Developing an LQR Controller for Differential Drive Robots

Ian Hogeboom-Burr

August 9, 2018

1 Introduction

In order to control a collection of differential drive robots, an LQR controller was designed using MATLAB. The control algorithm was built using a kinematic model of the system. This algorithm is capable of both trajectory and path tracking, and the controller gains are easily tunable. This controller is simulated using applications which plot the position of the robot in real-time.

2 Developing the Kinematic Model

Our goal is to develop a kinematic model which describes the motion of a two-wheeled differential drive robot. The robot has 3 degrees of freedom, and its position in the global coordinate system can be described as a 3-component vector, $\xi = [x, y, \theta]$, where θ is the angle of the robot's wheels relative to the global x-axis. We can also place a local coordinate system on the robot, with the unit directions X_f and X_p, where X_f points in the robot's forward direction (the direction in which the wheels are pointed), and X_p points in the direction perpendicular to the robot's wheels.

Since the robot has two wheels, and each of these wheels can be controlled independently, we have two potential inputs which determine the motion of the robot: ω_L and ω_R. These are the angular velocities at the left wheel and right wheel, respectively. Note: this model works for fixed standard wheels (wheels which can not rotate independently about their own axis). Take r to be the radius of the wheel, Q to be the point at the center of the chassis between the two wheels, and R to be distance between Q and each wheel. The instantaneous velocity at Q is a function of the angular velocities at each wheel, and can be written as follows:

$$V = \frac{1}{2} r(\omega_L + \omega_R)$$

The instantaneous velocity of Q must always point in the direction X_f, so the velocity of the robot in the forward direction is $V_f = V$. Because X_p is orthogonal to the direction of the robot's wheels, we have a nonholonomic constraint causing the velocity in the orthogonal direction to always be zero.

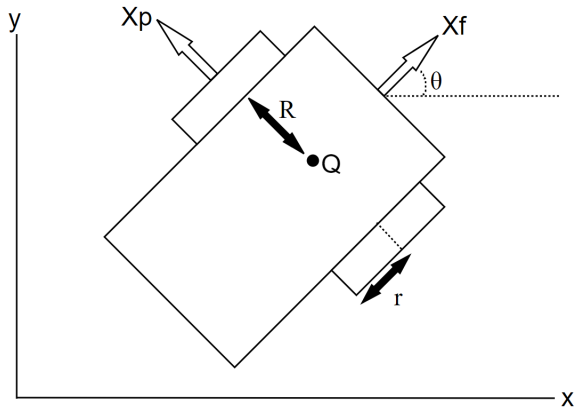

Figure 1: Model of the robot

To evaluate the rate of rotation of the robot, we first define positive rotation to be in the counter-clockwise direction. Forward rotation of the right wheel results in positive rotation, while negative rotation of the left wheel also results in positive rotation. We can describe the angular velocity at point Q as:

$$\omega = \frac{r}{2R}(\omega_R - \omega_L)$$

Looking at the geometry of the local and global coordinate systems, we can translate these equations of motion to the global reference frame by the following transformations:

$$\dot{x} = V\cos(\theta)$$
$$\dot{y} = V\sin(\theta)$$
$$\dot{\theta} = \omega$$

This system satisfies our conditions of motion in the X_f and X_p directions. Considering a time-step Ts and a zero-order hold on our system, we may develop a linear system in discrete-time:

$$x[k+1] = x[k] + V[k]\cos(\theta[k])Ts$$
$$y[k+1] = y[k] + V[k]\sin(\theta[k])Ts$$
$$\theta[k+1] = \theta[k] + \omega[k]Ts$$

We use these equations to predict changes in the position and orientation of the robot.

3 Linearizing the Kinematic Model

We start with the nonlinear discrete-time kinematic system, as developed in the previous section. Our goal is to approximate this system in a linear form, since linear equations are simple to work with in control applications. Putting our model into matrix form, we get:

$$q[k+1] = q[k] + Ts\begin{bmatrix}\cos(\theta[k]) & 0\\ \sin(\theta[k]) & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}V[k]\\ \omega[k]\end{bmatrix}$$

