Model Predictive Contouring Control for the ECE484 Crazyflie Race: Adapting Romero et al. (T-RO 2022) to Attitude Commands

ECE484 Crazyflie Drone project

November 9, 2026

Abstract

This report explains, step by step, how the Model Predictive Contouring Control (MPCC) approach of Romero, Sun, Foehn and Scaramuzza [1] is adapted to our drone racing task, and how it is implemented in `ece484_fly/control/mpcc_controller.py`. The paper controls a large racing quadrotor through its single-rotor thrusts with a rigid-body model. We can only send *collective thrust and attitude* commands to a Crazyflie 2.1, and the dynamics the simulator uses are an *identified* model in which the attitude responds to the command as a second-order system and the thrust as a first-order lag. We derive this model, define the contouring and lag errors on an arc-length parameterized path, formulate the optimal control problem with progress maximization, gate-aware weights, a clearance-based safe corridor and speed limits near gates whose pose is still unknown, and solve it at 100 Hz with acados (SQP real-time iteration, HPIPM). In simulation it finished all 90 evaluation episodes (three tracks, each nominal, with shifted gates and with wind, gusts and battery sag; section 11); the mean solve time is about 1 ms.

Contents

1 Overview **2**

2 The task and what the controller can observe **2**
2.1 Interface . . . 2
2.2 Nominal vs. true gate poses (simulation and hardware) . . . 3

3 Background: MPCC for time-optimal quadrotor flight **3**

4 Drone dynamics used in this project **4**
4.1 Why not the paper's model . . . 4
4.2 Equations . . . 4
4.3 Augmented model for MPCC . . . 5

5 Reference path **6**
5.1 Route . . . 6
5.2 Geometry: cubic spline . . . 6
5.3 Arc-length parameterization . . . 7
5.4 Gate-aware contour weight . . . 7
5.5 Safe corridor radius . . . 7
5.6 Replanning . . . 8

6 Contouring and lag errors **8**
6.1 Definitions . . . 8
6.2 Evaluating the path inside the solver . . . 9

7 The optimal control problem **9**
7.1 Discretization . . . 9
7.2 Cost . . . 9
7.3 Constraints . . . 10
7.4 Full problem . . . 11

8 Solving the problem **11**
8.1 Theory . . . 11
8.2 Implementation with acados . . . 12

9 Closing the loop **12**
9.1 From the observation to the model state . . . 12
9.2 Disturbance observer . . . 12
9.3 Progress state . . . 13
9.4 Warm start, failure handling . . . 13

10 Visualization **13**

11 Results **14**

12 Deployment notes and limitations **16**

A Settings (`MPCCSettings`) **17**

B How to run **17**

1 Overview

A drone race asks for two things at the same time: fly *fast*, and pass every gate *accurately*. Classical pipelines first compute a time-parameterized trajectory $\boldsymbol{p}_{\text{ref}}(t)$ and then track it. The timing is then fixed in advance: if the drone falls behind (a disturbance, a model error), the tracker tries to catch up with a reference point that may have become infeasible, and the plan cannot be faster than the planner's own assumptions.

MPCC removes the time from the reference. The reference is only a *geometric* path $\boldsymbol{p}_d(\theta)$, parameterized by its arc length θ. The controller adds θ to its own state and decides online how fast to move along the path, by trading off

- the *progress* θ (maximized: fly fast), against
- the *contouring error* (distance to the path, minimized: stay on the racing line, especially inside gates).

Because the speed profile is the result of an optimization over the true dynamics and limits, the drone automatically slows down where it must (tight turns, gates) and speeds up where it can.

Figure 1 shows our pipeline. Everything left of the dashed line already existed in the repository (track geometry, planners, arc-length path); the MPCC controller, the estimators and the visualization were added for this work.

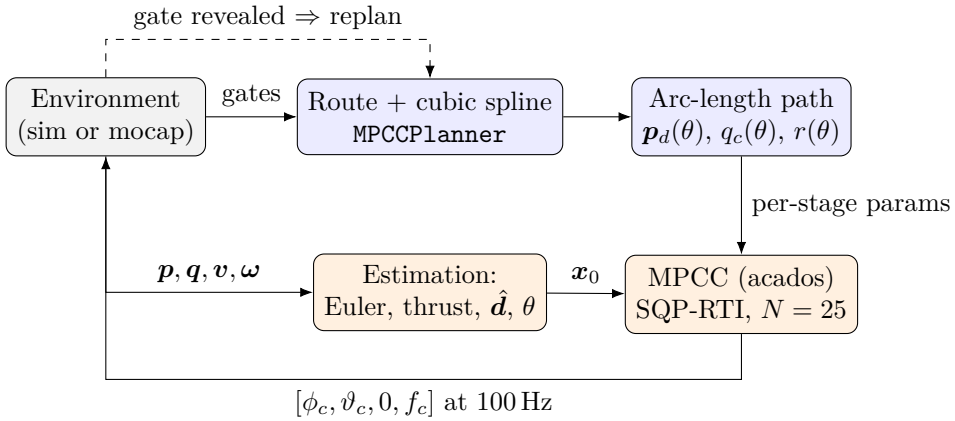


Figure 1: Controller architecture. The planner runs once at the start and again whenever the true pose of a gate or obstacle is revealed; the MPC runs at every environment step (100 Hz).

2 The task and what the controller can observe

2.1 Interface

At every environment step ($f_{\text{env}} = 100$ Hz) the controller receives the drone's position $\boldsymbol{p}$, orientation quaternion $\boldsymbol{q}$ (scipy order x, y, z, w), velocity $\boldsymbol{v}$, body angular velocity $\boldsymbol{\omega}$, the index of the next gate, and the poses of all gates and obstacles. In `attitude` control mode it must return

$$\boldsymbol{u}_{\text{env}} = [\,\phi_c,\ \vartheta_c,\ \psi_c,\ f_c\,] \tag{1}$$

i.e. roll, pitch, yaw in radians and the collective thrust f_c in Newtons. **Note:** the README lists the order as `[thrust, roll, pitch, yaw]`, but both the action space (`race_core.py`, `build_action_space`) and the hardware driver (`real_race_env.py`, `send_action`: `action[3]` is converted to PWM) use *thrust last*. The environment clips the thrust to $[4f_{\min}^{\text{rotor}}, 4f_{\max}^{\text{rotor}}] = [0.085, 0.8]$ N. The physics is integrated at 500 Hz and the staged command is applied at the attitude rate (500 Hz), so a command acts on the drone immediately.

Gates are square frames (0.72 m outer, 0.40 m opening) on a pole. They must be crossed along their local $+x$ axis. Obstacles are thin vertical poles. Touching anything, or leaving the safety box, ends the episode.

2.2 Nominal vs. true gate poses (simulation and hardware)

The observation reports the *nominal* (config file) pose of a gate or obstacle until the drone has come within `sensor_range` (0.7 m, measured in the horizontal plane); from then on the exact pose is reported (and stays revealed). This is implemented in the simulator (`race_core.py`, `_obs`) *and identically on the hardware*: in `real_race_env.py` the motion-capture system does measure the real gate and obstacle poses at reset (`_update_track_poses`), but `obs()` masks them with

```
gates_pos = np.where(gates_visited, gates.pos, gates.nominal_pos)
```

and `gates_visited` is only set once the drone is within `sensor_range`. Hence:

- On the real system, mocap knows the true positions from the start, but *the controller does not get them from the observation before it is close.* The deploy environment deliberately reproduces the simulator's partial observability.
- The drone's own state (position, attitude, velocity) comes from mocap plus an estimator on hardware, and exactly from the simulator in simulation.
- With `real_track_objects = false` the deploy environment uses the nominal poses everywhere, so in that practice mode nominal and true poses agree.

The controller is therefore built for partial knowledge: it plans with nominal poses, replans when a pose changes, and approaches gates whose pose is not revealed yet slowly enough to absorb the correction (section 7.3).

3 Background: MPCC for time-optimal quadrotor flight

We summarize the formulation of [1] (their Sections III and IV), in their notation.

Contouring control problem. Given a continuously differentiable path $\boldsymbol{p}_d(\theta)$ parameterized by arc length, the ideal objective is

$$J_{\text{MPCC}} = \sum_{k=0}^{N} q_c \, (e_k^c)^2 - \rho \, \theta_N, \qquad e_k^c = \min_{\theta} \|\boldsymbol{p}_k - \boldsymbol{p}_d(\theta)\|_2, \tag{2}$$

i.e. stay close to the path and get as far along it as possible. The minimization inside e_k^c is itself an optimization problem, so it is replaced by a progress variable $\hat{\theta}_k$ with dynamics $\hat{\theta}_{k+1} = \hat{\theta}_k + v_{\theta,k}\Delta t$ and the error $\boldsymbol{e}(\hat{\theta}_k) = \boldsymbol{p}_k - \boldsymbol{p}_d(\hat{\theta}_k)$ is split into a lag part (along the tangent) and a contour part (orthogonal to it), section 6. Maximizing θ_N is rewritten as maximizing $\sum_k v_{\theta,k}$ with weight μ.

