Final Report:
Highly Dynamic Quadcopter Control For Drone Racing

Alvin Shek and Tom Scherlis

***Abstract*— Drone racing involves navigating a small quadcopter through a series of gates at high speeds, often requiring highly dynamic flight and careful consideration of the constraints presented by the drone gates themselves. In this work, we present an approach to generate optimal minimum-snap reference trajectories given only the positions and orientations of the gates. We also present a controller based on state-feedback linearization and MPC in order to track these trajectories in real-time.**

Video: `https://youtu.be/uN9TzCkSSKk`
Code: `https://git.io/JfCZS`

I. Introduction

Quadcopters are highly dynamic, 6-degree-of-freedom nonlinear systems that often need to follow complex paths. Their prominence has grown significantly over the past decade from applications such as video-making and racing. We were inspired by the idea of designing autonomous quadcopters to participate in these races, and possibly other applications such as defense.

Typically, nonlinear systems can be approximated as linear and solved with fast, effective approaches such as Linear Quadratic Regulator(LQR) and Model Predictive Control (MPC). However, linear approximations only hold true close to their set point, and grossly miss the true values outside this neighborhood. For quadcopters, the small angle approximation assumes the quadcopter will remain near the neutral hovering state. While this assumption works well for applications in cinematography and surveillance, the model fails for use cases that require high-speed, dynamic maneuvering such as drone-racing and target tracking.

In this paper, we focus on the application of drone racing, where quadcopters are required to fly time-optimal paths through gates that are highly dynamic and violate small angle approximations. Differential flatness is a property that allows a nonlinear system to be separated into a linear system and a nonlinear transformation mapping between the two systems [11]. We exploit the differential flatness property of quadcopter dynamics to apply feedback linearization on the nonlinear dynamics, enabling us to solve a linear, transformed system for the optimal controls. We apply MPC on this flat system and compare its performance to LQR.

Tracking typically involves following a series of waypoints defined by position and orientation, but in this paper, we also present a method for generating optimal reference trajectories through a series of drone gates, such that we indirectly minimize the low-level controller efforts.

We will be using the MIT FlightGoggles simulator [9] for testing, as it has an implementation of the quadcopter dynamic. We use the ROS framework to develop our control software such that it runs in real-time with the simulator. We use ground-truth position for feedback.

II. Problem Statement

The problem is composed of two parts. First, we must generate a reference trajectory parameterized by time for the controller, given a list of gate positions and orientations. Second, we must design a controller to track this trajectory.

We define two coordinate frames: the fixed world frame (North-East-Down) and the quadcopter body frame. A quadcopter's state is typically made up of position, orientation, linear velocity, and angular velocity respectively:

$$\mathbf{x} = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, p, q, r]^T$$

where position and velocity are defined NED world frame. p, q, r respectively represent rate of change of angular velocity about x, y, z. ϕ, θ, ψ are euler angles about x, y, z respectively.

Most quadcopters take as input desired desired mass-normalized thrust and angular velocities:

$$\mathbf{u} = [T, p, q, r]^T$$

With this state and control space, our goal is to have the controller find some optimal control trajectory to take the quadcopter from initial state $\mathbf{x_0}$ and track a trajectory $\mathbf{x_g}$. For the other aspect of the project, our challenge was to convert a set of gate positions and orientations into a spline in order to determine $\mathbf{x_g}$.

For simulation, we use MIT Flightgoggles [9]. Flightgoggles provides a quadcopter dynamics model simulating the rigid body dynamics, rotor forces, and rigid body aerodynamic forces.

III. Related Work

In [12], differential flatness is also used to apply a linear controller to drone dynamics. However, rotor drag makes up a very significant portion of the dynamics calculations in [12]. In our use case, however, FlightGoggles only models rigid-body aerodynamics in the simulation, which allows us

to avoid this additional complexity.

[8] Presents an approach to quadcopter trajectory generation using minimum snap trajectories. We build on this work by adding directional constraints as a means to enforce travel direction through an obstacle such as a drone gate, window, or doorway. We also implemented two ways of determining trajectory durations. We also discuss the application of these trajectories to the example of drone racing.

