Learning Model Predictive Control for Drone Racing

Thomas Fork, Ziang Deng, and Lucas Moreira Medino

***Abstract*— We present an iterative approach for racing a drone using Learning Model Predictive Control (LMPC). We formulate drone racing as receeding horizon control with a time-varying convex quadratic program by linearizing the general, nonlinear problem. We demonstrate that our control problem can minimize lap time and run at high speed and then compare our controller to ones based off of LQR and MPC.**

Source Code:

`https://github.com/thomasfork/MEC231A_project`

Video:

`https://www.youtube.com/watch?v=lMFFV--E890`

I. Introduction

Drones and unmanned aerial vehicles are becoming more and more appealing to the industry for their ability to finish tasks that cannot be done in other ways [1]. Rapid maneuvering of drones in confined spaces is an open challenge in controls and robotics requiring accurate, precise, and high frequency planning, control, and perception. Racing epitomizes this problem and has inspired competitions and research alike [2]. Recent work [3] has shown that a conventional Model Predictive Control (MPC) problem can be augmented with a learned terminal cost and terminal set to learn near-optimal control schemes in an iterative environment. We explore the application of this controller to drone racing.

II. Problem Statement

A. Drone Model

Drones are inherently nonlinear systems with complex aerodynamic and electromechanical properties. In this work we use a simpler discrete time affine drone model presented in [4], eq. (2.16). This is a linear, time-invariant model of the form $x_{k+1} = Ax_k + Bu_k + C$. Where the state and input vectors are

$$x = \begin{bmatrix} x_1 \\ \dot{x}_1 \\ \theta_1 \\ \dot{\theta}_1 \\ x_2 \\ \dot{x}_2 \\ \theta_2 \\ \dot{\theta}_2 \\ x_3 \\ \dot{x}_3 \end{bmatrix} \qquad u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

and the sampling time is 50ms.
Here x_1, x_2, and x_3 are the position of the drone in Euclidean coordinates and θ_1 and θ_2 are the pitch and roll of the drone. u_1 and u_2 are pitch and roll commands and u_3 is a thrust command to the drone. In [4], x_3 points down however we modify the elements of A, B, and C such that x_3 points up in our simulations. Values for these matrices can be found in Appendix A

B. Track Model

We use a custom track built from a sequence of curved segments. Each segment is defined by a path length, radius of curvature (except for straight segments), and a boolean flag that determines whether or not the track curves left/right or up/down. This sequence defines the centerline of the track. We obtain boundaries of the track by sweeping a constant width and height rectangle along the centerline.

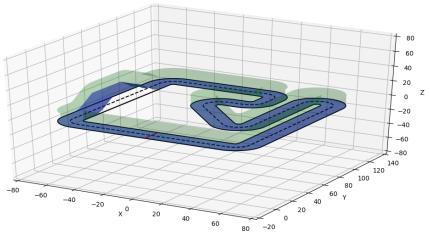

Fig. 1: Rendering of Custom Track Model

When flying a drone along the track it must

stay within this swept shape. These constraints are inherently nonlinear so they are linearized by:

1) Projecting a point $[x_1, x_2, x_3]$ to the closest point along the centerline
2) Computing the orientation of the track at that point
3) Converting track orientation, width, and height to four hyperplane constraints on the point

Additional details on track definition, the method for linearizing constraints, and numerical implementation can be found in Appendix B

III. LQR Controller

Our first comparison controller is an LQR controller applied to the affine drone dynamics. Since LQR is intended for linear systems, we first create an augmented system which is linear:

$$\dot{z} = \tilde{A}z + \tilde{B}u$$

$$z = \begin{bmatrix} x \\ 1 \end{bmatrix}, \tilde{A} = \begin{bmatrix} A & C \\ 0 & 0 \end{bmatrix}, \tilde{B} = \begin{bmatrix} B \\ 0 \end{bmatrix}$$

The infinite horizon Discrete Algebraic Riccati Equation (DARE) for this system does not converge as the added state variable is uncontrollable and incurs an ever-growing cost. Instead we use a modified DARE that is solved recursively and removes the cost associated with the uncontrollable state at each iteration. This results in a feedback policy $u = Kz$. Since z is an augmented state, we can decompose the feedback policy into

$$u = K_{linear}x + K_{affine}$$

Where K_{linear} is identical to the LQR feedback matrix if the affine dynamic term was removed and K_{affine} is a control signal that is necessary for the origin to be an equillibrium point.

