Constrained iLQR for Motion Planning in Autonomous Vehicles

Karmesh Yadav and Prateek Parmeshwar

***Abstract*— In this project we implemented an iterative Linear Quadratic Regulator (iLQR) algorithm that incorporates constraints in the environment for on-road autonomous motion planning. Since iLQR is based on the theory of dynamic programming, it does not inherently take constraints like obstacles, actuator limits, etc into account. Therefore a Constrained Iterative Linear Quadratic Regulator [1] [2] (CILQR) algorithm is used, which solves constrained optimal control problem in nonlinear systems efficiently. The algorithm is then deployed in an autonomous driving simulator, which will also be used for validation of the project.**

I. Introduction

Trajectory planning for autonomous vehicles is a hard problem. There are a lot of challenges involved such as need of both spatial and temporal constraint in highly dynamic environment, incorporation of future trajectory predictions of other vehicles, non-linear vehicle model and non-convex constraints. Broadly, motion planning can be classified into search based methods, sampling based methods and optimization based methods. Search based methods provide globally optimal solutions however they suffer from the drawback of being inefficient during spatio-temporal planning. Sampling based methods are only probabilistically complete and are highly computationally inefficient when it comes to collision check.

Optimization based methods formulate the motion planning problem as a mathematical optimization and algorithms such as the Iterative Linear Quadratic Regulator(iLQR) have a slight advantage over search based and sampling based methods in that the solution is optimal at least in a local sense and it works real-time. In this project we implement the iLQR algorithm for trajectory planning. The algorithm incorporates costs and constraints on state and control inputs as well as obstacles in the environment given their future trajectories. We also develop a low-level simulator in Python to validate our results, followed by deployment on a high-fidelity simulator - CARLA.

II. Methodology

The concept behind the iterative Linear Quadratic Regulator(iLQR) algorithm draws from LQR itself. In LQR, at every time step the dynamics of the system are linearized, and the cost of the sequence is quadratized around the current point in state space. The feedback gain is computed off that, as though the dynamics were both linear and consistent (i.e. did not change in different states). However, the limitations of this algortithm are that it only optimizes at a point and hence assumes that the current approximation of the dynamics hold for all time. iLQR is an extension of this algorithm where instead we optimize over the entire control sequence.
The basic flow of the algorithm is as follows [3]:

1) Initialize with a random control sequence U_0
2) Roll out the dynamics and do a forward pass using the control sequence and get the trajectory that results from applying the control sequence U on initial state x_0
3) Do a backward pass and estimate the value function at each (x, u) in X, U
4) Calculate an updated control signal $\hat{U}$ and evaluate the cost of the entire sequence.
   - If the cost of the sequence with this updated control signal is *less* than the cost evaluated using the original control sequence *and* their difference is under a certain threshold then we say that the algorithm has converged and we have our optimal control output.
   - If the cost of the sequence with this updated control signal is *less* than the cost evaluated using the original control sequence then we update $U = \hat{U}$ and go back to step 2
   - If the cost of the sequence with this updated control signal is *more* than the cost evaluated using the original control sequence then increase the lambda parameter and go back to step 2.

A. Algorithm in Detail

Given the state x and the dynamics f[4], the state update equation is given by,

$$\mathbf{x}_{t+1} = \mathbf{f}\left(\mathbf{x}_t, \mathbf{u}_t\right)$$

The entire control sequence is given by U and X denotes the entire trajectory that results from applying the control sequence U. The total cost which is a function of the control sequence and the initial state is defined as the sum of the running cost plus the terminal cost.

$$J\left(\mathbf{x}_0, \mathbf{U}\right) = \sum_{t=0}^{N-1} \ell\left(\mathbf{x}_t, \mathbf{u}_t\right) + \ell_f\left(\mathbf{x}_N\right)$$