Where,

$$q[k] = \begin{bmatrix}x[k]\\ y[k]\\ \theta[k]\end{bmatrix}$$

To approximate, we use a variation of linearization relying on first order Taylor Series approximations. First, we take a Jacobian of the above system with respect to the position vector q along the desired path d, evaluated at time k. Let H be the right-hand side of the above equation. Let the matrix A be defined as:

$$A = \frac{\partial H}{\partial q} = \frac{\partial}{\partial q}\begin{bmatrix}x_d[k] + V_d[k]\cos(\theta_d[k])Ts\\ y_d[k] + V_d[k]\sin(\theta_d[k])Ts\\ \theta_d[k] + \omega_d[k]Ts\end{bmatrix}$$

$$A = \begin{bmatrix}1 & 0 & -V_d[k]\sin(\theta_d[k])Ts\\ 0 & 1 & V_d[k]\cos(\theta_d[k])Ts\\ 0 & 0 & 1\end{bmatrix}$$

This matrix will hold information regarding the initial conditions at time k, as well as derivative data for how position affects changing system over the time-step Ts. We must also have a matrix, B, which describes how our system changes with control inputs. We apply the same strategy by taking the Jacobian:

$$B = \frac{\partial H}{\partial u} = \begin{bmatrix}\cos(\theta_d[k])Ts & 0\\ \sin(\theta_d[k])Ts & 0\\ 0 & Ts\end{bmatrix}$$

Where our control inputs are,

$$u = \begin{bmatrix} V_d \\ \omega_d \end{bmatrix}$$

Our final linear approximation in discrete-time is:

$$q\,[k+1] = A[k]q\,[k] + B[k]u[k]$$

4 Developing the Control Algorithm

4.1 Trajectory Tracking

In the trajectory tracking controller, the robot attempts to follow a path parameterized in x and y. The robot's desired velocity and angular velocity at every time-step match those of the parametrized path, so that the robot can keep up with the moving path. Since a parameterized path does not explicitly contain an angle, the desired angle of the robot is defined as the angle of the velocity vector of the trajectory at each point, with the desired angular velocity being the discrete-time derivative of those points. The desired velocity of the robot is the norm of the parametric speed in both directions, pointing from the current point on the trajectory to the next point. The subscript d denotes a desired value.

To optimize the robot's path, the controller evaluates whether it is more efficient to travel forwards or backwards, which may alter the desired angle at any time-step by +/- π. For instance, if the robot was at (0, 0) pointing along the positive x-axis, and the desired velocity points to (-1, 0), it would get a desired angle of π, causing the robot to turn around. Clearly it is more efficient for the robot to drive backwards in this scenario, so the desired angle should be 0 not π, and the robot should receive negative velocity inputs. The function in the simulation that achieves this is the GetInput function, which determines whether to alter the desired angle and/or flip the direction of the desired velocity by evaluating if the desired angle is further than $\pi/2$ radians from the current angle. The built-in 4-quadrant inverse tangent function in MATLAB $atan2$ is used to compute the desired angle, as it bounds angles in [-π, π]. All angles in the control algorithm must be kept in this range to ensure accurate angle tracking.

Our controller will work to develop control inputs to steer the robot along the path in two manners. Firstly, it will give desired inputs that match the velocity and angular velocity of the parametric path. Secondly, it will consider the error in pose of the robot and correct this error using PD control, which will move the robot onto the path if it strays. Our error is:

$$e = \begin{bmatrix} x - x_d \\ y - y_d \\ \theta - \theta_d \end{bmatrix}$$

We must also define two matrices for use in our linear quadratic regulator: the Q matrix assigns weights to the importance of the x, y, and theta errors, while the R matrix assigns costs (i.e. penalties) to the use of velocity and angular velocity, respectively. Note: Both the Q and R matrices must be symmetric and have non-zero diagonal entries. In Q, the diagonal entries symbolize the weights of the three state errors: x, y, and θ. Non-diagonal entries symbolize weights for the products of errors (e.g. the weight in row 1, column 2 would be the weight of the error in x multiplied by the error in y). R functions in the same manner. For simplicity, these non-diagonal values can be ignored for this system.

The GetControl function uses these matrices to produce a gain matrix using MATLAB's built in LQR function. The LQR function relies on the linearization developed earlier, and defines the control input u as $u = -Kq$, minimizing the LQR quadratic cost function for the given parameters and using the solution to optimize pole placement for the system. This returns a 2 x 3 matrix with gain entries corresponding to proportional and derivative gains for each of the three states, respectively, effectively creating a PD controller for this system. This gain matrix, when multiplied by the error vector, will produce inputs that minimize the error between the robot's defined position and current position. These error inputs can then be summed with the desired inputs to produce the final inputs to the robot.

$$u = -Ke + \begin{bmatrix} V_d \\ \omega_d \end{bmatrix}$$

The algorithm then sends the signal u to the robot, steering it towards the desired trajectory.