Model. The paper uses the full rigid-body model with quaternion $\boldsymbol{q}$, body rates $\boldsymbol{\omega}$, the four single-rotor thrusts $\boldsymbol{f}$ as states, and linear drag:

$$\dot{\boldsymbol{p}} = \boldsymbol{v}, \quad \dot{\boldsymbol{q}} = \tfrac{1}{2}\boldsymbol{q} \odot [0, \boldsymbol{\omega}], \quad \dot{\boldsymbol{v}} = \boldsymbol{g} + \tfrac{1}{m}\boldsymbol{R}(\boldsymbol{q})\boldsymbol{f}_T - \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^\top \boldsymbol{v}, \quad \dot{\boldsymbol{\omega}} = \boldsymbol{J}^{-1}(\boldsymbol{\tau} - \boldsymbol{\omega} \times \boldsymbol{J}\boldsymbol{\omega}). \tag{3}$$

The state is augmented to $\boldsymbol{x} = [\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{f}, \theta, v_\theta]$ and the inputs are the *rates* $\boldsymbol{u} = [\Delta v_\theta, \Delta \boldsymbol{f}]$, so that thrusts and progress speed change smoothly.

Optimal control problem. (their Eq. 17)

$$\begin{aligned} \min_{\boldsymbol{u}} \; & \sum_{k=0}^{N} \|\boldsymbol{e}_k^l\|_{q_l}^2 + \|\boldsymbol{e}_k^c\|_{q_c}^2 + \|\boldsymbol{\omega}_k\|_{Q_\omega}^2 + \|\Delta v_{\theta,k}\|_{r_{\Delta v}}^2 + \|\Delta \boldsymbol{f}_k\|_{R_{\Delta f}}^2 - \mu v_{\theta,k} \\ \text{s.t. } & \boldsymbol{x}_0 = \hat{\boldsymbol{x}}, \; \boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k), \; \underline{\boldsymbol{\omega}} \le \boldsymbol{\omega} \le \overline{\boldsymbol{\omega}}, \; \underline{\boldsymbol{f}} \le \boldsymbol{f} \le \overline{\boldsymbol{f}}, \; 0 \le v_\theta \le \overline{v_\theta}, \text{ rate bounds.} \end{aligned} \tag{4}$$

Gates. Instead of hard gate constraints (reported infeasible or too conservative), the contour weight depends on the reference point: $q_c(\boldsymbol{p}_d(\theta))$ is a sum of 3D Gaussians placed at the gates, so the drone may cut corners between gates but is pulled precisely through them.

Path and solver. Any continuously differentiable path can be used (min-snap, a time-optimal trajectory, or a point-mass time-optimal path); it is re-parameterized by arc length and represented by cubic splines. In every iteration the spline is evaluated at the progress predicted by the previous solution, and the problem is solved with a real-time iteration (RTI) scheme at 100 Hz.

4 Drone dynamics used in this project

4.1 Why not the paper's model

The paper's inputs are rotor thrusts. Our only actuator interface is $(\phi_c, \vartheta_c, \psi_c, f_c)$: the Crazyflie firmware runs its own attitude controller and motor mixing. Modelling the rigid body with torques would force us to also model that (unknown) onboard controller. Instead we use the model the simulator itself integrates for `physics = "so_rpy_rotor_drag"` (`drone_models/so_rpy_rotor_drag/model.py`), which is a *system-identified* model of the closed loop "Crazyflie + onboard attitude controller", fitted on the real `cf21B_500` drone. Using exactly this model has two advantages: in simulation there is no structural model mismatch, and on hardware it is the best model we have of the real closed-loop response.

4.2 Equations

Let $\boldsymbol{\eta} = [\phi, \vartheta, \psi]^\top$ be the roll, pitch and yaw angles in the extrinsic `xyz` convention (scipy), so the body-to-world rotation is

$$\boldsymbol{R}(\boldsymbol{\eta}) = \boldsymbol{R}_z(\psi)\boldsymbol{R}_y(\vartheta)\boldsymbol{R}_x(\phi) = \begin{bmatrix} c_\psi c_\vartheta & c_\psi s_\vartheta s_\phi - s_\psi c_\phi & c_\psi s_\vartheta c_\phi + s_\psi s_\phi \\ s_\psi c_\vartheta & s_\psi s_\vartheta s_\phi + c_\psi c_\phi & s_\psi s_\vartheta c_\phi - c_\psi s_\phi \\ -s_\vartheta & c_\vartheta s_\phi & c_\vartheta c_\phi \end{bmatrix}. \tag{5}$$

The model state is $(\boldsymbol{p}, \boldsymbol{\eta}, \boldsymbol{v}, \dot{\boldsymbol{\eta}}, f)$ where f is the thrust actually produced (the motors cannot change it instantly). With the command $(\boldsymbol{\eta}_c, f_c)$:

$$\dot{\boldsymbol{p}} = \boldsymbol{v}, \tag{6}$$

$$\dot{\boldsymbol{v}} = \frac{1}{m}\,\boldsymbol{R}(\boldsymbol{\eta}) \begin{bmatrix} 0 \\ 0 \\ c_0 + c_f f \end{bmatrix} + \boldsymbol{g} + \frac{1}{m}\boldsymbol{R}(\boldsymbol{\eta})\,\boldsymbol{D}\,\boldsymbol{R}(\boldsymbol{\eta})^\top \boldsymbol{v} + \boldsymbol{d}, \tag{7}$$

$$\ddot{\boldsymbol{\eta}} = \boldsymbol{a}_\eta \odot \boldsymbol{\eta} + \boldsymbol{b}_\eta \odot \dot{\boldsymbol{\eta}} + \boldsymbol{c}_\eta \odot \boldsymbol{\eta}_c, \tag{8}$$

$$\dot{f} = \frac{1}{\tau_f}\,(f_c - f). \tag{9}$$

$\boldsymbol{g} = [0, 0, -9.81]^\top$, $\boldsymbol{D}$ is a diagonal (negative) drag matrix and $\odot$ is the element-wise product. $\boldsymbol{d}$ is an external acceleration (wind, model error, battery sag); it is zero in the identified model and estimated online by us (section 9.2).

Euler-angle rates $\dot{\boldsymbol{\eta}}$ vs. body rates $\boldsymbol{\omega}$. These are different quantities. The body angular velocity $\boldsymbol{\omega}$ (what the IMU measures and what `obs["ang_vel"]` contains) is defined by $\dot{\boldsymbol{R}} = \boldsymbol{R}\,[\boldsymbol{\omega}]_\times$, while $\dot{\boldsymbol{\eta}}$ is the time derivative of the three angles. They are related by $\dot{\boldsymbol{\eta}} = \boldsymbol{W}(\boldsymbol{\eta})\,\boldsymbol{\omega}$ (section 9), and $\boldsymbol{W} = \boldsymbol{I}$ only at $\phi = \vartheta = 0$. We use $\dot{\boldsymbol{\eta}}$ as the state because the identified model eq. (8) is written in Euler angles: the simulator itself stores $\boldsymbol{\omega}$, converts it to $\dot{\boldsymbol{\eta}}$, evaluates eq. (8), and converts $\ddot{\boldsymbol{\eta}}$ back to $\dot{\boldsymbol{\omega}}$ (`rpy_rates_deriv2ang_vel_deriv`). Modelling in $(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})$ is therefore the same dynamics, written in the coordinates in which the fitted model is linear. The controller converts the measured $\boldsymbol{\omega}$ to $\dot{\boldsymbol{\eta}}$ once per step.

Interpretation. Equation (8) is three decoupled, linear, second-order systems: each Euler angle behaves like a damped spring driven by its command. With the identified parameters (table 1):

- roll/pitch: natural frequency $\omega_n = \sqrt{-a} = 13.7\,\text{rad/s}$, damping $\zeta = -b/(2\omega_n) = 0.46$ (about 19 % overshoot), and a *steady-state gain* $c/(-a) = 0.73$: a constant command of 0.7 rad settles at 0.51 rad. The MPC knows this and commands more than the tilt it actually needs;
- yaw: $\omega_n = 11.8\,\text{rad/s}$, $\zeta = 0.72$, gain 1.44. We always command $\psi_c = 0$;
- thrust: first-order lag with time constant $\tau_f = 80\,\text{ms}$. The produced thrust is $c_0 + c_f f$ with $c_f \approx 0.98$, so hovering needs $f_c = mg/c_f = 0.434\,\text{N}$;
- drag: $\boldsymbol{D}/m \approx -\text{diag}(0.50, 0.50, 0.54)\,\text{s}^{-1}$ in the body frame.