IV. Methodology

A. Optimal Spline Generation

To generate reference trajectories, we build on the minimum snap trajectory generation in [8]. Snap, the fourth derivative of position, is minimized as it indirectly minimizes angular velocity inputs p, q, r. We parameterize trajectories as fifth order piecewise polynomials parameterized by time. We build polynomials for $\mathbf{x} = ([x, y, z])^\top$. ψ is computed separately as a post-processing step to keep the quadcopter oriented forward along the trajectory.

$$\mathbf{x}_n(t) = \mathbf{A_n} + \mathbf{B_n}(t - t_n) + \mathbf{C_n}(t - t_n)^2 + \mathbf{D_n}(t - t_n)^3 + \mathbf{E_n}(t - t_n)^4 + \mathbf{F_n}(t - t_n)^5 \tag{1}$$

Where n corresponds to the index of the spline segment containing t, and t_n is the time associated with the start of segment n.

To minimize snap of a segment, we minimize the integral from 0 to d_n of the 4th derivative of the polynomial, where d_n is the duration of segment n, equal to $t_{n+1} - t_n$. To simplify things, we focus only on one dimension, x.

$$s_n = \int_0^{d_n} \left[\frac{d^4}{dt^4} x_n(t)\right]^2 dt \tag{2}$$

Thus, the total snap s over a given dimension is:

$$s = \sum_{n=0}^{N} s_n \tag{3}$$

Where N is the total number of segments on the trajectory. We therefore choose to minimize $\mathbf{s}^T\mathbf{s}$ to minimize snap in all dimensions.

1) Formulating the QP: To solve using a QP solver, we must put the problem into QP form:

$$\begin{aligned} \min \frac{1}{2} x^T H x + f^T x \\ \text{s.t. } Lx \leq b \end{aligned} \tag{4}$$

First, we frame s as a linear combination of our optimization variables, A, B, C, D, E, F. We also assume that t_n and d_n are known ahead of time for each segment. Thus, s_n for a single dimension is computed as:

$$s_n = \mathbf{k_n}^T \mathbf{Q_n} * \mathbf{k_n} \tag{5}$$

where k_n is a vector of coefficients: $[A_n, B_n, C_n, D_n, E_n, F_n]$ and $\mathbf{Q}_n$ is the positive semidefinite hermitian matrix.

To define Q, we need to plug d_n and t_n into (2). Replacing the square with an outer product allows us to compute s_n as a linear function of $\mathbf{k}_n$.

Q_n is derived in [13] as:

$$\mathbf{Q_n} = \begin{bmatrix} \mathbf{0}_{4\times4} & \mathbf{0}_{4\times2} \\ \mathbf{0}_{2\times4} & \mathbf{W_n} \end{bmatrix} \tag{6}$$

where:

$$\mathbf{W_n} = \begin{bmatrix} 24*24*d_n & 24*120*\frac{1}{2}*d_n^2 \\ 24*120*\frac{1}{2}*d_n^2 & 120*120*\frac{1}{3}*d_n^3 \end{bmatrix} \tag{7}$$

24 and 120 correspond to the coefficients that arise from differentiating eq 1. d_n is the duration of the segment. This can be generalized for any order polynomial minimizing the integral of any order derivative.

Thus, to solve for the full trajectory we set $x = [\mathbf{k}_0; \mathbf{k}_1; ...\mathbf{k}_N]$ Likewise, $\mathbf{H} = \text{diag}([\mathbf{Q}_0, \mathbf{Q}_1, ...\mathbf{Q}_N])$.

2) Waypoint constraints: Waypoint constraints define the value of a given derivative of the polynomial at time 0. For example, a zero-order waypoint constraint defines the position at the beginning of the segment. A first order constraint defines the velocity.

We can put these constraints into matrix form as follows. Start by letting $\mathbf{c_t}(t, o)$ correspond to the vector corresponding to the t coefficients of equation 1 at order o. For example, $\mathbf{c_t}(t, 0) = [t, t^2..., t^5]^T$, whereas $\mathbf{c_t}(t, 4) = [0, 0, 0, 0, t, t^2]^T$. Likewise, let $\mathbf{c_d}(o)$ correspond to the polynomial coefficients from differentiation at order o. For example, for $\mathbf{c_d}(0) = [1, 1, 1, 1, 1, 1]^T$ whereas $\mathbf{c_d}(4) = [0, 0, 0, 0, 24, 120]^T$. The overall coefficient vector $\mathbf{c}(t, o)$ is computed as the elementwise product of $\mathbf{c}_t$ and $\mathbf{c}_d$. Thus:

$$\frac{d^n x_n}{dt^n} = \mathbf{c}(t, n)^T \mathbf{k}_n \tag{8}$$

Thus, we can easily add each of these constraints as an equality constraint:

$$\mathbf{c}(t, n)^T * \mathbf{k}_n = \mathbf{b} \tag{9}$$

where b is the desired waypoint value. This can be relaxed to a pair of inequality constraints in order to enforce waypoints with a nonzero acceptable radius.

