LQR Trajectory Tracking Methods

Arba Shkreli, Elianne Sacher

September 8, 2022

Abstract

This report provides a detailed explanation of the theory and actual implementation of two methods of having a robot track a given trajectory in time, as well as a way by which to generate one just being given a sequence of ordered waypoints by the user.

Contents

1 Introduction

This report summarizes two different path tracking control projects that we have completed while working in the CoNG-Li lab. The goal of these projects was to have our robot, Turtlebot3, follow a given path reasonably without necessarily starting on it. The goal of the experiment, conducted both in simulation (simulated point object with Python and simulated robot with Gazebo), was to have the robot follow the path with minimal error. We intend for the models used and the data provided to be applicable for autonomous vehicles. The control objective of reaching the following the path was conducted with two different methods:

1. **LQR Trajectory Tracking.** In this method we linearize our delta system around a path of trajectories. The trajectory path contains both state references (coordinates and orientation) and control input references (linear and angular velocities). Our delta system is described by the state error, the difference between the state vectors and the reference state vectors, the control input error, the difference between the control input vectors, and the control input reference vectors. Upon linearization, we control the system by implementing an LQR controller on the system described.

2. **LQR Evolving Reference Point Tracking.** This second method breaks down the same path mentioned in the previous method into reference waypoints; however, this time we disregard the control input references. We gradually approach these waypoints, step by step, by performing LQR on the state error (same as described above), but instead of using the control input error we only use our control input vector. In this case we do not linearize the system. We use small enough time steps to consider the non-linear system as a linear system at each timestamp while calculating the time dependent control input matrix at each step.

Both of these methods were completed on a code generated state space model, on the ROS2 Gazebo platform, and on Turtlebot3s. In this report, we compare the advantages and disadvantages of each method with respect to robustness and noise.

NOTE: The term "trajectory" is an all-encompassing term in this report, which is taken to mean the desired poses *and* ideal control inputs to traverse the path determined, which assume that the vehicle actually starts out exactly on the intended path.

2 Theoretical Concepts Used

2.1 Relevant ROS2 Terminology

To implement our model on Tutlebot3 we used Robot Operating System (ROS). ROS2 is a set of software libraries and tools for building robot applications [Rob21]. In this report, we will be using some ROS2 terms, which include:

- ROS2 Graph: At the heart of any ROS2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate, which includes messages, topics, and discovery.

- ROS2 Node: A node is a participant in the ROS graph. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to Topics. Nodes can also provide or use Services and Actions. There are configurable Parameters associated with a node. Connections between nodes are established through a distributed discovery process. Nodes may be located in the same process, in different processes, or on different machines. These concepts will be described in more detail in the sections that follow.

- ROS2 Topics: ROS2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

- ROS2 Messages: ROS data type used when subscribing or publishing to a topic.

- ROS2 Discovery: The automatic process through which nodes determine how to talk to each other.

2.2 Spline Interpolation

Initially we wanted to see as a proof of concept whether the LQR controller worked on a simple reference trajectory of a straight line and circle. However, we would like for it to work on an arbitrary reference trajectory as well. That is, given a discrete set of (x, y) positions that are given in the order that it is wished for them to be covered, to generate a planned motion in time, which will be expressed as $\begin{bmatrix} x(t) & y(t) & \theta(t) \end{bmatrix}^T$. $x(t)$ and $y(t)$ will be derived first, from which we can compute $\theta(t) = \text{atan2}(\dot{y}, \dot{x})$. The idea was modified from what is partly discussed in [Jos16].

However, in order to fit an appropriate $x(t)$ and $y(t)$, we need to somehow assign a reasonable time by which each given waypoint is to be traversed, or rather a time that is realizable by the actuator involved. We chose to iterate through the ordered list of waypoints, and consider each pair of consecutive waypoints i and j. We compute a $\Delta t_{i,j}$ between the waypoints by assuming we know the maximum linear velocity that we allow the robot to have between them $v_{max_{i,j}}$. We can vary this velocity between different consecutive waypoint pairs, accounting for the possibility of changing circumstances such as a vehicle entering/exiting a busy road, which would result in a reduction/increase in maximum velocity. Therefore,

$$\Delta t_{i,j} = \frac{\text{Distance}\left((x_i, y_i), (x_j, y_j)\right)}{v_{max_{i,j}}} \tag{1}$$

and as a result we can embed a reasonable time by which each waypoint is reached.

It is possible that rather than a straight line connecting the two points, the fitted curve will be significantly longer, increasing the amount of time actually needed to reach the other next point than anticipated, creating a problem for the actuator to remain reasonably on schedule with its trajectory. We have noted so far the following strategies to mitigate this problem:

1. **Anticipating curvature.** Either programmatically or by the user's judgement, depending on the configuration of the points, curvature can be intuitively anticipated. Thus, when inputting the maximum velocity at which the vehicle can travel between two waypoints where a significant curve is expected, it is recommended to put not the greatest velocity that the vehicle can physically achieve, but instead something lower to account for the fact that because of the curvature, the distance covered will be greater than the straight-line one. Our demonstration of spline interpolated trajectories does not include an algorithmically-derived prediction of curvature between points, but we leave that as something to explore in future projects.

2. **Specifying waypoints that are closer together.** The closer waypoints are specified together, no matter what the overall trajectory's curvature is like, the distance between consecutive waypoints will be closer to the straight-line one. Thus, the reduction in maximum velocity between points can be either applied minimally or ignored.