Table 1: Parameters of `so_rpy_rotor_drag` for `cf21B_500` (loaded at run time with `drone_models.core.load_params`).

Symbol	Value	Meaning
m	0.04338 kg	mass (randomized by ± 5 g in the configs)
c_0, c_f	0, 0.98023	thrust map (`acc_coef`, `cmd_f_coef`)
τ_f	0.07994 s	thrust time constant (`thrust_time_coef`)
$\boldsymbol{a}_\eta$	$[-188.99, -188.99, -138.31]$ s^{-2}	angle stiffness (`rpy_coef`)
$\boldsymbol{b}_\eta$	$[-12.78, -12.78, -16.85]$ s^{-1}	angle damping (`rpy_rates_coef`)
$\boldsymbol{c}_\eta$	$[138.08, 138.08, 198.52]$ s^{-2}	command gain (`cmd_rpy_coef`)
$\boldsymbol{D}$	diag$(-0.0215, -0.0215, -0.0236)$ kg/s	linear drag
f_c range	$[0.085, 0.8]$ N	env. limits; we use $[0.12, 0.72]$ N

Table 2: Paper vs. this project.

	Romero et al. [1]	This work
Vehicle	0.85 kg racing quad, thrust-to-weight ≈ 4	43 g Crazyflie, we allow ≈ 1.66
Actuation	4 rotor thrusts (MPC output: body rates + collective thrust to Betaflight)	attitude angles + collective thrust only
Attitude	quaternion, body rates, torques from rotor thrusts	Euler angles, identified 2nd-order response to angle commands
Thrust dynamics	rotor thrusts are states, inputs are their rates	1st-order lag f (state) + command f_c (state), input is $\dot{f}_c$
Model states	$\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{f}, \theta, v_\theta$ (17)	$\boldsymbol{p}, \boldsymbol{\eta}, \boldsymbol{v}, \dot{\boldsymbol{\eta}}, f, \phi_c, \vartheta_c, f_c, \theta, v_\theta$ (18)
Gates	Gaussian contour weight only	Gaussian weight + soft safe corridor + slow approach to unrevealed gates
Obstacles	none	soft keep-out cylinders + corridor
Model error	–	disturbance observer $\hat{\boldsymbol{d}}$
Track knowledge	known	nominal until within 0.7 m, then replanning

4.3 Augmented model for MPCC

Like the paper, we do not let the optimizer choose the actuator commands directly, but their *rates*. The commands become states and the inputs are

$$\boldsymbol{u} = [\,\dot{\phi}_c,\ \dot{\vartheta}_c,\ \dot{f}_c,\ a_\theta\,]^\top \in \mathbb{R}^4. \tag{10}$$

This (i) yields smooth commands (the rates are penalized and bounded), (ii) lets us bound the commands themselves as *state* constraints, and (iii) is the direct analogue of the paper's $\Delta\boldsymbol{f}$. The

progress follows a double integrator, $\dot{\theta} = v_\theta$, $\dot{v}_\theta = a_\theta$ (paper: Δv_θ). The full state is

$$\boldsymbol{x} = [\underbrace{\boldsymbol{p}}_{3}, \underbrace{\boldsymbol{\eta}}_{3}, \underbrace{\boldsymbol{v}}_{3}, \underbrace{\dot{\boldsymbol{\eta}}}_{3}, \underbrace{f}_{1}, \underbrace{\phi_c, \vartheta_c, f_c}_{3}, \underbrace{\theta, v_\theta}_{2}]^\top \in \mathbb{R}^{18}, \tag{11}$$

with $\dot{\boldsymbol{x}} = F(\boldsymbol{x}, \boldsymbol{u}; \boldsymbol{d})$ given by eqs. (6) to (9) (with $\boldsymbol{\eta}_c = [\phi_c, \vartheta_c, 0]$), $\frac{d}{dt}(\phi_c, \vartheta_c, f_c) = (u_1, u_2, u_3)$ and the progress dynamics.

Commands sent to the drone are obtained by integrating the optimal rates over one control period $\delta = 10\,\text{ms}$:

$$[\phi_c, \vartheta_c, f_c]_{\text{sent}} = [\phi_c, \vartheta_c, f_c]_0 + \delta\,[u^\star_{0,1}, u^\star_{0,2}, u^\star_{0,3}], \tag{12}$$

which keeps the command state consistent from one step to the next.

5 Reference path

MPCC needs only a geometric, arc-length parameterized, continuously differentiable path. We reuse the planning module of the repository (`ece484_fly/planning`).

5.1 Route

`geometry.route` produces the ordered waypoints for all remaining laps: the start (plus a take-off point 0.3 m above it when starting from the floor) and, for every gate, a *pre* point 0.3 m in front of the center, the center, and a *post* point 0.6 m behind it, all on the gate axis. Pre/post points force the path to cross the gate perpendicularly. When the next gate would be entered from its back side, or the next point lies in front of the gate just left, a *detour* point (left/right/over the gate, 0.7 m from the center) is inserted; the candidate with the largest clearance to all frames, poles and obstacles is chosen.

5.2 Geometry: cubic spline

`CubicSplinePlanner` fits a C^2 cubic spline through the route, with a trapezoidal time allocation that is refined until peak speed and acceleration match its limits. For MPCC the timing is irrelevant: it only shapes the curve. We chose this geometry after an experiment:

- The point-mass time-optimal planner (`PointMassPlanner`) gives the shortest paths, but its geometry is a sequence of bang-bang parabolas. After replanning from a moving state it produced loops and near-cusps (velocity passing through zero) and passes gates within a 30° cone instead of perpendicularly (min. clearance to frames 0.16 m). With this reference our first MPCC version crashed into a gate frame on the circle track.
- The cubic spline has the largest clearance (0.199 m of the possible 0.2 m; it crosses all gates on their axis), no cusps, and the lowest peak curvature on the circle track (10.6 vs. 13.8 m^{-1} for point-mass).
- The min-snap planner used to break when replanning from a fast moving state (path length 465 m on circle in our test). The cause was the time allocation: uniformly stretching the segment times scales every velocity by $1/k$ *except* the fixed start velocity, so with $\|\boldsymbol{v}_0\| > v_{\max}$ the speed limit could never be met and the times grew without bound; a very long first segment with a fixed $\boldsymbol{v}_0$ makes a degree-7 polynomial travel far away before it turns back. The first segment is now checked against $\max(v_{\max}, \|\boldsymbol{v}_0\|)$ and all later ones against $v_{\max}$, which removes the loops. The cubic spline remains our default because of its clearance.

This matches the paper, which also obtains good results from a non-time-optimal (min-snap) path: the time optimality comes from the MPC, not from the reference.

5.3 Arc-length parameterization

`ArcLengthPath` samples the trajectory densely, removes duplicate points, resamples at equal chord length (2000 points), fits a cubic spline, measures its true arc length numerically, and refits on that arc length. The result is a cubic spline $\boldsymbol{p}_d(s)$, $s \in [0, L]$, with $\|\boldsymbol{p}_d'(s)\| \approx 1$, so $\boldsymbol{t}(s) = \boldsymbol{p}_d'(s)$ is the unit tangent and $\boldsymbol{p}_d''(s)$ the curvature vector (magnitude $\kappa = 1/\text{radius}$). From now on $\theta \equiv s$.

5.4 Gate-aware contour weight

`MPCCReference` adds the paper's dynamic weight allocation:

$$q_c(\theta) = q_{\text{nom}} + q_{\text{gate}} \sum_j \exp\Big(- \frac{\|\boldsymbol{p}_d(\theta) - \boldsymbol{g}_j\|^2}{2\sigma^2} \Big), \tag{13}$$

with $q_{\text{nom}} = 60$, $q_{\text{gate}} = 400$, $\sigma = 0.25\,\text{m}$ and $\boldsymbol{g}_j$ the gate centers (fig. 2). Like in the paper, the weight is a function of the *reference* point, so it is attained regardless of where the drone is. The paper uses normalized Gaussians; we add an explicit floor q_{nom} (their q_{nom} in Fig. 3) and scale the peak directly.

5.5 Safe corridor radius

Not in the paper, but important for safety: with a low q_c between gates MPCC cuts corners, and on our tracks the corners are next to gate frames and poles. We therefore compute, once per plan, the clearance of every path point to the collision geometry (gate frames as solid bands between the 0.4 m opening and the 0.72 m outer edge, gate poles, obstacle poles; all sampled every 4 cm, `geometry.collision_points`) and define the corridor radius

$$r(\theta) = \text{clip}\,\big(\,\text{clr}(\boldsymbol{p}_d(\theta)) - r_{\text{drone}},\ r_{\min},\ r_{\max}\big), \qquad r_{\text{drone}} = 0.13,\ r_{\min} = 0.05,\ r_{\max} = 0.25\ \text{m}. \tag{14}$$

If $\|\boldsymbol{p} - \boldsymbol{p}_d(\theta)\| \le r(\theta)$, the drone's ball of radius r_{drone} cannot touch any obstacle, since the ball around $\boldsymbol{p}_d(\theta)$ of radius clr is free. Inside a gate clr $\approx 0.2\,\text{m}$, so $r \approx 0.07\,\text{m}$. We use $\|\boldsymbol{e}^c\| \le r(\theta)$ as a soft constraint (section 7.3). With $\|\boldsymbol{e}^l\| \approx 0$ (proposition 1), $\|\boldsymbol{e}^c\| \approx \|\boldsymbol{p} - \boldsymbol{p}_d(\theta)\|$, which makes the argument apply.