3) Continuity Constraints: To enforce continuity along the spline, we construct constraints that the values of each derivative of interest must be equal for connecting segments. In our case, we enforce 2nd order continuity. For each pair of segments $x_n(t)$ and $x_{n+1}(t)$, we enforce the following:

$$\begin{aligned} x_n(d_n) &= x_{n+1}(0) \\ x'_n(d_n) &= x'_{n+1}(0) \\ x''_n(d_n) &= x''_{n+1}(0) \end{aligned} \tag{10}$$

These can be formulated as equality constraints in the form

$$\mathbf{c}(d_n, o)^T * \mathbf{k}_n + \mathbf{c}(0, o)^T * \mathbf{k}_{n+1} = 0 \tag{11}$$

Where o is the order of the continuity constraint. Adding these constraints to our QP formulation is as simple as adding two rows to $\mathbf{L}$ in the form $[\mathbf{0}_{1\times 6n}, \mathbf{c}(d_n, o)^T, \mathbf{c}(0, o)^T, \mathbf{0}_{1\times N-6(n+1)}]$, where one of the rows is made negative in order to enforce equality as a pair of inequality constraints. The zero vectors are used to pad the coefficient vectors to correspond with the correct pair of $\mathbf{k}$ vectors.

4) Direction constraints: Finally, to enforce the direction of flight at a given point, we introduce directional constraints. These are useful for constraining the direction of, say, the velocity vector at a given time without constraining it's magnitude. In order to construct these constraints, we need to optimize all 3 dimensions of the parametric splines together, since they relate x, y and z. As shown above for continuity constraints, we can easily constrain the sum of any set of polynomials at some time t, where each polynomial is characterized by its k_n vector.

In order to constrain direction, we constrain the motion in all directions orthogonal to the desired direction to zero, or some radius as desired. These directions are the basis vectors $\mathbf{n}_1$ and $\mathbf{n}_2$ null space of the desired direction vector:

$$\begin{bmatrix} \mathbf{c}(t,o)^T k_{x,n} \\ \mathbf{c}(t,o)^T k_{y,n} \\ \mathbf{c}(t,o)^T k_{z,n} \end{bmatrix} \cdot n_1 = 0 \\ \begin{bmatrix} \mathbf{c}(t,o)^T k_{x,n} \\ \mathbf{c}(t,o)^T k_{y,n} \\ \mathbf{c}(t,o)^T k_{z,n} \end{bmatrix} \cdot n_2 = 0 \tag{12}$$

An additional constraint can be added to enforce positive dot product with the desired direction vector if necessary.

5) Creating trajectories for drone races: For each gate, we add 3 constraints:

1) Position waypoint at the center of the gate
2) 1st order direction waypoint along the direction of the gate
3) Continuity constraints up to acceleration on the connecting spline segments

6) Determining Segment durations: So far, the trajectory optimization has assumed that d_n of each segment is known. We determined two ways to solve for d_n:

Higher level optimization: A cost function can be constructed to weight the total trajectory duration against the total snap (or jerk, acceleration, etc) along the trajectory by reusing the $\mathbf{H}$ matrix from eq 4, but generalized to some order o:

$$\mathbf{x}^T \mathbf{H} \mathbf{x} + \alpha \sum_{n=0}^{N} d_n \tag{13}$$

This nonlinear optimization problem can then be solved to find the optimal durations and coefficients for some aggressiveness determined by tuning α. In practice, however, this process was slow and often did not converge well, so we also use a second method:

Linear velocity assumption: d_n can be calculated based on the euclidean distance between the segment's endpoints and assuming some linear velocity. While not optimal, the results were acceptable and the solution is analytical. Using this assumption allows very fast ($\leq$10ms) computation of an optimal spline for around 15 gates. This allows the user to recompute the spline online as needed.