Once reasonable times have been assigned to each waypoint, we can finally perform the parametric curve fitting. That is, we seek to fit an

$$\mathbf{x}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} a_0 + a_1 t + a_2 t^2 + a_3 t^3 \\ b_0 + b_1 t + b_2 t^2 + b_3 t^3 \end{bmatrix} \tag{2}$$

that is the closest possible to all of the given waypoints in the least-squares sense. The reference we used for least-squares is in [MR19]. So, for N waypoints, waypoint j is embedded with a time of t_j is $\mathbf{x_j} = (x_j, y_j)$, and we want to ask the following optimization question:

$$\min_{a_0,\ldots,a_3,b_0,\ldots,b_3} \sum_{j=0}^{N-1} \|\mathbf{x_j} - \mathbf{x}(t_j)\|^2 \tag{3}$$

If we define the column vector holding both the $x(t)$ and $y(t)$ coefficients $\alpha = \begin{bmatrix} a_0 & ... & a_3 & b_0 & ... & b_3 \end{bmatrix}^T$, then we would like to find the least-squares solution α^* (assuming it is unique)

$$\alpha^* = (A^T A)^{-1} A^T b \tag{4}$$

Where

$$A = \begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_1 & t_1^2 & t_1^3 \\ & & & & ... & & & \\ 1 & t_{N-1} & t_{N-1}^2 & t_{N-1}^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_{N-1} & t_{N-1}^2 & t_{N-1}^3 \end{bmatrix} \tag{5}$$

And

$$b = \begin{bmatrix} x_0 & y_0 & x_1 & y_1 & ... & x_{N-1} & y_{N-1} \end{bmatrix}^T \tag{6}$$

When writing the code, the problem was expressed in CVXPY, which recognizes this simple problem as by the package's provided example.

2.3 System Modeling

We model the system a a two-wheeled vehicle that remains parallel to the ground at all times, as there exists a small passive spherical wheel that balances it and prohibits it from having a pitch angle, and otherwise does not affect the motion of the vehicle [AM21a]. The dynamics can be modeled as follows:

$$\dot{s} = f(s, u) = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \omega \end{bmatrix} \tag{7}$$

Where the state vector is $s = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$, where x and y are the 2D Cartesian coordinates of the center of the axle between the two wheels and the yaw angle about the axis perpendicular to both the ground and the axle, and the control input vector is $u = \begin{bmatrix} v & \omega \end{bmatrix}^T$, where v is the linear and ω is the angular velocity.

The desired path for the robot will be described by a set of waypoints, which could be generated by sampling a continuous arbitrary function or by some other means. Then, we will make use of the spline interpolation method to plan the ideal motion of the robot through the given waypoints, which includes the ideal intermittent states and ideal control inputs. The method will produce functions $s^*(t)$ and $u^*(t)$ for all times $t = 0, 1, ..., N-1$ where N is large enough. This includes all of the states $s^*(t)$ and control inputs $u^*(t)$. For the LQR trajectory tracking method, we wish for the robot to be able to eventually align itself with this planned trajectory using LQR to generate the actual control inputs for the robot. On the other hand, for the LQR evolving reference point method, we wish for the robot to be able to align with only the planned state references path ($s^*(t)$) using LQR to generate the actual control inputs for the robot.

2.4 Linearization and Discretization

2.4.1 Trajectory Path Method

To use LQR, we linearize our system about the planned trajectory by using the error between the state vector, control vector, and reference vector (which includes the coordinates, the orientation, linear velocity, and angular velocity) [AM21b].

$$\text{Denote } \tilde{s} = s(t) - s^*(t) = \begin{bmatrix} x(t) - x^*(t) \\ y(t) - y^*(t) \\ \theta(t) - \theta^*(t) \end{bmatrix} \text{ and } \tilde{u} = u(t) - u^*(t) = \begin{bmatrix} v(t) - v^*(t) \\ \omega(t) - \omega^*(t) \end{bmatrix} \tag{8}$$

Taking the derivative of $\tilde{s}$ will give us:

$$\dot{\tilde{s}} = \dot{s}(t) - \dot{s^*}(t) = f(s(t), u(t)) - f(s^*(t), u^*(t)) = \begin{bmatrix} v(t)\cos(\theta(t)) \\ v(t)\sin(\theta(t)) \\ \omega(t) \end{bmatrix} - \begin{bmatrix} v^*(t)\cos(\theta^*(t)) \\ v^*(t)\sin(\theta^*(t)) \\ \omega^*(t) \end{bmatrix} \tag{9}$$

By a first-order Taylor expansion using the Jacobian we linearize the system around the reference:

$$f(s(t), u(t)) - f(s^*(t), u^*(t)) \approx f(s^*(t), u^*(t)) + \frac{\partial f}{\partial s}\Big|_{\substack{u=u^* \\ s=s^*}} \cdot \tilde{s} + \frac{\partial f}{\partial u}\Big|_{\substack{u=u^* \\ s=s^*}} \cdot \tilde{u} - f(s^*(t), u^*(t)) \tag{10}$$

Therefore

$$\dot{\tilde{s}} = f(s(t), u(t)) \approx A(s_t^*, u_t^*)\tilde{s}(t) + B(s_t^*, u_t^*)\tilde{u}(t) \tag{11}$$

Where

$$A(s_t^*, u_t^*) = \begin{bmatrix} 0 & 0 & -v^*(t)\sin(\theta^*(t)) \\ 0 & 0 & v^*(t)\cos(\theta^*(t)) \\ 0 & 0 & 0 \end{bmatrix}, \; B(s_t^*, u_t^*) = \begin{bmatrix} \cos(\theta^*(t)) & 0 \\ \sin(\theta^*(t)) & 0 \\ 0 & 1 \end{bmatrix}$$

In order to apply our continuous model in code, we discretize it around evenly spaced time steps:

$$\begin{aligned} \tilde{s}_{t+1} =& \tilde{s}_t + \Delta t \cdot \dot{\tilde{s}}_t \\ =& \tilde{s}_t + \Delta t \cdot \big(A(t)\tilde{s}(t) + B(t)\tilde{s}(t)\big) \\ =& \big(I + \Delta t A(s_t^*, u_t^*)\big)\tilde{s}_t + \Delta t B(s_t^*, u_t^*)\tilde{u}_t \\ =& \tilde{A}(s_t^*, u_t^*)\tilde{s}_t + \tilde{B}(s_t^*, u_t^*)\tilde{u}_t \end{aligned} \tag{12}$$