Choice of r_{drone}. The simulator's collision geometry of the drone is a box with half extents $0.07 \times 0.07 \times 0.02\,\text{m}$, i.e. a horizontal half-diagonal of 0.099 m. Our first value $r_{\text{drone}} = 0.10\,\text{m}$ therefore left no margin at all inside a gate, and since the corridor is a *soft* constraint, small violations at speed ended in a frame hit. On the provided tracks this went unnoticed (10/10, but with gate offsets up to 0.12 m), on randomly generated tracks every MPCC failure was a gate-frame hit. With $r_{\text{drone}} = 0.13\,\text{m}$ (3 cm margin) the success rate on 60 random tracks went from 53/60 to 59/60 (nominal gates) and from 48/60 to 52/60 (shifted gates), while the provided tracks stay at 90/90 with unchanged lap times (`scripts/diagnose_mpcc.py`, `docs/distill_report.md`).

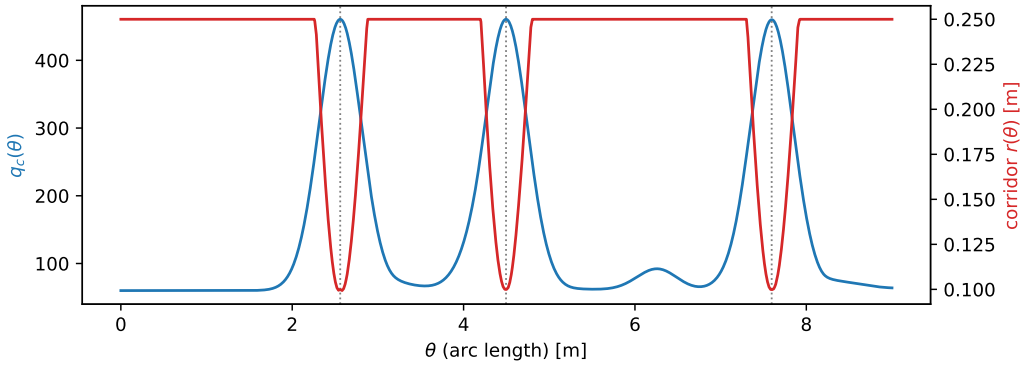


Figure 2: Contour weight $q_c(\theta)$ (blue) and safe-corridor radius $r(\theta)$ (red) along the first lap of the circle track. Dotted lines: arc length at which the path passes a gate. At every gate the weight peaks and the corridor narrows to $\approx 0.1\,\mathrm{m}$.

5.6 Replanning

The planner runs at start-up with the nominal track. Whenever an observed gate or obstacle pose changes (it was revealed), we replan the remaining laps. The new path starts at the current point of the old path, $\boldsymbol{p}_d(\theta)$, with start velocity $0.5\,\boldsymbol{t}(\theta)$. Starting on the old path and tangent to it makes the new path continue the old one smoothly; the small start speed matters because the spline is time-parameterized: a fast start (we first used the drone's 2.4 m/s) made it swing 1 m wide, out of the flight volume (partly the time-allocation problem described in section 5.2, since fixed; we keep the validated 0.5 m/s). After replanning, θ is re-projected onto the new path and the previously predicted progress increments are kept as a warm start. Planning takes a few tens of milliseconds.

6 Contouring and lag errors

6.1 Definitions

Let θ be the progress variable (the controller's guess of where the drone is along the path) and

$$\boldsymbol{e}(\theta) = \boldsymbol{p} - \boldsymbol{p}_d(\theta). \tag{15}$$

Since θ is arc length, $\boldsymbol{t}(\theta) = \boldsymbol{p}_d'(\theta)$ has unit norm. The error is split into its component along the tangent (the *lag error*) and the rest (the *contour error*):

$$e^l(\theta) = \boldsymbol{t}(\theta)^\top \boldsymbol{e}(\theta), \qquad \boldsymbol{e}^l = e^l\,\boldsymbol{t}, \tag{16}$$

$$\boldsymbol{e}^c(\theta) = \boldsymbol{e}(\theta) - e^l(\theta)\,\boldsymbol{t}(\theta) = (\boldsymbol{I} - \boldsymbol{t}\boldsymbol{t}^\top)\,\boldsymbol{e}(\theta). \tag{17}$$

$e^l > 0$ means the drone is *ahead* of the reference point, $e^l < 0$ behind it. $\boldsymbol{e}^c$ lies in the plane $\pi(\theta)$ normal to the path at $\boldsymbol{p}_d(\theta)$. The cost uses $\|\boldsymbol{e}^l\|_{q_l}^2 = q_l (e^l)^2$ and $\|\boldsymbol{e}^c\|_{q_c}^2 = q_c\,\boldsymbol{e}^{c\top}\boldsymbol{e}^c$ (paper Eqs. 10–11).

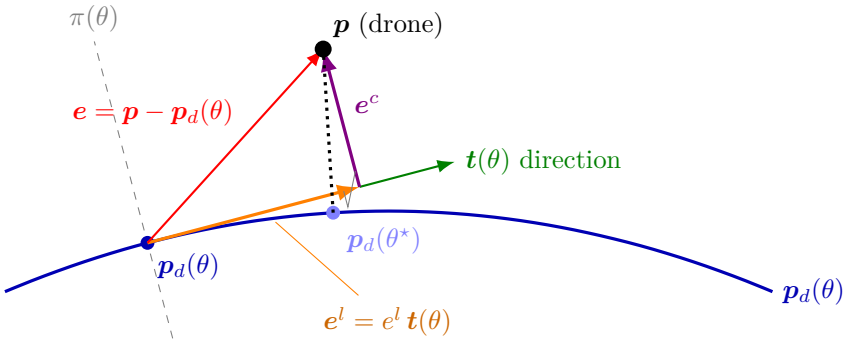


Figure 3: Error geometry. The true contour error (dotted) is the distance to the closest path point $\boldsymbol{p}_d(\theta^\star)$, which requires solving a minimization. MPCC instead uses the current progress θ and splits $\boldsymbol{e} = \boldsymbol{p} - \boldsymbol{p}_d(\theta)$ into a lag part along the tangent and a contour part in the normal plane. As $e^l \to 0$, $\theta \to \theta^\star$ and the approximation becomes exact.

Proposition 1 ([1], Prop. 1)**.** *If $e^l(\theta) = 0$, then $\boldsymbol{p}$ lies in the normal plane $\pi(\theta)$, so θ is a stationary point of $\|\boldsymbol{p} - \boldsymbol{p}_d(\cdot)\|^2$ (locally the closest point, $\theta = \theta^\star$), and $\|\boldsymbol{e}^c(\theta)\|$ equals the true contour error.*

Proof. $\frac{d}{d\theta}\frac{1}{2}\|\boldsymbol{p} - \boldsymbol{p}_d(\theta)\|^2 = -\boldsymbol{t}(\theta)^\top \boldsymbol{e}(\theta) = -e^l(\theta)$, which vanishes iff $e^l = 0$. □

Therefore q_l is chosen large ($q_l = 300$): it keeps θ synchronized with the drone and makes $\boldsymbol{e}^c$ a faithful contour error.

6.2 Evaluating the path inside the solver

The solver works with symbolic CasADi expressions and cannot call a scipy spline. We follow the paper's Algorithm 1 (spline evaluated at the progress predicted by the previous solution), and pass, *separately for every shooting node k*, a local second-order expansion of the path around the predicted progress $\bar{\theta}_k$:

$$\boldsymbol{p}_d(\theta_k) \approx \boldsymbol{p}_d(\bar{\theta}_k) + \boldsymbol{t}(\bar{\theta}_k)\,\delta_k + \tfrac{1}{2}\,\boldsymbol{p}_d''(\bar{\theta}_k)\,\delta_k^2, \qquad \delta_k = \theta_k - \bar{\theta}_k, \tag{18}$$

$$\boldsymbol{t}(\theta_k) \approx \boldsymbol{t}(\bar{\theta}_k) + \boldsymbol{p}_d''(\bar{\theta}_k)\,\delta_k. \tag{19}$$

Only ten numbers per node are needed ($\bar{\theta}_k$, $\boldsymbol{p}_d$, $\boldsymbol{t}$, $\boldsymbol{p}_d''$), and the expression is exact up to $O(\delta^3)$. Because $\bar{\theta}_k$ comes from the previous solution shifted by one control period, $|\delta_k|$ is of the order of a few millimetres in steady flight, so the error of the expansion is negligible. $\bar{\theta}_k$ beyond the end of the path use a straight extension ($\boldsymbol{p}_d'' = 0$).