To move on to the next time step, the algorithm needs to update the position of the robot at time k+1 (after the control input u was sent) so that it may restart the control procedure. This state estimation can be achieved in several ways. In the simulations, it is assumed that the robot's position is known exactly and the robot moves ideally. The updated position of the robot is found by inserting the control inputs u to the nonlinear discrete-time kinematic model of the system. In reality, the actual position of the robot can not be known to this degree of certainty - factors such as wheel slippage, measurement uncertainty, backlash, dynamics, and motor capabilities will affect how the robot's motion deviates from what the kinematic model predicts. In real-world scenarios, the robot's position at time k+1 would be estimated using localization methods, and then fed back into the controller.

It should be noted that if a constant point is fed to the trajectory tracker, the tracking algorithm will see the desired velocity as zero at all points, and apply a default constant desired velocity. The desired angular velocity in this situation will also be zero, and can be left as such.

Since the system becomes uncontrollable when V_d is zero, if a path with some zero velocity points is fed to the trajectory tracker these instances are replaced by $V_d = 0.00001$ to prohibit the algorithm from malfunctioning.

4.2 Path Tracking

The path tracking algorithm differs from the trajectory tracking method in a few distinct ways. The path tracking algorithm takes in a series of ordered points, but does not need to reach the points at a specific time. We define our desired velocity as a constant value that our robot will travel at, and set our desired angular velocity as zero. We define desired angle as the angle of a straight line from our current point to the desired point:

$$\theta_d = atan2(\frac{y_d - y}{x_d - x})$$

We also redefine our error vector as:

$$e = \begin{bmatrix} 0 \\ 0 \\ \theta - \theta_d \end{bmatrix}$$

We then continue with our controller as before, noting that the x and y weights in our Q matrix and the V cost in our R matrix are now irrelevant to our final input (but must be non-zero to prevent the LQR function from crashing). We then compute our final inputs as:

$$u = -Ke + \begin{bmatrix} V_d \\ 0 \end{bmatrix}$$

By altering the controller in this way we fundamentally change how our robot moves. The robot now moves forward at a constant speed, and does not use PD control to correct positional error in x and y. Instead, the robot only corrects its heading while driving, moving in a line from its current position to its desired position. The only LQR constants that matter now are the weight of the error in θ and the turning cost. By increasing the turning cost, the robot will move smoothly around curves. Decreasing the turning cost will cause the robot to turn sharply to face its desired point and then drive in a straight line. Altering the weight of the θ error will result in similar changes. The average of both wheel speeds will remain constant, so increasing one wheel's speed will decrease the other's speed by the same amount.

4.3 Implementation on Real Robots

So far, we have done all of our calculations using a unicycle model. On actual differential drive robots we do not control V and ω, we control the angular velocities at each wheel: ω_L and ω_R. We convert the unicycle inputs given by our controller to differential drive inputs using the following isomorphic map:

$$\omega_R = \frac{V}{r} + \frac{R\omega}{r}$$

$$\omega_L = \frac{V}{r} - \frac{R\omega}{r}$$

Furthermore, our robots can not take in continuous input values. The robot inputs must be integers in the range [-255, 255], where negative indicates backwards wheel rotation. The actual rotational speed in rad/s corresponding to these integer values varies based on battery charge, floor surface, and robot weight. Experiments confirmed that the map from integer wheel inputs to real rotational speeds is linear. To determine the specifics of this map, a calibration function was developed. The calibration function uses wheel encoders to estimate the rotational speed of each wheel at an input of 255 (ω_{255}), and then iterates through lower input values until it finds the maximum input which results in a rotational speed of zero (L). The linear map between control inputs and robot inputs is then:

$$input_{bot} = round(\omega_{controller}\frac{\omega_{255}}{255 - L} + sign(\omega_{controller})L)$$

Where *sign* takes the sign of its argument and returns +/- 1, and *round* is the standard integer rounding function. This map ensures that both positive and negative controller values get mapped to robot inputs that correspond closely to the true speeds. Values which map to an absolute value over 255 are capped at +/- 255, so it is important to tune the control algorithm to avoid saturating the controller. For the path tracking algorithm (the algorithm that is implemented on the real robots) saturation can be avoided by lowering the desired velocity and increasing the turning cost in the R matrix to avoid big input spikes from turning. For trajectory tracking, some trajectories will be impossible to track for the real robot due to saturation, as the desired inputs will exceed those possible by the robot; in these cases, attempt to reduce the frequency and amplitude of the parameterizations to reduce the desired inputs.