Where

$$\tilde{A}(s_t^*, u_t^*) = \begin{bmatrix} 1 & 0 & -\Delta t v^*(t)\sin(\theta^*(t)) \\ 0 & 1 & \Delta t v^*(t)\cos(\theta^*(t)) \\ 0 & 0 & 1 \end{bmatrix}, \; \tilde{B}(s_t^*, u_t^*) = \begin{bmatrix} \Delta t\cos(\theta^*(t)) & 0 \\ \Delta t\sin(\theta^*(t)) & 0 \\ 0 & \Delta t \end{bmatrix}$$

2.4.2 Evolving Reference Point Method

In the case of the evolving reference point method, we evaluate the controller of the our dynamic nonlinear system as a linear one. Since the time step we used is relatively small, we can calculate our time varying control input matrix, denoted as $\hat{B}$, at every time step, and for that time step use our discrete system as if it were linear. Our linearized system in this case, at each timestep, will be:

$$s(t) = \hat{A}s(t-1) + \hat{B}\big(s(t)\big)u(t) \tag{13}$$

Where

$$\hat{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; \hat{B}\big(s(t)\big) = \begin{bmatrix} \Delta t\cos(\theta^*(t)) & 0 \\ \Delta t\sin(\theta^*(t)) & 0 \\ 0 & \Delta t \end{bmatrix}$$

2.5 LQR

2.5.1 LQR Trajectory Tracking

As discussed, by the spline interpolation method we generate a set of ideal states and inputs that correspond to reasonably following given waypoints. Because the robot may not already start at this

point, and it may deviate from this planned motion, we use LQR to determine the actual control input to the robot in order to always tend to align with the desired path [SP08].

The performance index is defined as:

$$J(\tilde{s}_{t:N}, \tilde{u}_{t:N}) = \sum_{\tau=t}^{N} (\tilde{s}_\tau^T Q \tilde{s}_\tau + \tilde{u}_\tau^T R \tilde{u}_\tau) \tag{14}$$

Where

Q and R are our cost matrix which are fixed and predetermined, and $\tilde{s}_{t:N} = [\tilde{s}_t, \tilde{s}_{t+1}, ..., \tilde{s}_N]$, and analogously for $\tilde{u}_{t:N}$.

Our optimization question is:

$$\begin{aligned} \min_{\tilde{s}_{t:N}, \tilde{u}_{t:N}} J(\tilde{s}_{t:N}, \tilde{u}_{t:N}) = \min_{\tilde{s}_{t:N}, \tilde{u}_{t:N}} & \sum_{\tau=t}^{N} (\tilde{s}_\tau^T Q \tilde{s}_\tau + \tilde{u}_\tau^T R \tilde{u}_\tau) \\ \textbf{s.t.} \quad & (1)\ \tilde{s}_{\tau+1} = \tilde{A}(s_t^*, u_t^*)\tilde{s}_t + \tilde{B}(s_t^*, u_t^*)\tilde{u}_t \text{ for } t \le \tau \le N-1 \\ & (2)\ \tilde{s}_t = s_t - s_t^* \text{ and } \tilde{u}_t = u_t - u_t^* \end{aligned} \tag{15}$$

Another solution to this could be calculating the finite-horizon, discrete-time LQR by using the the Discrete-time Algebraic Riccati Equation (DARE):

$$\begin{aligned} P_{t-1} = & Q + \tilde{A}(s_t^*, u_t^*)^T P_t \tilde{A}(s_t^*, u_t^*) \\ & - \big(\tilde{A}(s_t^*, u_t^*)^T P_t \tilde{B}(s_t^*, u_t^*)\big)\big(R + \tilde{B}(s_t^*, u_t^*)^T P_t \tilde{B}(s_t^*, u_t^*)\big)^{-1}\big(\tilde{B}(s_t^*, u_t^*) P_t \tilde{A}(s_t^*, u_t^*)\big) \end{aligned} \tag{16}$$

Then, we can calculate the state-feedback gain matrix:

$$K_t = \big(R + \tilde{B}(s_t^*, u_t^*)^T P_{t+1} \tilde{B}(s_t^*, u_t^*)\big)^{-1} \tilde{B}(s_t^*, u_t^*)^T P_{t+1} \tilde{A}(s_t^*, u_t^*) \tag{17}$$

The control input vector for the delta system will then just be:

$$\tilde{u}(t) = -K_t \tilde{s}(t)$$

And in turn the control input vector for our system will be:

$$u(t) = \tilde{u}(t) + u^*(t) = -K_t \tilde{s}(t) + u^*(t)$$

2.5.2 LQR Evolving Reference Point Tracking

For this case, only will the ideal states from the spline interpolation will be considered. We will not have a reference control input to try to follow, but the LQR controller will only try to follow the reference poses. For fair comparison, the time steps and cost matrices will remain the same. The major difference when implementing the LQR will be the performance index, which will just be evaluated on the state error and the control input (rather than the control error) and the constraints (considering the different state and control input matrices) [SP08].

The performance index is defined as:

$$\hat{J}_i(s_t, s_i^*, u_t) = \sum_{t=0}^{\infty} ((s_i^* - s_t)^T Q (s_i^* - s_t) + u_t^T R u_t) \tag{18}$$

Where

s_i^* denotes the the i-th waypoint in the waypoint path and J_i denotes the performance index for approaching that point.

As denoted in the equation above, since we approach every waypoint in the path one waypoint at a time, both s_i^* and J_i will remain constant for each LQR implementation.