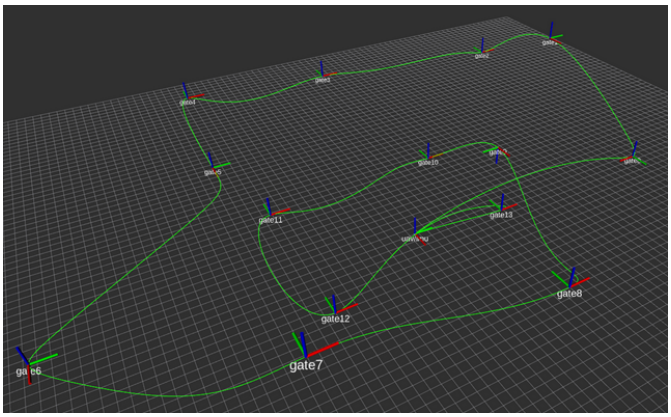

Fig 1: Trajectory Example

B. Controller

The first step for designing the controller is to map the original states and controls to the flat system. The new flat states can be represented as follows:

$$\mathbf{x} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \psi]^T$$

Linear position and velocity can be controlled using only roll(θ) and pitch(ϕ), leaving yaw(ψ) redundant. For this reason, roll and pitch can be treated as part of control space as desired attitude and handled by the our level attitude controller. Angular rates are inputs into a typical low-level quadcopter controller and so they are not needed in the state.

Control:

$$\mathbf{u} = [\mathbf{u_p}^T, u_\psi]^T \tag{14}$$

where u_p is related to linear acceleration:

$$\mathbf{u_p} = R_z^T(\psi) R_y^T(\theta) R_x^T(\phi) \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \frac{T_d}{m} \tag{15}$$

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \mathbf{u_p} + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{16}$$

and u_ψ is desired yaw rate r:

$$r = p\frac{s_\phi}{c_\theta} + r\frac{c_\phi}{c_\theta} \tag{17}$$

We now have a mapping from the flat states and their derivatives to the original states and their control inputs,

which means the system is differentially flat. The dynamics of the flat system can be expressed as a general linear system:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} + \mathbf{b}g \tag{18}$$

where A, B are defined as:

$$A = \begin{bmatrix} 0_{3x3} & I_{3x3} & 0_{3x1} \\ 0_{3x3} & 0_{3x3} & 0_{3x1} \\ 0_{1x3} & 0_{1x3} & 0_{1x1} \end{bmatrix} \tag{19}$$

$$B = \begin{bmatrix} 0_{3x3} & 0_{3x1} \\ I_{3x3} & 0_{3x1} \\ 0_{1x3} & 1 \end{bmatrix} \tag{20}$$

and b is a vector that factors in gravity, g, into z-axis acceleration:

$$b = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}^T \tag{21}$$

The above system can be fed into common algorithms like LQR and MPC. Now that we have the optimal controls of the flat system, we need to transform these back to the original control space.

The low-level controller built into FlightGoggles takes as input angular rates and mass-normalized thrust. Thrust is simply the norm of the acceleration vector, $\mathbf{u_p}$, but we perform an additional step of aligning the vector with the quadcopter's current z-axis rather than the desired z-axis. Given the current normal vector of the quadcopter, $\mathbf{n}$:

$$T_d = \max(0, \mathbf{n} \cdot \mathbf{u_p}) \tag{22}$$

The max operation ensures that the drone won't attempt to exert negative thrust. Because thrust is nonlinear, we cannot enforce positive thrust in our controller. This can lead to certain cases where the desired negative thrust is unattainable.

Given our derivation of acceleration vector $\mathbf{u_p}$ from (15), we can derive desired roll and pitch:

$$\mathbf{z} = R_z(\psi)\mathbf{u_p}\frac{-m}{T_d} = R_y^T(\theta)R_x^T(\phi)\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{23}$$

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} -s_\theta c_\phi \\ s_\phi \\ c_\theta c_\phi \end{bmatrix} \tag{24}$$

$$\begin{aligned} \phi_d &= \sin^{-1}(-z_2) \\ \theta_d &= \tan^{-1}(\frac{z_1}{z_3}) \end{aligned} \tag{25}$$

We can set desired yaw $\psi_d = 0$ since it is arbitrary, or atan2(y', x') in order to fly straight forwards at all times.