7 The optimal control problem

7.1 Discretization

Horizon $T = 1.0\,\mathrm{s}$ with $N = 25$ intervals of $\Delta t = 40\,\mathrm{ms}$. One second covers about 2.8 m at top speed, which is enough to see the next gate and the next turn. The dynamics are discretized with one step of the explicit Runge–Kutta method of order 4 per interval, $\boldsymbol{x}_{k+1} = \Phi_{\Delta t}(\boldsymbol{x}_k, \boldsymbol{u}_k)$. RK4 is stable for $\lambda\Delta t$ up to about 2.8; the fastest mode here is the attitude response with $|\lambda| \approx \omega_n = 13.7\,\mathrm{s}^{-1}$, i.e. $\lambda\Delta t \approx 0.55$.

7.2 Cost

For every stage $k = 0, \ldots, N-1$ (the terminal stage omits the input terms):

$$\ell_k = q_l\,(e_k^l)^2 + q_c(\bar{\theta}_k)\,\|\boldsymbol{e}_k^c\|^2 + w_\eta\|\dot{\boldsymbol{\eta}}_k\|^2 + w_{\mathrm{tilt}}(\phi_{c,k}^2 + \vartheta_{c,k}^2) + r_\eta(u_1^2 + u_2^2) + r_f u_3^2 + r_a u_4^2 - \mu\,v_{\theta,k}. \tag{20}$$

- $q_l(e^l)^2$, $q_c\|\boldsymbol{e}^c\|^2$, $-\mu v_\theta$: the MPCC core (paper Eq. 7). q_c is evaluated at $\bar{\theta}_k$ (a parameter) instead of at the decision variable θ_k, which is the same approximation as the paper's spline evaluation at the predicted progress.
- $w_\eta\|\dot{\boldsymbol{\eta}}\|^2$: replaces the paper's body-rate cost $\|\boldsymbol{\omega}\|^2_{Q_\omega}$ (it keeps rotations calm and is needed for a well-posed problem).
- w_{tilt}: small penalty on the tilt commands (smooth, less aggressive flight).
- r_η, r_f: command-rate penalties (paper: $R_{\Delta f}$); r_a: progress acceleration (paper: $r_{\Delta v}$). Without them the solution is noisy.

Weights: $q_l = 300$, $\mu = 12$, $w_\eta = 0.08$, $w_{\text{tilt}} = 0.5$, $r_\eta = 0.4$, $r_f = 4$, $r_a = 0.02$. The ratio μ/q_c sets the speed/accuracy trade-off.

7.3 Constraints

Hard bounds (stages $1..N$): $|\phi_c|, |\vartheta_c| \le 0.7\,\text{rad}$ (steady tilt $\le 0.51\,\text{rad}$), $0.12 \le f_c \le 0.72\,\text{N}$, $0 \le v_\theta \le \bar{v}_k$ (see below); inputs: $|\dot{\phi}_c|, |\dot{\vartheta}_c| \le 8\,\text{rad/s}$, $|\dot{f}_c| \le 6\,\text{N/s}$, $|a_\theta| \le 12\,\text{m/s}^2$. $v_\theta \ge 0$ prevents going backwards along the path (paper: non-reversal).

Soft safe corridor. $h_1(\boldsymbol{x}) = \|\boldsymbol{e}^c\|^2 - r(\bar{\theta}_k)^2 \le 0$, section 5.5.

Soft obstacle keep-out. For the two obstacles closest to the drone, $h_{2,3}(\boldsymbol{x}) = \|\boldsymbol{p}_{xy} - \boldsymbol{o}_{xy}\|^2 \ge r_{\text{obs}}^2$, $r_{\text{obs}} = 0.22\,\text{m}$ (infinite vertical cylinders around the poles).

Soft constraints are relaxed with slack variables $\boldsymbol{s} \ge 0$, $h(\boldsymbol{x}) \le \bar{h} + s$, penalized by $z\,s + Z\,s^2$ with $z = 10^4$, $Z = 10^5$. The linear term makes this an *exact penalty*: if the hard-constrained problem is feasible and z exceeds the corresponding Lagrange multiplier, the solution satisfies the constraint exactly; if not (e.g. after a disturbance), the problem stays feasible and the drone returns as fast as it can. This is why we use soft instead of hard state constraints: a hard constraint that is violated at $\boldsymbol{x}_0$ makes the QP infeasible.

Slow approach to unrevealed gates. The true pose of a gate is known only within 0.7 m. At 2.8 m/s that leaves 0.25 s to correct a shift of up to 0.15 m, which is not enough: in our first stress test 4 of 12 runs crashed into the first shifted gate. For every stage we therefore set

$$\bar{v}_k = \min\Big(v_{\max},\ \max\big(v_k^{\text{gate}},\ v_{\theta,0} - 0.8\,a_{\theta,\max}\,k\,\Delta t\big)\Big), \tag{21}$$

$$v_k^{\text{gate}} = \begin{cases} 1.5\ \text{m/s} & \text{if } \boldsymbol{p}_d(\bar{\theta}_k) \text{ is within } 1\,\text{m of an unrevealed gate,} \\ v_{\max} = 2.8\ \text{m/s} & \text{otherwise.} \end{cases} \tag{22}$$

The second term makes the cap reachable: without it the cap could appear closer than the progress can decelerate, the QP became infeasible and HPIPM returned NaN (we observed this). Gates stay revealed once seen, so only the first lap is slowed down.

Slowing down under large disturbances. A large disturbance estimate (section 9.2) means both that less control authority is left (e.g. battery sag eats the thrust margin) and that the model is less accurate (e.g. wind drag grows with airspeed, which a constant $\hat{\boldsymbol{d}}$ only tracks with a delay). We therefore scale $v_{\max}$ by $\max\left(1 - 0.15\,(\|\hat{\boldsymbol{d}}\| - 1),\ 0.6\right)$ for $\|\hat{\boldsymbol{d}}\| > 1\,\text{m/s}^2$. With strong wind, gusts and battery sag on the circle track, this raised the success rate from 7/10 to 10/10; without disturbances it has no effect.

7.4 Full problem

With $\boldsymbol{P}_k$ the per-stage parameters (path expansion, q_c, r^2, $\hat{\boldsymbol{d}}$, obstacles), the NLP solved at every control step is

$$\begin{aligned} \min_{\boldsymbol{x}_{0:N},\,\boldsymbol{u}_{0:N-1},\,\boldsymbol{s}} \quad & \sum_{k=0}^{N-1} \ell_k(\boldsymbol{x}_k,\boldsymbol{u}_k;\boldsymbol{P}_k) + \ell_N(\boldsymbol{x}_N;\boldsymbol{P}_N) + \sum_{k=1}^{N} \left(z^\top \boldsymbol{s}_k + \boldsymbol{s}_k^\top Z \boldsymbol{s}_k\right) \\ \text{s.t.} \quad & \boldsymbol{x}_0 = \hat{\boldsymbol{x}}_0, \qquad \boldsymbol{x}_{k+1} = \Phi_{\Delta t}(\boldsymbol{x}_k,\boldsymbol{u}_k;\boldsymbol{P}_k), \\ & \underline{\boldsymbol{x}}_k \le \boldsymbol{x}_k \le \overline{\boldsymbol{x}}_k, \quad \underline{\boldsymbol{u}} \le \boldsymbol{u}_k \le \overline{\boldsymbol{u}}, \\ & \underline{\boldsymbol{h}} - \boldsymbol{s}_k \le \boldsymbol{h}(\boldsymbol{x}_k;\boldsymbol{P}_k) \le \overline{\boldsymbol{h}} + \boldsymbol{s}_k, \quad \boldsymbol{s}_k \ge 0. \end{aligned} \tag{23}$$

8 Solving the problem

8.1 Theory

Direct multiple shooting. Equation (23) keeps all $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$ as variables and the integration Φ as equality constraints. This makes the problem large but very sparse (block-banded), and it is much better conditioned than single shooting for unstable or fast dynamics. Let $\boldsymbol{w} = (\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{x}_N)$; then eq. (23) reads $\min_{\boldsymbol{w}} J(\boldsymbol{w})$ s.t. $\boldsymbol{c}(\boldsymbol{w}) = 0$, $\boldsymbol{g}(\boldsymbol{w}) \le 0$.