The cost to go from any time step t is defined as,

$$J_t\left(\mathbf{x}, \mathbf{U}_t\right) = \sum_{i=t}^{N-1} \ell\left(\mathbf{x}_i, \mathbf{u}_i\right) + \ell_f\left(\mathbf{x}_N\right)$$

where

$$\mathbf{U}_t = \{\mathbf{u}_t, \mathbf{u}_{t+1}, \ldots, \mathbf{U}_{N-1}\}$$

The value function given by $V_t(\mathbf{x})$ is found using the control sequence that minimizes J_t

$$V_t(\mathbf{x}) = \min_{\mathbf{U}_t} J_t\left(\mathbf{x}, \mathbf{U}_t\right)$$

Given that the value function at the final time step is equal to the terminal cost, value function at previous time steps can be found recursively,

$$V(\mathbf{x}) = \min_{\mathbf{u}}[\ell(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u}))]$$

As mentioned in the basic flow of the algorithm, there is a forward pass that calculates the states at every time step. In the backwards pass, the value function which is a function of state and control at that time step is slightly perturbed and a control signal update which minimizes this value function is calculated.

$$Q(\delta\mathbf{x}, \delta\mathbf{u}) = \ell(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}) + V(\mathbf{f}(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}))$$

In order to calculate the control signal update, derivatives of dynamics and cost w.r.t state and control are precomputed. These derivatives include l_x, l_{xx}, l_u, l_{uu}, l_{ux}, f_x and f_u. The terms in the second order expansion of the value function (Q) are given by,

$$Q_{\mathrm{x}} = \ell_{\mathrm{x}} + \mathrm{f}_{\mathrm{x}}^T V_{\mathrm{x}}'$$

$$Q_{\mathbf{u}} = \ell_{\mathbf{u}} + \mathbf{f}_{\mathbf{u}}^T V_{\mathbf{x}}'$$

$$Q_{\mathrm{xx}} = \ell_{\mathrm{xx}} + \mathrm{f}_{\mathrm{x}}^T V_{\mathrm{xx}}' \mathrm{f}_{\mathrm{x}} + V_{\mathrm{x}}' \cdot \mathrm{f}_{\mathrm{xx}}$$

$$Q_{\mathrm{ux}} = \ell_{\mathrm{ux}} + \mathrm{f}_{\mathrm{u}}^T V_{\mathrm{xx}}' \mathrm{f}_{\mathrm{x}} + V_{\mathrm{x}}' \cdot \mathrm{f}_{\mathrm{ux}}$$

$$Q_{\mathrm{uu}} = \ell_{\mathrm{uu}} + \mathrm{f}_{\mathrm{u}}^T V_{\mathrm{xx}}' \mathrm{f}_{\mathrm{u}} + V_{\mathrm{x}}' \cdot \mathrm{f}_{\mathrm{uu}}$$

here V' is the value function at the next time step. By minimizing this expansion of the Q-function, we get an optimal update in the control. This optimal update contains a feedback term $\mathbf{K}$ and a feed-forward term $\mathbf{k}$. The optimal update is given by,

$$\delta\mathbf{u}^*(\delta\mathbf{x}) = \min_{\delta\mathbf{u}} Q(\delta\mathbf{x}, \delta\mathbf{u}) = \mathbf{k} + \mathbf{K}\delta\mathbf{x}$$

where $\mathbf{k} = -Q_{\mathrm{uu}}^{-1} Q_{\mathrm{u}}$ and $\mathbf{K} = -Q_{\mathrm{uu}}^{-1} Q_{\mathrm{ux}}$ The control update is then plugged back into the second order expansion of the Q-function to get a quadratic approximation of V and the first and second order derivatives are given by,

$$V_{\mathrm{x}} = Q_{\mathrm{x}} - \mathrm{K}^T Q_{\mathrm{uu}} \mathrm{k}$$

$$V_{\mathrm{xx}} = Q_{\mathrm{xx}} - \mathrm{K}^T Q_{\mathrm{uu}} \mathrm{K}$$

As the value function at the terminal time step is known, the optimal value function at the previous time steps are calculated recursively and an update control signal is calculated. Again, as mentioned in the basic flow, the cost of this new sequence is calculated and compared with the previous cost and the steps are repeated till convergence.

