmark comparison. **Top left:** Trajectory overlays showing all six controller–planner paths; MPPI+NavFn tracks closest to the reference while MPPI+Smac deviates most. **Top right:** CTE vs. time; MPPI+NavFn maintains the lowest error throughout while DWB exhibits the highest variance. **Bottom left:** Linear velocity profiles; MPPI produces the smoothest commands while DWB shows the most abrupt changes. **Bottom right:** Angular velocity profiles; LQR+Smac 2D requires fewer corrective turns than LQR+NavFn, consistent with Smac 2D's geometrically shorter paths.

B. Introduction of Metrics

Three metrics are used to evaluate performance:

- **Cross-Track Error (CTE)**: lateral deviation from the planned path, reported as mean, maximum, and standard deviation over each run.
- **Command smoothness**: mean absolute rate of change of linear velocity ($\mathrm{d}v/\mathrm{d}t$) and angular velocity ($\mathrm{d}\omega/\mathrm{d}t$); lower values indicate smoother motion.
- **Time-to-goal**: total elapsed time from goal dispatch to goal confirmation.

C. Per-metric Analysis

Table I summarizes the 3×2 matrix, with each cell averaged over three repeated runs per configuration. All 18 runs reached the goal.

Time-to-goal: DWB+Smac 2D is fastest at $22.9\,\mathrm{s}$, with DWB + NavFn ($23.9\,\mathrm{s}$) and both LQR configurations ($25.5\,\mathrm{s}$ each) within $2.6\,\mathrm{s}$. MPPI configurations are roughly $11\,\mathrm{s}$ to $15\,\mathrm{s}$ slower ($36.5\,\mathrm{s}$ to $37.7\,\mathrm{s}$), reflecting the per-cycle cost of evaluating many stochastic rollouts. Std devs of $0.2\,\mathrm{s}$ to $1.6\,\mathrm{s}$ indicate the per-configuration mean is stable across the three repeats.

Mean CTE: MPPI + NavFn is the outright winner at $0.012\,\mathrm{m}$ – roughly $4\times$ tighter than any other configuration – and is also the winner on max CTE and std CTE. DWB and LQR cluster tightly between 0.048 and $0.054\,\mathrm{m}$ and are effectively indistinguishable on this metric. MPPI+Smac, surprisingly, degrades to the worst mean CTE of the six ($0.081\,\mathrm{m}$), indicating MPPI is sensitive to the geometry of the planned path in a way LQR and DWB are not.

TABLE I

3×2 CONTROLLER $\times$ PLANNER BENCHMARK, MEAN $\pm$ STD OVER THREE REPEATED RUNS PER CONFIGURATION. BOLD ENTRIES INDICATE THE PER-COLUMN BEST VALUE (LOWEST CTE, LOWEST SMOOTHNESS RATE OF CHANGE, LOWEST TIME). ALL 18 RUNS REACHED THE GOAL.

Configuration	Mean CTE (m)	Max CTE (m)	Std CTE (m)	$\mathrm{d}v/\mathrm{d}t$	$\mathrm{d}\omega/\mathrm{d}t$	Time (s)	Goal
LQR + NavFn	0.051 ± 0.001	0.233 ± 0.000	0.066 ± 0.000	0.187 ± 0.001	0.946 ± 0.006	25.5 ± 0.6	3/3
LQR + Smac 2D	0.054 ± 0.002	0.205 ± 0.000	0.059 ± 0.001	0.183 ± 0.001	0.478 ± 0.010	25.5 ± 1.0	3/3
DWB + NavFn	0.049 ± 0.002	0.257 ± 0.000	0.077 ± 0.001	0.300 ± 0.025	1.158 ± 0.014	23.9 ± 0.8	3/3
DWB + Smac 2D	0.048 ± 0.003	0.199 ± 0.001	0.059 ± 0.002	0.311 ± 0.007	1.062 ± 0.017	$\mathbf{22.9 \pm 1.6}$	3/3
MPPI + NavFn	$\mathbf{0.012 \pm 0.001}$	$\mathbf{0.055 \pm 0.011}$	$\mathbf{0.011 \pm 0.001}$	$\mathbf{0.086 \pm 0.004}$	$\mathbf{0.169 \pm 0.009}$	37.7 ± 0.2	3/3
MPPI + Smac 2D	0.081 ± 0.002	0.285 ± 0.003	0.077 ± 0.001	0.097 ± 0.001	0.267 ± 0.003	36.5 ± 1.3	3/3

Smoothness: MPPI produces the smoothest linear and angular commands (0.086 and 0.169 respectively for MPPI + NavFn). LQR is second on both axes, and LQR + Smac's $\mathrm{d}\omega/\mathrm{d}t$ of 0.478 is roughly half that of LQR + NavFn (0.946), consistent with Smac 2D's shorter, straighter paths requiring fewer corrective turns. DWB has the highest rate of change on both velocity axes across both planners, matching its selection from a discrete trajectory set.

VI. DISCUSSION

LQR vs DWB: DWB and LQR track the path with comparable accuracy – both sit near $0.05\,\mathrm{m}$ mean CTE across planners – and finish within $2.6\,\mathrm{s}$ of each other. LQR's advantage is on command smoothness: $\mathrm{d}v/\mathrm{d}t$ is roughly $1.6\times$ lower than DWB and $\mathrm{d}\omega/\mathrm{d}t$ is 1.2–2.2$\times$ lower, reflecting LQR's analytic state-feedback law versus DWB's selection from a discrete trajectory set. LQR is also the cheaper controller per cycle, since its gain matrix is computed once at startup.