By itself the feedback policy $u = K_{linear}x + K_{affine}$ will asymptotically move the drone to the origin. To follow the track and reach the finish line we offset the state variable x with a target point situated ahead of the drone and on the centerline of the track

$$u = K_{linear}(x - x_{ref}) + K_{affine}$$

$$x_{ref} = [x_{1,ref}, 0, 0, 0, x_{2,ref}, 0, 0, 0, x_{3,ref}, 0]^\top$$

As the drone approaches x_{ref}, x_{ref} is moved further along the track until the drone reaches the finish line. Constraints are not explicitly handled by the LQR controller but the distance between the drone and x_{ref} was tuned to avoid constraint violation.

IV. MPC Controller

Our second comparison controller is an MPC controller that pursues a target point similar to LQR but is solved over a finite horizon with constraints on state and control output. At each time step a finite horizon optimal control problem is solved, and the first planned control output is applied to the drone. The optimal control problem used is:

$$J^* = \min_{\mathbf{U}_t, \mu_t, \boldsymbol{\eta_t}} \sum_{k=t}^{t+N-1} \big[||x_{k|t} - x_f||_Q + ||u_{k|t}||_R\big] + ||\eta_{k|t}||_{Q_\eta}] + ||\mu_t||_{Q_\mu} + ||x_{t+N|t} - x_f||_P \tag{1a}$$

$$\text{s.t. } x_{k+1|t} = Ax_{k|t} + Bu_{k|t} + C \tag{1b}$$

$$b_{x,l} \le F_x x_{k|t} \le b_{x,u} \quad \forall k = t, \cdots t+N \tag{1c}$$

$$b_{u,l} \le F_u u_{k|t} \le b_{u,u} \quad \forall k = t, \cdots t+N-1 \tag{1d}$$

$$b^k_{x,l} \le F^k_x x_{k|t} + E\eta_{k|t} \le b^k_{x,u} \quad \forall k = t, \cdots t+N-1 \tag{1e}$$

$$x_{t+N} = x_f + \mu_t \tag{1f}$$

$$x_{t|t} = x(t) \tag{1g}$$

$$u(t) = u_{t|t} \tag{1h}$$

Here $\mathbf{U_t}$ is a vector of the planned control outputs. Furthermore (1a) is the cost function on state, control output, and slack variables μ_t and $\boldsymbol{\eta}_t$. (1b) is the affine drone model from section II-A. Equation (1c) is a set of unchanging hyperplane constraints on the state of the drone active at all time and prediction steps and (1d) is a corresponding set of constraints on the controller output. (1e) is a set of hyperplane constraints that are different for all time and prediction steps and is made up of linearized track constraints. The slack variables $\eta_{k|t}$ and the slack matrix E make (1e) a soft constraint. Equation (1f) defines a terminal constraint with slack variable μ_t for the target point x_f. Each slack variable is penalized with respective cost matrices Q_η and Q_μ. Equations (1g) and (1h) are the receding horizon controller.

Each time the MPC controller is solved, the trajectory $[x_{t+1|t}, \cdots, x_{t+N|t}]$ is used as the reference about which to linearize the track constraints in

(1e) for the next solution. Additionally, the target point x_f is recomputed using the current position of the drone. Like in LQR, x_f is moved to guide the drone along the track and is placed far enough away that the terminal slack μ_t is always nonzero. This effectively makes the terminal slack cost Q_μ a terminal state cost that drives the drone along the track. The distance to x_f is tuned to avoid cases where the drone fails to circumnavigate nonconvex obstacles, e.g.a u-turn.

V. LMPC Controller

The LMPC controller implemented is similar to Equation (1) but with two key differences:

1) The terminal constraint and cost is replaced with the safe set and cost-to-go from [3]
2) The track constraints are linearized about the safe set rather than the previous trajectory
3) Unlike [3], we obtain a safe set by finding the point closest to the current drone state and the next K_{SS} points from the previous trajectory.

These differences from [3] are motivated by the nonconvexity of the track constraints when formulated in the global frame of reference. Since the convex hull of the safe set is not necessarily feasible, track constraints are linearized about the first N safe set points so that the track constraints establish a trust region of the safe set. This also helps guide the drone around nonconvex obstacles. Outside of the linearized track constraints the safe set cost is increased by the slack variable cost, which helps recover the desired safe set.