Our optimization question is:

$$\min_{s_{t:N}, s^*{}_{t:N}, u_{t:N}} \hat{J}_i\big(\tilde{s}_{t:N}, \tilde{u}_{t:N}\big) = \min_{s_{t:N}, s^*{}_{t:N}, u_{t:N}} \sum_{t=0}^{N} ((s_i^* - s_\tau)^T Q (s_i^* - s_\tau) + u_\tau^T R u_\tau) \quad \text{s.t. } s(t) = \hat{A} s(t-1) + \hat{B}\big(s(t)\big) u(t) \tag{21}$$

The optimal control sequence minimizing the performance index is given by:

$$u(t) = -K_t(s_i^* - s(t)), \text{ where } K_t = \Big(R + \hat{B}\big(s(t)\big)^T P_t \hat{B}\big(s(t)\big)\Big)^{-1} \Big(\hat{B}\big(s(t)\big)^T P_t \hat{A}\Big) \tag{22}$$

P is the unique positive definite solution of the discrete time algebraic Riccati equation (DARE):

$$P_{t-1} = Q + \hat{A}^T P_t \hat{A} - \big(\hat{A}^T P_t \hat{B}\big)\big(R + \hat{B}^T P_t \hat{B}\big)^{-1}\big(\hat{B} P_t \hat{A}\big) \tag{23}$$

3 Python Simulation

Before using ROS2 for either a more involved simulation or actual implementation on robots, we first wanted to test our controller with a Python simulation. This will enable us to easily add disturbances and check our model's robustness and sensitivity to noise.

3.1 CVXPY vs. DARE

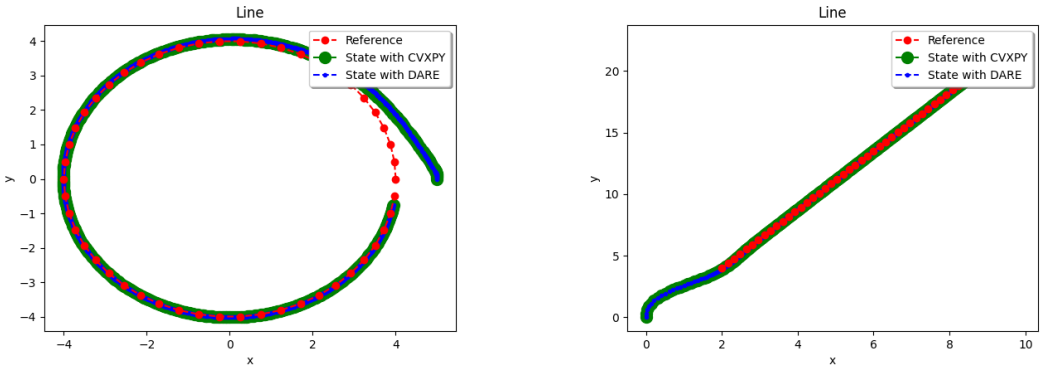
We came up with two different ways to implement in code our LQR trajectory function:

1. Addressing the LQR controller as an optimization question and solving it using CVXPY.
2. Solving the optimization question analytically by using DARE and recursively solving for the P matrix.

Both proved to be reasonable approaches for implementing the controllers, but how do they differ? We decided to test the two implementations firstly on simple paths: a line and a circle. We can specify either kind of trajectory by the number of waypoints along it that we wish, N. The line is defined by evenly spaced waypoints between a start and end point with ideal control inputs being a constant linear velocity through the points and zero angular velocity. The circle is specified by the desired center and radius, and the resulting trajectory starts from (assuming polar coordinate convention in radians) $\theta = 0$, with waypoints spaced out evenly θ-wise, and ends at $\theta - \frac{2\pi}{N}$. Later, we moved to testing the functional methods on a spline-interpolated path.

3.1.1 LQR Trajectory Tracking Implementation

After writing the two methods for generating the LQR trajectory tracking controller, we needed to simulate the movement of the object. We did so by letting it follow exactly the discrete system's dynamics (see equation 7) without any added disturbances. Both methods proved effective in terms of tracking the trajectories:



(a) Circle with 50 waypoints

(b) Line with 50 waypoints

Figure 1: Tracking Path

The code for producing these graphs can be found here.

In addition to testing generic trajectories such as the line and circle, we also wanted to test how well our spline-interpolated trajectories specified with a few discrete waypoints could be tracked by our LQR. We gave some arbitrary points, and wanted to see what kind of ideal trajectory was computed, and ultimately how well our LQR controller tracked it. To challenge the controller, we have the robot start at a point away from the path, but without modeling disturbances yet. The result can be seen in the following figure:

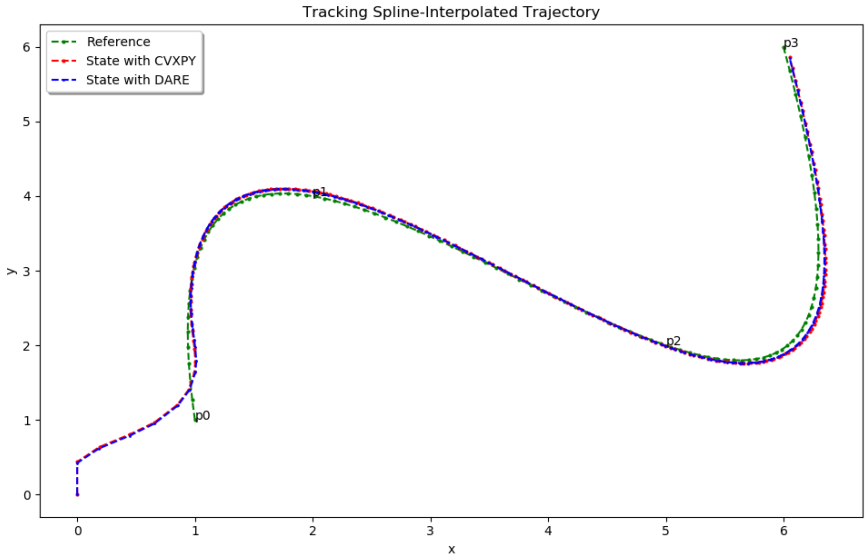


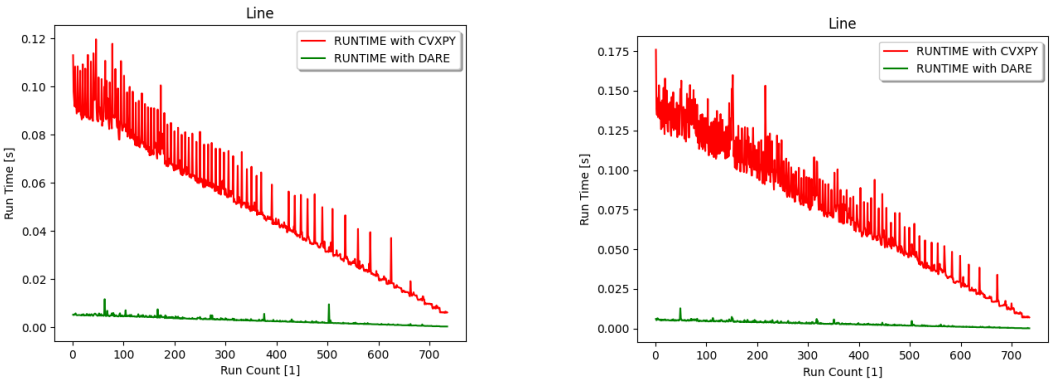
Figure 2: Tracking spline-interpolated trajectory discretized into 50 points through given waypoints $p_0 = (1, 1)$, $p_1 = (2, 4)$, $p_2 = (5, 2)$, $p_3 = (6, 6)$

The code for producing these graphs can be found here.

Once again we see minimal difference between what states CVXPY produces and what DARE produces, thus in real-life applications we would go with using the DARE solution due to significantly lower computational costs. Therefore, any ROS 2 implementation of the algorithm will use DARE rather than CVXPY.

For the given waypoints, we knew that the curve fitting should contain significant curvature between them, thus the additional information about the maximum velocities between each waypoint remained a fraction of the physical maximum expected from the vehicle, to render a realizable trajectory. As discussed, future work should go towards developing a formal method to infer curvature before parameterizing the trajectory. The goal should be generating ideal trajectory inputs that are always realizable by the actuator, while also making sure that resources are not left idle (that is, the robot should not be too slow in one segment when it could have been faster due to an overconservative estimate of maximum velocity allowed between any two waypoints). Another direction would be to study other ways to perform trajectory curve fitting.

Upon comparing the two method's ability to follow the path, we compared the run time of the two methods:



(a) Circle with 50 waypoints

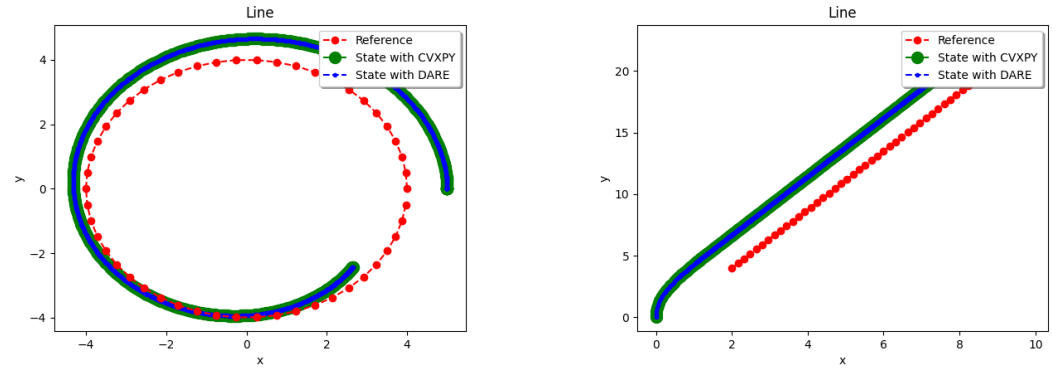
(b) Line with 50 waypoints

Figure 3: Run time

The code for producing these graphs can be found here.

3.1.2 LQR Evolving Reference Point Tracking Implementation

After writing the two methods for generating the LQR evolving reference point tracking controller, we needed to simulate the movement of the object. We did so by letting it follow exactly the discrete system's dynamics (see equation 7) without any added disturbances. Both methods proved effective in terms of tracking the trajectories:



(a) Circle with 50 waypoints

(b) Line with 50 waypoints

Figure 4: Tracking Path

The code for producing these graphs can be found here.

In addition to testing generic trajectories such as the line and circle, we also wanted to test how well our spline-interpolated trajectories specified with a few discrete waypoints could be tracked by our LQR. We gave some arbitrary points, and wanted to see what kind of ideal trajectory was computed, and ultimately how well our LQR controller tracked it. To challenge the controller, we have the robot start at a point away from the path, but without modeling disturbances yet. The result can be seen in the following figure:

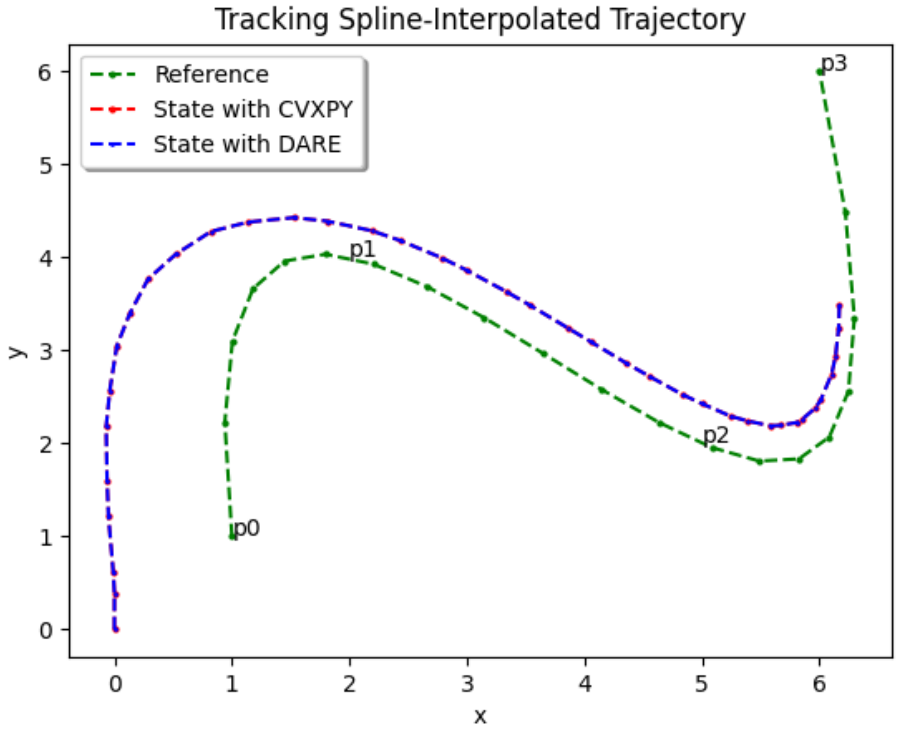
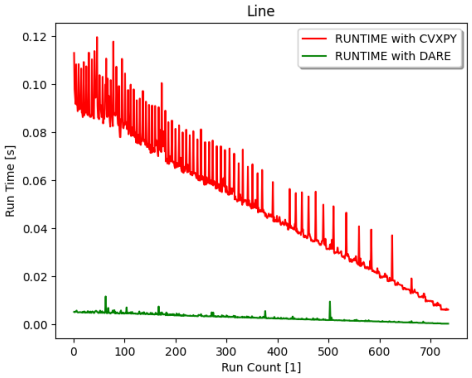


