Analysis of LQR and Linear MPC on Vehicle Trajectory Optimization

Nada Hussein
6.832
MIT
Cambridge, USA
nhussein@mit.edu

***Abstract*—In this paper, we focus on trajectory optimization in self-driving vehicles. We introduce implementations of LQR and linear MPC from scratch, and compare both methods on a wide array of trajectories in order to analyze best contexts for both. We successfully implement equivalent LQR and MPC methods, and compare them on straight, simple curve, and complex curve trajectories. We are able to achieve very similar performance in both, but find a clear advantage in MPC when we compare cross-track and heading errors along the trajectories. We also find that steeper curves in the trajectories lead to a decrease in fidelity in both methods.**

***Index Terms*—underactuated robotics, trajectory optimization, LQR, MPC**

I. Introduction

Autonomous vehicles are becoming more and more common on the roads. With these new vehicles, it's important to ensure that they are not only capable of generating accurate trajectories, but are also able to follow these trajectories optimally in order to prevent any unexpected error on the road. In this paper, we generate various types of cubic spline trajectories and test LQR and MPC on them, using steer angle and acceleration as input values. Both are able to provide a locally optimal trajectory given a locally linear model. We then compare both methods' performance and how they vary based on the situation.

II. Kinematic Model

Before implementing any trajectory optimization algorithms, we must design a kinematic model for a vehicle following a trajectory. We introduce 4 variables in the vehicle state vector:

$$[x, y, \psi, v] \tag{1}$$

where (x, y) represents our position, ψ represents the heading, and v represents the velocity. We can then represent our input vector with 2 variables:

$$[\delta, a] \tag{2}$$

where we define δ as our steering angle, and a as our acceleration. With these two vectors defined, we can then describe the system of equations to determine each next state:

$$x[n+1] = x[n] + v[n]\cos\psi[n]dt \tag{3}$$

$$y[n+1] = y[n] + v[n]\sin\psi[n]dt \tag{4}$$

$$\psi[n+1] = \psi[n] + \frac{v[n]}{L_f}\delta[n]dt \tag{5}$$

$$v[n+1] = v[n] + a[n]dt \tag{6}$$

where we define dt to be our discretized time step between states, and L_f to be the distance between the front of the car and its center of gravity. Here we use 0.1 sec and 0.5m, respectively. With this system of equations in place, we can then generate a state space model for use in both implementations of LQR and MPC.

III. Trajectory Optimization Algorithms

With our kinematic model in place, we now design our MPC and LQR algorithms.

A. Linear Quadratic Regulator (LQR)

We implement finite-horizon LQR using a similar process as seen in [1]. LQR assumes that our model is locally linear and time-varied. To implement LQR, we use the system of equations that determine the next state, as well as our errors to keep track of, $\bar{x}$:

$$\bar{x} = \begin{bmatrix} e_d \\ \frac{e_d - e_d^{prev}}{dt} \\ e_\psi \\ \frac{e_\psi - e_\psi^{prev}}{dt} \\ \Delta v \end{bmatrix} \tag{7}$$

I then decided to represent our state space model in terms of this error. With this, we can create the following state space model:

$$\bar{x}[n+1] = \begin{bmatrix} 1 & dt & 0 & 0 & 0 \\ 0 & 0 & v & 0 & 0 \\ 0 & 0 & 1 & dt & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \bar{x}[n] + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ v/L_f & 0 \\ 0 & dt \end{bmatrix} \begin{bmatrix} \delta[n] \\ a[n] \end{bmatrix} \tag{8}$$

With this state space model in place, we can then perform to find the best choices of steer angle and acceleration at each timestep.

To run through LQR, at every timestep, we find the nearest trajectory point to our ground-truth position with a simple distance metric so we can navigate to it. We can then use this

distance metric to find our cross track error, e_d. We also store our previous cross track error e_d^prev, to compute the discrete approximation of the derivative of cross track error. We follow a similar method to find our e_ψ and Δe_ψ, comparing the current heading with the heading of the track at the found closest index, and simply subtract our current and previous velocities for Δv.