Testing found the slope value of the map to be approximately 13 and the intercept value to be approximately 100 when running with a full battery on the floor of the design lab. This means the maximum rotational speed of a wheel is 11.9 rad/s (robot input of 255), which corresponds to a maximum linear speed of 0.388 m/s for the robot. Any robot input under 100 will result in stalling. These were the values used for real robot mapping in the simulation.

5 Using the Simulation Apps

5.1 Trajectory Tracking App

The graphic interface has three plots and a multitude of adjustable controls. When plotting, the main graph shows the current position of the robot, the desired position of the robot, and the position history of both the desired and actual robot up to the current point in time. The 'plot path' button commences plotting, and the 'end' switch stops the plot if the user chooses to quit before completion. The 'time length' slider allows the user to control how long the simulation will run for and must be adjusted prior to plotting. One unit of length on the grid is equivalent to one meter. The upper right plot shows the (distance to the desired position of the robot over time. This is the norm of the error in x and y. The lower right plot shows the control inputs over time. The switch under the control inputs plot allows the user to alternate between viewing unicycle inputs (V and ω) and differential drive inputs (ω_L and ω_R). These options can be alternated between during plotting. All units are SI. The check box under this plot allows the user to execute the control algorithm using the physical constraints on the real-world robot, showing the inputs between -255 and 255 that the robot would receive. This box must be checked prior to pressing plot path as constraining the robot will affect its motion, although it can be unchecked during the visualization with no effect on path if the user wishes to view the control inputs in physical units. If this box is checked, the user should increase the LQR costs found in the top right numerical entry boxes to avoid saturating the controller (saturation occurs if control inputs are getting capped at 255 or -255). The velocity cost should be increased by several orders of magnitude for optimal results, while in most cases the turning cost does not need to be adjusted. The upper entry fields allow the user to alter different parameters affecting the trajectory of the robot. The X-Curve and Y-Curve drop downs allow the user to choose the trajectory parametrization in x and y. The amplitude entry fields allow the user to scale these curves by multiplying them by a constant. The frequency edit-fields allow the user to change the frequency of these curves by multiplying the time term by the number entered. To choose a constant point choose '1' as the curve in each coordinate, and scale using amplitude to place the x and y coordinates of the point (e.g. to choose (5, -2) choose X-Curve: 1, Y-Curve: 1, X-Amplitude: 5, Y-Amplitude: -2). Frequency doe not affect constant points. The initial position entry fields allow the user to enter the initial position of the robot. The angle is measured in radians and is relative to the positive x-axis. Ensure the angle is entered in the range [-π, π]. The LQR costs and weights allow the

user to tune the LQR controller. The default values are tuned for accurate movement of an ideal robot. The desired velocity sets V_d for the robot if it is moving to a constant point.

5.2 Path Tracking App

The path tracking app is very similar to the trajectory tracking app. Rather than choosing trajectories in x and y, the user chooses the number of points the robot must reach and enters their coordinates in order in the coordinate table. While some of the LQR specifications are irrelevant in this model, they have been left so that the user can see which ones affect motion and which ones do not.

Both apps use time steps of 0.05 seconds. The dimensions of the real robots are used in the simulation; the chassis width is 126 mm and the wheel diameter is 65 mm. In both apps, the robot has reached a static point if it is within 5 cm of the point. In the path tracking app, this means the robot will start moving to the next point in the sequence (and so the error will jump up). While both apps are designed to plot the robot's position in real-time, the complexity of running the three plots simultaneously often slows MATLAB and causes the plots to run slower than desired.

6 Testing

Testing shows that the trajectory tracking controller can handle difficult trajectories and minimize error well. The controller does encounter issues when mapping to real robot inputs, as many trajectories move too quickly for the controller to handle without becoming saturated (see Figure 3). For this reason it is not feasible to implement the trajectory tracking controller on the real robots.