With the above high-level controller providing desired attitude, we tuned a low-level PD controller to output the angular rates to achieve this target. FlightGoggles provides a high-frequency(around 1kHz) PD controller converting angular rates to direct rotor torques. This mimics the inputs of most commercial quadcopters. This architecture is shown below:

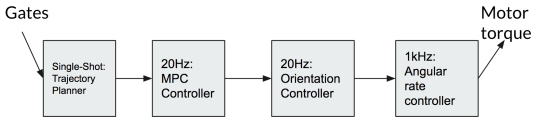

Fig 2. Control Architecture

We tuned our PD controller by producing step responses of desired attitutes that oscillated +/- some angle. With this, we increased K_p, the proportional gain to minimize time taken to reach a target, and increased K_d, the derivative gain to eliminate any oscillations.

We overall chose the following parameters for the MPC Controller:

$$\begin{aligned} Q = diag(20, 20, 20, 0.1, 0.1, 0.1, 1) \in R^7 \\ R = diag(0.1, 0.1, 0.1, 10) \in R^4 \\ S = Q * 10 \in R^7 \end{aligned} \tag{26}$$

The Q matrix represents the weights we place on error of different components of the state, and for our model, we weighed error in state the most since position-tracking was the most important.
The R matrix represents the weights we place on control effort, and for our model, we chose to minimize yaw-rate the most since we don't want the quadcopter to spin unnecessarily.
The S matrix represents the weights placed on final state error, and we chose to weight this more heavily to place priority on reaching a target state.
The MPC controller was run with a 10-step horizon at 20Hz, or overall 0.5 second horizon.

V. Experiments

We initially developed an LQR controller to test our system, and used acceleration from the reference trajectory as a feedforward. In practice, this feedforward becamse ineffective when error grew, which only caused error to grow further. We decided to also develop an MPC controller due to it's ability to track full trajectories rather than just stabilize the system. Overall, the MPC controller massively decreased tracking error and improved the cornering performance significantly. The below figure shows compares performance of both controllers in tracking error of the

quadcopter while moving at approximately 25mph:

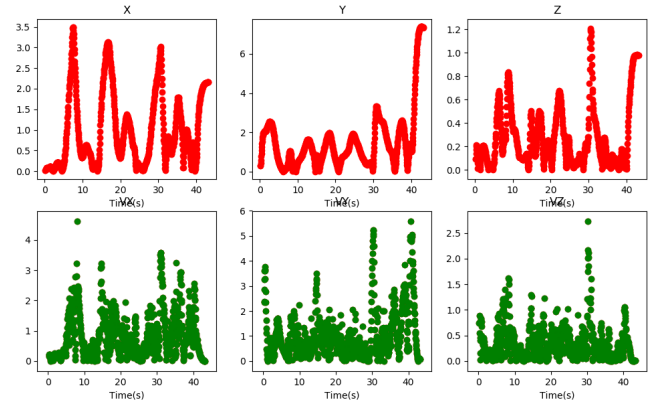

Fig 3. LQR Tracking Error

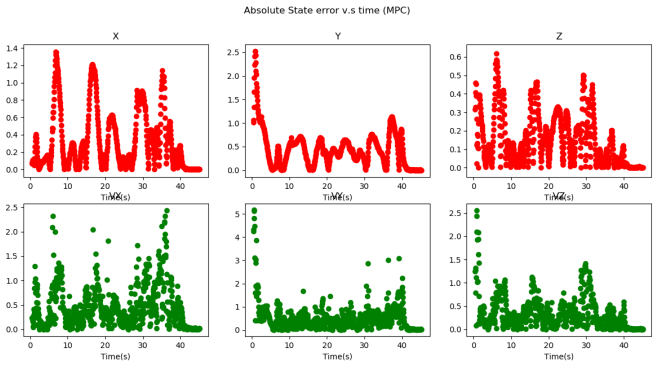

Fig 4. MPC Tracking Error

The above two figures compare absolute tracking error of linear position and velocity of both controllers. The individual plots are ordered in row-major order: x, y, z, $\dot{x}, \dot{y}, \dot{z}$. Comparing corresponding axes scales, LQR controller has nearly double the error of MPC. The oscillating absolute error indicates that both controllers oscillate above and below the desired reference trajectories.