From here, we take in our current state and create our new A and B matrices that are dependent on v[n]. Now that we have a state space model describing our error kinematics, we use a cost function to find an optimal gain K to find our optimal control. We define our cost function as follows:

$$J = \sum x^T Q x + u^T R u \tag{9}$$

We define Q and R as follows:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

This ensures an exactly even balance in cost consideration between performance (Q) and actuator effort (R). With this cost function in place, we can solve for the optimal control vector. We use the discrete-time Algebraic-Ricatti equation [2] for this, and solve iteratively over 150 iterations. We use the following to find our steady-state X, with initial X set as Q:

$$X_n = A^T X A - A^T X B (R + B^T X B)^{-1} B^T X A + Q \tag{11}$$

We iteratively solve this equation, setting X to X_n after each iteration, until we reach a difference between X and X_n of less than 0.01, and have converged to steady state. Once we have this, we can solve for K:

$$K = (B^T X B + R)^{-1} B^T X A \tag{12}$$

With this optimal gain, we then solve for our optimal control vector:

$$u^* = -K\bar{x} \tag{13}$$

From here we have our optimal steering angle and acceleration. We then update our state using the system of equations described in the kinematic model, and check whether this update leads us to be within a small threshold of the goal state, here defined as a 0.3m radius around the goal. If so, we exit out of the algorithm and have successfully completed the trajectory. Otherwise, we continue to perform LQR at each timestep until we reach the goal.

B. Model-Predictive Control (MPC)

We follow the same model as [1] to implement MPC, and follow the general structure found in [2].To implement MPC, we first need to define some error terms to include in our cost function. There are two important errors in this model: cross-track error, defined as e_{ct}, and heading error, defined as e_ψ. We define these errors as follows:

$$e_{ct}[n+1] = e_{ct}[n] + v[n]\sin(e_\psi[n])dt \tag{14}$$

$$e_\psi[n+1] = e_\psi[n] + \frac{v[n]}{L_f}\delta dt \tag{15}$$

We must then linearize our dynamic model such that we can represent it in state space form. We come up with the following derivatives:

$$\dot{x} = v\cos\psi \tag{16}$$

$$\dot{y} = v\sin\psi \tag{17}$$

$$\dot{v} = a \tag{18}$$

$$\dot{\psi} = \frac{v}{L}\tan\delta \tag{19}$$

Using these to linearize, we end up with the following state space model as seen in [1]:

$$\begin{bmatrix} x[n+1] \\ y[n+1] \\ \psi[n+1] \\ v[n+1] \end{bmatrix} = A \begin{bmatrix} x[n] \\ y[n] \\ \psi[n] \\ v[n] \end{bmatrix} + B \begin{bmatrix} \delta[n] \\ a[n] \end{bmatrix} + C \tag{20}$$

$$A = \begin{bmatrix} 1 & 0 & dt\cos\psi & -vdt\sin\psi \\ 0 & 1 & dt\sin\psi & vdt\cos\psi \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{dt}{L_f}\tan\delta & 1 \end{bmatrix} \tag{21}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ dt & 0 \\ 0 & \frac{vdt}{L_f\cos^2\delta} \end{bmatrix}, \tag{22}$$

$$C = \begin{bmatrix} vdt\psi\sin\psi \\ -vdt\psi\cos\psi \\ 0 \\ -vdt\frac{\delta}{L_f\cos^2\delta} \end{bmatrix} \tag{23}$$

To run MPC, at every time step we must find our reference look-ahead trajectory. I choose a look-ahead index value of 500. We then construct our reference trajectory by finding the closest neighbor to our current state, and returning the ground-truth trajectory from that index to 500 indices ahead. This will be the horizon we use to run MPC at each timestep. With this trajectory, we then go through 3 iterations of MPC to try to find an optimal trajectory along this horizon.

We then start with a list of accelerations and steering directions for this horizon that are all 0 for each timestep. We begin by predicting our trajectory in the future by simply updating our state for each acceleration and steering direction in the list. We then iterate on this acceleration and steering direction with linear MPC.

For every timestep in the horizon, MPC will first generate our A, B, and C matrices to represent our linearized model.

We use the CVXPY Python library for this [3]. We add a series of constraints to our problem:

$$\begin{bmatrix} x[n+1] \\ y[n+1] \\ \psi[n+1] \\ v[n+1] \end{bmatrix} = A \begin{bmatrix} x[n] \\ y[n] \\ \psi[n] \\ v[n] \end{bmatrix} + B \begin{bmatrix} \delta[n] \\ a[n] \end{bmatrix} + C \tag{24}$$

$$\begin{bmatrix} x[0] \\ y[0] \\ \psi[0] \\ v[0] \end{bmatrix} = \text{current state} \tag{25}$$

$$v[n] \leq \text{max speed} \forall n \tag{26}$$

$$v[n] \geq -\text{min speed} \forall n \tag{27}$$

$$|\delta[n]| \leq \text{max steer} \forall n \tag{28}$$

$$|a[n]| \leq \text{max acceleration} \forall n \tag{29}$$

We add these constraints to our CVXPY problem, and also add a cost representing the errors in equations 14 and 15. From here, CVXPY returns an optimal path of x, y, yaw, δ and acceleration values along the horizon we chose. We do this 3 times so we can try to converge on an optimal solution, and from here, we return our optimal trajectory over the horizon. We then take just the first step of this trajectory, update our state with it, and repeat for the next timestep until we have reached our goal.