The LMPC optimal control problem is thus:

$$J^* = \min_{\mathbf{U}_t, \mu_t, \boldsymbol{\eta_t}, \lambda} \sum_{k=t}^{t+N-1} \big[||x_{k|t} - x_f||_Q + ||u_{k|t}||_R + ||\eta_{k|t}||_{Q_\eta}\big] + ||\mu_t||_{Q_\mu} + \lambda Q_{SS} \tag{2a}$$

$$\text{s.t. } x_{k+1|t} = Ax_{k|t} + Bu_{k|t} + C \tag{2b}$$

$$b_{x,l} \le F_x x_{k|t} \le b_{x,u} \quad \forall k = t, \cdots t+N \tag{2c}$$

$$b_{u,l} \le F_u u_{k|t} \le b_{u,u} \quad \forall k = t, \cdots t+N-1 \tag{2d}$$

$$b^k_{x,l} \le F^k_x x_{k|t} + E\eta_{k|t} \le b^k_{x,u} \quad \forall k = t, \cdots t+N-1 \tag{2e}$$

$$x_{t+N} = \lambda X_{SS} + \mu_t \tag{2f}$$

$$\lambda_i \ge 0; \sum_i \lambda_i = 1 \tag{2g}$$

$$x_{t|t} = x(t) \tag{2h}$$

$$u(t) = u_{t|t} \tag{2i}$$

Where the new variables are

1) λ: Weighting coefficients for convex hull of safe set
2) Q_{SS}: Cost-to-go terms for safe set; λQ_{SS} corresponds to Barycentric interpolation of the cost to go
3) X_{SS}: States of safe set points; λX_{SS} corresponds to a point in the convex hull of the set of X_{SS}.

Similar to the track constraints, the safe set terms are updated before solving Equation 2. Given a previous trajectory $\{\hat{x}_k, \hat{u}_k\}$ which traverses the track, the safe set is found as:

$$I^* = argmin_k(||\hat{x}_k - x(t)||_2) \tag{3}$$

$$X_{SS} = \{\hat{x}_{I^*}, \cdots, \hat{x}_{I^*+K_{SS}-1}\} \tag{4}$$

Where $x(t)$ is the current position of the drone and K_{SS} is the number of elements to find for the safeset. The cost-to-go Q_{SS} is computed by finding how long it took the drone to reach the finish line from each point and multiplying by a scaling factor.

$$Q_{ss} = \{(t_{lap} - t) * \gamma_{SS}\}, t = I^*, \cdots, I^* + K_{ss} - 1 \tag{5}$$

In the event that the safe set crosses the finish

line, the cost-to-go for points after the finish line is decreased to preserve the gradient of Q_{SS}.

VI. Simulation and Results

We implemented the simulated drone and track in Python and used OSQP to solve the linearized convex LMPC QP. We ran all three controllers on the track shown in Figure 1 and simulated the system with no model mismatch - the same model was used for control and simulation. For each controller, we compared lap time, drone speed, and lateral and vertical position relative to the track centerline.

The same state and input costs were used for LQR and MPC controllers: $Q = I_{10}$ and $R = I_3$. For MPC we used a prediction horizon of 30, and used the terminal cost from solving the Ricatti equation for LQR. MCP and LMPC slack variables were hand-tuned to make constraint violation minimal. For LMPC we set all costs except safe set and slack variable costs to zero, this way lap time was minimized while remaining outside of the soft constraints.

We used the closed loop LQR trajectory as the initial trajectory for LMPC and ran it for 30 laps. The learning curve is shown in Figure 2. After 30 laps LMPC converged to a lap time of 59.05 seconds which is 71 seconds faster than the LQR controller and 19 seconds faster than the MPC controller.