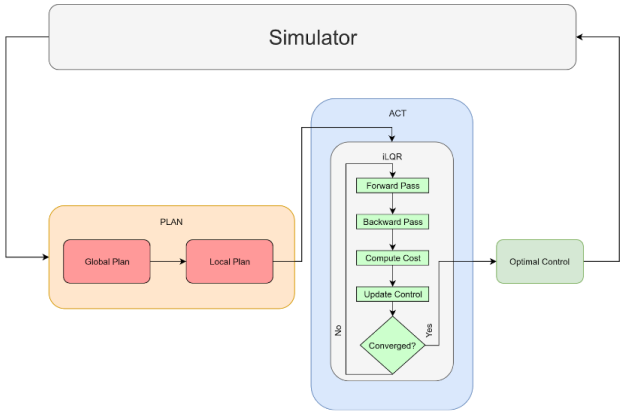

Fig. 1. Overall Architecture

B. Overview of Approach

The overall architecture comprises of the simulator subsystem, the planning subsystem and the controls subsystem. The simulator subsystem provides non-player character (NPC) information as well as a global plan in the form of waypoints. These waypoints are fed into the planning subsystem which fits a 3rd polynomial over a certain horizon to generate a local plan. This local plan serves as the desired trajectory for the controller to track. The controller subsystem then uses the iLQR algorithm and generates an optimal control sequence which is then fed back to the simulator.

C. Vehicle Model

The vehicle model used in this work is a kinematic bicycle model. The state of the vehicle is given by,

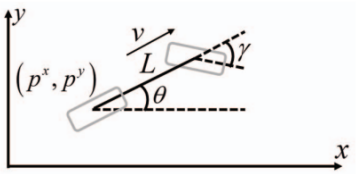

Fig. 2. Vehicle Kinematic Model

$$X = [x\ y\ v\ \theta]$$

where x, y is the $2D$ position of the vehicle, v is the vehicle speed and θ is the vehicle's yaw. The vehicle dynamics are given by,

$$\dot{x} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{v} \\ \dot{\theta} \end{bmatrix}$$

As we are dealing in discrete time domain the state update equation becomes,

$$X_{k+1} = \begin{bmatrix} x_{k+1} & y_{k+1} & v_{k+1} & \theta_{k+1} \end{bmatrix}^T = f\left(x_k, u_k\right)$$

where,

$$x_{k+1} = x_k + \cos\theta_k \left(v_k T_s + \frac{\dot{v}_k}{2} T_s^2\right)$$
$$y_{k+1} = y_k + \sin\theta_k \left(v_k T_s + \frac{\dot{v}_k}{2} T_s^2\right)$$

and

$$v_{k+1} = v_k + \dot{v}_k T_s$$
$$\theta_{k+1} = \theta_k + \dot{\theta}_k T_s$$

D. Cost

The cost function comprises of costs or penalties in state and controls, which are as follows:

- Acceleration
- Yaw Rate
- Path Following
- Desired Velocity Tracking

E. Constraints

The following constraints were added to the algorithm in the form of costs:

- Low and High Acceleration Bounds
- Low and High Yaw Rate Bounds
- Obstacle Constraints

Obstacles are modeled as ellipses where the axes are a function of the vehicle state, its dimensions, vehicle speed (for the major axis) and a safety margin. The ego-vehicle

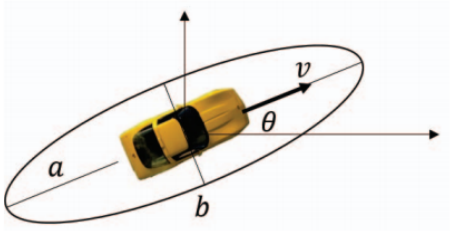

Fig. 3. NPC vehicle modeled as ellipse

itself is modeled as two circles. A constrained iLQR uses the

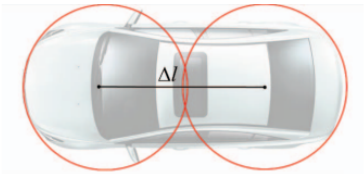

Fig. 4. Ego-vehicle modeled as two circles

same framework as the iLQR but in the backwards pass it transforms the cost and constraints into quadratic cost terms. If the cost function is not in quadratic form, then it needs to be quadratized around a point. The quadratization process is to compute the second-order Taylor series approximation of the function. If the constraint function is not linear it is linear it needs to be linearized. The linearization process is to replace the constraint function with its first order Taylor series expansion. The linearized constraint function f_x^k will be shaped by a barrier function,