IV. Results

Finally, we compare the performance of LQR and linear MPC on the same trajectories. We performed tests on a few different types of trajectories: straight paths, as well as two curved trajectories with different levels of curvature in order to compare both methods with respect to complexity in tracks.

A. Straight Paths

We use the following straight trajectory to test the simplest track for LQR and MPC:

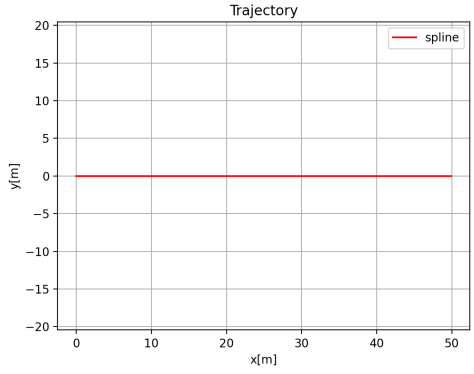

Fig. 1. Straight Trajectory

We see in figure 2 the trajectories that both methods came up with.

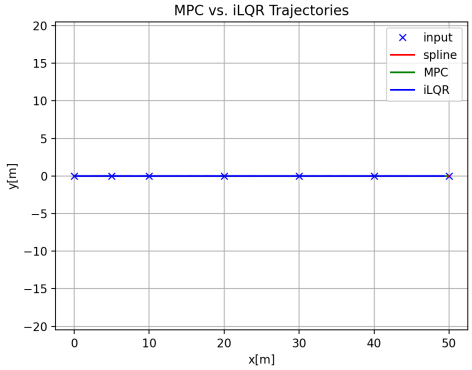

Fig. 2. Straight Trajectory MPC vs. LQR

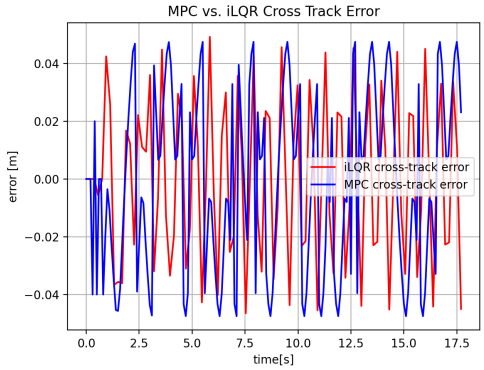

Fig. 3. Straight Trajectory Cross Track Error

Here we can see that on a straight trajectory, both methods work very well, to a point we are unable to distinguish them from the original line to follow. We can look closer at this by plotting both cross-track error and heading error over the trajectory, as seen in figures 3 and 4.

Here we see we end up with very small errors for both cross-track and angle error, and the error remains pretty consistent throughout since we do not introduce curves to the trajectory.

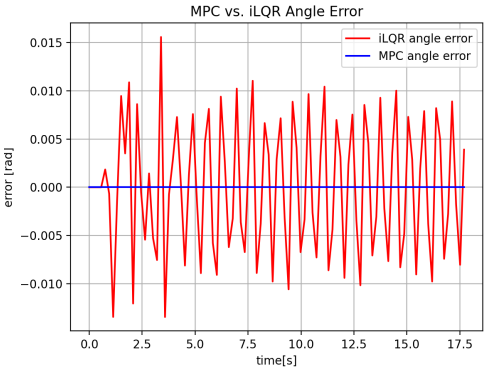

Fig. 4. Straight Trajectory Angle Error

B. Simple Curves

We then step up to a simple, wide curve as shown below:

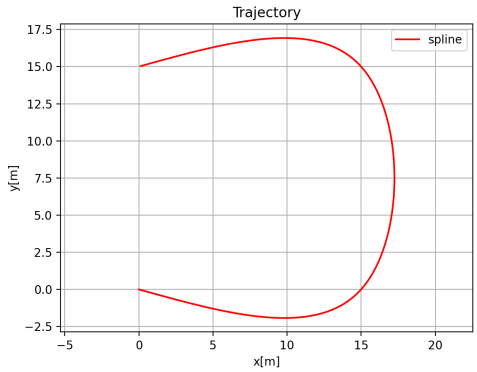

Fig. 5. Simple Curve Trajectory

We see in figure 6 the trajectories that both methods came up with:

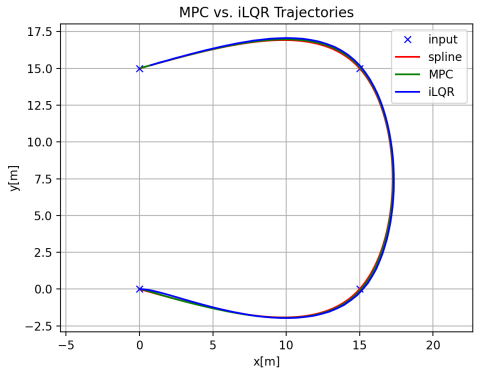

Fig. 6. Simple Curve Trajectory MPC vs. LQR

Here we can see that both methods perform very well, but we can see that the error increases at each curve. We can look closer at this in our error plots, seen in figures 7 and 8.

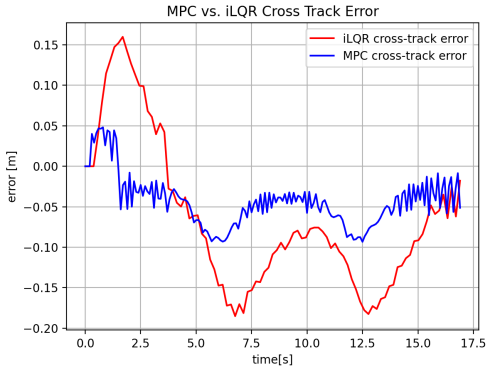

Fig. 7. Simple Curve Trajectory Cross Track Error

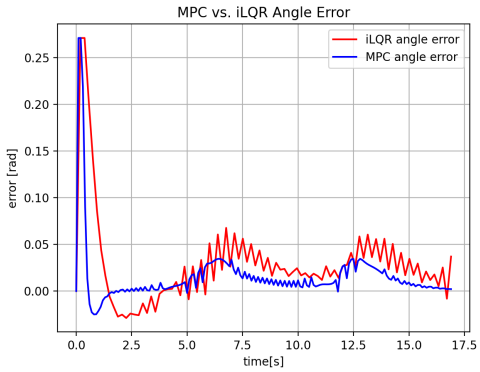

Fig. 8. Simple Curve Trajectory Angle Error

Here we can see that the errors in both cross-track and angle do increase as we approach the curves in the trajectory, but there is some difference between the two methods. We see that MPC performs much better, peaking at around 0.05m in cross-track error whereas LQR peaks at closer to 0.15m in cross track error. This trend continues in angle error as well, as we see MPC peak in angle error at around 0.03 rad while LQR peaks at 0.05 rad (discounting the initial error peak from starting the trajectory, which both methods see). Here we are able to tell that MPC is beginning to appear more robust to complex trajectories over LQR.

C. Complex Curves

Finally, we test a more complex spline, with multiple curves of various degrees of curvature:

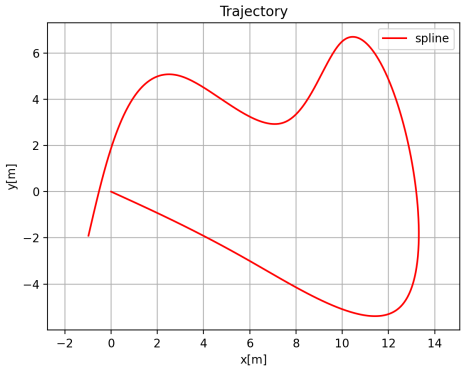

Fig. 9. Complex Curve Trajectory

We see in figure 10 the trajectories that both methods came up with.

Here we are able to see the biggest difference in performance between LQR and MPC. We can see that at each curve, LQR overshoots a bit more than MPC before correcting back to the trajectory again. We can analyze this deeper by looking at the error plots once more, seen in figures 11 and 12. Here we see that in both plots, LQR visibly overshoots each curve in the trajectory before correcting. For example in the curve

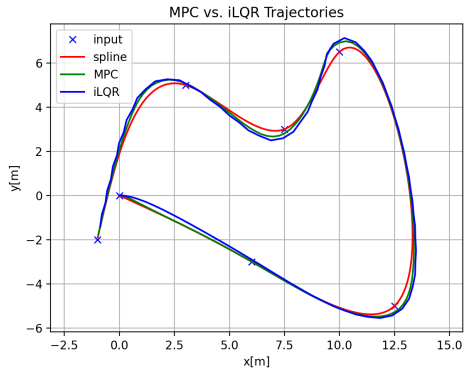

Fig. 10. Complex Curve Trajectory MPC vs. LQR

around (7.5, 3) on the trajectory, LQR overshoots by 0.4m, whereas MPC overshoots by only 0.2m. We also see this in angle error, where LQR has an error of 0.4rad while MPC only has a 0.2rad error.