Sequential quadratic programming (SQP). Starting from a guess $\boldsymbol{w}^i$, SQP solves the quadratic program obtained by a second-order model of the Lagrangian and a linearization of the constraints,

$$\min_{\Delta \boldsymbol{w}} \ \tfrac{1}{2}\Delta \boldsymbol{w}^\top \boldsymbol{H}^i \Delta \boldsymbol{w} + \nabla J^\top \Delta \boldsymbol{w} \quad \text{s.t.} \quad \boldsymbol{c} + \nabla \boldsymbol{c}^\top \Delta \boldsymbol{w} = 0,\ \boldsymbol{g} + \nabla \boldsymbol{g}^\top \Delta \boldsymbol{w} \le 0, \tag{24}$$

and sets $\boldsymbol{w}^{i+1} = \boldsymbol{w}^i + \Delta \boldsymbol{w}$. For the dynamics this gives $\Delta \boldsymbol{x}_{k+1} = \boldsymbol{A}_k \Delta \boldsymbol{x}_k + \boldsymbol{B}_k \Delta \boldsymbol{u}_k + \boldsymbol{b}_k$ with the sensitivities $\boldsymbol{A}_k = \partial \Phi / \partial \boldsymbol{x}$, $\boldsymbol{B}_k = \partial \Phi / \partial \boldsymbol{u}$ computed by algorithmic differentiation through the RK4 step.

Gauss–Newton Hessian. Except for the linear progress term, the cost is a sum of squares, $\ell = \|\boldsymbol{y}(\boldsymbol{x},\boldsymbol{u})\|^2 - \mu v_\theta$, with the residual

$$\boldsymbol{y} = \left[\sqrt{q_l}\, e^l,\ \sqrt{q_c}\, \boldsymbol{e}^c,\ \sqrt{w_\eta}\, \dot{\boldsymbol{\eta}},\ \sqrt{w_{\text{tilt}}}(\phi_c, \vartheta_c),\ \sqrt{r_\eta}(u_1,u_2),\ \sqrt{r_f} u_3,\ \sqrt{r_a} u_4\right]. \tag{25}$$

We use the generalized Gauss–Newton Hessian $\boldsymbol{H}_k = 2\,\boldsymbol{J}_k^\top \boldsymbol{J}_k$, $\boldsymbol{J}_k = \partial \boldsymbol{y} / \partial(\boldsymbol{u},\boldsymbol{x})$, and neglect the second derivatives of the dynamics. It is positive semidefinite by construction (the exact Hessian of $(e^l)^2$ and $\|\boldsymbol{e}^c\|^2$ is indefinite in general, because the path is curved), it needs only first derivatives, and it is accurate when the residuals are small, which is the regime we fly in. The linear term $-\mu v_\theta$ has zero Hessian, so nothing is lost.

Real-time iteration (RTI). Instead of iterating SQP to convergence at every sampling time, RTI performs *one* SQP iteration per control step, started from the previous solution shifted in time. Since the problem changes little between two steps (10 ms), one Newton-type step from a good guess tracks the optimal solution closely, and the iterate converges along with the system. The iteration is split into a *preparation* phase (linearization, Hessian, condensing, all independent of the new state) and a *feedback* phase (solve the QP once $\hat{\boldsymbol{x}}_0$ is known), which minimizes the delay between measurement and command.

The QP. The QP has optimal-control structure: stage-wise costs, dynamics coupling only neighbouring stages, and stage-wise inequalities. HPIPM solves it with a primal–dual interior point method in which each Newton system is solved by a structure-exploiting Riccati recursion, with cost linear in N instead of cubic. *Partial condensing* first eliminates the states inside blocks of stages, trading horizon length for block size, which suits the BLASFEO dense linear algebra kernels. Soft constraints only add diagonal blocks for the slacks.

8.2 Implementation with acados

acados [2] generates C code for the model, cost and constraint functions from CasADi expressions, and links them with its NLP layer (SQP/RTI), integrators and HPIPM. Our setup (`build_solver`):

Option	Value
`nlp_solver_type`	`SQP_RTI`
`qp_solver`	`PARTIAL_CONDENSING_HPIPM`, mode `BALANCE`, $\leq$50 iterations
`integrator_type`	`ERK`, 4 stages (RK4), 1 step per interval
`cost_type`	`EXTERNAL` with `cost_expr_ext_cost_custom_hess` $= 2\boldsymbol{J}^\top\boldsymbol{J}$
`hessian_approx`	`GAUSS_NEWTON` (uses the custom cost Hessian)
`levenberg_marquardt`	10^{-3} (regularization of the Hessian)
Constraints	`idxbx` (commands, v_θ), `idxbu`, nonlinear `con_h_expr` with `idxsh`
Parameters	19 per stage: $\bar{\theta}$, $\boldsymbol{p}_d$, $\boldsymbol{t}$, $\boldsymbol{p}_d''$, q_c, r^2, $\hat{\boldsymbol{d}}$, 2 obstacles

- We use an `EXTERNAL` cost instead of `NONLINEAR_LS` because the progress reward $-\mu v_\theta$ is linear, not a least-squares term. The custom Hessian keeps the Gauss–Newton structure.
- The inactive side of each soft constraint is set to `ACADOS_INFTY`, so acados masks it. (An earlier version used a large finite bound and an unused obstacle 30 m away; the resulting huge slack penalty made the QP produce NaN.)
- Code generation and compilation happen once (about 30 s) and are cached in `ece484_fly/control/_mpcc_build/<hash>`, where the hash covers all build-time settings and the source of the model and solver functions; a file lock makes parallel runs safe.
- At run time we only call `set(k, "p", ...)`, `set(k, "x"/"u", ...)` for the warm start, `set(0, "lbx"/"ubx", x0)`, `constraints_set(k, "ubx", ...)` for the speed caps, and `solve()`.

9 Closing the loop

9.1 From the observation to the model state

- $\boldsymbol{\eta}$ from the quaternion: `Rotation.from_quat(q).as_euler("xyz")`.
- $\dot{\boldsymbol{\eta}}$ from the body rates: $\dot{\boldsymbol{\eta}} = \boldsymbol{W}(\boldsymbol{\eta})\,\boldsymbol{\omega}$ with

$$\boldsymbol{W} = \begin{bmatrix} 1 & s_\phi t_\vartheta & c_\phi t_\vartheta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi/c_\vartheta & c_\phi/c_\vartheta \end{bmatrix} \tag{26}$$

  (the same map the simulator uses, `rotation.ang_vel2rpy_rates`).
- Thrust f: not observable. The motors start at rest ($f = 0$) and the commands are ours, so we propagate eq. (9) exactly: $f \leftarrow f + (1 - e^{-\delta/\tau_f})(f_c - f)$.
- Commands $(\phi_c, \vartheta_c, f_c)$: the last values sent.

9.2 Disturbance observer

The configs randomize the mass by ± 5 g (± 11 % of the weight), the stress tests add wind, gusts and battery sag, and real hardware always differs from the model. A constant acceleration error shifts the whole prediction. We estimate it with a first-order observer

$$\hat{\boldsymbol{d}} \leftarrow (1-\alpha)\hat{\boldsymbol{d}} + \alpha\Big(\frac{\boldsymbol{v}_t - \boldsymbol{v}_{t-1}}{\delta} - \boldsymbol{a}_{\text{model}}(\boldsymbol{\eta}_{t-1}, \boldsymbol{v}_{t-1}, f_{t-1})\Big), \qquad \alpha = 0.04, \tag{27}$$

(time constant ≈ 0.25 s, clipped to ± 3 m/s^2, disabled on the ground), and feed $\hat{\boldsymbol{d}}$ into eq. (7) as a parameter. This is offset-free MPC: a constant bias is absorbed without steady-state error. In the runs $\hat{d}_z$ settles at a few tenths of m/s^2, matching the mass randomization.

9.3 Progress state

As in the paper, θ is an internal state of the controller: at each step we take the progress predicted by the last solution, $\theta_0 \leftarrow \theta_0^\star + v_{\theta,0}^\star \delta$ (same for v_θ). We only re-project the position onto the path (a grid search followed by Newton steps on $\boldsymbol{t}^\top \boldsymbol{e} = 0$) at the start, after replanning, and when the lag error exceeds 0.6 m; the search window $[\theta - 0.3, \theta + 0.9]$ is biased forward so that θ cannot jump back onto a part of the path that was already flown. An early version re-projected at every step. That trapped the drone: when it cut inside a tight loop, the closest path point stayed on the branch it had already left, θ was pulled back at every step and the drone stalled in mid-air.

Stall recovery. MPCC is a non-convex problem. We found one more local minimum: when a replanned path has a hook (the route's post point behind gate j followed by a gate $j+1$ that was revealed closer than its nominal pose) and the drone has cut inside it, advancing θ first moves the reference point *away* from the drone. The RTI then converged to "wait one stage, then go", re-planned the same at every step, and the drone hovered with $v_\theta = 0$. If the drone is airborne, slower than 0.2 m/s and $v_\theta < 0.05$ for 0.3 s, we therefore move θ to the closest path point in $[\theta + 0.1, \theta + 1.5]$, which skips the hook.