The below figure demonstrates the MPC controller during a test run:

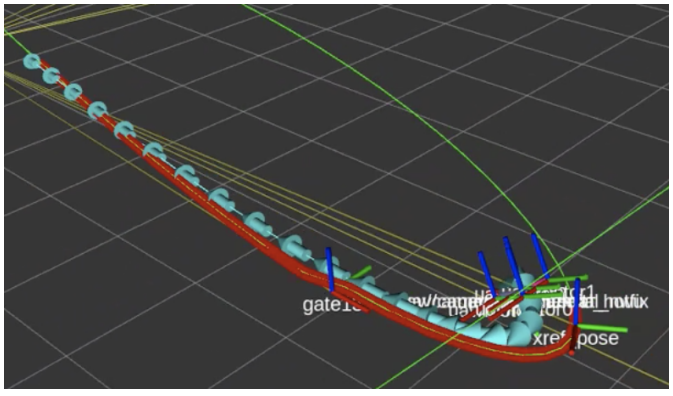

Fig 5. MPC on a sample track

In the figure above, the blue arrows show show MPC predicted trajectory over the time horizon. The green lines are part of the entire pre-generated spline. The red line shows the reference trajectory for the same time horizon. The figure shows how applying feedforward input given a future time horizon helps the drone handle sharp turns well. The LQR controller here would simply attempt to follow the current immediate reference pose using feedback error from the previous timestep. In the above scenario, the LQR controller causes the drone to undershoot the turn completely.

VI. Conclusion

Overall, we first implemented a method for generating optimal reference trajectories through a series of drone gates, such that we indirectly minimize the low-level controller efforts. The generator takes as input a series of drone gate positions and orientations and a hyperparameter of aggressiveness α that defines how fast to move through the trajectory. Second, we implemented feedback linearization on the differentially flat quadcopter dynamics and compared the performance of two controllers, MPC and LQR, on this system.

Through the optimal spline generation, we learned much about polynomial splines and how to formulate complex constraints such that they can be solved with quadratic programming. We also solidified our understanding of two popular controllers, LQR and MPC so we can comfortably apply these in industry. We learned the tradeoffs between horizon, time discretization granularity, and computation time in order to run our MPC controller in real time. We also explored the idea of feedback linearization to apply effective, but linear-based controllers to a nonlinear system. Possible future improvements could include the use of a rotor drag model in our simulation and differential flatness model, as well as learning-based techniques which we originally intended to use to improve disturbance rejection and overall performance. In practice, most drone races involve multiple laps, so ILC could be used to modify our reference trajectories in order to reduce repetive tracking errors. The low-level controller stack could be replaced with adaptive controllers to improve performance and reduce the sensitivity to tuning. Overall, much of our work focused on the software implementation and tuning of our trajectory generation and control systems. Going forward, we plan to reuse this work and specifically apply it to planning and control for an Autonomous Underwater Vehicle (AUV) built by the undergraduate robotics club.

References

[1] A Review of Control Algorithms for Autonomous Quadrotors: https://arxiv.org/pdf/1602.02622.pdf

[2] Dynamics modelling and linear control of quadcopter: https://ieeexplore.ieee.org/document/7813499

[3] Quadrotor control: modeling, nonlinear control design, and simulation: https://www.kth.se/polopoly˙fs/1.588039.1550155544!/Thesis%20KTH%20-%20Francesco%20Sabatino.pdf

[4] Teaching UAVs to Race: End-to-End Regression of Agile Controls in Simulation: https://arxiv.org/pdf/1708.05884.pdf

[5] Quadcopter Dynamics, Simulation, and Control: http://andrew.gibiansky.com/downloads/pdf/Quadcopter%20Dynamics,%20Simulation,

[6] OIL: Observational Imitation Learning: https://arxiv.org/pdf/1803.01129.pdf
[7] Autonomous Driving Motion Planning With Constrained Iterative LQR: https://ieeexplore-ieee-org.proxy.library.cmu.edu/stamp/stamp.jsp?tp=&arnumber=8671755
[8] Minimum Snap Trajectory Generation and Control for Quadrotors: http://www-personal.acfr.usyd.edu.au/spns/cdm/papers/Mellinger.pdf
[9] FlightGoggles: Photorealistic Sensor Simulation for Perception-driven Robotics using Photogrammetry and Virtual Reality: https://arxiv.org/abs/1905.11377
[10] A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning: https://www.ri.cmu.
[11] Differential Flatness of Mechanical Control Systems: A Catalog of Prototype Systems https://www.ri.cmu.edu/pub_files/2011/4/Ross-AISTATS11-NoRegret.pdf
[12] Differential Flatness of Quadrotor Dynamics Subject to Rotor Drag for Accurate Tracking of High-Speed Trajectories http://rpg.ifi.uzh.ch/docs/RAL18˙Faessler.pdf
[13] Minimum snap trajectory planning in MATLAB https://github.com/symao/minimum_snap_trajectory_generation