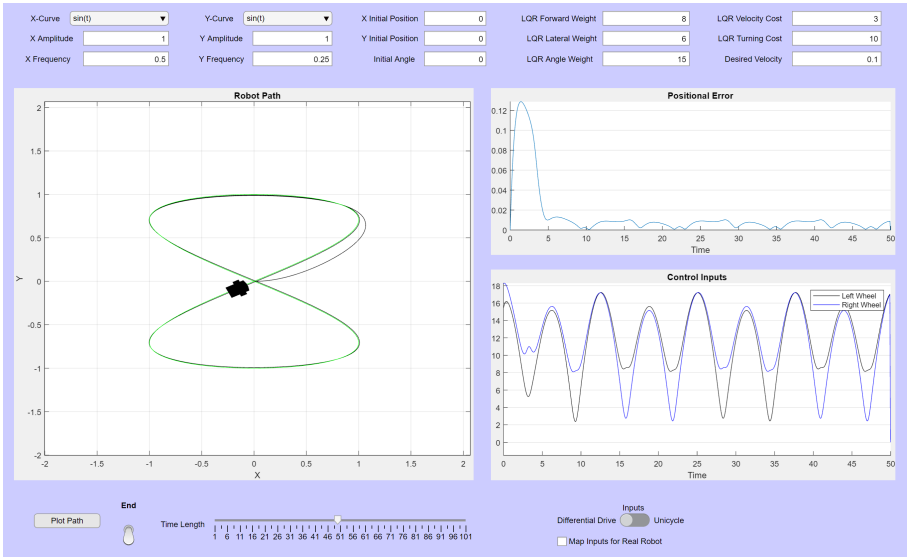

Figure 2: Trajectory tracking a figure-8 path. The robot deviates slightly at the start due to starting at the wrong angle, but quickly corrects itself and maintains a small positional error over the rest of the simulation.

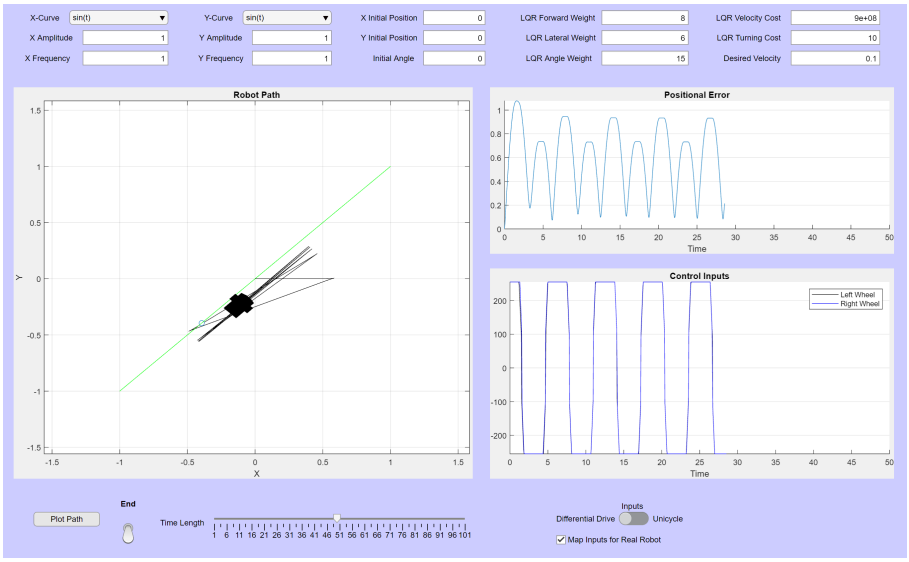

Figure 3: Trajectory tracking a forward-and-backward path using real robot constraints. The controller struggles with saturation due to limited linear speed.

Testing reveals that the path tracking controller accurately tracks a variety of curvilinear paths, and moves smoothly given a turning cost of 10. The robustness of this model shows its usefulness in application on the real robots. It can easily track any path, can be tuned to avoid saturation, and the user can specify a constant linear speed to travel at. Using a desired speed of 0.3 m/s generally results in unsaturated control, as this is well below the maximum linear speed of 0.388 m/s. It is possible that the controller could become saturated at this speed from certain paths if they contain many sharp turns that would cause the wheels to deviate greatly from their desired values. Additionally, when not constraining inputs to the real robots we find that desired speeds larger than 0.6 m/s result in poor tracking for a tolerance of 5 cm, as the robot overshoots points and may oscillate indefinitely without reaching them (see Figure 5). To overcome this, the robot requires smaller and smaller turning costs so that it can turn more sharply, resulting in jagged motion. This shows that, for the purposes of path-tracking, the physical constraints on the real robots' motors are not all that limiting, since speeds greater than 0.6 m/s can result in suboptimal movement.