LQR vs MPPI: Our custom MPPI is the outright accuracy and smoothness winner when paired with NavFn: mean CTE is roughly $4\times$ tighter than any other configuration and both smoothness rates are the lowest of the six. The cost is wall-clock time – MPPI configurations are $40\,\%$ to $50\,\%$ slower than LQR and DWB – and per-cycle compute, since MPPI evaluates many stochastic rollouts at every control step. LQR sits at the opposite end of this trade-off: a single matrix multiply per cycle, predictable runtime, and accuracy close to DWB but not MPPI. MPPI + Smac 2D also exposes a brittleness of the sampling approach, degrading to the worst mean CTE of the benchmark ($0.081\,\mathrm{m}$) when given sharper planner geometry. For a deployment where raw tracking quality matters most, MPPI + NavFn is the strongest choice; for a deployment where per-cycle compute is a hard constraint, LQR is the cheapest option that still reaches the goal reliably.

NavFn vs Smac 2D: Planner choice has a small effect on time-to-goal – all four DWB and LQR rows fall within $2.6\,\mathrm{s}$ of each other. The more interesting effect is on angular-command smoothness: LQR + Smac has roughly half the $\mathrm{d}\omega/\mathrm{d}t$ of LQR + NavFn (0.478 vs. 0.946), which matches the intuition that Smac 2D's shorter, straighter paths demand fewer corrective turns. Smac 2D, however, hurts MPPI's tracking substantially, underscoring that planner–controller pairings can interact in non-obvious ways. A likely cause is that Smac 2D's sharper geometric turns produce waypoints with rapid heading changes, which increase the variance of MPPI's stochastic rollouts and bias the weighted average toward suboptimal trajectories.

VII. CONCLUSION

We implemented LQR as a Nav2 controller plugin for TurtleBot3 and benchmarked it against DWB and a custom C++ MPPI controller across two planners, with three repeated runs per configuration. With body-frame error projection, a once-at-startup DARE solve, and the original specification gains $Q = \mathrm{diag}(1, 3, 1)$, $R = \mathrm{diag}(1, 0.5)$, LQR reached the goal in every run, matched DWB on mean cross-track error (around $0.05\,\mathrm{m}$), and produced roughly $1.6\times$ smoother linear-velocity commands than DWB. Our custom MPPI is the outright accuracy and smoothness winner – mean CTE $0.012\,\mathrm{m}$ with NavFn and the lowest $\mathrm{d}v/\mathrm{d}t$ and $\mathrm{d}\omega/\mathrm{d}t$ of the three controllers – at the cost of a $40\,\%$ to $50\,\%$ longer time-to-goal and a substantially higher per-cycle compute load; DWB+Smac 2D was the fastest configuration at $22.9\,\mathrm{s}$. Classical LQR, correctly linearized in the body frame and plumbed through the Nav2 lifecycle, thus occupies a useful middle ground: competitive accuracy with DWB, smoother commands than DWB, and a fraction of MPPI's runtime cost.

Future directions include re-deploying on the physical TurtleBot3 Burger and adapting the reference linearization to vary with instantaneous v_{ref} such as recomputing K on a coarser cadence to handle time-varying speed profiles. The benchmark should also be extended to a broader collection of start/goal pairs, obstacle layouts, and path geometries – in particular, the MPPI+Smac 2D degradation we observed deserves investigation across a wider range of planner outputs.

REFERENCES

[1] S. Macenski, F. Martín, R. White, and J. Clavero, "The Marathon 2: A Navigation System," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Las Vegas, NV, USA, Oct. 2020, pp. 2718–2725.

[2] R. Tedrake, *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation*, MIT, 2023. [Online]. Available: https://underactuated.csail.mit.edu/lqr.html

[3] G. Williams, A. Aldrich, and E. Theodorou, "Model Predictive Path Integral Control: From Theory to Parallel Computation," *J. Guid. Control Dyn.*, vol. 40, no. 2, pp. 344–357, 2017.

[4] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered Costmaps for Context-Sensitive Navigation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Chicago, IL, USA, Sep. 2014, pp. 4321–4326.

[5] Nav2 Contributors, "Writing a New Controller Plugin," *Nav2 Documentation*. [Online]. Available: https://docs.nav2.org/plugin_tutorials/docs/writing_new_nav2controller_plugin.html

[6] Nav2 Contributors, “Nav2 Pure Pursuit Tutorial,” *Nav2 Documentation*. [Online]. Available: https://github.com/ros-navigation/navigation2_tutorials/tree/126902457c5c646b136569886d6325f070c1073d/nav2_pure_pursuit_controller
[7] Nav2 Contributors, “Nav2 Official Documentation,” *Nav2 Documentation*. [Online]. Available: https://docs.nav2.org/
[8] Nav2 Contributors, “Nav2 Controller Server Source,” *GitHub*. [Online]. Available: https://github.com/ros-navigation/navigation2/tree/humble/nav2_controller
[9] P. Abbeel, “Linear Quadratic Regulator (LQR),” *CS287: Advanced Robotics*, UC Berkeley. [Online]. Available: https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/
[10] ROBOTIS, “TurtleBot3 Burger e-Manual,” *ROBOTIS e-Manual*. [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/
[11] G. Guennebaud and B. Jacob, “Eigen3 Documentation,” *Eigen*. [Online]. Available: https://eigen.tuxfamily.org/dox/
[12] ROS Contributors, “Pluginlib ROS Tutorials,” *ROS Documentation*. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Pluginlib.html