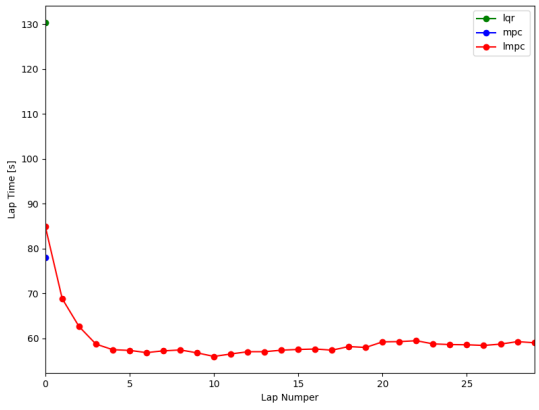

Fig. 2: LMPC Learning Curve

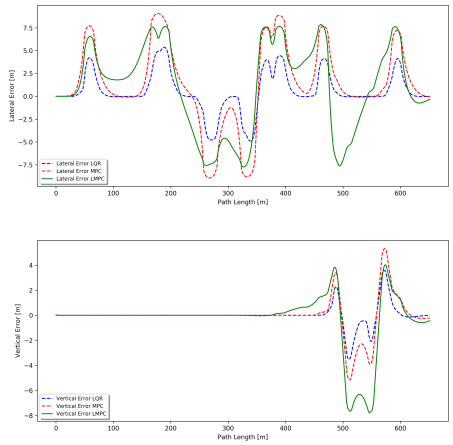

Fig. 3: Lateral and Vertical Position Comparison

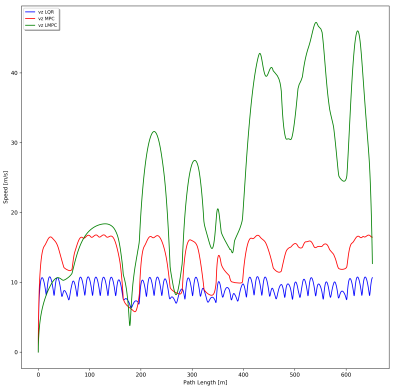

Fig. 4: Drone Speed Comparison

Figure 3 shows lateral and vertical position as a function of track path length for each controller. LQR closely followed the track centerline while LMPC and MPC cut corners and finished the lap faster. The MPC is shortsighted by the predicition horizon so it never learns to go faster than it can go within that horizon. Meanwhile the LMPC controller learns to go as fast as possible while satisfying constraints.

VII. Conclusion

In this paper we showed that a Learning Model Predictive control approach allows a drone to iteratively learn to race around a track and achieves shorter lap times than LQR and MPC. We also demonstrated LMPC in the presence of nonconvex constraints, higher dimensions, and longer prediction horizons than /ref8896988. Although the controller is successful in this scenario there remains

work to explore additional models and environments, such as tighter turns, more complex geometry, and realistic nonlinear drone models, which would have to be linearized for OSQP.

VIII. ACKNOWLEDGMENTS

The authors gratefully acknowledge advice and comments from Francesco Borrelli and Ugo Rosolia.

REFERENCES

[1] T. Witcher, "Rise of the drones," *Rise of the Drones*, vol. 85, no. 8, pp. 60–67, 2015. DOI: 10.1061/ciegag.0001027.

[2] E. Kaufmann, M. Gehrig, P. Foehn, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Beauty and the beast: Optimal methods meet learning for drone racing," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 690–696. DOI: 10.1109/ICRA.2019.8793631.

[3] U. Rosolia and F. Borrelli, "Learning how to autonomously race a car: A predictive control approach," *IEEE Transactions on Control Systems Technology*, vol. 28, no. 6, pp. 2713–2719, 2020. DOI: 10.1109/TCST.2019.2948135.

[4] P. Bouffard, "On-board model predictive control of a quadrotor helicopter: Design, implementation, and experiments," M.S. thesis, EECS Department, University of California, Berkeley, Dec. 2012. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-241.html.

APPENDIX

A. Drone Model

We flipped the signs of B_z and k_z such that x_3 points up, rather than down. Otherwise the matrices are identical to those from [4].

$$A = \begin{bmatrix} A_t & B_tC_\theta & 0 & 0 & 0 \\ 0 & A_\theta & 0 & 0 & 0 \\ 0 & 0 & A_t & B_tC_\theta & 0 \\ 0 & 0 & 0 & A_\theta & 0 \\ 0 & 0 & 0 & 0 & A_z \end{bmatrix}$$

$$B = \begin{bmatrix} B_l & 0 & 0 \\ 0 & B_l & 0 \\ 0 & 0 & B_z \end{bmatrix}$$

$$C = \begin{bmatrix} 0 \\ 0 \\ k_z \end{bmatrix}$$

And

$$A_t = \begin{bmatrix} 1 & 0.025 \\ 0 & 1 \end{bmatrix}$$

$$B_tC_\theta = \begin{bmatrix} 0.0031 & 0 \\ 0.2453 & 0 \end{bmatrix}$$

$$A_\theta = \begin{bmatrix} 0.7969 & 0.0225 \\ 1.7976 & 0.9767 \end{bmatrix}$$

$$B_l = \begin{bmatrix} 0 \\ 0 \\ 0.01 \\ 0.9921 \end{bmatrix}$$

$$B_z = \begin{bmatrix} 0.00021875 \\ 0.0175 \end{bmatrix}$$

$$k_z = \begin{bmatrix} -0.0031 \\ -0.2453 \end{bmatrix}$$

B. Track Model

Formally, the track is a parametric shape, defined by a an injective map

$$[s, e_l, e_n] \rightarrow [x_1, x_2, x_3]$$

Where s is the path length along the track, e_l is a lateral deviation from the centerline, and e_n is a normal deviation from the centerline. We ignore centerlines that twist, so e_l is always in the (x_1x_2) plane, and e_n is defined geometrically by $e_s \times e_l = e_n$, where e_s is a unit vector tangent to the centerline.