Figure 5: Tracking spline-interpolated trajectory discretized into 50 points through given waypoints $p_0 = (1, 1)$, $p_1 = (2, 4)$, $p_2 = (5, 2)$, $p_3 = (6, 6)$

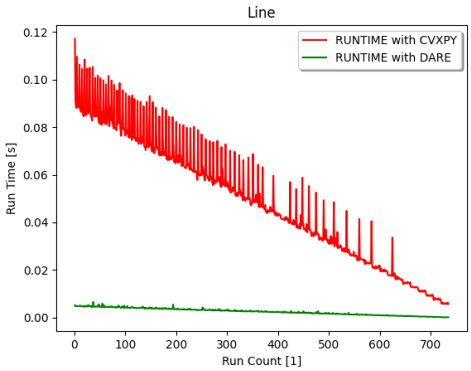
The code for producing these graphs can be found here.

Once again we see minimal difference between what states CVXPY produces and what DARE produces, thus in real-life applications we would go with using the DARE solution due to significantly lower computational costs. Therefore, any ROS 2 implementation of the algorithm will use DARE rather than CVXPY.

Upon comparing the two method's ability to follow the path, we compared the run time of the two methods:



(a) Circle with 50 waypoints



(b) Line with 50 waypoints

Figure 6: Run time

The code for producing these graphs can be found here.

3.1.3 Conclusion from CVXPY vs. DARE Comparison

This experimental comparison provided us with the following pros and cons list:

Pros for CVXPY over DARE	**Cons for CVXPY over DARE**
Modularity: One can easily add more conditions and generate more complex expressions as the code evolves.	Run time: CVXPY was one order of magnitude slower than the self generated code:

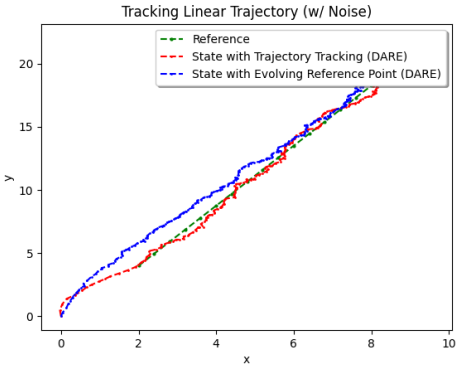
Since both methods for implementing the LQR function (CVXPY and DARE) proved to track the paths provided relatively accurately, we chose to use the DARE method for the ROS2 implementation. That is because the computation time when using CVXPY was too long. Additionally, from performing these experiment with both the LQR trajectory tracking method and the LQR evolving reference point tracking method was that the trajectory tracking method was superior in tracking the path. This was clear in all the different paths we plotted above.

3.2 Noise Simulation

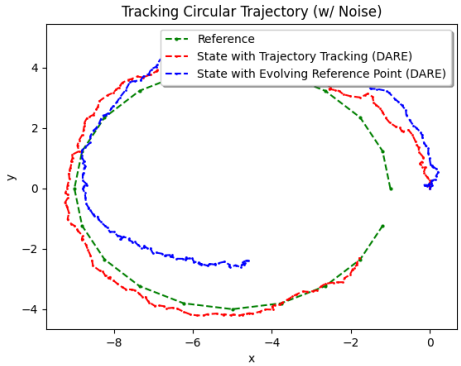
What remains next to do is test the controllers' robustness to system noise and disturbance. We tested the same three trajectories as before just with the DARE implementation, but we added Gaussian noise to the measurement of system state as well as actuator effort. To add noise to the state measurement, we took the current state and performed:

$$s_{t,noisy} = s_t + \mathbf{N} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \mathcal{N}(\mu = 0, \sigma = 0.05) \\ \mathcal{N}(\mu = 0, \sigma = 0.05) \\ \mathcal{N}(\mu = 0, \sigma = 0.005) \end{bmatrix} \tag{24}$$

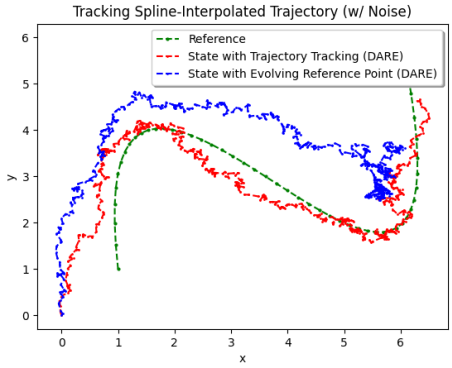
For the actuator effort noise, we added $\begin{bmatrix} \mathcal{N}(0, 0.05) & \mathcal{N}(0, 0.01) \end{bmatrix}^T$. The orientation noise we arbitrarily set to be smaller than the coordinate position noise, because we expect that the orientation would realistically not be provided by an external source, but is calculated locally. We added this noise for both the trajectory tracking method and the evolving reference point tracking method. To the input requested by the LQR controller, which calculated the control action using the noisy state measurements. The results are as follows:



(a) Line with 50 waypoints



(b) Circle with 50 waypoints



(c) Spline with 50 waypoints

Figure 7: Trajectories with Noise

The code for producing these graphs can be found here.