$$b_k^x = q_1 \exp\left(q_2 f_k^x\right)$$

where $q1$ and $q2$ are tuning parameters. Finally this function new constraint function is again quadratized using a second-order Taylor series expansion.

III. Results

To test the working of the algorithm with the constraints, we created a scenario in our PySimulator. The ego-vehicle standing behind a Non-Player-Character (NPC) has to follow a predefined global trajectory while the NPC also is following the same trajectory. We see different behaviour in the simulator based on the relative costs assigned to the different terms in the algorithm.

Case 1: Car Following In the first case higher weightage is given to following the given global path and lower weightage is given to tracking the desired speed. Therefore the best solution according to the algorithm is to wait for the NPC to go ahead and then accelerate and reach the desired path.

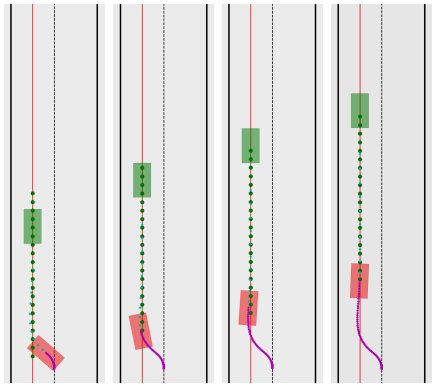

Fig. 5. Ego-vehicle following the NPC vehicle

Case 2: Overtaking Instead, if higher weightage is given to the velocity tracking error in comparison to the path following error, the ego-vehicle tries to overtake the NPC by accelerating and reaching the desired path.

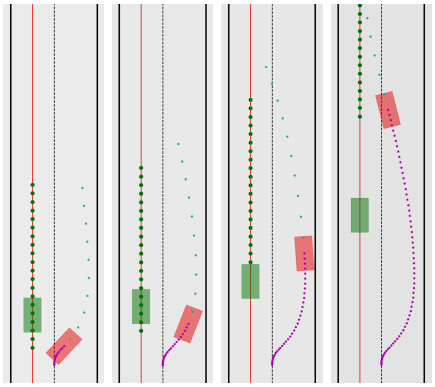

Fig. 6. Ego-vehicle overtaking the NPC vehicle

IV. Conclusion

In this project, we worked on applying the constrained iLQR algorithm for autonomous vehicle trajectory generation. Constraints on the states and controls are added within

the iLQR framework to let iLQR figure out the best control strategy. The constraints are linearized, shaped by barrier function, and then quadratized to be placed into the cost function of ILQR. To show the working of the algorithm, few different scenarios are tested out in the simulator. The behaviour of the vehicle changes based on the relative costs given.

ACKNOWLEDGMENT

We will like to thank Prof. Matt Travers and Saumya Saxena for the helping us decide the project. We will like to thank Yanjun Pan for the helpful discussions regarding the implementation.

REFERENCES

[1] Jianyu Chen, Wei Zhan, and Masayoshi Tomizuka. Constrained iterative lqr for on-road autonomous driving motion planning. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–7. IEEE, 2017.

[2] Jianyu Chen, Wei Zhan, and Masayoshi Tomizuka. Autonomous driving motion planning with constrained iterative lqr. *IEEE Transactions on Intelligent Vehicles*, 4(2):244–254, 2019.

[3] Blog on ILQR algorithm by Dr.Travis DeWolf at University of Waterloo, Canada. `https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/`. Accessed: December 14, 2019.

[4] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.