As stated in section II-B, We use a track centerline composed of line and arc segments which may curve in the (x_1x_2) plane or up/down. Position along this centerline can be computed as a function of s as follows:

Additionally, e_l and e_n can be added at the end by using the final orientation of the track and adjusting the returned point. We compute the orientation of the track as

$$A_{track} = [\hat{e}_s, \hat{e}_l, \hat{e}_n] =$$

$$\begin{bmatrix} cos(\theta)cos(\phi) & -sin(\theta) & -cos(\theta)sin(\phi) \\ sin(\theta)cos(\phi) & cos(\theta) & -sin(\theta)sin(\phi) \\ sin(\phi) & 0 & cos(\phi) \end{bmatrix}$$

Algorithm 1: Centerline Computation

Result: $[x_1, x_2, x_3]$ given s]
$x_1 = x_2 = x_3 = 0$
$\theta = 0$
$\phi = 0$
s_c = 0
$p := [x_1, x_2, x_3]$ **while** $s_c < s$ **do**
  move to next segment;
  if *s on segment* **then**
    if *segment is straight* **then**
      compute orientation from (θ, ϕ) and move p until s_c = s
      return p
    else
      compute arc segment center from (θ, ϕ, p), arc angle from radius and $s_c - s$, and corresponding new p on the arc
      return p
    end
  else
    if *segment is straight* **then**
      compute orientation from (θ, ϕ) and move p to end of segment
      s_c += length of segment
    else
      compute arc segment center from (θ, ϕ, p), arc angle from radius and length of segment, and new p at end of arc segment
      s_c += length of segment
      change θ or ϕ depending on radius and direction of curvature
    end
  end
end

And can compute

$$p = p(Algorithm\ 1) + e_l\hat{e}_l + e_n\hat{e}_n$$

Which completes the injective map

$$[s, e_l, e_n] \rightarrow [x_1, x_2, x_3]$$

This map loses injectivity whenever the track radius of curvature is smaller than its half-width; such tracks are ignored in this paper.

Although this map can be computed quickly and efficiently by keeping track of the starting positions and path length of each segment, it is expensive to invert, e.g. the map

$$[x_1, x_2, x_3] \rightarrow [s, e_l, e_n]$$

Is in difficult to compute even when the track definition is injective. As this computation is essential for linearizing track constraints, we speed it up by computing a sequence of waypoints along the track, defined as an array of

$$[s_j, p_j, A_{track,j}, \theta_j, \phi_j]$$

Given a point p_{query}, we can compute the index i of the closest waypoint by computing the Euclidean distance between the query point and each waypoint. We can then approximate the track coordinates of the point using s_i, p_i, and $A_{track,i}$:

$$[s, e_l, e_n] \approx [s_i, 0, 0] + (A_{track,i})^{-1}(p - p_i)$$

Which provides an efficient means for approximating the drones position along the track. This can be used for computing lap progress and completion, but is most important for linearizing constraints. The true track constraints are

$$-b_l <= e_l <= b_l$$

$$-b_n <= e_n <= b_n$$

Which are nonlinear in the state space model of the drone. To linearize these constraints about a point p_0 we can compute the waypoint $s_i, p_i, A_{track,i}$ closest to the given point and obtain linearized constraints as:

$$\begin{bmatrix} -inf \\ -b_l \\ -b_n \end{bmatrix} + A_{track,i} * p_0 <= A_{track,i} * p <= \begin{bmatrix} inf \\ b_l \\ b_n \end{bmatrix} + A_{track,i} * p_0$$

Which provides hyperplane constraints on arbitrary p given p_0 and is the method used throughout this paper for computing linearized track constraints. These constraints are overly conservative on the inner boundary of the track, and unsafe at the outer boundary of the track. For racing this is sufficient since the optimal drone behaviour is to tightly follow the inner boundaries of the track.