It is clear to see that under noisy conditions, the trajectory tracking method has once again proven to be superior.

The noise added is not informed by real conditions that vehicles like Turtlebots would experience. However, if we consider the positions to be given in the order of meters, a standard deviation of 0.05 for the noise would indicate that there is an uncertainty within 5 cm for each positional coordinate. We would like to test our robots under more realistic conditions, starting with the simulation environment Gazebo, and then proceed to using real Turtlebot3s making use of the OptiTrack motion capture system. We note that in the real implementation the noise would be mainly attributed to communication delay rather than an imprecise measurement from OptiTrack, though the extent to which this actually occurs will be discussed in Section 5.

The LQR Trajectory Tracking Method remains closer to the path even with noise, because it has the additional ideal control inputs to track as well as reference positions.

4 Gazebo Simulation

4.1 Discovering and Solving a Problem

The next stage of our controller's testing involved using a more realistic simulation platform, Gazebo. This allows to model the Turtlebot in its shape and inertial properties so that we find out whether the robot would actually move smoothly (without too much jerking nor tipping) from the calculated control inputs, insights we could not gain previously as the Python simulation essentially treated the robot as a point mass.

We encountered important issues in this stage, which solving provided crucial understanding of what subtleties to watch for once transferring a controller to a realistic setting. Initially the robot

appeared to accurately track the given path, but then would suddenly begin to inexplicably spin and never get on it again.

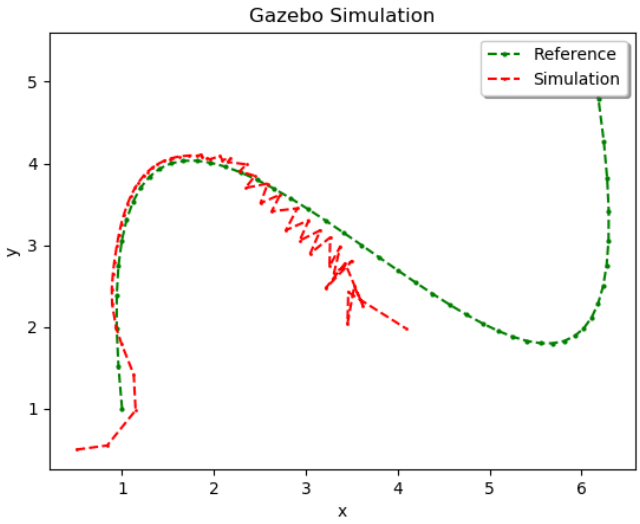


Figure 8: Problematic Spline-Interpolated Trajectory

This behavior was unexpected given that our Python simulation had gone quite smoothly, so we proceeded to investigate. We discovered two problems:

1. **Angular Error.** When calculating $\tilde{s}(t) = s(t) - s^*(t)$, we followed our formula without noting the subtleties that occur with the angular component. The formula conveys the *error*, which for orientation angles is not necessarily $\tilde{\theta}(t) = \theta(t) - \theta^*(t)$, as $\theta \in [0, 2\pi)$ in our case. This contributed to the spinning because suppose that the current angle is 1.9π and the reference angle is 0.1π. The way we implemented the controller would consider the error to be $\theta(t) - \theta^*(t) = 1.8\pi$, and the control action, which seeks to minimize the cost associated with angular error (among the other factors) over the remaining time steps, would not be to move counterclockwise (to go to 1.95π for example), because it falsely would believe that the error would increase. Therefore, it would take the action of going clockwise, which is extremely inefficient and noticeable in a vehicle with limited angular velocity. Thus, we were only able to notice this problematic behavior in the more realistic simulation.

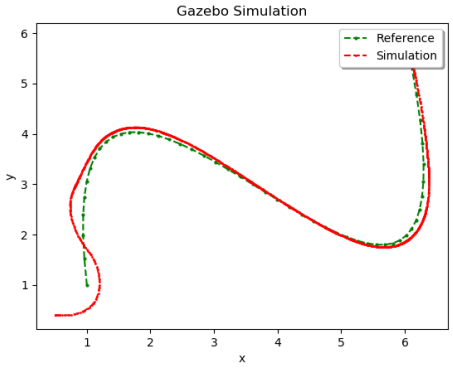
2. **Angle Normalization.** The way we kept all of the angles in our code within the range $[0, 2\pi)$ was not correct, and still generated angles outside that range.

To fix the first issue, we wrote a function that finds the smallest angle between the current and reference, which are both within the reinforced range $[0, 2\pi)$. If the absolute value of the difference between the angles is greater than π, then 2π is added to the smaller of the two, and then the difference that was expressed formulaically is carried out.

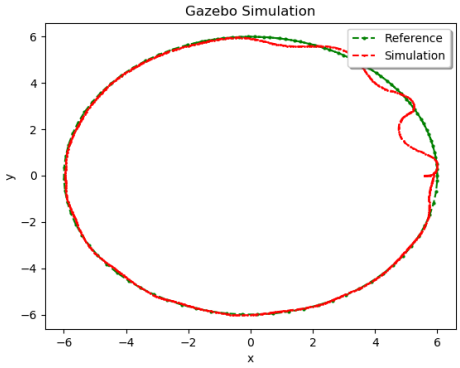
Fixing the second issue was done by rewriting our original angle normalization function, and making sure that all of the angles that we work with, whether they be reference angles or state angles, are normalized. For both fixes, the code can be found here as the `normalize` and `state_error` functions.

4.2 Results

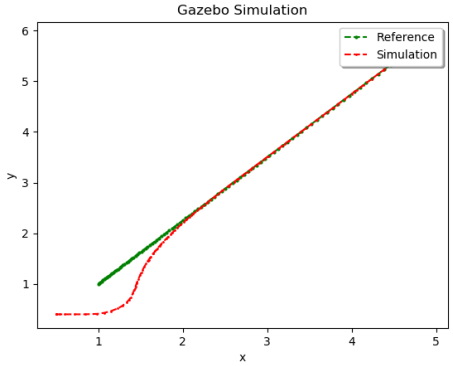
Once these changed were made, the tracking on Gazebo had a more reasonable and consistent character with that of the Python simulations. The trajectories differ in number of waypoints since the velocities are lower in the more realistic setting.