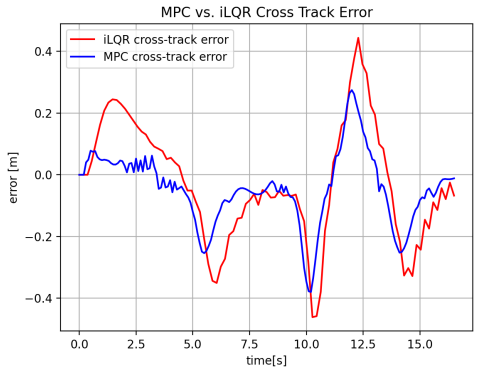

Fig. 11. Complex Curve Trajectory Cross Track Error

V. Discussion

Overall, LQR and MPC ended up having very similar results, as expected. However, there are some notable differences between the methods. We analyzed each method on three different trajectories: a straight path, a simple curved trajectory, and a complex curved trajectory. We can see in table I the total error for each method on each trajectory.

	LQR Errors		MPC Errors	
Track	Cross Track	Angle	Cross Track	Angle
Straight	2.2771 m	0.5588 rad	4.82915 m	0.0 rad
Simple Curve	9.21491 m	3.1543 rad	8.27762 m	2.8842 rad
Complex Curve	15.58711 m	8.1882 rad	14.25187 m	7.98946 rad

TABLE I
LQR vs MPC Errors

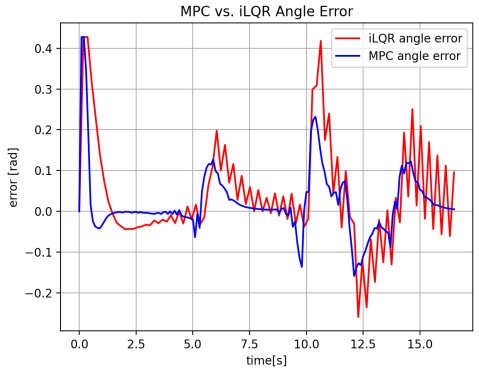

Fig. 12. Complex Curve Trajectory Angle Error

Here we can see that the error for both methods increases as we increase the complexity and curvature of the trajectories. This makes sense and is what we would expect, since we are deviating more from the current steering angle leading to some extra error until we have adjusted. We also see that LQR consistently performs slightly worse than MPC. We can see that our total cross-track error over the entire trajectory is consistently higher or comparable in LQR vs. MPC, and angle error is always higher in LQR. One reason for this may be the number of iterations on LQR is unable to converge well enough compared to MPC. It also might be due to the fact that MPC is able to take in constraints, whereas LQR simply operates on linear systems and cannot handle hard constraints. Finally, LQR comes up with the same optimal solution for the entire horizon, whereas MPC computes a new solution within a receding time window and therefore has more time to optimize more than LQR. All these difference can be explored in future work to make both methods more robust.

VI. Conclusion and Future Work

In conclusion, we were able to successfully implement LQR and MPC to perform trajectory optimization, and were able to get comparable results for both. However, we do find a clear difference in performance, where MPC performs slightly better than LQR across all trajectory types. There are many other algorithms to explore in this field tht are more robust, including Tube MPC, iterative LQR, as well as more constrained versions of both that are able to optimize much better than their base algorithms implemented here. In the future, we hope to implement more of these algorithms, as well as tune the current algorithms more rigorously, in order to develop a better class of algorithms for comparison purposes.

References

[1] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, Alexis Paques. (2018). PythonRobotics: a Python code collection of robotics algorithms.

[2] Russ Tedrake. Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832). Downloaded on 05/16/2021 from http://underactuated.mit.edu/

[3] Steven Diamond, Stephen Boyd (2016). CVXPY: A Python-embedded modeling language for convex optimization. Journal of Machine Learning Research, 17(83), 1–5.

[4] Y.Tassa,T.Erez,andE.Todorov,"Synthesisandstabilizationofcom- plex behaviors through online trajectory optimization," in Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on. IEEE, 2012, pp. 4906–4913.