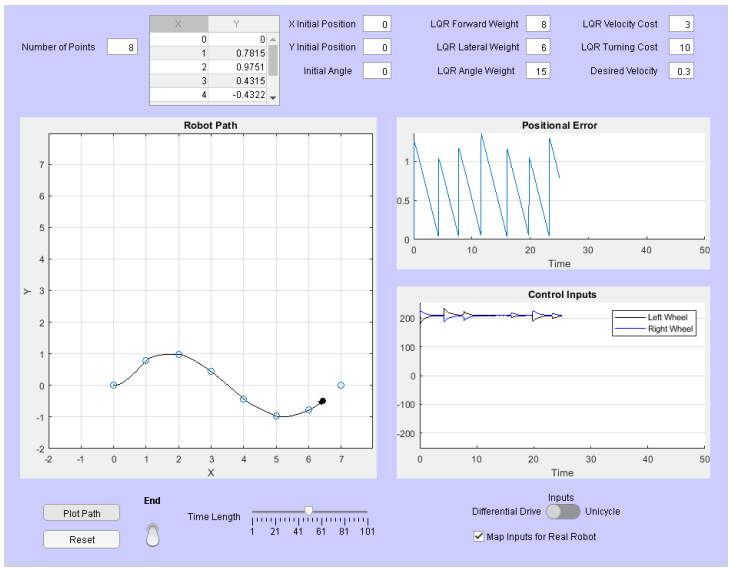

Figure 4: Path tracking a sinusoidal path using real robot constraints. The robot follows a curved path, and accurately tracks each point to within the 5 cm tolerance specified in the algorithm.

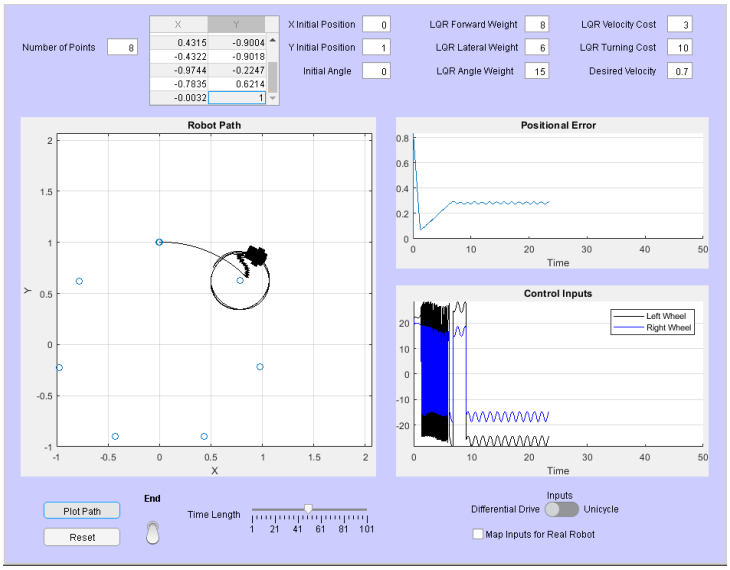

Figure 5: Path tracking a circular path without robot constraints. The robot overshoots the second point due to its high velocity and unreduced turning cost. It oscillates around the point without being able to reach it.

7 Conclusion

By evaluating the capabilities of the trajectory and path tracking algorithms using the simulations, it is clear that the LQR controller is efficient and robust in its tracking capabilities. The controller does have limitations and could encounter steady-state errors in certain applications due to the lack of integral feedback. The trajectory tracking controller also struggles with saturation when applied to the real robot, which is why trajectory tracking is not currently feasible to implement. Further research could be done into MPC control, although the computation required to execute MPC control on the robots is likely well beyond what is possible with the available equipment. Additionally, to more accurately model the robots, a dynamic model of the system could be created to replace the kinematic model. This could lead to better control in real environments.