(a) Spline-Interpolated Path with N=50: *link to video*



(b) Circular Path with N=100: *link to video*



(c) Linear Trajectory with N=100: *link to video*

Figure 9: Fixed Trajectories

The code for producing these graphs can be found here.

The figures include links to videos showing the corresponding Gazebo simulation. The robot does not always move smoothly with constant speed as it preferably should, but it is important to consider the tradeoff between accuracy, smoothness, and speed. Our work was mainly focused on the trajectory tracking being accurate. We would like to explore for more realistic settings how to make the motion as smooth and accurate as possible, since these are crucial in the world of autonomous driving.

5 Turtlebot3 Implementation

Finally, though not finished, we began to experiment our controller with real Turtlebot3s. When going from simulation to actual robots, our trajectories and maximum velocities had to change to better suit real conditions. A real Turtlebot achieves much lower linear speeds than what its Gazebo or Python-based counterpart can.

We have managed to reproduce the trajectories on a real testbed as well, though it was not a seamless process. What we find problematic is that tuning the number of waypoints and the number of times the LQR would be calculated between each waypoint is a time-consuming and non-generalized way to deploy our controller on a real system. Therefore, work solely focused on transferring the implementation to hardware is needed. So far, we have the robot successfully follow a circle and line trajectory, but have incurred some technical difficulties when it comes to the curvier spline-based trajectory.

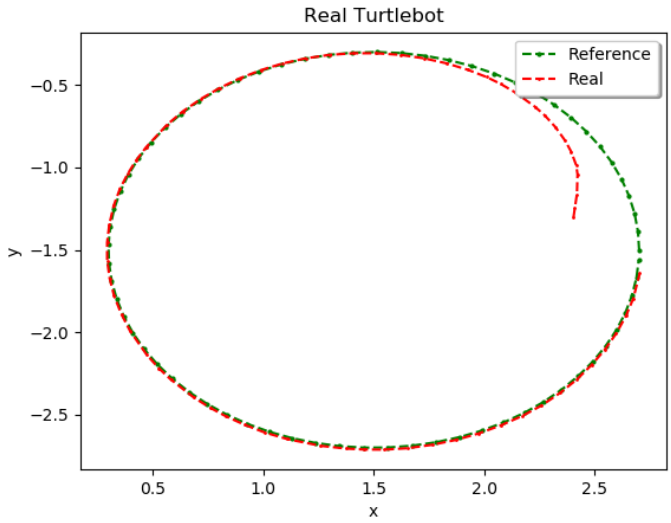


Figure 10: Lab Circular Trajectory *(link to video)*

The code for producing this graph can be found here.

6 Conclusion

In summary, our work consisted of trying two ways to track a given trajectory, as well as explore a method of generating a time-based trajectory through given waypoints that a robot is actually able to track. We determined that the LQR Trajectory Tracking method was more accurate with and without measurement noise, which we believe to arise from the fact that the method requires a more greatly specified trajectory. That is, in addition to the $\begin{bmatrix} x & y & \theta \end{bmatrix}^T$ at each discrete time step, ideal control inputs $\begin{bmatrix} v & \omega \end{bmatrix}^T$ are also tracked. Therefore, it becomes intuitively more difficult to veer from the path when the path is better defined.

Of course, if a more specified trajectory is needed for the positions to be more closely followed, then we need to be able to generate one. Our approach was to use spline interpolation through given waypoints with an assigned time by which they should be traversed in order to generate $s^*(t)$ and subsequently $u^*(t)$, of which an arbitrary number of discrete points can be taken for hardware implementation. The open question is how to assign a reasonable time to each user-given waypoint to perform this parametric curve fitting, without true knowledge of how long reaching each waypoint should take in the resulting trajectory. In other words, how to anticipate the curvature before performing the fit. We discussed some workarounds which so far have worked reasonably well, but we do not see them as being permanent or fully reliable solutions as the application settings become more serious.

Thus, we are left with many possible next points of study:

1. **Elaborate Turtlebot3 implementation.** Overall the process of having a real robot use our controller required tweaking of the number of waypoints to specify, as well as the number of times the actual control input would be calculated between each reference waypoint. We thus realize that hardware implementation of our controller needs to have a well-defined way to adapt to the physical limitations of the robot, which include speed limitations, frequency of receiving commands, etc, with little to no user intervention. The controller itself should incorporate physical velocity caps, for example, as the

2. **Make motion smoother while maintaining accuracy.** When the robot is significantly far away from the trajectory (typically in the beginning), it is useful to have more adjustment control inputs to get it on the path. However, when the robot is on the path within an acceptable margin, it is not useful to have as many control inputs. Therefore, we would like to explore

ways to adapt the number of control inputs over the course of the tracking, and more so how to do it with ROS2.

References

[AM21a] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2021. Chapter 2, http://www.cds.caltech.edu/~murray/amwiki/index.php?title=System_Modeling.

[AM21b] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2021. Chapter 5, http://www.cds.caltech.edu/~murray/amwiki/index.php?title=Linear_Systems.

[Jos16] Vrunda Joshi. Optimal trajectory generation for car-type mobile robot using spline interpolation. *ScienceDirect*, pages 601–606, 2016.

[MR19] Dan Margalit and Joseph Rabinoff. *Interactive Linear Algebra*. 2019. https://textbooks.math.gatech.edu/ila/least-squares.html.

[Rob21] Open Robotics. Ros2 concepts. 2021. https://docs.ros.org/en/foxy/Concepts.html.

[SP08] A.J Shaiju and Ian R. Petersen. Formulas for discrete time lqr, lqg, leqg and minimax lqg optimal control problems. 41, Issue 2:8773–8778, 2008.