9.4 Warm start, failure handling

The previous solution is interpolated in time by one control period ($\delta = 10\,\text{ms} < \Delta t$) and used as the initial guess; its θ-trajectory, shifted so it starts at θ_0, gives the expansion points $\bar{\theta}_k$. If the solver reports an error or returns non-finite values, we fall back to the shifted previous plan for up to 20 steps, then to a level attitude with hover thrust. In the final evaluation this path was never taken (table 3).

Algorithm 1 One control step (100 Hz)

if a gate/obstacle pose changed **then**
    replan path, corridor $r(\theta)$, weights $q_c(\theta)$
end if
$\hat{\boldsymbol{x}}_0 \leftarrow$ (pos, Euler, vel, Euler rates, thrust estimate, last commands)
update disturbance observer $\hat{\boldsymbol{d}}$
$\theta_0, v_{\theta,0} \leftarrow$ propagated from the last solution (re-project if $|e^l| > 0.6$ m; stall recovery)
warm start $(\boldsymbol{x}_k, \boldsymbol{u}_k) \leftarrow$ last solution shifted by δ; $\bar{\theta}_k \leftarrow \theta$ of the warm start
for every stage k: parameters $\boldsymbol{P}_k$ from $\boldsymbol{p}_d, \boldsymbol{t}, \boldsymbol{p}_d'', q_c, r$ at $\bar{\theta}_k$, $\hat{\boldsymbol{d}}$, closest obstacles; speed cap $\bar{v}_k$ (unrevealed gates, $\|\hat{\boldsymbol{d}}\|$)
one SQP-RTI iteration (acados)
$[\phi_c, \vartheta_c, f_c] \leftarrow [\phi_c, \vartheta_c, f_c] + \delta\, \boldsymbol{u}_{0,1:3}^\star$; clip to bounds
propagate the thrust estimate; **return** $[\phi_c, \vartheta_c, 0, f_c]$

10 Visualization

`scripts/sim.py` now calls `draw_controller_debug(env, controller)` (`ece484_fly/utils/render.py`) before every `env.render()`. Any controller may implement `debug_lines()` and `debug_points()`; the helper adds them as capsule and sphere markers to the MuJoCo viewer. The MPCC controller draws

- the **planned path ahead** of the drone (next 6 m from the current progress θ, light blue, semi-transparent), which changes when the path is replanned; and
- the **MPC prediction**: the 26 predicted positions $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_N$ over the next second (orange spheres).

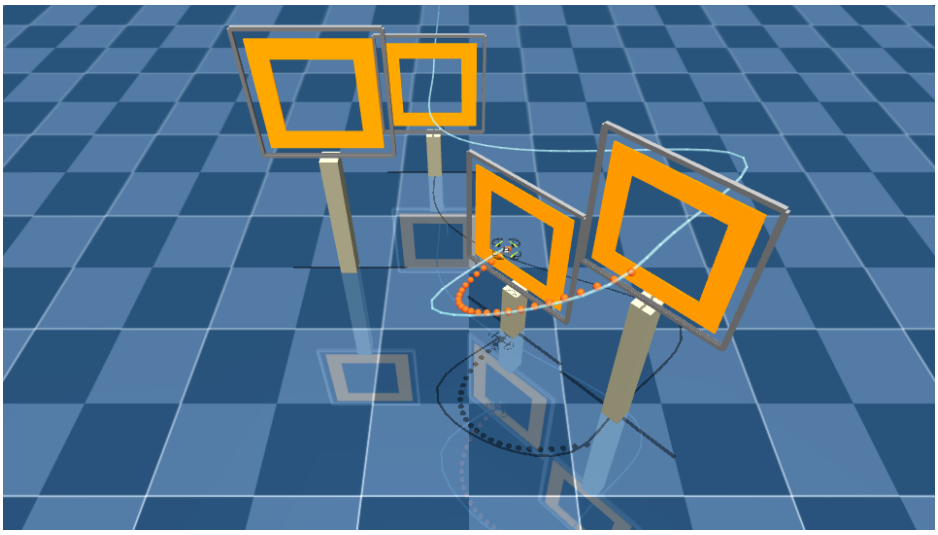


Figure 4: Simulator view with the overlays: planned path ahead (light blue) and the 1 s MPC prediction (orange).

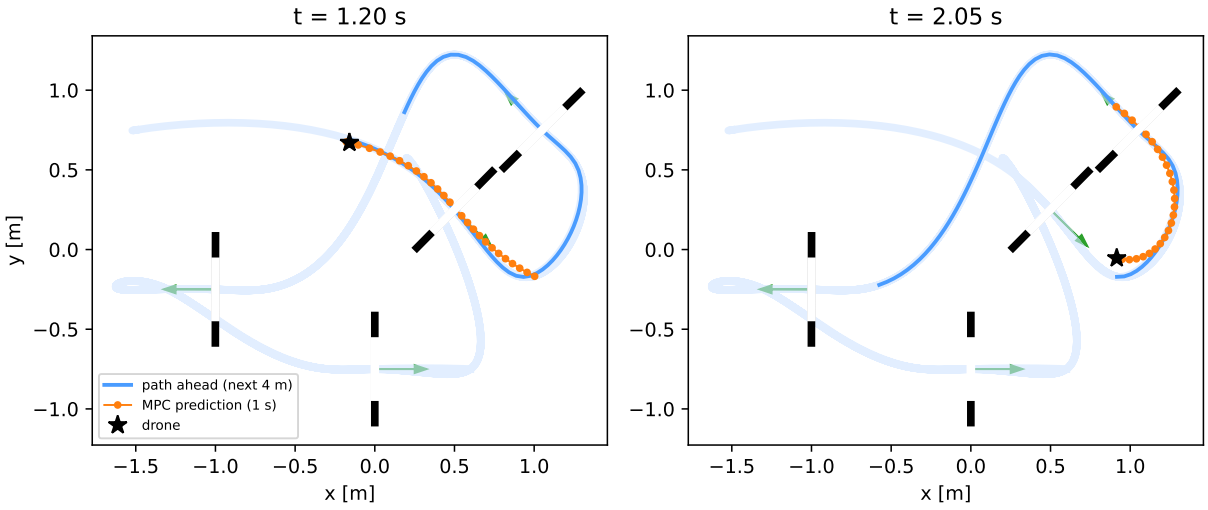


Figure 5: Two snapshots (top view): path ahead (blue), predicted positions (orange) and the drone (star). The prediction leaves the path between gates to cut corners and rejoins it inside gates.

11 Results

Setup. Simulation with the configs' default disturbances (action noise, random force noise of ±0.1 N, mass and inertia randomization, random start pose), `control_mode = "attitude"`, 5 laps. Besides the three tracks, we test: *shift*: gates moved by up to ±0.15 m (xy), ±0.1 m (z), ±0.15 rad (yaw), and obstacles by ±0.15 m, which is only revealed within 0.7 m; *wind*: 1.1 m/s steady wind, 0.8 m/s gusts and 10 % battery sag after 25 s. 10 episodes per configuration, different seeds.

Table 3: Simulation results, 10 episodes per configuration (5 laps each). Time: mean ± std. and best over the successful episodes. Gate offset: largest distance between the drone and the gate center one step after a gate was passed (over all gates and episodes; the opening half-width is 0.2 m). Solve: acados time per control step (mean / worst case). Fails: solver failures.

Config	Success	Time [s]	Best [s]	Gate offset [m]	Solve [ms]	Max solve [ms]	Fails
`circle`	10/10	25.10 ± 0.31	24.70	0.109	1.29	6.65	0
`uturn`	10/10	17.01 ± 0.18	16.82	0.079	1.31	3.32	0
`lemniscate`	10/10	25.83 ± 0.44	25.36	0.108	1.25	3.72	0
`circle_shift`	10/10	24.86 ± 0.76	23.91	0.100	1.26	6.34	0
`uturn_shift`	10/10	17.23 ± 0.39	16.67	0.105	1.24	5.74	0
`lemniscate_shift`	10/10	26.14 ± 0.64	25.14	0.119	1.23	4.89	0
`circle_wind`	10/10	27.03 ± 0.59	26.44	0.120	1.23	6.71	0
`uturn_wind`	10/10	18.53 ± 0.57	17.96	0.098	1.20	5.83	0
`lemniscate_wind`	10/10	28.43 ± 1.08	27.49	0.115	1.19	4.67	0

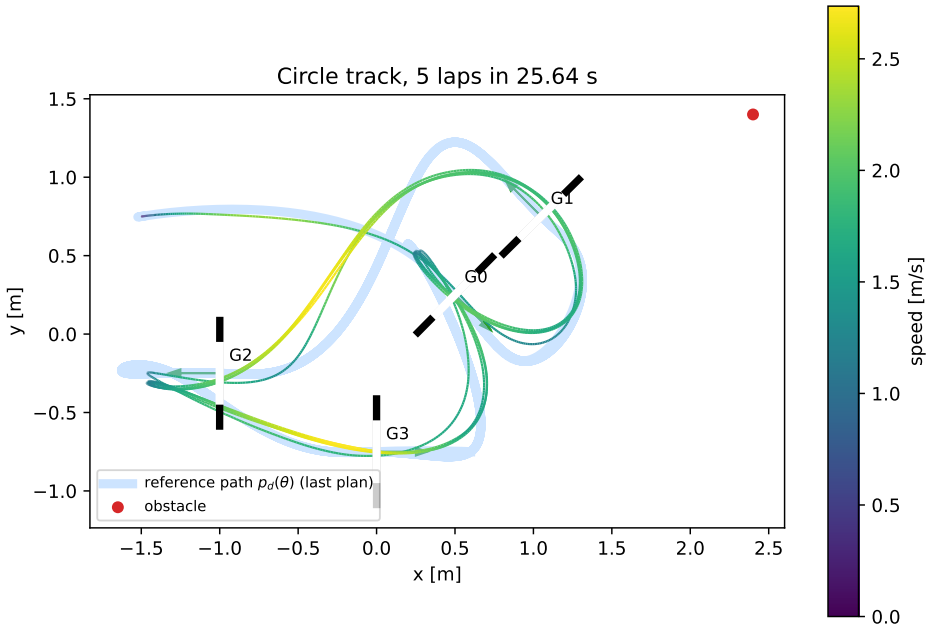


Figure 6: Five laps on the circle track (top view), colored by speed. Gates are drawn as black frames with the opening in white and the pass direction in green.

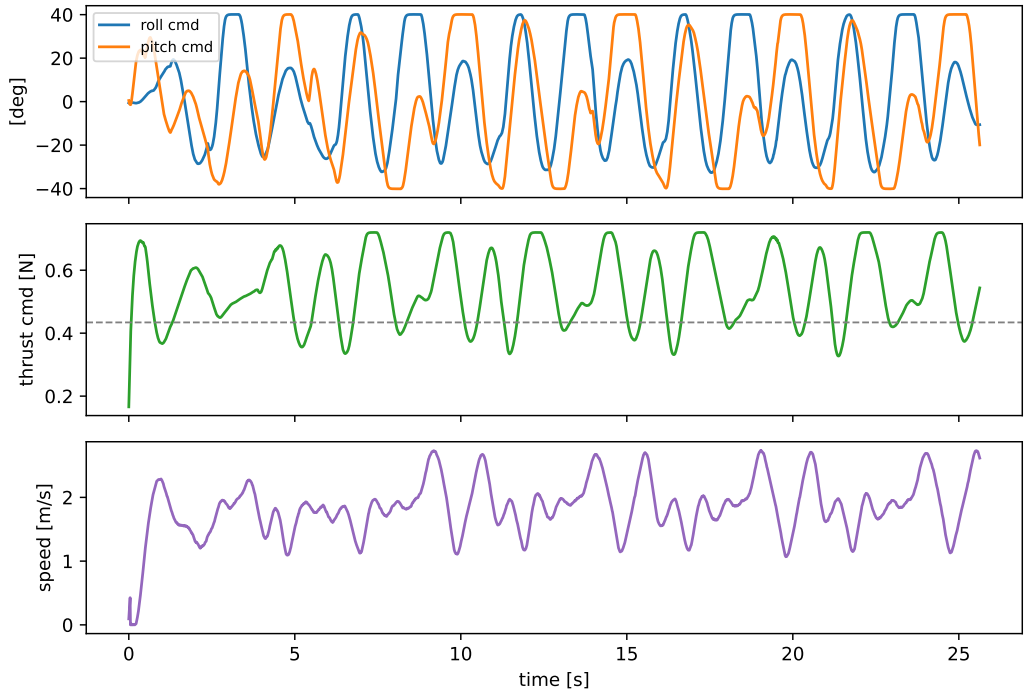


Figure 7: Commands and speed for the same flight. The commands are smooth because the optimizer chooses their rates.

Observations.

- Gates are passed close to their center even at 2.5–3 m/s, thanks to the Gaussian contour weight and the corridor. Between gates the drone deviates from the reference to shorten the turns, which is the intended MPCC behaviour.
- The first lap is slower than later laps: the speed cap near unrevealed gates trades about a second for robustness to shifted gates.
- The mean solve time is about 1.2 ms. The worst case in table 3 (up to 6.7 ms) was measured with nine simulations running in parallel on one machine; a single simulation stays around 2.5 ms. Both are below the 10 ms budget.
- No episode needed a solver fallback.

12 Deployment notes and limitations

- **Speed vs. safety.** $v_{\max} = 2.8\,\mathrm{m/s}$, $\mu = 12$ and the 0.25 m corridor are conservative choices. Faster laps need larger μ / $v_{\max}$ and a wider corridor where the clearance allows it; tune μ first, in small steps, as the paper also reports crashes for too large progress weights.
- **Latency.** In simulation the command acts within the same step. On hardware there is radio and estimator delay (tens of ms). A standard remedy is to forward-simulate $\hat{\boldsymbol{x}}_0$ over the known delay with the model and the commands already sent before solving.
- **Thrust estimate.** On hardware the thrust state starts after the zero-thrust unlock command; the open-loop estimate should be reset when the motors start. Battery voltage changes the thrust map; the disturbance observer absorbs it slowly ($\approx 0.25\,\mathrm{s}$).
- **Safety layer.** The README's must-haves (kill switch, stale-pose watchdog) are outside the controller and still required for deployment.
- **Euler angles.** The model uses Euler angles, singular at $\vartheta = \pm 90°$; the tilt is bounded far away from that.
- **Obstacles** are modelled as infinite vertical cylinders, which is conservative for flying over them.

A Settings (`MPCCSettings`)

Name	Value	Meaning
`N`, `dt`	25, 0.04 s	horizon (1 s)
`q_lag`, `mu`	300, 12	lag weight, progress reward
`q_c_nominal`, `q_c_gate`	60, 400	contour weight eq. (13)
`gate_sigma`	0.25 m	Gaussian width in eq. (13)
`w_drpy`, `w_tilt`	0.08, 0.5	Euler-rate and tilt-command penalties
`r_dcmd_rp`, `r_dcmd_f`	0.4, 4	command-rate penalties
`r_a_theta`	0.02	progress acceleration penalty
`slack_l1`, `slack_l2`	10^4, 10^5	soft-constraint penalties
`tilt_max`	0.7 rad	roll/pitch command bound
`f_min`, `f_max`	0.12, 0.72 N	thrust command bounds
`dcmd_rp_max`, `dcmd_f_max`	8 rad/s, 6 N/s	command rate bounds
`v_theta_max`, `a_theta_max`	2.8 m/s, 12 m/s^2	progress bounds
`r_obs`	0.22 m	obstacle keep-out radius
`r_drone`	0.10 m	drone radius + margin (corridor)
`r_tube_min`, `r_tube_max`	0.05, 0.25 m	corridor radius limits
`v_unrevealed`, `d_unrevealed`	1.5 m/s, 1.0 m	slow approach to unrevealed gates
`replan_start_speed`	0.5 m/s	spline start speed when replanning
`max_lag`	0.6 m	re-projection threshold for θ
`stall_time`	0.3 s	stall recovery trigger
`dob_gain`, `dob_max`	0.04, 3 m/s^2	disturbance observer
`dob_slow_start`	1 m/s^2	speed reduction starts above this $\|\hat{\boldsymbol{d}}\|$
`dob_slow_rate`, `dob_slow_min`	0.15, 0.6	reduction per m/s^2, minimum factor

B How to run

- In the config: `control_mode = "attitude"` and `file = "mpcc_controller.py"` (or pass `-controller mpcc_controller.py`).
- Run (inside `pixi shell`):

```
python scripts/sim.py -config circle.toml
     -controller mpcc_controller.py
```

  The first run generates and compiles the solver (about 30 s).
- With `render = true` the planned path and the MPC prediction are drawn.

References

[1] A. Romero, S. Sun, P. Foehn, D. Scaramuzza, "Model Predictive Contouring Control for Time-Optimal Quadrotor Flight," *IEEE Transactions on Robotics*, 2022. arXiv:2108.13205.

[2] R. Verschueren, G. Frison, D. Kouzoupis, J. Frey, N. van Duijkeren, A. Zanelli, B. Novoselnik, T. Albin, R. Quirynen, M. Diehl, "acados—a modular open-source framework for fast embedded optimal control," *Mathematical Programming Computation*, 2022.

[3] M. Diehl, H. G. Bock, J. P. Schlöder, "A real-time iteration scheme for nonlinear optimization in optimal feedback control," *SIAM J. Control and Optimization*, 2005.

[4] G. Frison, M. Diehl, "HPIPM: a high-performance quadratic programming framework for model predictive control," *IFAC World Congress*, 2020.